

# 01.관계형 데이터베이스 개요

## 1. 데이터베이스

흔히 현대사회를 가리켜 정보화 사회라고 한다. 그만큼 일상생활 속에서 수 없이 쏟아져 나오는 다양한 정보들이 우리의 생활과 밀접한 관계를 맺고 있는 것이다. 따라서 이런 다양한 정보들을 수집, 처리하고, 분석, 응용하는 것은 이제 사회 어느 곳에서도 꼭 필요한 요소가 되었다. 넓은 의미에서의 데이터베이스는 이러한 일상적인 정보들을 모아 놓은 것 자체를 의미한다. 그러나 일반적으로 데이터베이스라고 말할 때는 특정 기업이나 조직 또는 개인이 필요에 의해(ex: 부가가치가 발생하는) 데이터를 일정한 형태로 저장해 놓은 것을 의미한다.

예를 들어, 학교에서는 학생 관리를 목적으로 학생 개인의 정보를 모아둘 것이고, 기업에서는 직원들을 관리하기 위해 직원들의 이름, 부서, 월급 등의 정보를 모아둘 것이다. 그리고 이러한 정보들을 관리하기 위해서 엑셀과 같은 소프트웨어를 이용하여 보기 좋게 정리하여 저장해 놓을 것이다.

그러나 관리 대상이 되는 데이터의 양이 점점 많아지고 같은 데이터를 여러 사람이 동시에 여러 용도로 사용하게 되면서 단순히 엑셀 같은 개인이 관리하는 소프트웨어 만으로는 한계에 부딪히게 된다. 또한 경우에 따라서는 개인의 사소한 부주의로 인해 기업의 사활이 걸린 중요한 데이터가 손상되거나 유실되는 상황이 발생할 수도 있다.

따라서 많은 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.

- 데이터베이스의 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, DB2 와 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(Relational Database)

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 저장되고, 관계형 데이터베이스를 유일하게 조작할 수 있는 SQL 문장에 의해 관리되고 있으므로 관계형 데이터베이스와 SQL의 중요성은 아무리 강조해도 지나치

지 않다.

파일시스템의 경우, 하나의 파일을 많은 사용자가 동시에 검색할 수는 있지만 동시에 입력, 수정, 삭제할 수 없기 때문에 정보의 관리가 어려우므로, 하나의 파일을 여러 사용자나 어플리케이션에서 동시에 사용하기 위해서 원래의 데이터 파일을 여러 개 복사하여 사용하게 된다. 이렇게 여러 개의 데이터 파일이 존재하는 경우에 동일한 데이터가 여러 곳에 저장되는 문제가 발생하고, 하나의 원본 파일에 대한 변경 작업이 발생했을 때 모든 복사본 파일에 대한 변경 작업을 한꺼번에 병행 처리하지 않으면 서로 다른 정보 파일이 존재하기 때문에 데이터의 불일치성이 발생한다.

결과적으로 파일시스템은 분산된 데이터 간의 정합성을 유지하는데 과다한 노력이 필요하게 되고 데이터의 정합성을 보장하기 힘들게 된다.(단, 단일 사용자나 단일 어플리케이션이 파일시스템을 사용하는 경우 데이터베이스보다 처리 성능이 뛰어나므로 특정 업무에서는 아직도 파일시스템을 유용하게 사용하고 있다.)

이러한 문제에 대해 관계형 데이터베이스는 정규화를 통한 합리적인 테이블 모델링을 통해 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.

또한, 관계형 데이터베이스는 메타 데이터를 총괄 관리할 수 있기 때문에 데이터의 성격, 속성 또는 표현 방법 등을 체계화할 수 있고, 데이터 표준화를 통한 데이터 품질을 확보할 수 있는 장점을 가지고 있다.

그리고 DBMS는 인증된 사용자만이 참조할 수 있도록 보안 기능을 제공하고 있다. 테이블 생성 시에 사용할 수 있는 다양한 제약조건을 이용하여 사용자가 실수로 조건에 위배되는 데이터를 입력한 다든지, 관계를 연결하는 중요 데이터를 삭제하는 것을 방지하여 데이터 무결성(Integrity)을 보장할 수 있다.

추가로 DBMS는 시스템의 갑작스런 장애로부터 사용자가 입력, 수정, 삭제하던 데이터가 제대로 반영될 수 있도록 보장해주는 기능과, 시스템 다운, 재해 등의 상황에서도 데이터를 회복/복구할 수 있는 기능을 제공한다.

## 2. SQL(Structured Query Language)

SQL(Structured Query Language)은 관계형 데이터베이스에서 데이터 정의, 데이터 조작, 데이터 제어를 하기 위해 사용하는 언어이다. SQL의 최초 이름이 SEQUEL(Structured English QUery Language)이었기 때문에 ‘시큐얼’로 읽는 경우도 있지만, 표준은 SQL이므로 ‘에스큐엘’로 읽는 것을 권고한다.

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. 참고로 SQL 교육은 정확한 데이터를 출력하는 것이 목표이고, SQL 튜닝의 목적은 시스템에 큰 영향을 주는 SQL을 가장 효과적(응답시간, 자원 활용 최소화)으로 작성하는 것이 목표이다.

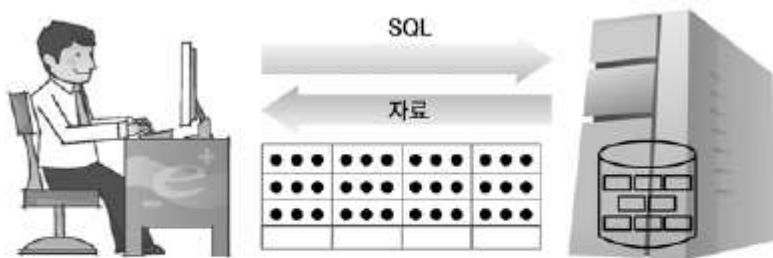
1986년부터 ANSI/ISO를 통해 표준화되고 정의된 SQL 기능은 벤더별 DBMS 개발의 목표가 된다. 일부 구체적인 용어는 다르더라도 대부분의 관계형 데이터베이스에서 ANSI/ISO 표준을 최대한 따르고 있기 때문에, SQL에 대한 지식은 다른 데이터베이스를 사용하더라도 상당 부분 기존 지식을

재활용할 수 있고, ANSI/ISO SQL-99, SQL-2003 이후 기준이 적용된 SQL 이라면 프로그램의 이식성을 높이는 데도 공헌한다.

각 벤더의 관계형 데이터베이스(RDBMS)는 표준화된 SQL 이외에도 벤더 차별화 및 이용 편리성을 위해 추가 기능이나 내장 함수 등에서 독자적 개발을 계속 진행하고 있다. 상호 호환성이 뛰어난 표준 기능과, 벤더별 특징을 가지고 있는 독자적 기능 중 어떤 기능을 선택할 지는 사용자의 몫이지만 가능한 ANSI/ISO 표준을 기준으로 할 것을 권고한다.

SQL 문장은 단순 스크립트가 아니라 이름에도 포함되어 있듯이, 일반적인 개발 언어처럼 독립된 하나의 개발 언어이다. 하지만 일반적인 프로그래밍 언어와는 달리 SQL 은 관계형 데이터베이스에 대한 전담 접속(다른 언어는 관계형 데이터베이스에 접속할 수 없다) 용도로 사용되며 독립되어 있다. 관계형 데이터베이스는 수학의 집합 논리에 입각한 것이므로, SQL 도 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 SQL 은 이러한 대화를 가능하도록 매개 역할을 하는 것이다. 결과적으로 SQL 문장을 배우는 것이 곧 관계형 데이터베이스를 배우는 기본 단계라 할 수 있다.



[그림 II-1-1] 사용자와 데이터베이스 간의 대화 과정

SQL 문장과 관련된 용어 중에서 먼저 테이블에 대한 내용은 건드리지 않고 단순히 조회를 하는 SELECT 문장이 있다. 그리고 테이블에 들어 있는 데이터에 변경을 가하는 UPDATE, DELETE, INSERT 문장은 테이블에 들어 있는 데이터들을 조작하는 종류의 SQL 문장들이다. 그 외, 테이블을 생성하고 수정하고 변경하고 삭제하는 테이블 관련 SQL 문장이 있고, 추가로 데이터에 대한 권한을 제어하는 SQL 문장도 있다.

[표 II-1-1] SQL 문장들의 종류

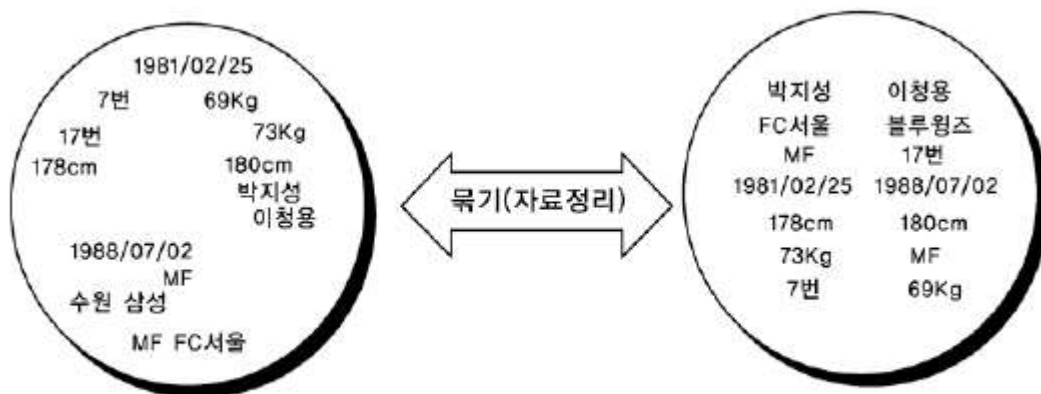
명령어의 종류	명령어	설명
데이터 조작어 (DML: Data Manipulation Language)	SELECT	데이터베이스에 들어 있는 데이터를 조회하거나 검색하기 위한 명령어를 말하는 것으로 RETRIEVE 라고도 한다.
	INSERT UPDATE DELETE	데이터베이스의 테이블에 들어 있는 데이터에 변형을 가하는 종류의 명령어들을 말한다. 예를 들어 데이터를 테이블에 새로운 행을 집어넣거나, 원하지 않는 데이터를 삭제하거나 수정하는 것들의 명령어들을 DML이라고 부른다.
데이터 정의어 (DDL: Data Definition 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.

이들 SQL 명령어는 3 가지 SAVEPOINT 그룹인 DDL, DML, DCL 로 나눌 수 있는데, TCL 의 경우 굳이 나눈다면 일부에서 DCL 로 분류하기도 하지만, 다소 성격이 다르므로 별도의 4 번째 그룹으로 분리할 것을 권고한다.

### 3. TABLE

월드컵 4 강 및 16 강으로 한국 축구에 대한 관심은 점점 높아지고 있다. 따라서 현재 K-League에 등록되어 있는 팀들의 정보와 선수들에 관련된 데이터에 관심을 두고, 선수정보를 데이터베이스화 한다.

다음은 K-리그 구단 홈페이지를 방문하여 팀 및 선수들의 정보를 찾아서 선수들의 이름과 소속 구단, 포지션, 생년월일, 키, 몸무게, 등번호를 노트에 적어본 것이다. 참고로 본 가이드의 K-리그 데이터는 팀명이나 일부 실명이 포함되어 있지만 전체적으로는 가공의 데이터이다.



[그림 II-1-2] K-리그 1차 자료 정리

별도의 정리 작업을 하지 않은 [그림 II-1-2]의 왼쪽 내용은 본인이 아니라면 알아보기도 힘들고 다른 사용자에게 큰 도움이 되지 않는다. 그러나 오른쪽의 내용은 선수별로 필요한 정보가 정리되

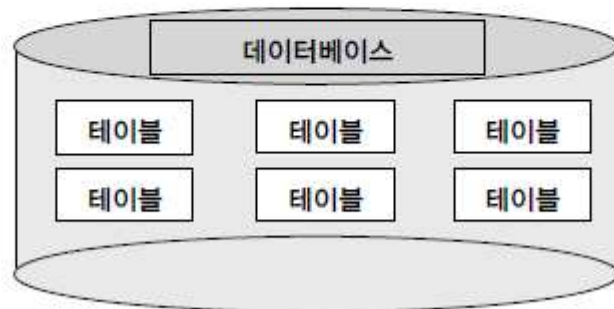


어 관심 있는 다른 사용자에게 도움이 될 수 있다.

그렇지만, 오른쪽의 내용도 한두 명의 선수에 대한 정보는 쉽게 볼 수 있지만 많은 선수들의 정보를 비교하기는 다소 어려워 보인다. 즉, 누가 키가 제일 큰지, 누가 몸무게가 제일 많은지를 판단하기가 어렵다. 엑셀처럼 키는 키대로, 몸무게는 몸무게대로 데이터의 순서를 정해서 비교하는 것이 바람직하다.

[표 II-1-2] K-리그 2차 자료 정리

선수	팀	포지션	백넘버	생년월일	키	몸무게	...
박지성	서울FC	MF	7	1981/02/25	178cm	73kg	⋮
이청용	블루윙즈	MF	17	1988/07/02	180cm	69kg	⋮
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮



[그림 II-1-3] 데이터베이스의 테이블

데이터는 관계형 데이터베이스의 기본 단위인 테이블 형태로 저장된다. 모든 자료는 테이블에 등록이 되고, 우리는 테이블로부터 원하는 자료를 꺼내 올 수 있다.

테이블은 어느 특정한 주제와 목적으로 만들어지는 일종의 집합이다. [표 II-1-2]처럼 K-리그 선수들의 정보들을 하나의 표에서 정리할 수 있다면, 이 표만 있다면 내가 좋아하는 선수들의 상세한 정보들을 볼 수 있고, 선수들의 정보를 상호간에 비교해 볼 수도 있다. 새로운 선수를 입력하려고 할 때 새로운 테이블을 생성할 필요 없이 데이터만 추가함으로써 선수들의 정보를 모두 관리할 수 있다.

[표 II-1-3] 테이블에 저장할 수 있는 자료들

선수	팀	팀연고지	포지션	등번호	생년월일	키	몸무게
최강조	일화천마	성남	MF	6	90/01/24	165	57
오춘식	대구FC	대구	MF	22	88/03/08	168	75
하리	아이파크	부산	FW	10	84/05/14	168	65
윤용구	드래곤즈	전남	MF	15	87/08/08	168	60
정도용	FC서울	서울	MF	40	86/05/28	168	68
전재호	일화천마	성남	MF	14	89/08/08	168	64
홍종하	제주유나이티드FC	제주	MF	32	88/12/21	169	74
오비나	시티즌	대전	MF	26	90/06/03	169	70
고창현	블루윙즈	수원	MF	8	93/09/15	170	64
이청용	블루윙즈	수원	MF	17	1988/07/02	180	69
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

[표 II-1-3]의 내용을 보면 선수, 팀, 팀연고지, 포지션, 등번호, 생년월일, 키, 몸무게가 각각의 칼럼이 되며, 해당 테이블은 반드시 하나 이상의 칼럼을 가져야 한다.

예를 들어 이청용 선수에 대한 정보는 아래와 같이 8개의 칼럼을 가지는 하나의 행으로 데이터화 되어 테이블에 저장된 것이다

이청용	블루윙즈	수원	MF	17	1988/07/02	180	69
-----	------	----	----	----	------------	-----	----

앞서 본 것처럼, 테이블에는 등록된 자료들이 있으며, 이 자료들은 삭제하지 않는 한 지속적으로 유지된다. 만약 우리가 자료를 입력하지 않는다면 테이블은 본래 만들어졌을 때부터 가지고 있던 속성을 그대로 유지하면서 존재하게 된다.

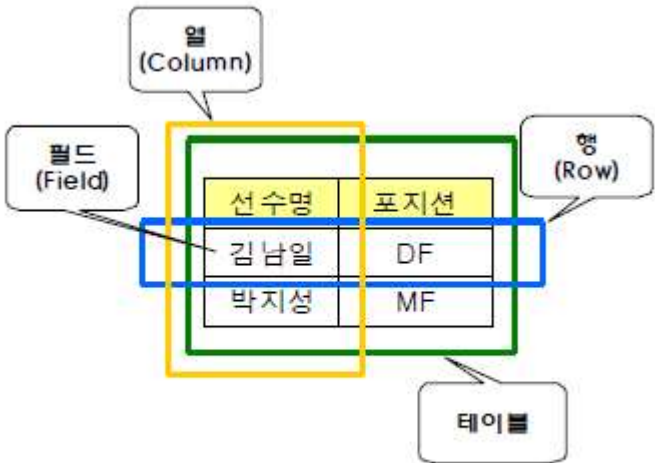
[표 II-1-4] 데이터가 있을 경우와 없을 경우

선수명	소속팀	포지션	선수명	소속팀	포지션
박지성	서울FC	MF			
이청용	블루윙즈	MF			
⋮	⋮	⋮			

테이블에 데이터가 있는 경우

테이블에 데이터가 없는 경우

테이블에 대해서 좀 더 상세히 살펴보면 테이블(TABLE)은 데이터를 저장하는 객체(Object)로서 관계형 데이터베이스의 기본 단위이다. 관계형 데이터베이스에서는 모든 데이터를 칼럼과 행의 2차원 구조로 나타낸다. 세로 방향을 칼럼(Column), 가로 방향을 행(Row)이라고 하고, 칼럼과 행이 겹치는 하나의 공간을 필드(Field)라고 한다. 선수정보 테이블을 예로 들면 선수명과 포지션 등의 칼럼이 있고, 각 선수에 대한 데이터를 행으로 구성하여 저장한다.



[그림 II-1-4] 테이블의 구조

[표 II-1-5] 테이블 용어

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

선수와 관련된 데이터를 저장할 때 모든 데이터를 하나의 테이블로 저장하지 않는다. [그림 II-1-5]를 보면 선수와 관련된 데이터를 선수 테이블과 구단 테이블이라는 복수의 테이블로 분할하여 저장하고 있다.

그리고 분할된 테이블은 그 칼럼의 값에 의해 연결된다. 이렇게 테이블을 분할하여 데이터의 불필요한 중복을 줄이는 것을 정규화(Normalization)라고 한다. 데이터의 정합성 확보와 데이터 입력/수정/삭제시 발생할 수 있는 이상현상(Anomaly)을 방지하기 위해 정규화는 관계형 데이터베이스 모델링에서 매우 중요한 프로세스이다.

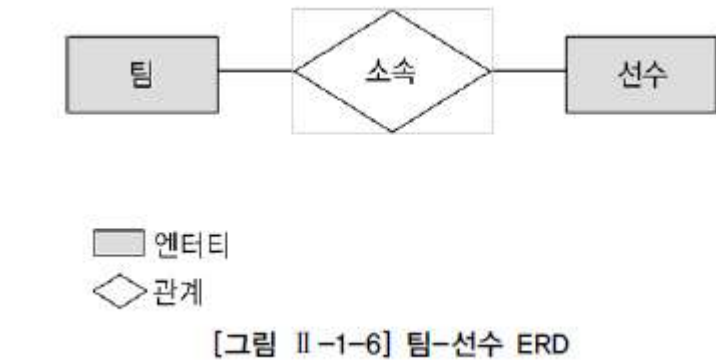


각 행을 한 가지 의미로 특정할 수 있는 한 개 이상의 칼럼을 기본키(Primary Key)라고 하며, 여기서는 <선수> 테이블의 '선수번호'와 <구단> 테이블의 '구단코드'가 기본키가 된다. 또, <선수> 테이블의 '구단코드'와 같이 다른 테이블의 기본 키로 사용되면서 테이블과의 관계를 연결하는 역할을 하는 칼럼을 외부키(Foreign Key)라고 한다.

[표 II-1-6] 테이블 관계 용어들	
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 기본키로 사용되고 있는 관계를 연결하는 칼럼

#### 4. ERD(Entity Relationship Diagram)

팀 정보와 선수 정보 간에는 어떤 의미의 관계가 존재하며, 다른 테이블과도 어떤 의미의 연관성이나 관계를 가지고 있다. ERD(Entity Relationship Diagram)는 이와 같은 관계의 의미를 직관적으로 표현할 수 있는 좋은 수단이다.



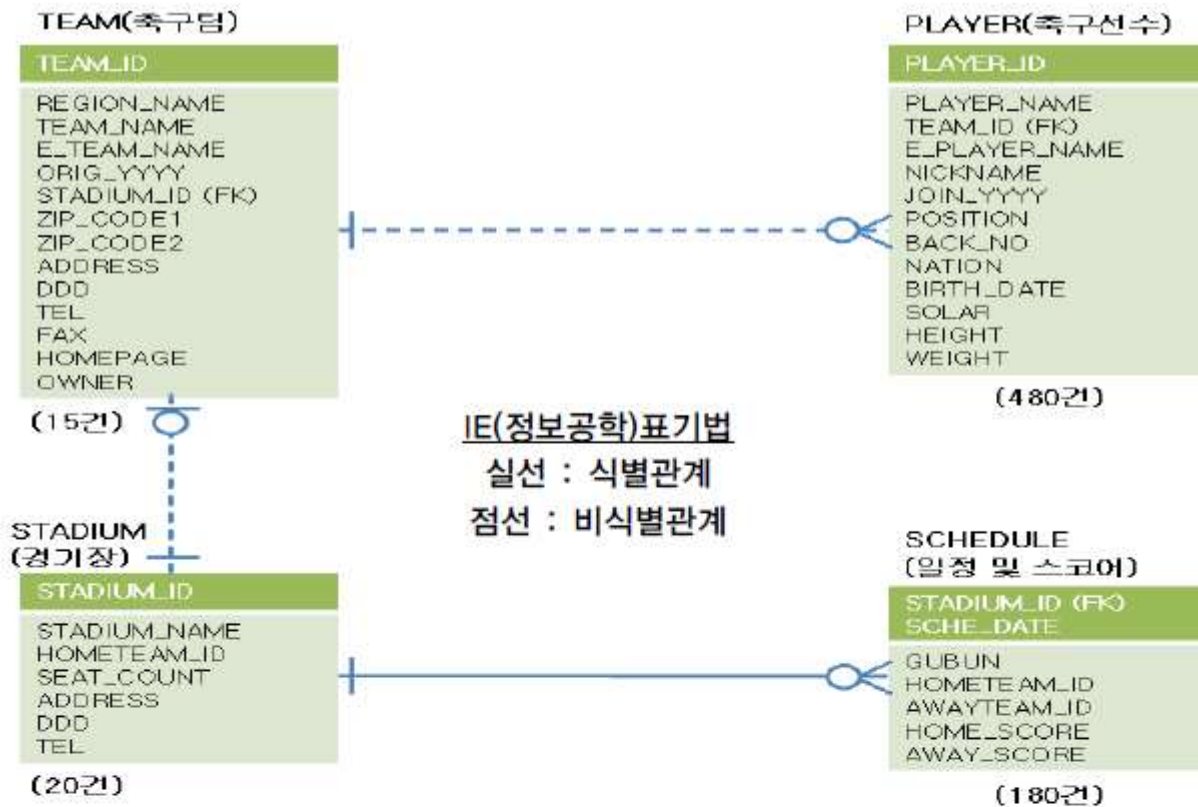
[그림 II-1-6]처럼 팀과 선수 간에는 “소속”이라는 관계가 맺어져 있다. 테이블 간 서로의 상관 관계를 그림으로 도식화한 것을 E-R 다이어그램이라고 하며, 간략히 ERD 라고 한다. ERD 의 구성 요소는 엔터티(Entity), 관계(Relationship), 속성(Attribute) 3 가지이며 현실 세계의 데이터는 이 3 가지 구성 요소로 모두 표현이 가능하다.

[그림 II-1-7]과 [그림 II-1-8]은 앞으로 사용하게 될 K-리그의 테이블 관계를 IE(Information Engineering) 표기법과 Barker(Case\*Method) 표기법으로 표현한 ERD 이다.

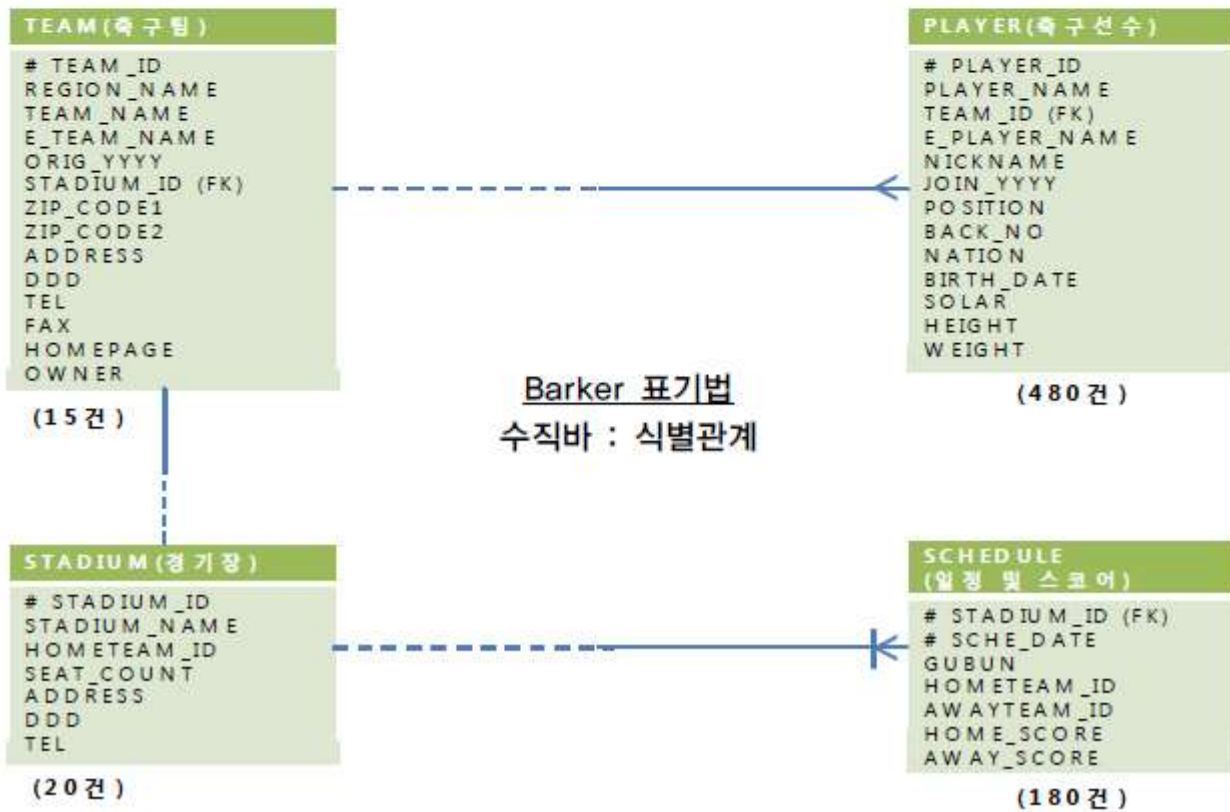
K-리그 테이블 간의 양방향 관계는 다음과 같다.

- 하나의 팀은 여러 명의 선수를 포함할 수 있다. - 한 명의 선수는 하나의 팀에 꼭 속한다.
- 하나의 팀은 하나의 전용 구장을 꼭 가진다. - 하나의 운동장은 하나의 홈팀을 가질 수 있다.
- 하나의 운동장은 여러 게임의 스케줄을 가질 수 있다. - 하나의 스케줄은 하나의 운동장에 꼭 배정된다.



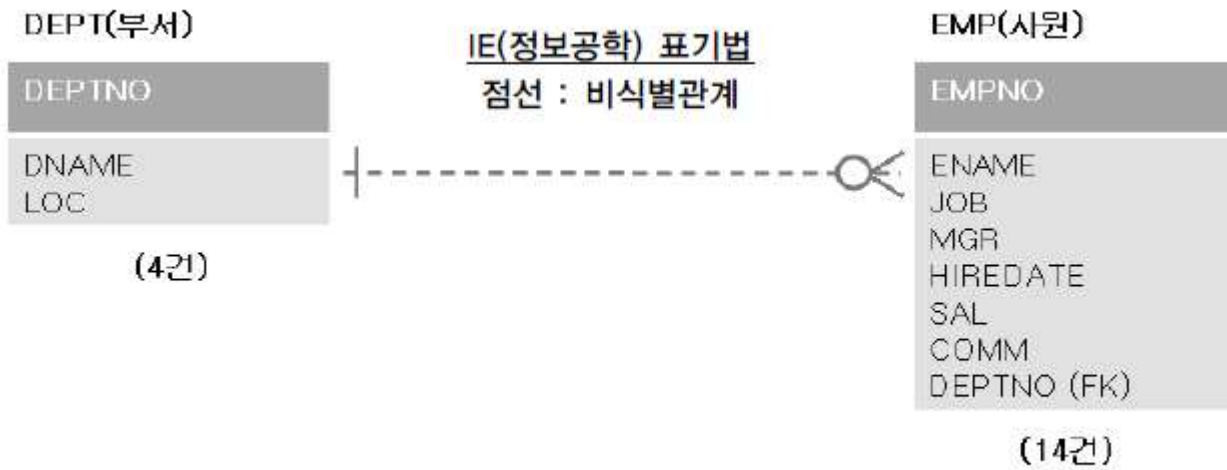


[그림 II-1-7] K-리그 ERD (IE 표기법)



[그림 II-1-8] K-리그 ERD (Barker 표기법)

[그림 II-1-9]와 [그림 II-1-10]은 앞으로 사용하게 될 부서-사원 테이블 간의 관계를 IE 표기법과 Barker 표기법으로 표현한 ERD이다.



[그림 II-1-9] 사원-부서 ERD (IE 표기법)

사원-부서 테이블 간의 양방향 관계는 다음과 같다.

- 하나의 부서는 여러 명의 사원을 보유할 수 있다.
- 한 명의 사원은 하나의 부서에 꼭 소속된다.



[그림 II-1-10] 사원-부서 ERD (Barker 표기법)

## 02.DDL

### 1. 데이터 유형

데이터 유형은 데이터베이스의 테이블에 특정 자료를 입력할 때, 그 자료를 받아들일 공간을 자료의 유형별로 나누는 기준이라고 생각하면 된다. 즉 특정 칼럼을 정의할 때 선언한 데이터 유형은 그 칼럼이 받아들일 수 있는 자료의 유형을 규정한다. 따라서 선언한 유형이 아닌 다른 종류의 데이터가 들어오려고 하면 데이터베이스는 에러를 발생시킨다.

예를 들어 선수의 몸무게 정보를 모아놓은 공간에 '박지성'이라는 문자가 입력되었을 때, 숫자가 의미를 가지는 칼럼 정보에 문자가 입력되었으니 잘못된 데이터라고 판단할 수 있는 것이다. 또한 데이터 유형과 더불어 지정한 크기(SIZE)도 중요한 기능을 제공한다. 즉 선언 당시에 지정한 데이터의 크기를 넘어선 자료가 입력되는 상황도 에러를 발생시키는 중요한 요인이기 때문이다. 데이터베이스에서 사용하는 데이터 유형은 다양한 형태로 제공된다. 벤더별로 SQL 문장의 차이는 적어지고 있지만, 데이터 유형과 내장형 함수 부분에서는 차이가 많은 편이다. 물론 데이터베이스 내부의 구조적인 차이점은 더 크지만 본 가이드 범위를 벗어나므로 여기서는 언급하지 않는다.

숫자 타입을 예를 들어 보면 ANSI/ISO 기준에서는 NUMERIC Type 의 하위 개념으로 NUMERIC, DECIMAL, DEC, SMALLINT, INTEGER, INT, BIGINT, FLOAT, REAL, DOUBLE PRECISION 을 소개하고 있다. SQL Server 와 Sybase 는 ANSI/ISO 기준의 하위 개념에 맞추어서 작은 정수형, 정수형, 큰 정수형, 실수형 등 여러 숫자 타입을 제공하고 있으며, 추가로 MONEY, SMALLMONEY 등의 숫자 타입도 가지고 있다. 반면, Oracle 은 숫자형 타입에 대해서 NUMBER 한 가지 숫자 타입의 데이터 유형만 지원한다. 사용자 입장에서 데이터 유형이나 내장형 함수까지 표준화가 되면 편리하겠지만, 벤더별 특화된 기능마다 장단점이 있으므로 사용자가 여러 상황을 고려해서 판단할 문제이다.

그리고 벤더에서 ANSI/ISO 표준을 사용할 때는 기능을 중심으로 구현하므로, 일반적으로 표준과 다른(ex: NUMERIC → NUMBER, WINDOW FUNCTION → ANALYTIC/RANK FUNCTION) 용어를 사용하는 것은 현실적으로 허용이 된다. 테이블의 칼럼이 가지고 있는 대표적인 4 가지 데이터 유형을 정리하였다. 아래 4 가지 유형 외에도 ANSI/ISO 에서는 Binary String Type, Binary Large Object String Type, National Character String Type, Boolean Type 등의 다양한 유형을 표시하고 있다.

[표 II-1-7] 자주 쓰이는 데이터 유형

데이터 유형	설명
CHARACTER(s)	<ul style="list-style-type: none"> <li>- 고정 길이 문자열 정보 (Oracle, SQL Server 모두 CHAR로 표현)</li> <li>- s는 기본 길이 1바이트, 최대 길이 Oracle 2,000바이트, SQL Server 8,000바이트</li> <li>- s만큼 최대 길이를 갖고 고정 길이를 가지고 있으므로 할당된 변수 값의 길이가 s보다 작을 경우에는 그 차이 길이만큼 공간으로 채워진다.</li> </ul>
VARCHAR(s)	<ul style="list-style-type: none"> <li>- CHARACTER VARYING의 약자로 가변 길이 문자열 정보 (Oracle은 VARCHAR2로 표현, SQL Server는 VARCHAR로 표현)</li> <li>- s는 최소 길이 1바이트, 최대 길이 Oracle 4,000바이트, SQL Server 8,000바이트</li> <li>- s만큼의 최대 길이를 갖지만 가변 길이로 조정이 되기 때문에 할당된 변수값의 바이트만 적용된다. (Limit 개념)</li> </ul>
NUMERIC	<ul style="list-style-type: none"> <li>- 정수, 실수 등 숫자 정보 (Oracle은 NUMBER로, SQL Server는 10가지 이상의 숫자 타입을 가지고 있음)</li> <li>- 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.</li> </ul>
DATETIME	<ul style="list-style-type: none"> <li>- 날짜와 시각 정보 (Oracle은 DATE로 표현, SQL Server는 DATETIME으로 표현)</li> <li>- Oracle은 1초 단위, SQL Server는 3.33ms(millisecond) 단위 관리</li> </ul>

문자열 유형의 경우, CHAR 유형과 VARCHAR 유형 중 어느 유형을 지정하는지에 대한 문제가 자주 논의된다. 중요한 것은 저장 영역과 문자열의 비교 방법이다. VARCHAR 유형은 가변 길이이므로 필요한 영역은 실제 데이터 크기뿐이다. 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.

또 하나는 비교 방법의 차이이다. 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 'AA' = 'AA '

예) VARCHAR 유형 'AA' ≠ 'AA '

가장 많이 사용하는 VARCHAR 유형에 대하여 예를 들어 설명하면, 영문 이름이 VARCHAR(40)으로 40 바이트가 지정되더라도 실제 'PARK,JISUNG'으로 데이터가 입력되는 경우 11 바이트의 공간만을 차지한다. 주민등록번호나 사번처럼 자료들이 고정된 길이의 문자열을 가지지 않는다면 데이터 유형은 VARCHAR 유형을 적용하는 것이 바람직하다.

예를 들자면, 팀이나 운동장의 주소는 정확히 얼마의 문자 길이를 사용할지 예측할 수 없는 경우가 대표적이다. CHAR가 아닌 VARCHAR, NUMERIC 유형에서 정의한 길이나 자릿수의 의미는 해당 데이터 유형이 가질 수 있는 최대한의 한계값을 정의한 것이라고 보아야 한다. 문자열(CHAR와 VARCHAR)에 대한 최대 길이와 NUMBER 칼럼의 정밀도(Precision)를 지정하는 것은 테이블 설계시 반드시 고려해야 할 중요 요소이다. 잘못된 판단은 추후 ALTER TABLE 명령으로 정정할 수는 있지만 데이터가 입력된 상황이라면 처리 과정이 쉽지 않다.



## 2. CREATE TABLE

테이블은 일정한 형식에 의해서 생성된다. 테이블 생성을 위해서는 해당 테이블에 입력될 데이터를 정의하고, 정의한 데이터를 어떠한 데이터 유형으로 선언할 것인지를 결정해야 한다.

### 가. 테이블과 칼럼 정의

테이블에 존재하는 모든 데이터를 고유하게 식별할 수 있으면서 반드시 값이 존재하는 단일 칼럼이나 칼럼의 조합들(후보키) 중에 하나를 선정하여 기본키 칼럼으로 지정한다. 선수 테이블을 예로 들면 ‘선수 ID’ 칼럼이 기본키로 적당할 것이다. 기본키는 단일 칼럼이 아닌 여러 개의 칼럼으로도 만들어질 수 있다.

그리고 테이블과 테이블 간에 정의된 관계는 기본키(PRIMARY KEY)와 외부키(FOREIGN KEY)를 활용해서 설정하도록 한다. 선수 테이블에 선수의 소속팀 정보가 같이 존재한다고 가정하면, 특정 팀의 이름이 변경되었을 경우 그 팀에 소속된 선수 데이터를 일일이 찾아서 수정을 하거나, 또한 팀이 해체되었을 경우 선수 관련 정보까지 삭제되는 수정/삭제 이상(Anomaly) 현상이 발생할 수 있다.

이런 이상 현상을 방지하기 위해 팀 정보를 관리하는 팀 테이블을 별도로 분리해서 팀 ID와 팀 이름을 저장하고, 선수 테이블에서는 팀 ID를 외부키로 참조하게 한다. 데이터 모델링 및 정규화에 대한 내용은 업무를 개선시킬 수 있는 고급 SQL을 작성하는데 필요한 내용이므로 이 부분도 기본적인 내용은 학습할 것을 권고한다.

- 아래는 선수 정보와 함께 K-리그와 관련 있는 다른 데이터들도 같이 살펴본 내용이다.

[표 II-1-8] K-리그의 테이블과 칼럼 정보

테이블	칼럼 설명
선수 정보 (PLAYER)	선수에 대한 상세 정보 (선수ID, 선수명, 소속팀ID, 영문선수명, 선수별명, 입단년도, 포지션, 등번호, 국적, 생년월일, 양/음력, 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)
경기일정 (SCHEDULE)	경기일정 및 스코어에 대한 정보 (운동장ID, 경기일자, 경기진행여부, 홈팀ID, 원정팀ID, 홈팀득점, 원정팀 득점)

- K-리그와는 별개로 회사의 부서와 사원 테이블의 칼럼들도 정리한다.

[표 II-1-9] 부서-사원 관계의 칼럼 정보

테이블	칼럼 설명
부서 (DEPT)	부서에 대한 상세 정보 (부서ID, 부서명, 부서지역)
사원 (EMP)	사원에 대한 상세 정보 (사원ID, 사원명, 업무, 매니저, 입사일자, 급여, 커미션, 부서ID)

### 나. CREATE TABLE

테이블을 생성하는 구문 형식은 다음과 같다.

CREATE TABLE 테이블이름 ( 컬럼명1 DATATYPE [DEFAULT 형식], 컬럼명2 DATATYPE [DEFAULT 형식], 컬럼명2 DATATYPE [DEFAULT 형식] );

다음은 테이블 생성 시에 주의해야 할 몇 가지 규칙이다.

- 테이블명은 객체를 의미할 수 있는 적절한 이름을 사용한다. 가능한 단수형을 권고한다.
- 테이블명은 다른 테이블의 이름과 중복되지 않아야 한다.
- 한 테이블 내에서는 컬럼명이 중복되게 지정될 수 없다.
- 테이블 이름을 지정하고 각 컬럼들은 괄호 "(" 로 묶어 지정한다.
- 각 컬럼들은 콤마 ","로 구분되고, 테이블 생성문의 끝은 항상 세미콜론 ";"으로 끝난다.
- 컬럼에 대해서는 다른 테이블까지 고려하여 데이터베이스 내에서는 일관성 있게 사용하는 것이 좋다.(데이터 표준화 관점)
- 컬럼 뒤에 데이터 유형은 꼭 지정되어야 한다.
- 테이블명과 컬럼명은 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

한 테이블 안에서 컬럼 이름은 달라야 하지만, 다른 테이블의 컬럼 이름과는 같을 수 있다. 예를 들면 선수 테이블의 TEAM\_ID, 팀 테이블의 TEAM\_ID 는 같은 컬럼 이름을 가지고 있다. 실제 DBMS 는 팀 테이블의 TEAM\_ID 를 PC 나 UNIX 의 디렉토리 구조처럼 'DB명+DB사용자명+테이블명+컬럼명' 처럼 계층적 구조를 가진 전체 경로로 관리하고 있다. 이처럼 같은 이름을 가진 컬럼들은 기본키와 외래키의 관계를 가지는 경우가 많으며, 향후 테이블 간의 조인 조건으로 주로 사용되는 중요한 연결고리 컬럼들이다.

[예제] 다음 조건의 형태로 선수 테이블을 생성한다.

테이블명 : PLAYER 테이블 설명 : K-리그 선수들의 정보를 가지고 있는 테이블 컬럼명 : PLAYER\_ID (선수 ID) 문자 고정 자릿수 7 자리, PLAYER\_NAME (선수명) 문자 가변 자릿수 20 자리, TEAM\_ID (팀 ID) 문자 고정 자릿수 3 자리, E\_PLAYER\_NAME (영문선수명) 문자 가변 자릿수 40 자리, NICKNAME (선수별명) 문자 가변 자릿수 30 자리, JOIN\_YYYY (입단년도) 문자 고정 자릿수 4 자리, POSITION (포지션) 문자 가변 자릿수 10 자리, BACK\_NO (등번호) 숫자 2 자리, NATION (국적) 문자 가변 자릿수 20 자리, BIRTH\_DATE (생년월일) 날짜, SOLAR (양/음) 문자 고정 자릿수 1 자리, HEIGHT (신장) 숫자 3 자리, WEIGHT (몸무게) 숫자 3 자리, 제약조건 : 기본키(PRIMARY KEY) → PLAYER\_ID (제약조건명은 PLAYER\_ID\_PK) 값이 반드시 존재 (NOT NULL) → PLAYER\_NAME, TEAM\_ID

[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.

테이블 생성 예제에서 추가적인 주의 사항 몇 가지를 확인하면 다음과 같다.

- 테이블 생성시 대/소문자 구분은 하지 않는다. 기본적으로 테이블이나 컬럼명은 대문자로 만들어진다.
- DATETIME 데이터 유형에는 별도로 크기를 지정하지 않는다.
- 문자 데이터 유형은 반드시

시 가질 수 있는 최대 길이를 표시해야 한다. - 칼럼과 칼럼의 구분은 콤마로 하되, 마지막 칼럼은 콤마를 찍지 않는다. - 칼럼에 대한 제약조건이 있으면 CONSTRAINT 를 이용하여 추가할 수 있다.

제약조건은 PLAYER\_NAME, TEAM\_ID 칼럼의 데이터 유형 뒤에 NOT NULL 을 정의한 사례와 같은 칼럼 LEVEL 정의 방식과, PLAYER\_PK PRIMARY KEY, PLAYER\_FK FOREIGN KEY 사례처럼 테이블 생성 마지막에 모든 제약조건을 기술하는 테이블 LEVEL 정의 방식이 있다. 하나의 SQL 문장 내에서 두 가지 방식은 혼용해서 사용할 수 있으며, 같은 기능을 가지고 있다.

## 다. 제약조건(CONSTRAINT)

제약조건(CONSTRAINT)이란 사용자가 원하는 조건의 데이터만 유지하기 위한 즉, 데이터의 무결성을 유지하기 위한 데이터베이스의 보편적인 방법으로 테이블의 특정 칼럼에 설정하는 제약이다. 테이블을 생성할 때 제약조건을 반드시 기술할 필요는 없지만, 이후에 ALTER TABLE 을 이용해서 추가, 수정하는 경우 데이터가 이미 입력된 경우라면 처리 과정이 쉽지 않으므로 초기 테이블 생성 시점부터 적합한 제약 조건에 대한 충분한 검토가 있어야 한다.

- 제약조건의 종류

[표 II-1-10] 제약조건의 종류

구분	설명
PRIMARY KEY (기본키)	테이블에 저장된 행 데이터를 고유하게 식별하기 위한 기본키를 정의한다. 하나의 테이블에 하나의 기본키 제약만 정의할 수 있다. 기본키 제약을 정의하면 DBMS는 자동으로 UNIQUE 인덱스를 생성하며, 기본키를 구성하는 칼럼에는 NULL을 입력할 수 없다. 결국 '기본키 제약 = 고유키 제약 & NOT NULL 제약'이 된다.
UNIQUE KEY (고유키)	테이블에 저장된 행 데이터를 고유하게 식별하기 위한 고유키를 정의한다. 단, NULL은 고유키 제약의 대상이 아니므로, NULL 값을 가진 행이 여러 개 있더라도 고유키 제약 위반이 되지 않는다.
NOT NULL	NULL 값의 입력을 금지한다. 디폴트 상태에서는 모든 칼럼에서 NULL을 허가하고 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.

- NULL 의미

NULL(ASCII 코드 00 번)은 공백(BLANK, ASCII 코드 32 번)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.

- DEFAULT 의미

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.

[예제] 다음 조건의 형태로 팀 테이블을 생성한다.



테이블명 : TEAM 테이블 설명 : K-리그 선수들의 소속팀에 대한 정보를 가지고 있는 테이블 칼럼명 : TEAM\_ID (팀 고유 ID) 문자 고정 자릿수 3 자리, REGION\_NAME (연고지 명) 문자 가변 자릿수 8 자리, TEAM\_NAME (한글 팀 명) 문자 가변 자릿수 40 자리, E-TEAM\_NAME (영문 팀 명) 문자 가변 자릿수 50 자리, ORIG\_YYYY (창단년도) 문자 고정 자릿수 4 자리, STADIUM\_ID (구장 고유 ID) 문자 고정 자릿수 3 자리, ZIP\_CODE1 (우편번호 앞 3 자리) 문자 고정 자릿수 3 자리, ZIP\_CODE2 (우편번호 뒷 3 자리) 문자 고정 자릿수 3 자리, ADDRESS (주소) 문자 가변 자릿수 80 자리, DDD (지역번호) 문자 가변 자릿수 3 자리, TEL (전화번호) 문자 가변 자릿수 10 자리, FAX (팩스번호) 문자 가변 자릿수 10 자리, HOMEPAGE (홈페이지) 문자 가변 자릿수 50 자리, OWNER (구단주) 문자 가변 자릿수 10 자리, 제약조건 : 기본 키(PRIMARY KEY) → TEAM\_ID (제약조건명은 TEAM\_ID\_PK) NOT NULL → REGION\_NAME, TEAM\_NAME, STADIUM\_ID (제약조건명은 미적용)

[예제] Oracle CREATE TABLE TEAM ( TEAM\_ID CHAR(3) NOT NULL, REGION\_NAME VARCHAR2(8) NOT NULL, TEAM\_NAME VARCHAR2(40) NOT NULL, E\_TEAM\_NAME VARCHAR2(50), ORIG\_YYYY CHAR(4), STADIUM\_ID CHAR(3) NOT NULL, 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), CONSTRAINT TEAM\_PK PRIMARY KEY (TEAM\_ID), CONSTRAINT TEAM\_FK FOREIGN KEY (STADIUM\_ID) REFERENCES STADIUM(STADIUM\_ID) ); 테이블이 생성되었다.

[예제] SQL Server CREATE TABLE TEAM ( TEAM\_ID CHAR(3) NOT NULL, REGION\_NAME VARCHAR(8) NOT NULL, TEAM\_NAME VARCHAR(40) NOT NULL, E\_TEAM\_NAME VARCHAR(50), ORIG\_YYYY CHAR(4), STADIUM\_ID CHAR(3) NOT NULL, ZIP\_CODE1 CHAR(3), ZIP\_CODE2 CHAR(3), ADDRESS VARCHAR(80), DDD VARCHAR(3), TEL VARCHAR(10), FAX VARCHAR(10), HOMEPAGE VARCHAR(50), OWNER VARCHAR(10), CONSTRAINT TEAM\_PK PRIMARY KEY (TEAM\_ID), CONSTRAINT TEAM\_FK FOREIGN KEY (STADIUM\_ID) REFERENCES STADIUM(STADIUM\_ID) ); 테이블이 생성되었다.

## 라. 생성된 테이블 구조 확인

테이블을 생성한 후 테이블의 구조가 제대로 만들어졌는지 확인할 필요가 있다. Oracle 의 경우 “DESCRIBE 테이블명;” 또는 간략히 “DESC 테이블명;”으로 해당 테이블에 대한 정보를 확인할 수 있다. SQL Server 의 경우 “sp\_help ‘dbo.테이블명’ ”으로 해당 테이블에 대한 정보를 확인할 수 있다.

[예제] 선수(PLAYER) 테이블의 구조를 확인한다.

[실행 결과] 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)

[실행 결과] SQL Server exec sp\_help 'dbo.PLAYER' go 칼럼이름 데이터 유형 길이 NULL 가능 -----  
-----  
----- 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

## 마. SELECT 문장을 통한 테이블 생성 사례

다음 절에서 배울 DML 문장 중에 SELECT 문장을 활용해서 테이블을 생성할 수 있는 방법(CTAS: Create Table ~ As Select ~)이 있다. 기존 테이블을 이용한 CTAS 방법을 이용할 수 있다면 칼럼 별로 데이터 유형을 다시 재정의 하지 않아도 되는 장점이 있다. 그러나 CTAS 기법 사용시 주의할 점은 기존 테이블의 제약조건 중에 NOT NULL 만 새로운 복제 테이블에 적용이 되고, 기본키, 고유



키, 외래키, CHECK 등의 다른 제약 조건은 없어진다는 점이다. 제약 조건을 추가하기 위해서는 뒤에 나오는 ALTER TABLE 기능을 사용해야 한다. SQL Server에서는 Select ~ Into ~ 를 활용하여 위와 같은 결과를 얻을 수 있다. 단, 칼럼 속성에 Identity 를 사용했다면 Identity 속성까지 같이 적용이 된다.

[예제] 선수(PLAYER) 테이블과 같은 내용으로 TEAM\_TEMP 라는 복사 테이블을 만들어 본다.

[예제] Oracle CREATE TABLE TEAM\_TEMP AS SELECT \* FROM TEAM; 테이블이 생성되었다.

[실행 결과] Oracle DESC TEAM\_TEMP; 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)

[예제] SQL Server SELECT \* INTO TEAM\_TEMP FROM TEAM; (1 개 행이 영향을 받음)

[실행 결과] SQL Server exec sp\_help 'dbo.TEAM\_TEMP' go 칼럼이름 데이터 유형 길이 NULL 가능 ---  
----- TEAM\_ID CHAR(3) 3 NO REGION\_NAME VARCHAR(8) 8 NO  
TEAM\_NAME VARCHAR(40) 40 NO E\_TEAM\_NAME VARCHAR(50) 50 YES ORIG\_YYYY CHAR(4) 4  
YES STADIUM\_ID CHAR(3) 3 NO ZIP\_CODE1 CHAR(3) 3 YES ZIP\_CODE2 CHAR(3) 3 YES ADDRESS  
VARCHAR(80) 80 YES DDD VARCHAR(3) 3 YES TEL VARCHAR(10) 10 YES FAX VARCHAR(10) 10 YES  
HOMEPAGE VARCHAR(50) 50 YES OWNER VARCHAR(10) 10 YES

### 3. ALTER TABLE

한 번 생성된 테이블은 특별히 사용자가 구조를 변경하기 전까지 생성 당시의 구조를 유지하게 된다. 처음의 테이블 구조를 그대로 유지하는 것이 최선이지만, 업무적인 요구 사항이나 시스템 운영상 테이블을 사용하는 도중에 변경해야 할 일들이 발생할 수도 있다. 이 경우 주로 칼럼을 추가/삭제하거나 제약조건을 추가/삭제하는 작업을 진행하게 된다.

#### 가. ADD COLUMN

다음은 기존 테이블에 필요한 칼럼을 추가하는 명령이다.

ALTER TABLE 테이블명 ADD 추가할 칼럼명 데이터 유형;

주의할 것은 새롭게 추가된 칼럼은 테이블의 마지막 칼럼이 되며 칼럼의 위치를 지정할 수는 없다.


[예제] PLAYER 테이블에 ADDRESS(데이터 유형은 가변 문자로 자릿수 80 자리로 설정한다.) 칼럼을 추가한다.

[예제] Oracle ALTER TABLE PLAYER ADD (ADDRESS VARCHAR2(80)); 테이블이 변경되었다.

[실행 결과] Oracle DESC 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) ADDRESS VARCHAR2(80) 추가된 열

[예제] SQL Server ALTER TABLE PLAYER ADD ADDRESS VARCHAR(80); 명령이 완료되었다.

[실행 결과] SQL Server exec sp\_help 'dbo.PLAYER' go 칼럼이름 데이터 유형 길이 NULL 가능 -----  
----- 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 ADDRESS VARCHAR(80) 80 YES  추가된 열

## 나. DROP COLUMN\

DROP COLUMN 은 테이블에서 필요 없는 칼럼을 삭제할 수 있으며, 데이터가 있거나 없거나 모두 삭제 가능하다. 한 번에 하나의 칼럼만 삭제 가능하며, 칼럼 삭제 후 최소 하나 이상의 칼럼이 테이블에 존재해야 한다. 주의할 부분은 한 번 삭제된 칼럼은 복구가 불가능하다. 다음은 테이블의 불필요한 칼럼을 삭제하는 명령이다.

ALTER TABLE 테이블명 DROP COLUMN 삭제할 칼럼명;

[예제] 앞에서 PLAYER 테이블에 새롭게 추가한 ADDRESS 칼럼을 삭제한다.

[예제] Oracle ALTER TABLE PLAYER DROP COLUMN ADDRESS; 테이블이 변경되었다.

[예제] SQL Server ALTER TABLE PLAYER DROP COLUMN ADDRESS; 명령이 완료되었다.

실행 결과에서 삭제된 칼럼 ADDRESS 가 존재하지 않는 것을 확인할 수 있다.

[실행 결과] Oracle DESC 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)

[실행 결과] SQL Server exec sp\_help 'dbo.PLAYER' go 칼럼이름 데이터 유형 길이 NULL 가능 -----  
-----  
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

## 다. MODIFY COLUMN

테이블에 존재하는 칼럼에 대해서 ALTER TABLE 명령을 이용해 칼럼의 데이터 유형, 디폴트 (DEFAULT) 값, NOT NULL 제약조건에 대한 변경을 포함할 수 있다. 다음은 테이블의 칼럼에 대한 정의를 변경하는 명령이다.

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

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

칼럼을 변경할 때는 몇 가지 사항을 고려해서 변경해야 한다.

- 해당 칼럼의 크기를 늘릴 수는 있지만 줄이지는 못한다. 이는 기존의 데이터가 훼손될 수 있기 때문이다.
- 해당 칼럼이 NULL 값만 가지고 있거나 테이블에 아무 행도 없으면 칼럼의 폭을 줄일 수 있다.
- 해당 칼럼이 NULL 값만을 가지고 있으면 데이터 유형을 변경할 수 있다.
- 해당 칼럼의 DEFAULT 값을 바꾸면 변경 작업 이후 발생하는 행 삽입에만 영향을 미치게 된다.
- 해당 칼럼에 NULL 값이 없을 경우에만 NOT NULL 제약조건을 추가할 수 있다.

[예제] TEAM 테이블의 ORIG\_YYYY 칼럼의 데이터 유형을 CHAR(4)→VARCHAR2(8)으로 변경하고, 향후 입력되는 데이터의 DEFAULT 값으로 '20020129'를 적용하고, 모든 행의 ORIG\_YYYY 칼럼에 NULL 이 없으므로 제약조건을 NULL → NOT NULL 로 변경한다.

[예제] Oracle ALTER TABLE TEAM\_TEMP MODIFY (ORIG\_YYYY VARCHAR2(8) DEFAULT '20020129' NOT NULL); 테이블이 변경되었다.

[예제] SQL Server ALTER TABLE TEAM\_TEMP ALTER COLUMN ORIG\_YYYY VARCHAR(8) NOT NULL; 명령이 완료되었다. ALTER TABLE TEAM\_TEMP ADD CONSTRAINT DF\_ORIG\_YYYY DEFAULT '20020129' FOR ORIG\_YYYY; 명령이 완료되었다.

실행 결과에서 테이블 구조의 변경 사항을 확인할 수 있다.

[실행 결과] Oracle DESC TEAM\_TEMP; 컬럼 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 NOT NULL VARCHAR2(8) 기본  
값 '20020129' 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)

[실행 결과] SQL Server exec sp\_help 'dbo.TEAM\_TEMP' go 컬럼 이름 데이터 유형 길이 NULL 가능 ---  
----- TEAM\_ID CHAR(3) 3 NO REGION\_NAME VARCHAR(8) 8  
NO TEAM\_NAME VARCHAR(40) 40 NO E\_TEAM\_NAME VARCHAR(50) 50 YES ORIG\_YYYY CHAR(4)  
4 YES STADIUM\_ID CHAR(3) 3 NO ZIP\_CODE1 CHAR(3) 3 YES ZIP\_CODE2 CHAR(3) 3 YES ADDRESS  
VARCHAR(80) 80 YES DDD VARCHAR(3) 3 YES TEL VARCHAR(10) 10 YES FAX VARCHAR(10) 10 YES  
HOMEPAGE VARCHAR(50) 50 YES OWNER VARCHAR(10) 10 YES constraint\_type constraint\_name  
constraint\_keys ----- DEFAULT on column  
ORIG\_YYYY DF\_ORIG\_YYYY ('20020129')

- RENAME COLUMN

아래는 테이블을 생성하면서 만들어졌던 컬럼명을 어떤 이유로 불가피하게 변경해야 하는 경우에 유용하게 쓰일 수 있는 RENAME COLUMN 문구이다.

ALTER TABLE 테이블명 RENAME COLUMN 변경해야 할 컬럼명 TO 새로운 컬럼명; /div>  
RENAME COLUMN 으로 컬럼명이 변경되면, 해당 컬럼과 관계된 제약조건에 대해서도 자동으로 변경되는 장점이 있지만, ADD/DROP COLUMN 기능처럼 ANSI/ISO 에 명시되어 있는 기능이 아니고 Oracle 등 일부 DBMS 에서만 지원하는 기능이다.

ALTER TABLE PLAYER RENAME COLUMN PLAYER\_ID TO TEMP\_ID; 테이블이 변경되었다. ALTER TABLE PLAYER RENAME COLUMN TEMP\_ID TO PLAYER\_ID; 테이블이 변경되었다.

SQL Server에서는 sp\_rename 저장 프로시저를 이용하여 컬럼 이름을 변경할 수 있다.

sp\_rename 변경해야 할 컬럼명, 새로운 컬럼명, 'COLUMN';

sp\_rename 'dbo.TEAM\_TEMP.TEAM\_ID', 'TEAM\_TEMP\_ID', 'COLUMN'; 주의: 엔터티 이름 부분을 변경하면 스크립트 및 저장 프로시저를 손상시킬 수 있다.

## 라. DROP CONSTRAINT

테이블 생성 시 부여했던 제약조건을 삭제하는 명령어 형태는 다음과 같다.

ALTER TABLE 테이블명 DROP CONSTRAINT 제약조건명;

[예제] PLAYER 테이블의 외래키 제약조건을 삭제한다.

[예제] Oracle ALTER TABLE PLAYER DROP CONSTRAINT PLAYER\_FK; 테이블이 변경되었다.

[예제] SQL Server ALTER TABLE PLAYER DROP CONSTRAINT PLAYER\_FK; 명령이 완료되었다.

## 마. ADD CONSTRAINT

테이블 생성 시 제약조건을 적용하지 않았다면, 생성 이후에 필요에 의해서 제약조건을 추가할 수 있다. 다음은 특정 칼럼에 제약조건을 추가하는 명령어 형태이다.

ALTER TABLE 테이블명 ADD CONSTRAINT 제약조건명 제약조건 (칼럼명);

[예제] PLAYER 테이블에 TEAM 테이블과의 외래키 제약조건을 추가한다. 제약조건명은 PLAYER\_FK 로 하고, PLAYER 테이블의 TEAM\_ID 칼럼이 TEAM 테이블의 TEAM\_ID 를 참조하는 조건이다.

[예제] Oracle ALTER TABLE PLAYER ADD CONSTRAINT PLAYER\_FK FOREIGN KEY (TEAM\_ID) REFERENCES TEAM(TEAM\_ID); 테이블이 변경되었다.

[예제] SQL Server ALTER TABLE PLAYER ADD CONSTRAINT PLAYER\_FK FOREIGN KEY (TEAM\_ID) REFERENCES TEAM(TEAM\_ID); 명령이 완료되었다.

[예제] PLAYER 테이블이 참조하는 TEAM 테이블을 제거해본다.

[예제] Oracle DROP TABLE TEAM; ERROR: 외래 키에 의해 참조되는 고유/기본 키가 테이블에 있다. ※ 테이블은 삭제되지 않음

[예제] SQL Server DROP TABLE TEAM; ERROR: 엔터티 'TEAM'은 FOREIGN KEY 제약 조건을 참조하므로 삭제할 수 없다. ※ 테이블은 삭제되지 않음

[예제] PLAYER 테이블이 참조하는 TEAM 테이블의 데이터를 삭제해본다.

[예제] Oracle DELETE TEAM WHERE TEAM\_ID = 'K10'; ERROR: 무결성 제약조건(SCOTT.PLAYER\_FK)이 위배되었다. 자식 레코드가 발견되었다. ※ 데이터는 삭제되지 않음

[예제] SQL Server DELETE TEAM WHERE TEAM\_ID = 'K10'; ERROR: FOREIGN KEY 제약 조건을 참조하므로 삭제할 수 없다. ※ 데이터는 삭제되지 않음

위와 같이 참조 제약조건을 추가하면 PLAYER 테이블의 TEAM\_ID 칼럼이 TEAM 테이블의 TEAM\_ID 칼럼을 참조하게 된다. 참조 무결성 옵션에 따라서 만약 TEAM 테이블이나 TEAM 테이블의 데이터를 삭제하려 할 경우 외부(PLAYER 테이블)에서 참조되고 있기 때문에 삭제가 불가능하게 제약을 할 수 있다. 즉, 외부키(FK)를 설정함으로써 실수에 의한 테이블 삭제나 필요한 데이터의 의도하지 않은 삭제와 같은 불상사를 방지하는 효과를 볼 수 있다.

## 4. RENAME TABLE

RENAME 명령어를 사용하여 테이블의 이름을 변경할 수 있다.

RENAME 변경전 테이블명 TO 변경후 테이블명;

SQL Server에서는 sp\_rename 을 이용하여 테이블 이름을 변경할 수 있다.

sp\_rename 변경전 테이블명, 변경후 테이블명;

[예제] RENAME 문장을 이용하여 TEAM 테이블명을 다른 이름으로 변경하고, 다시 TEAM 테이블로 변경한다.

[예제] Oracle RENAME TEAM TO TEAM\_BACKUP; 테이블 이름이 변경되었다. RENAME TEAM\_BACKUP TO TEAM; 테이블 이름이 변경되었다.

[예제] SQL Server sp\_rename 'dbo.TEAM','TEAM\_BACKUP'; 주의: 엔터티 이름 부분을 변경하면 스크립트 및 저장 프로시저를 손상시킬 수 있다. sp\_rename 'dbo.TEAM\_BACKUP','TEAM'; 주의: 엔터티 이름 부분을 변경하면 스크립트 및 저장 프로시저를 손상시킬 수 있다.

## 5. DROP TABLE



테이블을 잘못 만들었거나 테이블이 더 이상 필요 없을 경우 해당 테이블을 삭제해야 한다. 다음은 불필요한 테이블을 삭제하는 명령이다.

DROP TABLE 테이블명 [CASCADE CONSTRAINT];

DROP 명령어를 사용하면 테이블의 모든 데이터 및 구조를 삭제한다. CASCADE CONSTRAINT 옵션은 해당 테이블과 관계가 있었던 참조되는 제약조건에 대해서도 삭제한다는 것을 의미한다. SQL Server에서는 CASCADE 옵션이 존재하지 않으며 테이블을 삭제하기 전에 참조하는 FOREIGN KEY 제약 조건 또는 참조하는 테이블을 먼저 삭제해야 한다.

[예제] PLAYER 테이블을 제거한다.

[예제] Oracle DROP TABLE PLAYER; 테이블이 삭제되었다. DESC PLAYER; ERROR: 설명할 객체를 찾을 수 없다.

[예제] SQL Server DROP TABLE PLAYER; 명령이 완료되었다. exec sp\_help 'dbo.PLAYER'; 메시지 15009, 수준 16, 상태 1, 프로시저 sp\_help, 줄 66 데이터베이스 'northwind'에 엔터티 'dbo.player'이(가) 없거나 이 작업에 적합하지 않다.

## 6. TRUNCATE TABLE

TRUNCATE TABLE 은 테이블 자체가 삭제되는 것이 아니고, 해당 테이블에 들어있던 모든 행들이 제거되고 저장 공간을 재사용 가능하도록 해제한다. 테이블 구조를 완전히 삭제하기 위해서는 DROP TABLE 을 실행하면 된다.

TRUNCATE TABLE PLAYER;

[예제] TRUNCATE TABLE 을 사용하여 해당 테이블의 모든 행을 삭제하고 테이블 구조를 확인한다.

[예제] Oracle TRUNCATE TABLE TEAM; 테이블이 트렁케이팅되었다.

[예제] SQL Server TRUNCATE TABLE TEAM; 명령이 완료되었다.

[실행 결과] Oracle 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)

[실행 결과] SQL Server exec sp\_help 'dbo.TEAM' go 칼럼이름 데이터 유형 길이 NULL 가능 -----  
----- TEAM\_ID CHAR(3) 3 NO REGION\_NAME VARCHAR(8) 8 NO  
TEAM\_NAME VARCHAR(40) 40 NO E\_TEAM\_NAME VARCHAR(50) 50 YES ORIG\_YYYY CHAR(4) 4  
YES STADIUM\_ID CHAR(3) 3 NO ZIP\_CODE1 CHAR(3) 3 YES ZIP\_CODE2 CHAR(3) 3 YES ADDRESS  
VARCHAR(80) 80 YES DDD VARCHAR(3) 3 YES TEL VARCHAR(10) 10 YES FAX VARCHAR(10) 10 YES  
HOMEPAGE VARCHAR(50) 50 YES OWNER VARCHAR(10) 10 YES

[예제] DROP TABLE 을 사용하여 해당 테이블을 제거하고 테이블 구조를 확인한다.

[예제] Oracle DROP TABLE TEAM; 테이블이 삭제되었다. DESC TEAM; ERROR: 설명할 객체를 찾을 수 없다.

[예제] SQL Server DROP TABLE TEAM; 명령이 완료되었다. exec sp\_help 'dbo.TEAM'; 메시지 15009, 수준 16, 상태 1, 프로시저 sp\_help, 줄 66 데이터베이스 'northwind'에 엔터티 'dbo.TEAM'이(가) 없거나 이 작업에 적합하지 않다.

DROP TABLE 의 경우는 테이블 자체가 없어지기 때문에 테이블 구조를 확인할 수 없다. 반면 TRUNCATE TABLE 의 경우는 테이블 구조는 그대로 유지한 채 데이터만 전부 삭제하는 기능이다.

TRUNCATE 는 데이터 구조의 변경 없이 테이블의 데이터를 일괄 삭제하는 명령어로 DML 로 분류할 수도 있지만 내부 처리 방식이나 Auto Commit 특성 등으로 인해 DDL 로 분류하였다. 테이블에 있는 데이터를 삭제하는 명령어는 TRUNCATE TABLE 명령어 이외에도 다음 DML 절에서 살펴볼 DELETE 명령어가 있다. 그러나 DELETE 와 TRUNCATE 는 처리하는 방식 자체가 다르다. 테이블의 전체 데이터를 삭제하는 경우, 시스템 활용 측면에서는 DELETE TABLE 보다는 시스템 부하가 적은 TRUNCATE TABLE 을 권고한다. 단, TRUNCATE TABLE 의 경우 정상적인 복구가 불가능하므로 주의해야 한다.

## 03.DML

앞 절에서 테이블을 생성하고 생성된 테이블의 구조를 변경하는 명령어에 대해서 알아보았다. 지금 부터는 만들어진 테이블에 관리하기를 원하는 자료들을 입력, 수정, 삭제, 조회하는 DML(DATA MANIPULATION LANGUAGE) 사용 방법을 알아본다.

### 1. INSERT

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

▶ INSERT INTO 테이블명 (COLUMN\_LIST)VALUES (COLUMN\_LIST 에 넣을 VALUE\_LIST); ▶ INSERT INTO 테이블명 VALUES (전체 COLUMN 에 넣을 VALUE\_LIST);

해당 칼럼명과 입력되어야 하는 값을 서로 1:1 로 매핑해서 입력하면 된다. 해당 칼럼의 데이터 유형이 CHAR 나 VARCHAR2 등 문자 유형일 경우 『 ' 』(SINGLE QUOTATION)로 입력할 값을 입력한다. 숫자일 경우 『 ' 』(SINGLE QUOTATION)을 붙이지 않아야 한다. 첫 번째 유형은 테이블의 칼럼을 정의할 수 있는데, 이때 칼럼의 순서는 테이블의 칼럼 순서와 매치할 필요는 없으며, 정의하지 않은 칼럼은 Default 로 NULL 값이 입력된다. 단, Primary Key 나 Not NULL 로 지정된 칼럼은 NULL 이 허용되지 않는다. 두 번째 유형은 모든 칼럼에 데이터를 입력하는 경우로 굳이 COLUMN\_LIST 를 언급하지 않아도 되지만, 칼럼의 순서대로 빠짐없이 데이터가 입력되어야 한다.

[예제] 선수 테이블에 박지성 선수의 데이터를 일부 칼럼만 입력한다.

[예제] ▶ 테이블명 : PLAYER INSERT INTO PLAYER (PLAYER\_ID, PLAYER\_NAME, TEAM\_ID, POSITION, HEIGHT, WEIGHT, BACK\_NO) VALUES ('2002007', '박지성', 'K07', 'MF', 178, 73, 7); 1 개의 행이 만들어졌다.

[표 II-1-11]은 데이터베이스 내에 있는 PLAYER 테이블에 박지성 선수 정보가 입력되어 있는 것을 나타낸 것이다. INSERT 문장에서 BACK\_NO 가 마지막에 정의가 되었더라도 테이블에는 칼럼 순서대로 데이터가 입력되었다. 칼럼명이 정의되지 않은 경우 NULL 값이 입력되었다.

[표 II-1-11] 박지성 선수 자료가 들어가 있는 테이블

PLAYER_ID	PLAYER_NAME	TEAM_ID	E_PLAYER_NAME	NICKNAME	JOIN_YYYY	
2002007	박지성	K07				
POSITION	BACK_NO	NATION	BIRTH_DATE	SOLAR	HEIGHT	WEIGHT
MF	7				178	73

[예제] 해당 테이블에 이청용 선수의 데이터를 입력해본다.

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

[표 II-1-12] 이청용 선수 자료가 들어가 있는 테이블

PLAYER_ID	PLAYER_NAME	TEAM_ID	E_PLAYER_NAME	NICKNAME	JOIN_YYYY	
2002010	이청용	K07		BlueDragon	2002	
POSITION	BACK_NO	NATION	BIRTH_DATE	SOLAR	HEIGHT	WEIGHT
MF	17			1	180	69

데이터를 입력하는 경우 정의되지 않은 미지의 값인 E\_PLAYER\_NAME 은 두 개의 『 " 』SINGLE QUOTATION 을 붙여서 표현하거나, NATION 이나 BIRTH\_DATE 의 경우처럼 NULL 이라고 명시적으로 표현할 수 있다.

## 2. UPDATE

입력한 정보 중에 잘못 입력되거나 변경이 발생하여 정보를 수정해야 하는 경우가 발생할 수 있다. 다음은 UPDATE 문장의 기본 형태이다. UPDATE 다음에 수정되어야 할 칼럼이 존재하는 테이블명을 입력하고 SET 다음에 수정되어야 할 칼럼명과 해당 칼럼에 수정되는 값으로 수정이 이루어진다.

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

[예제] 선수 테이블의 백넘버를 일괄적으로 99 로 수정한다.

[예제] UPDATE PLAYER SET BACK\_NO = 99; 480 개의 행이 수정되었다.

[예제] 선수 테이블의 포지션을 일괄적으로 'MF' 로 수정한다.

[예제] UPDATE PLAYER SET POSITION = 'MF'; 480 개의 행이 수정되었다.

## 3. DELETE

테이블의 정보가 필요 없게 되었을 경우 데이터 삭제를 수행한다. 다음은 DELETE 문장의 기본적인 형태이다. DELETE FROM 다음에 삭제를 원하는 자료가 저장되어 있는 테이블명을 입력하고 실행한다. 이때 FROM 문구는 생략이 가능한 키워드이며, 뒤에서 배울 WHERE 절을 사용하지 않는다면 테이블의 전체 데이터가 삭제된다.

DELETE [FROM] 삭제를 원하는 정보가 들어있는 테이블명;

[예제] 선수 테이블의 데이터를 전부 삭제한다.

[예제] DELETE FROM PLAYER; 480 개의 행이 삭제되었다.

참고로 데이터베이스는 DDL 명령어와 DML 명령어를 처리하는 방식에 있어서 차이를 보인다. DDL(CREATE, ALTER, RENAME, DROP) 명령어인 경우에는 직접 데이터베이스의 테이블에 영향을 미치기 때문에 DDL 명령어를 입력하는 순간 명령어에 해당하는 작업이 즉시(AUTO COMMIT) 완료된다. 하지만 DML(INSERT, UPDATE, DELETE, SELECT) 명령어의 경우, 조작하려는 테이블을 메모리 버퍼에 올려놓고 작업을 하기 때문에 실시간으로 테이블에 영향을 미치는 것은 아니다. 따라서 버퍼에서 처리한 DML 명령어가 실제 테이블에 반영되기 위해서는 COMMIT 명령어를 입력하여 TRANSACTION 을 종료해야 한다.

그러나 SQL Server 의 경우는 DML 의 경우도 AUTO COMMIT 으로 처리되기 때문에 실제 테이블에 반영하기 위해 COMMIT 명령어를 입력할 필요가 없다. 테이블의 전체 데이터를 삭제하는 경우, 시스템 활용 측면에서는 삭제된 데이터를 로그로 저장하는 DELETE TABLE 보다는 시스템 부하가 적은 TRUNCATE TABLE 을 권고한다. 단, TRUNCATE TABLE 의 경우 삭제된 데이터의 로그가 없으므로 ROLLBACK 이 불가능하므로 주의해야 한다. 그러나 SQL Server 의 경우 사용자가 임의적으로 트랜잭션을 시작한 후 TRUNCATE TABLE 을 이용하여 데이터를 삭제한 이후 오류가 발견되어, 다시 복구를 원할 경우 ROLLBACK 문을 이용하여 테이블 데이터를 원 상태로 되돌릴 수 있다. 트랜잭션과 COMMIT, ROLLBACK 에 대해서는 다음 절에서 설명한다.

## 4. SELECT



사용자가 입력한 데이터는 언제라도 조회가 가능하다. 앞에서 입력한 자료들을 조회해보는 SQL 문은 다음과 같다. (별도 제공한 SQL SCRIPT 를 통해 모든 테이블의 데이터를 새롭게 생성한 후, 이후 본 가이드 내용을 진행하기 바란다.)

SELECT [ALL/DISTINCT] 보고 싶은 칼럼명, 보고 싶은 칼럼명, ... FROM 해당 칼럼들이 있는 테이블명; - ALL : Default 옵션이므로 별도로 표시하지 않아도 된다. 중복된 데이터가 있어도 모두 출력한다. - DISTINCT : 중복된 데이터가 있는 경우 1건으로 처리해서 출력한다.

[예제] 조회하기를 원하는 칼럼명을 SELECT 다음에 콤마 구분자(,)로 구분하여 나열하고, FROM 다음에 해당 칼럼이 존재하는 테이블명을 입력하여 실행시킨다. 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 2011123 비에라 K05 MF 21 176 73 2008460 서동원 K05 MF 22 184 78 2010019 안대현 K05 MF 25 179 72 2010018 양현정 K05 MF 14 176 72 2010022 유원섭 K05 MF 37 180 77 2012008 김수철 K05 MF 34 171 68 2012013 임다한 K05 DF 39 181 67 : : : : : : : 480 개의 행이 선택되었다.

- DISTINCT 옵션

[예제 및 실행 결과] SELECT ALL POSITION FROM PLAYER; ALL 은 생략 가능한 키워드이므로 아래 SQL 문장도 같은 결과를 출력한다, SELECT POSITION FROM PLAYER; 480 개의 행이 선택되었다.

[예제] SELECT DISTINCT POSITION FROM PLAYER;

[실행 결과] Oracle POSITION ----- GK DF FW MF 5 개의 행이 선택되었다.

실행 결과를 보면 480 개의 행이 모두 출력된 것이 아니라 포지션의 종류인 4 개의 행과 포지션 데이터가 아직 미정인 NULL 까지 5 건의 데이터만 출력이 되었다.

- WILDCARD 사용하기

입력한 정보들을 보기위해 PLAYER 테이블에서 보고 싶은 정보들이 있는 칼럼들을 선택하여 조회해보는 것이다. 해당 테이블의 모든 칼럼 정보를 보고 싶을 경우에는 와일드카드로 애스터리스크(\*)를 사용하여 조회할 수 있다.

SELECT \* FROM 테이블명;

[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 SOLAR 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 작은탱크 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 2011123 비에라 K05 Vieira 2011 MF 21 1984-02-25 1 176 73 2008460 서동원 K05 SEO, DONGWON 2008 MF 22 1985-08-14 1 184 78 2010019 안대현 K05 AN, DAEHYUN 2010 MF 25 1987-08-20 1 179 72 2010018 양현정 K05 YANG, HYUNJUNG 2010 MF 14 1987-07-25 1 176 72 2010022 유원섭 K05 YOU, WONSUOB 양마 2010 MF 37 1991-05-24 1 180 77 2012008 김수철 K05 KIM, SUCHEUL 2012 MF 34 1989-05-26 1 171 68 2012013 임다한 K05 LIM, DAHAN 달마 2012 DF 39 1989-07-21 1 181 67 : : : : : : : : : : : : 480 개의 행이 선택되었다.

실행 결과 화면을 보면 칼럼 레이블(LABEL)이 맨 위에 보이고, 레이블 밑에 점선이 보인다. 실질적인 자료는 다음 줄부터 시작된다. 레이블은 기본적으로 대문자로 보이고, 첫 라인에 보이는 레이블의 정렬은 다음과 같다.

- 좌측 정렬 : 문자 및 날짜 데이터 - 우측 정렬 : 숫자 데이터

본 가이드에서는 가독성을 위해 일부 칼럼의 좌정렬, 우정렬을 무시한 경우가 있으니 참고하기 바란다.

- ALIAS 부여하기

조회된 결과에 일종의 별명(ALIAS, ALIASES)을 부여해서 칼럼 레이블을 변경할 수 있다. 칼럼 별명(ALIAS)에 대한 사항을 정리하면 다음과 같다.

- 칼럼명 바로 뒤에 온다. - 칼럼명과 ALIAS 사이에 AS, as 키워드를 사용할 수도 있다. (option)  
- 이중 인용부호(Double quotation)는 ALIAS가 공백, 특수문자를 포함할 경우와 대소문자 구분이 필요할 경우 사용된다.

[예제] 입력한 선수들의 정보를 칼럼 별명을 이용하여 출력한다.

[예제] SELECT PLAYER\_NAME AS 선수명, POSITION AS 위치, HEIGHT AS 키, WEIGHT AS 몸무게  
FROM PLAYER; 칼럼 별명에서 AS 를 꼭 사용하지 않아도 되므로, 아래 SQL 은 위 SQL 과 같은 결과를 출력한다. SELECT PLAYER\_NAME 선수명, POSITION 위치, HEIGHT 키, WEIGHT 몸무게 FROM  
PLAYER;

[실행 결과] 선수명 위치 키 몸무게 ----- 정경량 MF 173 65 정은익 MF 176 63 레오마르 MF 183 77 명재용 MF 173 63 변재섭 MF 170 63 보띠 MF 174 68 비에라 MF 176 73 서동원 MF 184 78 안대현 MF 179 72 양현정 MF 176 72 유원섭 MF 180 77 김수철 MF 171 68 임다한 DF 181 67 : : : : 480 개의 행이 선택되었다.

[예제] 칼럼 별명을 적용할 때 별명 중간에 공백이 들어가는 경우 『" "』를 사용해야 한다. SQL Server 의 경우『" "』, 『" '』, 『( )』와 같이 3 가지의 방식으로 별명을 부여할 수 있다.

[예제] SELECT PLAYER\_NAME "선수 이름", POSITION "그라운드 포지션", HEIGHT "키", WEIGHT "몸무게"  
FROM PLAYER;

[실행 결과] 선수 이름 그라운드 포지션 키 몸무게 ----- 정경량 MF 173 65 정은익 MF 176 63 레오마르 MF 183 77 명재용 MF 173 63 변재섭 MF 170 63 보띠 MF 174 68 비에라 MF 176 73 서동원 MF 184 78 안대현 MF 179 72 양현정 MF 176 72 유원섭 MF 180 77 김수철 MF 171 68 임다한 DF 181 67 : : : : 480 개의 행이 선택되었다.

## 5. 산술 연산자와 합성 연산자

- 산술 연산자

산술 연산자는 NUMBER 와 DATE 자료형에 대해 적용되며 일반적으로 수학에서의 4 칩 연산과 동일하다. 그리고 우선순위를 위한 괄호 적용이 가능하다. 일반적으로 산술 연산을 사용하거나 특정 함수를 적용하게 되면 칼럼의 LABEL 이 길어지게 되고, 기존의 칼럼에 대해 새로운 의미를 부여한 것이므로 적절한 ALIAS 를 새롭게 부여하는 것이 좋다. 그리고 산술 연산자는 수학에서와 같이 (), \*, /, +, - 의 우선순위를 가진다.

[표 II-1-13] 산술 연산자의 종류

산술 연산자	설 명
( )	연산자 우선순위를 변경하기 위한 괄호 (괄호 안의 연산이 우선된다)
*	곱하기
/	나누기
+	더하기
-	빼기

[예제] 선수들의 키에서 몸무게를 뺀 값을 알아본다.

[예제] SELECT PLAYER\_NAME 이름, HEIGHT - WEIGHT "키-몸무게" FROM PLAYER;

[실행 결과] 이름 키-몸무게 --- ----- 정경량 108.00 정은익 113.00 레오마르 106.00 명재용 110.00 변재섭 107.00 보피 106.00 비에라 103.00 서동원 106.00 안대현 107.00 양현정 104.00 유원섭 103.00 김수철 103.00 임다한 114.00 ... 480 개의 행이 선택되었다.

[예제] 선수들의 키와 몸무게를 이용해서 BMI(Body Mass Index) 비만지수를 측정한다. ※ 예제에서 사용된 ROUND( ) 함수는 반올림을 위한 내장 함수로써 6 절에서 학습한다

[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 명재용 21.05 변재섭 21.80 보피 22.46 비에라 23.57 서동원 23.04 안대현 22.47 양현정 23.24 유원섭 23.77 김수철 23.26 임다한 20.45 ... 480 개의 행이 선택되었다.

- 합성(CONCATENATION) 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 2 개의 수직 바(II)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.

[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;

[실행 결과] 체격정보 정경량선수,173cm,65kg 정은익선수,176cm,63kg 레오마르선수,183cm,77kg 명재용선수,173cm,63kg 변재섭선수,170cm,63kg 보피선수,174cm,68kg 비에라선수,176cm,73kg 서동원선수,184cm,78kg 안대현선수,179cm,72kg 양현정선수,176cm,72kg 유원섭선수,180cm,77kg 김수철선수,171cm,68kg 임다한선수,181cm,67kg ... 480 개의 행이 선택되었다.

## 04.TCL

### 1. 트랜잭션 개요

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 분할할 수 없는 최소의 단위이다. 그렇기 때문에 전부 적용하거나 전부 취소한다. 즉, TRANSACTION은 ALL OR NOTHING의 개념인 것이다. 은행에서의 계좌이체 상황을 연상하면 트랜잭션을 이해하는데 도움이 된다. 계좌이체는 최소한 두 가지 이상의 작업으로 이루어져 있다. 우선 자신의 계좌에서 잔액을 확인하고 이체할 금액을 인출한 다음 나머지 금액을 저장한다. 그리고 이체할 계좌를 확인하고 앞에서 인출한 금액을 더한 다음에 저장하면 계좌이체가 성공한다.

계좌이체 사례 - STEP1. 100 번 계좌의 잔액에서 10,000 원을 뺀다. - STEP2. 200 번 계좌의 잔액에 10,000 원을 더한다.

계좌이체라는 작업 단위는 이런 두 개의 업데이트가 모두 성공적으로 완료되었을 때 종료된다. 둘 중 하나라도 실패할 경우 계좌이체는 원래의 금액을 유지하고 있어야만 한다. 만약 어떠한 장애에 의해 어느 쪽이든 한 쪽만 실행했을 경우, 이체한 금액은 어디로 증발해 버렸거나 마음대로 증가하게 된다. 당연히 그런 일이 있어서는 안 되므로 이러한 경우에는 수정을 취소하여 원 상태로 되돌려야 한다. 이런 계좌이체 같은 하나의 논리적인 작업 단위를 구성하는 세부적인 연산들의 집합을 트랜잭션이라 한다. 이런 관점에서 데이터베이스 응용 프로그램은 트랜잭션의 집합으로 정의할 수도 있다.

올바르게 반영된 데이터를 데이터베이스에 반영시키는 것을 커밋(COMMIT), 트랜잭션 시작 이전의 상태로 되돌리는 것을 롤백(ROLLBACK)이라고 하며, 저장점(SAVEPOINT) 기능과 함께 3 가지 명령어를 트랜잭션을 컨트롤하는 TCL(TRANSACTION CONTROL LANGUAGE)로 분류한다. 트랜잭션의 대상이 되는 SQL 문은 UPDATE, INSERT, DELETE 등 데이터를 수정하는 DML 문이다. SELECT 문장은 직접적인 트랜잭션의 대상이 아니지만, SELECT FOR UPDATE 등 배타적 LOCK을 요구하는 SELECT 문장은 트랜잭션의 대상이 될 수 있다. 트랜잭션의 특성을 살펴보면 [표 II-1-14]와 같다.

[표 II-1-14] 트랜잭션의 특성

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

계좌이체는 한 계좌에서 현금이 인출된 후에 다른 계좌로 입금이 되는데, 현금이 인출되기 전에 다른 계좌에 입금이 되는 것은 문제를 발생시킬 수 있다. 그리고 이체가 결정되기 전까지는 다른 사람이 이 계좌의 정보를 변경할 수 없다. 이것을 보통 문에 자물쇠를 채우듯이 한다고 하여 잠금(LOCKING)이라고 표현한다. 트랜잭션의 특성(특히 원자성)을 충족하기 위해 데이터베이스는 다양한 레벨의 잠금 기능을 제공하고 있는데, 잠금은 기본적으로 트랜잭션이 수행하는 동안 특정 데이터에

대해서 다른 트랜잭션이 동시에 접근하지 못하도록 제한하는 기법이다. 잠금이 걸린 데이터는 잠금을 실행한 트랜잭션만 독점적으로 접근할 수 있고 다른 트랜잭션으로부터 간섭이나 방해를 받지 않는 것이 보장된다. 그리고 잠금이 걸린 데이터는 잠금을 수행한 트랜잭션만이 해제할 수 있다.

## 2. COMMIT

입력한 자료나 수정한 자료에 대해서 또는 삭제한 자료에 대해서 전혀 문제가 없다고 판단되었을 경우 COMMIT 명령어를 통해서 트랜잭션을 완료할 수 있다. COMMIT 이나 ROLLBACK 이전의 데이터 상태는 다음과 같다.

- 단지 메모리 BUFFER 에만 영향을 받았기 때문에 데이터의 변경 이전 상태로 복구 가능하다.
- 현재 사용자는 SELECT 문장으로 결과를 확인 가능하다.
- 다른 사용자는 현재 사용자가 수행한 명령의 결과를 볼 수 없다.
- 변경된 행은 잠금(LOCKING)이 설정되어서 다른 사용자가 변경할 수 없다.

[예제] PLAYER 테이블에 데이터를 입력하고 COMMIT 을 실행한다.

[예제] Oracle INSERT INTO PLAYER (PLAYER\_ID, TEAM\_ID, PLAYER\_NAME, POSITION, HEIGHT, WEIGHT, BACK\_NO) VALUES ('1997035', 'K02', '이운재', 'GK', 182, 82, 1); 1 개의 행이 만들어졌다. COMMIT; 커밋이 완료되었다.

[예제] PLAYER 테이블에 있는 데이터를 수정하고 COMMIT 을 실행한다.

[예제] Oracle UPDATE PLAYER SET HEIGHT = 100; 480 개의 행이 수정되었다. COMMIT; 커밋이 완료되었다.

[예제] PLAYER 테이블에 있는 데이터를 삭제하고 COMMIT 을 실행한다.

[예제] Oracle DELETE FROM PLAYER; 480 개의 행이 삭제되었다. COMMIT; 커밋이 완료되었다.

COMMIT 명령어는 이처럼 INSERT 문장, UPDATE 문장, DELETE 문장을 사용한 후에 이런 변경 작업이 완료되었음을 데이터베이스에 알려 주기 위해 사용한다. COMMIT 이후의 데이터 상태는 다음과 같다.

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

### • SQL Server 의 COMMIT

Oracle 은 DML 을 실행하는 경우 DBMS 가 트랜잭션을 내부적으로 실행하며 DML 문장 수행 후 사용자가 임의로 COMMIT 혹은 ROLLBACK 을 수행해 주어야 트랜잭션이 종료된다. (일부 틀에서는 AUTO COMMIT 을 옵션으로 선택할 수 있다.) 하지만, SQL Server 는 기본적으로 AUTO COMMIT 모드이기 때문에 DML 수행 후 사용자가 COMMIT 이나 ROLLBACK 을 처리할 필요가 없다. DML 구문이 성공이면 자동으로 COMMIT 이 되고 오류가 발생할 경우 자동으로 ROLLBACK 처리된다. 위의 예제를 SQL Server 용으로 변경하면 아래와 같다.

[예제] PLAYER 테이블에 데이터를 입력한다.

[예제] SQL Server INSERT INTO PLAYER (PLAYER\_ID, TEAM\_ID, PLAYER\_NAME, POSITION, HEIGHT, WEIGHT, BACK\_NO) VALUES ('1997035', 'K02', '이운재', 'GK', 182, 82, 1); 1 개의 행이 만들어졌다.

[예제] PLAYER 테이블에 있는 데이터를 수정한다.

[예제] SQL Server UPDATE PLAYER SET HEIGHT = 100; 480 개의 행이 수정되었다.

[예제] PLAYER 테이블에 있는 데이터를 삭제한다.



[예제] SQL Server DELETE FROM PLAYER; 480 개의 행이 삭제되었다.

SQL Server 에서의 트랜잭션은 기본적으로 3 가지 방식으로 이루어진다.

1) AUTO COMMIT SQL Server 의 기본 방식이며, DML, DDL 을 수행할 때마다 DBMS 가 트랜잭션을 컨트롤하는 방식이다. 명령어가 성공적으로 수행되면 자동으로 COMMIT 을 수행하고 오류가 발생하면 자동으로 ROLLBACK 을 수행한다.

2) 암시적 트랜잭션 Oracle 과 같은 방식으로 처리된다. 즉, 트랜잭션의 시작은 DBMS 가 처리하고 트랜잭션의 끝은 사용자가 명시적으로 COMMIT 또는 ROLLBACK 으로 처리한다. 인스턴스 단위 또는 세션 단위로 설정할 수 있다. 인스턴스 단위로 설정하려면 서버 속성 창의 연결화면에서 기본 연결 옵션 중 암시적 트랜잭션에 체크를 해주면 된다. 세션 단위로 설정하기 위해서는 세션 옵션 중 SET IMPLICIT TRANSACTION ON 을 사용하면 된다.

3) 명시적 트랜잭션 트랜잭션의 시작과 끝을 모두 사용자가 명시적으로 지정하는 방식이다. BEGIN TRANSACTION (BEGIN TRAN 구문도 가능)으로 트랜잭션을 시작하고 COMMIT TRANSACTION(TRANSACTION 은 생략 가능) 또는 ROLLBACK TRANSACTION(TRANSACTION 은 생략 가능)으로 트랜잭션을 종료한다. ROLLBACK 구문을 만나면 최초의 BEGIN TRANSACTION 시점까지 모두 ROLLBACK 이 수행된다.

### 3. ROLLBACK

테이블 내 입력한 데이터나, 수정한 데이터, 삭제한 데이터에 대하여 COMMIT 이전에는 변경 사항을 취소할 수 있는데 데이터베이스에서는 롤백(ROLLBACK) 기능을 사용한다. 롤백(ROLLBACK)은 데이터 변경 사항이 취소되어 데이터의 이전 상태로 복구되며, 관련된 행에 대한 잠금(LOCKING)이 풀리고 다른 사용자들이 데이터 변경을 할 수 있게 된다.

[예제] PLAYER 테이블에 데이터를 입력하고 ROLLBACK 을 실행한다.

[예제] Oracle INSERT INTO PLAYER (PLAYER\_ID, TEAM\_ID, PLAYER\_NAME, POSITION, HEIGHT, WEIGHT, BACK\_NO) VALUES ('1999035', 'K02', '이운재', 'GK', 182, 82, 1); 1 개의 행이 만들어졌다. ROLLBACK; 롤백이 완료되었다.

[예제] PLAYER 테이블에 있는 데이터를 수정하고 ROLLBACK 을 실행한다.

[예제] Oracle UPDATE PLAYER SET HEIGHT = 100; 480 개의 행이 수정되었다. ROLLBACK; 롤백이 완료되었다.

[예제] PLAYER 테이블에 있는 데이터를 삭제하고 ROLLBACK 을 실행한다.

[예제] Oracle DELETE FROM PLAYER; 480 개의 행이 삭제되었다. ROLLBACK; 롤백이 완료되었다.

- SQL Server 의 ROLLBACK

SQL Server 는 위에서 언급한 바와 같이 AUTO COMMIT 이 기본 방식이므로 임의적으로 ROLLBACK 을 수행하려면 명시적으로 트랜잭션을 선언해야 한다. 위의 예제는 아래와 같이 변경된다.

[예제] PLAYER 테이블에 데이터를 입력하고 ROLLBACK 을 실행한다.

[예제] SQL Server BEGIN TRAN INSERT INTO PLAYER (PLAYER\_ID, TEAM\_ID, PLAYER\_NAME, POSITION, HEIGHT, WEIGHT, BACK\_NO) VALUES ('1999035', 'K02', '이운재', 'GK', 182, 82, 1); 1 개의 행이 만들어졌다. ROLLBACK; 롤백이 완료되었다.

[예제] PLAYER 테이블에 있는 데이터를 수정하고 ROLLBACK 을 실행한다.

[예제] SQL Server BEGIN TRAN UPDATE PLAYER SET HEIGHT = 100; 480 개의 행이 수정되었다. ROLLBACK; 롤백이 완료되었다.

[예제] PLAYER 테이블에 있는 데이터를 삭제하고 ROLLBACK 을 실행한다.

[예제] SQL Server BEGIN TRAN DELETE FROM PLAYER; 480 개의 행이 삭제되었다. ROLLBACK; 롤백이 완료되었다.

ROLLBACK 후의 데이터 상태는 다음과 같다.

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

COMMIT 과 ROLLBACK 을 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

## 4. SAVEPOINT

저장점(SAVEPOINT)을 정의하면 롤백(ROLLBACK)할 때 트랜잭션에 포함된 전체 작업을 롤백하는 것이 아니라 현 시점에서 SAVEPOINT 까지 트랜잭션의 일부만 롤백할 수 있다. 따라서 복잡한 대규모 트랜잭션에서 에러가 발생했을 때 SAVEPOINT 까지의 트랜잭션만 롤백하고 실패한 부분에 대해서만 다시 실행할 수 있다. (일부 틀에서는 지원이 안 될 수 있음) 복수의 저장점을 정의할 수 있으며, 동일이름으로 저장점을 정의했을 때는 나중에 정의한 저장점이 유효하다. 다음의 SQL 문은 SVPT1 이라는 저장점을 정의하고 있다.

SAVEPOINT SVPT1;

저장점까지 롤백할 때는 ROLLBACK 뒤에 저장점 명을 지정한다.

ROLLBACK TO SVPT1;

위와 같이 롤백(ROLLBACK)에 SAVEPOINT 명을 부여하여 실행하면 저장점 설정 이후에 있었던 데이터 변경에 대해서만 원래 데이터 상태로 되돌아가게 된다. SQL Server 는 SAVE TRANSACTION 을 사용하여 동일한 기능을 수행할 수 있다. 다음의 SQL 문은 SVTRI 이라는 저장점을 정의하고 있다.

SAVE TRANSACTION SVTRI;

저장점까지 롤백할 때는 ROLLBACK 뒤에 저장점 명을 지정한다.

ROLLBACK TRANSACTION SVTRI;

[예제] SAVEPOINT 를 지정하고, PLAYER 테이블에 데이터를 입력한 다음 롤백(ROLLBACK)을 이전에 설정한 저장점까지 실행한다.

[예제] Oracle SAVEPOINT SVPT1; 저장점이 생성되었다. INSERT INTO PLAYER (PLAYER\_ID, TEAM\_ID, PLAYER\_NAME, POSITION, HEIGHT, WEIGHT, BACK\_NO) VALUES ('1999035', 'K02', '이운재', 'GK', 182, 82, 1); 1 개의 행이 만들어졌다. ROLLBACK TO SVPT1; 롤백이 완료되었다.

[예제] SQL Server SAVE TRAN SVTRI; 저장점이 생성되었다. INSERT INTO PLAYER (PLAYER\_ID, TEAM\_ID, PLAYER\_NAME, POSITION, HEIGHT, WEIGHT, BACK\_NO) VALUES ('1999035', 'K02', '이운재', 'GK', 182, 82, 1); 1 개의 행이 만들어졌다. ROLLBACK TRAN SVTRI; 롤백이 완료되었다.

[예제] 먼저 SAVEPOINT 를 지정하고 PLAYER 테이블에 있는 데이터를 수정한 다음 롤백(ROLLBACK)을 이전에 설정한 저장점까지 실행한다.

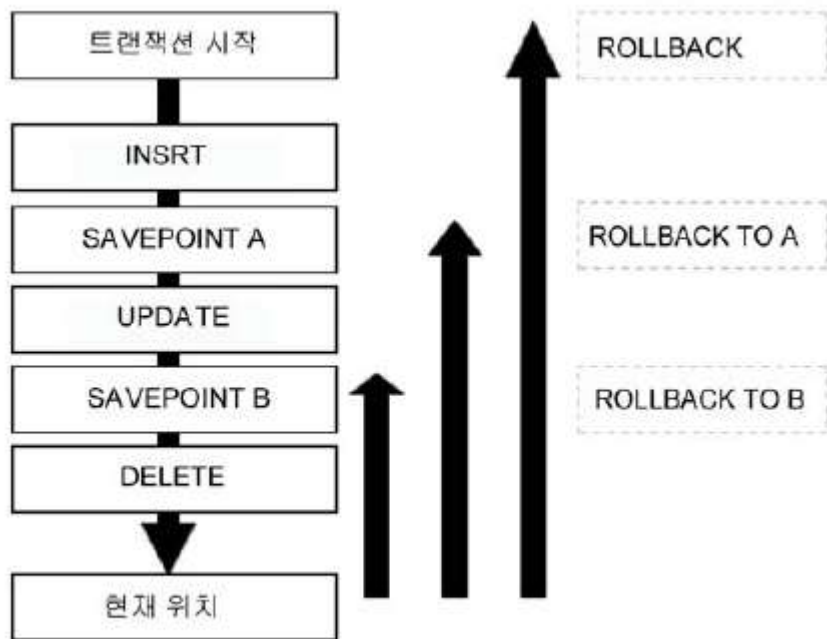
[예제] Oracle SAVEPOINT SVPT2; 저장점이 생성되었다. UPDATE PLAYER SET WEIGHT = 100; 480 개의 행이 수정되었다. ROLLBACK TO SVPT2; 롤백이 완료되었다.

[예제] SQL Server SAVE TRAN SVTR2; 저장점이 생성되었다. UPDATE PLAYER SET WEIGHT = 100; 480 개의 행이 수정되었다. ROLLBACK TRAN SVTR2; 롤백이 완료되었다.

[예제] SAVEPOINT 를 지정하고, PLAYER 테이블에 있는 데이터를 삭제한 다음 롤백(ROLLBACK)을 이전에 설정한 저장점까지 실행한다.

[예제] Oracle SAVEPOINT SVPT3; 저장점이 생성되었다. DELETE FROM PLAYER; 480 개의 행이 삭제되었다. ROLLBACK TO SVPT3; 롤백이 완료되었다.

[예제] SQL Server SAVE TRAN SVTR3; 저장점이 생성되었다. DELETE FROM PLAYER; 480 개의 행이 삭제되었다. ROLLBACK TRAN SVTR3; 롤백이 완료되었다.



[그림 II-1-11] ROLLBACK 원리 (Oracle 기준)

[그림 II-1-11]에서 보듯이 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 "ROLLBACK"을 실행했을 경우 반영안된 모든 변경 사항을 취소하고 트랜잭션 시작 위치로 되돌아간다.

[예제] 새로운 트랜잭션을 시작하기 전에 PLAYER 테이블의 데이터 건수와 몸무게가 100인 선수의 데이터 건수를 확인한다. ※ 몸무게를 확인할 수 있는 WHERE 절 조건과 데이터 건수를 집계하기 위한 COUNT 함수는 1장 5절과 6절에서 설명한다.

[예제 및 실행 결과] Oracle SELECT COUNT(\*) FROM PLAYER; COUNT(\*) ----- 480 1개의 행이 선택되었다. SELECT COUNT(\*) FROM PLAYER WHERE WEIGHT = 100; COUNT(\*) ----- 0 1개의 행이 선택되었다.

[예제] [그림 II-1-11]을 확인하기 위해 새로운 트랜잭션을 시작하고 SAVEPOINT A와 SAVEPOINT B를 지정한다. (둘에 AUTO COMMIT 옵션이 적용되어 있는 경우 해제함)

[예제 및 실행 결과] Oracle 새로운 트랜잭션 시작 INSERT INTO PLAYER (PLAYER\_ID, TEAM\_ID, PLAYER\_NAME, POSITION, HEIGHT, WEIGHT, BACK\_NO) VALUES ('1999035', 'K02', '이운재', 'GK', 182, 82, 1); 1개의 행이 만들어졌다. SAVEPOINT SVPT\_A; 저장점이 생성되었다. UPDATE PLAYER SET WEIGHT = 100; 481개의 행이 수정되었다. SAVEPOINT SVPT\_B; 저장점이 생성되었다. DELETE FROM PLAYER; 481개의 행이 삭제되었다. 현재 위치에서 [예제] CASE 1,2,3을 순서대로 수행해본다.

[예제] CASE1. SAVEPOINT B 저장점까지 롤백(ROLLBACK)을 수행하고 롤백 이후 데이터를 확인해 본다.

[예제 및 실행 결과] Oracle SELECT COUNT(\*) FROM PLAYER; COUNT(\*) ----- 0 1개의 행이 선택되었다. ROLLBACK TO SVPT\_B; 롤백이 완료되었다. SELECT COUNT(\*) FROM PLAYER; COUNT(\*) ----- 481 1개의 행이 선택되었다.

[예제] CASE2. SAVEPOINT A 저장점까지 롤백(ROLLBACK)을 수행하고 롤백 이후 데이터를 확인해 본다.

[예제 및 실행 결과] Oracle SELECT COUNT(\*) FROM PLAYER WHERE WEIGHT = 100; COUNT(\*) ----- 481 1개의 행이 선택되었다. ROLLBACK TO SVPT\_A; 롤백이 완료되었다. SELECT COUNT(\*) FROM PLAYER WHERE WEIGHT = 100; COUNT(\*) ----- 0 1개의 행이 선택되었다.

[예제] CASE3. 트랜잭션 최초 시점까지 롤백(ROLLBACK)을 수행하고 롤백 이후 데이터를 확인해 본다.

[예제 및 실행 결과] Oracle SELECT COUNT(\*) FROM PLAYER; COUNT(\*) ----- 481 1개의 행이 선택되었다. ROLLBACK; 롤백이 완료되었다. SELECT COUNT(\*) FROM PLAYER; COUNT(\*) ----- 480 1개의 행이 선택되었다.

- 앞서 배운 트랜잭션에 대해서 다시 한 번 정리한다.

해당 테이블에 데이터의 변경을 발생시키는 입력(INSERT), 수정(UPDATE), 삭제(DELETE) 수행시 그 변경되는 데이터의 무결성을 보장하는 것이 커밋(COMMIT)과 롤백(ROLLBACK)의 목적이다. 커밋(COMMIT)은 “변경된 데이터를 테이블이 영구적으로 반영해라”라는 의미를 갖는 것이고, 롤백(ROLLBACK)은 “변경된 데이터가 문제가 있으니 변경 전 데이터로 복귀하라”라는 의미이다. 저장점(SAVEPOINT/SAVE TRANSACTION)은 “데이터 변경을 사전에 지정한 저장점까지만 롤백하라”는 의미이다. Oracle의 트랜잭션은 트랜잭션의 대상이 되는 SQL 문장을 실행하면 자동으로 시작되고, COMMIT 또는 ROLLBACK을 실행한 시점에서 종료된다. 단, 다음의 경우에는 COMMIT과 ROLLBACK을 실행하지 않아도 자동으로 트랜잭션이 종료된다.

- CREATE, ALTER, DROP, RENAME, TRUNCATE TABLE 등 DDL 문장을 실행하면 그 이후 시점에 자동으로 커밋된다. - 부연하면, DML 문장 이후에 커밋 없이 DDL 문장이 실행되면 DDL 수행 전에 자동으로 커밋된다. - 데이터베이스를 정상적으로 접속을 종료하면 자동으로 트랜잭션이 커밋된다. - 애플리케이션의 이상 종료로 데이터베이스와의 접속이 단절되었을 때는 트랜잭션이 자동으로 롤백된다.

SQL Server의 트랜잭션은 DBMS가 트랜잭션을 컨트롤하는 방식인 AUTO COMMIT이 기본 방식이다. 다음의 경우는 Oracle과 같이 자동으로 트랜잭션이 종료된다.

- 애플리케이션의 이상 종료로 데이터베이스(인스턴스)와의 접속이 단절되었을 때는 트랜잭션이 자동으로 롤백된다.



## 05.WHERE 절

### 1. WHERE 조건절 개요

자료를 검색할 때 SELECT 절과 FROM 절만을 사용하여 기본적인 SQL 문장을 구성한다면, 테이블에 있는 모든 자료들이 결과로 출력되어 실제로 원하는 자료를 확인하기 어려울 수 있다. 사용자들은 자신이 원하는 자료만을 검색하기 위해서 SQL 문장에 WHERE 절을 이용하여 자료들에 대하여 제한할 수 있다. WHERE 절에는 두 개 이상의 테이블에 대한 조인 조건을 기술하거나 결과를 제한하기 위한 조건을 기술할 수도 있다. WHERE 절의 JOIN 조건에 대해서는 1 장 9 절에서 설명하고 FROM 절의 JOIN 에 대해서는 2 장 1 절에서 설명하도록 한다.

현실의 데이터베이스는 많은 사용자나 프로그램들이 동시에 접속하여 다량의 트랜잭션을 발생하고 있다. WHERE 조건절을 사용하지 않고 필요 없는 많은 자료들을 데이터베이스로부터 요청하는 SQL 문장은 대량의 데이터를 검색하기 위해 데이터베이스가 설치되어 있는 서버의 CPU 나 MEMORY 와 같은 시스템 자원(Resources)들을 과다하게 사용한다. 또한 많은 사용자들의 QUERY 에 대해 바로바로 처리를 해주지 못하게 되고, 또한 검색된 많은 자료들이 네트워크를 통해서 전달됨으로써 문제점들을 발생시킨다.

이런 문제점을 방지하기 위해 WHERE 절에 조건이 없는 FTS(Full Table Scan) 문장은 SQL 튜닝의 1차적인 검토 대상이 된다. (FTS 가 무조건 나쁜 것은 아니며 병렬 처리 등을 이용해 유용하게 사용하는 경우도 많다.) 기본적인 SQL 문장은 Oracle 의 경우 필수적으로 SELECT 절과 FROM 절로 이루어져 있다. SQL Server, Sybase 문장은 SELECT 목록에 상수, 변수 및 산술식(열 이름 없이)만 포함되는 경우는 FROM 절이 필요 없지만, 테이블의 칼럼이 사용된 경우는 FROM 절이 필요하다. WHERE 절은 조회하려는 데이터에 특정 조건을 부여할 목적으로 사용하기 때문에 FROM 절 뒤에 오게 된다.

SELECT [DISTINCT/ALL] 칼럼명 [ALIAS 명] FROM 테이블명 WHERE 조건식;

WHERE 절은 FROM 절 다음에 위치하며, 조건식은 아래 내용으로 구성된다.

- 칼럼(Column)명 (보통 조건식의 좌측에 위치) - 비교 연산자 - 문자, 숫자, 표현식 (보통 조건식의 우측에 위치) - 비교 칼럼명 (JOIN 사용시)

### 2. 연산자의 종류

WHERE 절에 조건식을 사용할 때, 사용되는 비교 연산자에 대해서 살펴본다. 연산자에 대해서 알아보기 전에 위에서 나왔던 조건을 조금 더 복잡하게 바꾸어 본다.

K-리그 일부 선수들의 이름과 포지션, 백넘버를 알고 싶다. 조건은 소속팀이 삼성블루윙즈이거나 전남드래곤즈에 소속된 선수들 중에서 포지션이 미드필더(MF:Mid Fielder) 이면서, 키는 170 센티미터 이상, 180 이하여야 한다.

위의 요구 조건을 모두 만족하는 Query 문장을 구성하기 위해서는 다양한 연산자들을 사용해야만 한다. WHERE 절에 사용되는 연산자는 3 가지 종류가 있다.

- 비교 연산자 (부정 비교 연산자 포함) - SQL 연산자 (부정 SQL 연산자 포함) - 논리 연산자

[표 II-1-15] 연산자의 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.
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)
	NOT IN (list)	list 값과 일치하지 않는다.
	IS NOT NULL	NULL 값을 갖지 않는다.

[표 II-1-16] 연산자의 우선순위

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

연산자의 우선순위를 살펴보면 다음과 같다.

- 괄호로 묶은 연산이 제일 먼저 연산 처리된다. - 연산자들 중에는 부정 연산자(NOT)가 먼저 처리되고, - 비교 연산자(=, >, >=, <, <=), SQL 비교 연산자(BETWEEN a AND b, IN (list), LIKE, IS NULL)가 처리되고, - 논리 연산자 중에서는 AND, OR 의 순으로 처리된다.

만일 이러한 연산에 있어서 연산자들의 우선순위를 염두에 두지 않고 WHERE 절을 작성한다면 테이블에서 자기가 원하는 자료를 찾지 못하거나, 혹은 틀린 자료인지도 모른 채 사용할 수도 있다. 실수하기 쉬운 비교 연산자와 논리 연산자의 경우 괄호를 사용해서 우선순위를 표시하는 것을 권고한다.

### 3. 비교 연산자

비교 연산자의 종류는 [표 II-1-17]과 같으며, 비교 연산자들을 적절히 사용하여 다양한 조건을 구성할 수 있다.

[표 II-1-17] 비교 연산자의 종류

연산자	연산자의 의미
=	같다.
>	보다 크다.
>=	보다 크거나 같다.
<	보다 작다.
<=	보다 작거나 같다.

앞의 요구 사항을 다음과 같이 비교 연산자를 적용하여 표현할 수 있다.

소속팀이 삼성블루윙즈이거나 전남드래곤즈에 소속된 선수들이어야 하고, 포지션이 미드필더(MF:Midfielder)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

각각의 예를 보면 비교 연산자로 소속팀, 포지션, 키와 같은 칼럼(Column)들을 특정한 값들과 조건을 비교하는데 사용되는 것을 알 수 있다.

[예제] 첫 번째 요구 사항인 소속팀이 삼성블루윙즈라는 조건을 WHERE 조건절로 옮겨서 SQL 문장을 완성하여 실행한다.

[예제 및 실행 결과] SELECT PLAYER\_NAME 선수이름, POSITION 포지션, BACK\_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM\_ID = K02; WHERE TEAM\_ID = K02 \* 3 행에 오류: ERROR: 열명이 부적합하다.

실행 결과는 “열(COLUMN)명이 부적합하다.”라는 에러 메시지를 보이고 SQL 문장의 세 번째 줄에 오류가 있다고 나와 있다. TEAM\_ID 라는 팀명의 데이터 타입은 CHAR(3)인데 비교 연산자 오른쪽에 K02의 값을 작은따옴표(')나 큰따옴표(")와 같은 인용 부호로 묶어서 처리하지 않았기 때문에 발생하는 에러이다. CHAR 변수나 VARCHAR2와 같은 문자형 타입을 가진 칼럼을 특정 값과 비교하기 위해서는 인용 부호(작은따옴표, 큰따옴표)로 묶어서 비교 처리를 해야 한다. 하지만 NUMERIC과 같은 숫자형 형태의 값은 인용부호를 사용하지 않는다.

[예제] 첫 번째 요구 사항을 수정하여 다시 실행한다.

[예제] SELECT PLAYER\_NAME 선수이름, POSITION 포지션, BACK\_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM\_ID = 'K02' ;

[실행 결과] 선수이름 포지션 백넘버 키 ----- 김성환 DF 5 183 가비 MF 10 177 강대희 MF 26 174 고종수 MF 22 176 고창 F 4 175 정준 MF 44 170 정진우 DF 7 179 데니스 FW 11 176 서정원 FW 14 173 : : : : 49 개의 행이 선택되었다.

[예제] 세 번째 요구 사항인 포지션이 미드필더(MF)인 조건을 WHERE 조건절로 옮겨서 SQL 문장을 완성하여 실행한다.

[예제] SELECT PLAYER\_NAME 선수이름, POSITION 포지션, BACK\_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE POSITION = 'MF';

[실행 결과] 선수이름 포지션 백넘버 키 ----- 가비 MF 10 177 강대희 MF 26 174 고종수 MF 22 176 고창현 MF 8 170 정기범 MF 28 173 정동현 MF 25 175 정두현 MF 4 175 정준 MF 44 170 오규찬 MF 24 178 윤원일 MF 45 176 장성철 MF 27 176 : : : : 162 개의 행이 선택되었다.

추가적으로 문자 유형간의 비교 조건이 발생하는 경우는 [표 11-1-18]과 같이 처리한다.

[표 11-1-18] 문자 유형 비교 방법

구분	비교 방법
비교 연산자의 양쪽이 모두 CHAR 유형 타입인 경우	길이가 서로 다른 CHAR형 타입이면 작은 쪽에 SPACE를 추가하여 길이를 같게 한 후에 비교한다.
	서로 다른 문자가 나올 때까지 비교한다.
	달라진 첫 번째 문자의 값에 따라 크기를 결정한다.
	BLANK의 수만 다르다면 서로 같은 값으로 결정한다.
비교 연산자의 어느 한 쪽이 VARCHAR 유형 타입인 경우	서로 다른 문자가 나올 때까지 비교한다.
	길이가 다르다면 짧은 것이 끝날 때까지만 비교한 후에 길이가 긴 것이 크다고 판단한다.
	길이가 같고 다른 것이 없다면 같다고 판단한다.
	VARCHAR는 NOT NULL까지 길이를 말한다.
상수값과 비교할 경우	상수 쪽을 변수 타입과 동일하게 바꾸고 비교한다.
	변수 쪽이 CHAR 유형 타입이면 위의 CAHAR 유형 타입의 경우를 적용한다.
	변수 쪽이 VARCHAR 유형 타입이면 위의 VARCHAR 유형 타입의 경우를 적용한다.

[예제] 네 번째 요구 사항인 "키가 170 센티미터 이상"인 조건도 WHERE 절로 옮겨서 SQL 문장을 완성하여 실행한다.

[예제] SELECT PLAYER\_NAME 선수이름, POSITION 포지션, BACK\_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE HEIGHT >= 170;

[실행 결과] 선수이름 포지션 백넘버 키 ----- 김성환 DF 5 183 가비 MF 10 177 강대희 MF 26 174 고종수 MF 22 176 고창현 MF 8 170 정기범 MF 28 173 정동현 MF 25 175 정두현 MF 4 175 정준 MF 44 170 정진우 DF 7 179 데니스 FW 11 176 : : : : 439 개의 행이 선택되었다.

문자 유형 칼럼의 경우 WHERE TEAM\_ID = K02 사례에서 ' ' 표시가 없는 경우 에러가 발생하였지만, 숫자 유형 칼럼의 경우 숫자로 변환이 가능한 문자열(Alpha Numeric)과 비교되면 상대 타입을 숫자 타입으로 바꾸어 비교한다. 예를 들면 [예제]의 WHERE HEIGHT >= 170 조건을 WHERE HEIGHT >= '170' 이라고 표현하더라도, HEIGHT 라는 칼럼이 숫자 유형의 변수이므로 내부적으로 '170' 이라는 문자열을 숫자 유형 170 으로 바꾸어 처리한다.

## 4. SQL 연산자



SQL 연산자는 SQL 문장에서 사용하도록 기본적으로 예약되어 있는 연산자로서 모든 데이터 타입에 대해서 연산이 가능한 4 가지 종류가 있다.

[표 II-1-19] SQL 연산자의 종류

연산자	연산자의 의미
BETWEEN a AND b	a와 b의 값 사이에 있으면 된다.(a와 b의 값이 포함됨)
IN (list)	리스트에 있는 값 중에서 어느 하나라도 일치하면 된다.
LIKE '비교문자열'	비교 문자열과 형태가 일치하면 된다.
IS NULL	NULL 값인 경우

앞의 요구 사항을 다음과 같이 비교 연산자와 SQL 비교 연산자를 적용하여 표현할 수 있다.

1) 소속팀코드 IN (삼성블루윙즈 코드(K02), 전남드래곤즈 코드(K07)) 2) 포지션 LIKE 미드필더(MF) 3) 키 BETWEEN 170 센티미터 AND 180 센티미터

- IN (list) 연산자

[예제] 소속팀 코드와 관련된 IN (list) 형태의 SQL 비교 연산자를 사용하여 WHERE 절에 사용한다.

[예제] SELECT PLAYER\_NAME 선수이름, POSITION 포지션, BACK\_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM\_ID IN ('K02','K07');

[실행 결과] 선수이름 포지션 백넘버 키 ----- DE-NIS FW 11 176 서정원 FW 14 173 손대호 DF 17 186 오규찬 MF 24 178 윤원일 MF 45 176 김동욱 MF 40 176 김희택 DF 서현욱 DF 정상호 DF 최철우 DF 정영광 GK 41 185 : : : : 100 개의 행이 선택되었다.

[예제] 사원 테이블에서 JOB 이 MANAGER 이면서 20 번 부서에 속하거나, JOB 이 CLERK 이면서 30 번 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 2 개의 행이 선택되었다.

사용자들이 잘 모르고 있는 다중 리스트를 이용한 IN 연산자는 SQL 문장을 짧게 만들어 주면서도 성능 측면에서도 장점을 가질 수 있는 매우 유용한 연산자이므로 적극적인 사용을 권고한다. 다만, 아래 SQL 문장과는 다른 결과가 나오게 되므로 용도를 구분해서 사용해야 한다.

[예제] SELECT ENAME, JOB, DEPTNO FROM EMP WHERE JOB IN ('MANAGER','CLERK') AND DEPTNO IN (20,30);

[실행 결과] ENAME JOB DEPTNO ----- SMITH CLERK 20 JONES MANAGER 20 BLAKE MANAGER 30 ADAMS CLERK 20 JAMES CLERK 30 5 개의 행이 선택되었다.

- LIKE 연산자

[예제] 요구 사항의 두 번째 조건에 대해서 LIKE 연산자를 WHERE 절에 적용해서 실행한다.

[예제] SELECT PLAYER\_NAME 선수이름, POSITION 포지션, BACK\_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE POSITION LIKE 'MF';

[예제] SELECT PLAYER\_NAME 선수이름, POSITION 포지션, BACK\_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE POSITION LIKE 'MF';

[실행 결과] 선수이름 포지션 백넘버 키 ----- 가비 MF 10 177 강대희 MF 26 174 고종수 MF 22 176 고창현 MF 8 170 정기범 MF 28 173 정동현 MF 25 175 정두현 MF 4 175 정준 MF 44 170 : : : : 162 개의 행이 선택되었다.

LIKE 의 사전적 의미는 ‘~와 같다’이다. 따라서 위와 같은 경우라면 비교 연산자인 ‘=’을 사용해서 작성해도 같은 결과를 얻을 수 있을 것이다. 그러나 만약 “장”씨 성을 가진 선수들을 조회할 경우는 어떻게 할까? 이런 문제를 해결하기 위해서 LIKE 연산자에서는 와일드카드(WildCard)를 사용할 수 있다. 와일드카드(WildCard)란 한 개 혹은 0 개 이상의 문자를 대신해서 사용하기 위한 특수 문자를 의미하며, 이를 조합하여 사용하는 것도 가능하므로 SQL 문장에서 사용하는 스트링 (STRING) 값으로 용이하게 사용할 수 있다.

[표 II-1-20] 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.

[예제] SELECT PLAYER\_NAME 선수이름, POSITION 포지션, BACK\_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE PLAYER\_NAME LIKE '장%';

[실행 결과] 선수이름 포지션 백넘버 키 ----- 장성철 MF 27 176 장윤정 DF 17 173 장서원 FW 7 180 장재우 FW 12 172 장대일 DF 7 184 장기봉 FW 12 180 장철우 DF 7 172 장형석 DF 36 181 장경진 DF 34 184 장성욱 MF 19 174 장철민 MF 24 179 장경호 MF 39 174 장동현 FW 39 178 13 개의 행이 선택되었다.

- BETWEEN a AND b 연산자

[예제] 세 번째로 키가 170 센티미터 이상 180 센티미터 이하인 선수들의 정보를 BETWEEN a AND b 연산자를 사용하여 WHERE 절을 완성한다.

[예제] SELECT PLAYER\_NAME 선수이름, POSITION 포지션, BACK\_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE HEIGHT BETWEEN 170 AND 180; BETWEEN a AND b는 범위에서 'a'와 'b'의 값을 포함하는 범위를 말하는 것이다.

[실행 결과] 선수이름 포지션 백넘버 키 ----- 장철우 DF 7 172 홍광철 DF 4 172 강정훈 MF 38 175 공오균 MF 22 177 정국진 MF 16 172 정동선 MF 9 170 최경규 MF 10 177 최내철 MF 24 177 배성재 MF 28 178 샴 MF 25 174 김관우 MF 8 175 : : : : 259 개의 행이 선택되었다.

- IS NULL 연산자

NULL(ASCII 00)은 값이 존재하지 않는 것으로 확정되지 않은 값을 표현할 때 사용한다. 따라서 어떤 값보다 크거나 작지도 않고 ‘ ’(공백, ASCII 32)이나 0(Zero, ASCII 48)과 달리 비교 자체가 불가능한 값인 것이다. 연산 관련 NULL의 특성은 다음과 같다.

- NULL 값과의 수치연산은 NULL 값을 리턴한다. - NULL 값과의 비교연산은 거짓(FALSE)을 리턴한다. - 어떤 값과 비교할 수도 없으며, 특정 값보다 크다, 적다라고 표현할 수 없다.

따라서 NULL 값의 비교는 비교 연산자인 “=”, “>”, “>=”, “<”, “<=”를 통해서 비교할 수도 없고, 만일 비교 연산을 하게 되면 결과는 거짓(FALSE)을 리턴하고, 수치 연산자(+, -, \*, / 등)를 통해서 NULL 값과 연산을 하게 되면 NULL 값을 리턴한다. NULL 값의 비교 연산은 IS NULL, IS NOT NULL 이라는 정해진 문구를 사용해야 제대로 된 결과를 얻을 수 있다.

[예제 및 실행 결과] SELECT PLAYER\_NAME 선수이름, POSITION 포지션, BACK\_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE POSITION = NULL; 선택된 레코드가 없다.

[예제]의 실행 결과로 “선택된 레코드가 없다.”라는 메시지가 출력되었다. 앞에서 살펴본 대로 WHERE 절에서 POSITION = NULL 을 사용했는데 문법 에러가 나지는 않았지만 WHERE 절의 조건이 거짓(FALSE)이 되어 WHERE 절의 조건을 만족하는 데이터를 한건도 얻지 못하게 된 것으로 의미 없는 SQL 이 되고 말았다.

[예제] POSITION 칼럼(Column) 값이 NULL 값인지를 판단하기 위해서는 IS NULL 을 사용하여 다음과 같이 SQL 문장을 수정하여 실행한다.

[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 3 개의 행이 선택되었다.

5. 논리 연산자

논리 연산자는 비교 연산자나 SQL 비교 연산자들로 이루어진 여러 개의 조건들을 논리적으로 연결시키기 위해서 사용되는 연산자라고 생각하면 된다. [표 II-1-21]을 보고 실제로 적용되는 예를 통해 사용방법을 이해한다.

[표 II-1-21] 논리 연산자의 종류

연산자	연산자의 의미
AND	앞에 있는 조건과 뒤에 오는 조건이 참(TRUE)이 되면 결과도 참(TRUE)이 된다. 즉, 앞의 조건과 뒤의 조건을 동시에 만족해야 하는 것이다.
OR	앞의 조건이 참(TRUE)이거나 뒤의 조건이 참(TRUE)이 되면 결과도 참(TRUE)이 된다. 즉, 앞뒤의 조건 중 하나만 참(TRUE)이면 된다.
NOT	뒤에 오는 조건에 반대되는 결과를 되돌려 준다.

[예제] 예를 들어 “소속이 삼성블루윙즈”인 조건과 “키가 170 센티미터 이상”인 조건을 연결해 보면 “소속이 삼성블루윙즈이고 키가 170 센티미터 이상인 조건을 가진 선수들의 자료를 조회”하는 것이 되는 것이다.

[예제] SELECT PLAYER\_NAME 선수이름, POSITION 포지션, BACK\_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM\_ID = 'K02' AND HEIGHT >= 170;

[실행 결과] 선수이름 포지션 백넘버 키 ----- 김반코비 MF 47 185 김선우 FW 33 174 김여성 MF 36 179 김용우 FW 27 175 김종민 MF 30 174 박용훈 MF 9 175 김만근 FW 34 177 김재민 MF 35 180 김현두 MF 12 176 이성용 DF 20 173 하태근 MF 29 182 : : : : 45 개의 행이 선택되었다

[예제] “소속이 삼성블루윙즈이거나 전남드래곤즈”인 조건을 SQL 비교 연산자로, “포지션이 미드필더(MF)”인 조건을 비교 연산자로 비교한 결과를 논리 연산 bg\_gray>[예제] SELECT PLAYER\_NAME 선수이름, POSITION 포지션, BACK\_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM\_ID IN ('K02','K07') AND POSITION = 'MF';

[실행 결과] 선수이름 포지션 백넘버 키 ----- 노병준 MF 22 177 최종우 MF 43 176 조진원 MF 9 176 실바 MF 45 173 윤용구 MF 15 168 김반 MF 14 174 김영수 MF 30 175 임관식 MF 29 172 이정호 MF 23 176 하기운 MF 32 180 김반코비 MF 47 185 : : : : 40 개의 행이 선택되었다.

실행 결과를 보면 소속이 (삼성블루윙즈이거나 전남드래곤즈이고) 포지션이 미드필더(MF)인 선수들의 데이터가 조회되었음을 확인할 수 있다

[예제] 요구 사항을 하나씩 하나씩 AND, OR 같은 논리 연산자를 사용하여 DBMS가 이해할 수 있는 SQL 형식으로 질문을 변경한다. 요구 사항을 순서대로 논리적인 조건을 적용한다.

소속팀이 삼성블루윙즈이거나 전남드래곤즈에 소속된 선수들이어야 하고, 포지션이 미드필더(MF:Midfielder)이어야 한다. 키는 170 센티미터 이상이고 180 이하여야 한다. 1) 소속팀이 삼성블루윙즈 OR 소속팀이 전남드래곤즈 2) AND 포지션이 미드필더 3) AND 키는 170 센티미터 이상 4) AND 키는 180 센티미터 이하

[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;

[실행 결과] 선수이름 포지션 백넘버 키 ----- 김성환 DF 5 183 가비 MF 10 177 강대희 MF 26 174 고종수 MF 22 176 고창현 MF 8 170 정기범 MF 28 173 정동현 MF 25 175 정두현 MF 4 175 정준 MF 44 170 정진우 DF 7 179 데니스 FW 11 176 : : : : 66개의 행이 선택되었다.

실행 결과의 내용을 보면 포지션이 미드필더(MF: MidFielder)가 아닌 선수들의 명단이 출력되었다. 원하는 데이터는 삼성블루윙즈이거나 전남드래곤즈 중 포지션이 미드필더(MF: Midfielder)인 선수들에 대한 자료만 요청했는데 포지션이 DF 나 FW 인 선수가 같이 출력된 것이다. [예제]에서 “소속팀 코드가 삼성블루윙즈(K02) 이거나 전남드래곤즈(K07)”라는 조건을 만족하고 “포지션이 미드필더(MF)”인 조건을 동시에 만족해야 하는데, 위의 SQL 문장에서는 괄호가 누락됨으로서 OR 논리 연산자보다 AND 논리 연산자를 먼저 실행하기 때문에 잘못된 결과를 나타낸 것이다. 논리 연산자들이 여러 개가 같이 사용되었을 때의 처리 우선순위는 ( ), NOT, AND, OR 의 순서대로 처리된다.

[표 II-1-22] 논리 연산자의 순서 사례

잘못 해석한 예	조건에 맞게 올바르게 수정한 예
WHERE 소속팀 = 삼성블루윙즈 OR 소속팀 = 전남드래곤즈 AND 포지션 = 미드필더(MF) AND 키 >= 170 센티미터 AND 키 <= 180 센티미터	WHERE (소속팀 = 삼성블루윙즈 OR 소속팀 = 전남드래곤즈) AND 포지션 = 미드필더(MF) AND 키 >= 170 센티미터 AND 키 <= 180 센티미터

[예제] 잘못된 결과를 보여 준 SQL 문장을 괄호를 사용하여 다시 적용한다.

[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;

[실행 결과] 선수이름 포지션 백넘버 키 ----- 가비 MF 10 177 강대희 MF 26 174 고종수 MF 22 176 고창현 MF 8 170 정기범 MF 28 173 정동현 MF 25 175 정두현 MF 4 175 정준 MF 44 170 오규찬 MF 24 178 윤원일 MF 45 176 김동욱 MF 40 176 : : : : 33개의 행이 선택되었다.

[예제] IN (list)와 BETWEEN a AND b 연산자를 활용하여 같은 결과를 출력하는 SQL 문장을 작성한다. 두개의 SQL 문장은 DBMS 내부적으로 같은 프로세스를 거쳐 수행되므로 당연히 실행 결과도 같다.

[예제] SELECT PLAYER\_NAME 선수이름, POSITION 포지션, BACK\_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM\_ID IN ('K02','K07') AND POSITION = 'MF' AND HEIGHT BETWEEN 170 AND 180 ; : 33 개의 행이 선택되었다.

앞서 살펴본 SQL 비교 연산자인 'IN' 과 논리 연산자인 'OR' 은 결과도 같고 내부적으로 처리하는 방법도 같다. 즉, 소속팀이 삼성블루윙즈이거나 전남드래곤즈인 선수들을 조회할 때 WHERE 절에 TEAM\_ID = 'K02' OR TEAM\_ID = 'K07' 라는 논리 연산자 조건과 TEAM\_ID IN ( 'K02', 'K07' ) 라는 SQL 연산자 조건은 같은 기능이다. 그리고 “HEIGHT >= 170 AND HEIGHT <= 180” 라는 비



교 연산자 조건과 “HEIGHT BETWEEN 170 AND 180”이라는 SQL 비교 연산자 조건도 결과도 같고 내부적으로 처리되는 방법도 같은 기능이다.

## 6. 부정 연산자

비교 연산자, SQL 비교 연산자에 대한 부정 표현을 부정 논리 연산자, 부정 SQL 연산자로 구분할 수 있다.

[표 II-1-23] 부정 연산자 종류

종류	연산자	연산자의 의미
부정 논리 연산자	!=	같지 않다.
	^=	같지 않다.
	◇	같지 않다. (ANIS/ISO 표준, 모든 운영체제에서 사용가능)
	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, b값을 포함하지 않는다)
	NOT IN (list)	list 값과 일치하지 않는다.
	IS NOT NULL	NULL 값을 갖지 않는다.

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

[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] Oracle 위의 SQL 과 아래 SQL 은 같은 내용을 나타내는 SQL 이다. SELECT PLAYER\_NAME 선수이름, POSITION 포지션, BACK\_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM\_ID = 'K02' AND POSITION ◇ 'MF' AND HEIGHT NOT BETWEEN 175 AND 185;

[실행 결과] 선수이름 포지션 백넘버 키 ----- 서정원 FW 14 173 손대호 DF 17 186 김선우 FW 33 174 이성용 DF 20 173 미트로 FW 19 192 최호진 GK 31 190 정유진 DF 37 188 손승준 DF 32 186 8 개의 행이 선택되었다.

[예제] 국적(NATION) 칼럼의 경우 내국인들은 별도 데이터를 입력하지 않았다. 국적 칼럼이 NULL 이 아닌 선수와 국적을 표시하라.

[예제] SELECT PLAYER\_NAME 선수이름, NATION 국적 FROM PLAYER WHERE NATION IS NOT NULL;

[실행 결과] 선수이름 국적 ----- 가비 루마니아 데니스 러시아 우르모브 유고 이고르 브라질 디디 브라질 하리 콜롬비아 빅토르 나이지리아 콜리 세네갈 김징요 브라질 미트로 보스니아 산드로 브라질 안드레 브라질 뚜파 브라질 마르코 브라질 히카르도 브라질 끌레베르 브라질 에디 브라질 마르코스 브라질 알리송 브라질 파울링뇨 브라질 제프유 미국 롤란 리투아니아 셀라하틴 김탈리아 올리베 브라질 김리네 브라질 야스민 크로아티아 코샤 브라질 27 개의 행이 선택되었다.

## 7. ROWNUM, TOP 사용

### • ROWNUM

Oracle 의 ROWNUM 은 칼럼과 비슷한 성격의 Pseudo Column 으로써 SQL 처리 결과 집합의 각 행에 대해 임시로 부여되는 일련번호이며, 테이블이나 집합에서 원하는 만큼의 행만 가져오고 싶을 때 WHERE 절에서 행의 개수를 제한하는 목적으로 사용한다.

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

추가적인 ROWNUM 의 용도로는 테이블 내의 고유한 키나 인덱스 값을 만들 수 있다. - UPDATE MY\_TABLE SET COLUMN1 = ROWNUM;

- TOP 절

SQL Server 는 TOP 절을 사용하여 결과 집합으로 출력되는 행의 수를 제한할 수 있다. TOP 절의 표현식은 다음과 같다.

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 지정할 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.

SQL 문장에서 ORDER BY 절이 사용되지 않으면 Oracle 의 ROWNUM 과 SQL Server 의 TOP 절은 같은 기능을 하지만, ORDER BY 절이 같이 사용되면 기능의 차이가 발생한다. 이 부분은 1 장 8 절 ORDER BY 절에서 설명하도록 한다.

## 06.함수(FUNCTION)

### 1. 내장 함수(BUILT-IN FUNCTION) 개요

함수는 다양한 기준으로 분류할 수 있는데, 벤더에서 제공하는 함수인 내장 함수(Built-in Function)와 사용자가 정의할 수 있는 함수(User Defined Function)로 나눌 수 있다. 본 절에서는 각 벤더에서 제공하는 데이터베이스를 설치하면 기본적으로 제공되는 SQL 내장 함수에 대해 설명한다.

내장 함수는 SQL 을 더욱 강력하게 해주고 데이터 값을 간편하게 조작하는데 사용된다. 내장 함수는 벤더별로 가장 큰 차이를 보이는 부분이지만, 핵심적인 기능들은 이름이나 표현법이 다르더라도 대부분의 데이터베이스가 공통적으로 제공하고 있다. 내장 함수는 다시 함수의 입력 값이 단일행 값이 입력되는 단일행 함수(Single-Row Function)와 여러 행의 값이 입력되는 다중행 함수(Multi-Row Function)로 나눌 수 있다. 다중행 함수는 다시 집계 함수(Aggregate Function), 그룹 함수(Group Function), 윈도우 함수(Window Function)로 나눌 수 있는데, 집계 함수는 다음 절에서, 그룹 함수는 2 장 5 절에서, 윈도우 함수는 2 장 6 절에서 설명하도록 하고 본 절에서는 단일행 함수에 대해서만 설명한다. 함수는 입력되는 값이 아무리 많아도 출력은 하나만 된다는 M:1 관계라는 중요한 특징을 가지고 있다. 단일행 함수의 경우 단일행 내에 있는 하나의 값 또는 여러 값이 입력 인수로 표현될 수 있다. 다중행 함수의 경우도 여러 레코드의 값들을 입력 인수로 사용하는 것이다.

함수명 (칼럼이나 표현식 [, Arg1, Arg2, ... ])

단일행 함수는 처리하는 데이터의 형식에 따라서 문자형, 숫자형, 날짜형, 변환형, NULL 관련 함수로 나눌 수 있다. 벤더에서 제공하는 내장 함수는 상당히 종류가 많고 벤더별로 사용법이 틀린 경우가 많아, 본 절에서는 Oracle 과 SQL Server 에서 공통으로 사용하는 중요 함수 위주로 설명을 한다. 함수에 대한 자세한 내용이나 버전에 따른 변경 내용은 벤더에서 제공하는 매뉴얼을 참조하기 바란다. 아래 함수의 예에서 SUBSTR / SUBSTRING 으로 표시한 것은 같은 기능을 하지만 다르게 표현되는 Oracle 내장 함수와 SQL Server 내장 함수를 순서대로 표현한 것이다.

[표 II-1-24] 단일행 함수의 종류

종류	내용	함수의 예
문자형 함수	문자를 입력하면 문자나 숫자 값을 반환한다.	LOWER, UPPER, 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,'YYYY' 'MM' 'DD')) / YEAR MONTH DAY
변환형 함수	문자, 숫자, 날짜형 값의 데이터 타입을 변환한다.	TO_NUMBER, TO_CHAR, TO_DATE / CAST, CONVERT
NULL 관련 함수	NULL을 처리하기 위한 함수	NVL/ISNULL, NULLIF, COALESCE

※ 주: Oracle함수/SQL Server함수 표시, '/' 없는 것은 공통 함수

단일행 함수의 중요한 특징은 다음과 같다.

- SELECT, WHERE, ORDER BY 절에 사용 가능하다. - 각 행(Row)들에 대해 개별적으로 작용하여 데이터 값들을 조작하고, 각각의 행에 대한 조작 결과를 리턴한다. - 여러 인자(Argument)를 입력해도 단 하나의 결과만 리턴한다. - 함수의 인자(Arguments)로 상수, 변수, 표현식이 사용 가능하고, 하나의 인수를 가지는 경우도 있지만 여러 개의 인수를 가질 수도 있다. - 특별한 경우가 아니면 함수의 인자(Arguments)로 함수를 사용하는 함수의 중첩이 가능하다.

## 2. 문자형 함수

문자형 함수는 문자 데이터를 매개 변수로 받아들여서 문자나 숫자 값의 결과를 돌려주는 함수이다. 몇몇 문자형 함수의 경우는 결과를 숫자로 리턴하는 함수도 있다.

[표 II-1-25] 단일행 문자형 함수의 종류

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)	Oracle, My SQL에서 유효한 함수이며 문자열1과 문자열2를 연결한다. 합성 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) SQL Server에서는 LTRIM 함수에 지정문자를 사용할 수 없다. 즉, 공백만 제거할 수 있다.
RTRIM (문자열 [, 지정문자 ])	문자열의 마지막 문자부터 확인해서 지정 문자가 나타나는 동안 해당 문자를 제거한다.(지정 문자가 생략되면 공백 값이 디폴트) SQL Server에서는 LTRIM 함수에 지정문자를 사용할 수 없다. 즉, 공백만 제거할 수 있다.
TRIM ([leading trailing both] 지정문자 FROM 문자열)	문자열에서 머리말, 꼬리말, 또는 양쪽에 있는 지정 문자를 제거한다. (leading   trailing   both 가 생략되면 both가 디폴트) SQL Server에서는 TRIM 함수에 지정문자를 사용할 수 없다. 즉, 공백만 제거할 수 있다.

※ 주: Oracle함수/SQL Server함수 표시, '/' 없는 것은 공통 함수

- 문자형 함수들이 적용되었을 때 리턴되는 값을 예를 들어 설명한다.



[표 II -1-26] 단일행 문자형 함수 사례

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' / 'RDBMS' + ' SQL'	'RDBMS SQL'
SUBSTR('SQL Expert', 5, 3) SUBSTRING('SQL Expert', 5, 3)	'Exp'
LENGTH('SQL Expert') / 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'

[예제] 'SQL Expert' 라는 문자형 데이터의 길이를 구하는 문자형 함수를 사용한다.

[예제 및 실행 결과] Oracle SELECT LENGTH('SQL Expert') FROM DUAL; LENGTH('SQL Expert') -----  
----- 10

예제 및 실행 결과를 보면 함수에 대한 결과 값을 마치 테이블에서 값을 조회했을 때와 비슷하게 표현한다. Oracle 은 SELECT 절과 FROM 절 두 개의 절을 SELECT 문장의 필수 절로 지정하였으므로 사용자 테이블이 필요 없는 SQL 문장의 경우에도 필수적으로 DUAL 이라는 테이블을 FROM 절에 지정한다. DUAL 테이블의 특성은 다음과 같다.

- 사용자 SYS 가 소유하며 모든 사용자가 액세스 가능한 테이블이다. - SELECT ~ FROM ~ 의 형식을 갖추기 위한 일종의 DUMMY 테이블이다. - DUMMY 라는 문자열 유형의 칼럼에 'X'라는 값이 들어 있는 행을 1건 포함하고 있다.

[예제 및 실행 결과] Oracle DESC DUAL; 칼럼 NULL 가능 데이터 유형 -----  
----- DUMMY VARCHAR2(1)

[예제 및 실행 결과] Oracle SELECT \* FROM DUAL; DUMMY ----- X 1 개의 행이 선택되었다.

반면 Sybase 나 SQL Server 의 경우에는 SELECT 절만으로도 SQL 문장이 수행 가능하도록 정의하였기 때문에 DUAL 이란 DUMMY 테이블이 필요 없다. 그러나 Sybase 나 SQL Server 의 경우에도 사용자 테이블의 칼럼을 사용할 때는 FROM 절이 필수적으로 사용되어야 한다.

[예제] 'SQL Expert' 라는 문자형 데이터의 길이를 구하는 문자형 함수를 사용한다.

[예제 및 실행 결과] Oracle SELECT LEN('SQL Expert') AS ColumnLength; ColumnLength -----  
10

[예제] 선수 테이블에서 CONCAT 문자형 함수를 이용해 축구선수란 문구를 추가한다.



[예제] SELECT CONCAT(PLOYER\_NAME, ' 축구선수') 선수명 FROM PLOYER; CONCAT 함수는 Oracle 의 '||' 합성 연산자와 같은 기능이다. SELECT PLOYER\_NAME || ' 축구선수' AS 선수명 FROM PLOYER;

SQL Server 에서 위의 예제와 같은 결과를 얻으려면 아래와 같이 수행하면 된다.

[예제] SQL Server SELECT PLOYER\_NAME + ' 축구선수' AS 선수명 FROM PLOYER;

[실행 결과] PLOYER\_ID 선수명 ----- 2011075 김성환 축구선수 2012123 가비 축구선수 2010089 강대희 축구선수 2007051 고종수 축구선수 2012015 고창현 축구선수 2009089 정기범 축구선수 2009083 정동현 축구선수 2011071 정두현 축구선수 2012025 정준 축구선수 2007040 정진우 축구선수 2007069 테니스 축구선수 2007274 서정원 축구선수 480 개의 행이 선택되었다.

실행 결과를 보면 실제적으로 함수가 모든 행에 대해 적용되어 '~ 축구선수' 라는 각각의 결과로 출력되었다. 특별한 제약 조건이 없다면 함수는 여러 개 중첩하여 사용이 가능하다. 함수 내부에 다른 함수를 사용하며 안쪽에 위치해 있는 함수부터 실행되어 그 결과 값이 바깥쪽의 함수에 인자 (Argument)로 사용되는 것이다.

함수 3 (함수 2 (함수 1 (칼럼이나 표현식 [, Arg1]) [, Arg2]) [, Arg3 ])

[예제] 경기장의 지역번호와 전화번호를 합친 번호의 길이를 구하시오. 연결연산자의 결과가 LENGTH(SQL Server 는 LEN 사용) 함수의 인수가 된다.

[예제] Oracle SELECT STADIUM\_ID, DDD||TEL as TEL, LENGTH(DDD||TEL) as T\_LEN FROM STADIUM;

[예제] SQL Server SELECT STADIUM\_ID, DDD+TEL as TEL, LEN(DDD+TEL) as T\_LEN FROM STADIUM;

[실행 결과] STADIUM\_ID TEL T\_LEN ----- D03 063273-1763 11 B02 031753-3956 11 C06 054282-2002 11 D01 061792-5600 11 B05 022128-2973 11 B01 031666-0496 11 C05 0556644-8468 12 C04 052220-2468 11 D02 042252-2002 11 B04 031259-2150 11 A02 0622468-8642 12 C02 051247-5771 11 A03 033459-3631 11 A04 0643631-2460 12 A05 053602-2011 11 F01 054 3 F02 051 3 F03 031 3 F04 055 3 F05 031 3 20 개의 행이 선택되었다.

### 3. 숫자형 함수

숫자형 함수는 숫자 데이터를 입력받아 처리하고 숫자를 리턴하는 함수이다.

[표 II-1-27] 단일행 숫자형 함수 종류

숫자형 함수	함수 설명
ABS(숫자)	숫자의 절대값을 돌려준다.
SIGN(숫자)	숫자가 양수인지, 음수인지 0인지를 구별한다.
MOD(숫자1, 숫자2)	숫자1을 숫자2로 나누어 나머지 값을 리턴한다. MOD 함수는 % 연산자로도 대체 가능함 (ex:7%3)
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 0이다.
TRUNC(숫자 [, m])	숫자를 소수 m자리에서 잘라서 버린다. m이 생략되면 디폴트 값은 0이다. SQL SERVER에서 TRUNC 함수는 제공되지 않는다.
SIN, COS, TAN,...	숫자의 삼각함수 값을 리턴한다.
EXP(), POWER(), SQRT(), LOG(), LN()	숫자의 지수, 거듭 제곱, 제곱근, 자연 로그 값을 리턴한다.

※ 주: Oracle 함수/SQL Server 함수 표시 ‘/’ 없는 것은 공통 함수

- 숫자형 함수들이 적용되었을 때 리턴되는 값을 예를 들어 설명한다.

[표 II-1-28] 단일행 숫자형 함수 사례

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

[예제] 소수점 이하 한 자리까지 반올림 및 내림하여 출력한다.

[예제] SQL Server SELECT ENAME, ROUND(SAL/12,1), TRUNC(SAL/12,1) FROM EMP;

[실행 결과] ENAME ROUND(SAL/12,I) TRUNC(SAL/12,I) ----- SMITH  
66.7 66.6 ALLEN 133.3 133.3 WARD 104.2 104.1 JONES 247.9 247.9 MARTIN 104.2 104.1 BLAKE 237.5  
237.5 CLARK 204.2 204.1 SCOTT 250 250 KING 416.7 416.6 TURNER 125 125 ADAMS 91.7 91.6 JAMES  
79.2 79.1 FORD 250 250 MILLER 108.3 108.3 14 개의 행이 선택되었다.

[예제] 정수 기준으로 반올림 및 올림하여 출력한다.

[예제] SQL Server SELECT ENAME, ROUND(SAL/12), CEILING(SAL/12) FROM EMP;

[실행 결과] ENAME ROUND(SAL/12) CEILING(SAL/12) ----- SMITH  
67 67 ALLEN 133 134 WARD 104 105 JONES 248 248 MARTIN 104 105 BLAKE 238 238 CLARK 204  
205 SCOTT 250 250 KING 417 417 TURNER 125 125 ADAMS 92 92 JAMES 79 80 FORD 250 250  
MILLER 108 109 14 개의 행이 선택되었다.

## 4. 날짜형 함수

날짜형 함수는 DATE 타입의 값을 연산하는 함수이다. Oracle 의 TO\_NUMBER(TO\_CHAR( )) 함수의 경우 변환형 함수로 구분할 수도 있으나 SQL Server 의 YEAR, MONTH, DAY 함수와 매핑하기 위하여 날짜형 함수에서 설명한다. EXTRACT/DATEPART 는 같은 기능을 하는 Oracle 내장 함수와 SQL Server 내장 함수를 표현한 것이다

[표 II-1-29] 단일행 날짜형 함수 종류

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), TO_NUMBER(TO_CHAR(d, 'MM')) / MONTH(d), TO_NUMBER(TO_CHAR(d, 'DD')) / DAY(d)	날짜 데이터에서 년/월/일 데이터를 출력할 수 있다. Oracle EXTRACT YEAR/MONTH/DAY 옵션이나 SQL Server DEPART YEAR/MONTH/DAY 옵션과 같은 기능이다. TO_NUMBER 함수 제외시 문자형으로 출력됨

※ 주: Oracle 함수/SQL Server 함수 표시, '/' 없는 것은 공통 함수

DATE 변수가 데이터베이스에 어떻게 저장되는지 살펴보면, 데이터베이스는 날짜를 저장할 때 내부적으로 세기(Century), 년(Year), 월(Month), 일(Day), 시(Hours), 분(Minutes), 초(Seconds)와 같은 숫자 형식으로 변환하여 저장한다. 날짜는 여러 가지 형식으로 출력이 되고 날짜 계산에도 사용되기 때문에 그 편리성을 위해서 숫자형으로 저장하는 것이다. 데이터베이스는 날짜를 숫자로 저장하기 때문에 덧셈, 뺄셈 같은 산술 연산자로도 계산이 가능하다. 즉, 날짜에 숫자 상수를 더하거나 빼 수 있다.

[표 II-1-30] 단일행 날짜형 데이터 연산

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

[예제] Oracle 의 SYSDATE 함수와 SQL Server 의 GETDATE( ) 함수를 사용하여 데이터베이스에서 사용하는 현재의 날짜 데이터를 확인한다. 날짜 데이터는 시스템 구성에 따라 다양하게 표현될 수 있으므로 사용자마다 다른 결과가 나올 수 있다.

[예제 및 실행 결과] Oracle SELECT SYSDATE FROM DUAL; SYSDATE ----- 12/07/18

[예제 및 실행 결과] SQL Server SELECT GETDATE() AS CURRENTTIME; CURRENTTIME -----  
----- 2012-07-18 13:10:02.047

[예제] 사원(EMP) 테이블의 입사일에서 년, 월, 일 데이터를 각각 출력한다. 아래 4 개의 SQL 문장은 같은 기능을 하는 SQL 문장이다.

[예제] Oracle 함수 SELECT EXTRACT(YEAR FROM HIREDATE) 입사년도, EXTRACT(MONTH FROM HIREDATE) 입사월, EXTRACT(DAY FROM 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; TO\_NUMBER 함수 제외시 문자형으로 출력됨 (ex: 01,02,03,...)

[예제] SQL Server 함수 SELECT ENAME, HIREDATE, DATEPART(YEAR, HIREDATE) 입사년도, DATEPART(MONTH, HIREDATE) 입사월, DATEPART(DAY, HIREDATE) 입사일 FROM EMP;

[예제] SQL Server 함수 SELECT ENAME, HIREDATE, YEAR(HIREDATE) 입사년도, MONTH(HIREDATE) 입사월, DAY(HIREDATE) 입사일 FROM EMP;

[실행 결과] ENAME HIREDATE 입사년도 입사월 입사일 -----  
SMITH 1980-12-17 1980 12 17 ALLEN 1981-02-20 1981 2 20 WARD 1981-02-22 1981 2 22 JONES 1981-04-02 1981 4 2 MARTIN 1981-09-28 1981 9 28 BLAKE 1981-05-01 1981 5 1 CLARK 1981-06-09 1981 6 9 SCOTT 1987-07-13 1987 7 13 KING 1981-11-17 1981 11 17 TURNER 1981-09-08 1981 9 8 ADAMS 1987-07-13 1987 7 13 JAMES 1981-12-03 1981 12 3 FORD 1981-12-03 1981 12 3 MILLER 1982-01-23 1982 1 23 14 개의 행이 선택되었다.

## 5. 변환형 함수

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

[표 II-1-31] 데이터 유형 변환의 종류

종류	설명
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 성능 저하가 발생할 수 있으며, 자동적으로 데이터베이스가 알아서 계산하지 않는 경우가 있어 에러를 발생할 수 있으므로 명시적인 데이터 유형 변환 방법을 사용하는 것이 바람직하다.

명시적 데이터 유형 변환에 사용되는 대표적인 변환형 함수는 다음과 같다.



[표 II-1-32] 단일행 변환형 함수의 종류

변환형 함수 - Oracle	함수 설명
TO_NUMBER(문자열)	alphanumeric 문자열을 숫자로 변환한다.
TO_CHAR(숫자 날짜 [, FORMAT])	숫자나 날짜를 주어진 FORMAT 형태로 문자열 타입으로 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 함수를 사용하여 출력 형식을 지정할 때, 숫자형과 날짜형의 경우 상당히 많은 포맷이 벤더별로 제공된다. 벤더별 데이터 유형과 함께 데이터 출력의 포맷 부분은 벤더의 고유 항목이 많으므로 매뉴얼을 참고하기 바라며, 아래는 대표적인 사례 몇 가지만 소개한다. [예제] 날짜를 정해진 문자 형태로 변형한다.

[예제 및 실행 결과] Oracle SELECT TO\_CHAR(SYSDATE, 'YYYY/MM/DD') 날짜, TO\_CHAR(SYSDATE, 'YYYY, MON, DAY') 문자형 FROM DUAL; 날짜 문자형 ----- 2012-07-19 2012, 7 월 , 일요일

[예제 및 실행 결과] SQL Server SELECT CONVERT(VARCHAR(10),GETDATE(),111) AS CURRENTDATE CURRNETDATE ----- 2012/07/19

[예제] 금액을 달러와 원화로 표시한다.

[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 \123,456,789 두 번째 칼럼의 L999에서 L은 로칼 화폐 단위를 의미한다.

[예제] 팀(Team) 테이블의 ZIP 코드 1과 ZIP 코드 2를 숫자로 변환한 후 두 항목을 더한 숫자를 출력한다.

[예제] Oracle SELECT TEAM\_ID, TO\_NUMBER(ZIP\_CODE1,'999') + TO\_NUMBER(ZIP\_CODE2,'999') 우편번호합 FROM TEAM;

[실행 결과] Oracle TEAM\_ID 우편번호합 ----- K05 750 K08 592 K03 840 K07 554 K09 359 K04 838 K11 333 K01 742 K10 331 K02 660 K12 869 K06 620 K13 777 K14 1221 K15 1665 15개의 행이 선택되었다.

[예제 및 실행 결과] SQL Server SELECT TEAM\_ID, CAST(ZIP\_CODE1 AS INT) + CAST(ZIP\_CODE2 AS INT) 우편번호합 FROM TEAM;

[실행 결과] SQL Server TEAM\_ID 우편번호합 ----- K05 750 K08 592 K03 840 K07 554 K09 359 K04 838 K11 333 K01 742 K10 331 K02 660 K12 869 K06 620 K13 777 K14 1221 K15 1665 15개의 행이 선택되었다.

## 6. CASE 표현

CASE 표현은 IF-THEN-ELSE 논리와 유사한 방식으로 표현식을 작성해서 SQL의 비교 연산 기능을 보완하는 역할을 한다. ANSI/ISO SQL 표준에는 CASE Expression이라고 표시되어 있는데, 함



수와 같은 성격을 가지고 있으며 Oracle 의 Decode 함수와 같은 기능을 하므로 단일행 내장 함수에서 같이 설명을 한다.

[예제] 일반 프로그램의 IF-THEN-ELSE-END 로직과 같다. 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;

[실행 결과] ENAME REVISED\_SALARY ----- SMITH 2000 ALLEN 2000 WARD 2000 JONES 2975 MARTIN 2000 BLAKE 2850 CLARK 2450 SCOTT 3000 KING 5000 TURNER 2000 ADAMS 2000 JAMES 2000 FORD 3000 MILLER 2000 14 개의 행이 선택되었다.

CASE 표현을 하기 위해서는 조건절을 표현하는 두 가지 방법이 있고, Oracle 의 경우 DECODE 함수를 사용할 수도 있다.

[표 II-1-33] 단일행 CASE 표현의 종류

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.

IF-THEN-ELSE 논리를 구현하는 CASE Expressions 은 Simple Case Expression 과 Searched Case Expression 두 가지 표현법 중에 하나를 선택해서 사용하게 된다.

CASE SIMPLE\_CASE\_EXPRESSION 조건 or SEARCHED\_CASE\_EXPRESSION 조건 ELSE 표현절 END

첫 번째 SIMPLE\_CASE\_EXPRESSION 은 CASE 다음에 바로 조건에 사용되는 칼럼이나 표현식을 표시하고, 다음 WHEN 절에서 앞에서 정의한 칼럼이나 표현식과 같은지 아닌지 판단하는 문장으로 EQUI(=) 조건만 사용한다면 SEARCHED\_CASE\_EXPRESSION 보다 간단하게 사용할 수 있는 장점이 있다. Oracle 의 DECODE 함수와 기능면에서 동일하다.

CASE EXPR WHEN COMPARISON\_EXPR THEN RETURN\_EXPR ELSE 표현절 END

[예제] 부서 정보에서 부서 위치를 미국의 동부, 중부, 서부로 구분하라.

[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;

[실행 결과] LOC AREA ----- NEW YORK EAST DALLAS CENTER CHICAGO CENTER BOSTON EAST 4 개의 행이 선택되었다.

두 번째 SEARCHED\_CASE\_EXPRESSION 은 CASE 다음에는 칼럼이나 표현식을 표시하지 않고, 다음 WHEN 절에서 EQUI(=) 조건 포함 여러 조건(>, >=, <, <=)을 이용한 조건절을 사용할 수 있기 때문에 SIMPLE\_CASE\_EXPRESSION 보다 훨씬 다양한 조건을 적용할 수 있는 장점이 있다.

CASE WHEN CONDITION THEN RETURN\_EXPR ELSE 표현절 END

[예제] 사원 정보에서 급여가 3000 이상이면 상등급으로, 1000 이상이면 중등급으로, 1000 미만이면 하등급으로 분류하라.

[예제] SELECT ENAME, CASE WHEN SAL >= 3000 THEN 'HIGH' WHEN SAL >= 1000 THEN 'MID' ELSE 'LOW' END AS SALARY\_GRADE FROM EMP;

[실행 결과] ENAME SALARY\_GRADE ----- SMITH LOW ALLEN MID WARD MID JONES MID MARTIN MID BLAKE MID CLARK MID SCOTT HIGH KING HIGH TURNER MID ADAMS MID JAMES LOW FORD HIGH MILLER MID 14 개의 행이 선택되었다.

CASE 표현은 함수의 성질을 가지고 있으므로, 다른 함수처럼 중첩해서 사용할 수 있다

[예제] 사원 정보에서 급여가 2000 이상이면 보너스를 1000 으로, 1000 이상이면 5000 으로, 1000 미만이면 0 으로 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;

[실행 결과] ENAME SAL BONUS ----- SMITH 800 0 ALLEN 1600 500 WARD 1250 500 JONES 2975 1000 MARTIN 1250 500 BLAKE 2850 1000 CLARK 2450 1000 SCOTT 3000 1000 KING 5000 1000 TURNER 1500 500 ADAMS 1100 500 JAMES 950 0 FORD 3000 1000 MILLER 1300 500 14 개의 행이 선택되었다.

## 7. NULL 관련 함수

### 가. NVL/ISNULL 함수

다시 한 번 NULL 에 대한 특성을 정리한다.

- 널 값은 아직 정의되지 않은 값으로 0 또는 공백과 다르다. 0 은 숫자이고, 공백은 하나의 문자이다. - 테이블을 생성할 때 NOT NULL 또는 PRIMARY KEY 로 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. - 결과값을 NULL 이 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.

[표 II-1-34] NULL 포함 연산의 결과

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

NVL/ISNULL 함수를 유용하게 사용하는 예는 산술적인 계산에서 데이터 값이 NULL 일 경우이다. 칼럼 간 계산을 수행하는 경우 NULL 값이 존재하면 해당 연산 결과가 NULL 값이 되므로 원하는 결과를 얻을 수 없는 경우가 발생한다. 이런 경우는 NVL 함수를 사용하여 숫자인 0(Zero)으로 변환을 시킨 후 계산을 해서 원하는 데이터를 얻는다. 관계형 데이터베이스의 중요한 데이터인 NULL 을 처리하는 주요 함수는 다음과 같다.

[표 II-1-35] 단일행 NULL 관련 함수의 종류

일반형 함수	함수 설명
NVL(표현식1, 표현식2) / 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.

※ 주: Oracle함수/SQL Server함수 표시, '/' 없는 것은 공통 함수

Oracle 의 경우 NVL 함수를 사용한다.

NVL (NULL 판단 대상, 'NULL 일 때 대체값' )

[예제 및 실행 결과] Oracle SELECT NVL(NULL, 'NVL-OK') NVL\_TEST FROM DUAL; NVL\_TEST -----  
- NVL-OK 1 개의 행이 선택되었다.

[예제 및 실행 결과] Oracle SELECT NVL('Not-Null', 'NVL-OK') NVL\_TEST FROM DUAL; NVL\_TEST ----  
---- Not-Null 1 개의 행이 선택되었다.

SQL Server 의 경우 ISNULL 함수를 사용한다.

ISNULL (NULL 판단 대상, 'NULL 일 때 대체값' )

[예제 및 실행 결과] SQL Server SELECT ISNULL(NULL, 'NVL-OK') ISNULL\_TEST ; ISNULL\_TEST -----  
---- NVL-OK 1 개의 행이 선택되었다.

[예제 및 실행 결과] SQL Server SELECT ISNULL('Not-Null', 'NVL-OK') ISNULL\_TEST ; ISNULL\_TEST -  
----- Not-Null 1 개의 행이 선택되었다.

[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 SELEC PLAYER\_NAME 선수명, POSITION, ISNULL(POSITION,'없음') 포지션 FROM  
PLAYER WHERE TEAM\_ID = 'K08'

[예제] NVL 함수와 ISNULL 함수를 사용한 SQL 문장은 벤더 공통적으로 CASE 문장으로 표현할 수 있다

[예제] SQL Server SELECT PLAYER\_NAME 선수명, POSITION, CASE WHEN POSITION IS NULL THEN '없음' ELSE POSITION END AS 포지션 FROM PLAYER WHERE TEAM\_ID = 'K08'

[실행 결과] 선수명 POSITION 포지션 ----- 차경복 DF DF 정학범 없음 안익수 없음  
차상광 없음 권찬수 GK GK 정경두 GK GK 정해운 GK GK 양영민 GK GK 가이모토 DF DF 정두영 DF DF  
정명희 DF DF 정영철 DF DF 박치국 MF MF 정상식 MF MF 서관수 FW FW 김성운 FW FW 김정운 FW  
FW 장동현 FW FW 45 개의 행이 선택되었다.

[예제] 급여와 커미션을 포함한 연봉을 계산하면서 NVL 함수의 필요성을 알아본다.

[예제] SELECT ENAME 사원명, SAL 월급, COMM 커미션, (SAL \* 12) + COMM 연봉 A, (SAL \* 12) +  
NVL(COMM,0) 연봉 B FROM EMP;

[실행 결과] 사원명 월급 커미션 연봉 A 연봉 B ----- 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 14 개의 행이 선택되었다.

실행 결과에서 월급에 커미션을 더해서 연봉을 계산하는 산술식이 있을 때 커미션에 NULL 값이 있는 경우 커미션 값에 NVL() 함수를 사용하지 않으면 연봉 A 의 계산 결과가 NULL 이 되어서 잘못된 계산한 결과를 확인할 수 있다. 따라서 연봉 B 결과와 같이 NVL(COMM,0)처럼 NULL 값을 0 으로 변환하여 연봉을 계산해야 하는 것이다. 물론 곱셈을 사용해야 하는 경우에는 NVL(COMM,1)을 해야 한다. 그러나 NVL 함수를 다중행 함수의 인자로 사용하는 경우는 오히려 불필요한 부하를 발생할 수 있으므로 굳이 NVL 함수를 사용할 필요가 없다. 다중행 함수는 입력 값으로 전체 건수가 NULL 값인 경우만 함수의 결과가 NULL 이 나오고 전체 건수 중에서 일부만 NULL 인 경우는 다중행 함수의 대상에서 제외한다. 예를 들면 100 명 중 10 명의 성적이 NULL 값일 때 평균을 구하는 다중행 함수 AVG 를 사용하면 NULL 값이 아닌 90 명의 성적에 대해서 평균값을 구하게 된다. 자세한 내용은 1 장 7 절에서 추가로 설명한다.

## 나. NULL 과 공집합

- 일반적인 NVL/ISNULL 함수 사용

STEP1. 정상적으로 매니저 정보를 가지고 있는 SCOTT 의 매니저를 출력한다.

[예제 및 실행 결과] SELECT MGR FROM EMP WHERE ENAME='SCOTT'; MGR ----- 7566 1 개의 행이 선택되었다. ☞ 'SCOTT'의 관리자(MGR=Manager)는 7566 사번을 가진 JONES 이다.

[예제 및 실행 결과] SELECT MGR FROM EMP WHERE ENAME='KING'; MGR ----- 1 개의 행이 선택되었다. ☞ 빈 칸으로 표시되었지만 실 데이터는 NULL 이다. ☞ 'KING'은 EMP 테이블에서 사장이므로 MGR(관리자) 필드에 NULL 이 입력되어 있다.

[예제 및 실행 결과] SELECT NVL(MGR,9999) MGR FROM EMP WHERE ENAME='KING'; MGR ----- 9999 1 개의 행이 선택되었다. ☞ NVL 함수로 NULL 을 0 으로 변경한다.

- 공집합의 NVL/ISNULL 함수 사용

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

STEP1. 공집합을 발생시키기 위해 사원 테이블에 존재하지 않는 'JSC'라는 이름으로 데이터를 검색한다.

[예제 및 실행 결과] SELECT MGR FROM EMP WHERE ENAME='JSC'; 데이터를 찾을 수 없다. ☞ EMP 테이블에 ENAME 이 'JSC' 란 사람은 없으므로 공집합이 발생한다.

STEP2. NVL/ISNULL 함수를 이용해 공집합을 9999 로 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 ----- 1 개의 행이 선택되었다. ☞ 빈 칸으로 표시되었지만 실 데이터는 NULL 이다. ☞ 다른 함수와 달리 집계 함수와 Scalar Subquery 의 경우는 인수의 결과 값이 공집합인 경우에도 NULL 을 출력한다.

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



[예제 및 실행 결과] SELECT NVL(MAX(MGR), 9999) MGR FROM EMP WHERE ENAME='JSC'; MGR -----  
9999 1 개의 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 9999 라는 값으로 변환한 사례이다.

Oracle 의 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 NULL 과 공집합을 쉽게 구분하기 힘들다. 개발자들은 NVL/ISNULL 함수를 사용해야 하는 경우와, 집계 함수를 포함한 NVL/ISNULL 함수를 사용해야 하는 경우와, 1 장 7 절에서 설명할 NVL/ISNULL 함수를 포함한 집계 함수를 사용하지 않아야 될 경우까지 잘 이해해서 NVL/ISNULL 함수를 정확히 사용해야 한다.

## 다. NULLIF

NULLIF 함수는 EXPR1 이 EXPR2 와 같으면 NULL 을, 같지 않으면 EXPR1 을 리턴한다. 특정 값을 NULL 로 대체하는 경우에 유용하게 사용할 수 있다.

NULLIF (EXPR1, EXPR2)

[예제] 사원 테이블에서 MGR 와 7698 이 같으면 NULL 을 표시하고, 같지 않으면 MGR 를 표시한다.

[예제] SELECT ENAME, EMPNO, MGR, NULLIF(MGR,7698) NUIF FROM EMP;

[예제] NULLIF 함수를 CASE 문장으로 표현할 수 있다. SELECT ENAME, EMPNO, MGR, CASE WHEN MGR = 7698 THEN NULL ELSE MGR END NUIF FROM EMP;

[실행 결과] ENAME EMPNO MGR NUIF ----- SMITH 7369 7902 7902 ALLEN  
7499 7698 WARD 7521 7698 JONES 7566 7839 7839 MARTIN 7654 7698 BLAKE 7698 7839 7839  
CLARK 7782 7839 7839 SCOTT 7788 7566 7566 KING 7839 TURNER 7844 7698 ADAMS 7876 7788  
7788 JAMES 7900 7698 FORD 7902 7566 7566 MILLER 7934 7782 7782 14 개의 행이 선택되었다.

실행 결과를 보면 MGR 의 값이 7698 이란 상수가 같은 경우 NUIF 칼럼에 NULL 이 표시되었다. KING 이 속한 행의 NUIF 칼럼에 NULL 이 표시된 것은 원래 MGR 데이터가 NULL 이었기 때문이다.

## 라. 기타 NULL 관련 함수 (COALESCE)

COALESCE 함수는 인수의 숫자가 한정되어 있지 않으며, 임의의 개수 EXPR 에서 NULL 이 아닌 최초의 EXPR 을 나타낸다. 만일 모든 EXPR 이 NULL 이라면 NULL 을 리턴한다.

COALESCE (EXPR1, EXPR2, ...)

[예제] 사원 테이블에서 커미션을 1 차 선택값으로, 급여를 2 차 선택값으로 선택하되 두 칼럼 모두 NULL 인 경우는 NULL 로 표시한다.

[예제] SELECT ENAME, COMM, SAL, COALESCE(COMM, SAL) COAL FROM EMP;

[예제] COALESCE 함수는 두개의 중첩된 CASE 문장으로 표현할 수 있다. 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;

[실행 결과] ENAME COMM SAL COAL ----- SMITH 800 800 ALLEN 300  
1600 300 WARD 500 1250 500 JONES 2975 2975 MARTIN 1400 1250 1400 BLAKE 2850 2850 CLARK  
2450 2450 SCOTT 3000 3000 KING 5000 5000 TURNER 0 1500 0 ADAMS 1100 1100 JAMES 950 950  
FORD 3000 3000 MILLER 1300 1300 14 개의 행이 선택되었다.

## 07.GROUP BY, HAVING 절

### 1. 집계 함수(Aggregate Function)

여러 행들의 그룹이 모여서 그룹당 단 하나의 결과를 돌려주는 다중행 함수 중 집계 함수(Aggregate Function)의 특성은 다음과 같다.

- 여러 행들의 그룹이 모여서 그룹당 단 하나의 결과를 돌려주는 함수이다. - GROUP BY 절은 행들을 소그룹화 한다. - SELECT 절, HAVING 절, ORDER BY 절에 사용할 수 있다.

ANSI/ISO 에서 데이터 분석 기능으로 분류한 함수 중 기본적인 집계 함수는 본 절에서 설명하고, ROLLUP, CUBE, GROUPING SETS 같은 GROUP 함수는 2 장 5 절에서, 다양한 분석 기능을 가진 WINDOW 함수는 2 장 6 절에서 설명한다.

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

자주 사용되는 주요 집계 함수들은 다음과 같다. 집계 함수는 그룹에 대한 정보를 제공하므로 주로 숫자 유형에 사용되지만, MAX, MIN, COUNT 함수는 문자, 날짜 유형에도 적용이 가능한 함수이다.

[표 II-1-36] 집계 함수의 종류

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.

[예제] 일반적으로 집계 함수는 GROUP BY 절과 같이 사용되지만 아래와 같이 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 480 보다 작은 것을 볼 수 있다. 그 이유는 COUNT(\*) 함수에 사용된 와일드카드(\*)는 전체 칼럼을 뜻하는데 전체 칼럼이 NULL 인 행은 존재할 수 없기 때문에 결국 COUNT(\*)는 전체 행의 개수를 출력한 것이고, COUNT(HEIGHT)는 HEIGHT 칼럼 값이 NULL 인 33 건은 제외된 건수의 합이다.

## 2. GROUP BY 절

WHERE 절을 통해 조건에 맞는 데이터를 조회했지만 테이블에 1차적으로 존재하는 데이터 이외의 정보, 예를 들면 각 팀별로 선수가 몇 명인지, 선수들의 평균 신장과 몸무게가 얼마나 되는지, 또는 각 팀에서 가장 큰 키의 선수가 누구인지 등의 2차 가공 정보도 필요하다. GROUP BY 절은 SQL 문에서 FROM 절과 WHERE 절 뒤에 오며, 데이터들을 작은 그룹으로 분류하여 소그룹에 대한 항목별로 통계 정보를 얻을 때 추가로 사용된다.

```
SELECT [DISTINCT] 컬럼명 [ALIAS명] FROM 테이블명 [WHERE 조건식] [GROUP BY 컬럼(Column)이나 표현식] [HAVING 그룹조건식];
```

GROUP BY 절과 HAVING 절은 다음과 같은 특성을 가진다.

- GROUP BY 절을 통해 소그룹별 기준을 정한 후, SELECT 절에 집계 함수를 사용한다.
- 집계 함수의 통계 정보는 NULL 값을 가진 행을 제외하고 수행한다.
- GROUP BY 절에서는 SELECT 절과는 달리 ALIAS 명을 사용할 수 없다.
- 집계 함수는 WHERE 절에는 올 수 없다. (집계 함수를 사용할 수 있는 GROUP BY 절보다 WHERE 절이 먼저 수행된다)
- WHERE 절은 전체 데이터를 GROUP 으로 나누기 전에 행들을 미리 제거시킨다.
- HAVING 절은 GROUP BY 절의 기준 항목이나 소그룹의 집계 함수를 이용한 조건을 표시할 수 있다.
- GROUP BY 절에 의한 소그룹별로 만들어진 집계 데이터 중, HAVING 절에서 제한 조건을 두어 조건을 만족하는 내용만 출력한다.
- HAVING 절은 일반적으로 GROUP BY 절 뒤에 위치한다.

일부 데이터베이스의 과거 버전에서 데이터베이스가 GROUP BY 절에 명시된 컬럼의 순서대로 오름차순 정렬을 자동으로 실시(비공식적인 지원이었음)하는 경우가 있었으나, 원칙적으로 관계형 데이터베이스 환경에서는 뒤에서 언급할 ORDER BY 절을 명시해야 데이터 정렬이 수행된다. ANSI/ISO 기준에서도 데이터 정렬에 대한 내용은 ORDER BY 절에서만 언급되어있지, GROUP BY 절에는 언급되어 있지 않다.

[예제] K-리그 선수들의 포지션별 평균키는 어떻게 되는가란 요구 사항을 접수하였다. GROUP BY 절을 사용하지 않고 집계 함수를 사용했을 때 어떤 결과를 보이는지 포지션별 평균키를 구해본다.

[예제 및 실행 결과] SELECT POSITION 포지션, AVG(HEIGHT) 평균키 FROM PLAYER; SELECT POSITION 포지션, AVG(HEIGHT) 평균키 \* 1행에 오류: ERROR: 단일 그룹의 집계 함수가 아니다.

GROUP BY 절에서 그룹 단위를 표시해 주어야 SELECT 절에서 그룹 단위의 컬럼과 집계 함수를 사용할 수 있다. 그렇지 않으면 예제와 같이 에러를 발생하게 된다.

[예제] SELECT 절에서 사용된 포지션이라는 한글 ALIAS 를 GROUP BY 절의 기준으로 사용해본다.

[예제 및 실행 결과] SELECT POSITION 포지션, AVG(HEIGHT) 평균키 FROM PLAYER GROUP BY POSITION 포지션; GROUP BY POSITION 포지션 \* 3행에 오류: ERROR: SQL 명령어가 올바르게 종료되지 않았다.

실행 결과를 살펴보면 GROUP BY 절에 “포지션”이라고 표시된 부분에 에러가 발생했다는 것을 알 수 있다. 컬럼에 대한 ALIAS 는 SELECT 절에서 정의하고 ORDER BY 절에서는 재사용할 수 있지만, GROUP BY 절에서는 ALIAS 명을 사용할 수 없다는 것을 보여 주는 사례이다.

[예제] 포지션별 최대키, 최소키, 평균키를 출력한다. (포지션별이란 소그룹의 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;
```

[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.

실행 결과를 보면 포지션별로 평균키 외에도 인원수, 키대상 인원수, 최대키, 최소키가 제대로 출력된 것을 확인할 수 있다. ORDER BY 절이 없기 때문에 포지션 별로 정렬은 되지 않았다. 추가로 포지션과 키 정보가 없는 선수가 3 명이라는 정보를 얻을 수 있으며, 포지션이 DF 인 172 명 중 30 명은 키에 대한 정보가 없는 것도 알 수 있다. GK, DF, FW, MF 의 최대키, 최소키, 평균키를 구할 때 키 값이 NULL 인 경우는 계산 대상에서 제외된다. 즉, 포지션 DF 의 최대키, 최소키, 평균키 결과는 키 값이 NULL 인 30 명을 제외한 142 명을 대상으로 수행한 통계 결과이다.

### 3. HAVING 절

[예제] K-리그 선수들의 포지션별 평균키를 구하는데, 평균키가 180 센티미터 이상인 정보만 표시하라는 요구 사항이 접수되었으므로 WHERE 절과 GROUP BY 절을 사용해 SQL 문장을 작성한다.

[예제 및 실행 결과] SELECT POSITION 포지션, ROUND(AVG(HEIGHT),2) 평균키 FROM PLAYER WHERE AVG(HEIGHT) >= 180 GROUP BY POSITION; WHERE AVG(HEIGHT) >= 180 \* 3행에 오류: ERROR: 집계 함수는 허가되지 않는다.

실행 결과에서 WHERE 절의 집계 함수 AVG(HEIGHT) 부분에서 “집계 함수는 허가되지 않는다”는 에러 메시지가 출력되었다. 즉, WHERE 절에는 AVG()라는 집계 함수는 사용할 수 없다. WHERE 절은 FROM 절에 정의된 집합(주로 테이블)의 개별 행에 WHERE 절의 조건절이 먼저 적용되고, WHERE 절의 조건에 맞는 행이 GROUP BY 절의 대상이 된다. 그런 다음 결과 집합의 행에 HAVING 조건절이 적용된다. 결과적으로 HAVING 절의 조건을 만족하는 내용만 출력된다. 즉, HAVING 절은 WHERE 절과 비슷하지만 그룹을 나타내는 결과 집합의 행에 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;

[예제] 포지션 평균키 ----- GK 186.26 DF 180.21 2개의 행이 선택되었다.

실행 결과에서 전체 4 개 포지션 중에서 평균 키가 180cm 가 넘는 2 개의 데이터만 출력된 것을 확인할 수 있다.

[예제] SQL 문장은 GROUP BY 절과 HAVING 절의 순서를 바꾸어서 수행한다.

[예제] SELECT POSITION 포지션, AVG(HEIGHT) 평균키 FROM PLAYER HAVING AVG(HEIGHT) >= 180 GROUP BY POSITION;

[실행 결과] 포지션 평균키 ----- GK 186.26 DF 180.21 2개의 행이 선택되었다.

GROUP BY 절과 HAVING 절의 순서를 바꾸어서 수행하더라도 문법 에러도 없고 결과물도 동일한 결과를 출력한다. 그렇지만, SQL 내용을 보면, 포지션이란 소그룹으로 그룹핑(GROUPING)되어 통계 정보가 만들어지고, 이후 적용된 결과 값에 대한 HAVING 절의 제한 조건에 맞는 데이터만을 출력하는 것이므로 논리적으로 GROUP BY 절과 HAVING 절의 순서를 지키는 것을 권고한다.

[예제] K-리그의 선수들 중 삼성블루윙즈(K02)와 FC 서울(K09)의 인원수는 얼마인가란 요구 사항이 접수되었다. WHERE 절과 GROUP BY 절을 사용한 SQL 과 GROUP BY 절과 HAVING 절을 사용한 SQL 을 모두 작성한다.

[예제 및 실행 결과] SELECT TEAM\_ID 팀ID, COUNT(\*) 인원수 FROM PLAYER WHERE TEAM\_ID IN ('K09', 'K02') GROUP BY TEAM\_ID; 팀ID 인원수 ----- K02 49 K09 49 2개의 행이 선택되었다.

[예제 및 실행 결과] SELECT TEAM\_ID 팀ID, COUNT(\*) 인원수 FROM PLAYER GROUP BY TEAM\_ID



HAVING TEAM\_ID IN ('K09', 'K02'); 팀ID 인원수 ----- K02 49 K09 49 2개의 행이 선택되었다.

GROUP BY 소그룹의 데이터 중 일부만 필요한 경우, GROUP BY 연산 전 WHERE 절에서 조건을 적용하여 필요한 데이터만 추출하여 GROUP BY 연산을 하는 방법과, GROUP BY 연산 후 HAVING 절에서 필요한 데이터만 필터링 하는 두 가지 방법을 사용할 수 있다. 같은 실행 결과를 얻는 두 가지 방법 중 HAVING 절에서 TEAM\_ID 같은 GROUP BY 기준 칼럼에 대한 조건을 추가할 수도 있으나, 가능하면 WHERE 절에서 조건절을 적용하여 GROUP BY 의 계산 대상을 줄이는 것이 효율적인 자원 사용 측면에서 바람직하다.

[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;

[실행 결과] 포지션 평균키 ----- GK 186.26 DF 180.21 FW 179.91 3개의 행이 선택되었다.

SQL 을 보면 SELECT 절에서 사용하지 않는 MAX 집계 함수를 HAVING 절에서 조건절로 사용한 사례이다. 즉, HAVING 절은 SELECT 절에 사용되지 않은 칼럼이나 집계 함수가 아니더라도 GROUP BY 절의 기준 항목이나 소그룹의 집계 함수를 이용한 조건을 표시할 수 있다. 이 부분은 1 장 8 절의 SELECT 문장의 실행 순서에서 추가 설명한다. 여기서 주의할 것은 WHERE 절의 조건 변경은 대상 데이터의 개수가 변경되므로 결과 데이터 값이 변경될 수 있지만, HAVING 절의 조건 변경은 결과 데이터 변경은 없고 출력되는 레코드의 개수만 변경될 수 있다. 실행 결과를 보면 다른 결과 값의 변경 없이 MAX(HEIGHT)가 189cm 로 190cm 미만인 MF 포지션의 데이터만 HAVING 조건에 의해 누락된 것을 확인할 수 있다. (다른 포포지션의 통계 정보는 다음과 같다. (GROUP BY 절 예제 참조)

포지션 인원수 키대상 최대키 최소키 평균키 ----- MF 162 162 189 165 176.31

#### 4. CASE 표현을 활용한 월별 데이터 집계

“집계 함수(CASE( ))~GROUP BY” 기능은, 모델링의 제1 정규화로 인해 반복되는 칼럼의 경우 구분 칼럼을 두고 여러 개의 레코드로 만들어진 집합을, 정해진 칼럼 수만큼 확장해서 집계 보고서를 만드는 유용한 기법이다. 부서별로 월별 입사자의 평균 급여를 알고 싶다는 고객의 요구사항이 있는데, 입사 후 1년마다 급여 인상이나 보너스 지급과 같은 일정이 정기적으로 잡힌다면 업무적으로 중요한 정보가 될 수 있다.

STEP1. 개별 데이터 확인

[예제] 먼저 개별 입사정보에서 월별 데이터를 추출하는 작업을 진행한다. 이 단계는 월별 정보가 있다면 생략 가능하다.

[예제] Oracle SELECT ENAME, DEPTNO, EXTRACT(MONTH FROM HIREDATE) 입사월, SAL FROM EMP;

[예제] SQL Server SELECT ENAME, DEPTNO, DATEPART(MONTH, HIREDATE) 입사월, SAL FROM EMP;

[예제] SQL Server SELECT ENAME, DEPTNO, MONTH(HIREDATE) 입사월, SAL FROM EMP;

[실행 결과] ENAME DEPTNO 입사월 SAL ----- SMITH 20 12 800 ALLEN 30 2 1600 WARD 30 2 1250 JONES 20 4 2975 MARTIN 30 9 1250 BLAKE 30 5 2850 CLARK 10 6 2450 SCOTT 20 7 3000 KING 10 11 5000 TURNER 30 9 1500 ADAMS 20 7 1100 JAMES 30 12 950 FORD 20 12 3000 MILLER 10 1 1300 14개의 행이 선택되었다.

STEP2. 월별 데이터 구분

[예제] 추출된 MONTH 데이터를 Simple Case Expression 을 이용해서 12 개의 월별 칼럼으로 구분한다. 실행 결과에서 보여 주는 ENAME 칼럼은 최종 리포트에서 요구되는 데이터는 아니지만, 정보의 흐름을 이해하기 위해 부가적으로 보여 주는 임시 정보이다. FROM 절에서 사용된 인라인 뷰는 2 장 4 절에서 설명한다.

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

```
[실행 결과] ENAME DEPTNO M01 M02 M03 M04 M05 M06 M07 M08 M09 M10 M11 M12 -----
-- SMITH 20 800 ALLEN 30 2975
1600 WARD 30 1250 JONES 20 2975
MARTIN 30 1250 BLAKE 30 2850 CLARK 10
2450 SCOTT 20 3000 KING 10 5000
TURNER 30 1500 ADAMS 20 1100 JAMES 30
950 FORD 20 3000 MILLER 10 1300
14개의 행이 선택되었다.
```

### STEP3. 부서별 데이터 집계

[예제] 최종적으로 보여주는 리포트는 부서별로 월별 입사자의 평균 급여를 알고 싶다는 요구사항이므로 부서별 평균값을 구하기 위해 GROUP BY 절과 AVG 집계 함수를 사용한다. 직원 개인에 대한 정보는 더 이상 필요 없으므로 제외한다. ORDER BY 절을 사용하지 않았기 때문에 부서번호별로 정렬이 되지는 않았다.

```
[예제] 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 ;
```

```
[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 1300 2450 5000 3개의 행이 선택되었다.
```

하나의 데이터에 여러 번 CASE 표현을 사용하고 집계 함수가 적용되므로 SQL 처리 성능 측면에서 나쁜 것이 아니냐는 생각을 할 수도 있다. 그렇지만, 같은 기능을 하는 리포트를 작성하기 위해 장문의 프로그램을 코딩하는 것에 비해, 위 방법을 활용하면 복잡한 프로그램이 아닌 하나의 SQL 문장으로 처리 가능하므로 DBMS 자원 활용이나 처리 속도에서 훨씬 효율적이다. 데이터의 건수가 많아질수록 처리 속도의 차이는 더 크질 수 있다. 개발자들은 가능한 하나의 SQL 문장으로 비즈니스적인 요구 사항을 처리할 수 있도록 노력해야 한다.

[예제] Simple Case Expression 으로 표현된 위의 SQL 과 같은 내용으로 Oracle 의 DECODE 함수를 사용한 SQL 문장을 작성한다.

```
[예제] 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 ;
```

DECODE 함수를 사용함으로써 SQL 문장이 조금 더 짧아졌다. CASE 표현과 Oracle 의 DECODE 함수는 표현상 서로 장단점이 있으므로 어떤 기능을 선택할 지는 사용자의 몫이다.

## 5. 집계 함수와 NULL

리포트의 빈칸을 NULL 이 아닌 ZERO 로 표현하기 위해 NVL(Oracle)/ISNULL(SQL Server) 함수를 사용하는 경우가 많은데, 다중 행 함수를 사용하는 경우는 오히려 불필요한 부하가 발생하므로 굳이 NVL 함수를 다중 행 함수 안에 사용할 필요가 없다. 다중 행 함수는 입력 값으로 전체 건수가 NULL 값인 경우만 함수의 결과가 NULL 이 나오고 전체 건수 중에서 일부만 NULL 인 경우는 NULL 인 행을 다중 행 함수의 대상에서 제외한다. 예를 들면 100 명 중 10 명의 성적이 NULL 값일 때 평균을 구하는 다중 행 함수 AVG 를 사용하면 NULL 값이 아닌 90 명의 성적에 대해서 평균값을 구하게 된다.

CASE 표현 사용시 ELSE 절을 생략하게 되면 Default 값이 NULL 이다. NULL 은 연산의 대상이 아닌 반면, SUM(CASE MONTH WHEN 1 THEN SAL ELSE 0 END)처럼 ELSE 절에서 0(Zero)을 지정하면 불필요하게 0 이 SUM 연산에 사용되므로 자원의 사용이 많아진다. 같은 결과를 얻을 수 있다면 가능한 ELSE 절의 상수값을 지정하지 않거나 ELSE 절을 작성하지 않도록 한다. 같은 이유로 Oracle 의 DECODE 함수는 4 번째 인자를 지정하지 않으면 NULL 이 Default 로 할당된다. 많이 실수하는 것 중에 하나가 Oracle 의 SUM(NVL(SAL,0)), SQL Server 의 SUM(ISNULL (SAL,0)) 연산이다. 개별 데이터의 급여(SAL)가 NULL 인 경우는 NULL 의 특성으로 자동적으로 SUM 연산에서 빠지는 데, 불필요하게 NVL/ISNULL 함수를 사용해 0(Zero)으로 변환시켜 데이터 건수만큼의 연산이 일어나게 하는 것은 시스템의 자원을 낭비하는 일이다. 리포트 출력 때 NULL 이 아닌 0 을 표시하고 싶은 경우에는 NVL(SUM(SAL),0)이나, ISNULL(SUM(SAL),0)처럼 전체 SUM 의 결과가 NULL 인 경우(대상 건수가 모두 NULL 인 경우)에만 한 번 NVL/ISNULL 함수를 사용하면 된다.

[예제] 팀별 포지션별 FW, MF, DF, GK 포지션의 인원수와 팀별 전체 인원수를 구하는 SQL 문장을 작성한다. 데이터가 없는 경우는 0 으로 표시한다.

[예제] SIMPLE\_CASE\_EXPRESSION 조건 - Oracle SELECT TEAM\_ID, NVL(SUM(CASE POSITION WHEN 'FW' THEN 1 ELSE 0 END),0) FW, NVL(SUM(CASE POSITION WHEN 'MF' THEN 1 ELSE 0 END),0) MF, NVL(SUM(CASE POSITION WHEN 'DF' THEN 1 ELSE 0 END),0) DF, NVL(SUM(CASE POSITION WHEN 'GK' THEN 1 ELSE 0 END),0) GK, COUNT(\*) SUM FROM PLAYER GROUP BY TEAM\_ID;

[예제] SIMPLE\_CASE\_EXPRESSION 조건 - Oracle CASE 표현의 ELSE 0, ELSE NULL 문구는 생략 가능하므로 다음과 같이 조금 더 짧게 SQL 문장을 작성할 수 있다. Default 값인 NULL이 적용됨. 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, COUNT(\*) SUM FROM PLAYER GROUP BY TEAM\_ID;

[예제] SEARCHED\_CASE\_EXPRESSION 조건 - Oracle SELECT TEAM\_ID, NVL(SUM(CASE WHEN POSITION = 'FW' THEN 1 END), 0) FW, NVL(SUM(CASE WHEN POSITION = 'MF' THEN 1 END), 0) MF, NVL(SUM(CASE WHEN POSITION = 'DF' THEN 1 END), 0) DF, NVL(SUM(CASE WHEN POSITION = 'GK' THEN 1 END), 0) GK, COUNT(\*) SUM FROM PLAYER GROUP BY TEAM\_ID;

[예제] SEARCHED\_CASE\_EXPRESSION 조건 - SQL Server SELECT TEAM\_ID, ISNULL(SUM(CASE WHEN POSITION = 'FW' THEN 1 END), 0) FW, ISNULL(SUM(CASE WHEN POSITION = 'MF' THEN 1 END), 0) MF, ISNULL(SUM(CASE WHEN POSITION = 'DF' THEN 1 END), 0) DF, ISNULL(SUM(CASE WHEN POSITION = 'GK' THEN 1 END), 0) GK, COUNT(\*) SUM FROM PLAYER GROUP BY TEAM\_ID;

[실행 결과] TEAM\_ID FW MF DF GK SUM ----- --- --- --- --- --- K14 0 1 1 0 2 K06 11 11 20 4 46

K13 1 0 1 1 3 K15 1 1 1 0 3 K02 10 18 17 4 49 K12 1 0 1 0 2 K04 13 11 18 4 46 K03 6 15 23 5 49 K07 9  
22 16 4 51 K05 10 19 17 5 51 K08 8 15 15 4 45 K11 1 1 1 0 3 K01 12 15 13 5 45 K10 5 15 13 3 36 K09 12  
18 15 4 49 15개의 행이 선택되었다.

4 개의 예제 SQL 문장은 같은 실행 결과를 출력한다. ORDER BY 절이 적용되지 않았으므로 TEAM\_ID 별로 정렬이 되어 있지 않다. TEAM\_ID 'K08'의 경우 POSITION 이 NULL 인 3 건이 포지션별 분류에는 빠져 있지만 COUNT(\*) SUM 에는 추가되어 있다.

[예제] GROUP BY 절 없이 전체 선수들의 포지션별 평균 키 및 전체 평균 키를 출력할 수도 있다.

[예제] SELECT ROUND(AVG(CASE WHEN POSITION = 'MF' THEN HEIGHT END),2) 미드필더,  
ROUND(AVG(CASE WHEN POSITION = 'FW' THEN HEIGHT END),2) 포워드, ROUND(AVG(CASE WHEN  
POSITION = 'DF' THEN HEIGHT END),2) 디펜더, ROUND(AVG(CASE WHEN POSITION = 'GK' THEN HEIGHT  
END),2) 골키퍼, ROUND(AVG(HEIGHT),2) 전체평균키 FROM PLAYER;

[실행 결과] 미드필더 포워드 디펜더 골키퍼 전체평균키 ----- 176.31 179.91  
180.21 186.26 179.31 1개의 행이 선택되었다.



## 08.ORDER BY 절

### 1. 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. 그리고 별도로 정렬 방식을 지정하지 않으면 기본적으로 오름차순이 적용되며, SQL 문장의 제일 마지막에 위치한다.

SELECT 칼럼명 [ALIAS 명] FROM 테이블명 [WHERE 조건식] [GROUP BY 칼럼(Column)이나 표현식] [HAVING 그룹조건식] [ORDER BY 칼럼(Column)이나 표현식 [ASC 또는 DESC]] ; ASC(Ascending) : 조회한 데이터를 오름차순으로 정렬한다.(기본 값이므로 생략 가능) DESC(Descending) : 조회한 데이터를 내림차순으로 정렬한다.

[예제] ORDER BY 절의 예로 선수 테이블에서 선수들의 이름, 포지션, 백넘버를 출력하는데 사람 이름을 내림차순으로 정렬하여 출력한다.

[예제] SELECT PLAYER\_NAME 선수명, POSITION 포지션, BACK\_NO 백넘버 FROM PLAYER ORDER BY PLAYER\_NAME DESC;

[실행 결과] 선수명 포지션 백넘버 ----- 히카르도 MF 10 황철민 MF 35 황연석 FW 16 황승주 DF 98 홍종하 MF 32 홍인기 DF 35 홍성요 DF 28 홍복표 FW 19 홍명보 DF 20 홍도표 MF 9 홍광철 DF 4 호제리오 DF 3 480 개의 행이 선택되었다.

[예제] ORDER BY 절의 예로 선수 테이블에서 선수들의 이름, 포지션, 백넘버를 출력하는데 선수들의 포지션 내림차순으로 출력한다. 칼럼명이 아닌 ALIAS 를 이용한다.

[예제] SELECT PLAYER\_NAME 선수명, POSITION 포지션, BACK\_NO 백넘버 FROM PLAYER ORDER BY 포지션 DESC;

[실행 결과] Oracle 선수명 포지션 백넘버 키 ----- 정학범 173 차상광 186 안익수 174 백영철 MF 22 173 조태용 MF 7 192 올리베 MF 29 190 김리네 MF 26 188 차스민 MF 33 186 480 개의 행이 선택되었다.

실행 결과에서 포지션에 아무 것도 없는 값들이 있다. 현재 선수 테이블에서 포지션 칼럼에 NULL 이 들어 있는데 포지션의 내림차순에서 NULL 값이 앞에 출력되었다는 것은 Oracle 이 NULL 값을 가장 큰 값으로 취급했다는 것을 알 수 있다. 반면 SQL Server 는 반대의 정렬 순서를 가진다. ORDER BY 절 사용 특징은 아래와 같다.

- 기본적인 정렬 순서는 오름차순(ASC)이다. - 숫자형 데이터 타입은 오름차순으로 정렬했을 경우에 가장 작은 값부터 출력된다. - 날짜형 데이터 타입은 오름차순으로 정렬했을 경우 날짜 값이 가장 빠른 값이 먼저 출력된다. 예를 들어 '01-JAN-2012' 는 '01-SEP-2012' 보다 먼저 출력된다. - Oracle 에서는 NULL 값을 가장 큰 값으로 간주하여 오름차순으로 정렬했을 경우에는 가장 마지막에, 내림차순으로 정렬했을 경우에는 가장 먼저 위치한다. - 반면, SQL Server 에서는 NULL 값을 가장 작은 값으로 간주하여 오름차순으로 정렬했을 경우에는 가장 먼저, 내림차순으로 정렬했을 경우에는 가장 마지막에 위치한다.

[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 NULL 인 데이터는 제외한다.

[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;

[실행 결과] 선수명 포지션 백넘버 키 ----- 서동명 GK 21 196 권정혁 GK 1 195 김석  
FW 20 194 정경두 GK 41 194 이현 GK 1 192 황연석 FW 16 192 미트로 FW 19 192 김대희 GK 31 192 조의  
손 GK 44 192 김창민 GK 1 191 우성용 FW 22 191 최동석 GK 1 190 샤샤 FW 10 190 447 개의 행이 선택  
되었다.

실행 결과를 보면 키가 192cm 인 선수가 5 명 있는데, ORDER BY 절에서 키가 큰 순서대로 출력하  
고, 키가 같으면 백넘버 순으로 정렬하라는 조건에 따라서 백넘버 순으로 정렬되어 있는 것을 확인  
할 수 있다. 칼럼명이나 ALIAS 명을 대신해서 SELECT 절의 칼럼 순서를 정수로 매핑하여 사용할  
수도 있다. 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, ALIAS 명, 칼럼 순서를 같이 혼용하는 것도 가능하다.

[예제] ORDER BY 절의 예로 선수 테이블에서 선수들의 이름, 포지션, 백넘버를 출력하는데 선수들  
의 백넘버 내림차순, 백넘버가 같은 경우 포지션, 포지션까지 같은 경우 선수명 순서로 출력한다.  
BACK\_NO 가 NULL 인 경우는 제외하고, 칼럼명이나 ALIAS 가 아닌 칼럼 순서를 매핑하여 사용하  
다.

[예제] SELECT PLAYER\_NAME 선수명, POSITION 포지션, BACK\_NO 백넘버 FROM PLAYER WHERE  
BACK\_NO IS NOT NULL ORDER BY 3 DESC, 2, 1;

[실행 결과] 선수명 포지션 백넘버 ----- 뚜따 FW 99 쿠키 FW 99 황승주 DF 98 무스타파  
MF 77 다보 FW 63 다오 DF 61 김충호 GK 60 최동우 GK 60 최주호 GK 51 안동원 DF 49 오재진 DF  
49 ... .. 439 개의 행이 선택되었다.

[예제] DEPT 테이블 정보를 부서명, 지역, 부서번호 내림차순으로 정렬해서 출력한다. 아래의 SQL  
문장은 출력되는 칼럼 레이블은 다를 수 있지만 결과는 모두 같다.

Case1. 칼럼명 사용 ORDER BY 절 사용

[예제] SELECT DNAME, LOC, DEPTNO FROM DEPT ORDER BY DNAME, LOC, DEPTNO DESC;

[실행 결과] DNAME LOC DEPTNO ----- ACCOUNTING NEW YORK 10  
OPERATIONS BOSTON 40 RESEARCH DALLAS 20 SALES CHICAGO 30 4 개의 행이 선택되었다.

Case2. 칼럼명 + ALIAS 명 사용 ORDER BY 절 사용

[예제] SELECT DNAME DEPT, LOC AREA, DEPTNO FROM DEPT ORDER BY DNAME, AREA, DEPTNO  
DESC;

[실행 결과] DEPT AREA DEPTNO ----- ACCOUNTING NEW YORK 10  
OPERATIONS BOSTON 40 RESEARCH DALLAS 20 SALES CHICAGO 30 4 개의 행이 선택되었다.

Case3. 칼럼 순서번호 + ALIAS 명 사용 ORDER BY 절 사용

[예제] SELECT DNAME, LOC AREA, DEPTNO FROM DEPT ORDER BY 1, AREA, 3 DESC;

[실행 결과] DNAME AREA DEPTNO ----- ACCOUNTING NEW YORK 10  
OPERATIONS BOSTON 40 RESEARCH DALLAS 20 SALES CHICAGO 30 4 개의 행이 선택되었다.

## 2. SELECT 문장 실행 순서

GROUP BY 절과 ORDER BY 가 같이 사용될 때 SELECT 문장은 6 개의 절로 구성이 되고, SELECT  
문장의 수행 단계는 아래와 같다.

5. SELECT 칼럼명 (ALIAS 명) 1. FROM 테이블명 2. WHERE 조건식 3. GROUP BY 칼럼(Column)이나 표현  
식 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)

위 순서는 옵티마이저가 SQL 문장의 SYNTAX, SEMANTIC 에러를 점검하는 순서이기도 하다. 예를 들면 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 DISTINCT 를 지정하거나 SQL 문장에 GROUP BY 절이 있거나 또는 SELECT 문에 UNION 연산자가 있으면 열 정의가 SELECT 목록에 표시되어야 한다. 이 부분은 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.

[예제] SELECT 절에 없는 EMP 칼럼을 ORDER BY 절에 사용한다.

[예제] SELECT EMPNO, ENAME FROM EMP ORDER BY MGR;

[실행 결과] EMPNO ENAME ----- 7902 FORD 7788 SCOTT 7900 JAMES 7499 ALLEN 7521 WARD 7844 TURNER 7654 MARTIN 7934 MILLER 7876 ADAMS 7698 BLAKE 7566 JONES 7782 CLARK 7369 SMITH 7839 KING 14 개의 행이 선택되었다.

실행 결과에서 ORDER BY 절에서 SELECT 절에서 정의하지 않은 칼럼을 사용해도 문제없음을 확인할 수 있다.

[예제] 인라인 뷰에 정의된 SELECT 칼럼을 메인쿼리에서 사용한다.

[예제 및 실행 결과] SELECT EMPNO FROM (SELECT EMPNO, ENAME FROM EMP ORDER BY MGR); 14 개의 행이 선택되었다.

실행 결과에서 2 장에서 배울 인라인 뷰의 SELECT 절에서 정의한 칼럼은 메인쿼리에서도 사용할 수 있는 것을 확인할 수 있다.

[예제] 인라인 뷰에 미정의된 칼럼을 메인쿼리에서 사용해본다.

[예제 및 실행 결과] SELECT MGR FROM (SELECT EMPNO, ENAME FROM EMP ORDER BY MGR);  
SELECT MGR FROM ; \* ERROR: "MGR": 부적합한 식별자

그러나 서브쿼리의 SELECT 절에서 선택되지 않은 칼럼들은 계속 유지되는 것이 아니라 서브쿼리 범위를 벗어나면 더 이상 사용할 수 없게 된다. (인라인 뷰도 동일함) GROUP BY 절에서 그룹핑 기준을 정의하게 되면 데이터베이스는 일반적인 SELECT 문장처럼 FROM 절에 정의된 테이블의 구조를 그대로 가지고 가는 것이 아니라, GROUP BY 절의 그룹핑 기준에 사용된 칼럼과 집계 함수에 사용될 수 있는 숫자형 데이터 칼럼들의 집합을 새로 만든다. GROUP BY 절을 사용하게 되면 그룹핑 기준에 사용된 칼럼과 집계 함수에 사용될 수 있는 숫자형 데이터 칼럼들의 집합을 새로 만드는데, 개별 데이터는 필요 없으므로 저장하지 않는다. GROUP BY 이후 수행 절인 SELECT 절이나 ORDER BY 절에서 개별 데이터를 사용하는 경우 에러가 발생한다. 결과적으로 SELECT 절에서는 그룹핑 기준과 숫자 형식 칼럼의 집계 함수를 사용할 수 있지만, 그룹핑 기준 외의 문자 형식 칼럼은 정할 수 없다.

[예제] GROUP BY 절 사용시 SELECT 절에 일반 칼럼을 사용해본다.

[예제 및 실행 결과] SELECT JOB, SAL FROM EMP GROUP BY JOB HAVING COUNT(\*) > 0 ORDER BY SAL; SELECT JOB, SAL ; \* ERROR: GROUP BY 표현식이 아니다.

[예제] GROUP BY 절 사용시 ORDER BY 절에 일반 칼럼을 사용해본다.

[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.

[예제] GROUP BY 절 사용시 ORDER BY 절에 집계 칼럼을 사용해본다.

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

[실행 결과] JOB ----- MANAGER PRESIDENT SALESMAN ANALYST CLERK 5 개의 행이 선택되었다.

SELECT SQL 에서 GROUP BY 절이 사용되었기 때문에 SELECT 절에 정의하지 않은 MAX, SUM,  
COUNT 집계 함수도 ORDER BY 절에서 사용할 수 있는 것을 실행 결과에서 확인할 수 있다.

### 3. Top N 쿼리

- ROWNUM

Oracle 에서 순위가 높은 N 개의 로우를 추출하기 위해 ORDER BY 절과 WHERE 절의 ROWNUM  
조건을 같이 사용하는 경우가 있는데 이 두 조건으로는 원하는 결과를 얻을 수 없다. Oracle 의 경  
우 정렬이 완료된 후 데이터의 일부가 출력되는 것이 아니라, 데이터의 일부가 먼저 추출된 후  
(ORDER BY 절은 결과 집합을 결정하는데 관여하지 않음) 데이터에 대한 정렬 작업이 일어나므로  
주의해야 한다.

[예제] 사원 테이블에서 급여가 높은 3 명만 내림차순으로 출력하고자 하는데, 잘못 사용된 SQL 의  
사례이다.

[예제] SELECT ENAME, SAL FROM EMP WHERE ROWNUM < 4 ORDER BY SAL DESC;

[실행 결과] ENAME SAL ----- ---- ALLEN 1600 WARD 1250 SMITH 800 3 개의 행이 선택되었다.

실행 결과의 3 명은 급여가 상위인 3 명을 출력한 것이 아니라, 급여 순서에 상관없이 무작위로 추출  
된 3 명에 한해서 급여를 내림차순으로 정렬한 결과이므로 원하는 결과를 출력한 것이 아니다.

[예제] ORDER BY 절이 없으면 ORACLE 의 ROWNUM 조건과 SQL SERVER 의 TOP 절은 같은 결  
과를 보인다. 그렇지만, ORDER BY 절이 사용되는 경우 ORACLE 은 ROWNUM 조건을 ORDER BY  
절보다 먼저 처리되는 WHERE 절에서 처리하므로, 정렬 후 원하는 데이터를 얻기 위해서는 2 장 4  
절에서 배울 인라인 뷰에서 먼저 데이터 정렬을 수행한 후 메인쿼리에서 ROWNUM 조건을 사용해야  
한다.

[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 3 개의 행이 선택되었다.

위 사례에서는 인라인 뷰를 사용하여 추출하고자 하는 집합을 정렬한 후 ROWNUM 을 적용시킴으  
로써 결과에 참여하는 순서와 추출되는 로우 순서를 일치시킴으로써 Top N 쿼리의 결과를 만들어  
내었다. 실행 결과를 보면 EMP 테이블의 데이터를 급여가 많은 순서부터 정렬을 수행한 후 상위 3  
건의 데이터를 출력한 것을 알 수 있다. 추가로, 원하는 추출 결과와 동일한 순서로 정렬된 인덱스  
가 존재한다면 그 인덱스를 사용하여 동일한 결과를 얻을 수도 있다.

- TOP ( )

반면 SQL Server 는 TOP 조건을 사용하게 되면 별도 처리 없이 관련 Order By 절의 데이터 정렬  
후 원하는 일부 데이터만 쉽게 출력할 수 있다.

TOP (Expression) [PERCENT] [WITH TIES]



TOP 절을 사용하여 결과 집합으로 반환되는 행 수를 제한할 수 있다. WITH TIES 옵션은 ORDER BY 절의 조건 기준으로 TOP N의 마지막 행으로 표시되는 추가 행의 데이터가 같을 경우 N+ 동일 정렬 순서 데이터를 추가 반환하도록 지정하는 옵션이다.

[예제] 사원 테이블에서 급여가 높은 2 명을 내림차순으로 출력하고자 한다.

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

[실행 결과] ENAME SAL ----- KING 5000 SCOTT 3000 2 개의 행이 선택되었다.

[예제] 사원 테이블에서 급여가 높은 2 명을 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 3 개의 행이 선택되었다.

TOP(2) WITH TIES 옵션은 동일 수치의 데이터를 추가로 더 추출하는 것으로, SCOTT 과 FORD의 급여가 공동 2 위이므로 TOP(2) WITH TIES의 실행 결과는 3건의 데이터가 출력된다.

## 09.조인(JOIN)

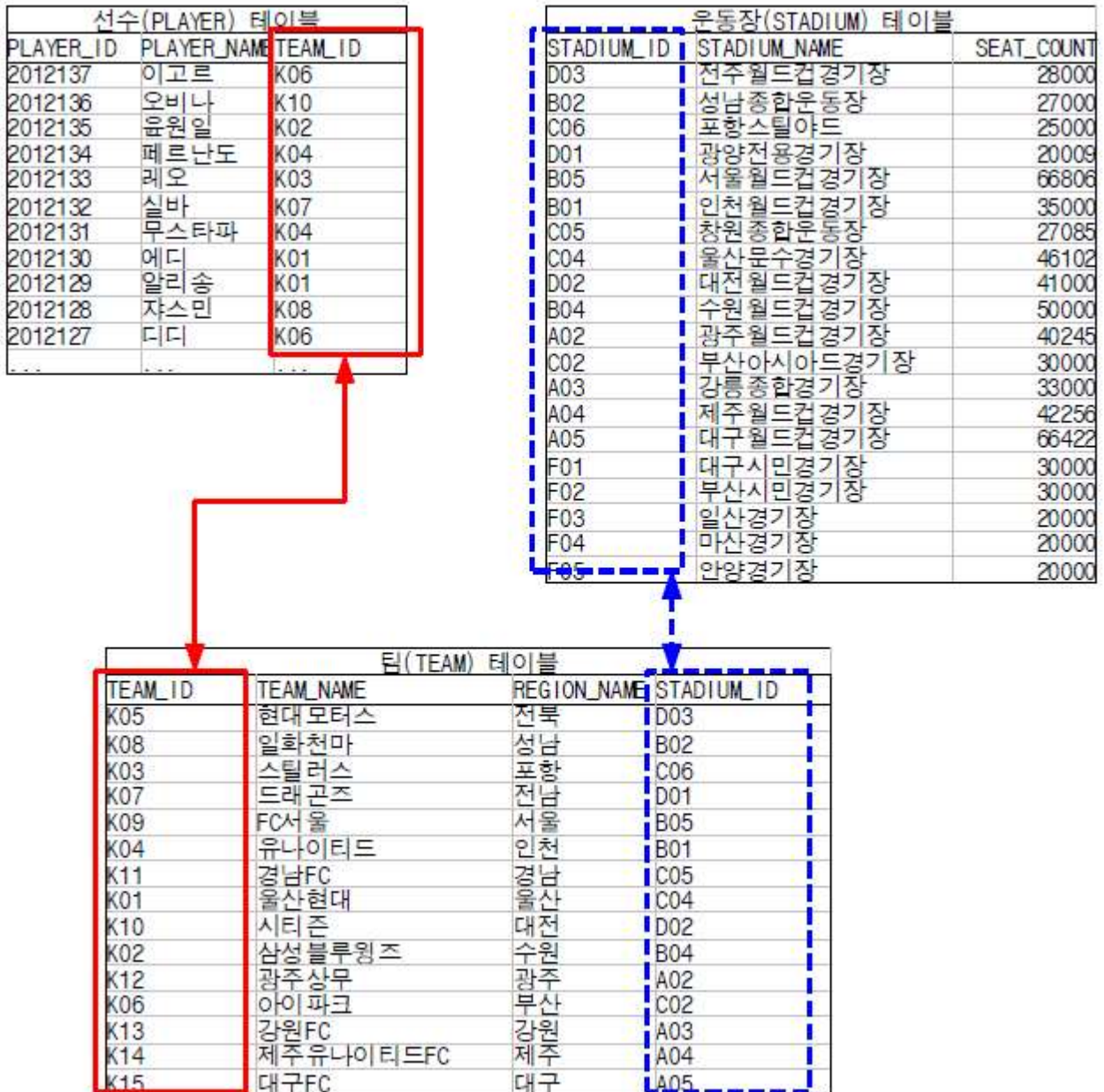
### 1. JOIN 개요

지금까지는 하나의 테이블에서 데이터를 출력하는 것을 살펴보았다. 하지만, 이것은 일상생활에서 발생하는 다양한 조건을 만족하는 SQL 문장을 작성하기에는 부족하다. 예를 들어, [그림 11-1-12]와 같이 선수들의 소속팀에 대한 정보나 프로 축구팀의 전용구장에 대한 정보 등 다른 정보가 들어있는 두 개 이상의 테이블과 연결 또는 결합하여 데이터를 출력하는 경우가 아주 많이 발생한다.

두 개 이상의 테이블 들을 연결 또는 결합하여 데이터를 출력하는 것을 JOIN 이라고 하며, 일반적으로 사용되는 SQL 문장의 상당수가 JOIN 이라고 생각하면 JOIN 의 중요성을 이해하기 쉬울 것이다. JOIN 은 관계형 데이터베이스의 가장 큰 장점이면서 대표적인 핵심 기능이라고 할 수 있다. 일반적인 경우 행들은 PRIMARY KEY(PK)나 FOREIGN KEY(FK) 값의 연관에 의해 JOIN 이 성립된다. 하지만 어떤 경우에는 이러한 PK, FK 의 관계가 없어도 논리적인 값들의 연관만으로 JOIN 이 성립 가능하다. 선수라는 테이블과 팀이라는 테이블이 있는 경우, 선수 테이블을 기준으로 필요한 데이터를 검색하고 이 데이터와 연관된 팀 테이블의 특정 행을 찾아오는 과정이 JOIN 을 이용하여 데이터를 검색하는 과정으로 볼 수 있는 것이다. 팀과 운동장 테이블도 조인 조건을 통해 필요한 데이터를 조합해서 가져올 수 있으며, 하나의 SQL 문장에서 선수, 팀, 운동장 등 여러 테이블을 조인해서 사용할 수도 있다.

다만 한 가지 주의할 점은 FROM 절에 여러 테이블이 나열되더라도 SQL 에서 데이터를 처리할 때는 단 두 개의 집합 간에만 조인이 일어난다는 것이다. FROM 절에 A, B, C 테이블이 나열되었더라도 특정 2 개의 테이블만 먼저 조인 처리되고, 2 개의 테이블이 조인되어서 처리된 새로운 데이터 집합과 남은 한 개의 테이블이 다음 차례로 조인되는 것이다. 이순서는 4 개 이상의 테이블이 사용되더라도 같은 프로세스를 반복한다.

예를 들어 A, B, C, D 4 개의 테이블을 조인하고자 할 경우 옵티마이저는 ( ( A JOIN D ) JOIN C ) JOIN B)와 같이 순차적으로 조인을 처리하게 된다. 먼저 A 와 D 테이블을 조인 처리하고, 그 결과 집합과 C 테이블을 다음 순서에 조인 처리하고, 마지막으로 3 개의 테이블을 조인 처리한 집합과 B 테이블을 조인 수행하게 된다. 이때 테이블의 조인 순서는 옵티마이저에 의해서 결정되고 과목 3 의 주요 튜닝 포인트가 된다. [그림 11-1-12]는 선수와 팀, 팀과 운동장 테이블 간의 관계를 설명한 것이다. 경기일정결과 테이블은 복잡성을 피하기 위해 설명상 제외하였다.



[그림 II-1-12] 테이블 간의 관계도

## 2. EQUI JOIN

EQUI(등가) JOIN 은 두 개의 테이블 간에 칼럼 값들이 서로 정확하게 일치하는 경우에 사용되는 방법으로 대부분 PK ↔ FK 의 관계를 기반으로 한다. 그러나 일반적으로 테이블 설계 시에 나타난 PK ↔ FK 의 관계를 이용하는 것이지 반드시 PK ↔ FK 의 관계로만 EQUI JOIN 이 성립하는 것은 아니다. 이 기능은 계층형(Hierarchical)이나 망형(Network) 데이터베이스와 비교해서 관계형 데이터베이스의 큰 장점이다. JOIN 의 조건은 WHERE 절에 기술하게 되는데 “=” 연산자를 사용해서 표현한다. 다음은 EQUI JOIN 의 대략적인 형태이다.

```
SELECT 테이블1.칼럼명, 테이블2.칼럼명, ... FROM 테이블1, 테이블2 WHERE 테이블1.칼럼명1 = 테이블2.칼럼명2; → WHERE 절에 JOIN 조건을 넣는다.
```

같은 내용을 ANSI/ISO SQL 표준 방식으로 표현하면 아래와 같다. ON 절에 대해서는 2 장 1 절에서 자세히 다룬다.

```
SELECT 테이블1.칼럼명, 테이블2.칼럼명, ... FROM 테이블1 INNER JOIN 테이블2 ON 테이블1.칼럼명1 = 테이블2.칼럼명2;
```

블2.칼럼명2: → ON 절에 JOIN 조건을 넣는다.

[예제] 선수 테이블과 팀 테이블에서 선수 이름과 소속된 팀의 이름을 출력하시오.

[예제] SELECT PLAYER,PLAYER\_NAME 선수명, TEAM,TEAM\_NAME 소속팀명 FROM PLAYER, TEAM WHERE PLAYER,TEAM\_ID = TEAM,TEAM\_ID; 또는 INNER JOIN을 명시하여 사용할 수도 있다. SELECT PLAYER,PLAYER\_NAME 선수명, TEAM,TEAM\_NAME 소속팀명 FROM PLAYER INNER JOIN TEAM ON PLAYER,TEAM\_ID = TEAM,TEAM\_ID;

위 SQL 을 보면 SELECT 구문에 단순히 칼럼명이 오지 않고 “테이블명.칼럼명”처럼 테이블명과 칼럼명이 같이 나타난다. 이렇게 특정 칼럼에 접근하기 위해 그 칼럼이 어느 테이블에 존재하는 칼럼 인지를 명시하는 것은 두 가지 이유가 있다.

먼저 모든 테이블에 칼럼들이 유일한 이름을 가진다면 상관없지만, JOIN 에 사용되는 두 개의 테이블에 같은 칼럼명이 존재하는 경우에는 DBMS 의 옵티마이저는 어떤 칼럼을 사용해야 할지 모르기 때문에 파싱 단계에서 에러가 발생된다.

두 번째는 개발자나 사용자가 조회할 데이터가 어느 테이블에 있는 칼럼을 말하는 것인지 쉽게 알 수 있게 하므로 SQL 에 대한 가독성이나 유지보수성을 높이는 효과가 있다. 하나의 SQL 문장 내에서 유일하게 사용하는 칼럼명이라면 칼럼명 앞에 테이블 명을 붙이지 않아도 되지만, 현재 두 집합에서 유일하다고 하여 미래에도 두 집합에서 유일하다는 보장은 없기 때문에 향후 발생할 오류를 방지하고 일관성을 위해 유일한 칼럼도 출력할 칼럼명 앞에 테이블명을 붙여서 사용하는 습관을 기르는 것을 권장한다.

조인 조건에 맞는 데이터만 출력하는 INNER JOIN 에 참여하는 대상 테이블이 N 개라고 했을 때, N 개의 테이블로부터 필요한 데이터를 조회하기 위해 필요한 JOIN 조건은 대상 테이블의 개수에서 하나를 뺀 N-1 개 이상이 필요하다. 즉 FROM 절에 테이블이 3 개가 표시되어 있다면 JOIN 조건은 3-1=2 개 이상이 필요하며, 테이블이 4 개가 표시되어 있다면 JOIN 조건은 4-1=3 개 이상이 필요하다. (옵티마이저의 발전으로 옵티마이저가 일부 JOIN 조건을 실행계획 수립 단계에서 추가할 수도 있지만, 예외적인 사항이다.)

JOIN 조건은 WHERE 절에 기술하며, JOIN 은 두 개 이상의 테이블에서 필요한 데이터를 출력하기 위한 가장 기본적인 조건이다. FROM 절에 조인 조건을 명시하는 또 다른 방법은 2 장 1 절에서 설명한다. (JOIN 조건이 없는 CROSS JOIN 도 2 장 1 절에서 설명한다.) 위 예제는 테이블 1 과 테이블 2 이름을 가진 두개 테이블에  $2 - 1 = 1$  인 한 개의 JOIN 조건(PLAYER,TEAM\_ID = TEAM,TEAM\_ID) 을 WHERE 절에 기술한 것이다.

## 가. 선수-팀 EQUI JOIN 사례

[그림 II-1-13]과 같이 선수(PLAYER) 테이블과 팀(Team) 테이블에서 K-리그 소속 선수들의 이름, 백넘버와 그 선수가 소속되어 있는 팀명 및 연고지를 알고 싶다는 요구사항을 확인한다.



선수(PLOYER) 테이블		
PLAYER-NAME	BACK_NO	TEAM_ID
이 고 르	21	K06
오 비 나	26	K10
윤 원 일	45	K02
페 르 난 도	44	K04
레 오	45	K03
실 바	45	K07
무 스 타 파	77	K04
에 디	7	K01
알리송	14	K014
자스민	33	K08
디디	8	K06
⋮	⋮	⋮



팀(TEAM) 테이블		
TEAM_ID	TEAM_NAME	REGION_NAME
K05	현대모터스	전부
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	수우 js
K12	광주상무	광주
K06	아이파크	부산
K13	강원FC	강원
K14	제주유나이티드FC	제주
K15	대구FC	대구

[그림 II-1-13] EQUI JOIN을 설명하기 위한 선수-팀 테이블 관계도

이 질의를 해결하기 위해 테이블 간의 관계를 이해할 필요가 있다. 우선 선수(PLOYER) 테이블과 팀(TEAM) 테이블에 있는 데이터와 이들 간의 관계를 나타내는 그림을 통해서 실제로 데이터들이 어떻게 연결되는지 살펴본다. 위와 같이 선수들의 정보가 들어 있는 선수(PLOYER) 테이블이 있고, 팀의 정보가 들어 있는 팀(TEAM) 테이블이 있다. 그런데 선수(PLOYER) 테이블에 있는 소속팀코드(TEAM\_ID) 칼럼이 팀(TEAM) 테이블의 팀코드(TEAM\_ID)와 PK(팀 테이블의 팀코드)와 FK(선수 테이블의 소속팀 코드)의 관계에 있다. 선수들과 선수들이 소속해 있는 팀명 및 연고지를 알아보기 위해서 선수 테이블의 소속팀코드를 기준으로 팀 테이블에 들어 있는 데이터를 다음과 같이 순서를 바꾸어 주면 아래 [그림 II-1-14]와 같이 바꿀 수 있다.

선수(PLOYER) 테이블			팀(TEAM) 테이블		
PLAYER-NAME	BACK_NO	TEAM_ID	TEAM_ID	TEAM_NAME	REGION_NAME
이 고 르	21	K06	K06	아이파크	부산
오 비 나	26	K10	K10	시티즌	대전
윤 원 일	45	K02	K02	삼성블루윙즈	수원
페 르 난 도	44	K04	K04	유나이티드	인천
레 오	45	K03	K03	스틸러스	포항
실 바	45	K07	K07	드래곤즈	전남
무 스 타 파	77	K04	K04	유나이티드	인천
에 디	7	K01	K01	울산현대	울산
알리송	14	K014	K01	울산현대	울산
자스민	33	K08	K08	일화천마	성남
디디	8	K06	K06	아이파크	부산
⋮	⋮	⋮	⋮	⋮	⋮

[그림 II-1-14] EQUI JOIN을 설명하기 위한 데이터 재배열 후

[그림 II-1-14]의 실바 선수를 예로 들면 백넘버는 45 번이고, 소속팀코드는 K07 번이다. K07 번 팀 코드의 팀명은 드래곤즈이고 연고지는 전남이라는 결과를 얻을 수 있게 된다.

[예제] [그림 II-1-14]의 데이터를 출력하기 위한 SELECT SQL 문장을 작성한다.

[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; 또는 INNER JOIN을 명시하여 사용할 수도 있다. SELECT PLAYER.PLAYER\_NAME, PLAYER.BACK\_NO, PLAYER.TEAM\_ID, TEAM.TEAM\_NAME, TEAM.REGION\_NAME FROM PLAYER INNER JOIN TEAM ON PLAYER.TEAM\_ID = TEAM.TEAM\_ID;

[실행 결과] PLAYER\_NAME BACK\_NO TEAM\_ID TEAM\_NAME REGION\_NAME -----  
----- 이고르 21 K06 아이파크 부산 오비나 26 K10 시티즌 대전 윤원일 45 K02 삼성블루윙즈 수원 페르난도 44 K04 유나이티드 인천 레오 45 K03 스틸러스 포항 실바 45 K07 드래곤즈 전남 무스타파 77 K04 유나이티드 인천 에디 7 K01 울산현대 울산 알리송 14 K01 울산현대 울산 자스민 33 K08 일화천마 성남 디디 8 K06 아이파크 부산 480개 항이 선택되었다.

위 예제를 확인하면 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. 그래서 SELECT 절에서 칼럼에 대한 ALIAS를 사용하는 것처럼 FROM 절의 테이블에 대해서도 ALIAS를 사용할 수 있다. 조회할 칼럼명 앞에 테이블명을 명시적으로 기술하는 것이 이론적으로는 가장 좋은 방법일 수 있지만, 테이블명이 길고 SQL의 복잡도가 높아지면 오히려 가독성이 떨어지기 때문에 테이블명 대신 ALIAS를 주로 사용한다. 단일 테이블을 사용하는 SQL 문장에서는 필요성은 없지만 사용하더라도 예러는 발생하지 않으며, 여러 테이블을 사용하는 조인을 이용하는 경우는 매우 유용하게 사용할 수 있다.

[예제] 칼럼과 테이블에 ALIAS를 적용하여 위 SQL을 수정한다. 실행 결과는 ALIAS 적용 전과 같음을 확인 할 수 있다.

[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; 또는 INNER JOIN을 명시하여 사용할 수도 있다. SELECT P.PLAYER\_NAME 선수명, P.BACK\_NO 백넘버, P.TEAM\_ID 팀코드, T.TEAM\_NAME 팀명, T.REGION\_NAME 연고지 FROM PLAYER P INNER JOIN TEAM T ON P.TEAM\_ID = T.TEAM\_ID;

[실행 결과] 선수명 백넘버 팀코드 팀명 연고지 ----- 이고르 21 K06 아이파크 부산 오비나 26 K10 시티즌 대전 윤원일 45 K02 삼성블루윙즈 수원 페르난도 44 K04 유나이티드 인천 레오 45 K03 스틸러스 포항 실바 45 K07 드래곤즈 전남 무스타파 77 K04 유나이티드 인천 에디 7 K01 울산현대 울산 알리송 14 K01 울산현대 울산 자스민 33 K08 일화천마 성남 디디 8 K06 아이파크 부산 ... 480 개 항이 선택되었다.

## 나. 선수-팀 WHERE 절 검색 조건 사례

지금까지는 EQUI JOIN에 대한 JOIN 조건만을 다루었는데, 추가로 WHERE 절에서 JOIN 조건 이외의 검색 조건에 대한 제한 조건을 덧붙여 사용할 수 있다. 즉, EQUI JOIN의 최소한의 연관 관계를 위해서 테이블 개수 - 1개의 JOIN 조건을 WHERE 절에 명시하고, 부수적인 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 SQL 문을 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; 또는 INNER JOIN을 명시하여 사용할 수도 있다. SELECT P.PLAYER\_NAME 선수명, P.BACK\_NO 백넘버, T.REGION\_NAME 연고지, T.TEAM\_NAME 팀명 FROM PLAYER P INNER JOIN TEAM T ON P.TEAM\_ID = T.TEAM\_ID WHERE P.POSITION = 'GK' ORDER BY P.BACK\_NO;

[실행 결과] 선수명 백넘버 연고지 팀명 ----- 최종문 1 전남 드래곤즈 정병지 1 포항 스틸러스 박유석 1 부산 아이파크 김승준 1 대전 시티즌 이현 1 인천 유나이티드 김운재 1 수원 삼성블루윙즈 정해운 1 성남 일화천마 권정혁 1 울산 울산현대 최동석 1 서울 FC서울 김창민 1 전북 현대모터스 김용

발 18 전북 현대모터스 한동진 21 인천 유나이티드 이은성 21 대전 시티즌 김준호 21 포항 스틸러스 조범철  
 21 수원 삼성블루윙즈 백민철 21 서울 FC서울 권찬수 21 성남 일화천마 서동명 21 울산 울산현대 강성일  
 30 대전 시티즌 김대희 31 포항 스틸러스 남현우 31 인천 유나이티드 정지혁 31 부산 아이파크 양영민 31  
 성남 일화천마 염동균 31 전남 드래곤즈 이무림 31 울산 울산현대 최관민 31 전북 현대모터스 최호진 31 수  
 원 삼성블루윙즈 우태식 31 서울 FC서울 김정래 33 전남 드래곤즈 최창주 40 울산 울산현대 정용대 40  
 부산 아이파크 정경진 41 부산 아이파크 정광수 41 수원 삼성블루윙즈 정경두 41 성남 일화천마 허인무 41  
 포항 스틸러스 정영광 41 전남 드래곤즈 조의손 44 서울 FC서울 정이섭 45 전북 현대모터스 양지원 45 울  
 산 울산현대 선원길 46 강원 강원FC 최주호 51 포항 스틸러스 최동우 60 전북 현대모터스 김중호 60 인천  
 유나이티드 43개 항이 선택되었다.

JOIN 조건을 기술할 때 주의해야 할 사항이 한 가지 있다. 만약 테이블에 대한 ALIAS 를 적용해서 SQL 문장을 작성했을 경우, WHERE 절과 SELECT 절에는 테이블명이 아닌 테이블에 대한 ALIAS 를 사용해야 한다는 점이다. 그러나, 권장 사항은 아니지만 하나의 SQL 문장 내에서 유일하게 사  
 용하는 칼럼명이라면 칼럼명 앞에 ALIAS 를 붙이지 않아도 된다.

[예제] 위 SQL 문장에서 FROM 절에서 테이블에 대한 ALIAS 를 정의했는데, SELECT 절이나 WHERE 절에서 테이블명을 사용한다면 DBMS 의 옵티마이저가 칼럼명이 부적합하다는 에러를 파싱  
 단계에서 발생시킨다. (SQL 문장의 파싱 순서는 FROM 절, WHERE 절, SELECT 절, ORDER BY 절  
 순서이다.)

[예제 및 실행 결과] SELECT PLAYER.PLAYER\_NAME 선수명, P.BACK\_NO 백넘버, T.REGION\_NAME 연고  
 지, 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; SELECT PLAYER.PLAYER\_NAME 선수명, P.BACK\_NO 백넘버, \* 1행에 오  
 류: ERROR: 열명이 부적합하다.

### 다. 팀-구장 EQUI JOIN 사례

팀(Team) 테이블				운동장(Stadium) 테이블		
TEAM ID	TEAM NAME	REGION NAMES	STADIUM ID	STADIUM ID	STADIUM NAME	SEAT COUNT
K05	현대모터스	전북	D03	D03	전주월드컵경기장	28000
K08	일화천마	성남	B02	B02	성남종합운동장	27000
K03	스틸러스	포항	C06	C06	포항스틸야드	25000
K07	드래곤즈	전남	D01	D01	광양전용경기장	20000
K09	FC서울	서울	B05	B05	서울월드컵경기장	66806
K04	유나이티드	인천	B01	B01	인천월드컵경기장	35000
K11	경남FC	경남	C05	C05	창원종합운동장	27085
K01	울산현대	울산	C04	C04	울산문수경기장	46102
K10	시티즌	대전	D02	D02	대전월드컵경기장	41000
K02	삼성블루윙즈	수원	B04	B04	수원월드컵경기장	50000
K12	광주상무	광주	A02	A02	광주월드컵경기장	40245
K06	아이파크	부산	C02	C02	부산아시안경기장	30000
K13	강원FC	강원	A03	A03	강릉종합경기장	33000
K14	제주유나이티드FC	제주	A04	A04	제주월드컵경기장	42256
K15	대구FC	대구	A05	A05	대구월드컵경기장	66422
				F01	대구시민경기장	30000
				F02	부산시민경기장	30000
				F03	일산경기장	20000
				F04	마산경기장	20000
				F05	안양경기장	20000

[그림 II-1-15] EQUI JOIN을 설명하기 위한 팀-구장 테이블 관계도

[예제] 이번에는 [그림 II-1-15]에 나와 있는 팀(Team) 테이블과 구장(Stadium) 테이블의 관계를  
 이용해서 소속팀이 가지고 있는 전용구장의 정보를 팀의 정보와 함께 출력하는 SQL 문을 작성한다.



[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; 또는 INNER JOIN을 명시하여 사용할 수도 있다. SELECT TEAM.REGION\_NAME, TEAM.TEAM\_NAME, TEAM.STADIUM\_ID, STADIUM.STADIUM\_NAME, STADIUM.SEAT\_COUNT FROM TEAM INNER JOIN STADIUM ON TEAM.STADIUM\_ID = STADIUM.STADIUM\_ID; 위 SQL문과 ALIAS를 사용한 아래 SQL문은 같은 결과를 얻을 수 있다. 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; 또는 INNER JOIN을 명시하여 사용할 수도 있다. SELECT T.REGION\_NAME, T.TEAM\_NAME, T.STADIUM\_ID, S.STADIUM\_NAME, S.SEAT\_COUNT FROM TEAM T INNER JOIN STADIUM S ON T.STADIUM\_ID = S.STADIUM\_ID; 중복이 되지 않는 칼럼의 경우 ALIAS를 사용하지 않아도 되므로, 아래 SQL 문은 위 SQL문과 같은 결과를 얻을 수 있다. 그러나 같은 이름을 가진 중복 칼럼의 경우는 테이블명이나 ALIAS가 필수 조건이다. SELECT REGION\_NAME, TEAM\_NAME, T.STADIUM\_ID, STADIUM\_NAME, 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 전남 드래곤즈 D01 광양 전용경기장 20009 서울 FC서울 B05 서울월드컵경기장 66806 인천 유나이티드 B01 인천월드컵경기장 35000 경남 경남FC C05 창원종합운동장 27085 울산 울산현대 C04 울산문수경기장 46102 대전 시티즌 D02 대전월드컵경기장 41000 수원 삼성블루윙즈 B04 수원월드컵경기장 50000 광주 광주상무 A02 광주 월드컵경기장 40245 부산 아이파크 C02 부산아시아드경기장 30000 강원 강원FC A03 강릉종합경기장 33000 제주 제주유나이티드FC A04 제주월드컵경기장 42256 대구 대구FC A05 대구월드컵경기장 66422  
 15개의 행이 선택되었다.

### 3. Non EQUI JOIN

Non EQUI(비등가) JOIN 은 두 개의 테이블 간에 칼럼 값들이 서로 정확하게 일치하지 않는 경우에 사용된다. Non EQUI JOIN 의 경우에는 “=” 연산자가 아닌 다른(Between, >, >=, <, <= 등) 연산자들을 사용하여 JOIN 을 수행하는 것이다. 두 개의 테이블이 PK-FK 로 연관관계를 가지거나 논리적으로 같은 값이 존재하는 경우에는 “=” 연산자를 이용하여 EQUI JOIN 을 사용한다. 그러나 두 개의 테이블 간에 칼럼 값들이 서로 정확하게 일치하지 않는 경우에는 EQUI JOIN 을 사용할 수 없다. 이런 경우 Non EQUI JOIN 을 시도할 수 있으나 데이터 모델에 따라서 Non EQUI JOIN 이 불가능한 경우도 있다. 다음은 Non EQUI JOIN 의 대략적인 형태이다. 아래 BETWEEN a AND b 조건은 Non EQUI JOIN 의 한 사례일 뿐이다.

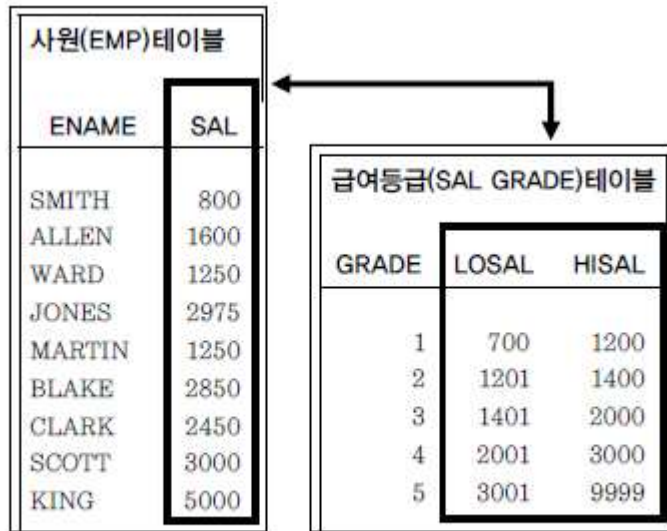
SELECT 테이블1.칼럼명, 테이블2.칼럼명, ... FROM 테이블1, 테이블2 WHERE 테이블1.칼럼명1 BETWEEN 테이블2.칼럼명1 AND 테이블2.칼럼명2;

[예제] Non EQUI JOIN 에 대한 샘플은 K-리그 관련 테이블로 구현되지 않으므로, 사원(EMP) 테이블과 가상의 급여등급(SAL\_GRADE) 테이블로 설명을 하도록 한다. 어떤 사원이 받고 있는 급여가 어느 등급에 속하는 등급인지 알고 싶다는 요구사항에 대한 Non EQUI JOIN 의 사례는 다음과 같다.

[예제] SELECT E.ENAME, E.JOB, E.SAL, S.GRADE FROM EMP E, SALGRADE S WHERE E.SAL BETWEEN S.LOSAL AND S.HISAL;

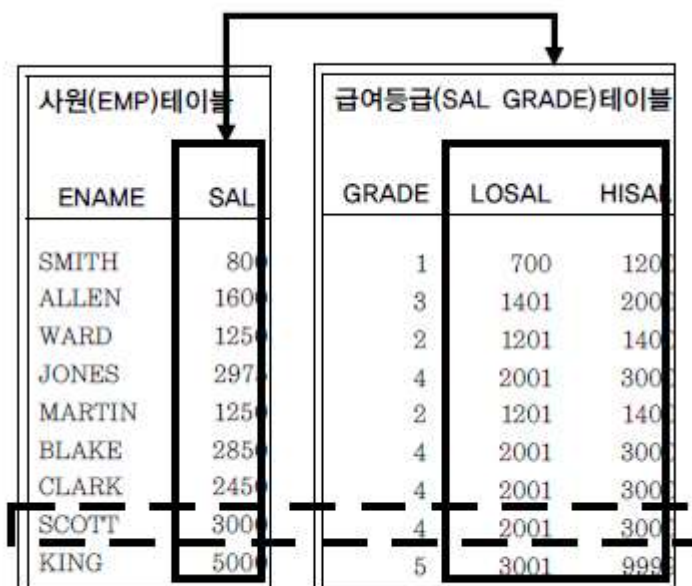
테이블 간의 관계를 설명하기 위해 먼저 사원(EMP) 테이블과 급여등급(SALGRADE) 테이블에 있는 데이터와 이들 간의 관계를 나타내는 [그림 II-1-16]을 가지고 실제적인 데이터들이 어떻게 연결되는지 설명한다. 급여등급(SALGRADE) 테이블에는 1 급(700 이상 ~ 1200 이하), 2 급(1201 이상 ~ 1400 이하), 3 급(1401 이상 ~ 2000 이하), 4 급(2001 이상 ~ 3000 이하), 5 급(3001 이상 ~ 9999 이하)으로 구분한 5 개의 급여등급이 존재한다고 가정한다.





[그림 II-1-16] Non EQUI JOIN을 설명하기 위한 두 개의 테이블 관계도

사원(EMP) 테이블에서 사원들의 급여가 급여등급(SALGRADE) 테이블의 등급으로 표시되기 위해서는 “=” 연산자로 JOIN 을 이용할 수가 없다. 그래서 사원들과 사원들의 급여가 급여등급 테이블의 어느 급여등급에 해당되는지 알아보기 위해서 사원 테이블에 들어 있는 데이터를 기준으로 급여등급 테이블의 어느 등급에 속하는지 1:1로 해당하는 값들을 나열해 보면 아래 [그림 II-1-17]과 같이 바꿀 수 있다.



[그림 II-1-17] Non EQUI JOIN을 설명하기 위한 데이터 재배열 후

[그림 II-1-17]을 보면 SCOTT 라는 사원을 예로 들어 급여는 3,000 달러(\$)이고, 3,000 달러(\$)는 급여등급 테이블에서 2,001 ~ 3,000 달러(\$) 사이의 4 급에 해당하는 급여등급이라는 값을 얻을 수 있다.

[예제] 사원 14 명 모두에 대해 아래 SQL 로 급여와 급여등급을 알아본다.

[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 ----- SMITH 800 1 JAMES 950 1 ADAMS 1100 1 WARD 1250 2 MARTIN 1250 2 MILLER 1300 2 TURNER 1500 3 ALLEN 1600 3 CLARK 2450 4 BLAKE 2850 4 JONES 2975 4 SCOTT 3000 4 FORD 3000 4 KING 5000 5 14개의 행이 선택되었다.

앞에서도 언급했지만 BETWEEN a AND b와 같은 SQL 연산자 뿐만 아니라 “=” 연산자가 아닌 “>”나 “<”와 같은 다른 연산자를 사용했을 경우에도 모두 Non EQUI JOIN에 해당한다. 단지 BETWEEN SQL 연산자가 Non EQUI JOIN을 설명하기 쉽기 때문에 예를 들어 설명한 것에 불과하며, 데이터 모델에 따라서 Non EQUI JOIN이 불가능한 경우도 있다. /p>

## 4. 3개 이상 TABLE JOIN

JOIN을 처음 설명할 때 나왔던 [그림 11-1-12]를 보면서 세 개의 테이블에 대한 JOIN을 구현해 보도록 한다. [그림 11-1-12]에서는 선수 테이블, 팀 테이블, 운동장 테이블을 예로 들었다. 선수들 별로 홈그라운드 경기장이 어디인지를 출력하고 싶다고 했을 때, 선수 테이블과 운동장 테이블이 서로 관계가 없으므로 중간에 팀 테이블이라는 서로 연관관계가 있는 테이블을 추가해서 세 개의 테이블을 JOIN해야만 원하는 데이터를 얻을 수 있다.

[예제] 앞의 예제에서 보았듯이 선수 테이블의 소속팀코드(TEAM\_ID)가 팀 테이블의 팀코드(TEAM\_ID)와 PK-FK의 관계가 있다는 것을 알 수 있고, 운동장 테이블의 운동장코드(STADIUM\_ID)와 팀 테이블의 전용구장코드(STADIUM\_ID)가 PK-FK 관계인 것을 생각하며 다음 SQL을 작성한다. 세 개의 테이블에 대한 JOIN이므로 WHERE 절에 2개 이상의 JOIN 조건이 필요하다.

[예제] [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; 또는 INNER JOIN을 명시하여 사용할 수도 있다. SELECT P.PLAYER\_NAME 선수명, P.POSITION 포지션, T.REGION\_NAME 연고지, T.TEAM\_NAME 팀명, S.STADIUM\_NAME 구장명 FROM PLAYER P INNER JOIN TEAM T ON P.TEAM\_ID = T.TEAM\_ID INNER JOIN STADIUM S ON 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 강성일 GK 대전 시티즌 대전월드컵경기장 강용 DF 포항 스틸러스 포항스틸야드 강정훈 MF 대전 시티즌 대전월드컵경기장 강철 DF 전남 드래곤즈 광양전용경기장 고관영 MF 전북 현대모터스 전주월드컵경기장 고규익 DF 광주 광주상무 광주월드컵경기장 고민기 FW 전북 현대모터스 전주월드컵경기장 고병운 DF 포항 스틸러스 포항스틸야드 고종수 MF 수원 삼성블루윙즈 수원월드컵경기장 고창현 MF 수원 삼성블루윙즈 수원월드컵경기장 공오균 MF 대전 시티즌 대전월드컵경기장 광경근 FW 인천 유나이티드 인천월드컵경기장 광기훈 FW 울산 울산현대 울산문수경기장 광기훈 FW 울산 울산현대 울산문수경기장 광치국 MF 성남 일화천마 성남종합운동장 ... 480개의 행이 선택되었다.

지금까지 JOIN에 대한 기본적인 사용법을 확인해 보았는데, JOIN이 필요한 기본적인 이유는 과목 1에서 배운 정규화에서부터 출발한다. 정규화란 불필요한 데이터의 정합성을 확보하고 이상현상(Anomaly) 발생을 피하기 위해, 테이블을 분할하여 생성하는 것이다. 사실 데이터웨어하우스 모델처럼 하나의 테이블에 모든 데이터를 집중시켜놓고 그 테이블로부터 필요한 데이터를 조회할 수도 있다. 그러나 이렇게 했을 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. 그렇지만 특정 요구조건을 만족하는 데이터들을 분할된 테이블로부터 조회하기 위해서는 테이블 간에 논리적인 연관관계가 필요하고 그런 관계성을 통해서 다양한 데이터들을 출력할 수 있는 것이다. 그리고, 이런 논리적인 관계를 구체적으로 표현하는 것이 바로 SQL 문장의 JOIN 조건인 것이다. 관계형 데이터베이스의 큰 장점이면서, SQL 튜닝의 중요 대상이 되는 JOIN을 잘못 기술하게 되면 시스템 자원 부족이나 과다한 응답시간 지연을 발생시키는 주요 원인이 되므로 JOIN 조건은 신중하게 작성해야 한다.