Structured Query Language

2과목 : SQL 기본 및 활용

과목소개

과목소개 II. SQL 기본 및 활용

- 제1장 SQL 기본
  - ✓ 제1절 관계형 데이터베이스 개요
  - ✓ 제2절 DDL
  - ✓ 제3절 DML
  - ✓ 제4절 TCL
  - ✓ 제5절 WHERE 절
  - ✓ 제6절 함수(FUNCTION)
  - ✓ 제7절 GROUP BY, HAVING 절
  - ✓ 제8절 ORDER BY 절
  - ✓ 제9절 조인(JOIN)
- 제2장 SQL 활용
  - ✓ 제1절 표준 조인
  - ✓ 제2절 집합 연산자
  - ✓ 제3절 계층형 질의와 셀프 조인
  - ✓ 제4절 서브쿼리
  - ✓ 제5절 그룹 함수
  - ✓ 제6절 윈도우 함수
  - ✓ 제7절 DCL
  - ✓ 제8절 절차형 SQL
- 제3장 SQL 최적화 기본 원리
  - ✓ 제1절 옵티마이저와 실행계획
  - ✓ 제2절 인덱스 기본
  - ✓ 제3절 조인 수행 원리

- SQL의 중요성
  - ✓ 오늘날 기업 또는 조직의 정보화에 있어서 관계형 데이터베이스는 거의 대부분이라 해도 과언이 아닐 정도로 데이터 저장소의 대부분을 차지하고 있다. 소프트웨어를 작성하는데 사용되는 언어는 많은 종류가 있지만 관계형 데이터베이스는 결국 SQL에 의해서만 데이터에 접근이 가능하기 때문에 데이터베이스를 기반으로 하는 정보시스템은 SQL 사용이 필수적인 요소이다.
  - ✓ 이 때문에 정보시스템을 개발하는 수많은 개발자들은 반드시 SQL을 익힐 수밖에 없고, 이러한 상황에 의해 SQL을 사용할 수 있는 개발자는 그 수를 헤아리기 어려울 정도로 많다. 그러나 이와 같은 SQL 사용 능력 보유자 수에도 불구하고 SQL의 수행 원리를 깊이 있게 이해하고 제대로 구사할 수 있는 전문적 지식을 갖춘 인재는 상대적으로 매우 빈약하다. 이것은 결과적으로 정보시스템의 성능과 품질을 저하시키는 중요 원인이 되기도 한다.
  - ✓ SQL 개발자(SQLD*, SQL Developer)란 데이터베이스와 데이터 모델링에 대한 지식을 바탕으로 응용 소프트웨어를 개발하면서 데이터를 조작하고 추출하는데 있어서 정확하고 최적의 성능을 발휘하는 SQL을 작성할 수 있는 개발자를 말한다.

Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본 소개

- 제1장 SQL 기본
  - ✓ 제1절 관계형 데이터베이스 개요
  - ✓ 제2절 DDL
  - ✓ 제3절 DML
  - ✓ 제4절 TCL
  - ✓ 제5절 WHERE 절
  - ✓ 제6절 함수(FUNCTION)
  - ✓ 제7절 GROUP BY, HAVING 절
  - ✓ 제8절 ORDER BY 절
  - ✓ 제9절 조인(JOIN)

- 제1장 소개

  ✓ 관계형 데이터베이스와 SQL의 관계를 이해하고, 데이터를 관리하기 위한 DDL, DML, TCL의 기능과 종류를 이해한다.

  ✓ 원하는 데이터를 조회하기 위한 SELECT 문장을 구성하는 WHERE 절, GROUP BY 절, HAVING 절, ORDER BY 절의 기능과 내장함수에 대해 알아본다.

  ✓ 집합 간의 관계를 연결하기 위한 WHERE 절을 이용한 기본적인 조인(JOIN)을 이해한다.

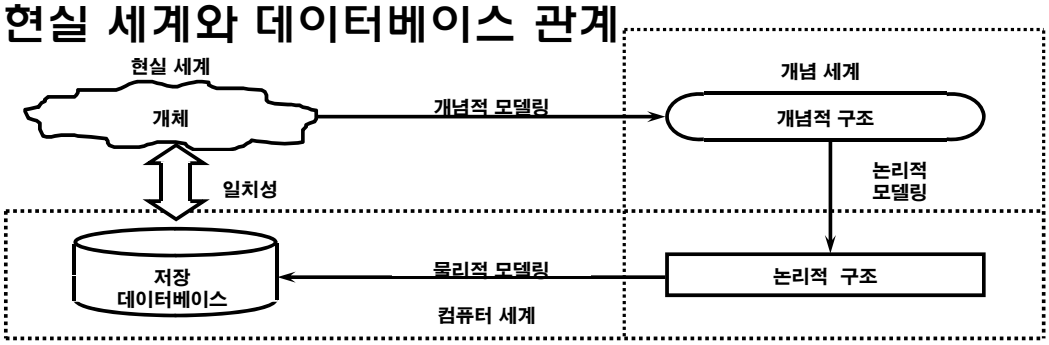
Structured Query Language

II. SQL 기본 및 활용

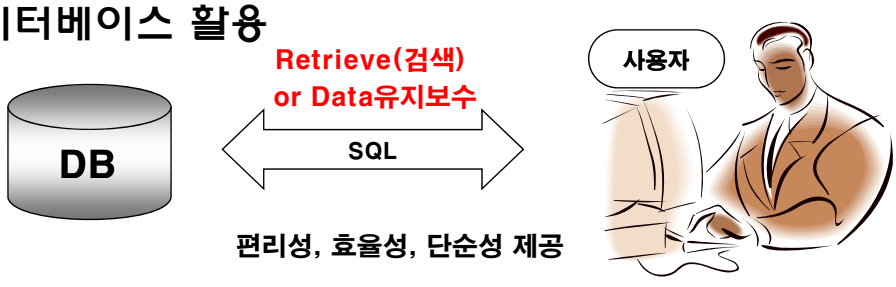
제1장 SQL 기본
제1절 관계형 데이터베이스 개요
학습하기

제1절 관계형 데이터베이스 개요

- 현실 세계와 데이터베이스 관계



- 데이터베이스 활용



- **DBMS 필요성**
  - ✓ 일반적으로 데이터베이스라고 말할 때는 특정 기업이나 조직 또는 개인이 필요에 의해(부가가치가 발생하는) 데이터를 일정한 형태로 저장해 놓은 것을 의미한다.

  ex) 학교: 학생 관리를 목적으로 학생 개개인의 정보 관리

  기업: 직원들을 관리하기 위해 직원들의 이름, 부서, 월급 등의 정보 관리

  - ✓ 관리 대상이 되는 데이터의 양이 점점 많아지고, 같은 데이터를 여러 사람이 동시에 여러 용도로 사용하게 되면서(동시성/병행 제어), 단순히 엑셀 같은 개인이 관리하는 소프트웨어 만으로는 한계에 부딪히게 된다.
  - ✓ 사용자들은 보다 효율적인 데이터의 관리 뿐만 아니라 예기치 못한 사건으로 인한 데이터의 손상을 피하고, 필요시 필요한 데이터를 복구하기 위한 강력한 기능의 소프트웨어를 필요로 하게 되었고 이러한 기본적인 요구사항을 만족시켜주는 시스템을 DBMS(Database Management System)라고 한다.

- **DBMS 발전**
  - ✓ 1960년대 : 플로우차트 중심의 개발 방법을 사용하였으며 파일 구조를 통해 데이터를 저장하고 관리하였다.
  - ✓ 1970년대 : 데이터베이스 관리 기법이 처음 태동되던 시기였으며 계층형(Hierarchical) 데이터베이스, 망형(Network) 데이터베이스 같은 제품들이 상용화 되었다.
  - ✓ 1980년대 : 현재 대부분의 기업에서 사용되고 있는 관계형 데이터베이스가 상용화되었으며 Oracle, Sybase, DB2와 같은 제품이 사용되었다.
  - ✓ 1990년대 : Oracle, Sybase, Informix, DB2, Teradata, SQL Server 외 많은 제품들이 보다 향상된 기능으로 정보시스템의 확실한 핵심 솔루션으로 자리잡게 되었으며, 인터넷 환경의 급속한 발전과 객체 지향 정보를 지원하기 위해 객체 관계형 데이터베이스로 발전하였다.

- 관계형 데이터베이스
  - ✓ 1970년 영국의 수학자였던 E.F. Codd 박사의 논문에서 처음으로 관계형 데이터베이스가 소개된 이후, IBM의 SQL 개발 단계를 거쳐서, Oracle을 선발로 여러 회사에서 상용화된 제품을 내놓았다. 이후 관계형 데이터베이스의 여러 장점이 알려지면서 기존의 파일시스템과 계층형, 망형 데이터베이스를 대부분 대체하면서 주력 데이터베이스가 되었다.
  - ✓ 현재 기업에서 사용하고 있는 대부분의 데이터베이스는 기존 관계형 데이터베이스에 객체 지원 기능을 추가한 객체 관계형 데이터베이스를 사용하고 있지만, 현실적으로 기업의 핵심 데이터는 대부분 관계형 데이터베이스 구조로 저장이 되고, 관계형 데이터베이스를 유일하게 조작할 수 있는 SQL 문장에 의해 관리되고 있으므로 관계형 데이터베이스와 SQL의 중요성은 아무리 강조해도 지나치지 않다.
  - ✓ 관계형 데이터베이스는 정규화를 통한 합리적인 테이블 모델링을 통해 데이터 이상(Anomaly) 현상을 제거하고 데이터 중복을 피할 수 있으며, 동시성 관리, 병행 제어를 통해 많은 사용자들이 동시에 데이터를 공유 및 조작할 수 있는 기능을 제공하고 있다.

- SQL(Structured Query Language)
  - ✓ SQL의 문법이 영어 문법과 흡사하기 때문에 SQL 자체는 다른 개발 언어에 비해 기초 단계 학습은 쉬운 편이지만, SQL이 시스템에 미치는 영향이 크므로 고급 SQL이나 SQL 튜닝의 중요성은 계속 커지고 있다.
  - ✓ SQL 기본 교육은 정확한 데이터를 출력하는 것이 목표이고, SQL 고급/튜닝 교육의 목적은 시스템에 큰 영향을 주는 SQL을 가장 효과적(응답시간, 자원 활용 최소화)으로 작성하는 것이 목표이다.
  - ✓ 1986년부터 ANSI/ISO를 통해 표준화되고 정의된 SQL 기능은 벤더별 DBMS 개발의 목표가 된다. 구체적인 용어는 다르더라도 대부분의 관계형 데이터베이스에서 ANSI/ISO 표준을 최대한 따르고 있기 때문에, SQL에 대한 지식은 다른 데이터베이스를 사용하더라도 상당 부분 기존 지식을 재활용할 수 있고, ANSI/ISO SQL-99, SQL-2003 이후 기준이 적용된 SQL이라면 프로그램의 이식성을 높이는 데도 공헌한다. (함수는 차이가 많음)

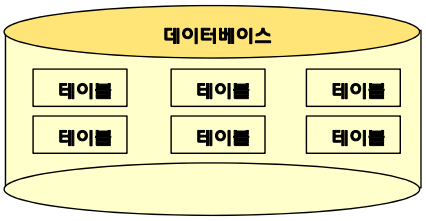
- SQL(Structured Query Language)
  - ✓ SQL 문장은 단순 스크립트가 아니라 이름에도 포함되어 있듯이, 일반적인 개발 언어처럼 독립된 하나의 개발 언어이다. 하지만 일반적인 프로그래밍 언어와는 달리 SQL은 관계형 데이터베이스에 대한 전담 접속(다른 언어는 관계형 데이터베이스에 접속할 수 없다) 용도로 사용되며 세미콜론(;)으로 분리되어 있는 SQL 문장 단위로 독립되어 있다.
  - ✓ 관계형 데이터베이스는 수학의 집합 논리에 입각한 것이므로, SQL도 데이터를 집합으로써 취급한다. 예를 들어 '포지션이 미드필더(MF)인 선수의 정보를 검색한다' 고 할 경우, 선수라는 큰 집합에서 포지션이 미드필더인 조건을 만족하는 요구 집합을 추출하는 조작이 된다.
  - ✓ 이렇게 특정 데이터들의 집합에서 필요로 하는 데이터를 꺼내서 조회하고 새로운 데이터를 입력/수정/삭제하는 행위를 통해서 사용자는 데이터베이스와 대화하게 된다. 그리고 SQL은 이러한 대화를 가능하도록 매개 역할을 하는 것이다. 결과적으로 SQL 문장을 배우는 것이 곧 관계형 데이터베이스를 배우는 기본 단계라 할 수 있다.

- SQL 종류

명령어의 종류	명령어	설명
데이터 조작어 (DML: Data Manipulation Language)	SELECT	데이터베이스에 들어 있는 데이터를 조회하거나 검색하기 위한 명령어를 말하는 것으로 RETRIEVE 라고도 한다. (가장 중요함)
	INSERT UPDATE DELETE	데이터베이스의 테이블에 들어 있는 데이터에 변형을 가하는 종류의 명령어들을 말한다. 예를 들어 데이터를 테이블에 새로운 행을 집어 넣거나, 원하지 않는 데이터를 삭제하거나 수정하는 것들의 명령어들을 DML이라고 부른다.
데이터 정의어 (DDL: Data Manipulation Language)	CREATE ALTER DROP RENAME	테이블과 같은 데이터 구조를 정의하는데 사용되는 명령어들로 그러한 구조를 생성하거나 변경하거나 삭제하거나 이름을 바꾸는 데이터 구조와 관련된 명령어들을 DDL이라고 부른다.
데이터 제어어 (DCL: Data Control Language)	GRANT REVOKE	데이터베이스에 접근하고 객체들을 사용하도록 권한을 주고 회수하는 명령어를 DCL이라고 부른다.
트랜잭션 제어어 (TCL: Transaction Control Language)	COMMIT ROLLBACK	논리적인 작업의 단위를 묶어서 DML에 의해 조작된 결과를 작업단위(트랜잭션) 별로 제어하는 명령어를 말한다.

제1절 관계형 데이터베이스 개요

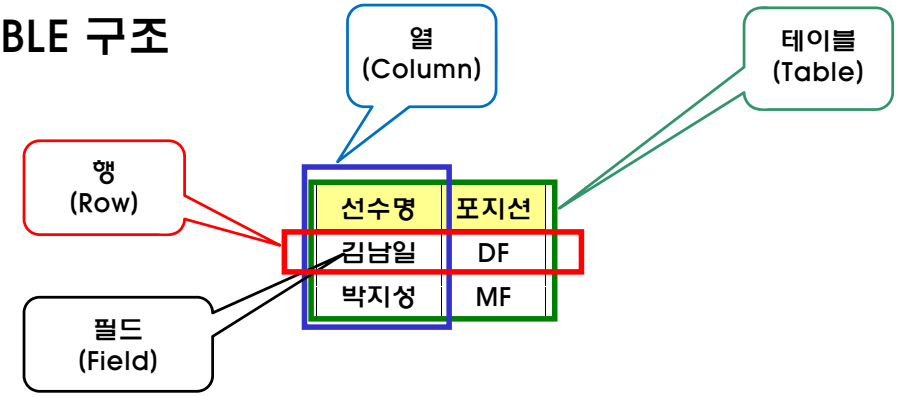
- TABLE
  - ✓ 테이블(TABLE)은 데이터를 저장하는 객체(Object)로서 관계형 데이터베이스의 기본 단위이다.
  - ✓ 모든 자료는 테이블에 등록이 되고, 테이블로부터 원하는 자료를 꺼내 올 수 있다. (SQL 이용)
  - ✓ 테이블은 어느 특정한 주제와 목적으로 만들어지는 일종의 집합이다.



선수	팀	팀 연고지	포지션	등 번호	생년월일	키	몸 무게
정성철	일화천마	성남	MF	6	90/01/24	165	57
윤용구	드래곤즈	전남	MF	15	87/08/08	168	60
정서연	FC서울	서울	MF	40	86/05/28	168	68
이청용	블루윙즈	수원	MF	17	88/07/02	180	69
...	...	...	...	...	...	...	...

제1절 관계형 데이터베이스 개요

- TABLE 구조



용어	설명
테이블 (Table)	행과 칼럼의 2차원 구조를 가진 데이터의 저장 장소이며, 데이터베이스의 가장 기본적인 구성 요소
칼럼/열 (Column)	2차원 구조를 가진 테이블에서 세로 방향으로 이루어진 하나하나의 특정 속성 (더이상 나눌 수 없는 특성)
로우/행 (Row)	2차원 구조를 가진 테이블에서 가로 방향으로 이루어진 연결된 데이터 ※ 현장에서는 레코드(Record)라고도 한다

- TABLE 관계

기본 키(Primary Key)　　외부 키(Foreign Key)　　기본 키(Primary Key)

선수번호	이름	팀 코드	포지션	등번호	키
1	김남일	K03	DF	33	177
2	박지성	K07	MF	7	178
3	이영표	K02	MF	22	176

<선수 테이블>

팀 코드	팀 명	연고지
K03	스틸러스	포항
K07	드래곤즈	전남
K02	블루윙즈	수원

<구단 테이블>

용어	설명
정규화 (Normalization)	테이블을 분할하여 데이터의 정합성을 확보하고, 불필요한 중복을 줄이는 프로세스
기본키 (Primary Key)	테이블에 존재하는 각 행을 한 가지 의미로 특정할 수 있는 한 개 이상의 칼럼
외부키 (Foreign Key)	다른 테이블의 기본 키로 사용되고 있는 관계를 연결하는 칼럼

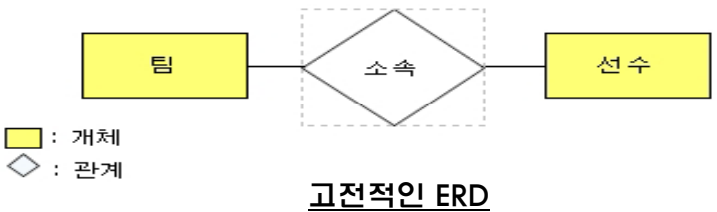
- ERD(Entity Relationship Diagram)

◇ ✓ 테이블 간 서로의 상관 관계를 그림으로 도식화한 것을 E-R 다이어그램이라고하며, 간략히 ERD라고 한다.

✓ ERD의 구성 요소는 개체(Entity), 관계(Relationship), 속성(Attribute) 3가지이다.

✓ 현실 세계의 데이터는 이 3가지 구성 요소로 모두 표현이 가능하다.

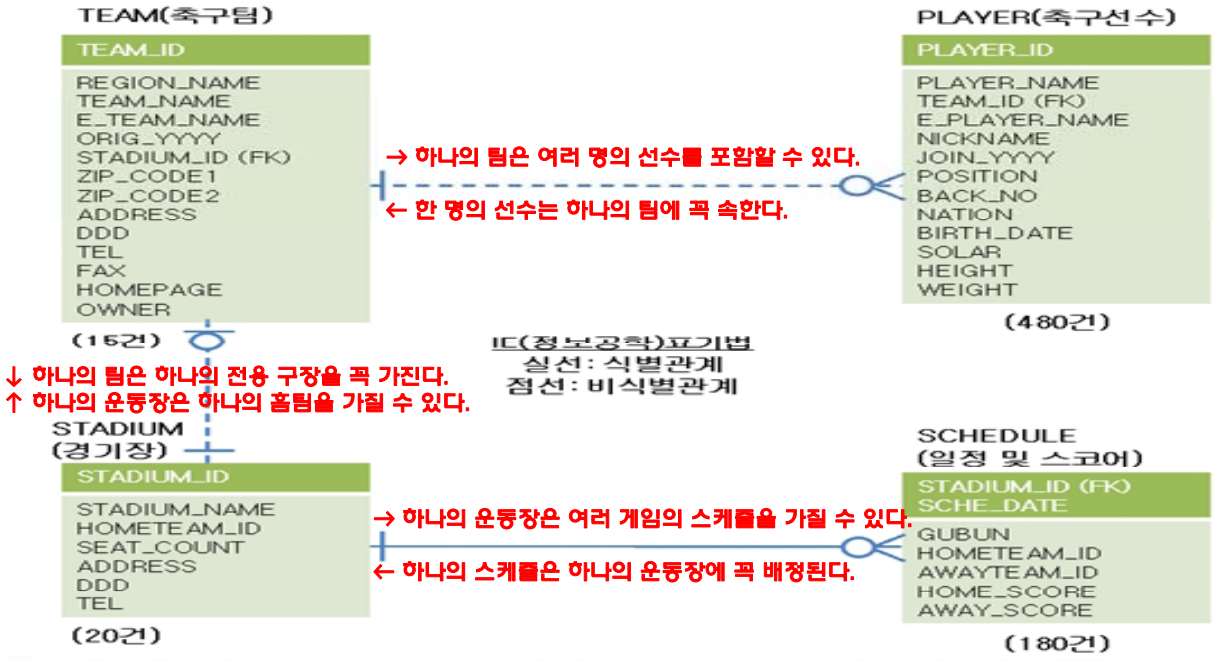
✓ 팀과 선수 간에는 "소속"이라는 관계가 맺어져 있다.



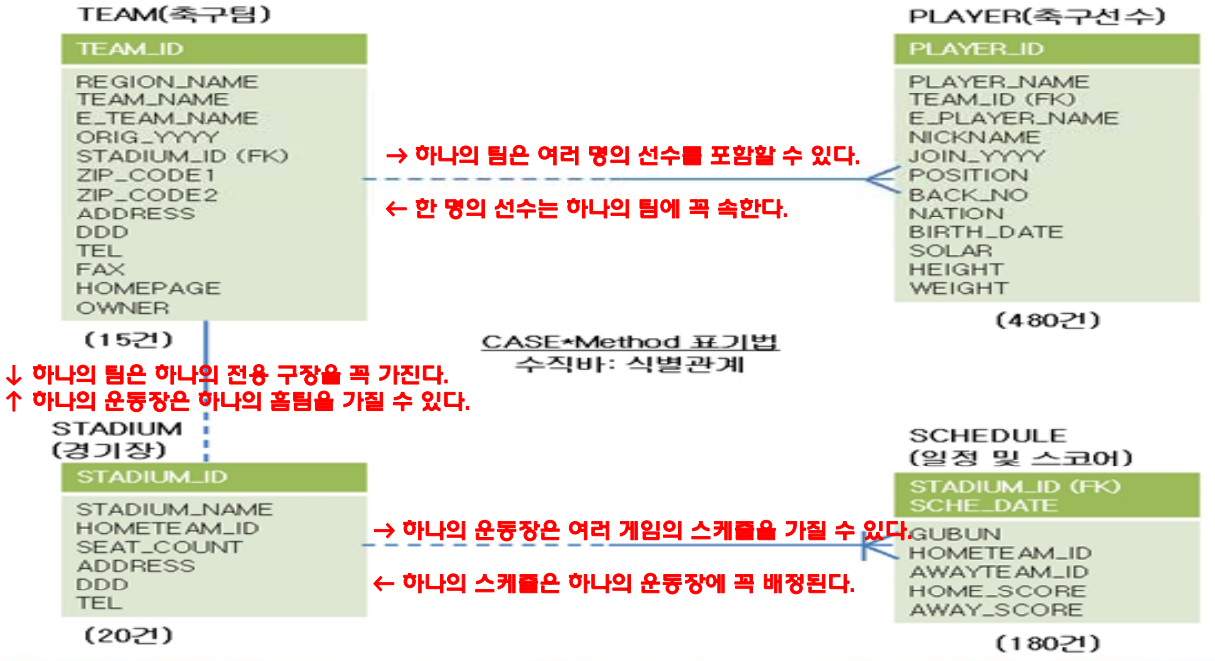
고전적인 ERD

(속성이 많은 경우 비효율적임)

- K-리그 ERD (정보공학[I/E] 표기법)

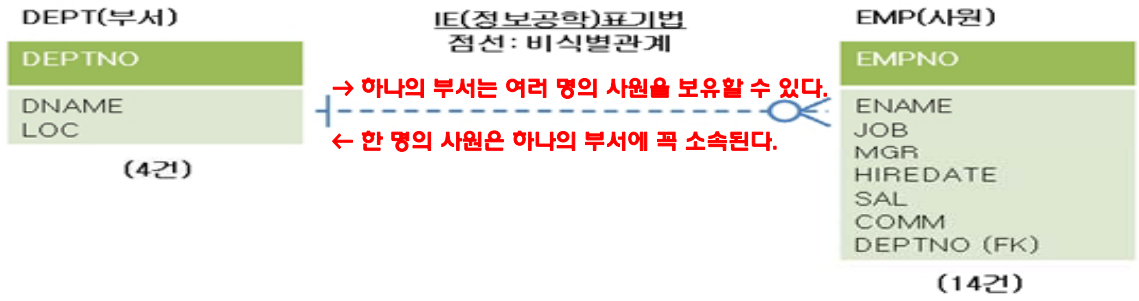


- K-리그 ERD (Case*Method 표기법)



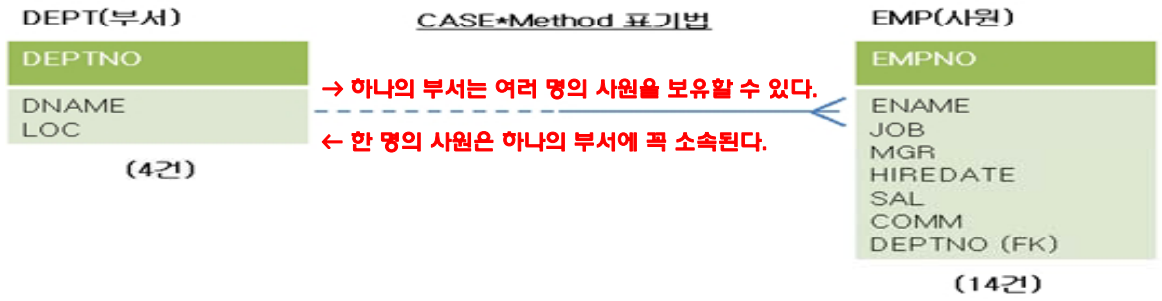
제1절 관계형 데이터베이스 개요

- 부서-사원 ERD (정보공학[I/E] 표기법)



제1절 관계형 데이터베이스 개요

- 부서-사원 ERD (Case*Method 표기법)



Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제1절 관계형 데이터베이스 개요
핵심정리 및 연습문제

핵심정리 제1절 관계형 데이터베이스 개요

- 특정 데이터들의 집합에서 필요로 하는 데이터를 꺼내서 조회하고 새로운 데이터를 입력/수정/삭제하는 행위를 통해서 사용자는 데이터베이스와 대화하게 된다.
- SQL(Structured Query Language)은 이러한 대화를 가능하도록 매개 역할을 하는 것이다.
- SQL은 관계형 데이터베이스의 데이터 정의, 데이터 조작, 데이터 제어를 위해 사용하는 언어이다.

문제. 다음 설명 중 맞는 것은 무엇인가?

① 데이터베이스에는 단 한 개의 테이블만 존재할 수 있다.

② 데이터베이스 내에 테이블이란 존재하지 않는다.

③ 아주 복잡한 자료도 테이블은 하나만 만드는 것이 바람직하다.

④ 모든 자료는 실질적으로 테이블에 저장이 되며, 테이블에 있는 자료들을 꺼내 볼 수 있다.

연습문제 제1절 관계형 데이터베이스 개요

문제. 다음 설명 중 맞는 것은 무엇인가?

① 데이터베이스에는 단 한 개의 테이블만 존재할 수 있다.

② 데이터베이스 내에 테이블이란 존재하지 않는다.

③ 아주 복잡한 자료도 테이블은 하나만 만드는 것이 바람직하다.

④ 모든 자료는 실질적으로 테이블에 저장이 되며, 테이블에 있는 자료들을 꺼내 볼 수 있다.

정답 : ④

해설 :

데이터베이스에는 자료의 성격에 따라 M개의 테이블을 생성한다.
모든 자료들은 테이블에 입력되며, 조회, 수정, 삭제 할 수 있다.

Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제2절 DDL 학습하기

제2절 DDL (Data Definition Language)

- 중요 데이터 유형

데이터 유형	설명
CHARACTER(s)	– 고정 길이 문자열 정보 (Oracle, SQL Server 모두 CHAR로 표현) – s는 기본 길이 1바이트, 최대 길이 Oracle 2,000바이트, SQL Server 8,000바이트 – s만큼 최대 길이를 갖고 고정 길이를 가지고 있으므로 해당 할당된 변수값의 길이가 s보다 작을 경우에는 그 차이 길이만큼 공간으로 채워진다.
VARCHAR(s)	– CHARACTER VARYING의 약자로 가변 길이 문자열 정보 (Oracle은 VARCHAR2로 표현, SQL Server는 VARCHAR로 표현) – s는 최소 길이 1바이트, 최대 길이 Oracle 4,000바이트, SQL Server 8,000바이트 – s만큼의 최대 길이를 갖지만 가변 길이로 조정이 되기 때문에 해당 할당된 변수값의 바이트만 적용된다. (Limit 개념)
NUMERIC	– 정수, 실수 등 숫자 정보 (Oracle은 NUMBER로, SQL Server는 10가지 이상의 숫자 타입을 가지고 있음) – Oracle은 처음에 전체 자리 수를 지정하고, 그 다음 소수 부분의 자리 수를 지정한다. 예를 들어, 정수 부분이 6자리이고 소수점 부분이 2자리인 경우에는 'NUMBER(8,2)'와 같이 된다.
DATETIME	– 날짜와 시각 정보 (Oracle은 DATE로 표현, SQL Server는 DATETIME으로 표현) – Oracle은 1초 단위, SQL Server는 3.33ms 단위로 관리

- 문자열 유형
  - ✓ VARCHAR유형은 가변 길이이므로 필요한 영역은 실제 데이터 크기분이다. 그렇기 때문에 길이가 다양한 칼럼과, 정의된 길이와 실제 데이터 길이에 차이가 있는 칼럼에 적합하다. 저장 측면에서도 CHAR 유형보다 작은 영역에 저장할 수 있으므로 장점이 있다.
  - ✓ CHAR에서는 문자열을 비교할 때 공백(BLANK)을 채워서 비교하는 방법을 사용한다. 공백 채우기 비교에서는 우선 짧은 쪽의 끝에 공백을 추가하여 2개의 데이터가 같은 길이가 되도록 한다. 그리고, 앞에서부터 한 문자씩 비교한다. 그렇기 때문에 끝의 공백만 다른 문자열은 같다고 판단된다. 그에 반해 VARCHAR 유형에서는 맨 처음부터 한 문자씩 비교하고 공백도 하나의 문자로 취급하므로 끝의 공백이 다르면 다른 문자로 판단한다.

예) CHAR 유형	예) VARCHAR 유형
'AA' = 'AA '	'AA' ≠ 'AA '

- K-리그 테이블과 칼럼 정보

테이블	칼럼 설명
선수 정보 (PLAYER)	선수에 대한 상세 정보 (선수ID, 선수명, 소속팀ID, 영문선수명, 선수별명, 입단년도, 포지션, 등번호, 국적, 생년월일, 양/음, 키, 몸무게)
팀 정보 (TEAM)	소속팀에 대한 상세 정보 (팀ID, 연고지명, 팀명, 영문팀명, 창단년도, 운동장ID, 우편번호1, 우편번호2, 주소, 지역번호, 전화번호, 팩스, 홈페이지, 구단주)
운동장 정보 (STADIUM)	운동장에 대한 상세 정보 (운동장ID, 운동장명, 홈팀ID, 좌석수, 주소, 지역번호, 전화번호)
경기 일정 (SCHEDULE)	경기 일정 및 스코어에 대한 정보 (운동장ID, 경기일자, 경기진행여부, 홈팀ID, 원정팀ID, 홈팀득점, 원정팀득점)

- ✓ 한 테이블 안에서 칼럼 이름은 달라야 하지만, 다른 테이블의 칼럼 이름과는 같을 수 있다.
- ✓ 예를 들면 선수 테이블의 소속팀ID, 팀 테이블의 팀ID는 같은 칼럼 이름을 가지고 있다. 실제 DBMS는 내부적으로는 팀 테이블의 TEAM_ID를 PC나 UNIX의 디렉토리 구조처럼 'DB명+DB사용자명+테이블명+칼럼명' 처럼 계층적 구조를 가진 전체 경로로 관리하고 있다.
- ✓ 이처럼 같은 이름을 가진 칼럼 들은 기본키와 외래키의 관계를 가지는 경우가 많으며, 향후 테이블 간의 조인 조건으로 주로 사용되는 중요한 연결고리 칼럼들이다.

- 테이블 생성 규칙
  - 테이블명은 객체를 의미할 수 있는 적절한 이름을 사용한다. 가능한 단수형을 권고한다.
  - 테이블명은 다른 테이블의 이름과 중복되지 않아야 한다.
  - 한 테이블 내에서는 칼럼명이 중복되게 지정될 수 없다.
  - 테이블 이름을 지정하고 각 칼럼들은 괄호 "()" 로 묶어 지정한다.
  - 각 칼럼들은 콤마","로 구분되고, 항상 끝은 세미콜론 ";"으로 끝난다.
  - 칼럼에 대해서는 다른 테이블까지 고려하여 데이터베이스 내에서는 일관성 있게 사용하는 것이 좋다. (데이터 표준화 관점)
  - 칼럼 뒤에 데이터 유형은 꼭 지정되어야 한다.
  - 테이블명과 칼럼명은 반드시 문자로 시작해야 하고, 벤더별로 길이에 대한 한계가 있다.
  - 벤더에서 사전에 정의한 예약어(Reserved word)는 쓸 수 없다.
  - A-Z, a-z, 0-9, _, $, # 문자만 허용된다.

※ 테이블명이 잘못된 사례

10_PLAYER	반드시 숫자가 아닌 문자로 시작되어야 함
T-PLAYER	특수 문자 '-'는 허용되지 않음

- 선수 테이블과 칼럼 정보

테이블명 : PLAYER
테이블 설명 : K-리그 선수들의 정보를 가지고 있는 테이블
칼럼명 : PLAYER_ID (선수ID) 문자 고정 자릿수 7자리,
PLAYER_NAME (선수명) 문자 가변 자릿수 20자리,
TEAM_ID (팀ID) 문자 고정 자릿수 3자리,
E_PLAYER_NAME (영문선수명) 문자 가변 자릿수 40자리,
NICKNAME (선수별명) 문자 가변 자릿수 30자리,
JOIN_YYYY (입단년도) 문자 고정 자릿수 4자리,
POSITION (포지션) 문자 가변 자릿수 10자리,
BACK_NO (등번호) 숫자 2자리,
NATION (국적) 문자 가변 자릿수 20자리,
BIRTH_DATE (생년월일) 날짜,
SOLAR (양/음) 문자 고정 자릿수 1자리,
HEIGHT (신장) 숫자 3자리,
WEIGHT (몸무게) 숫자 3자리,
제약조건 : 기본키(PRIMARY KEY) → PLAYER_ID
(제약조건명은 PLAYER_ID_PK)
값이 반드시 존재 (NOT NULL) → PLAYER_NAME, TEAM_ID

- 선수 테이블 생성 DDL(Data Definition Language)

[예제] Oracle

```
CREATE TABLE   PLAYER (
PLAYER_ID      CHAR(7) NOT NULL,
PLAYER_NAME    VARCHAR2(20) NOT NULL,
TEAM_ID        CHAR(3) NOT NULL,
E_PLAYER_NAME  VARCHAR2(40),
NICKNAME       VARCHAR2(30),
JOIN_YYYY      CHAR(4),
POSITION       VARCHAR2(10),
BACK_NO        NUMBER(2),
NATION         VARCHAR2(20),
BIRTH_DATE     DATE,
SOLAR          CHAR(1),
HEIGHT         NUMBER(3),
WEIGHT         NUMBER(3),
CONSTRAINT PLAYER_PK PRIMARY KEY (PLAYER_ID),
CONSTRAINT PLAYER_FK FOREIGN KEY (TEAM_ID)
REFERENCES TEAM(TEAM_ID)
);
테이블이 생성되었다.
```

[예제] SQL Server

```
CREATE TABLE   PLAYER (
PLAYER_ID      CHAR(7) NOT NULL,
PLAYER_NAME    VARCHAR(20) NOT NULL,
TEAM_ID        CHAR(3) NOT NULL,
E_PLAYER_NAME  VARCHAR(40),
NICKNAME       VARCHAR(30),
JOIN_YYYY      CHAR(4),
POSITION       VARCHAR(10),
BACK_NO        TINYINT,
NATION         VARCHAR(20),
BIRTH_DATE     DATE,
SOLAR          CHAR(1),
HEIGHT         SMALLINT,
WEIGHT         SMALLINT,
CONSTRAINT PLAYER_PK PRIMARY KEY (PLAYER_ID),
CONSTRAINT PLAYER_FK FOREIGN KEY (TEAM_ID)
REFERENCES TEAM(TEAM_ID)
);
테이블이 생성되었다.
```


- 제약 조건 종류

PRIMARY KEY (기본키)	테이블에 저장된 행 데이터를 고유하게 식별하기 위한 기본키를 정의한다. 하나의 테이블에 하나의 기본키 제약만 정의할 수 있다. 기본키 제약을 작성하면 DBMS는 자동으로 UNIQUE 인덱스를 작성하며, 기본키를 구성하는 칼럼에는 NULL을 입력할 수는 없다. 결국 '기본키 제약 = 고유키 제약 & NOT NULL 제약'이 된다.
UNIQUE (고유키)	테이블에 저장된 행 데이터를 고유하게 식별하기 위한 기본키를 정의한다. 단, NULL은 고유키 제약의 대상이 아니므로, NULL 값을 가진 행이 여러 개 가 있더라도 고유키 제약 위반이 되지 않는다.
NOT NULL	NULL 값의 입력을 금지한다. 디폴트 상태에서는 모든 칼럼에서 NULL을 허 가하고 있지만, 이 제약을 지정함으로써 해당 칼럼은 입력 필수가 된다. NOT NULL을 CHECK의 일부분으로 이해할 수도 있다.
CHECK	입력할 수 있는 값의 범위 등을 제한한다. CHECK 제약으로는 TRUE or FALSE로 평가할 수 있는 논리식을 지정한다.
FOREIGN KEY (외래키)	관계형 데이터베이스에서 테이블 간의 관계를 정의하기 위해 기본키를 다른 테이블의 외래키로 복사하는 경우 외래키가 생성된다. 외래키 지정시 참조 무결성 제약 옵션을 선택할 수 있다.

- 제약 조건
  - ✓ 제약조건은 PLAYER_NAME, TEAM_ID 칼럼의 데이터 유형 뒤에 NOT NULL을 정의한 사례와 같은 칼럼 LEVEL 방식과, PLAYER_PK PRIMARY KEY, PLAYER_FK FOREIGN KEY 사례처럼 테이블 생성 마지막에 모든 제약조건을 기술하는 테이블 LEVEL 방식이 있다. 하나의 SQL 문장 내에서 두 가지 방식은 혼용해서 사용할 수 있으며, 같은 기능을 가지고 있다. 또한 별도의 DDL 문장으로 수행할 수도 있다.
  - ✓ NULL은 공백(BLANK, ASCII 코드 32번)이나 숫자 0(ZERO, ASCII 48)과는 전혀 다른 값이며, 조건에 맞는 데이터가 없을 때의 공집합과도 다르다. 'NULL'은 '아직 정의되지 않은 미지의 값'이거나 '현재 데이터를 입력하지 못하는 경우'를 의미한다.
  - ✓ 데이터 입력 시에 칼럼의 값이 지정되어 있지 않을 경우 기본값(DEFAULT)을 사전에 설정할 수 있다. 데이터 입력시 명시된 값을 지정하지 않은 경우에 NULL 값이 입력되고, DEFAULT 값을 정의했다면 해당 칼럼에 NULL 값이 입력되지 않고 사전에 정의된 기본 값이 자동으로 입력된다.

- 생성된 테이블 구조 확인

Oracle

```
DESCRIBE PLAYER;

칼럼                      NULL 가능      데이터 유형
------------------------- -------------- -------------
PLAYER_ID                 NOT NULL       CHAR(7)
PLAYER_NAME               NOT NULL       VARCHAR2(20)
TEAM_ID                   NOT NULL       CHAR(3)
E_PLAYER_NAME                            VARCHAR2(40)
NICKNAME                                 VARCHAR2(30)
JOIN_YYYY                                CHAR(4)
POSITION                                 VARCHAR2(10)
BACK_NO                                  NUMBER(2)
NATION                                   VARCHAR2(20)
BIRTH_DATE                               DATE
SOLAR                                    CHAR(1)
HEIGHT                                   NUMBER(3)
WEIGHT                                   NUMBER(3)
```

- 생성된 테이블 구조 확인

SQL Server

```
exec sp_help 'dbo.PLAYER'
go
```

칼럼이름	데이터유형	길이	NULL가능
PLAYER_ID	CHAR(7)	7	NO
PLAYER_NAME	VARCHAR(20)	20	NO
TEAM_ID	CHAR(3)	3	NO
E_PLAYER_NAME	VARCHAR(40)	40	YES
NICKNAME	VARCHAR(30)	30	YES
JOIN_YYYY	CHAR(4)	4	YES
POSITION	VARCHAR(10)	10	YES
BACK_NO	TINYINT	1	YES
NATION	VARCHAR(20)	20	YES
BIRTH_DATE	DATE	3	YES
SOLAR	CHAR(1)	1	YES
HEIGHT	SMALLINT	2	YES
WEIGHT	SMALLINT	2	YES

- SELECT 문장을 통한 테이블 생성 사례
  - ✓ DML 문장 중에 SELECT 문장을 활용해서 테이블 생성할 (CTAS: Create Table ~ As Select ~) 수 있는 방법이 있다. 기존 테이블을 이용한 CTAS 방법을 이용할 수 있다면 칼럼별로 데이터 유형을 다시 재정의하지 않아도 되는 장점이 있다.
  - ✓ CTAS 기법 사용시 주의할 점은 기존 테이블의 제약조건 중에 NOT NULL만 새로운 복제 테이블에 적용이 되고, 기본키, 고유키, 외래키, CHECK 등의 다른 제약 조건은 없어진다는 점이다. 제약 조건을 추가하기 위해서는 뒤에 나오는 ALTER TABLE 기능을 사용해야 한다.

[예제] Oracle

```
CREATE TABLE TEAM_TEMP
AS  SELECT * FROM TEAM;
  테이블이 생성되었다.
```

[예제] SQL Server

```
SELECT * INTO TEAM_TEMP
FROM TEAM;
  테이블이 생성되었다.
```

- ALTER TABLE
  - ✓ ADD COLUMN
    - 주의할 것은 새롭게 추가된 칼럼은 테이블의 마지막 칼럼이 되며 칼럼의 위치를 지정할 수는 없다.

[예제]

```
ALTER TABLE 테이블명
ADD (추가할 칼럼명);

ALTER TABLE PLAYER
ADD (ADDRESS VARCHAR2(80));
테이블이 변경되었다.
```


- ALTER TABLE
  - ✓ DROP COLUMN
    - 한 번에 하나의 칼럼만 삭제 가능하며, 칼럼 삭제 후 최소 하나 이상의 칼럼이 테이블에 존재해야 한다. 주의할 부분은 한 번 삭제된 칼럼은 복구가 불가능하다.

[예제]

```
ALTER TABLE 테이블명
DROP COLUMN 삭제할 칼럼명;

ALTER TABLE PLAYER
DROP COLUMN ADDRESS;
테이블이 변경되었다.
```

- ALTER TABLE
  - ✓ MODIFY COLUMN
    - 테이블에 존재하는 칼럼에 대해서 MODIFY COLUMN명령을 이용해 칼럼의 데이터 유형, 디폴트(DEFAULT) 값, NOT NULL 제약조건에 대한 변경을 포함할 수 있다.

[예제] Oracle

```
ALTER TABLE 테이블명
MODIFY (칼럼명1 데이터 유형 [DEFAULT 식] [NOT NULL],
        칼럼명2 데이터 유형 ...);

ALTER TABLE TEAM_TEMP
MODIFY (ORIG_YYYY VARCHAR2(8) DEFAULT '20020129' NOT NULL);
  테이블이 변경되었다.
```

[예제] SQL Server

```
ALTER TABLE 테이블명
ALTER  (칼럼명1 데이터 유형 [DEFAULT 식] [NOT NULL],
        칼럼명2 데이터 유형 ...);
```


- ALTER TABLE
  - ✓ RENAME COLUMN
    - RENAME COLUMN으로 칼럼명이 변경되면, 해당 칼럼과 관계된 제약조건에 대해서도 자동으로 변경되는 장점이 있지만, ADD/DROP COLUMN 기능처럼 ANSI/ISO에 명시되어 있는 기능이 아니고 Oracle 등 일부 DBMS에서만 지원하는 기능이다.

[예제] Oracle

```
ALTER  TABLE  테이블명
RENAME COLUMN 변경해야 할 칼럼명 TO 새로운 칼럼명;

ALTER  TABLE  PLAYER
RENAME COLUMN PLAYER_ID TO TEMP_ID;
테이블이 변경되었다.
```

- ALTER TABLE
  - ✓ DROP CONSTRAINT
    - 테이블 생성 시 부여했던 제약조건을 삭제하는 명령어 형태는 다음과 같다.

[예제]

```
ALTER TABLE 테이블명
DROP  CONSTRAINT 제약조건명;

ALTER TABLE PLAYER
DROP  CONSTRAINT PLAYER_FK;
테이블이 변경되었다.
```


- ALTER TABLE
  - ✓ ADD CONSTRAINT
    - 테이블 생성 이후에 필요에 의해서 제약조건을 추가할 수 있다.

[예제]

```
ALTER TABLE 테이블명
ADD   CONSTRAINT 제약조건명 (칼럼명);

ALTER TABLE PLAYER
ADD   CONSTRAINT PLAYER_FK
FOREIGN KEY (TEAM_ID) REFERENCE TEAM(TEAM_ID)
테이블이 변경되었다.
```

- RENAME TABLE
  - ✓ RENAME 명령어를 사용하여 테이블의 이름을 변경할 수 있다.

[예제]

```
RENAME 변경전_테이블명 TO 변경후_테이블명

RENAME TEAM TO TEAM_BACKCUP;
테이블 이름이 변경되었다.

RENAME TEAM_BACKCUP TO TEAM;
테이블 이름이 변경되었다.
```


- DROP TABLE
  - ✓ 테이블을 잘못 만들었거나 테이블이 더 이상 필요없을 경우 해당 테이블을 삭제해야 한다.

[예제]

```
DROP TABLE 테이블명 [CASCADE CONSTRAINT];

DROP TABLE TEAM;
테이블이 삭제되었다.

DESC TEAM;
ERROR: 설명할 객체를 찾을 수 없습니다.
```

- TRUNCATE TABLE
  - ✓ TRUNCATE TABLE은 테이블 자체가 삭제되는 것이 아니고, 해당 테이블에 들어있던 모든 행들이 제거되고 저장공간을 재사용 가능하도록 해제된다.
  - ✓ 테이블 구조를 완전히 삭제하기 위해서는 DROP TABLE을 실행해야 된다.

[예제]

```
TRUNCATE TABLE PLAYER;

TRUNCATE TABLE TEAM;
테이블이 트렁케이트되었다.
```

실행 결과

```
DESC TEAM;
칼럼                      NULL 가능 데이터 유형
------------------------- -------- ------------
TEAM_ID                   NOT NULL CHAR(3)
REGION_NAME               NOT NULL VARCHAR2(4)
TEAM_NAME                 NOT NULL VARCHAR2(40)
E_TEAM_NAME                        VARCHAR2(50)
ORIG_YYYY                          CHAR(4)
STADIUM_ID                NOT NULL CHAR(3)
ZIP_CODE1                          CHAR(3)
ZIP_CODE2                          CHAR(3)
ADDRESS                            VARCHAR2(80)
DDD                                VARCHAR2(3)
TEL                                VARCHAR2(10)
FAX                                VARCHAR2(10)
HOMEPAGE                           VARCHAR2(50)
OWNER                              VARCHAR2(10)
```


- TRUNCATE TABLE
  - ✓ DROP TABLE의 경우는 테이블 자체가 없어지기 때문에 테이블 구조를 확인할 수 없다. 반면 TRUNCATE TABLE의 경우는 테이블 구조는 그대로 유지한 체 데이터만 전부 삭제하는 기능이다.
  - ✓ TRUNCATE는 데이터 구조의 변경 없이 테이블의 데이터를 일괄 삭제하는 명령어로 DML로 분류할 수도 있지만, 내부 처리 방식이나 Auto Commit 특성 등으로 인해 DDL로 분류하였다.
  - ✓ 테이블에 있는 데이터를 삭제하는 명령어는 TRUNCATE TABLE 명령어 이외에도 다음 DML 절에서 살펴볼 DELETE 명령어가 있다. 그러나 DELETE와 TRUNCATE는 처리하는 방식 자체가 다르다.
  - ✓ 테이블의 전체 데이터를 삭제하는 경우, 시스템 활용 측면에서는 DELETE TABLE 보다는 시스템 부하가 적은 TRUNCATE TABLE을 권고한다. 단, TRUNCATE TABLE의 경우 정상적인 복구가 불가능하므로 주의해야 한다.

Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제2절 DDL 핵심정리 및 연습문제

핵심정리 제2절 DDL (Data Definition Language)

- 데이터 유형은 데이터베이스의 테이블에 특정 자료를 입력할 때, 그 자료를 받아들일 공간을 자료의 유형별로 나누는 기준이라고 생각하면 된다.
- 중요한 데이터 유형으로는 CHARACTER(s), VARCHAR(s), NUMERIC, DATETIME 유형이 있다.
- 테이블은 CREATE TABLE 문장에 의해 생성되고,
- ALTER TABLE 문장에 의해 수정되며,
- DROP TABLE 문장에 의해 삭제되고,
- RENAME TABLE 문장에 의해 재명명된다.

문제. 데이터 유형에 대한 설명 중 틀린 것은 무엇인가?

① CHAR 유형은 고정 길이 문자형이다.
② VARCHAR 유형은 가변 길이 숫자형이다.
③ NUMERIC 유형은 숫자형 데이터를 표현한다.
④ DATE 유형은 날짜 데이터를 다룰 때 사용한다.

연습문제 제2절 DDL (Data Definition Language)

문제. 데이터 유형에 대한 설명 중 틀린 것은 무엇인가?

① CHAR 유형은 고정 길이 문자형이다.
② VARCHAR 유형은 가변 길이 숫자형이다.
③ NUMERIC 유형은 숫자형 데이터를 표현한다.
④ DATE 유형은 날짜 데이터를 다룰 때 사용한다.

정답 : ②
해설 :
VARCHAR 유형은 가변 길이 문자형이다.

문제. 다음 중 테이블 명으로 가능한 것은 무엇인가?

① EMP100
② 100EMP
③ EMP-100
④ 100_EMP

연습문제 제2절 DDL (Data Definition Language)

문제. 다음 중 테이블 명으로 가능한 것은 무엇인가?

① EMP100
② 100EMP
③ EMP-100
④ 100_EMP

정답 : ①
해설 :
테이블명과 칼럼명은 반드시 문자로 시작해야 한다
사용되는 글자는 A-Z, a-z, 0-9, _, $, # 만 허용함

Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제3절 DML 학습하기

제3절 DML (Data Manipulation Language)

- INSERT
  - ✓ 테이블에 데이터를 입력하는 방법은 두 가지 유형이 있으며 한 번에 한 건만 입력된다.

```
a. INSERT INTO 테이블명 (COLUMN_LIST)
     VALUES (COLUMN_LIST에 넣을 VALUE_LIST);

b. INSERT INTO 테이블명
     VALUES (전체 COLUMN에 넣을 VALUE_LIST);
```

  - ✓ 해당 칼럼의 데이터 유형이 CHAR나 VARCHAR2 등 문자 유형일 경우 『'』(SINGLE QUOTATION)로 입력할 값을 입력한다.
  - ✓ 숫자일 경우 『'』(SINGLE QUOTATION)을 붙이지 않아도 된다.

- INSERT (a type)
  - ✓ [예제] 선수 테이블에 박지성 선수의 데이터를 일부 칼럼만 입력한다.

[예제]

```
INSERT INTO PLAYER
      (PLAYER_ID, PLAYER_NAME, TEAM_ID, POSITION, HEIGHT, WEIGHT, BACK_NO)
       VALUES ('2002007', '박지성', 'K07', 'MF', 178, 73, 7);
1개의 행이 만들어졌다.
```

  - ✓ a 유형은 테이블의 칼럼을 정의할 수 있는데, 이때 칼럼의 순서는 테이블의 칼럼 순서와 매치할 필요는 없으며, 정의하지 않은 칼럼은 Default로 NULL 값이 입력된다.
  - ✓ 단, Primary Key나 Not NULL 로 지정된 칼럼은 NULL이 허용되지 않는다.

- INSERT (b type)
  - ✓ [예제] 선수 테이블에 이청용 선수의 전체 데이터를 입력한다.

[예제]

```
INSERT INTO PLAYER
VALUES ('2002010','이청용','K07','','BlueDragon','2002','MF','17',
        NULL,NULL,'1',180,69);
1개의 행이 만들어졌다.
```

  - ✓ b 유형은 모든 칼럼에 데이터를 입력하는 경우로 굳이 COLUMN_LIST를 언급하지 않아도 되지만, 칼럼의 순서대로 빠짐없이 데이터가 입력되어야 한다.
  - ✓ 데이터를 입력하는 경우 정의되지 않은 미지의 값인 E_PLAYER_NAME은 두개의 『''』 SINGLE QUOTATION을 붙여서 표현하거나, NATION이나 BIRTH_DATE의 경우처럼 NULL이라고 명시적으로 표현할 수 있다.

- UPDATE
  - ✓ UPDATE 다음에 수정되어야 할 칼럼이 존재하는 테이블명을 입력하고, SET 다음에 수정되어야 할 칼럼명과 해당 칼럼 수정값으로 변경이 이루어진다.

```
UPDATE  테이블명
SET     수정되어야 할 칼럼명 = 수정되기를 원하는 새로운 값;
```

  - ✓ [예제] 선수 테이블의 포지션을 일괄적으로 'MF'로 수정한다.

[예제]

```
UPDATE PLAYER
SET    POSITION = 'MF';
480개의 행이 수정되었다.
```

  - ✓ 일반적으로는5절에서 배울 WHERE 절을 사용하여 UPDATE 대상 행을 선별한다. WHERE 절을 사용하지 않는다면 테이블의 전체 데이터가 수정된다.

- DELETE
  - ✓ DELETE FROM 다음에 삭제를 원하는 자료가 저장되어 있는 테이블명을 입력하고 실행한다. 이때 FROM 문구는 생략이 가능한 키워드이다.

```
DELETE [FROM] 삭제를 원하는 정보가 있는 테이블명;
```

  - ✓ [예제] 선수 테이블의 데이터를 전부 삭제한다.

[예제]

```
DELETE FROM PLAYER;
480개의 행이 삭제되었다.
```

  - ✓ 일반적으로는5절에서 배울 WHERE 절을 사용하여 DELETE 대상 행을 선별한다. WHERE 절을 사용하지 않는다면 테이블의 전체 데이터가 삭제된다.

- SELECT
  - ✓ 조회하기를 원하는 칼럼명을 SELECT 다음에 콤마 구분자(,)로 구분하여 나열하고, FROM 다음에 해당 칼럼이 존재하는 테이블명을 입력하여 실행시킨다.

```
SELECT [ALL/DISTINCT] 보고싶은 칼럼명, 보고싶은 칼럼명, . . .
FROM   해당 칼럼들이 있는 테이블명;

- ALL      : 중복된 데이터가 있어도 모두 출력한다.
             Default 옵션이므로 별도로 표시하지 않아도 된다.
- DISTINCT : 중복된 데이터가 있는 경우 1건으로 처리해서 출력한다.
```

※ 별도 제공한 SQL SCRIPT를 통해 모든 테이블의 데이터를 새롭게 생성한 후, 이후 강의 내용 진행 권고함

- SELECT
  - ✓ [예제] 입력한 선수들의 데이터를 조회한다.

[예제]

```
SELECT PLAYER_ID, PLAYER_NAME, TEAM_ID, POSITION, HEIGHT, WEIGHT, BACK_NO
FROM   PLAYER;
```

실행 결과

PLAYER_ID	PLAYER_NAME	TEAM_ID	POSITION	BACK_NO	HEIGHT	WEIGHT
2007155	정경량	K05	MF	19	173	65
2010025	정은익	K05	MF	35	176	63
2012001	레오마르	K05	MF	5	183	77
2008269	명재용	K05	MF	7	173	63
2007149	변재섭	K05	MF	11	170	63
2012002	보띠	K05	MF	10	174	68
…	…	…	…	…	…	…

480개의 행이 선택되었다.

- SELECT
  - ✓ [예제] 선수 테이블의 포지션 정보를 ALL과 DISTINCT 옵션으로 확인해본다.

[예제] 및 실행 결과

```
SELECT ALL POSITION
FROM   PLAYER;

ALL은 생략 가능한 키워드이므로
아래 SQL 문장도 같은 결과를
출력한다.

SELECT POSITION
FROM   PLAYER;
480개의 행이 선택되었다.
```

[예제] 및 실행 결과

```
SELECT DISTINCT POSITION
FROM   PLAYER;

POSITION

GK
DF
FW
MF
5개의 행이 선택되었다.
```

※ 우측 예제는 POSITION에 데이터가 없는 NULL 값이 포함되어서 5개의 결과가 출력되었다.

- SELECT
  - ✓ 해당 테이블의 모든 칼럼 정보를 보고 싶을 경우에는 와일드카드로 애스터리스크(*)를 사용하여 조회할 수 있다.

```
SELECT  *
FROM    테이블명;
```

  - ✓ 실행 결과 화면을 보면 칼럼 레이블(LABLE)이 맨 위에 보여지고, 레이블 밑에 점선이 보여진다. 실질적인 자료는 다음 줄부터 시작된다.
  - ✓ 레이블은 기본적으로 대문자로 보여지고, 첫 라인에 보여지는 레이블의 정렬은 다음과 같다.
    - 좌측 정렬 : 문자 및 날짜 데이터
    - 우측 정렬 : 숫자 데이터

- SELECT
  - ✓ [예제] 입력한 선수들의 정보를 모두 조회한다.

[예제]

```
SELECT *
FROM   PLAYER;
```

실행 결과

PLAYER_ID	PLAYER_NAME	TEAM_ID	E_PLAYER_NAME	NICKNAME	JOIN_YYYY	POSITION	BACK_NO	NATION	BIRTH_DATE	SOL AR	HEIGHT	WEIGHT
2007155	정경량	K05	JEONG, KYUNGRYANG		2006	MF	19		1983-12-22	1	173	65
2010025	정은익	K05				MF	35		1991-03-09	1	176	63
2012001	레오마르	K05	Leomar Leiria	레오	2012	MF	5		1981-06-26	1	183	77
2008269	명재용	K05	MYUNG, JAEYOENG		2007	MF	7		1983-02-26	2	173	63
2007149	변재섭	K05	BYUN, JAESUB	작은뱅크	2007	MF	11		1985-09-17	2	170	63
2012002	보띠	K05	Raphael JoseBotti Zacarias Sena	Botti	2012	MF	10		1991-02-23	1	174	68
...	...	...	...	...	...	...	...	...	...	...	...	...

480개의 행이 선택되었다.

※ 본 교재는 가독성을 위해 일부 칼럼의 좌정렬, 우정렬을 무시한 경우가 있음

- SELECT – ALIAS
  - ✓ 조회된 결과에 일종의 별명(ALIAS, ALIASES)을 부여해서 칼럼 레이블을 변경할 수 있다.
    - 칼럼명 바로 뒤에 온다.
    - 칼럼명과 ALIAS 사이에 AS, as 키워드를 사용할 수도 있다. (option)

[예제]

```
SELECT PLAYER_NAME AS 선수명, POSITION AS 위치, HEIGHT AS 키,
       WEIGHT AS 몸무게
FROM   PLAYER;

칼럼 별명에서 AS를 꼭 사용하지 않아도 되므로, 아래 SQL은 위 SQL과 같은
결과를 출력한다.

SELECT PLAYER_NAME 선수명, POSITION 위치, HEIGHT 키, WEIGHT 몸무게
FROM   PLAYER;
```

- SELECT – ALIAS
  - ✓ ALIAS 사용시 이중 인용부호(Double Quotation)는 ALIAS가 공백, 특수 문자를 포함할 경우와 대소문자 구분이 필요할 경우 사용된다.

[예제]

```
SELECT PLAYER_NAME "선수 이름", POSITION "그라운드 포지션",
       HEIGHT "키", WEIGHT "Weight"
FROM   PLAYER;
```

실행 결과

선수 이름	그라운드 포지션	키	Weight
정경량	MF	173	65
정은익	MF	176	63
레오마르	MF	183	77

...
480개의 행이 선택되었다.

- SELECT – 산술 연산자
  - ✓ 산술 연산자는 NUMBER와 DATE 자료형에 대해 적용되며 일반적으로 수학에서의 4칙 연산과 동일하다.
  - ✓ 우선순위를 위한 괄호 적용이 가능하다.
  - ✓ 수학에서와 같이 (), *, /, +, – 의 우선 순위를 가진다.
  - ✓ 적절한 ALIAS를 새롭게 부여하는 것이 좋다.

산술연산자	설 명
( )	연산자 우선 순위를 변경하기 위한 괄호 (괄호 안의 연산이 우선된다)
*	곱하기
/	나누기
+	더하기
–	빼기

- SELECT – 산술 연산자
  - ✓ [예제] 선수들의 키와 몸무게를 이용해서 BMI(Body Mass Index) 비만 지수를 측정한다.

[예제]

```
SELECT PLAYER_NAME 이름, ROUND(WEIGHT/((HEIGHT/100)*(HEIGHT/100)),2) "BMI 비만지수"
FROM   PLAYER;
```

실행 결과

이름	BMI 비만지수
정경량	21.72
정은익	20.34
레오마르	22.99
김수철	23.26
임다한	20.45

...
480개의 행이 선택되었다.

※ 예제에서 사용된 ROUND() 함수는 반올림을 위한 내장 함수로써 6절에서 학습한다.

- SELECT – 합성 연산자
  - ✓ 문자와 문자를 연결하는 합성(CONCATENATION) 연산자를 사용하면 별도의 프로그램 도움 없이도 SQL 문장만으로도 유용한 리포트를 출력할 수 있다. 합성(CONCATENATION) 연산자의 특징은 다음과 같다.
    - 문자와 문자를 연결하는 경우 2개의 수직 바(||)에 의해 이루어진다. (Oracle)
    - 문자와 문자를 연결하는 경우 + 표시에 의해 이루어진다. (SQL Server)
    - 두 벤더 모두 공통적으로 CONCAT(string1, string2) 함수를 사용할 수 있다.
    - 칼럼과 문자 또는 다른 칼럼과 연결시킨다.
    - 문자 표현식의 결과에 의해 새로운 칼럼을 생성한다.

- SELECT – 합성 연산자
  - ✓ [예제] 다음과 같은 선수들의 출력 형태를 만들어 본다.

    출력 형태) 선수명 선수, 키 cm, 몸무게 kg

    예) 박지성 선수, 176 cm, 70 kg

[예제] - Oracle

```
SELECT PLAYER_NAME || '선수,' || HEIGHT || 'cm,' || WEIGHT || 'kg' 체격정보
FROM   PLAYER;
```

[예제] - SQL Server

```
SELECT PLAYER_NAME  + "선수, " + HEIGHT + "cm, "  + WEIGHT  + "kg" 체격정보
FROM   PLAYER;
```


- SELECT – 합성 연산자

실행 결과

체격정보
정경량선수,173cm,65kg
정은익선수,176cm,63kg
레오마르선수,183cm,77kg
명재용선수,173cm,63kg
변재섭선수,170cm,63kg
보띠선수,174cm,68kg
비에라선수,176cm,73kg
서동원선수,184cm,78kg
안대현선수,179cm,72kg
...
480개의 행이 선택되었다.

- **DUAL 테이블**
  - ✓ Oracle은 SELECT 절과 FROM 절 두개의 절이 SELECT SQL 문장의 필수 절로 지정되어 있으므로, 사용자 테이블이 필요 없는 SQL 문장의 경우에도 필수적으로 DUAL이라는 테이블을 FROM 절에 지정한다.
  - ✓ DUAL 테이블의 특성은 다음과 같다.
    - 사용자 SYS가 소유하며 모든 사용자가 액세스 가능한 테이블이다.
    - SELECT ~ FROM ~ 의 형식을 갖추기 위한 일종의 DUMMY 테이블이다.
    - DUMMY라는 문자열 유형 칼럼에 'X'라는 값이 들어 있는 행을 1건 포함하고 있다.
  - ✓ 반면 사이베이스(Sybase)나 MS SQL Server의 경우에는 SELECT 절만으로도 SQL 문장이 수행 가능하도록 정의하였기 때문에 DUAL이란 DUMMY 테이블이 필요 없다. 물론 사이베이스(Sybase)나 MS SQL Server의 경우에도 사용자 테이블의 칼럼을 사용할 때는 FROM 절이 필수적으로 사용되어야 한다.

Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제3절 DML 핵심정리 및 연습문제

핵심정리 제3절 DML

- INSERT INTO 문장에 의해 데이터가 입력되고,
- UPDATE 문장에 의해 데이터가 수정되고,
- DELETE 문장에 의해 데이터가 삭제되고,
- SELECT 문장에 의해 데이터가 조회된다.

- DDL보다는 DML이 더 많이 사용되고,
  DML 중 SELECT 문장이 가장 많이 사용된다.

연습문제 제3절 DML

문제. 데이터를 입력하기 위해 사용하는 SQL 명령어는 무엇인가?

① CREATE
② UPDATE
③ INSERT
④ ALTER

연습문제 제3절 DML

문제. 데이터를 입력하기 위해 사용하는 SQL 명령어는 무엇인가?

① CREATE
② UPDATE
③ INSERT
④ ALTER

정답 : ③
해설 :
데이터를 입력하기 위해서 "INSERT" 명령어를 사용한다..

Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제4절 TCL 학습하기

- **트랜잭션**
  - ✓ 트랜잭션은 데이터베이스의 논리적 연산단위이다.
  - ✓ 트랜잭션(TRANSACTION)이란 밀접히 관련되어 분리될 수 없는 한 개 이상의 데이터베이스 조작을 가리킨다.
  - ✓ 하나의 트랜잭션에는 하나 이상의 SQL 문장이 포함된다.
  - ✓ 트랜잭션은 의미적으로 분할할 수 없는 최소의 단위이다.
  - ✓ 그렇기 때문에 전부 적용하거나 전부 취소한다.
  - ✓ 즉, TRANSACTION은 ALL OR NOTHING의 개념인 것이다.
  - ✓ [사례] 계좌이체라는 작업단위는 두개의 스텝이 모두 성공적으로 완료되었을 때 종료된다. 둘 중 하나라도 실패할 경우 계좌이체는 원래의 금액을 유지하고 있어야만 한다.
    - STEP1. 100번 계좌의 잔액에서 10,000원을 뺀다.
    - STEP2. 200번 계좌의 잔액에 10,000원을 더한다.

- **트랜잭션 특성**

특성	설명
원자성 (atomicity)	트랜잭션에서 정의된 연산들은 모두 성공적으로 실행되던지 아니면 전혀 실행되지 않은 상태로 남아 있어야 한다. (all or nothing)
일관성 (consistency)	트랜잭션이 실행 되기 전의 데이터베이스 내용이 잘못 되어 있지 않다면 트랜잭션이 실행된 이후에도 데이터베이스의 내용에 잘못이 있으면 안된다.
고립성 (isolation)	트랜잭션이 실행되는 도중에 다른 트랜잭션의 영향을 받아 잘못된 결과를 만들어서는 안된다.
지속성 (durability)	트랜잭션이 성공적으로 수행되면 그 트랜잭션이 갱신한 데이터베이스의 내용은 영구적으로 저장된다.

  - ✓ 트랜잭션의 특성(특히 원자성)을 충족하기 위해 데이터베이스는 다양한 레벨의 잠금(LOCK) 기능을 제공하고 있는데, 잠금은 기본적으로 트랜잭션이 수행하는 동안 특정 데이터에 대해서 다른 트랜잭션이 동시에 접근하지 못하도록 제한하는 기법이다. (3권 참조)

제4절 TCL (Transaction Control Language)

- TCL
  - ✓ 트랜잭션을 콘트롤하는 TCL(TRANSACTION CONTROL LANGUAGE)
    - 커밋(COMMIT) :
      "변경된 데이터를 데이터베이스에 영구적으로 반영하라"
    - 롤백(ROLLBACK) :
      "변경된 데이터가 문제가 있으니 변경 전 데이터로 복귀하라"
    - 저장점(SAVEPOINT) :
      "데이터 변경을 사전에 지정한 저장점까지만 롤백하라"
  - ✓ COMMIT과 ROLLBACK을 사용함으로써 다음과 같은 효과를 볼 수 있다.
    - 데이터 무결성 보장
    - 영구적인 변경을 하기 전에 데이터의 변경 사항 확인 가능
    - 논리적으로 연관된 작업을 그룹핑하여 처리 가능
  - ✓ COMMIT이나 ROLLBACK(트랜잭션 철회) 이전 데이터 상태는 다음과 같다.
    - 현재 사용자는 SELECT 문장으로 결과를 확인 가능하다.
    - 다른 사용자는 현재 사용자가 수행한 명령의 결과를 볼 수 없다.
    - 변경된 행은 잠금(LOCKING)이 설정되어 다른 사용자가 변경할 수 없다.

제4절 TCL (Transaction Control Language)

- COMMIT
  - ✓ COMMIT 명령어는 INSERT 문장, UPDATE 문장, DELETE 문장 사용 후에 일련의 변경 작업이 완료되었음을 데이터베이스에 알려 주기 위해 사용한다.

```
INSERT INTO PLAYER
  (PLAYER_ID, TEAM_ID,
    PLAYER_NAME, POSITION, HEIGHT,
    WEIGHT, BACK_NO)
VALUES ('1997035', 'K02', '이운재',
        'GK', 182, 82, 1);
1개의 행이 만들어졌다.

UPDATE PLAYER
SET    HEIGHT = HEIGHT + 10;
10개의 행이 수정되었다.

COMMIT;
커밋이 완료되었다.
```

- COMMIT (트랜잭션 완료) 이후의 데이터 상태는 다음과 같다.

  - 데이터에 대한 변경 사항이 데이터베이스에 반영된다.
  - 이전 데이터는 영원히 잃어버리게 된다.
    (별도 로그 보관시 복구 가능)
  - 모든 사용자는 결과를 볼 수 있다.
  - 관련된 행에 대한 잠금(LOCKING)이 풀리고, 다른 사용자들이 행을 조작할 수 있게 된다.

- **ROLLBACK**
  - ✓ 테이블 내 입력한 데이터나, 수정한 데이터, 삭제한 데이터에 대하여 COMMIT 이전에는 변경 사항을 취소할 수 있는데 데이터베이스에서는 롤백(ROLLBACK)기능을 사용한다.

```
UPDATE PLAYER
SET    HEIGHT = HEIGHT + 10;
10개의 행이 수정되었다.

DELETE FROM PLAYER;
10개의 행이 삭제되었다.

ROLLBACK;
롤백이 완료되었다.
```

- ROLLBACK (트랜잭션 취소) 이후의 데이터 상태는 다음과 같다.

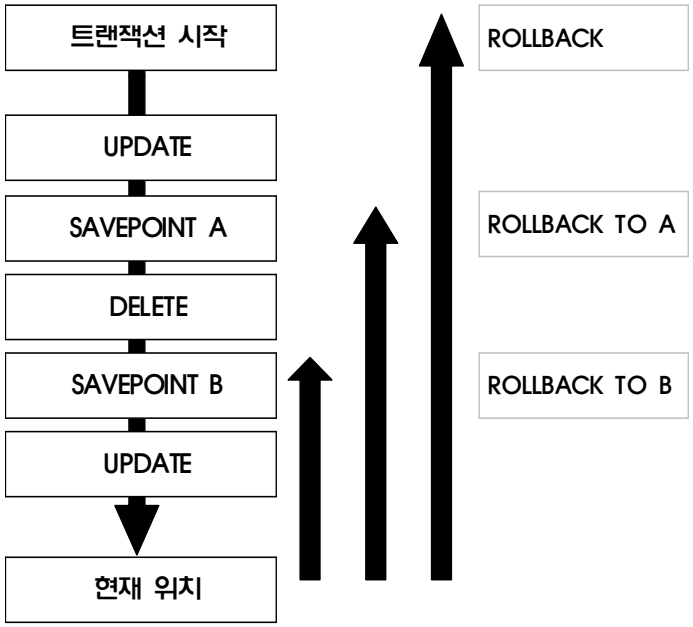
  - 데이터에 대한 변경 사항은 취소 된다.
  - 이전 데이터는 다시 재저장 된다.
  - 관련된 행에 대한 잠금(LOCKING)이 풀리고, 다른 사용자들이 행을 조작할 수 있게 된다.

- **SAVEPOINT**
  - ✓ 트랜잭션에 포함된 전체 작업(스텝)을 롤백하는 것이 아니라 현 시점에서 미리 지정한 SAVEPOINT(저장점)까지 트랜잭션의 일부만 롤백할 수 있다.
  - ✓ 복수의 저장점을 정의할 수 있으며, 동일이름으로 저장점을 정의했을 때는 나중에 정의한 저장점이 유효하다.

```
트랜잭션 내에서 SVPT1이라는 저장점을 정의한다.
SAVEPOINT SVPT1;
```

```
저장점까지 롤백할 때는 ROLLBACK 뒤에 저장점 명을 지정한다.
ROLLBACK TO SVPT1;
```

- SAVEPOINT



✓저장점 A로 되돌리고 나서 다시 B와 같이 미래 방향으로 되돌릴 수는 없다.

✓일단 특정 저장점까지 롤백하면 그 저장점 이후에 설정한 저장점이 무효가 되기 때문이다. 즉, 'ROLLBACK TO A'를 실행한 시점에서 저장점 A 이후에 정의한 저장점 B는 존재하지 않는다.

✓저장점 지정 없이 ROLLBACK을 실행했을 경우 반영 안 된 모든 변경 사항을 취소하고 트랜잭션 시작 위치로 되돌아간다

- COMMIT과 ROLLBACK
  - ✓ 트랜잭션은 트랜잭션의 대상이 되는 SQL문을 실행하면 자동으로 시작되고, COMMIT 또는 ROLLBACK을 실행한 시점에서 종료된다.
  - ✓ COMMIT과 ROLLBACK을 사용함으로써 다음과 같은 효과를 볼 수 있다.
    - 논리적으로 연관된 작업을 그룹핑하여 처리 가능
    - 영구적인 변경을 하기 전에 데이터의 변경 사항 확인 가능
    - 데이터 무결성 보장
  - ✓ 자동 커밋이 발생하는 경우
    - CREATE, ALTER, DROP, RENAME, TRUNCATE TABLE 등 DDL 문장을 실행하면 그 전후 시점에 자동으로 커밋된다. 부연하면, DML 문장 이후에 커밋 없어도 DDL 문장이 실행되면 DDL 수행 전에 자동으로 커밋된다.
    - 데이터베이스를 정상적으로 접속 종료하면 트랜잭션이 자동 커밋된다.
    - 애플리케이션의 이상 종료로 데이터베이스와의 접속이 단절되었을 때는 트랜잭션이 자동 롤백된다.

- SQL Server COMMIT 모드
  - ✓ Oracle은 DML을 실행하는 경우 DBMS가 트랜잭션을 내부적으로 실행하며 DML 문장 수행 후 사용자가 임의로 COMMIT 혹은 ROLLBACK을 수행해 주어야 트랜잭션이 종료된다. (일부 툴에서는 AUTO COMMIT을 옵션으로 선택할 수 있다)
  - ✓ SQL Server는 기본적으로 AUTO COMMIT 모드이기 때문에 DML 수행 후 사용자가 COMMIT이나 ROLLBACK을 처리할 필요가 없다. DML 구문이 성공이면 자동으로 COMMIT이 되고 오류가 발생할 경우 자동으로 ROLLBACK 처리된다.

- SQL Server 3가지 COMMIT 방식
  - ✓ AUTO COMMIT
    SQL Server의 기본 방식이며, DML, DDL 단위 SQL을 수행할 때마다 DBMS가 트랜잭션을 컨트롤하는 방식이다. 명령어가 성공적으로 수행되면 자동으로 COMMIT을 수행하고 오류가 발생하면 자동으로 ROLLBACK을 수행한다.
  - ✓ 암시적 트랜잭션
    Oracle과 같은 방식으로 처리된다. 즉, 트랜잭션의 시작은 DBMS가 처리하고 트랜잭션의 끝은 사용자가 명시적으로 COMMIT 또는 ROLLBACK으로 처리한다. 인스턴스 단위 또는 세션 단위로 설정할 수 있다. 인스턴스 단위로 설정하려면 서버 속성 창의 연결화면에서 기본연결 옵션 중 암시적 트랜잭션에 체크를 해주면 된다. 세션 단위로 설정하기 위해서는 세션 옵션 중 SET IMPLICIT TRANSACTION ON을 사용하면 된다.
  - ✓ 명시적 트랜잭션
    트랜잭션의 시작과 끝을 모두 사용자가 명시적으로 지정하는 방식이다. BEGIN TRANSACTION (BEGIN TRAN 구문도 가능)으로 트랜잭션을 시작하고 COMMIT TRANSACTION(TRANSACTION은 생략 가능) 또는 ROLLBACK TRANSACTION(TRANSACTION은 생략 가능)으로 트랜잭션을 종료한다. ROLLBACK 구문을 만나면 최초의 BEGIN TRANSACTION 시점까지 모두 ROLLBACK이 수행된다.

Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제4절 TCL 핵심정리 및 연습문제

핵심정리 제4절 TCL

- TRANSACTION은 ALL OR NOTHING 개념이다.
- COMMIT 문장을 통해 트랜잭션을 완료하고,
- ROLLBACK 문장을 통해 트랜잭션을 철회하며,
- SAVEPOINT/SAVE TRANSACTION을 통해 트랜잭션의 일부만 ROLLBACK 할 수 있다.

문제. Commit과 Rollback의 장점으로 적합하지 않은 것은 무엇인가?

① 데이터 무결성을 보장한다.
② 영구적인 변경을 하기 전에 데이터의 변경 사항 확인 가능
③ 영구적인 변경을 할 수 없게 한다.
④ 논리적으로 연관된 작업을 그룹핑하여 처리 가능

문제. Commit과 Rollback의 장점으로 적합하지 않은 것은 무엇인가?

① 데이터 무결성을 보장한다.
② 영구적인 변경을 하기 전에 데이터의 변경 사항 확인 가능
③ 영구적인 변경을 할 수 없게 한다.
④ 논리적으로 연관된 작업을 그룹핑하여 처리 가능

정답 : ③
해설 :
Commit과 Rollback의 장점은 다음과 같다.
- 데이터 무결성 보장
- 영구적인 변경을 하기 전에 데이터의 변경 사항 확인 가능
- 논리적으로 연관된 작업을 그룹핑하여 처리 가능

Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제5절 WHERE 절 학습하기

제5절 WHERE 절

- **WHERE 절**
  - ✓ 사용자들은 자신이 원하는 자료만을 검색하기 위해서 SQL 문장에 WHERE 절을 이용하여 자료들에 대하여 제한할 수 있다.
  - ✓ 두 개 이상의 테이블에 대한 INNER JOIN을 지원하는 다른 기능도 가지고 있다. (9절에서 설명)
  - ✓ WHERE 절은 조회하려는 데이터에 특정 조건을 부여할 목적으로 사용하기 때문에 FROM 절 뒤에 오게 된다.

```
SELECT [DISTINCT/ALL] 칼럼명 [ALIAS명]
FROM   테이블명
WHERE  조건식;

조건식 구성 :
칼럼(Column)명   (보통 조건식 좌측 위치) +  비교 연산자 +
문자/숫자/표현식 (보통 조건식 우측 위치) or 비교 칼럼명 (JOIN 사용시)
```

- 연산자 종류

구분	연산자	연산자의 의미
비교 연산자	=	같다.
	>	보다 크다.
	>=	보다 크거나 같다.
	<	보다 작다.
	<=	보다 작거나 같다.
SQL 연산자	BETWEEN a AND b	a와 b의 값 사이에 있으면 된다.(a와 b 값이 포함됨)
	IN (list)	리스트에 있는 값 중에서 어느 하나라도 일치하면 된다.
	LIKE '비교문자열'	비교문자열과 형태가 일치하면 된다.(%, _ 사용)
	IS NULL	NULL 값인 경우
논리 연산자	AND	앞에 있는 조건과 뒤에 오는 조건이 참(TRUE)이 되면 결과도 참(TRUE)이 된다. 즉, 앞의 조건과 뒤의 조건을 동시에 만족해야 한다.
	OR	앞의 조건이 참(TRUE)이거나 뒤의 조건이 참(TRUE)이 되어야 결과도 참(TRUE)이 된다. 즉, 앞뒤의 조건 중 하나만 참(TRUE)이면 된다.
	NOT	뒤에 오는 조건에 반대되는 결과를 되돌려 준다.

- 연산자 종류

구분	연산자	연산자의 의미
부정 비교 연산자	!=	같지 않다.
	^=	같지 않다.
	<>	같지 않다. (ISO 표준, 모든 운영체제에서 사용 가능)
	NOT 칼럼명 =	~와 같지 않다.
	NOT 칼럼명 >	~보다 크지 않다.
부정 SQL 연산자	NOT BETWEEN a AND b	a와 b의 값 사이에 있지 않다. (a, b 값을 포함하지 않는다) a보다 작거나 b보다 크거나 조건으로 바꿀 수 있다.
	NOT IN (list)	list 값과 일치하지 않는다.
	IS NOT NULL	NULL 값을 갖지 않는다.

- 연산자 우선 순위

연산 우선 순위	설 명
1	괄호 ()
2	NOT 연산자
3	비교 연산자, SQL 비교 연산자
4	AND
5	OR

✓ 실수하기 쉬운 비교 연산자와 논리 연산자의 경우 괄호를 사용해서 우선 순위를 표시하는 것을 권고한다.

- WHERE 조건

소속팀이 삼성블루윙즈이거나 전남드래곤즈에 소속된 선수들이어야 하고,
포지션이 미드필더(MF: MidFielders)이어야 한다.
키는 170 센티미터 이상이고 180 이하여야 한다.

1) 소속팀코드 = 삼성블루윙즈팀 코드(K02)
2) 소속팀코드 = 전남드래곤즈팀 코드(K07)
3) 포지션 = 미드필더 코드(MF)
4) 키 >= 170 센티미터
5) 키 <= 180 센티미터

✓ CHAR 변수나 VARCHAR2와 같은 문자형 타입을 가진 칼럼을 특정 값과 비교하기 위해서는 인용 부호(작은 따옴표, 큰 따옴표)로 묶어서 비교 처리를 해야 한다.

✓ NUMERIC과 같은 숫자형 형태의 값은 인용부호를 사용하지 않는다.

- WHERE 조건

[예제]

```
SELECT PLAYER_NAME 선수이름, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE (TEAM_ID  = 'K02' OR TEAM_ID = 'K07')
AND    POSITION = 'MF'
AND    HEIGHT >= 170
AND    HEIGHT <= 180;
```

- ( )가 없다면 잘못된 결과가 나옴
- 연산자 간 우선순위 중요함
- 문자형의 경우 홑따옴표 '' 사용

[예제]

```
SELECT PLAYER_NAME 선수이름, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  WHERE TEAM_ID IN ('K02','K07')
AND    POSITION = 'MF'
AND    HEIGHT BETWEEN 170 AND 180 ;

33개의 행이 선택되었다.
```

- 논리연산자 OR을 SQL 연산자인 IN으로 변경 가능함
- 비교연산자 >=, <= 조건을 SQL연산자인 BETWEEN 조건으로 변경 가능함

- WHERE 조건 – IN

✓ [예제] 사원 테이블에서 JOB이 MANAGER이면서 20번 부서에 속하거나, JOB이 CLERK이면서 30번 부서에 속하는 사원의 정보를 IN 연산자의 다중 리스트를 이용해 출력하라.

[예제]

```
SELECT ENAME, JOB, DEPTNO
FROM   EMP
WHERE (JOB, DEPTNO) IN (('MANAGER',20),('CLERK',30));
```

실행 결과

ENAME	JOB	DEPTNO
JONES	MANAGER	20
JAMES	CLERK	30

2개의 행이 선택되었다.

- WHERE 조건 – LIKE
  - ✓ 와일드 카드의 종류

와일드 카드	설명
%	0개 이상의 어떤 문자를 의미한다.
_	1개인 단일 문자를 의미한다.

[예제] "장"씨 성을 가진 선수들의 정보를 조회하는 WHERE 절을 작성한다.

```
SELECT PLAYER_NAME 선수이름, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  PLAYER_NAME LIKE '장%';
```

[예제] 포지션 데이터 2번째 자리의 값이 'F' 인 선수 정보를 조회하라.

```
SELECT PLAYER_NAME 선수이름, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  PLAYER_NAME LIKE  '_F%';
```


- WHERE 조건 – IS NULL
  - ✓ NULL : 아직 정의되지않은 미지의 값, 또는 현재 알수 없는 값
  - ✓ ' '(공백, ASCII 32)이나 0(Zero, ASCII 48)와는 다른 값이다.
  - ✓ NULL 값과의 수치연산은 NULL 값을 리턴한다.
  - ✓ NULL 값과의 비교연산은 거짓(FALSE)를 리턴한다.

    cf) 오라클의 경우 문법적으로 '= NULL' 조건에서 오류가 발생하지는 않지만, 데이터는 1건도 검색되지 않는다.
  - ✓ 어떤 값과 비교할 수 없으며,

    특정 값보다 크다, 작다라고 표현할 수 없다.
  - ✓ NULL 값의 비교 연산은 IS NULL, IS NOT NULL 이라는 정해진 문구를 사용해야 제대로 된 결과를 얻을 수 있다. (ANSI 기준)

• WHERE 조건 – IS NULL

[예제] 및 실행 결과

```
SELECT PLAYER_NAME 선수이름, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  POSITION = NULL;
선택된 레코드가 없다.
```

• POSITION = NULL 은 원하는 결과를 출력할 수 없다.

[예제]

```
SELECT PLAYER_NAME 선수이름, POSITION 포지션, TEAM_ID
FROM   PLAYER
WHERE POSITION IS NULL;
```

실행 결과

선수이름	포지션	TEAM_ID
정학범		K08
안익수		K08
차상광		K08

3개의 행이 선택되었다.

• WHERE 조건 – 부정연산자

✓ [예제] 삼성블루윙즈 소속인 선수들 중에서 포지션이 미드필더(MF:MidFielders)가 아니고, 키가 175 센티미터 이상 185 센티미터 이하가 아닌 선수들의 자료를 찾는다.

[예제]

```
SELECT  PLAYER_NAME 선수이름, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키
FROM    PLAYER
WHERE   TEAM_ID  = 'K02'
AND NOT POSITION = 'MF'
AND NOT HEIGHT BETWEEN 175 AND 185;
```

✓ [예제] 국적(NATION) 칼럼의 경우 내국인들은 별도 데이터를 입력하지 않았다(NULL). 국적 칼럼이 NULL이 아닌 선수와 국적을 표시하라(결과적으로 외국인 데이터 출력).

[예제]

```
SELECT PLAYER_NAME 선수이름, NATION 국적
FROM   PLAYER
WHERE  NATION IS NOT NULL;
27개의 행이 선택되었다.
```

- **WHERE 조건 – ROWNUM**
  - ✓ Oracle의 ROWNUM은 Pseudo Column(유사 칼럼, 사용자가 아닌 시스템이 관리하는)으로, 테이블이나 집합에서 원하는 만큼의 행만 가져오고 싶을 때 WHERE 절에서 사용할 수 있다.

한 건의 행만 가져오고 싶을 때는
- SELECT PLAYER_NAME FROM PLAYER WHERE ROWNUM = 1; 이나
- SELECT PLAYER_NAME FROM PLAYER WHERE ROWNUM <= 1; 이나
- SELECT PLAYER_NAME FROM PLAYER WHERE ROWNUM < 2; 처럼 사용할 수 있다.

두 건 이상의 N 행을 가져오고 싶을 때는 ROWNUM = N; 처럼 사용할 수 없으며
- SELECT PLAYER_NAME FROM PLAYER WHERE ROWNUM <= N; 이나
- SELECT PLAYER_NAME FROM PLAYER WHERE ROWNUM < N+1; 처럼 출력되는 행의 개수를 지정할 수 있다.

※ 추가적인 ROWNUM의 용도로는 테이블 내의 유니크한 키나 인덱스 값을 만들 수 있다.
UPDATE MY_TABLE SET COLUMN1 = ROWNUM;

- **WHERE 조건 – TOP**
  - ✓ SQL Server는 TOP 절을 사용하여 결과 집합으로 출력되는 행 수를 제한할 수 있다.

TOP (Expression) [PERCENT] [WITH TIES]
- Expression : 반환할 행의 수를 지정하는 숫자이다.
- PERCENT : 쿼리 결과 집합에서 처음 Expression%의 행만 반환됨을 나타낸다.
- WITH TIES : ORDER BY 절이 지정된 경우에만 사용할 수 있으며, TOP N [PERCENT]의 마지막 행과 같은 값이 있는 경우 추가 행이 출력되도록 선택할 수 있다.

한 건의 행만 가져오고 싶을 때는
- SELECT TOP(1) PLAYER_NAME FROM PLAYER; 처럼 사용할 수 있다.

두 건 이상의 N 행을 가져오고 싶을 때는
- SELECT TOP(N) PLAYER_NAME FROM PLAYER; 처럼 출력되는 행의 개수를 지정할 수 있다.

※ ORDER BY절 사용시 ROWNUM과 TOP의 차이점은 9절에서 추가 설명함

Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제5절 WHERE 절 핵심정리 및 연습문제

핵심정리 제5절 WHERE 절

- WHERE 조건절에 제한을 두어 원하는 자료만을 조회할 수 있다.
- WHERE 절에 사용되는 연산자는
  - ✓ 비교 연산자,
  - ✓ SQL 연산자,
  - ✓ 논리 연산자,
  - ✓ 부정 연산자가 있다.
- NULL 값의 비교 연산은 IS NULL, IS NOT NULL 이라는 정해진 문구를 사용해야 제대로 된 결과를 얻을 수 있다.
- 테이블이나 집합에서 원하는 만큼의 행만 가져오고 싶을 때 Oracle은 ROWNUM을 SQL Server는 TOP 기능을 사용할 수 있다.

연습문제 제5절 WHERE 절

문제. 다음 SQL 문장의 결과 출력되는 데이터는 무엇인가?

```
SELECT  PLAYER_NAME 선수명, E_PLAYER_NAME 선수영문명
FROM    PLAYER
WHERE   E_PLAYER_NAME LIKE '_A%';
```

① 선수의 영문 이름이 A로 시작하는 선수들의 이름
② 선수의 영문 이름이 A나 a로 시작하는 선수들의 이름
③ 선수의 영문 이름의 두번째 문자가 A인 선수들의 이름
④ 위치에 상관없이 선수의 영문 이름에 A를 포함하는 선수들의 이름

연습문제 제5절 WHERE 절

문제. 다음 SQL 문장의 결과 출력되는 데이터는 무엇인가?

```
SELECT  PLAYER_NAME 선수명, E_PLAYER_NAME 선수영문명
FROM    PLAYER
WHERE   E_PLAYER_NAME LIKE '_A%';
```

① 선수의 영문 이름이 A로 시작하는 선수들의 이름
② 선수의 영문 이름이 A나 a로 시작하는 선수들의 이름
③ 선수의 영문 이름의 두번째 문자가 A인 선수들의 이름
④ 위치에 상관없이 선수의 영문 이름에 A를 포함하는 선수들의 이름

정답: ③
해설 :
"_"와 "%"는 와일드카드(WILD CARD)로 하나의 글자, 또는 모든 문자를 대신하여 사용이 되므로 두 번째 문자가 대문자 A인 경우만 출력하게 된다.

Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제6절 FUNCTION 학습하기

제6절 FUNCTION

- **FUNCTION**
  - ✓ 함수는 다양한 기준으로 분류할 수 있는데, 벤더에서 제공하는 함수인 내장 함수(Built-in Function)와 사용자가 정의할 수 있는 함수(User Defined Function)로도 나눌 수 있다. (사용자 정의 함수는 2장 8절에서 학습함)
  - ✓ 내장 함수는 벤더별로 가장 큰 차이를 보이는 부분이지만, 핵심적인 기능들은 이름이나 사용법이 다르더라도 대부분의 데이터베이스가 공통적으로 제공하고 있다.

단일 행 함수, 6절 (Single-Row Function)	다중 행 함수 (Multi-Row Function) - 집계 함수(Aggregate Function), 7절 - 그룹 함수(Group Function), 2장5절 - 윈도우 함수(Window Function),2장6절

- FUNCTION
  - ✓ 일반적으로 함수는 입력되는 값이 많아도 출력은 하나만 된다는 M:1 관계라는 특징을 가지고 있다. (사용자 정의 함수의 경우 출력을 여러개 값이 나오게 할 수도 있다)
  - ✓ 단일 행 함수의 경우 단일 행 내에 있는 하나의 값 또는 여러 값이 입력 인수로 표현될 수 있다.
  - ✓ 단일 행 함수의 경우 각 행(Row)들에 대해 개별적으로 작용하여 데이터 값들을 조작하고, 각각의 행에 대한 조작 결과를 리턴한다.
  - ✓ 단일 행 함수의 경우 SELECT, WHERE, ORDER BY 절에 사용 가능하다.
  - ✓ 특별한 제약이 없다면 함수의 인자(Arguments)로 함수를 사용하는 함수의 중첩이 가능하다.

처리하는 데이터 형식에 따른 단일 행 함수 분류 (분류 기준은 다를 수 있음)
- 문자형
- 숫자형
- 날짜형
- 변환형
- NULL 관련 함수

- FUNCTION

종류	내용	함수의 예
문자형 함수	문자를 입력하면 문자나 숫자 값을 반환한다.	LOWER, UPPER, CONCAT, SUBSTR/SUBSTRING, LENGTH/LEN, LTRIM, RTRIM, TRIM, ASCII,
숫자형 함수	숫자를 입력하면 숫자 값을 반환한다.	ABS, MOD, ROUND, TRUNC, SIGN, CHR/CHAR, CEIL/CEILING, FLOOR, EXP, LOG, LN, POWER, SIN, COS, TAN
날짜형 함수	DATE 타입의 값을 연산한다.	SYSDATE/GETDATE, EXTRACT/DATEPART, TO_NUMBER(TO_CHAR(*d*,'DD'|'MM'|'YY')) / DAY|MONTH|YEAR
변환형 함수	문자, 숫자, 날짜형 값의 데이터 타입을 변환한다	TO_CHAR, TO_NUMBER, TO_DATE / CAST, CONVERT
NULL 관련함수	NULL을 처리하기 위한 함수	NVL/ISNULL, NULLIF, COALESCE

※ SUBSTR / SUBSTRING: 같은 기능을 하지만 다르게 표현되는 Oracle 내장 함수와 SQL Server 내장 함수를 표현함
함수에 대한 자세한 내용이나 버전에 따른 변경 내용은 벤더에서 제공하는 매뉴얼을 참조하기 바란다.

• 문자형 함수

문자형 함수	함수 설명
LOWER(문자열)	문자열의 알파벳 문자를 소문자로 바꾸어 준다.
UPPER(문자열)	문자열의 알파벳 문자를 대문자로 바꾸어 준다.
ASCII(문자)	문자나 숫자를 ASCII 코드 번호로 바꾸어 준다.
CHR/CHAR(ASCII번호)	ASCII 코드 번호를 문자나 숫자로 바꾸어 준다.
CONCAT (문자열1, 문자열2)	문자열1과 문자열2를 연결한다. 연결 연산자'||'(Oracle)나 '+'(SQL Server)와 동일하다.
SUBSTR/SUBSTRING (문자열, m[, n])	문자열 중 m위치에서 n개의 문자 길이에 해당하는 문자를 돌려준다. n이 생략되면 마지막 문자까지이다.
LENGTH/LEN(문자열)	문자열의 개수를 숫자값으로 돌려준다.
LTRIM (문자열 [, 지정문자])	문자열의 첫 문자부터 확인해서 지정 문자가 나타나면 해당 문자를 제거한다. (지정 문자가 생략되면 공백 값이 디폴트)
RTRIM (문자열 [, 지정문자])	문자열의 마지막 문자부터 확인해서 지정 문자가 나타나는 동안 해당 문자를 제거한다.(지정 문자가 생략되면 공백 값이 디폴트)
TRIM ([leading|trailing|both] 지정문자 FROM 문자열)	문자열에서 머리말, 꼬리말, 또는 양쪽에 있는 지정 문자를 제거한다. (leading | trailing | both 가 생략되면 both가 디폴트)

• 문자형 함수

문자형 함수 사용	결과 값 및 설명
LOWER('SQL Expert')	'sql expert'
UPPER('SQL Expert')	'SQL EXPERT'
ASCII('A')	65
CHR(65) / CHAR(65)	'A'
CONCAT('RDBMS',' SQL')	'RDBMS SQL'
SUBSTR('SQL Expert', 5, 3) SUBSTRING('SQL Expert', 5, 3)	'Exp'
LENGTH('SQL Expert') / LEN('SQL Expert')	ex) SELECT LENGTH('SQL Expert') FROM DUAL; ex) SELECT LEN('SQL Expert') ; 10
LTRIM('xxxYYZZxYZ','x') RTRIM('XXYYzzXYzz','z') TRIM('x' FROM 'xxYYZZxYZxx')	'YYZZxYZ' 'XXYYzzXY' 'YYZZxYZ'
RTRIM('XXYYZZXYZ ') → 공백 제거 및 CHAR와 VARCHAR 데이터 유형을 비교할 때 용이하게 사용된다.	'XXYYZZXYZ'

- 숫자형 함수

숫자형 함수	함수 설명
ABS(숫자)	숫자의 절대값을 돌려준다.
SIGN(숫자)	숫자가 양수인지, 음수인지 0 인지를 구별한다.
MOD(숫자1, 숫자2)	숫자1을 숫자2로 나누어 나머지 값을 리턴한다.
CEIL/CEILING(숫자)	숫자보다 크거나 같은 최소 정수를 리턴한다.
FLOOR(숫자)	숫자보다 작거나 같은 최대 정수를 리턴한다
ROUND(숫자 [, m])	숫자를 소수점 m자리에서 반올림하여 리턴한다. m이 생략되면 디폴트 값은 0 이다.
TRUNC(숫자 [, m])	숫자를 소수 m자리에서 잘라서 버린다. m이 생략되면 디폴트 값은 0 이다.
SIN, COS, TAN,...	숫자의 삼각함수 값을 리턴한다.
EXP(), POWER(), SQRT(), LOG(), LN()	숫자의 지수, 거듭 제곱, 제곱근, 자연 로그 값을 리턴한다.

- 숫자형 함수

숫자형 함수 사용	결과 값 및 설명
ABS(-15)	15
SIGN(-20) SIGN(0) SIGN(+20)	-1 0 1
MOD(7, 3)	1
CEIL(38.123) / CEILING(-38.123)	39 -38
FLOOR(38.123) FLOOR(-38.123)	38 -39
ROUND(38.5235, 3) ROUND(38.5235, 1) ROUND(38.5235, 0) ROUND(38.5235)	38.524 38.5 39 39 (Argument 0이 Default)
TRUNC(38.5235, 3) TRUNC(38.5235, 1) TRUNC(38.5235, 0) TRUNC(38.5235)	38.523 38.5 38 38 (Argument 0이 Default)

- 날짜형 함수

날짜형 함수	함수 설명
SYSDATE/GETDATE()	현재 날짜와 시각을 출력한다.
EXTRACT('YEAR'|'MONTH'|'DAY' from d) / DATEPART('YEAR'|'MONTH'|'DAY',*d*)	날짜 데이터에서 년/월/일 데이터를 출력할 수 있다. 시간/분/초도 가능함
TO_NUMBER(TO_CHAR(*d*,'YYYY')) / YEAR(*d*)	날짜 데이터에서 년 데이터를 출력할 수 있다. EXTRACT/DATEPART 년 옵션과 같은 기능이다.
TO_NUMBER(TO_CHAR(*d*,'MM')) / MONTH(*d*)	날짜 데이터에서 월 데이터를 출력할 수 있다. EXTRACT/DATEPART 월 옵션과 같은 기능이다.
TO_NUMBER(TO_CHAR(*d*,'DD')) / DAY(*d*)	날짜 데이터에서 일 데이터를 출력할 수 있다. EXTRACT/DATEPART 일 옵션과 같은 기능이다.

연산	결과	설명
날짜1 + 숫자	날짜	숫자만큼의 날수를 날짜에 더한다
날짜1 - 숫자	날짜	숫자만큼의 날수를 날짜에서 뺀다
날짜1 - 날짜2	날짜수	다른 하나의 날짜에서 하나의 날짜를 빼면 일수가 나온다.
날짜1 + 숫자/24	날짜	시간을 날짜에 더한다.

- 날짜형 함수
  - ✓ [예제] Oracle의 SYSDATE 함수와 SQL Server의 GETDATE() 함수를 사용하여 데이터베이스에서 사용하는 현재의 날짜 데이터를 확인한다.
    날짜 데이터는 시스템 구성에 따라 다양하게 표현될 수 있다.

[예제] 및 실행 결과
Oracle

```
SELECT SYSDATE FROM DUAL;

SYSDATE
--------
12/07/18
```

[예제] 및 실행 결과
SQL Server

```
SELECT GETDATE() ;

GETDATE()
-----------------------
2012-07-18 13:10:02.047
```

- **날짜형 함수**
  - ✓ [예제] 사원(EMP) 테이블의 입사일자에서 년,월,일 데이터를 각각 출력한다.

    아래 4개의 SQL 문장은 같은 기능을 하는 SQL 문장이다.

[예제] Oracle 함수

```
SELECT ENAME, HIREDATE,
       EXTRACT(YEAR  FROM HIREDATE) 입사년도,
       EXTRACT(MONTH FROM HIREDATE) 입사월,
       EXTRACT(DAY   FROM HIREDATE) 입사일
FROM   EMP;
```

[예제] SQL Server 함수

```
SELECT ENAME, HIREDATE,
       DATEPART(YEAR,  HIREDATE) 입사년도,
       DATEPART(MONTH, HIREDATE) 입사월,
       DATEPART(DAY,   HIREDATE) 입사일
FROM   EMP;
```

[예제] Oracle 함수

```
SELECT ENAME, HIREDATE,
       TO_NUMBER(TO_CHAR(HIREDATE,'YYYY')) 입사년도,
       TO_NUMBER(TO_CHAR(HIREDATE,'MM')) 입사월,
       TO_NUMBER(TO_CHAR(HIREDATE,'DD')) 입사일
FROM   EMP;
```

[예제] SQL Server 함수

```
SELECT ENAME, HIREDATE,
       YEAR(HIREDATE)  입사년도,
       MONTH(HIREDATE) 입사월,
       DAY(HIREDATE)   입사일
FROM   EMP;
```


- **변환형 함수**

숫자형 함수	함수 설명
ABS(숫자)	숫자의 절대값을 돌려준다.
SIGN(숫자)	숫자가 양수인지, 음수인지 0 인지를 구별한다.
MOD(숫자1, 숫자2)	숫자1을 숫자2로 나누어 나머지 값을 리턴한다.
CEIL/CEILING(숫자)	숫자보다 크거나 같은 최소 정수를 리턴한다.
FLOOR(숫자)	숫자보다 작거나 같은 최대 정수를 리턴한다
ROUND(숫자 [, m])	숫자를 소수점 m자리에서 반올림하여 리턴한다. m이 생략되면 디폴트 값은 0 이다.
TRUNC(숫자 [, m])	숫자를 소수 m자리에서 잘라서 버린다. m이 생략되면 디폴트 값은 0 이다.
SIN, COS, TAN,...	숫자의 삼각함수 값을 리턴한다.
EXP(), POWER(), SQRT(), LOG(), LN()	숫자의 지수, 거듭 제곱, 제곱근, 자연 로그 값을 리턴한다.

- **변환형 함수**
  - ✓ 변환형 함수는 특정 데이터 타입을 다양한 형식으로 출력하고 싶을 경우에 사용되는 함수이다. 변환형 함수는 크게 두 가지 방식이 있다.

명시적(Explicit) 데이터 유형 변환	데이터 변환형 함수로 데이터 유형을 변환하도록 명시해 주는 경우.
암시적(Implicit) 데이터 유형 변환	데이터베이스가 자동으로 데이터 유형을 변환하여 계산하는 경우.

  - ✓ 암시적 데이터 유형 변환의 경우 인덱스 미사용으로 인한 성능 저하가 발생할 수 있으며, 자동적으로 데이터베이스가 알아서 계산하지 않는 경우가 있어 에러를 발생할 수 있으므로 명시적인 데이터 유형 변환 방법을 사용하는 것이 바람직하다. (3권 참조)

- **명시적(Explicit) 데이터 유형 변환**

변환형 함수 - Oracle	함수 설명
TO_CHAR(숫자|날짜 [, FORMAT])	숫자나 날짜를 주어진 FORMAT 형태로 문자열 타입으로 변환한다.
TO_NUMBER(문자열)	문자열을 숫자로 변환한다.
TO_DATE(문자열 [, FORMAT])	문자열을 주어진 FORMAT 형태로 날짜 타입으로 변환한다.

변환형 함수 - SQL Server	함수 설명
CAST (expression AS data_type [(length)])	expression을 목표 데이터 유형으로 변환한다.
CONVERT (data_type [(length)], expression [, style])	expression을 목표 데이터 유형으로 변환한다.

- 변환형 함수

[예제] 및 실행 결과 Oracle

```
SELECT TO_CHAR(SYSDATE, 'YYYY/MM/DD') 숫자,
       TO_CHAR(SYSDATE, 'YYYY. MON, DAY') 문자
FROM   DUAL;

 숫자          문자
 2010-07-19    2010. 7월 , 월요일
```

[예제] 및 실행 결과 SQL Server

```
SELECT CONVERT(VARCHAR(10),GETDATE(),111) AS CURRENTDATE


CURRNETDATE
-----------
2012/07/19
```


- 변환형 함수

[예제] 및 실행 결과 Oracle

```
SELECT TO_CHAR(123456789/1200,'$999,999,999.99') 환율반영달러,
       TO_CHAR(123456789,'L999,999,999') 원화
FROM   DUAL;

 환율반영달러      원화
 $102,880.66       W123,456,789
```

[예제] 및 실행 결과 SQL Server

```
SELECT TEAM_ID, CAST(ZIP_CODE1 AS INT) + CAST(ZIP_CODE2 AS INT) 우편번호체크섬
FROM   TEAM;

15개의 행이 선택되었다.
```

- CASE 표현
  - ✓ CASE 표현은 IF-THEN-ELSE 논리와 유사한 방식으로 표현식을 작성해서 SQL의 비교 연산 기능을 보완하는 역할을 한다.
  - ✓ ANSI-SQL 표준에는 CASE Expression이라고 표시되어 있는데, 함수와 같은 성격을 가지고 있으며 Oracle의 Decode 함수와 같은 기능을 하므로 단일행 내장 함수에서 같이 설명을 한다.

[예제]

```
일반 프로그램의 IF-THEN-ELSE-END 로직

IF   SAL > 2000
     THEN  REVISED_SALARY = SAL
     ELSE  REVISED_SALARY = 2000
END-IF.
```

[예제]

```
같은 기능을 하는 CASE 표현이다.

SELECT ENAME,
       CASE   WHEN   SAL > 2000
              THEN   SAL
              ELSE   2000
       END    REVISED_SALARY
FROM   EMP;
```


- CASE 표현

CASE 표현	함수 설명
CASE SIMPLE_CASE_EXPRESSION 조건 ELSE 표현절 END	SIMPLE_CASE_EXPRESSION 조건이 맞으면 SIMPLE_CASE_EXPRESSION 조건내의 THEN 절을 수행하고, 조건이 맞지 않으면 ELSE 절을 수행한다.
CASE SEARCHED_CASE_EXPRESSION 조건 ELSE 표현절 END	SEARCHED_CASE_EXPRESSION 조건이 맞으면 SEARCHED_CASE_EXPRESSION 조건내의 THEN 절을 수행하고, 조건이 맞지 않으면 ELSE 절을 수행한다.
DECODE(표현식, 기준값1, 값1 [, 기준값2, 값2, ... , 디폴트값)	Oracle에서만 사용되는 함수로, 표현식의 값이 기준값1이면 값1을 출력하고, 기준값2이면 값2를 출력한다. 그리고 기준값이 없으면 디폴트 값을 출력한다. CASE 표현의 SIMPLE_CASE_EXPRESSION 조건과 동일하다.

※ CASE 표현과 Oracle의 DECODE 함수는 표현상 서로 장단점이 있으므로 특성을 고려하여 기능을 선택하면 된다.

- SIMPLE_CASE_EXPRESSION
  - ✓ SIMPLE_CASE_EXPRESSION은 CASE 다음에 바로 조건에 사용되는 칼럼이나 표현식을 표시하고, 다음 WHEN 절에서 앞에서 정의한 칼럼이나 표현식과 같은지 아닌지 판단하는 문장으로 EQUI(=) 조건만 사용한다면 SEARCHED_CASE_EXPRESSION보다 간단하게 사용할 수 있는 장점이 있다.
  - ✓ Oracle의 DECODE 함수와 기능면에서 동일하다.

[예제]

```
SELECT   LOC,
  CASE   LOC
  WHEN   'NEW YORK'   THEN 'EAST'
  WHEN   'BOSTON'     THEN 'EAST'
  WHEN   'CHICAGO'    THEN 'CENTER'
  WHEN   'DALLAS'     THEN 'CENTER'
  ELSE   'ETC'
  END    as   AREA
FROM     DEPT;
```


- SEARCHED_CASE_EXPRESSION
  - ✓ SEARCHED_CASE_EXPRESSION은 CASE 다음에는 칼럼이나 표현식을 표시하지 않고, 다음 WHEN 절에서 EQUI(=) 조건 포함 여러 조건(>, >=, <, <=)을 이용한 조건절을 사용할 수 있기 때문에 SIMPLE CASE EXPRESSION보다 훨씬 다양한 조건을 적용할 수 있는 장점이 있다.

[예제]

```
SELECT ENAME,
       CASE   WHEN   SAL >= 3000   THEN   'HIGH'
              WHEN   SAL >= 1000   THEN   'MID'
              ELSE   'LOW'
       END    AS     SALARY_GRADE
FROM   EMP;
```

- CASE 중첩
  - ✓ CASE 표현은 함수의 성질을 가지고 있으므로, 다른 함수처럼 중첩해서 사용할 수 있다
  - ✓ [예제] 사원 정보에서 급여가 2000 이상이면 보너스를 1000으로, 1000 이상이면 5000으로, 1000 미만이면 0으로 계산한다.

[예제]

```
SELECT ENAME, SAL,
       CASE  WHEN  SAL >= 2000
             THEN  1000
             ELSE (CASE WHEN SAL >= 1000
                        THEN 500
                        ELSE 0
                   END)
       END  as    BONUS
FROM   EMP;
```


- NULL
  - ✓ 널 값은 아직 정의되지 않은 값으로 0 또는 공백과 다르다. 0은 숫자이고, 공백은 하나의 문자이다.
  - ✓ 테이블을 생성할 때 NOT NULL 또는 PRIMARY KEY로 정의되지 않은 모든 데이터 유형은 널 값을 포함할 수 있다.
  - ✓ 널 값을 포함하는 연산의 경우 결과 값도 널 값이다. 모르는 데이터에 숫자를 더하거나 빼도 결과는 마찬가지로 모르는 데이터인 것과 같다.

연산	연산의 결과
NULL + 2, 2 + NULL	NULL
NULL - 2, 2 - NULL	NULL
NULL * 2, 2 * NULL	NULL
NULL / 2, 2 / NULL	NULL

- NULL 관련 함수
  - ✓ 결과값을 NULL이 아닌 다른 값을 얻고자 할 때 NVL/ISNULL 함수를 사용한다. NULL 값의 대상이 숫자 유형 데이터인 경우는 주로 0(Zero)으로, 문자 유형 데이터인 경우는 블랭크보다는 'x' 같이 해당 시스템에서 의미 없는 문자로 바꾸는 경우가 많다.

일반형 함수	함수 설명
NVL/ISNULL (표현식1, 표현식2)	표현식1의 결과값이 NULL이면 표현식2의 값을 출력한다. 단, 표현식1과 표현식2의 결과 데이터 타입이 같아야 한다. NULL 관련 가장 많이 사용되는 함수이므로 상당히 중요하다.
NULLIF (표현식1, 표현식2)	표현식1이 표현식2와 같으면 NULL을, 같지 않으면 표현식1을 리턴한다.
COALESCE (표현식1, 표현식2, …)	임의의 개수 표현식에서 NULL이 아닌 최초의 표현식을 나타낸다. 모든 표현식이 NULL이라면 NULL을 리턴한다

- NVL/ISNULL
  - ✓ [예제] 선수 테이블에서 성남 일화천마(K08) 소속 선수의 이름과 포지션을 출력하는데, 포지션이 없는 경우는 '없음'으로 표시한다.

[예제] - Oracle

```
SELECT PLAYER_NAME 선수명, POSITION,
       NVL(POSITION,'없음') 포지션
FROM   PLAYER
WHERE  TEAM_ID = 'K08'
```

[예제] - SQL Server

```
SELECT PLAYER_NAME 선수명, POSITION,
       ISNULL(POSITION,'없음') 포지션
FROM   PLAYER
WHERE  TEAM_ID = 'K08'
```

[예제] - 공통

```
SELECT PLAYER_NAME 선수명, POSITION,
       CASE  WHEN   POSITION  IS  NULL
             THEN  '없음'
             ELSE   POSITION
       END as 포지션
FROM   PLAYER
WHERE  TEAM_ID = 'K08'
```

- NVL/ISNULL
  - ✓ [예제] 급여와 커미션을 포함한 연봉을 계산하면서 NVL 함수의 필요성을 알아본다.

[예제] - Oracle

```
SELECT ENAME 사원명, SAL 월급, COMM 커미션,
       (SAL * 12) + COMM 연봉A, (SAL * 12) + NVL(COMM,0) 연봉B
FROM   EMP;
```

사원명	월급	커미션	연봉A	연봉B
SMITH	800			9600
ALLEN	1600	300	19500	19500
WARD	1250	500	15500	15500
JONES	2975			35700
MARTIN	1250	1400	16400	16400
BLAKE	2850			34200
CLARK	2450			29400
SCOTT	3000			36000
KING	5000			60000
TURNER	1500	0	18000	18000
ADAMS	1100			13200
JAMES	950			11400
FORD	3000			36000
MILLER	1300			15600

- NULL과 공집합
  - ✓ SELECT 1 FROM DUAL WHERE 1 = 2; 같은 조건이 대표적인 공집합을 발생시키는 쿼리이며, 위와 같이 조건에 맞는 데이터가 한 건도 없는 경우를 공집합이라고 하고, NULL 데이터와는 또 다르게 이해해야 한다.

NULL [예제] 및 실행 결과

```
SELECT MGR FROM EMP
WHERE ENAME='KING';

 MGR

1개의 행이 선택되었다.
```

☞ 빈 칸으로 표시되었지만 실 데이터는 NULL이다.
☞'KING'은 EMP 테이블에서 사장이므로 MGR(관리자) 필드에 NULL이 입력되어 있다.

STEP1.공집합 [예제] 및 실행 결과

```
SELECT MGR FROM EMP
WHERE ENAME='JSC';

데이터를 찾을 수 없다.
```

☞ EMP 테이블에 ENAME이 'JSC' 란 사람은 없으므로 공집합이 발생한다.

※ Oracle의 SQL*PLUS 같이 화면에서 데이터베이스와 직접 대화하는 환경이라면, 화면상에서 "데이터를 찾을 수 없다."라는 문구로 공집합을 구분할 수 있지만, 다른 개발 언어 내에 SQL 문장이 포함된 경우에는 NULL과 공집합을 쉽게 구분하기 힘든 어려움이 있다

- NULL과 공집합
  - ✓ STEP2. NVL/ISNULL 함수를 이용해 공집합을 9999로 바꾸고자 시도한다.

[예제] 및 실행 결과

```
SELECT NVL(MGR, 9999) MGR FROM EMP WHERE ENAME='JSC';
데이터를 찾을 수 없다.

☞ 많은 분들이 공집합을 NVL/ISNULL 함수를 이용해서 처리하려고 하는데,
   인수의 값이 공집합인 경우는 NVL/ISNULL 함수를 사용해도 역시 공집합이 출력된다.
☞ NVL/ISNULL 함수는 NULL 값을 대상으로 다른 값으로 바꾸는 함수이지
   공집합을 대상으로 하지 않는다.
```

  - ✓ STEP3. 적절한 집계 함수를 찾아서 NVL 함수 대신 적용한다.

[예제] 및 실행 결과

```
SELECT MAX(MGR) MGR FROM EMP WHERE ENAME='JSC';
 MGR

1개의 행이 선택되었다.

☞ 빈 칸으로 표시되었지만 실 데이터는 NULL이다.
☞ 다른 함수와 달리 집계 함수와 Scalar Subquery의 경우는
   인수의 값이 공집합인 경우에도 NULL을 출력한다.
```


- NULL과 공집합
  - ✓ STEP4. 집계 함수를 인수로 한 NVL/ISNULL 함수를 이용해서 공집합인 경우에도 빈칸이 아닌 9999로 출력하게 한다.

[예제] 및 실행 결과

```
SELECT NVL(MAX(MGR), 9999) MGR FROM EMP
WHERE ENAME='JSC';

 MGR
9999
1개의 행이 선택되었다.

☞ 공집합의 경우는 NVL 함수를 사용해도 공집합이 출력되므로,
   그룹함수와 NVL 함수를 같이 사용해서 처리한다.
☞ 예제는 그룹함수를 NVL 함수의 인자로 사용해서 인수의 값이
   공집합인 경우에도 원하는 9999라는 값으로 변환한 사례이다.
```

- NULLIF
  - ✓ NULLIF 함수는 EXPR1이 EXPR2 와 같으면 NULL을, 같지 않으면 EXPR1을 리턴한다. 특정 값을 NULL로 대체하는 경우에 유용하게 사용할 수 있다.
  - ✓ [예제] 사원 테이블에서 MGR와 7698이 같으면 NULL을 표시하고, 같지 않으면 MGR를 표시한다.

[예제]

```
SELECT ENAME, EMPNO, MGR, NULLIF(MGR, 7698) NUIF
FROM   EMP;
```

  - ✓ NULLIF 함수를 CASE 문장으로 표현할 수 있다.

[예제]

```
SELECT ENAME, EMPNO, MGR,
       CASE   WHEN   MGR = 7698
              THEN   NULL
              ELSE   MGR
       END    NUIF
FROM   EMP;
```


- COALESCE
  - ✓ COALESCE 함수는 인수의 숫자가 한정되어 있지 않으며, 임의의 개수 EXPR에서 NULL이 아닌 최초의 EXPR을 나타낸다. 만일 모든 EXPR이 NULL이라면 NULL을 리턴한다.
  - ✓ [예제] 사원 테이블에서 커미션을 1차 선택값으로, 급여를 2차 선택값으로 선택하되 두 칼럼 모두 NULL인 경우는 NULL로 표시한다.

[예제]

```
SELECT ENAME, COMM, SAL, COALESCE(COMM, SAL) COAL
FROM   EMP;
```

  - ✓ COALESCE 함수는 두개의 중첩된 CASE 문장으로 표현할 수 있다.

[예제]

```
SELECT ENAME, COMM, SAL,
       CASE   WHEN  COMM IS NOT NULL
              THEN  COMM
              ELSE (CASE WHEN   SAL IS NOT NULL
                         THEN   SAL
                         ELSE   NULL
                    END)
       END    COAL
FROM   EMP;
```

Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제6절 FUNCTION 핵심정리 및 연습문제

핵심정리 제6절 FUNCTION

- 사용자는 벤더에서 제공하는 내장 함수를 통해 데이터 값을 간편하게 조작할 수 있다.
- 단일행 함수는 처리하는 데이터의 형식에 따라서 문자형, 숫자형, 날짜형, 변환형, NULL 관련 함수로 나눌 수 있다.

문제. 다음 중 NULL과 관련이 없는 함수는?

① COALESCE
② ISNULL
③ NULLIF
④ NOTNULL

문제. 다음 중 NULL과 관련이 없는 함수는?

① COALESCE
② ISNULL
③ NULLIF
④ NOTNULL

정답: ④
해설 :
NULL과 관련이 있는 함수는 NVL/ISNULL, NULLIF, COALESCE 함수이다.

Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제7절 GROUP BY, HAVING 절 학습하기

- **집계 함수(Aggregate Function)**
  - ✓ 다중행 함수 중 집계 함수(Aggregate Function)의 특성은 다음과 같다.
    - 여러 행들의 그룹이 모여서 그룹당 단 하나의 결과를 돌려주는 함수이다.
    - SELECT 절, HAVING 절, ORDER BY 절에 사용할 수 있다.
    - 테이블 전체 집계를 위해 GROUP BY 절 없이도 집계 함수를 사용할 수 있다.
    - 집계 함수는 그룹에 대한 정보를 제공하므로 주로 숫자 유형에 사용되지만, MAX, MIN, COUNT 함수는 문자, 날짜 유형에도 적용이 가능한 함수이다.

단일 행 함수, 6절 (Single-Row Function)	다중 행 함수 (Multi-Row Function) - 집계 함수(Aggregate Function), 7절 - 그룹 함수(Group Function), 2장5절 - 윈도우 함수(Window Function),2장6절

- 집계 함수(Aggregate Function)

집계 함수명 ([DISTINCT | ALL] 칼럼이나 표현식)
– ALL : Default 옵션이므로 생략 가능함
– DISTINCT : 같은 값을 하나의 데이터로 간주할 때 사용하는 옵션임

집계 함수	사용 목적
COUNT(*)	NULL 값을 포함한 행의 수를 출력한다.
COUNT(표현식)	표현식의 값이 NULL 값인 것을 제외한 행의 수를 출력한다.
SUM([DISTINCT | ALL] 표현식)	표현식의 NULL 값을 제외한 합계를 출력한다.
AVG([DISTINCT | ALL] 표현식)	표현식의 NULL 값을 제외한 평균을 출력한다.
MAX([DISTINCT | ALL] 표현식)	표현식의 최대값을 출력한다. (문자,날짜 데이터 타입도 사용가능)
MIN([DISTINCT | ALL] 표현식)	표현식의 최소값을 출력한다. (문자,날짜 데이터 타입도 사용가능)
STDDEV([DISTINCT | ALL] 표현식)	표현식의 표준 편차를 출력한다.
VARIAN([DISTINCT | ALL] 표현식)	표현식의 분산을 출력한다.
기타 통계 함수	벤더별로 다양한 통계식을 제공한다.

- 집계 함수(Aggregate Function)

✓ [예제] 일반적으로 집계 함수는 GROUP BY 절과 같이 사용되지만 테이블 전체가 하나의 그룹이 되는 경우에는 GROUP BY 절 없이 단독으로도 사용 가능하다.

[예제]

```
SELECT COUNT(*) "전체 행수", COUNT(HEIGHT) "키 건수",
       MAX(HEIGHT) 최대키, MIN(HEIGHT) 최소키, ROUND(AVG(HEIGHT),2) 평균키
FROM   PLAYER;
```

실행 결과

전체 행수	키 건수	최대키	최소키	평균키
480	447	196	165	179.31

1개의 행이 선택되었다.

❖ 실행 결과를 보면 COUNT(HEIGHT)는 NULL 값이 아닌 키(HEIGHT) 칼럼의 건수만 출력하므로 COUNT(*)의 480보다 작은 것을 볼 수 있다. 그 이유는 COUNT(*) 함수에 사용된 와일드카드(*)는 전체 칼럼을 뜻하는데 전체 칼럼이 NULL인 행은 존재할 수 없기 때문에 결국 COUNT(*)는 전체 행의 개수를 출력한 것이고, COUNT(HEIGHT)는 HEIGHT 칼럼 값이 NULL인 33건은 제외된 건수의 합이다.

제7절 GROUP BY, HAVING 절

- GROUP BY 절
  - ✓ GROUP BY 절은 SQL 문에서 FROM 절과 WHERE 절 뒤에 오며 GROUP BY 절은 행들을 소그룹화 한다.
  - ✓ 데이터들을 작은 그룹으로 분류하여 소그룹에 대한 항목별로 통계 정보를 얻을 때 추가로 사용된다.
  - ✓ GROUP BY 절과 HAVING 절은 다음과 같은 특성을 가진다.
    - GROUP BY 절을 통해 소그룹별 기준을 정한 후, SELECT 절에 집계 함수를 사용한다.
    - GROUP BY 절에서는 SELECT 절과는 달리 칼럼 ALIAS 명을 사용할 수 없다.
    - WHERE 절은 전체 데이터를 GROUP으로 나누기 전에 행들을 미리 제거시킨다.
    - GROUP BY 절과 HAVING 절의 순서를 바꾸더라도 문법 에러가 없고 결과물도 동일한 결과를 출력한다. 그렇지만, 논리적으로 GROUP BY 절과 HAVING 절의 순서를 지키는 것을 권고한다.

제7절 GROUP BY, HAVING 절

- GROUP BY 절
  - ✓ GROUP BY 절과 HAVING 절은 다음과 같은 특성을 가진다.
    - 집계 함수의 통계 정보는 NULL 값을 가진 행을 제외하고 수행한다.
    - HAVING 절은 GROUP BY 절의 기준 항목이나 소그룹의 집계 함수를 이용한 조건을 표시할 수 있다.
    - 집계 함수는 WHERE 절에는 올 수 없다.
    - GROUP BY 절에 의한 소그룹별로 만들어진 집계 데이터 중, HAVING 절에서 제한 조건을 두어 조건을 만족하는 내용만 출력한다.
    - 일부 데이터베이스의 과거 버전에서는 데이터베이스가 GROUP BY 절에 명시된 칼럼(Column)의 순서대로 오름차순 정렬을 자동으로 실시하는 경우가 있었으나, 지금은 정렬을 위해서는 뒤에서 언급할 ORDER BY 절을 명시해야 정렬이 수행된다.

- GROUP BY 절
  - ✓ [예제] 포지션별 최대키, 최소키, 평균키를 출력한다. (포지션별이란 소그룹 조건을 제시하였기 때문에 GROUP BY 절을 사용한다)

[예제]

```
SELECT POSITION 포지션, COUNT(*) 인원수, COUNT(HEIGHT) 키대상,
       MAX(HEIGHT) 최대키, MIN(HEIGHT) 최소키, ROUND(AVG(HEIGHT),2) 평균키
FROM   PLAYER
GROUP  BY POSITION;
```

  - ✓ GROUP BY 절에서 그룹 단위를 표시해 주어야 SELECT 절에서 그룹 단위의 칼럼과 집계 함수를 사용할 수 있다.
  - ✓ 칼럼에 대한 ALIAS는 SELECT 절에서 정의하고 ORDER BY 절에서는 재활용할 수 있지만, GROUP BY 절에서는 ALIAS 명을 사용할 수 없다. (교재 참고)

- GROUP BY 절

[실행 결과]

포지션	인원수	키대상	최대키	최소키	평균키
	3	0			
GK	43	43	196	174	186.26
DF	172	142	190	170	180.21
FW	100	100	194	168	179.91
MF	162	162	189	165	176.31

5개의 행이 선택되었다.

- 포지션과 키 정보가 없는 선수가 3명이라는 정보를 얻을 수 있으며,
- 포지션이 DF인 172명 중 30명은 키에 대한 정보가 없는 것도 알 수 있다.
- ORDER BY 절이 없기 때문에 포지션 별로 정렬은 되지 않았다.

- HAVING 절
  - ✓ HAVING 절은 WHERE 절과 비슷하지만 GROUP BY절에 의해 만들어진 소그룹에 대한 조건이 적용된다는 점에서 차이가 있다.
  - ✓ [예제] HAVING 조건절에는 GROUP BY 절에서 정의한 소그룹의 집계 함수를 이용한 조건을 표시할 수 있으므로, HAVING 절을 이용해 평균키가 180 센티미터 이상인 정보만 표시한다.

[예제]

```
SELECT POSITION 포지션, ROUND(AVG(HEIGHT),2) 평균키
FROM   PLAYER
GROUP  BY  POSITION
HAVING AVG(HEIGHT) >= 180;
```

포지션	평균키
GK	186.26
DF	180.21

2개의 행이 선택되었다.

- HAVING 절
  - ✓ HAVING 절은 SELECT 절에 사용되지 않은 칼럼이나 집계 함수가 아니더라도 GROUP BY 절의 기준 항목이나 소그룹의 집계 함수를 이용한 조건을 표시할 수 있다.
  - ✓ [예제] 포지션별 평균키만 출력하는데, 최대키가 190cm 이상인 선수를 가지고 있는 포지션의 정보만 출력한다.

[예제]

```
SELECT POSITION 포지션, ROUND(AVG(HEIGHT),2) 평균키
FROM   PLAYER
GROUP  BY  POSITION
HAVING MAX(HEIGHT) >= 190;
```

포지션	평균키
GK	186.26
DF	180.21
FW	179.91

3개의 행이 선택되었다.

- CASE 표현을 활용한 월별 데이터 집계
  - ✓ "집계 함수(CASE( ))~GROUP BY" 기능은, 모델링의 제1정규화로 인해 반복되는 칼럼의 경우 구분 칼럼을 두고 여러 개의 레코드로 만들어진 집합을, 정해진 칼럼 수만큼 확장해서 집계 보고서를 만드는 유용한 기법이다.
  - ✓ [예제] 부서별로 월별 입사자의 평균 급여를 알고 싶다는 고객의 요구사항이 있는데, 입사 후 1년마다 급여 인상이나 보너스 지급같은 일정이 잡힌다면 업무적으로 중요한 정보가 될 수 있다. 먼저 개별 입사정보에서 월별 데이터를 확인한다.

[예제] Oracle 함수

```
SELECT ENAME, DEPTNO, EXTRACT(MONTH FROM HIREDATE) 입사월, SAL
FROM   EMP;
```

[예제] SQL Server 함수

```
SELECT ENAME, DEPTNO, DATEPART(MONTH, HIREDATE) 입사월, SAL
FROM   EMP;

SELECT ENAME, DEPTNO, MONTH(HIREDATE) 입사월, SAL
FROM   EMP;
```


- CASE 표현을 활용한 월별 데이터 집계

[예제] CASE 표현

```
SELECT DEPTNO,
       AVG(CASE MONTH WHEN  1 THEN SAL END) M01,
       AVG(CASE MONTH WHEN  2 THEN SAL END) M02,
       AVG(CASE MONTH WHEN  3 THEN SAL END) M03,
       AVG(CASE MONTH WHEN  4 THEN SAL END) M04,
       AVG(CASE MONTH WHEN  5 THEN SAL END) M05,
       AVG(CASE MONTH WHEN  6 THEN SAL END) M06,
       AVG(CASE MONTH WHEN  7 THEN SAL END) M07,
       AVG(CASE MONTH WHEN  8 THEN SAL END) M08,
       AVG(CASE MONTH WHEN  9 THEN SAL END) M09,
       AVG(CASE MONTH WHEN 10 THEN SAL END) M10,
       AVG(CASE MONTH WHEN 11 THEN SAL END) M11,
       AVG(CASE MONTH WHEN 12 THEN SAL END) M12
FROM  (SELECT ENAME, DEPTNO, EXTRACT(MONTH FROM HIREDATE) MONTH, SAL
       FROM   EMP)
GROUP  BY DEPTNO ;
```

✓같은 기능을 하는 리포트를 작성하기 위해 장문의 프로그램을 작성해야 하는 것에 비해, 위 방법을 활용하면 복잡한 프로그램이 아닌 하나의 SQL 문장으로 처리 가능하므로 DBMS 자원 활용이나 처리 속도에서 훨씬 효율적이다. 데이터의 건수가 많아질 수록 처리 속도 차이는 더 크진다.

- CASE 표현을 활용한 월별 데이터 집계

[예제] Oracle

```
SELECT DEPTNO,
  AVG(DECODE(MONTH, 1,SAL)) M01, AVG(DECODE(MONTH, 2,SAL)) M02,
  AVG(DECODE(MONTH, 3,SAL)) M03, AVG(DECODE(MONTH, 4,SAL)) M04,
  AVG(DECODE(MONTH, 5,SAL)) M05, AVG(DECODE(MONTH, 6,SAL)) M06,
  AVG(DECODE(MONTH, 7,SAL)) M07, AVG(DECODE(MONTH, 8,SAL)) M08,
  AVG(DECODE(MONTH, 9,SAL)) M09, AVG(DECODE(MONTH,10,SAL)) M10,
  AVG(DECODE(MONTH,11,SAL)) M11, AVG(DECODE(MONTH,12,SAL)) M12
FROM (SELECT ENAME, DEPTNO, EXTRACT(MONTH FROM HIREDATE) MONTH, SAL
        FROM EMP)
GROUP BY DEPTNO ;
```

실행 결과

DEPTNO	M01	M02	M03	M04	M05	M06	M07	M08	M09	M10	M11	M12
30		1425			2850				1375			950
20				2975			2050					1900
10	130					2450					5000	

- 집계 함수와 NULL 처리
  - ✓ 다중 행 함수는 입력 값으로 전체 건수가 NULL 값인 경우만 함수의 결과가 NULL이 나오고 전체 건수 중에서 일부만 NULL인 경우는 NULL 인 행을 다중 행 함수의 대상에서 제외한다. 예를 들면 100명 중 10명의 성적이 NULL 값일 때 평균을 구하는 다중 행 함수 AVG를 사용하면 NULL 값이 아닌 90명의 성적에 대해서 평균 값을 구하게 된다.
  - ✓ 초급 개발자가 많이 실수하는 것이 Oracle의 SUM(NVL(SAL,0)), SQL Server의 SUM(ISNULL(SAL,0)) 연산이다. 개별 데이터의 급여(SAL)가 NULL인 경우는 NULL의 특성으로 자동적으로 연산에서 빠지는 데, 불필요하게 NVL/ISNULL 함수를 사용해 0(Zero)으로 변환시켜 데이터 건수 만큼의 연산이 일어나게 하는 것은 시스템의 자원을 낭비하는 일이다. 리포트 출력 때 NULL이 아닌 0을 표시하고 싶은 경우에는 NVL(SUM(SAL),0)이나, ISNULL(SUM(SAL),0)처럼 전체 SUM의 결과가 NULL인 경우(대상 건수가 모두 NULL인 경우)에만 한 번 NVL/ISNULL 함수를 사용하면 된다..
  - ✓ CASE 표현 사용시 ELSE 절을 생략하게 되면 Default 값이 NULL이다. NULL은 연산의 대상이 아닌 반면, SUM(CASE MONTH WHEN 1 THEN SAL ELSE 0 END)처럼 ELSE 절에서 0(Zero)을 지정하면 불필요하게 0이 SUM 연산에 사용되므로 자원의 사용이 많아진다. 같은 결과를 얻을 수 있다면 가능한 ELSE 절의 상수값을 지정하지 않거나 ELSE 절을 작성하지 않도록 한다. 같은 이유로 Oracle의 DECODE 함수는 4번째 인자를 지정하지 않으면 NULL이 Default로 할당된다.

- 집계 함수와 NULL 처리

[예제] 개선 전 SQL (SIMPLE_CASE_EXPRESSION)

```
SELECT TEAM_ID,
       SUM(NVL((CASE  POSITION  WHEN 'FW'  THEN 1  ELSE 0 END),0) FW,
       SUM(NVL((CASE  POSITION  WHEN 'MF'  THEN 1  ELSE 0 END),0) MF,
       SUM(NVL((CASE  POSITION  WHEN 'DF'  THEN 1  ELSE 0 END),0) DF,
       SUM(NVL((CASE  POSITION  WHEN 'GK'  THEN 1  ELSE 0 END),0) GK,
       SUM(NVL(1),0)  SUM
FROM   PLAYER
GROUP  BY TEAM_ID;
```

[예제] 개선 후 SQL (SIMPLE_CASE_EXPRESSION)

```
SELECT TEAM_ID,
       NVL(SUM(CASE  POSITION  WHEN 'FW'  THEN 1  END),0) FW,
       NVL(SUM(CASE  POSITION  WHEN 'MF'  THEN 1  END),0) MF,
       NVL(SUM(CASE  POSITION  WHEN 'DF'  THEN 1  END),0) DF,
       NVL(SUM(CASE  POSITION  WHEN 'GK'  THEN 1  END),0) GK,
       NVL(SUM(1),0)  SUM
FROM   PLAYER
GROUP  BY TEAM_ID;
```


Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제7절 GROUP BY, HAVING 절
핵심정리 및 연습문제

핵심정리 제7절 GROUP BY, HAVING 절

- 집계 함수(Aggregate Function)는 여러 행들이 모여서 그룹당 단 하나의 결과를 돌려주는 다중행 함수의 일부로서 COUNT, SUM, AVG, MAX, MIN 함수가 있다.
- GROUP BY 절은 집합별 통계 정보의 기준을 명시하고,
- HAVING 절에는 집합에 대한 제한 조건을 두어 조건을 만족하는 내용만 출력한다.

연습문제 제7절 GROUP BY, HAVING 절

문제. 어떠한 데이터 타입도 사용이 가능한 집계 함수는 어느 것인가?

① COUNT
② SUM
③ AVG
④ STDDEV

연습문제 제7절 GROUP BY, HAVING 절

문제. 어떠한 데이터 타입도 사용이 가능한 집계 함수는 어느 것인가?

① COUNT
② SUM
③ AVG
④ STDDEV

정답: ①
해설 :
집계 함수는 집합에 대한 정보를 제공하므로 주로 숫자 유형에 사용된다.
추가로 MAX, MIN, COUNT 함수는 숫자 유형만 아니라
문자 유형, 날짜 유형에도 적용이 가능한 함수이다.

연습문제 제7절 GROUP BY, HAVING 절

문제. SQL 문장에서 집합별로 집계된 데이터에 대한 조회 조건을 제한하기 위해서 사용하는 절은 어느 것인가?

① WHERE 절
② GROUP BY 절
③ HAVING 절
④ FROM 절

연습문제 제7절 GROUP BY, HAVING 절

문제. SQL 문장에서 집합별로 집계된 데이터에 대한 조회 조건을 제한하기 위해서 사용하는 절은 어느 것인가?

① WHERE 절
② GROUP BY 절
③ HAVING 절
④ FROM 절

정답: ③
해설 :
일반적인 SQL 문장에서 조회하는 데이터를 제한하기 위해서는 WHERE 절을 사용하지만, 그룹별로 조회할 때 집계 데이터에 대한 제한 조건을 사용하기 위해서는 HAVING 절을 사용한다.

Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제8절 ORDER BY 절 학습하기

제8절 ORDER BY 절

- ORDER BY 정렬
  - ✓ ORDER BY 절은 SQL 문장으로 조회된 데이터들을 다양한 목적에 맞게 특정 칼럼을 기준으로 정렬하여 출력하는데 사용한다.
  - ✓ ORDER BY 절에 칼럼(Column)명 대신에 SELECT 절에서 사용한 ALIAS 명이나 칼럼 순서를 나타내는 정수도 사용 가능하다. SELECT 절의 칼럼명이 길거나 정렬 조건이 많을 경우 편리하게 사용할 수 있으나, 향후 유지보수성이나 가독성이 떨어지므로 가능한 칼럼명이나 ALIAS 명을 권고한다. ORDER BY 절에서 칼럼명, ALIAS명, 칼럼 순서를 같이 혼용하는 것도 가능하다.
  - ✓ 별도로 정렬 방식을 지정하지 않으면 기본적으로 오름차순이 적용되며, SQL 문장의 제일 마지막에 위치한다.
  - ✓ Oracle은 NULL 값을 가장 큰 값으로 취급한다. 반면 SQL Server는 최소값으로 간주한다.

제8절 ORDER BY 절

- ORDER BY 정렬
  - ✓ [예제] 한 개의 칼럼이 아닌 여러 가지 칼럼(Column)을 기준으로 정렬해본다. 먼저 키가 큰 순서대로, 키가 같은 경우 백넘버 순으로 ORDER BY 절을 적용하여 SQL 문장을 작성하는데, 키가 NULL인 데이터는 제외한다.

[ORDER BY 칼럼(Column)이나 표현식 [ASC 또는 DESC]] ;

- ASC(Ascending) : 조회한 데이터를 오름차순으로 정렬한다. (기본값으로 생략 가능)
- DESC(Descending) : 조회한 데이터를 내림차순으로 정렬한다.

[예제]

```
SELECT PLAYER_NAME 선수이름, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  HEIGHT IS NOT NULL
ORDER  BY HEIGHT DESC, BACK_NO ASC;
```

제8절 ORDER BY 절

- ORDER BY 정렬
  - ✓ [예제] DEPT 테이블 정보를 부서명, 지역, 부서번호 내림차순으로 정렬해서 출력한다. 아래의 SQL 문장은 출력되는 칼럼 레이블은 다를 수 있지만 결과 데이터는 모두 같다. ORDER BY 절에서 칼럼명, ALIAS명, 칼럼 순서를 같이 혼용하는 것도 가능하다.

[예제]

```
SELECT DNAME, LOC, DEPTNO
FROM   DEPT
ORDER  BY DNAME ASC, LOC ASC,
       DEPTNO DESC;
```

[예제]

```
SELECT DNAME DEPT, LOC AREA, DEPTNO
FROM   DEPT
ORDER  BY DNAME, AREA, DEPTNO DESC;
```

[예제]

```
SELECT DNAME, LOC, DEPTNO
FROM   DEPT
ORDER  BY DNAME, LOC, DEPTNO DESC;
```

[예제]

```
SELECT DNAME, LOC as AREA, DEPTNO
FROM   DEPT
ORDER  BY 1, AREA, 3 DESC;
```


제8절 ORDER BY 절

- SELECT 문장 실행 순서
  - ✓ GROUP BY 절과 ORDER BY가 같이 사용될 때 SELECT SQL 문장은 6개의 절로 구성이 되고, SELECT SQL 문장의 수행 단계는 아래와 같다. 아래는 옵티마이저가 SQL 문장의 SYNTAX 에러를 점검하는 순서이기도 하다.

```
5. SELECT   칼럼명 [ALIAS명]
1. FROM     테이블명
2. WHERE    조건식
3. GROUP BY 칼럼(Column)이나 표현식
4. HAVING   그룹조건식
6. ORDER BY 칼럼(Column)이나 표현식;
```

1. 발췌 대상 테이블을 참조한다. (FROM)
2. 발췌 대상 데이터가 아닌 것은 제거한다. (WHERE)
3. 행들을 소그룹화 한다. (GROUP BY)
4. 그룹핑된 값의 조건에 맞는 것만을 출력한다.(HAVING)
5. 데이터 값을 출력/계산한다. (SELECT)
6. 데이터를 정렬한다. (ORDER BY)

제8절 ORDER BY 절

- SELECT 문장 실행 순서
  - ✓ FROM 절에 정의되지 않은 테이블의 칼럼을 WHERE 절, GROUP BY 절, HAVING 절, SELECT 절, ORDER BY 절에서 사용하면 에러가 발생한다.
  - ✓ 그러나, ORDER BY 절에는 SELECT 목록에 나타나지 않은 문자형 항목이 포함될 수 있다. (단, SELECT DISTINCT를 지정하거나 문에 GROUP BY 절이 있거나 또는 SELECT 문에 UNION 연산자가 있으면 열 이름이 SELECT 목록에 나타나야 한다)

[예제]

```
SELECT EMPNO, ENAME
FROM   EMP
ORDER  BY MGR;
```

  - ✓ 관계형 데이터베이스가 데이터를 메모리에 올릴 때 행 단위로 모든 칼럼을 가져오게 되므로, SELECT 절에서 일부 칼럼만 선택하더라도 ORDER BY 절에서 메모리에 올라와 있는 다른 칼럼의 데이터를 사용할 수 있다. (단위 SQL 기준이며 서브쿼리를 벗어나는 경우 SELECT 절에 정의된 칼럼만 메인쿼리에서 재사용될 수 있다)

제8절 ORDER BY 절

- SELECT 문장 실행 순서
  - ✓ 실행 결과에서 2장에서 배울 인라인 뷰의 SELECT 절에서 정의한 칼럼만 메인쿼리에서 사용할 수 있는 것을 확인할 수 있다. (서브쿼리 동일)
  - ✓ [예제] 인라인 뷰에 정의된 SELECT 칼럼을 메인쿼리에서 사용할 수 있다.

[예제] 및 실행 결과

```
 SELECT EMPNO  FROM
(SELECT EMPNO, ENAME FROM EMP ORDER BY MGR);
14개의 행이 선택되었다.
```

  - ✓ [예제] 인라인 뷰에 미정의된 칼럼은 메인쿼리에서 사용할 수 없다.

[예제] 및 실행 결과

```
 SELECT MGR FROM
(SELECT EMPNO, ENAME FROM EMP ORDER BY MGR);

 SELECT MGR FROM ; *
 ERROR: "MGR": 부적합한 식별자
```

- SELECT 문장 실행 순서
  - ✓ GROUP BY 절에서 그룹핑 기준을 정의하게 되면 DBMS는 일반적인 SELECT 문장처럼 FROM 절에 정의된 테이블의 구조를 그대로 가지고 가는 것이 아니라, GROUP BY 절의 그룹핑 기준으로 행들을 소그룹화 한다.
  - ✓ [예제] GROUP BY 이후 소그룹화가 진행된 상황에서 수행 절인 SELECT 절이나 ORDER BY 절에서 개별 데이터를 사용하는 경우 에러가 발생한다.

[예제] 및 실행 결과

```
SELECT JOB, DEPTNO
FROM   EMP
GROUP  BY JOB
HAVING COUNT(*) > 0
ORDER  BY SAL;

SELECT JOB, DEPTNO ;  *
ERROR: GROUP BY 표현식이 아니다.
```

[예제] 및 실행 결과

```
SELECT JOB
FROM   EMP
GROUP  BY JOB
HAVING COUNT(*) > 0
ORDER  BY SAL;

ORDER  BY SAL;  *
ERROR: GROUP BY 표현식이 아니다.
```


- SELECT 문장 실행 순서
  - ✓ GROUP BY 절이 사용되었을 경우 GROUP BY 절의 그룹핑 기준으로 소그룹화된 데이터를 활용해서, SELECT 절에 정의하지 않은 MAX, SUM, COUNT 같은 집계 함수를 ORDER BY 절에서 사용할 수 있는 것을 확인할 수 있다.
  - ✓ [예제] GROUP BY 절 사용시 SELECT 절에 정의되지 않은 집계 칼럼을 ORDER BY 절에 사용해본다.

[예제] 및 실행 결과

```
SELECT JOB
FROM   EMP
GROUP  BY JOB
HAVING COUNT(*) > 0
ORDER  BY MAX(EMPNO), MAX(MGR), SUM(SAL), COUNT(DEPTNO), MAX(HIREDATE);

5개의 행이 선택되었다.
```

- Top N 쿼리
  - ✓ 5절 WHERE절에서 설명한 ROWNUM 사용시 주의할 점을 소개한다.
  - ✓ Oracle의 경우 ORDER BY 절과 WHERE 절의 ROWNUM 조건을 같이 사용하는 경우 정렬이 완료된 데이터의 일부가 출력되는 것이 아니라, 데이터의 일부가 먼저 추출된 후 데이터에 대한 정렬 작업이 일어나므로 원하는 데이터를 얻지 못할 수 있다.
  - ✓ [예제] 사원 테이블에서 급여가 높은 3명을 내림차순으로 출력하고자 한다.

[예제]

```
SELECT ENAME, SAL
FROM   EMP
WHERE  ROWNUM < 4
ORDER  BY SAL DESC;
```

실행 결과

ENAME	SAL
ALLEN	1600
WARD	1250
SMITH	800

3개의 행이 선택되었다.

  - ✓ 실행 결과의 3명은 급여가 상위인 3명을 출력한 것이 아니라, 무작위로 추출된 3명에 한해서(WHERE절 우선 수행) 급여를 내림차순으로 정렬한 결과이므로 문법적 오류가 발생하지 않더라도 업무적으로 잘못된 결과를 출력한 것이다.

- Top N 쿼리
  - ✓ [예제] 먼저 정렬된 데이터의 일부를 출력하고자 하는 경우는 2장에서 배울 인라인 뷰에서 먼저 데이터 정렬을 수행한 후 메인 쿼리에서 ROWNUM 조건을 사용해야 한다. (일반적으로 인라인 뷰가 메인쿼리보다 먼저 수행됨)

[예제]

```
SELECT ENAME, SAL
FROM (SELECT ENAME, SAL
      FROM   EMP
      ORDER BY SAL DESC)
WHERE ROWNUM < 4 ;
```

실행 결과

ENAME	SAL
KING	5000
SCOTT	3000
FORD	3000

3개의 행이 선택되었다.

  - ✓ EMP 테이블의 데이터를 급여가 많은 순서부터 인라인 뷰에서 정렬을 먼저 수행한 후 상위 3건의 데이터를 출력한 것이다.
  - ✓ ORDER BY 절이 없으면 Oracle의 Rownum 조건과 SQL Server의 TOP 절은 같은 결과를 보인다. 그렇지만, ORDER BY 절이 사용되는 경우 Oracle은 Rownum 조건을 Order By 절보다 먼저 처리되는 WHERE 절에서 처리하므로 정렬 후 원하는 데이터를 얻기 위해서는 위처럼 먼저 인라인뷰로 가공하는 절차를 거쳐야 한다.

- Top N 쿼리
  - ✓ 반면 SQL Server는 TOP 조건을 사용하게 되면 별도 처리 없이 관련 Order By 절의 데이터 정렬 후 원하는 일부 데이터만 쉽게 출력할 수 있다.
  - ✓ [예제] 사원 테이블에서 급여가 높은 2명을 내림차순으로 출력하고자 한다.

[예제]

```
SELECT TOP(2) ENAME, SAL
FROM   EMP
ORDER  BY SAL DESC;
```

※ 동일 급여에 대한 차순위 기준 미정

실행 결과

ENAME	SAL
KING	5000
SCOTT	3000

2개의 행이 선택되었다.

  - ✓ [예제] 사원 테이블에서 급여가 높은 2명을 내림차순으로 출력하는데 같은 급여를 받는 사원이 있으면 같이 출력한다.

[예제]

```
SELECT TOP(2) WITH TIES
       ENAME, SAL
FROM   EMP
ORDER  BY SAL DESC;
```

실행 결과

ENAME	SAL
KING	5000
SCOTT	3000
FORD	3000

3개의 행이 선택되었다.

Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제8절 ORDER BY 절
핵심정리 및 연습문제

핵심정리 제8절 ORDER BY 절

- ORDER BY 절은 조회된 자료를 원하는 칼럼 순으로 정렬하는 기능을 수행하고, SELECT 문장의 제일 마지막에 위치한다.
- SELECT 문장은 FROM 절, WHERE 절, GROUP BY 절, HAVING 절, SELECT 절, ORDER BY 절 순서로 실행된다.

연습문제 제8절 ORDER BY 절

문제. 다음과 같은 SQL 문장이 있다. 예제의 ORDER BY 절과 같은 결과를 갖는 구문은 어떤 것인가?

① ORDER BY 1 DESC, 2, 백넘버
② ORDER BY 선수명, 2, DESC 백넘버
③ ORDER BY PLAYER_NAME ASC, 2, 3
④ ORDER BY 선수명 ASC, 포지션, 3 DESC

연습문제 제8절 ORDER BY 절

문제. 다음과 같은 SQL 문장이 있다. 예제의 ORDER BY 절과 같은 결과를 갖는 구문은 어떤 것인가?

① ORDER BY 1 DESC, 2, 백넘버
② ORDER BY 선수명, 2, DESC 백넘버
③ ORDER BY PLAYER_NAME ASC, 2, 3
④ ORDER BY 선수명 ASC, 포지션, 3 DESC

정답: ④
해설 :
ORDER BY 절에서 정렬 기준이 생략되면 Default로 ASC(오름차순) 정렬이 되며, ORDER BY 절에는 칼럼(Column)명 대신에 SELECT 절에 기술한 칼럼(Column)의 순서 번호나 칼럼(Column)의 ALIAS 명을 대신해서 사용할 수 있다.

Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제9절 JOIN 학습하기

- JOIN
  - ✓ JOIN이 필요한 기본적인 이유는 과목1에서 배운 정규화에서부터 출발한다. 정규화란 불필요한 데이터의 정합성을 확보하고 이상현상(Anomaly) 발생을 피하기 위해, 테이블을 분할하여 생성하는 것이다.
  - ✓ 데이터웨어하우스 모델처럼 하나의 테이블에 모든 데이터를 집중시켜놓고(반정규화/비정규화) 그 테이블로부터 필요한 데이터를 조회할 수도 있다. 그러나 이렇게 됐을 경우, 가장 중요한 데이터의 정합성에 더 큰 비용을 지불해야 하며, 데이터를 추가, 삭제, 수정하는 작업 역시 상당한 노력이 요구될 것이다. 성능 측면에서도 간단한 데이터를 조회하는 경우에도 규모가 큰 테이블에서 필요한 데이터를 찾아야 하기 때문에 오히려 검색 속도가 떨어질 수도 있다.
  - ✓ 테이블을 정규화하여 데이터를 분할하게 되면 위와 같은 문제는 자연스럽게 해결된다.하지만 특정 요구조건을 만족하는 데이터들을 분할된 테이블로부터 조회하기 위해서는 테이블 간에 논리적인 연관관계가 필요하고 그런 관계성을 통해서 다양한 데이터들을 조회할 수 있는 것이다. 그리고, 이런 논리적인 관계를 성립시켜주는 것이 바로 JOIN 조건인 것이다. 유연한 조인 기능은 관계형 데이터베이스의 가장 큰 장점이라고 할 수있다.

- JOIN
  - ✓ 두 개 이상의 테이블 들을 연결 또는 결합하여 데이터를 조회하는 것을 JOIN이라고 하며, 일반적으로 사용되는 SQL 문장의 상당수가 JOIN이라고 생각하면 JOIN의 중요성을 이해하기 쉬울 것이다. 조인은 관계형 데이터베이스의 가장 큰 장점이면서 기본적인 기능이라고 할 수 있다.
  - ✓ 일반적인 경우 행들은 PRIMARY KEY(PK)나 FOREIGN KEY(FK) 값의 연관에 의해 JOIN이 성립된다. 하지만 어떤 경우에는 이러한 PK, FK의 관계가 없어도 논리적인 값들의 연관성만으로 JOIN이 성립 가능하다. 이 기능은 계층형(Hierarchical)이나 망형(Network) 데이터베이스와 비교해서 관계형 데이터베이스의 큰 장점이다.
  - ✓ 주의할 점은 FROM 절에 여러 테이블이 나열되더라도 SQL에서 데이터를 처리할 때는 단 두 개의 집합 간에만 조인이 일어난다는 것이다. FROM 절에 A, B, C 테이블이 나열되었더라도 특정 2개의 테이블만 먼저 조인 처리되고, 2개의 테이블이 조인되어서 처리된 새로운 데이터 집합과 남은 한 개의 테이블이 다음 차례로 조인 되는 것이다. 이 순서는 4개 이상의 테이블이 사용되더라도 같은 프로세스로 반복된다.
  - ✓ JOIN에 참여하는 대상 테이블이 N개라고 했을 때, N개의 테이블로부터 필요한 데이터를 조회하기 위해 필요한 JOIN 조건은 대상 테이블의 개수에서 하나를 뺀 N-1개 이상이 필요하다.

- 테이블 관계도

선수(PLAYER) 테이블

PLAYER_NAME	BACK_NO	TEAM_ID
이고르	21	K06
오비나	26	K10
윤원일	45	K02
페르난도	44	K04
레오	45	K03
실바	45	K07
무스타파	77	K04
에디	7	K01
알리송	14	K01
쟈스민	33	K08
디디	8	K06
…	…	…

팀(TEAM) 테이블

TEAM_ID	TEAM_NAME	REGION_NAME
K05	현대모터스	전북
K08	일화천마	성남
K03	스틸러스	포항
K07	드래곤즈	전남
K09	FC서울	서울
K04	유나이티드	인천
K11	경남FC	경남
K01	울산현대	울산
K10	시티즌	대전
K02	삼성블루윙즈	수원
K12	광주상무	광주
K06	아이파크	부산
K13	강원FC	강원
K14	제주유나이티드FC	제주
K15	대구FC	대구

- 팀의 정보가 들어 있는 팀(TEAM) 테이블과 선수들의 정보가 들어 있는 선수(PLAYER) 테이블이 있다.
- 팀(TEAM) 테이블의 팀코드(TEAM_ID) 칼럼과 선수(PLAYER) 테이블에 있는 소속팀코드(TEAM_ID) 칼럼은 PK(팀 테이블의 팀코드)와 FK(선수 테이블의 소속팀 코드)의 관계에 있다.

- EQUI JOIN
  - ✓ EQUI(등가) JOIN은 두 개의 테이블 간에 칼럼 값들이 서로 정확하게 일치하는 경우에 사용되는 방법으로 대부분 PK↔FK의 관계를 기반으로 한다. 그러나 일반적으로 테이블 설계 시에 나타난 PK↔FK의 관계를 이용하는 것이지 반드시 PK↔FK의 관계로만 EQUI JOIN이 성립하는 것은 아니다.
  - ✓ 고전적인 JOIN의 조건은 WHERE 절에 기술하게 되는데 EQUI JOIN은 "=" 연산자를 사용해서 표현한다.

```
SELECT 테이블1.칼럼명,  테이블2.칼럼명, …
FROM   테이블1,테이블2
WHERE  테이블1.칼럼명1 = 테이블2.칼럼명2;

→ WHERE 절에 JOIN 조건을 표시한다.
   FROM  절에 JOIN 조건을 표시하는 기능은 2장 1절에서 설명함
```

- EQUI JOIN
  - ✓ [예제] 선수 테이블과 팀 테이블 조인 SQL

[예제]

```
SELECT PLAYER.PLAYER_NAME, PLAYER.BACK_NO, PLAYER.TEAM_ID,
       TEAM.TEAM_NAME,  TEAM.REGION_NAME
FROM   PLAYER,TEAM
WHERE  PLAYER.TEAM_ID = TEAM.TEAM_ID;
```

  - ✓ SQL 문장에 단순히 칼럼명이 오지 않고 "테이블명.칼럼명"처럼 테이블명과 칼럼명이 같이 명시하는 것은 두 가지 이유가 있다.
    - 먼저 모든 테이블에 칼럼들이 유일한 이름을 가진다면 상관 없지만, JOIN에 사용되는 두 개의 테이블에 같은 칼럼명(TEAM_ID)이 존재하는 경우에는 DBMS의 옵티마이저는 어떤 칼럼을 사용해야 할지 모르기 때문에 파싱 단계에서 에러가 발생된다.
    - 두번째는 개발자나 사용자가 조회할 데이터가 어느 테이블에 있는 칼럼을 말하는 것인지 쉽게 알 수 있게 하므로 SQL에 대한 가독성이나 유지보수성을 높이는 효과가 있다.
    - 조회할 칼럼명 앞에 테이블명을 붙여서 사용하는 습관을 기르는 것을 권장하지만, 하나의 SQL 문장 내에서 유일하게 사용하는 칼럼명이라면 칼럼명 앞에 테이블 명을 붙이지 않아도 된다.

- EQUI JOIN
  - ✓ [예제] 선수명, 등번호와 선수들이 소속해 있는 팀ID, 팀명 및 연고지를 출력한다.

[예제]

```
SELECT PLAYER.PLAYER_NAME, PLAYER.BACK_NO, PLAYER.TEAM_ID,
       TEAM.TEAM_NAME,  TEAM.REGION_NAME
FROM   PLAYER,TEAM
WHERE  PLAYER.TEAM_ID = TEAM.TEAM_ID;
```

실행 결과

PLAYER_NAME	BACK_NO	TEAM_ID	TEAM_NAME	REGION_NAME
오비나	26	K10	시티즌	대전
윤원일	45	K02	삼성블루윙즈	수원
페르난도	44	K04	유나이티드	인천
실바	45	K07	드래곤즈	전남
…	…	…	…	…

480 Rows Selected.

  - ✓ JOIN 대상이 되는 테이블명이 조회하고자 하는 칼럼 앞에 반복해서 나오는 것을 알 수 있다. 긴 테이블명을 계속 되풀이해서 입력하다보면 개발 생산성이 떨어지는 문제점과 함께 개발자의 실수가 발생할 가능성이 높아지는 문제가 있다.

- EQUI JOIN – TABLE ALIAS
  - ✓ SELECT 절에서 칼럼에 대한 ALIAS를 사용하는 것처럼 FROM 절의 테이블에 대해서도 ALIAS를 사용할 수 있다. 단일 테이블을 사용하는 SQL 문장에서는 필요성은 없지만 사용하더라도 에러는 발생하지 않으며, 여러 테이블을 사용하는 조인을 이용하는 경우는 매우 유용하게 사용할 수 있다.
  - ✓ 만약 테이블에 대한 ALIAS를 적용해서 SQL 문장을 작성했을 경우, WHERE 절과 SELECT 절에는 테이블명이 아닌 테이블에 대한 ALIAS를 사용해야 한다(미사용시 문법 에러 발생)는 점이다. 그러나, 하나의 SQL 문장 내에서 유일하게 사용하는 칼럼명이라면 칼럼명 앞에 ALIAS를 붙이지 않아도 된다.
  - ✓ [예제] 칼럼과 테이블 ALIAS를 적용하여 위 SQL을 수정한다.

[예제]

```
SELECT P.PLAYER_NAME 선수명, P.BACK_NO 백넘버, P.TEAM_ID 팀코드,
       T.TEAM_NAME 팀명,     T.REGION_NAME 연고지
FROM   PLAYER P,   TEAM T
WHERE  P.TEAM_ID = T.TEAM_ID;
```


- EQUI JOIN
  - ✓ 테이블 (개수-1)개의 JOIN 조건을 WHERE 절에 명시하고, 부수적으로 제한 조건을 논리 연산자를 통하여 추가로 입력하는 것이 가능하다.
  - ✓ [예제] 위 SQL 문장의 WHERE 절에 포지션이 골키퍼인(골키퍼에 대한 포지션 코드는 'GK'임) 선수들에 대한 데이터만을 백넘버 순으로 출력하는 SQL문을 만들어 본다.

[예제]

```
SELECT P.PLAYER_NAME 선수명, P.BACK_NO 백넘버,
       T.REGION_NAME 연고지, T.TEAM_NAME 팀명
FROM   PLAYER P,    TEAM T
WHERE  P.TEAM_ID  = T.TEAM_ID
AND    P.POSITION = 'GK'
ORDER  BY P.BACK_NO;
```

실행 결과

선수명	백넘버	연고지	팀명
정병지	1	포항	스틸러스
김운재	1	수원	삼성블루윙즈
…	…	…	…

43 Rows Selected. 골키퍼만 선별된 것임

- 테이블 관계도

팀(TEAM) 테이블

TEAM_ID	TEAM_NAME	REGION_NAME	STADIUM_ID
K05	현대모터스	전북	D03
K08	일화천마	성남	B02
K03	스틸러스	포항	C06
K07	드래곤즈	전남	D01
K09	FC서울	서울	B05
K04	유나이티드	인천	B01
K11	경남FC	경남	C05
K01	울산현대	울산	C04
K10	시티즌	대전	D02
K02	삼성블루윙즈	수원	B04
K12	광주상무	광주	A02
K06	아이파크	부산	C02
K13	강원FC	강원	A03
K14	제주유나이티드FC	제주	A04
K15	대구FC	대구	A05

운동장(STADIUM) 테이블

STADIUM_ID	STADIUM_NAME	SEAT_COUNT
D03	전주월드컵경기장	28000
B02	성남종합운동장	27000
C06	포항스틸야드	25000
D01	광양전용경기장	20009
B05	서울월드컵경기장	66806
B01	인천월드컵경기장	35000
C05	창원종합운동장	27085
C04	울산문수경기장	46102
D02	대전월드컵경기장	41000
B04	수원월드컵경기장	50000
A02	광주월드컵경기장	40245
C02	부산아시아드경기장	30000
A03	강릉종합경기장	33000
A04	제주월드컵경기장	42256
A05	대구월드컵경기장	66422
F01	대구시민경기장	30000
F02	부산시민경기장	30000
F03	일산경기장	20000
F04	마산경기장	20000
F05	안양경기장	20000

- 운동장 정보가 들어 있는 스타디움(STADIUM) 테이블이 있고, 팀의 정보가 들어 있는 팀(TEAM) 테이블이 있다.
- 스타디움(STADIUM) 테이블의 운동장코드(STADIUM_ID) 칼럼과 팀(TEAM) 테이블의 운동장코드(STADIUM_ID) 칼럼은 PK(스타디움 테이블의 운동장코드)와 FK(팀 테이블의 운동장코드)의 관계에 있다.

- EQUI JOIN
  - ✓ [예제] 팀(TEAM) 테이블과 구장(STADIUM) 테이블의 관계를 이용해서 소속팀이 가지고 있는 전용구장의 정보를 팀의 정보와 함께 출력하는 SQL문을 작성한다. 아래 3개의 SQL은 같은 결과를 얻을 수 있다.

[예제]

```
SELECT TEAM.REGION_NAME, TEAM.TEAM_NAME, TEAM.STADIUM_ID,
       STADIUM.STADIUM_NAME, STADIUM.SEAT_COUNT
FROM   TEAM,STADIUM
WHERE  TEAM.STADIUM_ID = STADIUM.STADIUM_ID;
```

[예제]

```
SELECT T.REGION_NAME,  T.TEAM_NAME, T.STADIUM_ID,
       S.STADIUM_NAME, S.SEAT_COUNT
FROM   TEAM T, STADIUM S
WHERE  T.STADIUM_ID  = S.STADIUM_ID;
```

[예제]

```
SELECT REGION_NAME,  TEAM_NAME, T.STADIUM_ID,
       STADIUM_NAME, SEAT_COUNT
FROM   TEAM T, STADIUM S
WHERE  T.STADIUM_ID  = S.STADIUM_ID;
```

- EQUI JOIN
  - ✓ 실무에서는 테이블 ALIAS를 적용하고, 유지보수성과 가독성을 위해 유니크한 칼럼명에도 테이블 ALIAS를 적용한 두번째 SQL문장 형식을 권고한다.

[예제]

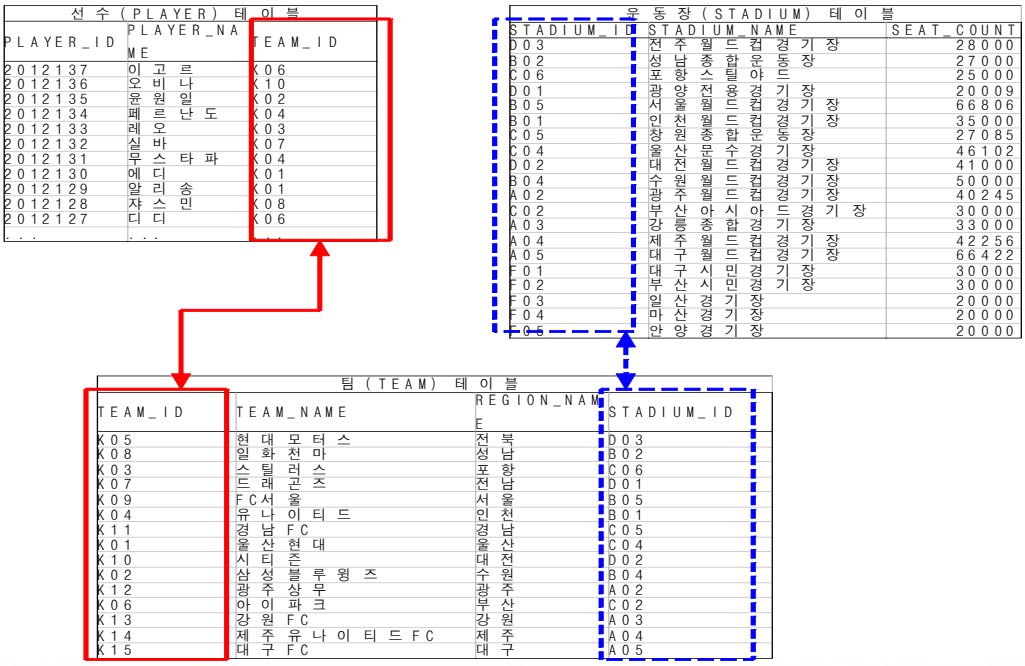
```
SELECT T.REGION_NAME,  T.TEAM_NAME, T.STADIUM_ID,
       S.STADIUM_NAME, S.SEAT_COUNT
FROM   TEAM T, STADIUM S
WHERE  T.STADIUM_ID  = S.STADIUM_ID;
```

실행 결과

REGION_NAME	TEAM_NAME	STADIUM_ID	STADIUM_NAME	SEAT_COUNT
전북	현대모터스	D03	전주월드컵경기장	28000
성남	일화천마	B02	성남종합운동장	27000
포항	스틸러스	C06	포항스틸야드	25000
…	…	…	…	…

15개의 행이 선택되었다.

- 테이블 관계도



- 3개 이상 테이블 JOIN
  - ✓ [예제] 선수명, 포지션과 선수들이 소속해 있는 팀명 및 연고지와 구장명을 출력한다. 세 개의 테이블에 대한 JOIN이므로 WHERE 절에 2개의 JOIN 조건이 들어 있다.

[예제]

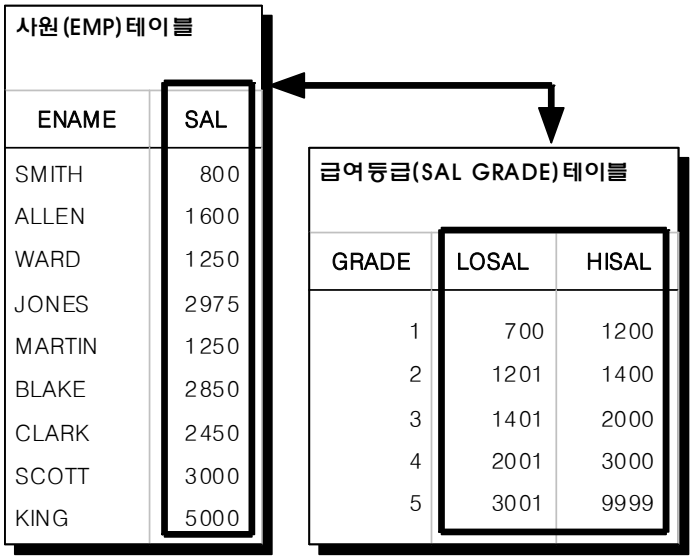
```
SELECT P.PLAYER_NAME 선수명, P.POSITION 포지션,
       T.REGION_NAME 연고지, T.TEAM_NAME 팀명,
       S.STADIUM_NAME 구장명
FROM   PLAYER P, TEAM T, STADIUM S
WHERE  P.TEAM_ID    = T.TEAM_ID
AND    T.STADIUM_ID = S.STADIUM_ID
ORDER  BY 선수명;
```

실행 결과

선수명	포지션	연고지	팀명	구장명
가비	MF	수원	삼성블루윙즈	수원월드컵경기장
가이모토	DF	성남	일화천마	성남종합운동장
강대희	MF	수원	삼성블루윙즈	수원월드컵경기장
…	…	…	…	…

480개의 행이 선택되었다.

- 테이블 관계도



- Non EQUI JOIN
  - ✓ 두 개의 테이블이 PK-FK로 연관관계를 가지거나 논리적으로 같은 값이 존재하는 경우에는 "=" 연산자를 이용하여 EQUI JOIN을 사용한다. 그러나 두 개의 테이블 간에 칼럼 값들이 서로 정확하게 일치하는 않는 경우에는 EQUI JOIN을 사용할 수 없다.
  - ✓ 이런 경우 Non EQUI JOIN을 시도하는데, "=" 연산자가 아닌 다른 (Between, >, >=, <, <= 등) 연산자들을 사용하여 JOIN을 수행하는 것이다.
  - ✓ 데이터 모델에 따라서 Non EQUI JOIN이 불가능한 경우도 있다. 아래 BETWEEN a AND b 조건은 Non EQUI JOIN의 한 사례일 뿐이다.

```
SELECT 테이블1.칼럼명, 테이블2.칼럼명, ...
FROM   테이블1, 테이블2
WHERE  테이블1.칼럼명1 BETWEEN 테이블2.칼럼명1 AND 테이블2.칼럼명2;
```


- Non EQUI JOIN
  - ✓ [예제] 사원별 급여와 어느 등급에 속하는지 알고 싶다는 요구사항에 대한 Non EQUI JOIN의 사례는 다음과 같다.

[예제]

```
SELECT E.ENAME 사원명, E.SAL 급여, S.GRADE 급여등급
FROM   EMP E, SALGRADE S
WHERE  E.SAL  BETWEEN  S.LOSAL AND S.HISAL;
```

실행 결과

사원명	급여	급여등급
JAMES	950	1
ADAMS	1100	1
WARD	1250	2
MARTIN	1250	2
...	...	...

14개의 행이 선택되었다.

Structured Query Language

II. SQL 기본 및 활용

제1장 SQL 기본
제9절 JOIN 핵심정리 및 연습문제

핵심정리 제9절 JOIN

- 두 개 이상의 테이블 들을 연결하여 데이터를 출력하는 것을 JOIN 이라고 하며,
- WHERE 절의 JOIN 조건에 대해서 EQUI JOIN과 Non EQUI JOIN으로 구분할 수 있다.

문제. 다음 SQL 문장에서 틀린 부분은 어디인가?

① SELECT PLAYER.PLAYER_NAME 선수명, TEAM.TEAM_NAME 팀명
② FROM PLAYER P, TEAM T
③ WHERE P.TEAM_ID = T.TEAM_ID
④ ORDER BY 선수명;

문제. 다음 SQL 문장에서 틀린 부분은 어디인가?

① SELECT PLAYER.PLAYER_NAME 선수명, TEAM.TEAM_NAME 팀명
② FROM PLAYER P, TEAM T
③ WHERE P.TEAM_ID = T.TEAM_ID
④ ORDER BY 선수명;

정답: ①
해설 :
FROM 절에 테이블에 대한 ALIAS를 사용했을 경우에, 중복된 이름이 있는 경우 SELECT 절에서는 반드시 ALIAS 명을 사용해야 한다.

문제. 4개의 테이블로부터 필요한 칼럼을 조회하려고 한다. 최소 몇 개의 JOIN 조건이 필요한가?

① 2개
② 3개
③ 4개
④ 5개

연습문제 제9절 JOIN

문제. 4개의 테이블로부터 필요한 칼럼을 조회하려고 한다. 최소 몇 개의 JOIN 조건이 필요한가?

① 2개
② 3개
③ 4개
④ 5개

정답: ②
해설 :
여러 테이블로부터 원하는 데이터를 조회하기 위해서는 전체 테이블 개수에서 최소 N-1 개 만큼의 JOIN 조건이 필요하다.

Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용 소개

장 소개 제2장 SQL 활용

- **제2장 SQL 활용**
  - ✓ 제1절 표준 조인
  - ✓ 제2절 집합 연산자
  - ✓ 제3절 계층형 질의와 셀프 조인
  - ✓ 제4절 서브쿼리
  - ✓ 제5절 그룹 함수
  - ✓ 제6절 윈도우 함수
  - ✓ 제7절 DCL
  - ✓ 제8절 절차형 SQL

- 제2장 소개
  - ✓ ANSI/ISO에서 정의한 FROM 절의 조인 조건과, 집합과 집합을 연결하는 집합 연산자의 사용법을 알아본다.
  - ✓ 데이터의 계층 구조를 질의하는 방법과 셀프 조인을 이해한다.
  - ✓ 메인쿼리와 주종의 개념을 가지는 서브쿼리에 대해 알아보고, 데이터 분석을 위한 그룹 함수와 윈도우 함수의 기능을 소개한다.
  - ✓ 데이터 액세스 권한에 대한 제어 SQL 문장과 사용자가 정의할 수 있는 절차형 SQL에 대해 살펴본다.

Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용
제1절 STANDARD JOIN 학습하기

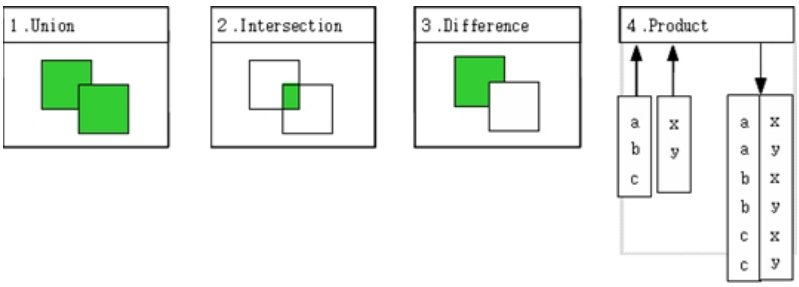
제1절 STANDARD JOIN

- STANDARD SQL
  - ✓ 1970년: Dr. E.F.Codd 관계형 DBMS (Relational DB) 논문 발표
  - ✓ 1974년: IBM SQL 개발
  - ✓ 1979년: Oracle 발표
  - ✓ 1980년: Sybase SQL Server 발표 (이후 Sybase ASE로 개명)
  - ✓ 1983년: IBM DB2 발표
  - ✓ 1986년: ANSI/ISO-SQL1 표준 제정
  - ✓ 1992년: ANSI/ISO-SQL2 표준 제정
  - ✓ 1993년: MS SQL Server 발표 (Windows OS, Sybase Code 활용)
  - ✓ 1999년: ANSI/ISO-SQL3 표준 제정
  - ✓ 2003년: ANSI/ISO-SQL4 추가 개정
  - ✓ 2008년: ANSI/ISO-SQL 추가 개정

제1절 STANDARD JOIN

- STANDARD SQL
  - ✓ 현재 우리가 사용하는 많은 시스템의 두뇌 역할을 하는 관계형 데이터베이스를 유일하게 접속할 수 있는 언어가 바로 SQL이다.
  - ✓ 초창기 SQL의 기본 기능을 정리했던 ANSI/ISO-SQL1에서 폭발적인 관계형 DBMS의 전성기를 주도했던 ANSI/ISO-SQL2를 통해 많은 기술적인 발전이 있었다.
  - ✓ 그러나, ANSI/ISO-SQL2의 경우 표준 SQL에 대한 명세가 부족한 부분도 있었고, DBMS 벤더 별로 문법이나 사용되는 용어의 차이가 너무 커져서 상호 호환성이나 SQL 학습 효율이 많이 부족한 문제가 발생하였다.
  - ✓ 이에 향후 SQL에서 ORDBMS 등 필요한 기능을 정리하고 호환 가능한 여러 기준을 제정한 것이 1999년에 정해진 ANSI/ISO-SQL3이다. 이후 가장 먼저 ANSI/ISO-SQL3의 기능을 시현한 것이 Oracle의 8i/9i 버전이라고 할 수 있다.
  - ✓ 참고로 2003년에 ANSI/ISO-SQL 기준이 소폭 추가 개정되었고 현재 사용되는 데이터베이스는 대부분 SQL 2003을 기준으로 하고 있다.

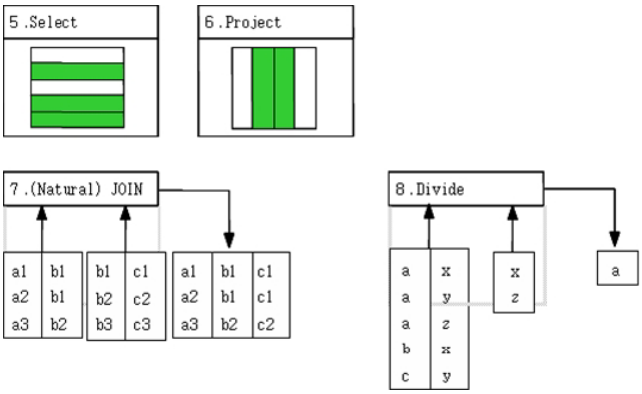
- E.F.CODD 일반 집합 연산자



✓ 일반 집합 연산자를 현재의 SQL과 비교하면

1. UNION 연산은 UNION 기능으로(이후 UNION ALL 추가),
2. INTERSECTION 연산은 INTERSECT 기능으로,
3. DIFFERENCE 연산은 EXCEPT(Oracle은 MINUS) 기능으로,
4. PRODUCT 연산은 CROSS JOIN 기능으로 구현되었다.

- E.F.CODD 순수 관계 연산자



✓ 순수 관계 연산자를 현재의 SQL 문장과 비교하면

5. SELECT 연산은 WHERE 절로 구현되었다. (SELECT 연산과 SELECT 절 의미 다름)
6. PROJECT 연산은 SELECT 절로 구현되었다.
7. (NATURAL) JOIN 연산은 다양한 JOIN 기능으로 구현되었다.
8. DIVIDE 연산은 현재 사용되지 않는다.

- FROM 절 JOIN 형태
  - ✓ ANSI/ISO SQL에서 규정한 JOIN 문법은 WHERE 절의 검색 조건과 테이블 간의 JOIN 조건을 구분 없이 사용하던 기존 방식을 그대로 사용할 수 있으면서, 추가된 선택 기능으로 테이블 간의 JOIN 조건을 FROM 절에서 명시적으로 정의할 수 있게 되었다.
    - INNER JOIN
    - NATURAL JOIN
    - USING 조건절
    - ON 조건절
    - CROSS JOIN
    - OUTER JOIN

- FROM 절 JOIN 형태
  - ✓ INNER JOIN은 WHERE 절에서부터 사용하던 JOIN의 DEFAULT 옵션으로 JOIN 조건에서 동일한 값이 있는 행만 반환한다. DEFAULT 옵션이므로 생략이 가능하지만, CROSS JOIN, OUTER JOIN과는 같이 사용할 수 없다.
  - ✓ NATURAL JOIN은 INNER JOIN의 하위 개념이므로 NATURAL INNER JOIN이라고 표시할 수 있으며, 결과는 NATURAL JOIN과 같다.
  - ✓ 새로운 SQL JOIN 문장 중에서 가장 중요하게 기억해야 하는 문장은 ON 조건절을 사용하는 경우이다. 과거 WHERE 절에서 JOIN 조건과 데이터 검증 조건이 같이 사용되어 용도가 불분명한 경우가 발생할 수 있었는데, WHERE 절의 JOIN 조건을 FROM 절의 ON 조건절로 분리하여 표시함으로써 사용자가 이해하기 쉽도록 한다.
  - ✓ ON 조건절의 경우 NATURAL JOIN처럼 JOIN 조건이 숨어 있지 않고, 명시적으로 JOIN 조건을 구분할 수 있고, NATURAL JOIN이나 USING 조건절처럼 칼럼명이 똑같아야 된다는 제약없이 칼럼명이 상호 다르더라도 JOIN 조건으로 사용할 수 있으므로 앞으로 가장 많이 사용될 것으로 예상된다. 다만, FROM 절에 테이블이 많이 사용될 경우 다소 복잡하게 보여 가독성이 떨어지는 단점이 있다.
  - ✓ 그런 측면에서 SQL Server의 경우 ON 조건절만 지원하고 NATURAL JOIN과 USING 조건절을 지원하지 않고 있는 것으로 보인다. 본 교재는 ANSI/ISO-SQL 기준에 NATURAL JOIN과 USING 조건절이 표시되어 있으므로 이부분도 설명을 하도록 한다.

- INNER JOIN
  - ✓ INNER JOIN은 OUTER(외부) JOIN과 대비하여 내부 JOIN이라고 하며 JOIN 조건에서 동일한 값이 있는 행만 반환한다.
  - ✓ INNER JOIN 표시는 그 동안 WHERE 절에서 사용하던 JOIN 조건을 FROM 절에서 정의하겠다는 표시이므로 USING 조건절이나 ON 조건절을 필수적으로 사용해야 한다.
  - ✓ [예제] 사원 번호와 사원 이름, 소속부서 코드와 소속부서 이름을 찾아본다.

[예제]

```
WHERE 절 JOIN 조건
SELECT EMP.DEPTNO, EMPNO, ENAME, DNAME
FROM   EMP, DEPT
WHERE  EMP.DEPTNO = DEPT.DEPTNO;

FROM 절 JOIN 조건
SELECT EMP.DEPTNO, EMPNO, ENAME, DNAME
FROM   EMP INNER JOIN DEPT
ON     EMP.DEPTNO = DEPT.DEPTNO;

INNER 키워드 생략
SELECT EMP.DEPTNO, EMPNO, ENAME, DNAME
FROM   EMP JOIN DEPT
ON     EMP.DEPTNO = DEPT.DEPTNO;
```


- NATURAL JOIN
  - ✓ NATURAL JOIN은 두 테이블 간의 동일한 이름을 갖는 모든 칼럼들에 대해 EQUI(=) JOIN을 수행한다.
  - ✓ NATURAL JOIN이 명시되었으면, 추가로 USING 조건절, ON 조건절, WHERE 절에서 JOIN 조건을 정의할 수 없다.
  - ✓ [예제] 사원 번호와 사원 이름, 소속부서 코드와 소속부서 이름을 찾아본다.

[예제]

```
FROM 절 JOIN 조건

SELECT DEPTNO, EMPNO, ENAME, DNAME
FROM   EMP NATURAL JOIN DEPT;
```

  - ✓ 별도의 JOIN 칼럼을 지정하지 않았지만, 두 개의 테이블에서 DEPTNO라는 공통된 칼럼을 자동으로 인식하여 JOIN을 처리한 것이다. JOIN에 사용된 칼럼들은 같은 데이터 유형이어야 하며, ALIAS나 테이블 명과 같은 접두사를 붙일 수 없다.
  - ✓ NATURAL JOIN은 JOIN이 되는 테이블의 데이터 성격(도메인)과 칼럼명 등이 동일해야 하는 제약 조건이 있다. 간혹 모델링 상의 부주의로 인해 동일한 칼럼명이더라도 다른 용도의 데이터를 저장하는 경우도 있으므로 주의해서 사용해야 한다.

- USING 조건절
  - ✓ NATURAL JOIN에서는 모든 일치되는 칼럼들에 대해 JOIN이 이루어지지만, FROM 절의 USING 조건절을 이용하면 같은 이름을 가진 칼럼들 중에서 원하는 칼럼에 대해서만 선택적으로 EQUI JOIN을 할 수가 있다.
  - ✓ [예제] DEPTNO, DNAME 2개의 칼럼을 이용해서 DEPT와 DEPT_TEMP 테이블을 조인한다

[예제]

```
FROM 절 JOIN 조건

SELECT DEPTNO, DNAME, DEPT.LOC
FROM   DEPT JOIN DEPT_TEMP
USING (DEPTNO, DNAME);
```

  - ✓ USING 조건절을 이용한 EQUI JOIN에서도 NATURAL JOIN과 마찬가지로 JOIN에 사용된 칼럼들은 같은 데이터 유형이어야 하며, ALIAS나 테이블 명과 같은 접두사를 붙일 수 없다.

- ON 조건절
  - ✓ JOIN 서술부(ON 조건절)와 非 JOIN 서술부(WHERE 조건절)를 분리하여 이해가 쉬우며, 칼럼명이 다르더라도 JOIN 조건을 사용할 수 있는 장점이 있다. (대부분의 DBMS가 지원)
  - ✓ [예제] 사원 테이블과 부서 테이블의 사원 번호와 사원 이름, 소속부서 코드와 소속부서 이름을 찾아본다.

[예제]

```
FROM 절 JOIN 조건

SELECT E.EMPNO, E.ENAME, E.DEPTNO, D.DNAME
FROM   EMP E JOIN DEPT D
ON    (E.DEPTNO = D.DEPTNO);
```

  - ✓ 임의의 JOIN 조건을 지정하거나, 이름이 다른 칼럼명을 JOIN 조건으로 사용하거나, JOIN 칼럼을 명시하기 위해서는 ON 조건절을 사용한다. ON 조건절에 사용된 괄호는 옵션 사항이다.

- ON 조건절
  - ✓ ON 조건절과 WHERE 검색 조건은 충돌 없이 사용할 수 있다.
  - ✓ [예제] 부서코드 30인 부서의 소속 사원 이름 및 소속 부서 코드, 부서 코드, 부서 이름을 찾아본다.

[예제]

```
FROM 절 JOIN 조건

SELECT E.ENAME, E.DEPTNO, D.DEPTNO, D.DNAME
FROM   EMP E JOIN DEPT D
ON     (E.DEPTNO = D.DEPTNO)
WHERE  E.DEPTNO = 30;
```

  - ✓ WHERE 절의 JOIN 조건과 같은 기능을 하면서도, 명시적으로 JOIN의 조건을 구분할 수 있으므로 가장 많이 사용될 것으로 예상된다. 다만, FROM 절에 테이블이 많이 사용될 경우 다소 복잡하게 보여 가독성이 떨어지는 단점이 있다.

- ON 조건절
  - ✓ [예제] 팀과 스타디움 테이블을 스타디움ID로 JOIN하여 팀이름, 스타디움ID, 스타디움 이름을 찾아본다.

[예제]

```
ON JOIN 조건
SELECT TEAM_NAME, TEAM.STADIUM_ID, STADIUM_NAME
FROM   TEAM JOIN STADIUM
ON     TEAM.STADIUM_ID = STADIUM.STADIUM_ID
ORDER BY STADIUM_ID;;

USING JOIN 조건
SELECT TEAM_NAME, STADIUM_ID, STADIUM_NAME
FROM   TEAM JOIN STADIUM
USING  (STADIUM_ID)
ORDER BY STADIUM_ID;;

WHERE 절 JOIN 조건
SELECT TEAM_NAME, TEAM.STADIUM_ID, STADIUM_NAME
FROM   TEAM, STADIUM
WHERE  TEAM.STADIUM_ID = STADIUM.STADIUM_ID
ORDER BY STADIUM_ID
```

- ON 조건절
  - ✓ [예제] 팀과 스타디움 테이블을 팀ID로 JOIN하여 팀이름, 팀ID, 스타디움 이름을 찾아본다. STADIUM에는 팀ID가 HOMETEAM_ID라는 칼럼으로 표시되어 있다

[예제]

```
ON JOIN 조건
SELECT TEAM_NAME, TEAM_ID, STADIUM_NAME
FROM   TEAM JOIN STADIUM
ON     TEAM.TEAM_ID = STADIUM.HOMETEAM_ID
ORDER BY TEAM_ID;

WHERE 절 JOIN 조건
SELECT TEAM_NAME, TEAM_ID, STADIUM_NAME
FROM   TEAM, STADIUM
WHERE  TEAM.TEAM_ID = STADIUM.HOMETEAM_ID
ORDER BY TEAM_ID;

위 SQL은 TEAM_ID와 HOMETEAM_ID라는 다른 이름의 칼럼을 사용하기 때문에 USING 조건절을 사용할 수 없다.
```


- ON 조건절 – 다중 조인
  - ✓ [예제] GK 포지션의 선수별 연고지명, 팀명, 구장명을 출력한다.

[예제]

```
ON JOIN 조건
SELECT P.PLAYER_NAME 선수명, P.POSITION 포지션, T.REGION_NAME 연고지명,
       T.TEAM_NAME 팀명, S.STADIUM_NAME 구장명
FROM   PLAYER P JOIN TEAM T
ON     P.TEAM_ID    = T.TEAM_ID
       JOIN   STADIUM S
ON     T.STADIUM_ID = S.STADIUM_ID
WHERE  P.POSITION = 'GK'
ORDER BY 선수명;

WHERE 절 JOIN 조건
SELECT P.PLAYER_NAME 선수명, P.POSITION 포지션, T.REGION_NAME 연고지명,
       T.TEAM_NAME 팀명, S.STADIUM_NAME 구장명
FROM   PLAYER P, TEAM T, STADIUM S
WHERE  P.TEAM_ID    = T.TEAM_ID
AND    T.STADIUM_ID = S.STADIUM_ID
AND    P.POSITION   = 'GK'
ORDER BY 선수명;
```

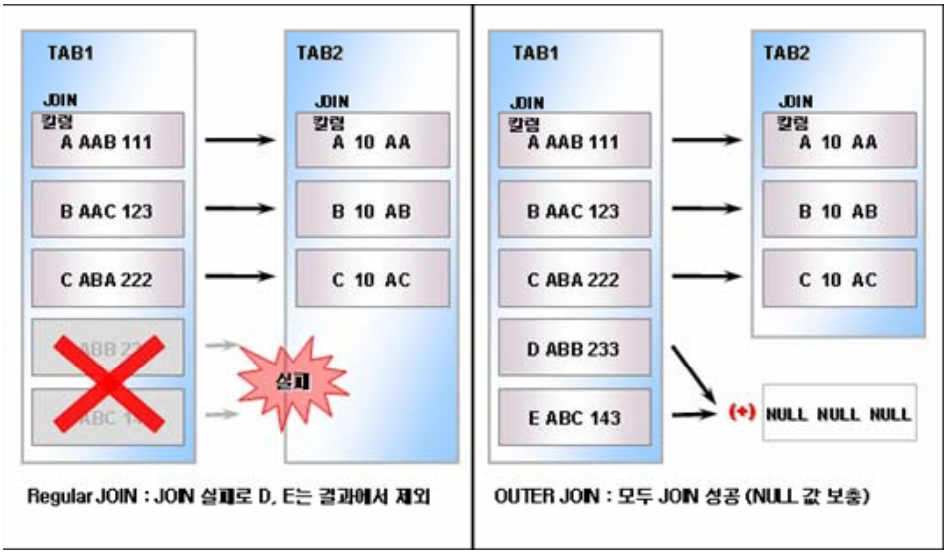
- CROSS JOIN
  - ✓ CROSS JOIN은 E.F.CODD 박사가 언급한 일반 집합 연산자의 PRODUCT의 개념으로 테이블 간 JOIN 조건이 없는 경우 생길 수 있는 모든 데이터의 조합을 말한다.
  - ✓ CARTESIAN PRODUCT 또는 CROSS PRODUCT와 같은 표현으로, 결과는 양쪽 집합의 M*N 건의 데이터 조합이 발생한다.
  - ✓ 정상적인 데이터 모델이라면 CROSS PRODUCT가 필요한 경우는 많지 않지만, 간혹 튜닝이나 리포트를 작성하기 위해 고의적으로 사용하는 경우가 있을 수 있다.
  - ✓ 데이터웨어하우스의 개별 DIMENSION(차원)을 FACT(사실) 칼럼과 JOIN하기 전에 모든 DIMENSION의 CROSS PRODUCT를 먼저 구할 때 유용하게 사용할 수 있다.
  - ✓ [예제] 조인 조건 없이 사원과 부서 테이블을 조인해본다.

[예제]

```
SELECT ENAME, DNAME
FROM   EMP CROSS JOIN DEPT
ORDER  BY ENAME;
56개의 행이 선택되었다.
```

  - ✓ 56건의 결과 데이터는 EMP 14건 * DEPT 4건의 데이터 조합 건수이다.

- OUTER JOIN
  - ✓ INNER(내부) JOIN과 대비하여 외부 JOIN이라고 불리며, JOIN 조건에서 동일한 값이 없는 행도 반환할 때 사용할 수 있다.



- OUTER JOIN
  - ✓ 과거 OUTER JOIN을 위해 Oracle은 JOIN 칼럼 뒤에 '(+)' 를 표시하였고, Sybase는 비교 연산자의 앞이나 뒤에 '(+)' 를 표시했었는데, JOIN 조건과 WHERE 절 검색 조건이 불명확한 단점, IN이나 OR 연산자 사용시 에러 발생, '(+)' 표시가 누락된 칼럼 존재시 OUTER JOIN 오류 발생, FULL OUTER JOIN 미지원 등 불편함이 많았다.
  - ✓ STANDARD JOIN을 사용함으로써 OUTER JOIN의 많은 문제점을 해결할 수 있고, 대부분의 관계형 DBMS 간에 호환성을 확보할 수 있으므로 명시적인 OUTER JOIN을 사용할 것을 적극적으로 권장한다.
  - ✓ 추가로 OUTER JOIN 역시 JOIN 조건을 FROM 절에서 정의하겠다는 표시이므로 USING 조건절이나 ON 조건절을 필수적으로 사용해야 한다.
  - ✓ LEFT/RIGHT OUTER JOIN의 경우에는 기준이 되는 테이블이 조인 수행시 무조건 드라이빙 테이블이 된다. 옵티마이저는 이 원칙에 위배되는 다른 실행계획을 고려하지 않는다. SQL 성능 저하의 원인이 될 수 있으므로 필요한 경우만 OUTER 조인을 사용해야 한다.

- LEFT OUTER JOIN
  - ✓ 조인 수행시 먼저 표기된 좌측 테이블에 해당하는 데이터를 모두 읽은 후, 나중 표기된 우측 테이블에서 JOIN 대상 데이터를 읽어 온다.
  - ✓ 우측 테이블의 JOIN 칼럼에서 조인 조건 값이 없는 경우에는 우측 테이블에서 가져오는 칼럼들은 NULL 값으로 채운다.
  - ✓ LEFT JOIN으로 OUTER 키워드를 생략해서 사용할 수 있다.
  - ✓ [예제] STADIUM에 등록된 운동장 중에는 홈팀이 없는 경기장도 있다. STADIUM과 TEAM을 JOIN 하되 홈팀이 없는 경기장의 정보도 같이 출력하도록 한다.

[예제]

```
SELECT STADIUM_NAME, STADIUM.STADIUM_ID, SEAT_COUNT, HOMETEAM_ID, TEAM_NAME
FROM   STADIUM LEFT OUTER JOIN TEAM
ON     STADIUM.HOMETEAM_ID = TEAM.TEAM_ID
ORDER BY HOMETEAM_ID;
```

- RIGHT OUTER JOIN
  - ✓ 조인 수행시 나중 표기된 우측 테이블에 해당하는 데이터를 모두 읽은 후, 먼저 표기된 좌측 테이블에서 JOIN 대상 데이터를 읽어 온다.
  - ✓ 좌측 테이블의 JOIN 칼럼에서 조인 조건 값이 없는 경우에는 좌측 테이블에서 가져오는 칼럼들은 NULL 값으로 채운다.
  - ✓ RIGHT JOIN으로 OUTER 키워드를 생략해서 사용할 수 있다.
  - ✓ [예제] 부서 및 사원 정보를 출력하되, 사원이 없는 부서의 정보도 같이 출력한다.

[예제]

```
SELECT E.ENAME, D.DEPTNO, D.DNAME
FROM   EMP E RIGHT OUTER JOIN DEPT D
ON     E.DEPTNO = D.DEPTNO;
```


- FULL OUTER JOIN
  - ✓ 조인 수행시 좌측, 우측 테이블의 모든 데이터를 읽어 JOIN하여 결과를 생성한다.
  - ✓ 즉, TABLE A와 B가 있을 때(TABLE 'A', 'B' 모두 기준이 됨), RIGHT OUTER JOIN과 LEFT OUTER JOIN의 결과를 합집합으로 처리한 결과와 동일하다.
  - ✓ 단, UNION ALL이 아닌 UNION 기능과 같으므로 중복되는 데이터는 삭제한다.
  - ✓ FULL JOIN으로 OUTER 키워드를 생략해서 사용할 수 있다.

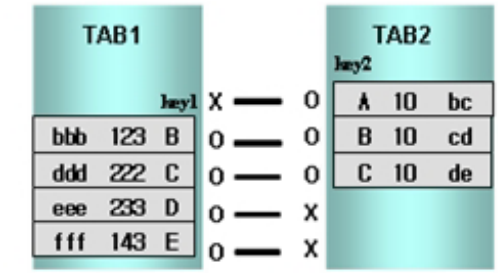
[예제]

```
SELECT *
FROM DEPT FULL OUTER JOIN DEPT_TEMP
ON   DEPT.DEPTNO = DEPT_TEMP.DEPTNO;
```

[예제]

```
SELECT L.DEPTNO, L.DNAME, L.LOC, R.DEPTNO, R.DNAME, R.LOC
FROM   DEPT L LEFT OUTER JOIN DEPT_TEMP R
ON     L.DEPTNO = R.DEPTNO
UNION
SELECT L.DEPTNO, L.DNAME, L.LOC, R.DEPTNO, R.DNAME, R.LOC
FROM   DEPT L RIGHT OUTER JOIN DEPT_TEMP R
ON     L.DEPTNO = R.DEPTNO;
```

- INNER vs OUTER vs CROSS JOIN 비교



INNER JOIN

```
SELECT  X.KEY1, Y.KEY2
  FROM  TAB1 X INNER JOIN TAB2 Y
    ON  (X.KEY1 = Y.KEY2)
```

CARTESIAN PRODUCT

```
SELECT  X.KEY1, Y.KEY2
  FROM  TAB1 X CROSS JOIN TAB2 Y
```

LEFT OUTER JOIN

```
SELECT  X.KEY1, Y.KEY2
  FROM  TAB1 X LEFT OUTER JOIN TAB2 Y
    ON  (X.KEY1 = Y.KEY2)
```

RIGHT OUTER JOIN

```
SELECT  X.KEY1, Y.KEY2
  FROM  TAB1 X RIGHT OUTER JOIN TAB2 Y
    ON  (X.KEY1 = Y.KEY2)
```

FULL OUTER JOIN

```
SELECT  X.KEY1, Y.KEY2
  FROM  TAB1 X FULL OUTER JOIN TAB2 Y
    ON  (X.KEY1 = Y.KEY2)
```


- INNER vs OUTER vs CROSS JOIN 비교
  - ✓ 첫번째, INNER JOIN의 결과는 다음과 같다.
    - 양쪽 테이블에 모두 존재하는 키 값이 B-B, C-C 인 2건이 출력된다.
  - ✓ 두번째, LEFT OUTER JOIN의 결과는 다음과 같다.
    - TAB1를 기준으로 키 값 조합이 B-B, C-C, D-NULL, E-NULL 인 4건이 출력된다.
  - ✓ 세번째, RIGHT OUTER JOIN의 결과는 다음과 같다.
    - TAB2을 기준으로 키 값 조합이 NULL-A, B-B, C-C 인 3건이 출력된다.
  - ✓ 네번째, FULL OUTER JOIN의 결과는 다음과 같다.
    - 양쪽 테이블을 기준으로 키 값 조합이 NULL-A, B-B, C-C, D-NULL, E-NULL 인 5건이 출력된다.
  - ✓ 다섯번째, CROSS JOIN(CARTESIAN PRODUCT)의 결과는 다음과 같다.
    - JOIN 가능한 모든 경우의 수를 표시하지만, 단, OUTER JOIN은 제외한다.
    - 양쪽 테이블 TAB1과 TAB2의 데이터를 곱한 개수인 4*3= 12건이 추출됨
    - 키 값 조합이 B-A, B-B, B-C, C-A, C-B, C-C, D-A, D-B, D-C, E-A, E-B, E-C 인 12건이 출력된다.

Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용
제1절 STANDARD JOIN
핵심정리 및 연습문제

핵심정리 제1절 STANDARD JOIN

- ANSI-SQL에서 규정한
  - ✓ INNER JOIN,
  - ✓ NATURAL JOIN,
  - ✓ USING 조건절,
  - ✓ ON 조건절,
  - ✓ CROSS JOIN,
  - ✓ OUTER JOIN 문법을 통해
- 사용자는 테이블 간의 JOIN 조건을
- FROM 절에서 명시적으로 정의할 수 있다.

연습문제 제1절 STANDARD JOIN

문제. 다음 JOIN 종류의 설명 중 틀린 것은 무엇인가?

① EQUI JOIN은 반드시 PK, FK 관계에 의해서만 성립된다.

② NON-EQUI JOIN은 등가 조건이 성립되지 않은 테이블에 JOIN을 걸어 주는 방법이다.

③ OUTER JOIN은 JOIN 조건을 만족하지 않는 데이터도 볼 수 있는 JOIN 방법이다.

④ SELF JOIN은 하나의 테이블을 논리적으로 분리시켜 EQUI JOIN을 이용하는 방법이다.

연습문제 제1절 STANDARD JOIN

문제. 다음 JOIN 종류의 설명 중 틀린 것은 무엇인가?

① EQUI JOIN은 반드시 PK, FK 관계에 의해서만 성립된다.

② NON-EQUI JOIN은 등가 조건이 성립되지 않은 테이블에 JOIN을 걸어 주는 방법이다.

③ OUTER JOIN은 JOIN 조건을 만족하지 않는 데이터도 볼 수 있는 JOIN 방법이다.

④ SELF JOIN은 하나의 테이블을 논리적으로 분리시켜 EQUI JOIN을 이용하는 방법이다.

정답 : ①

해설 :

EQUI JOIN은 반드시 PK, FK관계에 의해서만 성립되는 것은 아니다.

조인 칼럼이 맵핑이 가능하면 사용할 수 있다.

Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용
제2절 집합 연산자
학습하기

제2절 집합 연산자(SET OPERATOR)

- 집합 연산자(SET OPERATOR)
  - ✓ 기존의 조인에서는 FROM절에 검색하고자 하는 테이블을 나열하고, WHERE 절에 조인 조건을 기술하여 원하는 데이터를 조회할 수 있었다. 하지만 집합 연산자는 여러 개의 질의의 결과를 연결하여 하나로 결합하는 방식을 사용한다. 즉, 집합 연산자는 2개 이상의 질의 결과를 하나의 결과로 만들어 준다.
  - ✓ 일반적으로 집합 연산자를 사용하는 상황은 서로 다른 테이블에서 유사한 형태의 결과를 반환하는 것을 하나의 결과로 합치고자 할 때와 동일 테이블에서 서로 다른 질의를 수행하여 결과를 합치고자 할 때 사용할 수 있다. 이외에도 튜닝관점에서 실행계획을 분리하고자 하는 목적으로 사용할 수 있다.
  - ✓ 집합 연산자를 사용하기 위해서는 다음 제약조건을 만족해야 한다. SELECT 절의 칼럼 수가 동일하고 SELECT절의 동일 위치에 존재하는 칼럼의 데이터 타입이 상호 호환 가능(반드시 동일한 데이터 타입일 필요는 없음)해야 한다. 그렇지 않으면 데이터베이스가 오류를 반환한다.

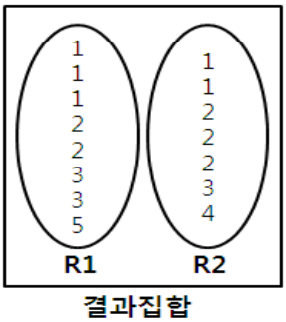
제2절 집합 연산자(SET OPERATOR)

- **집합 연산자 종류**

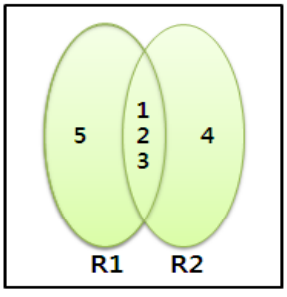
집합 연산자	연산자 의미
UNION	여러 개의 SQL문의 결과에 대한 합집합으로 결과에서 모든 중복된 행은 하나의 행으로 만든다.
UNION ALL	여러 개의 SQL문의 결과에 대한 합집합으로 중복된 행도 그대로 결과로 표시된다. 즉, 단순히 결과만 합쳐놓은 것이다. 일반적으로 여러 질의 결과가 상호 배타적(Exclusive)일 때 많이 사용한다. 개별 SQL문의 결과가 서로 중복되지 않는 경우, UNION과 결과가 동일하다. (결과의 정렬 순서에는 차이가 있을 수 있음)
INTERSECT	여러 개의 SQL문의 결과에 대한 교집합이다. 중복된 행은 하나의 행으로 만든다.
EXCEPT (MINUS)	앞의 SQL문의 결과에서 뒤의 SQL문의 결과에 대한 차집합이다. 중복된 행은 하나의 행으로 만든다. (오라클은 MINUS를 사용함, ANSI는 EXCEPT 용어 사용)

제2절 집합 연산자(SET OPERATOR)

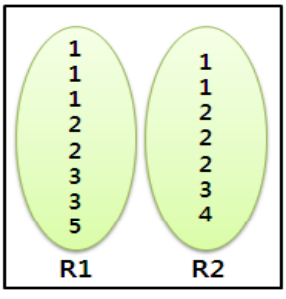
- **집합 연산자 연산**



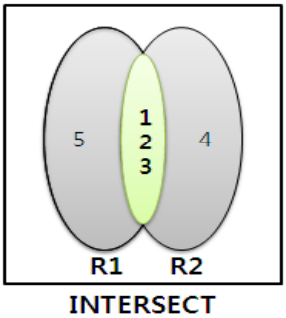
결과집합



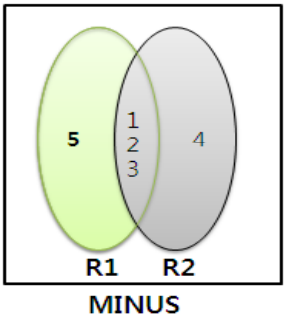
UNION



UNION ALL



INTERSECT



MINUS

- 집합연산자 사용
  - ✓ 집합 연산자는 사용상의 제약조건을 만족한다면 어떤 형태의 SELECT문이라도 이용할 수 있다. 집합 연산자는 2개 이상의 SELECT문을 연결하는 것이다.
  - ✓ ORDER BY는 집합 연산을 적용한 최종 결과에 대한 정렬 처리이므로 가장 마지막 줄에 한번만 기술한다.

```
SELECT PLAYER_NAME 선수명, BACK_NO 백넘버
FROM   PLAYER
WHERE  TEAM_ID = 'K02'
UNION  ALL
SELECT PLAYER_NAME 선수명, BACK_NO 백넘버
FROM   PLAYER
WHERE  TEAM_ID = 'K07'
ORDER  BY 1;
```


- UNION
  - ✓ [예제] K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과 K-리그 소속 선수 중 소속이 전남드레곤즈팀인 선수들의 집합의 합집합

```
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  TEAM_ID = 'K02'
UNION  ALL
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  TEAM_ID = 'K07' ;

100개의 행이 선택되었습니다.
```

- UNION
  - ✓ 삼성 블루윙즈팀인 선수들과 전남 드레곤즈팀의 선수들의 합집합이라는 것은 WHERE절에 IN 또는 OR 연산자로도 변환이 가능하다. 다만 IN 또는 OR 연산자를 사용할 경우에는 결과의 표시 순서가 달라질 수 있다.
  - ✓ 집합이라는 관점에서는 결과가 표시되는 순서가 틀렸다고 두 집합이 서로 다르다고 말할 수 없다. 만약, 결과의 동일한 표시 순서를 원한다면 ORDER BY 절을 사용해서 명시적으로 정렬 순서를 정의하는 것이 바람직하다.

```
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  TEAM_ID = 'K02' OR TEAM_ID = 'K07';
```

```
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  TEAM_ID IN ('K02', 'K07');
```


- UNION ALL
  - ✓ 예제에서 사용된 두개의 질의는 조건 팀이 다르므로 결과가 상호배타적인 경우이다. 같은 결과를 얻을 수 있다면, 데이터 중복을 검증하기 위한 비용이 발생하는 UNION 연산자가 아니라 더 효율적인 UNION ALL 연산자를 권고한다.

```
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  TEAM_ID = 'K02'
UNION  ALL
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  TEAM_ID = 'K07';

100개의 행이 선택되었습니다.
```

- UNION vs UNION ALL
  - ✓ [예제] K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과 K-리그 소속 선수 중 포지션이 골키퍼(GK)인 선수들의 집합의 합집합

```
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  TEAM_ID  = 'K02'
UNION
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  POSITION = 'GK';

88개의 행이 선택되었습니다.
```

결과 49건

결과 43건

49건+43건-중복4건=88건

※ WHERE TEAM_ID = 'K02' OR POSITION = 'GK'; 변환 가능

- UNION vs UNION ALL
  - ✓ [예제]에서 UNION 연산을 UNION ALL 연산으로 변경해본다. 결과가 다른 이유는 UNION은 결과에서 중복이 존재할 경우 중복을 제외시키지만 UNION ALL은 각각의 질의 결과를 단순히 결합시켜 줄 뿐 중복된 결과를 제외 시키지 않기 때문이다.

```
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  TEAM_ID  = 'K02'
UNION  ALL
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  POSITION = 'GK' ;

92개의 행이 선택되었습니다.
```

결과 49건

결과 43건

제2절 집합 연산자(SET OPERATOR)

- UNION vs UNION ALL
  - ✓ 일반적으로 현업에서 UNION이 꼭 필요한 경우는 많지 않다. 그리고, UNION ALL 사용시 중복되는 데이터가 배제해야 될 대상이 아니라 나름대로 의미를 가지는 경우도 많다.
  - ✓ 앞의 예제에서 차이가 나는 4명은 삼성블루윙즈이면서 골키퍼 포지션인 사람이 4명 있다는 정보를 가지는 것이다.
  - ✓ 중복을 배제한 UNION이 꼭 필요한 경우가 아니라면, UNION ALL을 우선적으로 고려한다.
  - ✓ RDBMS 초기에는 UNION만 있었으나 이후 실제 사용시 필요성에 의해 UNION ALL 연산자가 추가되었다.

제2절 집합 연산자(SET OPERATOR)

- UNION ALL
  - ✓ 이질적인 성격의 데이터를 한 화면에 보고 싶은 경우에도 UNION ALL을 유용하게 사용할 수 있다.
  - ✓ [예제] K-리그 소속 선수 중 포지션별 평균키에 대한 집합과 K-리그 소속 선수 중 팀별 평균키에 대한 집합의 합집합을 구한다.

```
SELECT 'P' 구분코드, POSITION 포지션, AVG(HEIGHT) 평균키
FROM   PLAYER
GROUP  BY POSITION
UNION  ALL
SELECT 'T' 구분코드, TEAM_ID 팀명,    AVG(HEIGHT) 평균키
FROM   PLAYER
GROUP  BY TEAM_ID
ORDER  BY 1, 2;
```

- UNION ALL
  - ✓ 포지션과 팀별이란 다른 기준에 의해 만들어진 이질적인 데이터를 하나의 결과로 확인할 수 있다. 데이터 성격을 구분하기 위한 구분코드 칼럼이 사용자에 의해 추가되었다.
  - ✓ 데이터 HEADING 부분은 첫 번째 SQL문에서 사용된 HEADING이 적용된다는 것을 알 수 있다. 두 번째 SELECT절에 사용된 '팀명'이 아니라 첫 번째 SELECT절에서 사용된 '포지션'이 최종 HEADING으로 사용되었다.

```
구분코드 포지션              평균키
-------- ----------  ----------------------
P        DF                 180.409
P        FW                  179.91
P        GK                 186.256
P        MF                 176.309
...
T        K01                180.089
T        K02                179.067
T        K03                179.911
T        K04                180.511
...
23 개의 행이 선택되었습니다
```


- INTERSECT
  - ✓ K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들 집합과 K-리그 소속 선수 중 포지션이 골키퍼(GK)인 선수들 집합의 교집합

```
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  TEAM_ID  = 'K02'                          결과 49건
INTERSECT
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키                  결과 43건
FROM   PLAYER
WHERE  POSITION = 'GK';

4개의 행이 선택되었습니다.                      49건과 43건의 중복4건
```

※ WHERE TEAM_ID = 'K02' AND POSITION = 'GK'; 변환 가능

- INTERSECT
  - ✓ INTERSECT 연산자는 IN 서브쿼리 또는 EXISTS 서브쿼리를 이용한 SQL문으로 변경 가능하다. (4절 학습)

```
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  TEAM_ID = 'K02'
AND    PLAYER_ID IN (SELECT PLAYER_ID FROM PLAYER
                     WHERE POSITION = 'GK');
```

```
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER X
WHERE  X.TEAM_ID = 'K02'
  AND  EXISTS (SELECT 1 FROM PLAYER Y
               WHERE Y.PLAYER_ID = X.PLAYER_ID
                 AND Y.POSITION  = 'GK');
```


- EXCEPT(MINUS)
  - ✓ [예제] K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합에서 K-리그 소속 선수 중 포지션이 미드필더(MF)가 아닌 선수들의 집합

```
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  TEAM_ID  = 'K02'
MINUS
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  POSITION = 'MF';

31개의 행이 선택되었습니다.
```

결과 49건

결과 18건

49건-MF18건=31건

※ SQL Server, DB2는 MINUS 대신 EXCEPT 사용

- EXCEPT(MINUS)
  - ✓ MINUS 연산자를 사용하지 않고 논리 연산자를 이용하여 동일한 결과의 SQL 문을 작성할 수 있다.

```
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE  TEAM_ID = 'K02' AND POSITION <> 'MF';
```

  - ✓ MINUS 연산자는 NOT EXISTS 또는 NOT IN 서브쿼리를 이용한 SQL문으로도 변경 가능하다.

```
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE TEAM_ID = 'K02'
AND    PLAYER_ID NOT IN (SELECT PLAYER_ID FROM PLAYER
                         WHERE POSITION = 'MF');
```


Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용
제2절 집합 연산자(SET OPERATOR)
핵심정리 및 연습문제

핵심정리 제2절 집합 연산자(SET OPERATOR)

- 두 개 이상의 테이블에서 JOIN을 사용하지 않고, SET 연산자는 여러 개의 SQL문을 연결하여 데이터를 결합하는 방식을 사용한다.
- UNION은 합집합,
- UNION ALL은 확장된 합집합,
- INTERSECT는 교집합,
- EXCEPT/MINUS는 차집합을 나타낸다.

연습문제 제2절 집합 연산자(SET OPERATOR)

문제. 다음 SET 연산자에 대한 설명 중 틀린 것은 무엇인가?

① UNION 연산자는 조회 결과에 대한 합집합을 나타내며 자동으로 정렬을 해준다.

② UNION ALL 연산자는 조회 결과를 정렬하고 중복되는 데이터를 한 번만 표현한다.

③ INTERSECT 연산자는 조회 결과에 대한 교집합을 의미한다.

④ EXCEPT 연산자는 조회 결과에 대한 차집합을 의미한다.

문제. 다음 SET 연산자에 대한 설명 중 틀린 것은 무엇인가?

① UNION 연산자는 조회 결과에 대한 합집합을 나타내며 자동으로 정렬을 해준다.

② UNION ALL 연산자는 조회 결과를 정렬하고 중복되는 데이터를 한 번만 표현한다.

③ INTERSECT 연산자는 조회 결과에 대한 교집합을 의미한다.

④ EXCEPT 연산자는 조회 결과에 대한 차집합을 의미한다.

정답 : ②

해설 :

UNION ALL 연산자는 조회 결과에 대해 별도의 정렬 작업을 하지 않는다. 또한 중복 데이터에 대해서도 삭제하지 않고 여러 번 중복 표현한다.

Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용

제3절 계층형 질의와 셀프 조인

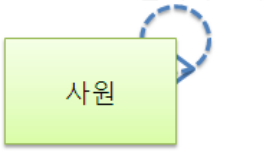
학습하기

제3절 계층형 질의와 셀프 조인

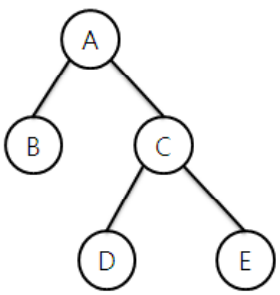
- **계층형 질의**
  - ✓ 테이블에 계층형 데이터가 존재하는 경우 데이터를 조회하기 위해서 계층형 질의(Hierarchical Query)를 사용한다. 계층형 데이터란 동일 테이블에 계층적으로 상위와 하위 데이터가 포함된 데이터를 말한다.
  - ✓ 예를 들어, 사원 테이블에서는 사원들 사이에 상위 사원(관리자)과 하위 사원 관계가 존재하고 조직 테이블에서는 조직들 사이에 상위 조직과 하위 조직 관계가 존재한다.
  - ✓ 엔터티를 순환관계 데이터 모델로 설계할 경우 계층형 데이터가 발생한다. 순환관계 데이터 모델의 예로는 조직도, 사원(관리자포함), 메뉴 등이 있다.

제3절 계층형 질의와 셀프 조인

- **계층형 구조**
  - ✓ (1)은 계층형 구조를 BARKER 표기법으로 표현한 ERD이다.
  - ✓ (2)의 계층형 구조에서 A의 하위 사원은 B, C이고,
    B 밑에는 하위 사원이 없고 C의 하위 사원은 D, E가 있다.
  - ✓ (3)의 표는 계층형 구조를 샘플 데이터로 표현한 것이다.



(1)순환관계 데이터 모델



(2)계층형 구조

사원	관리자
A	
B	A
C	A
D	C
E	C

(3)샘플 데이터

- Oracle 계층형 질의

```
SELECT    칼럼명...
FROM      테이블명
WHERE     조건...
START     WITH  조건
CONNECT BY   PRIOR 조건;
```

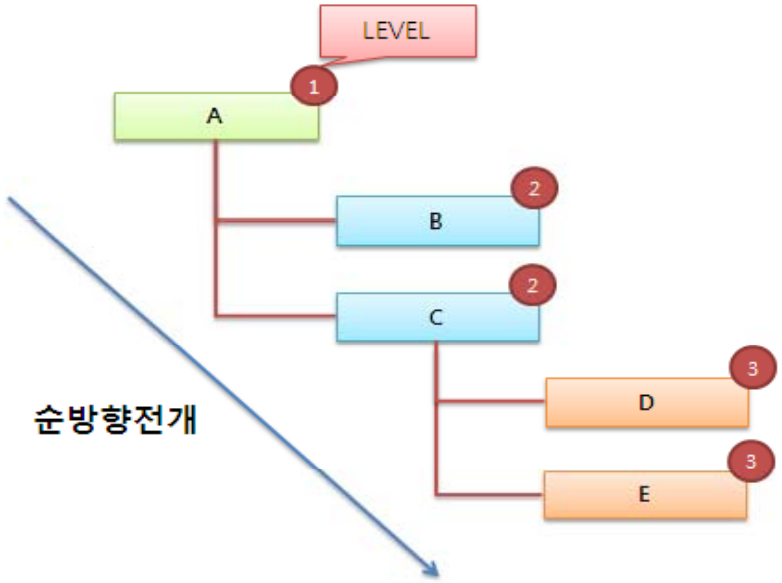
✓ START WITH절은 계층 구조 전개의 시작 위치를 지정하는 구문이다.
즉, 루트(시작) 데이터를 지정한다. (액세스)

✓ CONNECT BY절은 다음에 전개될 자식 데이터를 지정하는 구문이다.
자식 데이터는 CONNECT BY절에 주어진 조건을 만족해야 한다. (조인)

✓ "PRIOR 자식 = 부모" 형태를 사용하면 계층구조에서 부모 → 자식 방향으로 내려가는 순방향으로 전개된다.

✓ "PRIOR 부모 = 자식" 형태를 사용하면 계층구조에서 자식 → 부모 방향으로 올라가는 역방향으로 전개된다.

- Oracle 순방향 전개



- Oracle 순방향 전개

```
SELECT  LEVEL, LPAD(' ', 4 * (LEVEL-1)) || EMPNO 사원, MGR 관리자,
        CONNECT_BY_ISLEAF ISLEAF
FROM    EMP
START   WITH     MGR IS NULL
CONNECT BY PRIOR EMPNO = MGR;
```

```
LEVEL      사원          관리자   ISLEAF
-----  -------------  ------  -------
1          A                       0
2            B           A         1
2            C           A         0
3              D         C         1
3              E         C         1
```

가상 칼럼	설명
LEVEL	루트 데이터이면 1 그 하위 데이터이면 2이다. 리프(Leaf) 데이터까지 1씩 증가한다.
CONNECT_BY_ISLEAF	전개 과정에서 해당 데이터가 리프 데이터이면 1 그렇지 않으면 0이다.

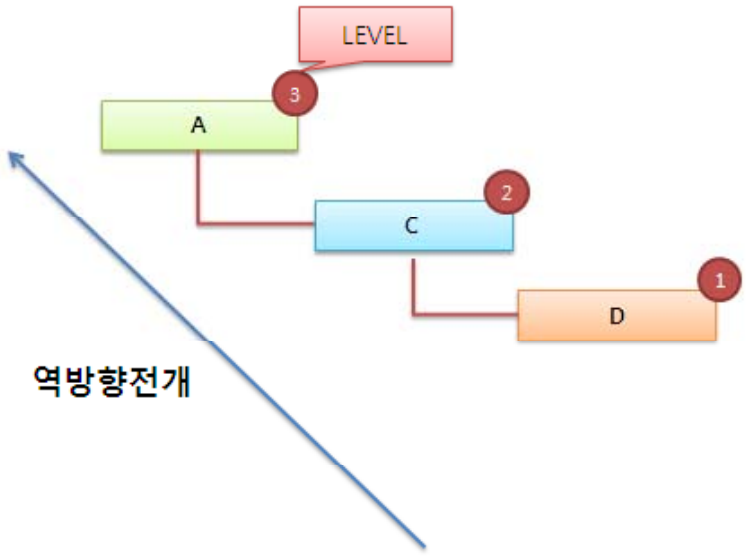
- Oracle 순방향 전개

```
SELECT  CONNECT_BY_ROOT EMPNO 시작사원, SYS_CONNECT_BY_PATH(EMPNO,'/') 경로,
        EMPNO 사원, MGR 관리자
FROM    EMP
START   WITH     MGR IS NULL
CONNECT BY PRIOR EMPNO = MGR;
```

```
시작사원   경로       사원       관리자
--------  -------  ----      -----
A         /A       A
A         /A/B     B         A
A         /A/C     C         A
A         /A/C/D   D         C
A         /A/C/E   E         C
```

가상 칼럼	설명
CONNECT_BY_ROOT	현재 전개할 데이터의 루트(시작) 데이터를 표시한다. 사용법 : CONNECT_BY_ROOT 칼럼
SYS_CONNECT_BY_PATH	루트(시작) 데이터부터 현재 전개할 데이터까지의 경로를 표시한다. 사용법 : SYS_CONNECT_BY_PATH(칼럼, 경로분리자)

- Oracle 역방향 전개



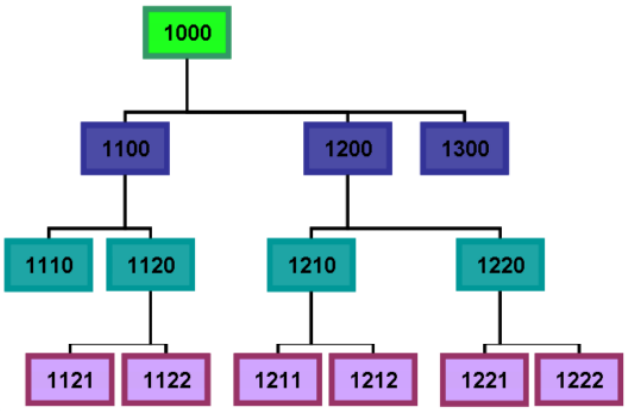
- Oracle 역방향 전개

```
SELECT  LEVEL, LPAD(' ', 4 * (LEVEL-1)) || EMPNO 사원, MGR 관리자,
        CONNECT_BY_ISLEAF ISLEAF
FROM    EMP
START   WITH   EMPNO = 'D'
CONNECT BY PRIOR MGR = EMPNO;
```

```
LEVEL      사원          관리자   ISLEAF
-----  -------------  ------  -------
1          D             C          0
2            C           A          0
3              A                      1
```

가상 칼럼	설명
LEVEL	루트 데이터이면 1 그 하위 데이터이면 2이다. 리프(Leaf) 데이터까지 1씩 증가한다.
CONNECT_BY_ISLEAF	전개 과정에서 해당 데이터가 리프 데이터이면 1 그렇지 않으면 0이다.

- SQL Server 계층형 질의
  - ✓ SQL Server 2000 버전까지는 계층형 질의를 작성할 수 있는 문법을 지원하지 않았다. SQL Server 2005 버전부터는 하나의 질의로 원하는 결과를 얻을 수 있게 되었다. (ANSI 포함됨)
  - ✓ [예제] 14명의 계층형 사원정보 구조이다. CTE(Common Table Expression)를 재귀 호출함으로써 Employees 데이터의 최상위부터 시작해 하위 방향으로 계층 구조를 전개하도록 한다.



- SQL Server 계층형 질의

```
WITH T_EMP_ANCHOR AS (
      SELECT EMPLOYEEID, MANAGERID, 0 AS LEVEL,
            CONVERT(VARCHAR(1000), EMPLOYEEID) AS SORT
      FROM   T_EMP
      WHERE  MANAGERID IS NULL /* 재귀 호출의 시작점 */
      UNION  ALL
      SELECT R.EMPLOYEEID, R.MANAGERID, A.LEVEL + 1,
            CONVERT(VARCHAR(1000), A.SORT + '/' + R.EMPLOYEEID) AS SORT
      FROM   T_EMP_ANCHOR A, T_EMP R
      WHERE  A.EMPLOYEEID = R.MANAGERID )
SELECT LEVEL, REPLICATE(' ', LEVEL) + EMPLOYEEID AS EMPLOYEEID, MANAGERID,
SORT
FROM   T_EMP_ANCHOR
ORDER  BY SORT
 GO
```

  - ✓ WITH 절의 CTE 쿼리를 보면, UNION ALL 연산자로 쿼리 두 개를 결합했다. 둘 중 위에 있는 쿼리를 '앵커 멤버' (Anchor Member)라고 하고, 아래에 있는 쿼리를 '재귀 멤버' (Recursive Member)라고 한다. (SQL Server 2005 온라인 매뉴얼 참조)
  - ✓ 앵커 멤버가 시작점이자 Outer 집합이 되어 Inner 집합인 재귀 멤버와 조인을 시작한다. 이어서, 앞서 조인한 결과가 다시 Outer 집합이 되어 재귀 멤버와 조인을 반복한다.

- SQL Server 계층형 질의

```
Level EmployeeID ManagerID Sort
----- ---------- --------- --------------------
    0 1000       NULL      1000
    1  1100      1000      1000/1100
    2   1110     1100      1000/1100/1110
    2   1120     1100      1000/1100/1120
    3    1121    1120      1000/1100/1120/1121
    3    1122    1120      1000/1100/1120/1122
    1  1200      1000      1000/1200
    2   1210     1200      1000/1200/1210
    3    1211    1210      1000/1200/1210/1211
    3    1212    1210      1000/1200/1210/1212
    2   1220     1200      1000/1200/1220
    3    1221    1220      1000/1200/1220/1221
    3    1222    1220      1000/1200/1220/1222
    1  1300      1000      1000/1300
(14개 행 적용됨)
```

✓ 실제 조직도와 비슷한 결과를 출력하도록, CTE에 Sort라는 정렬용 칼럼을 추가하고 쿼리 마지막에 order by 조건을 추가하였다. (단, 앵커 멤버와 재귀 멤버 양쪽에서 convert 함수 등으로 데이터 형식을 일치시켜야 한다)

- 셀프 조인

✓ 셀프 조인(Self Join)이란 동일 테이블 사이의 조인을 말한다. 따라서 FROM 절에 동일 테이블이 두 번 이상 나타난다.

✓ 동일 테이블 사이의 조인을 수행하면 테이블과 칼럼 이름이 모두 동일하기 때문에 식별을 위해 반드시 테이블 별칭(Alias)를 사용해야 한다. 그리고 칼럼에도 모두 테이블 별칭을 사용해서 어느 테이블의 칼럼인지 식별해줘야 한다. 이외 사항은 일반 조인과 동일하다.

✓ 셀프 조인은 동일한 테이블(EMP)이지만, 개념적으로는 두 개의 서로 다른 테이블(사원, 관리자)을 사용하는 것과 동일하다.

```
SELECT WORKER.ID  사원번호,   WORKER.NAME  사원명,
       MANAGER.ID 관리자사번, MANAGER.NAME 관리자명
FROM   EMP WORKER,  EMP MANAGER
WHERE  WORKER.MGR = MANAGER.ID;
```

- **셀프 조인**
  - ✓ [예제] 자신과 자신의 직속 관리자는 동일한 행에서 데이터를 구할 수 있으나 차상위 관리자는 바로 구할 수 없다. 차상위 관리자를 구하기 위해서는 자신의 직속 관리자를 기준으로 사원 테이블과 한번 더 조인(셀프 조인)을 수행해야 한다.

```
SELECT E1.사원, E1.관리자, E2.관리자 차상위관리자
FROM   사원 E1,    사원 E2
WHERE  E1.관리자 = E2.사원
ORDER  BY E1.사원;
```

사원	관리자	차상위관리자
B	A	
C	A	
D	C	A
E	C	A

✓결과에서 A사원(ex:사장)에 대한 정보는 누락되었다. 내부 조인(Inner Join)을 사용할 경우 자신의 관리자가 존재하지 않는 경우에는 관리자(E2) 테이블에서 조인할 대상이 존재하지 않기 때문에 해당 데이터는 결과에서 누락된다. 이를 방지하기 위해서는 아우터 조인을 사용해야 한다.

- **셀프 조인**
  - ✓ [예제] 아우터 조인을 사용해서 관리자가 존재하지 않는 데이터까지 모두 결과에 표시해본다.
  - ✓ 셀프 조인을 활용하면 똑 같은 포맷은 아니지만 계층형 질의 결과 내용을 추출할 수 있다.

```
SELECT E1.사원, E1.관리자, E2.관리자 차상위관리자
FROM   사원 E1  LEFT OUTER JOIN 사원 E2
  ON  (E1.관리자 = E2.사원)
ORDER  BY E1.사원;
```

사원	관리자	차상위관리자
A		
B	A	
C	A	
D	C	A
E	C	A

Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용
제3절 계층형 질의와 셀프 조인
핵심정리 및 연습문제

핵심정리 제3절 계층형 질의와 셀프 조인

- 테이블에 계층형 데이터가 존재하는 경우 데이터를 조회하기 위해서 계층형 질의(HIERARCHICAL QUERY)를 사용한다.
- 엔터티를 순환관계 데이터 모델로 설계할 경우 계층형 데이터가 발생한다. 순환관계 데이터 모델의 예로는 조직도, 사원(관리자포함), 메뉴 등이 있다.
- 셀프 조인(SELF JOIN)이란 동일 테이블 사이의 조인을 말하며, FROM 절에 동일 테이블이 두 번 이상 나타난다.

연습문제 제3절 계층형 질의와 셀프 조인

문제. 다음 중 SELF JOIN을 수행해야 할 때는 어떤 경우인가?

① 두 테이블에 공통 칼럼이 존재하고 두 테이블이 연관 관계가 있다.

② 두 테이블에 연관된 칼럼은 없으나 JOIN을 해야 한다.

③ 한 테이블 내에서 두 칼럼이 연관 관계가 있다.

④ 한 테이블 내에서 연관된 칼럼은 없으나 JOIN을 해야 한다.

연습문제 제3절 계층형 질의와 셀프 조인

문제. 다음 중 SELF JOIN을 수행해야 할 때는 어떤 경우인가?

① 두 테이블에 공통 칼럼이 존재하고 두 테이블이 연관 관계가 있다.

② 두 테이블에 연관된 칼럼은 없으나 JOIN을 해야 한다.

③ 한 테이블 내에서 두 칼럼이 연관 관계가 있다.

④ 한 테이블 내에서 연관된 칼럼은 없으나 JOIN을 해야 한다.

정답 : ③

해설 :

SELF JOIN은 하나의 테이블에서 두 개의 칼럼이 연관 관계를 가지고 있는 경우에 사용한다.

Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용
제4절 서브쿼리 학습하기

제4절 서브쿼리

- **서브쿼리**
  - ✓ 서브쿼리(Subquery)란 하나의 SQL문안에 포함되어 있는 또 다른 SQL문을 말한다. 즉, 서브쿼리가 메인쿼리에 포함되는 종속적인 관계이다.
  - ✓ 큰 개념의 조인에 포함시킬 수도 있으나, 보통은 조인과 구분한다.
  - ✓ 서브쿼리를 문법적으로 구분하는 가장 쉬운 방법은 SQL문이 괄호 (~~)로 묶여져 있으면 서브쿼리이다.
  - ✓ 일반적으로 서브쿼리의 위치에 따라 다음과 같이 나눌 수 있다.
    - WHERE절 : Nested Subquery (가장 먼저 개발됨)
    - FROM절 : Inline View (SQL 문장 내 절차성 효과)
    - SELECT절 : Scalar Subquery (가장 최근에 개발됨)
  - ✓ Scalar Subquery의 경우 함수적 특성을 가짐
  - ✓ 서브쿼리는 메인쿼리의 칼럼을 모두 사용할 수 있지만 메인쿼리는 서브쿼리의 칼럼을 사용할 수 없다. (예외적으로 인라인뷰에 정의된 칼럼은 메인쿼리에서 사용 가능)

- 서브쿼리와 조인
  - ✓ 조인과 서브쿼리를 논리적으로 구분하는 가장 좋은 방법은 두 개의 테이블 위치를 바꾸어 보는 것이다.
  - ✓ 조인은 두 개의 테이블 위치를 바꾸어 보더라도 같은 결과가 나오며, 서브쿼리의 경우는 주종의 관계이므로 일반적으로 다른 결과가 나오게 된다.
  - ✓ 만일 서브쿼리에서 두 개의 테이블 위치를 바꾸었는데도 같은 결과가 나온다면, 조인으로 바꿀 수 있다는 얘기가 되므로 일반적으로 서브쿼리를 집합적 개념을 사용할 수 있는 조인으로 바꾸는 것을 권고한다.
  - ✓ SQL문에서 서브쿼리 방식을 사용해야 할 때 잘못 판단하여 조인 방식을 사용하는 경우도 있다. 예를 들어, 결과는 부서(1) 레벨이고 사원(M) 테이블에서 체크해야 할 조건이 존재한다고 가정하자. 이런 상황에서 조인을 사용한다면 결과 집합은 사원(M) 레벨이 될 것이다. 이렇게 되면 원하는 결과를 만들기 위해 GROUP BY나 DISTINCT를 추가해서 결과를 다시 조직(1) 레벨로 만들어야 한다. 이와 같은 상황에서는 조인 방식이 아니라 서브쿼리 방식을 사용해야 한다. 조인과는 달리 서브쿼리를 사용해야 되는 경우도 많으므로, 조인과 서브쿼리의 용도를 정확히 알고 작성해야 한다.

- 서브쿼리 종류
  - ✓ 서브쿼리의 종류는 반환되는 데이터의 형태에 따라 다음과 같이 분류할 수 있다.

서브쿼리 종류	설명
Single Row (단일행) 서브쿼리	서브쿼리의 실행 결과가 항상 1건 이하인 서브쿼리를 의미한다. 단일행 서브쿼리는 단일행 비교 연산자와 함께 사용된다. 단일행 비교 연산자에는 =, <, <=, >, >=, <>이 있다.
Multi Row (다중행) 서브쿼리	서브쿼리의 실행 결과가 여러 건인 서브쿼리를 의미한다. 다중행 서브쿼리는 다중행 비교 연산자와 함께 사용된다. 다중행 비교 연산자에는 IN, ALL, ANY, SOME, EXISTS가 있다.
Multi Column (다중칼럼) 서브쿼리	서브쿼리의 실행 결과로 여러 칼럼을 반환한다. 메인쿼리의 조건절에 여러 칼럼을 동시에 비교할 수 있다. 서브쿼리와 메인쿼리에서 비교하고자 하는 칼럼 개수와 칼럼의 위치가 동일해야 한다.

- 단일행 서브쿼리
  - ✓ 서브쿼리가 단일행 비교 연산자(=, <, <=, >, >=, <>)와 함께 사용할 때는 서브쿼리의 결과 건수가 반드시 1건 이하이어야 한다. 만약, 서브쿼리의 결과 건수가 2건 이상을 반환하면 SQL문은 실행시간(Run Time) 오류가 발생한다. 이런 종류의 오류는 컴파일 할 때(Compile Time)는 알 수 없는 오류이다.

```
SELECT PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버
FROM   PLAYER_T
WHERE  TEAM_ID = (SELECT TEAM_ID
                  FROM PLAYER_T
                  WHERE PLAYER_NAME = '정남일')
ORDER  BY PLAYER_NAME;
```

  - ✓ 김남일 선수가 동명이인이 없다면 정상적인 수행이 되지만, 동명이인이 다른 팀에 있다면 두개 이상의 TEAM_ID가 반환되므로 런타임 오류가 발생한다.
  - ✓ 서브쿼리의 결과가 2건 이상 반환될 수 있다면 반드시 다중행 비교 연산자(IN, ALL, ANY, SOME)와 함께 사용해야 한다.

- 다중행 비교연산자

다중행 연산자	설명
IN (서브쿼리)	서브쿼리의 결과에 존재하는 임의의 값과 동일한 조건을 의미한다. (Multiple OR 조건)
비교연산자 ALL (서브쿼리)	서브쿼리의 결과에 존재하는 모든 값을 만족하는 조건을 의미한다. 비교 연산자로 ">"를 사용했다면 메인쿼리는 서브쿼리의 모든 결과 값을 만족해야 하므로, 서브쿼리 결과의 최대값보다 큰 모든 건이 조건을 만족한다.
비교연산자 ANY (서브쿼리), SOME	서브쿼리의 결과에 존재하는 어느 하나의 값이라도 만족하는 조건을 의미한다. 비교 연산자로 ">"를 사용했다면 메인쿼리는 서브쿼리의 값들 중 어떤 값이라도 만족하면 되므로, 서브쿼리의 결과의 최소값보다 큰 모든 건이 조건을 만족한다. (SOME은 ANY와 동일함)
EXISTS (서브쿼리)	서브쿼리의 결과를 만족하는 값이 존재하는지 여부를 확인하는 조건을 의미한다. 조건을 만족하는 건이 여러 건이더라도 1건만 찾으면 더 이상 검색하지 않는다.

- 다중행 서브쿼리
  - ✓ [예제] 선수들 중에서 '정현수' 라는 선수가 소속되어 있는 팀 정보를 출력하는 서브쿼리를 작성하면 다음과 같다.

```
SELECT REGION_NAME 연고지명, TEAM_NAME 팀명, E_TEAM_NAME 영문팀명
FROM   TEAM
WHERE  TEAM_ID = (SELECT TEAM_ID
                  FROM PLAYER
                  WHERE PLAYER_NAME = '정현수')
ORDER  BY TEAM_NAME;
```

ORA-01427: 단일행 하위 질의에 2개 이상의 행이 리턴되었습니다.

- 다중행 서브쿼리
  - ✓ [예제] 선수들 중에서 '정현수' 라는 선수가 소속되어 있는 팀 정보를 출력하는 서브쿼리를 다중행 비교 연산자 IN으로 바꾸어서 SQL문을 작성하면 다음과 같다. (동명이인을 확인할 수 있음)

```
SELECT REGION_NAME 연고지명, TEAM_NAME 팀명, E_TEAM_NAME 영문팀명
FROM   TEAM
WHERE  TEAM_ID IN (SELECT TEAM_ID
                   FROM PLAYER
                   WHERE PLAYER_NAME = '정현수')
ORDER  BY TEAM_NAME;
```

```
연고지명 팀명                     영문팀명
-------- ------------------------ ----------------------------------
전남     드래곤즈                 CHUNNAM DRAGONS FC
성남     일화천마                 SEONGNAM ILHWA CHUNMA FC

2개의 행이 선택되었다.
```

- 다중칼럼 서브쿼리
  - ✓ 다중 칼럼 서브쿼리는 서브쿼리의 결과로 여러 개의 칼럼이 반환되어 메인쿼리의 조건과 동시에 비교되는 것을 의미한다.
  - ✓ [예제] 소속팀별 키가 가장 작은 사람들의 정보를 해본다.

```
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션,
       BACK_NO 백넘버, HEIGHT 키
FROM   PLAYER
WHERE (TEAM_ID, HEIGHT) IN (SELECT TEAM_ID, MIN(HEIGHT)
                            FROM PLAYER
                            GROUP BY TEAM_ID)
ORDER BY TEAM_ID, PLAYER_NAME;
```

```
팀코드 선수명              포지션  백넘버    키
------ ------------------- ------- ------    ---
K01    마르코스             FW        44    170
K01    박정수               MF        8     170
...
```


- 서브쿼리 종류
  - ✓ 서브쿼리의 종류는 동작하는 방식에 따라 다음과 같이 분류할 수 있다. 앞에서 설명한 네스티드 서브쿼리들은 비연관 서브쿼리의 예제들이었다.

서브쿼리 종류	설명
Un-Correlated (비연관) 서브쿼리	서브쿼리가 메인쿼리 칼럼을 가지고 있지 않는 형태의 서브쿼리이다. 메인쿼리에 값(서브쿼리가 실행된 결과)을 제공하기 위한 목적으로 주로 사용한다.
Correlated (연관) 서브쿼리	서브쿼리가 메인쿼리의 칼럼을 사용하는 형태의 서브쿼리이다. 일반적으로 메인쿼리가 먼저 수행되어 읽혀진 데이터를 서브쿼리에서 조건이 맞는지 확인하고자 할 때 주로 사용된다.

- 연관 서브쿼리
  - ✓ 연관 서브쿼리(Correlated Subquery)는 서브쿼리 내에 메인쿼리 칼럼이 사용된 서브쿼리이다.
  - ✓ [예제] 가비 선수는 삼성블루윙즈팀 소속이므로 삼성블루윙즈팀 소속의 평균키를 구하고 그 평균키와 가비 선수의 키를 비교하여 적을 경우에 선수에 대한 정보를 출력한다. 만약, 평균키 보다 선수의 키가 크거나 같으면 조건에 맞지 않기 때문에 해당 데이터는 출력되지 않는다.
  - ✓ 이와 같은 작업을 메인쿼리에 존재하는 모든 행에 대해서 반복 수행한다.

```
SELECT T.TEAM_NAME 팀명, M.PLAYER_NAME 선수명, M.POSITION 포지션,
       M.BACK_NO 백넘버, M.HEIGHT 키
FROM   PLAYER M,   TEAM T
WHERE  M.TEAM_ID = T.TEAM_ID
AND    M.HEIGHT  < (SELECT AVG(S.HEIGHT)
                    FROM PLAYER S
                    WHERE S.TEAM_ID = M.TEAM_ID
                      AND S.HEIGHT IS NOT NULL
                    GROUP BY S.TEAM_ID)
ORDER BY 선수명;
```


- EXISTS 서브쿼리
  - ✓ EXISTS 서브쿼리는 항상 연관 서브쿼리로 사용된다.
  - ✓ EXISTS 서브쿼리의 특징은 아무리 조건을 만족하는 건이 여러 건이더라도 조건을 만족하는 1건만 찾으면 추가적인 검색을 진행하지 않는다.
  - ✓ 현업에서 조건을 만족하는지 여부를 묻는 로직이 많이 사용되는데, 성능에 부담이 적은 EXISTS절을 우선적으로 검토할 필요가 높다.
  - ✓ EXISTS 연산자의 왼쪽에는 칼럼명이나 상수가 표시되지 않는다.
  - ✓ 서브쿼리의 데이터가 필요한 경우가 아니라면 1,'X' 같은 업무적으로 의미없는 상수값을 선택하는 것이 좋다.
  - ✓ [예제] EXISTS 서브쿼리를 사용하여 '20120501' 부터 '20120521' 사이에 경기가 있는 경기장을 조회한다.

```
SELECT STADIUM_ID, STADIUM_NAME
FROM   STADIUM
WHERE  EXISTS (SELECT 1
               FROM SCHEDULE
               WHERE SCHE_DATE BETWEEN '20120501' AND '20120521')
```

- 연관 서브쿼리 업데이트
  - ✓ [예제] TEAM 테이블에 새롭게 추가된 STADIUM_NAME의 값을 STADIUM 테이블을 이용하여 일괄 변경하고자 할 때 다음과 같이 SQL문을 작성할 수 있다. 복잡한 프로그램을 하나의 SQL문장으로 대체할 수 있다.

```
UPDATE TEAM A
SET    A.STADIUM_NAME = (SELECT X.STADIUM_NAME
                         FROM STADIUM X
                        WHERE X.STADIUM_ID = A.STADIUM_ID);
```

  - ✓ 그러나, STADIUM_NAME 칼럼에 기존 데이터가 있는 경우 위 연관 서브쿼리를 사용은 주의해야 한다. 변경 작업 수행시 연관 서브쿼리 조인 조건이 맞지 않는 경우 서브쿼리의 공집합 결과가 STADIUM_NAME 칼럼에 NULL 값으로 업데이트 될 수 있는 리스크가 있다.
  - ✓ NULL 업데이트 리스크를 해결하기 위해서는 EXISTS 조건을 통해 안정적으로 조인 조건만 맞는 데이터를 업데이트 하거나, 공집합을 NULL로 리턴하는 그룹함수 특성을 사용하는 방법을 적용한다. (교재 외 내용임)

- 연관 서브쿼리 업데이트 (교재 외)
  - ✓ [예제] EXISTS 조건을 추가하여 조인 조건만 맞는 데이터만 변경한다.
    (ex: 100건 중 40건만 수정) ※ 권고 방법

```
UPDATE TEAM A
SET    A.STADIUM_NAME = (SELECT X.STADIUM_NAME
                         FROM STADIUM X
                        WHERE X.STADIUM_ID = A.STADIUM_ID)
WHERE  EXISTS           (SELECT 1
                         FROM STADIUM X
                        WHERE X.STADIUM_ID = A.STADIUM_ID);
```

  - ✓ [예제] 공집합을 NULL로 리턴하는 그룹함수 특성을 사용한다.
  - ✓ (ex: 100건 중 90건은 변경, 10건은 원래 값으로 수정)

```
UPDATE TEAM A
SET    A.STADIUM_NAME =
          (SELECT NVL(MAX(X.STADIUM_NAME), A.STADIUM_NAME)
             FROM STADIUM X
            WHERE X.STADIUM_ID = A.STADIUM_ID);
```

- INLINE VIEW
  - ✓ FROM절에서 사용되는 서브쿼리를 인라인 뷰(Inline View)라고 한다. FROM절에는 테이블 명이 오도록 되어 있다. FROM절에 사용된 서브쿼리의 결과가 마치 실행 시에 동적으로 생성된 테이블인 것처럼 사용할 수 있다.
  - ✓ 인라인 뷰는 SQL문이 실행될 때만 임시적으로 생성되는 동적인 뷰이기 때문에 데이터베이스에 해당 정보가 저장되지 않는다. 그래서 일반적인 뷰를 정적 뷰(Static View)라고 하고 인라인 뷰를 동적 뷰(Dynamic View)라고도 한다. 인라인 뷰는 테이블 명이 올 수 있는 곳에서 사용할 수 있다.
  - ✓ 중첩되어 사용할 수 있으며, 일반적으로 가장 안의 인라인 뷰부터 수행이 된다. (SQL 문장 내 절차적 프로그래밍 효과 가짐, 특별한 경우 뷰머지가 발생할수도 있음)

- INLINE VIEW
  - ✓ [예제] 인라인 뷰에 정의된 SELECT 칼럼을 메인쿼리에서 사용할 수 있다.

[예제] 및 실행 결과

```
SELECT EMPNO  FROM
(SELECT EMPNO, ENAME FROM EMP ORDER BY MGR);
14개의 행이 선택되었다.
```

  - ✓ [예제] 인라인 뷰에 미정의된 칼럼은 메인쿼리에서 사용할 수 없다.

[예제] 및 실행 결과

```
SELECT MGR FROM
(SELECT EMPNO, ENAME FROM EMP ORDER BY MGR);

SELECT MGR FROM ; *
ERROR: "MGR": 부적합한 식별자
```

- INLINE VIEW
  - ✓ 일반적으로 서브쿼리의 칼럼은 메인쿼리에서 사용할 수 없다. (반대는 상관 서브쿼리에서 가능함) 그러나, 인라인 뷰는 동적으로 생성된 테이블이기 때문에 인라인 뷰를 사용하는 것은 조인 방식을 사용하는 것과 같다. 결과적으로 인라인 뷰에서 정의된 칼럼은 SQL문에서 자유롭게 참조할 수 있다.
  - ✓ [예제] K-리그 선수들 중에서 포지션이 미드필더(MF) 선수들의 소속팀명 및 선수 정보를 출력하고자 한다. 인라인 뷰를 활용해서 SQL문을 만들어 보자.

```
SELECT T.TEAM_NAME 팀명, P.PLAYER_NAME 선수명, P.BACK_NO 백넘버
FROM  (SELECT TEAM_ID, PLAYER_NAME, BACK_NO
         FROM PLAYER
        WHERE POSITION = 'MF') P, TEAM T
WHERE  P.TEAM_ID = T.TEAM_ID
ORDER  BY 선수명;
```


- INLINE VIEW
  - ✓ 인라인 뷰의 중요 용도 중에 하나는 집합 간의 조인 회수를 줄이는 것이다. 조인 집합 간의 관계가 1:1이 아닌 1:M의 관계일 때 M쪽의 데이터를 인라인 뷰에서 먼저 GROUP BY 연산을 수행하여 통하여 메인쿼리의 집합과 1:1 조인을 수행할 수 있다.
  - ✓ [예제] 부서 기준으로 사원들의 평균급여와 최대급여를 구하고, 부서명 기준으로 정렬하라.

```
SELECT D.DNAME 부서명, E.AVG_SAL 평균급여, E.MAX_SAL 최대급여
FROM  (SELECT DEPTNO, AVG(SAL) AVG_SAL, MAX(SAL) MAX_SAL
         FROM EMP
        GROUP BY DEPTNO) E, DEPT D
WHERE  E.DEPTNO = D.DEPTNO
ORDER  BY 부서명;
```

- VIEW
  - ✓ 테이블을 실제로 데이터를 가지고 있는 반면, 뷰(View)는 실제 데이터를 가지고 있지 않다. 뷰는 단지 뷰 정의(View Definition, SQL 텍스트 파일)만을 가지고 있다. 질의에서 뷰가 사용되면 뷰 정의를 참조해서 DBMS 내부적으로 질의를 재작성(Rewrite)하여 질의를 수행한다.
  - ✓ Oracle의 경우 USER_VIEWS에서 해당 뷰에 사용된 SQL을 확인할 수 있다.
  - ✓ 뷰는 실제 데이터를 가지고 있지 않지만 테이블이 수행하는 역할을 수행하기 때문에 가상 테이블(Virtual Table)이라고도 한다.
  - ✓ 뷰는 다음과 같이 CREATE VIEW 문을 통해서 생성할 수 있다.

```
CREATE VIEW V_PLAYER_TEAM AS
SELECT P.PLAYER_NAME, P.POSITION, P.BACK_NO, P.TEAM_ID, T.TEAM_NAME
FROM   PLAYER P,   TEAM T
WHERE  P.TEAM_ID = T.TEAM_ID;
```


- VIEW의 장점

뷰의 장점	설명
독립성	테이블 구조가 변경되어 뷰를 사용하는 응용 프로그램은 변경하지 않아도 된다.
편리성	복잡한 질의를 뷰로 생성함으로써 관련 질의를 단순하게 작성할 수 있다. 또한 해당 형태의 SQL문을 자주 사용할 때 뷰를 이용하면 편리하게 사용할 수 있다.
보안성	직원의 급여정보와 같이 숨기고 싶은 정보가 존재한다면, 뷰 생성할 때 해당 칼럼을 빼고 생성함으로써 사용자에게 정보를 감출 수 있다.

- VIEW 사용
  - ✓ 뷰를 사용하기 위해서는 해당 뷰의 이름을 테이블과 똑같이 이용하면 된다.

```
SELECT PLAYER_NAME, POSITION, BACK_NO, TEAM_ID, TEAM_NAME
FROM   V_PLAYER_TEAM
WHERE  PLAYER_NAME LIKE '황%';
```

  - ✓ 뷰가 포함된 SQL을 파싱하는 시점에 DBMS가 내부적으로 SQL문을 다음과 같이 재작성한다. (VIEW Merge)

```
SELECT PLAYER_NAME, POSITION, BACK_NO, TEAM_ID, TEAM_NAME
FROM   (SELECT P.PLAYER_NAME, P.POSITION, P.BACK_NO, P.TEAM_ID,
               T.TEAM_NAME
        FROM   PLAYER P,   TEAM T
        WHERE  P.TEAM_ID = T.TEAM_ID)
WHERE PLAYER_NAME LIKE '황%';
```


- VIEW 생성 및 삭제
  - ✓ 뷰는 테이블뿐만 아니라 이미 존재하는 뷰를 참조해서도 생성할 수 있다.

```
CREATE VIEW V_PLAYER_TEAM_FILTER AS
SELECT PLAYER_NAME, POSITION, BACK_NO, TEAM_NAME
FROM   V_PLAYER_TEAM
WHERE  POSITION IN ('GK', 'MF');
```

  - ✓ 뷰를 제거하기 위해서는 DROP VIEW문을 사용한다.

```
DROP VIEW  V_PLAYER_TEAM;

DROP VIEW  V_PLAYER_TEAM_FILTER;
```

- 스칼라 서브쿼리
  - ✓ 스칼라 서브쿼리는 한 행, 한 칼럼(1Row, 1Column)만을 반환하는 서브쿼리를 말한다. 스칼라 서브쿼리는 칼럼을 쓸 수 있는 대부분의 곳에서 사용할 수 있다.
  - ✓ 스칼라 서브쿼리도 일종의 함수이므로 중첩해서 사용하고, OUTPUT이 두개 이상 나오는 경우나 OUTPUT의 데이터타입이 맞지 않는 경우 SYNTAX 에러가 발생합니다.
  - ✓ 스칼라 서브쿼리의 경우 FUNCTION(함수)의 특징을 가집니다. 함수는 기능적으로는 많은 INPUT이 있더라도 OUTPUT은 일반적으로 하나만 나온다는 것과 성능면에서는 전체 데이터를 일일이 모든 건수만큼 수행해야 한다는 점입니다.
  - ✓ 대량의 데이터 처리시 스칼라 서브쿼리를 남발하는 경우는 조인의 장점이나 집합적 개념을 적용하기 힘들므로, 같은 결과를 얻을 수 있다면 가능하면 스칼라 서브쿼리가 아니라 조인으로 대체하는 것이 좋습니다.

- 스칼라 서브쿼리 위치
  - ✓ Select List 항목

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

```
SELECT EMPNO, ENAME,
       (SELECT GRADE FROM SALGRADE WHERE EMP.SAL BETWEEN LOSAL AND HISAL)
        as GRADE
FROM   EMP;
```

– 스칼라 서브쿼리에서 출력된 값은 Select List의 한 항목 값으로 대체되고, 자료형이 서브쿼리에서 출력된 칼럼의 자료형과 일치하지 않으면, DBMS는 에러를 출력하게 됩니다.
– 만약 조회된 결과가 없다면, 즉 공집합이더라도 Select List의 값은 집계함수와 같이 NULL로 표시됩니다. 두 ROW 이상의 결과가 출력되면 DBMS는 에러를 출력하게 됩니다.

- 스칼라 서브쿼리 위치
  - ✓ 함수의 인자

```
SELECT EMPNO, ENAME,
       SUBSTR((SELECT DNAME  FROM DEPT  WHERE DEPTNO = E.DEPTNO), 1, 3)
       as DEPARTMENT_NAME
  FROM EMP E;
```

  - ✓ WHERE 절의 조건

```
SELECT EMPNO, ENAME
  FROM EMP E
 WHERE (SELECT DNAME  FROM DEPT    WHERE DEPTNO = E.DEPTNO)
     = (SELECT DNAME  FROM DEPT_2  WHERE DEPTNO = E.DEPTNO );
```

  - ✓ ORDER BY 절

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


- 스칼라 서브쿼리 위치
  - ✓ CASE 조건 절

```
SELECT EMPNO, ENAME, DEPTNO,
      (CASE WHEN  DEPTNO IN (SELECT DEPTNO  FROM DEPT)
            THEN 'USA'
            ELSE 'OTHER COUNTRY'
       END) LOCATION
  FROM EMP E;
```

  - ✓ CASE 결과 절

```
SELECT EMPNO, ENAME, DEPTNO,
      (CASE WHEN DEPTNO = 20
            THEN (SELECT DNAME  FROM DEPT  WHERE DEPTNO = 10)
            ELSE (SELECT DNAME  FROM DEPT  WHERE DEPTNO = 30)
       END) as NEW_DEPARTMENT
  FROM EMP E;
```

- 스칼라 서브쿼리 위치
  - ✓ HAVING 절

    [예제] 삼성 블루윙즈팀의 평균키보다 작은 팀의 이름과 해당 팀의 평균키를 구하는 SQL문을 작성하면 다음과 같다.

```
SELECT P.TEAM_ID 팀코드, T.TEAM_NAME 팀명, AVG(P.HEIGHT) 평균키
FROM   PLAYER P,     TEAM T
WHERE  P.TEAM_ID   = T.TEAM_ID
GROUP  BY P.TEAM_ID, T.TEAM_NAME
HAVING AVG(P.HEIGHT) < (SELECT AVG(HEIGHT)
                          FROM PLAYER
                         WHERE TEAM_ID ='K02')
```


Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용
제4절 서브쿼리
핵심정리 및 연습문제

핵심정리 제4절 서브쿼리

- 서브쿼리란 하나의 메인쿼리안에 포함되어 있는 종속적인 SQL 문장을 말하는 것으로 괄호( )로 묶어져 있다.
- 위치나 기능에 따라
  - ✓ WHERE 절 : NESTED SUBQUERY
  - ✓ FROM 절 : INLINE VIEW
  - ✓ SELECT절 외 : SCALAR SUBQUERY로 구분할 수 있다.
- 인라인뷰는 하나의 SQL 문장에 절차성을 줄 수 있다.
- 스칼라 서브쿼리는 함수의 성격을 가지고 있다.

연습문제 제4절 서브쿼리

문제. 일반적으로 FROM 절에 정의된 후 먼저 수행되어 SQL 문장 내에서 절차성을 주는 효과를 볼 수 있는 것은 어떤 유형의 서브쿼리 문장인가?

① SCALASR SUBQUERY
② NESTED SUBQUERY
③ CORRELATED SUBQUERY
④ INLINE VIEW

연습문제 제4절 서브쿼리

문제. 일반적으로 FROM 절에 정의된 후 먼저 수행되어 SQL 문장 내에서 절차성을 주는 효과를 볼 수 있는 것은 어떤 유형의 서브쿼리 문장인가?

① SCALASR SUBQUERY
② NESTED SUBQUERY
③ CORRELATED SUBQUERY
④ INLINE VIEW

정답 : ④
해설 :
FROM 절에 정의된 서브쿼리는 INLINE VIEW이다. INLINE VIEW는 일반적으로 메인쿼리보다 먼저 수행되므로 SQL 문장 내에서 절차성을 주는 효과를 얻을 수 있다.

연습문제 제4절 서브쿼리

문제. 다음 서브쿼리에 대한 설명 중 틀린 것을 고르시오.

① 다중행 연산자는 IN, ANY, ALL이 있으며, 서브쿼리의 결과로 하나 이상의 데이터가 RETURN되는 서브쿼리이다.
② TOP-N 서브쿼리는 INLINE VIEW의 정렬된 데이터를 ROWNUM을 이용해 결과 행수를 제한하거나, TOP (N) 조건을 사용하는 서브쿼리이다.
③ INLINE VIEW는 FROM 절에 사용되는 서브쿼리로서 실질적인 OBJECT는 아니지만 SQL 문장에서 마치 VIEW나 테이블처럼 사용되는 서브쿼리다.
④ 상호연관 서브쿼리는 처리 속도가 가장 빠르기 때문에 최대한 활용하는 것이 좋다.

문제. 다음 서브쿼리에 대한 설명 중 틀린 것을 고르시오.

① 다중행 연산자는 IN, ANY, ALL이 있으며, 서브쿼리의 결과로 하나 이상의 데이터가 RETURN되는 서브쿼리이다.

② TOP-N 서브쿼리는 INLINE VIEW의 정렬된 데이터를 ROWNUM을 이용해 결과 행수를 제한하거나, TOP (N) 조건을 사용하는 서브쿼리이다.

③ INLINE VIEW는 FROM 절에 사용되는 서브쿼리로서 실질적인 OBJECT는 아니지만 SQL 문장에서 마치 VIEW나 테이블처럼 사용되는 서브쿼리다.

④ 상호연관 서브쿼리는 처리 속도가 가장 빠르기 때문에 최대한 활용하는 것이 좋다.

정답 : ④

해설 :

상호 연관 서브쿼리는 서브쿼리가 메인쿼리의 행 수 만큼 실행되는 쿼리로서 실행 속도가 상대적으로 떨어지는 SQL 문장이다. 그러나, 복잡한 일반 배치 프로그램을 대체할 수 있기 때문에 조건에 맞는다면 적극적인 검토가 필요하다.

Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용
제5절 GROUP FUNCTION 학습하기

제5절 GROUP FUNCTION

- 데이터 분석 함수 (ANSI/ISO-SQL표준)
  - ■ AGGREGATE FUNCTIONS
    - ✓ GROUP FUNCTION의 한 부분이며, GROUP AGGREGATE FUNCTION 이라고도 부를 수 있다. 1장 7절에서 배웠던 COUNT, SUM, AVG, MAX, MIN 외 각종 집계 함수들이 포함되어 있다.
  - ■ GROUP FUNCTIONS
    - – 그룹 함수로는 집계 함수를 제외하면,
    - ✓ 소그룹 간의 소계를 계산하는 ROLLUP 함수,
    - ✓ GROUP BY 항목들간의 다차원적인 소계를 계산할 수 있는 CUBE 함수,
    - ✓ 특정 항목에 대한 소계를 계산하는 GROUPING SETS 함수가 있다.
  - ■ WINDOW FUNCTIONS
    - ✓ 분석 함수(ANALYTIC FUNCTION)나 순위 함수(RANK FUNCTION)로도 알려져 있는 윈도우 함수는 데이터웨어하우스에서 발전한 기능이며, 자세한 내용은 다음 6절에서 설명한다.

제5절 GROUP FUNCTION

- GROUP FUNCTIONS
  - ✓ ROLLUP은 GROUP BY의 확장된 형태로 사용하기가 쉬우며 병렬로 수행이 가능하기 때문에 매우 효과적일 뿐 아니라 시간 및 지역처럼 계층적 분류를 포함하고 있는 데이터의 집계에 적합하도록 되어 있다.
  - ✓ CUBE는 결합 가능한 모든 값에 대하여 다차원적인 집계를 생성하게 되므로 ROLLUP에 비해 다양한 데이터를 얻는 장점이 있는 반면에, 시스템에 부하를 많이 주는 단점이 있다.
  - ✓ GROUPING SETS는 원하는 부분의 소계만 손쉽게 추출할 수 있는 장점이 있다.
  - ✓ ROLLUP, CUBE, GROUPING SETS 사용시 정렬이 필요한 경우는 ORDER BY 절에 정렬 칼럼을 명시해야 한다.

- ROLLUP 함수
  - ✓ ROLLUP에 지정된 Grouping Columns의 List는 Subtotal을 생성하기 위해 사용되어지며, Grouping Columns의 수를 N이라고 했을 때 N+1 Level의 Subtotal이 생성된다.
  - ✓ ROLLUP의 인수는 계층 구조이므로 인수 순서가 바뀌면 수행 결과도 바뀌게 되므로 인수의 순서에도 주의해야 한다.
  - ✓ Oracle을 포함한 일부 DBMS의 과거 버전에서는 GROUP BY 절 사용시 자동적으로 정렬을 수행하였으나, 현재 대부분의 DBMS 버전은 집계 기능만 지원하고 있으며 정렬이 필요한 경우는 ORDER BY 절에 명시적으로 정렬 칼럼이 표시 되어야 한다.

- STEP1. 일반적인 GROUP BY 절 사용
  - ✓ [예제] 부서명과 업무명을 기준으로 사원 수와 급여 합을 집계한 일반적인 GROUP BY SQL 문장을 수행한다.

[예제]

```
SELECT DNAME,     JOB,
       COUNT(*)  "Total Empl",
       SUM(SAL)  "Total Sal"
FROM   EMP, DEPT
WHERE  DEPT.DEPTNO = EMP.DEPTNO
GROUP  BY   DNAME, JOB;
```

실행 결과

DNAME	JOB	Total Empl	Total Sal
SALES	MANAGER	1	2850
SALES	CLERK	1	950
ACCOUNTING	MANAGER	1	2450
RESEARCH	ANALYST	2	6000
ACCOUNTING	CLERK	1	1300
SALES	SALESMAN	4	5600
RESEARCH	MANAGER	1	2975
ACCOUNTING	PRESIDENT	1	5000
RESEARCH	CLERK	2	1900

9개의 행이 선택되었다.

- STEP1－2. GROUP BY 절 + ORDER BY 절 사용
  - ✓ [예제] 부서명과 업무명을 기준으로 집계한 일반적인 GROUP BY SQL 문장에 ORDER BY 절을 사용함으로써 부서, 업무별로 정렬이 이루어진다.

[예제]

```
SELECT DNAME,      JOB,
       COUNT(*)   "Total Empl",
       SUM(SAL)   "Total Sal"
FROM   EMP, DEPT
WHERE  DEPT.DEPTNO = EMP.DEPTNO
GROUP  BY   DNAME, JOB
ORDER  BY   DNAME, JOB;
```

실행 결과

DNAME	JOB	Total Empl	Total Sal
ACCOUNTING	CLERK	1	1300
ACCOUNTING	MANAGER	1	2450
ACCOUNTING	PRESIDENT	1	5000
RESEARCH	ANALYST	2	6000
RESEARCH	CLERK	2	1900
RESEARCH	MANAGER	1	2975
SALES	CLERK	1	950
SALES	MANAGER	1	2850
SALES	SALESMAN	4	5600

9개의 행이 선택되었다.

- STEP2. ROLLUP 함수 사용
  - ✓ [예제] 부서명과 업무명을 기준으로 집계한 일반적인 GROUP BY SQL 문장에 ROLLUP 함수를 사용한다.

[예제]

```
SELECT DNAME,      JOB,
       COUNT(*)   "Total Empl",
       SUM(SAL)   "Total Sal"
FROM   EMP, DEPT
WHERE  DEPT.DEPTNO = EMP.DEPTNO
GROUP  BY   ROLLUP  (DNAME, JOB);
```

※ ROLLUP의 경우 계층간 집계에 대해서는 LEVEL 별 순서(L1→L2→L3)를 정렬하지만, 계층내 GROUP BY 수행시 생성되는 표준 집계에는 별도의 정렬을 지원하지 않는다. L1, L2, L3 계층내 정렬을 위해서는 별도의 ORDER BY 절을 사용해야 한다.

DNAME	JOB	Total Empl	Total Sal
SALES	CLERK	1	950
SALES	MANAGER	1	2850
SALES	SALESMAN	4	5600
SALES		6	9400
RESEARCH	CLERK	2	1900
RESEARCH	ANALYST	2	6000
RESEARCH	MANAGER	1	2975
RESEARCH		5	10875
ACCOUNTING	CLERK	1	1300
ACCOUNTING	MANAGER	1	2450
ACCOUNTING	PRESIDENT	1	5000
ACCOUNTING		3	8750
		14	29025

13개의 행이 선택되었다.

- STEP2-2. ROLLUP 함수 + ORDER BY 절 사용
  - ✓ [예제] 부서명과 업무명 기준으로 집계한 일반적인 GROUP BY SQL 문장에 ROLLUP 함수를 사용한다. 추가로 ORDER BY 절을 사용해서 부서, 업무별로 정렬한다.

[예제]

```
SELECT DNAME,      JOB,
       COUNT(*)   "Total Empl",
       SUM(SAL)   "Total Sal"
FROM   EMP, DEPT
WHERE  DEPT.DEPTNO = EMP.DEPTNO
GROUP  BY   ROLLUP  (DNAME, JOB)
ORDER  BY   DNAME, JOB ;
```

DNAME	JOB	Total Empl	Total Sal
ACCOUNTING	CLERK	1	1300
ACCOUNTING	MANAGER	1	2450
ACCOUNTING	PRESIDENT	1	5000
ACCOUNTING		3	8750
RESEARCH	CLERK	2	1900
RESEARCH	ANALYST	2	6000
RESEARCH	MANAGER	1	2975
RESEARCH		5	10875
SALES	CLERK	1	950
SALES	MANAGER	1	2850
SALES	SALESMAN	4	5600
SALES		6	9400
		14	29025

13개의 행이 선택되었다.

- STEP3. GROUPING 함수 사용
  - ✓ [예제] ROLLUP이나 CUBE에 의한 소계가 계산된 결과에는 GROUPING(EXPR) = 1 이 표시되고, 그 외의 결과에는 GROUPING(EXPR) = 0 이 표시된다.

[예제]

```
SELECT   DNAME,          GROUPING(DNAME),
         JOB,            GROUPING(JOB),
         COUNT(*)       "Total Empl",
         SUM(SAL)       "Total Sal"
FROM     EMP, DEPT
WHERE    DEPT.DEPTNO = EMP.DEPTNO
GROUP    BY   ROLLUP  (DNAME, JOB);
```

DNAME	GROUPING (DNAME)	JOB	GROUPING (JOB)	Total Empl	Total Sal
SALES	0	CLERK	0	1	950
SALES	0	MANAGER	0	1	2850
SALES	0	SALESMAN	0	4	5600
SALES	0		1	6	9400
RESEARCH	0	CLERK	0	2	1900
RESEARCH	0	ANALYST	0	2	6000
RESEARCH	0	MANAGER	0	1	2975
RESEARCH	0		1	5	10875
ACCOUNTING	0	CLERK	0	1	1300
ACCOUNTING	0	MANAGER	0	1	2450
ACCOUNTING	0	PRESIDENT	0	1	5000
ACCOUNTING	0		1	3	8750
	1		1	14	29025

13개의 행이 선택되었다.

- STEP4. GROUPING 함수 + CASE 사용
  - ✓ [예제] ROLLUP 함수를 추가한 집계 보고서에서 집계 레코드를 구분할 수 있는 GROUPING 함수와 CASE 함수를 함께 사용한 SQL 문장을 작성한다.

[예제]

```
SELECT
  CASE  GROUPING(DNAME)
    WHEN 1      THEN 'All Departments'
    ELSE DNAME END    AS DNAME,
  CASE  GROUPING(JOB)
    WHEN 1      THEN 'All Jobs'
    ELSE JOB    END    AS JOB,
  COUNT(*)  "Total Empl",
  SUM(SAL)  "Total Sal"
FROM    EMP, DEPT
WHERE   DEPT.DEPTNO = EMP.DEPTNO
GROUP   BY    ROLLUP  (DNAME, JOB);
```

DNAME	JOB	Total Empl	Total Sal
SALES	CLERK	1	950
SALES	MANAGER	1	2850
SALES	SALESMAN	4	5600
SALES	All Jobs	6	9400
RESEARCH	CLERK	2	1900
RESEARCH	ANALYST	2	6000
RESEARCH	MANAGER	1	2975
RESEARCH	All Jobs	5	10875
ACCOUNTING	CLERK	1	1300
ACCOUNTING	MANAGER	1	2450
ACCOUNTING	PRESIDENT	1	5000
ACCOUNTING	All Jobs	3	8750
All Departments	All Jobs	14	29025

13개의 행이 선택되었다.

- STEP4-2. ROLLUP 함수 일부 사용
  - ✓ [예제] GROUP BY ROLLUP (DNAME, JOB) 조건에서 GROUP BY DNAME, ROLLUP(JOB) 조건으로 변경한 경우이다.

[예제]

```
SELECT
  CASE  GROUPING(DNAME)
    WHEN 1      THEN 'All Departments'
    ELSE DNAME END    AS DNAME,
  CASE  GROUPING(JOB)
    WHEN 1      THEN 'All Jobs'
    ELSE JOB    END    AS JOB,
  COUNT(*)  "Total Empl",
  SUM(SAL)  "Total Sal"
FROM    EMP, DEPT
WHERE   DEPT.DEPTNO = EMP.DEPTNO
GROUP   BY    DNAME, ROLLUP (JOB);
```

DNAME	JOB	Total Empl	Total Sal
SALES	CLERK	1	950
SALES	MANAGER	1	2850
SALES	SALESMAN	4	5600
SALES	All Jobs	6	9400
RESEARCH	CLERK	2	1900
RESEARCH	ANALYST	2	6000
RESEARCH	MANAGER	1	2975
RESEARCH	All Jobs	5	10875
ACCOUNTING	CLERK	1	1300
ACCOUNTING	MANAGER	1	2450
ACCOUNTING	PRESIDENT	1	5000
ACCOUNTING	All Jobs	3	8750

12개의 행이 선택되었다.

- STEP4－3. ROLLUP 함수 결합 칼럼 사용
  - ✓ [예제] JOB과 MGR는 하나의 집합으로 간주하고, 부서별, JOB & MGR에 대한 ROLLUP 결과를 출력한다.

[예제]

```
SELECT DNAME, JOB, MGR,
       SUM(SAL)   "Total Sal"
FROM   EMP, DEPT
WHERE  DEPT.DEPTNO = EMP.DEPTNO
GROUP  BY   ROLLUP (DNAME, (JOB, MGR));
```

※ ROLLUP 함수 사용시 괄호로 묶은 JOB과 MGR의 경우 하나의 집합(JOB+MGR) 칼럼으로 간주하여 괄호 내 각 칼럼별 집계를 구하지 않는다.

DNAME	JOB	MGR	Total Sal
SALES	CLERK	7698	950
SALES	MANAGER	7839	2850
SALES	SALESMAN	7698	5600
SALES			9400
RESEARCH	CLERK	7788	1100
RESEARCH	CLERK	7902	800
RESEARCH	ANALYST	7566	6000
RESEARCH	MANAGER	7839	2975
RESEARCH			10875
ACCOUNTING	CLERK	7782	1300
ACCOUNTING	MANAGER	7839	2450
ACCOUNTING	PRESIDENT		5000
ACCOUNTING			8750
			29025

14개의 행이 선택되었다.

- CUBE 함수
  - ✓ ROLLUP에서는 단지 가능한 Subtotal만을 생성하였지만, CUBE는 결합 가능한 모든 값에 대하여 다차원 집계를 생성한다. CUBE를 사용할 경우에는 내부적으로는 Grouping Columns의 순서를 바꾸어서 또 한 번의 Query를 추가 수행해야 한다. 뿐만 아니라 Grand Total은 양쪽의 Query 에서 모두 생성이 되므로 한 번의 Query에서는 제거되어야만 하므로 ROLLUP에 비해 시스템의 연산 대상이 많다.
  - ✓ 이처럼 Grouping Columns이 가질 수 있는 모든 경우에 대하여 Subtotal을 생성해야 하는 경우에는 CUBE를 사용하는 것이 바람직하나, ROLLUP에 비해 시스템에 많은 부담을 주므로 사용에 주의해야 한다.
  - ✓ CUBE 함수의 경우 표시된 인수들에 대한 계층별 집계를 구할 수 있으며, 이때 표시된 인수들 간에는 계층 구조인 ROLLUP 과는 달리 평등한 관계이므로 인수의 순서가 바뀌는 경우 행간에 정렬 순서는 바뀔 수 있어도 데이터 결과는 같다.
  - ✓ CUBE도 결과에 대한 정렬이 필요한 경우는 ORDER BY 절에 명시적으로 정렬 칼럼이 표시가 되어야 한다.

- STEP5. CUBE 함수 이용
  - ✓ [예제] STEP4. GROUP BY ROLLUP (DNAME, JOB) 조건에서 GROUP BY CUBE (DNAME, JOB) 조건으로 변경해서 수행한다.

[예제]

```
SELECT
 CASE  GROUPING(DNAME)
  WHEN 1     THEN 'All Departments'
  ELSE DNAME END   AS DNAME,
 CASE  GROUPING(JOB)
  WHEN 1     THEN 'All Jobs'
  ELSE JOB   END   AS JOB,
 COUNT(*)  "Total Empl",
 SUM(SAL)  "Total Sal"
FROM   EMP, DEPT
WHERE  DEPT.DEPTNO = EMP.DEPTNO
GROUP  BY   CUBE   (DNAME, JOB);
```

DNAME	JOB	Total Empl	Total Sal
All Departments	All Jobs	14	29025
All Departments	CLERK	4	4150
All Departments	ANALYST	2	6000
All Departments	MANAGER	3	8275
All Departments	SALESMAN	4	5600
All Departments	PRESIDENT	1	5000
SALES	All Jobs	6	9400
SALES	CLERK	1	950
SALES	MANAGER	1	2850
SALES	SALESMAN	4	5600
RESEARCH	All Jobs	5	10875
RESEARCH	CLERK	2	1900
RESEARCH	ANALYST	2	6000
RESEARCH	MANAGER	1	2975
ACCOUNTING	All Jobs	3	8750
ACCOUNTING	CLERK	1	1300
ACCOUNTING	MANAGER	1	2450
ACCOUNTING	PRESIDENT	1	5000

18개의 행이 선택되었다.

- STEP5-2. UNION ALL 사용 SQL
  - ✓ [예제] UNION ALL은 Set Operation 내용으로, 여러 SQL 문장을 연결하는 역할을 할 수 있다. 아래 SQL은 첫 번째 SQL 모듈부터 차례대로 결과가 나오므로 위 CUBE SQL과 결과 데이터는 같으나 행들의 정렬은 다를 수 있다.

[예제]

```
SELECT DNAME, JOB, COUNT(*) "Total Empl", SUM(SAL) "Total Sal"
FROM   EMP, DEPT
WHERE  DEPT.DEPTNO = EMP.DEPTNO
GROUP  BY   DNAME, JOB
UNION  ALL
SELECT DNAME, 'All Jobs', COUNT(*) "Total Empl", SUM(SAL) "Total Sal"
FROM   EMP, DEPT
WHERE  DEPT.DEPTNO = EMP.DEPTNO
GROUP  BY   DNAME
UNION  ALL
SELECT 'All Departments', JOB, COUNT(*) "Total Empl", SUM(SAL) "Total Sal"
FROM   EMP, DEPT
WHERE  DEPT.DEPTNO = EMP.DEPTNO
GROUP  BY   JOB
UNION  ALL
SELECT 'All Departments', 'All Jobs', COUNT(*) "Total Empl", SUM(SAL) "Total Sal"
FROM   EMP, DEPT
WHERE  DEPT.DEPTNO = EMP.DEPTNO ;
```

✓CUBE 함수를 사용하면서 가장 크게 개선되는 부분은 CUBE 사용 전 UNION ALL SQL에서 EMP, DEPT 테이블을 네 번이나 반복 액세스하는 부분을 CUBE 사용 SQL에서는 한 번으로 줄일 수 있는 부분이다.
✓기존에 같은 테이블을 네 번 액세스하는 이유가 되었던 부서와 업무별 소계와 총계 부분을 CUBE 함수를 사용함으로써 한 번의 액세스만으로 구현한다.
✓결과적으로 수행속도 및 자원 사용을 을 개선할 수 있으며, SQL 문장도 더 짧아졌으므로 가독성도 높아졌다.
ROLLUP 함수도 똑 같은 개선 효과를 얻을 수 있다.

- GROUPING SETS 함수
  - ✓ GROUPING SETS를 이용해 더욱 다양한 소계 집합을 만들 수 있는데, GROUP BY SQL 문장을 여러 번 반복하지 않아도 원하는 결과를 쉽게 얻을 수 있게 되었다.
  - ✓ GROUPING SETS에 표시된 인수들에 대한 개별 집계를 구할 수 있으며, 이때 표시된 인수들 간에는 계층 구조인 ROLLUP과는 달리 평등한 관계이므로 인수의 순서가 바뀌어도 결과는 같다.
  - ✓ GROUPING SETS 함수도 결과에 대한 정렬이 필요한 경우는 ORDER BY 절에 명시적으로 정렬 칼럼이 표시가 되어야 한다.

- STEP1. 일반 그룹함수를 이용한 SQL
  - ✓ [예제] 일반 그룹함수를 이용하여 부서별, JOB별 인원 수와 급여 합을 구하라.

[예제]

```
SELECT DNAME,
       'All Jobs' JOB,
       COUNT(*) "Total Empl",
       SUM(SAL) "Total Sal"
FROM   EMP, DEPT
WHERE  DEPT.DEPTNO = EMP.DEPTNO
GROUP  BY   DNAME
UNION  ALL
SELECT 'All Departments' DNAME,
       JOB,
       COUNT(*) "Total Empl",
       SUM(SAL) "Total Sal"
FROM   EMP, DEPT
WHERE  DEPT.DEPTNO = EMP.DEPTNO
GROUP  BY   JOB ;
```

실행 결과

DNAME	JOB	Total Empl	Total Sal
ACCOUNTING	All Jobs	3	8750
RESEARCH	All Jobs	5	10875
SALES	All Jobs	6	9400
All Departments	CLERK	4	4150
All Departments	SALESMAN	4	5600
All Departments	PRESIDENT	1	5000
All Departments	MANAGER	3	8275
All Departments	ANALYST	2	6000

8개의 행이 선택되었다.

※ 실행 결과는 별도의 ORDER BY 조건을 명시하지 않았기 때문에 DNAME이나 JOB에 대해서 정렬이 되어 있지 않다.

- STEP2. GROUPING SETS 사용 SQL
  - ✓ [예제] 일반 그룹함수를 GROUPING SETS 함수로 변경하여 부서별, JOB별 인원 수와 급여 합을 구하라. (인수의 순서가 JOB, DNAME으로 바뀌어도 결과는 같다)

[예제]

```
SELECT
  DECODE(GROUPING(DNAME),1,
  'All Departments', DNAME) AS DNAME,
  DECODE(GROUPING(JOB), 1,
  'All Jobs',         JOB)   AS JOB,
       COUNT(*) "Total Empl",
       SUM(SAL) "Total Sal"
FROM   EMP, DEPT
WHERE  DEPT.DEPTNO = EMP.DEPTNO
GROUP  BY   GROUPING SETS (DNAME, JOB);
```

※ 괄호로 묶은 집합 별로(괄호 내는 계층 구조가 아닌 하나의 데이터로 간주함) 집계를 구할 수 있다.

실행 결과

DNAME	JOB	Total Empl	Total Sal
All Departments	CLERK	4	4150
All Departments	SALESMAN	4	5600
All Departments	PRESIDENT	1	5000
All Departments	MANAGER	3	8275
All Departments	ANALYST	2	6000
ACCOUNTING	All Jobs	3	8750
RESEARCH	All Jobs	5	10875
SALES	All Jobs	6	9400

8개의 행이 선택되었다.

※ GROUPING SETS의 경우 일반 그룹함수를 이용한 SQL과 결과 데이터는 같으나 행들의 정렬 순서는 다를 수 있다.

- STEP3. 3개의 인수를 이용한 GROUPING SETS 이용
  - ✓ GROUPING SETS 함수 사용시 괄호로 묶은 집합별로(괄호 내는 계층구조가 아닌 하나의 데이터로 간주함) 집계를 구할 수 있다.
  - ✓ [예제] 부서-JOB-매니저 별 집계와, 부서-JOB 별 집계와, JOB-매니저 별 집계를 GROUPING SETS 함수를 이용해서 구해본다.

[예제]

```
SELECT DNAME, JOB, MGR,
       SUM(SAL)   "Total Sal"
FROM   EMP, DEPT
WHERE  DEPT.DEPTNO = EMP.DEPTNO
GROUP  BY   GROUPING SETS
          ((DNAME, JOB, MGR), (DNAME, JOB), (JOB, MGR));
```

DNAME	JOB	MGR	Total Sal
SALES	CLERK	7698	950
ACCOUNTING	CLERK	7782	1300
RESEARCH	CLER	7788	1100
RESEARCH	CLERK	7902	800
RESEARCH	ANALY	7566	6000
SALES	MANAGER	7839	2850
RESEARCH	MANAGER	7839	2975
ACCOUNTING	MANAGER	7839	2450
SALES	SALESMAN	7698	5600
ACCOUNTING	PRESIDENT		5000
	CLERK	7698	950
	CLERK	7782	1300
	CLERK	7788	1100
	CLERK	7902	800
	ANALYST	7566	6000
	MANAGER	7839	8275
	SALESMAN	7698	5600
	PRESIDENT		5000
SALES	MANAGER		2850
SALES	CLERK		950
ACCOUNTING	CLERK		1300
ACCOUNTING	MANAGER		2450
ACCOUNTING	PRESIDENT		5000
RESEARCH	MANAGER		2975
SALES	SALESMAN		5600
RESEARCH	ANALYST		6000
RESEARCH	CLERK		1900

실행 결과에서

- ✓ 첫 번째 10건의 데이터는 (DNAME+JOB+MGR) 기준의 집계이며,
- ✓ 두 번째 8건의 데이터는 (JOB+MGR) 기준의 집계이며,
- ✓ 세 번째 9건의 데이터는 (DNAME+JOB) 기준의 집계이다.

27개의 행이 선택되었다.

Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용
제5절 GROUP FUNCTION
핵심정리 및 연습문제

핵심정리 제5절 GROUP FUNCTION

- 데이터 분석을 위한 GROUP FUNCTION으로는
- 소그룹 간의 소계를 계산하는 ROLLUP 함수,
- GROUP BY 항목들 간의 다차원적인 소계를 계산할 수 있는 CUBE 함수,
- 특정 항목에 대한 소계를 계산하는 GROUPING SETS 함수가 있다.

연습문제 제5절 GROUP FUNCTION

문제. 그룹 내 순위 관련 WINDOW 함수의 특징이 틀린 것은 무엇인가?

① RANK 함수는 동일한 값에 대해서는 동일한 순위를 부여한다.
(같은 등수에 여럿이 존재하는 경우 등수가 SKIP 될 수 있음)

② DENSE_RANK 함수는 RANK 함수와 흡사하나, 동일한 순위를 하나의 건수로 취급하는 것이 틀린 점이다.
(같은 등수에 여럿이 존재하는 경우에도 등수가 SKIP 되지 않음)

③ CUMM_RANK 함수는 누적된 순위를 부여할 수 있다.
(등수를 누적 순위로 표현함)

④ RANK 함수가 동일한 값에 대해서는 동일한 순위를 부여하는데 반해, ROW_NUMBER 함수는 유니크한 순위를 부여한다.
(같은 등수가 존재할 수 없음)

문제. 그룹 내 순위 관련 WINDOW 함수의 특징이 틀린 것은 무엇인가?

① RANK 함수는 동일한 값에 대해서는 동일한 순위를 부여한다.
(같은 등수에 여럿이 존재하는 경우 등수가 SKIP 될 수 있음)

② DENSE_RANK 함수는 RANK 함수와 흡사하나, 동일한 순위를 하나의 건수로 취급하는 것이 틀린 점이다.
(같은 등수에 여럿이 존재하는 경우에도 등수가 SKIP 되지 않음)

③ CUMM_RANK 함수는 누적된 순위를 부여할 수 있다.
(등수를 누적 순위로 표현함)

④ RANK 함수가 동일한 값에 대해서는 동일한 순위를 부여하는데 반해, ROW_NUMBER 함수는 유니크한 순위를 부여한다.
(같은 등수가 존재할 수 없음)

정답 : ②

해설 :

ROLLUP, CUBE는 GROUP BY의 확장된 형태로 병렬로 수행이 가능하고 사용하기가 쉽기 때문에 효과적이다. 다차원적인 집계가 필요한 경우는 CUBE를 사용한다.

Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용
제6절 WINDOW FUNCTION
학습하기

- 데이터 분석 함수 (ANSI/ISO-SQL표준)
  - ■ AGGREGATE FUNCTIONS
    - ✓ GROUP FUNCTION의 한 부분이며, GROUP AGGREGATE FUNCTION 이라고도 부를 수 있다. 1장 7절에서 배웠던 COUNT, SUM, AVG, MAX, MIN 외 각종 집계 함수들이 포함되어 있다.
  - ■ GROUP FUNCTIONS
    - ✓ 그룹 함수로는 집계 함수를 제외하면, 소그룹 간의 소계를 계산하는 ROLLUP 함수, GROUP BY 항목들간의 다차원적인 소계를 계산할 수 있는 CUBE 함수, 특정 항목에 대한 소계를 계산하는 GROUPING SETS 함수가 있다. (2장 5절)
  - ■ WINDOW FUNCTIONS
    - ✓ 분석 함수(ANALYTIC FUNCTION)나 순위 함수(RANK FUNCTION)로도 알려져 있는 윈도우 함수(ANSI/ISO-SQL은 WINDOW FUNCTION 용어 사용함)는 데이터웨어하우스에서 발전한 기능이며, 부분적이나마 행과 행 간의 관계를 쉽게 정의하기 위해 만든 함수이다.

- WINDOW FUNCTION
  - ✓ 기존 관계형 데이터베이스는 칼럼과 칼럼간의 연산, 비교, 연결이나 집합에 대한 집계는 쉬운 반면, 행과 행간의 관계를 정의하거나, 행과 행간을 비교, 연산하는 것을 하나의 SQL 문으로 처리 하는 것은 매우 어려운 문제였다.
  - ✓ 부분적이나마 행과 행 간의 관계를 쉽게 정의하기 위해 만든 함수가 바로 WINDOW FUNCTION이다. 윈도우 함수를 활용하면 복잡한 프로그램을 하나의 SQL 문장으로 쉽게 해결할 수 있다.
  - ✓ 복잡하거나 자원을 많이 사용하는 튜닝 기법들을 대체할 수 있는 DBMS의 새로운 기능은 튜닝 관점에서도 최적화된 방법이므로 적극적으로 활용할 필요가 있다. 같은 결과가 나오는 수정된 튜닝 문장보다는 DBMS 벤더에서 최적화된 자원을 사용하도록 만들어진 새로운 기능을 사용하는 것이 일반적으로 더욱 효과가 좋기 때문이다.
  - ✓ WINDOW 함수는 기존에 사용하던 집계 함수도 있고, 새로이 WINDOW 함수 전용으로 만들어진 기능도 있다. 그리고, WINDOW 함수는 다른 함수와는 달리 중첩(NEST) 해서 사용하지는 못하지만, 서브 쿼리에서는 사용할 수 있다.

- WINDOW FUNCTION 종류
  - ✓ 첫 번째, 그룹 내 순위(RANK) 관련 함수는 RANK, DENSE_RANK, ROW_NUMBER 함수가 있다. ANSI/ISO-SQL 기준과 ORACLE, SQL SERVER 등 대부분의 DBMS에서 지원하고 있다.
  - ✓ 두 번째, 그룹 내 집계(AGGREGATE) 관련 함수는 일반적으로 많이 사용하는 SUM, MAX, MIN, AVG, COUNT 함수가 있다. ANSI/ISO-SQL 기준과 ORACLE, SQL SERVER 등 대부분의 DBMS에서 지원하고 있는데, SQL SERVER의 집계 함수는 뒤에서 설명할 OVER 절 내의 ORDER BY 구문을 지원하지 않는다.
  - ✓ 세 번째, 그룹 내 행 순서 관련 함수는 FIRST_VALUE, LAST_VALUE, LAG, LEAD 함수가 있다. ORACLE에서만 지원되는 함수이다.
  - ✓ 네 번째, 그룹 내 비율 관련 CUME_DIST, PERCENT_RANK 함수는 ANSI/ISO-SQL 기준과 ORACLE, SQL SERVER 등 대부분의 DBMS에서 지원하고 있다. NTILE 함수는 ANSI/ISO-SQL 기준에는 없지만, ORACLE, SQL SERVER에서 지원하고 있다. 마지막으로 RATIO_TO_REPORT 함수는 ORACLE에서만 지원되는 함수이다.
  - ✓ 다섯 번째, 선형 분석을 포함한 통계 분석 관련 함수가 있는데, 통계에 특화된 기능이므로 본 과정에서는 설명을 생략한다.

- WINDOW FUNCTION SYNTAX

```
SELECT WINDOW_FUNCTION (ARGUMENTS) OVER
( [PARTITION BY 칼럼] [ORDER BY 절] [WINDOWING 절] )
FROM 테이블 명;
```

  - ✓ WINDOW_FUNCTION : 기존에 사용하던 함수도 있고, 새롭게 WINDOW 함수용으로 추가된 함수도 있다.
  - ✓ ARGUMENTS (인수) : 함수에 따라 0~N 개의 인수가 지정될 수 있다.
  - ✓ OVER : WINDOW 함수에는 OVER 문구가 키워드로 꼭 포함된다
  - ✓ PARTITION BY 절 : 전체 집합을 기준에 의해 소그룹으로 나눌 수 있다.
  - ✓ ORDER BY 절 : 어떤 항목에 대해 순위를 지정할 지 ORDER BY 절을 기술한다.
  - ✓ WINDOWING 절 : WINDOWING 절은 함수의 대상이 되는 행 기준의 범위를 강력하게 지정할 수 있다.
    - ROWS는 물리적인 결과 행의 수를, RANGE는 논리적인 값에 의한 범위를 나타내는데, 둘 중의 하나를 선택해서 사용할 수 있다.

- RANK 함수
  - ✓ RANK 함수는 ORDER BY를 포함한 QUERY 문에서 특정 항목(칼럼)에 대한 순위를 구하는 함수이다. 이때 특정 범위(PARTITION) 내에서 순위를 구할 수도 있고 전체 데이터에 대한 순위를 구할 수도 있다. 또한 동일한 값에 대해서는 동일한 순위를 부여하게 된다.

  - ✓ [예제] 사원 데이터에서 급여가 높은 순서와 JOB 별로 급여가 높은 순서를 같이 출력한다.

[예제]

```
SELECT JOB, ENAME, SAL,
 RANK() OVER (ORDER BY SAL DESC) ALL_RANK,
 RANK() OVER (PARTITION BY JOB ORDER BY SAL DESC) JOB_RANK
FROM EMP;
```


- RANK 함수

JOB	ENAME	SAL	ALL_RANK	JOB_RANK
PRESIDENT	KING	5000	1	1
ANALYST	FORD	3000	2	1
ANALYST	SCOTT	3000	2	1
MANAGER	JONES	2975	4	1
MANAGER	BLAKE	2850	5	2
MANAGER	CLARK	2450	6	3
SALESMAN	ALLEN	1600	7	1
SALESMAN	TURNER	1500	8	2
CLERK	MILLER	1300	9	1
SALESMAN	WARD	1250	10	3
SALESMAN	MARTIN	1250	10	3
CLERK	ADAMS	1100	12	2
CLERK	JAMES	950	13	3
CLERK	SMITH	800	14	4

18개의 행이 선택되었다.

✓업무 구분이 없는 ALL_RANK 칼럼에서 FORD와 SCOTT, WARD와 MARTIN은 동일한 SALARY이므로 같은 순위를 부여한다.

✓업무를 PARTITION으로 구분한 JOB_RANK의 경우 같은 업무 내 범위에서만 순위를 부여한다.

✓하나의 SQL 문장에 ORDER BY SAL DESC 조건과 PARTITION BY JOB 조건이 충돌이 났기 때문에 JOB 별로는 정렬이 되지 않고, ORDER BY SAL DESC 조건으로 정렬이 되었다.

- DENSE_RANK 함수
  - ✓ DENSE_RANK 함수는 RANK 함수와 흡사하나, 동일한 순위를 하나의 건수로 취급하는 것이 틀린 점이다.
  - ✓ [예제] 사원데이터에서 급여가 높은 순서와, 동일한 순위를 하나의 등수로 간주한 결과도 같이 출력한다.

[예제]

```
SELECT JOB, ENAME, SAL,
 RANK() OVER (ORDER BY SAL DESC) RANK,
 DENSE_RANK() OVER (ORDER BY SAL DESC) DENSE_RANK
FROM EMP;
```


- DENSE_RANK 함수

JOB	ENAME	SAL	RANK	DENSE_RANK
PRESIDENT	KING	5000	1	1
ANALYST	FORD	3000	2	2
ANALYST	SCOTT	3000	2	2
MANAGER	JONES	2975	4	3
MANAGER	BLAKE	2850	5	4
MANAGER	CLARK	2450	6	5
SALESMAN	ALLEN	1600	7	6
SALESMAN	TURNER	1500	8	7
CLERK	MILLER	1300	9	8
SALESMAN	WARD	1250	10	9
SALESMAN	MARTIN	1250	10	9
CLERK	ADAMS	1100	12	10
CLERK	JAMES	950	13	11
CLERK	SMITH	800	14	12

18개의 행이 선택되었다.

✓FORD와 SCOTT, WARD와 MARTIN은 동일한 SALARY이므로 RANK와 DENSE_RANK 칼럼에서 모두 같은 순위를 부여한다.

✓그러나, RANK와 DENSE_RANK의 차이를 알 수 있는 데이터는 FORD와 SCOTT의 다음 순위인 JONES의 경우 RANK는 4등으로 DENSE_RANK는 3등으로 표시되어 있다.

✓마찬가지로 WARD와 MARTIN의 다음 순위인 ADAMS의 경우 RANK는 10등으로 DENSE_RANK는 12등으로 표시되어 있다.

- ROW_NUMBER 함수
  - ✓ ROW_NUMBER 함수는 RANK나 DENSE_RANK 함수가 동일한 값에 대해서는 동일한 순위를 부여하는데 반해, 동일한 값이라도 유니크한 순위를 부여한다.
  - ✓ [예제] 사원데이터에서 급여가 높은 순서와, 동일한 순위를 인정하지 않는 등수도 같이 출력한다.

[예제]

```
SELECT JOB, ENAME, SAL,
 RANK() OVER (ORDER BY SAL DESC) RANK,
 ROW_NUMBER() OVER (ORDER BY SAL DESC) ROW_NUMBER
FROM EMP;
```


- ROW_NUMBER 함수

JOB	ENAME	SAL	RANK	ROW_NUMBER
PRESIDENT	KING	5000	1	1
ANALYST	FORD	3000	2	2
ANALYST	SCOTT	3000	2	3
MANAGER	JONES	2975	4	4
MANAGER	BLAKE	2850	5	5
MANAGER	CLARK	2450	6	6
SALESMAN	ALLEN	1600	7	7
SALESMAN	TURNER	1500	8	8
CLERK	MILLER	1300	9	9
SALESMAN	WARD	1250	10	10
SALESMAN	MARTIN	1250	10	11
CLERK	ADAMS	1100	12	12
CLERK	JAMES	950	13	13
CLERK	SMITH	800	14	14

18개의 행이 선택되었다.

✓FORD와 SCOTT, WARD와 MARTIN은 동일한 SALARY이므로 RANK는 같은 순위를 부여했지만, ROW_NUMBER의 경우 동일한 순위를 배제하기 위해 유니크한 순위를 정한다. 위 경우는 같은 SALARY에서는 어떤 순서가 정해질지 알 수 없다. (Oracle의 경우 rowid가 적은 행이 먼저 나온다)

✓이 부분은 데이터베이스 별로 틀린 결과가 나올 수 있으므로, 만일 동일 값에 대한 순서까지 관리하고 싶으면 ROW_NUMBER() OVER (ORDER BY SAL DESC, ENAME) 같이 ORDER BY 절을 이용해 추가적인 정렬 기준을 정의해야 한다.

- MAX 함수
  - ✓ MAX 함수를 이용해 파티션별 윈도우의 최대값을 구할 수 있다.
  - ✓ [예제] 사원들의 급여와 같은 매니저를 두고 있는 사원들의 SALARY 중 최대값을 같이 구한다.

[예제]

```
SELECT MGR, ENAME, SAL,
       MAX(SAL) OVER (PARTITION BY MGR) AS MGR_MAX
FROM   EMP;
```

PARTITION BY MGR 구문을 통해 매니저별로 데이터를 파티션화 한다.

- MAX 함수

MGR	ENAME	SAL	MGR_MAX
7566	FORD	3000	3000
7566	SCOTT	3000	3000
7698	JAMES	950	1600
7698	ALLEN	1600	1600
7698	WARD	1250	1600
7698	TURNER	1500	1600
7698	MARTIN	1250	1600
7782	MILLER	1300	1300
7788	ADAMS	1100	1100
7839	BLAKE	2850	2975
7839	JONES	2975	2975
7839	CLARK	2450	2975
7902	SMITH	800	800
	KING	5000	5000

14개의 행이 선택되었다.

✓실행 결과를 확인하면 파티션 내의 최대값을 파티션 내 모든 행에서 MGR_MAX라는 칼럼 값으로 가질 수 있다.

- MAX 함수
  - ✓[예제] INLINE VIEW를 이용해 파티션별 최대값을 가진 행만 추출할 수도 있다.

[예제]

```
SELECT  MGR, ENAME, SAL
FROM
(SELECT MGR, ENAME, SAL,
        MAX(SAL) OVER
       (PARTITION BY MGR) AS IV_MAX_SAL
 FROM   EMP)
WHERE   SAL =  IV_MAX_SAL;
```

MGR	ENAME	SAL	MGR_MAX
7566	FORD	3000	3000
7566	SCOTT	3000	3000
7698	ALLEN	1600	1600
7782	MILLER	1300	1300
7788	ADAMS	1100	1100
7839	JONES	2975	2975
7902	SMITH	800	800
	KING	5000	5000

8개의 행이 선택되었다.

✓실행 결과를 보면 MGR 7566의 SCOTT, FORD 는 같은 최대값을 가지므로, WHERE SAL = IV_MAX_SAL 조건에 의해 두건 모두 추출되었다.

- MIN 함수
  - ✓ MIN 함수를 이용해 파티션별 윈도우의 최소값을 구할 수 있다.
  - ✓ [예제] 사원들의 급여와 같은 매니저를 두고 있는 사원들을 입사일자를 기준으로 정렬하고, SALARY 최소값을 같이 구한다.

[예제]

```
SELECT MGR, ENAME, HIREDATE, SAL,
       MIN(SAL) OVER(PARTITION BY MGR ORDER BY HIREDATE
                     RANGE UNBOUNDED PRECEDING) AS MGR_MIN
FROM   EMP;

RANGE UNBOUNDED PRECEDING :
현재 행을 기준으로 파티션 내의 첫 번째 행까지의 범위를 지정한다.
```

- MIN 함수

MGR	ENAME	HIREDATE	SAL	MGR_MIN
7566	FORD	1981-12-03	3000	3000
7566	SCOTT	1987-07-13	3000	3000
7698	ALLEN	1981-02-20	1600	1600
7698	WARD	1981-02-22	1250	1250
7698	TURNER	1981-09-08	1500	1250
7698	MARTIN	1981-09-28	1250	1250
7698	JAMES	1981-12-03	950	950
7782	MILLER	1982-01-23	1300	1300
7788	ADAMS	1987-07-13	1100	1100
7839	JONES	1981-04-02	2975	2975
7839	BLAKE	1981-05-01	2850	2850
7839	CLARK	1981-06-09	2450	2450
7902	SMITH	1980-12-17	800	800
	KING	1981-11-17	5000	5000

14개의 행이 선택되었다.

✓실행 결과를 확인하면 파티션 내의 최소값을 파티션 내 모든 행에서 MGR_MIN 이라는 칼럼 값으로 가질 수 있다.

- SUM 함수
  - ✓ SUM 함수를 이용해 파티션별 윈도우의 합을 구할 수 있다.
  - ✓ [예제] OVER 절 내에 ORDER BY 절을 추가해 파티션 내 데이터를 정렬하고 이전 SALARY 데이터까지의 누적값을 출력한다. (SQL SERVER의 경우 집계 함수 경우 OVER 절 내의 ORDER BY 절을 지원하지 않는다)

[예제] - Oracle

```
SELECT MGR, ENAME, SAL,
       SUM(SAL) OVER
      (PARTITION BY MGR ORDER BY SAL RANGE UNBOUNDED PRECEDING)
       AS MGR_SUM
FROM   EMP
```

RANGE UNBOUNDED PRECEDING :
현재 행을 기준으로 파티션 내의 첫 번째 행까지의 범위를 지정한다.

- SUM 함수

MGR	ENAME	SAL	MGR_SUM	참조
7566	SCOTT	3000	6000	
7566	FORD	3000	6000	
7698	JAMES	950	950	
7698	WARD	1250	3450	**
7698	MARTIN	1250	3450	**
7698	TURNER	1500	4950	
7698	ALLEN	1600	6550	
7782	MILLER	1300	1300	
7788	ADAMS	1100	1100	
7839	CLARK	2450	2450	
7839	BLAKE	2850	5300	
7839	JONES	2975	8275	
7902	SMITH	800	800	
	KING	5000	5000	

14개의 행이 선택되었다.

✓** 표시된 7699-WARD와 7699-MARTIN의 급여가 같으므로, 같은 ORDER로 취급하여 950+1250+1250=3450의 값이 되었다.

✓7698-TURNER의 경우 950+1250+1250+1500=4950의 누적합을 가진다.

- AVG 함수
  - ✓ AVG 함수와 파티션별 ROWS 윈도우를 이용해 원하는 조건에 맞는 데이터에 대한 통계값을 구할 수 있다.
  - ✓ [예제] EMP 테이블에서 같은 매니저를 두고 있는 사원들의 평균 SALARY를 구하는데, 조건은 같은 매니저 내에서 자기 바로 앞의 사번과 바로 뒤의 사번인 직원만을 대상으로 한다.

[예제]

```
SELECT MGR, ENAME, HIREDATE, SAL,
       ROUND (AVG(SAL) OVER (PARTITION BY MGR ORDER BY HIREDATE
       ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)) AS MGR_AVG
FROM   EMP;
```

ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING :
현재 행을 기준으로 파티션 내에서 앞의 한 건, 현재 행, 뒤의 한 건을 범위로 지정한다. (ROWS는 현재 행의 앞뒤 건수를 말하는 것임)

- AVG 함수

MGR	ENAME	HIREDATE	SAL	MGR_AVG
7566	FORD	1981-12-03	3000	3000
7566	SCOTT	1987-07-13	3000	3000
7698	ALLEN	1981-02-20	1600	1425
7698	WARD	1981-02-22	1250	1450
7698	TURNER	1981-09-08	1500	1333
7698	MARTIN	1981-09-28	1250	1233
7698	JAMES	1981-12-03	950	1100
7782	MILLER	1982-01-23	1300	1300
7788	ADAMS	1987-07-13	1100	1100
7839	JONES	1981-04-02	2975	2913
7839	BLAKE	1981-05-01	2850	2758
7839	CLARK	1981-06-09	2450	2650
7902	SMITH	1980-12-17	800	800
	KING	1981-11-17	5000	5000

14개의 행이 선택되었다.

✓실행 결과에서 ALLEN의 경우 파티션 내에서 첫 번째 데이터이므로 앞의 한 건은 평균값 집계 대상이 없다. 결과적으로 평균값 집계 대상은 본인의 데이터와 뒤의 한 건으로 평균값을 구한다. (1600+1250) / 2 = 1425의 값을 가진다.

✓TURNER의 경우 앞의 한건과, 본인의 데이터와, 뒤의 한 건으로 평균값을 구한다.(1250+1500+1250) / 3 = 1333의 값을 가진다.

✓JAMES의 경우 파티션 내에서 마지막 데이터이므로 뒤의 한 건을 제외한, 앞의 한 건과 본인의 데이터를 가지고 평균값을 구한다. (1250+950) / 2 = 1100의 값을 가진다.

- COUNT 함수
  - ✓ COUNT 함수와 파티션별 ROWS 윈도우를 이용해 원하는 조건에 맞는 데이터에 대한 통계값을 구할 수 있다.
  - ✓ [예제] 사원들을 급여 기준으로 정렬하고, 본인의 급여보다 급여가 50 이하가 적거나 130 이하로 많은 인원수를 출력하라.

[예제]

```
SELECT ENAME, SAL,
       COUNT(*) OVER (ORDER BY SAL
       RANGE BETWEEN 50 PRECEDING AND 150 FOLLOWING) AS MOV_COUNT
FROM   EMP;
```

RANGE BETWEEN 50 PRECEDING AND 150 FOLLOWING :
현재 행의 값을 기준으로 파티션 내에서 -50 에서 +150 까지 값을 가진 윈도우 내의 모든 행이 대상이 된다. (RANGE는 현재 행의 데이터 값을 기준으로 앞뒤 데이터 값의 범위를 말하는 것임)

- COUNT 함수

ENAME	SAL	MOV_CNT	참조,범위값
SMITH	800	2	(750~ 950)
JAMES	950	2	(900~1100)
ADAMS	1100	3	** (1050~1250)
WARD	1250	3	(1200~1400)
MARTIN	1250	3	(1200~1400)
MILLER	1300	3	(1250~1450)
TURNER	1500	2	(1450~1650)
ALLEN	1600	1	(1550~1750)
CLARK	2450	1	(2400~2600)
BLAKE	2850	4	(2800~3000)
JONES	2975	3	(2925~3125)
SCOTT	3000	3	(2950~3100)
FORD	3000	3	(2950~3100)
KING	5000	1	(4950~5100)

14개의 행이 선택되었다.

✓위 SQL 문장은 파티션이 지정되지 않았으므로 모든 건수를 대상으로 -50 ~ +100 기준에 맞는지 검사하게 된다. ORDER BY SAL로 정렬이 되어 있으므로 비교 연산이 쉬워진다.

✓** 표시된 ADAMS의 경우 자기가 가지고 있는 SALARY 1100을 기준으로 -50 에서 +150 까지 값을 가진 1050에서 1250까지의 값을 가진 JAMES(950), ADAMS(1100), WARD(1250) 3명의 데이터 건수를 구할 수 있다.

- FIRST_VALUE 함수
  - ✓ FIRST_VALUE 함수를 이용해 파티션별 윈도우에서 가장 먼저 나온 값을 구한다.
  - ✓ [예제] 부서별 직원들을 연봉이 높은 순서부터 정렬하고, 파티션 내에서 가장 먼저 나온 값을 출력한다. 같은 연봉이면 이름 순으로 정렬한다.

[예제]

```
SELECT DEPTNO, ENAME, SAL,
       FIRST_VALUE(ENAME) OVER
      (PARTITION BY DEPTNO ORDER BY SAL DESC, ENAME ASC
       ROWS UNBOUNDED PRECEDING) AS RICH_EMP
FROM   EMP;

RANGE UNBOUNDED PRECEDING :
현재 행을 기준으로 파티션 내의 첫 번째 행까지의 범위를 지정한다.
```

- FIRST_VALUE 함수

DEPTNO	ENAME	SAL	RICH_EMP
10	KING	5000	KING
10	CLARK	2450	KING
10	MILLER	1300	KING
20	FORD	3000	FORD
20	SCOTT	3000	FORD
20	JONES	2975	FORD
20	ADAMS	1100	FORD
20	SMITH	800	FORD
30	BLAKE	2850	BLAKE
30	ALLEN	1600	BLAKE
30	TURNER	1500	BLAKE
30	MARTIN	1250	BLAKE
30	WARD	1250	BLAKE
30	JAMES	950	BLAKE

14개의 행이 선택되었다.

✓SQL에서 같은 부서 내에 최고 급여를 받는 사람이 둘 있는 경우를 대비해서 이름을 두 번째 정렬 조건으로 추가한다.

✓같은 급여를 받는 FORD와 SCOTT 중 영문명 아스키코드가 적은 FORD가 최고 급여자로 선정되었다.

- LAST_VALUE 함수
  - ✓ LAST_VALUE 함수를 이용해 파티션별 윈도우에서 가장 나중에 나온 값을 구한다.
  - ✓ [예제] 부서별 직원들을 연봉이 높은 순서부터 정렬하고, 파티션 내에서 가장 마지막에 나온 값을 출력한다.

[예제]

```
SELECT DEPTNO, ENAME, SAL,
       LAST_VALUE(ENAME) OVER
      (PARTITION BY DEPTNO ORDER BY SAL DESC
       ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING)
       AS DEPT_POOR
FROM   EMP;

ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING:
현재 행을 포함해서 파티션 내의 마지막 행까지의 범위를 지정한다.
```

- LAST_VALUE 함수

DEPTNO	ENAME	SAL	DEPT_POOR
10	KING	5000	MILLER
10	CLARK	2450	MILLER
10	MILLER	1300	MILLER
20	SCOTT	3000	SMITH
20	FORD	3000	SMITH
20	JONES	2975	SMITH
20	ADAMS	1100	SMITH
20	SMITH	800	SMITH
30	BLAKE	2850	JAMES
30	ALLEN	1600	JAMES
30	TURNER	1500	JAMES
30	MARTIN	1250	JAMES
30	WARD	1250	JAMES
30	JAMES	950	JAMES

14개의 행이 선택되었다.

✓실행 결과에서 LAST_VALUE는 다른 함수와 달리 공동 등수를 인정하지 않고 가장 나중에 나온 행만을 처리한다.

✓만일 공동 등수가 있을 경우를 의도적으로 정렬하고 싶다면 별도의 정렬 조건을 가진 INLINE VIEW를 사용하거나, OVER () 내의 ORDER BY 조건에 칼럼을 추가해야 한다.

- LAG 함수
  - ✓ LAG 함수를 이용해 파티션별 윈도우에서 이전 몇 번째 행의 값을 가져올 수 있다.
  - ✓ [예제] 직원들을 입사일도가 낮은 기준으로 정렬을 하고, 본인보다 입사일자가 한 명 앞선 사원의 급여를 본인의 급여와 함께 출력한다.

[예제]

```
SELECT ENAME, HIREDATE, SAL,
       LAG(SAL) OVER
      (ORDER BY HIREDATE)
       AS PREV_SAL
FROM   EMP
WHERE  JOB = 'SALESMAN' ;
```

ENAME	HIREDATE	SAL	PREV_SAL
ALLEN	1981-02-20	1600	
WARD	1981-02-22	1250	1600
TURNER	1981-09-08	1500	1250
MARTIN	1981-09-28	1250	1500

4개의 행이 선택되었다.

- LAG 함수
  - ✓ [예제] LAG 함수는 3개의 ARGUMENTS 까지 사용할 수 있는데, 두 번째 인자는 몇번째 앞의 행을 가져올지 결정하는 것이고 (DEFAULT 1), 세번째 인자는 예를 들어 파티션의 첫 번째 행의 경우 가져올 데이터가 없어 NULL 값이 들어오는데 이 경우 다른 값으로 바꾸어 줄 수 있다. 결과적으로 NVL이나 ISNULL 기능과 같다.

[예제]

```
SELECT ENAME, HIREDATE, SAL,
       LAG(SAL, 2, 0) OVER
      (ORDER BY HIREDATE)
       AS PREV_SAL
FROM   EMP
WHERE  JOB = 'SALESMAN' ;
```

ENAME	HIREDATE	SAL	PREV_SAL
ALLEN	1981-02-20	1600	0
WARD	1981-02-22	1250	0
TURNER	1981-09-08	1500	1600
MARTIN	1981-09-28	1250	1250

4개의 행이 선택되었다.

✓LAG(SAL, 2, 0)의 기능은 두 건 앞의 SALARY를 가져오고, 가져올 건이 없는 경우는 0으로 처리한다.

- LEAD 함수
  - ✓ LEAD 함수를 이용해 파티션별 윈도우에서 이후 몇 번째 행의 값을 가져올 수 있다.
  - ✓ [예제] 직원들을 입사일자가 낮은 기준으로 정렬을 하고, 바로 다음에 입사한 인력의 입사일자를 함께 출력한다.

[예제]

```
SELECT ENAME, HIREDATE,
       LEAD(HIREDATE, 1) OVER
      (ORDER BY HIREDATE)
       AS "NEXTHIRED"
FROM   EMP
WHERE  JOB = 'SALESMAN' ;
```

ENAME	HIREDATE	NEXTHIRED
ALLEN	1981-02-20	1981-02-22
WARD	1981-02-22	1981-04-02
TURNER	1981-09-08	1981-09-28
MARTIN	1981-09-28	

4개의 행이 선택되었다.

✓LEAD 함수는 3개의 ARGUMENTS 까지 사용할 수 있는데, 두 번째 인자는 몇 번째 후의 행을 가져올지 결정하는 것이고 (DEFAULT 1), 세 번째 인자는 예를 들어 파티션의 마지막 행의 경우 가져올 데이터가 없어 NULL 값이 들어오는데 이 경우 다른 값으로 바꾸어 줄 수 있다.
✓결과적으로 NVL이나 ISNULL 기능과 같다.

- RATIO_TO_REPORT 함수
  - ✓ RATIO_TO_REPORT 함수를 이용해 파티션 내 전체 SUM(칼럼)값에 대한 행별 칼럼 값의 백분율을 소수점으로 구할 수 있다. 결과 값은 > 0 & <= 1 의 범위를 가진다. 그리고, 개별 RATIO의 합을 구하면 1이 된다.
  - ✓ [예제] SALESMAN JOB 인 사원들을 대상으로 전체 급여에서 본인이 차지하는 비율을 출력한다.

[예제]

```
SELECT ENAME, SAL,
 ROUND(RATIO_TO_REPORT(SAL)
       OVER (), 2) AS R_R
FROM   EMP
WHERE  JOB = 'SALESMAN';
```

ENAME	SAL	R_R	참조계산식
ALLEN	1600	0.29	(1600 / 5600)
WARD	1250	0.22	(1250 / 5600)
MARTIN	1250	0.22	(1250 / 5600)
TURNER	1500	0.27	(1500 / 5600)

4개의 행이 선택되었다.

✓실행 결과에서 전체 값은 1650 + 1250 + 1250 + 1500 = 5600 이 되고, RATIO_TO_REPORT 함수 연산의 분모로 사용된다.
✓개별 RATIO의 전체 합을 구하면 1이 되는 것을 확인할 수 있다. 0.29 + 0.22 + 0.22 + 0.27 = 1

- PERCENT_RANK 함수
  - ✓ PERCENT_RANK 함수를 이용해 파티션별 윈도우에서 제일 먼저 나오는 것을 0으로, 제일 늦게 나오는 것을 1으로 하여, 값이 아닌 행의 순서별 백분율을 구한다. 결과 값은 >= 0 & <= 1 의 범위를 가진다.
  - ✓ [예제] 같은 JOB을 가진 사원들의 집합에서 본인의 급여가 순서상 몇 번째 위치쯤에 있는지 0과 1 사이의 값으로 출력한다.

[예제]

```
SELECT DEPTNO, ENAME, SAL,
       PERCENT_RANK() OVER
      (PARTITION BY DEPTNO ORDER BY SAL DESC) AS P_R
FROM   EMP;
```

- PERCENT_RANK 함수

DEPTNO	ENAME	SAL	P_R
10	KING	5000	0
10	CLARK	2450	0.5
10	MILLER	1300	1
20	SCOTT	3000	0
20	FORD	3000	0
20	JONES	2975	0.5
20	ADAMS	1100	0.75
20	SMITH	800	1
30	BLAKE	2850	0
30	ALLEN	1600	0.2
30	TURNER	1500	0.4
30	MARTIN	1250	0.6
30	WARD	1250	0.6
30	JAMES	950	1

14개의 행이 선택되었다.

✓DEPTNO 10 의 경우 3건이므로 구간은 2개가 된다. 0과 1 사이를 2개의 구간으로 나누면 0, 0.5, 1이 된다.

✓DEPTNO 20 의 경우 5건이므로 구간은 4개가 된다. 0과 1 사이를 4개의 구간으로 나누면 0, 0.25, 0.5, 0.75, 1이 된다.

✓DEPTNO 30 의 경우 6건이므로 구간은 5개가 된다. 0과 1 사이를 5개의 구간으로 나누면 0, 0.2, 0.4, 0.6, 0.8, 1이 된다.

✓SCOTT, FORD와 WARD, MARTIN의 경우 ORDER BY SAL DESC 구문에 의해 급여가 같으므로 같은 ORDER로 취급한다.

- CUME_DIST 함수
  - ✓ CUME_DIST 함수를 이용해 파티션별 윈도우의 전체건수에서 현재 행보다 작거나 같은 건수에 대한 누적백분율을 구한다. 결과 값은 > 0 & <= 1 의 범위를 가진다.
  - ✓ [예제] 같은 JOB을 가진 사원들의 집합에서 본인의 급여가 누적 순서상 몇 번째 위치쯤에 있는지 0과 1 사이의 값으로 출력한다.

[예제]

```
SELECT DEPTNO, ENAME, SAL,
       CUME_DIST()
       OVER (PARTITION BY DEPTNO ORDER BY SAL DESC)
       AS CUME_DIST
FROM   EMP;
```

- CUME_DIST 함수

DEPTNO	ENAME	SAL	CUME_DIST
10	KING	5000	0.3333
10	CLARK	2450	0.6667
10	MILLER	1300	1.0000
20	SCOTT	3000	0.4000
20	FORD	3000	0.4000
20	JONES	2975	0.6000
20	ADAMS	1100	0.8000
20	SMITH	800	1.0000
30	BLAKE	2850	0.1667
30	ALLEN	1600	0.3333
30	TURNER	1500	0.5000
30	MARTIN	1250	0.8333
30	WARD	1250	0.8333
30	JAMES	950	1.0000

14개의 행이 선택되었다.

✓DEPTNO가 10인 경우 윈도우가 전체 3건이므로 0.3333 단위의 간격을 가진다. 즉, 0.3333, 0.6667, 1의 값이 된다.

✓DEPTNO가 20인 경우 윈도우가 전체 5건이므로 0.2000 단위의 간격을 가진다. 즉, 0.2000, 0.4000, 0.6000, 0.8000, 1의 값이 된다.

✓DEPTNO가 30인 경우 윈도우가 전체 6건이므로 0.1667 단위의 간격을 가진다. 즉, 0.1667, 0.3333, 0.5000, 0.6667, 0.8333, 1의 값이 된다.

✓* 표시가 있는 SCOTT, FORD와 WARD, MARTIN의 경우 ORDER BY SAL에 의해 SAL 이 같으므로 같은 ORDER로 취급한다. 다른 WINDOW 함수의 경우 동일 순서면 앞 행의 함수 결과 값을 따르는데, CUME_DIST의 경우는 동일 순서면 뒤 행의 함수 결과값을 기준으로 한다.

- NTILE 함수
  - ✓ NTILE 함수를 이용해 파티션별 전체 건수를 ARGUMENT 값으로 N 등분한 결과를 구할 수 있다.
  - ✓ [예제] 전체 사원을 급여가 높은 순서로 정렬하고, 급여를 기준으로 4개의 그룹으로 분류한다.

[예제]

```
SELECT ENAME, SAL,
       NTILE(4) OVER (ORDER BY SAL DESC) AS QUAR_TILE
FROM   EMP;
```

- NTILE 함수

DEPTNO	ENAME	SAL	QUAR_TILE
10	KING	5000	1
10	FORD	3000	1
10	SCOTT	3000	1
20	JONES	2975	1
20	BLAKE	2850	2
20	CLARK	2450	2
20	ALLEN	1600	2
20	TURNER	1500	2
30	MILLER	1300	3
30	WARD	1250	3
30	MARTIN	1250	3
30	ADAMS	1100	4
30	JAMES	950	4
30	SMITH	800	4

14개의 행이 선택되었다.

✓위 예제에서 NTILE(4)의 의미는 14명의 팀원을 4개 조로 나눈다는 의미이다.

✓전체 14명을 4개의 집합으로 나누면 몫이 3명, 나머지가 2명이 된다.

✓나머지 두 명은 앞의 조부터 할당한다. 즉, 4명 + 4명 + 3명 + 3명으로 조를 나누는 것이 가장 합리적이다.

Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용
제6절 WINDOW FUNCTION
핵심정리 및 연습문제

- 데이터 분석을 위한 WINDOW FUNCTION은 부분적이나마 행과 행 간의 관계를 쉽게 정의하기 위해 만든 함수이다.
- 첫 번째, 그룹 내 순위(RANK) 관련 함수는 RANK, DENSE_RANK, ROW_NUMBER 함수가 있다.
- 두 번째, 그룹 내 집계(AGGREGATE) 관련 함수는 일반적으로 많이 사용하는 SUM, MAX, MIN, AVG, COUNT 함수가 있다.
- 세 번째, 그룹 내 행 순서 관련 함수는 FIRST_VALUE, LAST_VALUE, LAG, LEAD 함수가 있다.
- 네 번째, 그룹 내 비율 관련 함수는 CUME_DIST, PERCENT_RANK, NTILE, RATIO_TO_REPORT 함수가 있다.

문제. 소계, 중계, 합계처럼 계층적 분류를 포함하고 있는 데이터의 집계에 적합한 GROUP 함수 두 가지는 무엇인가?

① ROLLUP, SUM
② ROLLUP, CUBE
③ GROUPING, SUM
④ CUBE, SUM

문제. 소계, 중계, 합계처럼 계층적 분류를 포함하고 있는 데이터의 집계에 적합한 GROUP 함수 두 가지는 무엇인가?

① ROLLUP, SUM
② ROLLUP, CUBE
③ GROUPING, SUM
④ CUBE, SUM

정답 : ③
해설 :
그룹 내 순위 관련 WINDOW FUNCTION으로는 RANK, DENSE_RANK, ROW_NUMBER 함수가 있고, ③의 지문은 DENSE_RANK 함수에 대한 설명이며, CUMM_RANK 함수는 존재하지 않는다.

Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용
제7절 DCL 학습하기

제7절 DCL(Data Control Language)

- DCL
  - ✓ 유저를 생성하고 권한을 제어할 수 있는 DCL(DATA CONTROL LANGUAGE) 명령어가 있다.
  - ✓ 대부분의 데이터베이스는 데이터 보호와 보안을 위해서 유저와 권한을 관리하고 있다.
  - ✓ 예를 들어 Oracle을 설치하면 기본적으로 제공되는 유저들인 SYS, SYSTEM, SCOTT 유저에 대해서 간단하게 알아본다.

유 저	역 할
SCOTT	Oracle 테스트용 샘플/교육용 유저 Default 패스워드 : TIGER
SYS	DBA ROLE을 부여받은 유저
SYSTEM	데이터베이스의 모든 시스템 권한을 부여받은 DBA 유저 Oracle 설치 완료 시에 패스워드 설정

제7절 DCL(Data Control Language)

- 시스템 권한
  - ✓ Oracle의 DBA 권한을 가지고 있는 SYSTEM 유저로 접속하면 유저 생성 권한(CREATE USER)을 다른 유저에게 부여할 수 있다.
  - ✓ [예제] SCOTT 유저에게 유저생성 권한(CREATE USER)을 부여한 후, SCOTT으로 접속 후 PJS 유저를 생성한다.

[예제] 및 실행 결과

```
CONN SYSTEM/MANAGER
연결되었다.

GRANT CREATE USER TO SCOTT;
권한이 부여되었다.

CONN SCOTT/TIGER
연결되었다.

CREATE USER PJS IDENTIFIED BY KOREA7;
사용자가 생성되었다.
```

- SESSION 생성 권한
  - ✓ Oracle의 PJS 유저가 생성됐지만 아무런 권한도 부여받지 못했기 때문에 로그인을 하면 CREATE SESSION 권한이 없다는 오류가 발생한다.
  - ✓ [예제] 유저가 로그인을 하려면 CREATE SESSION 권한을 부여받아야 한다.

[예제] 및 실행 결과

```
CONN SYSTEM/MANAGER
연결되었다.

GRANT CREATE SESSION TO PJS;
권한이 부여되었다.

CONN PJS/KOREA7
연결되었다.
```


- TABLE 생성 권한
  - ✓ Oracle의 PJS 유저는 로그인 권한만 부여되었기 때문에 테이블을 생성하려면 테이블 생성 권한(CREATE TABLE)이 불충분하다는 오류가 발생한다.
  - ✓ [예제] 유저가 로그인을 하려면 CREATE SESSION 권한을 부여받아야 한다.

[예제] 및 실행 결과

```
CONN SYSTEM/MANAGER
연결되었다.

GRANT CREATE TABLE TO PJS;
권한이 부여되었다.

CONN PJS/KOREA7
연결되었다.

CREATE TABLE MENU (
  MENU_SEQ  NUMBER NOT NULL,
  TITLE     VARCHAR2(10) );
테이블이 생성되었다.
```

- 오브젝트 권한
  - ✓ Oracle의 오브젝트(객체) 권한은 특정 오브젝트인 테이블, 뷰 등에 대한 SELECT, INSERT, DELETE, UPDATE 작업 명령어를 의미한다.
  - ✓ 오브젝트 권한과 오브젝트와의 관계

객체 권한	TABLE	VIEW	SEQUENCE	PROCEDURE
ALTER	○		○	
DELETE	○	○		
EXECUTE				○
INDEX	○			
INSERT	○	○		
REFERENCES	○			
SELECT	○	○	○	
UPDATE	○	○		

- 오브젝트 권한
  - ✓ [예제] Oracle의 PJS 유저로 접속하여 SCOTT 유저에게 MENU 테이블을 SELECT 할 수 있는 권한을 부여한다.

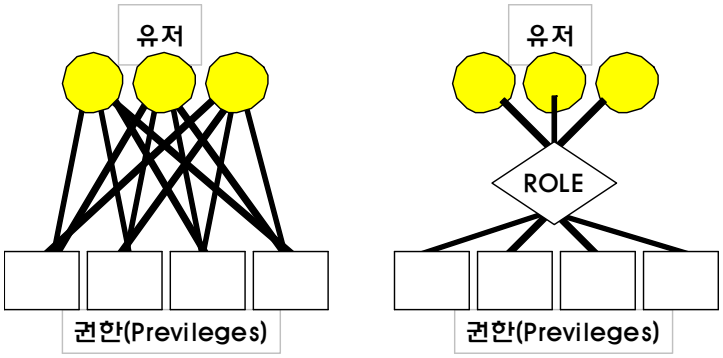
[예제] 및 실행 결과

```
CONN SCOTT/TIGER
연결되었다.

SELECT * FROM PJS.MENU;
ERROR: 테이블 또는 뷰가 존재하지 않는다.

CONN PJS/KOREA7
연결되었다.

GRANT SELECT ON MENU TO SCOTT;
권한이 부여되었다.

CONN SCOTT/TIGER
연결되었다.

SELECT * FROM PJS.MENU;
정상적으로 데이터 조회됨
```

✓ SCOTT 유저는 PJS.MENU 테이블을 SELECT하는 권한만 부여 받았기 때문에 UPDATE, INSERT, DELETE와 같은 다른 작업을 할 수 없다.

✓ 오브젝트 권한은 SELECT, INSERT, DELETE, UPDATE 등의 권한을 따로 따로 관리한다.

- ROLE을 이용한 권한 부여
  - ✓ Oracle의 데이터베이스 관리자는 유저가 생성될 때마다 각각의 권한들을 유저에게 부여하는 작업을 수행해야 하는데, 효율적인 관리를 위해 유저들과 권한들 사이에서 중개 역할을 하는 ROLE을 제공한다.
  - ✓ 왼쪽 그림은 권한을 직접 유저에게 할당할 때를 나타내는 것이며, 오른쪽 그림은 ROLE에 권한을 부여한 후 ROLE을 유저들에게 부여하는 것을 나타내고 있다.



- ROLE을 이용한 권한 부여
  - ✓ [예제] 이제 LOGIN_TABLE이라는 ROLE을 만들고, 이 ROLE을 이용하여 JISUNG 유저에게 권한을 부여한다.

[예제] 및 실행 결과

```
CONN SYSTEM/MANAGER
연결되었다.

CREATE ROLE LOGIN_TABLE;
롤이 생성되었다.

GRANT CREATE SESSION, CREATE TABLE TO LOGIN_TABLE;
권한이 부여되었다.

GRANT LOGIN_TABLE TO JISUNG;
권한이 부여되었다.

CONN JISUNG/KOREA7
연결되었다.

CREATE TABLE MENU2(
  MENU_SEQ NUMBER NOT NULL,
  TITLE    VARCHAR2(10));
테이블이 생성되었다.
```

- ROLE을 이용한 권한 부여
  - ✓ Oracle에서 가장 많이 사용하는 ROLE은 CONNECT와 RESOURCE이다.
  - ✓ CONNECT는 CREATE SESSION과 같은 로그인 권한이 포함되어 있고, RESOURCE는 CREATE TABLE과 같은 오브젝트 생성 권한이 포함되어 있다.
  - ✓ 일반적으로 Oracle에서 유저를 생성할 때 CONNECT와 RESOURCE ROLE을 사용하여 기본 권한을 부여한다.

CONNECT	RESOURCE
ALTER SESSION	CREATE CLUSTER
CREATE CLUSTER	CREATE INDEXTYPE
CREATE DATABASE LINK	CREATE OPERATOR
CREATE MENU_SEQUENCE	CREATE PROCEDURE
CREATE SESSION	CREATE MENU_SEQUENCE
CREATE SYNONYM	CREATE TABLE
CREATE TABLE	CREATE TRIGGER
CREATE VIEW	CREATE VIEW

- SQL Server 로그인
  - ✓ 첫 번째, Windows 인증 방식으로 Windows에 로그인한 정보를 가지고 SQL Server에 접속하는 방식이다. Microsoft Windows 사용자 계정을 통해 연결되면 SQL Server는 운영 체제의 Windows 보안 주체 토큰을 사용하여 계정 이름과 암호가 유효한지 확인한다. 즉, Windows에서 사용자 ID를 확인한다. SQL Server는 암호를 요청하지 않으며 ID의 유효성 검사를 수행하지 않는다.
  - ✓ 두 번째, 혼합 모드(Windows 인증 또는 SQL 인증) 방식으로 기본적으로 Windows 인증으로도 SQL Server에 접속 가능하며, Oracle의 인증과 같은 방식으로 사용자 아이디와 비밀번호로 SQL Server에 접속하는 방식이다. SQL 인증을 사용할 때는 암호(숫자+문자+특수문자 등을 혼합하여 사용)를 사용해야 한다.

- SQL Server 권한
  - ✓ SQL Server에서는 Oracle과 같이 Role을 자주 사용하지 않는다. 대신 위에서 언급한 서버 수준 역할 및 데이터베이스 수준 역할을 이용하여 로그인 및 사용자 권한을 제어한다.
  - ✓ 인스턴스 수준의 작업이 필요한 경우 서버 수준 역할을 부여하고 그보다 작은 개념인 데이터베이스 수준의 권한이 필요한 경우 데이터베이스 수준의 역할을 부여하면 된다.
  - ✓ 즉, 인스턴스 수준을 요구하는 로그인에는 서버 수준 역할을, 데이터베이스 수준을 요구하는 사용자에게는 데이터베이스 수준 역할을 부여한다.

Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용
제7절 DCL 핵심정리 및 연습문제

핵심정리 제7절 DCL(Data Control Language)

- 유저를 생성하고 권한을 제어할 수 있는 DCL(DATA CONTROL LANGUAGE) 명령어가 있으며,
- GRANT 문장을 통해 권한을 생성하고
- REVOKE 문장을 통해 권한을 회수한다.

연습문제 제7절 DCL(Data Control Language)

문제. 다음 중 맞는 내용은 무엇인가?

① 유저를 생성하면 생성한 유저로 바로 로그인 할 수 있다.

② 새롭게 생성된 유저라면 조건 없이 새로운 유저를 만들 수 있다.

③ 유저 생성은 누구나 할 수 있지만 권한 설정은 데이터베이스 관리자만 가능하다.

④ 다른 유저의 테이블은 그 테이블에 대한 권한 없이는 조회할 수 없다.

연습문제 제7절 DCL(Data Control Language)

문제. 다음 중 맞는 내용은 무엇인가?

① 유저를 생성하면 생성한 유저로 바로 로그인 할 수 있다.

② 새롭게 생성된 유저라면 조건 없이 새로운 유저를 만들 수 있다.

③ 유저 생성은 누구나 할 수 있지만 권한 설정은 데이터베이스 관리자만 가능하다.

④ 다른 유저의 테이블은 그 테이블에 대한 권한 없이는 조회할 수 없다.

정답 : ④

해설 :

테이블에 대한 권한은 각 테이블의 소유자가 가지고 있기 때문에 소유자로부터 권한을 받지 않으면 다른 유저의 테이블에 접근할 수 없다.

Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용

제8절 절차형 SQL

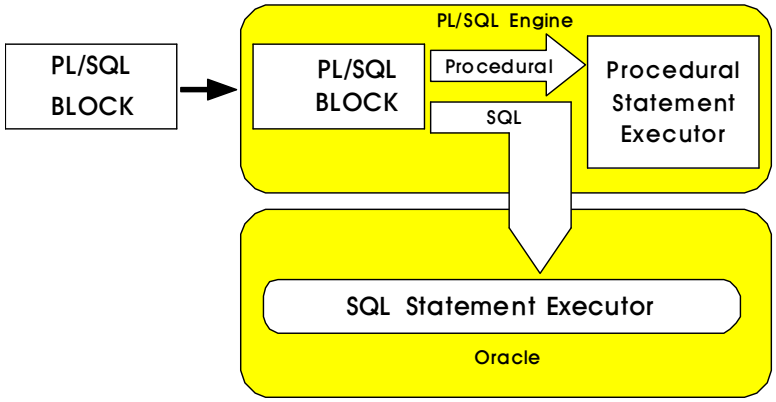
학습하기

- 절차형 SQL
  - ✓ 일반적인 개발 언어처럼 SQL에도 절차 지향적인 프로그램이 가능하도록 DBMS 벤더별로 PL(Procedural Language)/SQL(Oracle), SQL/PL(DB2), T-SQL(SQL Server) 등의 절차형 SQL을 제공하고 있다.
  - ✓ 절차형 SQL을 이용하여 SQL문의 연속적인 실행이나 조건에 따른 분기처리를 이용하여 특정 기능을 수행하는 저장 모듈을 생성할 수 있다.
  - ✓ 절차형 SQL을 이용하여 PROCEDURE, TRIGGER, USER DEFINED FUNCTION을 만들 수 있다.
  - ✓ PL/SQL과 관련된 내용은 상당히 다양하고 분량이 많기 때문에 상세한 내역은 각 DBMS 벤더의 매뉴얼을 참조한다.

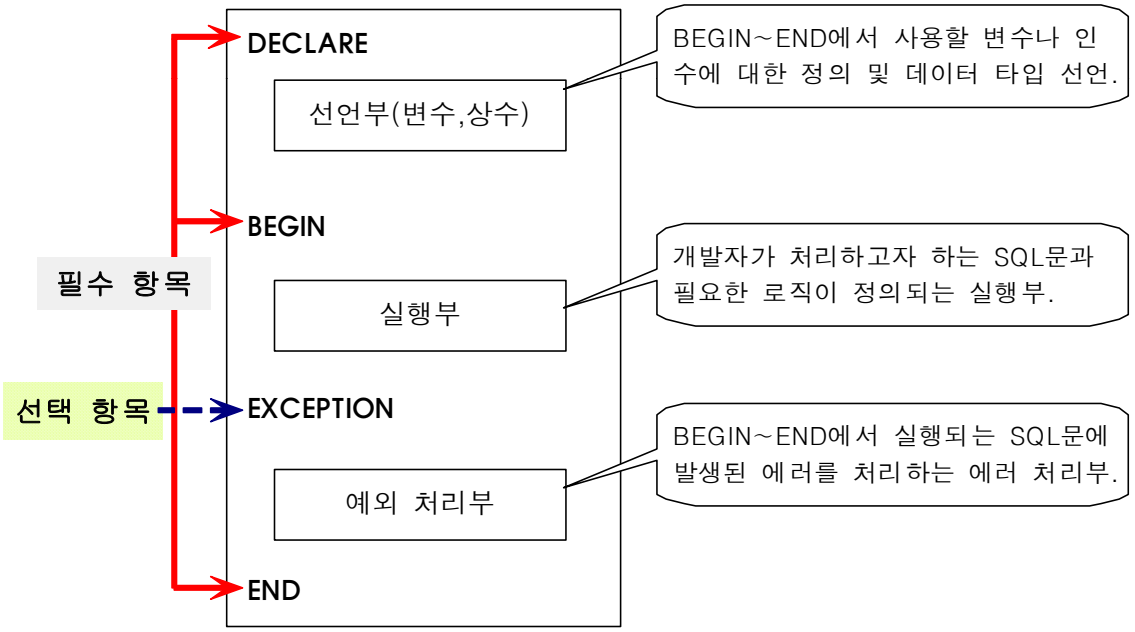
- PL/SQL 특징
  - ✓ Oracle의 PL/SQL은 Block 구조로 되어있고 Block 내에는 DML 문장과 QUERY 문장, 그리고 절차형 언어(IF, LOOP) 등을 사용할 수 있으며, 절차적 프로그래밍을 가능하게 하는 트랜잭션 언어이다.
  - ✓ PL/SQL을 이용하여 다양한 저장 모듈(Stored Module)을 개발할 수 있다.
  - ✓ 저장 모듈이란 PL/SQL 문장을 데이터베이스 서버에 저장하여 사용자와 애플리케이션 사이에서 공유할 수 있도록 만든 일종의 SQL 컴포넌트 프로그램이며, 독립적으로 실행되거나 다른 프로그램으로부터 실행될 수 있는 완전한 실행 프로그램이다.
  - ✓ 데이터베이스 내부에 저장됨으로 인해서 관리적인 문제가 발생할 수 있다.
  - ✓ 일반적으로 3개의 유형 중에는 데이터 무결성 및 데이터 집계를 위해 트리거 사용 빈도가 높다
  - ✓ 프로시저의 경우 DBA 외에 일반 사용자가 사용하는 경우가 적음
  - ✓ 스칼라 서브쿼리 이후 사용자정의함수의 사용 빈도는 줄어들고 있음

- PL/SQL 특징
  - ✓ PL/SQL은 Block 구조로 되어있어 각 기능별로 모듈화가 가능하다
  - ✓ 변수, 상수 등을 선언하여 SQL 문장 간 값을 교환한다.
  - ✓ IF, LOOP 등의 절차형 언어를 사용하여 절차적인 프로그램이 가능하도록 한다.
  - ✓ DBMS 정의 에러나 사용자 정의 에러를 정의하여 사용할 수 있다.
  - ✓ PL/SQL은 Oracle에 내장되어 있으므로 Oracle과 PL/SQL을 지원하는 어떤 서버로도 프로그램을 옮길 수 있다.
  - ✓ PL/SQL은 응용 프로그램의 성능을 향상시킨다.
  - ✓ PL/SQL은 여러 SQL 문장을 Block으로 묶고 한 번에 Block 전부를 서버로 보내기 때문에 통신량을 줄일 수 있다.

- PL/SQL 엔진
  - ✓ PL/SQL 엔진은 PL/SQL Block 프로그램을 입력받으면 SQL 문장과 프로그램 문장을 구분하여 처리한다.
  - ✓ 즉 프로그램 문장은 PL/SQL 엔진이 처리하고 SQL 문장은 Oracle 서버의 SQL Statement Executor가 실행하도록 작업을 분리하여 처리한다.



- PL/SQL 블록 구조



- PL/SQL 기본 문법(Syntax)

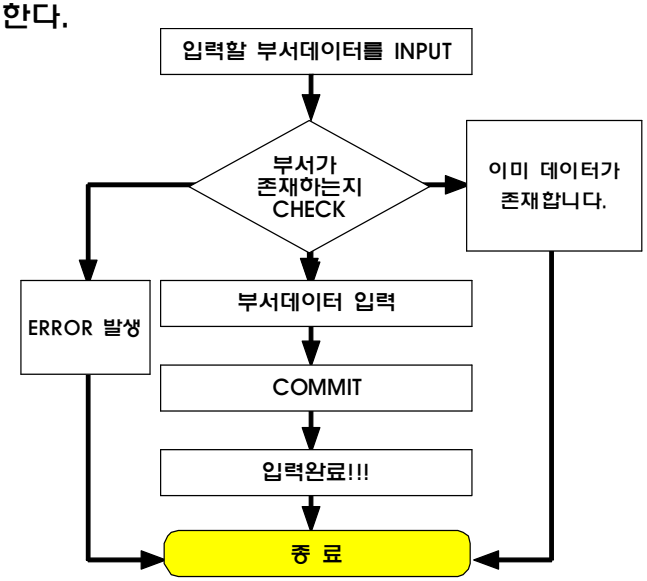
```
CREATE [OR REPLACE] Procedure [Procedure_name]
( argument1  [mode]  data_type1,
  argument2  [mode]  date_type2,
  ...  ... )
IS [AS]
  ...  ...
BEGIN
  ...  ...
EXCEPTION
  ...  ...
END;
/
```

✓ CREATE TABLE 명령어로 테이블을 생성하듯 CREATE 명령어로 데이터베이스 내에 프로시저를 생성할 수 있다. 삭제는 DROP 명령어 사용함.

✓ [OR REPLACE] 절은 데이터베이스 내에 같은 이름의 프로시저가 있을 경우, 기존의 프로시저를 무시하고 새로운 내용으로 덮어쓰기 하겠다는 의미이다.

- PL/SQL 기본 문법(Syntax)
  - ✓ argument는 프로시저가 호출될 때 프로시저 안으로 어떤 값이 들어오거나 혹은 프로시저에서 처리한 결과값을 운영 체제로 리턴시킬 매개 변수를 지정할 때 사용한다.
  - ✓ [mode] 부분에 지정할 수 있는 매개 변수의 유형은 3가지가 있다.
    - IN은 운영 체제에서 프로시저로 전달될 변수의 MODE이고,
    - OUT은 프로시저에서 처리된 결과가 운영체제로 전달되는 MODE이다.
    - 마지막으로 잘 쓰지는 않지만 INOUT MODE가 있는데
      이 MODE는 IN과 OUT 두 가지의 기능을 동시에 수행하는 MODE이다.
  - ✓ 마지막에 있는 슬래쉬("/")는 데이터베이스에게 프로시저를 컴파일하라는 명령어이다.

- Procedure 생성
  - ✓ [예제] SCOTT 유저가 소유하고 있는 DEPT 테이블에 새로운 부서를 등록하는 Procedure를 작성한다.



- Procedure 생성

```
CREATE OR REPLACE Procedure p_DEPT_insert                    ----------①
 ( v_DEPTNO  in  number,
   v_dname   in  varchar2,
   v_loc     in  varchar2,
   v_result  out varchar2)
IS
cnt number := 0;
BEGIN
  SELECT COUNT(*) INTO CNT                                   ----------②
  FROM   DEPT
  WHERE  DEPTNO = v_DEPTNO;
  if cnt > 0 then                                            ----------③
     v_result := '이미 등록된 부서번호이다';
  else
     INSERT INTO DEPT (DEPTNO, DNAME, LOC)                   ----------④
     VALUES (v_DEPTNO, v_dname, v_loc);
     COMMIT;                                                 ----------⑤
     v_result := '입력 완료!!';
  end if;
EXCEPTION                                                    ----------⑥
     WHEN OTHERS THEN
          ROLLBACK;
           v_result := 'ERROR 발생';
END;
/
```

✓PL/SQL에서 사용하는 SELECT 문장은 결과값이 반드시 있어야 하며, 그 결과 역시 반드시 하나여야 한다. 조회 결과가 없거나 하나 이상인 경우에는 에러를 발생시킨다.

✓일반적으로 대입 연산자는 "="을 사용하지만 PL/SQL에서는 ":="를 사용한다.

✓EXCEPTION에는 WHEN ~ THEN 절을 사용하여 에러의 종류별로 적절히 처리한다.

- Procedure 생성

① DEPT 테이블에 들어갈 칼럼 값(부서코드, 부서명, 위치)을 입력 받는다.

② 입력 받은 부서코드가 존재하는지 확인한다.

③ 부서코드가 존재하면 '이미 등록된 부서번호입니다'라는 메시지를 출력값에 넣는다.

④ 부서코드가 존재하지 않으면 입력받은 필드 값으로 새로운 부서 레코드를 입력한다.

⑤ 새로운 부서가 정상적으로 입력됐을 경우에는 COMMIT 명령어를 통해서 트랜잭션을 종료한다.

⑥ 에러가 발생하면 모든 트랜잭션을 취소하고 'ERROR 발생 '이라는 메시지를 출력값에 넣는다.

- Procedure 활용

```
SQL> SELECT * FROM DEPT;                                    ----------①
    DEPTNO DNAME          LOC
 --------- ----------     ---------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON

SQL> variable rslt varchar2(30);                            ----------②

SQL> EXECUTE p_DEPT_insert(10,'dev','seoul',:rslt);         ----------③
PL/SQL 처리가 정상적으로 완료되었다.

SQL> print rslt;                                            ----------④
RSLT
------------------------------
이미 등록된 부서번호이다
```


- Procedure 활용

```
SQL> EXECUTE p_DEPT_insert(50,'NewDev','seoul',:rslt); ----------⑤
PL/SQL 처리가 정상적으로 완료되었다.

SQL> print rslt;                                            ----------⑥
RSLT
------------------------------
입력 완료!

SQL> SELECT * FROM DEPT;                                    ----------⑦
    DEPTNO DNAME          LOC
 --------- ----------     ---------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 NewDev         SEOUL
```

- Procedure 활용

① DEPT 테이블을 조회하면 총 4개 행의 결과가 출력된다.

② Procedure를 실행한 결과값을 받을 변수를 선언한다. (BIND 변수)

③ 존재하는 DEPTNO(10)를 가지고 Procedure를 실행한다.

④ DEPTNO가 10인 부서는 이미 존재하기 때문에 변수 rslt를 print해 보면 '이미 등록된 부서번호이다' 라고 출력된다.

⑤ 이번에는 새로운 DEPTNO(50)를 가지고 입력한다.

⑥ rslt를 출력해 보면 '입력 완료!' 라고 출력된다.

⑦ DEPT 테이블을 조회하여 보면 DEPTNO가 50인 데이터가 정확하게 저장되었음을 확인할 수 있다.

- 사용자 정의 함수(User Defined Function)

✓ User Defined Function은 Procedure처럼 절차형 SQL을 로직과 함께 데이터베이스 내에 저장해 놓은 명령문의 집합을 의미한다. 앞에서 학습한 SUM, SUBSTR, NVL 등의 함수는 벤더에서 미리 만들어둔 내장 함수이고, 사용자가 별도의 함수를 만들 수도 있다.

✓ Function이 Procedure와 다른 점은 RETURN을 사용해서 일반적으로 하나의 값을 반드시 되돌려 줘야 한다는 것이다. 즉 Function은 Procedure와는 달리 SQL 문장에서 특정 작업을 수행하고 반드시 수행 결과값을 리턴한다.

- User Defined Function 생성
  - ✓ [예제] 절대값 UTIL_ABS 함수를 만드는데, INPUT 값으로 숫자만 들어온다고 가정한다.

[예제]

```
CREATE OR REPLACE Function UTIL_ABS
(v_input in  number)                    ..... ①
 return NUMBER
IS
 v_return number := 0;                  ..... ②
BEGIN
 if v_input < 0 then                    .....③
    v_return := v_input * -1;
 else
    v_return := v_input;
 end if;
 RETURN v_return;                       ..... ④
END;
/
```

① 숫자 값을 입력 받는다. 예제에서는 숫자 값만 입력된다고 가정한다.
② 리턴 값을 받아 줄 변수인 v_return를 선언한다.
③ 입력 값이 음수이면 -1을 곱하여 v_return 변수에 대입한다.
④ v_return 변수를 리턴한다.

- User Defined Function 활용
  - ✓ [예제] K-리그 8월 경기결과와 두 팀간의 점수차를 UTIL_ABS 함수를 사용하여 절대값으로 출력한다.

[예제]

```
SELECT SCHE_DATE 경기일자,
       HOMETEAM_ID || ' - ' || AWAYTEAM_ID 팀들,
       HOME_SCORE  || ' - ' || AWAY_SCORE SCORE,
       UTIL_ABS(HOME_SCORE - AWAY_SCORE) 점수차
FROM   SCHEDULE
WHERE  GUBUN = 'Y'
AND    SCHE_DATE BETWEEN '20120801' AND '20120831'
ORDER BY SCHE_DATE;
```

경기일자	팀들	SCORE	점수차
20120803	K01 - K03	3 - 0	3
20120803	K06 - K09	2 - 1	1
20120803	K08 - K07	1 - 0	1
...	...	...	...

- Trigger
  - ✓ Trigger란 특정한 테이블에 INSERT, UPDATE, DELETE와 같은 DML문이 수행되었을 때, 데이터베이스에서 자동으로 동작하도록 작성된 프로그램이다.
  - ✓ 사용자가 직접 호출하여 사용하는 것이 아니고, 생성 이후 조건이 맞으면 데이터베이스에서 자동적으로 수행하게 된다.
  - ✓ Trigger는 테이블과 뷰, 데이터베이스 작업을 대상으로 정의할 수 있으며, 전체 트랜잭션 작업에 대해 발생되는 Trigger와 각 행에 대해서 발생되는 Trigger가 있다.
  - ✓ 데이터웨어하우스의 실시간 집계성 테이블이나, Materialized View 생성시 사용할 수있다.
  - ✓ 데이터 무결성을 위해 FK(Foreign Key) 역할을 수행할 수도 있다.
  - ✓ Trigger는 데이터베이스 보안의 적용, 유효하지 않은 트랜잭션의 예방, 업무 규칙의 적용, 감사 제공 등에 사용될 수 있다.
  - ✓ OLTP성의 계정계, 기간계 시스템에서는 성능 부하를 발생할 위험이 있다.

- Trigger 생성
  - ✓ [예제] 먼저 관련 테이블을 생성한다.

주문정보 테이블 (ORDER_LIST)		
필드명	TYPE	길이
ORDER_DATE	CHAR	8
PRODUCT	VARCHAR2	10
QTY	NUMBER	
AMOUNT	NUMBER	

※ 테스트용으로 PK 누락함

일자별 판매집계 테이블 (SALES_PER_DATE)		
필드명	TYPE	길이
SALE_DATE	CHAR	8
PRODUCT	VARCHAR2	10
QTY	NUMBER	
AMOUNT	NUMBER	

※ 테스트용으로 PK 누락함

  - ✓ [예제] Trigger의 역할은 ORDER_LIST에 주문 정보가 입력되면 주문 정보의 주문 일자(ORDER_LIST.ORDER_DATE)와 주문 상품(ORDER_LIST.PRODUCT)을 기준으로 판매 집계 테이블(SALES_PER_DATE)에 해당 주문 일자의 주문 상품 레코드가 존재하면 판매 수량과 판매 금액을 더하고, 존재하지 않으면 새로운 레코드를 입력한다.

- Trigger 생성

```
CREATE OR REPLACE Trigger SUMMARY_SALES                    ..... ①
  AFTER INSERT
  ON    ORDER_LIST
  FOR   EACH ROW
DECLARE                                                    ..... ②
   o_date ORDER_LIST.order_date%TYPE;
   o_prod ORDER_LIST.product%TYPE;
BEGIN
   o_date  := :NEW.order_date;
   o_prod  := :NEW.product;
  UPDATE SALES_PER_DATE                                    ..... ③
     SET qty = qty + :NEW.qty,
         amount = amount + :NEW.amount
   WHERE sale_date = o_date
     AND product  = o_prod;
   if SQL%NOTFOUND then                                    ..... ④
      INSERT INTO SALES_PER_DATE
      VALUES(o_date, o_prod, :NEW.qty, :NEW.amount);
   end if;
END;
/
```


- Trigger 생성

① Trigger를 선언한다.
CREATE OR REPLACE Trigger SUMMARY_SALES : Trigger 선언문
AFTER INSERT : 레코드가 입력이 된 후 Trigger 발생
ON ORDER_LIST : ORDER_LIST 테이블에 Trigger 설정
FOR EACH ROW : 각 ROW마다 Trigger 적용

② o_date(주문일자), o_prod(주문상품) 값을 저장할 변수를 선언하고, 신규로 입력된 데이터를 저장한다.
: NEW는 신규로 입력된 레코드의 정보를 가지고 있는 구조체
: OLD는 수정, 삭제되기 전의 레코드를 가지고 있는 구조체

구분	:OLD	:NEW
INSERT	NULL	입력된 레코드 값
UPDATE	UPDATE되기 전의 레코드의 값	UPDATE된 후의 레코드 값
DELETE	레코드가 삭제되기 전 값	NULL

③ 먼저 입력된 주문 내역의 주문 일자와 주문 상품을 기준으로 SALES_PER_DATE 테이블에 업데이트한다.

④ 처리 결과가 SQL%NOTFOUND이면 해당 주문 일자의 주문 상품 실적이 존재하지 않으므로, SALES_PER_DATE 테이블에 새로운 집계 데이터를 입력한다.

- Trigger 활용
  - ✓ [예제] ORDER_LIST 테이블에 주문 정보를 입력한다.

```
SQL> INSERT INTO ORDER_LIST VALUES('20120901','MONOPACK',10,300000);
1개의 행이 만들어졌다.

SQL> COMMIT;
커밋이 완료되었다.

SQL> SELECT * FROM ORDER_LIST;
ORDER_DA PRODUCT            QTY      AMOUNT
-------- ---------- ---------- ----------
20120901 MONOPACK            10      300000

SQL> SELECT * FROM SALES_PER_DATE;
SALE_DAT PRODUCT            QTY      AMOUNT
-------- ---------- ---------- ----------
20120901 MONOPACK            10      300000
```


- Trigger 활용
  - ✓ [예제] 다시 한 번 주문 데이터를 입력하고, 두 테이블의 데이터를 확인한다.

```
SQL> INSERT INTO ORDER_LIST VALUES('20120901','MONOPACK',20,600000);
1개의 행이 만들어졌다.

SQL> COMMIT;
커밋이 완료되었다.

SQL> SELECT * FROM ORDER_LIST;
ORDER_DA PRODUCT            QTY      AMOUNT
-------- ---------- ---------- ----------
20120901 MONOPACK            10      300000
20120901 MONOPACK            20      600000

SQL> SELECT * FROM SALES_PER_DATE;
SALE_DAT PRODUCT            QTY      AMOUNT
-------- ---------- ---------- ----------
20120901 MONOPACK            30      900000
```

- 프로시저와 트리거의 차이점
  - ✓ 프로시저는 BEGIN ~ END 절 내에 COMMIT, ROLLBACK과 같은 트랜잭션 종료 명령어를 사용할 수 있지만, 데이터베이스 트리거는 BEGIN ~ END 절 내에 사용할 수 없다.
  - ✓ ROLLBACK을 하면 원 트랜잭션 뿐만 아니라 Trigger로 입력된 정보까지 하나의 트랜잭션으로 인식하여 두 테이블 모두 입력 취소가 된다. Trigger는 데이터베이스에 의해 자동 호출되지만 결국 INSERT, UPDATE, DELETE 문과 하나의 트랜잭션 안에서 일어나는 일련의 작업들이라 할 수 있다.

프로시저	트리거
CREATE Procedure 문법사용	CREATE Trigger 문법사용
EXECUTE 명령어로 실행	생성 후 자동으로 실행
COMMIT, ROLLBACK 실행 가능	COMMIT, ROLLBACK 실행 안됨

Structured Query Language

II. SQL 기본 및 활용

제2장 SQL 활용
제8절 절차형 SQL
핵심정리 및 연습문제

핵심정리 제8절 절차형 SQL

- 절차형 SQL을 이용하여 SQL 문장의 조건에 따른 분기 처리나 SQL 문장의 연속적인 실행을 이용하여 특정 기능을 수행하는 저장 모듈을 생성할 수 있다.
- 절차형 SQL을 이용하여 PROCEDURE, USER DEFINED FUNCTION , TRIGGER를 만들 수 있다.
- 절차형 SQL을 이용하여 SQL문의 연속적인 실행이나 조건에 따른 분기처리를 이용하는 PROCEDURE를 만들 수 있다.
- 앞에서 학습한 SUM, SUBSTR, NVL 등의 함수는 벤더에서 미리 만들어둔 내장 함수이고, 사용자가 별도로 만든 함수를 USER DEFINED FUNCTION이라고 한다.
- Trigger란 특정한 테이블에 INSERT, UPDATE, DELETE와 같은 DML 문이 수행되었을 때, 데이터베이스에서 자동으로 동작하도록 작성된 프로그램이다.

연습문제 제8절 절차형 SQL

문제. 다음 중 절차형 SQL을 이용하여 주로 만드는 것이 아닌 것은 무엇인가?

① PROCEDURE
② TRIGGER
③ USER DEFINED FUNCTION
④ BUILT-IN FUNCTION

문제. 다음 중 절차형 SQL을 이용하여 주로 만드는 것이 아닌 것은 무엇인가?

① PROCEDURE
② TRIGGER
③ USER DEFINED FUNCTION
④ BUILT-IN FUNCTION

정답 : ④
해설 :
절차형 SQL을 이용하여 PROCEDURE, TRIGGER, USER DEFINED FUNCTION을 만들 수 있다.

Structured Query Language

II. SQL 기본 및 활용

제3장 SQL 최적화 기본 원리 소개

- 제3장 SQL 최적화 기본 원리
  - ✓ 제1절 옵티마이저와 실행계획
  - ✓ 제2절 인덱스 기본
  - ✓ 제3절 조인 수행 원리

- 제3장 소개
  - ✓ SQL 최적화를 이해하기 위해서는 실행계획과 실행계획에 표시되는 사항 즉, 테이블 액세스 기법, 조인 기법, 조인 순서 등에 대한 이해가 필요하다.
  - ✓ 본 장에서는 최적의 실행계획을 생성하는 역할을 수행하는 옵티마이저와 테이블 액세스 기법으로서 가장 기본적인 두 가지 기법인 인덱스 스캔과 전체 테이블 스캔에 대해 설명한다.
  - ✓ 또한 NL(Nested Loops) Join, Hash Join, Sort Merge Join에 대한 조인 수행 원리에 대해 소개한다.

Structured Query Language

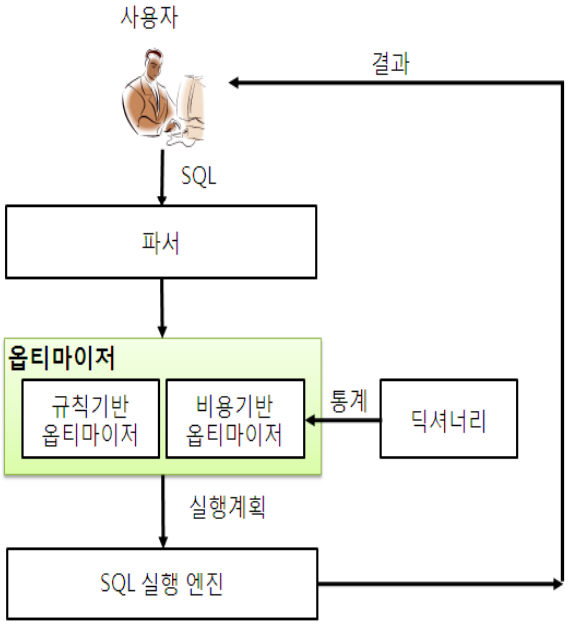
II. SQL 기본 및 활용

제3장 SQL 최적화 기본 원리
제1절 옵티마이저와 실행계획
학습하기

제1절 옵티마이저와 실행계획

- 옵티마이저
  - ✓ 옵티마이저(Optimizer)는 사용자가 질의한 SQL문에 대해 최적의 실행 방법을 결정하는 역할을 수행한다. 이러한 실행 방법을 실행계획(Execution Plan)이라고 한다.
  - ✓ JAVA, C등과 같은 프로그램 언어와는 달리 SQL은 사용자의 요구사항만 기술할 뿐 처리과정에 대한 기술은 하지 않는다. 사용자 요구사항을 만족하는 결과를 추출할 수 있는 다양한 실행 방법들 중에서 최적의 실행 방법을 결정하는 것이 바로 옵티마이저의 역할이다.
  - ✓ 옵티마이저가 선택한 실행 방법의 적절성 여부는 질의의 수행 속도에 가장 큰 영향 미치게 된다. 최적의 실행 방법 결정이라는 것은 어떤 방법으로 처리하는 것이 최소 일량으로 동일한 일을 처리할 수 있을지 결정하는 것이다.
  - ✓ 옵티마이저는 실제로 SQL문을 처리해보지 않은 상태에서 최적의 실행 방법을 결정해야 하는 어려움이 있다.

- 옵티마이저 종류



✓옵티마이저가 최적의 실행 방법을 결정하는 방식에 따라 규칙기반 옵티마이저(RBO, Rule-Based Optimizer)와 비용기반 옵티마이저(CBO, Cost-Based Optimizer)로 구분할 수 있다.

✓현재 대부분의 관계형 데이터베이스는 비용기반 옵티마이저만을 제공한다. 다만 하위 버전 호환성을 위해서만 규칙기반 옵티마이저 기능이 남아 있을 뿐이다.

- 규칙기반 옵티마이저(RBO)
  - ✓ 규칙기반 옵티마이저의 규칙은 보편 타당성에 근거한 것들이다. 이러한 규칙을 알고 있는 것은 옵티마이저의 최적화 작업을 이해하는데 도움이 된다. 아래는 일반적인 규칙 기반 옵티마이져의 15가지 규칙이다. 순위의 숫자가 낮을수록 높은 우선 순위이다.

순위	액세스 기법
1	Single row by rowid
2	Single row by cluster join
3	Single row by hash cluster key with unique or primary key
4	Single row by unique or primary key
5	Cluster join
6	Hash cluster key
7	Indexed cluster key
8	Composite index
9	Single column index
10	Bounded range search on indexed columns
11	Unbounded range search on indexed columns
12	Sort merge join
13	MAX or MIN of indexed column
14	ORDER BY on indexed column
15	Full table scan

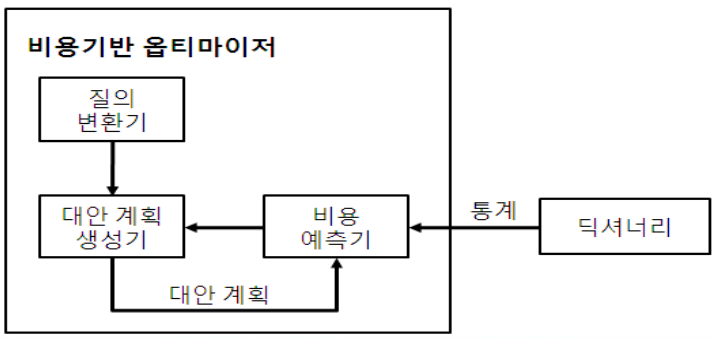
✓순위 2~14위까지는 인덱스를 활용하는 방법이다.

- RBO 우선 순위
  - ✓ 규칙 1. Single row by Rowid : ROWID를 통해서 테이블에서 하나의 행을 액세스하는 방식이다. ROWID는 행이 포함된 데이터 파일, 블록 등의 정보를 가지고 있기 때문에 다른 정보를 참조하지 않고도 바로 원하는 행을 액세스할 수 있다. 하나의 행을 액세스하는 가장 빠른 방법이다.
  - ✓ 규칙 4. Single row by unique or primary key : 유일 인덱스(Unique Index)를 통해서 하나의 행을 액세스하는 방식이다. 이 방식은 인덱스를 먼저 액세스하고 인덱스에 존재하는 ROWID를 추출하여 테이블의 행을 액세스한다.
  - ✓ 규칙 8. Composite index : 복합 인덱스에 동등('=' 연산자) 조건으로 검색하는 경우이다. 예를 들어, 만약 A+B 칼럼으로 복합 인덱스가 생성되어 있고, 조건절에서 WHERE A=10 AND B=1 형태로 검색하는 방식이다. 복합 인덱스 사이의 우선 순위 규칙은 다음과 같다. 인덱스 구성 칼럼의 개수가 더 많고 해당 인덱스의 모든 구성 칼럼에 대해 '='로 값이 주어질 수록 우선순위가 더 높다. (인덱스 매칭율)
    예를 들어, A+B로 구성된 인덱스와 A+B+C로 구성된 인덱스가 각각 존재하고 조건절에서 A, B, C 칼럼 모두에 대해 '='로 값이 주어진다면 A+B+C 인덱스가 우선 순위가 높다. 만약 조건절에서 A, B 칼럼에만 '='로 값이 주어진다면 A+B는 인덱스의 모든 구성 칼럼에 대해 값이 주어지고 A+B+C 인덱스 입장에서는 인덱스의 일부 칼럼에 대해서만 값이 주어졌기 때문에 A+B 인덱스가 우선 순위가 높게 된다.

- RBO 우선 순위
  - ✓ 규칙 9. Single column index : 단일 칼럼 인덱스에 '=' 조건으로 검색하는 경우이다. 만약 A 칼럼에 단일 칼럼 인덱스가 생성되어 있고, 조건절에서 A=10 형태로 검색하는 방식이다.
  - ✓ 규칙 10. Bounded range search on indexed columns : 인덱스가 생성되어 있는 칼럼에 양쪽 범위를 한정하는 형태로 검색하는 방식이다. 이러한 연산자에는 BETWEEN, LIKE 등이 있다. 만약 A 칼럼에 인덱스가 생성되어 있고, A BETWEEN '10' AND '20' 또는 A LIKE '1%' 형태로 검색하는 방식이다.
  - ✓ 규칙 11. Unbounded range search on indexed columns : 인덱스가 생성되어 있는 칼럼에 한쪽 범위만 한정하는 형태로 검색하는 방식이다. 이러한 연산자에는 >, >=, <, <= 등이 있습니다. 만약 A 칼럼에 인덱스가 생성되어 있고, A > '10' 또는 A < '20' 형태로 검색하는 방식이다.
  - ✓ 규칙 15. Full Table Scan : 전체 테이블을 액세스하면서 조건절에 주어진 조건을 만족하는 행만을 결과로 추출합니다. (일반적으로 속도가 느리지만,병렬 처리 가능함)

- 비용기반 옵티마이저(CBO)
  - ✓ RBO의 단순한 몇 개의 규칙만으로 현실의 모든 사항을 정확히 예측할 수는 없다. 비용기반 옵티마이저는 이러한 규칙기반 옵티마이저의 단점을 극복하기 위해서 출현하였다.
  - ✓ 비용기반 옵티마이저는 SQL문을 처리하는데 필요한 비용이 가장 적은 실행계획을 선택하는 방식이다. 여기서 비용이란 SQL문을 처리하기 위해 예상되는 소요시간 또는 자원 사용량을 의미한다.
  - ✓ 비용기반 옵티마이저는 비용을 예측하기 위해서 규칙기반 옵티마이저는 사용하지 않는 테이블, 인덱스, 칼럼 등의 다양한 객체 통계정보와 시스템 통계정보 등을 이용한다.
  - ✓ 통계정보가 없는 경우 비용기반 옵티마이저는 정확한 비용 예측이 불가능해져서 비효율적인 실행계획을 생성할 수 있다. 그렇기 때문에 정확한 통계정보를 유지하는 것은 비용기반 최적화에서 중요한 요소이다.

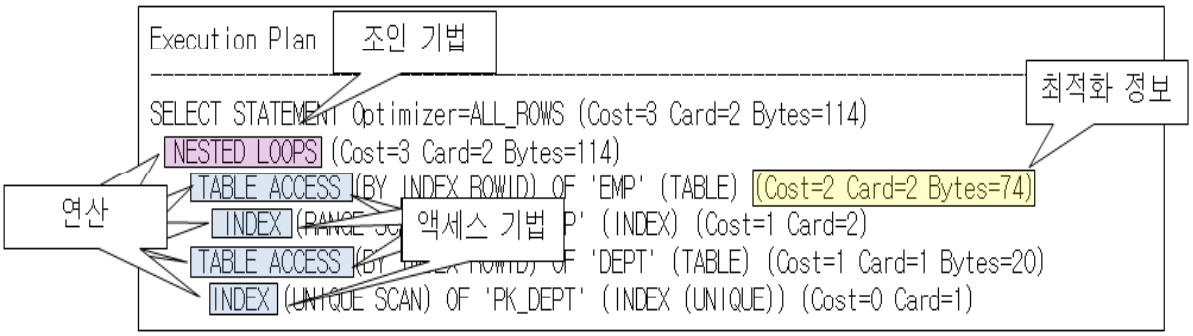
- CBO 옵티마이저
  - ✓ 비용기반 옵티마이저는 통계정보, DBMS 버전, DBMS 설정 정보 등의 차이로 인해 동일 SQL문도 서로 다른 실행계획이 생성될 수 있다. 또한 비용기반 옵티마이저의 다양한 한계들로 인해 실행계획의 예측 및 제어가 어렵다는 단점이 있다.
  - ✓ 비용기반 옵티마이저는 질의 변환기, 대안 계획 생성기, 비용 예측기 등의 모듈로 구성되어 있다.



- CBO 옵티마이저
  - ✓ 질의 변환기는 사용자가 작성한 SQL문을 작업하기가 보다 용이한 형태로 변환하는 모듈이다.
  - ✓ 대안 계획 생성기는 동일한 결과를 생성하는 다양한 대안 계획을 생성하는 모듈이다. 대안 계획의 생성이 너무 많아지면 최적화를 수행하는 시간이 그만큼 오래 걸린 수 있다. 그래서 대부분의 상용 옵티마이저들은 대안 계획의 수를 제약하는 다양한 방법을 사용한다. 이러한 현실적인 제약으로 인해 생성된 대안 계획들 중에서 최적의 대안 계획이 포함되지 않을 수도 있다.
  - ✓ 비용 예측기는 대안 계획 생성기에 의해서 생성된 대안 계획의 비용을 예측하는 모듈이다. 대안 계획의 정확한 비용을 예측하기 위해서 연산의 중간 집합의 크기 및 결과 집합의 크기, 분포도 등의 예측이 정확해야 한다. 보다 나은 예측을 위해 옵티마이저는 정확한 통계정보를 필요로 한다.

- 실행계획
  - ✓ 실행계획(Execution Plan)이란 SQL에서 요구한 사항을 처리하기 위한 절차와 방법을 의미한다. 실행계획을 생성한다는 것은 SQL을 어떤 순서로 어떻게 실행할 지를 결정하는 작업이다.
  - ✓ 동일한 SQL에 대해 결과를 낼 수 있는 다양한 처리 방법(실행계획)이 존재할 수 있지만 각 처리 방법마다 실행 시간(성능)은 서로 다를 수 있다. 옵티마이저는 다양한 처리 방법들 중에서 최적의 실행계획을 예측해서 생성해 준다.
  - ✓ 생성된 실행계획을 보는 방법은 데이터베이스 벤더마다 서로 다르다. 실행계획에서 표시되는 내용 및 형태도 약간씩 차이는 있지만 실행계획이 SQL 처리를 위한 절차와 방법을 의미한다는 기본적인 사항은 모두 동일하다.
  - ✓ 실행계획을 보고 SQL이 어떻게 실행되는지 정확히 이해할 수 있다면 보다 향상된 SQL 이해 및 활용을 할 수 있다. (SQL 튜닝의 출발)

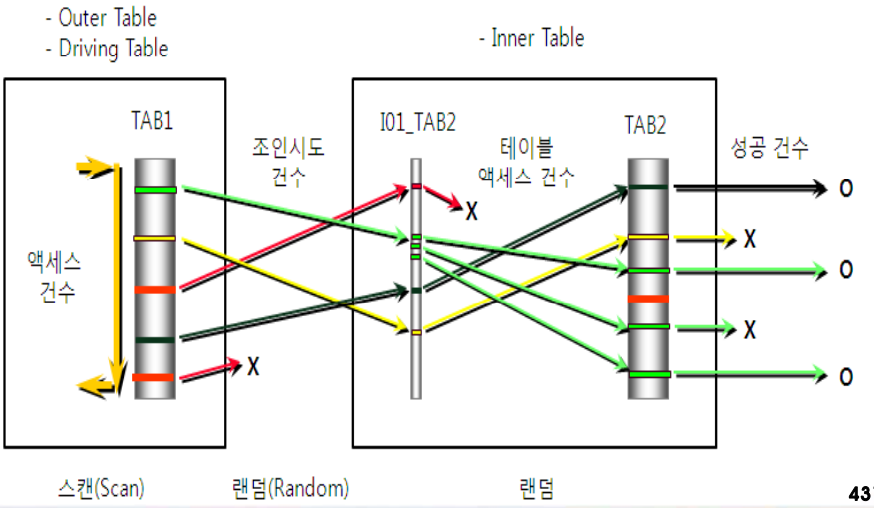
- **실행계획**
  - ✓ 실행계획을 구성하는 요소에는 조인 순서(Join Order), 조인 기법(Join Method), 액세스 기법(Access Method), 최적화 정보(Optimization Information), 연산(Operation) 등이 있다.
  - ✓ 최적화 정보는 실제로 SQL을 실행하고 얻은 결과가 아니라 통계 정보를 바탕으로 옵티마이저가 계산한 예상치이다. 만약 이러한 비용 사항이 실행계획에 표시되지 않았다면 이것은 규칙기반 최적화 방식으로 실행계획을 생성한 것이다.(아래는 오라클 사례)



- **실행계획**
  - ✓ 연산(Operation)은 여러 가지 조작을 통해서 원하는 결과를 얻어내는 일련의 작업이다. 연산에는 조인 기법(NL Join, Hash Join, Sort Merge Join 등), 액세스 기법(인덱스 스캔, 전체 테이블 스캔 등), 필터, 정렬, 집계, 뷰 등 다양하게 존재한다.
  - ✓ 조인 기법은 두 개의 테이블을 조인할 때 사용할 수 있는 방법으로서 여기에는 NL Join, Hash Join, Sort Merge Join 등이 있다. 그림에서 조인 기법은 NL Join을 사용하고 있다.
  - ✓ 그림에서 조인 순서는 EMP→DEPT이다. 논리적으로 가능한 조인 순서는 n! 만큼 존재한다. 여기서는 n은 FROM절에 존재하는 테이블 수이다. 그러나 현실적으로 옵티마이저가 적용 가능한 조인 순서는 이보다는 적거나 같다.
  - ✓ 액세스 기법은 하나의 테이블을 액세스할 때 사용할 수 있는 방법이다. 여기에는 인덱스를 이용하여 테이블을 액세스하는 인덱스 스캔(Index Scan)과 테이블 전체를 모두 읽으면서 조건을 만족하는 행을 찾는 전체 테이블 스캔(Full Table Scan) 등이 있다. 그림에서 액세스 기법은 인덱스 스캔을 사용하고 있다.
  - ✓ 최적화 정보에는 Cost, Card, Bytes가 있다. Cost는 상대적인 비용 정보이고 Card는 Cardinality의 약자로서 주어진 조건을 만족한 결과 집합 혹은 조인 조건을 만족한 결과 집합의 건수를 의미한다. Bytes는 결과 집합이 차지하는 메모리 양을 바이트로 표시한 것이다.

- SQL 처리 흐름도
  - ✓ SQL 처리 흐름도(Access Flow Diagram)란 SQL의 내부적인 처리 절차를 시각적으로 표현한 도표이다. 이것은 실행계획을 시각화한 것이다.
  - ✓ SQL 처리 흐름도에는 SQL문 처리를 위한 조인 순서, 테이블 액세스 기법과 조인 기법 등을 표현할 수 있다.

```
SELECT …
FROM   TAB1 A, TAB2 B
WHERE    A.KEY    = B.KEY
AND    A.COL1 = :조건1
AND    B.COL2 = :조건2
```



- SQL 처리 흐름도
  - ✓ 그림에서 조인 순서는 TAB1 → TAB2이다. 여기서 TAB1을 Outer Table 또는 Driving Table이라고 하고, TAB2를 Inner Table이라고 한다.
  - ✓ 테이블의 액세스 방법은 TAB1은 테이블 전체 스캔을 의미하고, TAB2는 I01_TAB2 이라는 인덱스를 통한 인덱스 스캔을 했음을 표시한 것이다.
  - ✓ 조인 방법은 NL Join을 수행했음을 표시한 것이다.
  - ✓ 그림에서 TAB1에 대한 액세스는 스캔(Scan) 방식이고 조인시도 및 I01_TAB2 인덱스를 통한 TAB2 액세스는 랜덤(Random) 방식이다.
  - ✓ 성능적인 관점을 살펴보기 위해서 SQL 처리 흐름도에 일량을 함께 표시할 수 있다.
  - ✓ 그림에서 건수(액세스 건수, 조인시도 건수, 테이블 액세스 건수, 성공 건수)라고 표시된 곳에 SQL 처리를 위해 작업한 건수 또는 처리 결과 건수 등의 일량을 함께 표시할 수 있다. 이것을 통해 어느 부분에서 비효율이 발생하고 있는지에 대한 힌트를 얻을 수 있다.

Structured Query Language

II. SQL 기본 및 활용

제3장 SQL 최적화 기본 원리
제1절 옵티마이저와 실행계획
핵심정리 및 연습문제

핵심정리 제1절 옵티마이저와 실행계획

- 사용자의 질의를 처리하는 다양한 실행 방법 중에서 최적의 실행 방법을 결정하는 것이 바로 옵티마이저의 역할이다.
- 옵티마이저는 규칙기반 옵티마이저와 비용기반 옵티마이저로 나눌 수 있다. 규칙기반 옵티마이저는 우선 규칙을 이용하여 실행계획을 생성하고 비용기반 옵티마이저는 사용자의 질의를 처리하는데 필요한 최소 비용이 드는 실행계획을 생성한다.
- 실행계획은 질의를 처리하기 위한 절차와 방법이 기술된다. 실행계획의 절차와 방법은 액세스 기법, 조인 기법, 조인 순서 등으로 표현된다.
- SQL 처리 흐름도는 실행계획을 시각화한 도표이다. 이것을 통해서 SQL의 처리 흐름을 보다 잘 이해할 수 있다.

문제. 옵티마이저에 대한 설명으로 적절하지 않는 것은 다음 중 무엇인가?

① 옵티마이저는 질의에 대해 실행계획을 생성한다.
② 비용기반 옵티마이저는 비용계산을 위해 다양한 통계정보를 사용한다.
③ 규칙기반 옵티마이저에서 제일 낮은 우선순위는 전체 테이블 스캔이다.
④ 비용기반 옵티마이저는 적절한 인덱스가 존재하면 반드시 인덱스를 사용한다.

연습문제 제1절 옵티마이저와 실행계획

문제. 옵티마이저에 대한 설명으로 적절하지 않는 것은 다음 중 무엇인가?

① 옵티마이저는 질의에 대해 실행계획을 생성한다.
② 비용기반 옵티마이저는 비용계산을 위해 다양한 통계정보를 사용한다.
③ 규칙기반 옵티마이저에서 제일 낮은 우선순위는 전체 테이블 스캔이다.
④ 비용기반 옵티마이저는 적절한 인덱스가 존재하면 반드시 인덱스를 사용한다.

정답 : ④
해설 :
비용기반 옵티마이저는 비용을 기반으로 최적화 작업을 수행한다. 따라서 인덱스 스캔보다 전체 테이블 스캔이 비용 낮다고 판단하면 적절한 인덱스가 존재하더라도 전체 테이블 스캔으로 SQL문을 수행할 수 있다.

문제. 실행계획에 대한 설명으로 적절하지 않는 것은 다음 중 무엇인가?
① 실행계획은 SQL문의 처리를 위한 절차와 방법이 표현된다.
② 실행계획은 액세스 기법, 조인 순서, 조인 방법 등으로 구성된다.
③ 실행계획이 다르면 결과도 달라질 수 있다.
④ 최적화 정보는 실행계획의 단계별 예상 비용을 표시한 것이다.

문제. 실행계획에 대한 설명으로 적절하지 않는 것은 다음 중 무엇인가?
① 실행계획은 SQL문의 처리를 위한 절차와 방법이 표현된다.
② 실행계획은 액세스 기법, 조인 순서, 조인 방법 등으로 구성된다.
③ 실행계획이 다르면 결과도 달라질 수 있다.
④ 최적화 정보는 실행계획의 단계별 예상 비용을 표시한 것이다.

정답 : ③
해설 :
동일 SQL문에 대해 실행계획이 다르다고 결과가 달라지지는 않는다. 그러나 실행계획의 차이로 성능이 달라질 수 있다.

문제. SQL 처리 흐름도에 대한 설명으로 적절하지 않는 것은 다음 중 무엇인가?

① 실행계획을 시각화한 것이다.
② 성능적인 측면의 표현은 고려하지 않는다.
③ 인덱스 스캔 및 전체 테이블 스캔 등의 액세스 기법을 표현할 수 있다.
④ SQL문의 처리 절차를 시각적으로 표현한 것이다.

연습문제 제1절 옵티마이저와 실행계획

문제. SQL 처리 흐름도에 대한 설명으로 적절하지 않는 것은 다음 중 무엇인가?

① 실행계획을 시각화한 것이다.
② 성능적인 측면의 표현은 고려하지 않는다.
③ 인덱스 스캔 및 전체 테이블 스캔 등의 액세스 기법을 표현할 수 있다.
④ SQL문의 처리 절차를 시각적으로 표현한 것이다.

정답 : ②
해설 :
SQL 처리 흐름도에서 성능적인 측면도 표현할 수 있다. 일량적인 측면의 표현과 인덱스 스캔 또는 테이블 스캔 등을 표현할 수 있다.

Structured Query Language

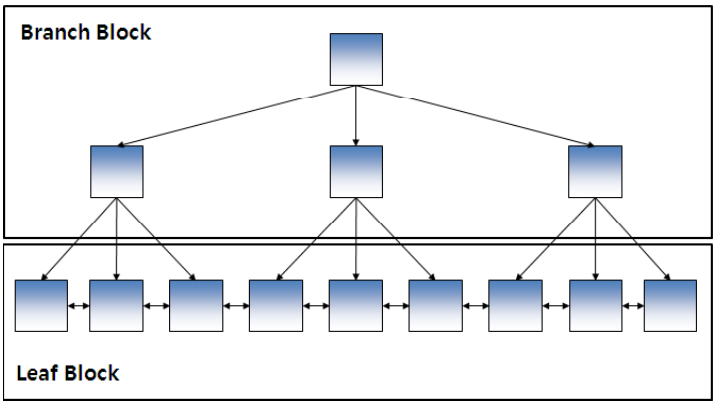
II. SQL 기본 및 활용

제3장 SQL 최적화 기본 원리
제2절 인덱스 기본 학습하기

제2절 인덱스 기본

- **인덱스 특징**
  - ✓ 인덱스는 원하는 데이터를 쉽게 찾을 수 있도록 돕는 책의 찾아보기와 유사한 개념이다. 인덱스는 테이블을 기반으로 선택적으로 생성할 수 있는 구조이다.
  - ✓ 테이블에 인덱스를 생성하지 않아도 되고 여러 개를 생성해도 된다.
  - ✓ 인덱스의 기본적인 목적은 검색 성능의 최적화이다. 즉, 검색 조건을 만족하는 데이터를 인덱스를 통해 효과적으로 찾을 수 있도록 돕는다.
  - ✓ 그렇지만 Insert, Update, Delete 등과 같은 DML 작업은 테이블과 인덱스를 함께 변경해야 하기 때문에 오히려 느려질 수 있다는 단점이 존재한다.
  - ✓ 인덱스의 칼럼 순서는 질의 성능에 중요한 영향을 미치는 요소이다.
  - ✓ 트리 기반 인덱스에는 B트리 인덱스 외에도 비트맵 인덱스(Bitmap Index), 리버스 키 인덱스(Reverse Key Index), 함수기반 인덱스(FBI, Function-Based Index) 등이 존재한다.

- B트리 기반 인덱스



✓ 관계형 DBMS에서 가장 일반적인 인덱스는 B트리 인덱스이다. B의 약자는 여러 의견이 있지만, Balance 의견이 가장 유력하다.

✓ 벤더별로 다양한 B트리(B+,B-,B*) 인덱스를 활용하는데 그림은 B+트리 인덱스에 대한 설명이다.

- B트리 기반 인덱스

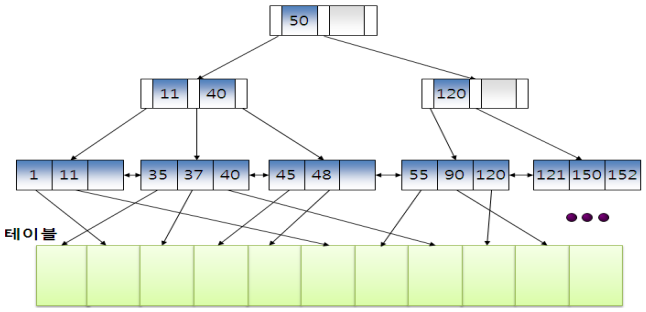
✓ B트리 인덱스는 브랜치 블록(Branch Block)과 리프 블록(Leaf Block)으로 구성된다. 브랜치 블록 중에서 가장 상위에서 있는 블록을 루트 블록(Root Block)이라고 한다. 브랜치 블록은 분기를 목적으로 하는 블록이다. 브랜치 블록은 다음 단계의 블록을 가리키는 포인터를 가지고 있다. 리프 블록은 트리의 가장 아래 단계에 존재한다.

✓ 리프 블록은 인덱스를 구성하는 칼럼의 데이터와 해당 데이터를 가지고 있는 행의 위치를 가리키는 레코드 식별자(RID, Record Identifier/Rowid)로 구성되어 있다. 인덱스 데이터는 인덱스를 구성하는 칼럼의 값으로 정렬된다. 만약 인덱스 데이터의 값이 동일하면 레코드 식별자의 순서로 저장된다.

✓ 리프 블록은 양방향 링크(Double Link)를 가지고 있다. 이것을 통해서 오름 차순(Ascending Order)과 내림 차순(Descending Order) 검색을 쉽게 할 수 있다.

✓ B트리 인덱스는 '=' 로 검색하는 일치(Exact Match) 검색과 'BETWEEN', '>' 등과 같은 연산자로 검색하는 범위(Range) 검색 모두에 적합한 구조이다.

- B트리 기반 인덱스



✓ 그림에서 37을 찾고자 한다면 루트 블록에서 50보다 작으므로 왼쪽 포인터로 이동한다. 37는 왼쪽 브랜치 블록의 11과 40 사이의 값이므로 가운데 포인터로 이동한다. 이동한 결과 해당 블록이 리프 블록이므로 37이 블록 내에 존재하는지 검색한다.

✓ 만약, 37과 50사이의 모든 값을 찾고자 한다면(BETWEEN 37 AND 50) 위와 동일한 방법으로 리프 블록에서 37를 찾고 50보다 큰 값을 만날 때까지 오른쪽으로 이동하면서 인덱스를 읽는다. 이것은 인덱스 데이터가 정렬되어 있고 리프 블록이 양방향 링크로 연결되어 있기 때문에 가능하다.

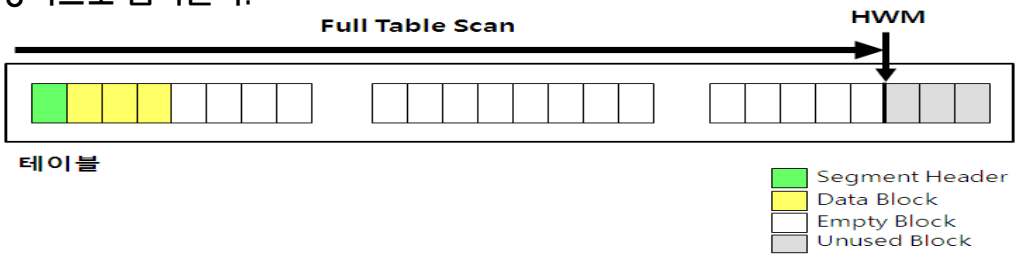
- SQL Server 클러스터형 인덱스

✓ SQL Server의 인덱스 종류는 저장 구조에 따라 클러스터형(clustered) 인덱스와 넌클러스터형(nonclustered) 인덱스로 나뉜다

✓ 그림은 Employee ID에 기반한 클러스터형 인덱스를 생성한 모습이다. 편의상, 삼각형(△) 모양의 B-트리 구조를 왼쪽으로 90도 돌려서 나타냈다

EmployeeID	LastName	FirstName	HireDate
1	Davolio	Nancy	1992-05-01 00:00:00.000
2	Fuller	Andrew	1992-08-14 00:00:00.000
3	Leverling	Janet	1992-04-01 00:00:00.000
4	Peacock	Margaret	1993-05-03 00:00:00.000
5	Buchanan	Steven	1993-10-17 00:00:00.000
6	Suyama	Michael	1993-10-17 00:00:00.000
7	King	Robert	1994-01-02 00:00:00.000
8	Callahan	Laura	1994-03-05 00:00:00.000
9	Dodsworth	Anne	1994-11-15 00:00:00.000

- **전체 테이블 스캔**
  - ✓ 전체 테이블 스캔(FTS) 방식으로 데이터를 검색한다는 것은 테이블에 존재하는 모든 데이터를 읽어 가면서 조건에 맞으면 결과로서 추출하고 조건에 맞지 않으면 버리는 방식으로 검색한다.



  - ✓ 오라클의 경우 그림과 같이 검색 조건에 맞는 데이터를 찾기 위해서 테이블의 고수위 마크(HWM, High Water Mark) 아래의 모든 블록을 읽는다. 고수위 마크는 테이블에 데이터가 쓰여졌던 블록 상의 최상위 위치(현재는 지워져서 데이터가 존재하지 않을 수도 있음)를 의미한다. 테이블 관리를 위해 HWM 사용함.
  - ✓ 전체 테이블 스캔 방식으로 데이터를 검색할 때 고수위 마크까지의 블록 내 모든 데이터를 읽어야 하기 때문에 모든 결과를 찾을 때까지 시간이 오래 걸릴 수 있다.

- **전체 테이블 스캔**

■ 옵티마이저가 연산으로서 FTS 방식을 선택하는 이유는 일반적으로 다음과 같다.

1) SQL문에 조건이 존재하지 않는 경우
SQL문에 조건이 존재하지 않는다는 것은 테이블에 존재하는 모든 데이터가 답이 된다는 것이다.

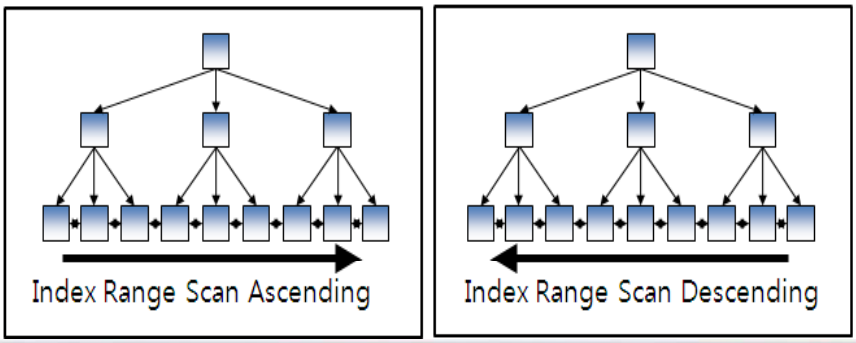
2) SQL문에 주어진 조건에 사용 가능한 인덱스가 존재하는 않는 경우
사용 가능한 인덱스가 존재하지 않는다면 데이터를 액세스할 수 있는 방법은 테이블의 모든 데이터를 읽으면서 주어진 조건을 만족하는지를 검사하는 방법뿐이다.
또한 주어진 조건에 사용 가능한 인덱스는 존재하나 함수를 사용하여 인덱스 칼럼을 변형한 경우에도 인덱스를 사용할 수 없다.

3) 옵티마이저의 취사 선택
조건을 만족하는 데이터가 많은 경우, 결과를 추출하기 위해서 테이블 대부분의 블록을 액세스해야 한다고 옵티마이저가 판단하면 조건에 사용 가능한 인덱스가 존재해도 전체 테이블 스캔 방식으로 읽을 수 있다.

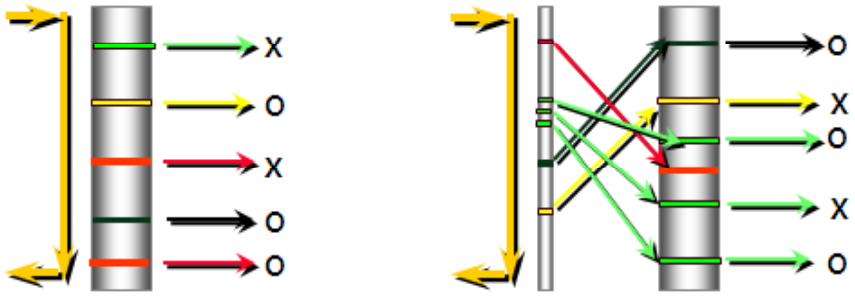
4) 그 밖의 경우
병렬처리 방식으로 처리하는 경우 또는 전체 테이블 스캔 방식의 힌트를 사용한 경우에 전체 테이블 스캔 방식으로 데이터를 읽을 수 있다.

- **인덱스 스캔**
  - ✓ 인덱스 스캔은 인덱스를 구성하는 칼럼의 값을 기반으로 데이터를 추출하는 액세스 기법이다. 인덱스의 리프 블록은 인덱스 구성하는 칼럼과 레코드 식별자로 구성되어 있다. 따라서 검색을 위해 인덱스의 리프 블록을 읽으면 인덱스 구성 칼럼의 값과 테이블의 레코드 식별자를 알 수 있다.
  - ✓ 인덱스에 존재하지 않는 칼럼의 값이 필요한 경우에는 현재 읽은 레코드 식별자를 이용하여 테이블을 액세스해야 한다. 그러나 SQL문에서 필요로 하는 모든 칼럼이 인덱스 구성 칼럼에 포함된 경우 테이블에 대한 액세스는 발생하지 않는다.
  - ✓ 인덱스는 인덱스 구성 칼럼의 순서로 정렬되어 있다. 인덱스의 구성 칼럼이 A+B라면 먼저 칼럼 A로 정렬되고 칼럼 A의 값이 동일할 경우에는 칼럼 B로 정렬된다. 그리고 칼럼 B까지 모두 동일하면 레코드 식별자로 정렬된다. 인덱스가 구성 칼럼으로 정렬되어 있기 때문에 인덱스를 경유하여 데이터를 읽으면 그 결과 또한 정렬되어 반환된다. 따라서 인덱스의 순서와 동일한 정렬 순서를 사용자가 원하는 경우에는 정렬 작업을 하지 않을 수 있다.
  - ✓ 인덱스 유일 스캔(Index Unique Scan), 인덱스 범위 스캔(Index Range Scan) 외에도 인덱스 전체 스캔(Index Full Scan), 인덱스 고속 전체 스캔(Fast Full Index Scan), 인덱스 스킵 스캔(Index Skip Scan) 등이 존재한다.

- **인덱스 스캔**
  - ✓ 인덱스 유일 스캔은 유일 인덱스(Unique Index)를 사용하여 단 하나의 데이터를 추출하는 방식이다. 유일 인덱스는 중복을 허락하지 않는 인덱스이다. 유일 인덱스 구성 칼럼에 모두 '='로 값이 주어지면 결과는 최대 1건이 된다. 인덱스 유일 스캔은 유일 인덱스 구성(복합) 칼럼에 대해 모두 '='로 값이 주어진 경우에만 가능한 인덱스 스캔 방식이다.
  - ✓ 인덱스 범위 스캔은 인덱스를 이용하여 한 건 이상의 데이터를 추출하는 방식이다. 유일 인덱스의 구성 칼럼 모두에 대해 '='로 값이 주어지지 않은 경우와 비유일 인덱스(Non-Unique Index)를 이용하는 모든 액세스 방식은 인덱스 범위 스캔 방식으로 데이터를 액세스하는 것이다. 인덱스 역순 범위 스캔은 내림 차순으로 데이터를 읽는 방식이다.



- **전체 테이블 스캔과 인덱스 스캔 방식 비교**
  - ✓ 데이터를 액세스하는 방법은 크게 두 가지로 나눠 볼 수 있다. 인덱스를 경유해서 읽는 인덱스 스캔 방식과 테이블의 전체 데이터를 모두 읽으면서 데이터를 추출하는 전체 테이블 스캔 방식이다.
  - ✓ 인덱스 스캔 방식은 사용 가능한 적절한 인덱스가 존재할 때만 이용할 수 있는 스캔 방식이지만, 전체 테이블 스캔 방식은 인덱스의 존재 유무와 상관없이 항상 이용 가능한 스캔 방식이다. 옵티마이저는 인덱스가 존재하더라도 전체 테이블 스캔 방식을 취사 선택할 수 있다.



전체 테이블 스캔 인덱스 스캔

- **전체 테이블 스캔과 인덱스 스캔 방식 비교**
  - ✓ 대용량 데이터 중에서 극히 일부의 데이터를 찾을 때, 인덱스 스캔 방식은 인덱스를 이용해 몇 번의 I/O만으로 원하는 데이터를 쉽게 찾을 수 있다. 그러나 전체 테이블 스캔은 테이블의 모든 데이터를 읽으면서 원하는 데이터를 찾아야 하기 때문에 비효율적인 검색을 하게 된다.
  - ✓ 반대로 테이블의 대부분의 데이터를 찾을 때는 한 블록씩 읽는 인덱스 스캔 방식 보다는 어차피 대부분의 데이터를 읽을 거라면 한번에 여러 블록씩 읽는 전체 테이블 스캔 방식이 유리할 수 있다.
  - ✓ 인덱스 스캔 방식에서는 불필요하게 다른 블록을 더 읽을 필요가 없다. 따라서 한번의 I/O 요청에 한 블록씩 데이터를 읽는다.
  - ✓ 그러나 전체 테이블 스캔은 데이터를 읽을 때 한번의 I/O 요청으로 여러 블록을 한꺼번에 읽는다. 어차피 테이블의 모든 데이터를 읽을 것이라면 한 번 읽기 작업을 할 때 여러 블록을 함께 읽는 것이 효율적이다.

Structured Query Language

II. SQL 기본 및 활용

제3장 SQL 최적화 기본 원리
제2절 인덱스 기본
핵심정리 및 연습문제

핵심정리 제2절 인덱스 기본

- 인덱스의 목적은 검색 성능의 최적화이다. 검색 조건을 만족하는 데이터를 인덱스를 통해 효과적으로 검색한다. 그러나 Insert, Update, Delete 등과 같은 DML 작업에 대해서는 테이블과 인덱스를 함께 변경해야 하기 때문에 성능이 떨어진다.
- B 트리 인덱스는 일치(Exact Match) 검색, 범위(Range) 검색 모두에 적합한 구조이다.
- 데이터를 액세스하는 방법은 인덱스 스캔과 전체 테이블 스캔 방식이 있다. 인덱스 스캔은 인덱스를 경유해서 테이블을 읽는 방식이고 전체 테이블 스캔은 테이블의 전체 데이터를 모두 읽으면서 데이터를 추출하는 방식이다.
- 인덱스 스캔 방식은 사용 가능한 적절한 인덱스가 존재할 때만 이용할 수 있는 스캔 방식이지만 전체 테이블 스캔 방식은 인덱스의 존재 유무와 상관없이 항상 이용 가능한 스캔 방식이다.

문제. 다음 설명 중 적절한 것은 무엇인가?

① 인덱스는 인덱스 구성 칼럼으로 항상 오름차순으로 정렬된다.
② 비용기반 옵티마이저는 인덱스 스캔이 항상 유리하다고 판단한다.
③ 규칙기반 옵티마이저는 적절한 인덱스가 존재하면 항상 인덱스를 사용하려고 한다.
④ 인덱스 범위 스캔은 항상 여러 건의 결과가 반환된다.

문제. 다음 설명 중 적절한 것은 무엇인가?

① 인덱스는 인덱스 구성 칼럼으로 항상 오름차순으로 정렬된다.
② 비용기반 옵티마이저는 인덱스 스캔이 항상 유리하다고 판단한다.
③ 규칙기반 옵티마이저는 적절한 인덱스가 존재하면 항상 인덱스를 사용하려고 한다.
④ 인덱스 범위 스캔은 항상 여러 건의 결과가 반환된다.

정답 : ③
해설 :
① 인덱스는 내림차순으로 생성되면 내림차순으로 정렬된다.
② 비용적인 측면에서는 전체 테이블 스캔이 유리할 수 있다.
③ 규칙기반 옵티마이저의 규칙에 따라 적절한 인덱스가 존재하면 전체 테이블 스캔보다는 항상 인덱스를 사용하려고 한다.
④ 인덱스 범위 스캔은 결과 건수만큼 반환한다. 결과가 없으면 한 건도 반환하지 않을 수 있다.

Structured Query Language

II. SQL 기본 및 활용

제3장 SQL 최적화 기본 원리
제3절 조인 수행 원리
학습하기

제3절 조인 수행 원리

- 조인
  - ✓ 조인이란 두 개 이상의 테이블을 하나의 집합으로 만드는 연산이다. SQL문에서 FROM 절에 두 개 이상의 테이블이 나열될 경우 조인이 수행된다.
  - ✓ 조인 연산은 두 테이블(집합) 사이에서 수행된다. FROM 절에 A, B, C라는 세 개의 테이블이 존재하더라도 세 개의 테이블이 동시에 조인이 수행되는 것은 아니다. 세 개의 테이블 중에서 먼저 두 개의 테이블에 대해 조인이 수행된다. 그리고 먼저 수행된 조인 결과와 나머지 테이블 사이에서 조인이 수행된다.이러한 작업은 FROM 절에 나열된 모든 테이블을 조인할 때까지 반복 수행한다.
  - ✓ 테이블 또는 조인 결과를 이용하여 조인을 수행할 때 조인 단계별로 다른 조인 기법을 사용할 수 있다. 예를 들어, A와 C테이블을 조인할 때는 NL Join 기법을 사용하고 해당 조인 결과와 B테이블을 조인할 때는 Hash Join 기법을 사용할 수 있다.

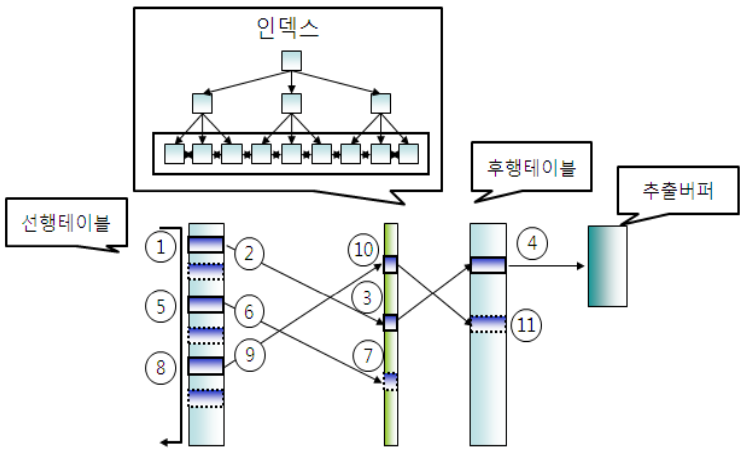
- 튜닝 Big3 조인
  - ✓ 조인은 여러 기준으로 분류할 수 있는데, 사용 빈도나 성능 관점에서는 NL Join, Sort Merge Join, Hash Join 3개의 조인이 가장 중요합니다.
  - ✓ Nested Loop 조인은 주로 랜덤액세스하며 소량의 업무 처리에 적합합니다. 연결 조건의 인덱스에 따라 성능이 좌우되며 드라이빙 테이블의 우선순위가 성능에 큰 영향을 미칩니다. Nested Loop 조인은 소규모 데이터를 인덱스를 사용하여 작업하는 OLTP 업무에 적당합니다.
  - ✓ Sort Merge 조인은 양쪽 테이블의 처리범위를 각자 엑세스하여 정렬한 결과를 차례로 스캔하면서 순차비교 함으로 인덱스를 사용하지 않습니다. 정렬 비용이 필요합니다. 드라이빙 테이블 의미가 없으며, Non-Equi 조인 가능.
  - ✓ Hash 조인은 Equi(=) 조인이 필요한 경우 Hash 함수를 사용하며 Sort 비용이 필요치 않으므로 좋은 효과를 볼 수 있습니다. 일반적으로 조인되는 두 집합의 크기가 차이가 나는 경우나 하드웨어 자원이 부족하지 않을 경우에는 성능이 좋으므로 SMJ 조인을 많이 대체하고 있습니다.

- Nested Loop 조인
  - ✓ NL Join은 프로그래밍에서 사용하는 중첩된 반복문과 유사한 방식으로 조인을 수행한다. 반복문의 외부에 있는 테이블을 선행 테이블 또는 외부 테이블(Outer Table)이라고 하고, 반복문의 내부에 있는 테이블을 후행 테이블 또는 내부 테이블(Inner Table)이라고 한다.

```
FOR 선행 테이블 읽음 ➔ 외부 테이블(Outer Table)
    FOR 후행 테이블 읽음 ➔ 내부 테이블(Inner Table)
        (선행 테이블과 후행 테이블 조인)
```

  - ✓ 먼저 선행 테이블의 조건을 만족하는 행을 추출하여 후행 테이블을 읽으면서 조인을 수행한다. 이 작업은 선행 테이블의 조건을 만족하는 모든 행의 수만큼 반복 수행한다.
  - ✓ NL Join에서는 선행 테이블의 조건을 만족하는 행의 수가 많으면(처리 주관 범위가 넓으면), 그 만큼 후행 테이블의 조인 작업은 반복 수행된다. 따라서 결과 행의 수가 적은(처리 주관 범위가 좁은) 테이블을 조인 순서상 선행 테이블로 선택하는 것이 전체 일량을 줄일 수 있다.
  - ✓ NL Join은 랜덤 방식으로 데이터를 액세스하기 때문에 처리 범위가 좁은 것이 유리하다.

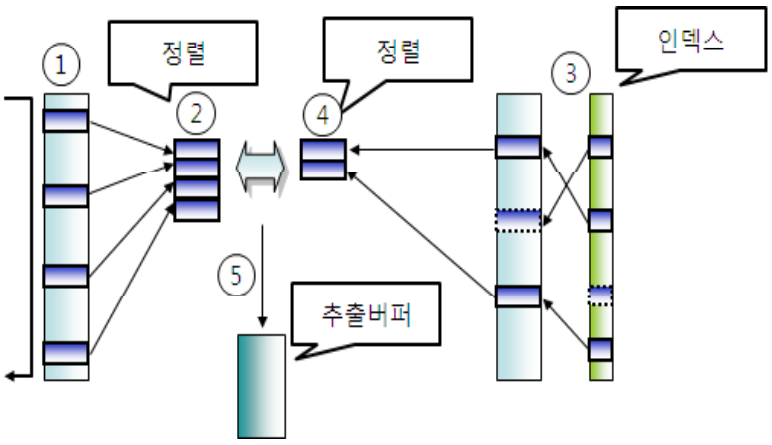
- Nested Loop 조인 작업 방법
  - ✓ 선행 테이블에서 주어진 조건을 만족하는 행을 찾음
  - ✓ 선행 테이블의 조인 키 값을 가지고 후행 테이블에서 조인 수행
  - ✓ 선행 테이블의 조건을 만족하는 모든 행에 대해 반복 작업 수행



✓만약 선행 테이블에 사용 가능한 인덱스가 존재한다면 인덱스를 통해 선행 테이블을 액세스할 수 있다. (여기서는 사용할 인덱스가 없음을 가정으로 설명한 것임)
✓NL Join 기법은 조인이 성공하면 바로 조인 결과를 사용자에게 보여 줄 수 있다.(추출버퍼 활용) 그래서 결과를 가능한 빨리 화면에 보여줘야 하는 온라인 프로그램에 적당한 조인 기법이다.
✓추출버퍼는 운반단위, Array Size, Prefetch Size라고도 한다

- Sort Merge Join
  - ✓ Sort Merge Join은 조인 칼럼을 기준으로 데이터를 정렬하여 조인을 수행한다. NL Join은 주로 랜덤 액세스 방식으로 데이터를 읽는 반면 Sort Merge Join은 주로 스캔 방식으로 데이터를 읽는다.
  - ✓ Sort Merge Join은 랜덤 액세스로 NL Join에서 부담이 되던 넓은 범위의 데이터를 처리할 때 이용되던 조인 기법이다. 그러나 Sort Merge Join은 정렬할 데이터가 많아 메모리에서 모든 정렬 작업을 수행하기 어려운 경우에는 임시 영역(디스크)을 사용하기 때문에 성능이 떨어질 수 있다.
  - ✓ 일반적으로 대량의 조인 작업에서 정렬 작업을 필요로 하는 Sort Merge Join 보다는 CPU 작업 위주로 처리하는 Hash Join이 성능상 유리하다. 그러나 Sort Merge Join은 Hash Join과는 달리 동등 조인 뿐만 아니라 비동등 조인에 대해서도 조인 작업이 가능하다는 장점이 있다.

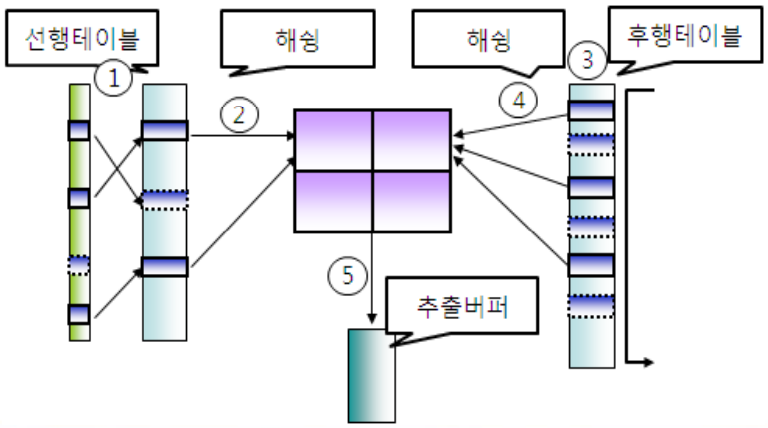
- Sort Merge Join 작업 방법
  - ✓ 독립적으로 각각의 테이블에서 데이터를 추출할 수 있다.
  - ✓ 독립적으로 각각의 테이블에서 조인키를 기준으로 정렬 작업 수행
  - ✓ 정렬 결과를 이용하여 조인을 수행하며 조인에 성공하면 추출버퍼에 넣음



✓Sort Merge Join은 조인 칼럼의 인덱스를 사용하지 않기 때문에 조인 칼럼의 인덱스가 존재하지 않을 경우에도 사용할 수 있는 조인 기법이다.

- Hash Join
  - ✓ Hash Join은 해쉬 기법을 이용하여 조인을 수행한다. 조인을 수행할 테이블의 조인 칼럼을 기준으로 해쉬 함수를 수행하여 서로 동일한 해쉬 값을 갖는 것들 사이에서 실제 값이 같은지를 비교하면서 조인을 수행한다.
  - ✓ Hash Join은 NL Join의 랜덤 액세스 문제점과 Sort Merge Join의 문제점인 정렬 작업의 부담을 해결 위한 대안으로 등장하였다.
  - ✓ Hash Join은 조인 칼럼의 인덱스를 사용하지 않기 때문에 조인 칼럼의 인덱스가 존재하지 않을 경우에도 사용할 수 있는 조인 기법이다. Hash Join은 해쉬 함수를 이용하여 조인을 수행하기 때문에 '=' 로 수행하는 조인 즉, 동등 조인에서만 사용할 수 있다.
  - ✓ Hash Join은 조인 작업을 수행하기 위해 해쉬 테이블을 메모리에 생성해야 한다. 생성된 해쉬 테이블의 크기가 메모리에 적재할 수 있는 크기보다 더 커지면 임시 영역(디스크)에 해쉬 테이블을 저장하면서 성능이 떨어진다. 그렇기 때문에 Hash Join을 할 때는 결과 행의 수가 적은 테이블을 선행 테이블로 사용하는 것이 좋다.

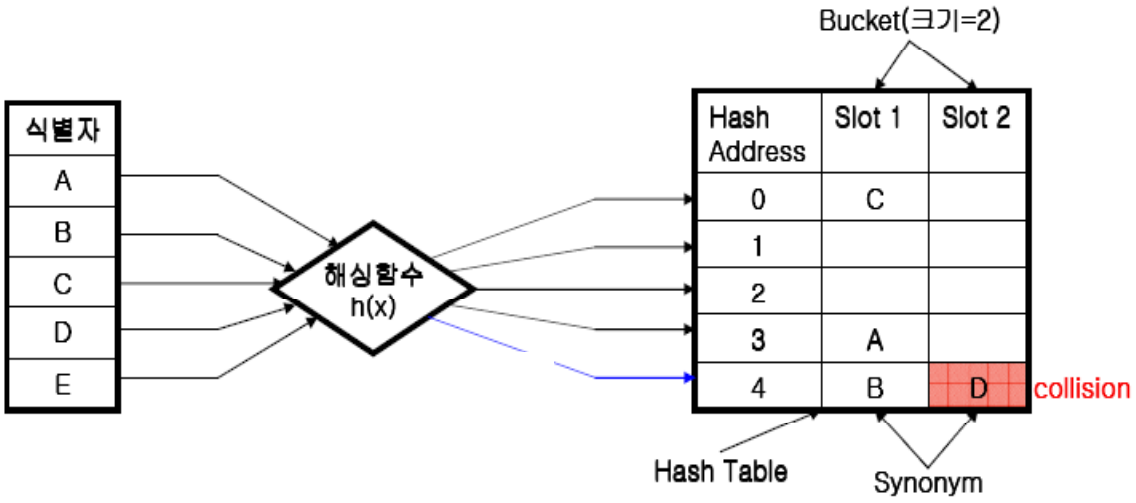
- Hash Join 작업 방법
  - ✓ 선행 테이블의 조인 키를 기준으로 해쉬 함수를 적용하여 모든 데이터에 대해 해쉬 테이블 생성 (조인 칼럼과 SELECT절에서 필요로 하는 칼럼도 함께 저장)
  - ✓ 후행 테이블의 조인 키를 기준으로 해쉬 함수를 적용하여 해당 버킷을 찾음
  - ✓ (조인 키를 이용해서 실제 조인될 데이터를 찾음, 조인에 성공하면 추출버퍼에 넣음)
  - ✓ 후행 테이블의 조건을 만족하는 모든 행에 대해서 반복 수행



✓Hash Join에서는 선행 테이블을 이용하여 먼저 해쉬 테이블을 생성한다고 해서 선행 테이블을 Build Input이라고도 하며,

✓후행 테이블은 만들어진 해쉬 테이블에 대해 해쉬 값의 존재를 검사 작업을 한다고 해서 Prove Input 이라고도 한다.

- 해싱 함수 (교재 외)



- 해싱 함수 (교재 외)
  - ✓ 해싱 함수
    1) 키 값으로부터 레코드의 물리적 주소로 사상 시키는 사상 함수
    2) 조건 : 충돌이 적게 발생하는 함수가 좋은 함수
    3) 해싱 함수에 따라 성능에 영향을 미친다
  - ✓ 해쉬 키
    – 해싱 함수가 레코드 주소를 계산하기 위해 사용하는 레코드의 키 값
  - ✓ 해쉬 어드레스
    – 해쉬 키를 이용하여 계산된 주소 임
  - ✓ 버킷
    1) 하나의 주소를 가지면서 하나 이상의 레코드를 저장할 수 있는 파일의 한 구역
    2) 여러 개의 슬롯(slot)으로 구성
  - ✓ 충돌(collision)
    – 서로 다른 레코드들이 같은 주소로 변환되는 경우
  - ✓ 슬롯(slot) : 한 개의 레코드를 저장할 수 있는 공간
  - ✓ Overflow : 빈 버킷이 없는 상태에 해쉬 주소가 다시 지정된 상태

Structured Query Language

II. SQL 기본 및 활용

제3장 SQL 최적화 기본 원리
제3절 조인 수행 원리
핵심정리 및 연습문제

핵심정리 제3절 조인 수행 원리

- SQL 문장에서 FROM 절에 두 개 이상의 테이블이 나열될 경우 조인이 수행된다.
- FROM 절에 세 개의 테이블이 존재하더라도 세 개의 테이블이 동시에 조인이 수행되는 것이 아니라 세 개의 테이블 중에서 먼저 두 개의 테이블에 대해 조인이 수행되고 먼저 수행된 조인 결과와 나머지 테이블 사이에서 조인이 수행된다.
- 조인을 수행할 때 각각의 조인별로 다른 조인 기법을 사용할 수 있다.
- NL Join은 중첩된 반복문과 유사한 조인 방식이다.
- Hash Join은 해쉬 기법을 이용한 조인 방식이다.
- Sort Merge Join은 데이터를 정렬을 이용한 조인 방식이다.

연습문제 제3절 조인 수행 원리

문제. 조인에 대한 설명으로 적절하지 않는 것은 다음 중 무엇인가?

① FROM절에 나열된 모든 테이블이 동시에 조인 작업이 수행된다.
② NL Join은 중첩된 반복문과 유사한 형식이다.
③ NL Join은 선행 테이블의 조건을 만족하는 건수만큼 반복 수행된다.
④ Hash Join은 작은 테이블을 선행 테이블로 사용하는 것이 성능관점에서 좋다.

문제. 조인에 대한 설명으로 적절하지 않는 것은 다음 중 무엇인가?

① FROM절에 나열된 모든 테이블이 동시에 조인 작업이 수행된다.

② NL Join은 중첩된 반복문과 유사한 형식이다.

③ NL Join은 선행 테이블의 조건을 만족하는 건수만큼 반복 수행된다.

④ Hash Join은 작은 테이블을 선행 테이블로 사용하는 것이 성능관점에서 좋다.

정답 : ①

해설 :

FROM절 아무리 많은 테이블이 나열되더라도 항상 2개씩 조인되다. 테이블과 테이블 사이 또는 앞에서 이미 수행된 조인의 결과 집합과 테이블, 조인 결과와 조인 결과 사이에서 조인이 처리된다.