자격증 스터디 교재

SQL 개발자(SQLD) 가이드

2020. 8.

데이터 전문가 포럼

(Data Professional Forum)

※ 한국데이터산업진흥원 http://www.dbguide.net 게시자료 활용

- 목 차 -

과목 1 데이터 모델링의 이해

제1장 데이터 모델링의 이해 ····· 2

제1절 데이터 모델의 이해 ····· 2

제2절 엔터티(Entity) ····· 25

제3절 속성(Attribute) ····· 34

제4절 관계(Relationship) ····· 41

제5절 식별자 ····· 49

제2장 데이터 모델과 성능 ····· 65

제1절 성능 데이터 모델링의 개요 ····· 65

제2절 정규화와 성능 ····· 68

제3절 반정규화와 성능 ····· 79

제4절 대량 데이터에 따른 성능 ····· 89

제5절 데이터베이스 구조와 성능 ····· 97

제6절 분산 데이터베이스와 성능 ····· 111

과목 2 SQL 기본 및 활용

제1장 SQL 기본 ····· 124

제1절 관계형 데이터베이스 개요 ····· 124

제2절 DDL(DATA DEFINITION LANGUAGE) ····· 134

제3절 DML(DATA MANIPULATION LANGUAGE) ····· 158

제4절 TCL(TRANSACTION CONTROL LANGUAGE) ····· 169

제5절 WHERE 절 ····· 183

제6절 함수(FUNCTION) ····· 204

제7절 GROUP BY, HAVING 절 ····· 237

제8절 ORDER BY 절 ····· 251

제9절 조인(JOIN) ····· 262

제2장 SQL 활용 ····· **277**
제1절 표준 조인(STANDARD JOIN) ····· 277
제2절 집합 연산자(SET OPERATOR) ····· 307
제3절 계층형 질의와 셀프 조인 ····· 321
제4절 서브쿼리 ····· 335
제5절 그룹 함수(GROUP 함수) ····· 348
제6절 윈도우 함수(WINDOW 함수) ····· 363
제7절 DCL(DATA CONTROL LANGUAGE) ····· 384
제8절 절차형 SQL ····· 398
제3장 SQL 최적화 기본 원리 ····· 422
제1절 옵티마이저와 실행계획 ····· 422
제2절 인덱스 기본 ····· 430
제3절 조인 수행 원리 ····· 437

[별첨] 제30~35회 기출문제 ····· **442**

과목 1 데이터 모델링의 이해

제1절 데이터 모델의 이해

1. 모델링의 이해

가. 모델링의 정의

인류의 가장 보편적인 특징이면서 욕구 중의 하나는 의사소통을 하면서 항상 그에 대한 기록을 남기는 것이다. 어떤 현상에 대해 기록하고 남겨 자신 스스로 또는 다른 사람에게 적절한 의미를 주기 위해 고대부터 기록의 문화는 발전해 왔다고 할 수 있다. 모델이라고 하는 것은 모형(模型), 축소형(縮小型)의 의미로서 사람이 살아가면서 나타날 수 있는 다양한 현상에 대해서 일정한 표기법에 의해 표현해 놓은 모형이라고 할 수 있다. 이 역시 사람이 어떤 목적을 달성하기 위해 커뮤니케이션의 효율성을 극대화한 고급화된 표현방법으로 설명될 수 있다.

사람이 살아가면서 나타날 수 있는 다양한 현상은 사람, 사물, 개념 등에 의해 발생된다고 할 수 있으며 모델링은 이것을 표기법에 의해 규칙을 가지고 표기하는 것 자체를 의미한다. 즉 모델을 만들어가는 일 자체를 모델링으로 정의할 수 있다.

복잡한 현실세계를 일정한 표기법에 의해 표현하는 일

[그림 Ⅰ-1-1] 모델링의 정의

다음은 모델링에 대한 다양한 정의를 보여준다.

1) Webster 사전

- 가설적 또는 일정 양식에 맞춘 표현(a hypothetical or stylized representation)
- 어떤 것에 대한 예비표현으로 그로부터 최종대상이 구축되도록 하는 계획으로서 기여하는 것

2) 복잡한 '현실세계'를 단순화시켜 표현하는 것이다. 3) 모델이란 사물 또는 사건에 관한 양상(Aspect)이나 관점(Perspective)을 연관된 사람이나 그룹을 위하여 명확하게 하는 것이다. 4) 모델이란 현실 세계의 추상화된 반영이다.

나. 모델링의 특징

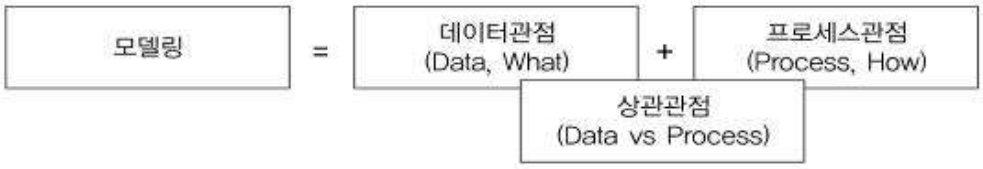
위의 정의를 요약하여 모델링의 특징을 요약하면 추상화, 단순화, 명확화의 3대 특징으로 요약할 수 있다.

1) 추상화(모형화, 가설적)는 현실세계를 일정한 형식에 맞추어 표현을 한다는 의미로 정리할 수 있다. 즉, 다양한 현상을 일정한 양식인 표기법에 의해 표현한다는 것이다.
2) 단순화는 복잡한 현실세계를 약속된 규약에 의해 제한된 표기법이나 언어로 표현하여 쉽게 이해할 수 있도록 하는 개념을 의미한다. 3) 명확화는 누구나 이해하기 쉽게 하기 위해 대상에 대한 애매모호함을 제거하고 정확(正確)하게 현상을 기술하는 것을 의미한다.

따라서 모델링을 다시 정의하면 '현실세계를 추상화, 단순화, 명확화하기 위해 일정한 표기법에 의해 표현하는 기법'으로 정리할 수 있다. 정보시스템 구축에서는 모델링을 계획/분석/설계 할 때 업무를 분석하고 설계하는데 이용하고 이후 구축/운영 단계에서는 변경과 관리의 목적으로 이용하게 된다.

다. 모델링의 세 가지 관점

시스템의 대상이 되는 업무를 분석하여 정보시스템으로 구성하는 과정에서 업무의 내용과 정보시스템의 모습을 적절한 표기법(Notation)으로 표현하는 것을 모델링이라고 한다면, 모델링은 크게 세 가지 관점인 데이터관점, 프로세스관점, 데이터와 프로세스의 상관관점으로 구분하여 설명할 수 있다.



[그림 Ⅰ-1-2] 모델링의 관점

1) 데이터관점 : 업무가 어떤 데이터와 관련이 있는지 또는 데이터간의 관계는 무엇인지에 대해서 모델링하는 방법(What, Data) 2) 프로세스관점 : 업무가 실제하고 있는 일은 무엇인지 또는 무엇을 해야 하는지를 모델링하는 방법(How, Process) 3) 데이터와 프로세스의 상관관점 : 업무가 처리하는 일의 방법에 따라 데이터는 어떻게 영향을 받고 있는지 모델링하는 방법(Interaction)으로 설명될 수 있다.

이 장에서는 데이터 모델링에 대한 기본 개념이 중요하므로 프로세스와 상관모델링에 대한 내용은 생략하고 데이터베이스를 구축하기 위한 데이터 모델링을 중심으로 설명한다.

2. 데이터 모델의 기본 개념의 이해

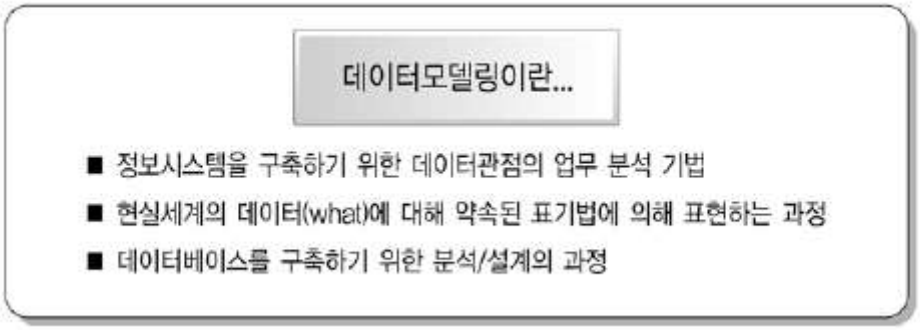
가. 모델링의 정의

데이터 모델은 데이터베이스의 골격을 이해하고 그 이해를 바탕으로 SQL문장을 기능과 성능적인 측면에서 효율적으로 작성하기 위해 꼭 알아야 하는 핵심요소이다. SQL 전문가를 위한 지식에서도 데이터베이스의 논리적인 구조를 이해하는 데이터 모델을 이해하는 것은 그 다음 SQL문장을 어떻게 구성할지에 대한 지식과 효율적인 구성에 대한 밑바탕의 지식을 쌓기 위한 핵심 이론이라 할 수 있다. 일반적으로 데이터 모델링은 다음과 같이 다양하게 정의될 수 있다.

- 정보시스템을 구축하기 위해, 해당 업무에 어떤 데이터가 존재하는지 또는 업무가 필요로 하는 정보는 무엇인지를 분석하는 방법
- 기업 업무에 대한 종합적인 이해를 바탕으로 데이터에 존재하는 업무 규칙(Business Rule)에 대하여 참(True) 또는 거짓(False)을 판별할 수 있는 사실(사실명제)을 데이터에 접근하는 방법(How), 사람(Who), 전산화와는 별개의(독립적인) 관점에서 이를 명확하게 표현하는 추상화 기법

이것을 좀 더 실무적으로 해석해 보면 업무에서 필요로 하는 데이터를 시스템 구축 방법론에 의해 분석하고 설계하여 정보시스템을 구축하는 과정으로 정의할 수 있다.

데이터 모델링을 하는 주요한 이유는 업무정보를 구성하는 기초가 되는 정보들을 일정한 표기법에 의해 표현함으로써 정보시스템 구축의 대상이 되는 업무 내용을 정확하게 분석하는 것이 첫 번째 목적이다. 두 번째는 분석된 모델을 가지고 실제 데이터베이스를 생성하여 개발 및 데이터관리에 사용하기 위한 것이다. 즉, 데이터 모델링이라는 것은 단지 데이터베이스만을 구축하기 위한 용도로만 쓰이는 것이 아니라 데이터 모델링 자체로서 업무를 설명하고 분석하는 부분에도 매우 중요한 의미를 가지고 있다고 할 수 있다.



나. 데이터 모델이 제공하는 기능

업무를 분석하는 관점에서 데이터 모델이 제공하는 기능은 다음과 같다.

- 시스템을 현재 또는 원하는 모습으로 가시화하도록 도와준다.
- 시스템의 구조와 행동을 명세화 할 수 있게 한다.
- 시스템을 구축하는 구조화된 틀을 제공한다.
- 시스템을 구축하는 과정에서 결정한 것을 문서화한다.

- 다양한 영역에 집중하기 위해 다른 영역의 세부 사항은 숨기는 다양한 관점을 제공한다.
- 특정 목표에 따라 구체화된 상세 수준의 표현방법을 제공한다.

3. 데이터 모델링의 중요성 및 유의점

데이터 모델링이 중요한 이유는 파급효과(Leverage), 복잡한 정보 요구사항의 간결한 표현(Conciseness), 데이터 품질(Data Quality)로 정리할 수 있다.

가. 파급효과(Leverage)

시스템 구축이 완성되어 가는 시점에서는 많은 애플리케이션들이 테스트를 수행하고 대규모의 데이터 이행을 성공적으로 수행하기 위한 많은 단위 테스트들이 수행되고 이러한 과정들이 반복된다. 각 단위 테스트들이 성공적으로 수행되고 완료되면 이를 전체를 묶어서 병행테스트, 통합테스트를 수행하게 된다. 만약, 이러한 시점에 데이터 모델의 변경이 불가피한 상황이 발생한다고 가정해 보자. 이를 위해서 데이터 구조의 변경에 따른 표준 영향 분석, 응용 변경 영향 분석 등 많은 영향 분석이 일어난다. 그 이후에 해당 분야의 실제적인 변경 작업이 발생하게 된다. 변경을 해야 하는 데이터 모델의 형태에 따라서 그 영향 정도는 차이가 있겠지만 이 시기의 데이터 구조의 변경으로 인한 일련의 변경작업은 전체 시스템 구축 프로젝트에서 큰 위험요소가 아닐 수 없다. 이러한 이유로 인해 시스템 구축 작업 중에서 다른 어떤 설계 과정보다 데이터 설계가 더 중요하다고 볼 수 있다.

나. 복잡한 정보 요구사항의 간결한 표현(Conciseness)

데이터 모델은 구축할 시스템의 정보 요구사항과 한계를 가장 명확하고 간결하게 표현할 수 있는 도구이다. 정보 요구사항을 파악하는 가장 좋은 방법은 수많은 페이지의 기능적인 요구사항을 파악하는 것보다 간결하게 그려져 있는 데이터 모델을 리뷰하면서 파악하는 것이 훨씬 빠른 방법이다. 데이터 모델은 건축물로 비유하자면 설계 도면에 해당한다. 이것은 건축물의 설계 도면이 건축물을 짓는 많은 사람들이 공유하면서 설계자의 생각대로 일사불란하게 움직여서 아름다운 건축물을 만들어 내는 것에 비유할 수 있다. 데이터 모델은 시스템을 구축하는 많은 관련자들이 설계자의 생각대로 정보요구사항을 이해하고 이를 운용할 수 있는 애플리케이션을 개발하고 데이터 정합성을 유지할 수 있도록 하는 것이다. 이렇게 이상적으로 역할을 할 수 있는 모델이 갖추어야 할 가장 중요한 점은 정보 요구사항이 정확하고 간결하게 표현되어야 한다는 것이다. 우리가 활용하고 있는 데이터 모델이 이와 같은 요소들이 충족된 모델인지를 확인해 볼 필요가 있다.

다. 데이터 품질(Data Quality)

데이터베이스에 담겨 있는 데이터는 기업의 중요한 자산이다. 이 데이터는 기간이 오래되면 될수록 활용가치는 훨씬 높아진다. 그런데 이러한 오래도록 저장되어진 데이터가 그저 그런 데이터, 정확성이 떨어지는 데이터라고 한다면 어떨까? 이것은 일부 시스템의 기능이 잘못되어 수정하는 성격의 일이 아니다. 이것은 해당 데이터로 얻을 수 있었던 소중한 비즈니스의 기회를 상실할 수도 있는 문제이다. 데이터 품질의 문제가 중요한 이유가 여기에 있

다. 데이터 품질의 문제는 데이터 구조가 설계되고 초기에 데이터가 조금 쌓일 때에는 인지하지 못하는 경우가 대부분이다. 이러한 데이터의 문제는 오랜 기간 숙성된 데이터를 전략적으로 활용하려고 하는 시점에 문제가 대두되기 때문이다.
데이터 품질의 문제가 야기되는 중대한 이유 중 하나가 바로 데이터 구조의 문제이다. 중복 데이터의 미정의, 데이터 구조의 비즈니스 정의의 불충분, 동일한 성격의 데이터를 통합하지 않고 분리함으로써의 나타나는 데이터 불일치 등의 데이터 구조의 문제로 인한 데이터 품질의 문제는 치유하기에 불가능한 경우가 대부분이다. 데이터 모델링을 할 때 유의점은 다음과 같다.

1) 중복(Duplication)

데이터 모델은 같은 데이터를 사용하는 사람, 시간, 그리고 장소를 파악하는데 도움을 준다. 이러한 지식 응용은 데이터베이스가 여러 장소에 같은 정보를 저장하는 잘못을 하지 않도록 한다.

2) 비유연성(Inflexibility)

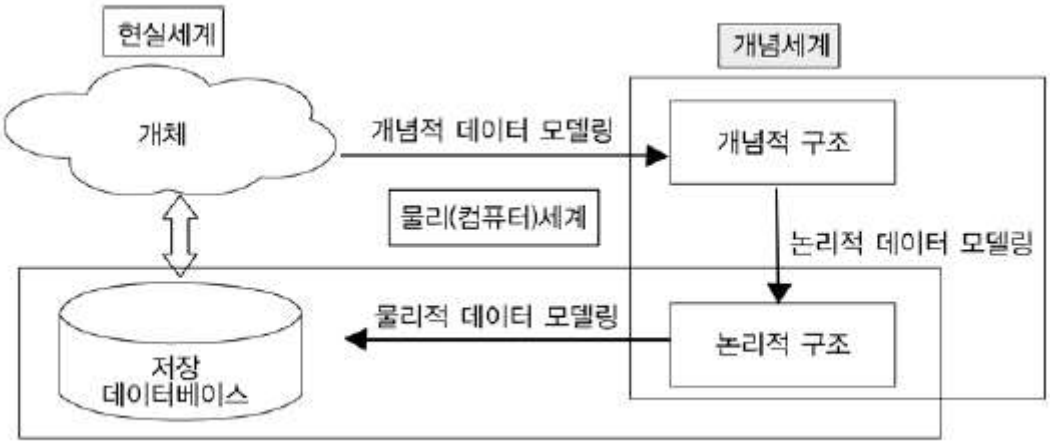
데이터 모델을 어떻게 설계했느냐에 따라 사소한 업무변화에도 데이터 모델이 수시로 변경됨으로써 유지보수의 어려움을 가중시킬 수 있다. 데이터의 정의를 데이터의 사용 프로세스와 분리함으로써 데이터 모델링은 데이터 혹은 프로세스의 작은 변화가 애플리케이션과 데이터베이스에 중대한 변화를 일으킬 수 있는 가능성을 줄인다.

3) 비일관성(Inconsistency)

데이터의 중복이 없더라도 비일관성은 발생한다. 예를 들어 신용 상태에 대한 갱신 없이 고객의 납부 이력 정보를 갱신하는 것이다. 개발자가 다른 데이터와 모순된다는 고려 없이 일련의 데이터를 수정할 수 있기 때문이다. 데이터 모델링을 할 때 데이터와 데이터간 상호 연관 관계에 대한 명확한 정의는 이러한 위험을 사전에 예방할 수 있도록 해준다.

4. 데이터 모델링의 3단계 진행

특별히 데이터 모델은 데이터베이스를 만들어내는 설계서로서 분명한 목표를 가지고 있다. 현실세계에서 데이터베이스까지 만들어지는 과정은 [그림 Ⅰ-1-3]과 같이 시간에 따라 진행되는 과정으로서 추상화 수준에 따라 개념적 데이터 모델, 논리적 데이터 모델, 물리적 데이터 모델로 정리할 수 있다.



[그림 Ⅰ-1-3] 현실세계와 데이터베이스 사이의 모델

처음 현실세계에서 추상화 수준이 높은 상위 수준을 형상화하기 위해 개념적 데이터 모델링을 전개한다. 개념적 데이터 모델은 추상화 수준이 높고 업무중심적이고 포괄적인 수준의 모델링을 진행한다. 참고로 EA기반의 전사적인 데이터 모델링을 전개할 때는 더 상위수준인 개괄적인 데이터 모델링을 먼저 수행하고 이후에 업무영역에 따른 개념적 데이터 모델링을 전개한다. 엔터티(Entity)중심의 상위 수준의 데이터 모델이 완성되면 업무의 구체적인 모습과 흐름에 따른 구체화된 업무중심의 데이터 모델을 만들어 내는데 이것을 논리적인 데이터 모델링이라고 한다. 논리적인 데이터 모델링 이후 데이터베이스의 저장구조에 따른 테이블스페이스 등을 고려한 방식을 물리적인 데이터 모델링이라고 한다. 이것을 요약하여 정리하면 [표 Ⅰ-1-1]과 같다.

[표 Ⅰ-1-1] 개념-논리-물리데이터 모델

데이터 모델링	내용	수준
개념적 데이터 모델링	추상화 수준이 높고 업무중심적이고 포괄적인 수준의 모델링 진행, 전사적 데이터 모델링, EA수립시 많이 이용	추상적
논리적 데이터 모델링	시스템으로 구축하고자 하는 업무에 대해 Key, 속성, 관계 등을 정확하게 표현, 재사용성이 높음	
물리적 데이터 모델링	실제로 데이터베이스에 이식할 수 있도록 성능, 저장 등 물리적인 성격을 고려하여 설계	구체적

가. 개념적 데이터 모델링(Conceptual Data Modeling)

개념적 데이터베이스 설계(개념 데이터 모델링)는 조직, 사용자의 데이터 요구사항을 찾고 분석하는데서 시작한다. 이 과정은 어떠한 자료가 중요하며 또 어떠한 자료가 유지되어야 하는지를 결정하는 것도 포함한다. 이 단계에 있어서의 주요한 활동은 핵심 엔터티와 그들 간의 관계를 발견하고, 그것을 표현하기 위해서 엔터티-관계 다이어그램을 생성하는 것이다. 엔터티-관계 다이어그램은 조직과 다양한 데이터베이스 사용자에게 어떠한 데이터가 중요한지 나타내기 위해서 사용된다. 데이터 모델링 과정이 전 조직에 걸쳐 이루어진다면,

그것은 전사적 데이터 모델(Enterprise Data Model)이라고 불린다. 개념 데이터 모델을 통해 조직의 데이터 요구를 공식화하는 것은 두 가지의 중요한 기능을 지원한다. 첫째, 개념 데이터 모델은 사용자와 시스템 개발자가 데이터 요구 사항을 발견하는 것을 지원한다. 개념 데이터 모델은 추상적이다. 그렇기 때문에 그 모델은 상위의 문제에 대한 구조화를 쉽게 하며, 사용자와 개발자가 시스템 기능에 대해서 논의할 수 있는 기반을 형성한다. 둘째, 개념 데이터 모델은 현 시스템이 어떻게 변형되어야 하는가를 이해하는데 유용하다. 일반적으로 매우 간단하게 고립된(Stand Alone) 시스템도 추상적 모델링을 통해서 보다 쉽게 표현되고 설명된다.

나. 논리적 데이터 모델링(Logical Data Modeling)

논리 데이터 모델링은 데이터베이스 설계 프로세스의 Input으로써 비즈니스 정보의 논리적인 구조와 규칙을 명확하게 표현하는 기법 또는 과정이라 할 수 있다. 논리 데이터 모델링의 결과로 얻어지는 논리 데이터 모델은 데이터 모델링이 최??리적인 스키마 설계를 하기 전 단계의 '데이터 모델'의 상태를 일컫는 말이다. 논리 데이터 모델링의 핵심은 어떻게 데이터에 액세스하고, 누가 데이터에 액세스하며, 그러한 액세스의 전산화와는 독립적으로 다시 말해서 누가(Who), 어떻게(How: Process) 그리고 전산화와는 별개로 비즈니스 데이터에 존재하는 사실들을 인식하여 기록하는 것이다.

데이터 모델링 과정에서 가장 핵심이 되는 부분이 논리 데이터 모델링이라고 할 수 있다. 데이터 모델링이란 모델링 과정이 아닌 별도의 과정을 통해서 조사하고 결정한 사실을 단지 ERD라는 그림으로 그려내는 과정을 말하는 것이 아니다. 시스템 구축을 위해서 가장 먼저 시작할 기초적인 업무조사를 하는 초기단계에서부터 인간이 결정해야 할 대부분의 사항을 모두 정의하는 시스템 설계의 전 과정을 지원하는 '과정의 도구'라고 해야 할 것이다.

이 단계에서 수행하는 또 한가지 중요한 활동은 정규화이다. 정규화는 논리 데이터 모델 상세화 과정의 대표적인 활동으로, 논리 데이터 모델의 일관성을 확보하고 중복을 제거하여 속성들이 가장 적절한 엔터티에 배치되도록 함으로써 보다 신뢰성있는 데이터구조를 얻는데 목적이 있다. 논리 데이터 모델의 상세화는 식별자 확정, 정규화, M:M 관계 해소, 참조 무결성 규칙 정의 등을 들 수 있으며, 추가적으로 이력 관리에 대한 전략을 정의하여 이를 논리 데이터 모델에 반영함으로써 데이터 모델링을 완료하게 된다.

다. 물리적 데이터 모델링(Physical Data Modeling)

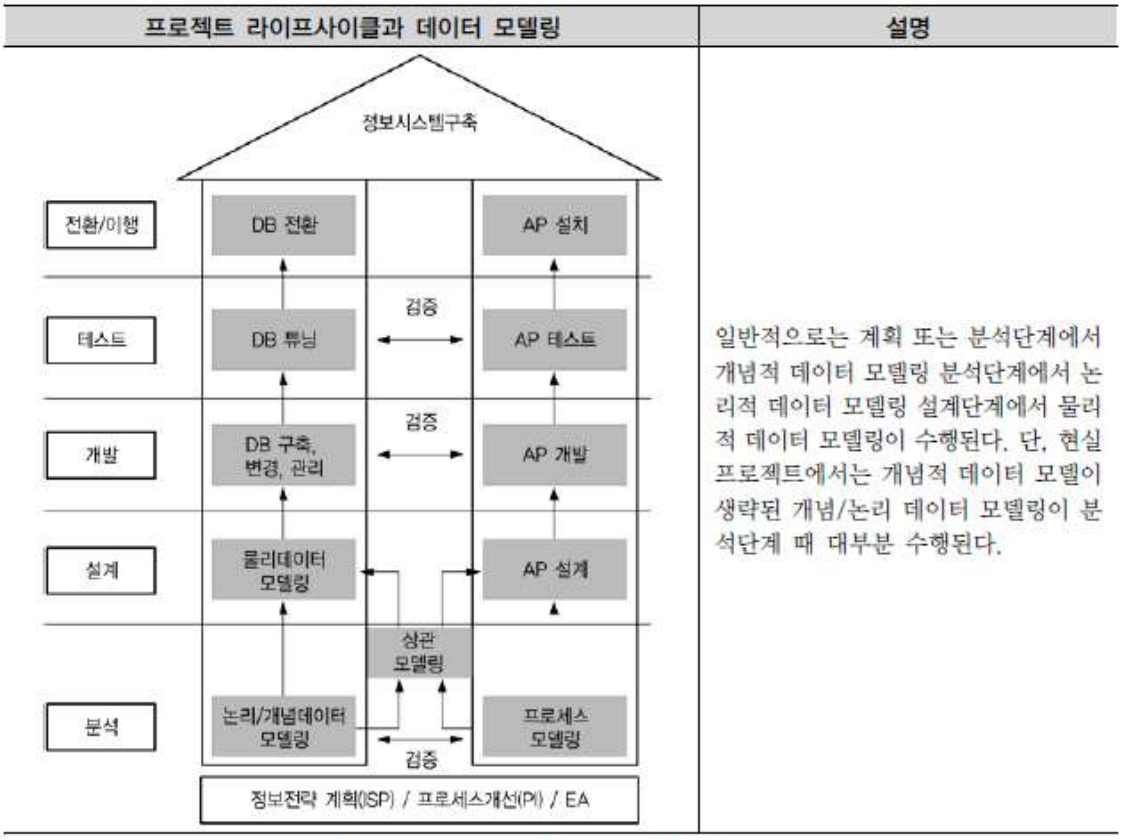
데이터베이스 설계 과정의 세 번째 단계인 물리 데이터 모델링은 논리 데이터 모델이 데이터 저장소로서 어떻게 컴퓨터 하드웨어에 표현될 것인가를 다룬다. 데이터가 물리적으로 컴퓨터에 어떻게 저장될 것인가에 대한 정의를 물리적 스키마라고 한다. 이 단계에서 결정되는 것은 테이블, 칼럼 등으로 표현되는 물리적인 저장구조와 사용될 저장 장치, 자료를 추출하기 위해 사용될 접근 방법 등이 있다. 계층적 데이터베이스 관리 시스템 환경에서는 데이터베이스 관리자가 물리적 스키마를 설계하고 구현하기 위해서 보다 많은 시간을 투자하여야 한다.

실질적인 현실 프로젝트에서는 개념적 데이터 모델링 논리적 데이터 모델링 물리적 데이터 모델링으로 수행하는 경우는 드물며 개념적 데이터 모델링과 논리적 데이터 모델을 한꺼

번에 수행하여 논리적인 데이터 모델링으로 수행하는 경우가 대부분이다. 프로젝트 생명주기에 따른 일반적인 데이터 모델은 다음과 같이 수행된다.

5. 프로젝트 생명주기(Life Cycle)에서 데이터 모델링

Waterfall 기반에서는 데이터 모델링의 위치가 분석과 설계단계로 구분되어 명확하게 정의할 수 있다. 정보공학이나 구조적 방법론에서는 보통 분석단계에서 업무중심의 논리적인 데이터 모델링을 수행하고 설계단계에서 하드웨어와 성능을 고려한 물리적인 데이터 모델링을 수행하게 된다. 나선형 모델, 예를 들어 RUP(Rational Unified Process나 마르미)에서는 업무크기에 따라 논리적 데이터 모델과 물리적 데이터 모델이 분석, 설계단계 양쪽에서 수행이 되며 비중은 분석단계에서 논리적인 데이터 모델이 더 많이 수행되는 형태가 된다.



[그림 Ⅰ-1-4] 프로젝트 생명주기에 따른 데이터 모델

데이터축과 애플리케이션축으로 구분되어 프로젝트가 진행되면서 각각에 도출된 사항은 상호 검증을 지속적으로 수행하면서 단계별 완성도를 높이게 된다. 단, 객체지향 개념은 데이터와 프로세스를 한꺼번에 바라보면서 모델링을 전개하므로 데이터 모델링과 프로세스 모델링을 구분하지 않고 일체형으로 진행(대표적인 예가 데이터(속성)와 프로세스(Method)가 같이 있는 클래스(Class))하게 된다.

6. 데이터 모델링에서 데이터독립성의 이해

가. 데이터독립성의 필요성

일체적 구성에서 기능화된 구성의 가장 큰 목적은 상호간 영향에서 벗어나 개별 형식이 가지는 고유의 기능을 유지시키며 그 기능을 극대화하는 것이다. 컴포넌트 기반의 모듈 구성도 각각이 고유한 기능을 가지면서 다른 기능을 가지고 있는 컴포넌트와 인터페이스를 가지게 하는 모습으로 정의할 수 있다. SOA의 '서비스'라고 하는 단위도 독립적인 비즈니스로 처리 가능한 단위를 서비스로 정의하고 그것이 다른 서비스에 비해 독립성을 구성하여 개별로도 의미가 있고 다른 서비스와 결합하여 프로세스로 제공해도 의미가 있는 단위(예, BPM)로 제공하게 된다.

이와 같이 어떤 단위에 대해 독립적인 의미를 부여하고 그것을 효과적으로 구현하게 되면 자신이 가지는 고유한 특징을 명확하게 할 뿐만 아니라 다른 기능의 변경으로부터 쉽게 변경되지 않고 자신의 고유한 기능을 가지고 기능을 제공하는 장점을 가지게 된다.

데이터독립성을 이해하기 위해서는, 데이터독립성이라고 하는 개념이 출현하게 된 배경을 이해할 필요가 있다. 데이터독립성의 반대말은 데이터종속성이다. 여기에서 종속의 주체는 보통 응용(Application)을 지칭하는 경우이다. 응용(Application)은 사용자 요구사항을 처리하는 사용자 접점의 인터페이스 오브젝트이다. 과거에 파일방식으로 데이터를 구성할 때는 데이터가 있는 파일과 데이터에 접근하기 위한 인덱스를 별도로 구현하여 접근하게 하였는데 사용자가 접근하는 방법(트랜잭션의 유형)에 따라 파일의 정렬순서, 인덱스의 정렬순서, 파일 구성 등을 제공하기 쉽게 별도로 구성하였다. 즉, 사용자 접근하는 유형에 따라 데이터를 구성하는 방법이 영향을 받게 된다. 메인프레임 환경에서 파일방식을 사용하여 데이터처리를 했던 메인프레임 세대는 개별로 처리했던 접근방법을 이해할 수 있으나 1990년대 이후의 Client/Sever 이후 세대는 파일처리 방식 이해가 난해할 수도 있다.

데이터독립성은 지속적으로 증가하는 유지보수 비용을 절감하고 데이터 복잡도를 낮추며 중복된 데이터를 줄이기 위한 목적이 있다. 또한 끊임없이 요구되는 사용자 요구사항에 대해 화면과 데이터베이스 간에 서로 독립성을 유지하기 위한 목적으로 데이터 독립성 개념이 출현했다고 할 수 있다.

[그림 Ⅰ-1-5] 데이터독립성의 필요성

데이터독립성은 미국 표준 협회(ANSI) 산하의 X3 위원회(컴퓨터 및 정보 처리)의 특별연구분과위원회에서 1978년에 DBMS와 그 인터페이스를 위해 제안한 'three-schema architecture'로 정의할 수 있다.

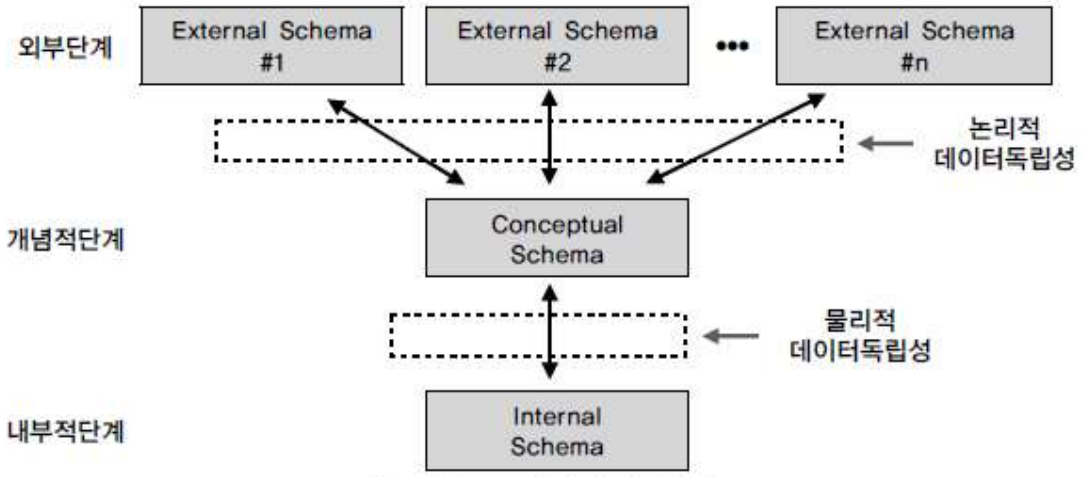
데이터독립성을 확보하게 되면 다음과 같은 효과를 얻을 수 있다.

- 각 View의 독립성을 유지하고 계층별 View에 영향을 주지 않고 변경이 가능하다.
- 단계별 Schema에 따라 데이터 정의어(DDL)와 데이터 조작어(DML)가 다름을 제공한다.

데이터독립성을 이해하기 위해서는 3단계로 표현된 ANSI 표준 모델을 살펴보면 되는데 특히 3단계인 구조, 독립성, 사상(Mapping) 3가지를 이해하면 된다.

나. 데이터베이스 3단계 구조

ANSI/SPARC의 3단계 구성의 데이터독립성 모델은 외부단계와 개념적 단계, 내부적 단계로 구성된 서로 간섭되지 않는 모델을 제시하고 있다.



[그림 Ⅰ-1-6] 데이터독립성

데이터독립성의 3단계에서 외부단계는 사용자와 가까운 단계로 사용자 개개인이 보는 자료에 대한 관점과 관련이 있스키마 구조데이터 유형의 공통적인 사항을 처리하는 통합된 뷰를 스키마 구조로 디자인한 형태이다. 우리가 쉽게 이해하는 데이터 모델은 사용자가 처리하는 통합된 뷰를 설계하는 도구로 이해해도 무방하다. 마지막으로 내부적 단계는 데이터가 물리적으로 저장된 방법에 대한 스키마 구조를 말한다.

다음에서 [그림 Ⅰ-1-6]의 3단계 구조의 상세 사항에 대해 각 구성별로 예를 들어 설명한다.

다. 데이터독립성 요소

[표 Ⅰ-1-2] 데이터독립성 구성요소

항목	내용	비고
외부스키마 (External Schema)	- View 단계 여러 개의 사용자 관점으로 구성, 즉 개개 사용자 단계로서 개개 사용자가 보는 개인적 DB 스키마 - DB의 개개 사용자나 응용프로그래머가 접근하는 DB 정의	사용자 관점 접근하는 특성에 따른 스키마 구성
개념스키마 (Conceptual) Schema	- 개념단계 하나의 개념적 스키마로 구성 모든 사용자 관점을 통합한 조직 전체의 DB를 기술하는 것 - 모든 응용시스템들이나 사용자들이 필요로 하는 데이터를 통합한 조직 전체의 DB를 기술한 것으로 DB에 저장되는 데이터와 그들간의 관계를 표현하는 스키마	통합관점
내부스키마 (Internal Schema)	- 내부단계, 내부 스키마로 구성, DB가 물리적으로 저장된 형식 - 물리적 장치에서 데이터가 실제적으로 저장되는 방법을 표현하는 스키마	물리적 저장구조

데이터베이스 스키마 구조는 3단계로 구분되고 각각은 상호 독립적인 의미를 가지고 고유한 기능을 가진다. 데이터 모델링은 통합관점의 뷰를 가지고 있는 개념 스키마를 만들어 가는 과정으로 이해할 수 있다.

라. 두 영역의 데이터독립성

이렇게 3단계로 개념이 분리되면서 각각의 영역에 대한 독립성을 지정하는 용어가 바로 논리적인 독립성과 물리적인 독립성이다.

[표 Ⅰ-1-3] 논리적, 물리적 데이터독립성

독립성	내용	특징
논리적 독립성	- 개념 스키마가 변경되어도 외부 스키마에는 영향을 미치지 않도록 지원하는 것 - 논리적 구조가 변경되어도 응용 프로그램에 영향 없음	- 사용자 특성에 맞는 변경가능 - 통합 구조 변경가능
물리적 독립성	- 내부스키마가 변경되어도 외부/개념 스키마는 영향을 받지 않도록 지원하는 것 - 저장장치의 구조변경은 응용프로그램과 개념스키마에 영향 없음	- 물리적 구조 영향 없이 개념구조 변경가능 - 개념구조 영향 없이 물리적인 구조 변경가능

즉, 논리적인 데이터독립성은 외부의 변경에도 개념스키마가 변하지 않는 특징을 가진다. 물론, 새로운 요건이 추가되거나 삭제될 경우 칼럼이 변형될 수 있지만 그러한 변화가 개별 화면이나 프로세스에 의해 변화된다기 보다는 전체 업무적인 요건을 고려하여 종합적으로 영향을 받음을 의미한다.

마. 사상(Mapping)

영어로 'Mapping'은 우리말로 '사상'이라고 번역되는데 이것은 상호 독립적인 개념을 연결시켜주는 다리를 뜻한다. 데이터독립성에서는 크게 2가지의 사상이 도출된다.

[표 Ⅰ-1-4] 사상(Mapping)

사상	내용	예
외부적/개념적 사상 (논리적 사상)	- 외부적 뷰와 개념적 뷰의 상호 관련성을 정의함	사용자가 접근하는 형식에 따라 다른 타입의 필드를 가질 수 있음. 개념적 뷰의 필드 타입은 변화가 없음
개념적/내부적 사상 (물리적 사상)	- 개념적 뷰와 저장된 데이터베이스의 상호관련성 정의	만약 저장된 데이터베이스 구조가 바뀐다면 개념적/내부적 사상이 바뀌어야 함. 그래야 개념적 스키마가 그대로 남아있게 됨

즉, 외부 화면이나 사용자에게 인터페이스하기 위한 스키마 구조는 전체가 통합된 개념적 스키마와 연결된다는 것이 논리적 사상이다. 또한 통합된 개념적 스키마 구조와 물리적으로 저장된 구조의 물리적인 테이블스페이스와 연결되는 구조가 물리적 사상이다. 데이터 독립성을 보장하기 위해서는 사상을 하는 스크립트(DDL)를 DBA가 필요할 때마다 변경해 주어야 한다. 즉, 각 단계(외부, 개념적, 내부적)의 독립성을 보장하기 위해서 변경사항이 발생했을 때 DBA가 적절하게 작업을 해주기 때문에 독립성이 보장된다고도 할 수 있다.

7. 데이터 모델링의 중요한 세 가지 개념

가. 데이터 모델링의 세 가지 요소

데이터 모델링을 구성하는 중요한 개념 세 가지가 있는데 이것은 데이터 모델에 대한 이해의 근간이 되므로 반드시 기억할 필요가 있다.

1) 업무가 관여하는 어떤 것(Things)
2) 어떤 것이 가지는 성격(Attributes)
3) 업무가 관여하는 어떤 것 간의 관계(Relationships)

이 세 가지는 데이터 모델링을 완성해 가는 핵심 개념으로서 결국 엔터티, 속성, 관계로 인식되는 것이다. 사물이나 사건 등을 바라 볼 때 전체를 지칭하는 용어를 어떤 것(Things)이라 하고, 그 어떤 것이 가지는 세부적인 사항을 성격(Attributes)이라고 할 수 있다. 또한 각각의 어떤 것은 다른 어떤 것과 연관성을 가질 수 있는데 이것을 관계(Relationship)라고 표현한다.

예를 들어 '이주일과 심순애가 존재하고 둘 사이는 서로 사랑하는 연인사이이다. 이주일은 키가 180cm에 성격은 친절하고 심순애는 키가 165cm에 세심하며 활달한 성격을 가지고 있다'는 시나리오를 살펴보자. 여기에서 '이주일, 심순애'는 어떤 것(Things)에 해당하고 '사랑하는 연인사이'가 어떤 것 간의 관계(Relationships)에 해당하며 '180cm에 성격은 친절, 세심하며 활달함'이 어떤 것이 가지는 성격(Attributes)에 해당한다. 위의 예와 같이 이

세상의 모든 사람, 사물, 개념 등은 어떤 것, 어떤 것 간의 관계, 성격의 구분을 통해서 분류할 수 있다. 바로 이러한 원리, 즉 자연계에 존재하는 모든 유형의 정보들을 세 가지 관점의 접근 방법을 통해 모델링을 진행하는 것이다.

나. 단수와 집합(복수)의 명명

데이터 모델링에서는 이 세 가지 개념에 대해서 단수와 복수의 개념을 분명하게 구분하고 있고 실제로 데이터 모델링을 할 때 많이 활용되는 용어이다.

[표 Ⅰ-1-5] 용어의 구분정의

개념	복수/집합개념 타입/클래스	개별/단수개념 어커런스/인스턴스
어떤 것 (Thing)	엔터티 타입(Entity Type)	엔터티(Entity)
	엔터티(Entity)	인스턴스(Instance), 어커런스(Occurrence)
어떤 것간의 연관 (Association between Things)	관계(Relationship)	패어링(Pairing)
어떤 것의 성격 (Characteristic of a Thing)	속성(Attribute)	속성값(Attribute Value)

어떤 것의 전체를 지칭하는 것을 영문으로 Entity Set, Entity Type의 복수의 의미를 가지는 Set이나 Type을 포함하여 표현하기도 한다. 그래서 엔터티 타입으로 표현하기도 하는데 실제 실무 현장에서는 엔터티로 짧게 명명한다. 즉 엔터티를 어떤 것에 대한 집합을 명명하여 지칭한다. 어떤 것에 대한 개별 지칭으로 엔터티가 단수명사로서 단어의 의미가 있지만, 엔터티를 집합개념으로 사용하는 경우에는 개별요소에 대해서는 인스턴스/어커런스를 단수의 개념으로 사용하여 부른다. 관계(Relationship)도 이를 복수로 통칭하여 관계로 표현하고 관계에 포함된 개별 연관성을 패어링이라고 부르기도 한다. 그러나 패어링이라는 용어는 실제 데이터 모델링을 할 때는 잘 사용하지 않으며 그냥 일반적으로 단수든 복수든 관계라고 표현하는 경우가 많다. 어떤 것이 가지는 성격(Attribute)에 대한 집합개념이 속성이고 그 안에 개별 값들을 속성값으로 구분하여 복수와 단수의 개념으로 구분할 수 있다.

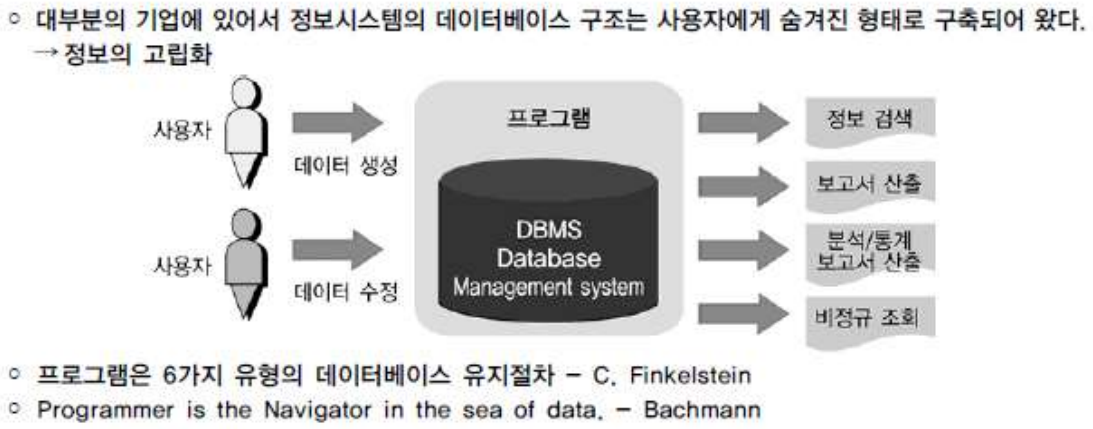
본 가이드에서는 현장 통용성을 반영하여 국내외적으로 가장 범용적으로 명명되고 있는 용어인 엔터티를 집합의 개념으로 지칭하고 인스턴스를 단수의 개념으로 명명하도록 한다. 데이터 모델의 핵심 요소인 이 세 가지를 이용하여 일정한 표기법에 의해 데이터 모델을 만들어 낼 수 있다. 다음은 다양한 표기법에 의해 생성되는 데이터 모델의 표기법을 설명한다.

8. 데이터 모델링의 이해관계자

가. 이해관계자의 데이터 모델링 중요성 인식

실제 업무시스템을 구축하는 실전 프로젝트에서는 데이터베이스를 전문적으로 하는 이른바 DBA(DataBase Administrator)가 데이터 모델링을 전적으로 하는 예는 거의 없다. 오히려 업무시스템을 개발하는 응용시스템 개발자가 데이터 모델링도 같이 하게 된다. 그 이유는 데이터 모델링이라는 과정이 단지 데이터베이스를 설계한다는 측면보다 업무를 이해하고 분석하여 표현하는 것이 중요하고, 표현된 내용을 바탕으로 프로젝트 관련자와 의사소통하고 프로그램이나 다른 표기법과 비교 검증하는 일을 수행하는 등 많은 시간을 업무를 분석하고 설계하는데 할애하기 때문에 업무영역별 개발팀에서 보통 데이터 모델링을 진행하게 되는 것이다. 물론 시스템이 대형화되면 모델링만을 전문적으로 담당하는 모델러를 투입하여 진행하는 경우도 있지만 이와 같은 경우도 실제 모델링 작업은 응용개발을 하는 사람이나 업무분석가(역할분담이 잘되어 있을 경우)가 담당하고 모델러나 DBA는 정확하게 모델링이 진행될 수 있도록 교육하고 제시하며 현안별로 직접 모델링을 진행하는 역할을 수행한다.

이와 같이 응용시스템을 개발하는 모든 시스템 엔지니어가 데이터 모델을 하거나 할 기회가 있음에도 불구하고 대부분의 사람들은 데이터 모델에 많은 관심을 갖지 않고 단지 프로그램을 개발하기 위한 프로그래밍 언어(Programming Language)에만 많은 관심을 두고 애플리케이션을 개발하는 데에 훨씬 많은 시간을 투자하는 경우가 많다. 그러나 분명한 사실은 정보시스템을 개발한다고 할 때 데이터 모델링, 데이터베이스 구축, 구축된 데이터의 적절한 활용은 다른 어떤 타스크보다 중요하다는 점이다.



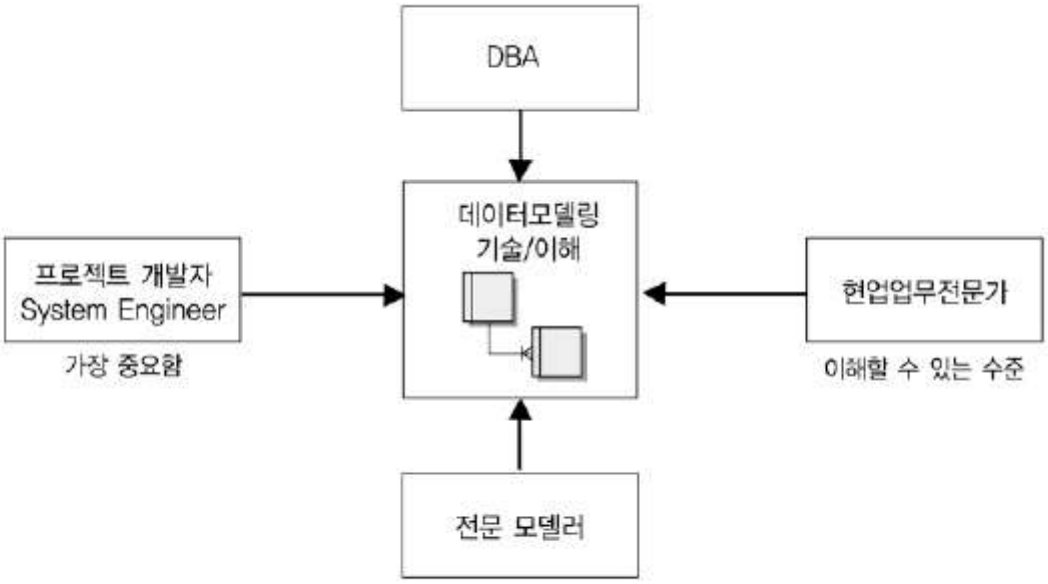
[그림 Ⅰ-1-7] 자료 처리의 중심은 데이터

우리가 구축하려는 시스템 대부분을 데이터에 기반한, 데이터가 중심에 있는 정보시스템을 구축하기 때문에 정보시스템의 핵심에 있는 데이터베이스 설계를 잘못했을 때 미치는 영향력은 모든 프로그램, 시간에 따라 입력되는 모든 데이터, 그리고 그 데이터베이스에 발생되는 모든 트랜잭션에 영향을 미칠 수 밖에 없게 된다. Bachmann은 '프로그래머는 데이터집합의 탐색자이다'라고 하였다. 그만큼 데이터에 대한 중요성을 높게 평가하는 것이다.

나. 데이터 모델링의 이해관계자

그러면 누가 데이터 모델링에 대해 연구하고 학습해야 하겠는가? 첫 번째는 정보시스템을 구축하는 모든 사람(전문적으로 코딩만하는 사람 포함)은 데이터 모델링도 전문적으로 할 수 있거나 적어도 완성된 모델을 정확하게 해석할 수 있어야 한다. 즉 프로젝트에 참여한 모든 IT기술자들은 데이터 모델링에 대해 정확하게 알고 있어야 한다는 것을 의미한다.

두 번째는 IT기술에 종사하거나 전공하지 않았더라도 해당 업무에서 정보화를 추진하는 위치에 있는 사람도 데이터 모델링에 대한 개념 및 세부사항에 대해 어느 정도 지식을 가지고 있어야 한다. 실제 프로젝트에서 보면 업무분석을 하는 도중에 현업의 업무담당자가 어느 사이에 데이터 모델링에 대해 상당한 이해를 하고 있음을 알게 된다. 그래야만 서로가 프로젝트 수행 중에 의사소통을 잘 할 수 있고 업무를 잘못 해석하여 잘못된 시스템을 구축하는 위험(Risk)을 줄일 수 있게 된다. 업무를 가장 잘 알고 있는 사람이 가장 훌륭한 모델러가 될 수 있다는 사실을 나타내는 예이다.



[그림 Ⅰ-1-8] 데이터 모델 이해관계자

9. 데이터 모델의 표기법인 ERD의 이해

가. 데이터 모델 표기법

데이터 모델에 대한 표기법으로 1976년 피터첸(Peter Chen)이 Entity-relationship model(E-R Model)이라는 표기법을 만들었다. 엔터티를 사각형으로 표현하고 관계를 마름모 속성을 타원형으로 표현하는 이 표기법은 데이터 모델링에 대한 이론을 배울 때 많이 활용되고 있다. 데이터베이스 설계에 대해 우리나라 대학에서는 주로 이 Chen의 모델 표기법을 통해 배우고 있다. [표 Ⅰ-1-6]은 엔터티와 속성 그리고 관계에 대한 다양한 표기법을 설명한 것이다

[표 Ⅰ-1-6] 표기법

표기법	설명
Chen 부서 1 소속 한다 N 직원	- 대학교재에서 가장 많이 이용하는 표기법 - 실무적으로 사용안함
IDEF1X 부서 ●----------◇ 직원	- 마름모와 원을 이용한 표기법으로 실무현장에서는 소수 활용 - ERWin
IE/Crow's Foot 부서 ┼ 포함한다 ──── 소속된다 ─<< 직원	- 까마귀발 모양의 표기법으로 가장 많이 사용함 - ERWin, ERStudio
Min-Max/ISO 부서 (0, N) 포함한다 ──── (1, 1) 소속된다 직원	- 기수성을 좀 더 정교하게 표현한 방법으로 많이 활용안됨
UML 《Relationship》 《Entity》 부서 1 직원을 포함하다 ──── 0..N 《Entity》 직원	- 스테레오타입을 이용하여 엔터티 표현 - UML로 표현하여 데이터 모델링 할 때 사용 - Rational Rose
Case*Method/Barker 부서 ┼ 1 ---------- 0..N <<─ 직원	- Crow's Foot을 적용하면서 관계 표기법 등 일부 다름(Barker's Notation) - DA#

데이터아키텍처 전문가(DAP) 관련 자격에서는 바커(Barker) 표기법을 적용하여 설명했다면, 본 가이드에서는 범용적인 Information Engineering(이하 IE) 표기법과 바커 표기법을 모두 적용하여 설명을 진행하도록 한다. 표기법은 바커 표기법이든 IE 표기법이든 상호간에 기술적으로 전환이 가능하기 때문에 한 가지만 정확하게 알고 있어도 다른 표기법을 이해하는데 큰 어려움이 없을 것이다.

나. ERD(Entity Relationship Diagram) 표기법을 이용하여 모델링하는 방법

ERD는 각 업무분석에서 도출된 엔터티와 엔터티간의 관계를 이해하기 쉽게 도식화된 다이어그램으로 표시하는 방법으로서 실제 프로젝트에서는 도식화된 그림 정도로만 생각하지 않고 해당 업무에서 데이터의 흐름과 프로세스와의 연관성을 이야기하는 데 가장 중요한 표기법이자 산출물이다.

UML 표준 표기법을 사용하는 오브젝트 모델링에서는 궁극적으로 해당 업무에 가장 적절한 클래스다이어그램을 그려내는 것이 가장 중요하다고 하면, 정보공학을 기반으로 하는 모델링에서는 해당 업무에 가장 적절한 ERD를 그려내는 것이 프로젝트의 지상과제이다. 오브젝트 모델링을 하더라도 관계형 데이터베이스를 대부분 사용하기 때문에 데이터베이스를 생성할 수 있는 데이터 모델 생성이 프로젝트에서 아주 중요한 타스크에 포함된다.

데이터분석이 어느 정도 완료되면 즉 엔터티, 관계, 속성 등이 데이터사전이나 각종 산출물에 의해 분석된 상태에서 ERD를 그리는 것이 원래 이론적인 작업 방법이지만, 실제 프로젝트에서는 분석된 엔터티와 관계, 속성 정보가 바로 ERD에 표현되며 내부 프로젝트 인원이나 해당 업무고객과 대화할 때 핵심 업무산출물로 항상 이용된다.

ERD를 그리는 것은 물론 어떻게 그리든 업무에는 전혀 지장을 주지 않지만 일정한 규칙을 지정하여 그림으로써 데이터 모델을 누구나 공통된 시각으로 파악할 수 있고 의사소통을 원활하게 하는 장점이 있다.

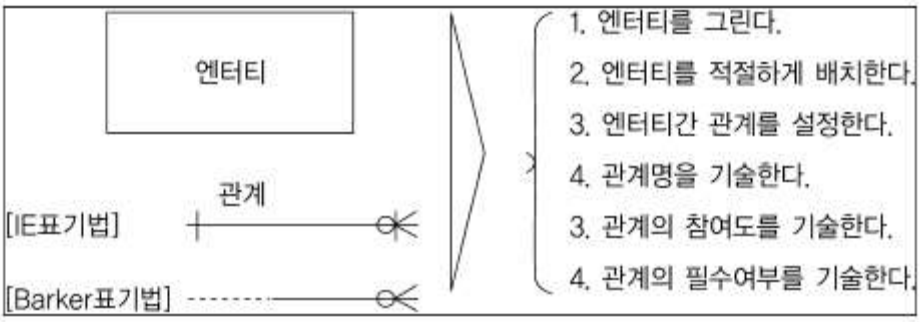
여기에서 제시하는 방법은 가이드일 뿐이며 프로젝트 상황과 엔터티의 관련 순서에 따라 얼마든지 다르게 배치될 수 있음을 숙지하고 배치방법에 대한 원칙을 다음과 같이 설명한다. 최근에는 전문 데이터 모델링 툴을 활용하여 ERD를 그리게 되므로 다음과 같이 엔터티, 속성, 관계 순으로 진행하기 보다는 엔터티와 관계를 바로 표현하는 방식으로 진행하기도 한다. 다만 업무를 분석하여 무엇을 도출할 지에 대한 관점을 제시하고, 전문 데이터 모델링 툴이 없을 때 어떤 순서로 표현해야 하는지 알 수 있도록 방법을 설명한다.

1) ERD 작업순서

ERD를 작성하는 작업순서는 다음과 같다.

① 엔터티를 그린다. ② 엔터티를 적절하게 배치한다. ③ 엔터티간 관계를 설정한다.

④ 관계명을 기술한다. ⑤ 관계의 참여도를 기술한다. ⑥ 관계의 필수여부를 기술한다.



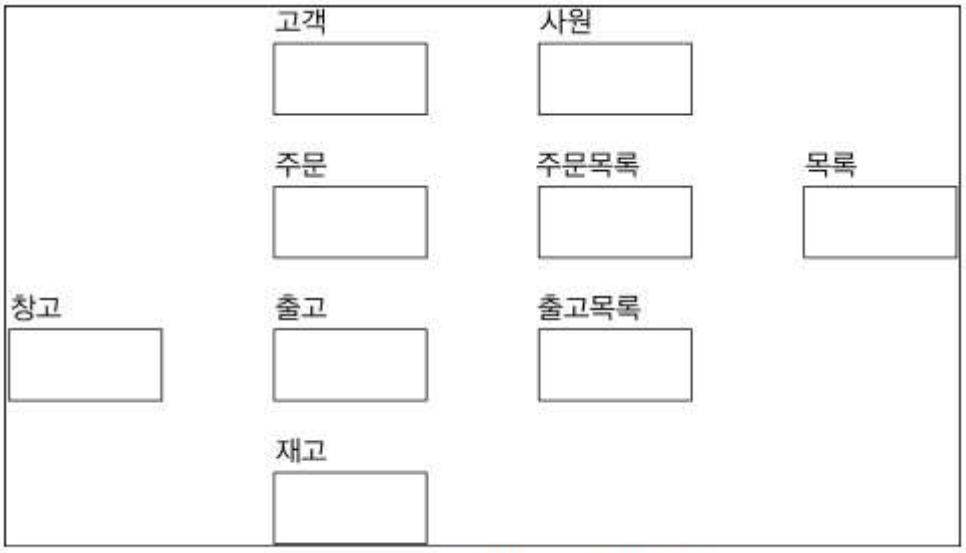
[그림 Ⅰ-1-9] ERD 작업순서

ERD는 엔터티와 엔터티 사이의 관계가 있는 정보를 나타내므로 두 개를 이용하여 작성하고, 이에 따라 Primary Key와 Foreign Key를 ERD 규칙에 따라 기술하도록 한다. 엔터티는 사각형으로 표기하여 기술한다.

2) 엔터티 배치

엔터티를 처음에 어디에 배치하는지는 데이터 모델링 툴을 사용하든 사용하지 않던 중요한 문제이다. 일반적으로 사람의 눈은 왼쪽에서 오른쪽, 위 쪽에서 아래쪽으로 이동하는 경

향이 있다. 따라서 데이터 모델링에서도 가장 중요한 엔터티를 왼쪽상단에 배치하고 이것을 중심으로 다른 엔터티를 나열하면서 전개하면 사람의 눈이 따라가기에 편리한 데이터 모델링을 전개할 수 있다. 해당 업무에서 가장 중요한 엔터티는 왼쪽 상단에서 조금 아래쪽 중앙에 배치하여 전체 엔터티와 어울릴 수 있도록 하면 향후 관계를 연결할 때 선이 꼬이지 않고 효과적으로 배치할 수 있게 된다.



[그림 Ⅰ-1-10] 엔터티 배치방법

[그림 Ⅰ-1-10]의 데이터 모델에서도 가장 중요한 엔터티인 고객과 주문을 왼쪽 상단에 배치하여 다른 엔터티를 연결하는 방식으로 엔터티를 배치하였다. 주문에 따라 출고가 이루어졌으므로 주문이 위에 출고가 아래에 위치해 있다.

두 번째 업무흐름에 중심이 되는 엔터티, 보통 업무 흐름에 있어서 중심이 되는 엔터티는 타 엔터티와 많은 관계를 가지고 있으므로 중앙에 배치하도록 한다. [그림 Ⅰ-1-10]에서는 주문, 출고, 주문목록, 출고목록이 업무의 중심엔터티에 해당한다.

세 번째는 업무를 진행하는 중심엔터티와 관계를 갖는 엔터티들은 중심에 배치된 엔터티를 주위에 배치하도록 한다. [그림 Ⅰ-1-10]에서는 창고, 고객, 사원, 재고가 이에 해당한다.

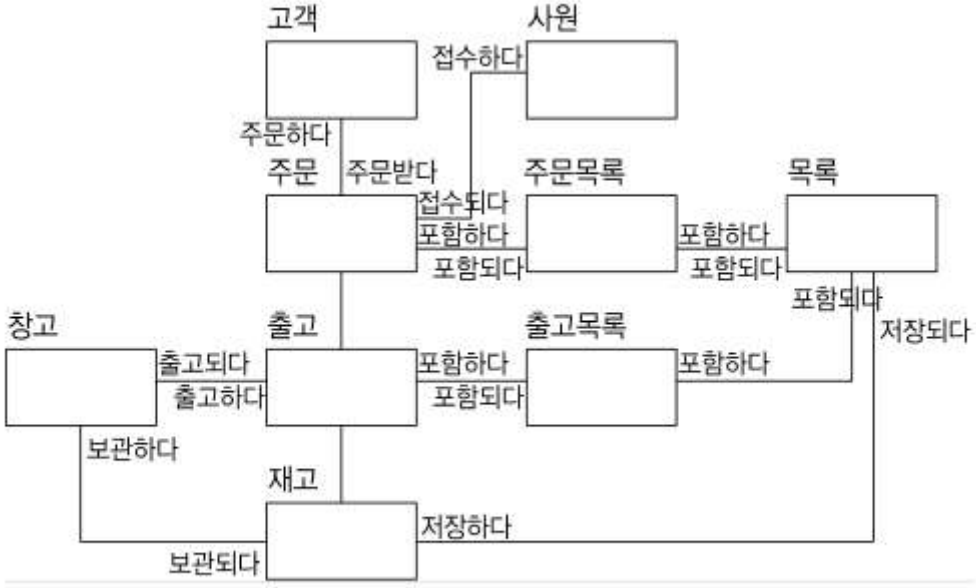
3) ERD 관계의 연결

언터티에 배치가 되면 관계를 정의한 분석서를 보고 서로 관련있는 엔터티간에 관계를 설정하도록 한다. 초기에는 모두 Primary Key로 속성이 상속되는 식별자 관계를 설정하도록 한다. 중복되는 관계가 발생되지 않도록 하고 Circle 관계도 발생하지 않도록 유의하여 작성하도록 한다.

[그림 Ⅰ-1-11] ERD 관계설정

4) ERD 관계명의 표시

관계설정이 완료되면 연결된 관계에 관계이름을 부여하도록 한다. 관계이름은 현재형을 사용하고 지나치게 포괄적인 용어(예, 이다, 가진다 등)는 사용하지 않도록 한다.

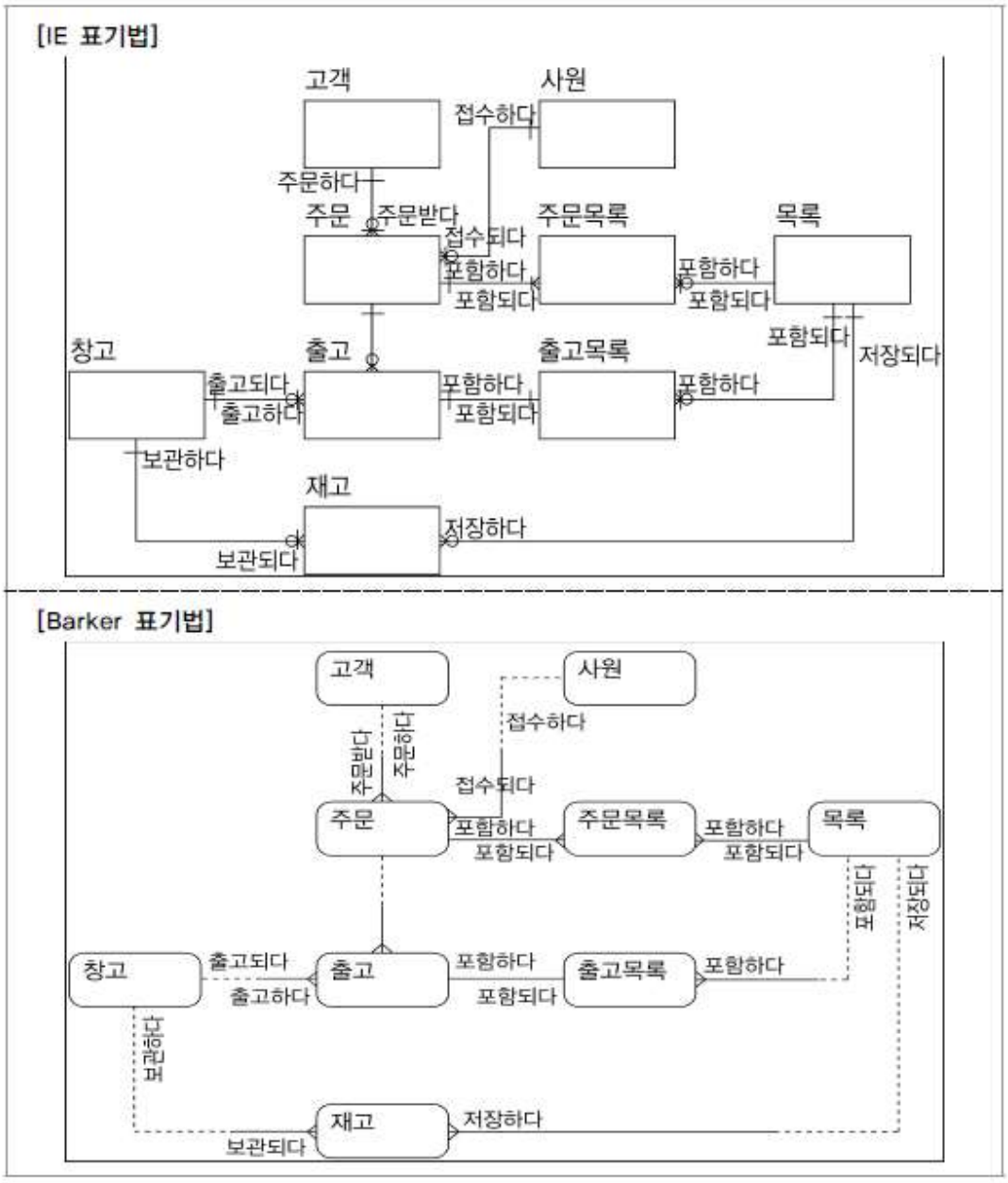


[그림 Ⅰ-1-12] ERD 관계명 표시

실제 프로젝트에서는 관계의 명칭을 크게 고려하지 않아도 무방하다. 왜냐하면 관계의 명칭이 나타나지 않아도 ERD의 흐름이 명확하게 드러나기 때문이다. 대부분의 관계는 엔터티의 성질과 주식별자를 보고 유추가 가능하다.

5) ERD 관계 관계차수와 선택성 표시

관계에 대한 이름을 모두 지정하였으면 관계가 참여하는 성격 중 엔터티내에 인스턴스들이 얼마나 관계에 참여하는 지를 나타내는 관계차수(Cardinality)를 표현한다. [그림 Ⅰ-1-13]은 관계의 관계차수를 지정한 ERD의 모습을 보여준다. 관계설명에서도 언급하겠지만 IE표기법으로는 하나(1, One)의 관계는 실선으로 표기하고 Barker표기법으로는 점선과 실선을 혼합하여 표기한다. 다수참여(Many)의 관계는 까마귀발과 같은 모양으로 그려준다. 또한 관계의 필수/선택표시는 관계선에 원을 표현하여 ERD를 그리도록 한다.



[그림 Ⅰ-1-13] 관계차수와 선택성 표시

10. 좋은 데이터 모델의 요소

일반적으로 시스템 구축 과정에서 생성되는 데이터 모델은 그 품질을 평가하는 것이 매우 어렵다. 사실 특정 데이터 모델이 업무 환경에서 요구하는 사항을 얼마나 잘 시스템적으로 구현할 수 있는가를 객관적으로 평가할 수 있다면 가장 좋은 평가의 방법일 것이다. 하지만 어디에도 이것을 객관적으로 평가할 수 있는 기준이 존재하지는 않는 것이 현실이다. 본 가이드에서는 이러한 상황에서 대체적으로 좋은 데이터 모델이라고 말할 수 있는 몇 가지의 요소들을 설명한다.

가. 완전성(Completeness)

업무에서 필요로 하는 모든 데이터가 데이터 모델에 정의되어 있어야 한다. 데이터 모델을 검증하기 위해서 가장 먼저 확인해야 할 부분이다. 이 기준이 충족되지 못하면 다른 어떤 평가 기준도 의미가 없어진다. 만약, 보험사의 데이터 모델에 고객의 직업을 관리하기 위한 속성이 존재하지 않는다면 어떨까? 이것은 심각한 데이터 모델의 문제점이다.

나. 중복배제(Non-Redundancy)

하나의 데이터베이스 내에 동일한 사실은 반드시 한 번만 기록하여야 한다. 예를 들면, 하나의 테이블에서 '나이' 칼럼과 '생년월일' 칼럼이 동시에 존재한다면 이것은 데이터 중복이라 볼 수 있다. 이러한 형태의 데이터 중복 관리로 인해서 여러 가지 바람직하지 않은 형태의 데이터 관리 비용을 지불할 수 있다. 예를 들면, 저장공간의 낭비, 중복 관리되고 있는 데이터의 일관성을 유지하기 위한 추가적인 데이터 조작 등이 대표적으로 낭비되는 비용이라고 볼 수 있다.

다. 업무규칙(Business Rules)

데이터 모델에서 매우 중요한 요소 중 하나가 데이터 모델링 과정에서 도출되고 규명되는 수많은 업무규칙(Business Rules)을 데이터 모델에 표현하고 이를 해당 데이터 모델을 활용하는 모든 사용자가 공유할 수 있도록 제공하는 것이다. 특히, 데이터 아키텍처에서 언급되는 논리 데이터 모델(Logical Data Model)에서 이러한 요소들이 포함되어야 함은 매우 중요하다. 예를 들면, 보험사의 사원들은 매월 여러 가지 항목에 대해서 급여를 지급받고 있고 이를 데이터로 관리하고 있다. 각 사원들은 월별로 하나 이상의 급여 항목(기본급, 상여금, 수당, 수수료, 등등)에 대해서 급여를 지급받는다. 여기에 더 나아가 각 사원은 사원 구분별(내근, 설계사, 계약직, 대리점 등)로 위의 급여 항목을 차등적으로 지급받는 다는 업무규칙이 있다. 이러한 내용을 데이터 모델에 나타내야 한다. 이렇게 함으로써 해당 데이터 모델을 사용하는 모든 사용자(개발자, 관리자 등)가 해당 규칙에 대해서 동일한 판단을 하고 데이터를 조작할 수 있게 된다.

라. 데이터 재사용(Data Reusability)

데이터의 재사용성을 향상시키고자 한다면 데이터의 통합성과 독립성에 대해서 충분히 고려해야 한다. 과거에 정보시스템이 생성·운영된 형태를 되짚어 보면 철저하게 부서 단위

의 정보시스템으로 설계되고 운용되어 왔다. 현재 대부분의 회사에서 진행하고 있는 신규 정보시스템의 구축 작업은 회사 전체 관점에서 공통 데이터를 도출하고 이를 전 영역에서 사용하기에 적절한 형태로 설계하여 시스템을 구축하게 된다. 이러한 형태의 데이터 설계에서 가장 중요하게 대두되는 것이 통합 모델이다. 통합 모델이어야만 데이터 재사용성을 향상시킬 수 있다. 또 한 측면에서 보면 과거 정보시스템의 데이터 구조의 가장 큰 특징은 데이터 모델이 별도로 존재하지 않고 애플리케이션의 부속품 정도로 인식되어져 왔던 것이 사실이다. 이러한 환경에서의 데이터는 프로세스의 흐름에 따라서 관리되게 마련이다. 이렇게 되면 데이터 중복이 많이 발생하고 데이터의 일관성 문제가 심각하게 초래된다. 데이터가 애플리케이션에 대해 독립적으로 설계되어야만 데이터 재사용성을 향상시킬 수 있다.

정보시스템은 비즈니스의 변화에 대해서 최적으로 적응하도록 끊임없이 요구된다. 하지만 일부 정보시스템의 데이터 모델은 이러한 변화에 대해서 현재의 데이터 구조를 거의 변화하지 않고도 변화에 대응할 수 있는 데이터 구조도 있을 것이고, 아주 적은 확장을 통해서 이러한 변화에 대응하는 것도 있을 것이다. 하지만, 이러한 변화에 대응하기 위해서 데이터 구조적으로 아주 많은 변화를 주어야만 한다면 변화의 대상이 되는 부분뿐만 아니라 정보시스템의 나머지 부분들도 많은 영향을 받게 될 것이다. 그래서 많은 기업들이 정보시스템을 구축하는 과정에서 데이터 구조의 확장성, 유연성에 많은 노력을 기울이고 있다.

결국 현대의 기업들이 동종의 타 기업으로부터 경쟁 우위에 자리매김하려고 하다면 구축하는 데이터 모델은 이러한 외부의 업무 환경 변화에 대해서 유연하게 대응할 수 있어야 한다. 특히 근래의 많은 패키지 시스템들이 가지고 있는 데이터 모델들은 확장성을 강조하기 위해서 많은 부분을 통합한 데이터 모델의 형태를 가지고 있다. 여기에서도 잘 나타나듯이 확장성을 담보하기 위해서는 데이터 관점의 통합이 불가피하다. 특히 정보시스템에서의 '행위의 주체'가 되는 집합의 통합, '행위의 대상'이 되는 집합의 통합, '행위 자체'에 대한 통합 등은 전체 정보시스템의 안정성, 확장성을 좌우하는 가장 중요한 요소이다.

데이터 모델이 갖추어야 하는 중요한 요소 중에 하나는 기업이 관리하고자 하는 데이터를 합리적으로 균형이 있으면서도 단순하게 분류하는 것이다. 아무리 효율적으로 데이터를 잘 관리할 수 있다고 하더라도 그것의 사용, 관리 측면이 복잡하다면 잘 만들어진 데이터 모델이라고 할 수 없다. 동종의 비즈니스를 영위하는 기업이라 하더라도 각 회사의 데이터 모델을 비교해 보면 그 복잡도에는 많은 차이를 나타낸다.

A보험사는 계약 업무를 수행하기 위해서 10개의 테이블을 정의하여 업무를 수행하는 반면에 B회사는 100개의 테이블을 정의하여 동일한 업무를 수행하고 있다. 두 회사의 데이터 모델의 차이점은 다음과 같다. 10개의 테이블을 가지고 업무를 수행하고 있는 A회사의 데이터 모델은 간결하지만 새로는 업무 환경의 변화에 대해서 확장성을 가지고 있다. B회사는 겉으로는 새로운 업무 환경의 변화터 모델의 한계로 인해 테이블의 수가 지속적으로 증가해 왔다. 이렇게 됨으로써 데이터 모델은 간결하지 못하고 동일한 형태로 관리되어야 하는 데이터가 복잡한 형태로 관리되고 그들과의 관계를 가지고 있는 다른 여러 가지의 데이터들 또한 복잡한 형태의 관계들이 불가피해 복잡성을 증가시켜 왔다. 결국 간결한 모델의 기본적인 전제는 통합이다. 합리적으로 잘 정돈된 방법으로 데이터를 통합하여 데이터의 집합을 정의하고 이를 데이터 모델로 잘 표현하여 활용한다면 웬만한 업무 변화에도 데이터 모델이 영향을 받지 않고 운용할 수 있게 된다.

마. 의사소통(Communication)

데이터 모델의 역할은 많다. 그 중에서도 중요한 것이 데이터 모델의 의사소통의 역할이다. 데이터 모델은 대상으로 하는 업무를 데이터 관점에서 분석하고 이를 설계하여 나오는 최종 산출물이다. 데이터를 분석 과정에서는 자연스럽게 많은 업무 규칙들이 도출된다. 이 과정에서 도출되는 많은 업무 규칙들은 데이터 모델에 엔터티, 서브타입, 속성, 관계 등의 형태로 최대한 자세하게 표현되어야 한다.

예를 들면, '사원' 테이블에는 어떠한 '사원구분'을 가지는 사원들이 존재하는지, '정규직', '임시직' 사원들이 같이 존재하는지, 아니면 또 다른 형태의 사원들이 존재하는지를 표현해야 한다. 더 나아가서 '호봉'이라는 속성은 '정규직'일 때에만 존재하는 속성인데 이러한 업무 규칙이 데이터 모델에 표현되어야 한다. 또한, 우리가 관리하는 사원들 중에서 '정규직' 사원들만이 '급여' 테이블과 관계를 가진다. 이러한 부분은 개별 관계로 데이터 모델에 표현되어야 한다.

이렇게 표현된 많은 업무 규칙들을 해당 정보시스템을 운용, 관리하는 많은 관련자들이 설계자가 정의한 업무 규칙들을 동일한 의미로 받아들이고 정보시스템을 활용할 수 있게 하는 역할을 하게 된다. 즉, 데이터 모델이 진정한 의사소통(Communication)의 도구로서의 역할을 하게 된다.

바. 통합성(Integration)

기업들이 과거부터 정보시스템을 구축해 왔던 방법은 개별 업무별로의 단위 정보시스템을 구축하여 현재까지 유지보수를 해오고 있는 것이 보통이다. 점진적인 확장과 보완의 방법으로 정보시스템을 구축해 왔기 때문에 동일한 성격의 데이터임에도 불구하고 전체 조직관점에서 보면 여러 곳에서 동일한 데이터가 존재하기 마련이다. 특히 이러한 데이터 중에서도 고객, 상품 등과 같이 마스터 성격의 데이터들이 분할되어 관리됨으로 인해 전체 조직 관점에서 데이터 품질, 관리, 활용 관점에서 많은 문제점들이 나타나고 있는 것이 현실이다. 가장 바람직한 데이터 구조의 형태는 동일한 데이터는 조직의 전체에서 한번 만 정의되고 이를 여러 다른 영역에서 참조, 활용하는 것이다. 물론 이 때에 성능 등의 부가적인 목적으로 의도적으로 데이터를 중복시키는 경우는 존재할 수 있다. 동일한 성격의 데이터를 한 번만 정의하기 위해서는 공유 데이터에 대한 구조를 여러 업무 영역에서 공동으로 사용하기 용이하게 정의할 수 있어야 한다. 이러한 이유로 데이터 아키텍처의 중요성이 한층 더 부각되고 있다.

제2절 엔터티(Entity)

1. 엔터티의 개념

데이터 모델을 이해할 때 가장 명확하게 이해해야 하는 개념 중에 하나가 바로 엔터티(Entity)이다. 이것은 우리말로 실체, 객체라고 번역하기도 하는데 실무적으로 엔터티라는 외래어를 많이 사용하기 때문에 본 가이드에서는 엔터티라는 용어를 그대로 사용하기로 한다.

엔터티에 대해서 데이터 모델과 데이터베이스에 권위자가 정의한 사항은 다음과 같다.

- 변별할 수 있는 사물 - Peter Chen (1976) -
- 데이터베이스 내에서 변별 가능한 객체 - C.J Date (1986) -
- 정보를 저장할 수 있는 어떤 것 - James Martin (1989) -
- 정보가 저장될 수 있는 사람, 장소, 물건, 사건 그리고 개념 등 - Thomas Bruce (1992) -
-

위 정의들의 공통점은 다음과 같다.

- 엔터티는 사람, 장소, 물건, 사건, 개념 등의 명사에 해당한다.
- 엔터티는 업무상 관리가 필요한 관심사에 해당한다.
- 엔터티는 저장이 되기 위한 어떤 것(Thing)이다.

엔터티란 “업무에 필요하고 유용한 정보를 저장하고 관리하기 위한 집합적인 것(Thing)”으로 설명할 수 있다. 또는, 엔터티는 업무 활동상 지속적인 관심을 가지고 있어야 하는 대상으로서 그 대상들 간에 동질성을 지닌 인스턴스들이나 그들이 행하는 행위의 집합으로 정의할 수 있다. 엔터티는 그 집합에 속하는 개체들의 특성을 설명할 수 있는 속성(Attribute)을 갖는데, 예를 들어 ‘학생’이라는 엔터티는 학번, 이름, 이수학점, 등록일자, 생일, 주소, 전화번호, 전공 등의 속성으로 특징지어질 수 있다. 이러한 속성 가운데에는 엔터티 인스턴스 전체가 공유할 수 있는 공통 속성도 있고, 엔터티 인스턴스 중 일부에만 해당하는 개별 속성도 있을 수 있다.

또한 엔터티는 인스턴스의 집합이라고 말할 수 있고, 반대로 인스턴스라는 것은 엔터티의 하나의 값에 해당한다고 정의할 수 있다. 예를 들어 과목은 수학, 영어, 국어가 존재할 수 있는데 수학, 영어, 국어는 각각이 과목이라는 엔터티의 인스턴스들이라고 할 수 있다. 또한 사건이라는 엔터티에는 사건번호2010-001, 2010-002 등의 사건이 인스턴스가 될 수 있다. 엔터티를 이해할 때 눈에 보이는(Tangible)한 것만 엔터티로 생각해서는 안되며 눈에 보이지 않는 개념 등에 대해서도 엔터티로서 인식을 할 수 있어야 한다. 실제 업무상에는 눈에 보이지 않는 것(Thing)이 엔터티로 도출되는 경우가 많기 때문에 더더욱 주의할 필요가 있다.

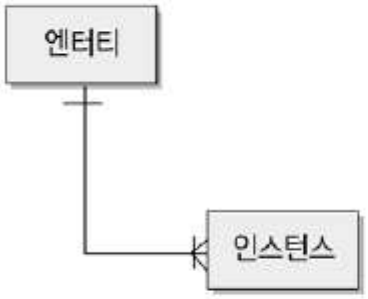
업무에 필요하고 유용한 정보를 저장하고 관리하기 위한 집합적인 것(thing)

[그림 Ⅰ-1-14] 엔터티 종류

2. 엔터티와 인스턴스에 대한 내용과 표기법

엔터티를 표현하는 방법은 각각의 표기법에 따라 조금씩 차이는 있지만 대부분 사각형으로 표현된다. 다만 이 안에 표현되는 속성의 표현방법이 조금씩 다를 뿐이다. 엔터티와 엔터티간의 ERD를 그리면 [그림 Ⅰ-1-15]와 같이 표현할 수 있다.

엔터티-인스턴스 ERD



엔터티는 인스턴스의 집합

엔터티-인스턴스의 예

엔터티	인스턴스
과 목	수 학
	영 어
강 사	이춘식
	조시형
사 건	2010-001
	2010-002

[그림 Ⅰ-1-15] 엔터티와 인스턴스

[그림 Ⅰ-1-15]에서 과목, 강사, 사건은 엔터티에 해당하고 수학, 영어는 과목이라는 엔터티의 인스턴스이고 이춘식, 조시형은 강사라는 엔터티의 인스턴스이며 사건번호인 2010-001, 2010-002는 사건 엔터티에 대한 인스턴스에 해당한다.

※ 참고 : 오브젝트 모델링에는 클래스(Class)와 오브젝트(Object)라는 개념이 있다. 클래스는 여러 개의 오브젝트를 포함하는 오브젝트 깡통이다. 이러한 개념은 정보공학의 엔터티가 인스턴스를 포함하는 개념과 비슷하다.

위의 엔터티와 인스턴스를 표현하면 [그림 Ⅰ-1-16]과 같다.

[그림 Ⅰ-1-16] 엔터티에 대한 표기법

3. 엔터티의 특징

엔터티는 다음과 같은 특징을 가지고 있으며 만약 도출된 엔터티가 다음의 성질을 만족하지 못하면 적절하지 않은 엔터티일 확률이 높다.

- 반드시 해당 업무에서 필요하고 관리하고자 하는 정보이어야 한다.(예. 환자, 토익의 응시횟수, …)
- 유일한 식별자에 의해 식별이 가능해야 한다.
- 영속적으로 존재하는 인스턴스의 집합이어야 한다.('한 개'가 아니라 '두 개 이상')
- 엔터티는 업무 프로세스에 의해 이용되어야 한다.
- 엔터티는 반드시 속성이 있어야 한다.
- 엔터티는 다른 엔터티와 최소 한 개 이상의 관계가 있어야 한다.

가. 업무에서 필요로 하는 정보

엔터티 특징의 첫 번째는 반드시 시스템을 구축하고자 하는 업무에서 필요로 하고 관리하고자 하는 정보여야 한다는 점이다. 예를 들어 환자라는 엔터티는 의료시스템을 개발하는 병원에서는 반드시 필요한 엔터티이지만 일반회사에서 직원들이 병에 걸려 업무에 지장을 준다하더라도 이 정보를 그 회사의 정보로서 활용하지는 않을 것이다. 즉 시스템 구축 대상인 해당업무에서 그 엔터티를 필요로 하는가를 판단하는 것이 중요하다.

업무에서 관리하고자 하는 영역(Business Boundary)에 대한 인식이 매우 중요함

[그림 Ⅰ-1-17] 엔터티 특성-필요성

사람이 살아가면서 환자는 발생할 수 밖에 없다. 그러나 일반회사의 인사시스템에서는 비록 직원들에 의해서 환자가 발생이 되지만 인사업무 영역에서 환자를 별도로 관리할 필요가 없다. 다른 예로 병원에서는 환자가 해당 업무의 가장 중요한 엔터티가 되어 꼭 관리해야 할 엔터티가 된다. 이와 같이 엔터티를 도출할 때는 업무영역 내에서 관리할 필요가 있는지를 먼저 판단하는 것이 중요하다.

나. 식별이 가능해야 함

두 번째는 식별자(Unique Identifier)에 의해 식별이 가능해야 한다는 점이다. 어떤 엔터티이건 임의의 식별자(일련번호)를 부여하여 유일하게 만들 수는 있지만, 엔터티를 도출하는 경우에 각각의 업무적으로 의미를 가지는 인스턴스가 식별자에 의해 한 개씩만 존재하는지 검증해 보아야 한다.

유일한 식별자는 그 엔터티의 인스턴스만의 고유한 이름이다. 두 개 이상의 엔터티를 대변하면 그 식별자는 잘못 설계된 것이다. 예를 들어 직원을 구분할 수 있는 방법은 이름이나 사원번호가 될 수가 있다. 그러나 이름은 동명이인(同名異人)이 될 수 있으므로 유일하게 식별될 수 없다. 사원번호는 회사에 입사한 사람에게 고유하게 부여된 번호이므로 유일한 식별자가 될 수 있는 것이다.

모두다 동일한 이름, 속성 관계

인스턴스 각각을 구분하기 위한 유일한 식별자가 존재해야 함

[그림 Ⅰ-1-18] 엔터티 특성-유일성

다. 인스턴스의 집합

세 번째는 영속적으로 존재하는 인스턴스의 집합이 되어야 한다는 점이다. 엔터티의 특징 중 “한 개”가 아니라 “두 개 이상”이라는 집합개념은 매우 중요한 개념이다. 두 개 이상

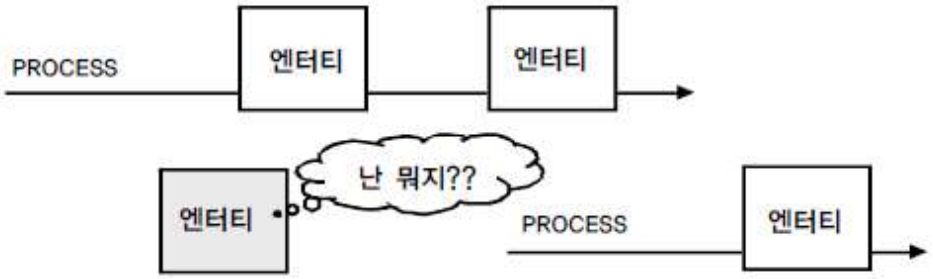
이라는 개념은 엔터티뿐만 아니라 엔터티간의 관계, 프로세스와의 관계 등 업무를 분석하고 설계하는 동안 설계자가 모든 업무에 대입해보고 검증해 보아야 할 중요한 개념이다. 하나의 엔터티는 여러 개의 인스턴스를 포함한다.

인스턴스가 한 개 밖에 없는 회사, 병원 엔터티는 집합이 아니므로 엔터티 성립이 안됨

[그림 Ⅰ-1-19] 엔터티 특성-인스턴스 수

라. 업무프로세스에 의해 이용

네 번째는 업무프로세스(Business Process)가 그 엔터티를 반드시 이용해야 한다는 점이다. 첫 번째 정의에서처럼 업무에서 반드시 필요하다고 생각하여 엔터티로 선정하였는데 업무프로세스에 의해 전혀 이용되지 않는다면 업무 분석이 정확하게 안되어 엔터티가 잘못 선정되거나 업무프로세스 도출이 적절하게 이루어지지 않았음을 의미한다. 이러한 경우는 데이터 모델링을 할 때 미처 발견하지 못하다가 프로세스 모델링을 하면서 데이터 모델과 검증을 하거나, 상관 모델링을 할 때 엔터티와 단위프로세스를 교차 점검하면서 문제점이 도출된다.



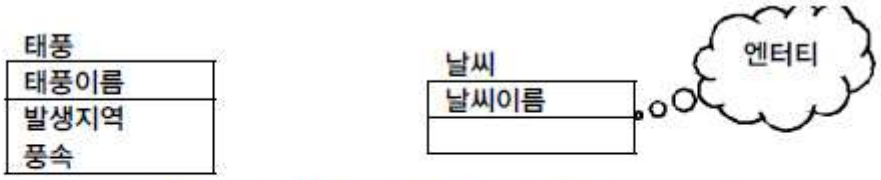
업무 프로세스에 의해 이용되지 않는 엔터티는 그 업무의 엔터티가 아님

[그림 Ⅰ-1-20] 엔터티 특성-프로세스 존재

[그림 Ⅰ-1-20]과 같이 업무프로세스에 의해 CREATE, READ, UPDATE, DELETE 등이 발생하지 않는 고립된 엔터티의 경우는 엔터티를 제거하거나 아니면 누락된 프로세스가 존재하는지 살펴보고 해당 프로세스를 추가해야 한다.

마. 속성을 포함

다섯 번째는 엔터티에는 반드시 속성(Attributes)이 포함되어야 한다는 점이다. 속성을 포함하지 않고 엔터티의 이름만 가지고 있는 경우는 관계가 생략되어 있거나 업무 분석이 미진하여 속성정보가 누락되는 경우에 해당한다. 또한 주식별자만 존재하고 일반속성은 전혀 없는 경우도 마찬가지로 적절한 엔터티라고 할 수 없다. 단, 예외적으로 관계엔터티(Associative Entity)의 경우는 주식별자 속성만 가지고 있어도 엔터티로 인정한다.

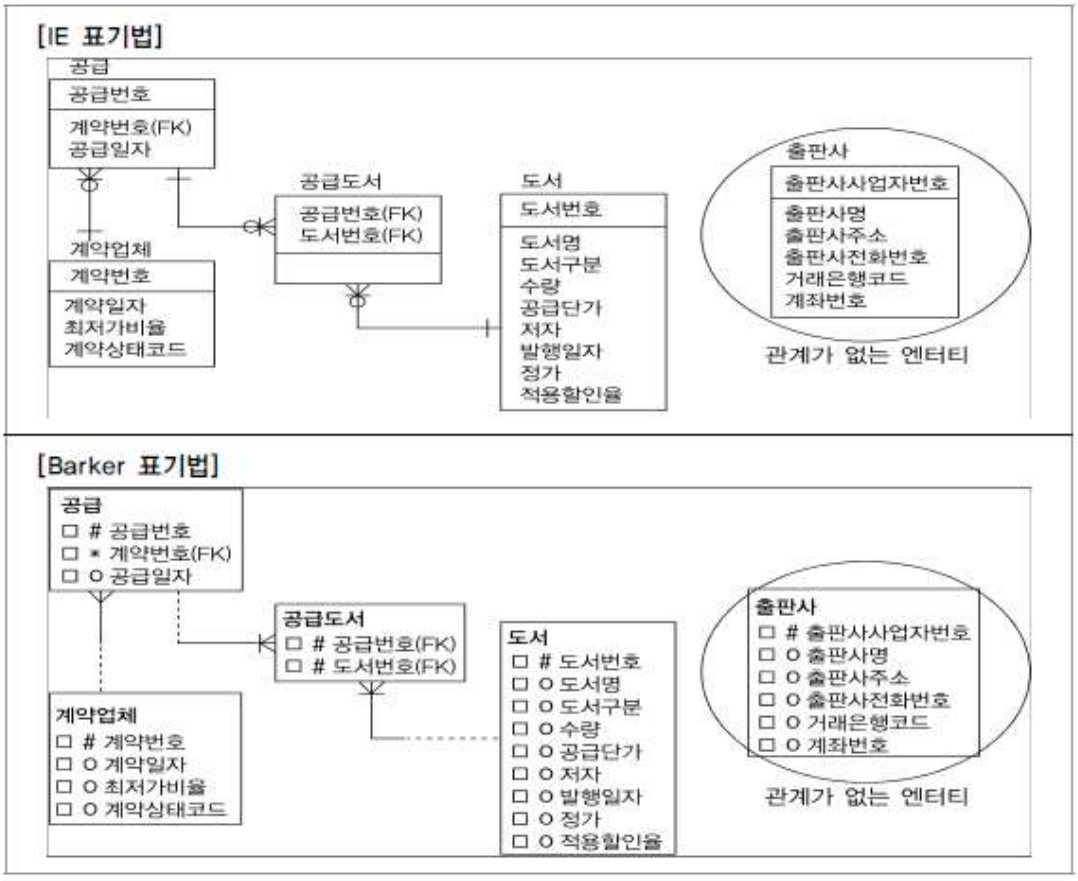


속성이 존재하지 않는 오브젝트는 엔터티가 될 수 없음

[그림 Ⅰ-1-21] 엔터티 특성-속성의 존재

바. 관계의 존재

여섯 번째는 엔터티는 다른 엔터티와 최소 한 개 이상의 관계가 존재해야 한다는 것이다. 기본적으로 엔터티가 도출되었다는 것은 해당 업무내에서 업무적인 연관성(존재적 연관성, 행위적 연관성)을 가지고 다른 엔터티와의 연관의 의미를 가지고 있음을 나타낸다. 그러나 관계가 설정되지 않은 엔터티의 도출은 부적절한 엔터티가 도출되었거나 아니면 다른 엔터티와 적절한 관계를 찾지 못했을 가능성이 크다.



엔터티가 관계가 없으면, 잘못된 엔터티이거나 관계가 누락되었을 가능성이 큼

[그림 Ⅰ-1-22] 엔터티 특성-관계의 존재

단, 데이터 모델링을 하면서 관계를 생략하여 표현해야 하는 경우는 다음과 같은 통계성 엔터티 도출, 코드성 엔터티 도출, 시스템 처리시 내부 필요에 의한 엔터티 도출과 같은 경우이다.

1) 통계를 위한 엔터티의 경우는 업무진행 엔터티로부터 통계업무만(Read Only)을 위해 별도로 엔터티를 다시 정의하게 되므로 엔터티간의 관계가 생략되는 경우에 해당한다.
2) 코드를 위한 엔터티의 경우 너무 많은 엔터티와 엔터티간의 관계 설정으로 인해 데이터 모델의 읽기효율성(Readability)이 저하되어 도저히 모델링 작업을 진행할 수 없게 된다. 또한 코드성 엔터티는 물리적으로 테이블과 프로그램 구현 이후에도 외부키에 의한 참조무결성을 체크하기 위한 규칙을 데이터베이스 기능에 맡기지 않는 경우가 대부분이기 때문에 논리적으로나 물리적으로 관계를 설정할 이유가 없다.
3) 시스템 처리시 내부 필요에 의한 엔터티(예를 들어, 트랜잭션 로그 테이블 등)의 경우 트랜잭션이 업무적으로 연관된 테이블과 관계 설정이 필요하지만 이 역시 업무적인 필요가 아니고 시스템 내부적인 필요에 의해 생성된 엔터티이므로 관계를 생략하게 된다.

4. 엔터티의 분류

엔터티는 엔터티 자신의 성격에 의해 실체유형에 따라 구분하거나 업무를 구성하는 모습에 따라 구분이 되는 발생시점에 의해 분류해 볼 수 있다.

가. 유무(有無)형에 따른 분류

일반적으로 엔터티는 유무형에 따라 유형엔터티, 개념엔터티, 사건엔터티로 구분된다. 유형엔터티(Tangible Entity)는 물리적인 형태가 있고 안정적이며 지속적으로 활용되는 엔터티로 업무로부터 엔터티를 구분하기가 가장 용이하다. 예를 들면, 사원, 물품, 강사 등이 이에 해당된다. 개념엔터티(Conceptual Entity)는 물리적인 형태는 존재하지 않고 관리해야 할 개념적 정보로 구분이 되는 엔터티로 조직, 보험상품 등이 이에 해당된다.

사건 엔터티(Event Entity)는 업무를 수행함에 따라 발생되는 엔터티로서 비교적 발생량이 많으며 각종 통계자료에 이용될 수 있다. 주문, 청구, 미납 등이 이에 해당된다.

나. 발생시점(發生時點)에 따른 분류

엔터티의 발생시점에 따라 기본/키엔터티(Fundamental Entity, Key Entity), 중심엔터티(Main Entity), 행위엔터티(Active Entity)로 구분할 수 있다.

1) 기본엔터티

기본엔터티란 그 업무에 원래 존재하는 정보로서 다른 엔터티와 관계에 의해 생성되지 않고 독립적으로 생성이 가능하고 자신은 타 엔터티의 부모의 역할을 하게 된다. 다른 엔터티로부터 주식별자를 상속받지 않고 자신의 고유한 주식별자를 가지게 된다. 예를 들어 사원, 부서, 고객, 상품, 자재 등이 기본엔터티가 될 수 있다.

2) 중심엔터티

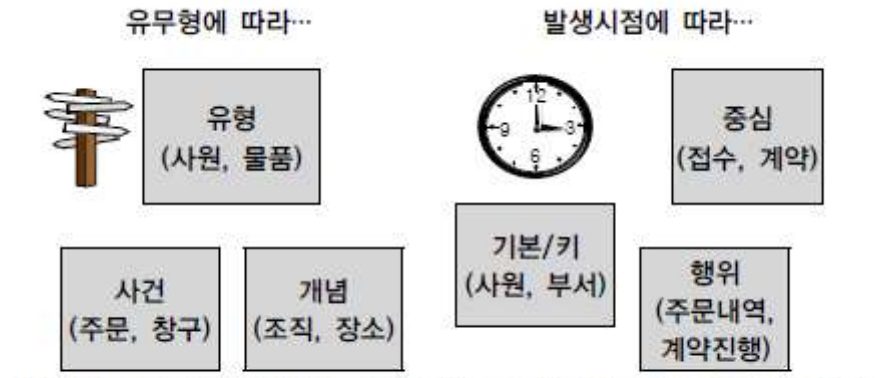
중심엔터티란 기본엔터티로부터 발생되고 그 업무에 있어서 중심적인 역할을 한다. 데이터의 양이 많이 발생되고 다른 엔터티와의 관계를 통해 많은 행위엔터티를 생성한다. 예를 들어 계약, 사고, 예금원장, 청구, 주문, 매출 등이 될 수 있다.

3) 행위엔터티

행위엔터티는 두 개 이상의 부모엔터티로부터 발생되고 자주 내용이 바뀌거나 데이터량이 증가된다. 분석초기 단계에서는 잘 나타나지 않으며 상세 설계단계나 프로세스와 상관모델링을 진행하면서 도출될 수 있다. 예를 들어 주문목록, 사원변경이력 등이 포함된다.

다. 엔터티 분류 방법의 예

[그림 Ⅰ-1-23]은 두 가지 엔터티 분류 방법에 대한 예를 나타낸 것이다.



[그림 Ⅰ-1-23] 엔터티 분류

이 밖에도 엔터티가 스스로 생성될 수 있는지 여부에 따라 독립엔터티인지 의존엔터티인지를 구분할 수도 있다.

5. 엔터티의 명명

엔터티를 명명하는 일반적인 기준은 용어를 사용하는 모든 표기법이 다 그렇듯이 첫 번째는 가능하면 현업업무에서 사용하는 용어를 사용한다. 두 번째는 가능하면 약어를 사용하지 않는다. 세 번째는 단수명사를 사용한다. 네 번째는 모든 엔터티에서 유일하게 이름이 부여되어야 한다. 다섯 번째는 엔터티 생성의미대로 이름을 부여한다.

첫 번째에서 네 번째에 해당하는 원칙은 대체적으로 잘 지켜진다. 그러나 다섯 번째 원칙인 “엔터티 생성의미대로 이름을 부여한다.”에 대해서는 적절하지 못한 엔터티명이 부여되는 경우가 빈번하게 발생한다. 중심엔터티에서도 간혹 적절하지 못한 엔터티명을 사용한 경우가 발생되고 행위엔터티의 경우에는 꽤 많은 경우에 적절하지 못한 엔터티명을 사용하

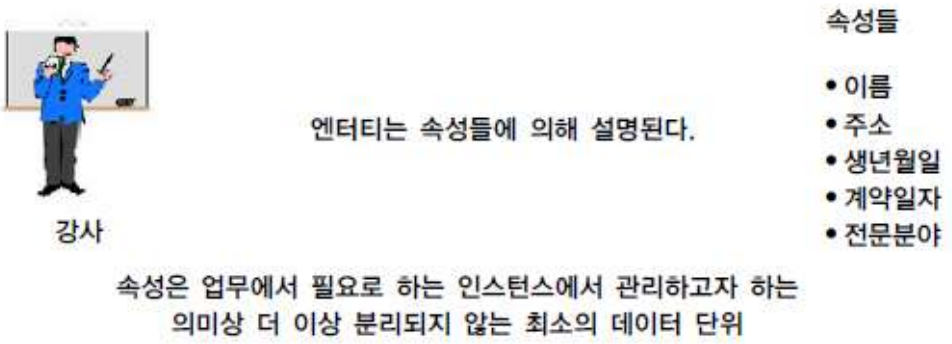
는 경우가 발생된다. 예를 들어, 고객이 어떤 제품들에 대해 주문하여 발생하는 행위엔터티에 대해 주문목록이라고도 할 수 있고 고객제품이라고 할 수 있다. 만약 고객제품이라고 하면 '고객이 주문한 제품'인지 아니면 '고객의 제품'인지 의미가 애매모호해질 수 있게 된다. 엔터티의 이름을 업무목적에 따라 생성되는 자연스러운 이름을 부여해야 하는데 이와 상관없이 임의로 이름을 부여하게 되면 프로젝트에서는 커뮤니케이션 오류로 인해 문제를 야기할 수 있게 된다.

제3절 속성(Attribute)

1. 속성 (Attribute)의 개념

속성이란 사전적인 의미로는 사물(事物)의 성질, 특징 또는 본질적인 성질, 그것이 없다면 실체를 생각할 수 없는 것으로 정의할 수 있다. 본질적 속성이란 어떤 사물 또는 개념에 없어서는 안 될 징표(徵表)의 전부이다. 이 징표는 사물이나 개념이 어떤 것인지를 나타내고 그것을 다른 것과 구별하는 성질이라고 할 수 있다.

이런 사전적인 정의 이외에 데이터 모델링 관점에서 속성을 정의하자면, "업무에서 필요로 하는 인스턴스로 관리하고자 하는 의미상 더 이상 분리되지 않는 최소의 데이터 단위"로 정의할 수 있다. 업무상 관리하기 위한 최소의 의미 단위로 생각할 수 있고, 이것은 엔터티에서 한 분야를 담당하고 있다.



[그림 Ⅰ-1-24] 속성의 정의

속성의 정의를 정리해 보면 다음과 같다.

- 업무에서 필요로 한다.
- 의미상 더 이상 분리되지 않는다.
- 엔터티를 설명하고 인스턴스의 구성요소가 된다.

의미상 더 이상 분리되지 않는다는 특징을 살펴보면 다음과 같다. 예를 들어 생년월일은 하나로서 의미가 있다. 만약 이것을 생년, 생월, 생일로 구분한다면 이것은 사실상 하나의 속성을 관리목적에 따라 구분했다라고 이해할 수 있다. 이러한 이유로 인해 S/W비용을 산정하는 기능점수(Function Point)를 산정할 때 이렇게 분리된 속성은 하나의 속성(DET)으로 계산하게 된다. 그러나 만약 서로 관련이 없는 이름, 주소를 하나의 속성 '이름주소'로 정의하면 어떻게 될까? 이것은 하나의 속성의 두 개의 의미를 갖기 때문에 기본속성으로서 성립하지 않게 된다. 이렇게 정리된 속성은 그냥 값의 의미로서 속성보다는 내역(Description)으로서 속성으로 예를 들어 인적사항이라는 속성으로 정의하여 관리할 수는 있다.

2. 엔터티, 인스턴스와 속성, 속성값에 대한 내용과 표기법

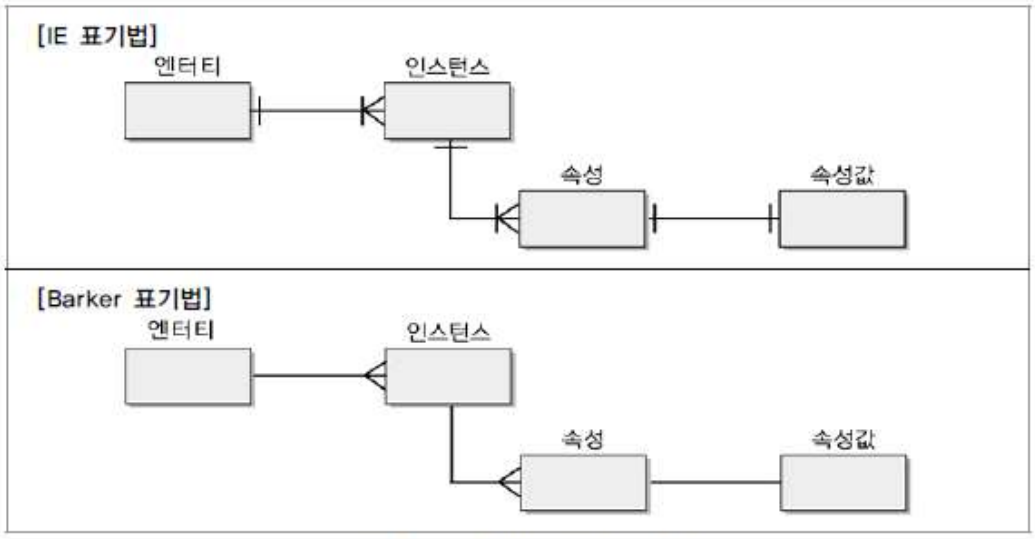
가. 엔터티, 인스턴스, 속성, 속성값의 관계

엔터티에는 두 개 이상의 인스턴스가 존재하고 각각의 엔터티에는 고유의 성격을 표현하는 속성정보를 두 개 이상 갖는다. 업무에서는 엔터티를 구성하는 특징이 무엇인지 또한 각각의 인스턴스들은 어떤 성격의 데이터로 구성되는지를 파악하는 작업이 필요하다. 분석단계에서 엔터티 내에 존재하는 여러 개의 인스턴스가 가지는 동일한 성격은 무엇인지를 파악하고 이에 이름을 부여하여 엔터티의 속성으로 기술하는 작업이 필요하다. 예를 들면 사원은 이름, 주소, 전화번호, 직책 등을 가질 수 있다. 사원이라는 엔터티에 속한 인스턴스들의 성격을 구체적으로 나타내는 항목이 바로 속성이다. 각각의 인스턴스는 속성의 집합으로 설명될 수 있다. 하나의 속성은 하나의 인스턴스에만 존재할 수 있다. 속성은 관계로 기술될 수 없고 자신이 속성을 가질 수도 없다.

엔터티 내에 있는 하나의 인스턴스는 각각의 속성들의 대해 한 개의 속성값만을 가질 수 있다. 예를 들면 사원의 이름은 홍길동이고 주소는 서울시 강남구이며, 전화번호는 123-4567, 직책은 대리이다. 이름, 주소, 전화번호, 직책은 속성이고 홍길동, 서울시 강남구, 123-4567, 대리는 속성값이다. 그러므로 속성값은 각각의 엔터티가 가지는 속성들의 구체적인 내용이라 할 수 있다.

엔터티, 인스턴스, 속성, 속성값에 대한 관계를 분석하면 다음과 같은 결과를 얻을 수 있다.

- 한 개의 엔터티는 두 개 이상의 인스턴스의 집합이어야 한다.
- 한 개의 엔터티는 두 개 이상의 속성을 갖는다.
- 한 개의 속성은 한 개의 속성값을 갖는다.



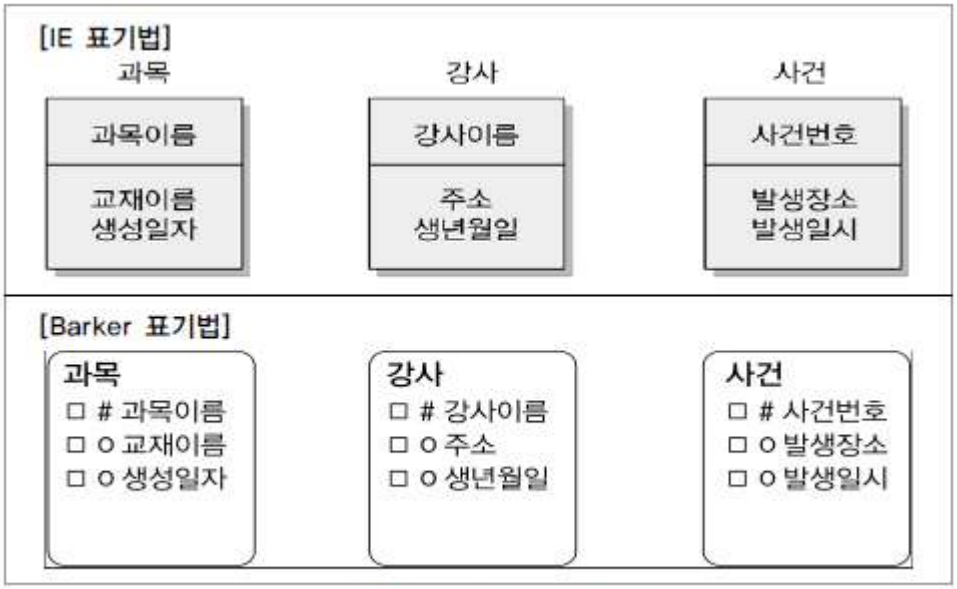
[그림 Ⅰ-1-25] 엔터티-속성의 관계

속성은 엔터티에 속한 엔터티에 대한 자세하고 구체적인 정보를 나타내며 각각의 속성은 구체적인 값을 갖게 된다.

예를 들어 사원이라는 엔터티에는 홍길동이라는 사람(엔터티)이 있을 수 있다. 홍길동이라는 사람의 이름은 홍길동이고 주소는 서울시 강서구이며 생년월일 1967년 12월 31일이다. 여기에 이름, 주소, 생년월일과 같은 각각의 값을 대표하는 이름들을 속성이라 하고 홍길동, 서울시 강서구, 1967년 12월 31일과 같이 각각의 이름에 대한 구체적인 값을 속성값(VALUE)이라고 한다.

나. 속성의 표기법

속성의 표기법은 엔터티 내에 이름을 포함하여 표현하면 된다.



[그림 Ⅰ-1-26] 속성의 표기법

3. 속성의 특징

속성은 다음과 같은 특징을 가지고 있으며 만약 도출된 속성이 다음의 성질을 만족하지 못하면 적절하지 않은 속성일 확률이 높다.

- 엔터티와 마찬가지로 반드시 해당 업무에서 필요하고 관리하고자 하는 정보이어야 한다. (예, 강사의 교재이름)
- 정규화 이론에 근간하여 정해진 주식별자에 함수적 종속성을 가져야 한다.
- 하나의 속성에는 한 개의 값만을 가진다. 하나의 속성에 여러 개의 값이 있는 다중값일 경우 별도의 엔터티를 이용하여 분리한다.

4. 속성의 분류

가. 속성의 특성에 따른 분류

속성은 업무분석을 통해 바로 정의한 속성을 기본속성(Basic Attribute), 원래 업무상 존재하지는 않지만 설계를 하면서 도출해내는 속성을 설계속성(Designed Attribute), 다른 속성으로부터 계산이나 변형이 되어 생성되는 속성을 파생속성(Derived Attribute)이라고 한다.

1) 기본속성

기본 속성은 업무로부터 추출한 모든 속성이 여기에 해당하며 엔터티에 가장 일반적이고 많은 속성을 차지한다. 코드성 데이터, 엔터티를 식별하기 위해 부여된 일련번호, 그리고 다른 속성을 계산하거나 영향을 받아 생성된 속성을 제외한 모든 속성은 기본속성이다. 주의해야 할 것은 업무로부터 분석한 속성이라도 이미 업무상 코드로 정의한 속성이 많다는 것이다. 이러한 경우도 속성의 값이 원래 속성을 나타내지 못하므로 기본속성이 되지 않는다.

2) 설계속성

설계속성은 업무상 필요한 데이터 이외에 데이터 모델링을 위해, 업무를 규칙화하기 위해 속성을 새로 만들거나 변형하여 정의하는 속성이다. 대개 코드성 속성은 원래 속성을 업무상 필요에 의해 변형하여 만든 설계속성이고 일련번호와 같은 속성은 단일(Unique)한 식별자를 부여하기 위해 모델 상에서 새로 정의하는 설계속성이다.

3) 파생속성

파생속성은 다른 속성에 영향을 받아 발생하는 속성으로서 보통 계산된 값들이 이에 해당된다. 다른 속성에 영향을 받기 때문에 프로세스 설계 시 데이터 정합성을 유지하기 위해 유의해야 할 점이 많으며 가급적 파생속성을 적게 정의하는 것이 좋다.

[그림 Ⅰ-1-27] 속성의 분류

이러한 분류 방식은 프로젝트에서 엄격하게 분류하여 속성정의서에 나열하는 경우도 있다. 이 때 파생속성의 경우는 그 값이 계산된 로직을 속성의 정의서의 기록함으로써 향후 속성 값에 대한 검증 시 활용되기도 한다. 파생속성은 그 속성이 가지고 있는 계산방법에

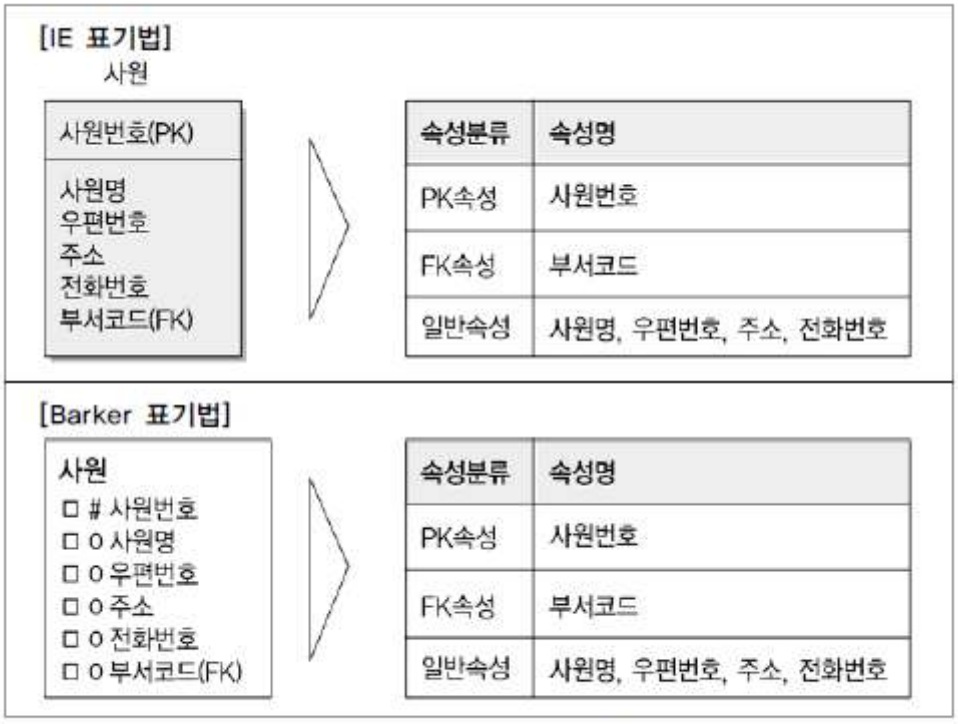
대해서 반드시 어떤 엔터티에 어떤 속성에 의해 영향을 받는지 정의가 되어야 한다.

예를 들어 '이자'라는 속성이 존재한다고 하면 이자는 원금이 1,000원이고 예치기간이 5개월이며 이자율이 5.0%에서 계산되는 속성값이다. 그렇다면 이자는 원금이 1,000원에서 2,000원으로 변하여도 영향을 받고 예치기간이 5개월에서 7개월로 증가하여도 값이 변하며 이자율이 5.0%에서 6.0%로 되어도 이자속성이 가지는 값은 변할 것이다. 한 번 값이 변해도 또 다시 영향을 미치는 속성값의 조건이 변한다면 이자의 값은 지속적으로 변경될 것이다.

이와 같이 타 속성에 의해 지속적으로 영향을 받아 자신의 값이 변하는 성질을 가지고 있는 속성이 파생속성이다. 파생속성은 될 수 있으면 꼭 필요한 경우에만 정의하도록 하여 업무로직이 속성내부에 숨지 않도록 하는 것이 좋다. 파생속성을 정의한 경우는 속성정의서에 파생속성이 가지는 업무로직을 기술하여 데이터의 정합성이 유지될 수 있도록 해야 하며 그 파생속성에 원인이 되는 속성을 이용하는 모든 애플리케이션에서는 값을 생성하고, 수정하고 삭제할 때 파생속성도 함께 고려해 주어야 한다. 파생속성은 일반 엔터티에서는 많이 사용하지 않으며 통계관련 엔터티나 배치 작업이 수행되면서 발생되는 엔터티의 경우 많이 이용된다.

나. 엔터티 구성방식에 따른 분류

엔터티를 식별할 수 있는 속성을 PK(Primary Key)속성, 다른 엔터티와의 관계에서 포함된 속성을 FK(Foreign Key)속성, 엔터티에 포함되어 있고 PK, FK에 포함되지 않은 속성을 일반속성이라 한다.



[그림 Ⅰ-1-28] 속성의 분류

또한 속성은 그 안에 세부 의미를 쪼갤 수 있는지에 따라 단순형 혹은 복합형으로 분류할 수 있다. 예를 들면 주소 속성은 시, 구, 동, 번지 등과 같은 여러 세부 속성들로 구성될 수 있는데 이를 복합 속성(Composite Attribute)이라 한다. 또한 나이, 성별 등의 속성은 더 이상 다른 속성들로 구성될 수 없는 단순한 속성이므로 단순 속성(Simple Attribute)이라 한다.

일반적으로 속성은 하나의 값을 가지고 있으나, 그 안에 동일한 성질의 여러 개의 값이 나타나는 경우가 있다. 이 때 속성 하나에 한 개의 값을 가지는 경우를 단일값(Single Value), 그리고 여러 개의 값을 가지는 경우를 다중값(Multi Value) 속성이라 한다. 주민등록번호와 같은 속성은 반드시 하나의 값만 존재하므로 이 속성은 단일값 속성(Single-Valued Attribute)이라 하고, 어떤 사람의 전화번호와 같은 속성은 집, 휴대전화, 회사 전화번호와 같이 여러 개의 값을 가질 수 있다. 자동차의 색상 속성도 차 지붕, 차체, 외부의 색이 다를 수 있다. 이런 속성을 다중값 속성(Multi-Valued Attribute)이라 한다. 다중값 속성의 경우 하나의 엔터티에 포함될 수 없으므로 1차 정규화를 하거나, 아니면 별도의 엔터티를 만들어 관계로 연결해야 한다.

5. 도메인(Domain)

각 속성은 가질 수 있는 값의 범위가 있는데 이를 그 속성의 도메인(Domain)이라 한다. 예를 들면 학생이라는 엔터티가 있을 때 학점이라는 속성의 도메인은 0.0에서 4.0 사이의 실수 값이며 주소라는 속성은 길이가 20자리 이내인 문자열로 정의할 수 있다. 여기서 물론 각 속성은 도메인 이외의 값을 갖지 못한다. 따라서 도메인을 좀더 이해하기 쉽게 정리하면, 엔터티 내에서 속성에 대한 데이터타입과 크기 그리고 제약사항을 지정하는 것이라 할 수 있다.

6. 속성의 명명(Naming)

C/S(Client/Server) 환경이든 Web 환경이든 속성명이 곧 사용자 인터페이스(User Interface)에 나타나기 때문에 업무와 직결되는 항목이다. 그래서 속성 이름을 정확하게 부여하고 용어의 혼란을 없애기 위해 용어사전이라는 업무사전을 프로젝트에 사용하게 된다. 또한 각 속성이 가지는 값의 종류와 범위를 명확하게 하기 위해 도메인정의를 미리 하여 정의하여 용어사전과 같이 사용한다. 용어사전과 도메인정의를 같이 사용하여 프로젝트를 진행할 경우 용어적 표준과 데이터타입의 일관성을 확보할 수 있게 된다.
속성명을 부여하는 원칙은 [그림 Ⅰ-1-29]와 같다.

1. 해당업무에서 사용하는 이름을 부여 한다.
2. 서술식 속성명은 사용하지 않는다.
3. 약어사용은 가급적 제한한다.
4. 전체 데이터모델에서 유일성 확보하는 것이 좋다.

[그림 Ⅰ-1-29] 속성의 명칭 부여

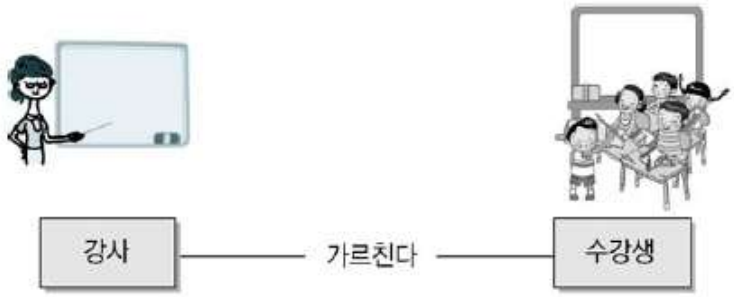
- 속성의 이름을 부여할 때는 현업에서 사용하는 이름을 부여하는 것이 가장 중요하다. 아무리 일반적인 용어라 할지라도 그 업무에서 사용되지 않으면 속성의 명칭으로 사용하지 않는 것이 좋다.
- 일반적으로는 서술식의 속성명은 사용하지 말아야 한다. 명사형을 이용하고 수식어가 많이 붙지 않도록 유의하여 작성한다. 수식어가 많으면 의미파악이 힘들고 상세 설계단계에서 물리속성으로 전환하는데 명확한 의미파악이 어렵게 된다. 소유격도 사용하지 않는다.
- 공용화되지 않은 업무에서 사용하지 않는 약어는 사용하지 않는 것이 좋다. 지나치게 약어를 많이 사용하면 업무분석자 내에서도 의사소통이 제약을 받으며 시스템을 운영할 때도 많은 불편을 초래할 수 있다.
- 가능하면 모든 속성의 이름은 유일하게 작성하는 것이 좋다. 물론 대량의 속성을 정의하는 경우 유일하게 작성하는 것이 어려울 수도 있지만 이렇게 하는 것이 나중에 데이터에 대한 흐름을 파악하고 데이터의 정합성을 유지하는데 큰 도움이 된다. 또한 반정규화(테이블통합, 분리, 칼럼의 중복 등)를 적용할 때 속성명의 충돌(Conflict)을 해결하여 안정적으로 반정규화를 적용할 수 있게 된다.

제4절 관계(Relationship)

1. 관계의 개념

가. 관계의 정의

관계(Relationship)를 사전적으로 정의하면 상호 연관성이 있는 상태로 말할 수 있다. 이것을 데이터 모델에 대입하여 정의해 보면, “엔터티의 인스턴스 사이의 논리적인 연관성으로서 존재의 형태로서나 행위로서 서로에게 연관성이 부여된 상태”라고 할 수 있다. 관계는 엔터티와 엔터티 간 연관성을 표현하기 때문에 엔터티의 정의에 따라 영향을 받기도 하고, 속성 정의 및 관계 정의에 따라서도 다양하게 변할 수 있다.



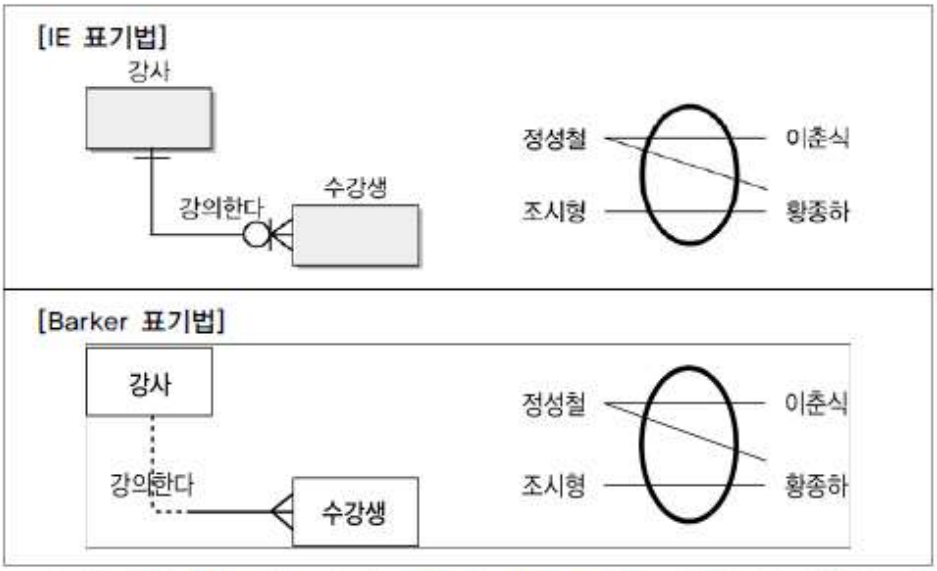
인스턴스 사이의 논리적인 연관성으로서 존재 또는 행위로서 서로에게 연관성이 부여된 상태

[그림 Ⅰ-1-30] 관계의 정의

나. 관계의 패어링

유의해야할 점은 관계는 엔터티 안에 인스턴스가 개별적으로 관계를 가지는 것(패어링)이고 이것의 집합을 관계로 표현한다는 것이다. 따라서 개별 인스턴스가 각각 다른 종류의 관계를 가지고 있다면 두 엔터티 사이에 두 개 이상의 관계가 형성될 수 있다.

각각의 엔터티의 인스턴스들은 자신이 관련된 인스턴스들과 관계의 어커런스로 참여하는 형태를 관계 패어링(Relationship Paring)이라 한다. [그림 Ⅰ-1-31]에서는 강사인 정성철은 이춘식과 황종하에게 강의를 하는 형태로 관계가 표현되어 있고 조시형은 황종하에게 강의를 하는 형태로 되어 있다. 이와 같이 엔터티내에 인스턴스와 인스턴스사이에 관계가 설정되어 있는 어커런스를 관계 패어링이라고 한다. 엔터티는 인스턴스의 집합을 논리적으로 표현하였다면 관계는 관계 패어링의 집합을 논리적으로 표현한 것이다.
최초의 ERD(Chen 모델)에서 관계는 속성을 가질 수 있었으나 요즘 ERD에서는 관계를 위해 속성을 도출하지는 않는다. 관계의 표현에는 이항 관계(Binary Relationship), 삼항 관계(Ternary Relationship), n항 관계가 존재할 수 있는데 실제에 있어서 삼항 관계 이상은 잘 나타나지 않는다.

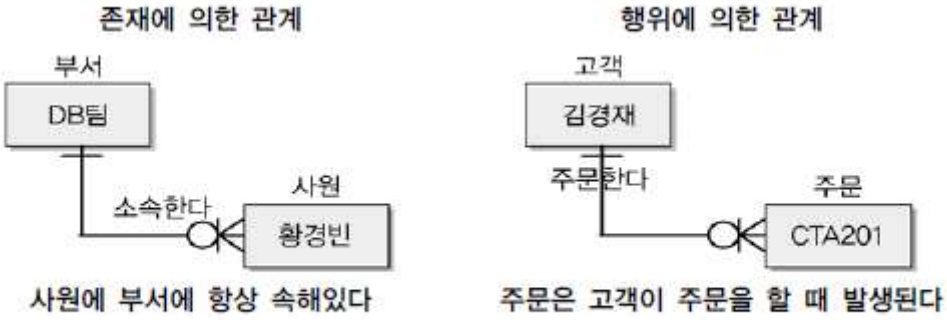


인스턴스 각각은 자신의 연관성을 가지고 있을 수 있음. 이것을 집합하여 '강의'라는 관계 도출

[그림 Ⅰ-1-31] 관계의 패어링

2. 관계의 분류

관계가 존재에 의한 관계와 행위에 의한 관계로 구분될 수 있는 것은 관계를 연결함에 있어 어떤 목적으로 연결되었느냐에 따라 분류하기 때문이다.



[그림 Ⅰ-1-32] 관계의 분류

[그림 Ⅰ-1-32]에서 왼쪽 편에 있는 모델은 황경빈이란 사원이 DB팀에 소속되어 있는 상태를 나타낸다. '소속된다'라는 의미는 행위에 따른 이벤트에 의해 발생되는 의미가 아니고 그냥 황경빈사원이 DB팀에 소속되어 있기 때문에 나타나는 즉 존재의 형태에 의해 관계가 형성되어 있는 것이다.

반면에 오른편에 있는 김경재 고객은 '주문한다'라는 행위를 하여 CTA201이라는 주문번호를 생성하였다. 주문 엔터티의 CTA201 주문번호는 김경재 고객이 '주문한다'라는 행위에 의해 발생되었기 때문에 두 엔터티 사이의 관계는 행위에 의한 관계가 되는 것이다.

UML(Unified Modeling Language)에는 클래스다이어그램의 관계중 연관관계(Association)와 의존관계(Dependency)가 있다. 이 둘의 차이는 연관관계는 항상 이용하는 관계로 존재적 관계에 해당하고 의존관계는 상대방 클래스의 행위에 의해 관계가 형성될 때 구분하여 표현한다는 것이다. 즉, ERD에서는 존재적 관계와 행위에 의한 관계를 구분하지 않고 표현했다면 클래스다이어그램에서는 이것을 구분하여 연관관계와 의존관계로 표현하고 있는 것이다. 연관관계는 표현방법이 실선으로 표현되고 소스코드에서 멤버변수로 선언하여 사용하게 하고, 의존관계는 점선으로 표현되고 행위를 나타내는 코드인 Operation(Method)에서 파라미터 등으로 이용할 수 있도록 되어 있다.

3. 관계의 표기법

관계에서는 표기법이 상당히 복잡하고 여러 가지 의미를 가지고 있다. 다음 3가지 개념과 함께 표기법을 이해할 필요가 있다.

- 관계명(Membership) : 관계의 이름
- 관계차수(Cardinality) : 1:1, 1:M, M:N
- 관계선택사양(Optionality) : 필수관계, 선택관계

가. 관계명(Membership)

관계명은 엔터티가 관계에 참여하는 형태를 지칭한다. 각각의 관계는 두 개의 관계명을 가지고 있다. 또한 각각의 관계명에 의해 두 가지의 관점으로 표현될 수 있다.

[그림 Ⅰ-1-33] 관계의 관계명

엔터티에서 관계가 시작되는 편을 관계시작점(The Beginning)이라고 부르고 받는 편을 관계끝점(The End)이라고 부른다. 관계 시작점과 끝점 모두 관계이름을 가져야 하며 참여자의 관점에 따라 관계이름이 능동적(Active)이거나 수동적(Passive)으로 명명된다. 관계명은 다음과 같은 명명규칙에 따라 작성해야 한다.

- 애매한 동사를 피한다. 예를 들면 '관계된다', '관련이 있다', '이다', '한다' 등은 구체적이지 않아 어떤 행위가 있는지 또는 두 참여자간 어떤 상태가 존재하는지 파악할 수 없다.
- 현재형으로 표현한다. 예를 들면 '수강을 신청했다', '강의를 할 것이다'라는 식으로 표현해서는 안된다. '수강 신청한다', '강의를 한다'로 표현해야 한다.

나. 관계차수(Degree/Cardinality)

두 개의 엔터티간 관계에서 참여자의 수를 표현하는 것을 관계차수(Cardinality)라고 한다. 가장 일반적인 관계차수 표현방법은 1:M, 1:1, M:N이다. 가장 중요하게 고려해야 할 사항은 한 개의 관계가 존재하느냐 아니면 두 개 이상의 멤버쉽이 존재하는지를 파악하는 것이 중요하다.

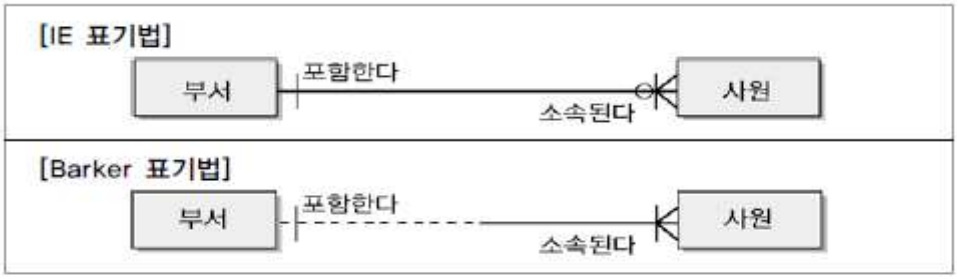
관계차수를 표시하는 방법은 여러 가지 방법이 있지만 Crow's Foot 모델에서는 선을 이용하여 표현한다. 한 개가 참여하는 경우는 실선을 그대로 유지하고 다수가 참여한 경우는(Many) 까마귀발과 같은 모양으로 그려준다.

1) 1:1(ONE TO ONE) 관계를 표시하는 방법

[그림 Ⅰ-1-34] 관계차수(1:1)

관계에 참여하는 각각의 엔터티는 관계를 맺는 다른 엔터티의 엔터티에 대해 단지 하나의 관계만을 가지고 있다.

2) 1:M(ONE TO MANY) 관계를 표시하는 방법



한 명의 사원은 한 부서에 소속되고 한 부서에는 여러 사원을 포함한다

[그림 Ⅰ-1-35] 관계차수(1:M)

관계에 참여하는 각각의 엔터티는 관계를 맺는 다른 엔터티의 엔터티에 대해 하나나 그 이상의 수와 관계를 가지고 있다. 그러나 반대의 방향은 단지 하나만의 관계를 가지고 있다.

3) M:M(MANY TO MANY) 관계를 표시하는 방법

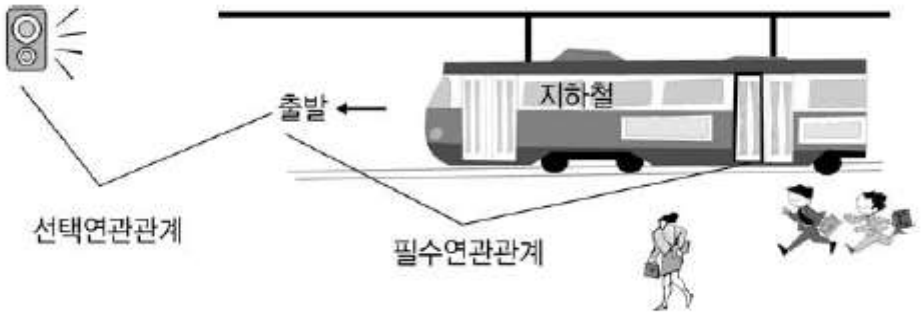
[그림 Ⅰ-1-36] 관계차수(M:M)

관계에 참여하는 각각의 엔터티는 관계를 맺는 다른 엔터티의 엔터티에 대해 하나나 그 이상의 수와 관계를 가지고 있다. 반대의 방향도 동일하게 관계에 참여하는 각각의 엔터티는 관계를 맺는 다른 엔터티의 엔터티에 대해 하나 또는 그 이상의 수와 관계를 가지고 있다. 이렇게 M:N 관계로 표현된 데이터 모델은 이후에 두 개의 주식별자를 상속받은 관계엔터티를 이용하여 3개의 엔터티로 구분하여 표현한다.

다. 관계선택사양(Optionality)

요즈음 웬만한 대도시에는 지하철이 많이 운행된다. 만약 지하철 문이 닫히지 않았는데 지하철이 떠난다면 무슨 일이 발생할까? 아마도 어떤 사람은 머리만 지하철 안에 들어오고 몸은 밖에 있는 채로 끌려갈 것이고, 또 어떤 사람은 가방만 지하철에 실어 보내는 사람도 있을 것이고, 지하철과 승강기 사이에 몸이 낄 수도 있을 것이다. 물론 지하철운행과 지하철문의 관계는 이렇게 설계되지 않아 위와 같은 어처구니없는 일은 발생하지 않을 것이다. “반드시 지하철의 문이 닫혀야만 지하철은 출발한다.” 지하철출발과 지하철문닫힘은 필수(Mandatory)적으로 연결 관계가 있는 것이다. 이와 같은 것이 데이터 모델의 관계에서는 필수참여관계(Mandatory)가 된다.

또 지하철 안내방송시스템의 예를 들어보자. 지하철의 출발을 알리는 안내방송은 지하철의 출발과 상관없이 방송해도 아무런 문제가 발생하지 않는다. 물론 정해진 시간에 방송을 하면 승객에게 정보로서 유익하겠지만 꼭 그렇게 할 필요는 없다. 그래서 가끔씩 시스템의 녹음된 여성의 목소리가 아닌 시끄러운 남자 기사가 방송을 하는 경우가 있다. 안내방송시스템이 고장이 나도 지하철운행에는 별로 영향을 주지 않는다. 방송시점도 조금씩 다르게 나타나도 지하철이 출발하는 것과는 밀접하게 연관되지 않는다. 이와 같이 지하철의 출발과 지하철방송과는 정보로서 관련은 있지만 서로가 필수적인(Mandatory) 관계는 아닌 선택적인 관계(Optional)가 되는 것이다.



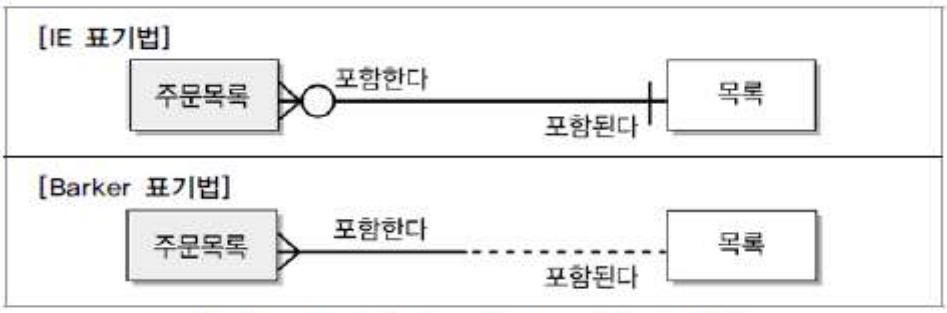
[그림 Ⅰ-1-37] 관계선택사양

이와 같은 것이 데이터 모델 관계에서는 선택참여관계(Optional)가 된다. 참여하는 엔터티가 항상 참여하는지 아니면 참여할 수도 있는지를 나타내는 방법이 필수(Mandatory Membership)와 선택참여(Optional Membership)이다.

필수참여는 참여하는 모든 참여자가 반드시 관계를 가지는, 타 엔터티의 참여자와 연결이 되어야 하는 관계이다. 예를 들면 주문서는 반드시 주문목록을 가져야 하며 주문목록이 없는 주문서는 의미가 없으므로 주문서와 주문목록은 필수참여관계가 되는 것이다. 반대로 목록은 주문이 될 수도 있고 주문이 되지 않은 목록이 있을 수도 있으므로 목록과 주문목록과의 관계는 선택참여(Optional Membership)가 되는 것이다. 선택참여된 항목은 물리속성에서 Foreign Key로 연결될 경우 Null을 허용할 수 있는 항목이 된다. 만약 선택참여로 지정해야 할 관계를 필수참여로 잘못 지정하면 애플리케이션에서 데이터가 발생할 때 반드시 한 개의 트랜잭션으로 제어해야 하는 제약사항이 발생한다. 그러므로 설계단계에서 필수참여와 선택참여는 개발시점에 업무 로직과 직접적으로 관련된 부분이므로 반드시 고려되어야 한다.

선택참여관계는 ERD에서 관계를 나타내는 선에서 선택참여하는 엔터티 쪽을 원으로 표시한다. 필수참여는 아무런 표시를 하지 않는다.

만약 관계가 표시된 양쪽 엔터티에 모두 선택참여가 표시된다면, 즉 0:0(Zero to Zero)의 관계가 된다면 그 관계는 잘못될 확률이 많으므로 관계설정이 잘못되었는지를 검토해 보아야 한다.



하나의 주문목록에는 한 개의 목록을 항상 포함하고
한 목록은 여러 개의 주문 목록에 의해 포함될 수 있다.

[그림 Ⅰ-1-38] 관계선택참여

관계선택사양은 관계를 통한 상대방과의 업무적인 제약조건을 표현하는 것으로서 간단하면서 아주 중요한 표기법이다. 이것을 어떻게 설정했는지에 따라 참조무결성 제약조건의 규칙이 바뀌게 되므로 주의 깊게 모델링을 해야 한다.

4. 관계의 정의 및 읽는 방법

가. 관계 체크사항

두 개의 엔터티 사이에서 관계를 정의할 때 다음 사항을 체크해 보도록 한다.

- 두 개의 엔터티 사이에 관심있는 연관규칙이 존재하는가?
- 두 개의 엔터티 사이에 정보의 조합이 발생되는가?
- 업무기술서, 장표에 관계연결에 대한 규칙이 서술되어 있는가?
- 업무기술서, 장표에 관계연결을 가능하게 하는 동사(Verb)가 있는가?

나. 관계 읽기

데이터 모델을 읽는 방법은 먼저 관계에 참여하는 기준 엔터티를 하나 또는 각(Each)으로 읽고 대상 엔터티의 개수(하나, 하나 이상)를 읽고 관계선택사양과 관계명을 읽도록 한다.

- 기준(Source) 엔터티를 한 개(One) 또는 각(Each)으로 읽는다.
- 대상(Target) 엔터티의 관계참여도 즉 개수(하나, 하나 이상)를 읽는다.
- 관계선택사양과 관계명을 읽는다.

[그림 Ⅰ-1-39] 관계의 읽는 방법

위의 관계를 정의를 한 사항에 대해서 뒷부분만 의문문으로 만들면 바로 관계를 도출하기 위한 질문 문장으로 만들 수 있다. 위의 질문을 업무를 분석하는 자기 스스로에게 질문하거나, 장표나 업무기술서 또는 업무를 잘 알고 있는 업무담당 고객과 대화를 하면서 관계를 완성해 갈 수 있다. 예를 들어, 주문과 제품과 관계를 질문하기 원할 때 “한 주문에 대해서 하나의 제품만을 주문합니까?”라고 할 수도 있고 또는 “한 제품은 하나의 주문에 대해서만 주문을 접수받을 수 있습니까?”라고 질문할 수 있다. 이러한 질문 방법은 엔터티간 관계설정뿐 아니라 업무의 흐름도 분석이 되는 실제 프로젝트에서 효과적인 방법이 된다.

제5절 식별자

1. 식별자(Identifiers) 개념

엔터티는 인스턴스들의 집합이라고 하였다. 여러 개의 집합체를 담고 있는 하나의 통에서 각각을 구분할 수 있는 논리적인 이름이 있어야 한다. 이 구분자를 식별자(Identifier)라고 한다. 식별자란 하나의 엔터티에 구성되어 있는 여러 개의 속성 중에 엔터티를 대표할 수 있는 속성을 의미하며 하나의 엔터티는 반드시 하나의 유일한 식별자가 존재해야 한다. 보통 식별자와 키(Key)를 동일하게 생각하는 경우가 있는데 식별자라는 용어는 업무적으로 구분이 되는 정보로 생각할 수 있으므로 논리 데이터 모델링 단계에서 사용하고 키는 데이터베이스 테이블에 접근을 위한 매개체로서 물리 데이터 모델링 단계에서 사용한다.

[그림 Ⅰ-1-40] 식별자의 정의

2. 식별자의 특징

주식별자인지 아니면 외부식별자인지 등에 따라 특성이 다소 차이가 있다. 먼저 주식별자일 경우 다음과 같은 특징을 갖는다.

- 주식별자에 의해 엔터티내에 모든 인스턴스들이 유일하게 구분되어야 한다.
- 주식별자를 구성하는 속성의 수는 유일성을 만족하는 최소의 수가 되어야 한다.
- 지정된 주식별자의 값은 자주 변하지 않는 것이어야 한다.
- 주식별자가 지정이 되면 반드시 값이 들어와야 한다.

[표 Ⅰ-1-7] 주식별자의 특징

특징	내용	비고
유일성	주식별자에 의해 엔터티내에 모든 인스턴스들을 유일하게 구분함	예) 사원번호가 주식별자가 모든 직원들에 대해 개인별로 고유하게 부여됨
최소성	주식별자를 구성하는 속성의 수는 유일성을 만족하는 최소의 수가 되어야 함	예) 사원번호만으로도 고유한 구조인데 사원분류코드+사원번호로 식별자가 구성될 경우 부절한 주식별자 구조임
불변성	주식별자가 한 번 특정 엔터티에 지정되면 그 식별자의 값은 변하지 않아야 함	예) 사원번호의 값이 변한다는 의미는 이전기록이 말소되고 새로운 기록이 발생되는 개념임
존재성	주식별자가 지정되면 반드시 데이터 값이 존재 (Null 안됨)	예) 사원번호 없는 회사직원은 있을 수 없음

대체식별자의 특징은 주식별자의 특징과 일치하지만 외부식별자는 별도의 특징을 가지고 있다. 외부식별자의 경우 주식별자 특징과 일치하지 않으며 참조무결성 제약조건(Referential Integrity)에 따른 특징을 가지고 있다.

3. 식별자 분류 및 표기법

가. 식별자 분류

식별자의 종류는 자신의 엔터티 내에서 대표성을 가지는가에 따라 주식별자(Primary Identifier)와 보조식별자(Alternate Identifier)로 구분하고 엔터티 내에서 스스로 생성되었는지 여부에 따라 내부식별자와 외부식별자(Foreign Identifier)로 구분할 수 있다. 또한 단일 속성으로 식별이 되는가에 따라 단일식별자(Single Identifier)와 복합식별자(Composit Identifier)로 구분할 수 있다. 원래 업무적으로 의미가 있던 식별자 속성을 대체하여 일련번호와 같이 새롭게 만든 식별자를 구분하기 위해 본질식별자와 인조식별자로도 구분할 수 있다.

[그림 Ⅰ-1-41] 식별자의 분류

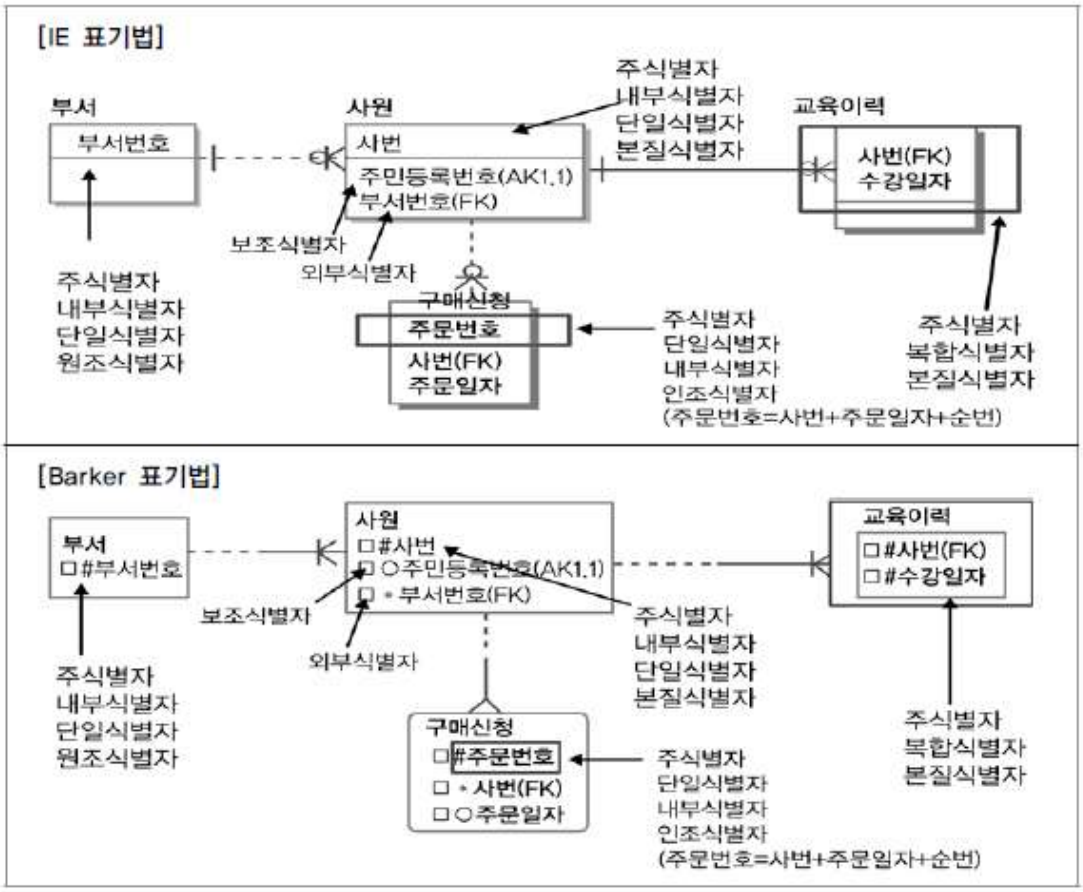
[그림 Ⅰ-1-41]의 식별자에 대한 분류체계를 좀 더 상세하게 설명하면 [표 Ⅰ-1-8]과 같이 표현할 수 있다.

[표 Ⅰ-1-8] 식별자의 분류체계

분류	식별자	설명
대표성 여부	주식별자	엔터티 내에서 각 어커런스를 구분할 수 있는 구분자이며, 타 엔터티와 참조관계를 연결할 수 있는 식별자
	보조식별자	엔터티 내에서 각 어커런스를 구분할 수 있는 구분자이나 대표성을 가지지 못해 참조관계 연결을 못함
스스로 생성여부	내부식별자	엔터티 내부에서 스스로 만들어지는 식별자
	외부식별자	타 엔터티와의 관계를 통해 타 엔터티로부터 받아오는 식별자
속성의 수	단일식별자	하나의 속성으로 구성된 식별자
	복합식별자	둘 이상의 속성으로 구성된 식별자
대체 여부	본질식별자	업무에 의해 만들어지는 식별자
	인조식별자	업무적으로 만들어지지는 않지만 원조식별자가 복잡한 구성을 가지고 있기 때문에 인위적으로 만든 식별자

나. 식별자 표기법

식별자에 대한 위의 분류방법을 데이터 모델에서 표현하면 [그림 Ⅰ-1-42]와 같이 분류하여 설명할 수 있다.



[그림 Ⅰ-1-42] 식별자의 분류-데이터 모델

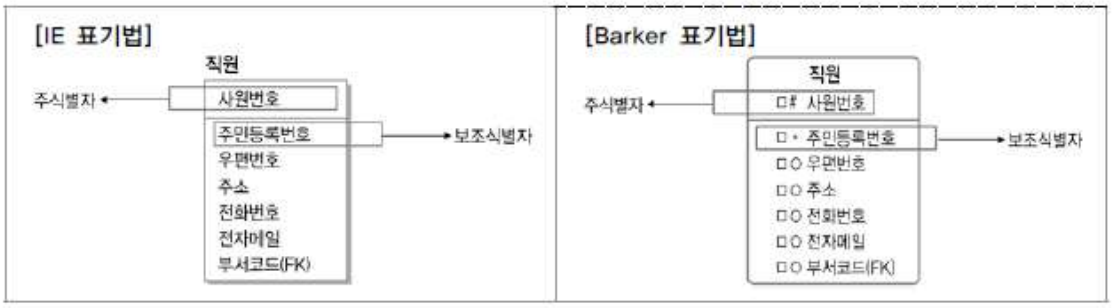
4. 주식별자 도출기준

데이터 모델링 작업에서 중요한 작업 중의 하나가 주식별자 도출작업이다. 주식별자를 도출하기 위한 기준을 정리하면 다음과 같다.

- 해당 업무에서 자주 이용되는 속성을 주식별자로 지정한다.
- 명칭, 내역 등과 같이 이름으로 기술되는 것들은 가능하면 주식별자로 지정하지 않는다.
- 복합으로 주식별자로 구성할 경우 너무 많은 속성이 포함되지 않도록 한다.

가. 해당 업무에서 자주 이용되는 속성을 주식별자로 지정하도록 함

예를 들면, 직원이라는 엔터티가 있을 때 유일하게 식별가능한 속성으로는 주민등록번호와 사원번호가 존재할 수 있다. 사원번호가 그 회사에서 직원을 관리할 때 흔히 사용되므로 사원번호를 주식별자로 지정하고 주민등록번호는 보조식별자로 사용할 수 있다.

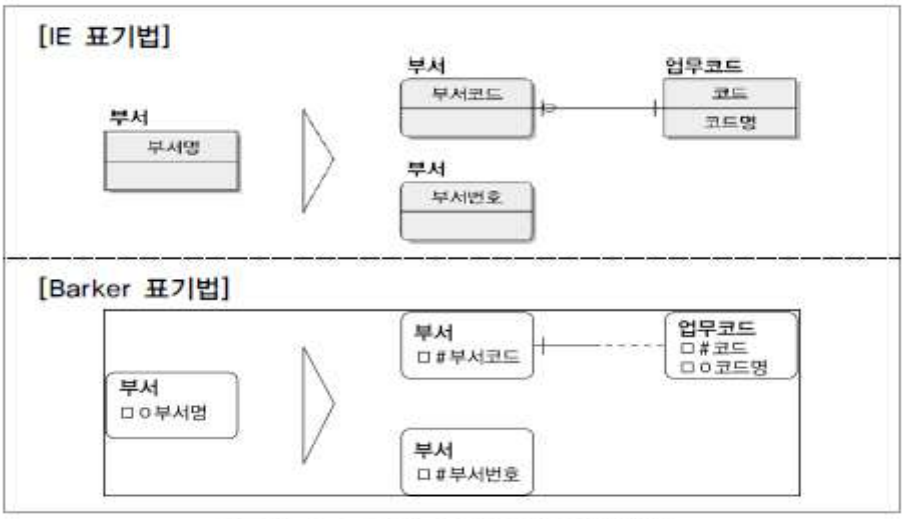


[그림 Ⅰ-1-43] 식별자의 주식별자

나. 명칭, 내역 등과 같이 이름으로 기술되는 것은 피함

명칭, 내역 등과 같이 이름으로 기술되는 것들은 가능하면 주식별자로 지정하지 않도록 한다. 예를 들어, 한 회사에 부서이름이 100개가 있다고 할 때, 각각의 부서이름은 유일하게 구별될 수 있다고 하여 부서이름을 주식별자로 지정하지 않도록 해야 한다. 만약 부서이름을 주식별자로 선정하면 물리데이터베이스로 테이블을 생성하여 데이터를 읽을 때 항상 부서이름이 WHERE 조건절에 기술되는 현상이 발생된다. 부서이름은 많은 경우 20자 이상이 될 수 있으므로 조건절에 정확한 부서이름을 기술하기는 쉬운 일이 아니다.

이와 같이 명칭이나 내역이 있고 인스턴스들을 식별할 수 있는 다른 구분자가 존재하지 않을 경우는 새로운 식별자를 생성하도록 한다. 보통 일련번호와 코드를 많이 사용한다.

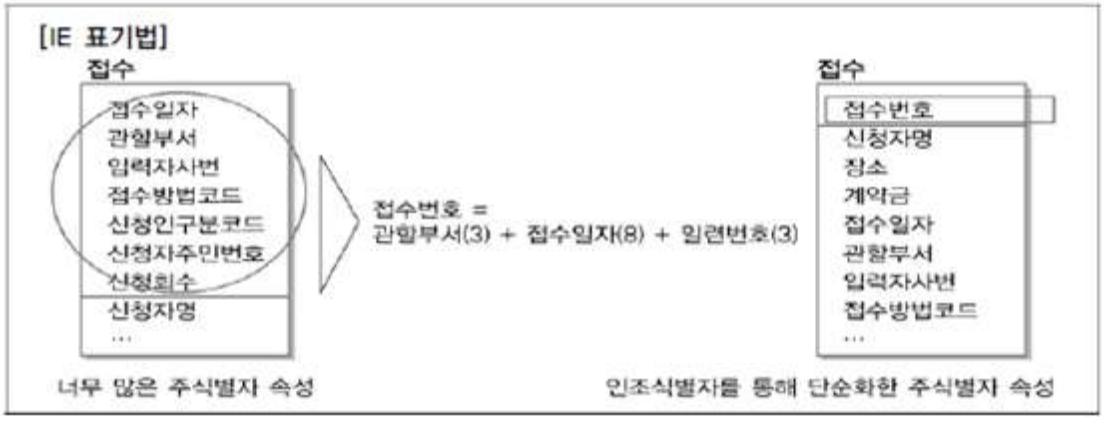


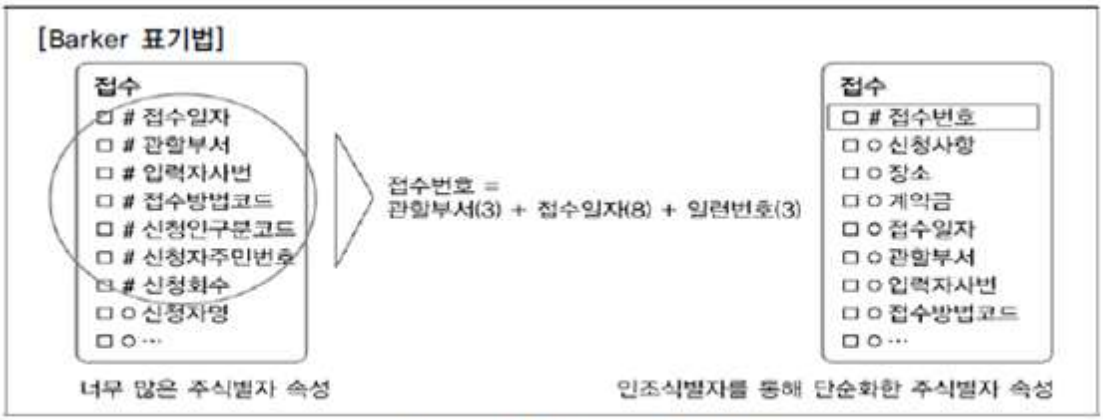
[그림 Ⅰ-1-44] 주식별자-명칭/내역

부서명과 같은 경우는 부서코드를 부여하여 코드엔터티에 등록한 후 부서코드로 주식별자를 지정하는 방법과 부서일련번호(부서번호)를 주식별자로 하고 부서명은 보조식별자로 활용하는 두 가지 방법이 있다.

다. 속성의 수가 많아지지 않도록 함

주식별자로 선정하기 위한 속성이 복합으로 구성되어 주식별자가 될 수 있을 때 가능하면 주식별자 선정하기 위한 속성의 수가 많지 않도록 해야 한다. 그러나 만약 주식별자로 선정된 속성들이 자신이 가지고 있는 자식엔터티로부터 손자엔터티, 그리고 증손자엔터티까지 계속해서 상속이 되는 속성이고 복잡한 데이터 모델이 구현되어 물리데이터베이스에서 조인으로 인한 성능저하가 예상되는 모습을 가지고 있다면 속성의 반정규화 측면에서 하나의 테이블에 많은 속성이 있는 것이 인정될 수도 있다. 하지만 일반적으로 주식별자의 속성의 개수가 많다는 것(일반적으로 7~8개 이상)은 새로운 인조식별자(Artificial Identifier)를 생성하여 데이터 모델을 구성하는 것이 데이터 모델을 한층 더 단순하게 하고 애플리케이션을 개발할 때 조건절을 단순하게 할 수 있는 방법이 될 수 있다.





[그림 Ⅰ-1-45] 주식별자-복합속성

[그림 Ⅰ-1-45]는 왼편의 복잡한 주식별자를 임의의 주식별자를 생성하여 단순화한 예를 보여준다. 왼편 그림에서는 접수라는 엔터티에 어떤 부류의 사람이 특정접수방법에 의해 특정 날짜에 여러 번 신청하는 것을 관할부성에서 접수담당자가 입력한 대로 데이터가 식별될 수 있는 업무 규칙이 있는 경우이다. 접수의 주식별자가 접수일자, 관할부서, 입력자사번, 접수방법코드, 신청인구분코드, 신청자주민번호, 신청회수 등 7개 이상의 복잡한 속성을 가지고 있다. 이러한 모델의 경우 실제 테이블에 Primary Key는 7개가 생성될 것이고 만약 특정 신청인의 계약금 하나만 가져온다고 하더라도 다음과 같이 복잡한 SQL문장을 구사해야 한다.

```
SELECT 계약금
  FROM 접수
WHERE 접수.접수일자 = '2010.07.15"
   AND 접수.관할부서 = '1001'
   AND 접수.입력자사번 = 'AB45588'
   AND 접수.접수방법코드 = 'E'
   AND 접수.신청인구분코드 = '01'
   AND 접수.신청인주민번호 = '7007171234567'
   AND 접수.신청횟수 = '1'
```

이렇게 된 SQL문장을 접수번호라고 하는 인조식별자로 대체했다고 하면 특정신청인의 계약금 조회는 다음과 같이 간단하게 할 수 있다.

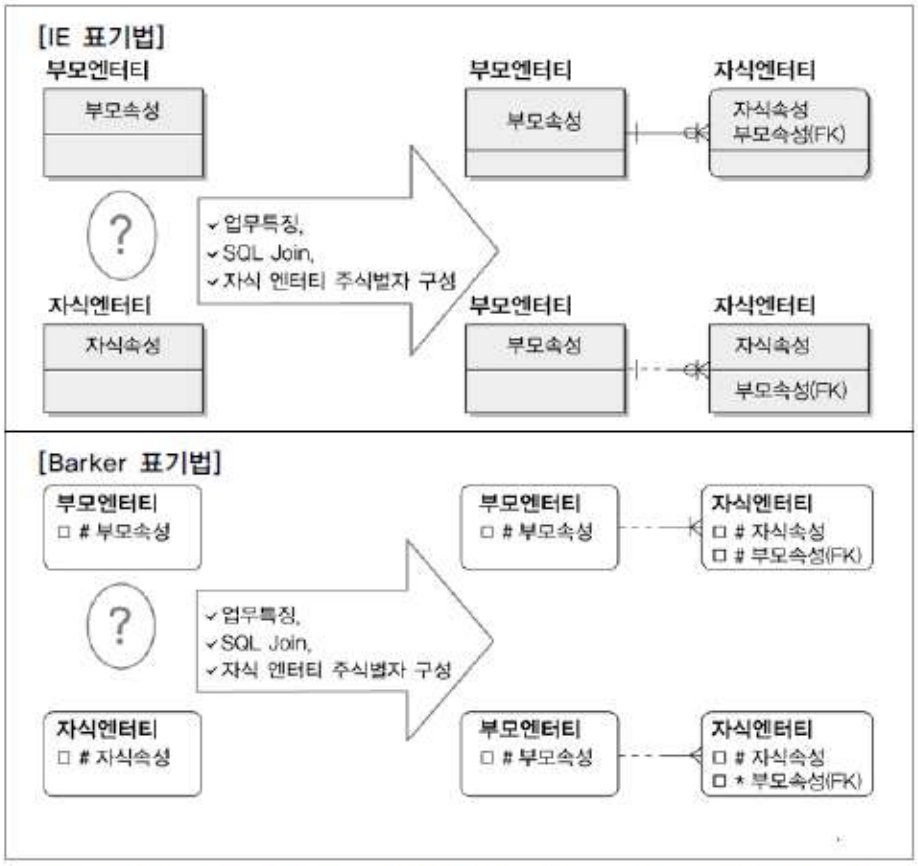
```
SELECT 계약금
  FROM 접수
 WHERE 접수.접수일자 = '100120100715001"
```

만약 접수 엔터티가 자식과 손자엔터티를 가지고 있고 자식과 손자엔터티에 있는 데이터를 서로 조인하여 가져오고자 한다면 아무리 간단한 SQL문장이라도 쉽게 A4용지 한 페이지는 넘어갈 것이다. 이렇게 모델상에 표현하는 문장의 간편성뿐만 아니라 애플리케이션 구성에 있어서도 복잡한 소스구성을 피하기 위하여 과도한 복합키는 배제하도록 노력해야 한다.

5. 식별자관계와 비식별자관계에 따른 식별자

가. 식별자관계와 비식별자 관계의 결정

외부식별자(Foreign Identifier)는 자기 자신의 엔터티에서 필요한 속성이 아니라 다른 엔터티와의 관계를 통해 자식 쪽에 엔터티에 생성되는 속성을 외부식별자라 하며 데이터베이스 생성 시에 Foreign Key역할을 한다. 관계와 속성을 정의하고 주식별자를 정의하면 논리적인 관계에 의해 자연스럽게 외부식별자가 도출되지만 중요하게 고려해야 할 사항이 있다. 엔터티에 주식별자가 지정되고 엔터티간 관계를 연결하면 부모쪽의 주식별자를 자식엔터티의 속성으로 내려 보낸다. 이 때 자식엔터티에서 부모엔터티로부터 받은 외부식별자를 자신의 주식별자로 이용할 것인지 또는 부모와 연결이 되는 속성으로서만 이용할 것인지를 결정해야 한다.



엔터티 사이 관계유형은 업무특징, 자식엔터티의 주식별자구성, SQL 전략에 의해 결정된다.

[그림 Ⅰ-1-46] 식별자/비식별자관계 조정

나. 식별자관계

부모로부터 받은 식별자를 자식엔터티의 주식별자로 이용하는 경우는 Null값이 오면 안 되므로 반드시 부모엔터티가 생성되어야 자기 자신의 엔터티가 생성되는 경우이다. 부모로부터 받은 속성을 자식엔터티가 모두 사용하고 그것만으로 주식별자로 사용한다면 부모엔터티와 자식엔터티의 관계는 1:1의 관계가 될 것이고 만약 부모로부터 받은 속성을 포함하여 다른 부모엔터티에서 받은 속성을 포함하거나 스스로 가지고 있는 속성과 함께 주식별자로 구성되는 경우는 1:M 관계가 된다.

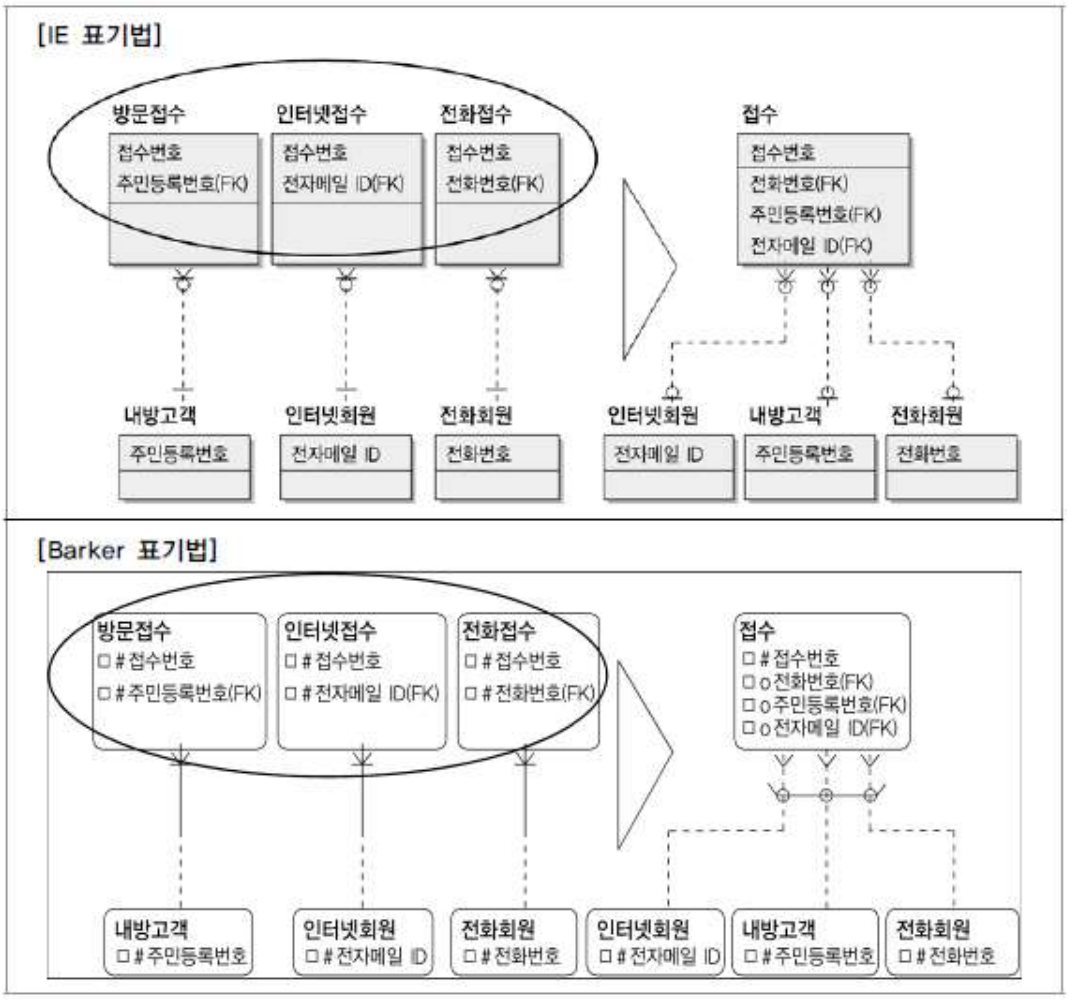
[그림 Ⅰ-1-47] 외부식별자의 주식별자 역할

[그림 Ⅰ-1-47]에서 발령엔터티는 반드시 사원엔터티가 있어야 자신도 생성될 수 있고 자신의 주식별자도 부모엔터티의 외부식별자 사원번호와 자신의 속성 발령번호로 이루어져 있음을 알 수 있다. 이 때 사원과 발령의 관계는 1:M 관계이다. 또한 사원과 임시직사원의 관계와 주식별자를 보면, 임시직사원의 주식별자는 사원의 주식별자와 동일하게 이용되는 경우를 볼 수 있다. 1:1 관계에서 이와 같이 나타나며 주식별자가 동일하며 엔터티 통합의 대상이 됨을 알 수 있다. 이와 같이 자식엔터티의 주식별자로 부모의 주식별자가 상속이 되는 경우를 식별자 관계(Identifying Relationship)라고 지칭한다.

다. 비식별자관계

부모엔터티로부터 속성을 받았지만 자식엔터티의 주식별자로 사용하지 않고 일반적인 속성으로만 사용하는 경우가 있다. 이과 같은 경우를 비식별자 관계(Non-Identifying Relationship)라고 하며 다음의 네 가지 경우에 비식별자 관계에 의한 외부속성을 생성한다.

1) 자식엔터티에서 받은 속성이 반드시 필수가 아니어도 무방하기 때문에 부모 없는 자식이 생성될 수 있는 경우이다.
2) 엔터티별로 데이터의 생명주기(Life Cycle)를 다르게 관리할 경우이다. 예를 들어 부모엔터티에 인스턴스가 자식의 엔터티와 관계를 가지고 있었지만 자식만 남겨두고 먼저 소멸될 수 있는 경우가 이에 해당된다. 이에 대한 방안으로 물리데이터베이스 생성 시 Foreign Key를 연결하지 않는 임시적인 방법을 사용하기도 하지만 데이터 모델상에서 관계를 비식별자관계로 조정하는 것이 가장 좋은 방법이다.
3) 여러 개의 엔터티가 하나의 엔터티로 통합되어 표현되었는데 각각의 엔터티가 별도의 관계를 가질 때이며 이에 해당된다.

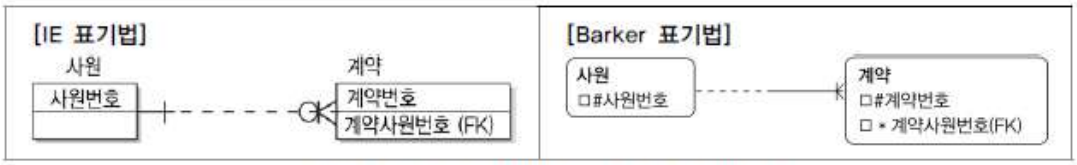


엔터티가 통합되면서 원래 식별자관계가 비식별자관계로 될 수 밖에 없게 됨

[그림 Ⅰ-1-48] 외부식별자의 비식별자 역할

[그림 Ⅰ-1-48]은 접수엔터티가 인터넷접수, 내방접수, 전화접수가 하나로 통합되어 표현되어 있는 경우에 해당되며 통합된 엔터티에 각각의 엔터티가 별도의 관계를 가지고 있고 각각의 관계로부터 받은 주식별자를 접수엔터티의 주식별자로 사용할 수 없는 모습을 보여준다.

4) 자식엔터티에 주식별자로 사용하여도 되지만 자식엔터티에서 별도의 주식별자를 생성하는 것이 더 유리하다고 판단될 때 비식별자 관계에 의한 외부식별자로 표현한다.

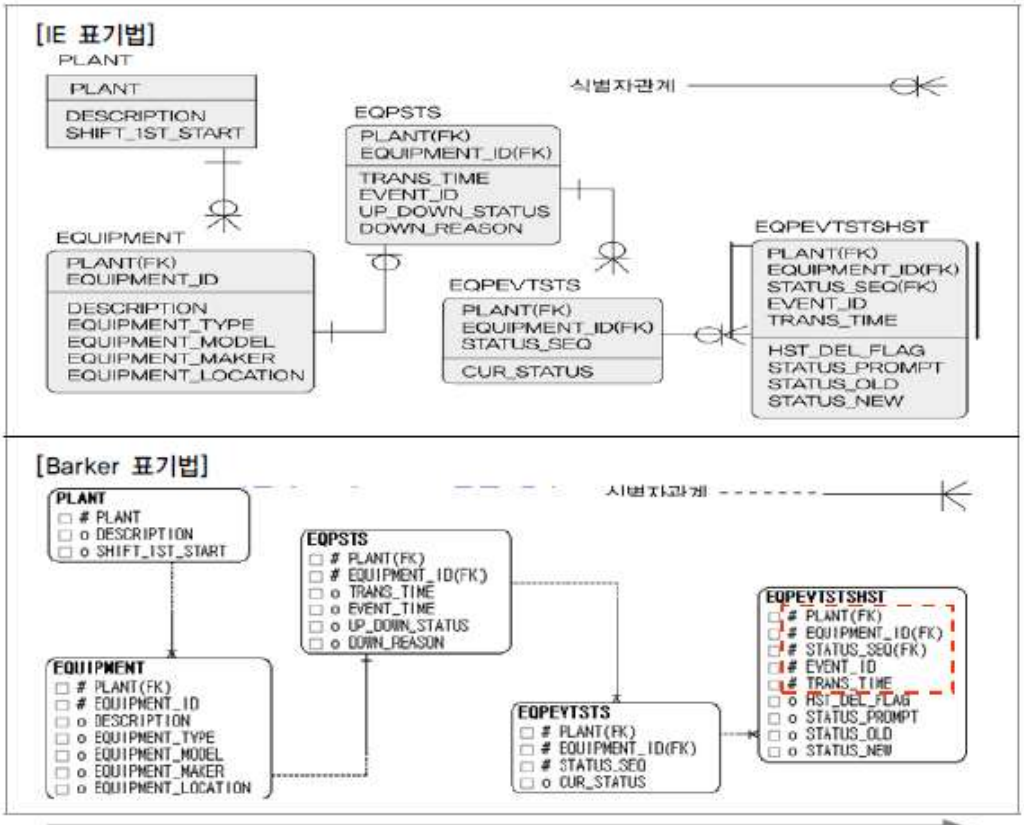


[그림 Ⅰ-1-49] 자식엔터티의 독립 주식별자

[그림 Ⅰ-1-49]는 계약이 반드시 사원에 의해 이루어져 사원번호와 계약번호로 주식별자를 구성할 수 있지만 계약번호 단독으로도 계약 엔터티의 주식별자를 구성할 수 있으므로 하나만 가지고 있는 것이 더 효율적이라고 판단하여 계약번호만 주식별자로 하고 계약사원번호는 일반속성 외부식별자로서 사용하게 된 경우이다.

라. 식별자 관계로만 설정할 경우의 문제점

단지 식별자관계와 비식별자관계에 대한 설정을 고려하지 않은 것이 개발의 복잡성을 증가시키는 요인이 될까?



부모에서 자식으로 식별자 관계로 연결되므로 인해 주식별자의 속성 수가 많아지게 된다.

[그림 Ⅰ-1-50] 식별자관계 연결의 주식별자

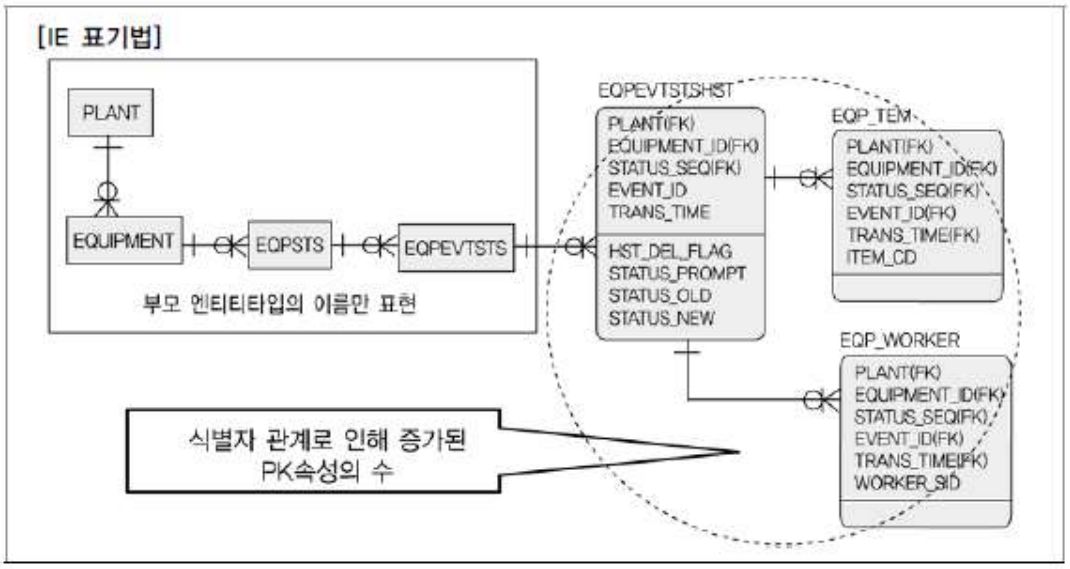
[그림 Ⅰ-1-50]의 데이터 모델은 원시 엔터티이었던 PLANT의 경우 단지 한 개의 속성만이 PK속성이었으나 EQPEVTSTSHST 엔터티의 경우 부모로부터 모두 식별자관계 연결로 인해 PK속성의 개수가 무려 5개나 설정될 수 밖에 없게 된 사례이다.

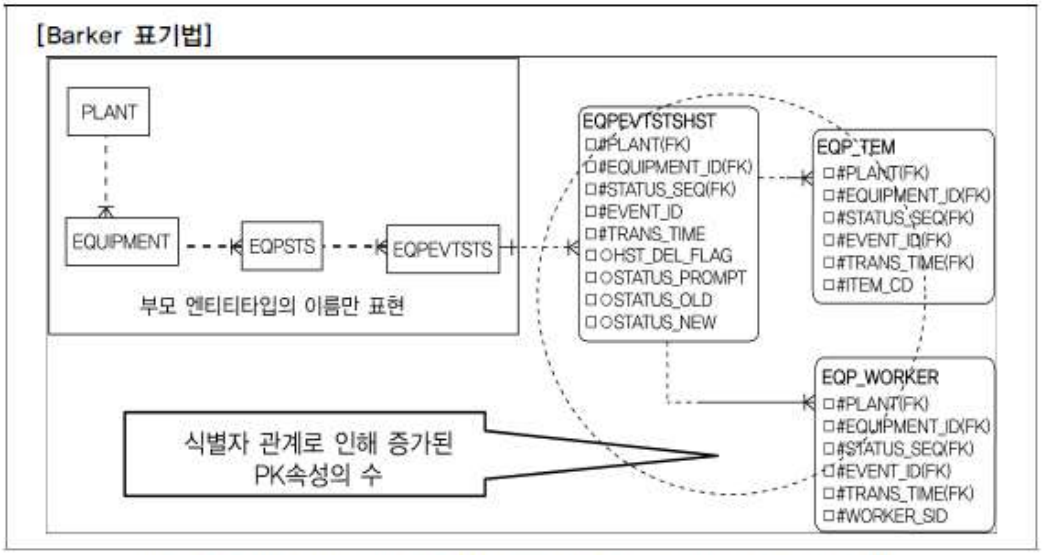
즉, PLANT 엔터티에는 PK속성의 수가 1개이고 관계가 1:M으로 전개되었으므로 자식 엔터티는 PLANT 엔터티의 PK속성의 수 + 1이 성립된다. 물론 1개 이상의 속성의 추가되어야 1:M 관계를 만족할 수 있다. 이와 같은 원리에 의해 1:M 관계의 식별자관계의 PK속성의 수는 다음과 같다.

원 부모엔터티 : 1개
2대 부모엔터티 : 2개 이상 = 원부모 1개 + 추가 1개 이상 +
3대 부모엔터티 : 3개 이상 = 원부모 1개 + 2대 1개 + 추가 1개 이상
3대 부모엔터티 : 3개 이상 = 원부모 1개 + 2대 1개 + 3대 1개 + 추가 1개 이상
4대 부모엔터티 : 4개 이상 = 원부모 1개 + 2대 1개 + 3대 1개 + 4 1개 + 추가 1개 이상 ....

이와 같은 규칙에 의해 지속적으로 식별자 관계를 연결한 데이터 모델의 PK속성의 수는 데이터 모델의 흐름이 길어질수록 증가할 수 밖에 없는 구조를 가지게 된다.

[그림 I-1-50]의 예시 모델에서 EQPEVTSTSHST에 설정된 PK속성을 이용하여 해당 이력의 수리 ITEM과 작업자에 대한 엔터티가 별도로 발생이 가능한 모델은 다음 [그림 I-1-51]과 같이 PK속성 수가 많은 데이터 모델을 만들 수밖에 없다.





세 개의 테이블에서 정보를 가져오는 SQL 구문을 만들면 어떻게 될까?

[그림 Ⅰ-1-51] 식별자관계 연결의 주식별자

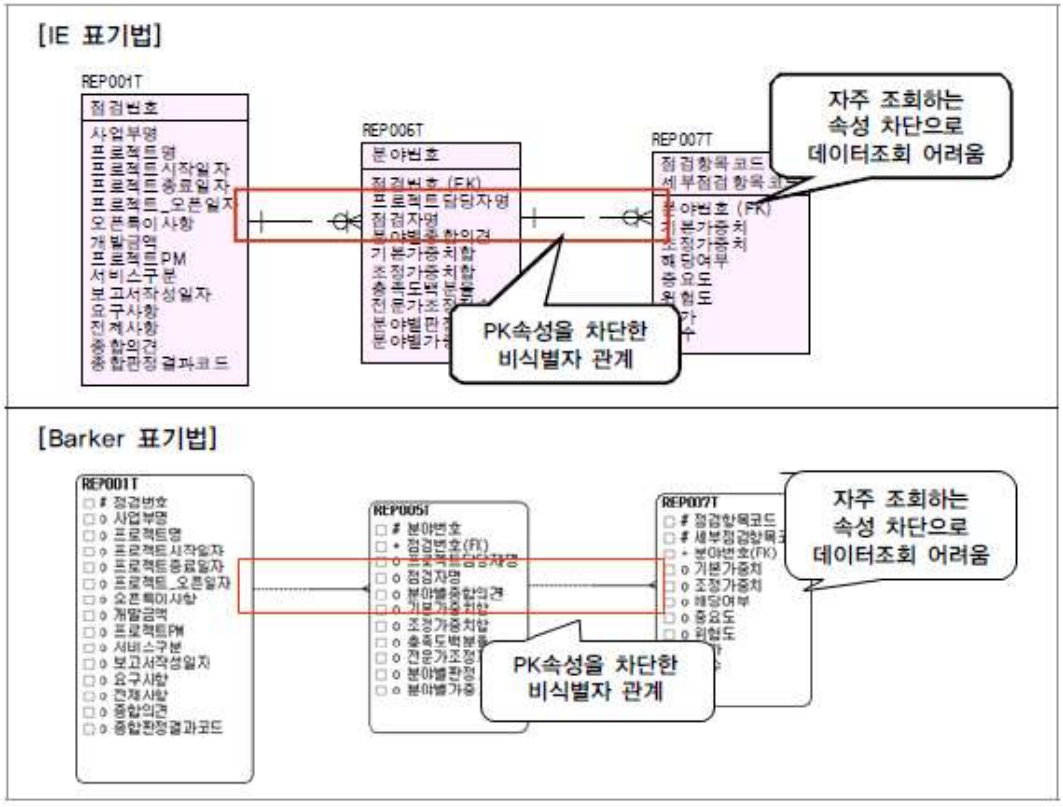
예시 모델의 맨 하위에 있는 EQPEVTSTSHST에서 다시 새로운 엔터티 EQP_ITEM, EQP_WORKER와 관계를 맺고 있을 때 이 세 개의 엔터티에서 정보를 가져오는 SQL구문을 작성해 보자.

```
SELECT A.EVENT_ID,
       A.TRANS_TIME,
       A.HST_DEL_FLAG,
       A.STATUS_PROMPT,
       A.STATUS_OLD,
       A.STATUS_NEW
  FROM EQPEVTSTSHST A, EQP_ITEM B, EQP_WORKER C
WHERE A.PLANT = B.PLANT
   AND A.EQUIPMENT_ID = B.EQUIPMENT_ID
   AND A.STATUS_SEQ = B.STATUS_SEQ
   AND A.EVENT_ID = B.EVENT_ID
   AND A.TRANS_TIME = B.TRANS_TIME
   AND B.ITEM_CD = 'A001'
   AND A.PLANT = C.PLANT
   AND A.EQUIPMENT_ID = C.EQUIPMENT_ID
   AND A.STATUS_SEQ = C.STATUS_SEQ
   AND A.EVENT_ID = C.EVENT_ID
   AND A.TRANS_TIME = C.TRANS_TIME
   AND C.WORKER_SID = 'A012008001'
```

고작 3개 정도의 엔터티를 조인했을 뿐인데 SQL구문의 WHERE절이 매우 길어진 사실을 확인할 수 있다. 실제로 프로젝트에서는 개발자가 개발할 때 당연히 데이터 모델을 참조하면서 엔터티와 관계를 이용하여 개발해야 하는데 생성된 엔터티 스키마 정보만을 보고 개발하는 경우가 많다. 그런데 위와 같이 조인에 참여하는 주식별자속성의 수가 많을 경우 정확하게 조인관계를 설정하지 않고 즉, 누락하여 개발하는 경우가 간혹 발견되기도 한다. 정리하면 식별자 관계만으로 연결된 데이터 모델의 특징은 주식별자 속성이 지속적으로 증가할 수 밖에 없는 구조로서 개발자 복잡성과 오류가능성을 유발시킬 수 있는 요인이 될 수 있다는 사실을 기억해야 한다.

마. 비식별자 관계로만 설정할 경우의 문제점

일반적으로 각각의 엔터티에는 중요한 기준 속성이 있는데 이러한 기준속성은 부모엔터티에 있는 PK속성으로부터 상속되어 자식엔터티에 존재하는 경우가 많다. 이러한 속성의 예로 '주민등록번호', '사원번호', '주문번호', '목록번호' 등이 있다. 이런 속성은 부모엔터티를 조회할 때도 당연히 쓰이지만 자식엔터티의 데이터를 조회할 때도 해당 조건이 조회의 조건으로 걸리는 경우가 다수이다. 그런데 데이터 모델링을 전개할 때 각 엔터티 간의 관계를 비식별자 관계로 설정하면 이런 유형의 속성이 자식엔터티로 상속이 되지 않아 자식엔터티에서 데이터를 처리할 때 쓸데없이 부모엔터티까지 찾아가야 하는 경우가 발생된다.



테이블 REP007T에서 점검번호='301'인 데이터를 조회해 보자

[그림 Ⅰ-1-52] 비식별자관계

[그림 Ⅰ-1-52]의 예에서는 REP001T, REP005T, REP007T 간의 관계가 비식별자 관계로 연결되면서 점검번호, 분야번호 속성이 관계를 타고 자식엔터티로 내려가는 것을 차단하였다. 이러한 모델에서는 만약 위 모델에 있는 맨 하위에 있는 REP007T 엔터티에서 어떤 점검에 대한 정보를 보려고 하면 불필요한 조인이 다량으로 유발되면서 SQL구문도 길어지고 성능이 저하되는 현상이 발생이 된다.

[표 Ⅰ-1-9] 식별자와 비식별자관계 비교

비식별자관계	식별자 관계
[IE 표기법] [Barker 표기법] 	[IE 표기법] [Barker 표기법]
SELECT A. 기본가중치, A. 조정가중치 FROM REP007T A, REP005T, REP001T C WHERE A. 분야번호 = B. 분야번호 AND B. 점검번호 = C. 점검번호 AND C. 점검번호 = '105'	SELECT A. 기본가중치, A. 조정가중치 FROM REP007T A, REP005T, REP001T C WHERE A.점검번호 = '105'
테이블 REP007T에서 점검번호='301'인 데이터를 조회	

[표 Ⅰ-1-9]의 조회 패턴을 보면 고작 점검번호='301' 정도로서 간단하면서도 이 업무에서는 가장 근간이 되는 조회의 패턴 정보로 여겨지는 조건이다. 이 조건은 아마도 웬만한 업무처리에는 많이 포함될 것으로 보이는데 단순하게 걸리는 이 하나의 조회 조건도 왼쪽과 같이 비식별자 관계로만 데이터 모델링을 전개하다 보면 SQL구문에 많은 조인이 걸리게 되고 그에 따라 복잡성이 증가하고 성능이 저하되게 되는 것이다.

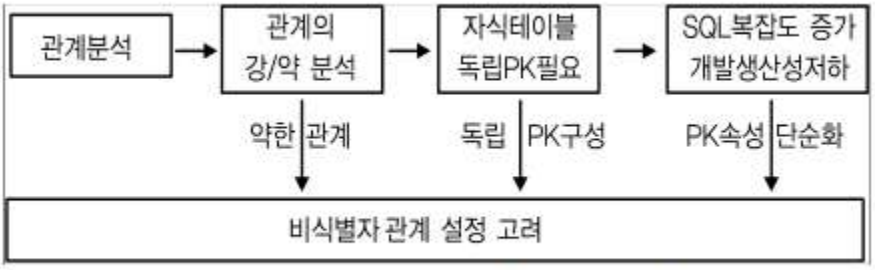
오른쪽과 같이 식별자관계를 통해 연결하다보면, 부모의 모든 주식별자 속성을 상속받음으로 인해 맨 하위에 있는 자식엔터티에서 바로 조회의 조건을 이용하여 원하는 정보를 가져올 수 있다. 이러한 경우는 당연히 성능과 개발의 용이성 측면에서는 식별자관계가 우위에 있음을 보여준다.

따라서 이 두 가지 경우에 대해서 일정한 규칙을 가지고 데이터 모델링을 하는 기술이 필요하며, 다음에 제시된 고려사항을 데이터 모델링에 반영한다면 효과적인 데이터 모델을 만들어 내는데 유용하게 활용할 수 있다.

바. 식별자관계와 비식별자관계 모델링

1) 비식별자관계 선택 프로세스

실제로 프로젝트를 전개할 때 식별자관계와 비식별자관계를 취사선택하여 연결하는 내공은 높은 수준의 기술을 요하고 있다. 특히 식별자관계에서 비식별자관계를 파악하는 기술이 필요한데 다음 흐름(Flow)에 따라 비식별자관계를 선정한다면 합리적으로 관계를 설정하는 모습이 될 수 있다. 기본적으로 식별자관계로 모든 관계가 연결되면서 다음 조건에 해당할 경우 비식별자관계로 조정하면 된다.



[그림 Ⅰ-1-53] 비식별자관계 설정 고려사항

여기에서 가장 중요한 요인은 자식엔터티의 독립된 주식별자 구성이 필요한지를 분석하는 부분이다. 독립적으로 주식별자를 구성한다는 의미는 업무적 필요성과 성능상 필요여부를 모두 포함하는 의미로 이해하면 된다.

2) 식별자와 비식별자관계 비교

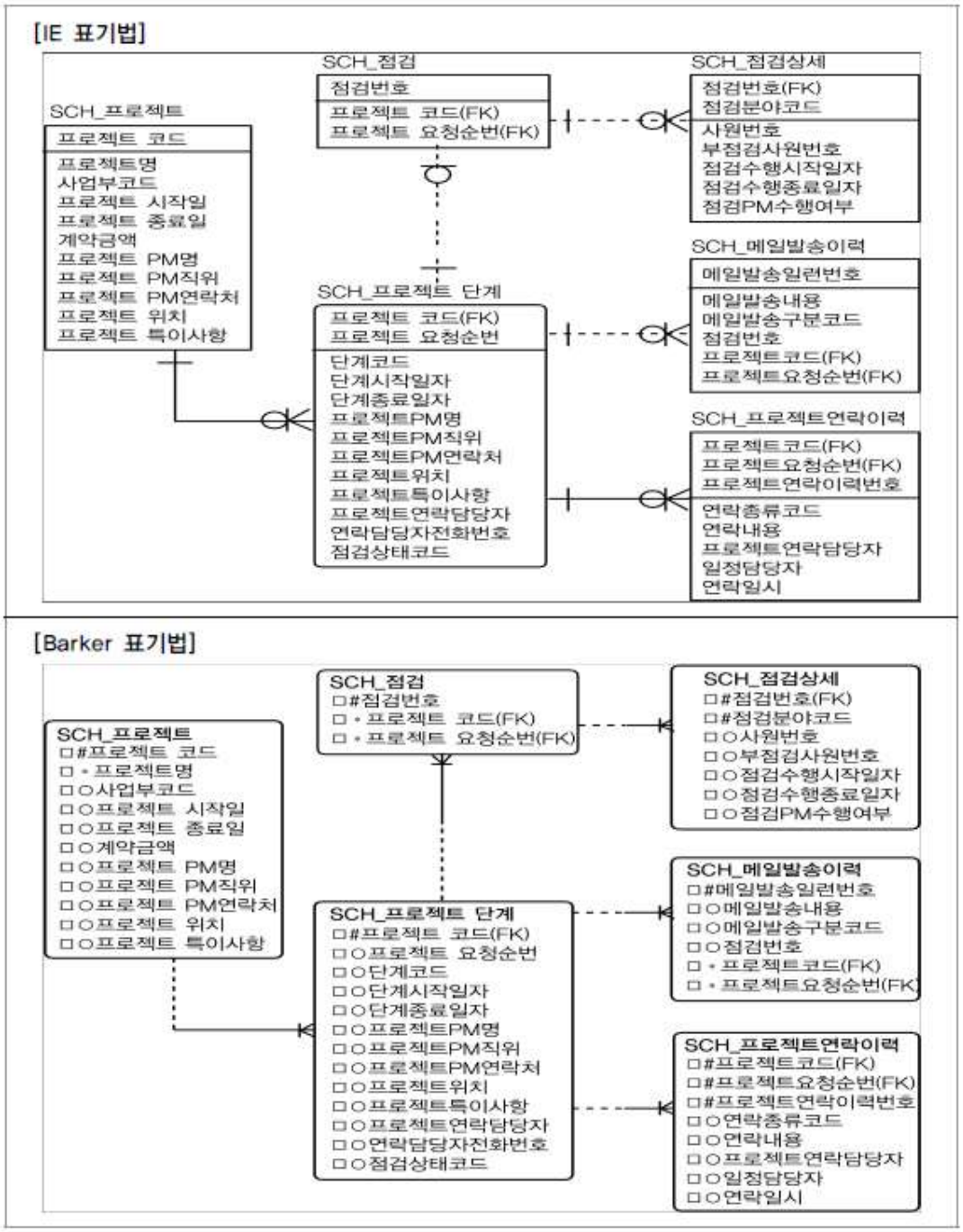
강한 관계인 식별자관계와 약한 관계인 비식별자관계를 비교하면 [표 Ⅰ-1-10]과 같이 나타낼 수 있다.

[표 Ⅰ-1-10] 식별자와 비식별자관계 비교

항목	식별자관계	비식별자관계
목적	강한 연결관계 표현	약한 연결관계 표현
자식 주식별자 영향	자식 주식별자의 구성에 포함됨	자식 일반 속성에 포함됨
표기법	실선 표현	점선 표현
연결 고려사항	- 반드시 부모엔터티 종속 - 자식 주식별자구성에 부모 주식별자포함 필요 - 상속받은 주식별자속성을 타 엔터티에 이전 필요	- 약한 종속관계 - 자식 주식별자구성을 독립적으로 구성 - 자식 주식별자구성에 부모 주식별자 부분 필요 - 상속받은 주식별자속성을 타 엔터티에 차단 필요 - 부모쪽의 관계참여가 선택관계

3) 식별자와 비식별자를 적용한 데이터 모델

이러한 기준에 의해 식별자와 비식별자 관계가 적절하게 설정된 데이터 모델은 [그림 Ⅰ-1-54]의 사례와 같이 균형감 있게 나타난다.



[그림 Ⅰ-1-54] 식별자관계와 비식별자관계의 적절한 선택

상기 모델은 업무의 특성에 따라 식별자관계와 비식별자관계를 적절하게 선택함으로써 데이터 모델의 균형감을 갖추었다고 볼 수 있다.

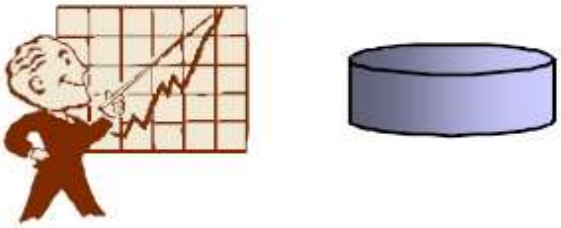
제1절 성능 데이터 모델링의 개요

1. 성능 데이터 모델링의 정의

인프라가 갖추어 지지 않은 환경에서 과연 빠른 속도로 이동할 수 있을까? 길이 굽이굽이 굽어져 있고 곳곳에 신호등이 있는 도로에서 아무리 성능이 좋은 차라고 할지라도 과연 그 길을 빠르게 지날 수 있을까? 데이터베이스에서 기본적으로 설계단계에서부터 성능을 고려하지 않고 설계를 하는 것은 빠르게 지나갈 수 없는 길을 지나가는 차에게 빨리 와달라고 요청하는 것과 다름이 없다.

데이터의 용량의 커질수록 기업의 의사결정의 속도가 빨라질수록 데이터를 처리하는 속도는 빠르게 처리되어야 할 필요성을 반증해 준다. 일반적으로 실무 프로젝트에서 보면 잘못된 테이블 디자인 위에서 개발된 애플리케이션의 성능이 저하되는 경우, 개발자가 구축한 SQL구문에 대해서만 책망을 하는 경우가 많이 있다. 물론 개발자가 SQL구문을 잘못 구성하여 성능이 저하되는 경우도 있지만 근본적으로 디자인이 잘못되어 SQL구문을 잘 못 작성하도록 구성될 수밖에 없는 경우도 빈번하게 발생되고 있음을 기억해야 한다.
성능이 저하되는 데이터 모델의 경우 크게 세 가지 경우를 고려하여 그 성능을 향상시킬 수 있다. 데이터 모델 구조에 의해 성능이 저하될 수도 있고 데이터가 대용량이 됨으로 인해 불가피하게 성능이 저하되어 나타나는 경우도 있다. 또한 인덱스 특성을 충분히 고려하지 않고 인덱스를 생성함으로 인해 성능이 저하되어 나타나는 경우도 있다.

일반적으로 성능이라고 하면 데이터조회의 성능을 의미하곤 한다. 그 이유는 데이터입력/수정/삭제는 일시적이고 빈번하지 않고 단건 처리가 많은 반면 데이터조회의 경우는 반복적이고 빈번하며 여러 건을 처리하는 경우가 많기 때문이다. 이러한 특징은 일반적인 트랜잭션의 성격이 조회의 패턴을 가지고 있다는 것이고 업무에 따라서는 입력/수정/삭제의 성능이 중요한 경우도 있다.
따라서 데이터 모델링을 할 때 어떤 작업 유형에 따라 성능 향상을 도모해야 하는지 목표를 분명하게 해야 정확한 성능향상 모델링을 할 수 있음을 기억해야 한다. 성능 데이터 모델링이란 데이터베이스 성능향상을 목적으로 설계단계의 데이터 모델링 때부터 정규화, 반정규화, 테이블통합, 테이블분할, 조인구조, PK, FK 등 여러 가지 성능과 관련된 사항이 데이터 모델링에 반영될 수 있도록 하는 것으로 정의할 수 있다.



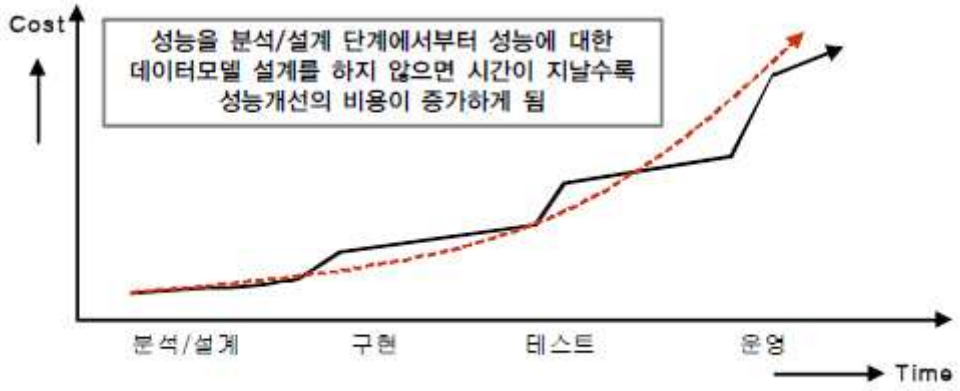
성능 데이터 모델링이란 데이터베이스 성능향상을 목적으로 설계단계의 데이터 모델링 때부터 성능과 관련된 사항이 데이터 모델링에 반영될 수 있도록 하는 것이다.

[그림 Ⅰ-2-1] 성능 데이터 모델링

성능 데이터 모델링이 단순히 반정규화만을 의미하지 않음을 주목해야 한다. 성능데이터 모델링은 정규화를 통해서도 수행할 수 있고 인덱스의 특징을 고려해서 칼럼의 순서도 변형할 수 있다. 또한 대량의 데이터특성에 따라 비록 정규화된 모델이라도 테이블을 수직 또는 수평분할하여 적용하는 방법도 있고 논리적인 테이블을 물리적인 테이블로 전환할 때 데이터 처리의 성격에 따라 변환하는 방법도 성능 데이터 모델링의 범주에 포함될 수 있다.

2. 성능 데이터 모델링 수행시점

성능 향상을 위한 비용은 프로젝트 수행 중에 있어서 사전에 할수록 비용이 들지 않는다. 특히 분석/설계 단계에서 데이터 모델에 성능을 고려한 데이터 모델링을 수행할 경우 성능저하에 따른 재업무(Rework) 비용을 최소화 할 수 있는 기회를 가지게 된다. 분석/설계단계에서 데이터 모델은 대충하고, 성능이 저하되는 SQL문장을 튜닝하고, 부족한 하드웨어 용량(CPU, Memory 등)을 증설하는 등의 작업은 추가적인 비용을 소진하게 하는 원인이 된다. 특히 데이터의 증가가 빠를수록 성능저하에 따른 성능개선비용은 기하급수적으로 증가하게 된다.



[그림 Ⅰ-2-2] 성능 향상 그래프

많은 프로젝트에서 분석/설계단계 때부터 치밀하게 성능에 대비한 설계를 하지 않고, 성능이 저하된 결과만을 대상으로 문제발생 시점에 근시안적인 튜닝을 적용하고 있다. 마치 SQL 튜닝이 모든 것인 것처럼 그것이 마법인 것처럼 SQL문장에만 집중하여 튜닝을 하는 프로젝트의 현장이 아직도 많이 있다.

따라서 분석/설계 단계에서 데이터베이스 처리 성능을 향상시킬 수 있는 방법을 주도면밀하게 고려해야 한다. 만약 어떤 트랜잭션이 해당 비즈니스 처리에 핵심적이고 사용자 업무처리에 있어 중요함을 가지고 있고 성능이 저하되면 안되는 특징을 가지고 있다면, 프로젝트 초기에 운영환경에 대비한 테스트 환경을 구현하고 그곳에 트랜잭션을 발생시켜 실제 성능을 테스트해 보아야 한다. 이 때 데이터 모델의 구조도 변경하면서 어떠한 구조가 해당 사이트에 성능상 가장 적절한 구조인지를 검토하여 성능이 좋은 모습으로 디자인하는 전략이 요구된다.

3. 성능 데이터 모델링 고려사항

일반적으로 성능 데이터 모델은 다음과 같은 프로세스로 진행하는 것이 데이터 모델링 단계에서 성능을 충분히 고려할 수 있는 방안이 된다.

① 데이터 모델링을 할 때 정규화를 정확하게 수행한다.

② 데이터베이스 용량산정을 수행한다.

③ 데이터베이스에 발생되는 트랜잭션의 유형을 파악한다.

④ 용량과 트랜잭션의 유형에 따라 반정규화를 수행한다.

⑤ 이력모델의 조정, PK/FK조정, 슈퍼타입/서브타입 조정 등을 수행한다.

⑥ 성능관점에서 데이터 모델을 검증한다.

데이터 모델링을 할 때 기본적으로 정규화를 완벽하게 수행해야 한다. 정규화된 모델이 데이터를 주요 관심사별로 분산시키는 효과가 있기 때문에 그 자체로 성능을 향상시키는 효과가 있다. 일단 정규화가 완성된 모델에 대해서 해당 데이터 모델의 각각의 엔터티에 어느 정도 트랜잭션이 들어오는지 살펴볼 필요가 있다. 이 때 가장 좋은 방법이 엔터티에 대한 용량산정을 하는 것이다. 각각의 엔터티(테이블)에 대한 용량산정을 수행하면 어떤 엔터티(테이블)에 데이터가 집중되는지 파악할 수 있다. 이 용량산정은 엔터티별로 데이터가 대용량인지를 구분하게 하기 때문에 테이블에 대한 성능고려를 엄격하게 적용해야 하는지 기준이 될 수 있다.

또한 데이터 모델에 발생되는 트랜잭션의 유형을 파악할 필요가 있다. 트랜잭션의 유형에 대한 파악은 CRUD 매트릭스를 보고 파악하는 것도 좋은 방법이 될 수 있고 객체지향 모델링을 적용한다면 시퀀스 다이어그램을 보면 트랜잭션의 유형을 파악하기에 용이하다. 또한 화면에서 처리된 데이터의 종류들을 보면 이벤트(입력, 수정, 삭제, 조회)에 따라 테이블에 데이터가 어떻게 처리되는지를 유추할 수 있다. 트랜잭션의 유형을 파악하게 되면 SQL문장의 조인관계 테이블에서 데이터조회의 칼럼들을 파악할 수 있게 되어 그에 따라 성능을 고려한 데이터 모델을 설계할 수 있다.

이렇게 파악된 용량산정과 트랜잭션의 유형데이터를 근거로 정확하게 테이블에 대해 반정규화를 적용하도록 한다. 반정규화는 테이블, 속성, 관계에 대해 포괄적인 반정규화의 방법을 적용해야 한다. 또한 대량 데이터가 처리되는 이력모델에 대해 성능고려를 하고 PK/FK의 순서가 인덱스 특성에 따라 성능에 영향을 미치는 영향도가 크기 때문에 반드시 PK/FK를 성능이 우수한 순서대로 칼럼의 순서를 조정해야 한다.

전체적으로 성능에 대한 충분한 고려가 되었는지를 데이터 모델 검토를 통해 다시 한 번 확인하도록 한다. 데이터 모델 검토 시에 일반적인 데이터 모델 규칙만을 검증하지 말고 충분하게 성능이 고려되었는지를 체크리스트에 포함하여 검증하도록 한다.

제2절 정규화와 성능

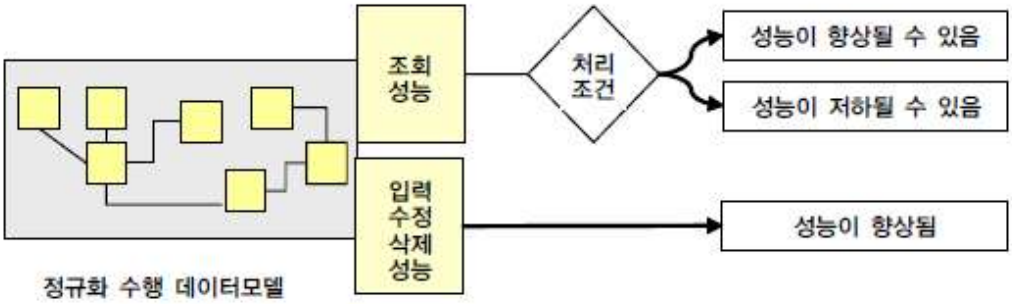
1. 정규화를 통한 성능 향상 전략

데이터 모델링을 하면서 정규화를 하는 것은 기본적으로 데이터에 대한 중복성을 제거하여 주고 데이터가 관심사별로 처리되는 경우가 많기 때문에 성능이 향상되는 특징을 가지고 있다. 물론 엔터티가 계속 발생되므로 SQL문장에서 조인이 많이 발생하여 이로 인한 성능 저하가 나타나는 경우도 있지만 이런 부분은 사례별로 유의하여 반정규화를 적용하는 전략이 필요하다.

정규화를 수행하면 항상 조회 성능이 저하되어 나타날까?

데이터처리의 성능이 무엇인지 정확히 구분하여 인식할 필요가 있다. 데이터베이스에서 데이터를 처리할 때 성능이라고 하면 조회 성능과 입력/수정/삭제 성능의 두 부류로 구분된다. 이 두 가지 성능이 모두 우수하면 좋겠지만 데이터 모델을 구성하는 방식에 따라 두 성능이 Trade-Off 되어 나타나는 경우가 많이 있다.

정규화를 수행한다는 것은 데이터를 결정하는 결정자에 의해 함수적 종속을 가지고 있는 일반속성을 의존자로 하여 입력/수정/삭제 이상을 제거하는 것이다. 데이터의 중복속성을 제거하고 결정자에 의해 동일한 의미의 일반속성이 하나의 테이블로 집약되므로 한 테이블의 데이터 용량이 최소화되는 효과가 있다. 따라서 정규화된 테이블은 데이터를 처리할 때 속도가 빨라질 수도 있고 느려질 수도 있는 특성이 있다.



일반적으로 정규화를 수행해야 데이터처리의 성능이 향상되며
데이터의 조회처리 트랜잭션시에 성능저하가 나타날 수 있음

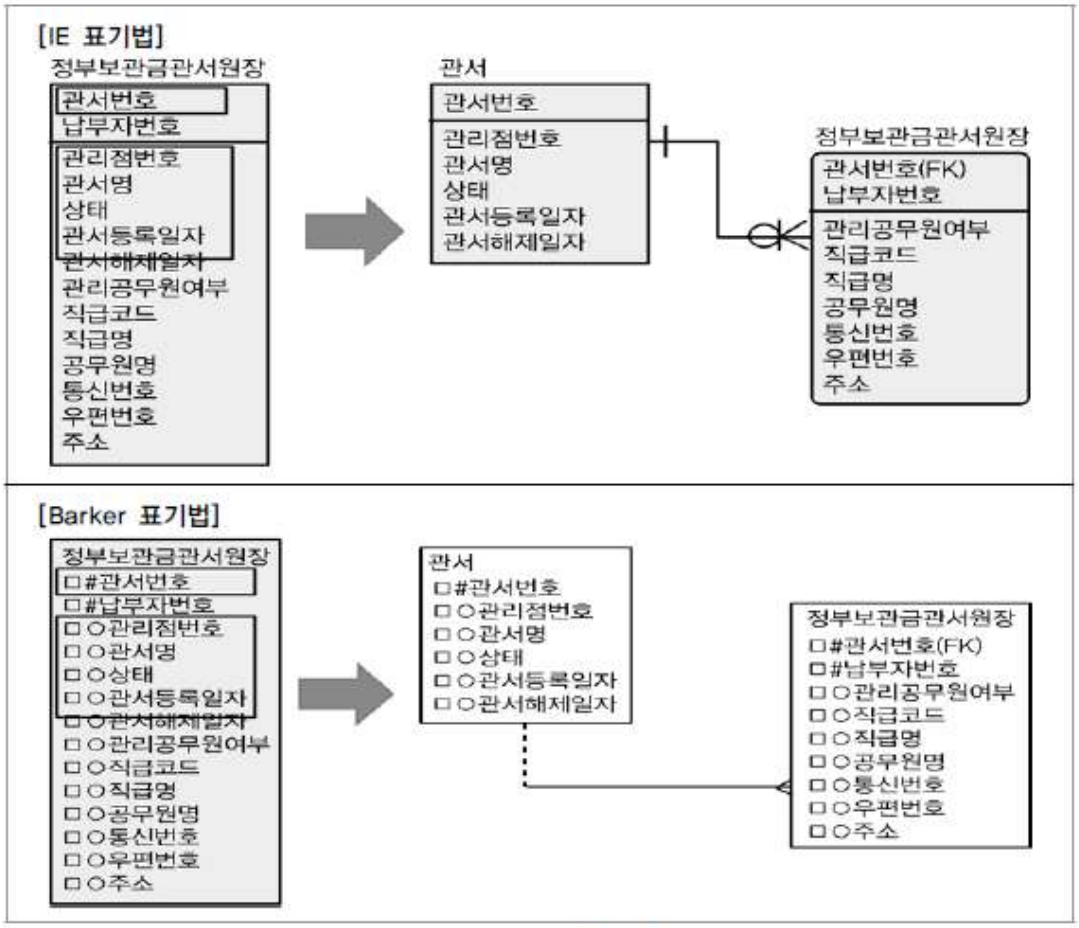
[그림 Ⅰ-2-3] 정규화 수행과 성능

[그림 Ⅰ-2-3]을 보면 정규화 수행 모델은 데이터를 입력/수정/삭제할 때 일반적으로 반정규화된 테이블에 비해 처리 성능이 향상된다. 단 데이터를 조회할 때에는 처리 조건에 따라 조회 성능이 향상될 수도 있고 저하될 수도 있다.

따라서 일반적으로 정규화가 잘 되어 있으면 입력/수정/삭제의 성능이 향상되고 반정규화를 많이 하면 조회의 성능이 향상된다고 인식될 수 있다. 그러나 데이터 모델링을 할 때 반정규화만이 조회 성능을 향상시킨다는 고정관념은 탈피되어야 한다. 정규화를 해서 성능이 저하되기는커녕 정규화를 해야만 성능이 향상되는 경우가 아주 많이 나타나기 때문이다.

2. 반정규화된 테이블의 성능저하 사례1

정규화하여 조인이 발생하면 성능이 심각하게 저하되는가? 다음 예를 살펴보면 2차 정규화를 적용한 테이블에 대해서 조인을 하더라도 PK Unique Index를 이용하면 조인 성능 저하는 미미하게 발생된다.



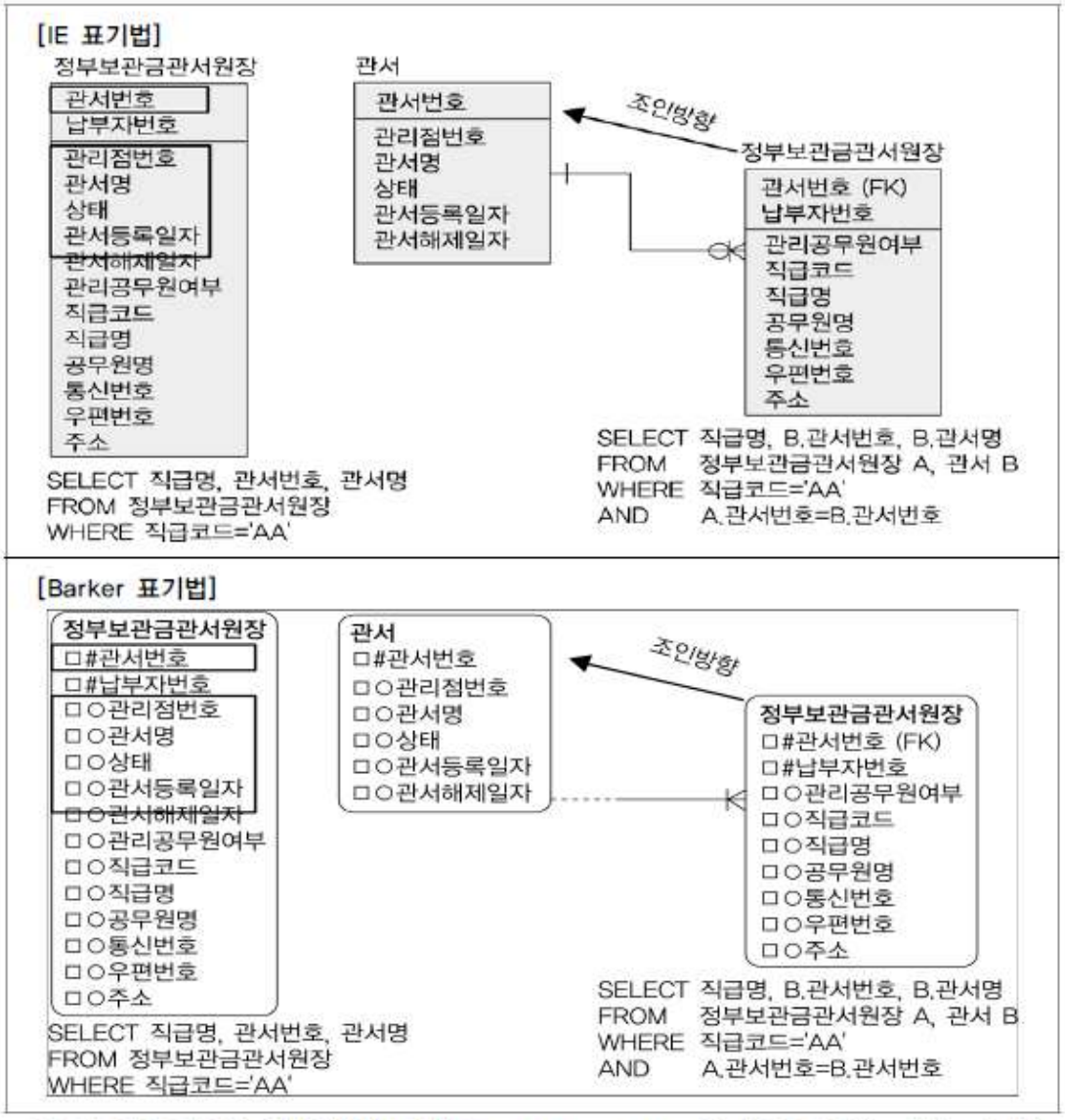
[그림 Ⅰ-2-4] 정규화

[그림 Ⅰ-2-4]의 왼쪽 그림은 2차 정규화가 안 된 반정규화된 테이블의 모습이고 오른쪽 그림은 부분키 종속을 정규화하여 두 개의 테이블로 분리해 2차 정규화된 테이블의 모습이다.

2차 정규화가 안 된 테이블은 직급명과 함께 반정규화된 관서번호, 관서명을 조회하면 하나의 테이블에서 데이터가 조회가 된다. 2차 정규화된 테이블은 관서번호, 관서명이 관서 테이블에만 존재하기 때문에 두 개의 테이블을 조인하여 처리해야 한다.

정부보관금관서원장에서 데이터를 조회하는 것이나, 관서와 정부보관금관서원장을 조인하여 데이터를 조회하나 처리 성능은 사용자가 느끼기에는 거의 차이가 나지 않는다. PK가 걸려있는 방향으로 조인이 걸려 Unique Index를 곧바로 찾아서 데이터를 조회하기 때문에, 하나의 테이블에서 조회하는 작업과 비교했을 때 미미하게 성능 차이가 날 뿐 사용자에게

크게 영향을 줄 만큼 성능이 저하되는 일은 없는 것이다.
게다가 위의 예를 '관서등록일자가 2010년 이후 관서를 모두 조회하라'는 SQL 구문을 처리하는 것으로 바꾸면, 2차 정규화된 테이블이 훨씬 빠르다. [그림 Ⅰ-2-5]에서와 같이 왼쪽 테이블에서는 불필요하게 납부자번호만큼 누적된 데이터를 읽어서 결과를 구분하여 보여주어야 하지만 오른쪽은 관서수만큼만 존재하는 데이터를 읽어 곧바로 결과를 보여주기 때문이다. 이렇게 단순한 예만 보아도 정규화를 수행하면 무조건 조회성능이 저하된다는 고정관념이 틀렸다는 것을 알 수 있다.

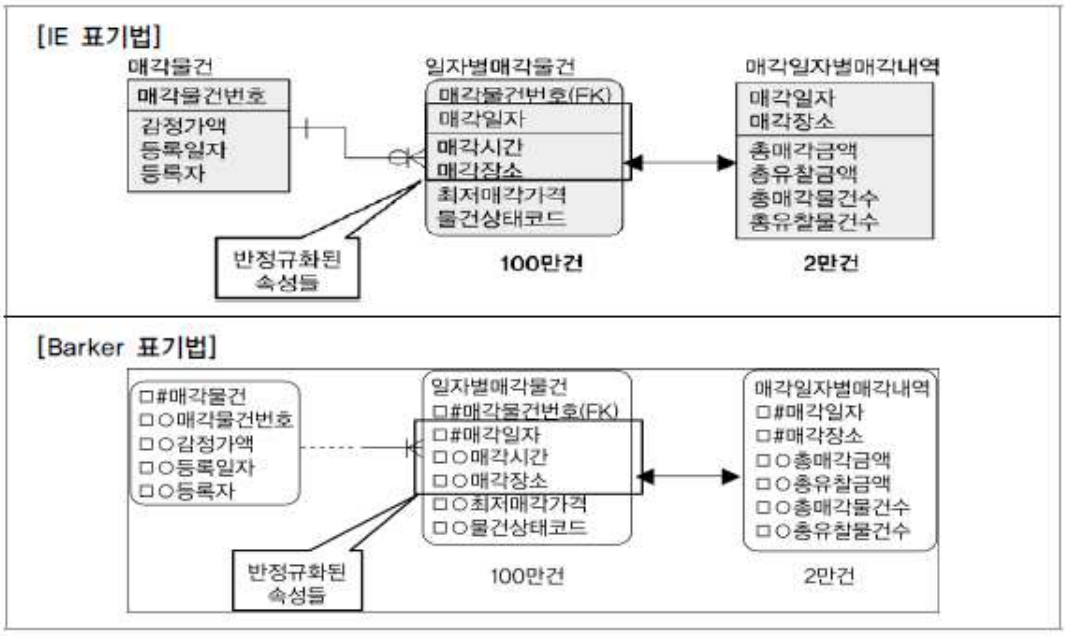


2차 정규화를 적용한 테이블에서 조회해도 PK Unique Index를 이용 조인성능저하는 미미함

[그림 Ⅰ-2-5] 정규화와 SQL

3. 반정규화된 테이블의 성능저하 사례2

두 개의 엔터티가 통합되어 반정규화된 또 다른 경우를 살펴본다. 이 업무에서는 어떤 물건을 매각할 때 매각일자를 정하고 그 일자에 해당하는 매각시간과 매각장소가 결정하는 속성의 성격을 가지고 있다. 즉 매각일자가 결정자가 되고 매각시간과 매각장소가 의존자가 되는 함수적 종속관계가 형성되는 관계이다. 매각일자는 5천 건이 있고 일자별매각물건은 100만 건이 있는 것으로 가정하자.



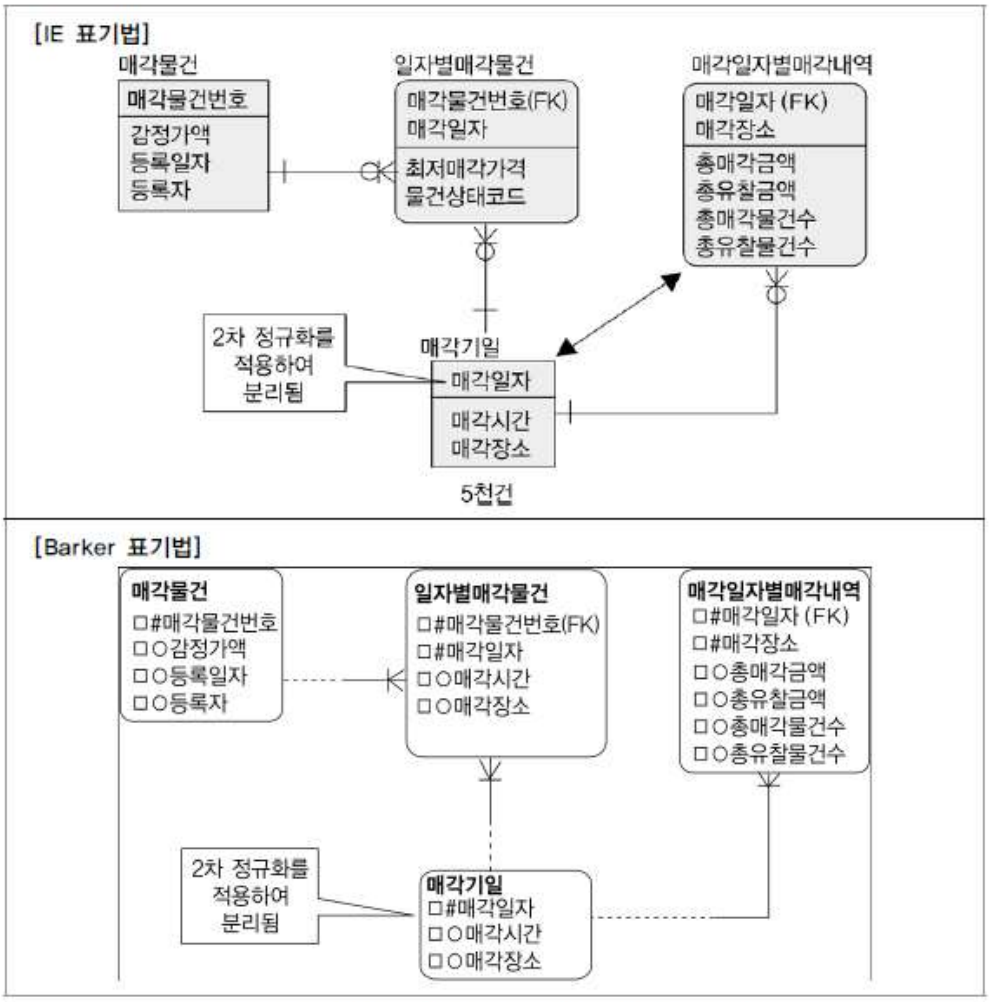
특정 매각장소에 대해 매각일자를 찾아 매각내역을 조회하려면
100만 건의 데이터를 읽어 매각일자를 DISTINCT하여 매각일자별매각내역과 조인이 된다.

[그림 Ⅰ-2-6] 반정규화된 데이터 모델

예를 들어 [그림 Ⅰ-2-6]의 모델에서 '서울 7호'에서 매각된 총매각금액, 총유찰금액을 산출하는 조회용 SQL문장을 작성하면 다음과 같다.

```
SELECT B.총매각금액 , B.총유찰금액
  FROM (SELECT DISTINCT 매각일자
          FROM 일자별매각물건
         WHERE 매각장소 = '서울 7호') A, <== 100만건의 데이터를 읽어 DISTINCT함
         매각일자별매각내역 B
 WHERE A.매각일자 = B.매각일자
    AND A.매각장소 = B.매각장소;
```

대량의 데이터에서 조인 조건이 되는 대상을 찾기 위해 인라인뷰를 사용하기 때문에 성능이 저하된다. 이를 정규화하려면 복합식별자 중에서 일반속성이 주식별자 속성 중 일부에만 종속관계를 가지고 있으므로 2차 정규화의 대상이 된다. 2차 정규화를 적용하면 [그림 Ⅰ-2-7]과 같은 모델이 된다.



특정 매각장소에 대해 매각일자를 찾아 매각내역을 조회하려면
500건의 매각기일과 매각일자별매각내역과 조인이 된다.

[그림 Ⅰ-2-7] 정규화된 데이터 모델

2차 정규화를 적용하여 매각일자를 PK로 하고 매각시간과 매각장소는 일반속성이 되었다. 정규화를 적용했기 때문에 매각일자를 PK로 사용하는 매각일자별매각내역과도 관계가 연결된다. 따라서 업무흐름에 따른 적확한 데이터 모델링 표기도 가능해지고, 드라이빙이 된 테이블 이 5천 건의 매각기일 테이블이 되므로 성능도 향상된다.

만약 위의 모델에서 '서울 7호'에서 매각된 총매각금액, 총유찰금액을 산출하는 조회용 SQL 문장을 작성하면 다음과 같이 나온다.

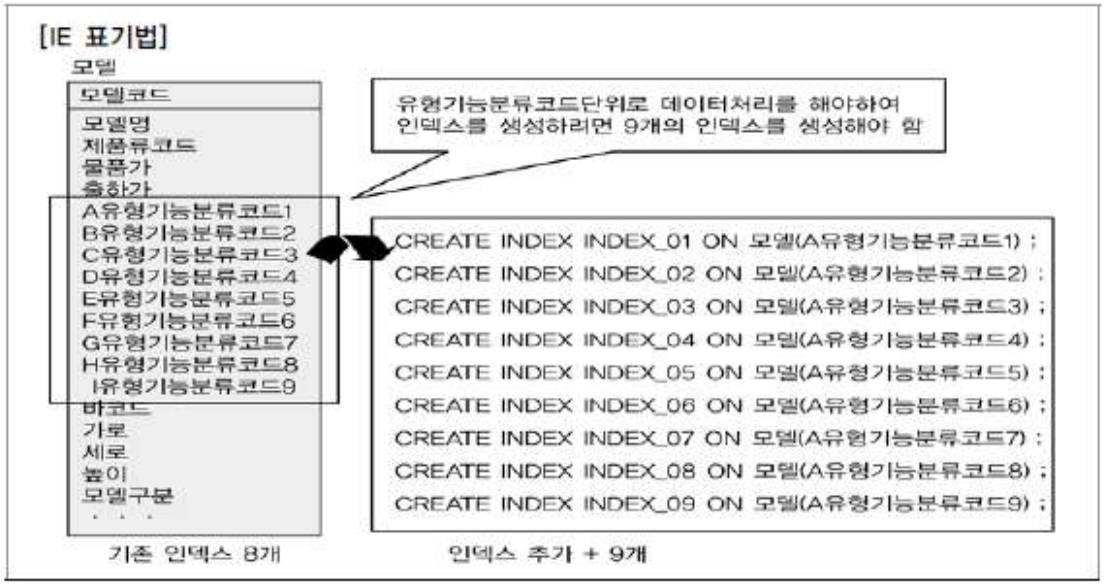
```
SELECT B.총매각금액 , B.총유찰금액
  FROM 매각기일 A,
       매각일자별매각내역 B
 WHERE A.매각장소 = '서울 7호' <== 5천건의 데이터를 읽음
   AND A.매각일자 = B.매각일자 AND A.매각장소 = B.매각장소;
```

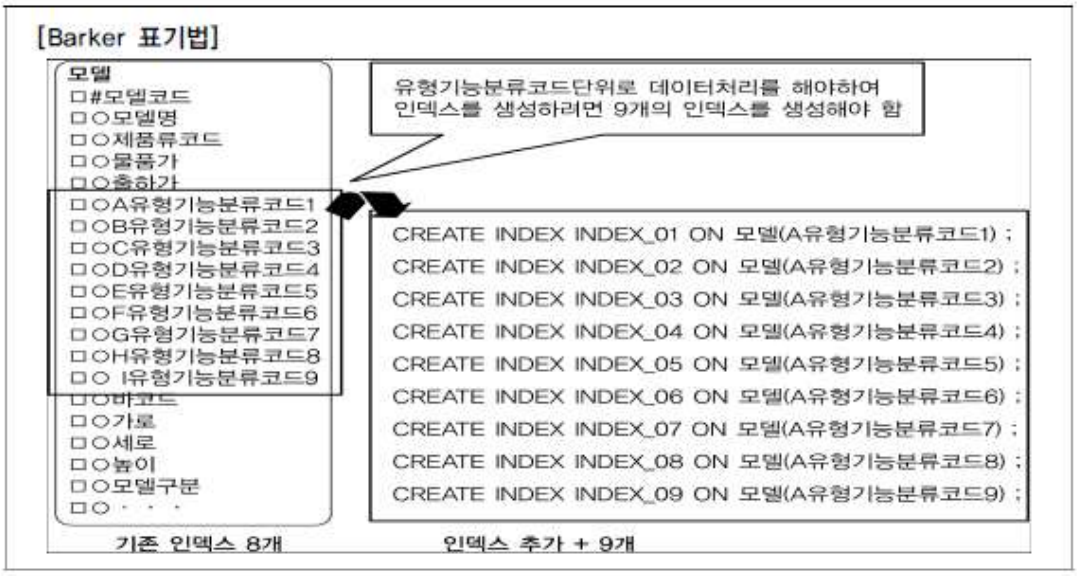
매각기일 테이블이 정규화 되면서 드라이빙이 되는 대상 테이블의 데이터가 5천 건으로 줄어들어 조회 처리가 빨라진다.

4. 반정규화된 테이블의 성능저하 사례3

다음 사례는 동일한 속성 형식을 두 개 이상의 속성으로 나열하여 반정규화한 경우에 해당한다. 계층형 데이터베이스를 많이 사용했던 과거 데이터 모델링의 습관이 남아서인지 관계형 데이터베이스에서도 동일한 속성을 한 테이블에 속성1, 속성2, 속성3 데이터 모델링을 하는 경우가 많이 있다.

[그림 Ⅰ-2-8]을 보면, 모델이라고 하는 테이블에 업무적으로 필요한 8개의 인덱스가 이미 생성되어 있다. 데이터는 30만 건이고 온라인 환경의 데이터베이스라고 가정하자. 유형기능분류코드에 따라 데이터를 조회하는 경우가 많이 나타나 인덱스를 생성하려면 유형기능분류코드 각각에 대해 인덱스를 생성해야 하므로 9개나 되는 인덱스를 추가 생성해야 한다.





[그림 Ⅰ-2-8] 반정규화된 데이터 모델

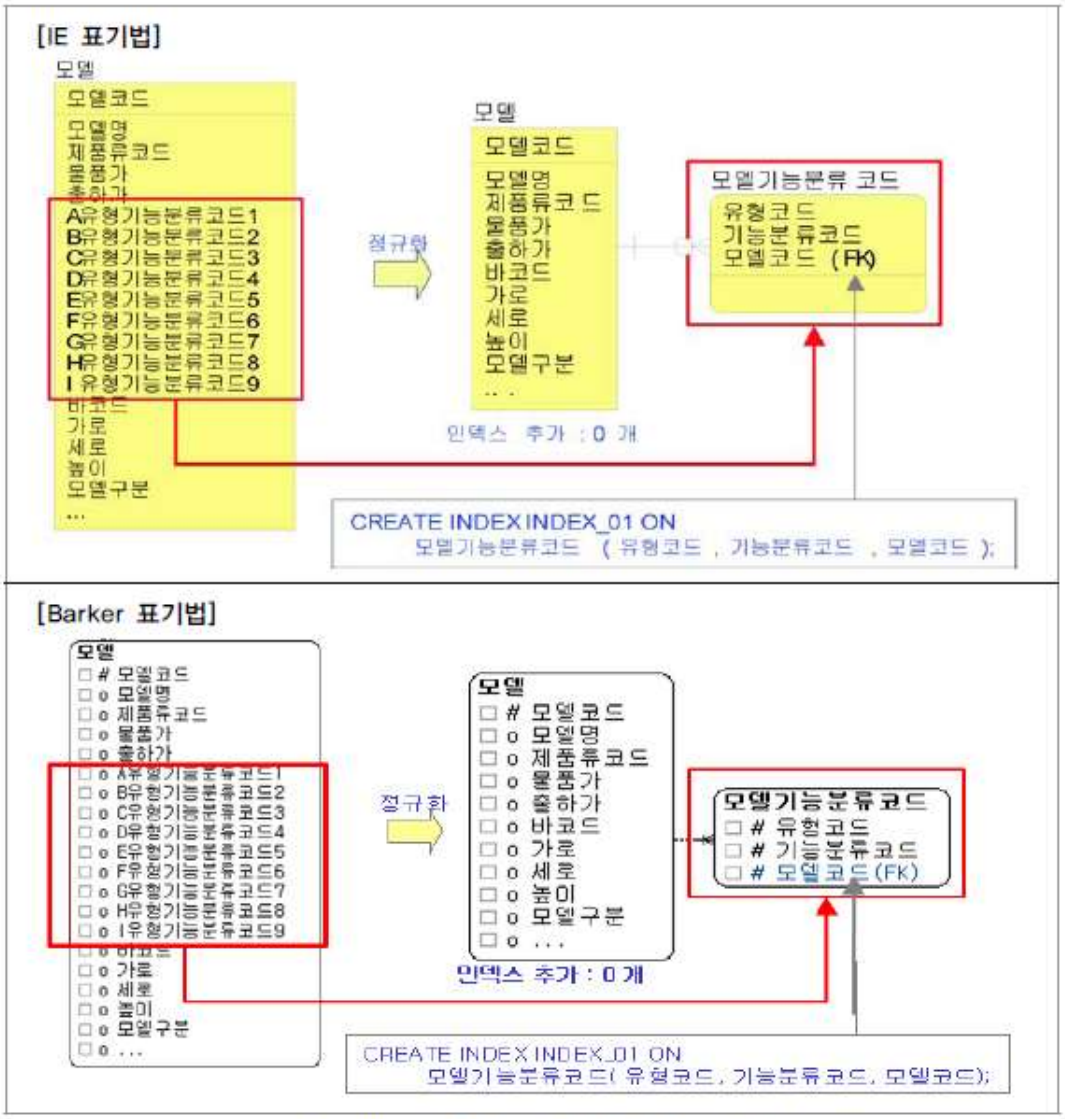
참고로, 한 테이블에 인덱스가 많아지면 조회 성능은 향상되지만 데이터 입력/수정/삭제 성능은 저하된다. 그래서 일반 업무처리(온라인성 업무)에서는 인덱스 수를 가급적 7~8개가 넘지 않도록 하는 것이 좋다. 그런데 [그림 Ⅰ-2-8]의 모델은 다른 필요한 인덱스 이외에 유형기능분류코드 속성에 해당하는 인덱스를 9개나 추가적으로 생성해야 하므로 실전프로젝트에서는 어쩔 수 없이 인덱스를 생성하지 않거나 A유형기능분류코드1 하나만 인덱스를 생성하는 경우가 생긴다. 이에 따라 A유형기능분류코드1, A유형기능분류코드2, A유형기능분류코드3...을 이용하면 SQL의 성능이 저하되는 경우가 많다.

만약 각 유형코드별로 조건을 부여하여 모델코드와 모델명을 조회하는 SQL문장을 작성한다면 다음과 같이 나온다.

```
SELECT 모델코드, 모델명
  FROM 모델
 WHERE ( A유형기능분류코드1 = '01' )
       OR ( B유형기능분류코드2 = '02' )
       OR ( C유형기능분류코드3 = '07' )
       OR ( D유형기능분류코드4 = '01' )
       OR ( E유형기능분류코드5 = '02' )
       OR ( F유형기능분류코드6 = '07' )
       OR ( G유형기능분류코드7 = '03' )
       OR ( H유형기능분류코드8 = '09' )
       OR ( I유형기능분류코드9 = '09' )
```

각 유형별로 모두 인덱스가 걸려 있어야 인덱스에 의해 데이터를 찾을 수 있다. 이런 모델은 다음과 같이 정규화를 적용해야 한다.

중복속성에 대한 분리가 1차 정규화의 정의임을 고려하면 모델 테이블은 1차 정규화의 대상이 된다. 로우단위의 대상도 1차 정규화의 대상이 되지만 칼럼 단위로 중복이 되는 경우도 1차 정규화의 대상이 된다. 따라서 모델에 대해 1차 정규화를 적용하면 [그림 Ⅰ-2-9]와 같이 분리될 수 있다.



[그림 Ⅰ-2-9] 정규화된 데이터 모델

하나의 테이블에 9개가 반복적으로 나열이 되어 있을 때는 인덱스 생성이 어려웠지만 정규화되어 분리된 이후에는 인덱스 추가 생성이 0개가 되었다. 또한 분리된 테이블 모델기능분류코드에서 PK인덱스를 생성하여 이용함으로써 성능이 향상될 수 있다.
만약 각 유형코드별로 조건을 부여하여 모델코드와 모델명을 조회하는 SQL문장을 작성한다면 다음과 같이 작성된다.

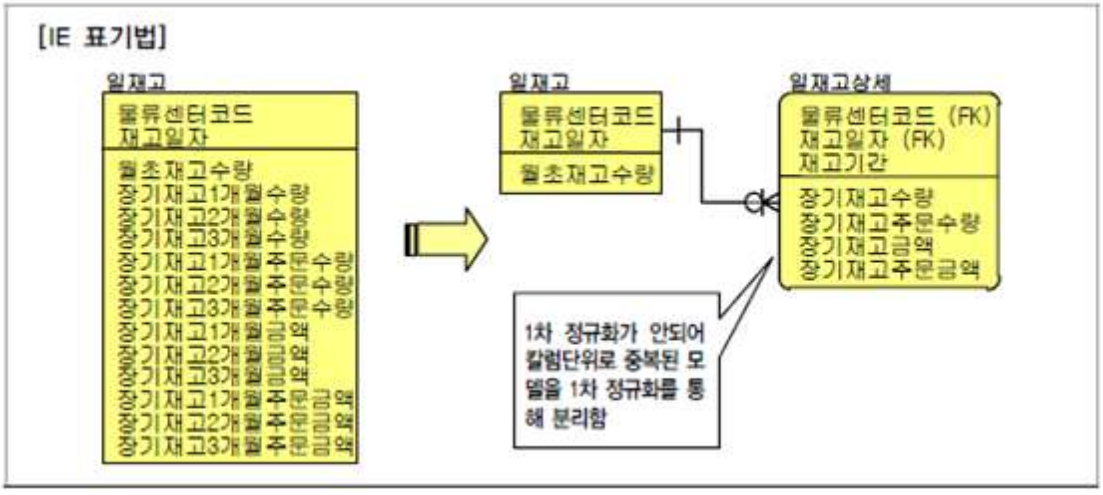
```
SELECT A.모델코드, A.모델명
  FROM 모델 A, 모델기능분류코드 B
 WHERE ( B.유형코드 = 'A' AND B.기능분류코드 = '01' AND A.모델코드 = B.모델코드 )
     OR ( B.유형코드 = 'B' AND B.기능분류코드 = '02' AND A.모델코드 = B.모델코드 )
     OR ( B.유형코드 = 'C' AND B.기능분류코드 = '07' AND A.모델코드 = B.모델코드 )
     OR ( B.유형코드 = 'D' AND B.기능분류코드 = '01' AND A.모델코드 = B.모델코드 )
     OR ( B.유형코드 = 'E' AND B.기능분류코드 = '02' AND A.모델코드 = B.모델코드 )
     OR ( B.유형코드 = 'F' AND B.기능분류코드 = '07' AND A.모델코드 = B.모델코드 )
     OR ( B.유형코드 = 'G' AND B.기능분류코드 = '03' AND A.모델코드 = B.모델코드 )
     OR ( B.유형코드 = 'H' AND B.기능분류코드 = '09' AND A.모델코드 = B.모델코드 )
     OR ( B.유형코드 = 'I' AND B.기능분류코드 = '09' AND A.모델코드 = B.모델코드 )
```

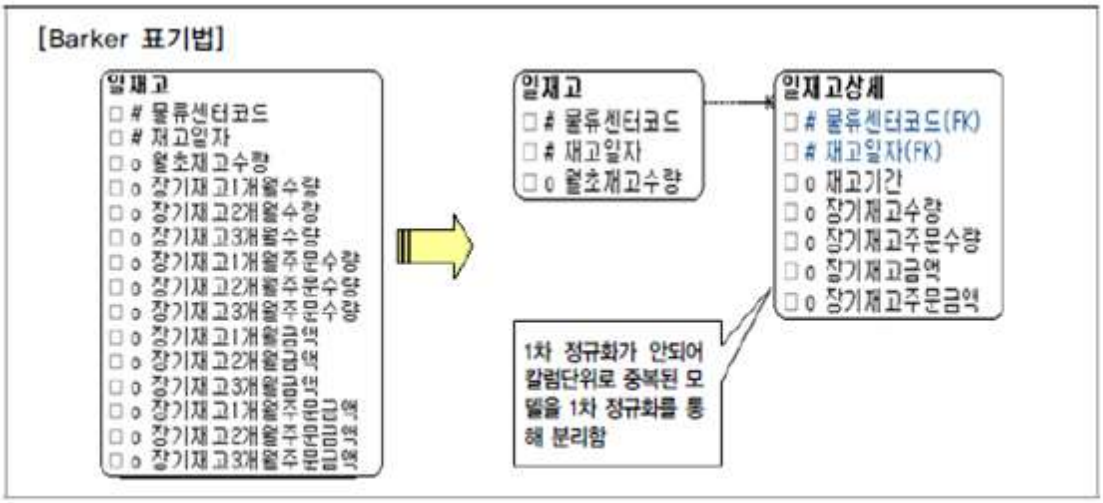
위 SQL구문은 유형코드+기능분류코드+모델코드에 인덱스가 걸려 있으므로 인덱스를 통해 데이터를 조회함으로써 성능이 향상된다.

실전 프로젝트에서도 많은 데이터 모델이 칼럼 단위에서 중복된 경우가 발견된다. 이에 대한 파급효과 계산 없이 무조건 칼럼 단위로 COL1, COL2, COL3... 이런 식으로 데이터 모델링을 하는 것은 근본적으로 SQL문장의 성능을 나쁘게 하는 결과를 초래할 수도 있으므로 인덱스 생성 영향도를 파악한 이후에 적용하는 것이 좋은 방법이 된다

5. 반정규화된 테이블의 성능저하 사례4

[그림 Ⅰ-2-10]과 같은 경우도 동일한 사례이다.



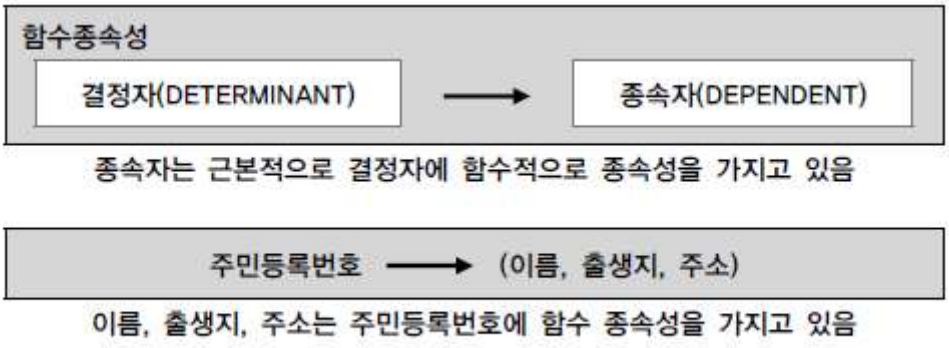


[그림 Ⅰ-2-10] 반정규화된 테이블의 정규화

일재고와 일재고 상세를 구분함으로써 일재고에 발생되는 트랜잭션의 성능저하를 예방할 수 있게 되었다.

6. 함수적 종속성(Functional Dependency)에 근거한 정규화 수행 필요

함수의 종속성(Functional Dependency)은 데이터들이 어떤 기준값에 의해 종속되는 현상을 지칭하는 것이다. 이 때 기준값을 결정자(Determinant)라 하고 종속되는 값을 종속자(Dependent)라고 한다.



[그림 Ⅰ-2-11] 함수의 종속성

[그림 Ⅰ-2-11]에서 보면 사람이라는 엔터티는 주민등록번호, 이름, 출생지, 호주라는 속성이 존재한다. 여기에서 이름, 출생지, 호주라는 속성은 주민등록번호 속성에 종속된다. 만약 어떤 사람의 주민등록번호가 신고되면 그 사람의 이름, 출생지, 호주가 생성되어 단지 하나의 값만을 가지게 된다. 이를 기호로 표시하면, 다음과 같이 표현할 수 있다.

주민등록번호 -> (이름, 출생지, 호주)

즉 “주민등록번호가 이름, 출생지, 호주를 함수적으로 결정한다.”라고 말할 수 있다. 실세계의 데이터들은 대부분 이러한 함수 종속성을 가지고 있다.

함수의 종속성은 데이터가 가지고 있는 근본적인 속성으로 인식되고 있다. 정규화의 궁극적인 목적은 반복적인 데이터를 분리하고 각 데이터가 종속된 테이블에 적절하게(프로세스에 의해 데이터의 정합성이 지켜질 수 있어야 함) 배치되도록 하는 것이므로 이 함수의 종속성을 이용하여 정규화 작업이나 각 오브젝트에 속성을 배치하는 작업에 이용이 되는 것이다.

기본적으로 데이터는 속성간의 함수종속성에 근거하여 정규화되어야 한다. 프로젝트 수행에서 정규화는 선택사항이 아니라 필수사항이다. 데이터의 구조를 함수적 종속관계에 의해 정규화 사상에 맞게 분리하는 것은 마치 건축물을 지을 때 부실공사를 방지하기 위해 하중, 균형, 내구성을 고려한 기본적인 설계를 하는 것과 같다. 정규화 이론은 건축으로 따지면 기본적인 설계 원칙만큼이나 중요한 이론이다.

프로젝트에서 정규화가 적용된 모델을 설계하기 위해서는 정규화에 대한 기본이론을 정확하고 구체적으로 이해하고 있어야 한다. 전문적인 IT프로젝트를 진행하는 설계자가 정규화의 이론을 모르고 데이터 모델링을 설계하는 것은 불가능하다. 눈이 어두운 사람이 감각에 의해 길을 찾아가는 것과 같다. 분명하게 자기가 가는 길을 명확하게 검증받을 수 있도록, 지도를 가지고 길을 가듯이 정규화의 이론을 숙지하고 정보시스템의 근간인 데이터를 설계해야 한다.

제3절 반정규화와 성능

1. 반정규화를 통한 성능향상 전략

가. 반정규화의 정의

반정규화(=역정규화) 용어는 조금 다르게 표현되어도 그 의미는 동일하다. 여기에서 반정규화는 '반(Half)'의 의미가 아닌 한자로 반대하다의 의미를 가진 '反'의 의미이다. 영어로는 De-Normalization이다. 비정규화는 아예 정규화를 수행하지 않은 모델을 지칭할 때 사용한다.

반정규화를 정의하면 정규화된 엔터티, 속성, 관계에 대해 시스템의 성능향상과 개발(Development)과 운영(Maintenance)의 단순화를 위해 중복, 통합, 분리 등을 수행하는 데이터 모델링의 기법을 의미한다. 협의의 반정규화는 데이터를 중복하여 성능을 향상시키기 위한 기법이라고 정의할 수 있고 좀 더 넓은 의미의 반정규화는 성능을 향상시키기 위해 정규화된 데이터 모델에서 중복, 통합, 분리 등을 수행하는 모든 과정을 의미한다.

데이터 무결성이 깨질 수 있는 위험을 무릅쓰고 데이터를 중복하여 반정규화를 적용하는 이유는 데이터를 조회할 때 디스크 I/O량이 많아서 성능이 저하되거나 경로가 너무 멀어 조인으로 인한 성능저하가 예상되거나 칼럼을 계산하여 읽을 때 성능이 저하될 것이 예상되는 경우 반정규화를 수행하게 된다.

중복성의 원리를 활용하여 데이터조회시 성능을 향상시키는 역할을 할 수 있음

[그림 Ⅰ-2-12] 반정규화

기본적으로 정규화는 입력/수정/삭제에 대한 성능을 향상시킬 뿐만 아니라 조회에 대해서도 성능을 향상시키는 역할을 한다. 그러나 정규화만을 수행하면 엔터티의 갯수가 증가하고 관계가 많아져 일부 여러 개의 조인이 걸려야만 데이터를 가져오는 경우가 있다. 이러한 경우 업무적으로 조회에 대한 처리성능이 중요하다고 판단될 때 부분적으로 반정규화를 고려하게 되는 것이다. 또한 정규화의 함수적 종속관계는 위반하지 않지만 데이터의 중복성을 증가시켜야만 데이터조회의 성능을 향상시키는 경우가 있다. 이러한 경우 반정규화를 통해서 성능을 향상시킬 수 있게 되는 것이다.

프로젝트에서는 설계단계에서 반정규화를 적용하게 되는데 반정규화를 기술적으로 수행하지 않는 경우에는 다음과 같은 현상이 발생된다.

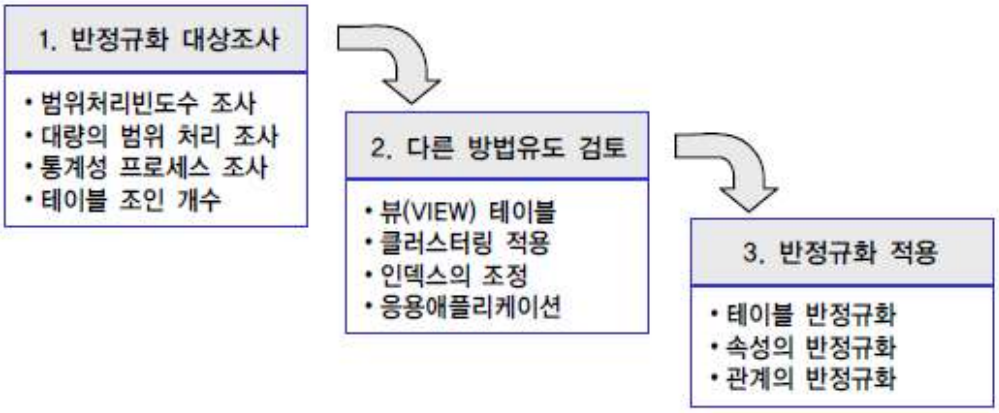
- 성능이 저하된 데이터베이스가 생성될 수 있다.
- 구축단계나 시험단계에서 반정규화를 적용할 때 수정에 따른 노력비용이 많이 들게 된다.

나. 반정규화의 적용방법

반정규화도 하나의 난이도 높은 데이터 모델링의 실무기술이다. 보통 프로젝트에서는 칼럼 중복을 통해서만 반정규화를 수행하게 된다. 칼럼의 반정규화가 많은 이유는 개발을 하다가 SQL문장 작성이 복잡해지고 그에 따라 SQL단위 성능 저하가 예상이 되어 다른 테이블에서 조인하여 가져와야 할 칼럼을 기준이 되는 테이블에 중복하여 SQL문장을 단순하게 처리하도록 하기 위해 요청하는 경우가 많다. 이 때문에 칼럼의 반정규화 유형이 많이 나타나게 된다. 이렇게 무분별하게 칼럼의 반정규화를 많이 하게 되는 것은 데이터에 대한 무결성을 깨뜨리는 결정적인 역할을 하는 경우가 많이 있다.

반정규화에 대한 필요성이 결정이 되면 칼럼의 반정규화 뿐만 아니라 테이블의 반정규화와 관계의 반정규화를 종합적으로 고려하여 적용해야 한다. 또한 반정규화를 막연하게 중복을 유도하는 것만을 수행하기 보다는 성능을 향상시킬 수 있는 다른 방법들을 고려하고 그 이후에 반정규화를 적용하도록 해야 한다.

반정규화를 적용할 때는 기본적으로 데이터 무결성이 깨질 가능성이 많이 있기 때문에 반드시 데이터 무결성을 보장할 수 있는 방법을 고려한 이후에 반정규화를 적용하도록 해야 한다. 정규화와 반정규화 사이에는 Trade-Off 관계 즉, 마치 저울추가 양쪽에 존재하여 한쪽이 무거워지면 다른 쪽은 위로 올라가는 것처럼 정규화만을 강조하다 보면 성능의 이슈가 발생될 수 있고 반정규화를 과도하게 적용하다 보면 데이터 무결성이 깨질 수 있는 위험이 증가하게 되는 것이다. 따라서 반정규화를 적용할 때에는 데이터 무결성이 중요함을 알고 데이터 무결성이 충분히 유지될 수 있도록 프로세스 처리에 있어서 안정성이 먼저 확인이 되어야 한다.



[그림 Ⅰ-2-13] 반정규화 절차

반정규화를 적용하기 위해서는 [그림 Ⅰ-2-13]에 나타난 것처럼 먼저 반정규화의 대상을 조사하고 다른 방법을 적용할 수 있는지 검토하고 그 이후에 반정규화를 적용하도록 한다.

- 반정규화의 대상을 조사한다.

일단 전체 데이터의 양을 조사하고 그 데이터가 해당 프로세스를 처리할 때 성능저하가 나타날 수 있는지 검증해야 한다. 데이터가 대량이고 성능이 저하될 것으로 예상이 되면 다음 4가지 경우를 고려하여 반정규화를 고려하게 된다.

- 자주 사용되는 테이블에 접근(Access)하는 프로세스의 수가 많고 항상 일정한 범위만을 조회하는 경우에 반정규화를 검토한다.
- 테이블에 대량의 데이터가 있고 대량의 데이터 범위를 자주 처리하는 경우에 처리범위를 일정하게 줄이지 않으면 성능을 보장할 수 없을 경우에 반정규화를 검토한다.
- 통계성 프로세스에 의해 통계 정보를 필요로 할 때 별도의 통계테이블(반정규화 테이블)을 생성한다.
- 테이블에 지나치게 많은 조인(JOIN)이 걸려 데이터를 조회하는 작업이 기술적으로 어려울 경우 반정규화를 검토한다.

- 반정규화의 대상에 대해 다른 방법으로 처리할 수 있는지 검토한다.

가급적이면 데이터를 중복하여 데이터 무결성을 깨뜨릴 위험을 제어하기 위하여 반정규화를 결정하기 이전에 성능을 향상시킬 수 있는 다른 방법을 모색하도록 한다.

- 지나치게 많은 조인(JOIN)이 걸려 데이터를 조회하는 작업이 기술적으로 어려울 경우 뷰(VIEW)를 사용하면 이를 해결할 수도 있다. 뷰가 조회의 성능을 향상시키는 역할을 수행하지는 않는다. 다만 개발자별로 SQL문장을 만드는 방법에 따라 성능저하가 나타날 수 있으므로 성능을 고려한 뷰를 생성하여 개발자가 뷰를 통해 접근하게 함으로써 성능저하의 위험을 예방하는 것도 좋은 방법이 된다.
- 대량의 데이터처리나 부분처리에 의해 성능이 저하되는 경우에 클러스터링을 적용하거나 인덱스를 조정함으로써 성능을 향상시킬 수 있다. 클러스터링을 적용하는 방법은 대량의 데이터를 특정 클러스터링 팩트에 의해 저장방식을 다르게 하는 방법이다. 이 방법의 경우 데이터를 입력/수정/삭제하는 경우 성능이 많이 저하되므로 조회중심의 테이블이 아니라면 생성하면 안되는 오브젝트이다. 다만, 조회가 대부분이고 인덱스를 통해 성능향상이 불가능하다면 클러스터링을 고려할 만하다. 또한 인덱스를 통해 성능을 충분히 확보할 수 있다면 인덱스를 조정하여 반정규화를 회피하도록 한다.
- 대량의 데이터는 Primary Key의 성격에 따라 부분적인 테이블로 분리할 수 있다. 즉 파티셔닝 기법(Partitioning)이 적용되어 성능저하를 방지할 수 있다. 인위적인 테이블을 통합/분리하지 않고 물리적인 저장기법에 따라 성능을 향상시킬 수 있는 파티셔닝을 고려해 볼 수 있다. 이 경우는 데이터가 특정 기준(파티셔닝 키)에 의해 다르게 저장되고 파티셔닝 키에 따른 조회가 될 때 성능이 좋아지는 특성이 있다. 따라서 특정 기준에 의해 물리적인 저장공간이 구분될 수 있고 트랜잭션이 들어올 때 일정한 기준에 의해 들어온다면 파티셔닝 테이블을 적용하여 조회의 성능을 향상시키는 것도 좋은 방법이 될 수 있다.
- 응용 애플리케이션에서 로직을 구사하는 방법을 변경함으로써 성능을 향상시킬 수 있다. 응용 메모리 영역에 데이터를 처리하기 위한 값을 캐쉬한다든지 중간 클래스 영역에 데이터를 캐쉬하여 공유하게 하여 성능을 향상 시키는 것도 성능을 향상시키는 방법이 될 수 있다.

- 반정규화를 적용한다.

반정규화를 적용하기 이전에 사전에 충분히 성능에 대한 고려가 이루어져서 반정규화를 적용해야겠다는 판단이 들었다면 이 때 반정규화의 세 가지 규칙을 고려하여 반정규화를 적용하도록 한다. 반정규화를 하는 대상으로는 테이블, 속성, 관계에 대해 적용할 수 있으며 꼭 테이블과 속성, 관계에 대해 중복으로 가져가는 방법만이 반정규화가 아니고 테이블, 속성, 관계를 추가할 수도 있고 분할할 수도 있으며 제거할 수도 있다.

성능을 향상시킬 수 있는 포괄적인 방법을 적용하여 반정규화를 적용하는 것이 전문화된 반정규화의 기법임을 기억할 필요가 있다.

2. 반정규화의 기법

넓은 의미에서 반정규화를 고려할 때 성능을 향상시키기 위한 반정규화는 여러 가지가 나타날 수 있다.

가. 테이블 반정규화

[표 Ⅰ-2-1] 테이블의 반정규화

기법분류	기법	내용
테이블병합	1:1 관계 테이블병합	1:1 관계를 통합하여 성능향상
	1:M 관계 테이블병합	1:M 관계 통합하여 성능향상
	슈퍼/서브타입 테이블병합	슈퍼/서브 관계를 통합하여 성능향상
테이블분할	수직분할	칼럼단위의 테이블을 디스크 I/O를 분산처리 하기 위해 테이블을 1:1로 분리하여 성능향상(트랜잭션의 처리되는 유형을 파악이 선행되어야 함)
	수평분할	로우 단위로 집중 발생되는 트랜잭션을 분석하여 디스크 I/O및 데이터접근의 효율성을 높여 성능을 향상하기 위해 로우단위로 테이블을 쪼갬(관계가 없음)
테이블추가	중복테이블 추가	다른 업무이거나 서버가 다른 경우 동일한 테이블구조를 중복하여 원격조인을 제거하여 성능을 향상
	통계테이블 추가	SUM, AVG 등을 미리 수행하여 계산해 둠으로써 조회 시 성능을 향상
	이력테이블 추가	이력테이블 중에서 마스터 테이블에 존재하는 레코드를 중복하여 이력테이블에 존재하는 방법은 반정규화의 유형
	부분테이블 추가	하나의 테이블의 전체 칼럼 중 자주 이용하는데 자주 이용하는 집중화된 칼럼들이 있을 때 디스크 I/O를 줄이기 위해 해당 칼럼들을 모아놓은 별도의 반정규화된 테이블을 생성

나. 칼럼 반정규화

[표 Ⅰ-2-2] 칼럼의 반정규화

반정규화 기법	내용
중복칼럼 추가	조인에 의해 처리할 때 성능저하를 예방하기 위해 즉, 조인을 감소시키기 위해 중복된 칼럼을 위치시킴
파생칼럼 추가	트랜잭션이 처리되는 시점에 계산에 의해 발생되는 성능저하를 예방하기 위해 미리 값을 계산하여 칼럼에 보관함. Derived Column이라고 함
이력테이블 칼럼추가	대량의 이력데이터를 처리할 때 불특정 날 조회나 최근 값을 조회할 때 나타날 수 있는 성능저하를 예방하기 위해 이력테이블에 기능성 칼럼(최근값 여부, 시작과 종료일자 등)을 추가함
PK에 의한 칼럼 추가	복합의미를 갖는 PK를 단일 속성으로 구성하였을 경우 발생됨. 단일 PK안에서 특정 값을 별도로 조회하는 경우 성능저하가 발생될 수 있음. 이 때 이미 PK안에 데이터가 존재하지만 성능향상을 위해 일반속성으로 포함하는 방법이 PK의한 칼럼추가 반정규화임
응용시스템 오작동을 위한 칼럼 추가	업무적으로는 의미가 없지만 사용자가 데이터처리를 하다가 잘못 처리하여 원래 값으로 복구하기를 원하는 경우 이전 데이터를 임시적으로 중복하여 보관하는 기법. 칼럼으로 이것을 보관하는 방법은 오작동 처리를 위한 임시적인 기법이지만 이것을 이력데이터 모델로 풀어내면 정상적인 데이터 모델의 기법이 될 수 있음

다. 관계 반정규화

[표 Ⅰ-2-3] 관계의 반정규화

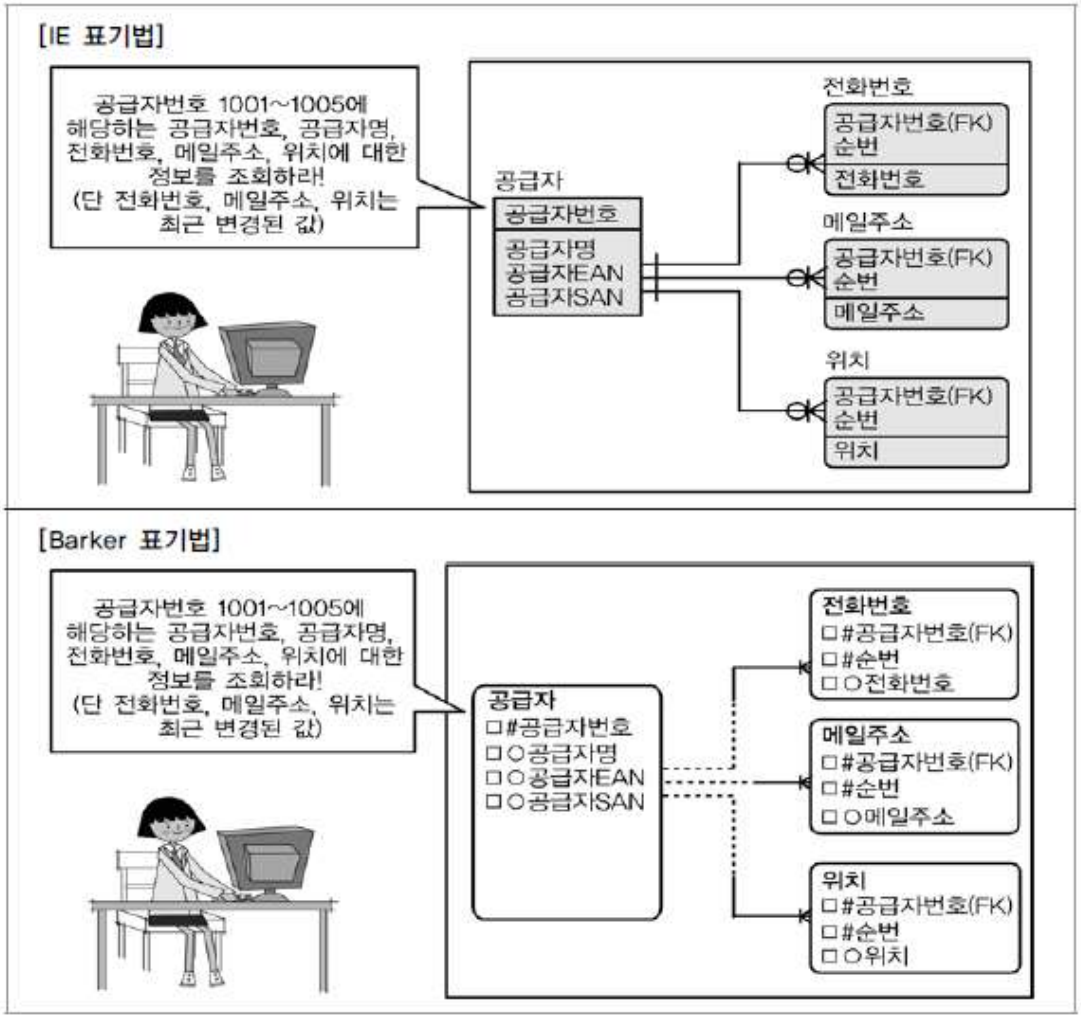
반정규화 기법	내용
중복관계 추가	데이터를 처리하기 위한 여러 경로를 거쳐 조인이 가능하지만 이 때 발생할 수 있는 성능저하를 예방하기 위해 추가적인 관계를 맺는 방법이 관계의 반정규화임

테이블과 칼럼의 반정규화는 데이터 무결성에 영향을 미치게 되나 관계의 반정규화는 데이터 무결성을 깨뜨릴 위험을 갖지 않고서도 데이터처리의 성능을 향상시킬 수 있는 반정규화의 기법이 된다. 데이터 모델 전체가 관계로 연결되어 있고 관계가 서로 먼 친척간에 조인관계가 빈번하게 되어 성능저하가 예상이 된다면 관계의 반정규화를 통해 성능향상을 도모할 필요가 있다

테이블과 칼럼의 반정규화는 데이터 무결성에 영향을 미치게 되나 관계의 반정규화는 데이터 무결성을 깨뜨릴 위험을 갖지 않고서도 데이터처리의 성능을 향상시킬 수 있는 반정규화의 기법이 된다. 데이터 모델 전체가 관계로 연결되어 있고 관계가 서로 먼 친척간에 조인관계가 빈번하게 되어 성능저하가 예상이 된다면 관계의 반정규화를 통해 성능향상을 도모할 필요가 있다.

3. 정규화가 잘 정의된 데이터 모델에서 성능이 저하될 수 있는 경우

[그림 Ⅰ-2-14]는 공급자라고 하는 엔터티가 마스터이고 전화번호와 메일주소 위치가 각각 변경되는 내용이 이력형태로 관리되는 데이터 모델이다. 이 모델에서 공급자정보를 가져오는 경우를 가정해 보자.



[그림 Ⅰ-2-14] 정규화 모델의 문제점

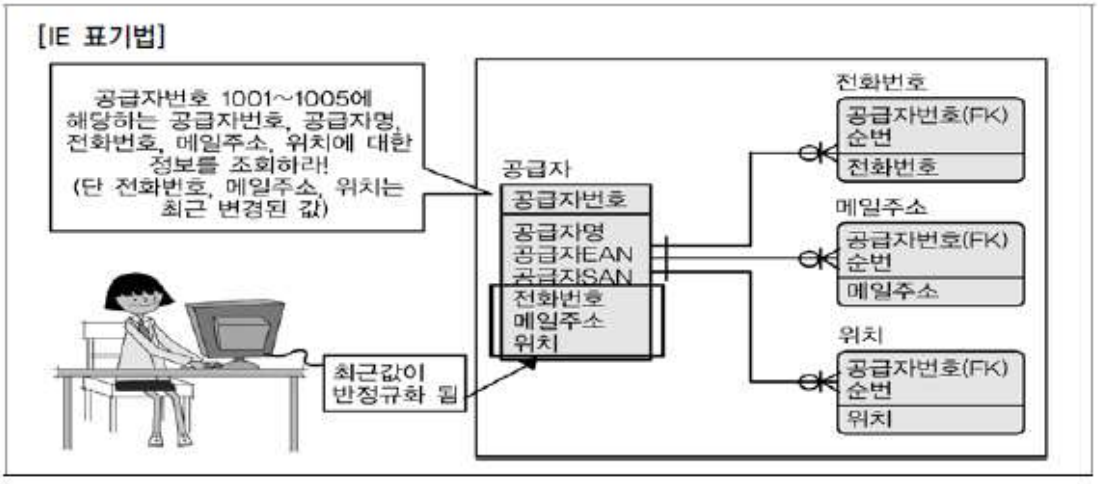
공급자와 전화번호, 메일주소, 위치는 1:M 관계이므로 한 명의 공급자당 여러 개의 전화번호, 메일주소, 위치가 존재한다. 따라서 가장 최근에 변경된 값을 가져오기 위해서는 조금 복잡한 조인이 발생될 수 밖에 없다.

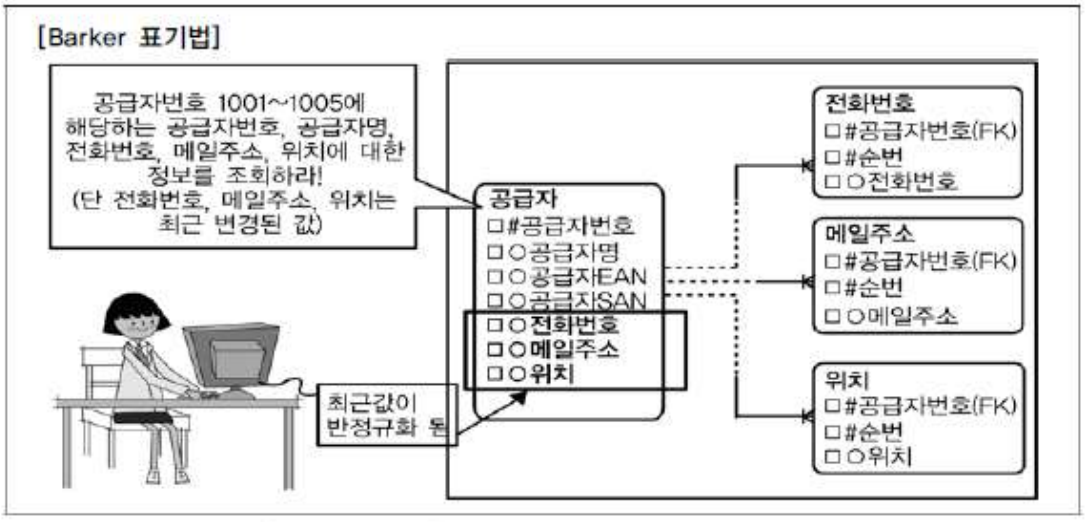
다음 SQL은 위와 같은 조건을 만족하는 SQL구문이 된다.

```
SELECT A.공급자명, B.전화번호, C.메일주소, D.위치
  FROM 공급자 A,
     (SELECT X.공급자번호, X.전화번호
        FROM 전화번호 X,
          (SELECT 공급자번호, MAX(순번) 순번
             FROM 전화번호
```

```
            WHERE 공급자번호 BETWEEN '1001' AND '1005'
            GROUP BY 공급자번호) Y
      WHERE X.공급자번호 = Y.공급자번호
         AND X.순번 = Y.순번) B,
    (SELECT X.공급자번호, X.메일주소
       FROM 메일주소 X,
            (SELECT 공급자번호, MAX(순번) 순번
               FROM 메일주소
              WHERE 공급자번호 BETWEEN '1001' AND '1005'
              GROUP BY 공급자번호) Y
      WHERE X.공급자번호 = Y.공급자번호
         AND X.순번 = Y.순번) C,
    (SELECT X.공급자번호, X.위치
        FROM 위치 X,
            (SELECT 공급자번호, MAX(순번) 순번
               FROM 위치
              WHERE 공급자번호 BETWEEN '1001' AND '1005'
              GROUP BY 공급자번호) Y
       WHERE X.공급자번호 = Y.공급자번호
          AND X.순번 = Y.순번) D
WHERE A.공급자번호 = B.공급자번호
   AND A.공급자번호 = C.공급자번호
   AND A.공급자번호 = D.공급자번호
   AND A.공급자번호 BETWEEN '1001' AND '1005'
```

정규화 된 모델이 적절하게 반정규화 되지 않으면 위와 같은 복잡한 SQL구문은 쉽게 나올 수 있다. 이른바 A4용지 5장으로 작성된 SQL이 쉽지 않게 발견될 수 있는 것이다.
위의 모델을 적절하게 반정규화를 적용하면 즉, 가장 최근에 변경된 값을 마스터에 위치시키면 다음과 같이 아주 간단한 SQL구문이 작성 된다.





[그림 Ⅰ-2-15] 반정규화를 통한 성능향상 사례

위에서 복잡하게 작성된 SQL문장이 반정규화를 적용하므로 인해 다음과 같이 간단하게 작성이 되어 가독성도 높아지고 성능도 향상되어 나타났다.

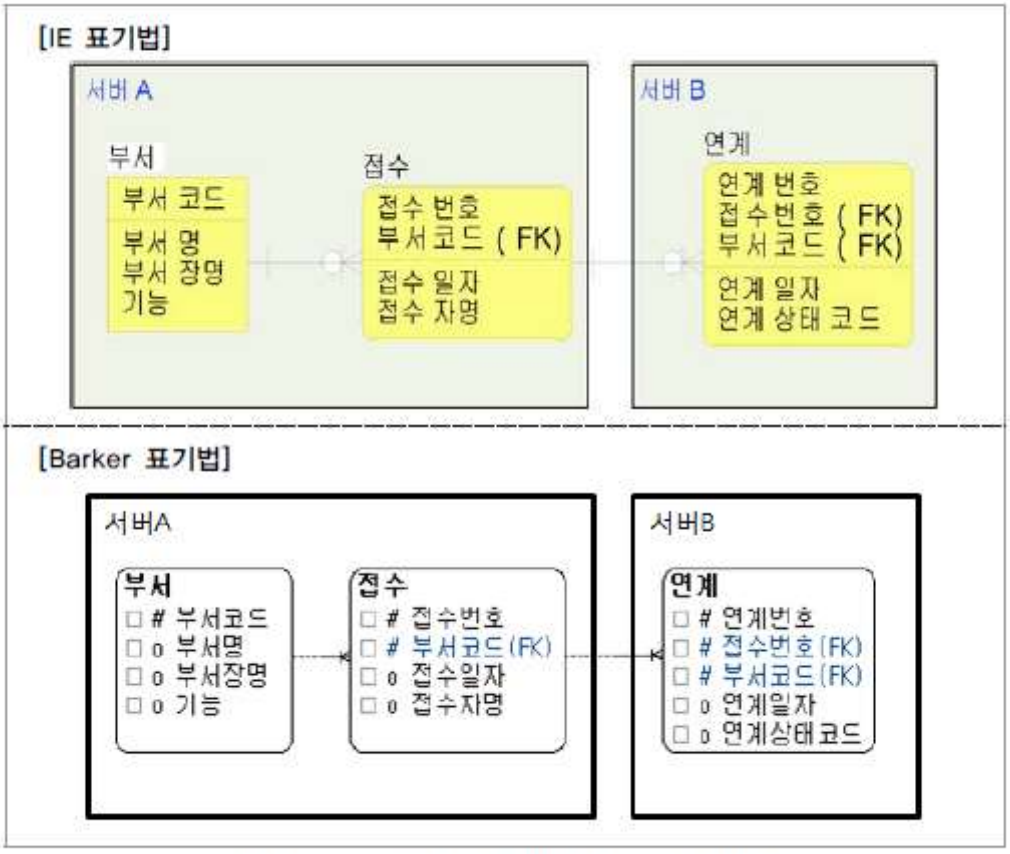
```
SELECT 공급자명, 전화번호, 메일주소, 위치
 FROM 공급자
WHERE 공급자번호 BETWEEN '1001' AND '1005'
```

결과만 보면 너무 당연하고 쉬운 것 같지만 기억해야 할 사실은 위 내용들은 모두 실제로 프로젝트를 할 때도 이와 같이 SQL문장의 성능과 단순성을 고려하지 않고 무모하게 설계되는 경우가 많이 있다는 점이다.

4. 정규화가 잘 정의된 데이터 모델에서 성능이 저하된 경우

업무의 영역이 커지고 다른 업무와 인터페이스가 많아짐에 따라 데이터베이스서버가 여러 대인 경우가 있다. [그림 Ⅰ-2-16]은 데이터베이스서버가 분리 되어 분산데이터베이스가 구성되어 있을 때 반정규화를 통해 성능을 향상시킬 수 있는 경우이다.

서버A에 부서와 접수 테이블이 있고 서버B에 연계라는 테이블이 있는데 서버B에서 데이터를 조회할 때 빈번하게 조회되는 부서번호가 서버A에 존재하기 때문에 연계, 접수, 부서 테이블이 모두 조인이 걸리게 된다. 게다가 분산데이터베이스 환경이기 때문에 다른 서버간에도 조인이 걸리게 되어 성능이 저하되는 것이다.



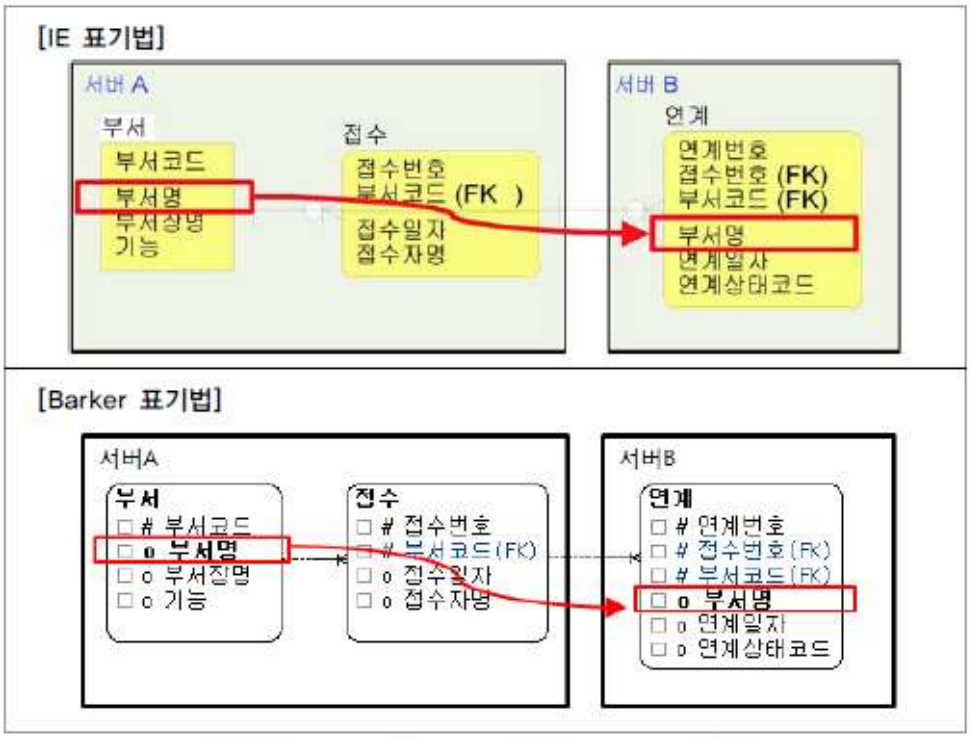
[그림 Ⅰ-2-16] 다른 서버간 정규화된 사례

위의 모델을 통해 서버B의 연계테이블에서 부서명에 따른 연계상태코드를 가져오는 SQL구문은 다음과 같이 작성된다.

```
SELECT C.부서명, A.연계상태코드
  FROM 연계 A, 접수 B, 부서 C <== 서버A와 서버B가 조인이 걸림
WHERE A.부서코드 = B.부서코드
   AND A.접수번호 = B.접수번호
   AND B.부서코드 = C.부서코드
   AND A.연계일자 BETWEEN '20040801' AND '20040901'
```

Oracle의 경우 DB LINK 조인이 발생하여 일반조인보다 성능이 저하될 것이다.

위의 분산 환경에 따른 데이터 모델을 다음과 같이 서버A에 있는 부서테이블의 부서명을 서버B의 연계테이블에 부서명으로 속성 반정규화를 함으로써 조회 성능을 향상시킬 수 있다.



[그림 Ⅰ-2-17] 다른 서버간 반정규화된 사례

[그림 Ⅰ-2-17]의 모델에 대한 SQL구문은 다음과 같이 작성된다.

```
SELECT 부서명, 연계상태코드
  FROM 연계
WHERE 연계일자 BETWEEN '20040801' AND '20040901'
```

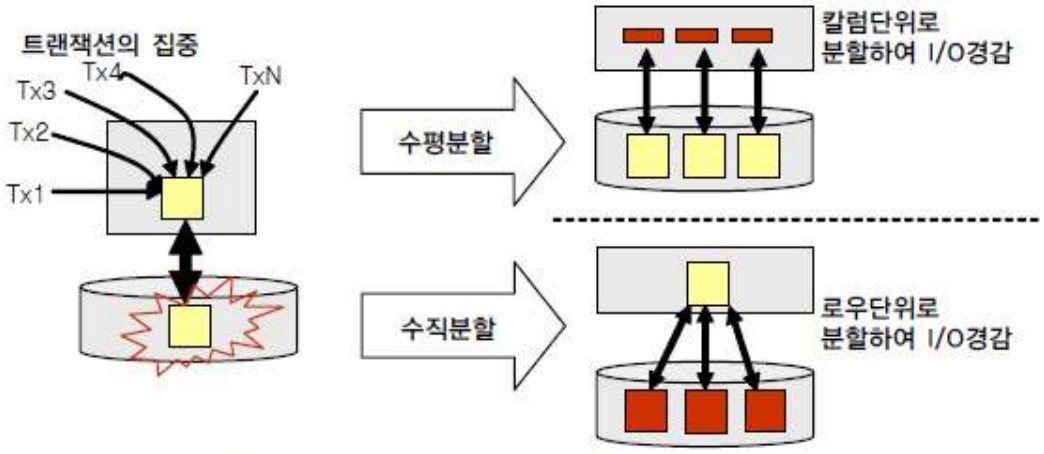
SQL구문도 간단해지고 분산되어 있는 서버간에도 DB LINK 조인이 발생하지 않아 성능이 개선되었다.

반정규화를 적용할 때 기억해야 할 내용은 데이터를 입력, 수정, 삭제할 때는 성능이 떨어지는 점을 기억해야 하고 데이터의 무결성 유지에 주의를 해야 한다.

제4절 대량 데이터에 따른 성능

1. 대량 데이터발생에 따른 테이블 분할 개요

아무리 설계가 잘되어 있는 데이터 모델이라고 하더라도 대량의 데이터가 하나의 테이블에 집약되어 있고 하나의 하드웨어 공간에 저장되어 있으면 성능저하를 피하기가 힘들다. 이런 원리는 하나의 고속도로 차선을 넓게 시공하여 건설해도 교통량이 많게 되면 이 넓은 도로가 정체현상을 보이는 것과 비슷한 원리로 이해할 수 있다. 일의 처리되는 양이 한군데에 몰리는 현상은 어떤 업무에 있어서 중요한 업무에 해당되는 데이터가 특정 테이블에 있는 경우에 발생이 되는데 이런 경우 트랜잭션이 분산 처리될 수 있도록 테이블단위에서 분할의 방법을 적용할 필요가 있는 것이다.



대량의 데이터가 존재하는 테이블에 많은 트랜잭션이 발생하여 성능이 저하되는
테이블 구조에 대해 수평/수직 분할 설계를 통해 성능저하를 예방할 수 있음

[그림 Ⅰ-2-18] 테이블 수평/수직분할에 의한 성능향상

한 테이블에 데이터가 대량으로 집중되거나 하나의 테이블에 여러 개의 칼럼이 존재하여 디스크에 많은 블록을 점유하는 경우는 모두 성능저하를 유발할 수 있는 경우이다. 하나의 테이블에 대량의 데이터가 존재하는 경우에는 인덱스의 Tree구조가 너무 커져 효율성이 떨어져 데이터를 처리(입력, 수정, 삭제, 조회)할 때 디스크 I/O를 많이 유발하게 된다. 또한 한 테이블에 많은 수의 칼럼이 존재하게 되면 데이터가 디스크의 여러 블록에 존재하므로 인해 디스크에서 데이터를 읽는 I/O량이 많아지게 되어 성능이 저하되게 된다.

대량의 데이터가 처리되는 테이블에 성능이 저하되는 이유는 SQL문장에서 데이터를 처리하기 위한 I/O의 양이 증가하기 때문이다. 당연하게 데이터의 양이 많아지면 그것을 처리하기 위한 I/O량이 많아질 것이라고 막연하게 생각할 수 있지만, 인덱스를 적절하게 구성하여 이용하게 하면 I/O를 줄일 수 있을 것이라고 생각할 수 있다. 조회조건에 따른 인덱스를 적절하게 이용하면 해당 테이블에 데이터가 아무리 많아도 원하는 데이터만 접근하면 되기 때문에 I/O의 양이 그다지 증가하지 않을 것으로 생각할 수 있다. 그러나 대량의 데이터가 하나의 테이블에 존재하게 되면 인덱스를 생성할 때 인덱스의 크기(용량)가 커지게 되고 그

렇게 되면 인덱스를 찾아가는 단계가 깊어지게 되어 조회의 성능에도 영향을 미치게 된다. 인덱스 크기가 커질 경우 조회의 성능에는 영향을 미치는 정도가 작지만 데이터를 입력/수정/삭제하는 트랜잭션의 경우 인덱스의 특성상 일량이 증가하여 더 많이 성능의 저하를 유발하게 된다. 또한 데이터에 대한 범위 조회시 더 많은 I/O 유발할 수 있게 되어 성능저하를 유발할 수 있게 된다.

칼럼이 많아지게 되면 물리적인 디스크에 여러 블록에 데이터가 저장되게 된다. 따라서 데이터를 처리할 때 여러 블록에서 데이터를 I/O해야 하는 즉 SQL문장의 성능이 저하될 수 특징을 가지게 된다. 물론, 테이블에 칼럼이 많아지는 현상은 정규화이론인 함수적 종속성에 근거하여 당연히 하나의 테이블에 설계할 수는 있다. 그러나 대량 데이터를 가진 테이블에서 불필요하게 많은 양의 I/O를 유발하여 성능이 저하되는 경우에는 이것을 기술적으로 분석하여 성능을 향상하는 방법으로 분할할 수 있다.

[그림 Ⅰ-2-19] 디스크에 데이터저장의 개념

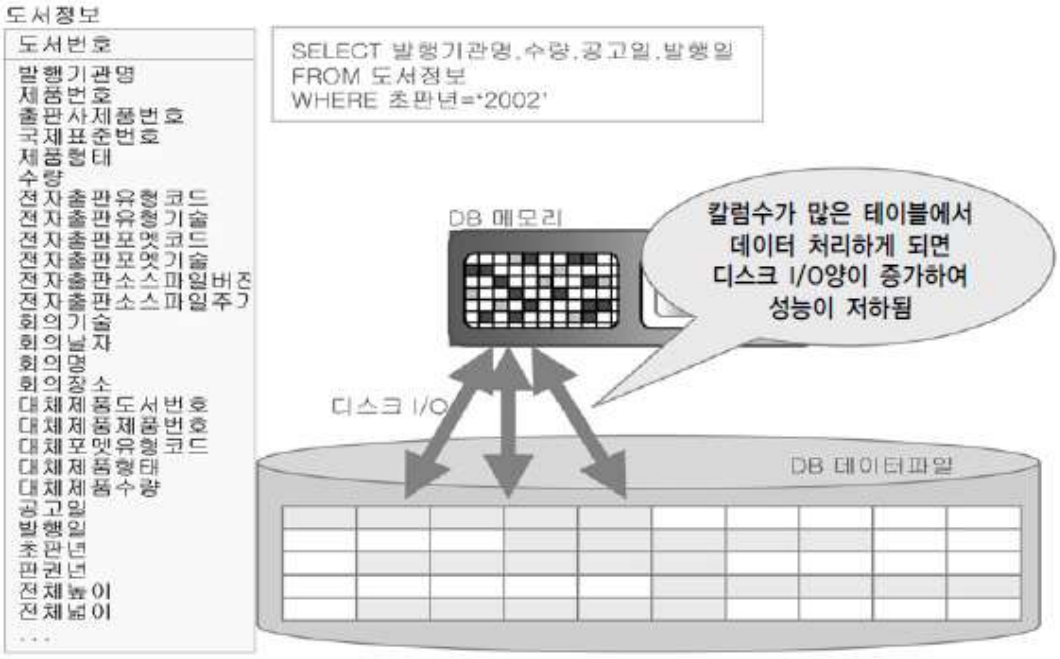
프로젝트를 수행할 때 때로는 하나의 테이블에 300개 이상의 칼럼을 가지고 있는 경우가 있다. 컴퓨터 화면 하나에는 볼 수가 없어서 스크롤을 하면서 하나의 테이블에 있는 칼럼을 구경해야 할 정도이다. 이렇게 많은 칼럼은 로우체이닝과 로우마이그레이션이 많아지게 되어 성능이 저하된다.

로우 길이가 너무 길어서 데이터 블록 하나에 데이터가 모두 저장되지 않고 두 개 이상의 블록에 걸쳐 하나의 로우가 저장되어 있는 형태가 로우체이닝(Row Chaining) 현상이다. 또한 로우마이그레이션(Row Migration)은 데이터 블록에서 수정이 발생하면 수정된 데이터를 해당 데이터 블록에서 저장하지 못하고 다른 블록의 빈 공간을 찾아 저장하는 방식이다. 로우체이닝과 로우마이그레이션이 발생하여 많은 블록에 데이터가 저장되면 데이터베이스 메모리에서 디스크와 I/O(입력/출력)가 발생할 때 불필요하게 I/O가 많이 발생하여 성능이 저하된다.

2. 한 테이블에 많은 수의 칼럼을 가지고 있는 경우

도서정보라고 하는 테이블에 칼럼수가 아주 많은 경우를 생각해 보자. 생략된 칼럼까지 합하면 대략 200개라고 가정한다. 만약 하나의 로우의 길이가 10KByte라고 하고 블록은 2K단위로 쪼개져 있다고 가정한다. 또한 블록에 데이터는 모두 채워진다고 가정하면 대략 하나의 로우는 5블록에 걸쳐 데이터가 저장될 것이다.

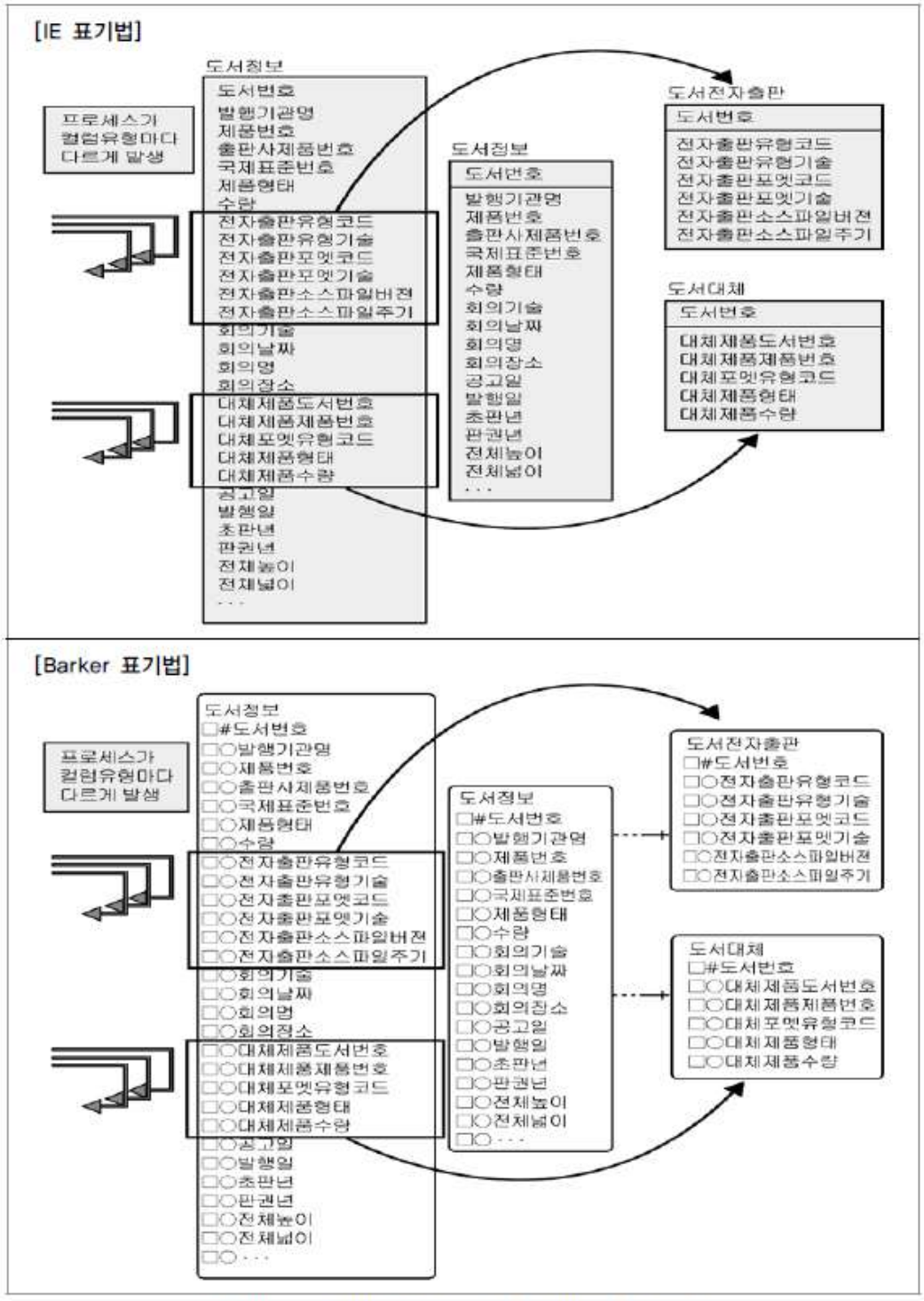
이 때 칼럼의 앞쪽에 위치한 발행기관명, 수량, 중간에 위치한 공고일, 발행일에 대한 정보를 가져오려면 물리적으로 칼럼의 값이 블록에 넓게 산재되어 있어 디스크 I/O가 많이 일어나게 된다.



[그림 Ⅰ-2-20] 칼럼수가 많은 테이블

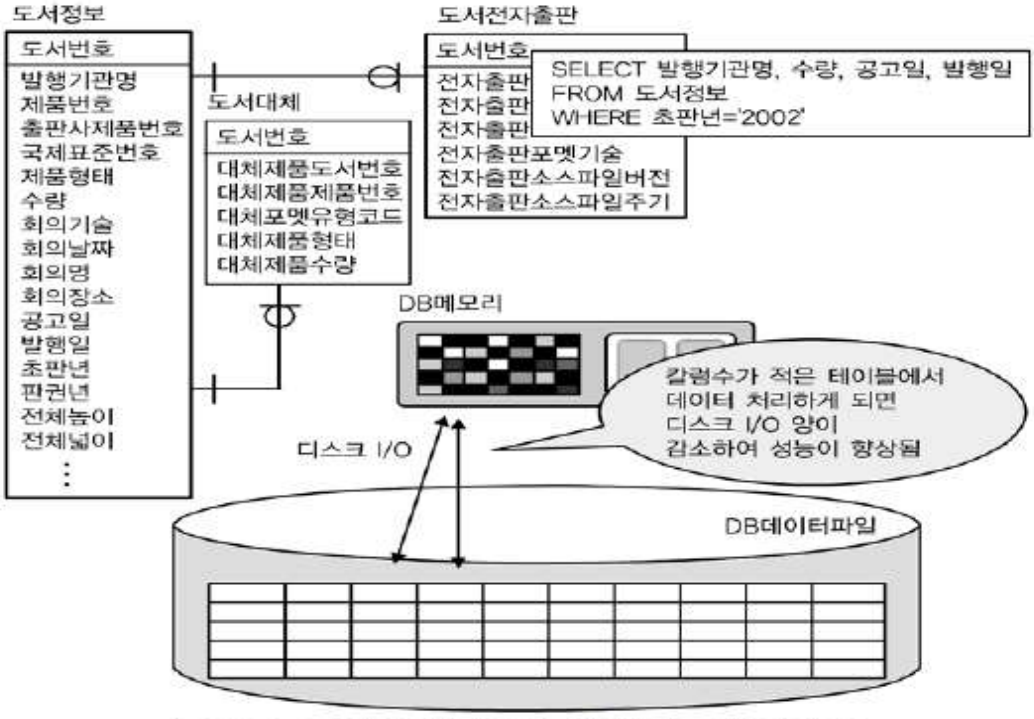
200개의 칼럼을 동시에 조회하여 화면에 보여주는 경우는 드문 사례이다. 만약 200개의 칼럼이 가지고 있는 값을 모두 한 화면에 보여주기 위해서는 화면을 몇 번 스크롤 하면서 보여야 한다. 즉 이렇게 많은 칼럼을 가지고 있는 테이블에 대해서는 트랜잭션이 발생될 때 어떤 칼럼에 대해 집중적으로 발생하는지 분석하여 테이블을 쪼개어 주면 디스크 I/O가 감소하게 되어 성능이 개선되게 된다.

[그림 Ⅰ-2-21]의 데이터 모델을 살펴보자.



[그림 Ⅰ-2-21] 칼럼수가 많은 테이블의 1:1 분리

도서정보 테이블에는 전자출판유형에 대한 트랜잭션이 독립적으로 발생이 되는 경우가 많고 대체제품에 대한 유형의 트랜잭션이 독립적으로 발생되는 경우가 많이 있어 1:1 관계로 분리하였다. 분리된 테이블은 디스크에 적어진 칼럼이 저장이 되므로 로우마이그레이션과 로우체이닝이 많이 줄어들 수 있다. 따라서 아래와 같이 발행기관명, 수량, 중간에 위치한 공고일, 발행일을 가져오는 동일한 SQL구문에 대해서도 디스크 I/O가 줄어들어 성능이 개선되게 된다.



[그림 Ⅰ-2-22] 칼럼수가 많은 테이블의 1:1 분리의 I/O

많은 수의 칼럼을 가지는 데이터 모델 형식도 실전 프로젝트에서 흔히 나타나는 현상이다. 트랜잭션을 분석하여 적절하게 1:1 관계로 분리함으로써 성능향상이 가능하도록 해야 한다.

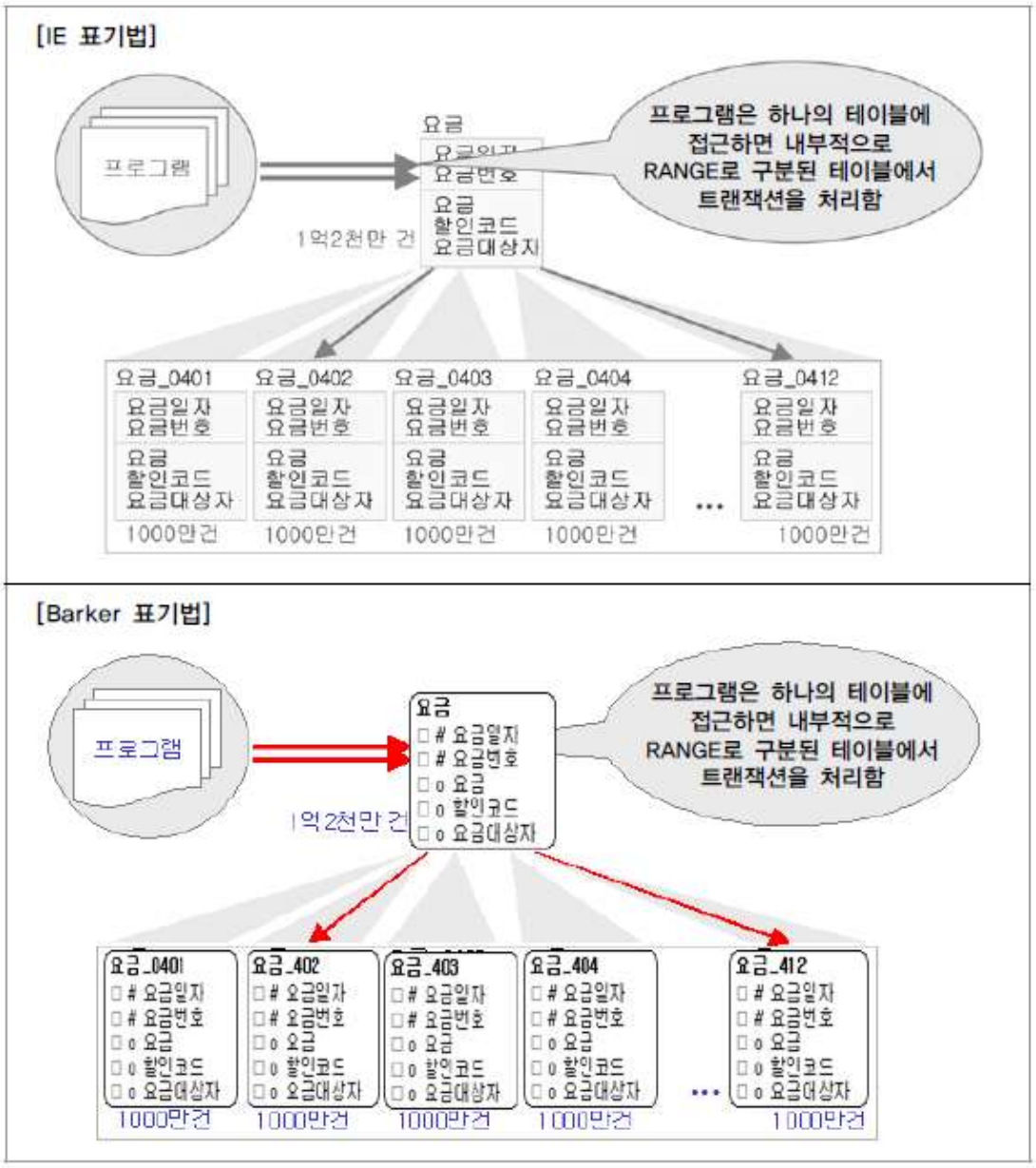
3. 대량 데이터 저장 및 처리로 인해 성능

테이블에 많은 양의 데이터가 예상될 경우 파티셔닝을 적용하거나 PK에 의해 테이블을 분할하는 방법을 적용할 수 있다. Oracle의 경우 크게 LIST PARTITION(특정값 지정), RANGE PARTITION(범위), HASH PARTITION(해쉬적용), COMPOSITE PARTITION(범위와 해쉬가 복합) 등이 가능하다.

데이터량이 몇 천만건을 넘어서면 아무리 서버사양이 훌륭하고 인덱스를 잘 생성해준다고 하더라고 SQL문장의 성능이 나오지 않는다. 이 때는 논리적으로는 하나의 테이블로 보이지만 물리적으로 여러 개의 테이블스페이스에 쪼개어 저장될 수 있는 구조의 파티셔닝을 적용하도록 한다.

가. RANGE PARTITION 적용

다음은 요금테이블에 PK가 요금일자+요금번호로 구성되어 있고 데이터건수가 1억2천만 건인 대용량 테이블의 경우이다. 하나의 테이블로는 너무 많은 데이터가 존재하므로 인해 성능이 느린 경우에 해당된다. 이 때 요금의 특성상 항상 월단위로 데이터 처리를 하는 경우가 많으므로 PK인 요금일자의 년+월을 이용하여 12개의 파티션 테이블을 만들었다. 하나의 파티션 테이블당 평균 1,000만 건의 데이터가 있다고 가정한다.



[그림 Ⅰ-2-23] 파티셔닝의 적용-RANGE

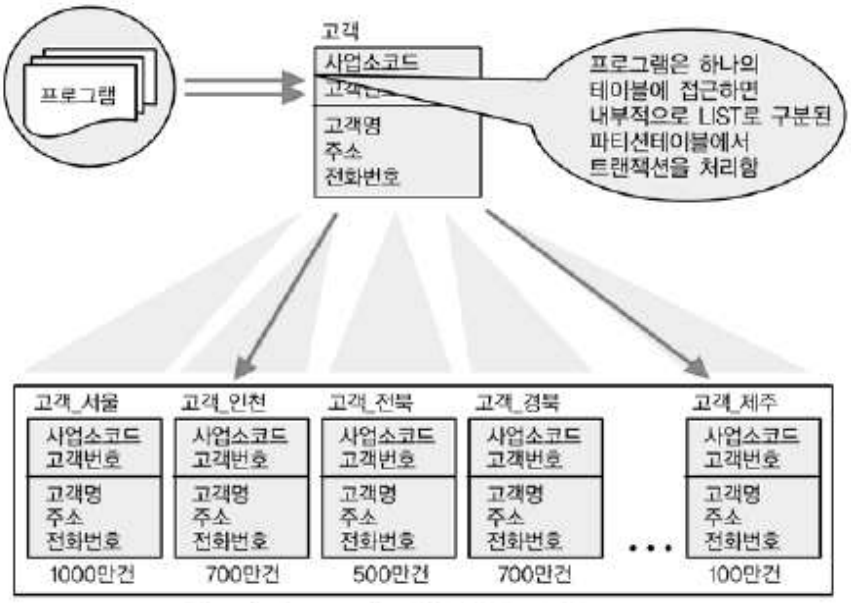
SQL문장을 처리할 때는 마치 하나의 테이블처럼 보이는 요금 테이블을 이용하여 처리하면 되지만 DBMS 내부적으로는 SQL WHERE 절에 비교된 요금일자에 의해 각 파티션에 있는 정보를 찾아가므로 평균 1,000만 건의 데이터가 있는 곳을 찾아도 되어 성능이 개선될 수 있다.

가장 많이 사용하는 파티셔닝의 기준이 RANGE PARTITION이다. 대상 테이블이 날자 또는 숫자값으로 분리가 가능하고 각 영역별로 트랜잭션이 분리된다면 RANGE PARTITION을 적용한다. 또한 RANGE PARTITION은 데이터보관주기에 따라 테이블에 데이터를 쉽게 지우는 것이 가능하므로(파티션 테이블을 DROP하면 되므로) 데이터보관주기에 다른 테이블관리가 용이하다.

나. LIST PARTITION 적용

지점, 사업소, 사업장, 핵심적인 코드값 등으로 PK가 구성되어 있고 대량의 데이터가 있는 테이블이라면 값 각각에 의해 파티셔닝이 되는 LIST PARTITION을 적용할 수 있다.

[그림 Ⅰ-2-24]의 예는 고객이라고 하는 테이블에 데이터가 1억 건이 있는데 하나의 테이블에서 데이터를 처리하기에는 SQL문장의 성능이 저하되어 지역을 나타내는 사업소코드별로 LIST PARTITION을 적용한 예이다.



[그림 Ⅰ-2-24] 파티셔닝의 적용-LIST

LIST PARTITION은 대용량 데이터를 특정값에 따라 분리 저장할 수는 있으나 RANGE PARTITION과 같이 데이터 보관주기에 따라 쉽게 삭제하는 기능은 제공될 수 없다.

다. HASH PARTITION 적용

기타 HASH PARTITION은 지정된 HASH 조건에 따라 해싱 알고리즘이 적용되어 테이블이 분리되며 설계자는 테이블에 데이터가 정확하게 어떻게 들어갔는지 알 수 없다. 역시 성능향상을 위해 사용하며 데이터 보관주기에 따라 쉽게 삭제하는 기능은 제공될 수 없다.

데이터량이 대용량이 되면 파티셔닝의 적용은 필수적으로 파티셔닝 기준을 나눌 수 있는 조건에 따라 적절한 파티셔닝을 방법을 선택하여 성능을 향상 시키도록 한다.

4. 테이블에 대한 수평분할/수직분할의 절차

테이블에 대한 수평분할/수직분할에 대한 결정은 다음의 4가지 원칙을 적용하면 된다.

1) 데이터 모델링을 완성한다.

2) 데이터베이스 용량산정을 한다.

3) 대량 데이터가 처리되는 테이블에 대해서 트랜잭션 처리 패턴을 분석한다.

4) 칼럼 단위로 집중화된 처리가 발생하는지, 로우단위로 집중화된 처리가 발생되는지 분석하여 집중화된 단위로 테이블을 분리하는 것을 검토한다.

용량산정은 어느 테이블에 데이터의 양이 대용량이 되는지 분석하는 것이다. 특정 테이블의 용량이 대용량인 경우 칼럼의 수가 너무 많은 지 확인한다. 칼럼의 수가 많은 경우 트랜잭션의 특성에 따라 테이블을 1:1 형태로 분리할 수 있는지 검증하면 된다. 칼럼의 수가 적지만 데이터용량이 많아 성능저하가 예상이 되는 경우 테이블에 대해 파티셔닝 전략을 고려하도록 한다. 이 때 임의로 파티셔닝할 것인지 데이터가 발생되는 시간에 따라 파티셔닝을 할 것인지를 설명된 기준에 따라 적용하면 된다.

제5절 데이터베이스 구조와 성능

1. 슈퍼타입/서브타입 모델의 성능고려 방법

가. 슈퍼/서브타입 데이터 모델의 개요

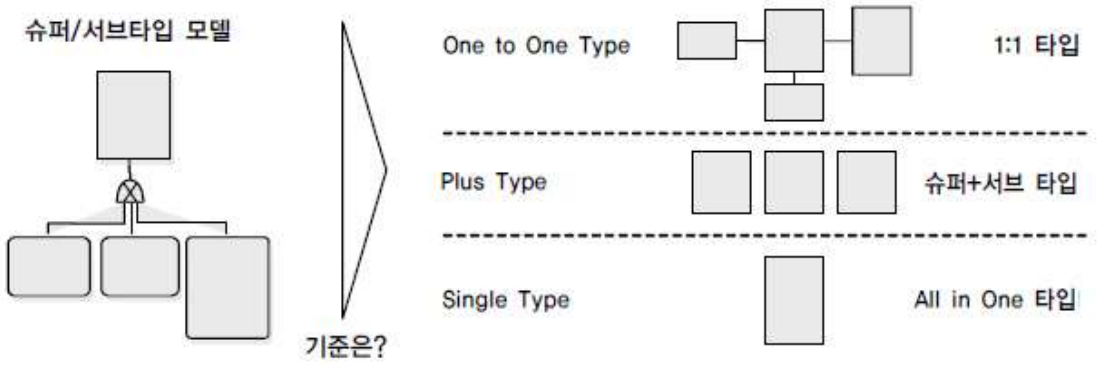
Extended ER모델이라고 부르는 이른바 슈퍼/서브타입 데이터 모델은 최근에 데이터 모델링을 할 때 자주 쓰이는 모델링 방법이다. 이 모델이 자주 쓰이는 이유는 업무를 구성하는 데이터의 특징을 공통과 차이점의 특징을 고려하여 효과적으로 표현할 수 있기 때문이다. 즉, 공통의 부분을 슈퍼타입으로 모델링하고 공통으로부터 상속받아 다른 엔터티와 차이가 있는 속성에 대해서는 별도의 서브엔터티로 구분하여 업무의 모습을 정확하게 표현하면서 물리적인 데이터 모델로 변환을 할 때 선택의 폭을 넓힐 수 있는 장점이 있다. 이러한 장점 때문에 많은 프로젝트에서 슈퍼/서브타입을 활용한 데이터 모델의 사례가 증가하고 있다.

당연히 슈퍼/서브타입의 데이터 모델은 논리적인 데이터 모델에서 이용되는 형태이고 분석/설계단계를 구분하자면, 분석단계에서 많이 쓰이는 모델이다. 따라서 물리적인 데이터 모델을 설계하는 단계에서는 슈퍼/서브타입 데이터 모델을 일정한 기준에 의해 변환을 해야 한다. 그런데 실제로 프로젝트 현장에서는 이것을 변환하는 방법에 대해 정확한 노하우가 없기 때문에 막연하게 1:1로 변환하거나 아니면 하나의 테이블로 구성해 버리는 현상이 나타난다.

물리적인 데이터 모델이 성능을 고려한 데이터 모델이 되어야 한다는 점을 고려하면 이렇게 막연하게 슈퍼/서브타입을 아무런 기준없이 변환하는 것 자체가 성능이 저하될 수 있는 위험이 있음을 기억해야 한다.

나. 슈퍼/서브타입 데이터 모델의 변환

성능을 고려한 슈퍼타입과 서브타입의 모델 변환의 방법을 알아보면 [그림 Ⅰ-2-25]와 같다.



[그림 Ⅰ-2-25] 슈퍼타입과 서비스타입의 변환

슈퍼/서브타입에 대한 변환을 잘못하면 성능이 저하되는 이유는 트랜잭션 특성을 고려하지 않고 테이블이 설계되었기 때문이다.

이것을 3가지 경우의 수로 정리하면 설명하면 다음과 같다.

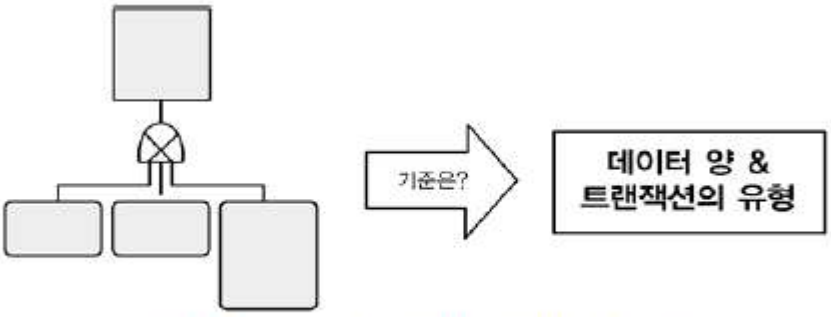
1) 트랜잭션은 항상 일괄로 처리하는데 테이블은 개별로 유지되어 Union연산에 의해 성능이 저하될 수 있다.

2) 트랜잭션은 항상 서브타입 개별로 처리하는데 테이블은 하나로 통합되어 있어 불필요하게 많은 양의 데이터가 집약되어 있어 성능이 저하되는 경우가 있다.

3) 트랜잭션은 항상 슈퍼+서브 타입을 공통으로 처리하는데 개별로 유지되어 있거나 하나의 테이블로 집약되어 있어 성능이 저하되는 경우가 있다.

해당 테이블에 발생되는 성능이 중요한 트랜잭션이 빈번하게 처리되는 기준에 따라 테이블을 설계해야 이러한 성능저하 현상을 예방할 수 있음을 기억해야 한다.

슈퍼/서브타입을 성능을 고려한 물리적인 데이터 모델로 변환하는 기준은 데이터 양과 해당 테이블에 발생되는 트랜잭션의 유형에 따라 결정된다.



[그림 Ⅰ-2-26] 슈퍼/서브타입 변환 기준

데이터의 양은 데이터량이 소량일 경우 성능에 영향을 미치지 않기 때문에 데이터처리의 유연성을 고려하여 가급적 1:1 관계를 유지하는 것이 바람직하다. 그러나 데이터용량이 많아지는 경우 그리고 해당 업무적인 특징이 성능에 민감한 경우는 트랜잭션이 해당 테이블에 어떻게 발생되는지에 따라 3가지 변환방법을 참조하여 상황에 맞게 변환하도록 해야 한다.

다. 슈퍼/서브 타입 데이터 모델의 변환기술

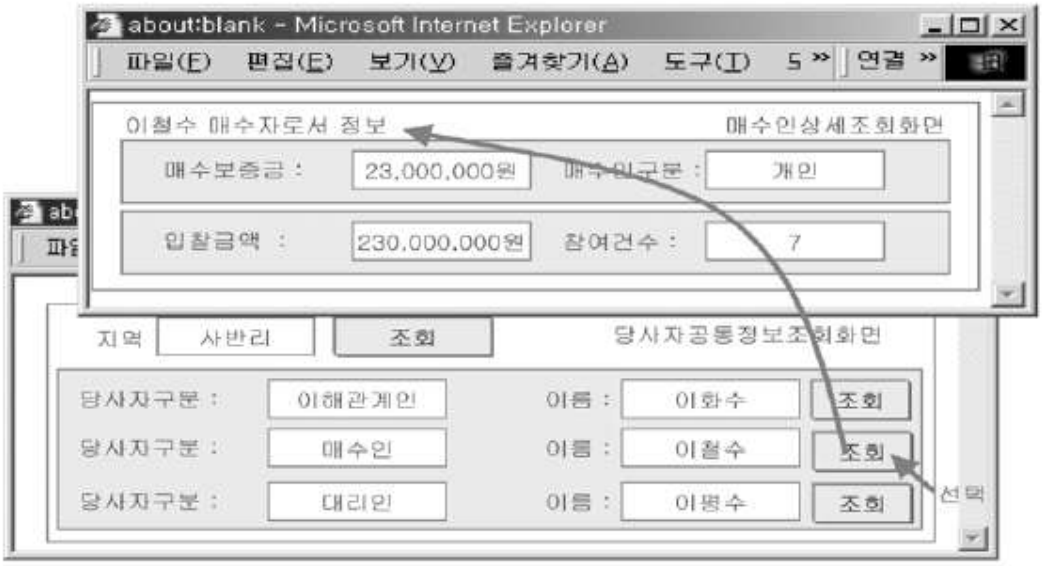
논리적인 데이터 모델에서 설계한 슈퍼타입/서브타입 모델을 물리적인 데이터 모델로 전환할 때 주로 어떤 유형의 트랜잭션이 발생하는지 검증해야 한다. 물론 데이터량이 아주 작다면, 예를 들어 10만 건도 되지 않는다면 그리고 시스템을 운영하는 중에도 증가하지 않는다면 트랜잭션의 성격을 고려하지 않고 전체를 하나의 테이블로 묶어도 좋은 방법이다.

그러나 데이터량이 많이 존재하고 지속적으로 증가하는 양도 많다면 슈퍼타입/서브타입에 대해 물리적인 데이터 모델로 변환하는 세 가지 유형에 대해 세심하게 적용을 해야 한다.

1) 개별로 발생되는 트랜잭션에 대해서는 개별 테이블로 구성

업무적으로 발생되는 트랜잭션이 슈퍼타입과 서브타입 각각에 대해 발생하는 것이다.

[그림 Ⅰ-2-27]의 업무화면을 보면 공통으로 처리하는 슈퍼타입테이블인 당사자 정보를 미리 조회하고 원하는 내용을 클릭하면 거기에 따라서 서브타입인 세부적인 정보 즉 이해관계인, 매수인, 대리인에 대한 내용을 조회하는 형식이다. 즉 슈퍼타입이 각 서브타입에 대해 기준역할을 하는 형식으로 사용할 때 이러한 유형의 트랜잭션이 발생이 된다.



[그림 Ⅰ-2-27] 트랜잭션 유형

위와 같이 슈퍼타입과 서브타입각각에 대해 독립적으로 트랜잭션이 발생이 되면 슈퍼타입에도 꼭 필요한 속성만을 가지게 하고 서브타입에도 꼭 필요한 속성 및 자신이 타입에 맞는 데이터만 가지게 하기 위해서 모두 분리하여 1:1 관계를 갖도록 한다.

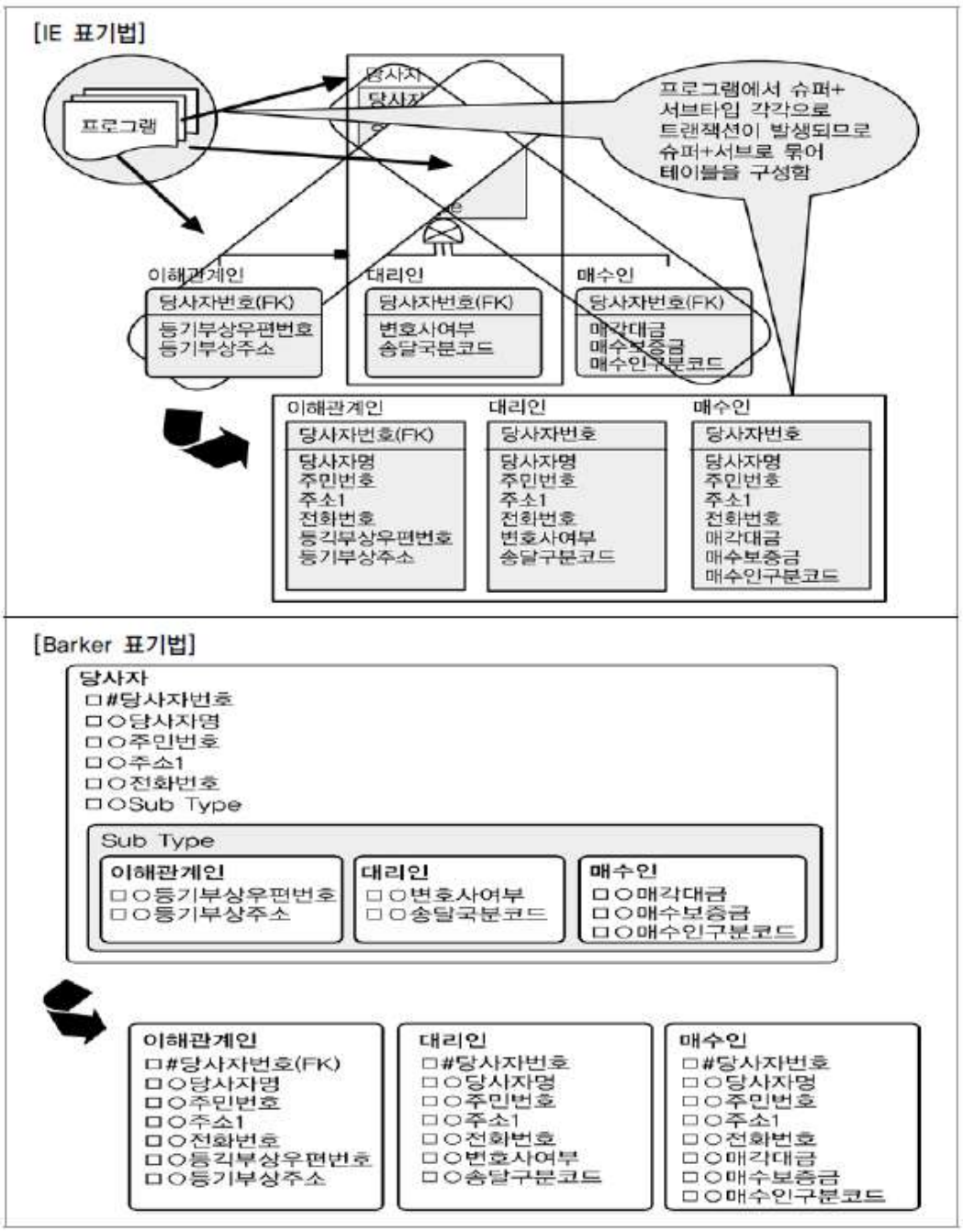
실전프로젝트에서는 데이터량이 대용량으로 존재하는 경우에 공통으로 이용하는 슈퍼타입의 속성의 수가 너무 많아져 디스크 I/O가 많아지는 것을 방지하기 위해 위와 같이 각각을 1:1 관계로 가져가는 경우도 있다.

2) 슈퍼타입+서브타입에 대해 발생되는 트랜잭션에 대해서는 슈퍼타입+서브타입 테이블로 구성

만약 대리인이 10만 건, 매수인 500만 건, 이해관계인 500만 건의 데이터가 존재한다고 가정하고 슈퍼타입과 서브타입이 모두 하나의 테이블로 통합되어 있다고 가정하자. 매수인, 이해관계인에 대한 정보는 배제하고 10만 건뿐인 대리인에 대한 데이터만 처리할 경우 다른 테이블과 같이 데이터가 1천10만 건이 저장되어 있는 곳에서 처리해야 하므로 불필요한 성능저하 현상이 유발된다. 즉 대리인에 대한 처리가 개별적으로 많이 발생하는데 매수인과 이해관계인의 데이터까지 포함되어 있으므로 최대 10만 건을 읽어 처리할 수 있는 업무가 최대 1천10만 건을 읽어 처리하는 경우가 발생될 수 있다.

이와 같이 슈퍼타입과 서브타입을 묶어 트랜잭션이 발생하는 업무특징을 가지고 있을 때에는 다음 데이터 모델과 같이 슈퍼타입+각서브타입을 하나로 묶어 별도의 테이블로 구성하는 것이 효율적이다.

업무적인 특성상 실전 프로젝트에서 슈퍼타입/서브타입모델은 위와 같이 각각이 슈퍼타입+서브타입으로 묶여 구성하는 경우가 많다.



[그림 Ⅰ-2-28] Plus Type 변환

3) 전체를 하나로 묶어 트랜잭션이 발생할 때는 하나의 테이블로 구성

대리인 10만 건, 매수인 500만 건, 이해관계인 500만 건의 데이터가 존재한다고 하더라고 데이터를 처리할 때 대리인, 매수인, 이해관계인을 항상 통합하여 처리한다고 하면 테이블을 개별로 분리해야 불필요한 조인을 유발하거나 불필요한 UNION ALL과 같은 SQL구문이 작성되어 성능이 저하된다. 비록 슈퍼타입과 서브타입의 테이블들을 하나로 묶었을 때 각각의 속성별로 제약사항(NULL/NOT NULL, 기본값, 체크값)을 정확하게 지정하지 못할지라도 대용량이고 성능향상이 필요하다면 하나의 테이블로 묶어서 만들어 준다.

3가지 전개 방식이 아주 간단한 원리 같은데 이것도 실전 프로젝트에서 적용하면 쉽지 않은 경우가 많이 나타난다. 때로는 각각의 유형이 혼합되어 있는 경우도 있다. 혼합된 트랜잭션 유형이 있는 경우는 많이 발생하는 트랜잭션 유형에 따라 구성하면 된다.

라. 슈퍼/서브타입 데이터 모델의 변환타입 비교

[표 Ⅰ-2-4] 슈퍼/서브타입 데이터 모델 변환타입 비교

구분	OneToOne Type	Plus Type	Single Type
특징	개별 테이블 유지	슈퍼+서브타입 테이블	하나의 테이블
확장성	우수함	보통	나쁨
조인성능	나쁨	나쁨	우수함
I/O량 성능	좋음	좋음	나쁨
관리용이성	좋지않음	좋지않음	좋음(1개)
트랜잭션 유형에 따른 성택 방법	개별 테이블로 접근이 많은 경우 선택	슈퍼+서브 형식으로 데이터를 처리하는 경우 선택	전체를 일괄적으로 처리하는 경우 선택

[표 Ⅰ-2-4]와 같이 각 성능이 좋을 수도 나쁠 수도 있기 때문에 변환모델의 선택은 철저하게 데이터베이스에 발생되는 트랜잭션의 유형에 따라 선택을 해야 한다.

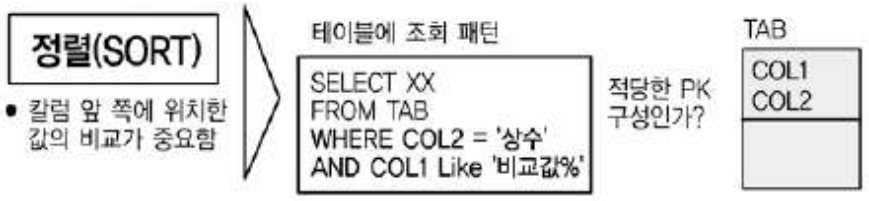
2. 인덱스 특성을 고려한 PK/FK 데이터베이스 성능향상

가. PK/FK 칼럼 순서와 성능개요

데이터를 조회할 때 가장 효과적으로 처리될 수 있도록 접근경로를 제공하는 오브젝트가 바로 인덱스이다. 일반적으로 데이터베이스 테이블에서는 균형 잡힌 트리구조의 B*Tree구조를 많이 사용한다. 우리는 B*Tree구조의 내부 알고리즘까지는 알 필요가 없더라도 그 구조를 이용할 때 정렬되어 있는 특징으로 인해 데이터베이스 설계에 이 특징에 따라 설계에 반영해야 할 요소에 대해서는 반드시 알고 있어야 좋은 데이터 모델을 만들어 낼 수 있게 된다.

프로젝트에서 PK/FK설계는 업무적 의미로도 매우 중요한 의미를 가지고 있지만 데이터를 접근할 때 경로를 제공하는 성능의 측면에서도 중요한 의미를 가지고 있기 때문에 성능을 고려한 데이터베이스 설계가 될 수 있도록 설계단계 말에 칼럼의 순서를 조정할 필요가 있다.

일반적으로 프로젝트에서는 PK/FK 칼럼 순서의 중요성을 인지하지 못한 채로 데이터 모델링이 되어 있는 그 상태대로 바로 DDL을 생성함으로써 데이터베이스 데이터처리 성능에 문제를 유발하는 경우가 빈번하게 발생이 된다.



테이블에 발생되는 트랜잭션의 조회 패턴에 따라 PK/FK 칼럼의 순서를 조정해야 함

[그림 Ⅰ-2-29] 인덱스 특성에 따른 PK/FK순서

간단한 것 같지만 실전 프로젝트에서는 아주 중요한 내용이 바로 PK순서이다. 성능저하 현상이 많은 부분이 PK가 여러 개의 속성으로 구성된 복합식별자 일 때 PK순서에 대해 별로 고려하지 않고 데이터 모델링을 한 경우에 해당된다.

특히 물리적인 데이터 모델링 단계에서는 스스로 생성된 PK순서 이외에 다른 엔터티로부터 상속받아 발생되는 PK순서까지 항상 주의하여 표시하도록 해야 한다.

PK는 해당 해당테이블의 데이터를 접근할 가장 빈번하게 사용되는 유일한 인덱스(Unique Index)를 모두 자동 생성한다. PK순서를 결정하는 기준은 인덱스 정렬구조를 이해한 상태에서 인덱스를 효율적으로 이용할 수 있도록 PK순서를 지정해야 한다. 즉 인덱스의 특징은 여러 개의 속성이 하나의 인덱스로 구성되어 있을 때 앞쪽에 위치한 속성의 값이 비교자로 있어야 인덱스가 좋은 효율을 나타낼 수 있다. 앞쪽에 위치한 속성 값이 가급적 '=' 아니면 최소한 범위 'BETWEEN' '< >'가 들어와야 인덱스를 이용할 수 있는 것이다.

데이터 모델링 때 결정한 PK순서와는 다르게 DDL문장을 통해 PK순서를 다르게 생성할 수도 있다. 그러나 대부분의 프로젝트에서는 데이터 모델의 PK 순서에 따라 그대로 PK를 생성한다. 만약 다르게 생성한다고 하더라도 데이터 모델과 데이터베이스 테이블의 구조가 다른 것처럼 보여 유지보수에 어려움이 많을 것이다. 또한 FK라고 하더라도 데이터를 조회할 때 조인의 경로를 제공하는 역할을 수행하므로 FK에 대해서는 반드시 인덱스를 생성하도록 하고 인덱스 칼럼의 순서도 조회의 조건을 고려하여 접근이 가장 효율적인 칼럼 순서대로 인덱스를 생성하도록 주의해야 한다.

나. PK칼럼의 순서를 조정하지 않으면 성능이 저하 이유

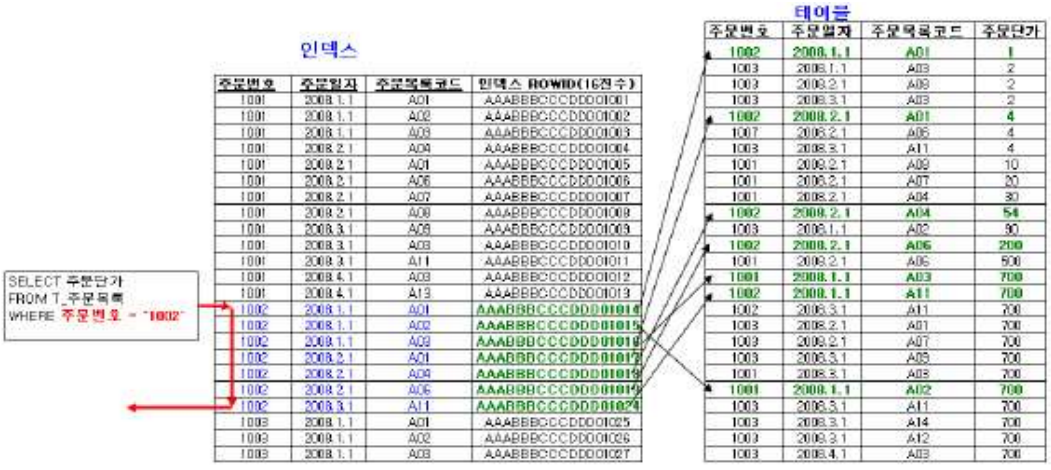
먼저 데이터 모델링에서 엔터티를 설계하면 그에 따라 DDL이 생성이 되고 생성된 DDL에 따라 인덱스가 생성된다. 이 때 우리가 알아야 할 구조는 인덱스의 정렬구조에 해당된다.

[그림 Ⅰ-2-30]은 인덱스의 정렬구조가 생성되는 구조를 보여주고 있다.

[그림 Ⅰ-2-30] 데이터 모델과 인덱스 구조

[그림 Ⅰ-2-30]에서 보면 테이블에서 데이터 모델의 PK순서에 따라 DDL이 그대로 생성이 되고 테이블의 데이터가 주문번호가 가장먼저 정렬되고 그에 따라 주문일자가 정렬이 되고 마지막으로 주문목록코드가 정렬되는 것을 알 수 있다. 이러한 정렬 구조로 인해 데이터를 접근하는 트랜잭션의 조건에 따라 다른 인덱스 접근방식을 보여주게 된다.

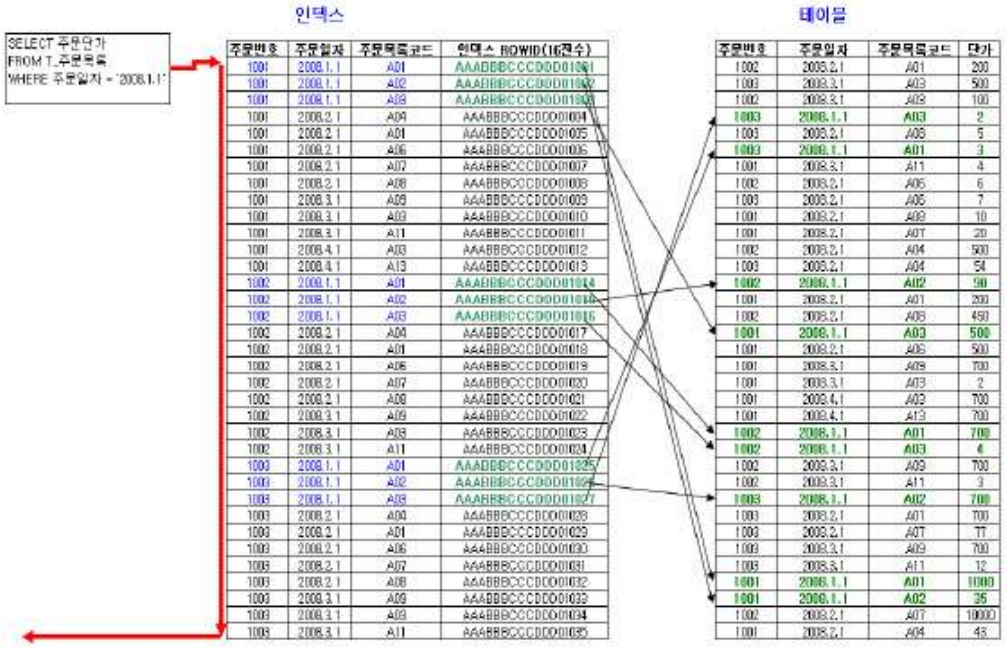
위와 같은 인덱스의 정렬 구조에서 SQL구문의 조건에 따라 인덱스를 처리하는 범위가 달라지게 된다. 맨 앞에 있는 인덱스 칼럼에 대해 조회 조건이 들어올 때 데이터를 접근하는 방법은 [그림 Ⅰ-2-31]과 같다.



맨 앞에 있는 인덱스 칼럼에 조회 조건이 들어올 때 Scan방법

[그림 Ⅰ-2-31] 조건에 따른 인덱스 Scan구조(1)

인덱스의 정렬된 첫 번째 칼럼에 비교가 되었기 때문에 순차적으로 데이터를 찾아가게 된다. 맨 앞에 있는 칼럼이 제외된 상태에서 데이터를 조회 할 경우 데이터를 비교하는 범위가 매우 넓어지게 되어 성능 저하를 유발하게 된다.



맨 앞에 있는 인덱스 칼럼에 조회 조건이 들어오지 않을 때 Scan방법
[그림 Ⅰ-2-32] 조건에 따른 인덱스 Scan구조(2)

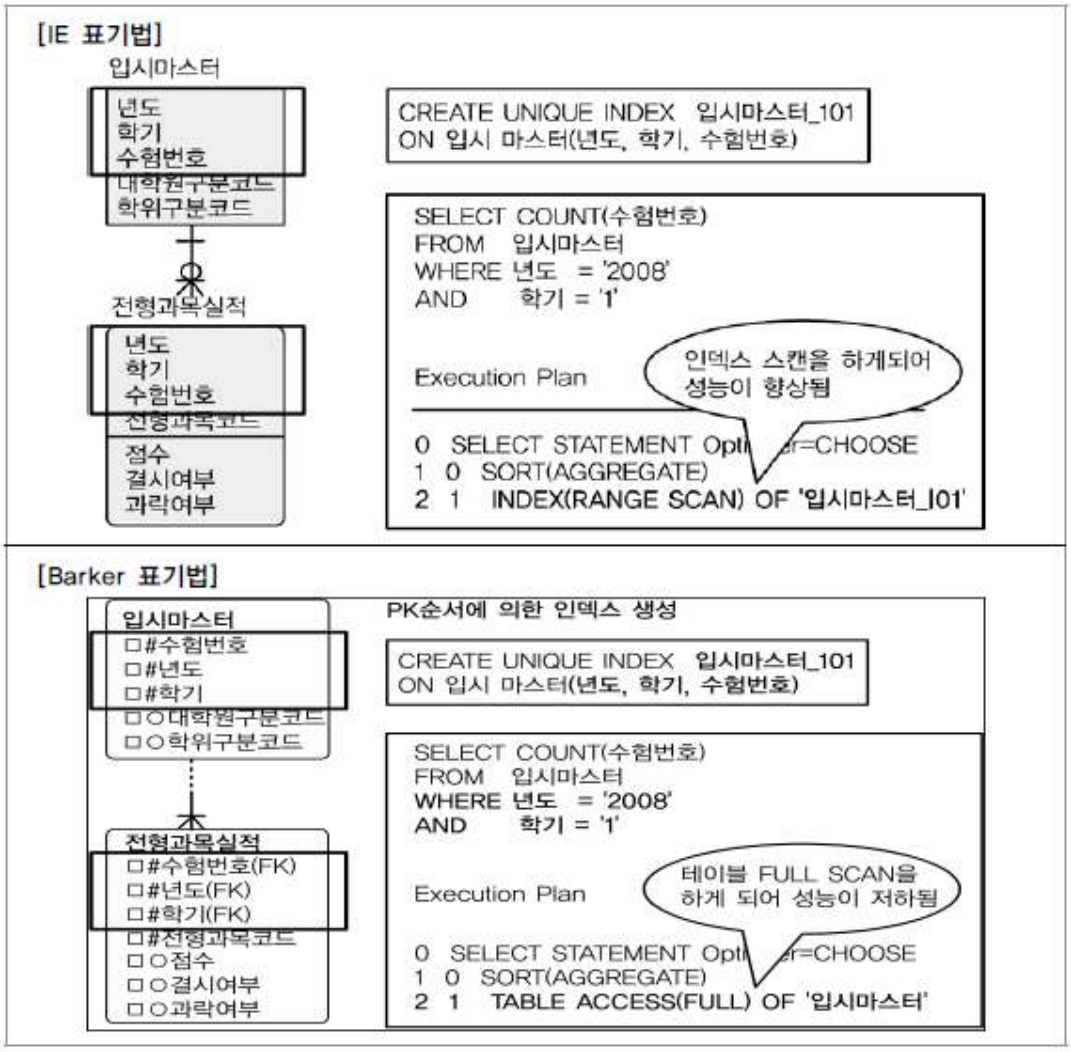
[그림 Ⅰ-2-32]의 예에서는 주문번호에 대한 비교값이 들어오지 않으므로 인해 인덱스 전체를 읽어야만 원하는 데이터를 찾을 수 있게 된다. 이러한 이유로 인덱스를 읽고 테이블 블록에서 읽어 처리하는데 I/O가 많이 발생하게 되므로 옵티마이저는 차라리 테이블에 가서 전체를 읽는 방식으로 처리하게 된다.

이러한 모습으로 인덱스의 정렬구조를 이해한 상태에서 인덱스에 접근하는 접근유형을 비교해보면 어떠한 인덱스를 태워야 하는지 어떠한 조건이 들어와야 데이터를 처리하는 양을 줄여 성능을 향상시킬 수 있는지 알 수 있게 된다.

정리하면, PK의 순서를 인덱스 특징에 맞게 고려하지 않고 바로 그대로 생성하게 되면, 테이블에 접근하는 트랜잭션의 특징에 효율적이지 않은 인덱스가 생성되어 있으므로 인덱스의 범위를 넓게 이용하거나 Full Scan을 유발하게 되어 성능이 저하된다고 정리할 수 있다. 다음은 실전 프로젝트에서 발생되는 예를 통해 PK나 FK의 성능저하 사항을 알아보도록 한다.

다. PK순서를 잘못 지정하여 성능이 저하된 경우 - 간단한 오류

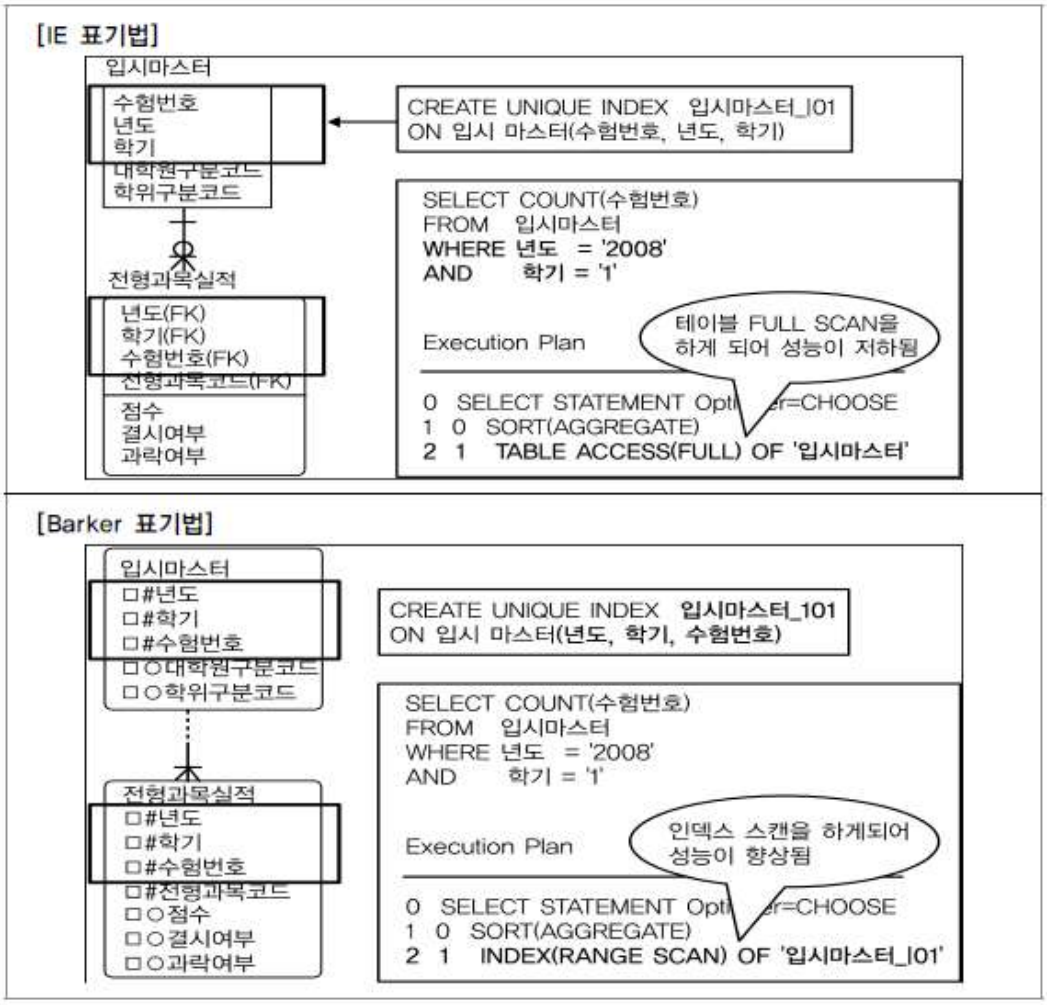
입시마스터라는 테이블의 PK는 수험번호+년도+학기로 구성되어 있고 전형과목실적 테이블은 입시마스터 테이블에서 상속받은 수험번호+년도+학기에 전형과목코드로 PK가 구성되어 있는 복합식별자 구조의 테이블이다. 입시마스터에는 200만 건의 데이터가 있고 학사는 4학기로 구성되어 있고 데이터는 5년간 보관되어 있다. 그러므로 한 학기당 평균 2만 건의 데이터가 있다고 가정하자.



[그림 Ⅰ-2-33] PK순서에 의한 인덱스 생성 개선 전

이 테이블 구조에서 다음과 같은 SQL구문이 실행되면 입시마스터 테이블에 있는 인덱스 입시마스터_I01을 이용할 수 있을까?

```
SELECT COUNT(수험번호)
  FROM 입시마스터
WHERE 년도 = '2008'
   AND 학기 = '1'
```



[그림 Ⅰ-2-34] PK순서에 의한 인덱스 생성 개선 후

입시마스터_I01 인덱스가 수험번호+년도+학기 중 수험번호에 대한 값이 WHERE절에 들어오지 않으므로 FULL TABLE SCAN이 발생, 200만 건의 데이터를 모두 읽게 되어 성능이 저하되었다.

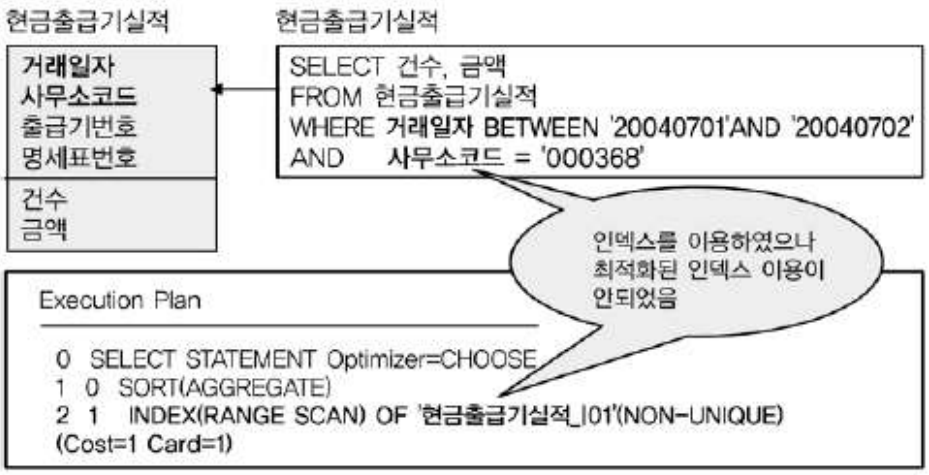
입시마스터 테이블에 데이터를 조회할 때 년도와 학기에 대한 내용이 빈번하게 들어오므로 [그림 Ⅰ-2-34]와 같이 PK순서를 변경함으로써 인덱스를 이용 가능하도록 할 수 있다. 즉, 생성된 인덱스가 정상적으로 이용이 되어 평균 2만 건의 데이터를 처리함으로써 성능이 개선된 모습이다.

라. PK순서를 잘못 지정하여 성능이 저하된 경우 - 복잡한 오류

현금출급기실적의 PK는 거래일자+사무소코드+출급기번호+명세표번호로 되어 있는데 대부분의 SQL문장에서는 조회를 할 때 사무소코드가 '='로 들어오고 거래일자에 대해서는 'BETWEEN' 조회를 하고 있다. 이 때 SQL은 정상적으로 인덱스를 이용할 수 있지만 인덱스 효율이 떨어져 성능이 저하되는 경우에 해당된다.

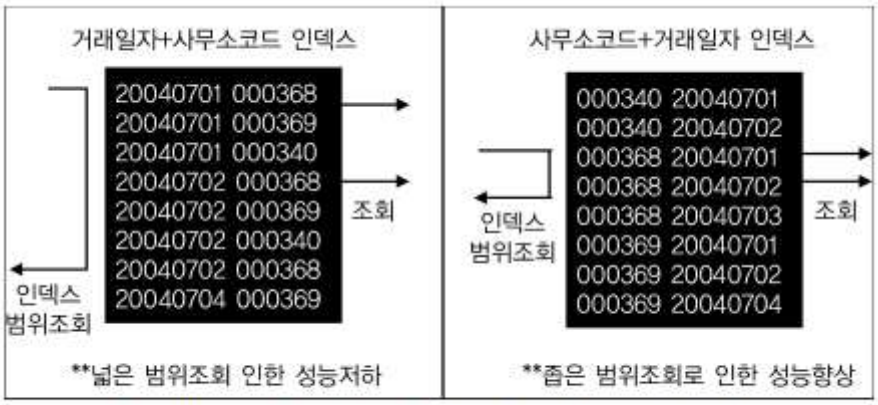
해당 테이블에 발생하는 SQL은 다음과 같이 작성되었다.

```
SELECT 건수, 금액
  FROM 현금출급기실적
WHERE 거래일자 BETWEEN '20040701'
   AND '20040702' AND 사무소코드 = '000368'
```



[그림 Ⅰ-2-35] PK순서에 의한 인덱스 생성

실행계획을 분석해 보면 인덱스가 정상적으로 이용되었기 때문에 SQL문장은 튜닝이 잘된 것으로 착각할 수 있다. 문제는 인덱스를 이용하기는 하는데 얼마나 효율적으로 이용하는지 검증이 필요하다. 아래 그림은 거래일자+사무소코드 순서로 인덱스를 구성한 경우와 사무소코드+거래일자 순서로 인덱스를 구성한 경우 데이터를 처리하는 범위의 차이를 보여주는 그림이다. 거래일자+사무소코드로 구성된 그림을 보면 BETWEEN 비교를 한 거래일자 '20040701'이 인덱스의 앞에 위치하기 때문에 범위가 넓어졌고 사무소코드+거래일자로 구성된 인덱스의 경우 '='비교를 한 사무소코드 '000368'이 인덱스 앞에 위치하여 범위가 좁아졌다.



[그림 Ⅰ-2-36] PK순서에 의한 인덱스 이용 범위

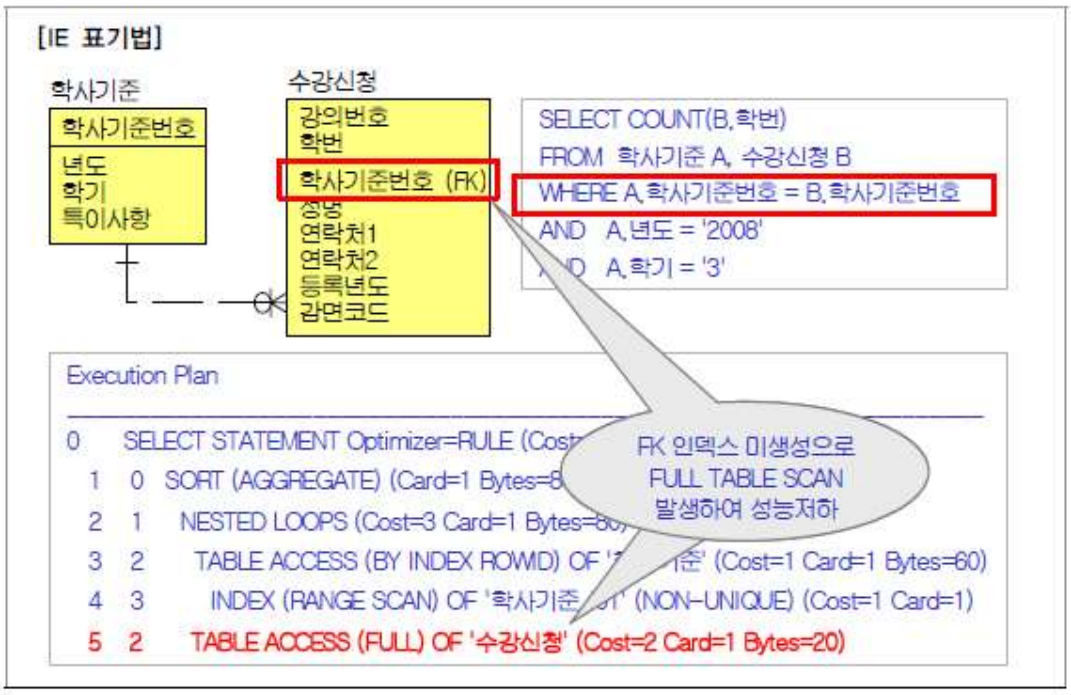
그러므로 이 경우 인덱스순서를 고려하여 데이터 모델의 PK순서를 거래일자+사무소코드+출급기번호+명세표번호에서 사무소코드+거래일자+출급기번호+명세표번호로 수정하여 성능을 개선할 수 있다. 물론 테이블의 PK구조를 그대로 둔 상태에서 인덱스만 하나 더 만들어도 성능은 개선될 수 있다. 이 때 이미 만들어진 PK 인덱스가 전혀 사용되지 않는다면 입력, 수정, 삭제시 불필요한 인덱스로 인해 더 성능이 저하되어 좋지 않다. 최적화된 인덱스 생성을 위해 PK순서변경을 통한 인덱스 생성이 바람직하다.

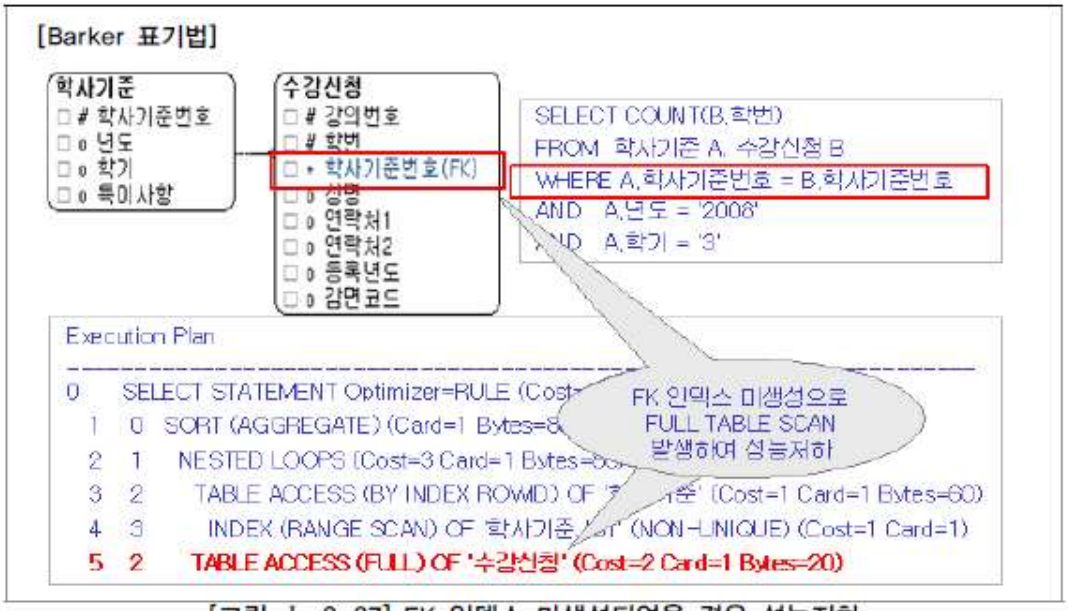
그러면 테이블의 PK의 속성이 A, B가 있을 때 A+B형태로도 빈번하게 조회가 되고 B+A로도 빈번하게 조회되는 경우에는 어떻게 할 것인가? 이 때는 좀 더 자주 이용되는 조회의 형태대로 PK순서를 구성하여 이용하게 하고 순서를 바꾼 인덱스를 추가로 생성하는 것이 필요하다.

3. 물리적인 테이블에 FK제약이 걸려있지 않을 경우 인덱스 미생성으로 성능저하

물리적인 테이블에 FK를 사용하지 않아도 데이터 모델 관계에 의해 상속받은 FK속성들은 SQL WHERE 절에서 조인으로 이용되는 경우가 많이 있으므로 FK 인덱스를 생성해야 성능이 좋은 경우가 빈번하다.

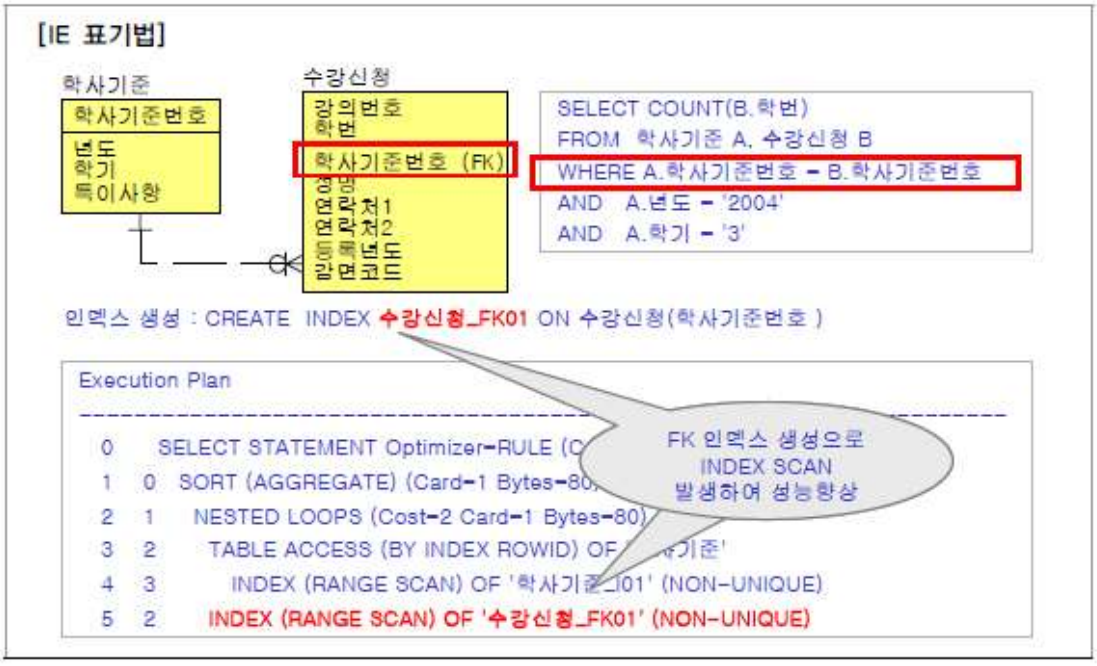
다음 그림은 학사기준과 수강신청에 대한 데이터 모델이다. 물리적인 테이블에는 두 테이블사이에 FK 참조무결성 관계가 걸려 있지 않는다고 가정한다. 또한 학사기준에는 데이터가 5만 건이 있고 수강신청에 데이터가 500만 건이 있다고 가정하자.

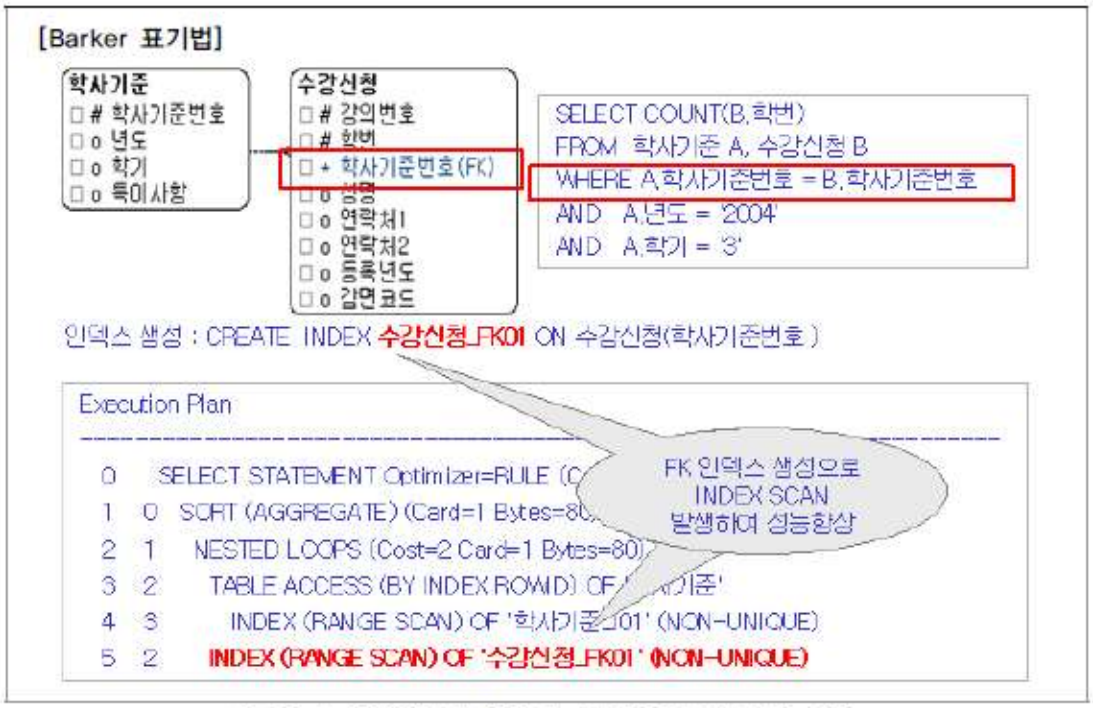




[그림 Ⅰ-2-37] FK 인덱스 미생성되었을 경우 성능저하

비록 수강신청 테이블에 있는 학사기준번호가 SQL WHERE 절에 비교자로 들어오지는 않았지만 수강신청 테이블에서 상속받은 학사기준번호에 대해 인덱스를 생성하지 않으므로 인해 학사기준과 수강신청 테이블이 조인이 되면서 500만 건의 수강신청 테이블이 FULL TABLE SCAN이 발생되어 성능이 저하되었다. 이 때는 수강신청 테이블에 FK 인덱스를 생성하여 성능을 개선할 수 있다.





[그림 Ⅰ-2-38] FK 인덱스 생성으로 성능저하 해결

비록 물리적으로 학사기준과 수강신청이 연결되어 있지 않다고 하더라도 학사기준으로부터 상속받은 FK에 대해 FK인덱스를 생성함으로써 SQL문장이 조인이 발생할 때 성능저하를 예방할 수 있다.

FK인덱스를 적절하게 설계하여 구축하지 않았을 경우 개발초기에는 데이터량이 얼마 되지 않아 성능저하가 나타나지 않다가 시스템을 오픈하고 데이터량이 누적될수록 SQL성능이 나빠짐으로 인해 데이터베이스서버에 심각한 장애현상을 초래하는 경우가 많이 있다.

그러므로 물리적인 테이블에 FK 제약 걸었을 때는 반드시 FK인덱스를 생성하도록 하고 FK제약이 걸리지 않았을 경우에는 FK인덱스를 생성하는 것을 기본정책으로 하되 발생되는 트랜잭션에 의해 거의 활용되지 않았을 때에만 FK 인덱스를 지우는 방법으로 하는 것이 적절한 방법이 된다.

제6절 분산 데이터베이스와 성능

1. 분산 데이터베이스의 개요

1990년대에는 데이터베이스를 분산하여 저장하고 그것을 하나의 데이터베이스로 인식하여 사용하는 기술은 아주 난이도가 높은 고급기술로 인식되었다. 2000년도에 클라우드 컴퓨팅, SOA를 인식하듯 분산 데이터베이스를 인식하고 연구·도입하려는 기업이 많았었다. DBMS의 기능이 강해지고 네트워크 속도가 빨라지면서 분산 데이터베이스가 초기에 예상한 만큼 확산되지는 않았지만, 여전히 많은 데이터베이스는 네트워크를 통한 데이터베이스 간의 공유체계를 통해 분산 데이터베이스를 활용하고 있다.
분산데이터베이스의 정의는 다음과 같다.

- 여러 곳으로 분산되어있는 데이터베이스를 하나의 가상 시스템으로 사용할 수 있도록 한 데이터베이스
- 논리적으로 동일한 시스템에 속하지만, 컴퓨터 네트워크를 통해 물리적으로 분산되어 있는 데이터들의 모임. 물리적 Site 분산, 논리적으로 사용자 통합·공유

즉, 분산 데이터베이스는 데이터베이스를 연결하는 빠른 네트워크 환경을 이용하여 데이터베이스를 여러 지역 여러 노드로 위치시켜 사용성/성능 등을 극대화 시킨 데이터베이스라고 정의할 수 있다.

2. 분산 데이터베이스의 투명성(Transparency)

분산데이터베이스가 되기 위해서는 6가지 투명성(Transparency)을 만족해야 한다.

1) 분할 투명성 (단편화) : 하나의 논리적 Relation이 여러 단편으로 분할되어 각 단편의 사본이 여러 site에 저장
2) 위치 투명성 : 사용하려는 데이터의 저장 장소 명시 불필요. 위치정보가 System Catalog에 유지되어야 함
3) 지역사상 투명성 : 지역DBMS와 물리적 DB사이의 Mapping 보장. 각 지역시스템 이름과 무관한 이름 사용 가능
4) 중복 투명성 : DB 객체가 여러 site에 중복 되어 있는지 알 필요가 없는 성질
5) 장애 투명성 : 구성요소(DBMS, Computer)의 장애에 무관한 Transaction의 원자성 유지
6) 병행 투명성 : 다수 Transaction 동시 수행시 결과의 일관성 유지, Time Stamp, 분산 2단계 Locking을 이용 구현

전통적인 분산 데이터베이스 구축과 같이, 분산 환경의 데이터베이스를 위와 같은 특징 모두를 만족하면서 구축하는 사례는 최근에는 드물다. 최근에는 분산 환경의 데이터베이스를 구축하기보다 통합하여 데이터베이스를 구축하는 사례가 더 많이 있다. 그럼에도 불구하고 위의 분산 환경의 데이터베이스를 업무적인 특징 및 지역적인 특징에 따라 적절하게 활용하기만 하면, 다양한 장점을 제공하는 특징을 가지고 있기 때문에 대량 데이터처리의 지역적 처리나 글로벌 처리 등에서는 분산 데이터베이스가 유용하게 활용되고 있다.

3. 분산 데이터베이스의 적용 방법 및 장단점

가. 분산 데이터베이스 적용방법

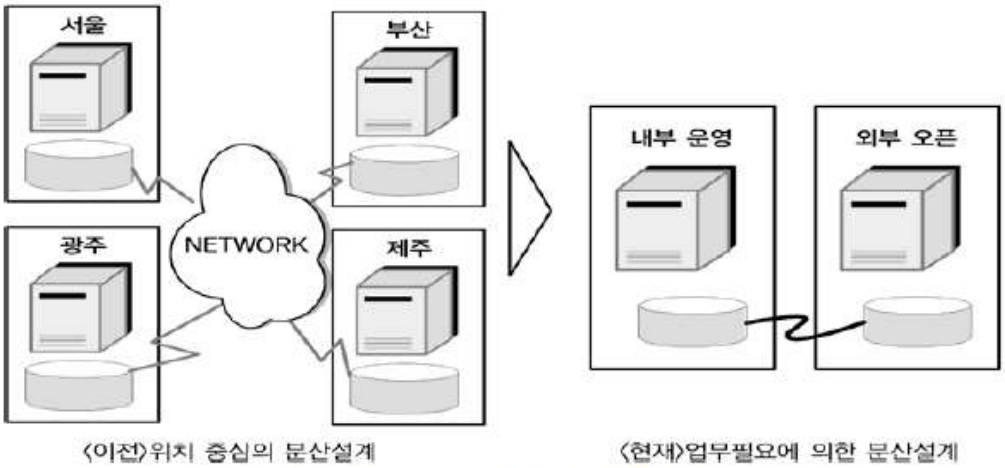
분산 환경의 데이터베이스를 성능이 우수하게 현장에서 가치 있게 사용하는 방법은 업무의 흐름을 보고 업무구성에 따른 아키텍처 특징에 따라 데이터베이스를 구성하는 것이다. 단순히 분산 환경에서 데이터베이스를 구축하는 것이 목적이 아니라, 업무의 특징에 따라 데이터베이스 분산구조를 선택적으로 설계하는 능력이 필요한 것이다. 이러한 측면만을 보았을 때는 데이터베이스 분산설계라는 측면보다는 데이터베이스 구조설계(아키텍처)라는 의미로 이해해도 무방할 것이다.

나. 분산 데이터베이스 장단점

장점	단점
- 지역 자치성, 점증적 시스템 용량 확장 - 신뢰성과 가용성 - 효용성과 융통성 - 빠른 응답 속도와 통신비용 절감 - 데이터의 가용성과 신뢰성 증가 - 시스템 규모의 적절한 조절 - 각 지역 사용자의 요구 수용 증대	- 소프트웨어 개발 비용 - 오류의 잠재성 증대 - 처리 비용의 증대 - 설계, 관리의 복잡성과 비용 - 불규칙한 응답 속도 - 통제의 어려움 - 데이터 무결성에 대한 위협

4. 분산 데이터베이스의 활용 방향성

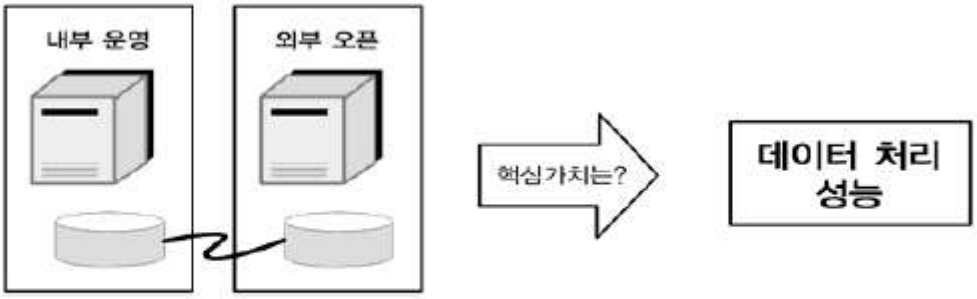
분산 데이터베이스는 업무적인 기능이 다양해지고 데이터의 양이 기하급수적으로 증가하는 최근 데이터베이스 환경에서 적용하는 고급화된 기술이다. 업무적인 특징에 따라 분산 데이터베이스를 활용하는 기술이 필요하다.



[그림 Ⅰ-2-39] 분산설계 방향성

5. 데이터베이스 분산구성의 가치

데이터를 분산 환경으로 구성하였을 때 가장 핵심적인 가치는 바로 통합된 데이터베이스에서 제공할 수 없는 빠른 성능을 제공한다는 점이다. 원거리 또는 다른 서버에 접속하여 처리하므로 인해 발생되는 네트워크 부하 및 트랜잭션 집중에 따른 성능 저하의 원인을 분산된 데이터베이스 환경을 구축하므로 빠른 성능을 제공하는 것이 가능해 진다. 바로 이 점 때문에 분산 환경의 데이터베이스를 구축하게 되는 것이다.



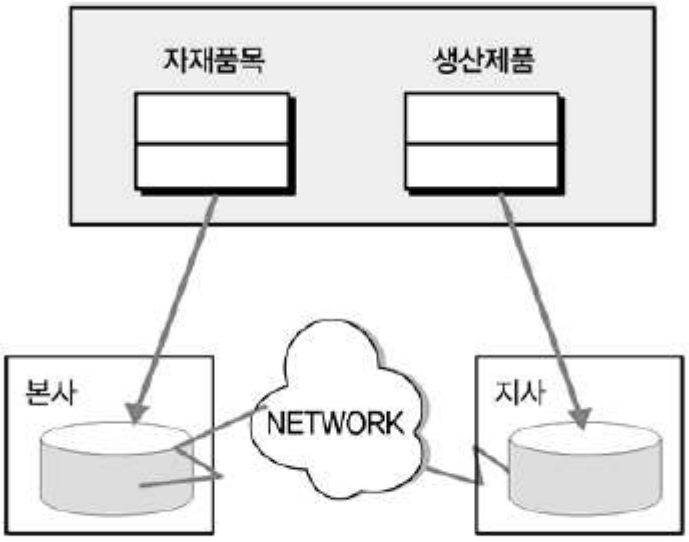
[그림 Ⅰ-2-40] 분산 데이터베이스 핵심가치

6. 분산 데이터베이스의 적용 기법

데이터베이스의 분산의 종류에는 테이블 위치 분산과 테이블 분할 분산, 테이블 복제 분산, 테이블 요약 분산 전략이 있다. 그 중에서도 가장 많이 사용하는 방식의 테이블의 복제 분할 분산의 방법이고 이 방법은 성능이 저하되는 많은 데이터베이스에서 가장 유용하게 적용할 수 있는 기술적인 방법이 된다. 분산 환경으로 데이터베이스를 설계하는 방법은 일단 통합 데이터 모델링을 하고 각 테이블별로 업무적인 특징에 따라 지역 또는 서버별로 테이블을 분산 배치나 복제 배치하는 형태로 설계할 수 있다.

가. 테이블 위치 분산

테이블 위치 분산은 테이블의 구조는 변하지 않는다. 또한 테이블이 다른 데이터베이스에 중복되어 생성되지도 않는다. 다만 설계된 테이블의 위치를 각각 다르게 위치시키는 것이다. 예를 들어, 자재품목은 본사에서 구입하여 관리하고 각 지사별로 자재품목을 이용하여 제품을 생산한다고 하면 [그림 Ⅰ-2-41]과 같이 데이터베이스를 본사와 지사단위로 분산시킬 수 있다.



[그림 Ⅰ-2-41] 테이블별 위치 분산

[그림 Ⅰ-2-41]의 분산방법은 설계된 테이블 각각이 지역별로 분산되어 생성되는 경우이다.

각각의 테이블마다 위치가 다르게 지정되어야 한다면 [그림 Ⅰ-2-42]의 표와 같이 각각 테이블마다 위치를 표기하여 테이블을 생성하도록 한다.

테이블 / 위치	자재품목	생산제품	협력회사	사원	부서
본사	•		•		•
지사		•		•	

[그림 Ⅰ-2-42] 테이블별 위치 분산

테이블별 위치 분산은 정보를 이용하는 형태가 각 위치별로 차이가 있을 경우에 이용한다. 테이블의 위치가 위치별로 다르므로 테이블의 위치를 파악할 수 있는 도식화된 위치별 데이터베이스 문서가 필요하다.

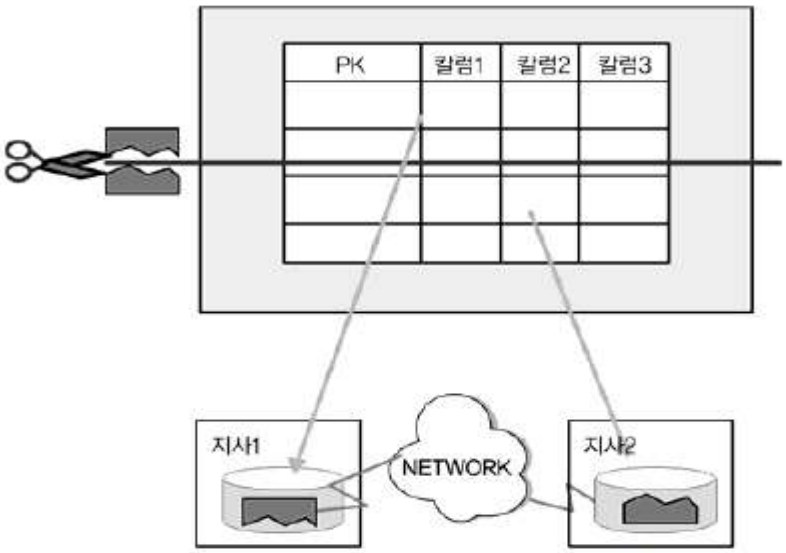
나. 테이블 분할(Fragmentation) 분산

테이블 분할 분산은 단순히 위치만 다른 곳에 두는 것이 아니라 각각의 테이블을 쪼개어 분산하는 방법이다. 테이블을 분할하여 분산하는 방법은 테이블을 나누는 기준에 따라 두

가지로 구분된다. 첫 번째는 테이블의 로우(Row)단위로 분리하는 수평분할(Horizontal Fragmentation)이 있고 두 번째는 테이블을 칼럼(Column) 단위로 분할하는 수직분할(Vertical Fragmentation)이 있다.

- 수평분할(Horizontal Fragmentation)

지사(Node)에 따라 테이블을 특정 칼럼의 값을 기준으로 로우(Row)를 분리한다. 칼럼은 분리되지 않는다. 모든 데이터가 각 지사별로 분리되어 있는 형태를 가지고 있다. 각 지사에 있는 데이터와 다??며, 데이터를 한군데 집합시켜 놓아도 Primary Key에 의해 중복이 발생되지 않는다.



[그림 Ⅰ-2-43] 테이블 분할 분산 - 수평분할

이와 같이 수평분할을 이용하는 경우는 각 지사(Node)별로 사용하는 로우(Row)가 다를 때 이용한다. 데이터를 수정할 때는 타 지사에 있는 데이터를 원칙적으로 수정하지 않고 자신의 데이터에 대해서 수정하도록 한다. 각 지사에 존재하는 테이블에 대해서 통합처리를 해야 하는 경우는 조인(JOIN)이 발생하여 성능 저하가 예상되므로 통합처리 프로세스가 많은지를 먼저 검토한 이후에 많지 않은 경우에 수평분할을 해야 한다.

데이터가 지사별로 별도로 존재하므로 중복은 발생하지 않는다. 대신 타 지사에 있는 데이터가 지사구분이 변경되면 단순히 수정이 발생하는 것 이외에 변경된 지사로 데이터를 이송해야 한다. 한 시점에는 한 지사(Node)에서 하나의 데이터만이 존재하므로 데이터의 무결성은 보장되는 형태이다.

지사(Node)별로 데이터베이스를 운영하는 경우는 데이터베이스가 속한 서버가 지사(Node)에 존재하던지 아니면 본사에 통합해서 존재하건 간에 데이터베이스 테이블들은 수평 분할하여 존재한다.

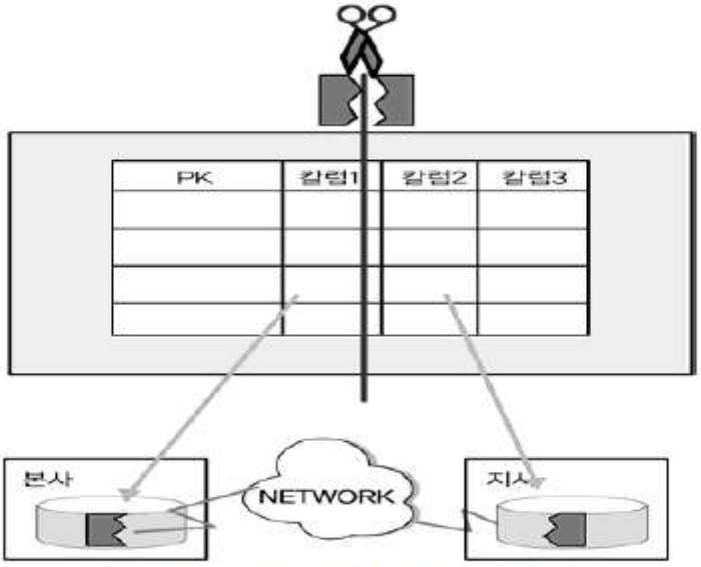
지사별로 운영하는 테이블들의 예를 들어보자. 테이블 고객, 생산제품, 협력회사, 사원, 부서 테이블이 지사1과 지사2에서 일의 시작과 끝이 항상 다르게 발생한다고 하면 각 테이블은 지사별로 수평 분할하여 [그림 Ⅰ-2-44]의 표와 같이 생성되게 된다.

위치 \ 테이블	고객	생산제품	협력회사	사원	부서
지사 1	◓	◓	◓	◓	◓
지사 2	◒	◒	◒	◒	◒

[그림 Ⅰ-2-44] 테이블 수평분할

- 수직분할(Vertical Fragmentation)

지사(Node)에 따라 테이블 칼럼을 기준으로 칼럼(Row)을 분리한다. 로우(Row) 단위로는 분리되지 않는다. 모든 데이터가 각 지사별로 분리되어 있는 형태를 가지고 있다. 칼럼을 기준으로 분할하였기 때문에 각각의 테이블에는 동일한 Primary Key구조와 값을 가지고 있어야 한다. 지사별로 쪼개어진 테이블들을 조합하면 Primary Key가 동일한 데이터의 조합이 가능해야 하며 하나의 완전한 테이블이 구성되어야 한다. 데이터를 한군데 집합시켜 놓아도 동일한 Primary Key는 하나로 표현하면 되므로 데이터 중복은 발생되지 않는다.



[그림 Ⅰ-2-45] 테이블 수직분할

예를 들어 제품의 재고량은 각 지사별로 관리하고 제품에 대한 단가는 본사에서 관리한다고 하면 본사 테이블에는 제품번호, 단가가 존재하고 지사에는 제품번호, 재고량이 존재한다. 이를 본사와 지사단위로 분리된 칼럼의 모습을 도식화 하면 다음과 같이 나타난다.

위치 \ 테이블	제품	분할 칼럼
본사	◐	제품번호, 단가
지사	◑	제품번호, 재고량

[그림 Ⅰ-2-46] 테이블 수직분할 모델링

테이블의 전체 칼럼 데이터를 보기 위해서는 각 지사(Node)별로 흩어져 있는 테이블들을 조인(JOIN)하여 가져와야 하므로 가능하면 통합하여 처리하는 프로세스가 많은 경우에는 이용하지 않도록 한다. 일반적으로 실제 프로젝트에서는 이와 같이 칼럼을 쪼개는 테이블의 수직분할 분산 환경을 구성하는 사례는 드물다.

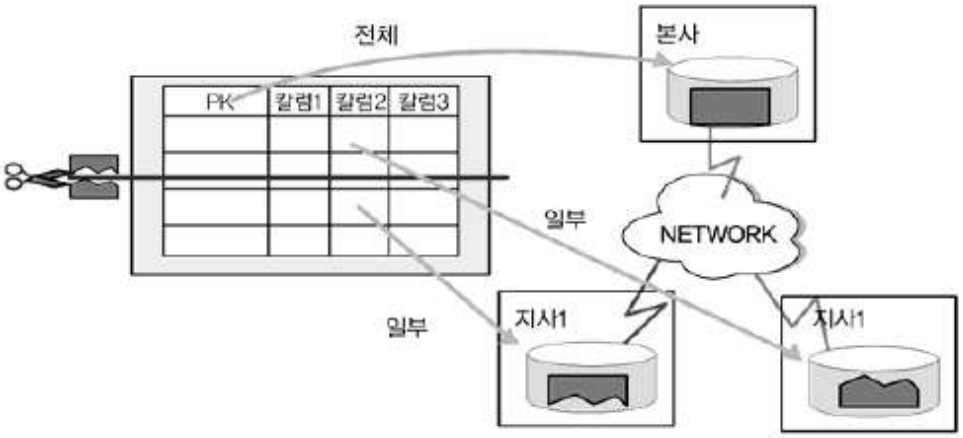
다. 테이블 복제(Replication) 분산

테이블 복제(Replication) 분산은 동일한 테이블을 다른 지역이나 서버에서 동시에 생성하여 관리하는 유형이다.

마스터 데이터베이스에서 테이블의 일부의 내용만 다른 지역이나 서버에 위치시키는 부분복제(Segment Replication)가 있고 마스터 데이터베이스의 테이블의 내용을 각 지역이나 서버에 존재시키는 광역복제(Broadcast Replication)가 있다.

- **부분복제(Segment Replication)**

통합된 테이블을 한군데(본사)에 가지고 있으면서 각 지사별로는 지사에 해당된 로우(Row)를 가지고 있는 형태이다. 지사에 존재하는 데이터는 반드시 본사에 존재하게 된다. 즉 본사의 데이터는 지사데이터의 합이 되는 것이다. 각 지사에서 데이터 처리가 용이할 뿐만 아니라 전체 데이터에 대한 통합처리도 본사에 있는 통합 테이블을 이용하게 되므로 여러 테이블에 조인(JOIN)이 발생하지 않는 빠른 작업 수행이 가능해진다.



[그림 Ⅰ-2-47] 테이블 복제 분산 - 부분복제

[그림 Ⅰ-2-47]을 보면, 본사 데이터베이스에 있는 테이블에는 테이블의 전체 내용이 들어가고 각 지사 데이터베이스에 있는 테이블에는 지사별로 관계된 데이터만 들어가게 된다. 수평분할 분산과 마찬가지로 지사간에는 데이터의 중복이 발생하지 않으나 본사와 지사간에는 데이터의 중복이 항상 발생하게 되는 경우이다.

보통 전국에 있는 고객을 관리할 때 본사에는 전국고객에 대한 정보를 관리하고 지사에는 각 지사와 거래하는 고객정보를 관리한다. 본사의 데이터를 이용하여 통계, 이동 등을 관리하며 지사에 있는 데이터를 이용하여 지사별로 빠른 업무수행을 한다. 보통 지사에 데이터가 먼저 발생하고 본사에 데이터는 지사에 데이터를 이용하여 통합하여 발생된다.

위치 \ 테이블	고객
본사	●
지사 1	◐
지사 2	◑

본사에는 전국의 고객정보를 관리하고 지사 1의 고객테이블에서는 지사 1에 속한 고객정보를 지사 2의 고객테이블에서는 지사 2에 속한 고객정보를 관리한다.

[그림 Ⅰ-2-48] 테이블 복제 분산 - 부분복제

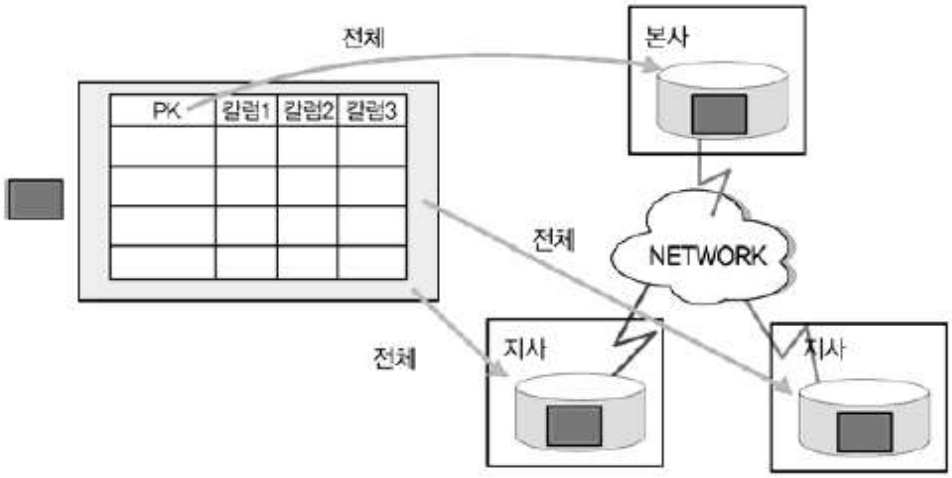
실제 프로젝트에서 많이 사용하는 데이터베이스 분산기법에 해당한다. 각 지사별로 업무수행이 용이하고 본사에 있는 데이터를 이용하여 보고서를 출력하거나 통계를 산정하는 등 다양한 업무형태로 이용 가능하다.

다른 지역간의 데이터를 복제(Replication)하는데 많은 시간이 소요되고 데이터베이스와 서버에 부하(Load)가 발생하므로 보통 실시간(On-Line) 처리에 의해 복사하는 것보다는 야간에 배치 작업에 의해 수행되는 경우가 많이 있다.

또한 본사와 지사 양쪽 모두 데이터를 수정하여 전송하는 경우 데이터의 정합성을 일치시키는 것이 어렵기 때문에 가능하면 한쪽(지사)에서 데이터의 수정이 발생하여 본사로 복제(Replication)를 하도록 한다.

- 광역복제(Broadcast Replication)

통합된 테이블을 한군데(본사)에 가지고 있으면서 각 지사에도 본사와 동일한 데이터를 모두 가지고 있는 형태이다. 지사에 존재하는 데이터는 반드시 본사에 존재하게 된다. 모든 지사에 있는 데이터량과 본사에 있는 데이터량이 다 동일하다. 본사와 지사모두 동일한 정보를 가지고 있으므로 본사나 지사나 데이터처리에 특별한 제약을 받지는 않는다.



[그림 Ⅰ-2-49] 테이블 복제 분산 - 광역복제

예를 들어, 본사에서 코드테이블에 데이터에 대해 입력, 수정, 삭제가 발생하고 각 지사에서는 코드데이터를 이용하는 프로세스가 발생한다. 즉 본사에서는 데이터를 관리하고 지사에서는 이 데이터를 읽어 업무프로세스를 발생시키는 것이다.

위치 \ 테이블	코드
본사	●
지사 1	●
지사 2	●

본사, 지사1, 지사2 모두 동일한 양의 코드 테이블의 데이터를 가지고 있다.

[그림 Ⅰ-2-50] 테이블 복제 분산 - 광역복제

광역복제(Broadcast Replication) 역시 실제 프로젝트에서 많이 사용하는 데이터베이스 분산기법에 해당한다. 부분복제의 경우는 지사에서 데이터에 대한 입력, 수정, 삭제가 발생하여 본사에서 이용하는 방식이 많은 반면 광역복제(Broadcast Replication)의 경우에는 본사에서 데이터가 입력, 수정, 삭제가 되어 지사에서 이용하는 형태가 차이점이다.

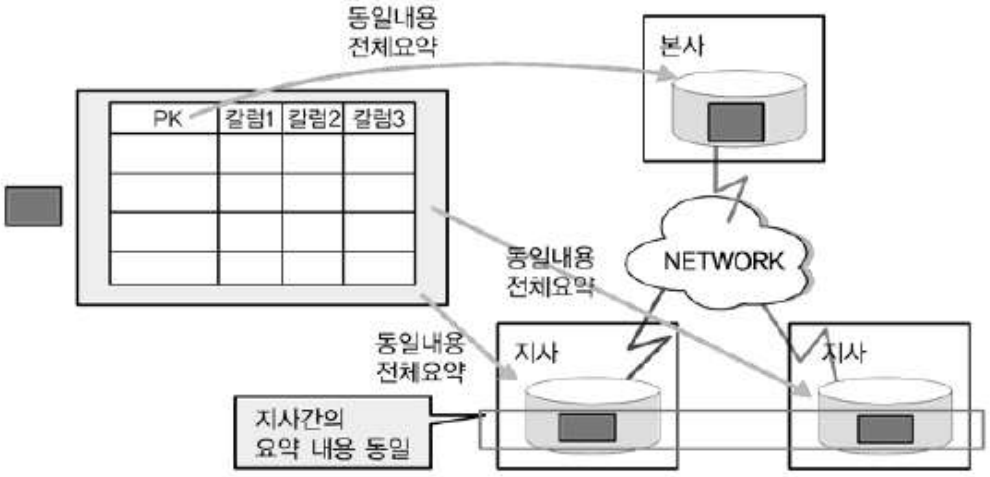
부분복제와 마찬가지로 데이터를 복제(Replication)하는데 많은 시간이 소요되고 데이터베이스와 서버에 부하(Load)가 발생하므로 보통 실시간(On-Line) 처리에 의해 복사하는 것보다는 배치에 의해 복제가 되도록 한다.

라. 테이블 요약(Summarization) 분산

테이블 요약(Summarization) 분산은 지역간에 또는 서버 간에 데이터가 비슷하지만 서로 다른 유형으로 존재하는 경우 있다. 요약의 방식에 따라, 동일한 테이블 구조를 가지고 있으면서 분산되어 있는 동일한 내용의 데이터를 이용하여 통합된 데이터를 산출하는 방식의 분석요약(Rollup Summarization)과 분산되어 있는 다른 내용의 데이터를 이용하여 통합된 데이터를 산출하는 방식의 통합요약(Consolidation Summarization)이 있다.

- 분석요약(Rollup Replication)

분석요약(Rollup Replication)은 각 지사별로 존재하는 요약정보를 본사에 통합하여 다시 전체에 대해서 요약정보를 산출하는 분산방법이다.



[그림 Ⅰ-2-51] 테이블 요약 분산 - 분석요약

[그림 Ⅰ-2-51]에서 보면, 테이블에 있는 모든 칼럼(Column)과 로우(Row)가 지사에도 동일하게 존재하지만, 각 지사에는 동일한 내용에 대해 지사별로 요약되어 있는 정보를 가지고 있고 본사에는 각 지사의 요약정보를 통합하여 재산출하여 전체에 대한 요약정보를 가지고 있는 것으로 표시되어 있다.

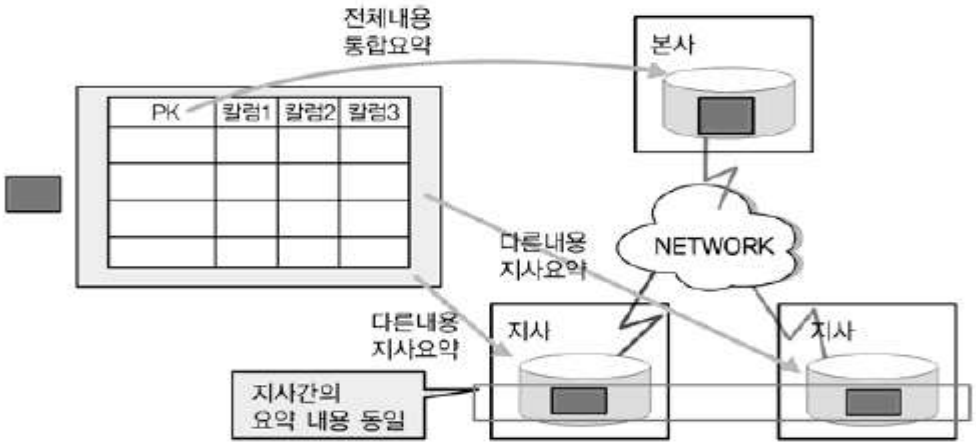
예를 들어, 제품별 판매실적이라는 테이블이 존재한다고 가정하자. 각 지사에서는 취급제품이 동일하다. 지사별로 판매된 제품에 대해서 지사별로 판매실적이 관리된다. 지사1과 지사2에도 동일한 제품이 취급이 되므로 이를 본사에서 판매실적을 집계할 경우에는 통합된 판매실적을 관리할 수 있는 것이다.

[그림 Ⅰ-2-52] 테이블 요약 분산 – 분석요약

각종 통계데이터를 산정할 경우에, 모든 지사의 데이터를 이용하여 처리하면 성능이 지연되고 각 지사 서버에 부하를 주기 때문에 업무에 장애가 발생할 수 있다. 통합 통계데이터에 대한 정보제공에 용이한 분산방법이다. 본사에 분석 요약된 테이블을 생성하고 데이터는 역시 일반 업무가 종료되는 야간에 수행하여 생성한다.

- 통합요약(Consolidation Replication)

통합요약(Consolidation Replication)은 각 지사별로 존재하는 다른 내용의 정보를 본사에 통합하여 다시 전체에 대해서 요약정보를 산출하는 분산방법이다.



[그림 Ⅰ-2-53] 테이블 요약 분산 – 통합요약

[그림 Ⅰ-2-53]에서 보면, 테이블에 있는 모든 칼럼(Column)과 로우(Row)가 지사에도 동일하게 존재하지만 각 지사에는 타지사와 다른 요약정보를 가지고 있고 본사에는 각 지사의 요약정보를 데이터를 같은 위치에 두는 것으로 통합하여 전체에 대한 요약정보를 가지고 있는 것으로 표시된다.

위치 \ 테이블	판매실적
본사	●
지사 1	◓
지사 2	◒

지사 1과 지사 2에 판매실적이 존재하지만 서로 다른 내용으로 존재한다. 본사에서는 모든 지사의 판매실적을 통합한 실적 데이터가 생성된다.

[그림 Ⅰ-2-54] 테이블 요약분산 - 통합요약

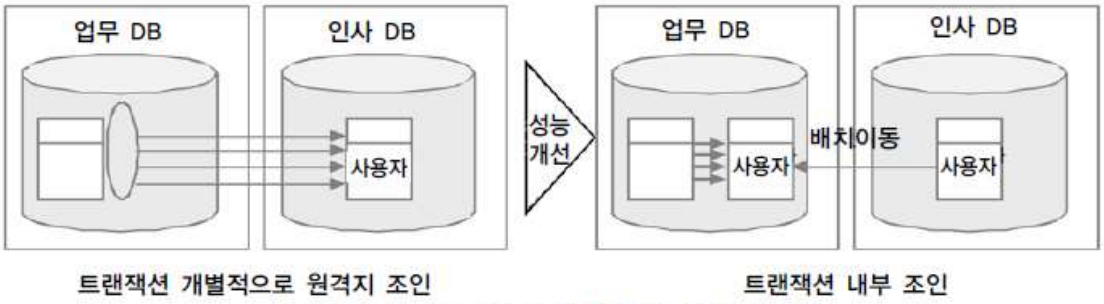
본사에 통계데이터를 산정하는 유형은 분석요약과 비슷하나 통합요약은 단지 지사에서 산출한 요약정보를 한군데 취합하여 보여주는 형태이다. 분석요약은 지사에 있는 데이터를 이용하여 본사에서 통합하여 요약 데이터를 산정하였지만 통합요약에서는 지사에서 요약한 정보를 본사에서 취합하여 각 지사별로 데이터를 비교하기 위해 이용되는 것이다.

각종 통계데이터를 산정할 경우에, 모든 지사의 데이터를 조인하여 처리하면 성능이 지연되고 각 지사 서버에 부하(LOAD)를 주기 때문에 업무에 장애가 발생할 수 있다. [그림 Ⅰ-2-54]와 같은 방법은 통합 통계데이터에 대한 정보제공에 용이한 분산방법이다. 본사에 통합 요약된 테이블을 생성하고 데이터는 역시 일반 업무가 종료되는 야간에 수행하여 생성하는 것이 일반적인 적용방법이다.

7. 분산 데이터베이스를 적용하여 성능이 향상된 사례

프로젝트를 수행할 때 단순한 분산 환경의 원리를 이해하지 않고 데이터베이스를 설계하여 성능이 저하되는 경우가 빈번하다. 특히 복제분산의 원리를 간단하게 응용하면 많은 업무적인 특성이 있는 곳에서 그 성능을 향상시켜 설계할 수 있다.

[그림 Ⅰ-2-55]는 개인정보를 관리하는 데이터베이스가 인사 데이터베이스일 때 분산이 안된 경우의 각 서버에 독립적으로 테이블이 있을 때 트랜잭션과 복제분산을 통해 테이블의 정보가 양쪽에 있을 때 트랜잭션 처리의 특성을 보여주는 그림이다. 단순한 개념도 이지만 위의 원리가 복잡한 업무처리에서 효과적으로 성능을 향상시킬 수 있음을 주목해야 한다.



[그림 Ⅰ-2-55] 업무 특성에 따른 분산환경 구성

데이터베이스 분산 설계는 다음과 같은 경우에 적용하면 효과적이다.

- 성능이 중요한 사이트에 적용해야 한다.
- 공통코드, 기준정보, 마스터 데이터 등에 대해 분산환경을 구성하면 성능이 좋아진다.
- 실시간 동기화가 요구되지 않을 때 좋다. 거의 실시간(Near Real Time)의 업무적인 특징을 가지고 있을 때도 분산 환경을 구성할 수 있다
- 특정 서버에 부하가 집중이 될 때 부하를 분산할 때도 좋다.
- 백업 사이트(Disaster Recovery Site)를 구성할 때 간단하게 분산기능을 적용하여 구성할 수 있다.

과목 2 SQL 기본 및 활용

제1절 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 곳에서나 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 개개인의 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, DB2와 같은 제품이 사용되었다.
- 1990년대 : Oracle, Sybase, Informix, DB2, Teradata, SQL Server 외 많은 제품들이 보다 향상된 기능으로 정보시스템의 확실한 핵심 솔루션으로 자리잡게 되었으며, 인터넷 환경의 급속한 발전과 객체 지향 정보를 지원하기 위해 객체 관계형 데이터베이스로 발전하였다.

- **관계형 데이터베이스(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 저장이 되고, 관계형 데이터베이스를 유일하게 조작할 수 있는 SQL 문장에 의해 관리되고 있으므로 관계형 데이터베이스와 SQL의 중요성은 아무리 강조해도 지나치지 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 입력한다든지, 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.

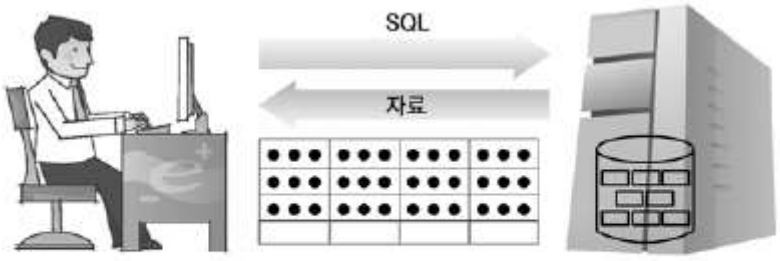
1986년부터 ANSI/ISO를 통해 표준화되고 정의된 SQL 기능은 벤더별 DBMS 개발의 목표가 된다. 일부 구체적인 용어는 다르더라도 대부분의 관계형 데이터베이스에서 ANSI/ISO 표준을 최대한 따르고 있기 때문에, SQL에 대한 지식은 다른 데이터베이스를 사용하더라도 상당 부분 기존 지식을 재활용할 수 있고, ANSI/IS0 SQL-99, SQL-2003 이후 기준이 적용된 SQL이라면 프로그램의 이식성을 높이는 데도 공헌한다.

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 SQL은 관계형 데이터베이스에 대한 전담 접속(다른 언어는 관계형 데이터베이스에 접속할 수 없다) 용도로 사용되며 독립되어 있다.

관계형 데이터베이스는 수학의 집합 논리에 입각한 것이므로, SQL도 데이터를 집합으로써 취급한다. 예를 들어 '포지션이 미드필더(MF)인 선수의 정보를 검색한다'고 할 경우, 선수라는 큰 집합에서 포지션이 미드필더인 조건을 만족하는 요구 집합을 추출하는 조작이 된다.

이렇게 특정 데이터들의 집합에서 필요로 하는 데이터를 꺼내서 조회하고 새로운 데이터를 입력/수정/삭제하는 행위를 통해서 사용자는 데이터베이스와 대화하게 된다. 그리고 SQL은 이러한 대화를 가능하도록 매개 역할을 하는 것이다. 결과적으로 SQL 문장을 배우는 것이 곧 관계형 데이터베이스를 배우는 기본 단계라 할 수 있다.



[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.

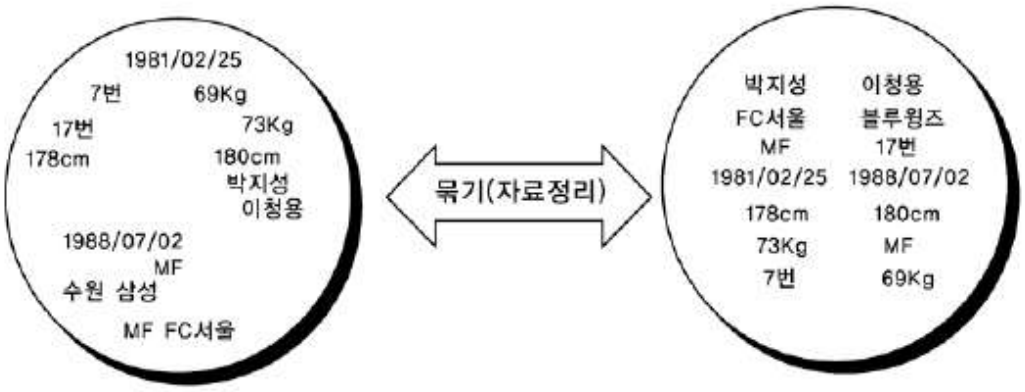
명령어의 종류	명령어	설명
데이터 정의어 (DDL: Data Definition Language)	CREATE ALTER DROP RENAME	테이블과 같은 데이터 구조를 정의하는데 사용되는 명령어들로 그러한 구조를 생성하거나 변경하거나 삭제하거나 이름을 바꾸는 데이터 구조와 관련된 명령어들을 DDL이라고 부른다.
데이터 제어어 (DCL: Data Control Language)	GRANT REVOKE	데이터베이스에 접근하고 객체들을 사용하도록 권한을 주고 회수하는 명령어를 DCL이라고 부른다.
트랜잭션 제어어 (TCL: Transaction Control Language)	COMMIT ROLLBACK	논리적인 작업의 단위를 묶어서 DML에 의해 조작된 결과를 작업단위(트랜잭션) 별로 제어하는 명령어를 말한다.

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

3. TABLE

월드컵 4강 및 16강으로 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 정리되어 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 추가함으로서 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

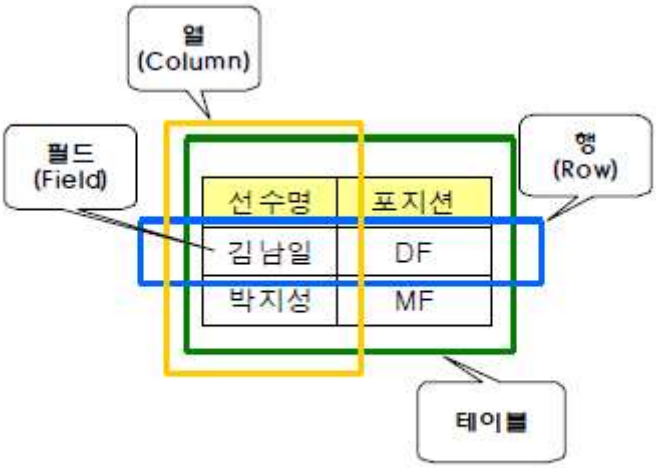
선수명	소속팀	포지션
박지성	서울FC	MF
이청용	블루윙즈	MF
⋮	⋮	⋮

테이블에 데이터가 있는 경우

선수명	소속팀	포지션

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.

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

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

[그림 Ⅱ-1-5] 테이블의 정규화

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.

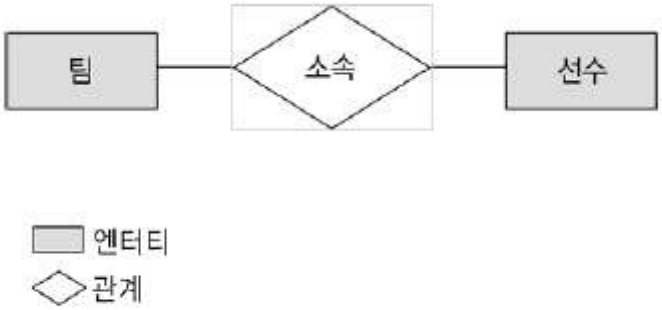
[표 Ⅱ-1-6] 테이블 관계 용어들

용어	설명
정규화 (Normalization)	테이블을 분할하여 데이터의 정합성을 확보하고, 불필요한 중복을 줄이는 프로세스

용어	설명
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] 팀-선수 ERD

[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 ERD라고 한다. ERD의 구성 요소는 엔터티(Entity), 관계(Relationship), 속성(Attribute) 3가지이며 현실 세계의 데이터는 이 3가지 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 ERD이다.

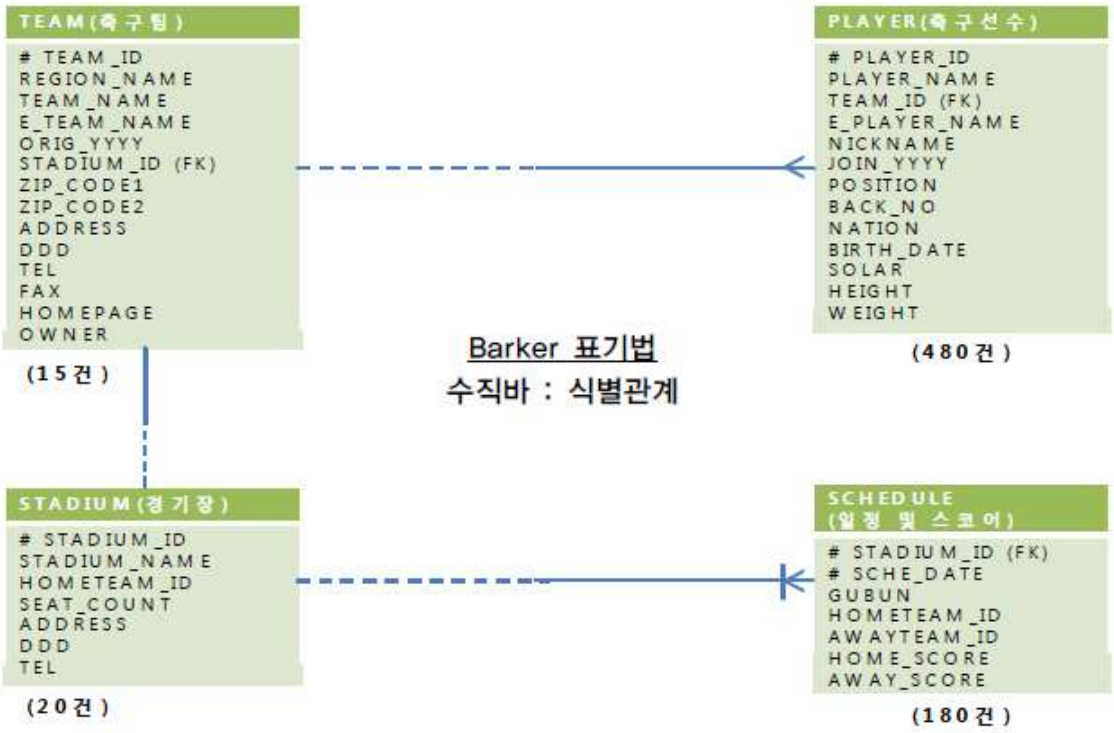
K-리그 테이블 간의 양방향 관계는 다음과 같다.

- 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.

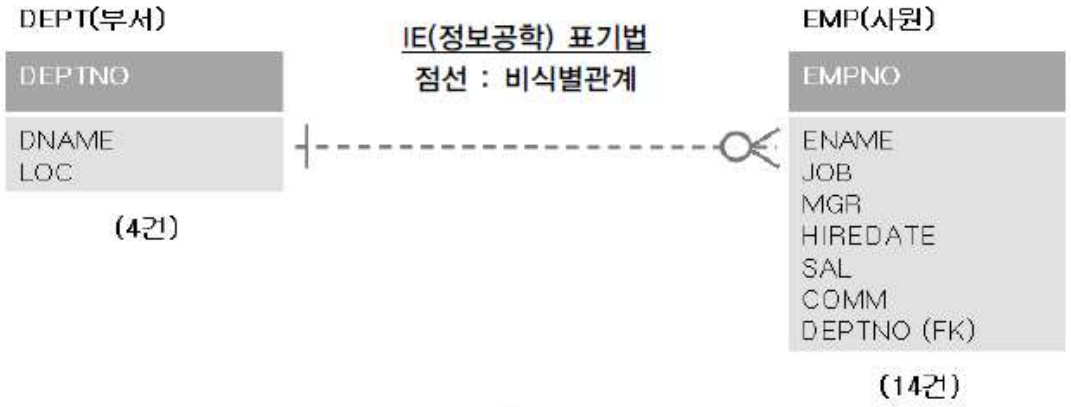
- 하나의 운동장은 여러 게임의 스케줄을 가질 수 있다.
- 하나의 스케줄은 하나의 운동장에 꼭 배정된다.

[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)

제2절 DDL(DATA DEFINITION LANGUAGE)

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 Type의 하위 개념으로 NUMERIC, DECIMAL, DEC, SMALLINT, INTEGER, INT, BIGINT, FLOAT, REAL, DOUBLE PRECISION을 소개하고 있다. SQL Server와 Sybase는 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, Oracle은 숫자형 타입에 대해서 NUMBER 한 가지 숫자 타입의 데이터 유형만 지원한다. 사용자 입장에서는 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 4가지 데이터 유형을 정리하였다. 아래 4가지 유형 외에도 ANSI/ISO에서는 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)	- 고정 길이 문자열 정보 (Oracle, SQL Server 모두 CHAR로 표현) - s는 기본 길이 1바이트, 최대 길이 Oracle 2,000바이트, SQL Server 8,000바이트 - s만큼 최대 길이를 갖고 고정 길이를 가지고 있으므로 할당된 변수 값의 길이가 s보다 작을 경우에는 그 차이 길이만큼 공간으로 채워진다.

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

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 40바이트가 지정되더라도 실제 'PARK,JISUNG'으로 데이터가 입력되는 경우 11바이트의 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 '선수ID' 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 팀ID와 팀 이름을 저장하고, 선수 테이블에서는 팀ID를 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 (
     칼럼명1 DATATYPE [DEFAULT 형식],
     칼럼명2 DATATYPE [DEFAULT 형식],
     칼럼명2 DATATYPE [DEFAULT 형식]
) ;
```

다음은 테이블 생성 시에 주의해야 할 몇 가지 규칙이다.

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

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

한 테이블 안에서 칼럼 이름은 달라야 하지만, 다른 테이블의 칼럼 이름과는 같을 수 있다.

예를 들면 선수 테이블의 TEAM_ID, 팀 테이블의 TEAM_ID는 같은 칼럼 이름을 가지고 있다. 실제 DBMS는 팀 테이블의 TEAM_ID를 PC나 UNIX의 디렉토리 구조처럼 'DB명+DB사용자명+테이블명+칼럼명'처럼 계층적 구조를 가진 전체 경로로 관리하고 있다.

이처럼 같은 이름을 가진 칼럼들은 기본키와 외래키의 관계를 가지는 경우가 많으며, 향후 테이블 간의 조인 조건으로 주로 사용되는 중요한 연결고리 칼럼들이다.

[예제] 다음 조건의 형태로 선수 테이블을 생성한다.

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

[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 칼럼명은 대문자로 만들어진다.
- DATETIME 데이터 유형에는 별도로 크기를 지정하지 않는다.
- 문자 데이터 유형은 반드시 가질 수 있는 최대 길이를 표시해야 한다.
- 칼럼과 칼럼의 구분은 콤마로 하되, 마지막 칼럼은 콤마를 찍지 않는다.
- 칼럼에 대한 제약조건이 있으면 CONSTRAINT를 이용하여 추가할 수 있다.

제약조건은 PLAYER_NAME, TEAM_ID 칼럼의 데이터 유형 뒤에 NOT NULL을 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 TABLE을 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 개가 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 00번)은 공백(BLANK, ASCII 코드 32번)이나 숫자 0(ZERO, ASCII 48)과는 전혀 다른 값이며, 조건에 맞는 데이터가 없을 때의 공집합과도 다르다. 'NULL'은 '아직 정의되지 않은 미지의 값'이거나 '현재 데이터를 입력하지 못하는 경우'를 의미한다.

- 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 기본값이 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 3자리, REGION_NAME (연고지 명) 문자 가변 자릿수 8자리, TEAM_NAME (한글 팀 명) 문자 가변 자릿수 40자리, E-TEAM_NAME (영문 팀 명) 문자 가변 자릿수 50자리 , ORIG_YYYY (창단년도) 문자 고정 자릿수 4자리, STADIUM_ID (구장 고유 ID) 문자 고정 자릿수 3자리, ZIP_CODE1 (우편번호 앞 3자리) 문자 고정 자릿수 3자리, ZIP_CODE2 (우편번호 뒷 3자리) 문자 고정 자릿수 3자리, ADDRESS (주소) 문자 가변 자릿수 80자리, DDD (지역번호) 문자 가변 자릿수 3자리, TEL (전화번호) 문자 가변 자릿수 10자리, FAX (팩스번호) 문자 가변 자릿수 10자리, HOMEPAGE (홈페이지) 문자 가변 자릿수 50자리, OWNER (구단주) 문자 가변 자릿수 10자리,
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. Oracle의 경우 “DESCRIBE 테이블명;” 또는 간략히 “DESC 테이블명;”으로 해당 테이블에 대한 정보를 확인할 수 있다. SQL Server의 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 칼럼별로 데이터 유형을 다시 재정의 하지 않아도 되는 장점이 있다.

그러나 CTAS 기법 사용시 주의할 점은 기존 테이블의 제약조건 중에 NOT NULL만 새로운 복제 테이블에 적용이 되고, 기본키, 고유키, 외래키, CHECK 등의 다른 제약 조건은 없어진다는 점이다. 제약 조건을 추가하기 위해서는 뒤에 나오는 ALTER TABLE 기능을 사용해야 한다.

SQL Server에서는 Select ~ Into ~ 를 활용하여 위와 같은 결과를 얻을 수 있다. 단, 칼럼 속성에 Identity를 사용했다면 Identity 속성까지 같이 적용이 된다.

[예제] 선수(PLAYER) 테이블과 같은 내용으로 TEAM_TEMP라는 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           NOTNULL       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;
(1개 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 80자리로 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 COLUMN은 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 ADDRESS가 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'을 적용하고, 모든 행의 ORIG_YYYY 칼럼에 NULL이 없으므로 제약조건을 NULL → NOT NULL로 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 VARCAHR(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;
```

```
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            NOT NULL      VARCHAR2(8) ☜ 기본값 '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
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

constraint_type                    constraint_name      constraint_keys
-----------------------            ------------         -------------
DEFAULT on column ORIG_YYYY DF_ORIG_YYYY                ('20020129')
```

- RENAME COLUMN

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

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

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

```
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 칼럼 이름을 변경할 수 있다.

```
sp_rename 변경해야 할 칼럼명, 새로운 칼럼명, '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_FK로 하고, PLAYER 테이블의 TEAM_ID 칼럼이 TEAM 테이블의 TEAM_ID를 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_rename을 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 TABLE은 테이블 자체가 삭제되는 것이 아니고, 해당 테이블에 들어있던 모든 행들이 제거되고 저장 공간을 재사용 가능하도록 해제한다. 테이블 구조를 완전히 삭제하기 위해서는 DROP TABLE을 실행하면 된다.

```
TRUNCATE TABLE PLAYER;
```

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

```
[예제] Oracle
TRUNCATE TABLE TEAM;
테이블이 트렁케이트되었다.
```

```
[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 TABLE을 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 TABLE의 경우는 테이블 자체가 없어지기 때문에 테이블 구조를 확인할 수 없다. 반면 TRUNCATE TABLE의 경우는 테이블 구조는 그대로 유지한 체 데이터만 전부 삭제하는 기능이다. TRUNCATE는 데이터 구조의 변경 없이 테이블의 데이터를 일괄 삭제하는 명령어로 DML로 분류할 수도 있지만 내부 처리 방식이나 Auto Commit 특성 등으로 인해 DDL로 분류하였다.

테이블에 있는 데이터를 삭제하는 명령어는 TRUNCATE TABLE 명령어 이외에도 다음 DML 절에서 살펴볼 DELETE 명령어가 있다. 그러나 DELETE와 TRUNCATE는 처리하는 방식 자체가 다르다. 테이블의 전체 데이터를 삭제하는 경우, 시스템 활용 측면에서는 DELETE TABLE 보다는 시스템 부하가 적은 TRUNCATE TABLE을 권고한다. 단, TRUNCATE TABLE의 경우 정상적인 복구가 불가능하므로 주의해야 한다.

제3절 DML(DATA MANIPULATION LANGUAGE)

앞 절에서 테이블을 생성하고 생성된 테이블의 구조를 변경하는 명령어에 대해서 알아보았다. 지금부터는 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_LIST에 넣을 VALUE_LIST);

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

해당 칼럼명과 입력되어야 하는 값을 서로 1:1로 매핑해서 입력하면 된다. 해당 칼럼의 데이터 유형이 CHAR나 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 Default로 NULL 값이 입력된다. 단, Primary Key나 Not NULL 로 지정된 칼럼은 NULL이 허용되지 않는다.

두 번째 유형은 모든 칼럼에 데이터를 입력하는 경우로 굳이 COLUMN_LIST를 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);
1개의 행이 만들어졌다.
```

[표 Ⅱ-1-11]은 데이터베이스 내에 있는 PLAYER 테이블에 박지성 선수 정보가 입력되어 있는 것을 나타낸 것이다. INSERT 문장에서 BACK_NO가 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);
1개의 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_NAME은 두 개의 『 '' 』SINGLE QUOTATION을 붙여서 표현하거나, NATION이나 BIRTH_DATE의 경우처럼 NULL이라고 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 99로 수정한다.

```
[예제]
UPDATE PLAYER
SET BACK_NO = 99;
480개의 행이 수정되었다.
```

[예제] 선수 테이블의 포지션을 일괄적으로 'MF'로 수정한다.

```
[예제]
UPDATE PLAYER
SET POSITION = 'MF';
480개의 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;
480개의 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 TRANSACTION을 종료해야 한다.

그러나 SQL Server의 경우는 DML의 경우도 AUTO COMMIT으로 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 TABLE을 권고한다.

단, TRUNCATE TABLE의 경우 삭제된 데이터의 로그가 없으므로 ROLLBACK이 불가능하므로 주의해야 한다. 그러나 SQL Server의 경우 사용자가 임의적으로 트랜잭션을 시작한 후 TRUNCATE TABLE을 이용하여 데이터를 삭제한 이후 오류가 발견되어, 다시 복구를 원할 경우 ROLLBACK 문을 이용하여 테이블 데이터를 원 상태로 되돌릴 수 있다. 트랜잭션과 COMMIT, ROLLBACK에 대해서는 다음 절에서 설명한다.

4. SELECT

사용자가 입력한 데이터는 언제라도 조회가 가능하다. 앞에서 입력한 자료들을 조회해보는 SQL 문은 다음과 같다. (별도 제공한 SQL SCRIPT를 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
 :  :               :     :     :     :     :
480개의 행이 선택되었다.
```

- DISTINCT 옵션

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

```
[예제 및 실행 결과]
SELECT ALL POSITION
 FROM PLAYER;

ALL은 생략 가능한 키워드이므로 아래 SQL 문장도 같은 결과를 출력한다,

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

```
[예제]
SELECT DISTINCT POSITION
FROM PLAYER;
```

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

실행 결과를 보면 480개의 행이 모두 출력된 것이 아니라 포지션의 종류인 4개의 행과 포지션 데이터가 아직 미정인 NULL까지 5건의 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 앙마 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
 : : : : : : : : : : : : : :
480개의 행이 선택되었다.
```

실행 결과 화면을 보면 칼럼 레이블(LABLE)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 AS를 꼭 사용하지 않아도 되므로,
아래 SQL은 위 SQL과 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
: : : :
480개의 행이 선택되었다.
```

[예제] 칼럼 별명을 적용할 때 별명 중간에 공백이 들어가는 경우 『" " 』를 사용해야 한다. SQL Server의 경우『" "』, 『' 』', 『[ ]』와 같이 3가지의 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
: : : :
480개의 행이 선택되었다.
```

5. 산술 연산자와 합성 연산자

- 산술 연산자

산술 연산자는 NUMBER와 DATE 자료형에 대해 적용되며 일반적으로 수학에서의 4칙 연산과 동일하다. 그리고 우선순위를 위한 괄호 적용이 가능하다. 일반적으로 산술 연산을 사용하거나 특정 함수를 적용하게 되면 칼럼의 LABEL이 길어지게 되고, 기존의 칼럼에 대해 새로운 의미를 부여한 것이므로 적절한 ALIAS를 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
보띠     106.00
비에라   103.00
서동원   106.00
안대현   107.00
양현정   104.00
유원섭   103.00
김수철   103.00
임다한   114.00
… …
480개의 행이 선택되었다.
```

[예제] 선수들의 키와 몸무게를 이용해서 BMI(Body Mass Index) 비만지수를 측정한다.
※ 예제에서 사용된 ROUND( ) 함수는 반올림을 위한 내장 함수로써 6절에서 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
보띠    22.46
비에라  23.57
서동원  23.04
안대현  22.47
양현정  23.24
유원섭  23.77
김수철  23.26
임다한  20.45
… …
480개의 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 2개의 수직 바(||)에 의해 이루어진다. (Oracle)
- 문자와 문자를 연결하는 경우 + 표시에 의해 이루어진다. (SQL Server)
- 두 벤더 모두 공통적으로 CONCAT (string1, string2) 함수를 사용할 수 있다.
- 칼럼과 문자 또는 다른 칼럼과 연결시킨다.
- 문자 표현식의 결과에 의해 새로운 칼럼을 생성한다.

[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
보띠선수,174cm,68kg
비에라선수,176cm,73kg
서동원선수,184cm,78kg
안대현선수,179cm,72kg
양현정선수,176cm,72kg
유원섭선수,180cm,77kg
김수철선수,171cm,68kg
임다한선수,181cm,67kg
...
480개의 행이 선택되었다.
```

제4절 TCL(TRANSACTION CONTROL LANGUAGE)

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. 100번 계좌의 잔액에서 10,000원을 뺀다.
- STEP2. 200번 계좌의 잔액에 10,000원을 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 3가지 명령어를 트랜잭션을 콘트롤하는 TCL(TRANSACTION CONTROL LANGUAGE)로 분류한다.

트랜잭션의 대상이 되는 SQL문은 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 실행되던지 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.

COMMIT이나 ROLLBACK 이전의 데이터 상태는 다음과 같다.

- 단지 메모리 BUFFER에만 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 COMMIT을 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);
1개의 행이 만들어졌다.

COMMIT;
커밋이 완료되었다.
```

[예제] PLAYER 테이블에 있는 데이터를 수정하고 COMMIT을 실행한다.

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

COMMIT;
커밋이 완료되었다.
```

[예제] PLAYER 테이블에 있는 데이터를 삭제하고 COMMIT을 실행한다.

```
[예제] Oracle
DELETE FROM PLAYER;
480개의 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 Server의 COMMIT

Oracle은 DML을 실행하는 경우 DBMS가 트랜잭션을 내부적으로 실행하며 DML 문장 수행 후 사용자가 임의로 COMMIT 혹은 ROLLBACK을 수행해 주어야 트랜잭션이 종료된다. (일부 툴에서는 AUTO COMMIT을 옵션으로 선택할 수 있다.)

하지만, SQL Server는 기본적으로 AUTO COMMIT 모드이기 때문에 DML 수행 후 사용자가 COMMIT이나 ROLLBACK을 처리할 필요가 없다. DML 구문이 성공이면 자동으로 COMMIT이 되고 오류가 발생할 경우 자동으로 ROLLBACK 처리된다. 위의 예제를 SQL Server용으로 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);
1개의 행이 만들어졌다.
```

[예제] PLAYER 테이블에 있는 데이터를 수정한다.

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

[예제] PLAYER 테이블에 있는 데이터를 삭제한다.

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

SQL Server에서의 트랜잭션은 기본적으로 3가지 방식으로 이루어진다.

1) AUTO COMMIT

SQL Server의 기본 방식이며, DML, DDL을 수행할 때마다 DBMS가 트랜잭션을 컨트롤하는 방식이다. 명령어가 성공적으로 수행되면 자동으로 COMMIT을 수행하고 오류가 발생하면 자동으로 ROLLBACK을 수행한다.

2) 암시적 트랜잭션

Oracle과 같은 방식으로 처리된다. 즉, 트랜잭션의 시작은 DBMS가 처리하고 트랜잭션의 끝은 사용자가 명시적으로 COMMIT 또는 ROLLBACK으로 처리한다. 인스턴스 단위 또는 세션 단위로 설정할 수 있다. 인스턴스 단위로 설정하려면 서버 속성 창의 연결화면에서 기본연결 옵션 중 암시적 트랜잭션에 체크를 해주면 된다. 세션 단위로 설정하기 위해서는 세션 옵션 중 SET IMPLICIT TRANSACTION ON을 사용하면 된다.

3) 명시적 트랜잭션

트랜잭션의 시작과 끝을 모두 사용자가 명시적으로 지정하는 방식이다. BEGIN TRANSACTION (BEGIN TRAN 구문도 가능)으로 트랜잭션을 시작하고 COMMIT

TRANSACTION(TRANSACTION은 생략 가능) 또는 ROLLBACK TRANSACTION (TRANSACTION은 생략 가능)으로 트랜잭션을 종료한다. ROLLBACK 구문을 만나면 최초의 BEGIN TRANSACTION 시점까지 모두 ROLLBACK이 수행된다.

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 ROLLBACK을 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);
1개의 행이 만들어졌다.

ROLLBACK;
롤백이 완료되었다.
```

[예제] PLAYER 테이블에 있는 데이터를 수정하고 ROLLBACK을 실행한다.

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

ROLLBACK;
롤백이 완료되었다.
```

[예제] PLAYER 테이블에 있는 데이터를 삭제하고 ROLLBACK을 실행한다.

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

ROLLBACK;
롤백이 완료되었다.
```

■ SQL Server의 ROLLBACK

SQL Server는 위에서 언급한 바와 같이 AUTO COMMIT이 기본 방식이므로 임의적으로 ROLLBACK을 수행하려면 명시적으로 트랜잭션을 선언해야 한다. 위의 예제는 아래와 같이 변경된다.

[예제] PLAYER 테이블에 데이터를 입력하고 ROLLBACK을 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);
1개의 행이 만들어졌다.

ROLLBACK;
롤백이 완료되었다.
```

[예제] PLAYER 테이블에 있는 데이터를 수정하고 ROLLBACK을 실행한다.

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

ROLLBACK;
롤백이 완료되었다.
```

[예제] PLAYER 테이블에 있는 데이터를 삭제하고 ROLLBACK을 실행한다.

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

ROLLBACK;
롤백이 완료되었다.
```

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

COMMIT과 ROLLBACK을 사용함으로써 다음과 같은 효과를 볼 수 있다.

- 데이터 무결성 보장
- 영구적인 변경을 하기 전에 데이터의 변경 사항 확인 가능
- 논리적으로 연관된 작업을 그룹핑하여 처리 가능

4. SAVEPOINT

저장점(SAVEPOINT)을 정의하면 롤백(ROLLBACK)할 때 트랜잭션에 포함된 전체 작업을 롤백하는 것이 아니라 현 시점에서 SAVEPOINT까지 트랜잭션의 일부만 롤백할 수 있다. 따라서 복잡한 대규모 트랜잭션에서 에러가 발생했을 때 SAVEPOINT까지의 트랜잭션만 롤백하고 실패한 부분에 대해서만 다시 실행할 수 있다. (일부 툴에서는 지원이 안 될 수 있음)

복수의 저장점을 정의할 수 있으며, 동일이름으로 저장점을 정의했을 때는 나중에 정의한 저장점이 유효하다. 다음의 SQL문은 SVPT1이라는 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 Server는 SAVE TRANSACTION을 사용하여 동일한 기능을 수행할 수 있다. 다음의 SQL문은 SVTR1이라는 저장점을 정의하고 있다.

```
SAVE TRANSACTION SVTR1;
```

저장점까지 롤백할 때는 ROLLBACK 뒤에 저장점 명을 지정한다.

```
ROLLBACK TRANSACTION SVTR1;
```

[예제] SAVEPOINT를 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);
1개의 행이 만들어졌다.

ROLLBACK TO SVPT1;
롤백이 완료되었다.
```

[예제] SQL Server

```
SAVE TRAN SVTR1;
저장점이 생성되었다.

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.

ROLLBACK TRAN SVTR1;
롤백이 완료되었다.
```

[예제] 먼저 SAVEPOINT를 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;
480개의 행이 수정되었다.

ROLLBACK TO SVPT2;
롤백이 완료되었다.
```

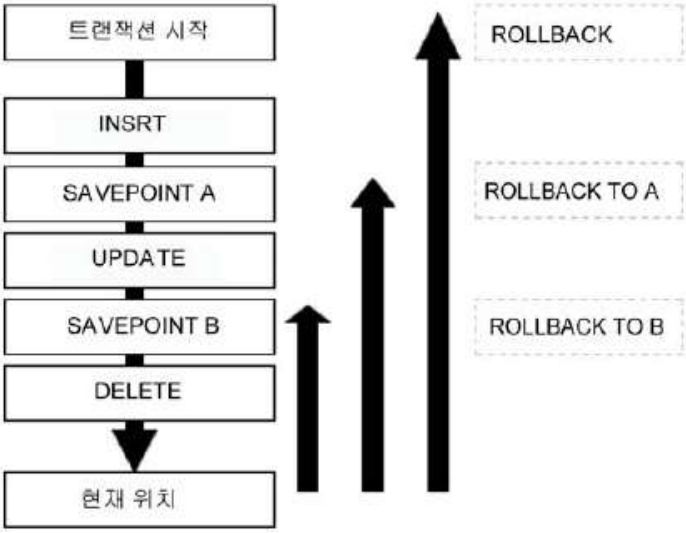
```
[예제] SQL Server
SAVEPOINT SVPT2;
저장점이 생성되었다.

UPDATE PLAYER
SET WEIGHT = 100;
480개의 행이 수정되었다.

ROLLBACK TO SVPT2;
롤백이 완료되었다.
```

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

```
[예제] Oracle
SAVEPOINT SVPT3;
저장점이 생성되었다.

DELETE FROM PLAYER;
480개의 행이 삭제되었다.

ROLLBACK TO SVPT3;
롤백이 완료되었다.
```

```
[예제] SQL Server
SAVE TRAN SVTR3;
저장점이 생성되었다.

DELETE FROM PLAYER;
480개의 행이 삭제되었다.

ROLLBACK TRAN SVTR3;
롤백이 완료되었다.
```

[그림 Ⅱ-1-11]은 여러 개의 SAVEPOINT 설정 후 롤백(ROLLBACK)이 이루어지는 범위에 대해 표현한 것이다.



[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. (툴에 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 전후 데이터를 확인해본다.

```
[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 전후 데이터를 확인해본다.

[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 전후 데이터를 확인해본다.

```
[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 "?? 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 전후 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.

제5절 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 1장 9절에서 설명하고 FROM 절의 JOIN에 대해서는 2장 1절에서 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 CPU나 MEMORY와 같은 시스템 자원(Resources)들을 과다하게 사용한다. 또한 많은 사용자들의 QUERY에 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. (FTS가 무조건 나쁜 것은 아니며 병렬 처리 등을 이용해 유용하게 사용하는 경우도 많다.)

기본적인 SQL 문장은 Oracle의 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 [ALIAS명]
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 3가지 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, OR의 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
3행에 오류:
ERROR: 열명이 부적합하다.
```

실행 결과는 “열(COLUMN)명이 부적합하다.”라는 에러 메시지를 보이고 SQL 문장의 세 번째 줄에 오류가 있다고 나와 있다. TEAM_ID라는 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
:	:	:	:

49개의 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
:	:	:	:

162개의 행이 선택되었다.

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

[표 Ⅱ-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
:	:	:	:

439개의 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, HEIGHT라는 칼럼이 숫자 유형의 변수이므로 내부적으로 '170'이라는 문자열을 숫자 유형 170으로 바꾸어 처리한다.

4. SQL 연산자

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

[실행 결과]

선수이름	포지션	백넘버	키
데니스	FW	11	176
서정원	FW	14	173
손대호	DF	17	186
오규찬	MF	24	178
윤원일	MF	45	176
김동욱	DF	40	176
김회택	DF		
서현옥	DF		
정상호	DF		
최철우	DF		
정영광	GK	41	185
:	:	:	:

100개의 행이 선택되었다.

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

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

```
[실행 결과]
 ENAME        JOB          DEPTNO
 JONES        MANAGER          20
 JAMES        CLERK            30
 2개의 행이 선택되었다.
```

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            30
 JAMES        CLERK            30
 5개의 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';
```

[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
:	:	:	:

162개의 행이 선택되었다.

LIKE의 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 0개 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
장서연	FW	7	180
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

13개의 행이 선택되었다.

- BETWEEN a AND b 연산자

[예제] 세 번째로 키가 170 센티미터 이상 180센티미터 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	24	179
정동선	MF	9	170

```
최경규      MF          10     177
최내철      MF          24     177
배성재      MF          28     178
샴          MF          25     174
김관우      MF           8     175
:           :           :      :
259개의 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 = NULL을 사용했는데 문법 에러가 나지는 않았지만 WHERE 절의 조건이 거짓(FALSE)이 되어 WHERE 절의 조건을 만족하는 데이터를 한건도 얻지 못하게 된 것으로 의미 없는 SQL이 되고 말았다.

[예제] POSITION 칼럼(Column) 값이 NULL 값인지를 판단하기 위해서는 IS NULL을 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
3개의 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]울 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, TEAM_ID
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
이성용	DF	20	173
하태근	MF	29	182

: : : : 45개의 행이 선택되었다

[예제] “소속이 삼성블루윙즈이거나 전남드래곤즈”인 조건을 SQL 비교 연산자로, “포지션이 미드필더(MF)”인 조건을 비교 연산자로 비교한 결과를 논리 연산자로 묶어서 처리한다.

[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

하기윤	MF	32	180
김반코비	MF	47	185
:	:	:	:

45개의 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 DF나 FW인 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, OR의 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;
:
33개의 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 비교 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 SQL과 아래 SQL은 같은 내용을 나타내는 SQL이다.

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

8개의 행이 선택되었다.

[예제] 국적(NATION) 칼럼의 경우 내국인들은 별도 데이터를 입력하지 않았다. 국적 칼럼이 NULL이 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
뚜따	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
쟈스민	크로아티아
코샤	브라질
27개의 행이 선택되었다.	

7. ROWNUM, TOP 사용

- ROWNUM

Oracle의 ROWNUM은 칼럼과 비슷한 성격의 Pseudo Column으로써 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.

한 건의 행만 가져오고 싶을 때는

- SELECT PLAYER_NAME FROM PLAYER WHERE ROWNUM = 1; 이나
- SELECT PLAYER_NAME FROM PLAYER WHERE ROWNUM <= 1; 이나
- SELECT PLAYER_NAME FROM PLAYER WHERE ROWNUM <2; 처럼 사용할 수 있다.

두 건 이상의 N 행을 가져오고 싶을 때는 ROWNUM = N; 처럼 사용할 수 없으며

- SELECT PLAYER_NAME FROM PLAYER WHERE ROWNUM <= N; 이나
- SELECT PLAYER_NAME FROM PLAYER WHERE ROWNUM <= N+1; 처럼 출력되는 행의 한계를 지정할 수 있다.

추가적인 ROWNUM의 용도로는 테이블 내의 고유한 키나 인덱스 값을 만들 수 있다.

- UPDATE MY_TABLE SET COLUMN1 = ROWNUM;

• TOP 절

SQL Server는 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 Oracle의 ROWNUM과 SQL Server의 TOP 절은 같은 기능을 하지만, ORDER BY 절이 같이 사용되면 기능의 차이가 발생한다. 이 부분은 1장 8절 ORDER BY 절에서 설명하도록 한다.

제6절 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 SQL을 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 2장 5절에서, 윈도우 함수는 2장 6절에서 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 Oracle과 SQL Server에서 공통으로 사용하는 중요 함수 위주로 설명을 한다.

함수에 대한 자세한 내용이나 버전에 따른 변경 내용은 벤더에서 제공하는 매뉴얼을 참조하기 바란다. 아래 함수의 예에서 SUBSTR / SUBSTRING으로 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. Oracle은 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 SYS가 소유하며 모든 사용자가 액세스 가능한 테이블이다.
- SELECT ~ FROM ~ 의 형식을 갖추기 위한 일종의 DUMMY 테이블이다.
- DUMMY라는 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
1개의 행이 선택되었다.
```

반면 Sybase나 SQL Server의 경우에는 SELECT 절만으로도 SQL 문장이 수행 가능하도록 정의하였기 때문에 DUAL이란 DUMMY 테이블이 필요 없다. 그러나 Sybase나 SQL Server의 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(PLAYER_NAME, ' 축구선수') 선수명
FROM PLAYER;

CONCAT 함수는 Oracle의 '||' 합성 연산자와 같은 기능이다.
SELECT PLAYER_NAME || ' 축구선수' AS 선수명
FROM PLAYER;
```

SQL Server에서 위의 예제와 같은 결과를 얻으려면 아래와 같이 수행하면 된다.

```
[예제] SQL Server
 SELECT PLAYER_NAME  + ' 축구선수' AS 선수명
 FROM PLAYER;
```

```
[실행 결과]
 PLAYER_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       데니스 축구선수
 2007274       서정원 축구선수
  |              |
 480개의 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.

```
함수3 (함수2 (함수1 (칼럼이나 표현식 [, Arg1]) [, Arg2]) [, Arg3 ])
```

[예제] 경기장의 지역번호와 전화번호를 합친 번호의 길이를 구하시오.

연결연산자의 결과가 LENGTH(SQL Server는 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 a s 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
20개의 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: Oracle함수/SQL Server함수 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) SIGN(0) SIGN(+20)	-1 0 1
MOD(7,3) / 7%3	1
CEIL(38.123) / CEILING(38.123) CEILING(-38.123)	39 39 -38
FLOOR(38.123) FLOOR(-38.123)	38 -39
ROUND(38.5235, 3) ROUND(38.5235, 1) ROUND(38.5235, 0) ROUND(38.5235)	38.524 38.5 39 39 (인수 0이 Default)
TRUNC(38.5235, 3) TRUNC(38.5235, 1) TRUNC(38.5235, 0) TRUNC(38.5235)	38.523 38.5 38 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/12,1)        TRUNC(SAL/12,1)
 --------       ---------------        --------------
 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
 14개의 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

14개의 행이 선택되었다.

4. 날짜형 함수

날짜형 함수는 DATE 타입의 값을 연산하는 함수이다. Oracle의 TO_NUMBER(TO_CHAR( )) 함수의 경우 변환형 함수로 구분할 수도 있으나 SQL Server의 YEAR, MONTH,DAY 함수와 매핑하기 위하여 날짜형 함수에서 설명한다. EXTRACT/DATEPART는 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: Oracle함수/SQL Server함수 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 뺄 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] Oracle의 SYSDATE 함수와 SQL Server의 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 입사일자에서 년, 월, 일 데이터를 각각 출력한다. 아래 4개의 SQL 문장은 같은 기능을 하는 SQL 문장이다.

[예제] Oracle 함수

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

[예제] 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

14개의 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;

날자          문자형
---------     ----------------
2012-07-19    2012. 7월 , 월요일
```

[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 코드1과 ZIP 코드2를 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, 함수와 같은 성격을 가지고 있으며 Oracle의 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
14개의 행이 선택되었다.
```

CASE 표현을 하기 위해서는 조건절을 표현하는 두 가지 방법이 있고, Oracle의 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 Expressions은 Simple Case Expression과 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_EXPRESSION은 CASE 다음에 바로 조건에 사용되는 칼럼이나 표현식을 표시하고, 다음 WHEN 절에서 앞에서 정의한 칼럼이나 표현식과 같은지 아닌지 판단하는 문장으로 EQUI(=) 조건만 사용한다면 SEARCHED_CASE_EXPRESSION보다 간단하게 사용할 수 있는 장점이 있다. Oracle의 DECODE 함수와 기능면에서 동일하다.

```
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
4개의 행이 선택되었다.
```

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

```
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
14개의 행이 선택되었다.
```

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

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

[예제]

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

[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
14개의 행이 선택되었다.
```

7. NULL 관련 함수

가. NVL/ISNULL 함수

다시 한 번 NULL에 대한 특성을 정리한다.

- 널 값은 아직 정의되지 않은 값으로 0 또는 공백과 다르다.
  0은 숫자이고, 공백은 하나의 문자이다.
  - 테이블을 생성할 때 NOT NULL 또는 PRIMARY KEY로 정의되지 않은 모든 데이터 유형은 널 값을 포함할 수 있다.
  - 널 값을 포함하는 연산의 경우 결과 값도 널 값이다.
  모르는 데이터에 숫자를 더하거나 빼도 결과는 마찬가지로 모르는 데이터인 것과 같다.
  - 결과값을 NULL이 아닌 다른 값을 얻고자 할 때 NVL/ISNULL 함수를 사용한다.
  NULL 값의 대상이 숫자 유형 데이터인 경우는 주로 0(Zero)으로, 문자 유형 데이터인 경우는 블랭크보다는 'x' 같이 해당 시스템에서 의미 없는 문자로 바꾸는 경우가 많다.

[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 NULL일 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 NULL을 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

Oracle의 경우 NVL 함수를 사용한다.

```
NVL (NULL 판단 대상,'NULL일 때 대체값')
```

[예제 및 실행 결과] Oracle

```
SELECT NVL(NULL, 'NVL-OK') NVL_TEST
FROM DUAL;

NVL_TEST
-------
NVL-OK
1개의 행이 선택되었다.
```

[예제 및 실행 결과] Oracle

```
SELECT NVL('Not-Null', 'NVL-OK') NVL_TEST
FROM DUAL;

NVL_TEST
-------
Not-Null
1개의 행이 선택되었다.
```

SQL Server의 경우 ISNULL 함수를 사용한다.

```
ISNULL (NULL 판단 대상,'NULL일 때 대체값')
```

[예제 및 실행 결과] SQL

```
Server SELECT ISNULL(NULL, 'NVL-OK') ISNULL_TEST ;

ISNULL_TEST
---------
NVL-OK
1개의 행이 선택되었다.
```

[예제 및 실행 결과] SQL Server

```
SELECT ISNULL('Not-Null', 'NVL-OK') ISNULL_TEST ;


ISNULL_TEST
---------
Not-Null
1개의 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
정명휘          DF              DF
정영철          DF              DF
곽치국          MF              MF
정상식          MF              MF
서관수          FW              FW
김성운          FW              FW
김정운          FW              FW
장동현          FW              FW
  |               |               |
45개의 행이 선택되었다.
```

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

[예제]

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

[실행 결과]

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

14개의 행이 선택되었다.

실행 결과에서 월급에 커미션을 더해서 연봉을 계산하는 산술식이 있을 때 커미션에 NULL 값이 있는 경우 커미션 값에 NVL() 함수를 사용하지 않으면 연봉A의 계산 결과가 NULL이 되어서 잘못 계산한 결과를 확인할 수 있다. 따라서 연봉B 결과와 같이 NVL(COMM,0)처럼 NULL 값을 0으로 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 NULL이 나오고 전체 건수 중에서 일부만 NULL인 경우는 다중행 함수의 대상에서 제외한다.

예를 들면 100명 중 10명의 성적이 NULL 값일 때 평균을 구하는 다중행 함수 AVG를 사용하면 NULL 값이 아닌 90명의 성적에 대해서 평균값을 구하게 된다. 자세한 내용은 1장 7절에서 추가로 설명한다.

나. NULL과 공집합

- 일반적인 NVL/ISNULL 함수 사용

STEP1. 정상적으로 매니저 정보를 가지고 있는 SCOTT의 매니저를 출력한다.

[예제 및 실행 결과]

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

MGR
-----
7566 1개의 행이 선택되었다.
```

☞ 'SCOTT'의 관리자(MGR=Manager)는 7566 사번을 가진 JONES이다.

STEP2. 매니저에 NULL이 들어있는 KING의 매니저를 출력한다.

[예제 및 실행 결과]

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

MGR
-----
1개의 행이 선택되었다.
```

☞ 빈 칸으로 표시되었지만 실 데이터는 NULL이다.

☞ 'KING'은 EMP 테이블에서 사장이므로 MGR(관리자) 필드에 NULL이 입력되어 있다.

STEP3. 매니저가 NULL인 경우 빈칸이 아닌 9999로 출력하기 위해 NVL/ISNULL 함수를 사용한다.

[예제 및 실행 결과]

```
SELECT NVL(MGR,9999) MGR FROM EMP WHERE ENAME='KING';
MGR
-----
9999
1개의 행이 선택되었다.
```

☞ NVL 함수로 NULL을 0으로 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 ENAME이'JSC'란 사람은 없으므로 공집합이 발생한다.

STEP2. NVL/ISNULL 함수를 이용해 공집합을 9999로 바꾸고자 시도한다.

[예제 및 실행 결과]

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

데이터를 찾을 수 없다.

☞ 많은 분들이 공집합을 NVL/ISNULL 함수를 이용해서 처리하려고 하는데, 인수의 값이 공집합인 경우는 NVL/ISNULL 함수를 사용해도 역시 공집합이 출력된다.

☞ NVL/ISNULL 함수는 NULL 값을 대상으로 다른 값으로 바꾸는 함수이지 공집합을 대상으로 하지 않는다.

STEP3. 적절한 집계 함수를 찾아서 NVL 함수 대신 적용한다.

[예제 및 실행 결과]

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


1개의 행이 선택되었다.
```

☞ 빈 칸으로 표시되었지만 실 데이터는 NULL이다.

☞ 다른 함수와 달리 집계 함수와 Scalar Subquery의 경우는 인수의 결과 값이 공집합인 경우에도 NULL을 출력한다.

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

[예제 및 실행 결과]

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


MGR
-----
9999
1개의 행이 선택되었다.
```

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

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

개발자들은 NVL/ISNULL 함수를 사용해야 하는 경우와, 집계 함수를 포함한 NVL/ISNULL 함수를 사용해야 하는 경우와, 1장 7절에서 설명할 NVL/ISNULL 함수를 포함한 집계 함수를 사용하지 않아야 될 경우까지 잘 이해해서 NVL/ISNULL 함수를 정확히 사용해야 한다.

다. NULLIF

NULLIF 함수는 EXPR1이 EXPR2와 같으면 NULL을, 같지 않으면 EXPR1을 리턴한다. 특정 값을 NULL로 대체하는 경우에 유용하게 사용할 수 있다.

```
NULLIF (EXPR1, EXPR2)
```

[예제] 사원 테이블에서 MGR와 7698이 같으면 NULL을 표시하고,
같지 않으면 MGR를 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

14개의 행이 선택되었다.

실행 결과를 보면 MGR의 값이 7698이란 상수가 같은 경우 NUIF칼럼에 NULL이 표시되었다. KING이 속한 행의 NUIF 칼럼에 NULL이 표시된 것은 원래 MGR 데이터가 NULL이었기 때문이다.

라. 기타 NULL 관련 함수 (COALESCE)

COALESCE 함수는 인수의 숫자가 한정되어 있지 않으며, 임의의 개수 EXPR에서 NULL이 아닌 최초의 EXPR을 나타낸다. 만일 모든 EXPR이 NULL이라면 NULL을 리턴한다.

```
COALESCE (EXPR1, EXPR2, …)
```

[예제] 사원 테이블에서 커미션을 1차 선택값으로, 급여를 2차 선택값으로 선택하되 두 칼럼 모두 NULL인 경우는 NULL로 표시한다.

[예제]

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

[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

14개의 행이 선택되었다.

제7절 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/ISO에서 데이터 분석 기능으로 분류한 함수 중 기본적인 집계 함수는 본 절에서 설명하고, ROLLUP, CUBE, GROUPING SETS 같은 GROUP 함수는 2장 5절에서, 다양한 분석 기능을 가진 WINDOW 함수는 2장 6절에서 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 NULL값이 아닌 키(HEIGHT) 칼럼의 건수만 출력하므로 COUNT(*)의 480보다 작은 것을 볼 수 있다. 그 이유는 COUNT(*) 함수에 사용된 와일드카드(*)는 전체 칼럼을 뜻하는데 전체 칼럼이 NULL인 행은 존재할 수 없기 때문에 결국 COUNT(*)는 전체 행의 개수를 출력한 것이고, COUNT(HEIGHT)는 HEIGHT 칼럼 값이 NULL인 33건은 제외된 건수의 합이다.

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] 칼럼명 [ALIAS명]

FROM 테이블명

[WHERE 조건식]

[GROUP BY 칼럼(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 GROUP으로 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 칼럼의 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 칼럼과 집계 함수를 사용할 수 있다. 그렇지 않으면 예제와 같이 에러를 발생하게 된다.

[예제] SELECT 절에서 사용된 포지션이라는 한글 ALIAS를 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. 칼럼에 대한 ALIAS는 SELECT 절에서 정의하고 ORDER BY 절에서는 재활용할 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 3명이라는 정보를 얻을 수 있으며, 포지션이 DF인 172명 중 30명은 키에 대한 정보가 없는 것도 알 수 있다. GK, DF, FW, MF의 최대키, 최소키, 평균키를 구할 때 키 값이 NULL인 경우는 계산 대상에서 제외된다. 즉, 포지션 DF의 최대키, 최소키, 평균키 결과는 키 값이 NULL인 30명을 제외한 142명을 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 4개 포지션 중에서 평균 키가 180cm가 넘는 2개의 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 FC서울(K09)의 인원수는 얼마인가란 요구 사항이 접수되었다. WHERE 절과 GROUP BY 절을 사용한 SQL과 GROUP BY 절과 HAVING 절을 사용한 SQL을 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 BY의 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.
```

SQL을 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 1장 8절의 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 189cm로 190cm 미만인 MF 포지션의 데이터만

HAVING 조건에 의해 누락된 것을 확인할 수 있다. (다른 포지션은 모두 MAX(HEIGHT)가 190 이상이다.) 누락된 MF 포지션의 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 제1정규화로 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 Expression을 이용해서 12개의 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 2장 4절에서 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     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 Expression으로 표현된 위의 SQL과 같은 내용으로 Oracle의 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 Oracle의 DECODE 함수는 표현상 서로 장단점이 있으므로 어떤 기능을 선택할 지는 사용자의 몫이다.

5. 집계 함수와 NULL

리포트의 빈칸을 NULL이 아닌 ZERO로 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 NULL이 나오고 전체 건수 중에서 일부만 NULL인 경우는 NULL인 행을 다중 행 함수의 대상에서 제외한다. 예를 들면 100명 중 10명의 성적이 NULL 값일 때 평균을 구하는 다중 행 함수 AVG를 사용하면 NULL 값이 아닌 90명의 성적에 대해서 평균값을 구하게 된다.

CASE 표현 사용시 ELSE 절을 생략하게 되면 Default 값이 NULL이다. NULL은 연산의 대상이 아닌 반면, SUM(CASE MONTH WHEN 1 THEN SAL ELSE 0 END)처럼 ELSE 절에서 0(Zero)을 지정하면 불필요하게 0이 SUM 연산에 사용되므로 자원의 사용이 많아진다. 같은 결과를 얻을 수 있다면 가능한 ELSE 절의 상수값을 지정하지 않거나 ELSE 절을 작성하지 않도록 한다. 같은 이유로 Oracle의 DECODE 함수는 4번째 인자를 지정하

지 않으면 NULL이 Default로 할당된다.

많이 실수하는 것 중에 하나가 Oracle의 SUM(NVL(SAL,0)), SQL Server의 SUM(ISNULL (SAL,0)) 연산이다. 개별 데이터의 급여(SAL)가 NULL인 경우는 NULL의 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 NULL이 아닌 0을 표시하고 싶은 경우에는 NVL(SUM(SAL),0)이나, ISNULL(SUM(SAL),0)처럼 전체 SUM의 결과가 NULL인 경우(대상 건수가 모두 NULL인 경우)에만 한 번 NVL/ISNULL 함수를 사용하면 된다.

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

4개의 예제 SQL 문장은 같은 실행 결과를 출력한다. ORDER BY 절이 적용되지 않았으므로 TEAM_ID 별로 정렬이 되어 있지 않다. TEAM_ID 'K08'의 경우 POSITION이 NULL인 3건이 포지션별 분류에는 빠져 있지만 COUNT(*) SUM에는 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.
```

제8절 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 [ALIAS명]
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

480개의 행이 선택되었다.
```

[예제] ORDER BY 절의 예로 선수 테이블에서 선수들의 이름, 포지션, 백넘버를 출력하는데 선수들의 포지션 내림차순으로 출력한다. 칼럼명이 아닌 ALIAS를 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
쟈스민     MF      33    186

480개의 행이 선택되었다.
```

실행 결과에서 포지션에 아무 것도 없는 값들이 있다. 현재 선수 테이블에서 포지션 칼럼에 NULL이 들어 있는데 포지션의 내림차순에서 NULL 값이 앞에 출력되었다는 것은 Oracle이 NULL 값을 가장 큰 값으로 취급했다는 것을 알 수 있다. 반면 SQL Server는 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.

- Oracle에서는 NULL 값을 가장 큰 값으로 간주하여 오름차순으로 정렬했을 경우에는 가장 마지막에, 내림차순으로 정렬했을 경우에는 가장 먼저 위치한다.
- 반면, SQL Server에서는 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 NULL인 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
조의손    GK       44    192
김창민    GK        1    191
우성용    FW       22    191
최동석    GK        1    190
샤샤      FW       10    190

447개의 행이 선택되었다.
```

실행 결과를 보면 키가 192cm인 선수가 5명 있는데, ORDER BY 절에서 키가 큰 순서대로 출력하고, 키가 같으면 백넘버 순으로 정렬하라는 조건에 따라서 백넘버 순으로 정렬되어 있는 것을 확인할 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, ALIAS명, 칼럼 순서를 같이 혼용하는 것도 가능하다.

[예제] ORDER BY 절의 예로 선수 테이블에서 선수들의 이름, 포지션, 백넘버를 출력하는데 선수들의 백넘버 내림차순, 백넘버가 같은 경우 포지션, 포지션까지 같은 경우 선수명 순서로 출력한다. BACK_NO가 NULL인 경우는 제외하고, 칼럼명이나 ALIAS가 아닌 칼럼 순서를 매핑하여 사용한다.

```
[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
...       ...      ..
439개의 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

4개의 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

4개의 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

4개의 행이 선택되었다.
```

2. SELECT 문장 실행 순서

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

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

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

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 DISTINCT를 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

14개의 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);
14개의 행이 선택되었다.
```

실행 결과에서 2장에서 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
5개의 행이 선택되었다.
```

SELECT SQL에서 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

Oracle에서 순위가 높은 N개의 로우를 추출하기 위해 ORDER BY 절과 WHERE 절의 ROWNUM 조건을 같이 사용하는 경우가 있는데 이 두 조건으로는 원하는 결과를 얻을 수 없다. Oracle의 경우 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 3명만 내림차순으로 출력하고자 하는데, 잘못 사용된 SQL의 사례이다.

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

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

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

[예제] ORDER BY 절이 없으면 ORACLE의 ROWNUM 조건과 SQL SERVER의 TOP 절은 같은 결과를 보인다. 그렇지만, ORDER BY 절이 사용되는 경우 ORACLE은 ROWNUM 조건을 ORDER BY 절보다 먼저 처리되는 WHERE 절에서 처리하므로, 정렬 후 원하는 데이터를 얻기 위해서는 2장 4절에서 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
3개의 행이 선택되었다.
```

위 사례에서는 인라인 뷰를 사용하여 추출하고자 하는 접합을 정렬한 후 ROWNUM을 적용시킴으로써 결과에 참여하는 순서와 추출되는 로우 순서를 일치시킴으로써 Top N 쿼리의 결과를 만들어내었다. 실행 결과를 보면 EMP 테이블의 데이터를 급여가 많은 순서부터 정렬을 수행한 후 상위 3건의 데이터를 출력한 것을 알 수 있다.

추가로, 원하는 추출 결과와 동일한 순서로 정렬된 인덱스가 존재한다면 그 인덱스를 사용하여 동일한 결과를 얻을 수도 있다.

- TOP ( )

반면 SQL Server는 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 2명을 내림차순으로 출력하고자 한다.

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

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

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

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

```
[실행 결과]
ENAME    SAL
-----    ---
KING     5000
SCOTT    3000
FORD     3000
3개의 행이 선택되었다.
```

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

제9절 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 Ⅱ-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 JOIN이라고 하며, 일반적으로 사용되는 SQL 문장의 상당수가 JOIN이라고 생각하면 JOIN의 중요성을 이해하기 쉬울 것이다. JOIN은 관계형 데이터베이스의 가장 큰 장점이면서 대표적인 핵심 기능이라고 할 수 있다.

일반적인 경우 행들은 PRIMARY KEY(PK)나 FOREIGN KEY(FK) 값의 연관에 의해 JOIN이 성립된다. 하지만 어떤 경우에는 이러한 PK, FK의 관계가 없어도 논리적인 값들의 연관만으로 JOIN이 성립 가능하다.

선수라는 테이블과 팀이라는 테이블이 있는 경우, 선수 테이블을 기준으로 필요한 데이터를 검색하고 이 데이터와 연관된 팀 테이블의 특정 행을 찾아오는 과정이 JOIN을 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 SQL에서 데이터를 처리할 때는 단 두 개의 집합 간에만 조인이 일어난다는 것이다. FROM 절에 A, B, C 테이블이 나열되었더라도 특정 2개의 테이블만 먼저 조인 처리되고, 2개의 테이블이 조인되어서 처리된 새로운 데이터 집합과 남은 한 개의 테이블이 다음 차례로 조인되는 것이다. 이순서는 4개 이상의 테이블이 사용되더라도 같은 프로세스를 반복한다.

예를 들어 A, B, C, D 4개의 테이블을 조인하고자 할 경우 옵티마이저는 (((A JOIN D) JOIN C) JOIN B)와 같이 순차적으로 조인을 처리하게 된다. 먼저 A와 D 테이블을 조인 처리하고, 그 결과 집합과 C 테이블을 다음 순서에 조인 처리하고, 마지막으로 3개의 테이블을 조인 처리한 집합과 B 테이블을 조인 수행하게 된다. 이때 테이블의 조인 순서는 옵티마이저에 의해서 결정되고 과목3의 주요 튜닝 포인트가 된다.

[그림 Ⅱ-1-12]는 선수와 팀, 팀과 운동장 테이블 간의 관계를 설명한 것이다. 경기일정 결과 테이블은 복잡성을 피하기 위해 설명상 제외하였다.

선수(PLAYER) 테이블

PLAYER_ID	PLAYER_NAME	TEAM_ID
2012137	이고르	K06
2012136	오비나	K10
2012135	윤원일	K02
2012134	페르난도	K04
2012133	레오	K03
2012132	실바	K07
2012131	무스타파	K04
2012130	에디	K01
2012129	알리송	K01
2012128	자스민	K08
2012127	디디	K06
...	...	...

운동장(STADIUM) 테이블

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

팀(TEAM) 테이블

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

[그림 Ⅱ-1-12] 테이블 간의 관계도

2. EQUI JOIN

EQUI(등가) JOIN은 두 개의 테이블 간에 칼럼 값들이 서로 정확하게 일치하는 경우에 사용되는 방법으로 대부분 PK ↔ FK의 관계를 기반으로 한다. 그러나 일반적으로 테이블 설계 시에 나타난 PK ↔ FK의 관계를 이용하는 것이지 반드시 PK ↔ FK의 관계로만 EQUI JOIN이 성립하는 것은 아니다. 이 기능은 계층형(Hierarchical)이나 망형(Network) 데이터베이스와 비교해서 관계형 데이터베이스의 큰 장점이다. JOIN의 조건은 WHERE 절에 기술하게 되는데 "=" 연산자를 사용해서 표현한다. 다음은 EQUI JOIN의 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 2장 1절에서 자세히 다룬다.

```
SELECT 테이블1.칼럼명, 테이블2.칼럼명, ...
FROM 테이블1 INNER JOIN 테이블2
ON 테이블1.칼럼명1 = 테이블2.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 SQL을 보면 SELECT 구문에 단순히 칼럼명이 오지 않고 “테이블명.칼럼명”처럼 테이블명과 칼럼명이 같이 나타난다. 이렇게 특정 칼럼에 접근하기 위해 그 칼럼이 어느 테이블에 존재하는 칼럼인지를 명시하는 것은 두 가지 이유가 있다.

먼저 모든 테이블에 칼럼들이 유일한 이름을 가진다면 상관없지만, JOIN에 사용되는 두 개의 테이블에 같은 칼럼명이 존재하는 경우에는 DBMS의 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 SQL에 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 JOIN에 참여하는 대상 테이블이 N개라고 했을 때, N개의 테이블로부터 필요한 데이터를 조회하기 위해 필요한 JOIN 조건은 대상 테이블의 개수에서 하나를 뺀 N-1개 이상이 필요하다. 즉 FROM 절에 테이블이 3개가 표시되어 있다면 JOIN 조건은 3-1=2개 이상이 필요하며, 테이블이 4개가 표시되어 있다면 JOIN 조건은 4-1=3개 이상이 필요하다. (옵티마이저의 발전으로 옵티마이저가 일부 JOIN 조건을 실행계획 수립 단계에서 추가할 수도 있지만, 예외적인 사항이다.)

JOIN 조건은 WHERE 절에 기술하며, JOIN은 두 개 이상의 테이블에서 필요한 데이터를 출력하기 위한 가장 기본적인 조건이다. FROM 절에 조인 조건을 명시하는 또 다른 방법은 2장 1절에서 설명한다. (JOIN 조건이 없는 CROSS JOIN도 2장 1절에서 설명한다.)

위 예제는 테이블1과 테이블2 이름을 가진 두개 테이블에 2 - 1 = 1인 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(PLAYER) 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
쟈스민	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(PLAYER) 테이블과 팀(TEAM) 테이블에 있는 데이터와 이들 간의 관계를 나타내는 그림을 통해서?.

위와 같이 선수들의 정보가 들어 있는 선수(PLAYER) 테이블이 있고, 팀의 정보가 들어 있는 팀(TEAM) 테이블이 있다. 그런데 선수(PLAYER) 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 소속팀 코드를 기준으로 팀 테이블에 들어 있는 데이터를 다음과 같이 순서를 바꾸어 주면 아래 [그림 Ⅱ-1-14]와 같이 바꿀 수 있다.

선수(PLAYER) 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
쟈스민	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 45번이고, 소속팀코드는 K07번이다. K07번 팀코드의 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
쟈스민        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 에러는 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
쟈스민	33	K08	일화천마	성남
디디	8	K06	아이파크	부산
…	…	…	…	…

480개 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 SQL문을 만들어 본다.

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

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
김용발 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 수원 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 울산 울산현대
선원길 46 강원 강원FC
최주호 51 포항 스틸러스
최동우 60 전북 현대모터스
김충호 60 인천 유나이티드
43개 항이 선택되었다.
```

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

[예제] 위 SQL 문장에서 FROM 절에서 테이블에 대한 ALIAS를 정의했는데, SELECT 절이나 WHERE 절에서 테이블명을 사용한다면 DBMS의 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 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;
SELECT PLAYER.PLAYER_NAME 선수명, P.BACK_NO 백넘버, *
1행에 오류:
ERROR: 열명이 부적합하다.
```

다. 팀-구장 EQUI JOIN 사례

팀(TEAM) 테이블

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

↔

운동장(STADIUM) 테이블

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

[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) 테이

블의 관계를 이용해서 소속팀이 가지고 있는 전용구장의 정보를 팀의 정보와 함께 출력하는 SQL문을 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        광양전용경기장      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        광주월드컵경기장    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) JOIN은 두 개의 테이블 간에 칼럼 값들이 서로 정확하게 일치하지 않는 경우에 사용된다. Non EQUI JOIN의 경우에는 “=” 연산자가 아닌 다른(Between, >, >=, <, <= 등) 연산자들을 사용하여 JOIN을 수행하는 것이다.

두 개의 테이블이 PK-FK로 연관관계를 가지거나 논리적으로 같은 값이 존재하는 경우에는 “=” 연산자를 이용하여 EQUI JOIN을 사용한다. 그러나 두 개의 테이블 간에 칼럼 값들이 서로 정확하게 일치하지 않는 경우에는 EQUI JOIN을 사용할 수 없다. 이런 경우 Non EQUI JOIN을 시도할 수 있으나 데이터 모델에 따라서 Non EQUI JOIN이 불가능한 경우도 있다.

다음은 Non EQUI JOIN의 대략적인 형태이다. 아래 BETWEEN a AND b 조건은 Non EQUI JOIN의 한 사례일 뿐이다.

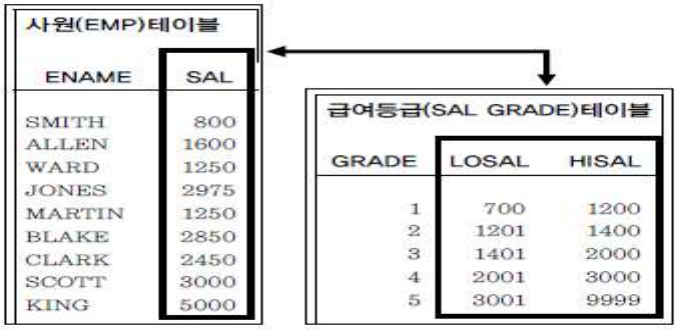
```
SELECT 테이블1.칼럼명, 테이블2.칼럼명, ...
FROM   테이블1, 테이블2
WHERE 테이블1.칼럼명1 BETWEEN 테이블2.칼럼명1 AND 테이블2.칼럼명2;
```

[예제] Non EQUI JOIN에 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 JOIN의 사례는 다음과 같다.

```
[예제]
```

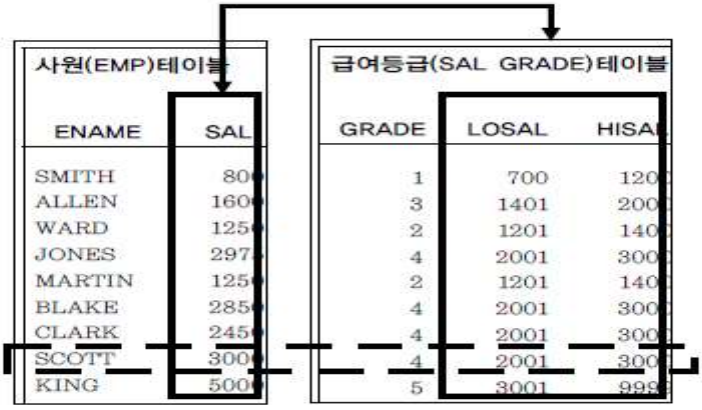
```
SELECT E.ENAME, E.JOB, E.SAL, S.GRADE
FROM EMP E, SALGRADE S
WHERE E.SAL BETWEEN S.LOSAL AND S.HISAL;
```

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 1급(700 이상 ~ 1200 이하), 2급(1201 이상 ~ 1400 이하), 3급(1401 이상 ~ 2000 이하), 4급(2001 이상 ~ 3000 이하), 5급(3001 이상 ~ 9999 이하)으로 구분한 5개의 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 JOIN을 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 SCOTT라는 사원을 예로 들어 급여는 3,000달러($)이고, 3,000달러($)는 급여등급 테이블에서 2,001 ~ 3,000달러($) 사이의 4급에 해당하는 급여등급이라는 값을 얻을 수 있다.

[예제] 사원 14명 모두에 대해 아래 SQL로 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 Ⅱ-1-12]를 보면서 세 개의 테이블에 대한 JOIN을 구현해 보도록 한다. [그림 Ⅱ-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 JOIN 해야만 원하는 데이터를 얻을 수 있다.

[예제] 앞의 예제에서 보았듯이 선수 테이블의 소속팀코드(TEAM_ID)가 팀 테이블의 팀코

드(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    팀 명        구 장 명
 ------      -----     -----     ---------    -------------
 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
 고규억      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
곽경근	FW	인천	유나이티드	인천월드컵경기장
곽기훈	FW	울산	울산현대	울산문수경기장
곽기훈	FW	울산	울산현대	울산문수경기장
곽치국	MF	성남	일화천마	성남종합운동장
…	…	… … …		

480개의 행이 선택되었다.

지금까지 JOIN에 대한 기본적인 사용법을 확인해 보았는데, JOIN이 필요한 기본적인 이유는 과목1에서 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 됐을 경우, 가장 중요한 데이터의 정합성에 더 큰 비용을 지불해야 하며, 데이터를 추가, 삭제, 수정하는 작업 역시 상당한 노력이 요구될 것이다.

성능 측면에서도 간단한 데이터를 조회하는 경우에도 규모가 큰 테이블에서 필요한 데이터를 찾아야 하기 때문에 오히려 검색 속도가 떨어질 수도 있다.

테이블을 정규화하여 데이터를 분할하게 되면 위와 같은 문제는 자연스럽게 해결 된다. 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 중요 원인이 되므로 JOIN 조건은 신중하게 작성해야 한다.

제1절 표준 조인(STANDARD JOIN)

1. STANDARD SQL 개요

1970년: Dr. E.F.Codd 관계형 DBMS(Relational DB) 논문 발표
1974년: IBM SQL 개발
1979년: Oracle 상용 DBMS 발표
1980년: Sybase SQL Server 발표 (이후 Sybase ASE로 개명)
1983년: IBM DB2 발표
1986년: ANSI/ISO SQL 표준 최초 제정 (SQL-86, SQL1)
1992년: ANSI/ISO SQL 표준 개정 (SQL-92, SQL2)
1993년: MS SQL Server 발표 (Windows OS, Sybase Code 활용)
1999년: ANSI/ISO SQL 표준 개정 (SQL-99, SQL3)
2003년: ANSI/ISO SQL 표준 개정 (SQL-2003)
2008년: ANSI/ISO SQL 표준 개정 (SQL-2008)

국내뿐만 아니라 전 세계적으로 많이 사용되고 있는 관계형 데이터베이스의 경우 오브젝트 개념을 포함한 여러 새로운 기능들이 꾸준히 개발되고 있으며, 현재 기업형 DBMS는 순수 관계형 데이터베이스가 아닌 객체 지원 기능이 포함된 객체관계형(Object Relational) 데이터베이스를 대부분 사용하고 있다. 현재 우리가 사용하는 많은 시스템의 두뇌 역할을 하는 관계형 데이터베이스를 유일하게 접속할 수 있는 언어가 바로 SQL이다. 사용자와 개발자 입장에서는 SQL의 진화 및 변화가 가장 큰 관심 내용인데, 초창기 SQL의 기본 기능을 정리했던 최초의 SQL-86 표준과 관계형 DBMS의 폭발적인 전성기를 주도했던 ANSI/ISO SQL2 세대를 지나면서 많은 기술적인 발전이 있었다.

그러나, ANSI/ISO SQL2의 경우 표준 SQL에 대한 명세가 부족한 부분이 있었고, DBMS 벤더 별로 문법이나 사용되는 용어의 차이가 너무 커져서 상호 호환성이나 SQL 학습 효율이 많이 부족한 문제가 발생하였다.

이에 향후 SQL에서 필요한 기능을 정리하고 호환 가능한 여러 기준을 제정한 것이 1999년에 정해진 ANSI/ISO SQL3이다. 이후 가장 먼저 ANSI/ISO SQL3의 기능을 시현한 것이 Oracle의 8i/9i 버전이라고 할 수 있다. 참고로 2003년에 ANSI/ISO SQL 기준이 소폭 추가 개정되었고 현재 사용되는 데이터베이스는 대부분 SQL-2003 표준을 기준으로 하고 있다. 다른 벤더의 DBMS도 2006년 이후 발표된 버전에서 ANSI/ISO SQL-99와 SQL-2003의 핵심적인 기능은 만족스러운 수준으로 구현된 것으로 평가 받고 있다. 마지막으로 2008년에 진행된 추가 개정 내용은 아직 사용자 레벨에 큰 영향을 미치지 않고 있다.

아직도 벤더별로 일부 기능의 개발이 진행 중인 경우도 있고 벤더별 특이한 기술 용어는 여전히 호환이 안 되고 있지만, ANSI/ISO SQL 표준을 통해 STANDARD JOIN을 포함한 많은 기능이 상호 벤치마킹하고 발전하면서 DBMS 간에 평준화를 이루어 가고 있다고 볼 수 있다.

예를 들면, IBM DB2나 SYBASE ASE DBMS는 과거 버전부터 CASE 기능이나 FULL OUTER JOIN 기능을 지원하였지만, Oracle DBMS는 양쪽(FULL) OUTER JOIN의 경우

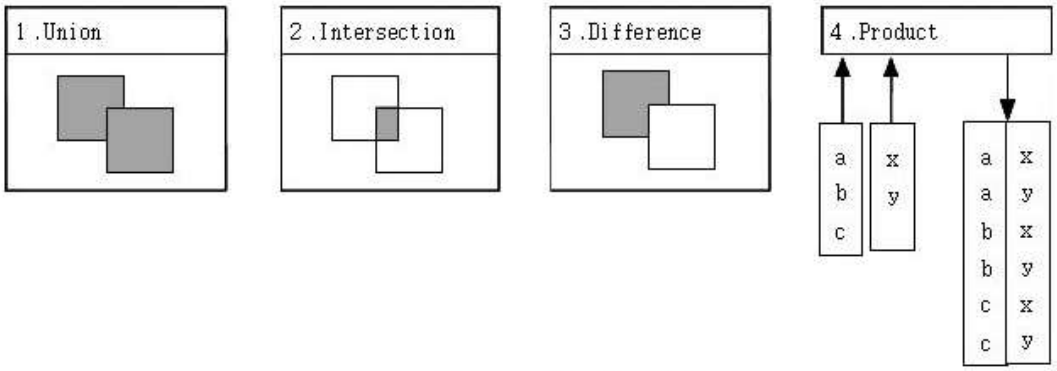
(+) 표시를 이용한 두 개의 SQL 문장을 UNION 오퍼레이션으로 처리하거나, CASE 기능을 구현하기 위해 DECODE 함수를 복잡하게 구현해야 하는 불편함이 있었다. 이런 불편 사항은 Oracle에서 표준 SQL에 포함된 CASE 기능과 FULL OUTER JOIN 기능을 추가함으로써 문제가 해결되었다.(참고로, Oracle DECODE 함수가 CASE 기능보다 장점도 있으므로 Oracle 사용자는 요구사항에 따라 DECODE나 CASE 함수를 선택할 수 있다.)

결과적으로 사용자 입장에서는 ANSI/ISO SQL의 새로운 기능들을 사용함으로써 보다 쉽게 데이터를 추출하거나 SQL 튜닝의 효과를 함께 얻을 수 있게 되었다.

대표적인 ANSI/ISO 표준 SQL의 기능은 다음 내용을 포함한다.

- STANDARD JOIN 기능 추가 (CROSS, OUTER JOIN 등 새로운 FROM 절 JOIN 기능들)
- SCALAR SUBQUERY, TOP-N QUERY 등의 새로운 SUBQUERY 기능들
- ROLLUP, CUBE, GROUPING SETS 등의 새로운 리포팅 기능
- WINDOW FUNCTION 같은 새로운 개념의 분석 기능들

가. 일반 집합 연산자



[그림 Ⅱ-2-1] E.F.CODD 일반 집합 연산자

현재 사용하는 SQL의 많은 기능이 관계형 데이터베이스의 이론을 수립한 E.F.Codd 박사의 논문에 언급이 되어 있다. 논문에 언급된 8가지 관계형 대수는 다시 각각 4개의 일반 집합 연산자와 순수 관계 연산자로 나눌 수 있으며, 관계형 데이터베이스 엔진 및 SQL의 기반 이론이 되었다.

일반 집합 연산자를 현재의 SQL과 비교하면,

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

첫 번째, UNION 연산은 수학적 합집합을 제공하기 위해, 공통 교집합의 중복을 없애기 위한 사전 작업으로 시스템에 부하를 주는 정렬 작업이 발생한다. 이후 UNION ALL 기능

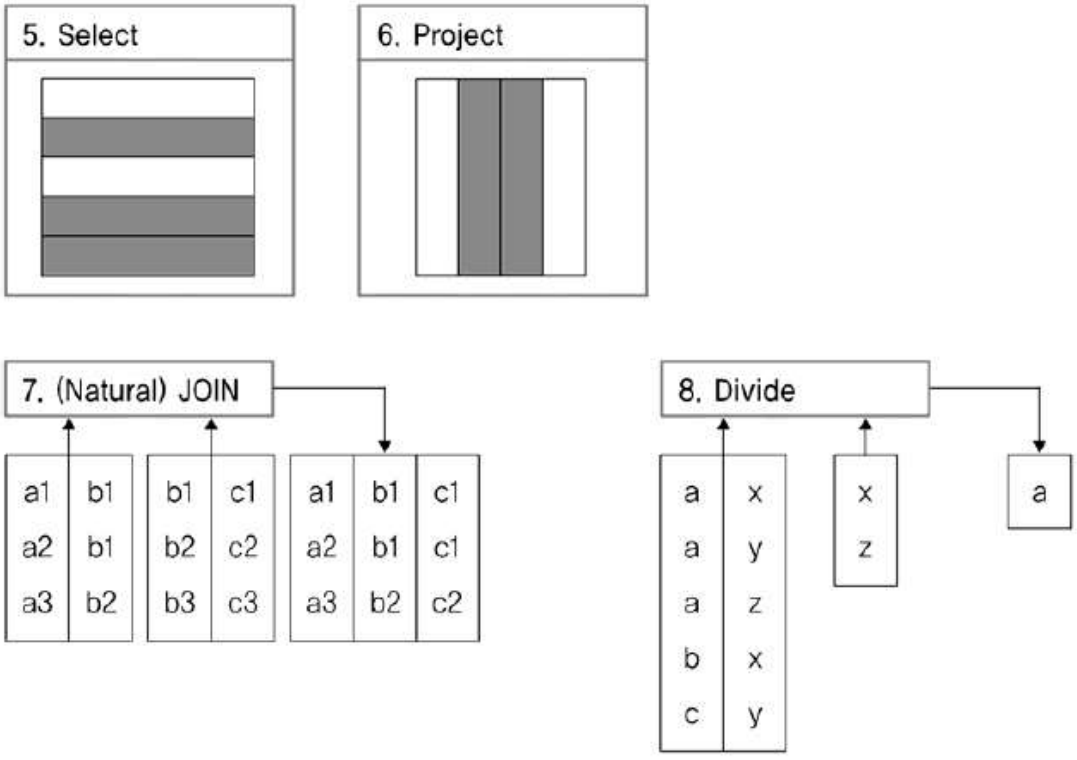
이 추가되었는데, 특별한 요구 사항이 없다면 공통집합을 중복해서 그대로 보여 주기 때문에 정렬 작업이 일어나지 않는 장점을 가진다. 만일 UNION과 UNION ALL의 출력 결과가 같다면, 응답 속도 향상이나 자원 효율화 측면에서 데이터 정렬 작업이 발생하지 않는 UNION ALL을 사용하는 것을 권고한다.

두 번째, INTERSECTION은 수학의 교집합으로써 두 집합의 공통집합을 추출한다.

세 번째, DIFFERENCE는 수학의 차집합으로써 첫 번째 집합에서 두 번째 집합과의 공통집합을 제외한 부분이다. 대다수 벤더는 EXCEPT를, Oracle은 MINUS 용어를 사용한다. (SQL 표준에는 EXCEPT로 표시되어 있으며, 벤더에서 SQL 표준 기능을 구현할 때 다른 용어를 사용하는 것은 현실적으로 허용되고 있다.)

네 번째, PRODUCT의 경우는 CROSS(ANIS/ISO 표준) PRODUCT라고 불리는 곱집합으로, JOIN 조건이 없는 경우 생길 수 있는 모든 데이터의 조합을 말한다. 양쪽 집합의 M*N 건의 데이터 조합이 발생하며, CARTESIAN(수학자 이름) PRODUCT라고도 표현한다.

나. 순수 관계 연산자



[그림 Ⅱ-2-2] E.F.CODD 순수 관계 연산자

순수 관계 연산자는 관계형 데이터베이스를 구현하기 위해 새롭게 만들어진 연산자이다. 순수 관계 연산자를 현재의 SQL 문장과 비교하면 다음과 같다.

1. SELECT 연산은 WHERE 절로 구현되었다.
2. PROJECT 연산은 SELECT 절로 구현되었다.

3. (NATURAL) JOIN 연산은 다양한 JOIN 기능으로 구현되었다.
4. DIVIDE 연산은 현재 사용되지 않는다.

다섯 번째, SELECT 연산은 SQL 문장에서는 WHERE 절의 조건절 기능으로 구현이 되었다. (SELECT 연산과 SELECT 절의 의미가 다름을 유의하자.)

여섯 번째, PROJECT 연산은 SQL 문장에서는 SELECT 절의 칼럼 선택 기능으로 구현이 되었다.

일곱 번째, JOIN 연산은 WHERE 절의 INNER JOIN 조건과 함께 FROM 절의 NATURAL JOIN, INNER JOIN, OUTER JOIN, USING 조건절, ON 조건절 등으로 가장 다양하게 발전하였다.

여덟 번째, DIVIDE 연산은 나눗셈과 비슷한 개념으로 왼쪽의 집합을 'XZ'로 나누었을 때, 즉 'XZ'를 모두 가지고 있는 'A'가 답이 되는 기능으로 현재 사용되지 않는다.

관계형 데이터베이스의 경우 요구사항 분석, 개념적 데이터 모델링, 논리적 데이터 모델링, 물리적 데이터 모델링 단계를 거치게 되는데, 이 단계에서 엔터티 확정 및 정규화 과정, 그리고 M:M (다대다) 관계를 분해하는 절차를 거치게 된다.

특히 정규화 과정의 경우 데이터 정합성과 데이터 저장 공간의 절약을 위해 엔터티를 최대한 분리하는 작업으로, 일반적으로 3차 정규형이나 보이스코드 정규형까지 진행하게 된다. 이런 정규화를 거치면 하나의 주제에 관련 있는 엔터티가 여러 개로 나누어지게 되고, 이 엔터티들이 주로 테이블이 되는데 이렇게 흩어진 데이터를 연결해서 원하는 데이터를 가져오는 작업이 바로 JOIN이라고 할 수 있다.

관계형 데이터베이스에 있어서 JOIN은 SQL의 가장 중요한 기능이므로 충분히 이해할 필요가 있다.

2. FROM 절 JOIN 형태

ANSI/ISO SQL에서 표시하는 FROM 절의 JOIN 형태는 다음과 같다.

- INNER JOIN
- NATURAL JOIN
- USING 조건절
- ON 조건절
- CROSS JOIN
- OUTER JOIN

ANSI/ISO SQL에서 규정한 JOIN 문법은 WHERE 절을 사용하던 기존 JOIN 방식과 차이가 있다. 사용자는 기존 WHERE 절의 검색 조건과 테이블 간의 JOIN 조건을 구분 없이 사용하던 방식을 그대로 사용할 수 있으면서, 추가된 선택 기능으로 테이블 간의 JOIN 조건을 FROM 절에서 명시적으로 정의할 수 있게 되었다.

INNER JOIN은 WHERE 절에서부터 사용하던 JOIN의 DEFAULT 옵션으로 JOIN 조건에

서 동일한 값이 있는 행만 반환한다. DEFAULT 옵션이므로 생략이 가능하지만, CROSS JOIN, OUTER JOIN과는 같이 사용할 수 없다.

NATURAL JOIN은 INNER JOIN의 하위 개념으로 NATURAL JOIN은 두 테이블 간의 동일한 이름을 갖는 모든 칼럼들에 대해 EQUI(=) JOIN을 수행한다. NATURAL INNER JOIN이라고도 표시할 수 있으며, 결과는 NATURAL JOIN과 같다.

새로운 SQL JOIN 문장 중에서 가장 중요하게 기억해야 하는 문장은 ON 조건절을 사용하는 경우이다. 과거 WHERE 절에서 JOIN 조건과 데이터 검증 조건이 같이 사용되어 용도가 불분명한 경우가 발생할 수 있었는데, WHERE 절의 JOIN 조건을 FROM 절의 ON 조건절로 분리하여 표시함으로써 사용자가 이해하기 쉽도록 한다.

ON 조건절의 경우 NATURAL JOIN처럼 JOIN 조건이 숨어 있지 않고, 명시적으로 JOIN 조건을 구분할 수 있고, NATURAL JOIN이나 USING 조건절처럼 칼럼명이 똑같아야 된다는 제약 없이 칼럼명이 상호 다르더라도 JOIN 조건으로 사용할 수 있으므로 앞으로 가장 많이 사용될 것으로 예상된다. 다만, FROM 절에 테이블이 많이 사용될 경우 다소 복잡하게 보여 가독성이 떨어지는 단점이 있다.

그런 측면에서 SQL Server의 경우 ON 조건절만 지원하고 NATURAL JOIN과 USING 조건절을 지원하지 않고 있는 것으로 보인다. 본 가이드는 ANSI/ISO SQL 기준에 NATURAL JOIN과 USING 조건절이 표시되어 있으므로 이 부분도 설명을 하도록 한다.

3. INNER JOIN

INNER JOIN은 OUTER(외부) JOIN과 대비하여 내부 JOIN이라고 하며 JOIN 조건에서 동일한 값이 있는 행만 반환한다. INNER JOIN 표시는 그 동안 WHERE 절에서 사용하던 JOIN 조건을 FROM 절에서 정의하겠다는 표시이므로 USING 조건절이나 ON 조건절을 필수적으로 사용해야 한다.

[예제] 사원 번호와 사원 이름, 소속부서 코드와 소속부서 이름을 찾아본다.

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

위 SQL과 아래 SQL은 같은 결과를 얻을 수 있다.

FROM 절 JOIN 조건
SELECT EMP.DEPTNO, EMPNO, ENAME, DNAME
```

```
FROM EMP INNER JOIN DEPT
ON EMP.DEPTNO = DEPT.DEPTNO;

INNER는 JOIN의 디폴트 옵션으로 아래 SQL문과 같이 생략 가능하다.

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

```
[실행 결과]
DEPTNO      EMPNO       ENAME       DNAME
------      -----       ------      ---------
20          7369        SMITH       RESEARCH
30          7499        ALLEN       SALES
30          7521        WARD        SALES
20          7566        JONES       RESEARCH
30          7654        MARTIN      SALES
30          7698        BLAKE       SALES
10          7782        CLARK       ACCOUNTING
20          7788        SCOTT       RESEARCH
10          7839        KING        ACCOUNTING
30          7844        TURNER      SALES
20          7876        ADAMS       RESEARCH
30          7900        JAMES       SALES
20          7902        FORD        RESEARCH
10           7934       MILLER      ACCOUNTING
14개의 행이 선택되었다.
```

위에서 사용한 ON 조건절에 대해서는 뒤에서 추가 설명하도록 한다.

4. NATURAL JOIN

NATURAL JOIN은 두 테이블 간의 동일한 이름을 갖는 모든 칼럼들에 대해 EQUI(=) JOIN을 수행한다. NATURAL JOIN이 명시되면, 추가로 USING 조건절, ON 조건절, WHERE 절에서 JOIN 조건을 정의할 수 없다. 그리고, SQL Server에서는 지원하지 않는 기능이다.

[예제] 사원 번호와 사원 이름, 소속부서 코드와 소속부서 이름을 찾아본다.

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

```
[실행 결과]
DEPTNO EMPNO ENAME      DNAME
------ ------ ------    ------
20     7369   SMITH     RESEARCH
30     7499   ALLEN     SALES
30     7521   WARD      SALES
20     7566   JONES     RESEARCH
30     7654   MARTIN    SALES
30     7698   BLAKE     SALES
10     7782   CLARK     ACCOUNTING
20     7788   SCOTT     RESEARCH
10     7839   KING      ACCOUNTING
30     7844   TURNER    SALES
20     7876   ADAMS     RESEARCH
30     7900   JAMES     SALES
20     7902   FORD      RESEARCH
10     7934   MILLER    ACCOUNTING
14개의 행이 선택되었다.
```

위 SQL은 별도의 JOIN 칼럼을 지정하지 않았지만, 두 개의 테이블에서 DEPTNO라는 공통된 칼럼을 자동으로 인식하여 JOIN을 처리한 것이다. JOIN에 사용된 칼럼들은 같은 데이터 유형이어야 하며, ALIAS나 테이블 명과 같은 접두사를 붙일 수 없다.

```
[예제]
SELECT EMP.DEPTNO, EMPNO, ENAME, DNAME
FROM EMP NATURAL JOIN DEPT;

ERROR: NATURAL JOIN에 사용된 열은 식별자를 가질 수 없음
```

NATURAL JOIN은 JOIN이 되는 테이블의 데이터 성격(도메인)과 칼럼명 등이 동일해야 하는 제약 조건이 있다. 간혹 모델링 상의 부주의로 인해 동일한 칼럼명이더라도 다른 용도의 데이터를 저장하는 경우도 있으므로 주의해서 사용해야 한다.

[예제] 아래 '*' 와일드카드처럼 별도의 칼럼 순서를 지정하지 않으면 NATURAL JOIN의 기준이 되는 칼럼 들이 다른 칼럼보다 먼저 출력된다. (ex: DEPTNO가 첫 번째 칼럼이 된다.) 이때 NATURAL JOIN은 JOIN에 사용된 같은 이름의 칼럼을 하나로 처리한다.

```
[예제]
SELECT *
FROM EMP NATURAL JOIN DEPT;
```

```
[실행 결과]
DEPTNO EMPNO ENAME JOB MGR HIREDATE SAL COMM DNAME LOC
-----  ----- ----- -------- --- -------- ---- ---- --------- ------
20      7369 SMITH CLERK 7902 1980-12-17 800 RESEARCH DALLAS
30      7499 ALLEN SALESMAN 7698 1981-02-20 1600 300 SALES CHICAGO
30      7521 WARD SALESMAN 7698 1981-02-22 1250 500 SALES CHICAGO
20      7566 JONES MANAGER 7839 1981-04-02 2975 RESEARCH DALLAS
30      7654 MARTIN SALESMAN 7698 1981-09-28 1250 1400 SALES CHICAGO
30      7698 BLAKE MANAGER 7839 1981-05-01 2850 SALES CHICAGO
10      7782 CLARK MANAGER 7839 1981-06-09 2450 ACCOUNTING NEW YORK
20      7788 SCOTT ANALYST 7566 1987-07-13 3000 RESEARCH DALLAS
10      7839 KING PRESIDENT 1981-11-17 5000 ACCOUNTING NEW YORK
30      7844 TURNER SALESMAN 7698 1981-09-08 1500 SALES CHICAGO
20      7876 ADAMS CLERK 7788 1987-07-13 1100 RESEARCH DALLAS
30      7900 JAMES CLERK 7698 1981-12-03 950 0 SALES CHICAGO
20      7902 FORD ANALYST 7566 1981-12-03 3000 RESEARCH DALLAS
10      7934 MILLER CLERK 7782 1982-01-23 1300 ACCOUNTING NEW YORK
14개의 행이 선택되었다.
```

[예제] 반면, INNER JOIN의 경우 첫 번째 테이블, 두 번째 테이블의 칼럼 순서대로 데이터가 출력된다. 이때 NATURAL JOIN은 JOIN에 사용된 같은 이름의 칼럼을 하나로 처리하지만, INNER JOIN은 별개의 칼럼으로 표시한다.

```
[예제]
SELECT *
FROM EMP INNER JOIN DEPT
ON EMP.DEPTNO = DEPT.DEPTNO;
```

```
[실행 결과]
```

```
EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO DEPTNO DNAME LOC
----  -----  ------ --- ------- --- ---- ----- ----- -------- -----
7369  SMITH  CLERK 7902 1980-12-17 800 20 20 RESEARCH DALLAS
7499  ALLEN  SALESMAN 7698 1981-02-20 1600 300 30 30 SALES CHICAGO
7521  WARD   SALESMAN 7698 1981-02-22 1250 500 30 30 SALES CHICAGO
7566  JONES  MANAGER 7839 1981-04-02 2975 20 20 RESEARCH DALLAS
7654  MARTIN SALESMAN 7698 1981-09-28 1250 1400 30 30 SALES CHICAGO
7698  BLAKE  MANAGER 7839 1981-05-01 2850 30 30 SALES CHICAGO
7782  CLARK  MANAGER 7839 1981-06-09 2450 10 10 ACCOUNTING NEW YORK
7788  SCOTT  ANALYST 7566 1987-07-13 3000 20 20 RESEARCH DALLAS
7839  KING   PRESIDENT 1981-11-17 5000 10 10 ACCOUNTING NEW YORK
7844  TURNER SALESMAN 7698 1981-09-08 1500 0 30 30 SALES CHICAGO
7876  ADAMS CLERK 7788 1987-07-13 1100 20 20 RESEARCH DALLAS
7900  JAMES  CLERK 7698 1981-12-03 950 30 30 SALES CHICAGO
7902  FORD   ANALYST 7566 1981-12-03 3000 20 20 RESEARCH DALLAS
7934  MILLER CLERK 7782 1982-01-23 1300 10 10 ACCOUNTING NEW YORK
14개의 행이 선택되었다.
```

[예제] NATURAL JOIN과 INNER JOIN의 차이를 자세히 설명하기 위해 DEPT_TEMP 테이블을 임시로 만든다.

```
[예제] Oracle
CREATE TABLE DEPT_TEMP
 AS SELECT * FROM DEPT;
```

```
[예제] SQL Server
SELECT * INTO DEPT_TEMP
FROM DEPT;
```

```
[예제]
UPDATE DEPT_TEMP
SET DNAME = 'R&D'
WHERE DNAME = 'RESEARCH';

UPDATE DEPT_TEMP
SET DNAME = 'MARKETING'
WHERE DNAME = 'SALES';

SELECT * FROM DEPT_TEMP;
```

```
[실행 결과]
DEPTNO DNAME LOC
-------- ---------- ---------
10 ACCOUNTING NEW YORK
20 R&D DALLAS
```

```
30 MARKETING CHICAGO
40 OPERATIONS BOSTON
4개의 행이 선택되었다.
```

부서번호 20과 30의 DNAME이 'R&D'와 'MARKETING'으로 변경된 것을 확인할 수 있다.

[예제] 세 개의 칼럼명이 모두 같은 DEPT와 DEPT_TEMP 테이블을 NATURAL [INNER] JOIN으로 수행한다.

```
[예제]
SELECT *
FROM DEPT NATURAL INNER JOIN DEPT_TEMP;

INNER는 DEFAULT 옵션으로 아래와 같이 생략 가능하므로 위 SQL과 아래 SQL은 같은 결과를 얻을 수 있다.

SELECT *
FROM DEPT NATURAL JOIN DEPT_TEMP;
```

```
[실행 결과]
DEPTNO DNAME       LOC
------ ---------- ----------
10     ACCOUNTING NEW YORK
40     OPERATIONS BOSTON
2개의 행이 선택되었다.
```

위 SQL의 경우 DNAME의 내용이 바뀐 부서번호 20, 30의 데이터는 실행 결과에서 제외된 것을 알 수 있다.

[예제] 다음에는 같은 조건이지만 출력 칼럼에서 차이가 나는 일반적인 INNER JOIN을 수행한다.

```
[예제]
SELECT *
FROM DEPT JOIN DEPT_TEMP
ON DEPT.DEPTNO = DEPT_TEMP.DEPTNO
AND DEPT.DNAME = DEPT_TEMP.DNAME
AND DEPT.LOC = DEPT_TEMP.LOC;

위 SQL과 아래 SQL은 같은 결과를 얻을 수 있다.

SELECT *
FROM DEPT, DEPT_TEMP
WHERE DEPT.DEPTNO = DEPT_TEMP.DEPTNO
```

```
AND DEPT.DNAME = DEPT_TEMP.DNAME
AND DEPT.LOC = DEPT_TEMP.LOC;
```

```
[실행 결과]
DEPTNO DNAME LOC DEPTNO DNAME LOC
------ ---------- -------- ------ ---------- ------
10 ACCOUNTING NEW YORK
10 ACCOUNTING NEW YORK
40 OPERATIONS BOSTON
40 OPERATIONS BOSTON
2개의 행이 선택되었다.
```

위 SQL의 경우 DNAME의 내용이 바뀐 부서번호 20, 30의 경우는 결과에서 제외된 것을 알 수 있다. 차이가 나는 부분은 NATURAL JOIN은 JOIN에 사용된 같은 이름의 칼럼을 하나로 처리하지만, INNER JOIN의 경우는 2개의 칼럼으로 표시된다.

5. USING 조건절

NATURAL JOIN에서는 모든 일치되는 칼럼들에 대해 JOIN이 이루어지지만, FROM 절의 USING 조건절을 이용하면 같은 이름을 가진 칼럼들 중에서 원하는 칼럼에 대해서만 선택적으로 EQUI JOIN을 할 수가 있다. 다만, 이 기능은 SQL Server에서는 지원하지 않는다.

[예제] 세 개의 칼럼명이 모두 같은 DEPT와 DEPT_TEMP 테이블을 DEPTNO 칼럼을 이용한 [INNER] JOIN의 USING 조건절로 수행한다.

```
[예제]
SELECT *
FROM DEPT JOIN DEPT_TEMP
USING (DEPTNO);
```

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

위 SQL의 '*' 와일드카드처럼 별도의 칼럼 순서를 지정하지 않으면 USING 조건절의 기준이 되는 칼럼이 다른 칼럼보다 먼저 출력된다. (ex: DEPTNO가 첫 번째 칼럼이 된다.) 이

때 USING JOIN은 JOIN에 사용된 같은 이름의 칼럼을 하나로 처리한다.

[예제] USING 조건절을 이용한 EQUI JOIN에서도 NATURAL JOIN과 마찬가지로 JOIN 칼럼에 대해서는 ALIAS나 테이블 이름과 같은 접두사를 붙일 수 없다. (DEPT.DEPTNO → DEPTNO)

```
[예제]
잘못된 사례:
SELECT DEPT.DEPTNO, DEPT.DNAME, DEPT.LOC, DEPT_TEMP.DNAME,
DEPT_TEMP.LOC
FROM DEPT JOIN DEPT_TEMP USING (DEPTNO);
ERROR:
USING 절의 열 부분은 식별자를 가질 수 없음

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

```
[실행 결과]
DEPTNO   DNAME        LOC        DNAME        LOC
-------  ---------    ---------  -----------  --------
10       ACCOUNTING   NEW YORK   ACCOUNTING   NEW YORK
20       RESEARCH     DALLAS     R&D          DALLAS
30       SALES        CHICAGO    MARKETING    CHICAGO
40       OPERATIONS   BOSTON     OPERATIONS   BOSTON

4개의 행이 선택되었다.
```

[예제] 이번에는 DEPT와 DEPT_TEMP 테이블의 일부 데이터 내용이 변경되었던 DNAME 칼럼을 조인 조건으로 [INNER] JOIN의 USING 조건절을 수행한다.

```
[예제]
SELECT *
FROM DEPT JOIN DEPT_TEMP
USING (DNAME);
```

```
[실행 결과]
DNAME        DEPTNO   LOC        DEPTNO   LOC
----------   ------   ---------  -------  ---------
ACCOUNTING   10       NEW YORK   10       NEW YORK
OPERATIONS   40       BOSTON     40       BOSTON
2개의 행이 선택되었다.
```

위 SQL의 경우 DNAME의 내용이 바뀐 부서번호 20, 30의 경우는 결과에서 제외된 것을 알 수 있다. 그리고 USING에 사용된 DNAME이 첫 번째 칼럼으로 출력된 것과 함께, JOIN 조건에 참여하지 않은 DEPTNO와 LOC가 2개의 칼럼으로 표시된 것을 알 수 있다.

[예제] 이번에는 세 개의 칼럼명이 모두 같은 DEPT와 DEPT_TEMP 테이블을 LOC와 DEPTNO 2개 칼럼을 이용한 [INNER] JOIN의 USING 조건절로 수행한다.

```
[예제]
SELECT *
FROM DEPT JOIN DEPT_TEMP
USING (LOC, DEPTNO);
```

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

USING에 사용된 LOC, DEPTNO가 첫 번째, 두 번째 칼럼으로 출력되고, JOIN 조건에 참여하지 않은 DNAME 칼럼은 2개의 칼럼으로 표시된 것을 알 수 있다.

[예제] 이번에는 DEPTNO, DNAME 2개의 칼럼을 이용한 [INNER] JOIN의 USING 조건절로 수행한다.

```
[예제]
SELECT *
FROM DEPT JOIN DEPT_TEMP
USING (DEPTNO, DNAME);
```

```
[실행 결과]
DEPTNO       DNAME        LOC          LOC
------       ----------   --------     --------
10           ACCOUNTING   NEW YORK     NEW YORK
40           OPERATIONS   BOSTON       BOSTON
2개의 행이 선택되었다.
```

위 SQL의 경우 DNAME의 내용이 바뀐 부서번호 20, 30의 경우는 결과에서 제외된 것을 알 수 있다. 그리고 USING에 사용된 DEPTNO, DNAME이 첫 번째, 두 번째 칼럼으로 출

력된 것과 함께, JOIN 조건에 참여하지 않은 LOC가 2개의 칼럼으로 표시된 것을 알 수 있다

6. ON 조건절

JOIN 서술부(ON 조건절)와 비 JOIN 서술부(WHERE 조건절)를 분리하여 이해가 쉬우며, 칼럼명이 다르더라도 JOIN 조건을 사용할 수 있는 장점이 있다.

[예제] 사원 테이블과 부서 테이블에서 사원 번호와 사원 이름, 소속부서 코드, 소속부서 이름을 출력한다.

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

```
[실행 결과]
EMPNO      ENAME      DEPTNO     DNAME
-----      -------    ------     -----------
7369       SMITH      20          RESEARCH
7499       ALLEN      30         SALES
7521       WARD       30         SALES
7566       JONES      20         RESEARCH
7654       MARTIN     30          SALES
7698       BLAKE      30         SALES
7782       CLARK      10         ACCOUNTING
7788       SCOTT      20         RESEARCH
7839       KING       10         ACCOUNTING
7844       TURNER     30         SALES
7876       ADAMS      20         RESEARCH
7900       JAMES      30         SALES
7902       FORD       20         RESEARCH
7934       MILLER     10         ACCOUNTING
14개의 행이 선택되었다.
```

NATURAL JOIN의 JOIN 조건은 기본적으로 같은 이름을 가진 모든 칼럼들에 대한 동등 조건이지만, 임의의 JOIN 조건을 지정하거나, 이름이 다른 칼럼명을 JOIN 조건으로 사용하거나, JOIN 칼럼을 명시하기 위해서는 ON 조건절을 사용한다. ON 조건절에 사용된 괄호는 옵션 사항이다.

USING 조건절을 이용한 JOIN에서는 JOIN 칼럼에 대해서 ALIAS나 테이블 명과 같은 접두사를 사용하면 SYNTAX 에러가 발생하지만, 반대로 ON 조건절을 사용한 JOIN의 경우는 ALIAS나 테이블 명과 같은 접두사를 사용하여 SELECT에 사용되는 칼럼을 논리적으로 명확하게 지정해주어야 한다. (DEPTNO → E.DEPTNO)

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

가. WHERE 절과의 혼용

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

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

```
[실행 결과]
ENAME        DEPTNO       DEPTNO       DNAME
-------      ------       ------       ------
ALLEN        30           30           SALES
WARD         30           30           SALES
MARTIN       30           30           SALES
BLAKE        30           30           SALES
TURNER       30           30           SALES
JAMES        30           30           SALES
6개의 행이 선택되었다.
```

나. ON 조건절 + 데이터 검증 조건 추가

ON 조건절에 JOIN 조건 외에도 데이터 검색 조건을 추가할 수는 있으나, 검색 조건 목적인 경우는 WHERE 절을 사용할 것을 권고한다. (다만, 아우터 조인에서 조인의 대상을 제한하기 위한 목적으로 사용되는 추가 조건의 경우는 ON 절에 표기되어야 한다.)

[예제] 매니저 사원번호가 7698번인 사원들의 이름 및 소속 부서 코드, 부서 이름을 찾아본다.

```
[예제]
SELECT E.ENAME, E.MGR, D.DEPTNO, D.DNAME
FROM EMP E JOIN DEPT D
```

```
ON (E.DEPTNO = D.DEPTNO AND E.MGR = 7698);

  위 SQL과 아래 SQL은 같은 결과를 얻을 수 있다.

SELECT E.ENAME, E.MGR, D.DEPTNO, D.DNAME
FROM EMP E JOIN DEPT D ON (E.DEPTNO = D.DEPTNO)
WHERE E.MGR = 7698;
```

```
[실행 결과]
ENAME        MGR    DEPTNO       DNAME
-------      ----   ------       ------
ALLEN        7698   30           SALES
WARD         7698   30           SALES
MARTIN       7698   30           SALES
TURNER       7698   30           SALES
JAMES        7698   30           SALES
5개의 행이 선택되었다.
```

다. ON 조건절 예제

[예제] 팀과 스타디움 테이블을 스타디움ID로 JOIN하여 팀이름, 스타디움ID, 스타디움 이름을 찾아본다.

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

위 SQL은 STADIUM_ID라는 공통된 칼럼이 있기 때문에 아래처럼 USING 조건절로 구현할 수도 있다.

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

위 SQL은 고전적인 방식인 WHERE 절의 INNER JOIN으로 구현할 수도 있다.

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

```
ORDER BY STADIUM_ID
```

```
[실행 결과]
TEAM_NAME STADIUM_ID STADIUM_NAME
------------- --------- -------------
광주상무 A02 광주월드컵경기장
강원FC A03 강릉종합경기장
제주유나이티드FC A04 제주월드컵경기장
대구FC A05 대구월드컵경기장
유나이티드 B01 인천월드컵경기장
일화천마 B02 성남종합운동장
삼성블루윙즈 B04 수원월드컵경기장
FC서울 B05 서울월드컵경기장
아이파크 C02 부산아시아드경기장
울산현대 C04 울산문수경기장
경남FC C05 창원종합운동장
스틸러스 C06 포항스틸야드
드래곤즈 D01 광양전용경기장
시티즌 D02 대전월드컵경기장
 15개의 행이 선택되었다.
```

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

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

위 SQL은 고전적인 방식인 WHERE 절의 INNER JOIN으로 구현할 수도 있다.

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

위 SQL은 TEAM_ID와 HOMETEAM_ID라는 다른 이름의 칼럼을 사용하기 때문에
```

USING 조건절을 사용할 수는 없다.
[실행 결과] TEAM_NAME TEAM_ID STADIUM_NAME ---------- ------- ---------- 울산현대 K01 울산문수경기장 삼성블루윙즈 K02 수원월드컵경기장 스틸러스 K03 포항스틸야드 유나이티드 K04 인천월드컵경기장 현대모터스 K05 전주월드컵경기장 아이파크 K06 부산아시아드경기장 드래곤즈 K07 광양전용경기장 일화천마 K08 성남종합운동장 FC서울 K09 서울월드컵경기장 시티즌 K10 대전월드컵경기장 경남FC K11 창원종합운동장 광주상무 K12 광주월드컵경기장 강원FC K13 강릉종합경기장 제주유나이티드FC K14 제주월드컵경기장 15개의 행이 선택되었다.

라. 다중 테이블 JOIN

[예제] 사원과 DEPT 테이블의 소속 부서명, DEPT_TEMP 테이블의 바뀐 부서명 정보를 출력한다.

[예제] SELECT E.EMPNO, D.DEPTNO, D.DNAME, T.DNAME New_DNAME FROM EMP E JOIN DEPT D ON (E.DEPTNO = D.DEPTNO) JOIN DEPT_TEMP T ON (E.DEPTNO = T.DEPTNO); 위 SQL은 고전적인 방식인 WHERE 절의 INNER JOIN으로 구현할 수도 있다. SELECT E.EMPNO, D.DEPTNO, D.DNAME, T.DNAME New_DNAME FROM EMP E, DEPT D, DEPT_TEMP T WHERE E.DEPTNO = D.DEPTNO AND E.DEPTNO = T.DEPTNO;
[실행 결과]

```
EMPNO DEPTNO DNAME NEW_DNAME
------ ------ --------- -----------
7369 20 RESEARCH R&D
7499 30 SALES MARKETING
7521 30 SALES MARKETING
7566 20 RESEARCH R&D
7654 30 SALES MARKETING
7698 30 SALES MARKETING
7782 10 ACCOUNTING ACCOUNTING
7788 20 RESEARCH R&D
7839 10 ACCOUNTING ACCOUNTING
7844 30 SALES MARKETING
7876 20 RESEARCH R&D
7900 30 SALES MARKETING
7902 20 RESEARCH R&D
7934 10 ACCOUNTING ACCOUNTING
14개의 행이 선택되었다.
```

[예제] GK 포지션의 선수별 연고지명, 팀명, 구장명을 출력한다.

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

위 SQL은 고전적인 방식인 WHERE 절의 INNER JOIN으로 구현할 수도 있다.

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

```
ORDER BY 선수명;
```

```
[실행 결과]
선수명 포지션 연고지명 팀명 구장명
----- ---- ------ -------- ----------
강성일 GK 대전 시티즌 대전월드컵경기장
권정혁 GK 울산 울산현대 울산문수경기장
권찬수 GK 성남 일화천마 성남종합운동장
김대희 GK 포항 스틸러스 포항스틸야드
김승준 GK 대전 시티즌 대전월드컵경기장
김용발 GK 전북 현대모터스 전주월드컵경기장
김운재 GK 수원 삼성블루윙즈 수원월드컵경기장
김정래 GK 전남 드래곤즈 광양전용경기장
김준호 GK 포항 스틸러스 포항스틸야드
김창민 GK 전북 현대모터스 전주월드컵경기장
김충호 GK 인천 유나이티드 인천월드컵경기장
남현우 GK 인천 유나이티드 인천월드컵경기장
박유석 GK 부산 아이파크 부산아시아드경기장
43개의 행이 선택되었다.
```

[예제] 홈팀이 3점 이상 차이로 승리한 경기의 경기장 이름, 경기 일정, 홈팀 이름과 원정팀 이름 정보를 출력한다.

```
[예제]
SELECT ST.STADIUM_NAME, SC.STADIUM_ID, SCHE_DATE, HT.TEAM_NAME,
AT.TEAM_NAME, HOME_SCORE, AWAY_SCORE
FROM SCHEDULE SC JOIN STADIUM ST
ON SC.STADIUM_ID = ST.STADIUM_ID
JOIN TEAM HT
ON SC.HOMETEAM_ID = HT.TEAM_ID
JOIN TEAM AT
ON SC.AWAYTEAM_ID = AT.TEAM_ID
WHERE HOME_SCORE >= AWAY_SCORE + 3;

위 SQL은 고전적인 방식인 WHERE 절의 INNER JOIN으로 구현할 수도 있다.

SELECT ST.STADIUM_NAME, SC.STADIUM_ID, SCHE_DATE, HT.TEAM_NAME,
AT.TEAM_NAME, HOME_SCORE, AWAY_SCORE
FROM SCHEDULE SC, STADIUM ST, TEAM HT, TEAM AT
WHERE HOME_SCORE> = AWAY_SCORE + 3
```

```
AND SC.STADIUM_ID = ST.STADIUM_ID
AND SC.HOMETEAM_ID = HT.TEAM_ID
AND SC.AWAYTEAM_ID = AT.TEAM_ID;
```

FROM 절에 4개의 테이블이 JOIN에 참여하였으며, HOME TEAM과 AWAY TEAM의 팀 이름을 구하기 위해 TEAM 테이블을 HT와 AT 두 개의 ALIAS로 구분하였다.

```
[실행 결과]
STADIUM_NAME STADIUM_ID SCHE_DATE TEAM_NAME TEAM_NAME HOME_SCORE AWAY_SCORE
------------ --------- -------- --------- --------- --------- ---------
서울월드컵경기장 B05 20120714 FC서울 삼성블루윙즈 3 0
부산아시아드경기장 C02 20120727 아이파크 시티즌 3 0
울산문수경기장 C04 20120803 울산현대 스틸러스 3 0
성남종합운동장 B02 20120317 일화천마 유나이티드 6 0
창원종합운동장 C05 20120427 경남FC 아이파크 5 2
5개의 행이 선택되었다.
```

7. CROSS JOIN

CROSS JOIN은 E.F.CODD 박사가 언급한 일반 집합 연산자의 PRODUCT의 개념으로 테이블 간 JOIN 조건이 없는 경우 생길 수 있는 모든 데이터의 조합을 말한다.

두 개의 테이블에 대한 CARTESIAN PRODUCT 또는 CROSS PRODUCT와 같은 표현으로, 결과는 양쪽 집합의 M*N 건의 데이터 조합이 발생한다. (아래 56건의 데이터는 EMP 14건 * DEPT 4건의 데이터 조합 건수이다.)

[예제] 사원 번호와 사원 이름, 소속부서 코드와 소속부서 이름을 찾아본다.

```
[예제]
SELECT ENAME, DNAME
FROM EMP CROSS JOIN DEPT
ORDER BY ENAME;
```

```
[실행 결과]
ENAME DNAME
-------- ---------
ADAMS SALES
ADAMS RESEARCH
```

```
ADAMS OPERATIONS
ADAMS ACCOUNTING
ALLEN OPERATIONS
ALLEN RESEARCH
ALLEN ACCOUNTING
ALLEN SALES
BLAKE SALES
BLAKE OPERATIONS
BLAKE RESEARCH
BLAKE ACCOUNTING
CLARK SALES
CLARK RESEARCH
CLARK OPERATIONS
CLARK ACCOUNTING
56개의 행이 선택되었다.
```

[예제] NATURAL JOIN의 경우 WHERE 절에서 JOIN 조건을 추가할 수 없지만, CROSS JOIN의 경우 WHERE 절에 JOIN 조건을 추가할 수 있다. 그러나, 이 경우는 CROSS JOIN이 아니라 INNER JOIN과 같은 결과를 얻기 때문에 CROSS JOIN을 사용하는 의미가 없어지므로 권고하지 않는다.

```
[예제]
SELECT ENAME, DNAME
FROM EMP CROSS JOIN DEPT
WHERE EMP.DEPTNO = DEPT.DEPTNO;

위 SQL과 아래 SQL은 같은 결과를 얻을 수 있다.

SELECT ENAME, DNAME
FROM EMP INNER JOIN DEPT
WHERE EMP.DEPTNO = DEPT.DEPTNO;
```

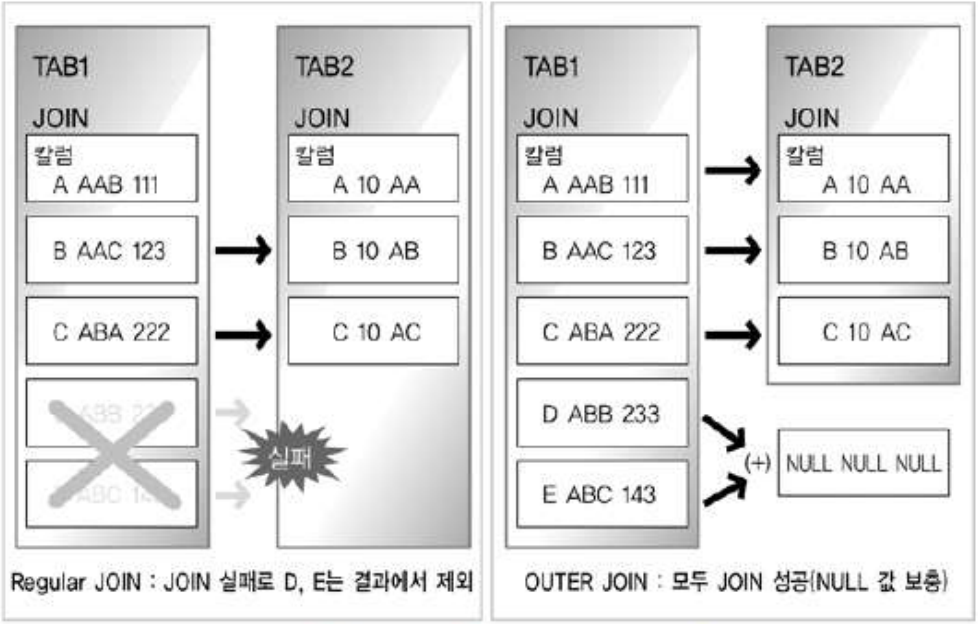
```
[실행 결과]
ENAME DNAME
------- ---------
SMITH RESEARCH
ALLEN SALES
WARD SALES
JONES RESEARCH
```

```
MARTIN SALES
BLAKE SALES
CLARK ACCOUNTING
SCOTT RESEARCH
KING ACCOUNTING
TURNER SALES
ADAMS RESEARCH
JAMES SALES
FORD RESEARCH
MILLER ACCOUNTING
14개의 행이 선택되었다.
```

정상적인 데이터 모델이라면 CROSS PRODUCT가 필요한 경우는 많지 않지만, 간혹 튜닝이나 리포트를 작성하기 위해 고의적으로 사용하는 경우가 있을 수 있다. 그리고 데이터 웨어하우스의 개별 DIMENSION(차원)을 FACT(사실) 칼럼과 JOIN하기 전에 모든 DIMENSION의 CROSS PRODUCT를 먼저 구할 때 유용하게 사용할 수 있다.

8. OUTER JOIN

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



[그림 Ⅱ-2-3] OUTER JOIN 설명

[그림 Ⅱ-2-3]은 TAB1 테이블이 TAB2 테이블을 JOIN 하되, TAB2의 JOIN 데이터가 있는 경우는 TAB2의 데이터를 함께 출력하고, TAB2의 JOIN 데이터가 없는 경우에도 TAB1의 모든 데이터를 표시하고 싶은 경우이다. TAB1의 모든 값에 대해 TAB2의 데이터가 반드시 존재한다는 보장이 없는 경우 OUTER JOIN을 사용하여 해결이 가능하다.

과거 OUTER JOIN을 위해 Oracle은 JOIN 칼럼 뒤에 '(+)'를 표시하였고, Sybase는 비교 연산자의 앞이나 뒤에 '(+)'를 표시했었는데, JOIN 조건과 WHERE 절 검색 조건이 불명확한 단점, IN이나 OR 연산자 사용시 에러 발생, '(+)' 표시가 누락된 칼럼 존재시 OUTER JOIN 오류 발생, FULL OUTER JOIN 미지원 등 불편함이 많았다.

STANDARD JOIN을 사용함으로써 OUTER JOIN의 많은 문제점을 해결할 수 있고, 대부분의 관계형 DBMS 간에 호환성을 확보할 수 있으므로 명시적인 OUTER JOIN을 사용할 것을 적극적으로 권장한다.

추가로 OUTER JOIN 역시 JOIN 조건을 FROM 절에서 정의하겠다는 표시이므로 USING 조건절이나 ON 조건절을 필수적으로 사용해야 한다. 그리고, LEFT/RIGHT OUTER JOIN의 경우에는 기준이 되는 테이블이 조인 수행시 무조건 드라이빙 테이블이 된다. 옵티마이저는 이 원칙에 위배되는 다른 실행계획을 고려하지 않는다.

가. LEFT OUTER JOIN

조인 수행시 먼저 표기된 좌측 테이블에 해당하는 데이터를 먼저 읽은 후, 나중 표기된 우측 테이블에서 JOIN 대상 데이터를 읽어 온다. 즉, Table A와 B가 있을 때(Table 'A'가 기준이 됨), A와 B를 비교해서 B의 JOIN 칼럼에서 같은 값이 있을 때 그 해당 데이터를 가져오고, B의 JOIN 칼럼에서 같은 값이 없는 경우에는 B 테이블에서 가져오는 칼럼들은

NULL 값으로 채운다.

그리고 LEFT JOIN으로 OUTER 키워드를 생략해서 사용할 수 있다.

[예제] STADIUM에 등록된 운동장 중에는 홈팀이 없는 경기장도 있다. STADIUM과 TEAM을 JOIN 하되 홈팀이 없는 경기장의 정보도 같이 출력하도록 한다.

[예제]

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

OUTER는 생략 가능한 키워드이므로 아래 SQL은 같은 결과를 얻을 수 있다.

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

[실행 결과]

```
STADIUM_NAME   STADIUM_ID   SEAT_COUNT   HOMETEAM_ID   TEAM_NAME
------------ --------- ---------- ----------- ----------
울산문수경기장 C04 46102 K01 울산현대
수원월드컵경기장 B04 50000 K02 삼성블루윙즈
포항스틸야드 C06 25000 K03 스틸러스
인천월드컵경기장 B01 35000 K04 유나이티드
전주월드컵경기장 D03 28000 K05 현대모터스
부산아시아드경기장 C02 30000 K06 아이파크
광양전용경기장 D01 20009 K07 드래곤즈
성남종합운동장 B02 27000 K08 일화천마
서울월드컵경기장 B05 66806 K09 FC서울
대전월드컵경기장 D02 41000 K10 시티즌
창원종합운동장 C05 27085 K11 경남FC
광주월드컵경기장 A02 40245 K12 광주상무
강릉종합경기장 A03 33000 K13 강원FC
제주월드컵경기장 A04 42256 K14 제주유나이티드FC
대구월드컵경기장 A05 66422 K15 대구FC
안양경기장 F05 20000
```

```
마산경기장 F04 20000
일산경기장 F03 20000
부산시민경기장 F02 30000
대구시민경기장 F01 30000

20개의 행이 선택되었다.
```

INNER JOIN이라면 홈팀이 배정된 15개의 경기장만 출력 되었겠지만, LEFT OUTER JOIN을 사용하였기 때문에 홈팀이 없는 대구시민경기장, 부산시민경기장, 일산경기장, 마산경기장, 안양경기장의 정보까지 추가로 출력되었다.

나. RIGHT OUTER JOIN

조인 수행시 LEFT JOIN과 반대로 우측 테이블이 기준이 되어 결과를 생성한다. 즉, TABLE A와 B가 있을 때(TABLE 'B'가 기준이 됨), A와 B를 비교해서 A의 JOIN 칼럼에서 같은 값이 있을 때 그 해당 데이터를 가져오고, A의 JOIN 칼럼에서 같은 값이 없는 경우에는 A 테이블에서 가져오는 칼럼들은 NULL 값으로 채운다. 그리고 RIGHT JOIN으로 OUTER 키워드를 생략해서 사용할 수 있다.

[예제] DEPT에 등록된 부서 중에는 사원이 없는 부서도 있다. DEPT와 EMP를 조인하되 사원이 없는 부서 정보도 같이 출력하도록 한다.

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

OUTER는 생략 가능한 키워드이므로 아래 SQL은 같은 결과를 얻을 수 있다.

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

```
[실행 결과]
ENAME DEPTNO DNAME LOC
----- ------ ---------- --------
CLARK 10 ACCOUNTING NEW YORK
KING 10 ACCOUNTING NEW YORK
MILLER 10 ACCOUNTING NEW YORK
```

```
JONES 20 RESEARCH DALLAS
FORD 20 RESEARCH DALLAS
ADAMS 20 RESEARCH DALLAS
SMITH 20 RESEARCH DALLAS
SCOTT 20 RESEARCH DALLAS
WARD 30 SALES CHICAGO
TURNER 30 SALES CHICAGO
ALLEN 30 SALES CHICAGO
JAMES 30 SALES CHICAGO
BLAKE 30 SALES CHICAGO
MARTIN 30 SALES CHICAGO 40
OPERATIONS BOSTON
15개의 행이 선택되었다.
```

INNER JOIN이라면 사원 정보와 함께 사원이 배정된 3개의 부서 정보와 14명의 사원 정보만 출력 되었겠지만, RIGHT OUTER JOIN을 사용하였기 때문에 사원이 배정되지 않은 부서번호 40의 OPERATIONS 부서의 LOC 정보까지 출력되었다.

다. FULL OUTER JOIN

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

[예제] DEPT 테이블과 DEPT_TEMP 테이블의 FULL OUTER JOIN 사례를 만들기 위해 DEPT_TEMP의 DEPTNO를 수정한다. 결과적으로 DEPT_TEMP 테이블의 새로운 DEPTNO 데이터는 DETP 테이블의 DEPTNO와 2건은 동일하고 2건은 새로운 DEPTNO가 생성된다.

```
[예제]
UPDATE DEPT_TEMP
SET DEPTNO = DEPTNO + 20;

SELECT * FROM DEPT_TEMP;
```

```
[실행 결과]
DEPTNO DNAME LOC
```

```
------ ---------- ----------
30 ACCOUNTING NEW YORK
40 R&D DALLAS
50 MARKETING CHICAGO
60 OPERATIONS BOSTON
4개의 행이 선택되었다.
```

[예제] DEPTNO 기준으로 DEPT와 DEPT_TEMP 데이터를 FULL OUTER JOIN으로 출력한다. 예제에 사용된 UNION(중복 데이터는 제거됨)은 다음 절에서 설명하도록 한다.

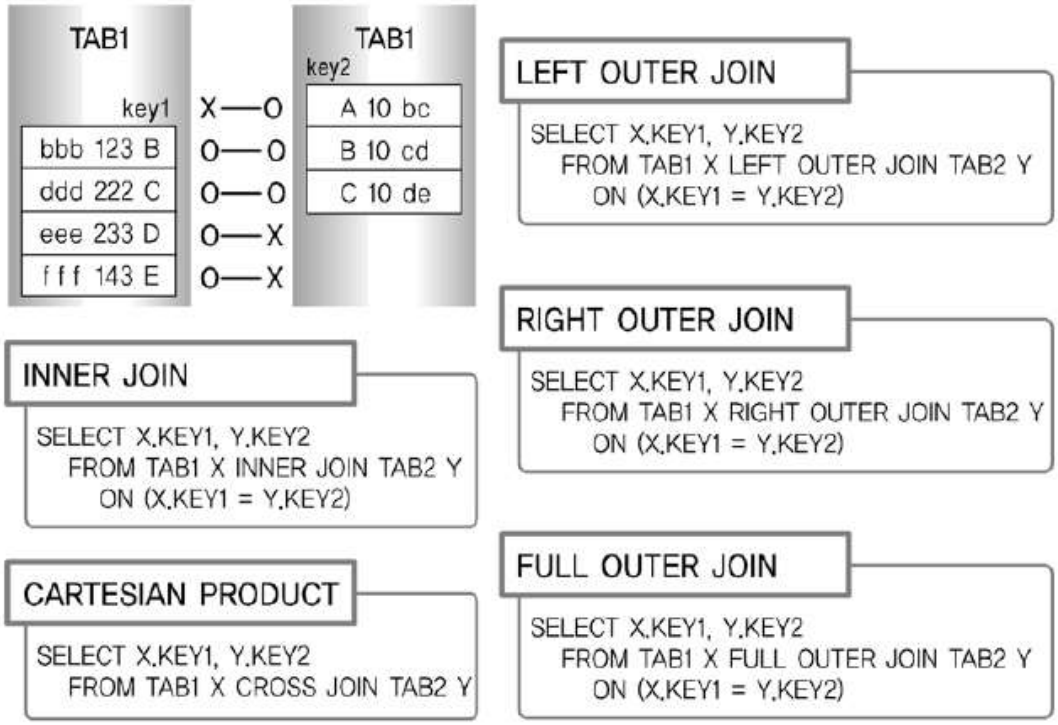
```
[예제]
SELECT *
FROM DEPT FULL OUTER JOIN DEPT_TEMP
ON DEPT.DEPTNO = DEPT_TEMP.DEPTNO;

OUTER는 생략 가능한 키워드이므로 아래 SQL은 같은 결과를 얻을 수 있다.

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

위 SQL과 아래 SQL은 같은 결과를 얻을 수 있다.

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

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

```
40 R&D DALLAS
50 MARKETING CHICAGO
60 OPERATIONS BOSTON
6개의 행이 선택되었다.
```

INNER JOIN이라면 부서번호가 동일한 30, 40 부서의 2개 정보만 출력되었겠지만, FULL OUTER JOIN을 사용하였기 때문에 DEPT 테이블에만 있는 부서번호 10, 20의부서와 DEPT_TEMP 테이블에만 있는 부서번호 50, 60의 부서 정보까지 같이 출력되었다.

9. INNER vs OUTER vs CROSS JOIN 비교



[그림 Ⅱ-2-4] INNER vs OUTER vs CROSS JOIN 문장 비교

첫 번째, INNER JOIN의 결과는 다음과 같다.

양쪽 테이블에 모두 존재하는 키 값이 B-B, C-C 인 2건이 출력된다.

두 번째, LEFT OUTER JOIN의 결과는 다음과 같다.

TAB1을 기준으로 키 값 조합이 B-B, C-C, D-NULL, E-NULL 인 4건이 출력된다.

세 번째, RIGHT OUTER JOIN의 결과는 다음과 같다.

TAB2를 기준으로 키 값 조합이 NULL-A, B-B, C-C 인 3건이 출력된다.

네 번째, FULL OUTER JOIN의 결과는 다음과 같다.

양쪽 테이블을 기준으로 키 값 조합이 NULL-A, B-B, C-C, D-NULL, E-NULL 인 5건이 출력된다.

다섯 번째, CROSS JOIN의 결과는 다음과 같다.

JOIN 가능한 모든 경우의 수를 표시하지만 단, OUTER JOIN은 제외한다.

양쪽 테이블 TAB1과 TAB2의 데이터를 곱한 개수인 4 * 3 = 12건이 추출됨

키 값 조합이 B-A, B-B, B-C, C-A, C-B, C-C, D-A, D-B, D-C, E-A, E-B, E-C 인 12건이 출력된다.

제2절 집합 연산자(SET OPERATOR)

두 개 이상의 테이블에서 조인을 사용하지 않고 연관된 데이터를 조회하는 방법 중에 또 다른 방법이 있는데 그 방법이 바로 집합 연산자(Set Operator)를 사용하는 방법이다. 기존의 조인에서는 FROM 절에 검색하고자 하는 테이블을 나열하고, WHERE 절에 조인 조건을 기술하여 원하는 데이터를 조회할 수 있었다.

하지만 집합 연산자는 여러 개의 질의의 결과를 연결하여 하나로 결합하는 방식을 사용한다. 즉, 집합 연산자는 2개 이상의 질의 결과를 하나의 결과로 만들어 준다.

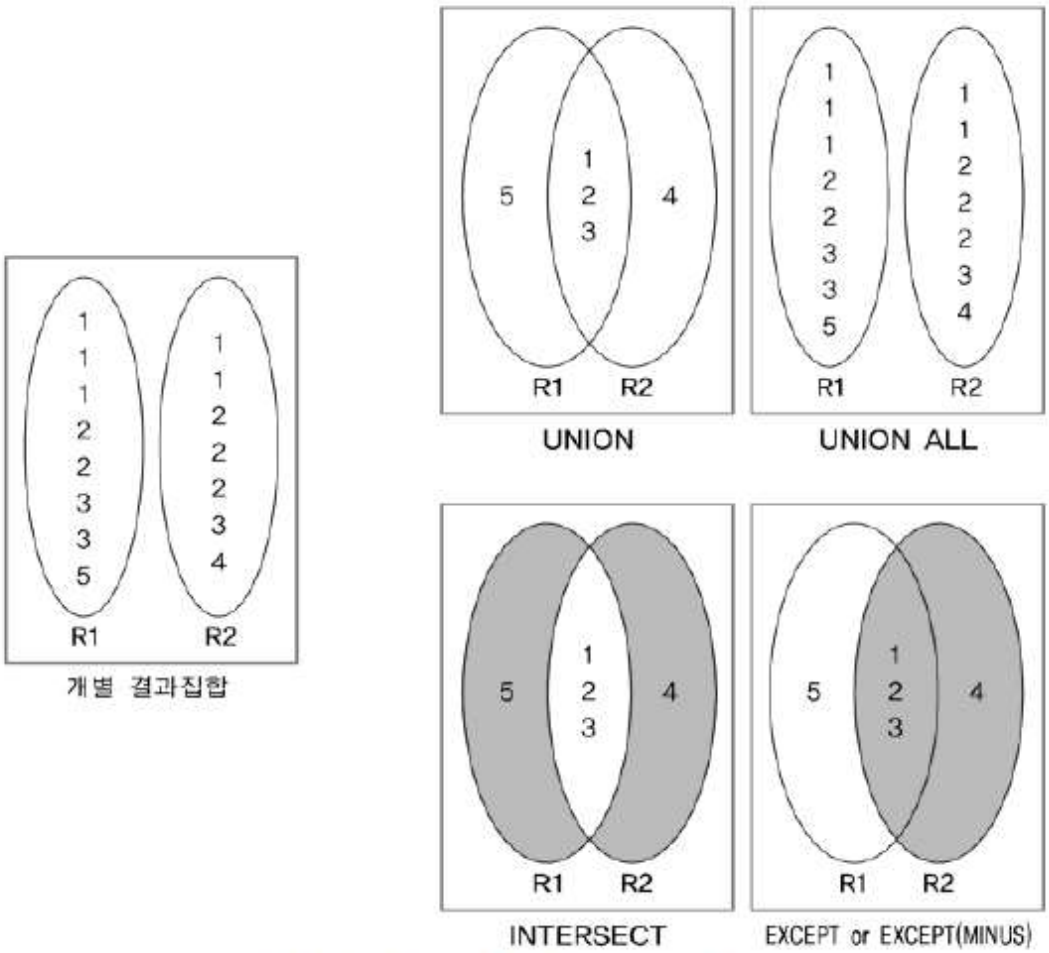
일반적으로 집합 연산자를 사용하는 상황은 서로 다른 테이블에서 유사한 형태의 결과를 반환하는 것을 하나의 결과로 합치고자 할 때와 동일 테이블에서 서로 다른 질의를 수행하여 결과를 합치고자 할 때 사용할 수 있다. 이외에도 튜닝관점에서 실행계획을 분리하고자 하는 목적으로도 사용할 수 있다.

집합 연산자를 사용하기 위해서는 다음 제약조건을 만족해야 한다. SELECT 절의 칼럼 수가 동일하고 SELECT 절의 동일 위치에 존재하는 칼럼의 데이터 타입이 상호 호환 가능(반드시 동일한 데이터 타입일 필요는 없음)해야 한다. 그렇지 않으면 데이터베이스가 오류를 반환한다.

[표 Ⅱ-2-1] 집합 연산자의 종류

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

집합 연산자는 개별 SQL문의 결과 집합에 대해 합집합(UNION/UNION ALL), 교집합(INTERSECT), 차집합(EXCEPT)으로 집합간의 관계를 가지고 작업을 한다.



[그림 Ⅱ-2-5] 집합 연산자의 연산

집합 연산자를 가지고 연산한 결과는 [그림 Ⅱ-2-5]와 같다. [그림 Ⅱ-2-5]의 왼쪽에 존재하는 R1, R2는 각각의 SQL문을 실행해서 생성된 개별 결과 집합을 의미한다.

[그림 Ⅱ-2-5]에서 보면 알 수 있듯이 UNION ALL을 제외한 다른 집합 연산자에서는 SQL문의 결과 집합에서 먼저 중복된 건을 배제하는 작업을 수행한 후에 집합 연산을 적용한다(논리적인 관점의 처리임).

UNION 연산에서 R1 = {1, 2, 3, 5}, R2 = {1, 2, 3, 4}가 되고, 이것의 합집합(R1 ∪ R2)의 결과는 {1, 2, 3, 4, 5}이다. UNION ALL 연산은 중복에 대한 배제 없이 2개의 결과 집합을 단순히 합친 것과 동일한 결과이다.

UNION ALL의 결과는 {1, 1, 1, 2, 2, 3, 3, 5, 1, 1, 2, 2, 2, 3, 4}이다. INTERSECT 연산에서 R1 = {1, 2, 3, 5}, R2 = {1, 2, 3, 4}가 되어, 이것의 교집합(R1 ∩ R2)의 결과는 {1, 2, 3}이다. EXCEPT 연산에서는 R1 = {1, 2, 3, 5}, R2 = {1, 2, 3, 4}가 되고, 이것의 차집합(R1 R2)의 결과는 {5}이다.

EXCEPT 연산에서는 순서가 중요하다. 만약 순서가 바뀌어서 R2 R1의 차집합이었다면 결과는 {4}가 된다.

집합 연산자를 사용하여 만들어지는 SQL문의 형태는 다음과 같다.

```
SELECT 칼럼명1, 칼럼명2, ...
FROM 테이블명1
[WHERE 조건식 ]
[[GROUP BY 칼럼(Column)이나 표현식
[HAVING 그룹조건식 ] ]
집합 연산자
SELECT 칼럼명1, 칼럼명2, ...
FROM 테이블명2
[WHERE 조건식 ]
[[GROUP BY 칼럼(Column)이나 표현식
[HAVING 그룹조건식 ] ]
[ORDER BY 1, 2 [ASC또는 DESC ] ;
-------------------------------------------------------------
SELECT PLAYER_NAME 선수명, BACK_NO 백넘버
FROM PLAYER
WHERE TEAM_ID = 'K02'
UNION
SELECT PLAYER_NAME 선수명, BACK_NO 백넘버
FROM PLAYER
WHERE TEAM_ID = 'K07' ORDER BY 1;
```

집합 연산자는 사용상의 제약조건을 만족한다면 어떤 형태의 SELECT문이라도 이용할 수 있다. 집합 연산자는 여러 개의 SELECT문을 연결하는 것에 지나지 않는다. ORDER BY는 집합 연산을 적용한 최종 결과에 대한 정렬 처리이므로 가장 마지막 줄에 한번만 기술한다.

아래 질문에 대해 집합 연산자를 사용하여 처리하는 방법을 알아보자.

[집합 연산자를 연습하기 위한 질문]

1) K-리그 소속 선수들 중에서 소속이 삼성블루윙즈팀인 선수들과전남드레곤즈팀인 선수들에 대한 내용을 모두보고 싶다.
2) K-리그 소속 선수들 중에서 소속이 삼성블루윙즈팀인 선수들과 포지션이 골키퍼(GK)인 선수들을 모두 보고 싶다.
3) K-리그 소속 선수들에 대한 정보 중에서 포지션별 평균키와 팀별 평균키를 알고 싶다.

4) K-리그 소속 선수를 중에서 소속이 삼성블루윙즈팀이면서 포지션이 미드필더(MF)가 아닌 선수들의 정보를 보고 싶다.
5) K-리그 소속 선수들 중에서 소속이 삼성블루윙즈팀이면서 포지션이 골키퍼(GK)인 선수들의 정보를 보고 싶다.

SQL문을 작성하기에 전에 [집합 연산자를 연습하기 위한 질문]을 다음과 같이 집합 연산자를 사용 형태로 해석할 수 있다.

[질문을 집합 연산의 개념으로 해석한 결과]

1) K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과K-리그 소속 선수 중 소속이 전남드레곤즈팀인 선수들의 집합의 합집합
2) K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과K-리그 소속 선수 중 포지션이 골키퍼(GK)인 선수들의 집합의 합집합
3) K-리그 소속 선수 중 포지션별 평균키에 대한 집합과K-리그 소속 선수 중 팀별 평균키에 대한 집합의 합집합
4) K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과K-리그 소속 선수 중 포지션이 미드필더(MF))인 선수들의 집합의 차집합
5) K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과K-리그 소속 선수 중 포지션이 골키퍼(GK)인 선수들의 집합의 교집합

위의 결과를 집합 연산자를 사용하여 SQL문을 작성하고 그 결과를 확인해 보도록 하자. 먼저 첫 번째 질문에 대한 SQL문을 작성하고 실행해 보자.

[질문1]

1) K-리그 소속 선수들 중에서 소속이 삼성블루윙즈팀인 선수들과전남드레곤즈팀인 선수들에 대한 내용을 모두보고 싶다.
1) K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과K-리그 소속 선수 중 소속이 전남드레곤즈팀인 선수들의 집합의 합집합

[예제]

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

```
FROM PLAYER
WHERE TEAM_ID = 'K07'
```

```
[실행 결과]
팀코드 선수명 포지션 백넘버 키
------------------------
K02  가비    MF     10  177
K02  강대희  MF     26  174
K02  고종수  MF     22  176
K02  고창현  MF      8  170
K02  김강진  DF     43  181
K07  강철    DF      3  178
K07  김반    MF     14  174
K07  김영수  MF     30  175
K07  김정래  GK     33  185
K07  김창원  DF      5  183
........
100개의 행이 선택되었다.
```

첫 번째 질문에 대한 SQL문에서 삼성 블루윙즈팀인 선수들과 전남 드레곤즈팀의 선수들의 합집합이라는 것은 WHERE 절에 IN 또는 OR 연산자로도 변환이 가능하다. 다만 IN 또는 OR 연산자를 사용할 경우에는 결과의 표시 순서가 달라질 수 있다. 집합이라는 관점에서는 결과가 표시되는 순서가 틀렸다고 두 집합이 서로 다르다고 말할 수 없다. 만약, 결과의 동일한 표시 순서를 원한다면 ORDER BY절을 사용해서 명시적으로 정렬 순서를 정의하는 것이 바람직하다.

첫 번째 질문에 대한 SQL문을 IN 또는 OR 연산자를 사용한 SQL문으로 변경해 보고 결과의 표시 순서가 다름을 확인해 보자.

```
[예제] (비교)
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버,
       HEIGHT 키
FROM PLAYER
WHERE TEAM_ID = 'K02' OR TEAM_ID = 'K07';
------------------------------------------------------------
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버,
```

```
        HEIGHT 키
FROM PLAYER
WHERE TEAM_ID IN ('K02', 'K07');
```

```
[실행 결과]
팀    선수명 포지션 백넘버 키
---- ----- ---- ----- ---
K07  김회택  TM
K07  서현옥  TC
K07  정상호  TC
K07  최철우  TC
K07  정영광  GK     41  185
K02  정호    TM
K02  왕선재  TC
K02  코샤    TC
K02  윤성효  TC
K02  정광수  GK     41  182
...........
100개의 행이 선택되다.
```

두 번째 질문에 대해 SQL문을 작성하고 결과를 확인해 보자.

[질문2]

2) K-리그 소속 선수들 중에서 소속이 삼성블루윙즈팀인 선수들과 포지션이 골키퍼(GK)인 선수들을 모두 보고 싶다.

2) K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과K-리그 소속 선수 중 포지션이 골키퍼(GK)인 선수들의 집합의 합집합

[예제]

```
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백
       넘버, HEIGHT 키
FROM PLAYER
WHERE TEAM_ID = 'K02'
UNION
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백
```

```
        넘버, HEIGHT 키
FROM PLAYER
WHERE POSITION = 'GK';
```

```
[실행 결과]
팀코드 선수명 포지션 백넘버 키
---- ----- ----- ---- --
K01 권정혁 GK 1 195
K01 서동명 GK 21 196
K01 양지원 GK 45 181
K01 이무림 GK 31 185
K01 최창주 GK 40 187
K02 가비   MF 10 177
K02 강대희 MF 26 174
K02 고종수 MF 22 176
K02 고창현 MF 8 170
K02 김강진 DF 43 181
...
88개의 행이 선택되었다.
```

두 번째 질문에 대한 실행 결과는 첫 번째 실행 결과와 비교해 보면 집합의 대상만 차이가 날뿐 다른 점은 없다. 마찬가지로 두 번째 질문에 대해 OR 연산자를 사용한 SQL문으로 변경하면 다음과 같다. 여기서는 서로 다른 칼럼에 조건을 사용했기 때문에 IN 연산자를 사용할 수 없다.

```
[예제] (비교)
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버,
        HEIGHT 키
FROM PLAYER
WHERE TEAM_ID = 'K02' OR POSITION = 'GK';
```

만약, 두 번째 질문에 대한 SQL문에서 UNION이라는 집합 연산자 대신에 UNION ALL이라는 집합 연산자를 사용하면 어떻게 될지 한번 수행해 보자.

```
[예제]
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버,
```

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

```
[실행 결과]
팀코드 선수명 포지션 백넘버 키
----  ----- ----  ---- ---
K02   정호    TM
K02   왕선재  TC
K02   코샤    TC
K02   윤성효  TC
K02   정광수  GK    41  182
K04   남현우  GK    31  180
K04   김충호  GK    60  185
K04   이현    GK     1  192
K04   한동진  GK    21  183
K10   강성일  GK    30  182
...
92개의 행이 선택되었다.
```

수행 결과에서 알 수 있듯이 결과 건수가 UNION은 88건이었으나 UNION ALL은 92건으로 결과 건수가 늘어났다. 두 SQL문의 결과가 서로 다르다. 결과가 다른 이유는 UNION은 결과에서 중복이 존재할 경우 중복을 제외시키지만 UNION ALL은 각각의 질의 결과를 단순히 결합시켜 줄 뿐 중복된 결과를 제외시키지 않기 때문이다. 이와 같이 결과 집합에 중복이 존재하면 UNION과 UNION ALL의 결과는 달라진다. UNION ALL에서 중복된 결과들을 확인해 보고자 할 때는 ORDER BY절을 사용하면 용이하다. 아래 SQL문을 통해 중복된 결과를 확인해 보자.

```
[예제]
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버,
        HEIGHT 키
FROM PLAYER
```

```
WHERE TEAM_ID = 'K02'
UNION ALL
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버,
HEIGHT 키
FROM PLAYER
WHERE POSITION = 'GK' ORDER BY 1, 2, 3, 4, 5;
```

```
[실행 결과]
팀코드 선수명 포지션 백넘버 키
----  ----  ----- ---- ---
K02   김운재  GK     1 182
K02   김운재  GK     1 182
K02   정광수  GK    41 182
K02   정광수  GK    41 182
K02   조범철  GK    21 185
K02   조범철  GK    21 185
K02   최호진  GK    31 190
K02   최호진  GK    31 190
...
92개의 행이 선택되었다.
```

결과에서 삼성블루윙즈팀(K02)에서 포지션이 골키퍼(GK)인 사람이 중복 표시되째 질문에 대한 SQL문을 작성하고 결과를 확인해 보자.

```
[질문3]
3) K-리그 소속 선수들에 대한 정보 중에서 포지션별 평균키와 팀별 평균키를 알고 싶다.
3) K-리그 소속 선수 중 포지션별 평균키에 대한 집합과K-리그 소속 선수 중 팀별 평균
   키에 대한 집합의 합집합
```

```
[예제]
SELECT 'P' 구분코드, POSITION 포지션, AVG(HEIGHT) 평균키
FROM PLAYER
GROUP BY POSITION
UNION
SELECT 'T' 구분코드, TEAM_ID 팀명, AVG(HEIGHT) 평균키
```

```
FROM PLAYER
GROUP BY TEAM_ID
ORDER BY 1;
```

```
[실행 결과]
구분코드 포지션 평균키
------ ----- --------
P        DF   180.409
P        FW    179.91
P        GK   186.256
P        MF   176.309
P        TC   178.833
T       K01   180.089
T       K02   179.067
T       K03   179.911
T       K04   180.511
T       K05   180.422
...
23개의 행이 선택되었다.
```

세 번째 질문에서는 평균키에 대한 값들의 합집합을 구하는 것이다. 합집합을 구하기 위해 SQL문에서 그룹함수를 사용했다. 그룹함수도 집합 연산자에서 사용이 가능하다는 것을 알 수 있다. 또한 실제로 테이블에는 존재하지 않지만 결과 행을 구분하기 위해 SELECT 절에 칼럼('구분코드')을 추가할 수 있다는 것을 알 수 있다. 이와 같이 목적을 위해 SELECT 절에 임의의 칼럼을 추가하는 것은 다른 모든 SQL문에서 적용 가능하다.

집합 연산자의 결과를 표시할 때 HEADING 부분은 첫 번째 SQL문에서 사용된 HEADING이 적용된다는 것을 알 수 있다. SQL문에서 첫 번째 SELECT 절에서는 '포지션' HEADING을 사용하였고 두 번째 SELECT 절에서는 '팀명' HEADING을 사용하였다. 그러나 결과에는 '포지션' HEADING으로 표시되었다.

네 번째 질문인 삼성 블루윙즈팀인 집합과 포지션이 미드필더(MF)인 선수들의 차집합에 대한 SQL문을 작성하고 결과를 확인해 보자.

```
[질문4]
4) K-리그 소속 선수를 중에서 소속이 삼성블루윙즈팀이면서 포지션이 미드필더(MF)가
   선수들의 정보를 보고 싶다.
```

4) K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과K-리그 소속 선수 중 포지션이 미드필더(MF))인 선수들의 집합의 차집합

```
[예제] Oracle
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백
       넘버, HEIGHT 키
FROM PLAYER
WHERE TEAM_ID = 'K02'
MINUS
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백
       넘버, HEIGHT 키
FROM PLAYER
WHERE POSITION = 'MF'
ORDER BY 1, 2, 3, 4, 5;
```

SQL Server에서는 MINUS대신 EXCEPT를 사용할 수 있다.

```
[실행 결과]
팀코드 선수명 포지션 백넘버 키
---- ------- ----- ---- --
K02   김강진   DF     43 181
K02   김관희   FW     39 180
K02   김만근   FW     34 177
K02   김병국   DF      2 183
K02   김병근   DF      3 175
K02   왕선재   TC
K02   윤성효   TC
K02   윤화평   FW     42 182
K02   이성용   DF     20 173
K02   정광수   GK     41 182
...
31개의 행이 선택되었다.
```

차집합은 앞의 집합의 결과에서 뒤의 집합의 결과를 빼는 것이다. 이번 SQL문은 삼성블루윙즈팀의 선수들 중에서 포지션이 미드필더(MF)인 선수들의 정보를 빼는 것이다. 해당

SQL문은 다른 형태의 SQL문으로 변경 가능하다. EXCEPT 연산자의 앞에 오는 SQL문의 조건은 만족하고 뒤에 오는 SQL문의 조건은 만족하지 않는 SQL문과 동일한 결과를 얻을 수 있다. 그러므로 EXCEPT 연산자를 사용하지 않고 논리 연산자를 이용하여 동일한 결과의 SQL문을 작성할 수 있다. SQL문을 실행하여 결과가 동일한지 직접 확인해 보자.

```
[예제]
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백
       넘버, HEIGHT 키
FROM PLAYER
WHERE TEAM_ID = 'K02' AND POSITION <>'MF'
ORDER BY 1, 2, 3, 4, 5;
```

MINUS 연산자는 NOT EXISTS 또는 NOT IN 서브쿼리를 이용한 SQL문으로도 변경 가능하다. (NOT EXISTS와 NOT IN에 대한 설명은 제5절 서브쿼리에서 참고)

```
[예제]
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백
       넘버, HEIGHT 키
FROM PLAYER X
WHERE X.TEAM_ID = 'K02'
AND NOT EXISTS (SELECT 1 FROM PLAYER Y WHERE Y.PLAYER_ID =
                X.PLAYER_ID AND POSITION = 'MF')
ORDER BY 1, 2, 3, 4, 5;
-----------------------------------------------------------------
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백
       넘버, HEIGHT 키
FROM PLAYER
WHERE TEAM_ID = 'K02'
AND PLAYER_ID NOT IN (SELECT PLAYER_ID FROM PLAYER WHERE POSITION
                      = 'MF')
ORDER BY 1, 2, 3, 4, 5;
```

이제 마지막으로 삼성블루윙즈팀이면서 포지션이 골키퍼인 선수들인 교집합을 얻기 위한 SQL문을 작성해 보자.

```
[질문 5]
5) K-리그 소속 선수들 중에서 소속이 삼성블루윙즈팀이면서 포지션이 골키퍼(GK)인
```

선수들의 정보를 보고 싶다.

5) K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과K-리그 소속 선수 중 포지션이 골키퍼(GK)인 선수들의 집합의 교집합

```
[예제]
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백
       넘버, HEIGHT 키
FROM PLAYER
WHERE TEAM_ID = 'K02'
INTERSECT
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백
       넘버, HEIGHT 키
FROM PLAYER WHERE POSITION = 'GK'
ORDER BY 1, 2, 3, 4, 5;
```

```
[실행 결과]
팀코드 선수명 포지션 백넘버 키
---- ------ ----- ---- --
K02   김운재   GK      1 182
K02   정광수   GK     41 182
K02   조범철   GK     21 185
K02   최호진   GK     31 190
...
4개의 행이 선택되었다.
```

교집합의 결과는 소속이 삼성 블루윙즈팀인 선수의 집합이면서 포지션이 골키퍼인 집합인 두 개의 조건을 만족하는 집합이다. 이것은 INTERSECT 연산자의 앞에 오는 SQL문의 조건은 만족하면서 뒤의 SQL문의 조건을 만족하는 것과 동일한 결과를 얻을 수 있다. 다음과 같이 INTERSECT 연산자를 사용하지 않고도 논리 연산자만으로 결과가 동일한 SQL문을 작성할 수 있다. SQL문을 실행하여 결과가 동일한지 직접 확인해 보자.

```
[예제]
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백
       넘버, HEIGHT 키
FROM PLAYER
WHERE TEAM_ID = 'K02' AND POSITION = 'GK'
ORDER BY 1, 2, 3, 4, 5;
```

INTERSECT 연산자는 EXISTS 또는 IN 서브쿼리를 이용한 SQL문으로 변경 가능하다. (EXISTS와 IN에 대한 설명은 제5절 서브쿼리에서 참고)

[예제]

```
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키
FROM PLAYER X
WHERE X.TEAM_ID = 'K02'
AND EXISTS (SELECT 1 FROM PLAYER Y WHERE Y.PLAYER_ID = X.PLAYER_ID
            AND Y.POSITION = 'GK')
ORDER BY 1, 2, 3, 4, 5;
```

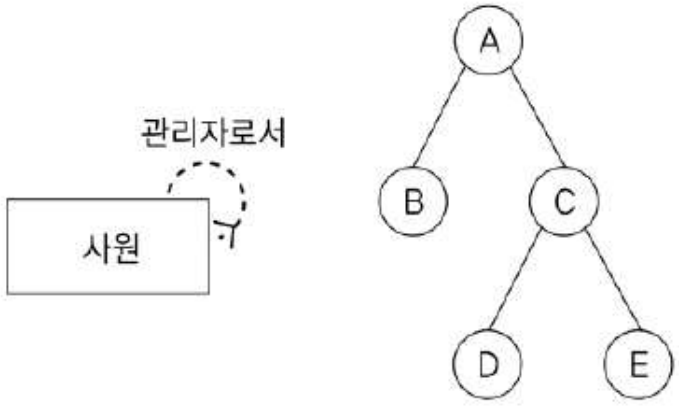
[예제]

```
SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키
FROM PLAYER
WHERE TEAM_ID = 'K02'
AND PLAYER_ID IN (SELECT PLAYER_ID FROM PLAYER WHERE POSITION =
                  'GK')
ORDER BY 1, 2, 3, 4, 5;
```

제3절 계층형 질의와 셀프 조인

1. 계층형 질의

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



사원	관리자
A	
B	A
C	A
D	C
E	C

(1) 순환관계 데이터 모델 (2) 계층형 구조 (3) 샘플 데이터

[그림 Ⅱ-2-6] 계층형 데이터

[그림 Ⅱ-2-6]은 사원에 대한 순환관계 데이터 모델을 표현한 것이다. (2)계층형 구조에서 A의 하위 사원은 B, C이고 B 밑에는 하위 사원이 없고 C의 하위 사원은 D, E가 있다. 계층형 구조를 데이터로 표현한 것이 (3)샘플 데이터이다. 계층형 데이터 조회는 DBMS 벤더와 버전에 따라 다른 방법으로 지원한다. 여기서는 Oracle과 SQL Server 기준으로 설명한다.

가. Oracle 계층형 질의

Oracle은 계층형 질의를 지원하기 위해서 [그림 Ⅱ-2-7]과 같은 계층형 질의 구문을 제공한다.

```
SELECT…
FROM 테이블
WHERE condition AND condition…
START WITH condition
CONNECT BY [NOCYCLE] condition AND condition…
[ORDER SIBLINGS BY column, column, …]
```

[그림 Ⅱ-2-7] 계층형 질의 구문

- START WITH절은 계층 구조 전개의 시작 위치를 지정하는 구문이다. 즉, 루트 데이터를 지정한다.(액세스) - CONNECT BY절은 다음에 전개될 자식 데이터를 지정하는 구문이다. 자식 데이터는 CONNECT BY절에 주어진 조건을 만족해야 한다.(조인) - PRIOR : CONNECT BY절에 사용되며, 현재 읽은 칼럼을 지정한다. PRIOR 자식 = 부모 형태를 사용하면 계층구조에서 자식 데이터에서 부모 데이터(자식 → 부모) 방향으로 전개하는 순방향 전개를 한다. 그리고 PRIOR 부모 = 자식 형태를 사용하면 반대로 부모 데이터에서 자식 데이터(부모 → 자식) 방향으로 전개하는 역방향 전개를 한다. - NOCYCLE : 데이터를 전개하면서 이미 나타났던 동일한 데이터가 전개 중에 다시 나타난다면 이것을 가리켜 사이클(Cycle)이 형성되었다라고 말한다. 사이클이 발생한 데이터는 런타임 오류가 발생한다. 그렇지만 NOCYCLE를 추가하면 사이클이 발생한 이후의 데이터는 전개하지 않는다. - ORDER SIBLINGS BY : 형제 노드(동일 LEVEL) 사이에서 정렬을 수행한다. - WHERE : 모든 전개를 수행한 후에 지정된 조건을 만족하는 데이터만 추출한다.(필터링)

Oracle은 계층형 질의를 사용할 때 다음과 같은 가상 칼럼(Pseudo Column)을 제공한다.

[표 Ⅱ-2-2] 계층형 질의에서 사용되는 가상 칼럼

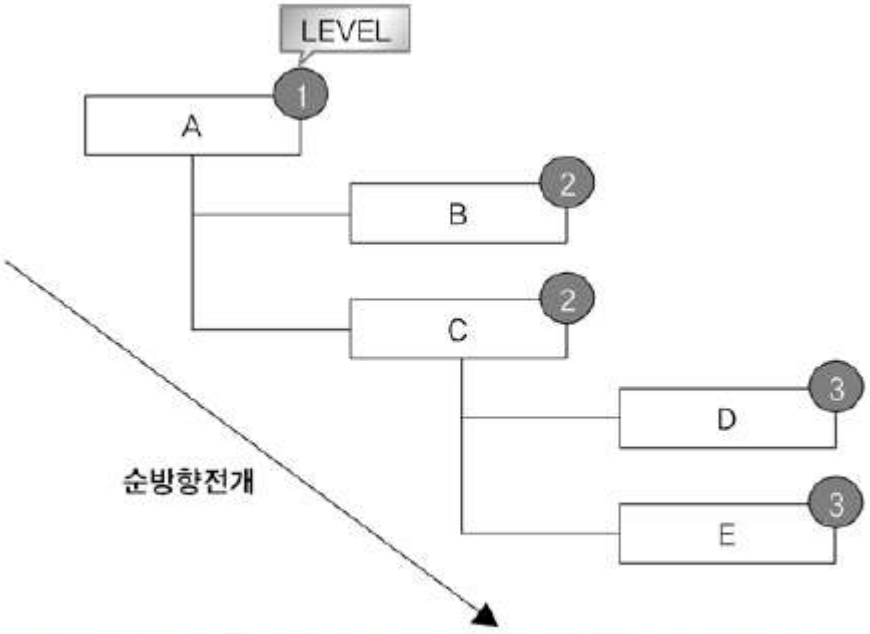
가상 칼럼	설명
LEVEL	루트 데이터이면 1, 그 하위 데이터이면 2이다. 리프(Leaf) 데이터까지 1씩 증가한다.
CONNECT_BY_ISLEAF	전개 과정에서 해당 데이터가 리프 데이터이면 1, 그렇지 않으면 0이다.
CONNECT_BY_ISCYCLE	전개 과정에서 자식을 갖는데, 해당 데이터가 조상으로서 존재하면 1, 그렇지 않으면 0이다. 여기서 조상이란 자신으로부터 루트까지의 경로에 존재하는 데이터를 말한다. CYCLE 옵션을 사용했을 때만 사용할 수 있다.

다음은 [그림 Ⅱ-2-6]의 (3)샘플 데이터를 계층형 질의 구문을 이용해서 조회한 것이다. 여기서는 결과 데이터를 들여쓰기 하기 위해서 LPAD 함수를 사용하였다.

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

```
[실행 결과]
 LEVEL     사원     관리자 ISLEAF
 ----- -------- ----- ------
 1         A                0
 2           B      A       1
 2           C      A       0
 3             D    C       1
 3             E    C       1
```

A는 루트 데이터이기 때문에 레벨이 1이다. A의 하위 데이터인 B, C는 레벨이 2이다. 그리고 C의 하위 데이터인 D, E는 레벨이 3이다. 리프 데이터는 B, D, E이다. 관리자 → 사원 방향을 전개이기 때문에 순방향 전개이다. [그림 Ⅱ-2-8]은 계층형 질의에 대한 논리적인 실행 모습이다.



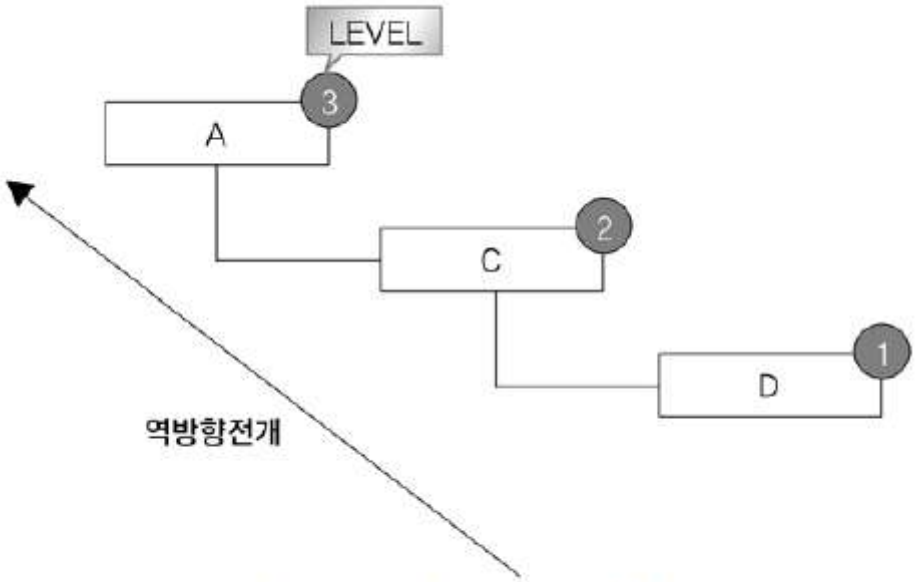
[그림 Ⅱ-2-8] 순방향 계층형 질의 결과의 논리적인 모습

다음 예제는 사원 'D'로부터 자신의 상위관리자를 찾는 역방향 전개의 예이다.

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

```
[실행 결과]
LEVEL      사원      관리자 ISLEAF
----- --------- ----- -----
1        D            C       0
2          C          A       0
3             A             1
```

본 예제는 역방향 전개이기 때문에 하위 데이터에서 상위 데이터로 전개된다. 결과를 보면 내용을 제외하고 표시 형태는 순방향 전개와 동일하다. D는 루트 데이터이기 때문에 레벨이 1이다. D의 상위 데이터인 C는 레벨이 2이다. 그리고 C의 상위 데이터인 A는 레벨이 3이다. 리프 데이터는 A이다. 루트 및 레벨은 전개되는 방향에 따라 반대가 됨을 알 수 있다. [그림 Ⅱ-2-9]는 역방향 전개에 대한 계층형 질의에 대한 논리적인 실행 모습이다.



[그림 Ⅱ-2-9] 역방향 계층형 질의 결과의 논리적인 모습

Orcle은 계층형 질의를 사용할 때 사용자 편의성을 제공하기 위해서 [표 Ⅱ-2-3]과 같은 함수를 제공한다

[표 Ⅱ-2-3] 계층형 질의에서 사용되는 함수

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

SYS_CONNECT_BY_PATH, CONNECT_BY_ROOT를 사용한 예는 다음과 같다.

```
[예제]
SELECT CONNECT_BY_ROOT 사원 루트사원,
SYS_CONNECT_BY_PATH(사원, '/') 경로, 사원, 관리자
FROM 사원
START WITH 관리자 IS NULL
CONNECT BY PRIOR 사원 = 관리자
```

```
[실행 결과]
루트사원    경로     사원 관리자
------- ------- ---- -----
A       /A      A
A       /A/B    B    A
A       /A/C    C    A
A       /A/C/D  D    C
A       /A/C/E  E    C
```

START WITH를 통해 추출된 루트 데이터가 1건 이기 때문에 루트사원은 모두 A이다. 경로는 루트로부터 현재 데이터까지의 경로를 표시한다. 예를 들어, D의 경로는 A → C → D 이다.

나. SQL Server 계층형 질의

SQL Server 2000 버전까지는 계층형 질의를 작성할 수 있는 문법을 지원하지 않았다. 조직도처럼 계층적 구조를 가진 데이터는 저장 프로시저를 재귀 호출하거나 While 루프 문에서 임시 테이블을 사용하는 등 (순수한 쿼리가 아닌) 프로그램 방식으로 전개해야만 했다. 그러나 SQL Server 2005 버전부터는 하나의 질의로 원하는 결과를 얻을 수 있게 되었다.

먼저, Northwind 데이터베이스에 접속하여 Employees 테이블의 데이터를 조회해 보자.

```
USE NORTHWIND
GO
SELECT EMPLOYEEID, LASTNAME, FIRSTNAME, REPORTSTO
FROM EMPLOYEES
GO
**********************************************************************
EmployeeID LastName   FirstName   ReportsTo
--------- --------    -------   --------
         1  Davolio      Nancy          2
         2  Fulle       Andrew       NULL
         3  Leverling    Janet          2
         4  Peacock     Margaret        2
         5  Buchanan    Steven          2
         6  Suyama      Michael         5
         7  King        Robert          5
         8  Callahan     Laura          2
         9  Dodsworth    Anne          5
(9개 행 적용됨)
```

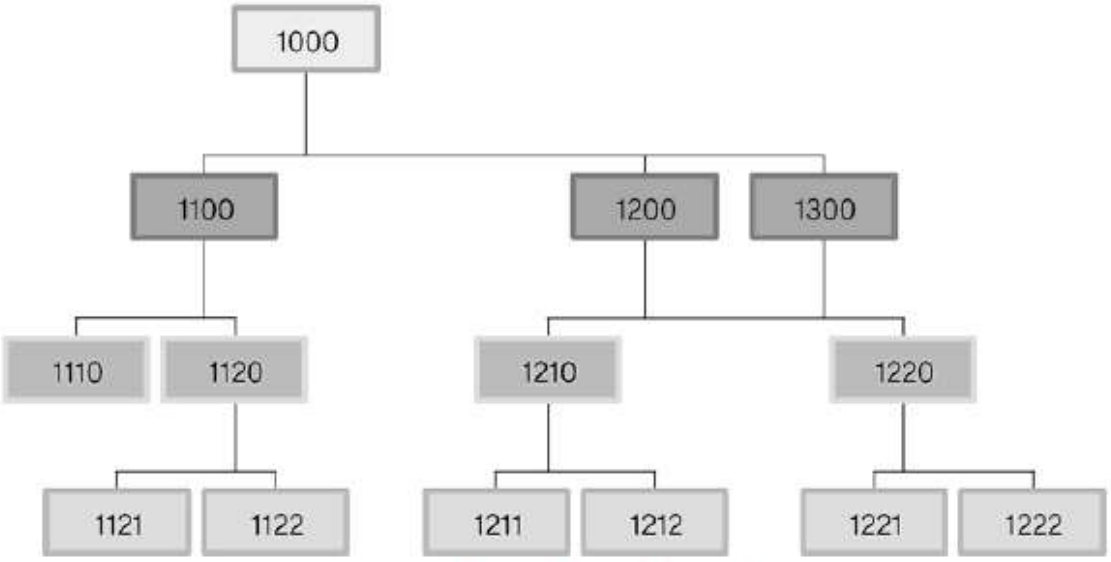
총 9개 로우가 있는데, ReportsTo 칼럼이 상위 사원에 해당하며 EmployeeID 칼럼과 재귀적 관계를 맺고 있다. EmployeeID가 2인 Fuller 사원을 살펴보면, ReportsTo 칼럼 값이 NULL이므로 계층 구조의 최상위에 있음을 알 수 있다. CTE(Common Table Expression)를 재귀 호출함으로써 Employees 데이터의 최상위부터 시작해 하위 방향으로 계층 구조를 전개하도록 작성한 쿼리와 결과는 다음과 같다.

```
WITH EMPLOYEES_ANCHOR AS (
    SELECT EMPLOYEEID, LASTNAME, FIRSTNAME, REPORTSTO, 0 AS LEVEL
    FROM EMPLOYEES
    WHERE REPORTSTO IS NULL /* 재귀 호출의 시작점 */
    UNION ALL
    SELECT R.EMPLOYEEID, R.LASTNAME, R.FIRSTNAME,
    R.REPORTSTO, A.LEVEL + 1
    FROM EMPLOYEES_ANCHOR A, EMPLOYEES R
    WHERE A.EMPLOYEEID = R.REPORTSTO )
SELECT LEVEL, EMPLOYEEID, LASTNAME, FIRSTNAME, REPORTSTO
FROM EMPLOYEES_ANCHOR
GO
************************************************************************
Level EmployeeID LastName    FirstName    ReportsTo
----  --------   -------     -----        --------
 0       2       Fuller      Andrew       NULL
 1       1       Davolio     Nancy        2
 1       3       Leverling   Janet        2
 1       4       Peacock     Margaret     2
 1       5       Buchanan    Steven       2
 1       8       Callahan    Laura        2
 2       6       Suyama      Michael      5
 2       7       King        Robert       5
 2       9       Dodsworth   Anne         5
(9개 행 적용됨)
```

WITH 절의 CTE 쿼리를 보면, UNION ALL 연산자로 쿼리 두 개를 결합했다. 둘 중 위에 있는 쿼리를 '앵커 멤버'(Anchor Member)라고 하고, 아래에 있는 쿼리를 '재귀 멤버'(Recursive Member)라고 한다. 아래는 재귀적 쿼리의 처리 과정이다.

1. CTE 식을 앵커 멤버와 재귀 멤버로 분할한다. 2. 앵커 멤버를 실행하여 첫 번째 호출 또는 기본 결과 집합(T0)을 만든다. 3. Ti는 입력으로 사용하고 Ti+1은 출력으로 사용하여 재귀 멤버를 실행한다. 4. 빈 집합이 반환될 때까지 3단계를 반복한다. 5. 결과 집합을 반환한다. 이것은 T0에서 Tn까지의 UNION ALL이다.

정리하자면 다음과 같다. 먼저, 앵커 멤버가 시작점이자 Outer 집합이 되어 Inner 집합인 재귀 멤버와 조인을 시작한다. 이어서, 앞서 조인한 결과가 다시 Outer 집합이 되어 재귀 멤버와 조인을 반복하다가 조인 결과가 비어 있으면 즉, 더 조인할 수 없으면 지금까지 만들어진 결과 집합을 모두 합하여 리턴한다. [그림 Ⅱ-2-10]에 있는 조직도를 쿼리로 출력했을 때, 대부분 사용자는 아래와 같은 결과를 기대할 것이다.(보기 편하도록 각 로우 앞쪽에 자신의 레벨만큼 빈칸을 삽입했다.)



[그림 Ⅱ-2-10] 조직도 예제

```
EmployeeID   ManagerID
--------     ---------
1000          NULL
 1100         1000
  1110        1100
  1120        1100
   1121       1120
   1122       1120
1200          1000
 1210         1200
  1211        1210
  1212        1210
1220          1200
 1221         1220
 1222         1220
1300          1000
```

아래에 t_emp 데이터의 최상위부터 시작해 하위 방향으로 계층 구조를 전개하도록 작성한 쿼리와 그 결과이다.

```
WITH T_EMP_ANCHOR AS (
        SELECT EMPLOYEEID, MANAGERID, 0 AS LEVEL
        FROM T_EMP
        WHERE MANAGERID IS NULL /* 재귀 호출의 시작점 */
        UNION ALL
        SELECT R.EMPLOYEEID, R.MANAGERID, A.LEVEL + 1
        FROM T_EMP_ANCHOR A, T_EMP R
        WHERE A.EMPLOYEEID = R.MANAGERID )
SELECT LEVEL,
REPLICATE(' ', LEVEL) + EMPLOYEEID AS EMPLOYEEID, MANAGERID
FROM T_EMP_ANCHOR
GO
**********************************************************************
  Level EmployeeID ManagerID
   ---   -------  ---------
    0     1000       NULL
    1      1100      1000
    1      1200      1000
    1      1300      1000
    2       1210     1200
    2       1220     1200
    3        1221    1220
    3        1222    1220
    3        1211    1210
    3        1212    1210
    2      1110      1100
    2      1120      1100
    3       1121     1120
    3       1122     1120
(14개 행 적용됨)
```

보다시피, 계층 구조를 단순히 하위 방향으로 전개했을 뿐 [그림 Ⅱ-2-10]에 있는 조직

도와는 많이 다른 모습이다. 앞서 보았듯이, CTE 재귀 호출로 만들어낸 계층 구조는 실제와 다른 모습으로 출력된다. 따라서 조직도와 같은 모습으로 출력하려면 order by 절을 추가해 원하는 순서대로 결과를 정렬해야 한다. 실제 조직도와 같은 모습의 결과를 출력하도록, CTE에 Sort라는 정렬용 칼럼을 추가하고 쿼리 마지막에 order by 조건을 추가해보자. (단, 앵커 멤버와 재귀 멤버 양쪽에서 convert 함수 등으로 데이터 형식을 일치시켜야 한다.)

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

CTE 안에서 Sort 칼럼에 사번(=EmployeeID)을 재귀적으로 더해 나가면 정렬 기준으로 삼을 수 있는 값이 만들어진다. 아래는 Sort 칼럼으로 정렬하여 출력한 결과인데, [그림 Ⅱ-2-10]에 있는 조직도의 모습과 일치한다.

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

가상의 Sort 칼럼을 추가해 정렬하는 게 아쉽기는 하지만, SQL Server에서 계층 구조를 실제 모습대로 출력하려면 현재(2005, 2008 버전 기준)로서는 감수해야 할 수밖에 없다.

2. 셀프 조인

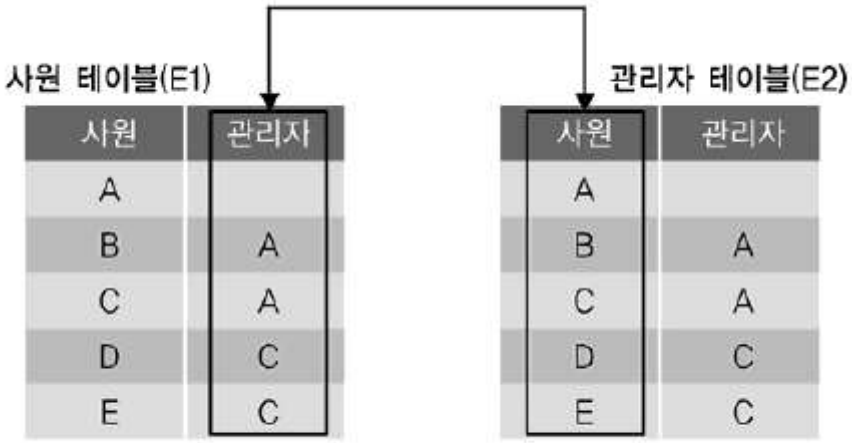
셀프 조인(Self Join)이란 동일 테이블 사이의 조인을 말한다. 따라서 FROM 절에 동일 테이블이 두 번 이상 나타난다. 동일 테이블 사이의 조인을 수행하면 테이블과 칼럼 이름이 모두 동일하기 때문에 식별을 위해 반드시 테이블 별칭(Alias)를 사용해야 한다. 그리고 칼럼에도 모두 테이블 별칭을 사용해서 어느 테이블의 칼럼인지 식별해줘야 한다. 이외 사항은 조인과 동일하다.

셀프 조인에 대한 기본적인 사용법은 다음과 같다

```
SELECT ALIAS명1.칼럼명, ALIAS명2.칼럼명, ...
FROM 테이블1 ALIAS명1, 테이블2 ALIAS명2
WHERE ALIAS명1.칼럼명2 = ALIAS명2.칼럼명1;
```

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

계층형 질의에서 살펴보았던 사원이라는 테이블 속에는 사원과 관리자가 모두 하나의 사원이라는 개념으로 동일시하여 같이 입력되어 있다. 이것을 이용해서 다음 문제를 셀프 조인으로 해결해 보면 다음과 같다. "자신과 상위, 차상위 관리자를 같은 줄에 표시하라." 이 문제를 해결하기 위해서는 FROM 절에 사원 테이블을 두 번 사용해야 한다.



[그림 Ⅱ-2-11] 셀프 조인 개념

셀프 조인은 동일한 테이블(사원)이지만 [그림 Ⅱ-2-11]과 같이 개념적으로는 두 개의 서로 다른 테이블(사원, 관리자)을 사용하는 것과 동일하다. 동일 테이블을 다른 테이블인 것처럼 처리하기 위해 테이블 별칭을 사용한다. 여기서는 E1(사원), E2(관리자) 테이블 별칭을 사용하였다. 차상위 관리자를 구하기 위해서 E1.관리자 = E2.사원 조인 조건을 사용한다. 셀프 조인을 이용한 SQL문은 다음과 같다.

```
[예제]
SELECT E1.사원, E1.관리자, E2.관리자 차상위_관리자
FROM 사원 E1, 사원 E2
WHERE E1.관리자 = E2.사원
ORDER BY E1.사원;
```

```
[실행 결과]
사원  관리자  차상위_관리자
---- ------ ----------
 B    A
 C    A
 D    C        A
 E    C        A
```

자신과 자신의 직속 관리자는 동일한 행에서 데이터를 구할 수 있으나 차상위 관리자는 바로 구할 수 없다. 차상위 관리자를 구하기 위해서는 자신의 직속 관리자를 기준으로 사원 테이블과 한번 더 조인(셀프 조인)을 수행해야 한다. 결과 표시를 위해 SELECT절에 2개의 '관리자' 칼럼을 사용되었다. 한 명은 자신의 직속 관리자(E1.관리자)이고 다른 한 명은 자신의 차상위 관리자(E2.관리자)이다. 결과를 보면, B와 C의 관리자는 A이고 차상위 관리자는 없다. D와 E의 관리자는 C이고 차상위 관리자는 A이다. 결과에서 A에 대한 정보는 누락되었다. 내부 조인(Inner Join)을 사용할 경우 자신의 관리자가 존재하지 않는 경우에는 관리자(E2) 테이블에서 조인할 대상이 존재하지 않기 때문에 해당 데이터는 결과에서 누락된다. 이를 방지하기 위해서는 아우터 조인을 사용해야 한다. 다음은 아우터 조인을 사용한 예이다.

```
[예제]
SELECT E1.사원, E1.관리자, E2.관리자 차상위_관리자
FROM 사원 E1 LEFT OUTER JOIN 사원 E2
ON (E1.관리자 = E2.사원)
ORDER BY E1.사원;
```

```
[실행 결과]
사원  관리자 차상위_관리자
---- ----- ----------
 A
 B     A
 C     A
 D     C      A
 E     C      A
```

아우터 조인을 사용해서 관리자가 존재하지 않는 데이터까지 모두 결과에 표시되었다.

제4절 서브쿼리

서브쿼리(Subquery)란 하나의 SQL문안에 포함되어 있는 또 다른 SQL문을 말한다. 서브쿼리는 알려지지 않은 기준을 이용한 검색을 위해 사용한다. 서브쿼리는 [그림 Ⅱ-2-12]와 같이 메인쿼리가 서브쿼리를 포함하는 종속적인 관계이다.

[그림 Ⅱ-2-12] 메인쿼리와 서브쿼리

조인은 조인에 참여하는 모든 테이블이 대등한 관계에 있기 때문에 조인에 참여하는 모든 테이블의 칼럼을 어느 위치에서라도 자유롭게 사용할 수 있다. 그러나 서브쿼리는 메인쿼리의 칼럼을 모두 사용할 수 있지만 메인쿼리는 서브쿼리의 칼럼을 사용할 수 없다.

질의 결과에 서브쿼리 칼럼을 표시해야 한다면 조인 방식으로 변환하거나 함수, 스칼라 서브쿼리(Scalar Subquery) 등을 사용해야 한다. 조인은 집합간의 곱(Product)의 관계이다. 즉, 1:1 관계의 테이블이 조인하면 1(= 1 * 1) 레벨의 집합이 생성되고, 1:M 관계의 테이블을 조인하면 M(= 1 * M) 레벨의 집합이 생성된다. 그리고 M:N 관계의 테이블을 조인하면 MN(= M * N) 레벨의 집합이 결과로서 생성된다. 예를 들어, 조직(1)과 사원(M) 테이블을 조인하면 결과는 사원 레벨(M)의 집합이 생성된다. 그러나 서브쿼리는 서브쿼리 레벨과는 상관없이 항상 메인쿼리 레벨로 결과 집합이 생성된다. 예를 들어, 메인쿼리로 조직(1), 서브쿼리로 사원(M) 테이블을 사용하면 결과 집합은 조직(1) 레벨이 된다.

SQL문에서 서브쿼리 방식을 사용해야 할 때 잘못 판단하여 조인 방식을 사용하는 경우가 있다. 예를 들어, 결과는 조직 레벨이고 사원 테이블에서 체크해야 할 조건이 존재한다고 가정하자. 이런 상황에서 SQL문을 작성할 때 조인을 사용한다면 결과 집합은 사원(M) 레벨이 될 것이다. 이렇게 되면 원하는 결과가 아니기 때문에 SQL문에 DISTINCT를 추가해서 결과를 다시 조직(1) 레벨로 만든다. 이와 같은 상황에서는 조인 방식이 아니라 서브쿼리 방식을 사용해야 한다. 메인쿼리로 조직을 사용하고 서브쿼리로 사원 테이블을 사용하면 결과 집합은 조직 레벨이 되기 때문에 원하는 결과가 된다.

서브쿼리를 사용할 때 다음 사항에 주의해야 한다.

① 서브쿼리를 괄호로 감싸서 사용한다.

② 서브쿼리는 단일 행(Single Row) 또는 복수 행(Multiple Row) 비교 연산자와 함께 사용 가능하다. 단일 행 비교 연산자는 서브쿼리의 결과가 반드시 1건 이하이어야 하고 복수 행 비교 연산자는 서브쿼리의 결과 건수와 상관 없다.

③ 서브쿼리에서는 ORDER BY를 사용하지 못한다. ORDER BY절은 SELECT절에서 오직 한 개만 올 수 있기 때문에 ORDER BY절은 메인쿼리의 마지막 문장에 위치해야 한다.

서브쿼리가 SQL문에서 사용이 가능한 곳은 다음과 같다.

- SELECT 절 - FROM 절 - WHERE 절 - HAVING 절 - ORDER BY 절
- INSERT문의 VALUES 절 - UPDATE문의 SET 절

서브쿼리의 종류는 동작하는 방식이나 반환되는 데이터의 형태에 따라 분류할 수 있다. 동작하는 방식에 따라 서브쿼리를 분류하면 [표 Ⅱ-2-4]와 같이 두 가지로 나눌 수 있다.

[표 Ⅱ-2-4] 동작하는 방식에 따른 서브쿼리 분류

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

서브쿼리는 메인쿼리 안에 포함된 종속적인 관계이기 때문에 논리적인 실행순서는 항상 메인쿼리에서 읽혀진 데이터에 대해 서브쿼리에서 해당 조건이 만족하지를 확인하는 방식으로 수행되어야 한다. 그러나 실제 서브쿼리의 실행순서는 상황에 따라 달라질 수 있다. 반환되는 데이터의 형태에 따라 서브쿼리는 [표 Ⅱ-2-5]와 같이 세가지로 분류된다.

[표 Ⅱ-2-5] 반환되는 데이터의 형태에 따른 서브쿼리 분류

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

1. 단일 행 서브 쿼리

서브쿼리가 단일 행 비교 연산자(=, <, <=, >, >=, <>)와 함께 사용할 때는 서브쿼리의 결과 건수가 반드시 1건 이하이어야 한다. 만약, 서브쿼리의 결과 건수가 2건 이상을 반환하면 SQL문은 실행시간(Run Time) 오류가 발생한다. 이런 종류의 오류는 컴파일 할 때(Compile Time)는 알 수 없는 오류이다. 단일 행 서브쿼리의 예로 '정남일' 선수가 소속된 팀의 선수들에 대한 정보를 표시하는 문제를 가지고 설명해 보면 다음과 같다.

[그림 Ⅱ-2-13] 단일 행 서브쿼리의 예제1

[그림 Ⅱ-2-13]은 2개의 SQL문으로 구성되어 있다. 정남일 선수의 소속팀을 알아내는 SQL문(서브쿼리 부분)과 이 결과를 이용해서 해당 팀에 소속된 선수들의 정보를 출력하는 SQL문(메인쿼리 부분)으로 구성된다. 정남일 선수가 소속된 팀의 선수들에 대한 정보를 표시하는 문제를 서브쿼리 방식의 SQL문으로 작성하면 다음과 같다.

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

```
[실행 결과] 선수명 포지션 백넘버
------- ----- -----
강철 DF 3
김반 MF 14
김영수 MF 30
김정래 GK 33
김창원 DF 5
김회택 TM
꼬레아 FW 16
노병준 MF 22
51개의 행이 선택되었다.
```

정남일 선수의 소속팀을 알아내는 서브쿼리가 먼저 수행되어 정남일 선수의 소속팀 코드가 반환된다.

메인쿼리는 서브쿼리에서 반환된 결과를 이용해서 조건을 만족하는 선수들의 정보를 출력한다. 만약, 정남일 선수가 동명이인이었다면 2건 이상의 결과가 반환되어 SQL문은 오류가 발생될 것이다. 테이블 전체에 하나의 그룹함수를 적용할 때는 그 결과값이 1건이 생성되기 때문에 단일 행 서브쿼리로서 사용 가능하다. 선수들 중에서 키가 평균 이하인 선수들의 정보를 출력하는 문제를 가지고 그룹함수를 사용한 서브쿼리를 알아보도록 한다.

[그림 Ⅱ-2-14] 단일 행 서브쿼리의 예제2

[그림 Ⅱ-2-14]는 2개의 SQL문으로 구성되어 있다. 선수들의 평균키를 알아내는 SQL문(서브쿼리 부분)과 이 결과를 이용해서 키가 평균 이하의 선수들의 정보를 출력하는 SQL문(메인쿼리 부분)으로 구성된다. [그림 Ⅱ-2-14]를 SQL문으로 작성하면 다음과 같다.

```
[예제] SELECT PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버
       FROM PLAYER
       WHERE HEIGHT <= (SELECT AVG(HEIGHT)
                        FROM PLAYER)
       ORDER BY PLAYER_NAME;
[실행 결과] 선수명 포지션 백넘버
           ------- ------ -----
           가비 MF 10
           강대희 MF 26
           강용 DF 2
           강정훈 MF 38
           강철 DF 3
           고규억 DF 29
           고민기 FW 24
           고종수 MF 22
           228개의 행이 선택되었다.
```

12. 다중 행 서브쿼리

서브쿼리의 결과가 2건 이상 반환될 수 있다면 반드시 다중 행 비교 연산자(IN, ALL, ANY, SOME)와 함께 사용해야 한다. 그렇지 않으면 SQL문은 오류를 반환한다. 다중 행 비교 연산자는 다음과 같다.

[표 Ⅱ-2-6] 다중 행 비교 연산자

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

선수들 중에서 '정현수'라는 선수가 소속되어 있는 팀 정보를 출력하는 서브쿼리를 작성하면 다음과 같다.

```
[예제] SELECT REGION_NAME 연고지명, TEAM_NAME 팀명, E_TEAM_NAME 영문팀명
       FROM TEAM
       WHERE TEAM_ID = (SELECT TEAM_ID
                        FROM PLAYER
                        WHERE PLAYER_NAME = '정현수')
       ORDER BY TEAM_NAME;
ORA-01427: 단일 행 하위 질의에 2개 이상의 행이 리턴되었다.
```

위의 SQL문은 서브쿼리의 결과로 2개 이상의 행이 반환되어 단일 행 비교 연산자인 '='로는 처리가 불가능하기 때문에 에러가 반환되었다. 따라서 다중 행 비교 연산자로 바꾸어서 SQL문을 작성하면 다음과 같다.

```
[예제] SELECT REGION_NAME 연고지명, TEAM_NAME 팀명, E_TEAM_NAME 영문팀명
       FROM TEAM
       WHERE TEAM_ID IN (SELECT TEAM_ID
                         FROM PLAYER
                         WHERE PLAYER_NAME = '정현수')
       ORDER BY TEAM_NAME;
[실행 결과] 연고지명 팀명 영문팀명
         ------ ----- ------------------------
         전남 드래곤즈 CHUNNAM DRAGONS
         FC 성남 일화천마 SEONGNAM ILHWA CHUNMA FC
         2개의 행이 선택되었다.
```

실행 결과를 보면 '정현수'란 이름을 가진 선수가 두 명이 존재한다. 소속팀은 각각 전남 드래곤즈팀(K07)과 성남 일화천마팀(K08)이다. 본 예제에서는 동명이인에 대한 내용을 예로 들었지만, 서브쿼리의 실행 결과가 2건 이상이 나오는 모든 경우에 다중 행 비교 연산자

를 사용해야 한다.

3. 다중 칼럼 서브쿼리

다중 칼럼 서브쿼리는 서브쿼리의 결과로 여러 개의 칼럼이 반환되어 메인쿼리의 조건과 동시에 비교되는 것을 의미한다.

소속팀별 키가 가장 작은 사람들의 정보를 출력하는 문제를 가지고 다중 칼럼 서브쿼리를 알아보도록 한다. 소속팀별 키가 가장 작은 사람들의 정보는 GROUP BY를 이용하여 찾을 수 있으므로 다음과 같이 SQL문을 작성할 수 있다.

```
[예제] SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키
       FROM PLAYER
       WHERE (TEAM_ID, HEIGHT) IN (SELECT TEAM_ID, MIN(HEIGHT)
                                   FROM PLAYER
                                   GROUP BY TEAM_ID)
       ORDER BY TEAM_ID, PLAYER_NAME;
[실행 결과] 팀코드 선수명 포지션 백넘버 키
         ----- -------- ------ ---- ---
           K01 마르코스 FW 44 170
           K01 박정수 MF 8 170
           K02 고창현 MF 8 170
           K02 정준 MF 44 170
           K03 김중규 MF 42 170
           19개의 행이 선택되었다.
```

SQL문의 실행 결과를 보면 서브쿼리의 결과값으로 소속팀코드(TEAM_ID)와 소속팀별 가장 작은 키를 의미하는 MIN(HEIGHT)라는 두 개의 칼럼을 반환했다. 메인쿼리에서는 조건절에 TEAM_ID와 HEIGHT 칼럼을 괄호로 묶어서 서브쿼리 결과와 비교하여 원하는 결과를 얻었다.

실행 결과에서 보면 하나 팀에서 키가 제일 작은 선수 한 명씩만 반환된 것이 아니라 같은 팀에서 여러 명이 반환된 것을 확인할 수 있다. 이것은 동일 팀 내에서 조건(팀별 가장 작은 키)을 만족하는 선수가 여러 명이 존재하기 때문이다. 그러나 이 기능은 SQL Server에서는 지원되지 않는 기능이다.

4. 연관 서브쿼리

연관 서브쿼리(Correlated Subquery)는 서브쿼리 내에 메인쿼리 칼럼이 사용된 서브쿼리이다.

선수 자신이 속한 팀의 평균 키보다 작은 선수들의 정보를 출력하는 SQL문을 연관 서브쿼리를 이용해서 작성해 보면 다음과 같다

```
[예제] SELECT T.TEAM_NAME 팀명, M.PLAYER_NAME 선수명, M.POSITION 포지션, M.BACK_NO 백넘버, M.HEIGHT 키
       FROM PLAYER M, TEAM T
       WHERE M.TEAM_ID = T.TEAM_ID AND M.HEIGHT <( SELECT AVG(S.HEIGHT)
                                                   FROM PLAYER S
                                                   WHERE S.TEAM_ID = M.TEAM_ID
                                                   AND S.HEIGHT IS NOT NULL
                                                   GROUP BY S.TEAM_ID )
       ORDER BY 선수명;
[실행 결과] 팀명 선수명 포지션 백넘버 키
           -------- ----- ----- ----- --
           삼성블루윙즈 가비 MF 10 177
           삼성블루윙즈 강대희 MF 26 174
           스틸러스 강용 DF 2 179
           시티즌 강정훈 MF 38 175
           드래곤즈 강철 DF 3 178
           현대모터스 고관영 MF 32 180
           현대모터스 고민기 FW 24 178
           삼성블루윙즈 고종수 MF 22 176
           224의 행이 선택되었다.
```

예를 들어, 가비 선수는 삼성블루윙즈팀 소속이므로 삼성블루윙즈팀 소속의 평균키를 구하고 그 평균키와 가비 선수의 키를 비교하여 적을 경우에 선수에 대한 정보를 출력한다. 만약, 평균키 보다 선수의 키가 크거나 같으면 조건에 맞지 않기 때문에 해당 데이터는 출력되지 않는다. 이와 같은 작업을 메인쿼리에 존재하는 모든 행에 대해서 반복 수행한다.

EXISTS 서브쿼리는 항상 연관 서브쿼리로 사용된다. 또한 EXISTS 서브쿼리의 특징은 아무리 조건을 만족하는 건이 여러 건이더라도 조건을 만족하는 1건만 찾으면 추가적인 검색을 진행하지 않는다.

다음은 EXISTS 서브쿼리를 사용하여 '20120501' 부터 '20120502' 사이에 경기가 있는 경기장을 조회하는 SQL문이다.

```
[예제] SELECT STADIUM_ID ID, STADIUM_NAME 경기장명
       FROM STADIUM A
       WHERE EXISTS (SELECT 1
                     FROM SCHEDULE X
                     WHERE X.STADIUM_ID = A.STADIUM_ID
                     AND X.SCHE_DATE BETWEEN '20120501' AND '20120502')
[실행 결과] ID 경기장명
           --- ----------------------------------
           B01 인천월드컵경기장
           B04 수원월드컵경기장
           B05 서울월드컵경기장
           C02 부산아시아드경기장
           4개의 행이 선택되었다.
```

5. 그밖에 위치에서 사용하는 서브쿼리

가. SELECT 절에 서브쿼리 사용하기

다음은 SELECT 절에서 사용하는 서브쿼리인 스칼라 서브쿼리(Scalar Subquery)에 대해서 알아본다. 스칼라 서브쿼리는 한 행, 한 칼럼(1 Row 1 Column)만을 반환하는 서브쿼리를 말한다. 스칼라 서브쿼리는 칼럼을 쓸 수 있는 대부분의 곳에서 사용할 수 있다.

선수 정보와 해당 선수가 속한 팀의 평균 키를 함께 출력하는 예제로 스칼라 서브쿼리를 설명하면 다음과 같다.

[그림 Ⅱ-2-15] 스칼라 서브쿼리

[그림 Ⅱ-2-15]는 2개의 SQL문으로 구성되어 있다. 선수들의 정보를 출력하는 SQL문(메인쿼리 부분)과 해당 선수의 소속팀별 평균키를 알아내는 SQL문(서브쿼리 부분)으로 구성된다. 여기서 선수의 소속팀별 평균키를 알아내는 스칼라 서브쿼리는 메인쿼리의 결과 건수만큼 반복수행 된다. [그림 Ⅱ-2-15]를 SQL문으로 작성하면 다음과 같다.

```
[예제] SELECT PLAYER_NAME 선수명, HEIGHT 키, (SELECT AVG(HEIGHT)
                                              FROM PLAYER X
                                              WHERE X.TEAM_ID = P.TEAM_ID) 팀평균키
       FROM PLAYER P
[실행 결과] 선수명 키 팀평균키
            ------- ---- -------------
            가비 177 179.067
            가이모토 182 178.854
            강대희 174 179.067
            강성일 182 177.485
            강용 179 179.911
            강정훈 175 177.485
            강철 178 178.391
            고관영 180 180.422
            480개의 행이 선택되었다.
```

스칼라 서브쿼리 또한 단일 행 서브쿼리이기 때문에 결과가 2건 이상 반환되면 SQL문은 오류를 반환한다.

나. FROM 절에서 서브쿼리 사용하기

FROM 절에서 사용되는 서브쿼리를 인라인 뷰(Inline View)라고 한다. FROM 절에는 테이블 명이 오도록 되어 있다. 그런데 서브쿼리가 FROM 절에 사용되면 어떻게 될까? 서브쿼리의 결과가 마치 실행 시에 동적으로 생성된 테이블인 것처럼 사용할 수 있다. 인라인 뷰는 SQL문이 실행될 때만 임시적으로 생성되는 동적인 뷰이기 때문에 데이터베이스에 해당 정보가 저장되지 않는다. 그래서 일반적인 뷰를 정적 뷰(Static View)라고 하고 인라인 뷰를 동적 뷰(Dynamic View)라고도 한다. 뷰에 대해서는 뒤에서 좀더 설명하기로 한다.

인라인 뷰는 테이블 명이 올 수 있는 곳에서 사용할 수 있다. 서브쿼리의 칼럼은 메인쿼리에서 사용할 수 없다고 했다. 그러나 인라인 뷰는 동적으로 생성된 테이블이다. 인라인 뷰를 사용하는 것은 조인 방식을 사용하는 것과 같다. 그렇기 때문에 인라인 뷰의 칼럼은 SQL문 자유롭게 참조할 수 있다.

K-리그 선수들 중에서 포지션이 미드필더(MF)인 선수들의 소속팀명 및 선수 정보를 출력하고자 한다. 인라인 뷰를 활용해서 SQL문을 만들어 보자.

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

```
[실행 결과] 팀명 선수명 백넘버
           --------- ------- -----
           삼성블루윙즈 가비 10
           삼성블루윙즈 강대희 26
           시티즌 강정훈 38
           현대모터스 고관영 32
           삼성블루윙즈 고종수 22
           삼성블루윙즈 고창현 8
           시티즌 공오균 22
           일화천마 곽치국 32
           162개의 행이 선택되었다.
```

SQL문을 보면 선수들 중에서 포지션이 미드필더(MF) 선수들을 인라인 뷰를 통해서 추출하고 인라인 뷰의 결과와 TEAM 테이블과 조인해서 팀명(TEAM_NAME)을 출력하고 있다.

인라인 뷰에서는 ORDER BY절을 사용할 수 있다. 인라인 뷰에 먼저 정렬을 수행하고 정렬된 결과 중에서 일부 데이터를 추출하는 것을 TOP-N 쿼리라고 한다. TOP-N 쿼리를 수행하기 위해서는 정렬 작업과 정렬 결과 중에서 일부 데이터만을 추출할 수 있는 방법이 필요하다. Oracle에서는 ROWNUM이라는 연산자를 통해서 결과로 추출하고자 하는 데이터 건수를 제약할 수 있다.

```
[예제] Oracle
       SELECT PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키
       FROM (SELECT PLAYER_NAME, POSITION, BACK_NO, HEIGHT
              FROM PLAYER
             WHERE HEIGHT IS NOT NULL
             ORDER BY HEIGHT DESC)
       WHERE ROWNUM <= 5;
```

```
[예제] SQL Server
   SELECT TOP(5) PLAYER_NAME AS 선수명, POSITION AS 포지션, BACK_NO AS 백넘버, HEIGHT AS 키
   FROM PLAYER
   WHERE HEIGHT IS NOT NULL
   ORDER BY HEIGHT DESC
```

```
[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
           5개의 행이 선택되었다.
```

당 SQL문의 인라인 뷰에서 선수의 키를 내림차순으로 정렬(가장 키가 큰 선수부터 출력)한 후 메인쿼리에서 ROWNUM을 사용해서 5명의 선수의 정보만을 추출하였다. 이것은 모든 선수들 중에서 가장 키가 큰 5명의 선수를 출력한 것이다. 만약, 다른 선수 중에서 키가 192인 선수가 더 존재하더라도 해당 SQL문에서는 데이터가 출력되지 않는다. 이런 데이터까지 추출하고자 한다면 분석함수의 RANK관련 함수를 사용해야 한다.

다. HAVING 절에서 서브쿼리 사용하기

HAVING 절은 그룹함수와 함께 사용될 때 그룹핑된 결과에 대해 부가적인 조건을 주기 위해서 사용한다.

평균키가 삼성 블루윙즈팀의 평균키보다 작은 팀의 이름과 해당 팀의 평균키를 구하는 SQL문을 작성하면 다음과 같다.

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

```
[실행 결과] 팀코드 팀명 평균키
           ---- ----------- ------
           K13 강원FC 173.667
```

```
K15 대구FC 175.333
K11 경남FC 176.333
K14 제주유나이티드FC 169.5
K12 광주상무 173.5
K07 드래곤즈 178.391
K08 일화천마 178.854
K10 시티즌 177.485
8개의 행이 선택되었다.
```

라. UPDATE문의 SET 절에서 사용하기

현재 TEAM 테이블에는 STADIUM_NAME 칼럼이 없다. TEAM 테이블에 STADIUM_NAME을 추가(ALTER TABLE ADD COLUMN)하였다고 가정하자. TEAM 테이블에 추가된 STADIUM_NAME의 값을 STADIUM 테이블을 이용하여 변경하고자 할 때 다음과 같이 SQL문을 작성할 수 있다.

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

서브쿼리를 사용한 변경 작업을 할 때 서브쿼리의 결과가 NULL을 반환할 경우 해당 컬럼의 결과가 NULL이 될 수 있기 때문에 주의해야 한다.

마. INSERT문의 VALUES절에서 사용하기

PLAYER 테이블에 '홍길동'이라는 선수를 삽입하고자 한다. 이때 PLAYER_ID의 값을 현재 사용중인 PLAYER_ID에 1을 더한 값으로 넣고자 한다. 다음과 같이 SQL문을 SQL문을 작성할 수 있다.

```
INSERT INTO PLAYER(PLAYER_ID, PLAYER_NAME, TEAM_ID)
VALUES((SELECT TO_CHAR(MAX(TO_NUMBER(PLAYER_ID))+1)
FROM PLAYER), '홍길동', 'K06');
```

6. 뷰(View)

테이블은 실제로 데이터를 가지고 있는 반면, 뷰(View)는 실제 데이터를 가지고 있지 않다. 뷰는 단지 뷰 정의(View Definition)만을 가지고 있다. 질의에서 뷰가 사용되면 뷰 정의를 참조해서 DBMS 내부적으로 질의를 재작성(Rewrite)하여 질의를 수행한다.

뷰는 실제 데이터를 가지고 있지 않지만 테이블이 수행하는 역할을 수행하기 때문에 가상 테이블(Virtual Table)이라고도 한다. 뷰는 [표 Ⅱ-2-7]과 같은 장점을 갖는다.

[표 Ⅱ-2-7] 뷰 사용의 장점

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

뷰는 다음과 같이 CREATE VIEW문을 통해서 생성할 수 있다.

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

해당 뷰는 선수 정보와 해당 선수가 속한 팀명을 함께 추출하는 것이다. 뷰의 명칭은 'V_PLAYER_TEAM'이다. 뷰는 테이블뿐만 아니라 이미 존재하는 뷰를 참조해서도 생성할 수 있다.

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

V_PLAYER_TEAM_FILTER 뷰는 이미 앞에서 생성했던 V_PLAYER_TEAM 뷰를 기반으로 해서 생성된 뷰다. V_PLAYER_TEAM_FILTER 뷰는 선수 포지션이 골키퍼(GK), 미드필더(MF)인 선수만을 추출하고자 하는 뷰이다.(뷰를 포함하는 뷰를 잘못 생성하는 경우 성능상의 문제를 유발할 수 있으므로, 뷰와 SQL의 수행원리를 잘 이해하고 사용하기 바란다) 뷰를 사용하기 위해서는 해당 뷰의 이름을 이용하면 된다.

뷰를 사용하는 방법은 다음과 같다.

```
[예제] SELECT PLAYER_NAME, POSITION, BACK_NO, TEAM_ID, TEAM_NAME
       FROM V_PLAYER_TEAM
       WHERE PLAYER_NAME LIKE '황%'
[실행 결과] PLAYER_NAME POSITION BACK_NO TEAM_ID TEAM_NAME
            ----------- -------- ------- ------- ---------
            황철민 MF 35 K06 아이파크
            황승주 DF 98 K05 현대모터스
            황연석 FW 16 K08
            일화천마 3개의 행이 선택되었다.
```

이것은 V_PLAYER_TEAM 뷰에서 성이 '황'씨인 선수만을 추출하는 SQL문이다. 결과로서

3건이 추출되었다. 뷰를 사용하는 경우에는 DBMS가 내부적으로 SQL문을 다음과 같이 재작성한다.

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

이것은 앞에서 설명했던 인라인 뷰와 유사한 모습임을 알 수 있다. 이와 같은 형태로 사용되기 때문에 뷰는 데이터를 저장하지 않고도 데이터를 조회할 수 있다.

뷰를 제거하기 위해서는 DROP VIEW문을 사용한다.

```
DROP VIEW V_PLAYER_TEAM;
DROP VIEW V_PLAYER_TEAM_FILTER;
```

제5절 그룹 함수(GROUP 함수)

1. 데이터 분석 개요

ANSI/ISO SQL 표준은 데이터 분석을 위해서 다음 세 가지 함수를 정의하고 있다.

- AGGREGATE FUNCTION
- GROUP FUNCTION
- WINDOW FUNCTION

■ AGGREGATE FUNCTION

GROUP AGGREGATE FUNCTION이라고도 부르며, GROUP FUNCTION의 한 부분으로 분류할 수 있다. 1장 7절에서 설명한 COUNT, SUM, AVG, MAX, MIN 외 각종 집계 함수들이 포함되어 있다.

■ GROUP FUNCTION

결산 개념의 업무를 가지는 원가나 판매 시스템의 경우는 소계, 중계, 합계, 총 합계 등 여러 레벨의 결산 보고서를 만드는 것이 중요 업무 중의 하나이다. 개발자들이 이런 보고서를 작성하기 위해서는 SQL이 포함된 3GL으로 배치 프로그램을 작성하거나, 레벨별 집계를 위한 여러 단계의 SQL을 UNION, UNION ALL로 묶은 후 하나의 테이블을 여러 번 읽어 다시 재정렬하는 복잡한 단계를 거쳐야 했다.

그러나 그룹 함수를 사용한다면 하나의 SQL로 테이블을 한 번만 읽어서 빠르게 원하는 리포트를 작성할 수 있다. 추가로, 소계/합계를 표시하기 위해 GROUPING 함수와 CASE 함수를 이용하면 쉽게 원하는 포맷의 보고서 작성도 가능하다. 그룹 함수로는 집계 함수를 제외하고, 소그룹 간의 소계를 계산하는 ROLLUP 함수, GROUP BY 항목들 간 다차원적인 소계를 계산 할 수 있는 CUBE 함수, 특정 항목에 대한 소계를 계산하는 GROUPING SETS 함수가 있다.

ROLLUP은 GROUP BY의 확장된 형태로 사용하기가 쉬우며 병렬로 수행이 가능하기 때문에 매우 효과적일 뿐 아니라 시간 및 지역처럼 계층적 분류를 포함하고 있는 데이터의 집계에 적합하도록 되어 있다.

CUBE는 결합 가능한 모든 값에 대하여 다차원적인 집계를 생성하게 되므로 ROLLUP에 비해 다양한 데이터를 얻는 장점이 있는 반면에, 시스템에 부하를 많이 주는 단점이 있다.

GROUPING SETS는 원하는 부분의 소계만 손쉽게 추출할 수 있는 장점이 있다.

ROLLUP, CUBE, GROUPING SETS 결과에 대한 정렬이 필요한 경우는 ORDER BY 절에 정렬 칼럼을 명시해야 한다.

■ WINDOW FUNCTION

분석 함수(ANALYTIC FUNCTION)나 순위 함수(RANK FUNCTION)로도 알려져 있는 윈도우 함수는 데이터웨어하우스에서 발전한 기능이며, 자세한 내용은 다음 절에서 설명한다.

2. ROLLUP 함수

ROLLUP에 지정된 Grouping Columns의 List는 Subtotal을 생성하기 위해 사용되어지며, Grouping Columns의 수를 N이라고 했을 때 N+1 Level의 Subtotal이 생성된다. 중요한 것은, ROLLUP의 인수는 계층 구조이므로 인수 순서가 바뀌면 수행 결과도 바뀌게 되므로 인수의 순서에도 주의해야 한다.

ROLLUP과 CUBE의 효과를 알아보기 위해 단계별로 데이터를 출력해본다.

STEP 1. 일반적인 GROUP BY 절 사용

[예제] 부서명과 업무명을 기준으로 사원수와 급여 합을 집계한 일반적인 GROUP BY SQL 문장을 수행한다.

[예제]

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

[실행 결과]

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

9개의 행이 선택되었다.

Oracle을 포함한 일부 DBMS의 과거 버전에서는 GROUP BY 절 사용시 자동적으로 정렬을 수행하였으나, 현재 대부분의 DBMS 버전은 집계 기능만 지원하고 있으며 정렬이 필요한 경우는 ORDER BY 절에 명시적으로 정렬 칼럼이 표시되어야 한다.

STEP 1-2. GROUP BY 절 + ORDER BY 절 사용

[예제] 부서명과 업무명을 기준으로 집계한 일반적인 GROUP BY SQL 문장에 ORDER BY 절을 사용함으로써 부서, 업무별로 정렬이 이루어진다.

[예제]

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

[실행 결과]

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

9개의 행이 선택되었다.

STEP 2. ROLLUP 함수 사용

[예제] 부서명과 업무명을 기준으로 집계한 일반적인 GROUP BY SQL 문장에 ROLLUP 함수를 사용한다.

[예제]

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

[실행 결과]

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

13개의 행이 선택되었다.

실행 결과에서 2개의 GROUPING COLUMNS(DNAME, JOB)에 대하여 다음과 같은 추가 LEVEL의 집계가 생성된 것을 볼 수 있다.

L1 - GROUP BY 수행시 생성되는 표준 집계 (9건)

L2 - DNAME 별 모든 JOB의 SUBTOTAL (3건)

L3 - GRAND TOTAL (마지막 행, 1건)

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

STEP 2-2. ROLLUP 함수 + ORDER BY 절 사용

[예제] 부서명과 업무명을 기준으로 집계한 일반적인 GROUP BY SQL 문장에 ROLLUP 함수를 사용한다. 추가로 ORDER BY 절을 사용해서 부서, 업무별로 정렬한다.

[예제]

```
SELECT DNAME, JOB,
       COUNT(*) "Total Empl",
       SUM(SAL) "Total Sal"
FROM EMP, DEPT
```

```
WHERE DEPT.DEPTNO = EMP.DEPTNO
GROUP BY ROLLUP (DNAME, JOB)
ORDER BY DNAME, JOB ;
```

[실행 결과]

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

13개의 행이 선택되었다.

STEP 3. GROUPING 함수 사용

ROLLUP, CUBE, GROUPING SETS 등 새로운 그룹 함수를 지원하기 위해 GROUPING 함수가 추가되었다.

- ROLLUP이나 CUBE에 의한 소계가 계산된 결과에는 GROUPING(EXPR) = 1 이 표시되고,
- 그 외의 결과에는 GROUPING(EXPR) = 0 이 표시된다.

GROUPING 함수와 CASE/DECODE를 이용해, 소계를 나타내는 필드에 원하는 문자열을 지정할 수 있어, 보고서 작성시 유용하게 사용할 수 있다.

[예제] ROLLUP 함수를 추가한 집계 보고서에서 집계 레코드를 구분할 수 있는 GROUPING 함수가 추가된 SQL 문장이다.

[예제]

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

[실행 결과]

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

13개의 행이 선택되었다.

부서별, 업무별과 전체 집계를 표시한 레코드에서는 GROUPING 함수가 1을 리턴한 것을 확인할 수 있다. 그리고 전체 합계를 나타내는 결과 라인에서는 부서별 GROUPING 함수와 업무별 GROUPING 함수가 둘 다 1인 것을 알 수 있다.

STEP 4. GROUPING 함수 + CASE 사용

[예제] ROLLUP 함수를 추가한 집계 보고서에서 집계 레코드를 구분할 수 있는 GROUPING 함수와 CASE 함수를 함께 사용한 SQL 문장을 작성한다.

[예제]

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

Oracle의 경우는 DECODE 함수를 사용해서 좀더 짧게 표현할 수 있다.

```
SELECT
DECODE(GROUPING(DNAME), 1, 'All Departments', DNAME) AS DNAME,
```

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

[실행 결과]

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

13개의 행이 선택되었다.

부서별과 전체 집계를 표시한 레코드에서 'ALL JOBS'와 'ALL DEPARTMENTS'라는 사용자 정의 텍스트를 확인할 수 있다. 일부 DBMS는 GROUPING_ID라는 비슷한 용도의 함수를 추가로 사용할 수도 있으므로 참조하기 바란다.

STEP 4-2. ROLLUP 함수 일부 사용

[예제] GROUP BY ROLLUP (DNAME, JOB) 조건에서 GROUP BY DNAME, ROLLUP(JOB) 조건으로 변경한 경우이다.

[예제]

```
SELECT
CASE GROUPING(DNAME) WHEN 1 THEN 'All Departments' ELSE DNAME END AS DNAME,
CASE GROUPING(JOB) WHEN 1 THEN 'All Jobs' ELSE JOB END AS JOB,
COUNT(*) "Total Empl",
SUM(SAL) "Total Sal"
```

```
FROM EMP, DEPT
WHERE DEPT.DEPTNO = EMP.DEPTNO
GROUP BY DNAME, ROLLUP(JOB)
```

[실행 결과]

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

12개의 행이 선택되었다.

결과는 마지막 ALL DEPARTMENTS &ALL JOBS 줄만 계산이 되지 않았다. ROLLUP이 JOB 칼럼에만 사용되었기 때문에 DNAME에 대한 집계는 필요하지 않기 때문이다.

삭제된 레코드)

All Departments	All Jobs	14	29025

STEP 4-3. ROLLUP 함수 결합 칼럼 사용

[예제] JOB과 MGR는 하나의 집합으로 간주하고, 부서별, JOB &MGR에 대한 ROLLUP 결과를 출력한다.

[예제]

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

☞ JOB, MGR을 소계시 하나의 집합으로 간주하여 구분하지 않음

[실행 결과]

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

14개의 행이 선택되었다.

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

3. CUBE 함수

ROLLUP에서는 단지 가능한 Subtotal만을 생성하였지만, CUBE는 결합 가능한 모든 값에 대하여 다차원 집계를 생성한다. CUBE를 사용할 경우에는 내부적으로는 Grouping Columns의 순서를 바꾸어서 또 한 번의 Query를 추가 수행해야 한다. 뿐만 아니라 Grand Total은 양쪽의 Query 에서 모두 생성이 되므로 한 번의 Query에서는 제거되어야만 하므로 ROLLUP에 비해 시스템의 연산 대상이 많다.

이처럼 Grouping Columns이 가질 수 있는 모든 경우에 대하여 Subtotal을 생성해야 하는 경우에는 CUBE를 사용하는 것이 바람직하나, ROLLUP에 비해 시스템에 많은 부담을 주므로 사용에 주의해야 한다.

CUBE 함수의 경우 표시된 인수들에 대한 계층별 집계를 구할 수 있으며, 이때 표시된 인수들 간에는 계층 구조인 ROLLUP과는 달리 평등한 관계이므로 인수의 순서가 바뀌는 경우 행간에 정렬 순서는 바뀔 수 있어도 데이터 결과는 같다.

그리고 CUBE도 결과에 대한 정렬이 필요한 경우는 ORDER BY 절에 명시적으로 정렬 칼럼이 표시가 되어야 한다.

STEP 5. CUBE 함수 이용

[예제] GROUP BY ROLLUP (DNAME, JOB) 조건에서 GROUP BY CUBE (DNAME, JOB) 조건으로 변경해서 수행한다.

[예제]

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

[실행 결과]

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

18개의 행이 선택되었다.

CUBE는 GROUPING COLUMNS이 가질 수 있는 모든 경우의 수에 대하여 Subtotal을 생성하므로 GROUPING COLUMNS의 수가 N이라고 가정하면, 2의 N승 LEVEL의 Subtotal을 생성하게 된다.

실행 결과에서 CUBE 함수 사용으로 ROLLUP 함수의 결과에다 업무별 집계까지 추가해서 출력할 수 있는데, ROLLUP 함수에 비해 업무별 집계를 표시한 5건의 레코드가 추가된 것을 확인할 수 있다. (All Departments - CLERK, ANALYST, MANAGER, SALESMAN, PRESIDENT 별 집계가 5건 추가되었다.)

STEP 5-2. UNION ALL 사용 SQL

UNION ALL은 Set Operation 내용으로, 여러 SQL 문장을 연결하는 역할을 할 수 있다.

위 SQL은 첫 번째 SQL 모듈부터 차례대로 결과가 나오므로 위 CUBE SQL과 결과 데이터는 같으나 행들의 정렬은 다를 수 있다.

[예제]

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

CUBE 함수를 사용하면서 가장 크게 개선되는 부분은 CUBE 사용 전 SQL에서 EMP, DEPT 테이블을 네 번이나 반복 액세스하는 부분을 CUBE 사용 SQL에서는 한 번으로 줄일 수 있는 부분이다. 기존에 같은 테이블을 네 번 액세스하는 이유가 되었던 부서와 업무별 소계와 총계 부분을 CUBE 함수를 사용함으로써 한 번의 액세스만으로 구현한다.

결과적으로 수행속도 및 자원 사용율을 개선할 수 있으며, SQL 문장도 더 짧아졌으므로 가독성도 높아졌다. 실행 결과는 STEP5의 결과와 동일하다. ROLLUP 함수도 똑 같은 개선 효과를 얻을 수 있다.

4. GROUPING SETS 함수

GROUPING SETS를 이용해 더욱 다양한 소계 집합을 만들 수 있는데, GROUP BY SQL 문장을 여러 번 반복하지 않아도 원하는 결과를 쉽게 얻을 수 있게 되었다.

GROUPING SETS에 표시된 인수들에 대한 개별 집계를 구할 수 있으며, 이때 표시된 인수들 간에는 계층 구조인 ROLLUP과는 달리 평등한 관계이므로 인수의 순서가 바뀌어도 결과는 같다.

그리고 GROUPING SETS 함수도 결과에 대한 정렬이 필요한 경우는 ORDER BY 절에 명시적으로 정렬 칼럼이 표시가 되어야 한다.

■ 일반 그룹함수를 이용한 SQL

[예제] 일반 그룹함수를 이용하여 부서별, JOB별 인원수와 급여 합을 구하라.

[예제]

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

[실행 결과]

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

8개의 행이 선택되었다.

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

■ GROUPING SETS 사용 SQL

[예제] 일반 그룹함수를 GROUPING SETS 함수로 변경하여 부서별, JOB별 인원수와 급여 합을 구하라.

[예제]

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

[실행 결과]

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

8개의 행이 선택되었다.

GROUPING SETS 함수 사용시 UNION ALL을 사용한 일반 그룹함수를 사용한 SQL과 같은 결과를 얻을 수 있으며, 괄호로 묶은 집합 별로(괄호 내는 계층 구조가 아닌 하나의 데이터로 간주함) 집계를 구할 수 있다. GROUPING SETS의 경우 일반 그룹함수를 이용한 SQL과 결과 데이터는 같으나 행들의 정렬 순서는 다를 수 있다.

■ GROUPING SETS 사용 SQL - 순서 변경

[예제] 일반 그룹함수를 GROUPING SETS 함수로 변경하여 부서별, JOB별 인원수와 급여 합을 구하는데 GROUPING SETS의 인수들의 순서를 바꾸어 본다.

[예제]

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

[실행 결과]

DNAME	JOB	Total Empl	Total Sal
All Departments	CLERK	4	4150
All Departments	SALESMAN	4	5600
All Departments	PRESIDENT	1	5000
All Departments	MANAGER	3	8275
All Departments	ANALYST	2	6000

ACCOUNTING	All Jobs	3	8750
RESEARCH	All Jobs	5	10875
SALES	All Jobs	6	9400

8개의 행이 선택되었다.

GROUPING SETS 인수들은 평등한 관계이므로 인수의 순서가 바뀌어도 결과는 같다. (JOB과 DNAME의 순서가 바뀌었지만 결과는 같다.)

■ 3개의 인수를 이용한 GROUPING SETS 이용

[예제] 부서-JOB-매니저 별 집계와, 부서-JOB 별 집계와, JOB-매니저 별 집계를 GROUPING SETS 함수를 이용해서 구해본다.

[예제]

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

GROUPING SETS 함수 사용시 괄호로 묶은 집합별로(괄호 내는 계층구조가 아닌 하나의 데이터로 간주함) 집계를 구할 수 있다.

[실행 결과]

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

MANAGER		7839	8275
SALESMAN		7698	5600
PRESIDENT			5000
SALES	MANAGER		2850
SALES	CLERK		950
ACCOUNTING	CLERK		1300
ACCOUNTING	MANAGER		2450
ACCOUNTING	PRESIDENT		5000
RESEARCH	MANAGER		2975
SALES	SALESMAN		5600
RESEARCH	ANALYST		6000
RESEARCH	CLERK		1900

27개의 행이 선택되었다.

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

제6절 윈도우 함수(WINDOW 함수)

1. WINDOW FUNCTION 개요

기존 관계형 데이터베이스는 칼럼과 칼럼간의 연산, 비교, 연결이나 집합에 대한 집계는 쉬운 반면, 행과 행간의 관계를 정의하거나, 행과 행간을 비교, 연산하는 것을 하나의 SQL 문으로 처리 하는 것은 매우 어려운 문제였다. PL/SQL, SQL/PL, T-SQL, PRO*C 같은 절차형 프로그램을 작성하거나, INLINE VIEW를 이용해 복잡한 SQL 문을 작성해야 하던 것을 부분적이나마 행과 행간의 관계를 쉽게 정의하기 위해 만든 함수가 바로 WINDOW FUNCTION이다. 윈도우 함수를 활용하면 복잡한 프로그램을 하나의 SQL 문장으로 쉽게 해결할 수 있다.

분석 함수(ANALYTIC FUNCTION)나 순위 함수(RANK FUNCTION)로도 알려져 있는 윈도우 함수(ANSI/ISO SQL 표준은 WINDOW FUNCTION이란 용어를 사용함)는 데이터웨어하우스에서 발전한 기능이다. SQL 사용자 입장에서는 INLINE VIEW 이후 SQL의 중요한 기능이 추가되었다고 할 수 있으며, 많은 프로그램이나 튜닝 팁을 대체할 수 있을 것이다.

복잡하거나 자원을 많이 사용하는 튜닝 기법들을 대체할 수 있는 DBMS의 새로운 기능은 튜닝 관점에서도 최적화된 방법이므로 적극적으로 활용할 필요가 있다. 같은 결과가 나오는 변형된 튜닝 문장보다는 DBMS 벤더에서 최적화된 자원을 사용하도록 만들어진 새로운 기능을 사용하는 것이 일반적으로 더욱 효과가 좋기 때문이다.

WINDOW 함수는 기존에 사용하던 집계 함수도 있고, 새로이 WINDOW 함수 전용으로 만들어진 기능도 있다. 그리고 WINDOW 함수는 다른 함수와는 달리 중첩(NEST)해서 사용하지는 못하지만, 서브쿼리에서는 사용할 수 있다.

- WINDOW FUNCTION 종류

WINDOW FUNCTION의 종류는 크게 다섯 개의 그룹으로 분류할 수 있는데 벤더별로 지원하는 함수에는 차이가 있다.

첫 번째, 그룹 내 순위(RANK) 관련 함수는 RANK, DENSE_RANK, ROW_NUMBER 함수가 있다. ANSI/ISO SQL 표준과 Oracle, SQL Server 등 대부분의 DBMS에서 지원하고 있다.

두 번째, 그룹 내 집계(AGGREGATE) 관련 함수는 일반적으로 많이 사용하는 SUM, MAX, MIN, AVG, COUNT 함수가 있다. ANSI/ISO SQL 표준과 Oracle, SQL Server 등 대부분의 DBMS에서 지원하고 있는데, SQL Server의 경우 집계 함수는 뒤에서 설명할 OVER 절 내의 ORDER BY 구문을 지원하지 않는다.

세 번째, 그룹 내 행 순서 관련 함수는 FIRST_VALUE, LAST_VALUE, LAG, LEAD 함수가 있다. Oracle에서만 지원되는 함수이기는 하지만, FIRST_VALUE, LAST_VALUE 함수는 MAX, MIN 함수와 비슷한 결과를 얻을 수 있고, LAG, LEAD 함수는 DW에서 유용하게 사용되는 기능이므로 같이 설명하도록 한다.

네 번째, 그룹 내 비율 관련 함수는 CUME_DIST, PERCENT_RANK, NTILE, RATIO_TO_REPORT 함수가 있다. CUME_DIST, PERCENT_RANK 함수는 ANSI/ISO SQL

표준과 Oracle DBMS에서 지원하고 있으며, NTILE 함수는 ANSI/ISO SQL 표준에는 없지만, Oracle, SQL Server에서 지원하고 있다. 마지막으로 RATIO_TO_REPORT 함수는 Oracle에서만 지원되는 함수이기는 하나, 현업에서 유용한 기능을 구현하는데 참조하기 위해 설명하도록 한다.

다섯 번째, 선형 분석을 포함한 통계 분석 관련 함수가 있는데, 통계에 특화된 기능이므로 본 가이드에서는 설명을 생략한다. 아래는 Oracle의 통계 관련 함수를 참조로 표시한 것이다.

```
CORR, COVAR_POP, COVAR_SAMP, STDDEV, STDDEV_POP, STDDEV_SAMP,
VARIANCE, VAR_POP, VAR_SAMP, REGR_(LINEAR REGRESSION), REGR_SLOPE,
REGR_INTERCEPT, REGR_COUNT, REGR_R2, REGR_AVGX, REGR_AVGY,
REGR_SXX, REGR_SYY, REGR_SXY
```

- WINDOW FUNCTION SYNTAX

- WINDOW 함수에는 OVER 문구가 키워드로 필수 포함된다.

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

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

```
BETWEEN 사용 타입
ROWS | RANGE BETWEEN
 UNBOUNDED PRECEDING | CURRENT ROW | VALUE_EXPR PRECEDING/FOLLOWING
AND
 UNBOUNDED FOLLOWING | CURRENT ROW | VALUE_EXPR PRECEDING/FOLLOWING

BETWEEN 미사용 타입
ROWS | RANGE
 UNBOUNDED PRECEDING | CURRENT ROW | VALUE_EXPR PRECEDING
```

2. 그룹 내 순위 함수

가. RANK 함수

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

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

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

[실행 결과]

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

13개의 행이 선택되었다.

업무 구분이 없는 ALL_RANK 칼럼에서 FORD와 SCOTT, WARD와 MARTIN은 동일한 SALARY이므로 같은 순위를 부여한다. 그리고 업무를 PARTITION으로 구분한 JOB_RANK의 경우 같은 업무 내 범위에서만 순위를 부여한다.

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

[예제] 앞의 SQL문의 결과는 JOB과 SALARY 기준으로 정렬이 되어있지 않다. 새로운 SQL에서는 전체 SALARY 순위를 구하는 ALL_RANK 칼럼은 제외하고, 업무별로 SALARY 순서를 구하는 JOB_RANK만 알아보도록 한다.

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

[실행 결과]

JOB	ENAME	SAL	JOB_RANK
ANALYST	FORD	3000	1
ANALYST	SCOTT	3000	1
CLERK	MILLER	1300	1
CLERK	ADAMS	1100	2
CLERK	JAMES	950	3
CLERK	SMITH	800	4
MANAGER	JONES	2975	1
MANAGER	BLAKE	2850	2
MANAGER	CLARK	2450	3
PRESIDENT	KING	5000	1
SALESMAN	ALLEN	1600	1
SALESMAN	TURNER	1500	2
SALESMAN	MARTIN	1250	3
SALESMAN	WARD	1250	3

13개의 행이 선택되었다.

업무별로 SALARY 순서를 구하는 JOB_RANK만 사용한 경우 파티션의 기준이 된 JOB과 SALARY 별로 정렬이 되어 있는 것을 알 수 있다.

나. DENSE_RANK 함수

DENSE_RANK 함수는 RANK 함수와 흡사하나, 동일한 순위를 하나의 건수로 취급하는 것이 틀린 점이다.

[예제] 사원데이터에서 급여가 높은 순서와, 동일한 순위를 하나의 등수로 간주한 결과도 같이 출력한다.

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

[실행 결과]

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

13개의 행이 선택되었다.

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

그러나 RANK와 DENSE_RANK의 차이를 알 수 있는 데이터는 FORD와 SCOTT의 다음 순위인 JONES의 경우 RANK는 4등으로 DENSE_RANK는 3등으로 표시되어 있다. 마찬가지로 WARD와 MARTIN의 다음 순위인 ADAMS의 경우 RANK는 12등으로 DENSE_RANK는 10등으로 표시되어 있다.

다. ROW_NUMBER 함수

ROW_NUMBER 함수는 RANK나 DENSE_RANK 함수가 동일한 값에 대해서는 동일한 순위를 부여하는데 반해, 동일한 값이라도 고유한 순위를 부여한다.

[예제] 사원데이터에서 급여가 높은 순서와, 동일한 순위를 인정하지 않는 등수도 같이 출력한다.

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

[실행 결과]

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

14개의 행이 선택되었다.

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

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

3. 일반 집계 함수

가. SUM 함수

SUM 함수를 이용해 파티션별 윈도우의 합을 구할 수 있다.

[예제] 사원들의 급여와 같은 매니저를 두고 있는 사원들의 SALARY 합을 구한다.

```
[예제]
SELECT MGR, ENAME, SAL, SUM(SAL) OVER (PARTITION BY MGR) MGR_SUM
```

```
FROM EMP;

PARTITION BY MGR 구문을 통해 매니저별로 데이터를 파티션화 한다.
```

[실행 결과]

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

14개의 행이 선택되었다.

[예제] OVER 절 내에 ORDER BY 절을 추가해 파티션 내 데이터를 정렬하고 이전 SALARY 데이터까지의 누적값을 출력한다. (SQL Server의 경우 집계 함수의 경우 OVER 절 내의 ORDER BY 절을 지원하지 않는다.)

```
[예제] SELECT MGR, ENAME, SAL,
       SUM(SAL) OVER (PARTITION BY MGR ORDER BY SAL RANGE UNBOUNDED PRECEDING)
       as MGR_SUM
FROM EMP

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

[실행 결과]

MGR	ENAME	SAL	MGR_SUM
7566	SCOTT	3000	6000
7566	FORD	3000	6000

7698	JAMES		950	950
7698	WARD	*	1250	3450
7698	MARTIN	*	1250	3450
7698	TURNER		1500	4950
7698	ALLEN		1600	6550
7782	MILLER		1300	1300
7788	ADAMS		1100	1100
7839	CLARK		2450	2450
7839	BLAKE		2850	5300
7839	JONES		2975	8275
7902	SMITH		800	800
	KING		5000	5000

14개의 행이 선택되었다.

* 표시된 7699-WARD와 7699-MARTIN의 급여가 같으므로, 같은 ORDER로 취급하여 950+ 1250+ 1250=3450의 값이 되었다. 7698-TURNER의 경우 950+ 1250+ 1250+ 1500=4950의 누적합을 가진다.

나. MAX 함수

MAX 함수를 이용해 파티션별 윈도우의 최대값을 구할 수 있다.

[예제] 사원들의 급여와 같은 매니저를 두고 있는 사원들의 SALARY 중 최대값을 같이 구한다.

```
[예제]
SELECT MGR, ENAME, SAL, MAX(SAL) OVER (PARTITION BY MGR) as MGR_MAX
FROM EMP;
```

[실행 결과]

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

```
7839            CLARK                                2450          2975
7902            SMITH                                 800           800
                KING                                 5000          5000
14개의 행이 선택되었다.
```

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

[예제] 추가로, INLINE VIEW를 이용해 파티션별 최대값을 가진 행만 추출할 수도 있다.

```
[예제]
SELECT MGR, ENAME, SAL
FROM (SELECT MGR, ENAME, SAL, MAX(SAL) OVER (PARTITION BY MGR)
                as IV_MAX_SAL
        FROM EMP)
WHERE SAL = IV_MAX_SAL ;
```

```
[실행 결과]
MGR             ENAME                                SAL
----            ------                               ---
7566            FORD                                 3000
7566            SCOTT                                3000
7698            ALLEN                                1600
7782            MILLER                               1300
7788            ADAMS                                1100
7839            JONES                                2975
7902            SMITH                                 800
                KING                                 5000
8개의 행이 선택되었다.
```

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

다. MIN 함수

MIN 함수를 이용해 파티션별 윈도우의 최소값을 구할 수 있다.

[예제] 사원들의 급여와 같은 매니저를 두고 있는 사원들을 입사일자를 기준으로 정렬하고, SALARY 최소값을 같이 구한다.

```
[예제]
SELECT MGR, ENAME, HIREDATE, SAL,
          MIN(SAL) OVER(PARTITION BY MGR ORDER BY HIREDATE) as MGR_MIN
FROM EMP;
```

[실행 결과]

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

14개의 행이 선택되었다.

라. AVG 함수

AVG 함수와 파티션별 ROWS 윈도우를 이용해 원하는 조건에 맞는 데이터에 대한 통계값을 구할 수 있다.

[예제] EMP 테이블에서 같은 매니저를 두고 있는 사원들의 평균 SALARY를 구하는데, 조건은 같은 매니저 내에서 자기 바로 앞의 사번과 바로 뒤의 사번인 직원만을 대상으로 한다.

```
[예제]
SELECT MGR, ENAME, HIREDATE, SAL,
          ROUND (AVG(SAL) OVER (PARTITION BY MGR ORDER BY HIREDATE
                    ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)) as MGR_AVG
```

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

[실행 결과]

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

14개의 행이 선택되었다.

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

TURNER의 경우 앞의 한건과, 본인의 데이터와, 뒤의 한 건으로 평균값을 구한다. (1250 + 1500 + 1250) / 3 = 1333의 값을 가진다. JAMES의 경우 파티션 내에서 마지막 데이터이므로 뒤의 한 건을 제외한, 앞의 한 건과 본인의 데이터를 가지고 평균값을 구한다. (1250 + 950) / 2 = 1100의 값을 가진다.

마. COUNT 함수

COUNT 함수와 파티션별 ROWS 윈도우를 이용해 원하는 조건에 맞는 데이터에 대한 통계 값을 구할 수 있다.

[예제] 사원들을 급여 기준으로 정렬하고, 본인의 급여보다 50 이하가 적거나 150 이하로 많은 급여를 받는 인원수를 출력하라.

[예제]

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

RANGE BETWEEN 50 PRECEDING AND 150 FOLLOWING :

현재 행의 급여값을 기준으로 급여가 -50에서 +150의 범위 내에 포함된 모든 행이 대상이 된다. (RANGE는 현재 행의 데이터 값을 기준으로 앞뒤 데이터 값의 범위를 표시하는 것임)

[실행 결과]

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

14개의 행이 선택되었다.

위 SQL 문장은 파티션이 지정되지 않았으므로 모든 건수를 대상으로 -50 ~ +150 기준에 맞는지 검사하게 된다. ORDER BY SAL로 정렬이 되어 있으므로 비교 연산이 쉬워진다. ** 표시된 ADAMS의 경우 자기가 가지고 있는 SALARY 1100을 기준으로 -50에서 +150까지 값을 가진 1050에서 1250까지의 값을 가진 JAMES(950), ADAMS(1100), WARD(1250) 3명의 데이터 건수를 구할 수 있다.

4. 그룹 내 행 순서 함수

가. FIRST_VALUE 함수

FIRST_VALUE 함수를 이용해 파티션별 윈도우에서 가장 먼저 나온 값을 구한다. SQL Server에서는 지원하지 않는 함수이다. MIN 함수를 활용하여 같은 결과를 얻을 수도 있다.

[예제] 부서별 직원들을 연봉이 높은 순서부터 정렬하고, 파티션 내에서 가장 먼저 나온 값

을 출력한다.

```
[예제]
SELECT DEPTNO, ENAME, SAL, FIRST_VALUE(ENAME)
        OVER (PARTITION BY DEPTNO ORDER BY SAL DESC
        ROWS UNBOUNDED PRECEDING) as DEPT_RICH
FROM EMP;
RANGE UNBOUNDED PRECEDING :
현재 행을 기준으로 파티션 내의 첫 번째 행까지의 범위를 지정한다.
```

[실행 결과]

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

14개의 행이 선택되었다.

실행 결과를 보면 같은 부서 내에 최고 급여를 받는 사람이 둘 있는 경우, 즉, * 표시가 있는 부서번호 20의 SCOTT과 FORD 중에서 어느 사람이 최고 급여자로 선택될지는 위의 SQL 문만 가지고는 판단할 수 없다. FIRST_VALUE는 다른 함수와 달리 공동 등수를 인정하지 않고 처음 나온 행만을 처리한다. 위처럼 공동 등수가 있을 경우에 의도적으로 세부 항목을 정렬하고 싶다면 별도의 정렬 조건을 가진 INLINE VIEW를 사용하거나, OVER () 내의 ORDER BY 절에 칼럼을 추가해야 한다.

[예제] 앞의 SQL 문장에서 같은 값을 가진 FIRST_VALUE를 처리하기 위해 ORDER BY 정렬 조건을 추가한다.

```
[예제]
SELECT DEPTNO, ENAME, SAL,
```

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

[실행 결과]

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

14개의 행이 선택되었다.

SQL에서 같은 부서 내에 최고 급여를 받는 사람이 둘 있는 경우를 대비해서 이름을 두 번째 정렬 조건으로 추가한다. 실행 결과를 확인하면 부서번호 20의 최고 급여자가 이전의 SCOTT 값에서 ASCII 코드가 적은 값인 FORD로 변경 된 것을 확인할 수 있다.

나. LAST_VALUE 함수

LAST_VALUE 함수를 이용해 파티션별 윈도우에서 가장 나중에 나온 값을 구한다. SQL Server에서는 지원하지 않는 함수이다. MAX 함수를 활용하여 같은 결과를 얻을 수도 있다.

[예제] 부서별 직원들을 연봉이 높은 순서부터 정렬하고, 파티션 내에서 가장 마지막에 나온 값을 출력한다.

```
[예제]
SELECT DEPTNO, ENAME, SAL,
        LAST_VALUE(ENAME) OVER (PARTITION BY DEPTNO ORDER BY SAL DESC
        ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) as DEPT_POOR
```

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

[실행 결과]

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

14개의 행이 선택되었다.

실행 결과에서 LAST_VALUE는 다른 함수와 달리 공동 등수를 인정하지 않고 가장 나중에 나온 행만을 처리한다. 만일 공동 등수가 있을 경우를 의도적으로 정렬하고 싶다면 별도의 정렬 조건을 가진 INLINE VIEW를 사용하거나, OVER () 내의 ORDER BY 조건에 칼럼을 추가해야 한다.

다. LAG 함수

LAG 함수를 이용해 파티션별 윈도우에서 이전 몇 번째 행의 값을 가져올 수 있다. SQL Server에서는 지원하지 않는 함수이다.

[예제] 직원들을 입사일자가 빠른 기준으로 정렬을 하고, 본인보다 입사일자가 한 명 앞선 사원의 급여를 본인의 급여와 함께 출력한다.

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

```
[실행 결과]
ENAME        HIREDATE          SAL                    MGR_MIN
------       ---------         ---                    -------
ALLEN        1981-02-20        1600
WARD         1981-02-22        1250                      1600
TURNER       1981-09-08        1500                      1250
MARTIN       1981-09-28        1250                      1500
4개의 행이 선택되었다.
```

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

```
[예제]
SELECT ENAME, HIREDATE, SAL,
        LAG(SAL, 2, 0) OVER (ORDER BY HIREDATE) as PREV_SAL
FROM EMP
WHERE JOB = 'SALESMAN'
LAG(SAL, 2, 0)의 기능은 두 행 앞의 SALARY를 가져오고, 가져올 값이 없는 경우는 0으로 처리한다.
```

```
[실행 결과]
ENAME        HIREDATE          SAL                    PREV_SAL
------       ---------         ---                    --------
ALLEN        1981-02-20        1600                          0
WARD         1981-02-22        1250                          0
TURNER       1981-09-08        1500                       1600
MARTIN       1981-09-28        1250                       1250
4개의 행이 선택되었다.
```

라. LEAD 함수

LEAD 함수를 이용해 파티션별 윈도우에서 이후 몇 번째 행의 값을 가져올 수 있다. 참고로 SQL Server에서는 지원하지 않는 함수이다.

[예제] 직원들을 입사일자가 빠른 기준으로 정렬을 하고, 바로 다음에 입사한 인력의 입사일자를 함께 출력한다.

```
[예제]
SELECT ENAME, HIREDATE,
        LEAD (HIREDATE, 1) OVER (ORDER BY HIREDATE) as "NEXTHIRED"
FROM EMP;
```

```
[실행 결과]
ENAME                HIREDATE                NEXTDATE
------               ---------               ---------
ALLEN                1981-02-20              1981-02-22
WARD                 1981-02-22              1981-04-02
TURNER               1981-09-08              1981-09-28
MARTIN               1981-09-28
4개의 행이 선택되었다.
```

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

5. 그룹 내 비율 함수

가. RATIO_TO_REPORT 함수

RATIO_TO_REPORT 함수를 이용해 파티션 내 전체 SUM(칼럼)값에 대한 행별 칼럼 값의 백분율을 소수점으로 구할 수 있다. 결과 값은 >0 &<= 1 의 범위를 가진다. 그리고 개별 RATIO의 합을 구하면 1이 된다. SQL Server에서는 지원하지 않는 함수이다.

[예제] JOB이 SALESMAN인 사원들을 대상으로 전체 급여에서 본인이 차지하는 비율을 출력한다.

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

```
[실행 결과]
ENAME                    SAL          R_R
------                   ----         ----
ALLEN                    1600         0.29          (1600 / 5600)
WARD                     1250         0.22          (1250 / 5600)
TURNER                   1250         0.22          (1250 / 5600)
MARTIN                   1500         0.27          ( 500 / 5600)
4개의 행이 선택되었다.
```

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

```
0.29 + 0.22 + 0.22 + 0.27 = 1
```

나. PERCENT_RANK 함수

PERCENT_RANK 함수를 이용해 파티션별 윈도우에서 제일 먼저 나오는 것을 0으로, 제일 늦게 나오는 것을 1로 하여, 값이 아닌 행의 순서별 백분율을 구한다. 결과 값은 >= 0 &<= 1 의 범위를 가진다. 참고로 SQL Server에서는 지원하지 않는 함수이다.

[예제] 같은 부서 소속 사원들의 집합에서 본인의 급여가 순서상 몇 번째 위치쯤에 있는지 0과 1 사이의 값으로 출력한다.

```
[예제]
SELECT DEPTNO, ENAME, SAL,
        PERCENT_RANK() OVER (PARTITION BY DEPTNO ORDER BY SAL DESC) as P_R
FROM EMP;
```

```
[실행 결과]
DEPTNO      ENAME          SAL        P_R
--------    ------         ---        ---
10          KING           5000         0
10          CLARK          2450       0.5
10          MILLER         1300         1
20          SCOTT          3000         0
20          FORD           3000         0
20          JONES          2975       0.5
```

20	ADAMS	1100	0.75
20	SMITH	800	1
30	BLAKE	2850	0
30	ALLEN	1600	0.2
30	TURKER	1500	0.4
30	MARTIN	1250	0.6
30	WARD	1250	0.6
30	JAMES	950	1

14개의 행이 선택되었다.

DEPTNO 10의 경우 3건이므로 구간은 2개가 된다. 0과 1 사이를 2개의 구간으로 나누면 0, 0.5, 1이 된다. DEPTNO 20의 경우 5건이므로 구간은 4개가 된다. 0과 1 사이를 4개의 구간으로 나누면 0, 0.25, 0.5, 0.75, 1이 된다. DEPTNO 30의 경우 6건이므로 구간은 5개가 된다. 0과 1 사이를 5개의 구간으로 나누면 0, 0.2, 0.4, 0.6, 0.8, 1이 된다. SCOTT, FORD와 WARD, MARTIN의 경우 ORDER BY SAL DESC 구문에 의해 급여가 같으므로 같은 ORDER로 취급한다.

다. CUME_DIST 함수

CUME_DIST 함수를 이용해 파티션별 윈도우의 전체건수에서 현재 행보다 작거나 같은 건수에 대한 누적백분율을 구한다. 결과 값은 >0 &<= 1 의 범위를 가진다. 참고로 SQL Server에서는 지원하지 않는 함수이다.

[예제] 같은 부서 소속 사원들의 집합에서 본인의 급여가 누적 순서상 몇 번째 위치쯤에 있는지 0과 1 사이의 값으로 출력한다.

```
[예제]
SELECT DEPTNO, ENAME, SAL,
CUME_DIST() OVER (PARTITION BY DEPTNO ORDER BY SAL DESC) as CUME_DIST
FROM EMP;
```

[실행 결과]

DEPTNO	ENAME	SAL	CUME_DIST
10	KING	5000	0.3333
10	CLARK	2450	0.6667
10	MILLER	1300	1.0000
20	SCOTT	3000	0.4000

20	FORD	3000	0.4000
20	JONES	2975	0.6000
20	ADAMS	1100	0.8000
20	SMITH	800	1.0000
30	BLAKE	2850	0.1667
30	ALLEN	1600	0.3333
30	TURKER	1500	0.5000
30	MARTIN	1250	0.8333
30	WARD	1250	0.8333
30	JAMES	950	1.0000

14개의 행이 선택되었다.

DEPTNO가 10인 경우 윈도우가 전체 3건이므로 0.3333 단위의 간격을 가진다. 즉, 0.3333, 0.6667, 1의 값이 된다. DEPTNO가 20인 경우 윈도우가 전체 5건이므로 0.2000 단위의 간격을 가진다. 즉, 0.2000, 0.4000, 0.6000, 0.8000, 1의 값이 된다. DEPTNO가 30인 경우 윈도우가 전체 6건이므로 0.1667 단위의 간격을 가진다. 즉, 0.1667, 0.3333, 0.5000, 0.6667, 0.8333, 1의 값이 된다. * 표시가 있는 SCOTT, FORD와 ** 표시가 있는 WARD, MARTIN의 경우 ORDER BY SAL에 의해 SAL 이 같으므로 같은 ORDER로 취급한다. 다른 WINDOW 함수의 경우 동일 순서면 앞 행의 함수 결과 값을 따르는데, CUME_DIST의 경우는 동일 순서면 뒤 행의 함수 결과값을 기준으로 한다.

라. NTILE 함수

NTILE 함수를 이용해 파티션별 전체 건수를 ARGUMENT 값으로 N 등분한 결과를 구할 수 있다.

[예제] 전체 사원을 급여가 높은 순서로 정렬하고, 급여를 기준으로 4개의 그룹으로 분류한다.

```
[예제]
SELECT ENAME, SAL, NTILE(4) OVER (ORDER BY SAL DESC) as QUAR_TILE
FROM EMP
```

[실행 결과]

DEPTNO	ENAME	SAL	QUAR_TILE
10	KING	5000	1
10	FORD	3000	1
10	SCOTT	3000	1
20	JONES	2975	1
20	BLAKE	2850	2

20	CLARK	2450	2
20	ALLEN	1600	2
20	TURNER	1500	2
30	MILLER	1300	3
30	WARD	1250	3
30	MARTIN	1250	3
30	ADAMS	1100	4
30	JAMES	950	4
30	SMITH	800	4

14개의 행이 선택되었다.

위 예제에서 NTILE(4)의 의미는 14명의 팀원을 4개 조로 나눈다는 의미이다. 전체 14명을 4개의 집합으로 나누면 몫이 3명, 나머지가 2명이 된다. 나머지 두 명은 앞의 조부터 할당한다. 즉, 4명 + 4명 + 3명 + 3명으로 조를 나누게 된다.

제7절 DCL(data control language)

1. DCL 개요

지금까지 살펴본 SQL 문장을 분류하면 테이블 생성과 조작에 관련된 명령어(DDL)와, 데이터를 조작하기 위한 명령어(DML), 그리고 TRANSACTION을 제어하기 위한 명령어(TCL)이다. 추가로, 이런 명령어들 이외에도 유저를 생성하고 권한을 제어할 수 있는 DCL(DATA CONTROL LANGUAGE) 명령어가 있다.

2. 유저와 권한

다른 부서 간에 또는 다른 회사 간에 데이터를 공유하기 위해 데이터베이스를 오픈해야 하는 경우가 가끔 발생한다. 물론 데이터베이스를 오픈하는 것 자체가 문제가 될 수 있다. 즉, 운영 시스템에서 사용하던 유저를 오픈하면 데이터의 손실 우려가 너무 커지게 되는 것이다. 이런 경우에 새로운 유저를 생성하고, 생성한 유저에게 공유할 테이블이나 기타 오브젝트에 대한 접근 권한만을 부여한다면 문제는 쉽게 해결할 수 있다. 일반적으로 회원제 웹사이트를 방문하여 서비스를 이용하려면 먼저 회원 가입을 해야 한다. 유저 아이디, 패스워드, 기타 개인정보를 입력하고 약관에 동의하면 회원 가입이 된다. 그리고 유저 아이디와 패스워드로 로그인하면 웹사이트의 서비스를 이용할 수 있게 된다. 그러나 영화나 유료 게임과 같은 특정 컨텐츠를 이용하려면 '권한이 없다'라는 메시지를 볼 수 있다. 여기서 유저 아이디와 패스워드를 유저라 할 수 있고, 유료 서비스에 대한 결재 여부를 권한이라 할 수 있다. 대부분의 데이터베이스는 데이터 보호와 보안을 위해서 유저와 권한을 관리하고 있는데, 예를 들어 Oracle을 설치하면 기본적으로 제공되는 유저들인 SYS, SYSTEM, SCOTT 유저에 대해서 [표 Ⅱ-2-8]을 통해서 간단하게 알아본다.

[표 Ⅱ-2-8] Oracle에서 제공하는 유저들

유저	역할
SCOTT	Oracle 테스트용 샘플 유저 Default 패스워드 : TIGER
SYS	DBA ROLE을 부여받은 유저
SYSTEM	데이터베이스의 모든 시스템 권한을 부여받은 DBA 유저 Oracle 설치 완료 시에 패스워드 설정

Oracle과 SQL Server의 사용자에 대한 아키텍처는 다른 면이 많다. Oracle은 유저를 통해 데이터베이스에 접속을 하는 형태이다. 즉, 아이디와 비밀번호 방식으로 인스턴스에 접속을 하고 그에 해당하는 스키마에 오브젝트 생성 등의 권한을 부여받게 된다. SQL Server는 인스턴스에 접속하기 위해 로그인이라는 것을 생성하게 되며, 인스턴스 내에 존재하는 다수의 데이터베이스에 연결하여 작업하기 위해 유저를 생성한 후 로그인과 유저를 매핑해 주어야 한다. 더 나아가 특정 유저는 특정 데이터베이스 내의 특정 스키마에 대해 권한을 부여받을 수 있다. SQL Server 로그인은 두 가지 방식으로 가능하다. 첫 번째, Windows 인증 방식으로 Windows에 로그인한 정보를 가지고 SQL Server에 접속하는 방식이다. Microsoft Windows 사용자 계정을 통해 연결되면 SQL Server는 운영 체제의 Windows

보안 주체 토큰을 사용하여 계정 이름과 암호가 유효한지 확인한다. 즉, Windows에서 사용자 ID를 확인한다. SQL Server는 암호를 요청하지 않으며 ID의 유효성 검사를 수행하지 않는다. Windows 인증은 기본 인증 모드이며 SQL Server 인증보다 훨씬 더 안전하다. Windows 인증은 Kerberos 보안 프로토콜을 사용하고, 암호 정책을 적용하여 강력한 암호에 대해 적합한 복잡성 수준을 유지하도록 하며, 계정 잠금 및 암호 만료를 지원한다. SQL Server가 Windows에서 제공하는 자격 증명을 신뢰하므로 Windows 인증을 사용한 연결을 트러스트된 연결이라고도 한다. 두 번째, 혼합 모드(Windows 인증 또는 SQL 인증) 방식으로 기본적으로 Windows 인증으로도 SQL Server에 접속 가능하며, Oracle의 인증과 같은 방식으로 사용자 아이디와 비밀번호로 SQL Server에 접속하는 방식이다. SQL 인증을 사용할 때는 강력한 암호(숫자+문자+특수문자 등을 혼합하여 사용)를 사용해야 한다. 예를 들어, 아래 [그림 Ⅱ-1-16]을 보면 SCOTT이라는 LOGIN 이름으로 인스턴스 INST1에 접속을 하여 미리 매핑되어 있는 SCOTT이라는 유저를 통해 PRODUCT라는 스키마에 속해 있는 ITEM이라는 테이블의 데이터를 액세스하고 있다.

[그림 Ⅱ-2-16] SQL Server 접속 경로

가. 유저 생성과 시스템 권한 부여

유저를 생성하고 데이터베이스에 접속한다. 하지만 데이터베이스에 접속했다고 해서 테이블, 뷰, 인덱스 등과 같은 오브젝트(OBJECT)를 생성할 수는 없다. 사용자가 실행하는 모든 DDL 문장(CREATE, ALTER, DROP, RENAME 등)은 그에 해당하는 적절한 권한이 있어야만 문장을 실행할 수 있다. 이러한 권한을 시스템 권한이라고 하며 약 100개 이상의 종류가 있다. 일반적으로 시스템 권한은 일일이 유저에게 부여되지 않는다. 100개 이상의 시스템 권한을 일일이 사용자에게 설정하는 것은 너무 복잡하고, 특히 유저로부터 권한을 관리하기가 어렵기 때문이다. 그래서 롤(ROLE)을 이용하여 간편하고 쉽게 권한을 부여하게 된다. 롤에 대한 자세한 설명은 차후에 하도록 하고 먼저 유저를 생성하고 권한을 부여한다. 새로운 유저를 생성하려면 일단 유저 생성 권한(CREATE USER)이 있어야 한다.

[예제] SCOTT 유저로 접속한 다음 PJS 유저(패스워드: KOREA7)를 생성해 본다.

```
[예제] Oracle
CONN SCOTT/TIGER
연결되었다.
```

```
CREATE USER PJS IDENTIFIED BY KOREA7;
CREATE USER PJS IDENTIFIED BY KOREA7;

 * 1행에 오류: ERROR: 권한이 불충분하다
```

현재 SCOTT 유저는 유저를 생성할 권한을 부여받지 못했기 때문에 권한이 불충분하다는 오류가 발생한다. Oracle의 DBA 권한을 가지고 있는 SYSTEM 유저로 접속하면 유저 생성 권한(CREATE USER)을 다른 유저에게 부여할 수 있다.

[예제] SCOTT 유저에게 유저생성 권한(CREATE USER)을 부여한 후 다시 PJS 유저를 생성한다.

```
[예제 및 실행 결과] Oracle
GRANT CREATE USER TO SCOTT;
권한이 부여되었다.

CONN SCOTT/TIGER
연결되었다.

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

SQL Server는 유저를 생성하기 전 먼저 로그인을 생성해야 한다. 로그인을 생성할 수 있는 권한을 가진 로그인은 기본적으로 sa이다.

[예제] sa로 로그인을 한 후 SQL 인증을 사용하는 PJS라는 로그인(패스워드: KOREA7)을 생성해 본다. 로그인 후 최초로 접속할 데이터베이스는 AdventureWorks 데이터베이스로 설정한다.

```
[예제 및 실행 결과] SQL Server
CREATE LOGIN PJS WITH PASSWORD='KOREA7', DEFAULT_DATABASE=AdventureWorks
```

SQL Server에서의 유저는 데이터베이스마다 존재한다. 그러므로 유저를 생성하기 위해서는 생성하고자 하는 유저가 속할 데이터베이스로 이동을 한 후 처리해야 한다.

```
[예제 및 실행 결과] SQL Server
USE ADVENTUREWORKS;
```

```
GO
CREATE USER PJS FOR LOGIN PJS WITH DEFAULT_SCHEMA = dbo;
```

[예제] 생성된 PJS 유저로 로그인한다.

```
[예제 및 실행 결과] Oracle
CONN PJS/KOREA7;
오류:
ERROR: 사용자 PJS는 CREATE SESSION 권한을 가지고 있지 않음; 로그온이 거절되었다.
```

PJS 유저가 생성됐지만 아무런 권한도 부여받지 못했기 때문에 로그인을 하면 CREATE SESSION 권한이 없다는 오류가 발생한다. 유저가 로그인을 하려면 CREATE SESSION 권한을 부여받아야 한다.

[예제] PJS 유저가 로그인할 수 있도록 CREATE SESSION 권한을 부여한다.

```
[예제 및 실행 결과] Oracle
CONN SCOTT/TIGER
연결되었다.

GRANT CREATE SESSION TO PJS;
권한이 부여되었다.

CONN PJS/KOREA7
연결되었다.
```

[예제] PJS 유저로 테이블을 생성한다.

```
[예제 및 실행 결과] Oracle
SELECT * FROM TAB;
선택된 레코드가 없다.

CREATE TABLE MENU (
  MENU_SEQ NUMBER NOT NULL,
  TITLE VARCHAR2(10) );

CREATE TABLE MENU ( *
1행에 오류:
ERROR: 권한이 불충분하다.
```

[예제 및 실행 결과] SQL Server
```
CREATE TABLE MENU (
  MENU_SEQ INT NOT NULL,
  TITLE VARCHAR(10) );
```
데이터베이스 'AdventureWorks'에서 CREATE TABLE 사용 권한이 거부되었다.

PJS 유저는 로그인 권한만 부여되었기 때문에 테이블을 생성하려면 테이블 생성 권한(CREATE TABLE)이 불충분하다는 오류가 발생한다.(Oracle, SQL Server)

[예제] SYSTEM 유저를 통하여 PJS 유저에게 CREATE TABLE 권한을 부여한 후 다시 테이블을 생성한다.

[예제 및 실행 결과] Oracle
```
CONN SYSTEM/MANAGER
연결되었다.

GRANT CREATE TABLE TO PJS;
권한이 부여되었다.

CONN PJS/KOREA7
연결되었다.

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

[예제 및 실행 결과] SQL Server
```
GRANT CREATE TABLE TO PJS;
권한이 부여되었다.

스키마에 권한을 부여한다.
GRANT Control ON SCHEMA::dbo TO PJS
권한이 부여되었다.

PJS로 로그인한다.
CREATE TABLE MENU (
  MENU_SEQ INT NOT NULL,
  TITLE VARCHAR(10) );
```

테이블이 생성되었다.

나. OBJECT에 대한 권한 부여

앞에서 PJS 유저를 생성하여 로그인하고 테이블을 만드는 과정에서 몇 가지의 권한에 대해서 살펴보았다. 이제는 특정 유저가 소유한 객체(OBJECT) 권한에 대해 알아본다. 오브젝트 권한은 특정 오브젝트인 테이블, 뷰 등에 대한 SELECT, INSERT, DELETE, UPDATE 작업 명령어를 의미한다. [표 Ⅱ-2-9]는 오브젝트 권한과 오브젝트와의 관계를 보여 주고 있다.

[표 Ⅱ-2-9] 오브젝트 권한과 오브젝트와의 관계 (Oracle 사례)

객체 권한	테이블	VIEWS	SEQUENCE	PROCEDURE
ALTER	○		○	
DELETE	○	○		
EXECUTE				○
INDEX	○			
INSERT	○	○		
REFERENCES	○			
SELECT	○	○	○	
UPDATE	○	○		

[표 Ⅱ-2-10] 오브젝트 권한과 오브젝트와의 관계 (SQL Server 사례)

객체 권한	테이블	VIEWS	FUNCTION	PROCEDURE
ALTER	○		○	
DELETE	○	○	○	
EXECUTE				○
INDEX	○			
INSERT	○	○		
REFERENCES	○			
SELECT	○	○	○	
UPDATE	○	○		

앞에서 PJS 유저가 생성한 MENU 테이블을 SCOTT 유저를 통해서 조회하면 어떻게 될까? SCOTT, PJS 뿐만 아니라 모든 유저는 각각 자신이 생성한 테이블 외에 다른 유저의 테이블에 접근하려면 해당 테이블에 대한 오브젝트 권한을 소유자로부터 부여받아야 한다. 우리가 남의 집에 방문했을 때 집주인의 허락 없이는 집에 들어갈 수 없는 것과 같은 이치이다. SQL Server도 같은 방식으로 동작한다. 한 가지 다른 점은 위에서 언급했듯이 유저는 단지 스키마에 대한 권한만을 가진다. 다시 말하면 테이블과 같은 오브젝트는 유저가 소유하는 것이 아니고 스키마가 소유를 하게 되며 유저는 스키마에 대해 특정한 권한을 가지는 것이다. 먼저 SCOTT 유저로 접속하여 PJS.MENU 테이블을 조회한다. 다른 유저가 소유한 객체에 접근하기 위해서는 객체 앞에 객체를 소유한 유저의 이름을 붙여서 접근해야 한다. SQL Server는 객체 앞에 소유한 유저의 이름을 붙이는 것이 아니고 객체가 속한 스키마

이름을 붙여야 한다.

```
[예제 및 실행 결과] Oracle
CONN SCOTT/TIGER
연결되었다.

SELECT * FROM PJS.MENU;

SELECT * FROM PJS.MENU *
1행에 오류:
ERROR: 테이블 또는 뷰가 존재하지 않는다.
```

```
[예제 및 실행 결과] SQL Server
SCOTT로 로그인한다.

SELECT * FROM dbo.MENU;

개체이름 'dbo.MENU'이(가) 잘못되었다.
```

SCOTT 유저는 PJS 유저로부터 MENU 테이블을 SELECT할 수 있는 권한을 부여받지 못했기 때문에 MENU 테이블을 조회할 수 없다.

[예제] PJS 유저로 접속하여 SCOTT 유저에게 MENU 테이블을 SELECT 할 수 있는 권한을 부여한다.

```
[예제 및 실행 결과] Oracle
CONN PJS/KOREA7
연결되었다.

INSERT INTO MENU VALUES (1, '화이팅');
1개의 행이 만들어졌다.

COMMIT;
커밋이 완료되었다.

GRANT SELECT ON MENU TO SCOTT;
권한이 부여되었다.
```

```
[예제 및 실행 결과] SQL Server
PJS로 로그인한다.
```

```
INSERT INTO MENU VALUES (1, '화이팅');
1개의 행이 만들어졌다.

GRANT SELECT ON MENU TO SCOTT;
권한이 부여되었다.
```

다시 한 번 SCOTT 유저로 접속하여 PJS.MENU 테이블을 조회한다. 이제 PJS.MENU 테이블을 SELECT하면 테이블 자료를 볼 수 있다. SCOTT 유저는 PJS.MENU 테이블을 SELECT하는 권한만 부여 받았기 때문에 UPDATE, INSERT, DELETE와 같은 다른 작업을 할 수 없다. 오브젝트 권한은 SELECT, INSERT, DELETE, UPDATE 등의 권한을 따로따로 관리한다.

[예제] PJS.MENU 테이블에 UPDATE를 시도한다.

[예제 및 실행 결과] Oracle

```
CONN SCOTT/TIGER
연결되었다.

SELECT *
FROM PJS.MENU;

MENU_SEQ TITLE
-------- --------
       1 파이팅

UPDATE PJS.MENU
SET TITLE = '코리아'
WHERE MENU_SEQ = 1;
UPDATE PJS.MENU * 1행에 오류:
ERROR: 권한이 불충분하다
```

[예제 및 실행 결과] SQL Server

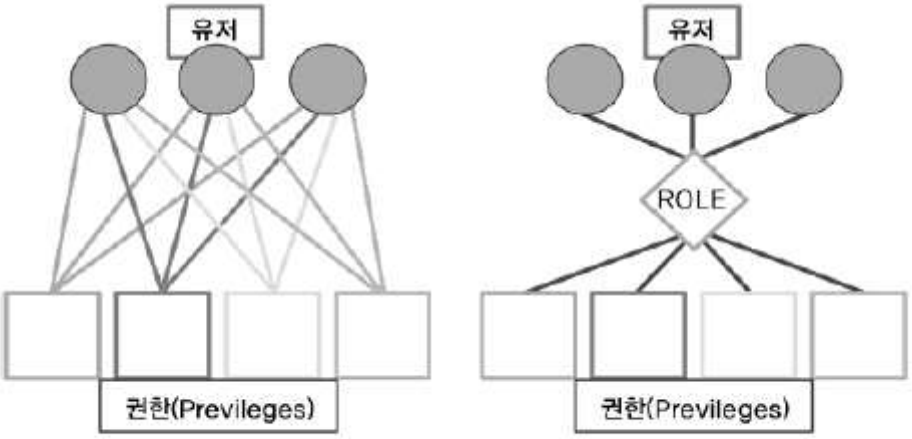
```
SCOTT으로 로그인한다.
SELECT *
FROM PJS.MENU;

MENU_SEQ TITLE
-------- ----------
       1 파이팅
```

```
UPDATE PJS.MENU
SET TITLE = '코리아'
WHERE MENU_SEQ = 1;

개체 'MENU', 데이터베이스 'AdventureWorks', 스키마 'dbo'에 대한 UPDATE 권한이
거부되었다.
```

권한이 부족하여 UPDATE를 할 수 없다는 오류가 나타난다. PJS 유저에게 UPDATE 권한을 부여한 후 다시 시도하면 업데이트가 가능하다.

3. Role을 이용한 권한 부여

유저를 생성하면 기본적으로 CREATE SESSION, CREATE TABLE, CREATE PROCEDURE 등 많은 권한을 부여해야 한다. 데이터베이스 관리자는 유저가 생성될 때마다 각각의 권한들을 유저에게 부여하는 작업을 수행해야 하며 간혹 권한을 빠뜨릴 수도 있으므로 각 유저별로 어떤 권한이 부여되었는지를 관리해야 한다. 하지만 관리해야 할 유저가 점점 늘어나고 자주 변경되는 상황에서는 매우 번거로운 작업이 될 것이다. 이와 같은 문제를 줄이기 위하여 많은 데이터베이스에서 유저들과 권한들 사이에서 중개 역할을 하는 ROLE을 제공한다. 데이터베이스 관리자는 ROLE을 생성하고, ROLE에 각종 권한들을 부여한 후 ROLE을 다른 ROLE이나 유저에게 부여할 수 있다. 또한 ROLE에 포함되어 있는 권한들이 필요한 유저에게는 해당 ROLE만을 부여함으로써 빠르고 정확하게 필요한 권한을 부여할 수 있게 된다. [그림 Ⅱ-2-17]에서는 유저들과 권한들 사이 간 ROLE의 역할을 보여 주고 있다. 왼쪽 그림은 권한을 직접 유저에게 할당할 때를 나타내는 것이며, 오른쪽 그림은 ROLE에 권한을 부여한 후 ROLE을 유저들에게 부여하는 것을 나타내고 있다.



[그림 Ⅱ-2-17] ROLE 의 개념

ROLE에는 시스템 권한과 오브젝트 권한을 모두 부여할 수 있으며, ROLE은 유저에게 직접 부여될 수도 있고, 다른 ROLE에 포함하여 유저에게 부여될 수도 있다.

[예제] JISUNG 유저에게 CREATE SESSION과 CREATE TABLE 권한을 가진 ROLE을 생성한 후 ROLE을 이용하여 다시 권한을 할당한다. 권한을 취소할 때는 REVOKE를 사용한다.

[예제 및 실행 결과] Oracle

```
CONN SYSTEM/MANAGER
연결되었다.

REVOKE CREATE SESSION, CREATE TABLE FROM JISUNG;
권한이 취소되었다.

CONN JISUNG/KOREA7
ERROR: 사용자 JISUNG은 CREATE SESSION 권한을 가지고 있지 않음. 로그온이 거절되었다.
```

[예제 및 실행 결과] SQL Server

```
sa로 로그인한다.

REVOKE CREATE TABLE FROM PJS;
권한이 취소되었다.

PJS로 로그인한다.

CREATE TABLE MENU (
  MENU_SEQ INT NOT NULL,
  TITLE VARCHAR(10) );
데이터베이스 'AdventureWorks'에서 CREATE TABLE사용 권한이 거부되었다.
```

[예제] 이제 LOGIN_TABLE이라는 ROLE을 만들고, 이 ROLE을 이용하여 JISUNG 유저에게 권한을 부여한다.

[예제 및 실행 결과] Oracle

```
CONN SYSTEM/MANAGER
연결되었다.

CREATE ROLE LOGIN_TABLE;
롤이 생성되었다.

GRANT CREATE SESSION, CREATE TABLE TO LOGIN_TABLE;
권한이 부여되었다.

GRANT LOGIN_TABLE TO JISUNG;
권한이 부여되었다.

CONN JISUNG/KOREA7
연결되었다.

CREATE TABLE MENU2(
  MENU_SEQ NUMBER NOT NULL,
  TITLE VARCHAR2(10));
테이블이 생성되었다.
```

이와 같이 ROLE을 만들어 사용하는 것이 권한을 직접 부여하는 것보다 빠르고 안전하게 유저를 관리할 수 있는 방법이다. Oracle에서는 기본적으로 몇 가지 ROLE을 제공하고 있다. 그 중 가장 많이 사용하는 ROLE은 CONNECT와 RESOURCE이다. 참조를 위해 [표 Ⅱ-2-11]은 CONNECT와 RESOURCE ROLE에 부여된 권한 목록을 정리한 것이다. CONNECT는 CREATE SESSION과 같은 로그인 권한이 포함되어 있고, RESOURCE는 CREATE TABLE과 같은 오브젝트의 생성 권한이 포함되어 있다. 일반적으로 유저를 생성할 때 CONNECT와 RESOURCE ROLE을 사용하여 기본 권한을 부여한다.

[표 Ⅱ-2-11] CONNECT ROLE과 RESOURCE ROLE에 포함된 권한 목록 (Oracle 사례)

CONNECT	RESOURCE
ALTER SESSION	CREATE CLUSTER
CREATE CLUSTER	CREATE INDEXTYPE
CREATE DATABASE LINK	CREATE OPERATOR
CREATE MENU_SEQUENCE	CREATE PROCEDURE
CREATE SESSION	CREATE MENU_SEQUENCE
CREATE SYNONYM	CREATE TABLE
CREATE TABLE	CREATE TRIGGER
CREATE VIEW	CREATE

유저를 삭제하는 명령어는 DROP USER이고, CASCADE 옵션을 주면 해당 유저가 생성한 오브젝트를 먼저 삭제한 후 유저를 삭제한다.

[예제] 앞에서 MENU라는 테이블을 생성했기 때문에 CASCADE 옵션을 사용하여 JISUNG 유저를 삭제한 후, 유저 재생성 및 기본적인 ROLE을 부여한다.

[예제 및 실행 결과] Oracle

```
CONN SYSTEM/MANAGER
연결되었다.

DROP USER JISUNG CASCADE;
사용자가 삭제되었다.
☞ JISUNG 유저가 만든 MENU 테이블도 같이 삭제되었다.

CREATE USER JISUNG IDENTIFIED BY KOREA7;
사용자가 생성되었다.

GRANT CONNECT, RESOURCE TO JISUNG;
권한이 부여되었다.

CONN JISUNG/KOREA7
연결되었다.

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

SQL Server에서는 위와 같이 ROLE을 생성하여 사용하기보다는 기본적으로 제공되는 ROLE에 멤버로 참여하는 방식으로 사용한다. 특정 로그인이 멤버로 참여할 수 있는 서버 수준 역할(ROLE)은 [표 Ⅱ-2-12]와 같다.

[표 Ⅱ-2-12] 서버 수준 역할명 (SQL Server 사례)

서버 수준 역할명	설명
public	모든 SQL Server 로그인은 PUBLIC 서버 역할에 속한다. 서버 보안 주체에게 보안 객체에 대한 특정 사용 권한이 부여되지 않았거나 거부된 경우 사용자는 해당 개체에 대해 PUBLIC 으로 부여된 사용 권한을 상속 받는다. 모든 사용자가 개체를 사용할 수 있도록 하려는 경우에만 개체에 PUBLIC 권한을 할당해야 한다.
bulkadmin	BULK INSERT문을 수행할 수 있다.
dbcreator	데이터베이스를 생성, 변경, 삭제 및 복원할 수 있다.
diskadmin	디스크 파일을 관리하는데 사용된다.
processadmin	SQL Server의 인스턴스에서 실행중인 프로세스를 종료할 수 있다.
securityadmin	로그인 및 해당 속성을 관리한다. 서버 및 데이터베이스 수준의 사용 권한을 부여(GRANT), 거부(DENY), 취소(REVOKE)할 수 있다. 또한, 로그인의 암호를 다시 설정할 수 있다.
serveradmin	서버 차원의 구성 옵션을 변경하고 서버를 종료할 수 있다.
setupadmin	연결된 서버를 추가하거나 제거할 수 있다.
sysadmin	서버에서 모든 작업을 수행할 수 있다. 기본적으로 Windows BUILTIN ₩Administrators 그룹의 멤버인 로컬 관리자 그룹은 sysamdin 고정 서버 역할의 멤버이다.

데이터베이스에 존재하는 유저에 대해서는 아래와 같은 데이터베이스 역할의 멤버로 참여할 수 있다.

[표 Ⅱ-2-13] 데이터베이스 수준 역할명 (SQL Server 사례)

데이터베이스 수준 역할명	설명
db_accessadmin	Windows 로그인, Windows 그룹 및 SQL Server 로그인의 데이터베이스에 대한 액세스를 추가하거나 제거할 수 있다.
db_backupoperator	데이터베이스를 백업할 수 있다.
db_datareader	모든 사용자 테이블의 모든 데이터를 읽을 수 있다.
db_datawriter	모든 사용자 테이블에서 데이터를 추가, 삭제, 변경할 수 있다.
db_ddladmin	데이터베이스에서 모든 DDL 명령을 수행할 수 있다.
db_denydatareader	데이터베이스 내에 있는 사용자 테이블의 데이터를 읽을 수 없다.
db_denydatawriter	데이터베이스 내의 모든 사용자 테이블에 있는 데이터를 추가, 삭제, 변경할 수 없다.
db_owner	데이터베이스 내에 있는 모든 구성 및 유지관리 작업을 수행할 수 있고 데이터베이스를 삭제할 수도 있다.
db_securityadmin	역할 멤버 자격을 수정하고 사용 권한 관리를 할 수 있다. 이 역할에 보안 주체를 추가하면 원하지 않는 권한 상승이 설정될 수 있다.

SQL Server에서는 Oracle과 같이 Role을 자주 사용하지 않는다. 대신 위에서 언급한 서버 수준 역할 및 데이터베이스 수준 역할을 이용하여 로그인 및 사용자 권한을 제어한다. 인스턴스 수준의 작업이 필요한 경우 서버 수준 역할을 부여하고 그보다 작은 개념인 데이터베

이스 수준의 권한이 필요한 경우 데이터베이스 수준의 역할을 부여하면 된다. 즉, 인스턴스 수준을 요구하는 로그인에는 서버 수준 역할을, 데이터베이스 수준을 요구하는 사용자에게는 데이터베이스 수준 역할을 부여한다.

제8절 절차형 SQL

1. 절차형 SQL 개요

일반적인 개발 언어처럼 SQL에도 절차 지향적인 프로그램이 가능하도록 DBMS 벤더별로 PL(Procedural Language)/SQL(Oracle), SQL/PL(DB2), T-SQL(SQL Server) 등의 절차형 SQL을 제공하고 있다. 절차형 SQL을 이용하면 SQL문의 연속적인 실행이나 조건에 따른 분기처리를 이용하여 특정 기능을 수행하는 저장 모듈을 생성할 수 있다. 본 절에서는 절차형 SQL을 이용하여 만들 수 있는 저장 모듈인 Procedure, User Defined Function, Trigger에 대해서 간단하게 살펴본다. (상세한 내역은 각 DBMS 벤더의 매뉴얼을 참조한다.)

2. PL/SQL 개요

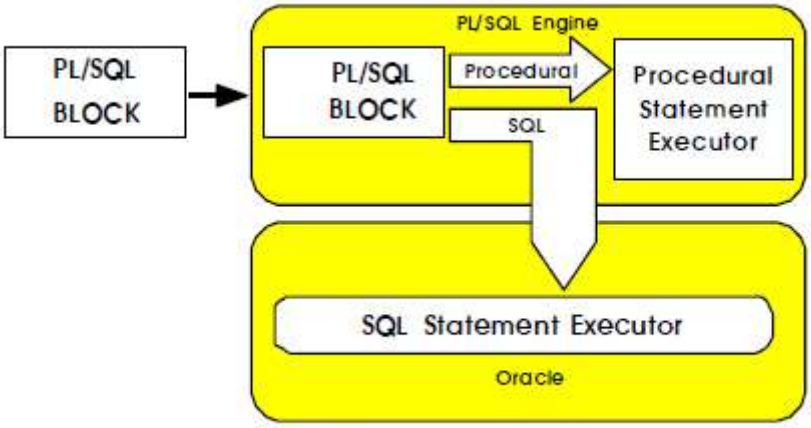
가. PL/SQL 특징

Oracle의 PL/SQL은 Block 구조로 되어있고 Block 내에는 DML 문장과 QUERY 문장, 그리고 절차형 언어(IF, LOOP) 등을 사용할 수 있으며, 절차적 프로그래밍을 가능하게 하는 트랜잭션 언어이다. 이런 PL/SQL을 이용하여 다양한 저장 모듈(Stored Module)을 개발할 수 있다. 저장 모듈이란 PL/SQL 문장을 데이터베이스 서버에 저장하여 사용자와 애플리케이션 사이에서 공유할 수 있도록 만든 일종의 SQL 컴포넌트 프로그램이며, 독립적으로 실행되거나 다른 프로그램으로부터 실행될 수 있는 완전한 실행 프로그램이다. Oracle의 저장 모듈에는 Procedure, User Defined Function, Trigger가 있다.

PL/SQL의 특징은 다음과 같다.

- PL/SQL은 Block 구조로 되어있어 각 기능별로 모듈화가 가능하다. - 변수, 상수 등을 선언하여 SQL 문장 간 값을 교환한다. - IF, LOOP 등의 절차형 언어를 사용하여 절차적인 프로그램이 가능하도록 한다. - DBMS 정의 에러나 사용자 정의 에러를 정의하여 사용할 수 있다. - PL/SQL은 Oracle에 내장되어 있으므로 Oracle과 PL/SQL을 지원하는 어떤 서버로도 프로그램을 옮길 수 있다. - PL/SQL은 응용 프로그램의 성능을 향상시킨다. - PL/SQL은 여러 SQL 문장을 Block으로 묶고 한 번에 Block 전부를 서버로 보내기 때문에 통신량을 줄일 수 있다.

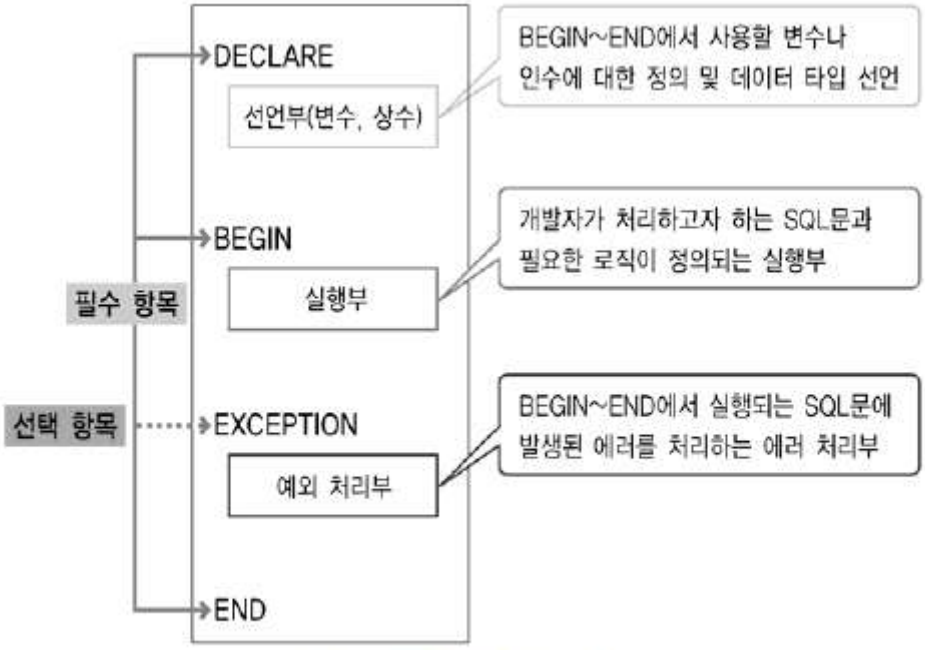
[그림 Ⅱ-2-18]은 PL/SQL Architecture이다. PL/SQL Block 프로그램을 입력받으면 SQL 문장과 프로그램 문장을 구분하여 처리한다. 즉 프로그램 문장은 PL/SQL 엔진이 처리하고 SQL 문장은 Oracle 서버의 SQL Statement Executor가 실행하도록 작업을 분리하여 처리한다.



[그림 Ⅱ-2-18] Oracle의 PL/SQL 엔진

나. PL/SQL 구조

다음은 PL/SQL의 블록 구조를 표현한 내용이다.



[그림 Ⅱ-2-19] PL/SQL 블록 구조

- DECLARE : BEGIN ~ END 절에서 사용될 변수와 인수에 대한 정의 및 데이터 타입을 선언하는 선언부이다. - BEGIN ~ END : 개발자가 처리하고자 하는 SQL문과 여러 가지 비교문, 제어문을 이용하여 필요한 로직을 처리하는 실행부이다. - EXCEPTION : BEGIN ~ END 절에서 실행되는 SQL문이 실행될 때 에러가 발생하면 그 에러를 어떻게 처리할 것이지를 정의하는 예외 처리부이다.

다. PL/SQL 기본 문법(Syntax)

앞으로 살펴볼 User Defined Function이나 Trigger의 생성 방법이나 사용 목적은 다르지만 기본적인 문법은 비슷하기 때문에 여기에서는 Stored Procedure를 통해서 PL/SQL에 대한 기본적인 문법을 정리한다.

```
CREATE [OR REPLACE] Procedure [Procedure_name]
( argument1 [mode] data_type1,
  argument2 [mode] date_type2,
 ... ... )
 IS [AS]
   ... ...
 BEGIN
   ... ...
 EXCEPTION
   ... ...
 END;
 /
```

다음은 생성된 프로시저를 삭제하는 명령어이다.

```
DROP Procedure [Procedure_name];
```

CREATE TABLE 명령어로 테이블을 생성하듯 CREATE 명령어로 데이터베이스 내에 프로시저를 생성할 수 있다. 이렇게 생성한 프로시저는 데이터베이스 내에 저장된다. 프로시저는 개발자가 자주 실행해야 하는 로직을 절차적인 언어를 이용하여 작성한 프로그램 모듈이기 때문에 필요할 때 호출하여 실행할 수 있다. [OR REPLACE] 절은 데이터베이스 내에 같은 이름의 프로시저가 있을 경우, 기존의 프로시저를 무시하고 새로운 내용으로 덮어쓰기 하겠다는 의미이다. Argument는 프로시저가 호출될 때 프로시저 안으로 어떤 값이 들어오거나 혹은 프로시저에서 처리한 결과값을 운영 체제로 리턴시킬 매개 변수를 지정할 때 사용한다. [mode] 부분에 지정할 수 있는 매개 변수의 유형은 3가지가 있다. 먼저 IN은 운영체제에서 프로시저로 전달될 변수의 MODE이고, OUT은 프로시저에서 처리된 결과가 운영체제로 전달되는 MODE이다. 마지막으로 잘 쓰지는 않지만 INOUT MODE가 있는데 이 MODE는 IN과 OUT 두 가지의 기능을 동시에 수행하는 MODE이다. 마지막에 있는 슬래쉬(“/”)는 데이터베이스에게 프로시저를 컴파일하라는 명령어이다. 앞에서 잠깐 언급했지만 PL/SQL과 관련된 내용은 상당히 다양하고 분량이 많기 때문에 본 가이드에서는 간단한 문법과 사용 목적에 초점을 맞춰 이해하기 바란다.

3. T-SQL 개요

가. T-SQL 특징

T-SQL은 근본적으로 SQL Server를 제어하기 위한 언어로서, T-SQL은 엄격히 말하면, MS사에서 ANSI/ISO 표준의 SQL에 약간의 기능을 더 추가해 보완적으로 만든 것이다. T-SQL을 이용하여 다양한 저장 모듈(Stored Module)을 개발할 수 있는데, T-SQL의 프로그래밍 기능은 아래와 같다.

- 변수 선언 기능 @@이라는 전역변수(시스템 함수)와 @이라는 지역변수가 있다.
- 지역변수는 사용자가 자신의 연결 시간 동안만 사용하기 위해 만들어지는 변수이며 전역변수는 이미 SQL서버에 내장된 값이다.
- 데이터 유형(Data Type)을 제공한다. 즉 int, float, varchar 등의 자료형을 의미한다.
- 연산자(Operator) 산술연산자(+, -, *, /)와 비교연산자(=, <, >, <>) 논리연산자(and, or, not) 사용이 가능하다.
- 흐름 제어 기능 IF-ELSE와 WHILE, CASE-THEN 사용이 가능하다.
- 주석 기능

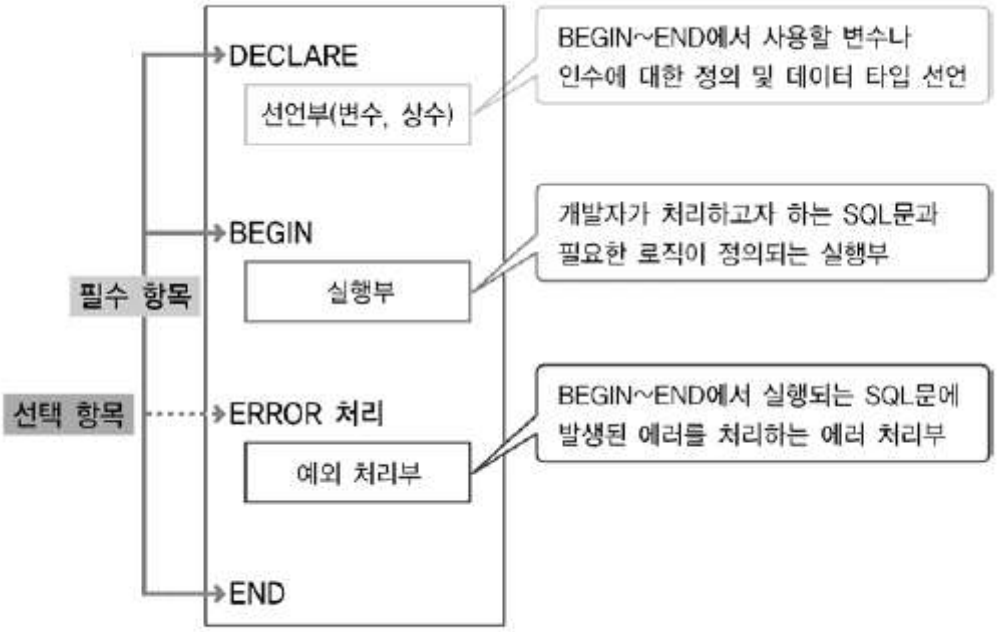
  한줄 주석 : -- 뒤의 내용은 주석

  범위 주석 : /* 내용 */ 형태를 사용하며, 여러 줄도 가능함

T-SQL과 타 DBMS가 제공하는 SQL은 약간만 다를 뿐 그 맥락은 같이 하기 때문에, 조금의 변경 사항만 적용하면 같은 기능을 수행할 수 있다. 그리고 많은 사람들이 SQL 서버에 엔터프라이즈 매니저의 UI를 통하여 접근하는 경우가 많은데, 실??이 더 바람직하다.

나. T-SQL 구조

다음은 T-SQL의 구조를 표현한 내용이다. PL/SQL과 유사하다.



[그림 Ⅱ-2-20] T-SQL 구조

- DECLARE : BEGIN ~ END 절에서 사용될 변수와 인수에 대한 정의 및 데이터 타입을 선언하는 선언부이다.
- BEGIN ~ END : 개발자가 처리하고자 하는 SQL문과 여러 가지 비교문, 제어문을 이용하여 필요한 로직을 처리하는 실행부이다. T-SQL에서는 BEGIN, END 문을 반드시 사용해야하는 것은 아니지만 블록 단위로 처리하고자 할 때는 반드시 작성해야 한다.
- ERROR 처리 : BEGIN ~ END 절에서 실행되는 SQL문이 실행될 때 에러가 발생하면 그 에러를 어떻게 처리할 것이지를 정의하는 예외 처리부이다.

다. T-SQL 기본 문법(Syntax)

앞으로 살펴볼 User Defined Function이나 Trigger의 생성 방법과 사용 목적은 Stored Procedure와 다르지만 기본적인 문법은 비슷하기 때문에 여기에서는 Stored Procedure를 통해서 T-SQL에 대한 기본적인 문법을 정리한다.

```
CREATE Procedure [schema_name.]Procedure_name
 @parameter1 data_type1 [mode],
 @parameter2 date_type2 [mode],
 ... ...
 WITH
 AS
```

```
 … …
BEGIN
 … …
ERROR 처리
 … …
END;
```

다음은 생성된 프로시저를 삭제하는 명령어이다.

```
DROP Procedure [schema_name.]Procedure_name;
```

CREATE TABLE 명령어로 테이블을 생성하듯 CREATE 명령어로 데이터베이스 내에 프로시저를 생성할 수 있다. 이렇게 생성한 프로시저는 데이터베이스 내에 저장된다. 프로시저는 개발자가 자주 실행해야 하는 로직을 절차적인 언어를 이용하여 작성한 프로그램 모듈이기 때문에 필요할 때 호출하여 실행할 수 있다.

프로시저의 변경이 필요할 경우 Oracle은 [CREATE OR REPLACE]와 같이 하나의 구문으로 처리하지만 SQL Server는 CREATE 구문을 ALTER 구문으로 변경하여야 한다. @parameter는 프로시저가 호출될 때 프로시저 안으로 어떤 값이 들어오거나 혹은 프로시저에서 처리한 결과 값을 리턴 시킬 매개 변수를 지정할 때 사용한다.

[mode] 부분에 지정할 수 있는 매개 변수(@parameter)의 유형은 4가지가 있다.

① VARYING

결과 집합이 출력 매개 변수로 사용되도록 지정합니다. CURSOR 매개변수에만 적용된다.

② DEFAULT

지정된 매개변수가 프로시저를 호출할 당시 지정되지 않을 경우 지정된 기본값으로 처리한다. 즉, 기본 값이 지정되어 있으면 해당 매개 변수를 지정하지 않아도 프로시저가 지정된 기본 값으로 정상적으로 수행이 된다.

③ OUT, OUTPUT

프로시저에서 처리된 결과 값을 EXECUTE 문 호출 시 반환한다.

④ READONLY

자주 사용되지는 않는다. 프로시저 본문 내에서 매개 변수를 업데이트하거나 수정할 수 없음을 나타낸다. 매개 변수 유형이 사용자 정의 테이블 형식인 경우 READONLY를 지정해야 한다.

WITH 부분에 지정할 수 있는 옵션은 3가지가 있다.

① RECOMPILE

데이터베이스 엔진에서 현재 프로시저의 계획을 캐시하지 않고 프로시저가 런타임에 컴파일 된다. 데이터베이스 엔진에서 저장 프로시저 안에 있는 개별 쿼리에 대한 계획을 삭제하려 할 때 RECOMPILE 쿼리 힌트를 사용한다.

② ENCRYPTION

CREATE PROCEDURE 문의 원본 텍스트가 알아보기 어려운 형식으로 변환된다. 변조된 출력은 SQL Server의 카탈로그 뷰 어디에서도 직접 표시되지 않는다. 원본을 볼 수 있는 방법이 없기 때문에 반드시 원본은 백업을 해두어야 한다.

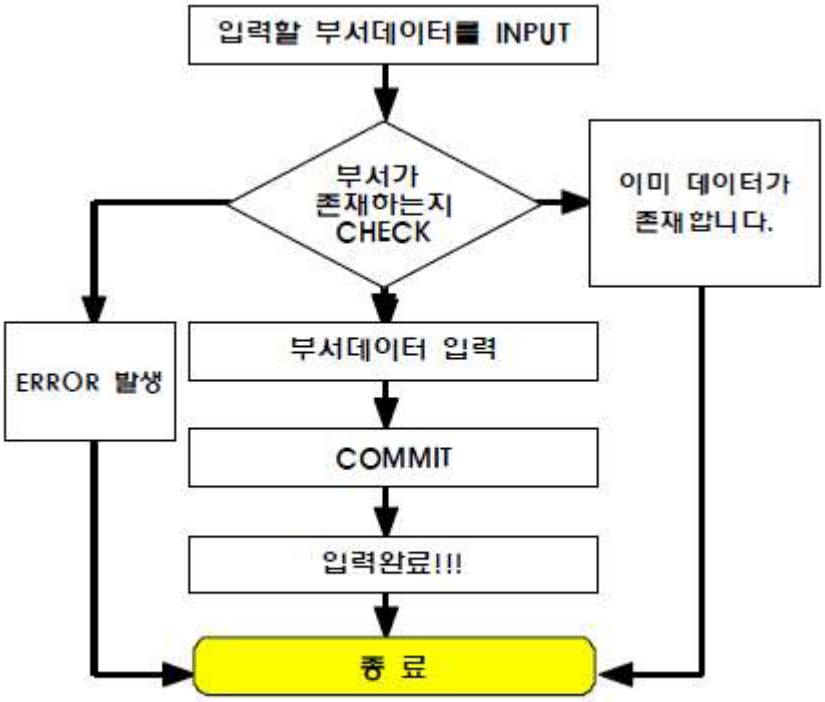
③ EXECUTE AS

해당 저장 프로시저를 실행할 보안 컨텍스트를 지정한다.

앞에서 잠깐 언급했지만 T-SQL과 관련된 내용은 상당히 다양하고 분량이 많기 때문에 본 가이드에서는 간단한 문법과 사용 목적에 초점을 맞춰 이해하기 바란다.

4. Procedure의 생성과 활용

[그림 Ⅱ-2-21]은 앞으로 생성할 Procedure의 기능을 Flow Chart로 나타낸 그림이다.



[그림 Ⅱ-2-21] 부서 입력 FLOW CHART

[예제] SCOTT 유저가 소유하고 있는 DEPT 테이블에 새로운 부서를 등록하는 Procedure를 작성한다. SCOTT 유저가 기본적으로 소유한 DEPT 테이블의 구조는 [표 Ⅱ-2-14]와 같다.

[표 Ⅱ-2-14] DEPT 테이블 구조

DEPT			
칼럼ID	TYPE	길이	인덱스
DEPTNO	NUMBER	2	PK
DNAME	VARCHAR2	14	
LOC	VARCHAR2	13	

[예제] Oracle

```
CREATE OR REPLACE Procedure p_DEPT_insert    -------------①
( v_DEPTNO in   number,
  v_dname    in   varchar2,
  v_loc      in   varchar2,
  v_result   out varchar2)
IS
cnt number := 0;
BEGIN
 SELECT COUNT(*) INTO CNT                    -------------②
 FROM DEPT
 WHERE DEPTNO = v_DEPTNO
   AND ROWNUM = 1;
 if cnt >0 then                              -------------③
   v_result := '이미 등록된 부서번호이다';
 else
    INSERT INTO DEPT (DEPTNO, DNAME, LOC) -------------④
    VALUES (v_DEPTNO, v_dname, v_loc);
    COMMIT;                                  -------------⑤
    v_result := '입력 완료!!';
 end if;
EXCEPTION                                    -------------⑥
    WHEN OTHERS THEN
          ROLLBACK;
          v_result := 'ERROR 발생';
END;
/
```

[예제] SQL Server

```
CREATE Procedure dbo.p_DEPT_insert           -------------①
```

```
@v_DEPTNO int,
@v_dname varchar(30),
@v_loc     varchar(30),
@v_result varchar(100) OUTPUT
AS
DECLARE @cnt int
SET @cnt = 0
BEGIN
 SELECT @cnt=COUNT(*)                          -------------②
 FROM DEPT
 WHERE DEPTNO = @v_DEPTNO
 IF @cnt >0                                    -------------③
 BEGIN
    SET @v_result = '이미 등록된 부서번호이다'
    RETURN
 END
 ELSE
 BEGIN
   BEGIN TRAN
   INSERT INTO DEPT (DEPTNO, DNAME, LOC) -------------④
   VALUES (@v_DEPTNO, @v_dname, @v_loc)
   IF @@ERROR<>0
   BEGIN
        ROLLBACK                               -------------⑥
        SET @v_result = 'ERROR 발생'
        RETURN
   END
   ELSE
   BEGIN
       COMMIT                                  -------------⑤
        SET @v_result = '입력 완료!!'
        RETURN
   END
  END
END
```

DEPT 테이블은 DEPTNO 칼럼이 PRIMARY KEY로 설정되어 있으므로, DEPTNO 칼럼에는 유일한 값을 넣어야만 한다.

[예제]에 대한 설명은 다음과 같다.

① DEPT 테이블에 들어갈 칼럼 값(부서코드, 부서명, 위치)을 입력 받는다.

② 입력 받은 부서코드가 존재하는지 확인한다.

③ 부서코드가 존재하면 '이미 등록된 부서번호입니다'라는 메시지를 출력 값에 넣는다.

④ 부서코드가 존재하지 않으면 입력받은 필드 값으로 새로운 부서 레코드를 입력한다.

⑤ 새로운 부서가 정상적으로 입력됐을 경우에는 COMMIT 명령어를 통해서 트랜잭션을 종료한다.

⑥ 에러가 발생하면 모든 트랜잭션을 취소하고 'ERROR 발생'라는 메시지를 출력값에 넣는다.

앞에 있는 프로시저를 작성하면서 주의해야 할 몇 가지 문법적 요소가 있다.

첫째, PL/SQL 및 T-SQL에서는 다양한 변수가 있다. 예제에서 나온 cnt라는 변수를 SCALAR 변수라고 한다. SCALAR 변수는 사용자의 임시 데이터를 하나만 저장할 수 있는 변수이며 거의 모든 형태의 데이터 유형을 지정할 수 있다.

둘째, PL/SQL에서 사용하는 SQL 구문은 대부분 지금까지 살펴본 것과 동일하게 사용할 수 있지만 SELECT 문장은 다르다. PL/SQL에서 사용하는 SELECT 문장은 결과값이 반드시 있어야 하며, 그 결과 역시 반드시 하나여야 한다. 조회 결과가 없거나 하나 이상인 경우에는 에러를 발생시킨다. T-SQL에서는 결과 값이 없어도 에러가 발생하지 않는다.

셋째, T-SQL을 비롯하여 일반적으로 대입 연산자는 “=”을 사용하지만 PL/SQL에서는 “:=”를 사용한다.

넷째, 에러 처리를 담당하는 EXCEPTION에는 WHEN ~ THEN 절을 사용하여 에러의 종류별로 적절히 처리한다. OTHERS를 이용하여 모든 에러를 처리할 수 있지만 정확하게 에러를 처리하는 것이 좋다. T-SQL에서는 에러 처리를 다양하게 처리할 수 있으며 위의 예제는 그 한 예이다.

다음은 지금까지 작성한 프로시저를 실행하여 기능을 테스트한 과정이다.

[실행 결과] Oracle

```
SQL> SELECT * FROM DEPT;                          -----------------①
     DEPTNO DNAME LOC
------- ------- ---------
     10  ACCOUNTING NEW YORK
     20  RESEARCH DALLAS
     30  SALES CHICAGO
     40 OPERATIONS BOSTON
SQL> variable rslt varchar2(30);                  -----------------②

SQL> EXECUTE p_DEPT_insert(10,'dev','seoul',:rslt);  -----------------③
PL/SQL 처리가 정상적으로 완료되었다.
```

```
SQL> print rslt;                                          ------------------④
RSLT
--------------------------------
이미 등록된 부서번호이다

SQL> EXECUTE p_DEPT_insert(50,'NewDev','seoul',:rslt);  -----------------⑤
PL/SQL 처리가 정상적으로 완료되었다.

SQL> print rslt;                                          -----------------⑥
RSLT
--------------------------------
입력 완료!!

SQL> SELECT * FROM DEPT;                                 -----------------⑦
    DEPTNO DNAME LOC
    ------ -------- ---------
        10 ACCOUNTING NEW YORK
        20 RESEARCH DALLAS
        30 SALES CHICAGO
        40 OPERATIONS BOSTON
        50 NewDev SEOUL
5개의 행이 선택되었다.
```

① DEPT 테이블을 조회하면 총 4개 행의 결과가 출력된다.

② Procedure를 실행한 결과 값을 받을 변수를 선언한다. (BIND 변수)

③ 존재하는 DEPTNO(10)를 가지고 Procedure를 실행한다.

④ DEPTNO가 10인 부서는 이미 존재하기 때문에 변수 rslt를 print해 보면 '이미 등록된 부서번호이다' 라고 출력된다.

⑤ 이번에는 새로운 DEPTNO(50)를 가지고 입력한다.

⑥ rslt를 출력해 보면 '입력 완료!!' 라고 출력된다.

⑦ DEPT 테이블을 조회하여 보면 DEPTNO가 50인 데이터가 정확하게 저장되었음을 확인할 수 있다.

T-SQL로 작성한 프로시저를 실행하기 위해서는 일반적으로 SQL Server에서 제공하는 기본 클라이언트 프로그램인 SQL Server MANAGEMENT STUDIO를 사용한다.

[실행 결과] SQL Server

```
SELECT * FROM DEPT;                          -----------------①
```

```
DEPTNO   DNAME      LOC
-------  ----------  ---------
      10  ACCOUNTING NEW YORK
      20  RESEARCH     DALLAS
      30  SALES         CHICAGO
      40  OPERATIONS BOSTON
DECALRE @v_result VARCHAR(100)              ------------------②
EXECUTE dbo.p_DEPT_insert 10, 'dev', 'seoul',
        @v_result=@v_result OUTPUT           ------------------③
SELECT @v_result AS RSLT                    ------------------④
RSLT
---------------------------------
이미 등록된 부서번호이다
DECALRE @v_result VARCHAR(100)              ------------------⑤
EXECUTE dbo.p_DEPT_insert 50, 'dev', 'seoul',
        @v_result=@v_result OUTPUT          ------------------⑥
SELECT @v_result AS RSLT                    ------------------⑦
RSLT
---------------------------------
입력 완료!

SELECT * FROM DEPT;                          ----------------⑧
   DEPTNO    DNAME     LOC
  ------- --------    ---------
        10 ACCOUNTING NEW YORK
        20 RESEARCH     DALLAS
        30 SALES         CHICAGO
        40 OPERATIONS BOSTON
        50 NewDev       SEOUL
5개의 행에서 선택되었다.
```

① DEPT 테이블을 조회하면 총 4개 행의 결과가 출력된다.

② Procedure를 실행한 결과 값을 받을 변수를 선언한다.

③ 존재하는 DEPTNO(10)를 가지고 Procedure를 실행한다.

④ DEPTNO가 10인 부서는 이미 존재하기 때문에 변수 rslt를 print해 보면 '이미 등록된 부서번호이다'라고 출력된다.

⑤ Procedure를 실행한 결과 값을 받을 변수를 선언한다.

⑥ 이번에는 새로운 DEPTNO(50)를 가지고 입력한다.

⑦ rslt를 출력해 보면 '입력 완료!'라고 출력된다.

⑧ DEPT 테이블을 조회하여 보면 DEPTNO가 50인 데이터가 정확하게 저장되었음을 확인할 수 있다.

5. User Defined Function의 생성과 활용

User Defined Function은 Procedure처럼 절차형 SQL을 로직과 함께 데이터베이스 내에 저장해 놓은 명령문의 집합을 의미한다. 앞에서 학습한 SUM, SUBSTR, NVL 등의 함수는 벤더에서 미리 만들어둔 내장 함수이고, 사용자가 별도의 함수를 만들 수도 있다.

Function이 Procedure와 다른 점은 RETURN을 사용해서 하나의 값을 반드시 되돌려 줘야 한다는 것이다. 즉 Function은 Procedure와는 달리 SQL 문장에서 특정 작업을 수행하고 반드시 수행결과 값을 리턴한다.

[예제] K-리그 8월 경기결과와 두 팀간의 점수차를 ABS 함수를 사용하여 절대값으로 출력한다.

[예제] Oracle

```
SELECT SCHE_DATE 경기일자,
       HOMETEAM_ID || ' - ' || AWAYTEAM_ID 팀들,
       HOME_SCORE || ' - ' || AWAY_SCORE SCORE,
       ABS(HOME_SCORE - AWAY_SCORE) 점수차
FROM SCHEDULE
WHERE GUBUN = 'Y'
AND   SCHE_DATE BETWEEN '20120801' AND '20120831'
ORDER BY SCHE_DATE;
```

[예제] SQL Server

```
SELECT SCHE_DATE 경기일자,
       HOMETEAM_ID + ' - ' + AWAYTEAM_ID AS 팀들,
       HOME_SCORE + ' - ' + AWAY_SCORE AS SCORE,
       ABS(HOME_SCORE - AWAY_SCORE) AS 점수차
FROM SCHEDULE
WHERE GUBUN = 'Y'
AND SCHE_DATE BETWEEN '20120801' AND '20120831'
ORDER BY SCHE_DATE;
```

[실행 결과]

경기일자	팀들	SCORE	점수차
20120803	K01 - K03	3 - 0	3
20120803	K06 - K09	2 - 1	1
20120803	K08 - K07	1 - 0	1
20120804	K05 - K04	2 - 1	1
20120804	K10 - K02	0 - 3	3
20120811	K07 - K10	1 - 1	0
20120811	K03 - K08	2 - 0	2
20120811	K09 - K05	0 - 1	1
20120811	K04 - K02	0 - 2	2
20120811	K01 - K06	0 - 0	0
20120818	K05 - K01	0 - 2	2
20120818	K02 - K09	1 - 2	1
20120818	K08 - K10	3 - 1	2
20120818	K04 - K07	1 - 0	1
20120818	K06 - K03	3 - 1	2
20120824	K02 - K01	1 - 1	0
20120824	K05 - K03	3 - 3	0
20120824	K08 - K06	4 - 3	1
20120825	K10 - K04	1 - 1	0
20120825	K09 - K07	1 - 1	0
20120828	K04 - K08	2 - 3	1
20120828	K09 - K10	2 - 0	2
20120828	K03 - K02	0 - 0	0
20120828	K01 - K07	0 - 1	1
20120828	K06 - K05	1 - 1	0

25개의 행이 선택되었다.

[예제]에서 사용한 ABS 함수를 만드는데, INPUT 값으로 숫자만 들어온다고 가정한다.

[예제] Oracle

```
CREATE OR REPLACE Function UTIL_ABS
(v_input in number)        ---------------- ①
 return NUMBER
IS
 v_return number := 0;    ---------------- ②
```

```
BEGIN
 if v_input <0 then ---------------- ③
    v_return := v_input * -1;
 else v_return := v_input;
 end if;
RETURN v_return; ---------------- ④
END;
/
```

[예제] SQL Server

```
CREATE Function dbo.UTIL_ABS
(@v_input int)              ---------------- ①
RETURNS int
AS
BEGIN
  DECLARE @v_return int  ---------------- ②
  SET @v_return=0
  IF @v_input <0          ---------------- ③
     SET @v_return = @v_input * -1
  ELSE
     SET @v_return = @v_input
  RETURN @v_return;       ---------------- ④
END
```

[예제]에서 생성한 UTIL_ABS Function의 처리 과정은 다음과 같다.

① 숫자 값을 입력 받는다. 예제에서는 숫자 값만 입력된다고 가정한다.

② 리턴 값을 받아 줄 변수인 v_return를 선언한다.

③ 입력 값이 음수이면 -1을 곱하여 v_return 변수에 대입한다.

④ v_return 변수를 리턴한다.

[예제] 함수를 이용하여 앞의 SQL을 수정하여 실행한다.

[예제] Oracle

```
SELECT SCHE_DATE 경기일자,
        HOMETEAM_ID || ' - ' || AWAYTEAM_ID 팀들,
        HOME_SCORE || ' - ' || AWAY_SCORE SCORE,
        UTIL_ABS(HOME_SCORE - AWAY_SCORE) 점수차
FROM SCHEDULE
```

```
WHERE GUBUN = 'Y'
AND  SCHE_DATE BETWEEN '20120801' AND '20120831'
ORDER BY SCHE_DATE;
```

[예제] SQL Server

```
SELECT SCHE_DATE 경기일자,
        HOMETEAM_ID + ' - ' + AWAYTEAM_ID AS 팀들,
        HOME_SCORE + ' - ' + AWAY_SCORE AS SCORE,
       dbo.UTIL_ABS(HOME_SCORE - AWAY_SCORE) AS 점수차
FROM SCHEDULE
WHERE GUBUN = 'Y' AND SCHE_DATE BETWEEN '20120801' AND '20120831'
ORDER BY SCHE_DATE;
```

[실행 결과]

```
경기일자   팀들        SCORE     점수차
------- --------    ------    ------
20120803 K01 - K03   3 - 0       3
20120803 K06 - K09   2 - 1       1
20120803 K08 - K07   1 - 0       1
20120804 K05 - K04   2 - 1       1
20120804 K10 - K02   0 - 3       3
25개의 행이 선택되었다.
```

실행 결과는 앞의 ABS 내장함수를 사용한 SQL 문장과 같은 결과를 확인할 수 있다.

6. Trigger의 생성과 활용

Trigger란 특정한 테이블에 INSERT, UPDATE, DELETE와 같은 DML문이 수행되었을 때, 데이터베이스에서 자동으로 동작하도록 작성된 프로그램이다. 즉 사용자가 직접 호출하여 사용하는 것이 아니고 데이터베이스에서 자동적으로 수행하게 된다.

Trigger는 테이블과 뷰, 데이터베이스 작업을 대상으로 정의할 수 있으며, 전체 트랜잭션 작업에 대해 발생되는 Trigger와 각 행에 대해서 발생되는 Trigger가 있다.

요구 사항은 다음과 같다고 가정한다. 어떤 쇼핑몰에 하루에 수만 건의 주문이 들어온다. 주문 데이터는 주문일자, 주문상품, 수량, 가격이 있으며, 사장을 비롯한 모든 임직원이 일자별, 상품별 총 판매수량과 총 판매가격으로 구성된 주문 실적을 온라인상으로 실시간 조회한다고 했을 때, 한 사람의 임직원이 조회할 때마다 수만 건의 데이터를 읽어 계산해야 한다. 가끔 한 번씩 조회한다면 문제가 없을 수도 있으나 빈번하게 조회작업이 일어난다면 조회작업에 많은 시간을 허비할 수 있다.

[예제] 트리거(Trigger)를 사용하여 주문한 건이 입력될 때마다, 일자별 상품별로 판매수량과 판매금액을 집계하여 집계자료를 보관하도록 한다. 먼저 관련 테이블을 생성한다.

[표 Ⅱ-2-15] 주문정보 테이블

ORDER_LIST		
필드명	TYPE	길이
ORDER_DATE	CHAR	8
PRODUCT	VARCHAR2	10
QTY	NUMBER	
AMOUNT	NUMBER	

[표 Ⅱ-2-16] 일자별 판매집계 테이블

SALES_PER_DATE		
필드명	TYPE	길이
SALE_DATE	CHAR	8
PRODUCT	VARCHAR2	10
QTY	NUMBER	
AMOUNT	NUMBER	

[예제] Oracle

```
CREATE TABLE ORDER_LIST (
ORDER_DATE CHAR(8)     NOT NULL,
PRODUCT VARCHAR2(10) NOT NULL,
QTY        NUMBER NOT NULL,
AMOUNT   NUMBER NOT NULL);

CREATE TABLE SALES_PER_DATE (
SALE_DATE   CHAR(8) NOT NULL,
PRODUCT     VARCHAR2(10) NOT NULL,
QTY           NUMBER NOT NULL,
AMOUNT      NUMBER NOT NULL);
```

[예제] SQL Server

```
CREATE TABLE ORDER_LIST (
ORDER_DATE CHAR(8) NOT NULL,
PRODUCT     VARCHAR(10) NOT NULL,
QTY           INT NOT NULL,
AMOUNT      INT NOT NULL);

CREATE TABLE SALES_PER_DATE (
SALE_DATE CHAR(8) NOT NULL,
```

```
PRODUCT VARCHAR(10) NOT NULL,
QTY     INT NOT NULL,
AMOUNT  INT NOT NULL);
```

[예제] 이제 Trigger를 작성한다.

Trigger의 역할은 ORDER_LIST에 주문 정보가 입력되면 주문 정보의 주문 일자(ORDER_LIST.ORDER_DATE)와 주문 상품(ORDER_LIST.PRODUCT)을 기준으로 판매 집계 테이블(SALES_PER_DATE)에 해당 주문 일자의 주문 상품 레코드가 존재하면 판매 수량과 판매 금액을 더하고 존재하지 않으면 새로운 레코드를 입력한다.

[예제] Oracle

```
CREATE OR REPLACE Trigger SUMMARY_SALES ---------------- ①
  AFTER INSERT
  ON ORDER_LIST
  FOR EACH ROW
DECLARE ---------------- ②
  o_date ORDER_LIST.order_date%TYPE;
  o_prod ORDER_LIST.product%TYPE;
BEGIN
 o_date := :NEW.order_date;
 o_prod := :NEW.product;
 UPDATE SALES_PER_DATE ---------------- ③
 SET qty = qty + :NEW.qty,
  amount = amount + :NEW.amount
WHERE sale_date = o_date
  AND product = o_prod;
if SQL%NOTFOUND then ---------------- ④
   INSERT INTO SALES_PER_DATE
   VALUES(o_date, o_prod, :NEW.qty, :NEW.amount);
end if;
END;
/
```

SUMMARY_SALES Trigger의 처리절차를 설명하면 다음과 같다.

① Trigger를 선언한다.

CREATE OR REPLACE Trigger SUMMARY_SALES : Trigger 선언문

AFTER INSERT : 레코드가 입력이 된 후 Trigger 발생

ON ORDER_LIST : ORDER_LIST 테이블에 Trigger 설정

FOR EACH ROW : 각 ROW마다 Trigger 적용

② o_date(주문일자), o_prod(주문상품) 값을 저장할 변수를 선언하고, 신규로 입력된 데이터를 저장한다.

: NEW는 신규로 입력된 레코드의 정보를 가지고 있는 구조체

: OLD는 수정, 삭제되기 전의 레코드를 가지고 있는 구조체 [표 Ⅱ-2-17] 참조

③ 먼저 입력된 주문 내역의 주문 일자와 주문 상품을 기준으로 SALES_PER_DATE 테이블에 업데이트한다.

④ 처리 결과가 SQL%NOTFOUND이면 해당 주문 일자의 주문 상품 실적이 존재하지 않으며, SALES_ PER_DATE 테이블에 새로운 집계 데이터를 입력한다.

[표 Ⅱ-2-17] Trigger에서 사용하는 레코드 구조체 비교(1)

구분	:OLD	:NEW
INSERT	NULL	입력된 레코드 값
UPDATE	UPDATE되기 전의 레코드의 값	UPDATE된 후의 레코드 값
DELETE	레코드가 삭제되기 전 값	NULL

[예제] SQL Server

```
CREATE Trigger dbo.SUMMARY_SALES                ---------------- ①
  ON ORDER_LIST
  AFTER INSERT
AS
DECLARE
  @o_date DATETIME,@o_prod INT,@qty int, @amount int
BEGIN
   SELECT @o_date=order_date, @o_prod=product, @qty=qty, @amount=amount
   FROM inserted                                ---------------- ②
   UPDATE SALES_PER_DATE                        ---------------- ③
     SET qty = qty + @qty,
         amount = amount + @amount
   WHERE sale_date = @o_date
     AND product = @o_prod;
   IF @@ROWCOUNT=0                              ---------------- ④
      INSERT INTO SALES_PER_DATE
    VALUES(@o_date, @o_prod, @qty, @amount)
END
```

SUMMARY_SALES Trigger의 처리절차를 설명하면 다음과 같다.

① Trigger를 선언한다.

CREATE Trigger SUMMARY_SALES : Trigger 선언문

ON ORDER_LIST : ORDER_LIST 테이블에 Trigger 설정

AFTER INSERT : 레코드가 입력이 된 후 Trigger 발생

② o_date(주문일자), o_prod(주문상품), qty(수량), amount(금액) 값을 저장할 변수를 선언하고, 신규로 입력된 데이터를 저장한다.

inserted는 신규로 입력된 레코드의 정보를 가지고 있는 구조체

deleted는 수정, 삭제되기 전의 레코드를 가지고 있는 구조체. [표 Ⅱ-2-18] 참조

③ 먼저 입력된 주문 내역의 주문 일자와 주문 상품을 기준으로 SALES_PER_DATE 테이블에 업데이트한다.

④ 처리 결과가 0건이면 해당 주문 일자의 주문 상품 실적이 존재하지 않으며, SALES_PER_DATE 테이블에 새로운 집계 데이터를 입력한다.

[표 Ⅱ-2-18] Trigger에서 사용하는 레코드 구조체 비교(2)

구분	deleted	inserted
INSERT	NULL	입력된 레코드 값
UPDATE	UPDATE되기 전의 레코드의 값	UPDATE된 후의 레코드 값
DELETE	레코드가 삭제되기 전 값	NULL

[예제] ORDER_LIST 테이블에 주문 정보를 입력한다.

[예제] Oracle

```
SQL> SELECT * FROM ORDER_LIST;
선택된 레코드가 없다.

SQL> SELECT * FROM SALES_PER_DATE;
선택된 레코드가 없다.

SQL> INSERT INTO ORDER_LIST VALUES('20120901', 'MONOPACK', 10, 300000);
1개의 행이 만들어졌다.

SQL> COMMIT;
커밋이 완료되었다.
```

[예제] SQL Server

```
SELECT * FROM ORDER_LIST;
```

```
선택된 레코드가 없다.

SELECT * FROM SALES_PER_DATE;
선택된 레코드가 없다.

INSERT INTO ORDER_LIST VALUES('20120901', 'MONOPACK', 10, 300000);
1개의 행이 만들어졌다.
```

[예제] 주문 정보와 판매 집계 테이블에 같은 데이터가 들어왔는지 확인한다

[실행 결과]

```
SQL> SELECT * FROM ORDER_LIST;
ORDER_DATG PRODUCT     QTY      AMOUNT
---------  --------  --------   -------
20120901   MONOPACK    10       300000

SQL> SELECT * FROM SALES_PER_DATE;
SALE_DATG PRODUCT    QTY      AMOUNT
-------- --------  --------  --------
20120901  MONOPACK    10      300000
```

[예제] 다시 한 번 같은 데이터를 입력해보고, 두 테이블의 데이터를 확인한다.

[실행 결과] Oracle

```
SQL> INSERT INTO ORDER_LIST VALUES('20120901','MONOPACK',20,600000);
1개의 행이 만들어졌다.
SQL> COMMIT;
커밋이 완료되었다.
SQL> SELECT * FROM ORDER_LIST;
ORDER_DATG PRODUCT QTY AMOUNT
--------- ---------- ------ -------
20120901 MONOPACK 10 300000
20120901 MONOPACK 20 600000
SQL> SELECT * FROM SALES_PER_DATE;
SALE_DATG PRODUCT QTY AMOUNT
-------- --------- ----- -------
20120901 MONOPACK 30 900000
```

[실행 결과] SQL Server

```
INSERT INTO ORDER_LIST VALUES('20120901','MONOPACK',20,600000);
1개의 행이 만들어졌다.

SELECT * FROM ORDER_LIST;
ORDER_DATG PRODUCT QTY AMOUNT
--------- --------- ----- --------
20120901 MONOPACK   10 300000
20120901 MONOPACK   20 600000

SELECT * FROM SALES_PER_DATE;
SALE_DATG PRODUCT QTY AMOUNT
-------- --------- ---- --------
20120901 MONOPACK  30   900000
```

[예제] 이번에는 다른 상품으로 주문 데이터를 입력한 후 두 테이블의 결과를 조회해 보고 트랜잭션을 ROLLBACK 수행한다. 판매 데이터의 입력 취소가 일어나면, 주문 정보 테이블과 판매 집계 테이블에 동시에 입력(수정) 취소가 일어나는지 확인해본다.

[실행 결과] Oracle

```
SQL> INSERT INTO ORDER_LIST VALUES('20120901','MULTIPACK',10,300000);
  1개의 행이 만들어졌다.
SQL> SELECT * FROM ORDER_LIST;
ORDER_DA PRODUCT QTY AMOUNT
-------- -------- ------ -------
20120901 MONOPACK 10 300000
20120901 MONOPACK 20 600000
20120901 MULTIPACK 10 300000

SQL> SELECT * FROM SALES_PER_DATE;
SALE_DATG PRODUCT QTY AMOUNT
-------- -------- ------ -------
20120901 MONOPACK 30 900000
20120901 MULTIPACK 10 300000

SQL> ROLLBACK;
롤백이 완료되었다.
```

```
SQL> SELECT * FROM ORDER_LIST;
ORDER_DATG PRODUCT QTY AMOUNT
-------- -------- ------ -------
20120901 MONOPACK 10 300000
20120901 MONOPACK 20 600000

SQL> SELECT * FROM SALES_PER_DATE;
SALE_DATG PRODUCT QTY AMOUNT
-------- --------- ------ -------
20120901 MONOPACK 30 900000
```

[실행 결과] SQL Server

```
BEGIN TRAN
INSERT INTO ORDER_LIST VALUES('20120901','MULTIPACK',10,300000);
1개의 행이 만들어졌다.

SELECT * FROM ORDER_LIST;
ORDER_DATG PRODUCT QTY AMOUNT
--------- --------- ------ -------
20120901 MONOPACK 10 300000
20120901 MONOPACK 20 600000
20120901 MULTIPACK 10 300000

SELECT * FROM SALES_PER_DATE;
SALE_DATG PRODUCT QTY AMOUNT
-------- --------- ------ -------
20120901 MONOPACK 30 900000
20120901 MULTIPACK 10 300000

ROLLBACK;
롤백이 완료되었다.

SELECT * FROM ORDER_LIST;
ORDER_DATG PRODUCT QTY AMOUNT
--------- -------- ------ -------
20120901 MONOPACK 10 300000
20120901 MONOPACK 20 600000
```

```
SELECT * FROM SALES_PER_DATE;
SALE_DATG PRODUCT QTY AMOUNT
-------- -------- ------ -------
20120901 MONOPACK 30 900000
```

ROLLBACK을 하면 하나의 트랜잭션이 취소가 되어 Trigger로 입력된 정보까지 하나의 트랜잭션으로 인식하여 두 테이블 모두 입력 취소가 되는 것을 보여 주고 있다. Trigger는 데이터베이스에 의해 자동 호출되지만 결국 INSERT, UPDATE, DELETE 문과 하나의 트랜잭션 안에서 일어나는 일련의 작업들이라 할 수 있다. Trigger는 데이터베이스 보안의 적용, 유효하지 않은 트랜잭션의 예방, 업무 규칙 자동 적용 제공 등에 사용될 수 있다.

7. 프로시저와 트리거의 차이점

프로시저는 BEGIN ~ END 절 내에 COMMIT, ROLLBACK과 같은 트랜잭션 종료 명령어를 사용할 수 있지만, 데이터베이스 트리거는 BEGIN ~ END 절 내에 사용할 수 없다.

[표 Ⅱ-2-19] 프로시저와 트리거의 차이점

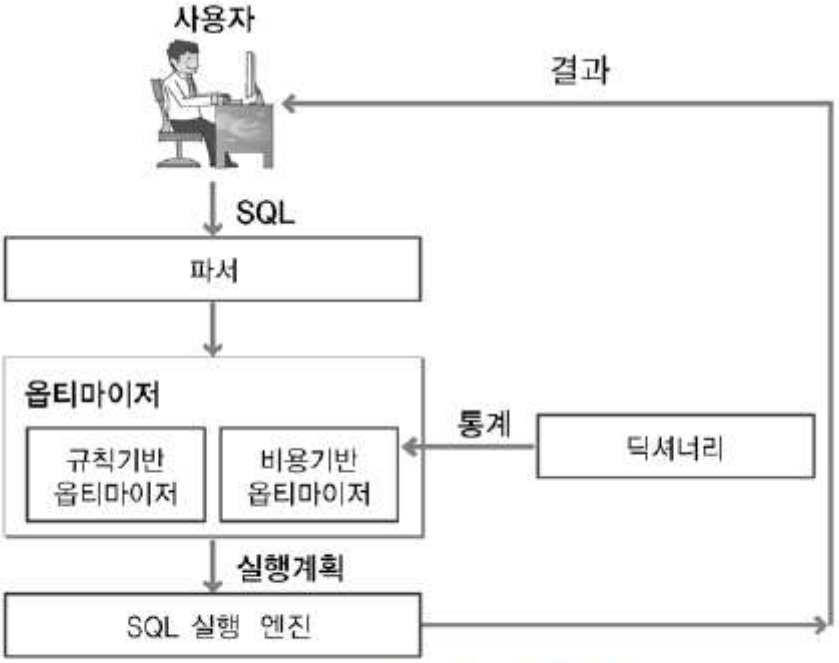
프로시저	트리거
CREATE Procedure 문법사용	CREATE Trigger 문법사용
EXECUTE 명령어로 실행	생성 후 자동으로 실행
COMMIT, ROLLBACK 실행 가능	COMMIT, ROLLBACK 실행 안됨

제1절 옵티마이저와 실행계획

1. 옵티마이저

옵티마이저(Optimizer)는 사용자가 질의한 SQL문에 대해 최적의 실행 방법을 결정하는 역할을 수행한다. 이러한 최적의 실행 방법을 실행계획(Execution Plan)이라고 한다. 관계형 데이터베이스는 궁극적으로 SQL문을 통해서만 데이터를 처리할 수 있다. JAVA, C등과 같은 프로그램 언어와는 달리 SQL은 사용자의 요구사항만 기술할 뿐 처리과정에 대한 기술은 하지 않는다. 그러므로 사용자의 요구사항을 만족하는 결과를 추출할 수 있는 다양한 실행 방법이 존재할 수 있다. 다양한 실행 방법들 중에서 최적의 실행 방법을 결정하는 것이 바로 옵티마이저의 역할이다. 관계형 데이터베이스는 옵티마이저가 결정한 실행 방법대로 실행 엔진이 데이터를 처리하여 결과 데이터를 사용자에게 전달할 뿐이다. 옵티마이저가 선택한 실행 방법의 적절성 여부는 질의의 수행 속도에 가장 큰 영향 미치게 된다. 이런 의미에서 관계형 데이터베이스에서 진정한 프로그래머는 옵티마이저라고 할 수 있다.

최적의 실행 방법 결정이라는 것은 어떤 방법으로 처리하는 것이 최소 일량으로 동일한 일을 처리할 수 있을지 결정하는 것이다. 그러나 이러한 결정을 옵티마이저는 실제로 SQL문을 처리해보지 않은 상태에서 결정해야 하는 어려움이 있다. 옵티마이저가 최적의 실행 방법을 결정하는 방식에 따라 [그림 Ⅱ-3-1]과 같이 규칙기반 옵티마이저(RBO, Rule Based Optimizer)와 비용기반 옵티마이저(CBO, Cost Based Optimizer)로 구분할 수 있다.



[그림 Ⅱ-3-1] 옵티마이저의 종류

현재 대부분의 관계형 데이터베이스는 비용기반 옵티마이저만을 제공한다. 비록 규칙기반 옵티마이저를 제공하더라도 신규 기능들에 대해서는 더 이상 지원하지 않는다. 다만 하위 버전 호환성을 위해서만 규칙기반 옵티마이저가 남아 있을 뿐이다. 그렇지만 규칙기반 옵티마이저의 규칙은 보편 타당성에 근거한 것들이다. 이러한 규칙을 알고 있는 것은 옵티마이저의 최적화 작업을 이해하는데 도움이 된다.

가. 규칙기반 옵티마이저

규칙기반 옵티마이저는 규칙(우선 순위)을 가지고 실행계획을 생성한다. 실행계획을 생성하는 규칙을 이해하면 누구나 실행계획을 비교적 쉽게 예측할 수 있다. 규칙기반 옵티마이저가 실행계획을 생성할 때 참조하는 정보에는 SQL문을 실행하기 위해서 이용 가능한 인덱스 유무와 (유일, 비유일, 단일, 복합 인덱스)종류, SQL문에서 사용하는 연산자(=, <, <>, LIKE, BETWEEN 등)의 종류 그리고 SQL문에서 참조하는 객체(힙 테이블, 클러스터 테이블 등)의 종류 등이 있다. 이러한 정보에 따라 우선 순위(규칙)가 정해져 있고, 이 우선 순위를 기반으로 실행계획을 생성한다. 결과적으로 규칙기반 옵티마이저는 우선 순위가 높은 규칙이 적은 일량으로 해당 작업을 수행하는 방법이라고 판단하는 것이다. [그림 Ⅱ-3-2]는 Oracle의 규칙기반 옵티마이저의 15가지 규칙이다. 순위의 숫자가 낮을수록 높은 우선 순위이다.

순위	액세스 기법
1	Single row by rowid
2	Single row by cluster join
3	Single row by hash cluster key with unique or primary key
4	Single row by unique or primary key
5	Cluster join
6	Hash cluster key
7	Indexed cluster key
8	Composite index
9	Single column index
10	Bounded range search on indexed columns
11	Unbounded range search on indexed columns
12	Sort merge join
13	MAX or MIN of indexed column
14	ORDER BY on indexed column
15	Full table scan

[그림 Ⅱ-3-2] 규칙기반 옵티마이저의 규칙

규칙기반 옵티마이저의 우선 순위 규칙 중에서 주요한 규칙에 대해서만 간략히 설명한다.

규칙 1. Single row by rowid : ROWID를 통해서 테이블에서 하나의 행을 액세스하는 방식이다. ROWID는 행이 포함된 데이터 파일, 블록 등의 정보를 가지고 있기 때문에 다른 정보를 참조하지 않고도 바로 원하는 행을 액세스할 수 있다. 하나의 행을 액세스하는 가장 빠른 방법이다.

규칙 4. Single row by unique or primary key : 유일 인덱스(Unique Index)를 통해서 하나의 행을 액세스하는 방식이다. 이 방식은 인덱스를 먼저 액세스하고 인덱스에 존재하는 ROWID를 추출하여 테이블의 행을 액세스한다.

규칙 8. Composite index : 복합 인덱스에 동등('=' 연산자) 조건으로 검색하는 경우이다. 예를 들어, 만약 A+B 칼럼으로 복합 인덱스가 생성되어 있고, 조건절에서 WHERE A=10 AND B=1 형태로 검색하는 방식이다. 복합 인덱스 사이의 우선 순위 규칙은 다음과 같다. 인덱스 구성 칼럼의 개수가 더 많고 해당 인덱스의 모든 구성 칼럼에 대해 '='로 값이 주어질 수록 우선순위가 더 높다. 예를 들어, A+B로 구성된 인덱스와 A+B+C로 구성된 인덱스가 각각 존재하고 조건절에서 A, B, C 칼럼 모두에 대해 '='로 값이 주어진다면 A+B+C 인덱스가 우선 순위가 높다. 만약 조건절에서 A, B 칼럼에만 '='로 값이 주어진다면 A+B는 인덱스의 모든 구성 칼럼에 대해 값이 주어지고 A+B+C 인덱스 입장에서는 인덱스의 일부 칼럼에 대해서만 값이 주어졌기 때문에 A+B 인덱스가 우선 순위가 높게 된다.

규칙 9. Single column index : 단일 칼럼 인덱스에 '=' 조건으로 검색하는 경우이다. 만약 A 칼럼에 단일 칼럼 인덱스가 생성되어 있고, 조건절에서 A=10 형태로 검색하는 방식이다.

규칙 10. Bounded range search on indexed columns : 인덱스가 생성되어 있는 칼럼에 양쪽 범위를 한정하는 형태로 검색하는 방식이다. 이러한 연산자에는 BETWEEN, LIKE 등이 있다. 만약 A 칼럼에 인덱스가 생성되어 있고, A BETWEEN '10' AND '20' 또는 A LIKE '1%' 형태로 검색하는 방식이다.

규칙 11. Unbounded range search on indexed columns : 인덱스가 생성되어 있는 칼럼에 한쪽 범위만 한정하는 형태로 검색하는 방식이다. 이러한 연산자에는 >, >=, <, <= 등이 있다. 만약 A 칼럼에 인덱스가 생성되어 있고, A >'10' 또는 A <'20' 형태로 검색하는 방식이다.

규칙 15. Full table scan : 전체 테이블을 액세스하면서 조건절에 주어진 조건을 만족하는 행만을 결과로 추출한다.

규칙기반 옵티마이저는 인덱스를 이용한 액세스 방식이 전체 테이블 액세스 방식보다 우선 순위가 높다. 따라서 규칙기반 옵티마이저는 해당 SQL문에서 이용 가능한 인덱스가 존재한다면 전체 테이블 액세스 방식보다는 항상 인덱스를 사용하는 실행계획을 생성한다.

규칙기반 옵티마이저가 조인 순서를 결정할 때는 조인 칼럼 인덱스의 존재 유무가 중요한

판단의 기준이다. 조인 칼럼에 대한 인덱스가 양쪽 테이블에 모두 존재한다면 앞에서 설명한 규칙에 따라 우선 순위가 높은 테이블을 선행 테이블(Driving Table)로 선택한다. 한쪽 조인 칼럼에만 인덱스가 존재하는 경우에는 인덱스가 없는 테이블을 선행 테이블로 선택해서 조인을 수행한다. 조인 칼럼에 모두 인덱스가 존재하지 않으면 FROM 절의 뒤에 나열된 테이블을 선행 테이블로 선택한다. 만약 조인 테이블의 우선 순위가 동일하다면 FROM 절에 나열된 테이블의 역순으로 선행 테이블을 선택한다.

규칙기반 옵티마이저의 조인 기법의 선택은 다음과 같다. 양쪽 조인 칼럼에 모두 인덱스가 없는 경우에는 Sort Merge Join을 사용하고 둘 중하나라도 조인 칼럼에 인덱스가 존재한다면 일반적으로 NL Join을 사용한다.

다음 SQL문을 이용해서 규칙기반 옵티마이저의 최적화 과정을 알아보자.

```
SELECT ENAME
FROM EMP
WHERE JOB = 'SALESMAN'
AND SAL BETWEEN 3000 AND 6000

INDEX
---------------------------------
EMP_JOB : JOB
EMP_SAL : SAL
PK_EMP : EMPNO (UNIQUE)
```

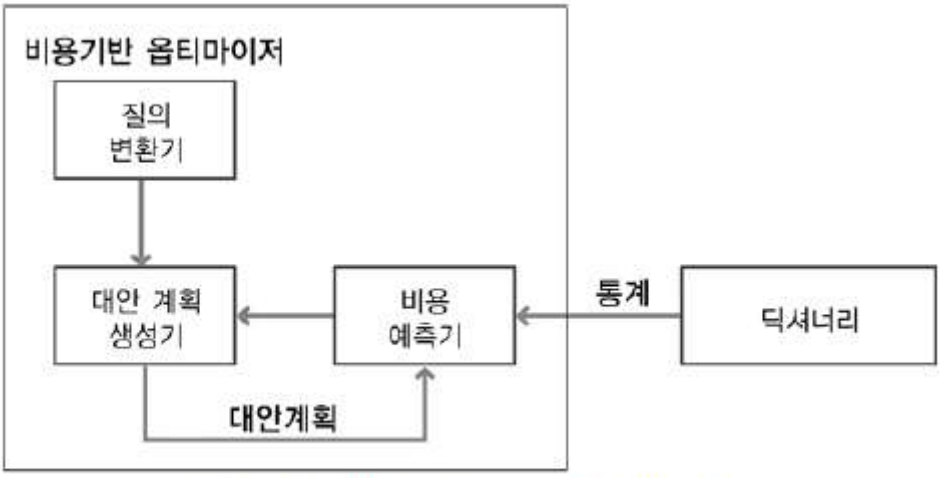
조건절에서 JOB 칼럼의 조건은 '=', SAL 칼럼의 조건은 'BETWEEN'으로 값이 주어졌고 각각의 칼럼에 단일 칼럼 인덱스가 존재한다. 우선 순위 규칙에 따라 JOB 조건은 규칙 9의 단일 칼럼 인덱스를 만족하고 SAL 조건은 규칙 10의 인덱스상의 양쪽 한정 검색을 만족한다. 따라서 우선 순위가 높은 EMP_JOB 인덱스를 이용해서 조건을 만족하는 행에 대해 EMP 테이블을 액세스하는 방식을 선택할 것이다. 다음은 규칙기반 옵티마이저가 생성한 실행계획이다.

```
Execution Plan
------------------------------------------------------------

SELECT STATEMENT Optimizer=CHOOSE
  TABLE ACCESS (BY INDEX ROWID) OF 'EMP'
    INDEX (RANGE SCAN) OF 'EMP_JOB' (NON-UNIQUE)
```

나. 비용기반 옵티마이저

규칙기반 옵티마이저는 조건절에서 '=' 연산자와 'BETWEEN' 연산자가 사용되면 규칙에 따라 '=' 칼럼의 인덱스를 사용하는 것이 보다 적은 일량 즉, 보다 적은 처리 범위로 작업을 할 것이라고 판단한다. 그러나 실제로는 'BETWEEN' 칼럼을 사용한 인덱스가 보다 일량이 적을 수 있다. 단순한 몇 개의 규칙만으로 현실의 모든 사항을 정확히 예측할 수는 없다. 비용기반 옵티마이저는 이러한 규칙기반 옵티마이저의 단점을 극복하기 위해서 출현하였다.

비용기반 옵티마이저는 SQL문을 처리하는데 필요한 비용이 가장 적은 실행계획을 선택하는 방식이다. 여기서 비용이란 SQL문을 처리하기 위해 예상되는 소요시간 또는 자원 사용량을 의미한다. 비용기반 옵티마이저는 비용을 예측하기 위해서 규칙기반 옵티마이저가 사용하지 않는 테이블, 인덱스, 칼럼 등의 다양한 객체 통계정보와 시스템 통계정보 등을 이용한다. 통계정보가 없는 경우 비용기반 옵티마이저는 정확한 비용 예측이 불가능해져서 비효율적인 실행계획을 생성할 수 있다. 그렇기 때문에 정확한 통계정보를 유지하는 것은 비용기반 최적화에서 중요한 요소이다.



[그림 Ⅱ-3-3] 비용기반 옵티마이저의 구성 요소

[그림 Ⅱ-3-3]과 같이 비용기반 옵티마이저는 질의 변환기, 대안 계획 생성기, 비용 예측기 등의 모듈로 구성되어 있다. 질의 변환기는 사용자가 작성한 SQL문을 처리하기에 보다 용이한 형태로 변환하는 모듈이다.

대안 계획 생성기는 동일한 결과를 생성하는 다양한 대안 계획을 생성하는 모듈이다. 대안 계획은 연산의 적용 순서 변경, 연산 방법 변경, 조인 순서 변경 등을 통해서 생성된다. 동일한 결과를 생성하는 가능한 모든 대안 계획을 생성해야 보다 나은 최적화를 수행할 수 있다. 그러나 대안 계획의 생성이 너무 많아지면 최적화를 수행하는 시간이 그만큼 오래 걸린 수 있다. 그래서 대부분의 상용 옵티마이저들은 대안 계획의 수를 제약하는 다양한 방법을 사용한다. 이러한 현실적인 제약으로 인해 생성된 대안 계획들 중에서 최적의 대안 계획이 포함되지 않을 수도 있다.

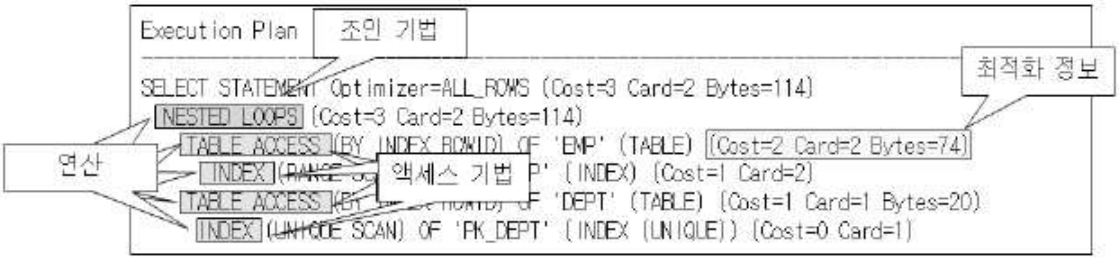
비용 예측기는 대안 계획 생성기에 의해서 생성된 대안 계획의 비용을 예측하는 모듈이다. 대안 계획의 정확한 비용을 예측하기 위해서 연산의 중간 집합의 크기 및 결과 집합의 크기, 분포도 등의 예측이 정확해야 한다. 보다 나은 예측을 위해 옵티마이저는 정확한 통계정보를 필요로 한다. 또한 대안 계획을 구성하는 각 연산에 대한 비용 계산식이 정확해야 한다.

앞에서 규칙기반 옵티마이저는 항상 인덱스를 사용할 수 있다면 전체 테이블 스캔 보다는 인덱스를 사용하는 실행계획을 생성한다고 했다. 그렇지만 비용기반 옵티마이저는 인덱스를 사용하는 비용이 전체 테이블 스캔 비용보다 크다고 판단되면 전체 테이블 스캔을 수행하는 방법으로 실행계획을 생성할 수도 있다. 비용기반 옵티마이저는 통계정보, DBMS 버전, DBMS 설정 정보 등의 차이로 인해 동일 SQL문도 서로 다른 실행계획이 생성될 수 있다. 또한 비용기반 옵티마이저의 다양한 한계들로 인해 실행계획의 예측 및 제어가 어렵다는 단점이 있다.

2. 실행계획

실행계획(Execution Plan)이란 SQL에서 요구한 사항을 처리하기 위한 절차와 방법을 의미한다. 실행계획을 생성한다는 것은 SQL을 어떤 순서로 어떻게 실행할 지를 결정하는 작업이다. 동일한 SQL에 대해 결과를 낼 수 있는 다양한 처리 방법(실행계획)이 존재할 수 있지만 각 처리 방법마다 실행 시간(성능)은 서로 다를 수 있다. 옵티마이저는 다양한 처리 방법들 중에서 가장 효율적인 방법을 찾아준다. 즉, 옵티마이저는 최적의 실행계획을 생성해 준다.

생성된 실행계획을 보는 방법은 데이터베이스 벤더마다 서로 다르다. 실행계획에서 표시되는 내용 및 형태도 약간씩 차이는 있지만 실행계획이 SQL 처리를 위한 절차와 방법을 의미한다는 기본적인 사항은 모두 동일하다. 실행계획을 보고 SQL이 어떻게 실행되는지 정확히 이해할 수 있다면 보다 향상된 SQL의 이해 및 활용이 가능하다. Oracle의 실행계획 형태는 [그림 Ⅱ-3-4]와 같다. 실행계획을 구성하는 요소에는 조인 순서(Join Order), 조인 기법(Join Method), 액세스 기법(Access Method), 최적화 정보(Optimization Information), 연산(Operation) 등이 있다.



[그림 Ⅱ-3-4] 실행계획 정보의 구성요소

조인 순서는 조인작업을 수행할 때 참조하는 테이블의 순서이다. 예를 들어, FROM 절에 A, B 두 개의 테이블이 존재할 때 조인 작업을 위해 먼저 A 테이블을 읽고 B 테이블을 읽는 작업을 수행한다면 조인 순서는 A → B이다. [그림 Ⅱ-3-4]에서 조인 순서는 EMP → DEPT이다. 논리적으로 가능한 조인 순서는 n! 만큼 존재한다. 여기서는 n은 FROM 절에 존재하는 테이블 수이다. 그러나 현실적으로 옵티마이저가 적용 가능한 조인 순서는 이보다는 적거나 같다.

조인 기법은 두 개의 테이블을 조인할 때 사용할 수 있는 방법으로서 여기에는 NL Join, Hash Join, Sort Merge Join 등이 있다. [그림 Ⅱ-3-4]에서 조인 기법은 NL Join을 사용하고 있다.

액세스 기법은 하나의 테이블을 액세스할 때 사용할 수 있는 방법이다. 여기에는 인덱스를 이용하여 테이블을 액세스하는 인덱스 스캔(Index Scan)과 테이블 전체를 모두 읽으면서 조건을 만족하는 행을 찾는 전체 테이블 스캔(Full Table Scan) 등이 있다. [그림 Ⅱ-3-4]에서 액세스 기법은 인덱스 스캔을 사용하고 있다.

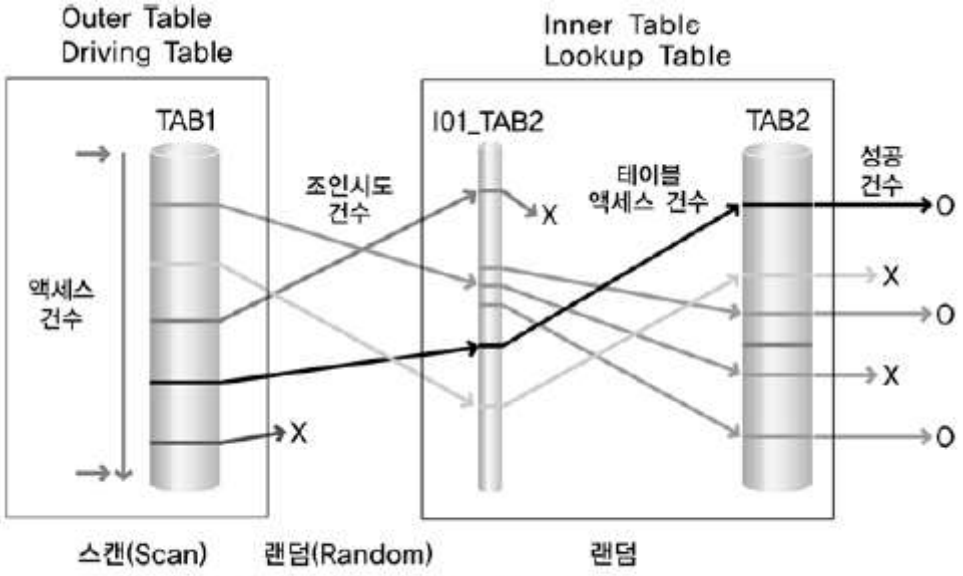
최적화 정보는 옵티마이저가 실행계획의 각 단계마다 예상되는 비용 사항을 표시한 것이다. 실행계획에 비용 사항이 표시된다는 것은 비용기반 최적화 방식으로 실행계획을 생성했다는 것을 의미한다. 최적화 정보에는 Cost, Card, Bytes가 있다. Cost는 상대적인 비용 정보이고 Card는 Cardinality의 약자로서 주어진 조건을 만족한 결과 집합 혹은 조인 조건

을 만족한 결과 집합의 건수를 의미한다. Bytes는 결과 집합이 차지하는 메모리 양을 바이트로 표시한 것이다. 이러한 비용 정보는 실제로 SQL을 실행하고 얻은 결과가 아니라 통계 정보를 바탕으로 옵티마이저가 계산한 예상치이다. 만약 이러한 비용 사항이 실행계획에 표시되지 않았다면 이것은 규칙기반 최적화 방식으로 실행계획을 생성한 것이다. [그림 Ⅱ-3-4] 실행계획의 예에서는 비용 정보가 표시되어 있으므로 비용기반 최적화 방식으로 생성된 실행계획이다.

연산(Operation)은 여러 가지 조작을 통해서 원하는 결과를 얻어내는 일련의 작업이다. 연산에는 조인 기법(NL Join, Hash Join, Sort Merge Join 등), 액세스 기법(인덱스 스캔, 전체 테이블 스캔 등), 필터, 정렬, 집계, 뷰 등 다양한 종류가 존재한다. 예를 들어, SQL에서 정렬을 목적으로 ORDER BY를 수행했다면 정렬 연산이 표시되다.

3. SQL 처리 흐름도

SQL 처리 흐름도(Access Flow Diagram)란 SQL의 내부적인 처리 절차를 시각적으로 표현한 도표이다. 이것은 실행계획을 시각화한 것이다. [그림 Ⅱ-3-5]와 같이 액세스 처리 흐름도에는 SQL문의 처리를 위해 어떤 테이블을 먼저 읽었는지(조인 순서), 테이블을 읽기 위해서 인덱스 스캔을 수행했는지 또는 테이블 전체 스캔을 수행했는지(액세스 기법)과 조인 기법 등을 표현할 수 있다. 예를 들어, [그림 Ⅱ-3-5]에서 조인 순서는 TAB1 → TAB2이다. 여기서 TAB1을 Outer Table 또는 Driving Table이라고 하고, TAB2를 Inner Table 또는 Lookup Table이라고 한다. 테이블의 액세스 방법은 TAB1은 테이블 전체 스캔을 의미하고 TAB2는 I01_TAB2 이라는 인덱스를 통한 인덱스 스캔을 했음을 표시한 것이다. 조인 방법은 NL Join을 수행했음을 표시한 것이다. [그림 Ⅱ-3-5]에서 TAB1에 대한 액세스는 스캔(Scan) 방식이고 조인시도 및 I01_TAB2 인덱스를 통한 TAB2 액세스는 랜덤(Random) 방식이다. 대량의 데이터를 랜덤 방식으로 액세스하게 되면 많은 I/O가 발생하여 성능상 좋지 않다.



[그림 Ⅱ-3-5] SQL 처리 흐름도

성능적인 관점을 살펴보기 위해서 SQL 처리 흐름도에 일량을 함께 표시할 수 있다. [그림 Ⅱ-3-5]에서 건수(액세스 건수, 조인 시도 건수, 테이블 액세스 건수, 성공 건수)라고 표시된 곳에 SQL 처리를 위해 작업한 건수 또는 처리 결과 건수 등의 일량을 함께 표시할 수 있다. 이것을 통해 어느 부분에서 비효율이 발생하고 있는지에 대한 힌트를 얻을 수 있다.

다음은 [그림 Ⅱ-3-5]가 다음 SQL문에 대한 SQL 처리 흐름도라는 가정으로 설명한다.

```
SELECT …
FROM TAB1 A, TAB2 B
WHERE A.KEY = B.KEY
AND A.COL1 = :condition1
AND B.COL2 = :condition2
```

[그림 Ⅱ-3-5]에서 액세스 건수는 SQL 처리를 위해 TAB1을 액세스한 건수이다. 여기서는 TAB1의 A.COL1 칼럼에 이용 가능한 인덱스가 존재하지 않아 전체 테이블 스캔을 수행했음을 의미한다. 따라서 액세스 건수는 TAB1 테이블의 총 건수와 동일하다.

조인 시도 건수는 TAB1을 액세스한 후 즉, 테이블에서 읽은 해당 건에 대해 A.COL1 = :condition1 조건을 만족한 건만이 TAB2와 조인을 시도한다. 즉, TAB1을 액세스한 후 A.COL1 = :condition1 조건을 만족하지 않는다면 더 이상 조인 작업을 진행할 필요가 없다. 따라서 조인 시도 건수는 TAB1에 주어진 조건인 A.COL1 = :condition1을 만족한 건수가 된다.

테이블 액세스 건수는 B.KEY 칼럼만으로 구성된 인덱스(B.KEY 칼럼만으로 구성된 인덱스라고 가정함)인 I01_TAB2에서 B.KEY = A.KEY (TAB1은 이미 읽혀졌기 때문에 A.KEY 값은 상수임) 조건을 만족한 건만이 TAB2 테이블을 액세스한다. 즉, 조인 시도한 건들 중에서 B.KEY = A.KEY 조건까지 만족한 건과 같다.

성공 건수는 SQL 실행을 통해 사용자에게 답으로서 보여지는 결과 건수이다. TAB2 테이블을 액세스해서 B.COL2 = :condition2 조건까지 만족해야 비로서 사용자에게 보여질 수 있다.

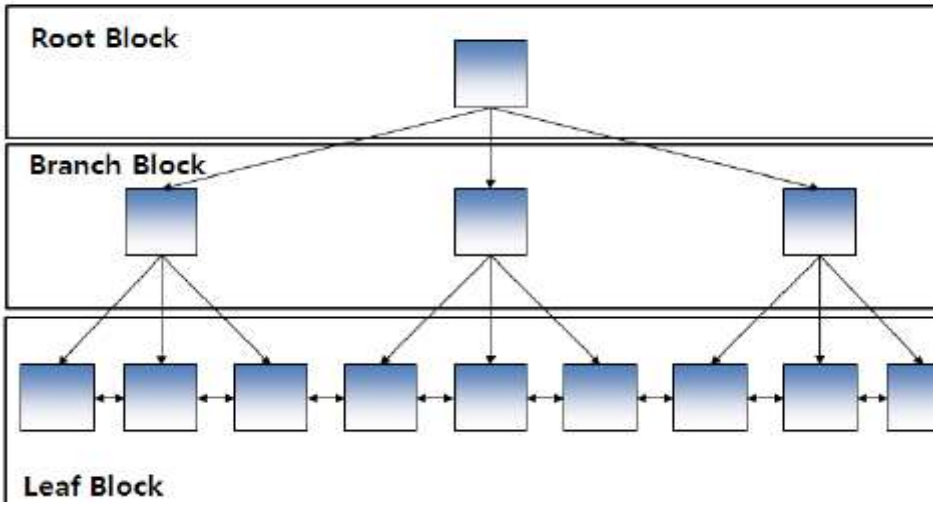
제2절 인덱스 기본

1. 인덱스 특징과 종류

인덱스는 원하는 데이터를 쉽게 찾을 수 있도록 돕는 책의 찾아보기와 유사한 개념이다. 인덱스는 테이블을 기반으로 선택적으로 생성할 수 있는 구조이다. 테이블에 인덱스를 생성하지 않아도 되고 여러 개를 생성해도 된다. 인덱스의 기본적인 목적은 검색 성능의 최적화이다. 즉, 검색 조건을 만족하는 데이터를 인덱스를 통해 효과적으로 찾을 수 있도록 돕는다. 그렇지만 Insert, Update, Delete 등과 같은 DML 작업은 테이블과 인덱스를 함께 변경해야 하기 때문에 오히려 느려질 수 있다는 단점이 존재한다.

가. 트리 기반 인덱스

DBMS에서 가장 일반적인 인덱스는 B-트리 인덱스이다.



[그림 Ⅱ-3-6]과 같이 B-트리 인덱스는 브랜치 블록(Branch Block)과 리프 블록(Leaf Block)으로 구성된다. 브랜치 블록 중에서 가장 상위에서 있는 블록을 루트 블록(Root Block)이라고 한다. 브랜치 블록은 분기를 목적으로 하는 블록이다. 브랜치 블록은 다음 단계의 블록을 가리키는 포인터를 가지고 있다. 리프 블록은 트리의 가장 아래 단계에 존재한다.

리프 블록은 인덱스를 구성하는 칼럼의 데이터와 해당 데이터를 가지고 있는 행의 위치를 가리키는 레코드 식별자(RID, Record Identifier/Rowid)로 구성되어 있다. 인덱스 데이터는 인덱스를 구성하는 칼럼의 값으로 정렬된다. 만약 인덱스 데이터의 값이 동일하면 레코드 식별자의 순서로 저장된다. 리프 블록은 양방향 링크(Double Link)를 가지고 있다. 이것을 통해서 오름 차순(Ascending Order)과 내림 차순(Descending Order) 검색을 쉽게 할 수

있다. B-트리 인덱스는 '='로 검색하는 일치(Exact Match) 검색과 'BETWEEN', '>' 등과 같은 연산자로 검색하는 범위(Range) 검색 모두에 적합한 구조이다.

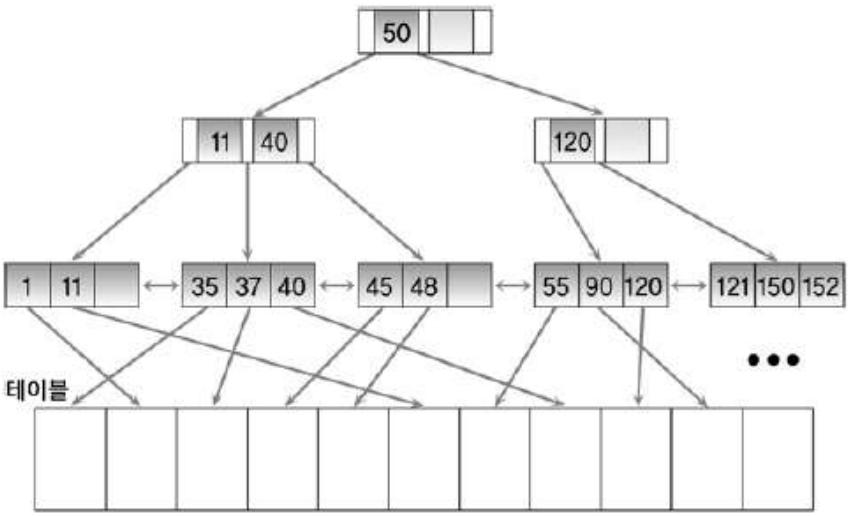
[그림 Ⅱ-3-7]은 브랜치 브록이 3개의 포인터로 구성된 B-트리 인덱스의 예이다. 인덱스에서 원하는 값을 찾는 과정은 다음과 같다.

1단계. 브랜치 블록의 가장 왼쪽 값이 찾고자 하는 값보다 작거나 같으면 왼쪽 포인터로 이동

2단계. 찾고자 하는 값이 브랜치 블록의 값 사이에 존재하면 가운데 포인터로 이동

3단계. 오른쪽에 있는 값보다 크면 오른쪽 포인터로 이동

이 과정을 리프 블록을 찾을 때까지 반복한다. 리프 블록에서 찾고자 하는 값이 존재하면 해당 값을 찾은 것이고, 해당 값이 없으면 해당 값은 존재하지 않아 검색에 실패하게 된다.



[그림 Ⅱ-3-7] B-트리 인덱스 검색

예를 들어, [그림 Ⅱ-3-7]에서 37을 찾고자 한다면 루트 블록에서 50보다 작으므로 왼쪽 포인터로 이동한다. 37는 왼쪽 브랜치 블록의 11과 40 사이의 값이므로 가운데 포인터로 이동한다. 이동한 결과 해당 블록이 리프 블록이므로 37이 블록 내에 존재하는지 검색한다. 본 예에서는 리프 블록에 37이 존재한다. 검색하고자 하는 값을 찾은 것이다. 만약, SQL문에서 다른 칼럼이 더 필요하면 리프 블록에 존재하는 레코드 식별자를 이용해서 테이블을 액세스한다.

만약, 37과 50사이의 모든 값을 찾고자 한다면(BETWEEN 37 AND 50) 위와 동일한 방법으로 리프 블록에서 37을 찾고 50보다 큰 값을 만날 때까지 오른쪽으로 이동하면서 인덱스를 읽는다. 이것은 인덱스 데이터가 정렬되어 있고 리프 블록이 양방향 링크로 연결되어 있기 때문에 가능하다. 인덱스를 경유해서 반환된 결과 데이터는 인덱스 데이터와 동일한 순서로 갖게 되는 특징을 갖는다.

인덱스를 생성할 때 동일 칼럼으로 구성된 인덱스를 중복해서 생성할 수 없다. 그렇지만 인덱스 구성 칼럼은 동일하지만 칼럼의 순서가 다르면 서로 다른 인덱스로 생성할 수 있다. 예를 들어, JOB+SAL 칼럼 순서의 인덱스와 SAL+JOB 칼럼 순서의 인덱스를 별도의 인덱스를 생성할 수 있다. 인덱스의 칼럼 순서는 질의의 성능에 중요한 영향을 미치는 요소이다.

Oracle에서 트리 기반 인덱스에는 B-트리 인덱스 외에도 비트맵 인덱스(Bitmap Index), 리버스 키 인덱스(Reverse Key Index), 함수기반 인덱스(FBI, Function-Based Index) 등이 존재한다.

나. SQL Server의 클러스터형 인덱스

SQL Server의 인덱스 종류는 저장 구조에 따라 클러스터형(clustered) 인덱스와 비클러스터형(nonclustered) 인덱스로 나뉜다. 여기서는 클러스터형 인덱스에 대해서만 설명하기로 한다. 클러스터형 인덱스는 두 가지 중요한 특징이 있다.

첫째, 인덱스의 리프 페이지가 곧 데이터 페이지다. 따라서 테이블 탐색에 필요한 레코드 식별자가 리프 페이지에 없다(인덱스 키 칼럼과 나머지 칼럼을 리프 페이지에 같이 저장하기 때문에 테이블을 랜덤 액세스할 필요가 없다). 클러스터형 인덱스의 리프 페이지를 탐색하면 해당 테이블의 모든 칼럼 값을 곧바로 얻을 수 있다. 흔히 클러스터형 인덱스를 사전에 비유한다. 예를 들어, 영한사전은 알파벳 순으로 정렬되어 있으며 각 단어 바로 옆에 한글 설명이 붙어 있다. 전문서적 끝 부분에 있는 찾아보기(=색인)가 페이지 번호만 알려주는 것과 비교하면 그 차이점을 알 수 있다.

둘째, 리프 페이지의 모든 로우(=데이터)는 인덱스 키 칼럼 순으로 물리적으로 정렬되어 저장된다. 테이블 로우는 물리적으로 한 가지 순서로만 정렬될 수 있다. 그러므로 클러스터형 인덱스는 테이블당 한 개만 생성할 수 있다.(전화번호부 한 권을 상호와 인명으로 동시에 정렬할 수 없는 것과 마찬가지다.)

[그림 Ⅱ-3-8]은 Employee ID, Last Name, First Name, Hire Date로 구성된 Employees 테이블에 대해 Employee ID에 기반한 클러스터형 인덱스를 생성한 모습이다. B-트리 구조를 편의상, 삼각형 모양을 왼쪽으로 90도 돌려서 나타냈다.

EmployeeID	LastName	FirstName	HireDate
1	Davolio	Nancy	1992-05-01 00:00:00.000
2	Fuller	Andrew	1992-08-14 00:00:00.000
3	Leverling	Janet	1992-04-01 00:00:00.000
4	peacock	Margaret	1993-05-03 00:00:00.000
5	Buchanan	Steven	1993-10-17 00:00:00.000
6	Suyama	Michael	1993-10-17 00:00:00.000
7	King	Robart	1994-01-02 00:00:00.000
8	Callahan	Laura	1994-03-05 00:00:00.000
9	Dodsworth	Anne	1994-11-15 00:00:00.000

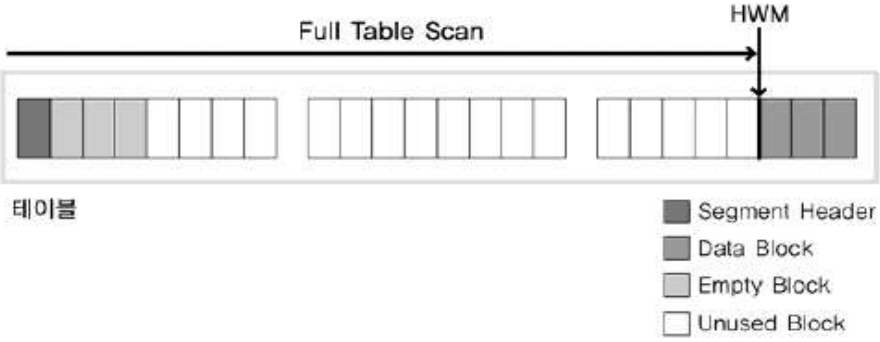
[그림 Ⅱ-3-8] Employees_pk 클러스터형 인덱스

[그림 Ⅱ-3-8]에 표시된 것처럼, 리프 블록에 인덱스 키 칼럼 외에도 테이블의 나머지 칼럼이 모두 함께 있다.

2. 전체 테이블 스캔과 인덱스 스캔

가. 전체 테이블 스캔

전체 테이블 스캔 방식으로 데이터를 검색한다는 것은 테이블에 존재하는 모든 데이터를 읽어 가면서 조건에 맞으면 결과로서 추출하고 조건에 맞지 않으면 버리는 방식으로 검색한다.



[그림 Ⅱ-3-9] 전체 테이블 스캔

Oracle의 경우 [그림 Ⅱ-3-9]와 같이 검색 조건에 맞는 데이터를 찾기 위해서 테이블의 고수위 마크(HWM, High Water Mark) 아래의 모든 블록을 읽는다. 고수위 마크는 테이블에 데이터가 쓰여졌던 블록 상의 최상위 위치(현재는 지워져서 데이터가 존재하지 않을 수

도 있음)를 의미한다. 전체 테이블 스캔 방식으로 데이터를 검색할 때 고수위 마크까지의 블록 내 모든 데이터를 읽어야 하기 때문에 모든 결과를 찾을 때까지 시간이 오래 걸릴 수 있다.

이와 같이 전체 테이블 스캔 방식은 테이블에 존재하는 모든 블록의 데이터를 읽는다. 그러나 이것은 결과를 찾기 위해 꼭 필요해서 모든 블록을? 모든 블록을 읽은 것이다. 따라서 이렇게 읽은 블록들은 재사용성이 떨어진다. 그래서 전체 테이블 스캔 방식으로 읽은 블록들은 메모리에서 곧 제거될 수 있도록 관리된다.

옵티마이저가 연산으로서 전체 테이블 스캔 방식을 선택하는 이유는 일반적으로 다음과 같다.

1) SQL문에 조건이 존재하지 않는 경우

SQL문에 조건이 존재하지 않는다는 것은 테이블에 존재하는 모든 데이터가 답이 된다는 것이다. 그렇기 때문에 테이블의 모든 블록을 읽으면서 무조건 결과로서 반환하면 된다.

2) SQL문의 주어진 조건에 사용 가능한 인덱스가 존재하는 않는 경우

사용 가능한 인덱스가 존재하지 않는다면 데이터를 액세스할 수 있는 방법은 테이블의 모든 데이터를 읽으면서 주어진 조건을 만족하는지를 검사하는 방법뿐이다. 또한 주어진 조건에 사용 가능한 인덱스는 존재하나 함수를 사용하여 인덱스 칼럼을 변형한 경우에도 인덱스를 사용할 수 없다.

3) 옵티마이저의 취사 선택

조건을 만족하는 데이터가 많은 경우, 결과를 추출하기 위해서 테이블의 대부분의 블록을 액세스해야 한다고 옵티마이저가 판단하면 조건에 사용 가능한 인덱스가 존재해도 전체 테이블 스캔 방식으로 읽을 수 있다.

4) 그 밖의 경우

병렬처리 방식으로 처리하는 경우 또는 전체 테이블 스캔 방식의 힌트를 사용한 경우에 전체 테이블 스캔 방식으로 데이터를 읽을 수 있다.

나. 인덱스 스캔

여기서는 데이터베이스에서 주로 사용되는 트리 기반 인덱스를 중심으로 설명한다. 인덱스 스캔은 인덱스를 구성하는 칼럼의 값을 기반으로 데이터를 추출하는 액세스 기법이다. 인덱스의 리프 블록은 인덱스 구성하는 칼럼과 레코드 식별자로 구성되어 있다. 따라서 검색을 위해 인덱스의 리프 블록을 읽으면 인덱스 구성 칼럼의 값과 테이블의 레코드 식별자를 알 수 있다. 인덱스에 존재하지 않는 칼럼의 값이 필요한 경우에는 현재 읽은 레코드 식별자를 이용하여 테이블을 액세스해야 한다. 그러나 SQL문에서 필요로 하는 모든 칼럼이 인덱스 구성 칼럼에 포함된 경우 테이블에 대한 액세스는 발생하지 않는다.

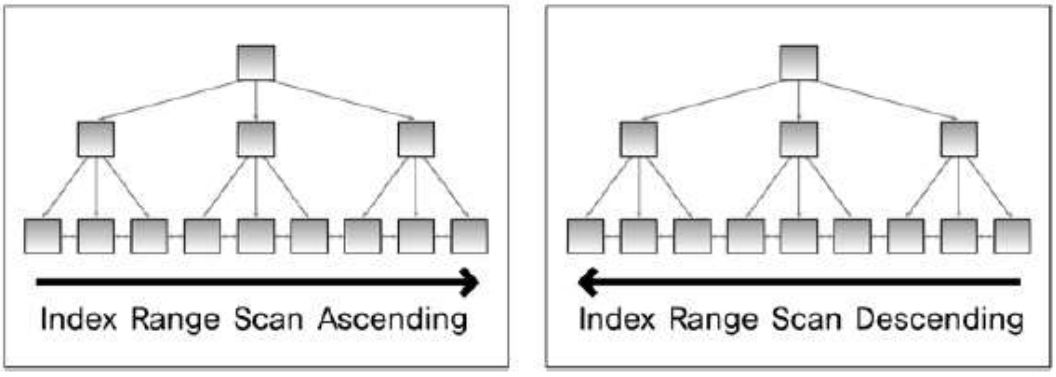
인덱스는 인덱스 구성 칼럼의 순서로 정렬되어 있다. 인덱스의 구성 칼럼이 A+B라면 먼

저 칼럼 A로 정렬되고 칼럼 A의 값이 동일할 경우에는 칼럼 B로 정렬된다. 그리고 칼럼 B까지 모두 동일하면 레코드 식별자로 정렬된다. 인덱스가 구성 칼럼으로 정렬되어 있기 때문에 인덱스를 경유하여 데이터를 읽으면 그 결과 또한 정렬되어 반환된다. 따라서 인덱스의 순서와 동일한 정렬 순서를 사용자가 원하는 경우에는 정렬 작업을 하지 않을 수 있다.

인덱스 스캔 중에서 자주 사용되는 인덱스 유일 스캔(Index Unique Scan), 인덱스 범위 스캔(Index Range Scan), 인덱스 역순 범위 스캔(Index Range Scan Descending)에 대해 간단히 설명하면 다음과 같다. 제공되는 인덱스 스캔 방식은 데이터베이스 벤더마다 다를 수 있다.

1) 인덱스 유일 스캔은 유일 인덱스(Unique Index)를 사용하여 단 하나의 데이터를 추출하는 방식이다. 유일 인덱스는 중복을 허락하지 않는 인덱스이다. 유일 인덱스 구성 칼럼에 모두 '='로 값이 주어지면 결과는 최대 1건이 된다. 인덱스 유일 스캔은 유일 인덱스 구성 칼럼에 대해 모두 '='로 값이 주어진 경우에만 가능한 인덱스 스캔 방식이다.

2) 인덱스 범위 스캔은 인덱스를 이용하여 한 건 이상의 데이터를 추출하는 방식이다. 유일 인덱스의 구성 칼럼 모두에 대해 '='로 값이 주어지지 않은 경우와 비유일 인덱스(Non-Unique Index)를 이용하는 모든 액세스 방식은 인덱스 범위 스캔 방식으로 데이터를 액세스하는 것이다. [그림 Ⅱ-3-10]의 왼쪽 그림과 같은 방식으로 인덱스를 읽는다.



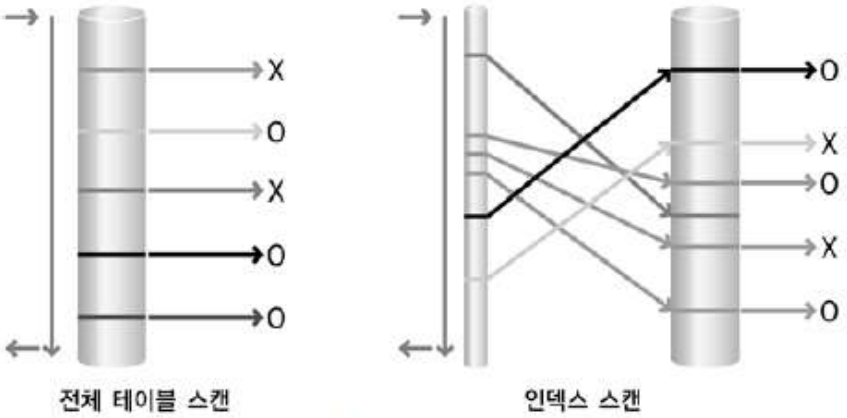
[그림 Ⅱ-3-10] 인덱스 오름/내림 차순 범위 스캔

3) 인덱스 역순 범위 스캔은 [그림 Ⅱ-3-10]의 오른쪽 그림과 같이 인덱스의 리프 블록의 양방향 링크를 이용하여 내림 차순으로 데이터를 읽는 방식이다. 이 방식을 이용하여 최대값(Max Value)을 쉽게 찾을 수 있다. 이 또한 인덱스 범위 스캔의 일종이다.

이외에도 인덱스 전체 스캔(Index Full Scan), 인덱스 고속 전체 스캔(Fast Full Index Scan), 인덱스 스킵 스캔(Index Skip Scan) 등이 존재한다.

다. 전체 테이블 스캔과 인덱스 스캔 방식의 비교

데이터를 액세스하는 방법은 크게 두 가지로 나눠 볼 수 있다. 인덱스를 경유해서 읽는 인덱스 스캔 방식과 테이블의 전체 데이터를 모두 읽으면서 데이터를 추출하는 전체 테이블 스캔 방식이다. 인덱스 스캔 방식은 사용 가능한 적절한 인덱스가 존재할 때만 이용할 수 있는 스캔 방식이지만 전체 테이블 스캔 방식은 인덱스의 존재 유무와 상관없이 항상 이용 가능한 스캔 방식이다. 앞에서 설명한 것처럼 옵티마이저는 인덱스가 존재하더라도 전체 테이블 스캔 방식을 취사 선택할 수 있다. [그림 Ⅱ-3-11]은 전체 테이블 스캔과 인덱스 스캔에 대한 SQL 처리 흐름도 표현의 예이다.



[그림 Ⅱ-3-11] 전체 테이블 스캔과 인덱스 스캔에 대한 SQL 처리 흐름도 표현 예시

인덱스 스캔은 인덱스에 존재하는 레코드 식별자를 이용해서 검색하는 데이터의 정확한 위치를 알고서 데이터를 읽는다. 그렇기 때문에 인덱스 스캔 방식에서는 불필요하게 다른 블록을 더 읽을 필요가 없다. 따라서 한번의 I/O 요청에 한 블록씩 데이터를 읽는다. 그러나 전체 테이블 스캔은 데이터를 읽을 때 한번의 I/O 요청으로 여러 블록을 한꺼번에 읽는다. 어차피 테이블의 모든 데이터를 읽을 것이라면 한 번 읽기 작업을 할 때 여러 블록을 함께 읽는 것이 효율적이다.

대용량 데이터 중에서 극히 일부의 데이터를 찾을 때, 인덱스 스캔 방식은 인덱스를 이용해 몇 번의 I/O만으로 원하는 데이터를 쉽게 찾을 수 있다. 그러나 전체 테이블 스캔은 테이블의 모든 데이터를 읽으면서 원하는 데이터를 찾아야 하기 때문에 비효율적인 검색을 하게 된다. 그러나 반대로 테이블의 대부분의 데이터를 찾을 때는 한 블록씩 읽는 인덱스 스캔 방식 보다는 어차피 대부분의 데이터를 읽을 거라면 한번에 여러 블록씩 읽는 전체 테이블 스캔 방식이 유리할 수 있다.

제3절 조인 수행 원리

조인이란 두 개 이상의 테이블을 하나의 집합으로 만드는 연산이다. SQL문에서 FROM 절에 두 개 이상의 테이블이 나열될 경우 조인이 수행된다. 조인 연산은 두 테이블 사이에서 수행된다. FROM 절에 A, B, C라는 세 개의 테이블이 존재하더라도 세 개의 테이블이 동시에 조인이 수행되는 것은 아니다. 세 개의 테이블 중에서 먼저 두 개의 테이블에 대해 조인이 수행된다. 그리고 먼저 수행된 조인 결과와 나머지 테이블 사이에서 조인이 수행된다. 이러한 작업은 FROM 절에 나열된 모든 테이블을 조인할 때까지 반복 수행한다.

예를 들어, A, B, C 세 개의 테이블을 조인할 때를 가정으로 설명하면 다음과 같다. 먼저 A와 B 두 테이블을 먼저 조인하면 해당 조인 결과와 나머지 C 테이블을 조인한다(A → B → C). 만약, A와 C 테이블을 먼저 조인한다면 해당 조인 결과와 나머지 B 테이블을 조인한다(A → C → B). 테이블 또는 조인 결과를 이용하여 조인을 수행할 때 조인 단계별로 다른 조인 기법을 사용할 수 있다. 예를 들어, A와 B 테이블을 조인할 때는 NL Join 기법을 사용하고 해당 조인 결과와 C 테이블을 조인할 때는 Hash Join 기법을 사용할 수 있다.

조인 기법은 두 개의 테이블을 조인할 때 사용할 수 있는 방법이다. 여기서는 조인 기법 중에서 자주 사용되는 NL Join, Hash Join, Sort Merge Join에 대해서 조인 원리를 간단하게 설명한다.

1. NL Join

NL Join은 프로그래밍에서 사용하는 중첩된 반복문과 유사한 방식으로 조인을 수행한다. 반복문의 외부에 있는 테이블을 선행 테이블 또는 외부 테이블(Outer Table)이라고 하고, 반복문의 내부에 있는 테이블을 후행 테이블 또는 내부 테이블(Inner Table)이라고 한다.

```
FOR 선행 테이블 읽음 → 외부 테이블(Outer Table)
    FOR 후행 테이블 읽음 → 내부 테이블(Inner Table)
        (선행 테이블과 후행 테이블 조인)
```

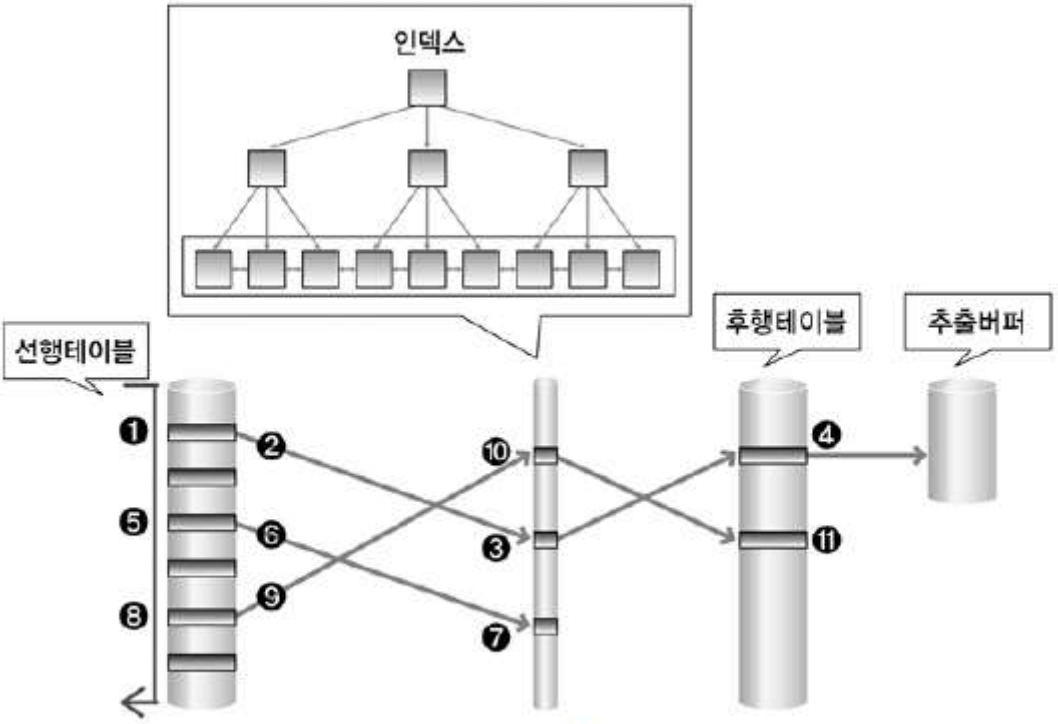
먼저 선행 테이블의 조건을 만족하는 행을 추출하여 후행 테이블을 읽으면서 조인을 수행한다. 이 작업은 선행 테이블의 조건을 만족하는 모든 행의 수만큼 반복 수행한다. NL Join에서는 선행 테이블의 조건을 만족하는 행의 수가 많으면(처리 주관 범위가 넓으면), 그 만큼 후행 테이블의 조인 작업은 반복 수행된다. 따라서 결과 행의 수가 적은(처리 주관 범위가 좁은) 테이블을 조인 순서상 선행 테이블로 선택하는 것이 전체 일량을 줄일 수 있다. NL Join은 랜덤 방식으로 데이터를 액세스하기 때문에 처리 범위가 좁은 것이 유리하다.

NL Join의 작업 방법은 다음과 같다.

① 선행 테이블에서 주어진 조건을 만족하는 행을 찾음

② 선행 테이블의 조인 키 값을 가지고 후행 테이블에서 조인 수행

③ 선행 테이블의 조건을 만족하는 모든 행에 대해 1번 작업 반복 수행



[그림 Ⅱ-3-12] NL Join

[그림 Ⅱ-3-12]의 예를 이용하여 NL Join의 수행 방식을 알아보도록 하자. [그림 Ⅱ-3-12]에서 인덱스는 B-트리 인덱스의 리프 블록만을 그린 것임을 표현한 것이다.

① 선행 테이블에서 조건을 만족하는 첫 번째 행을 찾음

→ 이때 선행 테이블에 주어진 조건을 만족하지 않는 경우 해당 데이터는 필터링 됨

② 선행 테이블의 조인 키를 가지고 후행 테이블에 조인 키가 존재하는지 찾으러 감

→ 조인 시도

③ 후행 테이블의 인덱스에 선행 테이블의 조인 키가 존재하는지 확인

→ 선행 테이블의 조인 값이 후행 테이블에 존재하지 않으면 선행 테이블 데이터는 필터링 됨 (더 이상 조인 작업을 진행할 필요 없음)

④ 인덱스에서 추출한 레코드 식별자를 이용하여 후행 테이블을 액세스

→ 인덱스 스캔을 통한 테이블 액세스

후행 테이블에 주어진 조건까지 모두 만족하면 해당 행을 추출버퍼에 넣음

⑤ ~ ⑪ 앞의 작업을 반복 수행함

추출버퍼는 SQL문의 실행결과를 보관하는 버퍼로서 일정 크기를 설정하여 추출버퍼에 결과가 모두 차거나 더 이상 결과가 없어서 추출버퍼를 채울 것이 없으면 결과를 사용자에게 반환한다. 추출버퍼는 운반단위, Array Size, Prefetch Size라고도 한다.

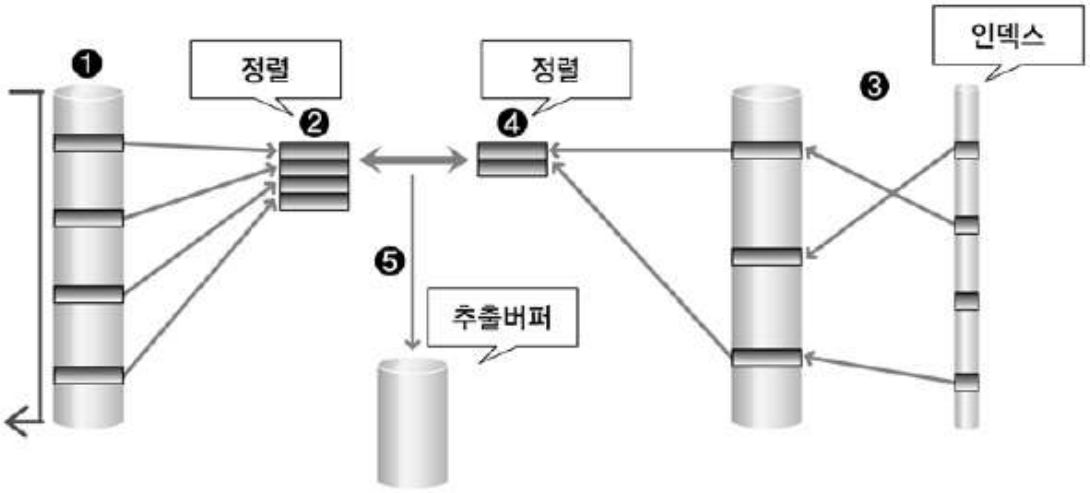
[그림 Ⅱ-3-12]에서 만약 선행 테이블에 사용 가능한 인덱스가 존재한다면 인덱스를 통해 선행 테이블을 액세스할 수 있다. (여기서는 사용할 인덱스가 없음을 가정으로 설명한

것임) NL Join 기법은 조인이 성공하면 바로 조인 결과를 사용자에게 보여 줄 수 있다. 그래서 결과를 가능한 빨리 화면에 보여줘야 하는 온라인 프로그램에 적당한 조인 기법이다.

2. Sort Merge Join

Sort Merge Join은 조인 칼럼을 기준으로 데이터를 정렬하여 조인을 수행한다. NL Join은 주로 랜덤 액세스 방식으로 데이터를 읽는 반면 Sort Merge Join은 주로 스캔 방식으로 데이터를 읽는다. Sort Merge Join은 랜덤 액세스로 NL Join에서 부담이 되던 넓은 범위의 데이터를 처리할 때 이용되던 조인 기법이다. 그러나 Sort Merge Join은 정렬할 데이터가 많아 메모리에서 모든 정렬 작업을 수행하기 어려운 경우에는 임시 영역(디스크)을 사용하기 때문에 성능이 떨어질 수 있다.

일반적으로 대량의 조인 작업에서 정렬 작업을 필요로 하는 Sort Merge Join 보다는 CPU 작업 위주로 처리하는 Hash Join이 성능상 유리하다. 그러나 Sort Merge Join은 Hash Join과는 달리 동등 조인 뿐만 아니라 비동등 조인에 대해서도 조인 작업이 가능하다는 장점이 있다.



[그림 Ⅱ-3-13] Sort Merge Join

Sort Merge Join의 동작은 [그림 Ⅱ-3-13]과 같다.

① 선행 테이블에서 주어진 조건을 만족하는 행을 찾음

② 선행 테이블의 조인 키를 기준으로 정렬 작업을 수행

① ~ ②번 작업을 선행 테이블의 조건을 만족하는 모든 행에 대해 반복 수행

③ 후행 테이블에서 주어진 조건을 만족하는 행을 찾음

④ 후행 테이블의 조인 키를 기준으로 정렬 작업을 수행

③ ~ ④번 작업을 후행 테이블의 조건을 만족하는 모든 행에 대해 반복 수행

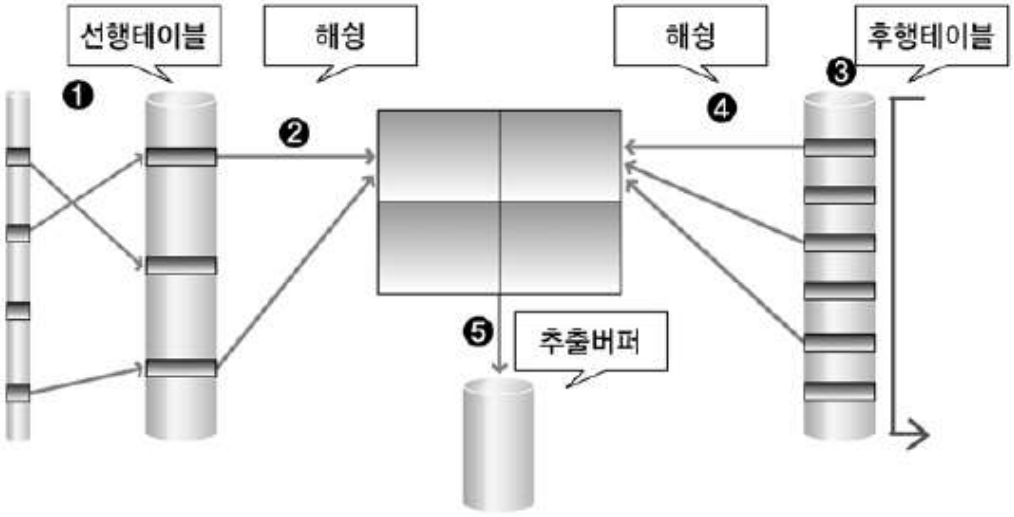
⑤ 정렬된 결과를 이용하여 조인을 수행하며 조인에 성공하면 추출버퍼에 넣음

Sort Merge Join은 조인 칼럼의 인덱스를 사용하지 않기 때문에 조인 칼럼의 인덱스가 존재하지 않을 경우에도 사용할 수 있는 조인 기법이다. Sort Merge Join에서 조인 작업을 위해 항상 정렬 작업이 발생하는 것은 아니다. 예를 들어, 조인할 테이블 중에서 이미 앞 단계의 작업을 수행하는 도중에 정렬 작업이 미리 수행되었다면 조인을 위한 정렬 작업은 발생하지 않을 수 있다.

3. Hash Join

Hash Join은 해싱 기법을 이용하여 조인을 수행한다. 조인을 수행할 테이블의 조인 칼럼을 기준으로 해쉬 함수를 수행하여 서로 동일한 해쉬 값을 갖는 것들 사이에서 실제 값이 같은지를 비교하면서 조인을 수행한다.

Hash Join은 NL Join의 랜덤 액세스 문제점과 Sort Merge Join의 문제점인 정렬 작업의 부담을 해결 위한 대안으로 등장하였다.



[그림 Ⅱ-3-14] Hash Join

Hash Join의 동작은 [그림 Ⅱ-3-14]와 같다.

① 선행 테이블에서 주어진 조건을 만족하는 행을 찾음

② 선행 테이블의 조인 키를 기준으로 해쉬 함수를 적용하여 해쉬 테이블을 생성

→ 조인 칼럼과 SELECT 절에서 필요로 하는 칼럼도 함께 저장됨

① ~ ②번 작업을 선행 테이블의 조건을 만족하는 모든 행에 대해 반복 수행

③ 후행 테이블에서 주어진 조건을 만족하는 행을 찾음

④ 후행 테이블의 조인 키를 기준으로 해쉬 함수를 적용하여 해당 버킷을 찾음

→ 조인 키를 이용해서 실제 조인될 데이터를 찾음

⑤ 조인에 성공하면 추출버퍼에 넣음

③ ~ ⑤번 작업을 후행 테이블의 조건을 만족하는 모든 행에 대해서 반복 수행

Hash Join은 조인 칼럼의 인덱스를 사용하지 않기 때문에 조인 칼럼의 인덱스가 존재하지 않을 경우에도 사용할 수 있는 조인 기법이다. Hash Join은 해쉬 함수를 이용하여 조인을 수행하기 때문에 '='로 수행하는 조인 즉, 동등 조인에서만 사용할 수 있다. 해쉬 함수를 적용한 값은 어떤 값으로 해쉉될 지 알 수 없다. 해쉬 함수가 적용될 때 동일한 값은 항상 같은 값으로 해쉉됨이 보장된다. 그러나 해쉬 함수를 적용할 때 보다 큰 값이 항상 큰 값으로 해쉉되고 작은 값이 항상 작은 값으로 해쉉된다는 보장은 없다. 그렇기 때문에 Hash Join은 동등 조인에서만 사용할 수 있다.

[그림 Ⅱ-3-14]와 같이 Hash Join은 조인 작업을 수행하기 위해 해쉬 테이블을 메모리에 생성해야 한다. 생성된 해쉬 테이블의 크기가 메모리에 적재할 수 있는 크기보다 더 커지면 임시 영역(디스크)에 해쉬 테이블을 저장한다. 그러면 추가적인 작업이 필요해 진다. 그렇기 때문에 Hash Join을 할 때는 결과 행의 수가 적은 테이블을 선행 테이블로 사용하는 것이 좋다. 선행 테이블의 결과를 완전히 메모리에 저장할 수 있다면 임시 영역에 저장하는 작업이 발생하지 않기 때문이다.

Hash Join에서는 선행 테이블을 이용하여 먼저 해쉬 테이블을 생성한다고 해서 선행 테이블을 Build Input이라고도 하며, 후행 테이블은 만들어진 해쉬 테이블에 대해 해쉬 값의 존재여부를 검사한다고 해서 Prove Input이라고도 한다.

[별첨] 제30~35회 기출문제

제35회 SQL 개발자(SQLD) 자격검정 기출문제

1과목 문제 및 정답 링크 : https://cafe.naver.com/sqlpd/10736
2과목 문제 및 정답 링크 : https://cafe.naver.com/sqlpd/10740
단답형 문제 및 정답 링크 : https://cafe.naver.com/sqlpd/10738

제34회 SQL 개발자(SQLD) 자격검정 기출문제

전체 문제 및 정답 링크 : https://cafe.naver.com/sqlpd/10604

제33회 SQL 개발자(SQLD) 자격검정 기출문제

1과목 문제 및 정답 링크 : https://cafe.naver.com/sqlpd/9118
단답형 문제 및 정답 링크 : https://cafe.naver.com/sqlpd/9117

제32회 SQL 개발자(SQLD) 자격검정 기출문제

1과목 문제 및 정답 링크 : https://cafe.naver.com/sqlpd/8505
2과목 문제 및 정답 링크 : https://cafe.naver.com/sqlpd/8507
단답형 문제 및 정답 링크 : https://cafe.naver.com/sqlpd/8495

제31회 SQL 개발자(SQLD) 자격검정 기출문제

1과목 문제 및 정답 링크 : https://cafe.naver.com/sqlpd/7927
2과목 문제 및 정답 링크 : https://cafe.naver.com/sqlpd/7946
단답형 문제 및 정답 링크 : https://cafe.naver.com/sqlpd/7926

제30회 SQL 개발자(SQLD) 자격검정 기출문제

1과목 문제 및 정답 링크 : https://cafe.naver.com/sqlpd/7596
2과목 문제 및 정답 링크 : https://cafe.naver.com/sqlpd/7597
단답형 문제 및 정답 링크 : https://cafe.naver.com/sqlpd/7598