# SQLD 개념 요약

## 과목 1 - 데이터 모델링의 이해

### 제 1장. 데이터 모델링의 이해

제 1절 데이터 모델의 이해
제 2절 엔터티
제 3절 속성
제 4절 관계
제 5절 식별자

### 제 2장. 데이터 모델과 성능

제 1절 성능 데이터 모델링의 개요
제 2절 정규화와 성능
제 3절 반정규화와 성능
제 4절 대량 데이터에 따른 성능
제 5절 데이터베이스 구조와 성능
제 6절 분산 데이터베이스와 성능

## 과목 2 - SQL 기본 및 활용

### 제 1장. SQL 기본

제 1절 관계형 데이터베이스 개요
제 2절 DDL
제 3절 DML
제 4절 DCL
제 5절 WHERE 절
제 6절 함수
제 7절 GROUP BY, HAVING 절
제 8절 ORDER BY 절
제 9절 조인

### 제 2장. SQL 활용

제 1절 표준 조인
제 2절 집합 연산자
제 3절 계층형 질의와 셀프조인
제 4절 서브쿼리
제 5절 그룹함수
제 6절 윈도우 함수
제 7절 DCL
제 8절 절차형 SQL

### 제 3장. SQL 최적화 기본 원리

제 1절 옵티마이저와 실행계획
제 2절 인덱스 기본
제 3절 조인 수행원리

# 1과목 데이터 모델링의 이해

## 1-1. 데이터 모델링의 이해

### 1-1-1. 데이터 모델의 이해

*** 모델링**

일정한 표기법에 의해 규칙을 가지고 표기하는 것
( 커뮤니케이션 효율성 극대화한 고급화된 표현방법 )

*** 모델링 특징 3가지**

**추상화** (모형화, 가설적) : 현실세계 일정한 형식에 맞추어 표현 (일정한 양식 표기법)
**단순화** : 복잡한 헌실세계 약속된 규약에 의해 제한된 표기법/언어로 표현 쉽게 이해
**명확화** : 누구나 이해하기 쉽게 대상의 애매모호함 제거 정확하게 현상을 기술

*** 정보시스템 구축에서 모델링 활용**

계획/분석/설계 할 때 업무를 분석하고 설계하는데 이용
구축/운영 단계에서는 변경과 관리의 목적으로 이용

*** 모델링 관점 3가지**

**데이터 관점** (Data, What) : 업무와 데이터 or 데이터간의 관계
**프로세스 관점** (Process, How) : 업무가 실제 하는 일 or 무엇을 해야 하는지
**상관 관점** (Interaction) : 업무가 처리하는 일의 방법에 따라 데이터가 받는 영향

*** 데이터 모델링 정의**

정보시스템 구축 위한 데이터 관점 업부 분석기법
현실세계 데이터 약속된 표기법에 의해 표현
데이터베이스 구축 위한 분석/설계 과정

*** 데이터 모델링 기능**

시스템 가시화 도움
시스템 구조와 행동 명세화 가능
시스템 구축 구조화된 틀 제공
시스템 구축 과정에서 결정한 것 문서화
다양한 영역 집중 위해 다른 영역 세부사항 숨김 (다양한 관점 제공)
특정 목표에 따라 구체화된 상세 수준의 표현방법을 제공

*** 데이터 모델링 중요성**

**파급효과**가 크다 (Leverage) : 데이터 구조 변경으로 인한 일련의 변경작업 위험요소 해결
복잡한 정보 요구사항의 **간결한 표현** (Conciseness) : 요구사항, 한계 명확하고 간결하게
**데이터 품질**을 유지 (Data Quality) : 오래된 데이터의 정확성, 신뢰성 해결

*** 데이터 모델링 유의점 3가지**

**중복** (Duplication) : 데이터베이스가 여러 장소에 같은 정보 저장하는 것 주의
**비유연성** (Inflexibility) : 사소한 업무변화에 데이터모델 수시로 변경되면 유지보수 어려움
→ 데이터의 정의를 데이터의 사용 프로세스와 분리
**비일관성** (Inconsistency) : 데이터의 중복이 없어도 비일관성 발생 가능
→ 모델링 할 때 데이터간 상호 연관관계 명확히 정의

*** 데이터 모델링의 진행 3단계**

[ 개념적 (추상) → 논리적 → 물리적 (구체) ]

**개념** (계획/분석) : 추상화, 업무중심, 포괄, 전사적, EA수립 추상적
**논리** (분석) : [KEY, 속성, 관계] 표현, 재사용성, 정규화 ↓
**물리** (설계) : 데이터베이스 이식, 성능/저장 등 물리적 성격 고려 구체적

*** 프로젝트 생명주기(Life Cycle)에서 데이터 모델링**

Waterfall 기반 : 분석과 설계단계로 구분되어 명확하게 정의
정보공학/구조적 방법론 : 분석단계 - 업무중심 논리적 모델링
설계단계 - 하드웨어 및 성능 고려한 물리적 모델링
나선형 모델 (RUP, 마르미) : 업무크기 따라 논리적/물리적 모델링이 분석/설계 양쪽 수행
비중은 분석단계에서 논리적 모델이 더 많이 수행
데이터/애플리케이션 축 구분 진행, 상호검증, 단계별 완성도

일반적으로 [계획/분석 : 개념] [분석 : 논리] [설계 : 물리] but 현실에서 보통 개념 생략
객체지향 개념은 데이터/프로세스 모델링 구분 X 일체형으로 진행

*** 데이터 독립성 필요성**

유지보수 비용 증가
데이터 중복성 증가
데이터 복잡성 증가
요구사항 대응 저하

*** 데이터 독립성 효과**

각 View의 독립성 유지, 계층별 View에 영향 주지 않고 변경 가능.
단계별 Schema에 따라 DDL과 DML의 다름을 제공
구조 / 독립성/ 사상(Mapping)

*** ANSI / SPARC 데이터베이스 3단계 구조 및 독립성**

**외부단계 / 개념단계 / 내부적** 단계 (서로 간섭 X)

**논리적 독립성** (외부-개념) : 개념스키마 변경 → 외부스키마 영향 X
( 사용자 특성에 맞는 변경 가능, 통합 구조 변경가능 )
**물리적 독립성** (개념-내부) : 내부스키마 변경 → 외부/개념스키마 영향 X
( 물리적 구조, 개념 구조 상호간 영향 없이 서로 변경가능 )

데이터 모델링은 통합관점 뷰를 가지는 개념 스키마를 만들어가는 과정 !!

*** 사상 (Mapping)**

상호 독립적인 개념을 연결시켜주는 다리
**논리적 사상** : 외부화면 및 사용자 인터페이스 스키마 구조는 개념스키마와 연결됨
**물리적 사상** : 개념스키마 구조와 물리적 저장된 구조(테이블스페이스)와 연결됨

*** 좋은 데이터 모델의 요소 6가지**

**완전성 / 중복배제 / 업무규칙 / 데이터 재사용 / 의사소통 / 통합성**

*** 데이터 모델링의 요소 3가지**

업무가 관여하는 **어떤 것** (Thing) : 엔터티타입, 엔터티 / 엔터티, 인스턴스, 어커런스
그것이 가지는 **성격** (Attributes) : 속성 / 속성값
그것들 간의 **관계** (Relationships) : 관계 / 페어링

**1-1-2. 엔터티**

*** 엔터티의 개념**

명사, 업무상 관리 필요한 관심사, 저장이 되기 위한 어떤 것(Thing)

*** 엔터티의 특징**

업무에서 꼭 필요한 정보
유일한 식별자로 식별이 가능해야 함
인스턴스 2개 이상의 집합
업무 프로세스에 의해 이용됨
반드시 속성이 있어야 함 (주식별자만 존재하고 일반속성 없어도 적절X, 관계엔터티 예외)
다른 엔터티와의 관계가 최소 1개 이상 존재 (통계성, 코드성, 내부필요 엔터티 예외)

*** 엔터티의 분류**

**유무형 : 유형, 개념, 사건**
**발생시점 : 기본 → 중심 → 행위**

*** 엔터티의 명명**

1 가능한 현업 업무에서 사용하는 용어 사용
2 가능하면 약어 사용 X
3 단수명사 사용
4 모든 엔터티에서 유일한 이름 부여
5 생성 의미대로 이름 부여

**1-1-3. 속성**

*** 속성의 개념**

업무에서 필요, 의미상 더이상 분리 X, 인스턴스의 구성요소 (엔터티 설명)

*** 인스턴스 - 속성 - 엔터티의 관계**

1개의 엔터티는 2개 이상의 인스턴스 집합이어야 함
1개의 인스턴스는 2개 이상의 속성을 가짐
1개의 속성은 1개의 속성값을 가짐

*** 속성의 특징**

반드시 해당 업무에서 필요하고 관리하고자하는 정보
정규화 이론에 근간, 정해진 주식별자에 함수적 종속성을 가져야 함
하나의 속성에는 1개의 값만을 가짐. 다중값일 경우 별도의 엔터티 이용하여 분리

*** 속성의 분류**

특성에 따른 분류 : **기본속성** - 업무로부터 추출한 모든 속성
**설계속성** - 새로 만들거나 정의하는 속성 (코드성, 일련번호 등)
**파생속성** - 다른 속성의 영향 받아 발생 (계산된 값 등)

구성 방식에 따른 분류 : **PK(기본키)** / **FK(외래키)** / **일반속성**

*** 도메인**

속성이 가질 수 있는 값의 범위. [데이터타입/크기/제약사항] 지정

*** 속성의 명명**

해당 업무에서 사용하는 이름 부여
서술식 속성명 사용 X
약어 사용 가급적 X
전체 데이터 모델에서 유일성 확보하는 것이 좋음

**1-1-4. 관계**

*** 관계의 정의**

인스턴스 사이의 논리적인 연관성. 존재 or 행위로서 서로에게 연관성이 부여된 상태.

*** 관계와 페어링**

인스턴스간의 개별적 관계 = 페어링 → 이것의 집합을 논리적 표현 = 관계

*** 관계의 분류**

존재에 의한 관계 / 행위에 의한 관계

*** 관계의 표기법**

관계명 : 엔터티가 관계에 참여하는 형태. 각 관계는 2개의 관계명 및 관점을 가짐
관계차수 : 1:1, 1:M, M:M (관계 엔터티 이용)
관계선택사양 : 필수참여(Mandatory), 선택참여(Optional)

*** 관계의 정의**

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?

*** 관계의 읽기**

각 / 기준엔터티 / 관계차수 / 관련엔터티 / 필수or선택 / 관계명

| | | | | | | |
|---|---|---|---|---|---|---|
| ex) | 각각의 | 사원은 | 한 | 부서에 | 때때로 | 속한다 |
| | 각 | 부서에는 | 여러 | 사원이 | 항상 | 소속된다 |

**1-1-5. 식별자**

*** 식별자 개념**

엔터티를 구분짓는 논리적인 이름
엔터티를 대표할 수 있는 속성
엔터티에는 반드시 하나의 유일한 식별자가 존재함

*** 식별자 특징**

**유일성** : 주식별자에 의해 엔터티내에 모든 인스턴스 유일하게 구분함
**최소성** : 주식별자를 구성하는 속성의 수는 유일성을 만족하는 최소의 수여야 함
**불변성** : 주식별자가 한 번 특정 엔터티에 지정되면 그 식별자 값은 변하지 않아야 함
**존재성** : 주식별자가 지정되면 반드시 데이터 값이 존재해야 함 (Null X)

*** 식별자 분류**

**대표성 여부 : 주식별자 / 보조식별자**
**스스로 생성 여부 : 내부식별자 / 외부식별자**
**속성의 수 : 단일식별자 / 복합식별자**
**대체 여부 : 본질식별자 / 인조식별자**

*** 주식별자 도출 기준**

해당 업무에서 자주 이용되는 속성을 지정
명칭, 내역 등과 같이 이름으로 기술되는 것들은 피함
→ 구분자가 존재하지 않을 경우 새로운 식별자 생성 (일련번호, 코드 등)
복합으로 주식별자 구성할 경우 너무 많은 속성 포함되지 않도록
→ 주식별자 개수가 많을 경우 새로운 인조식별자를 생성

*** 식별자와 비식별자 관계**

**# 식별자 관계**

부모의 주식별자 → 자식의 주식별자로 상속 / Null X / 1:1 or 1:M 관계
강한 연결관계, 부모에 종속, 상속받은 주식별자 타 엔터티 이전 필요

문제 - 주식별자 속성이 지속적 증가 가능 → 복잡성과 오류가능성 유발

**# 비식별자 관계**

부모로부터 속성을 받아 일반속성으로 사용 (약한 종속)
상속받은 속성 타 엔터티에 차단, 부모쪽 관계참여 선택관계

**- 사용하는 경우 -**

- 부모 없는 자식 생성 가능한 경우
- 엔터티별로 데이터 생명주기 다르게 관리하는 경우
- 여러개 엔터티가 하나로 통합 표현, 각각 별도의 관계 가질 경우
- 자식 엔터티에서 별도의 주식별자 생성이 더 유리할 경우

문제 - 속성이 자식에게 상속X → 부모엔터티까지 조인 현상 발생 →
불필요한 조인 발생, SQL구문 길어져 성능 저하

## 1-2. 데이터 모델과 성능

### 1-2-1. 성능 데이터 모델링의 개요

*** 성능 데이터 모델링의 정의**

세가지 경우 고려 : 데이터모델 구조, 대용량의 데이터, 인덱스 특성

반정규화만을 의미 X, 정규화 또는 인덱스 특징 고려한 칼럼 순서 변형도 가능,
데이터 특성에 따라 정규화된 모델의 테이블 수직 또는 수평분할 방법,
논리적 테이블을 물리적 전환할 때 데이터 처리 성격에 따라 변환

*** 성능 데이터 모델링 수행시점**

사전에 할수록 비용절감이 가능 (분석/설계 단계에서 하는 것이 Best - 재업무비용 최소화)

*** 성능 데이터 모델링 고려사항**

데이터 모델링할 때 정규화 수행
데이터베이스 용량산정 수행
데이터베이스에 발생되는 트랜잭션 유형 파악
용량과 트랜잭션 유형에 따라 반정규화 수행
이력모델 조정, PK/FK 조정, 슈퍼타입/서브타입 조정 수행
성능관점에서 데이터 모델 검증

**1-2-2. 정규화와 성능**

*** 정규화**

다양한 유형의 검사를 통해 데이터 모델을 좀 더 구조화하고 개선시켜 나가는 절차
중복성 제거, 관심사별로 처리되는 경우가 많아 성능이 향상됨

성능 = [ **조회 / 입력, 수정, 삭제** ] 2가지로 분류

결정자에 의해 함수적 종속 가진 일반속성을 의존자로 하여 입력/수정/삭제이상 제거
중복속성 제거, 결정자에 의해 동일한 의미의 일반속성 하나의 테이블로 집약
한 테이블의 데이터 용량 최소화, 입력/수정/삭제 → 향상, 조회 → 향상 or 저하

반정규화된 테이블의 성능이 더 떨어지는 경우 존재

## 1-2-3. 반정규화와 성능

### * 반정규화의 정의

정규화된 엔터티, 속성, 관계에 대하여 시스템의 성능향상, 개발과 운영 단순화를 위하여 중복, 통합, 분리 등을 수행하는 데이터 모델링 기법 (데이터 중복하여 성능 향상)

무결성 깨질 위험 감수 → 디스크 I/O량 감소, 긴 경로의 조인으로 인한 성능저하 해결
중복성의 원리를 활용하여 데이터 조회시 성능 향상

정규화도 조회 성능 향상 But 일부 여러 조인 걸려야 데이터 가저오는 경우 (비식별관계)
→ 조회에 대한 처리성능 중요하다고 판단되면 부분적 반정규화
정규화의 종속관계는 위반 X, 데이터 중복성을 증가시켜 조회 성능 향상

프로젝트의 설계 단계에서 반정규화를 적용 (구축 및 시험단계에서 적용시 노력비용↑)

### * 반정규화 적용방법

반정규화 대상조사 : 범위처리 빈도수, 대량 범위처리, 통계성 프로세스, 테이블 조인 개수
↓
다른 방법 검토 : 뷰 테이블, 클러스터링, 인덱스 조정, 응용애플리케이션
↓
반정규화 적용 : 테이블, 속성, 관계의 반정규화

### * 반정규화의 기법

#### # 테이블 반정규화

테이블 병합 : 1:1 관계 병합, 1:M 관계 병합, 슈퍼/서브타입 병합
테이블 분할 : 수직분할 (칼럼단위, 테이블 1:1로 분리), 수평분할 (로우단위)
테이블 추가 : 중복테이블 - 동일한 테이블구조 중복 원격조인 제거
통계테이블 - SUM, AVG 등 미리 계산, 조회시 성능 향상
이력테이블 - 마스터 테이블 레코드 중복하여 이력테이블에 존재
부분테이블 - 전체 칼럼 중 자주 이용하는 칼럼들을 모아놓은 별도 테이블

**# 칼럼 반정규화**

중복칼럼 추가 : 조인을 감소시키기 위해 중복 칼럼 위치시킴
파생칼럼 추가 : 계산에 의한 성능저하 예방 미리 계산하여 칼럼에 보관 (Derived 컬럼)
이력테이블 칼럼추가 : 기능성 칼럼 (최근값 여부, 시작 및 종료일자 등) 추가
PK에 의한 칼럼추가 : 복합의미 PK 단일속성 구성시 발생. 일반속성으로 PK데이터 추가
응용시스템 오작동 위한 칼럼추가 : 이전 데이터 임시적 중복보관 (원래값 복구 용도)

**# 관계 반정규화**

중복관계 추가 : 여러 경로를 거친 조인을 방지하기 위해 추가적인 관계를 맺음

테이블과 칼럼의 반정규화는 데이터 무결성에 영향을 미침
관계의 반정규화는 데이터 무결성 깨뜨릴 위험 X, 데이터처리 성능은 향상

정규화가 잘 정의된 데이터 모델에서 성능이 저하된 경우 존재

### 1-2-4. 대량 데이터에 따른 성능

*** 대량 데이터 발생에 따른 테이블 분할**

**수직분할** : **컬럼** 단위로 분할하여 I/O 경감
**수평분할** : **로우** 단위로 분할하여 I/O 경감

*** 성능 저하의 원인**

하나의 테이블에 데이터 대량집중 : 테이블 구조 너무 커져 효율성 ↓, 디스크 I/O ↑
하나의 테이블에 여러개의 컬럼 존재 : 디스크 점유량↑, 데이터 읽는 I/O량 ↑
대량의 데이터가 처리되는 테이블 : SQL문장에서 데이터 처리 위한 I/O량 ↑, 인덱스 구성
대량의 데이터가 하나의 테이블에 존재 : 인덱스의 크기 ↑ 성능저하
컬럼이 많아지는 경우 : 로우 체이닝, 로우 마이그레이션 발생

*** 해결 방안**

한 테이블 많은 컬럼 - 수직 분할
대량 데이터 저장 및 처리 - 파티셔닝, PK에 의한 테이블 분할
ㄴ Range (날짜/숫자별), List (특정 값 ex.지역), Hash

*** 테이블에 대한 수직/수평분할 절차**

데이터 모델 완성
데이터베이스 용량 산정
대량의 데이터가 처리되는 테이블에 대해 트랜잭션 처리 패턴을 분석
컬럼 단위로 집중화된 처리가 발생하는지, 로우 단위로 발생하는지 분석하여 테이블 분리

칼럼 수 많을 경우 → 1:1 형태 수직 분할
칼럼 수 적지만 데이터 용량 많을 경우 → 파티셔닝 고려

## 1-2-5. 데이터베이스 구조와 성능

*** 슈퍼/서브타입 데이터 모델**

최근 가장 많이 쓰임 : 업무를 구성하는 데이터를 공통/차이점 특징 고려 효과적 표현 가능
**슈퍼타입** : 공통의 부분
**서브타입** : 공통으로부터 상속받아 다른 엔티티와 차이가 있는 속성
논리적 데이터 모델에서 이용되는 형태. (분석단계에서 많이 쓰임)
물리적 데이터 모델로 설계시 문제점 나타남
(적당한 노하우 X → 1:1 또는 All in one 타입이 되어버려 성능 저하)

*** 슈퍼/서브타입 데이터 모델의 변환**

**# 성능저하의 원인 3가지**

트랜잭션은 일괄처리, 테이블은 개별로 유지되어 Union 연산에 의해 성능 저하
트랜잭션은 서브타입 개별로 처리, 테이블은 하나로 통합되어 불필요한 많은 데이터
트랜잭션은 슈퍼+서브타입을 공통으로 처리, 테이블은 개별로 유지됨 or 하나로 집약

**# 슈퍼/서브 타입의 변환 기준**

데이터의 양과 해당 테이블에 발생되는 트랜잭션 유형
데이터가 소량 : 데이터 처리 유연성 고려하여 가급적 1:1 관계 유지
데이터가 대량 : 3가지의 변화방법 (개별 테이블, 슈퍼+서브타입 테이블, 하나의 테이블)

*** 슈퍼/서브 타입의 데이터 모델 변환 기술**

- 개별로 발생되는 트랜잭션 → 개별 테이블로 구성 (One to One Type)
- 슈퍼+서브타입에 대해 발생되는 트랜잭션 → 슈퍼+서브타입 테이블로 구성 (Plus Type)
- 전체를 하나로 묶어 트랜잭션이 발생 → 하나의 테이블로 구성 (Single Type)

**쪼개질수록** [ **확장성 ↑, 조인성능 ↓, I/O 성능 ↑, 관리용이성 ↓** ]

*** PK/FK 컬럼 순서와 성능개요**

인덱스 중요성 : 데이터 조작시 가장 효과적으로 처리될 수 있도록 접근경로 제공 오브젝트
PK/FK 설계 중요성 : 데이터 접근할 때 접근경로 제공. 설계단계 마지막에 컬럼순서 조정
PK 순서의 중요성 : 물리적인 모델링 단계에서 스스로 생성 PK 이외에 상속 PK 순서도 주의
FK 순서의 중요성 : 조인의 경로 제공 역할 수행. 조회 조건 고려하여 반드시 인덱스 생성

*** PK 순서를 조정하지 않으면 성능 저하되는 이유**

조회 조건에 따라 인덱스를 처리하는 범위가 달라짐
PK의 순서를 인덱스 특징에 맞게 생성하지 않고 자동으로 생성하게 되면 테이블에 접근하는 트랜잭션이 비효율적인 인덱스에 의하여 인덱스를 넓은 범위로 스캔하거나 풀 스캔 유발

*** 물리적 테이블에 FK 제약이 걸려있지 않을 경우 인덱스 미생성으로 성능 저하**

물리적으로 두 테이블 사이에 FK 참조 무결성 관계를 걸어 상속받은 FK에 대해 인덱스 생성

**1-2-6. 분산 데이터베이스와 성능**

*** 분산 데이터베이스의 개요**

- 빠른 네트워크 환경을 이용하여 데이터베이스를 여러 지역에서 노드로 위치시켜 사용성과 성능을 극대화 시킨 데이터베이스
- 분산되어있는 데이터베이스를 하나의 가상 시스템으로 사용할 수 있도록 한 데이터베이스
- 논리적으로 동일한 시스템, 네트워크 통해 물리적으로 분산되어있는 데이터들의 모임
- 물리적 Site 분산, 논리적으로 사용자 통합 및 공유

*** 분산 데이터베이스의 투명성 6가지**

**분할 투명성**(단편화) : 하나의 논리적 릴레이션을 여러 단편으로 분할, 그 사본을 여러 Site에 저장
**위치 투명성** : 사용하려는 데이터의 저장 장소 알 필요 X, 위치정보 시스템 카탈로그에 유지
**지역사상 투명성** : 지역DBMS, 물리적DB 사이 Mapping 보장. 각 지역시스템 이름과 무관한 이름 사용 가능
**중복 투명성** : DB 객체가 여러 Site에 중복 되어 있는지 알 필요 X
**장애 투명성** : 구성요소(DBMS,컴퓨터)의 장애에 무관하게 트랜잭션 원자성 유지
**병행 투명성** : 다수 트랜잭션 동시 수행시 결과의 일관성 유지
(Time Stamp, 분산 2단계 Locking을 이용하여 구현)

*** 분산 데이터베이스의 적용방법**

단순히 분산 환경에서 데이터베이스를 구축하는 것이 목적이 아니라, 업무의 특징에 따라 데이터베이스 분산구조를 선택적으로 설계

*** 분산 데이터베이스 장단점**

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

*** 분산 데이터베이스의 활용 방향성**

업무적인 특징에 따라 **위치 중심**의 분산, **업무 필요**에 의한 분산을 설계

*** 데이터베이스 분산구성의 가치**

통합된 데이터베이스에서 제공할 수 없는 빠른 성능을 제공 ( 데이터 처리 성능 ↑ )
(원거리 또는 다른 서버에 접속하여 처리함으로 인해 발생되는 네트워크 부하 및 트랜잭션 집중에 따른 성능 저하의 원인을 분산된 데이터베이스 환경을 구축하여 빠른 성능 제공)

*** 테이블 위치 분산**

테이블 구조 변경 X
테이블이 다른 데이터베이스에 중복으로 생성 X
정보를 이용하는 형태가 각 위치별로 차이가 있을 경우 사용
테이블 위치를 파악할 수 있는 도식화된 위치별 데이터베이스 문서 필요

*** 테이블 분할 분산**

위치만 다른 곳에 두는 것이 아니라 각각의 테이블을 쪼개어서 분산

**# 수평분할** - 특정 칼럼의 값 기준으로 **로우 단위 분리**
칼럼은 분리 X
Primary Key에 의해 중복이 발생 X
자사별로 사용하는 Row가 다를 때 사용
**데이터 수정** : 타 지사에 있는 데이터 원칙적으로 수정 X, 자신 데이터만 수정
**각 지사의 테이블 통합처리** : 조인이 발생하여 성능저하 예상, 통합처리 프로세스가 많은지 검토한 후 많지 않은 경우 수평분할 진행
**데이터 무결성 보장** : 데이터가 지사별로 별도로 존재하므로 중복 발생 X
타 지사 데이터의 지사구분이 변경되면 단순히 수정이 발생하는 것 이외에 변경된 지사로 데이터를 이송해야함
한 시점에는 한 지사(Node)에서 하나의 데이터만 존재
**지사별로 데이터베이스를 운영하는 경우** : 데이터베이스가 속한 서버가 지사에 존재하던지 아니면 본사에 통합해서 존재하건 간에 데이터베이스 테이블들은 수평 분할하여 존재

**# 수직분할** - 칼럼을 기준으로 **칼럼 단위로 분리**
로우 단위로 분리 X
각 테이블은 동일한 기본키 구조와 값을 가지고 있어야 함
데이터를 한군데 집합시켜 놓아도 동일한 기본키는 하나로 표현하면 되므로 데이터 중복이 발생되지 않음
테이블 전체 칼럼 데이터 조회 : 가능하면 통합하여 처리하는 프로세스가 많은 경우에는 이용하지 않도록 한다
실제 프로젝트에서 수직분할 분산 환경을 구성하는 사례는 드물다

*** 테이블 복제 분산**

동일한 테이블을 다른 지역이나 서버에서 동시에 생성하여 관리
프로젝트에서 많이 사용하는 데이터베이스 분산 기법

**# 부분복제** - 마스터 데이터베이스에서 테이블의 일부 내용만 다른 지역이나 서버에 위치

통합된 테이블을 본사에 가지고 있으면서 각 지사별로는 지사에 해당하는 로우를 가지고 있는 형태 ( 본사 데이터 = 지사 데이터들의 합 )

여러 테이블에 조인이 발생하지 않는 빠른 작업 수행 가능 (각 지사에서 데이터 처리 용이할 뿐 아니라 전체 데이터 통합처리도 본사의 통합 테이블 이용)

본사 데이터는 통계, 이동 등 관리 / 지사 데이터 이용하여 지사별 빠른 업무

지사에 데이터가 선발생, 본사는 지사 데이터 통합하여 발생 (광역복제와 차이)

다른 지역간 데이터 복제는 실시간 처리보다 배치 처리를 이용 (시간, 부하 ↑)

데이터의 정합성 일치 어려움 가능하면 [ 지사 수정 발생 → 본사 복제 ] 권장

**# 광역복제** - 통합된 테이블을 본사에 가지고 있으며 각 지사에 본사와 동일한 데이터 분배

본사나 지사나 데이터처리에 특별한 제약 X

본사에서 데이터 입력, 수정, 삭제가 되어 지사에서 이용 (부분복제와 차이)

다른 지역간 데이터 복제는 실시간 처리보다 배치 처리를 이용 (시간, 부하 ↑)

*** 테이블 요약 분산**

지역/서버 간에 데이터가 비슷하지만 서로 다른 유형으로 존재하는 경우

**# 분석요약** - 각 지사별로 존재하는 요약 정보를 본사에 통합하여 다시 전체에 대해 요약

동일한 테이블 구조를 가지고 있으면서 분산되어 있는 동일한 내용의
데이터를 이용하여 통합된 데이터를 산출하는 방식

테이블에 있는 모든 칼럼과 로우가 지사에도 동일하게 존재하지만,
각 지사에는 동일한 내용에 대해 지사별로 요약된 정보를 가지고 있고
본사에는 각 지사의 요약 정보를 통합하여 재산출한 전체 요약정보를 가짐

각종 통계 데이터 산정 : 모든 지사의 데이터를 이용하여 처리하면 성능이
지연되고 각 지사 서버에 부하, 업무장애 발생 가능

지사에 있는 데이터를 이용하여 본사에서 통합하여 요약 데이터 산정
통합 통계 데이터에 대한 정보제공에 용이

본사에 분석 요약된 테이블을 생성하고 데이터는 역시 일반 업무가 종료되는
야간에 수행하여 생성

**# 통합요약** - 각 지사별로 존재하는 다른 내용 정보를 본사에 통합, 다시 전체의 요약 산출

테이블에 있는 모든 칼럼과 로우가 지사에도 동일하게 존재하지만,
각 지사에는 타 지사와 다른 요약정보를 가지고 있고 본사에는 각 지사의
요약 정보를 데이터를 같은 위치에 두는 것으로 통합하여 전체 요약정보 가짐

본사에 통계 데이터 산정 : 분석요약과 비슷하나 단지 지사에서 산출한 요약
정보를 한군데 취합하여 보여주는 형태.
모든 지사의 데이터를 이용하여 처리하면 성능이
지연되고 각 지사 서버에 부하를 주기 때문에
업무장애 발생 가능

지사에서 요약한 정보를 본사에서 취합, 각 지사별 데이터 비교하기 용이
통계 데이터에 대한 정보 제공에 용이

본사에 통합 요약된 테이블을 생성하고 데이터는 역시 일반 업무가 종료되는
야간에 수행하여 생성

*** 분산 데이터베이스를 적용하여 성능이 향상된 사례**

분산 환경의 원리를 이해하지 않고 데이터베이스를 설계하여 성능이 저하되는 경우 빈번
복제분산의 원리를 간단히 응용하면 많은 업무적 특성이 있는 곳에서 성능 향상해 설계가능

**데이터베이스 분산 설계는 다음과 같은 경우에 적용하면 효과적이다.**

- 성능이 중요한 사이트
- 공통코드, 기준정보, 마스터 데이터 등에 대해 분산 환경을 구성하면 성능이 좋아짐
- 실시간 동기화가 요구되지 않을 때 좋음.
  ( 준 실시간의 업무적 특징을 가져도 분산 환경 구성 가능 )
- 특정 서버에 부하가 집중될 때 부하를 분산시키는 용도로 좋다
- 백업 사이트(Disaster Recovery Site) 구성 시, 간단하게 분산 기능 적용하여 구성 가능

# 과목 2 SQL 기본 및 활용

## 2-1 SQL 기본

### 2-1-1. 관계형 데이터베이스 개요

*** 데이터베이스 정의**

특정 기업이나 조직 또는 개인이 필요에 의해 데이터를 일정한 형태로 저장해 놓은 것
데이터베이스 관리 소프트웨어 → DBMS

*** 데이터베이스 발전**

플로우차트 (1960) → 계층형, 망형 (1970) → 관계형 (1980) → 객체관계형 (1990)

*** 관계형 데이터베이스**

- 1970 영국의 수학자 E.F.Codd 박사의 논문에서 처음 소개

- 파일 시스템 단점 : 동시에 입력/수정/삭제할 수 없기 때문에 정보의 관리가 어려움
  복사본 파일을 만들어 사용할 경우 데이터의 불일치성이 발생

- 관계형 DB의 장점 : 정규화를 통해 이상 현상을 제거하고 데이터 중복을 피함
  동시성 관리, 병행 제어를 통해 데이터를 공유
  데이터의 표현 방법 등 체계화 할 수 있고, 데이터 표준화, 품질 확보
  보안기능, 데이터 무결성 보장, 데이터 회복/복구 기능

*** SQL (Structured Query Language)**

최초 이름은 SEQUEL, 1986부터 ANSI/ISO를 통해 표준화되고 정의됨
각 벤더의 RDBMS는 표준화된 SQL 이외에 벤더 차별화 및 이용 편리성 위해 추가기능 구현
SQL 명령어는 3가지 SAVEPOINT 그룹인 DDL, DML, DCL로 구분
TCL의 경우 굳이 나눈다면 일부에서 DCL로 분류, but 성격이 다르므로 별도의 4번째로 분리

DML - Select, Insert, Update, Delete
DDL - Create, Alter, Drop, Rename
DCL - Grant, Revoke
TCL - Commit, Rollback

*** 테이블**

데이터를 저장하는 객체로서 관계형 데이터베이스의 기본 단위

*** 테이블의 분할**

데이터의 불필요한 중복을 줄이는 것이 정규화(Normalization). 이상현상(Anomaly) 방지
기본키 ( PK / Primary Key ), 외래키 ( FK / Foreign key )

*** ERD (Entitiy Relationship Diagram)**

구성요소 : 엔티티(Entity), 관계(Relationship), 속성(Attribute) 3가지
표기법 : IE(Information Engineering) 표기법, Barker(Case Method) 표기법

**2-1-2. DDL**

*** 데이터 유형**

**# 숫자 타입**

ANSI/ISO 기준 : Numeric, Decimal, Dec, Small Int, Integer, Int, Big int, Float, Real, Double Precision

SQL Server / Sybase : 작은 정수, 정수, 큰 정수, 실수 등 + Money, Small Money
Oracle : 숫자형 타입에 대해서 Number 한 가지 타입만 지원

벤더에서 ANSI/ISO 표준을 사용할 땐 기능을 중심으로 구현, 표준과 다른 용어 사용 허용

**# 테이블의 칼럼이 가지고 있는 대표적인 4가지 유형**

Character(s) : 고정길이 문자열, CHAR로 표현, s는 기본길이 1바이트, 최대 Oracle 2,000 SQL Server 8,000바이트, 최대 s만큼의 길이를 가짐. 빈 공간은 채워짐

Varchar(s) : Character Varying의 약자. 가변길이 문자열, Oracle은 VARCHAR2, SQL Server는 VARCHAR로 표현, s는 최소 1바이트, 최대 Oracle 4,000
SQL Server 8,000 바이트, s만큼의 최대 길이 but 가변적으로 줄어들어 조정

Numeric : 정수, 실수 등 숫자 정보. Oracle은 Number, SQL Server는 10가지 이상 타입
Oracle은 처음 전체 자리 수, 그 뒤에 그 중 소수 부분 자리 수 지정

Datetime : 날짜와 시각 정보, Oracle은 DATE, SQL Server는 DATETIME으로 표현
Oracle은 1초 단위, SQL Server는 3.33ms(밀리세컨) 단위 관리

**# 문자열 유형**

CHAR(고정길이) 과 VARCHAR(가변길이) 의 차이
CHAR에서 문자열 비교 = 공백을 채워서 비교
VARCHAR에서 비교 = 맨 처음부터 한 문자씩 비교 ( 공백도 하나의 문자로 취급 )

*** CREATE TABLE**

테이블과 칼럼 정의 : 후보키중 하나를 선정하여 기본키 지정

**CREATE TABLE 테이블명 (**
**칼럼명1 DataType [Default 형식]**
**칼럼명2 DateType [Default 형식] );**

테이블 생성시 대/소문자 구분은 하지 않는다. 기본적으로 테이블이나 칼럼명 = 대문자로
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 X
칼럼에 대한 제약조건이 있으면 CONSTRAINT를 이용하여 추가할 수 있다.

*** 제약조건**

데이터의 무결성을 유지하기 위한 데이터베이스의 보편적인 방법

**PRIMARY KEY** : 기본키. 하나의 테이블에 한 개만 지정 가능. 자동으로 UNIQUE 인덱스 생성
NULL값 입력 불가 ( 기본키 제약 = 고유키 & NOT NULL 제약 )
**UNIQUE KEY** : NULL 가능, 행을 고유하게 식별하기 위한 고유키
**NOT NULL** : NULL 값 입력 금지.
**CHECK** : 입력할 수 있는 값의 범위 등을 제한. TRUE or FALSE 논리식을 지정
**FOREIGN KEY** : 외래키. 참조 무결성 옵션 선택 가능

NULL : 공집합, 0, 공백 등과 다른 정의되지 않은 미지의 값 or 데이터 입력을 못하는 경우
DEFAULT : 데이터 입력 시 칼럼의 값이 지정되지 않을 때, 기본값.

*** 생성된 테이블 구조 확인**

[Oracle] **DESCRIBE 테이블명;**
[SQL Server] **exec sp_help 'dbo.테이블명' go**

*** SELECT 문장을 통한 테이블 생성 사례**

SELECT 문장을 활용, 테이블 생성할 수 있는 방법 (**CTAS** : Create Table ~ As Select ~)
CTAS 사용시 주의할 점 : 기존 테이블 제약조건중 **NOT NULL만 새로운 테이블에 복제**
기본키, 고유키, 외래키, CHECK 등 **다른 제약조건 사라짐**
제약 조건 추가 위해선 ALTER TABLE 기능 사용해야 함
SQL Server에서는 Select ~ Into ~를 활용하여 같은 결과
단, 칼럼 속성에 identity 사용 = identity 속성 같이 적용

[Oracle] **CREATE TABLE 테이블명_TEMP AS SELECT * FROM 테이블명;**
[SQL] **SELECT * INTO 테이블명_TEMP FROM 테이블명;**

*** ALTER TABLE**

**ADD COLUMN** : **ALTER TABLE 테이블명 ADD 추가칼럼명 데이터유형;**
새롭게 추가된 칼럼은 테이블의 마지막 칼럼. 위치는 지정 불가능.

**DROP COLUMN** : **ALTER TABLE 테이블명 DROP COLUMN 삭제할 칼럼명;**

**MODIFY COLUMN** : ALTER TABLE을 이용해 칼럼의 데이터 유형, 디폴트 값, NOT NULL
제약조건에 대한 변경을 포함

[Oracle] **ALTER TABLE 테이블명 MODIFY (칼럼명 데이터유형 DEFAULT NOT NULL);**
[SQL Server] **ALTER TABLE 테이블명 ALTER COLUMN 칼럼명 데이터유형 DEFAULT NOT NULL;**

칼럼을 변경할 때는 몇 가지 사항을 고려해서 변경해야 한다
- 해당 칼럼의 크기를 늘릴 수는 있지만 줄이지는 못한다
- 해당 칼럼이 NULL 값만 가지고 있거나 테이블에 아무 행도 없으면 칼럼 폭 줄이기 가능
- 해당 칼럼이 NULL 값만을 가지고 있으면 데이터 유형 변경 가능
- 해당 칼럼의 DEFAULT 값을 바꾸면 변경 작업 이후 발생하는 행 삽입에만 영향을 미침
- 해당 칼럼에 NULL 값이 없을 경우에만 NOT NULL 제약조건 추가 가능

**RENAME COLUMN**
[Oracle] **ALTER TABLE 테이블명 RENAME COLUMN (구)칼럼명 TO (신)칼럼명;**
[SQL Server] **sp_rename 'dbo.테이블명.칼럼명(구)', '테이블명.칼럼명(신)', 'COLUMN';**

**DROP CONSTRAINT** : **ALTER TABLE 테이블명 DROP CONSTRAINT 제약조건명;**

**ADD CONSTRAINT** :
**ALTER TABLE 테이블명 ADD CONSTRAINT 제약조건명 제약조건 (칼럼명);**

참조 무결성 제약조건을 추가하여 실수에 의한 테이블 삭제나 필요한 데이터의 의도하지 않은 삭제와 같은 불상사를 방지하는 효과를 볼 수 있다.

*** RENAME TABLE**

[Oracle] **RENAME (구)테이블명 TO (신)테이블명;**
[SQL Server] **sp_rename 'dbo.(구)테이블명', '(신)테이블명';**

*** DROP TABLE**

**DROP TABLE 테이블명 CASCADE CONSTRAINT;**
CASCADE CONSTRAINT 옵션은 해당 테이블과 관계가 있었던 참조되는 제약조건에 대해서도 함께 삭제한다는 것을 의미한다. SQL Server에서는 CASCADE 옵션 존재 X
(테이블을 삭제하기 전에 참조하는 FOREIGN KEY 제약조건 또는 참조하는 테이블 삭제)

*** TRUNCATE TABLE**

**TRUNCATE TABLE 테이블명;**
테이블 자체가 삭제되는 것이 아니라 해당 테이블의 모든 행들이 제거되고 저장 공간을 재사용 가능하도록 해제한다. 구조 자체 삭제는 DROP TABLE을 이용
DML로 분류할 수도 있지만 내부 처리방식이나 Auto Commit 특성으로 DDL로 분류
DELETE 명령어와 처리 방식 자체가 다름. 전체 데이터 삭제시 부하가 더 적어서 권장
But 오토커밋이기 때문에 정상적인 복구가 불가능하므로 사용시 주의 요망

**2-1-3. DML**

*** INSERT**

**INSERT INTO 테이블명 ( 칼럼리스트 / 생략 = 전체칼럼 )**
**VALUES ( 리스트 순서에 맞춰 입력할 값 1:1 매핑하여 작성 )**
칼럼의 데이터가 문자형일 경우 ' (Single quotation)으로 묶어서 입력. 숫자는 X

*** UPDATE**

**UPDATE 테이블명**
**SET 칼럼명 = 값;**

*** DELETE**

**DELETE FROM 테이블명;** (FROM 생략 가능)

*** SELECT**

**SELECT [ALL | DISTINCT] 칼럼1, 칼럼2, ...**
**FROM 테이블명;**
ALL : DEFAULT 옵션 ( 중복 데이터 모두 출력 )
DISTINCT : 중복 제거하여 1건으로 출력

**ALIAS 부여** : 조회된 결과에 별명(ALIAS, ALIASES)을 부여하여 칼럼 레이블은 변경 가능
칼럼명 바로 뒤에 온다 칼럼명과 ALIAS 사이에 AS 키워드 사용 가능 (선택)
ALIAS가 공백,특수문자를 포함할 경우나 대소문자 구분이 필요할 경우,
" ( Double quotation )으로 묶어 사용한다.

*** 산술 연산자**

NUMBER와 DATE 자료형에 적용 ( 수학에서의 사칙연산과 동일 )
우선순위 = [ ( ) ] → [ * ] → [ / ] → [ + ] → [ - ]

*** 합성(Concatenation) 연산자**

문자와 문자를 연결하는 경우
[Oracle] → **||**
[SQL Server] → **+**

두 벤더 모두 공통적으로 CONCAT (string1, string2, ...) 함수를 이용해 동일하게 표현 가능

**2-1-4. TCL**

*** 트랜잭션**

데이터베이스의 **논리적 연산단위**
밀접히 관련되어 분리될 수 없는 한 개 이상의 데이터베이스 조작
하나의 트랜잭션에는 하나 이상의 SQL 문장이 포함
트랜잭션은 분할할 수 없는 최소의 단위
따라서, 전부 적용하거나 전부 취소. 즉, 트랜잭션은 ALL OR NOTHING

*** 트랜잭션을 컨트롤하는 TCL**

**COMMIT** : 문제없이 처리된 트랜잭션을 데이터베이스에 반영시키는 것
**ROLLBACK** : 트랜잭션 수행 이전의 상태로 되돌리는 것

트랜잭션 대상이 되는 SQL : UPDATE, INSERT, DELETE 등 데이터를 수정하는 DML문
SELECT FOR UPDATE 등 배타적 LOCK을 요구하는 SELECT 문

*** 트랜잭션의 특성**

**원자성(Atomicity)** : 트랜잭선의 연산은 모두 적용되던지, 아니면 모두 취소되어야 한다.
**일관성(Consistency)** : 트랜잭션의 실행 전 DB에 이상이 없다면 실행 후에도 같아야 한다.
**고립성(Isolation)** : 트랜잭션 실행 중, 다른 트랜잭션의 영향을 받아서는 안된다.
**지속성(Durability)** : 트랜잭션이 성공적으로 수행되면 영구적으로 반영되어 저장된다.

*** COMMIT**

입력, 수정, 삭제한 자료에 대하여 문제가 없을 경우 COMMIT 명령어로 **변경 사항을 적용**

COMMIT 이전의 상태 : 단지 Memory Buffer에만 영향을 주고 이전 상태로 복구가 가능
**현재 사용자**는 SELECT 문으로 **변경 결과 확인 가능**한 상태
**다른 사용자**는 현재 사용자가 수행한 결과 **확인 불가능**
변경된 행은 잠금(Locking)이 설정되어 다른 사용자가 변경 불가능

COMMIT 이후의 상태 : 데이터에 대한 변경 사항이 데이터베이스에 영구적 반영.
이전 데이터는 영원히 잃어버린다.
모든 사용자가 결과를 조회할 수 있다.
관련 행에 잠금이 해제되고, 다른 사용자가 행을 조작할 수 있다.

**SQL Server는 기본적으로 DML 구문도 Auto Commit** 모드이다. (Auto / 암시적 / 명시적)

*** ROLLBACK**

테이블 내 입력, 수정, 삭제한 데이터에 대해서 COMMIT 이전에 **변경 사항을 취소**하는 기능
변경 사항이 취소되고, 관련 행 잠금이 풀리며 다른 사용자들이 데이터 변경 가능

롤백 후 데이터 상태 : 데이터에 대한 변경사항은 취소됨
이전 데이터가 다시 재저장됨.
관련 행에 대한 잠금이 풀리고 다른 사용자들이 행 조작 가능

**COMMIT과 ROLLBACK을 사용함으로써 얻을 수 있는 효과**

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

*** SAVEPOINT (저장점)**

저장점을 정의하면 롤백을 할 경우 전체 롤백이 아닌 저장점까지의 일부만 롤백할 수 있다.
- 복수의 저장점을 정의할 수 있다.
- 동일 이름으로 저장점을 저장시 나중에 정의한 저장점이 유효.
- 저장점을 정의하고 저장점으로 롤백한다.

[Oracle] **SAVEPOINT 포인트이름;**
→ **ROLLBACK TO 포인트이름;**

[SQL Server] **SAVE TRANSACTION 포인트이름;**
→ **ROLLBACK TRANSACTION 포인트이름;**

**한 저장점으로 되돌리고 나서 그보다 더 미래로 다시 되돌릴 수 없다.**
특정한 저장점까지 롤백하면 그 저장점 이후에 설정한 저장점은 모두 무효가 된다.
저장점 없이 롤백하면 모든 변경사항을 취소한다.

## 2-1-5. WHERE 절

### * WHERE 절

원하는 자료만을 검색하기 위해서 WHERE 절 이용하여 자료를 제한
WHERE 절에 조건이 없는 FTS(Full Table Scan) 문장은 SQL 튜닝 1차 검토 대상
(FTS가 무조건 나쁜 것은 아님. 병렬 처리 등을 이용해 유용하게 사용하는 경우도 많음)

### * 연산자의 종류

비교 연산자 : =, >, >=, <, <=
SQL 연산자 : BETWEEN a AND b, IN (list), LIKE '비교문자열', IS NULL
논리 연산자 : AND, OR, NOT
부정 비교 연산자 : !=, ^=, <>, NOT 칼럼명 =, NOT 칼럼명 >
부정 SQL 연산자 : NOT BETWEEN a AND b, NOT IN (list), IS NOT NULL

### * 문자 유형간의 비교 방법

#### # 비교 연산자의 양쪽이 모두 CHAR 타입인 경우

길이가 서로 다른 CHAR 이면 작은쪽에 Space 추가하여 길이 같게 한 후에 비교
서로 다른 문자가 나올 때까지 비교
달라진 첫 번째 문자의 값에 따라 크기를 결정
Blank의 수만 다르다면 서로 같은 값으로 결정

#### # 비교 연산자의 어느 한 쪽이 VARCHAR 타입인 경우

서로 다른 문자가 나올 때까지 비교
길이가 다르다면 짧은 쪽이 끝날 때까지만 비교한 후 길이가 긴 것이 크다고 판단
길이가 같고 다른 것이 없다면 같다고 판단
VARCHAR는 NOT NULL까지 길이를 말함

#### # 상수값과 비교할 경우

상수 쪽을 변수 타입과 동일하게 바꾸고 비교
변수 쪽이 CHAR 타입이면 위의 CHAR 타입 경우를 적용
변수 쪽이 VARCHAR 타입이면 위의 VARCHAR 타입 경우를 적용

*** 연산자의 우선순위**

[ **( )** ] → [ **NOT** ] → [ **비교연산자, SQL 비교연산자** ] → [ **AND** ] → [ **OR** ]

*** SQL 연산자**

BETWEEN a AND b : a와 b 값 사이에 있는 값들. (a, b 포함)
IN (list) : 리스트에 있는 값 중에서 하나라도 일치 여부
LIKE '비교문자열' : 형태 일치 여부 ( 와일드카드 : **%** (0개 이상 문자열), **_** (1개 단일 문자) )
IS NULL : NULL 값인지 여부

*** 논리 연산자**

AND : 앞의 조건과 뒤의 조건이 모두 TRUE 일 경우 TRUE 반환. (동시만족)
OR : 앞이나 뒤의 조건 중 한 개라도 TRUE 이면 TRUE 반환.
NOT : 뒤에 오는 조건절에 반대되는 결과 반환.

*** 부정 연산자**

**# 부정 논리 연산자**

!=, ^=, <> → 같지 않다 (ANSI/ISO 표준은 <>. 모든 운영체제 사용 가능)
NOT 칼럼명 = → ~와 같지 않다
NOT 칼럼명 > → ~보다 크지 않다

**# 부정 SQL 연산자**

NOT BETWEEN a AND b : a부터 b까지의 값 사이에 포함되지 않는다 (a, b도 값에 포함)
NOT IN (list) : list에 일치하는 값이 없다
IS NOT NULL : NULL 값을 갖지 않는다

*** ROWNUM, TOP**

WHERE 절에서 행의 개수를 제한하는 목적으로 사용
1건의 행은 [ = ] 연산자 사용 가능, 2건 이상부터는 [ = ] 사용 불가
[Oracle] **SELECT 칼럼명 FROM 테이블명 ROWNUM <= N or ROWNUM < N;**
고유키나 인덱스 생성 가능 → UPDATE 테이블명 SET 칼럼명 = ROWNUM;
[SQL Server] **SELECT TOP(N) 칼럼명 FROM 테이블명;**
→ **TOP(Expression) / PERCENT / WITH TIES**
Expression : 행 수 지정, PERCENT : 결과 집합의 처음 몇 % 행만 반환,

WITH TIES : ORDER BY 절이 지정된 경우만 사용가능, 마지막 행 같은 값 추가 출력

**2-1-6. 함수**

*** 내장함수 (Built-In Function)**

함수는 다양한 기준으로 분류 가능
- 벤더에서 제공하는 함수인 내장 함수 **(Built-In Function)**
- 사용자가 정의할 수 있는 함수 **(User Defined Function)**

SQL을 더욱 강력하게 해주고 데이터 값을 간편하게 조작하는데 사용.
벤더별로 가장 큰 차이를 보이는 부분이지만 핵심적인 기능들은 이름/표기법이 달라도 비슷
내장함수는 다시 함수의 입력 값에 따라 **단일행 함수**와 **다중행 함수**로 나뉜다
함수는 입력값이 아무리 많아도 출력값은 하나라는 M:1 관계라는 중요한 특징을 가짐

**단일행 함수** : 단일행 내에 있는 하나의 값 또는 여러 값이 입력 인수로 사용
**다중행 함수** : 여러 레코드의 값들을 입력 인수로 사용

*** 단일행 함수의 종류**

**문자형, 숫자형, 날짜형, 변환형, NULL관련 함수**

**# 문자형 함수** : **LOWER**(문자열) - 문자열의 알파벳 문자를 소문자로 변경
**UPPER**(문자열) - 문자열의 알파벳 문자를 대문자로 변경
**ASCII**(문자) - 문자나 숫자를 ASCII 코드 번호로 출력
**CHR/CHAR**(ASCII번호) - ASCII 코드 번호를 문자나 숫자로 출력
**CONCAT**(문자열1, 문자열2) - 문자열은 연결 ( || or + 와 동일 기능 )
**SUBSTR/SUBSTRING**(문자열, m[, n]) - 문자열 중 m 위치부터 n개의 문자 반환 ( n 생략 시 마지막 문자까지 )
**LENGTH/LEN**(문자열) - 문자열의 개수(길이)를 숫자값으로 반환
**LTRIM**(문자열[, 지정문자]) - 문자열 앞쪽부터 확인해 지정 문자가 처음 나타나는 동안 해당 문자를 제거(기본값 공백)
**RTRIM**(문자열[, 지정문자]) - 문자열 뒤부터 확인해 지정 문자가 처음 나타나는 동안 해당 문자를 제거(기본값 공백)
**TRIM**([ leading | trailing | both ] 지정문자 FROM 문자열) - 머리말, 꼬리말, 양쪽 지정문자 제거 기본값 both
SQL Server에서는 TRIM 함수에 지정문자 사용 불가 (공백만 제거 가능)
**Oracle 함수 / SQL Server 함수로 표시. 공통함수는 표시 X**

**@ DUAL 테이블은 데이터 사전과 함께 Oracle에 의해 자동으로 생성되는 기본 테이블**
**SYS의 스키마에 있으며 모든 사용자가 액세스 가능. VARCHAR2(1) 정의의 DUMMY 열 1개**

**값을 가지는 하나의 행도 포함. 사용자가 계산이나 사용자 함수 등을 실행 할 경우 사용**

**# 숫자형 함수** : **ABS**(숫자) - 절대값 반환
**SIGN**(숫자) - 양수, 0, 음수 판별 (부호)
**MOD**(숫자1, 숫자2) - 숫자1을 숫자2로 나누어 나머지 값 반환 ( % )
**CEIL/CEILING**(숫자) - 크거나 같은 최소 정수 반환 (정수 값으로 올림)
**FLOOR**(숫자) - 작거나 같은 최대 정수 반환 (정수 값으로 버림)
**ROUND**(숫자[, m]) - 소수점 m자리에서 반올림 (생략시 0)
**TRUNC**(숫자[, m]) - 소수 m자리 뒤로 잘라서 버림 (생략시 0) **[Oracle Only]**
**SIN, COS, TAN,** ... - 삼각함수 값 반환
**EXP(), POWER(), SQRT(), LOG(), LN()** - 지수, 거듭제곱, 제곱근, 자연로그

**# 날짜형 함수** : **SYSDATE/GETDATE()** - 현재 날짜와 시각 반환
**EXTRACT('YEAR'|'MONTH'|'DAY' from d)** - 년/월/일 데이터를 추출
**/ DATEPART('YEAR'|'MONTH'|'DAY', d)** (시간/분/초도 가능함)
**TO_NUMBER(TO_CHAR(d,'YYYY')) / YEAR(d)** - 위와 같은 기능
(TO_NUMBER 제외시 문자형 출력)

**날짜+숫자 = 날짜 / 날짜-숫자 = 날짜 / 날짜-날짜 = 날짜 수 / 날짜+숫자/24 = 날짜+시간**

**# 변환형 함수**

**명시적(Explicit) 변환** : 데이터 변환형 함수로 변환하도록 명시
**암시적(Implicit) 변환** : 데이터베이스가 자동으로 변환하여 계산

[ Oracle ]
**TO_NUMBER(문자열)** - alphanumeric 문자열을 숫자로 변환
**TO_CHAR(숫자|날짜[, FORMAT])** - 숫자/날짜를 주어진 FORMAT으로 문자열 타입 변환
**TO_DATE(문자열[,FORMAT])** - 문자열을 주어진 FORMAT으로 날짜 타입 변환

[ SQL Server ]
**CAST (expression AS data_type [(length)])** - expression을 목표 타입으로 변환
**CONVERT (data_type [(length)] expression[, style])** - expression 목표 타입 변환

*** CASE 표현**

CASE 표현은 IF-THEN-ELSE 논리와 유사한 방식으로 표현식을 작성하여 SQL의 비교 연산 기능을 보완하는 역할을 함. ANSI/ISO 표준에는 CASE Expression이라고 표시. 함수와 같은 성격을 가지고 있으며 Oracle의 DECODE 함수와 같은 기능을 하므로 단일행 내장함수에서 같이 설명함.

**SELECT 칼럼명,**
**CASE**
**WHEN 조건**
**THEN 조건이 TRUE일 때 반환**
**ELSE 조건이 FALSE일 때 반환**
**END AS 칼럼명**
**FROM 테이블명;**

CASE 표현에는 조건절 표현 방법이 두 가지. Oracle의 경우 DECODE 함수 사용도 가능

1. **SIMPLE_CASE_EXPRESSION** 조건 TRUE이면 해당 THEN 절 수행, FALSE이면 ELSE 절 수행
( **EQUI(=) 조건만 사용**할 경우 간단하게 사용 **DECODE와 기능 동일** )

**CASE**
**SIMPLE_CASE_EXPRESSION 조건** ( 인수 WHEN 조건 THEN 반환값 )
**ELSE 표현절** ( ELSE 반환값 )
**END**

2. **SEARCHED_CASE_EXPRESSION** 조건 TRUE 이면 해당 THEN 절 수행, FALSE이면 ELSE 절
( = 뿐만 아니라 **부등호 등을 이용한 여러 조건절 사용** 가능. **다양한 조건** 적용 가능 )

**CASE**
**SEARCHED_CASE_EXPRESSION 조건** ( WHEN 조건 THEN 반환값 * N )
**ELSE 표현절** ( ELSE 반환값 )
**END**

3. Oracle 함수로 표현식의 값이 기준값1 이면 값1 출력, 기준값2 이면 값2 출력.
기준값이 없으면 디폴트 값 출력. CASE의 SIMPLE_CASE_EXPRESSION 조건과 동일

**DECODE(표현식, 기준값1, 값1, 기준값2, 값2, … , DEFAULT 값) DEFAULT 생략시 NULL**

*** NVL / ISNULL 함수**

테이블 생성시 NOT NULL, PK로 지정하지 않은 모든 데이터 유형은 NULL값을 가질 수 있음
NULL 값을 포함하는 연산의 경우 결과 값도 NULL이다.

**NVL/ISNULL**(판단대상, 대체값) : 판단대상의 값이 NULL이면 대체값 출력
( 판단대상과 대체값 데이터 타입 같아야 함)

**NULLIF**(판단대상, 비교대상) : 판단대상 = 비교대상 이면 NULL, 아닐 경우 판단대상 리턴

**COALESCE**(표현식1, 표현식2, ...) NULL이 아닌 최초의 표현식 리턴. 없으면 NULL 리턴

*** 공집합**

**SELECT 1** 와 같은 조건이 대표적인 공집합을 발생시키는 쿼리
**FROM DUAL** 조건에 맞는 데이터가 한 건도 없는 경우를 공집합이라 함
**WHERE 1 = 2;** NULL과는 또 다르게 이해해야 함

**인수 값이 공집합인 경우 NVL/ISNULL 사용해도 공집합**이 출력.

NVL/ISNULL → 널 값을 대체값으로 표현하고 싶을 때 이용
NULLIF → 특정 값을 NULL로 대체하는 경우에 유용하게 이용
COALESCE 함수는 두 개의 중첩된 CASE 문장으로 표현이 가능.

### 2-1-7. GROUP BY HAVING 절

*** 집계함수 (Aggregate Function)**

여러 행들의 그룹이 모여서 그룹당 단 하나의 결과를 돌려주는 함수
GROUP BY 절은 행들을 소그룹화
SELECT 절, HAVING 절, ORDER BY 절에 사용

**집계함수명( ALL | DISTINCT 칼럼/표현식 )** - Default = ALL

주로 숫자 유형에 사용하며 **MAX, MIN, COUNT 함수는 문자, 날짜에도 적용** 가능

**COUNT**(*) : NULL 값을 포함한 모든 행의 수를 출력
**COUNT**(표현식) : NULL 값인 것을 제외한 행의 수를 출력
**SUM**([DISTINCT|ALL] 표현식) : NULL을 제외한 합계 출력
**AVG**([DISTINCT|ALL] 표현식) : NULL을 제외한 평균 출력
**MAX**([DISTINCT|ALL] 표현식) : 최대값 출력 (문자, 날짜 데이터 사용 가능)
**MIN**([DISTINCT|ALL] 표현식) : 최소값 출력 (문자, 날짜 데이터 사용 가능)
**STDDEV**([DISTINCT|ALL] 표현식) : 표준 편차를 출력
**VARIAN**([DISTINCT|ALL] 표현식) : 분산을 출력
+ 벤더별로 기타 다양한 통계 함수를 제공

*** GROUP BY 절**

FROM 절과 WHERE 절 뒤에 오며, 데이터들을 작은 그룹으로 분류하여 소그룹에 대한 항목별 통계 정보를 얻을 때 추가로 사용한다

**SELECT DISTINCT 칼럼명 ALIAS명**
**FROM 테이블명**
**WHERE 조건식**
**GROUP BY 칼럼/표현식**
**HAVING 그룹의 조건식;**

**GROUP BY 절과 HAVING 절의 특성**

- GROUP BY 절을 통해 소그룹별 기준을 정한 뒤, SELECT 절에 집계 함수 사용
- 집계 함수의 통계 정보는 NULL 값을 가진 행을 제외하고 수행
- 집계 함수는 WHERE 절에는 올 수 없음 (GROUP BY 절보다 WHERE 절이 먼저 수행)
- WHERE 절은 전체 데이터를 GROUP으로 나누기 전에 행들을 미리 제거하는 역할
- HAVING 절은 GROUP BY 절의 기준 항목이나 소그룹의 집계 함수를 이용한 조건을 표시할 수 있음
- GROUP BY 절에 의한 소그룹별로 만들어진 집계 데이터 중, HAVING 절에서 제한 조건을 두어 조건을 만족하는 내용만을 출력
- HAVING 절은 일반적으로 GROUP BY 절 뒤에 위치
- GROUP BY 절에서는 ALIAS명을 사용할 수 없다
- 원칙적으로 ORDER BY 절 명시해야 정렬 (일부 DB 과거 버전에서 자동 오름차순 정렬)

*** HAVING 절**

- WHERE 절에는 집계 함수를 사용할 수 없다
- GROUP BY 절보다 HAVING 절을 앞에 사용해도 같은 결과가 나오지만, 논리적으로 순서를 지키는 것을 권고한다.
- 가능하면 WHERE 절에서 조건절을 적용하여 GROUP BY 계산 대상을 줄이는 것이 효과적
- HAVING 절은 SELECT 절에 사용되지 않은 칼럼이나 집계 함수가 아니더라도, GROUP BY 절의 기준 항목이나 소그룹의 집계 함수를 이용한 조건을 표시 가능
- WHERE 절 조건 변경은 대상 데이터 개수가 변경되므로 결과 데이터 값이 변경 가능성O, HAVING 절 조건 변경은 결과 데이터 변경은 없고 출력되는 레코드 개수만 변경 가능성O

*** CASE 표현을 활용한 월별 데이터 집계**

제 1정규화로 인해 반복되는 칼럼의 경우 구분 칼럼을 두고 여러 개의 레코드로 만들어진 집합을 정해진 칼럼 수만큼 확장해서 집계 보고서를 만드는 기법 (DECODE도 동일기능)

하나의 데이터에 여러 번 CASE 표현, 집계함수 사용 후 그룹으로 묶음

*** 집계 함수와 NULL 처리**

다중행 함수에 NVL함수를 사용하면 부하 발생 → 굳이 사용할 필요 없음.

다중행 함수는 입력 값으로 전체 건수가 NULL 값인 경우만 함수의 결과가 NULL
전체 건수 중에서 일부만 NULL인 경우는 NIULL인 행을 연산에서 제외

CASE 표현 사용시는 ELSE 절을 생략하면 Default 값이 NULL
→ 같은 결과를 얻을 수 있다면 가능한 ELSE 절의 상수값을 지정하지 않거나 ELSE절 생략

Oracle의 DECODE 함수는 4번째 인자 지정하지 않으면 Default 값이 NULL

**2-1-8. ORDER BY 절**

*** ORDER BY 정렬**

ORDER BY 절에 칼럼명 대신 SELECT 절에서 사용한 ALIAS 명이나 칼럼 순서를 나타내는 정수도 사용이 가능하다

기본적인 정렬 순서는 오름차순(ASC)이며 SQL 문장의 제일 마지막에 위치한다

**SELECT 칼럼명 ALIAS명**
**FROM 테이블명**
**WHERE 조건식**
**GROUP BY 칼럼/표현식**
**HAVING 그룹조건식**
**ORDER BY 칼럼/표현식 [ASC|DESC];**

숫자형 타입은 오름차순시 작은 값부터 출력
날짜형 타입은 오름차순시 빠른 날부터 출력
Oracle은 NULL 값을 가장 큰 값, SQL Server는 NULL 값을 가장 작은 값으로 간주

칼럼의 순서는 향후 유지보수성이나 가독성이 떨어지므로 가능한 칼럼명이나 ALIAS명 권고

*** SELECT 문장의 실행 순서**

**5. SELECT 칼럼명 ALIAS명**
**1. FROM 테이블명**
**2. WHERE 조건식**
**3. GROUP BY 칼럼/표현식**
**4. HAVING 그룹조건식**
**6. ORDER BY 칼럼/표현식;**

**1. 발췌 대상 테이블 참조 (FROM)**
**2. 발췌 대상 데이터가 아닌 것은 제거 (WHERE)**
**3. 행들을 소그룹화 (GROUP BY)**
**4. 그룹핑된 값의 조건에 맞는 것만을 출력 (HAVING)**
**5. 데이터 값을 출력/계산 (SELECT)**
**6. 데이터를 정렬 (ORDER BY)**

옵티마이저가 SQL 문장의 SYNTAX, SEMANTIC 에러를 점검하는 순서이기도 하다

**FROM 절에 정의되지 않은 칼럼을 WHERE 절, GROUP BY 절, HAVING 절, SELECT 절, ORDER BY 절에 사용하면 에러 발생.**

(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 DB가 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

서브쿼리의 SELECT 절에서 선택되지 않은 칼럼들은 계속 유지되는 것이 아니라 서브쿼리의 범위를 벗어나면 더 이상 사용할 수 없게 된다

GROUP BY 절에서 그룹핑 기준을 정의하게 되면 DB는 일반적인 SELECT 문장처럼 FROM 절에 정의된 테이블의 구조를 그대로 가지고 오는 것이 아니라, GROUP BY 절의 그룹핑 기준에 사용된 컬럼과 집계 함수에 사용될 수 있는 숫자형 데이터 칼럼들의 집합을 새로 만든다. 이 때, 개별 데이터는 필요 없으므로 저장하지 않는다. GROUP BY 절 이후 수행되는 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.**

*** 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.

ORDER BY 절은 결과 집합을 결정하는데 관여하지 않는다.
→ 인라인 뷰를 활용하여 추출하고자 하는 집합을 먼저 정렬한 후 ROWNUM을 적용시킴으로써 결과에 참여하는 순서와 추출되는 행의 순서를 일치시킨다
ROWNUM 조건이 ORDER BY 절보다 먼저 처리되는 WHERE 절에서 처리하므로 정렬 후 원하는 데이터를 얻기 위해서는 **인라인 뷰에서 먼저 정렬을 수행한 후 메인쿼리에서 ROWNUM 조건을 사용**해야 한다.

**# TOP ( )** : SQL Server는 TOP 조건을 사용하게 되면 별도 처리 없이 관련 ORDER BY 절의 데이터 정렬 후 원하는 일부 데이터만 쉽게 출력한다

TOP 절을 사용하여 결과 집합으로 반환되는 행의 수를 제한할 수 있다

**WITH TIES 옵션**은 ORDER BY 절의 조건 기준으로 TOP N의 마지막 행으로 표시되는 추가 행의 데이터가 같을 경우 N + 동일 정렬 순서 데이터를 추가로 반환하도록 하는 옵션

**TOP (Expression) PERCENT WITH TIES**

**2-1-9. JOIN**

*** JOIN 개요**

**두 개 이상의 테이블들을 연결 또는 결합하여 데이터를 출력**하는 것을 JOIN이라고 한다
JOIN은 관계형 데이터베이스의 가장 큰 장점이면서 대표적인 핵심 기능
일반적인 경우 행들은 PK나 FK 값의 연관에 의해 JOIN이 성립된다
어떤 경우에는 이러한 PK, FK 관계가 없어도 논리적인 값들의 연관만으로 JOIN 성립이 가능

하나의 SQL 문장에서 여러 테이블을 조인해서 사용할 수도 있다. 다만 한 가지 주의할 점은 FROM 절에 여러 테이블이 나열되더라도 SQL에서 데이터를 처리할 때는 **두 개의 집합 간에만 JOIN이 일어난다**.

FROM 절에 A, B, C 3개의 테이블이 나열되었더라도 특정 2개의 테이블만 먼저 조인되고, 그 조인된 새로운 결과 집합과 남은 한 개의 테이블이 다음 차례로 조인되는 것이다.

*** EQUI JOIN**

EQUI JOIN은 두 테이블 간에 **칼럼 값들이 서로 정확히 일치하는 경우**에 사용되는 방법.

대부분 PK-FK 관계를 기반으로 한다. 그러나 일반적으로 테이블 설계시에 나타난 PK-FK의 관계를 이용하는 것이지 반드시 PK-FK 관계로만 EUQI JOIN이 성립하는 것은 아니다.

JOIN의 조건은 WHERE 절에 기술하게 되는데 "=" 연산자를 사용해서 표현한다.

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

SELECT 테이블1.칼럼명, 테이블2.칼럼명,
FROM 테이블1 **INNER JOIN** 테이블2 **ON** 테이블1.칼럼명1 = 테이블2.칼럼명2
→ ON 절에 JOIN 조건을 넣는다.

**N개의 테이블의 조인**에서 JOIN 조건은 대상 태이블의 개수에서 하나를 뺀 **N-1개 이상 필요**

JOIN 조건은 WHERE 절에 기술하며, JOIN은 두 개 이상의 테이블에서 필요한 데이터를 출력하기 위한 가장 기본적인 조건이다.

조건절에 ALIAS명 사용이 가능하다.
WHERE 절에서 JOIN 조건 이외의 검색 조건에 대한 조건을 덧붙여 사용할 수 있다.
( 조인 조건 명시 후, 논리 연산자를 이용하여 부수적인 제한 조건 추가로 입력 )

**JOIN 시 주의사항**
만약 테이블에 대한 ALIAS명을 적용하여 SQL 문장을 작성했을 경우, WHERE 절과 SELECT 절에는 테이블명이 아닌 ALIAS를 사용해야 한다.
권장사항은 아니지만 하나의 SQL 문장 내에서 유일한 컬럼명이면 ALIAS를 붙이지 않아도 O

*** Non EQUI JOIN**

Non EQUI JOIN은 두 개의 테이블 간 **칼럼 값들이 서로 정확하게 일치하지 않는 경우** 사용

"=" 연산자가 아닌 다른(Between, 부등호 등) 연산자를 사용하여 JOIN을 수행하는 것.

두 개의 테이블이 PK-FK로 연관관계를 가지거나 논리적으로 같은 값이 존재하는 경우에는 "=" 연산자를 이용하여 EQUI JOIN을 사용한다. 그러나 컬럼 값들이 서로 정확히 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**.

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

*** JOIN이 필요한 이유**

JOIN이 필요한 기본적인 이유는 정규화에서부터 출발한다.

정규화란 불필요한 데이터의 정합성을 확보하고 이상현상 발생을 피하기 위해, 테이블을 분할하여 생성하는 것이다. 사실 데이터 웨어하우스 모델처럼 하나의 테이블에 모든 데이터를 집중시켜놓고 그 테이블로부터 필요한 데이터를 조회할 수도 있다. 그러나 이러한 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. 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 JOIN 조건인 것이다.

관계형 데이터베이스의 큰 장점이면서, SQL 튜닝의 주요 대상이 되는 JOIN을 잘못 기술하게 되면, 시스템 자원 부족이나 과다한 응답시간 지연을 발생시키는 주요 원인이 되므로 JOIN 조건은 신중하게 작성해야 한다.

## 2-2. SQL 활용

### 2-2-1. 표준 조인

*** STANDARD SQL 개요**

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

**일반 집합 연산자**

UNION 연산은 **UNION** 기능으로
INTERSECTION 연산은 **INTERSECT** 기능으로
DIFFERENCE 연산은 **EXCEPT** 기능으로 ( Oracle은 **MINUS** )
PRODUCT 연산은 **CROSS JOIN** 기능으로

**순수 관계 연산자**

SELECT 연산은 WHERE 절로 구현
PROJECT 연산은 SELECT 절로 구현
(NATURAL) JOIN 연산은 다양한 JOIN 기능으로 구현
DIVIDE 연산은 현재 사용되지 않는다.

*** FROM 절의 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/ISO에서 규정한 JOIN 문법은 기존 JOIN 방식과 차이가 있다.

기존 WHERE 절 방식을 그대로 사용 가능,
추가된 선택 기능으로 **FROM 절**에서 **JOIN 조건을 명시적으로 정의** 가능.

**INNER JOIN**은 WHERE 절부터 사용하던 JOIN의 Default 옵션. **동일값**이 있는 행만 반환.
Default 옵션이므로 생략이 가능하지만, CROSS JOIN, OUTER JOIN과는 같이 사용 불가능.

**NATURAL JOIN**은 INNER JOIN의 하위 개념. 두 테이블 간의 **동일한 이름을 갖는 모든 컬럼들에 대해 EQUI JOIN**을 수행. NATURAL INNER JOIN이라고도 표시할 수 있음.

과거 WHERE 절에서 JOIN 조건과 데이터 검증 조건이 같이 사용되어 용도가 불분명한 경우가 발생할 수 있었는데 **WHERE 절의 JOIN 조건**을 **FROM 절의 ON 조건절로 분리**하여 표시함으로써 사용자가 이해하기 쉽도록 한다.

NATURAL JOIN처럼 JOIN 조건이 숨어있지 않고, **명시적으로 JOIN 조건을 구분**할 수 있고, NATURAL JOIN이나 USING 조건절처럼 칼럼명이 똑같아야 한다는 제약 없이 칼럼명이 상호 다르더라도 JOIN 조건으로 사용할 수 있으므로 앞으로 가장 많이 사용될 것으로 예상.

FROM 절에 테이블이 많이 사용될 경우 가독성이 떨어지는 단점
→ **SQL Server의 경우 ON 조건절만 지원** / NATURAL JOIN, USING 지원 X

*** INNER JOIN**

내부 JOIN이라고 하며 JOIN 조건에서 동일한 값이 있는 행만 반환.
그 동안 WHERE 절에서 사용하던 JOIN 조건을 FROM 절에서 정의하겠다는 표시
USING 조건절이나 ON 조건절을 필수적으로 사용.
JOIN에 사용된 두 테이블의 컬럼이 모두 출력된다.
중복 테이블의 경우 별개의 컬럼으로 표시한다.

*** NATURAL JOIN**

두 테이블 간 동일한 이름을 갖는 모든 칼럼들에 대해 EQUI JOIN 수행
NATURAL JOIN이 명시되면 WHERE 절에서 JOIN 조건 정의 X (USING, ON 조건절)
SQL Server에서는 지원 X
JOIN 컬럼을 지정하지 않아도 공통된 칼럼으로 JOIN 처리 함
JOIN에 사용된 컬럼은 같은 데이터 타입이여야 함
ALIAS나 테이블명과 같은 접두사를 붙일 수 X
별도로 순서를 지정하지 않으면 JOIN의 기준이 되는 칼럼들이 앞쪽에 출력된다.
JOIN에 사용된 같은 이름의 칼럼을 하나로 처리한다.

*** USING 조건절**

FROM 절에 USING 조건절을 이용하여 같은 이름의 칼럼 중 원하는 칼럼에 대해서만 EQUI JOIN을 할 수 있다 (NATURAL JOIN은 모든 일치 칼럼. USING은 원하는 칼럼)
SQL Server에서는 지원하지 않는다. JOIN 칼럼에 대해 ALIAS나 테이블명과 같은 접두사를 붙일 수 없다. JOIN에 참여한 칼럼은 1개로 표시된다.

*** ON 조건절**

ON 조건절을 사용하면 칼럼명이 달라도 JOIN 사용 가능. 임의의 JOIN 조건 사용 가능
이름이 다른 칼럼명을 JOIN 조건으로 사용 가능. JOIN 칼럼을 명시 가능.
ALIAS 및 테이블명과 같은 접두사를 사용 가능.
ON 조건절에 데이터 검색 조건을 추가할 수 있다. (WHERE 절 사용을 권고)

*** CROSS JOIN**

코드 박사의 일반 집합 연산자 중 PRODUCT의 개념.
테이블 간 JOIN 조건이 없는 경우 생길 수 있는 모든 데이터의 조합
CARTESIAN PRODUCT / CROSS PRODUCT 와 같은 표현
결과는 양쪽 집합의 M*N 건의 데이터 조합이 발생한다.

정상적인 데이터 모델이라면 CROSS JOIN이 필요한 경우는 많지 않지만, 간혹 튜닝이나 리포트 작성을 위해 고의적으로 사용하는 경우가 있을 수 있다.

그리고 DW의 개별 차원을 사실 칼럼과 JOIN 하기 전에 모든 차원의 CROSS PRODUCT를 먼저 구할 때 유용하게 사용할 수 있다.

*** OUTER JOIN**

**JOIN 조건에서 동일한 값이 없는 행도 반환할 때 사용**

+ 표시를 사용했었으나 검색 조건 불명확, IN / ON 연산자 사용 시 에러 발생,
+ 표시가 누락된 칼럼 존재 시 OUTER JOIN 오류 발생, FULL OUTER JOIN 미지원 등 불편함이 많아 STANDARD JOIN을 사용함으로써 많은 문제점 해결 및 호환성 확보

USING, ON 조건절 필수 사용. LEFT/RIGHT 경우 기준 테이블이 무조건 드라이빙 테이블

**# LEFT/RIGHT OUTER JOIN**

좌측 테이블에서 데이터를 먼저 읽은 후, 우측 테이블에서 JOIN 대상을 읽어옴.
좌측 테이블이 기준이 됨. OUTER 생략 가능. (RIGHT는 좌우만 변경 나머지 같음)

**# FULL OUTER JOIN**

조인되는 모든 테이블의 데이터를 읽어 JOIN 한다. LEFT/RIGHT 조인 결과의 합집합

**2-2-2. 집합 연산자**

*** 집합 연산자 (SET_OPERATOR)**

연관된 데이터를 조회하는 방법 중 하나
여러 개의 질의의 결과를 연결하여 하나로 결합하는 방식
→ 2개 이상의 질의 결과를 하나의 결과로 만들어준다

**# 일반적으로 집합 연산자를 사용하는 상황**

서로 다른 테이블에서 유사한 형태의 결과를 반환하는 것을 하나의 결과로 합치고자 할 때 사용.

동일 테이블에서 서로 다른 질의를 수행하여 결과를 합치고자 할 때 사용

이외에도 튜닝 관점에서 실행계획을 분리하고자 하는 목적으로도 사용

**# 집합 연산자를 사용하기 위한 제약조건 (지키지 않을 시 DB가 오류 반환)**

SELECT 절의 칼럼수가 동일
SELECT 절의 동일 위치 칼럼의 데이터 타입이 상호 호환 가능 (반드시 동일 필요 X)

**# 집합 연산자의 종류**

**UNION** : 합집합, 중복된 행 하나로 표시
**UNION ALL** : 합집합, 중복된 행 그대로 표시 (단순히 결과만 합쳐놓음) 일반적으로 질의 결과들이 상호 배타적일 때 많이 사용. 중복 없을 시 UNION과 결과가 동일하다. (정렬 순서에는 차이가 있을 수 있음)

**INTERSECT** : 교집합, 중복된 행 하나로 표시
**EXCEPT** : 앞의 SQL 결과에서 뒤의 결과를 뺀 차집합, 중복 행 하나로 표시
( Oracle에서는 MINUS 사용 )

SELECT 칼럼명1, 칼럼명2, ...
FROM 테이블명1
[ WHERE 조건식 ] [ GROUP BY 칼럼/표현식 HAVING 그룹조건식 ]
**집합연산자**
SELECT 문
[ ORDER BY 칼럼명 [ ASC | DESC ] ]

집합 연산자는 사용상의 제약조건을 만족한다면 어떤 형태의 SELECT 문이라도 이용이 가능. 집합 연산자는 여러개의 SELECT 문을 연결하는 것에 지나지 않음
ORDER BY는 최종 결과에 대한 정렬 처리이므로 가장 마지막 줄에 한 번만.

### 2-2-3. 계층형 질의와 셀프 조인

*** 계층형 질의**

테이블에 계층형 데이터가 존재하는 경우 데이터를 조회하기 위해 사용

동일 테이블에 계층적으로 상/하위 데이터가 포함된 데이터를 계층형 데이터라 함
ex) 사원 테이블의 사원들 사이에 하위 사원과 상위 사원(관리자) 관계

엔터티를 순환관계 데이터 모델로 설계할 경우 계층형 데이터 발생
계층형 데이터 조회는 DBMS 벤더와 버전에 따라 다른 방법으로 지원한다

**# Oracle 계층형 질의**

**# 계층형 질의에서 사용되는 가상 칼럼**

**LEVEL** : 루트 데이터를 1로 시작하여 하위로 내려갈수록 Leaf 까지 1씩 증가.
**CONNECT_BY_ISLEAF** : 리프데이터이면 1, 그렇지 않으면 0
**CONNECT_BY_ISCYCLE** : 자식을 갖는데, 해당 데이터가 조상으로서 존재하면 1,
그렇지 않으면 0. CYCLE 옵션 사용시에만 사용 가능.

**# 계층형 질의 구문**

**SELECT 칼럼 .....**
**FROM 테이블**
**WHERE 조건 .....**
**START WITH 조건**
**CONEECT BY [NOCYCLE] 조건 .....**
**[ORDER SIBLINGS BY 칼럼, ,,,,,,]**

**START WITH 절** (엑세스) : 계층 구조 전개의 시작 위치를 지정하는 구문

**CONNECT BY 절** (조인) : 다음에 전개될 자식 데이터를 지정하는 구문
자식 데이터는 CONNECT BY 절 주어진 조건 만족

**PRIOR** : CONNECT BY 절에 사용되며, 현재 읽은 칼럼 지정
PRIOR 자식 = 부모 : [ 부모 → 자식 ] 순방향 전개
PRIOR 부모 = 자식 : [ 자식 → 부모 ] 역방향 전개

**NOCYCLE** : 데이터 전개 시, 이미 나타났던 동일한 데이터가 전개 중에 다시 나타난다면 이것을 CYCLE 형성이라 함
사이클이 발생한 데이터는 런타임 오류 발생
→ NOCYCLE 추가 - 사이클 발생 이후의 데이터는 전개 X

**ORDER SIBLINGS BY** - 형제 노드(동일 Level) 사이에서 정렬

**WHERE** : 모든 전개를 수행한 후에 지정된 조건을 만족하는 데이터만 추출

**# 계층형 질의에서 사용되는 함수**

**SYS_CONEECT_BY_PATH(칼럼, 경로분리자)**
루트 데이터부터 현재 전개할 데이터까지의 경로를 표시

**CONNECT_BY_ROOT 칼럼**
현재 전개할 데이터의 루트 데이터를 표시. 단항 연산자

**# SQL Server 계층형 질의**

SQL Server 2000 까지는 계층형 질의 문법 지원 X
조직도처럼 계층적 구조 가진 데이터는 저장 프로시저 재귀 호출이나
While 루프 문에서 임시 테이블을 사용하는 등 프로그램 방식으로 전개
SQL Server 2005 버전부터 하나의 질의로 수행 가능

CTE(Common Table Expression)를 재귀 호출함으로써 상위부터 하위 방향 전개

```
WITH 테이블명_ANCHOR AS
( SELECT 하위칼럼명, 칼럼명, 상위칼럼명, 0 AS LEVEL
  FROM 테이블명
  WHERE 상위칼럼명 IS NULL /* 재귀 호출의 시작점 */
  UNION ALL
  SELECT R.칼럼명, R.칼럼명, R.계층칼럼명, A.LEVEL + 1
  FROM 테이블명_ANCHOR A, 테이블명 R
  WHERE A.하위칼럼 = R.상위칼럼 )
```

WITH 절의 CTE 쿼리를 보면 **UNION** 연산자로 쿼리 두 개를 결합.

**위**의 쿼리를 **앵커 멤버**, **아래**의 쿼리를 **재귀 멤버**라고 함

**# 재귀적 쿼리의 처리 과정**

1. CTE 식을 앵커 멤버와 재귀 멤버로 분할
2. 앵커 멤버를 실행하여 첫 번째 호출 또는 기본 결과 집합 T(0) 생성
3. T(i)는 입력으로 사용하고 T(i)+1은 출력으로 사용하여 재귀 멤버 실행
4. 빈 집합이 반활될 때가지 3단계 반복
5. 결과 집합을 반환. T(0)에서 T(n)까지의 UNION ALL

앵커 멤버가 시작점이자 Outer 집합이 되어 Inner 집합인 재귀 멤버와 조인 시작
앞서 조인한 결과가 다시 Outer 집합이 되어 재귀 멤버와 조인을 반복
조인 결과가 비어 있으면 (더 조인할 수 없으면) 지금까지 결과 집합 합쳐 리턴

계층 구조를 단순히 하위 방향으로 전개했을 뿐 조직도(실제)와는 다른 모습
조직도와 같은 모습으로 출력하려면 ORDER BY 절 추가해 원하는 순서로 정렬
→ 정렬 기준으로 삼을 수 있는 정렬용 칼럼 추가하여 ORDER BY 조건 추가
( 앵커 멤버와 재귀 멤버 양쪽에서 Convert 함수 등으로 데이터 형식 일치 )

*** 셀프 조인**

동일 테이블 사이의 조인 (FROM 절에 동일 테이블이 두 번 이상 나타남)
테이블과 칼럼명이 모두 동일하기 때문에 식별을 위해 반드시 ALIAS 사용
칼럼에도 모두 테이블 ALIAS를 사용해서 어느 테이블의 칼럼인지 식별
이외 나머지 사항은 조인과 동일

```
SELECT ALIAS명1.칼럼명1, ALIAS명2.칼럼명1, ....
FROM 테이블명 ALIAS명1, 테이블명 ALIAS명2
WHERE ALIAS명1.칼럼명2 = ALIAS명2.칼럼명1
```

OUTER JOIN을 이용해서 상위가 존재하지 않는 데이터까지 모두 결과 표시 가능

**2-2-4. 서브쿼리**

*** 서브쿼리**

하나의 SQL문 안에 포함되어 있는 또 다른 SQL문을 말한다.

*** 서브쿼리 사용시 주의사항**

서브쿼리를 괄호로 감싸서 사용
서브쿼리는 단일 행 또는 복수 행 비교 연산자와 함께 사용이 가능하다
  단일 행 비교 연산자 - 서브쿼리의 결과가 반드시 1건 이하
  복수 행 비교 연산자 - 서브쿼리의 결과 건수와 상관 없다
서브쿼리에서는 ORDER BY를 사용하지 못한다. ORDER BY 절은 SELECT 절에서
오직 한 개만 올 수 있기 때문에 ORDER BY 절은 메인쿼리의 마지막 문장 위치.

*** 서브쿼리가 SQL 문에서 사용이 가능한 곳**

SELECT 절, FROM 절, WHERE 절, HAVING 절, OREDER BY 절,
INSERT 문의 VALUES 절, UPDATE 문의 SET 절

*** 동작 방식에 따른 서브쿼리 분류**

**Un-Correlated(비연관)** 서브쿼리 : 서브쿼리가 메인쿼리 칼럼을 갖고 있지 않은 형태.
메인쿼리에 값을 제공하기 위한 목적으로 주로 사용

**Correlated(연관)** 서브쿼리 : 서브쿼리가 메인쿼리 칼럼을 가지고 있는 형태. 일반적으로
메인쿼리가 먼저 수행되어 읽혀진 데이터를 서브쿼리에서
조건이 맞는지 확인하고자 할 때 주로 사용

*** 반환되는 데이터의 형태에 따른 서브쿼리 분류**

**Single Row(단일 행)** 서브쿼리 : 서브쿼리 결과가 항상 1건 이하인 서브쿼리.
단일 행 비교 연산자와 함께 사용 (등호, 부등호)

**Mulit Row(다중 행)** 서브쿼리 : 실행 결과가 여러 건인 서브쿼리. 다중 행 비교 연산자와
함께 사용 (IN, ALL, ANY, SOME, EXISTS 등)

**Multi Column(다중 컬럼)** 서브쿼리 : 실행 결과로 여러 컬럼을 반환. 메인쿼리의
조건절에 여러 칼럼을 동시에 비교 가능.
서브쿼리와 메인쿼리에서 비교하고자 하는 칼럼

개수와 위치가 동일해야 한다.

---

* **Correlated(연관) 서브쿼리** : 서브쿼리 내에 메인쿼리 칼럼이 사용된 서브쿼리

* **그 밖의 위치에서 사용하는 서브쿼리**

**SELECT 절에 서브쿼리 사용**
→ **스칼라 서브쿼리** (Scalar Subquery) : 한 행, 한 칼럼만을 반환

**FROM 절에서 서브쿼리 사용**
→ **인라인 뷰** (Inline View) : SQL 문이 실행될 때만 임시적으로 생성되는 동적인
뷰이기 때문에 DB에 해당 정보가 저장되지 X.
그래서 일반 뷰를 정적 뷰(Static View)라고 하고
인라인 뷰를 동적 뷰(Dynamic View)라고도 한다.

**HAVING 절에서 서브쿼리 사용**
→ 그룹함수와 함께 사용될 때 그룹핑된 결과에 대해 부가 조건을 주기 위해 사용

**UPDATE 문의 SET 절에서 사용**
→ 서브쿼리를 사용한 변경 작업을 할 때 서브쿼리의 결과가 NULL을 반환할 경우
해당 컬럼의 결과가 NULL이 될 수 있기 때문에 주의

**INSERT 문의 VALUES 절에서 사용**

*** 뷰 (View)**

테이블은 실제로 데이터를 가지고 있는 반면, 뷰는 실제 데이터를 가지고 있지 않다.
뷰는 단지 뷰의 정의만을 가지고 있다. 질의에서 뷰가 사용되면 뷰 정의를 참조해서
DBMS 내부적으로 질의를 재작성하여 수행한다. 뷰는 실제 데이터를 가지고 있지 않지만
테이블이 수행하는 역할을 수행하기 때문에 가상 테이블이라고도 한다.

**뷰의 장점** : **독립성** - 테이블 구조가 변경되어도 뷰를 사용하는 응용 프로그램은 변경 X
**편리성** - 복잡한 질의를 뷰로 생성함으로써 관련 질의를 단순하게 작성 가능
또한 해당 형태의 SQL문을 자주 사용할 때 뷰를 이용하면 편리
**보안성** - 직원의 급여정보와 같이 숨기고 싶은 정보가 존재한다면, 뷰를
생성할 때 해당 칼럼을 빼고 생성하여 사용자에게 정보를 감추기 가능

2-2-5. 그룹 함수

*** 데이터 분석 개요**

# ANSI/ISO SQL 표준은 데이터 분석을 위해 다음 세 가지 함수를 정의

**AGGREGATE FUNCTION** : GROUP AGGREGATE FUNCTION이라고도 부르며,
GROUP FUNCTION의 한 부분으로 분류할 수 있다.
각종 **집계 함수**들이 포함되어 있다.

**GROUP FUNCTION** :

**ROLLUP**은 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.

**CUBE**는 결합 가능한 모든 값에 대하여 다차원적인 집계를 생성하게 되므로 ROLLUP에 비해 다양한 데이터를 얻는 장점이 있는 반면, 시스템에 부하를 많이 주는 단점이 있다.

**GROUPING SETS**는 원하는 부분의 소계만 손쉽게 추출할 수 있는 장점이 있다.

ROLLUP, CUBE, GROUPING SETS **결과에 대한 정렬이 필요한 경우는 ORDER BY 절**에 정렬 칼럼을 명시해야 한다.

**WINDOW FUNCTION** : 분석함수나 순위함수로도 알려져 있는 윈도우 함수는 DW에서 발전한 기능이며 다음 절에서 자세히 설명.

*** ROLLUP 함수**

ROLLUP에 지정된 Grouping Columns의 List는 **Subtotal을 생성**하기 위해 사용되며, Grouping Columns의 수를 N이라고 했을 때 N+1 Level의 Subtotal이 생성된다.
→ 인수1로 묶인 인수2의 계, 인수2로 묶인 인수3의 계, .... , 총 합계

ROLLUP의 인수는 **계층 구조**이므로 순서가 바뀌면 수행 결과도 바뀌니 순서에 주의.

GROUP BY 절에 인수를 묶어 사용 → **GROUP BY ROLLUP ( 인수1, 인수2, ....)**

※ **GROUPING 함수** 사용 : **GROUPING(EXPR) = (소계 결과 1 / 그 외에는 0)**
CASE/DECODE를 함께 사용해 소계 필드에 원하는 문자열

지정 가능하여, 보고서 작성시 유용하게 사용 가능

*** CUBE 함수**

ROLLUP에서는 단지 가능한 Subtotal만을 생성하였지만, CUBE는 **결합 가능한 모든 값에 대하여 다차원 집계**를 생성한다.

CUBE를 사용할 경우에는 내부적으로는 Grouping Columns의 순서를 바꾸어서 또 한 번의 Query를 추가 수행해야 한다.

뿐만 아니라, Grand Total은 양 쪽의 Query에서 모두 생성이 되므로 한 번의 Query에서는 제거되어야 하기 때문에 ROLLUP에 비해 시스템 연산 대상이 많다.

이처럼 Grouping Columns가 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, 이 때 표시된 인수들 간에는 계층 구조인 ROLLUP과는 달리 평등한 관계이므로 인수의 순서가 바뀌는 경우 행간에 정렬 순서는 바뀔 수 있어도 데이터 결과는 같다.

GROUP BY 절에 인수를 묶어 사용 → **GROUP BY CUBE ( 인수1, 인수2, .... )**

*** GROUPING SETS 함수**

더욱 다양한 소계 집합을 만들 수 있다. GROUP BY 문장을 여러 번 반복하지 않아도 원하는 결과를 쉽게 얻을 수 있다. 표시된 **인수들에 대한 개별 집계**를 구할 수 있으며, 이 때 표시된 인수들 간에는 계층구조인 ROLLUP과는 달리 **평등한 관계**이므로 인수의 순서가 바뀌어도 결과는 같다. 결과에 대한 정렬이 필요한 경우 ORDER BY 절에 명시.

UNION ALL을 사용한 일반 그룹함수를 사용한 SQL과 같은 결과를 얻을 수 있으며,
괄호로 묶은 집합 별로 집계를 구할 수 있다. (괄호 내는 계층 구조 X)
일반 그룹함수를 이용한 SQL과 결과는 같으나 행들의 정렬 순서는 다를 수 있다.

GROUP BY 절에 인수를 묶어 사용 → **GROUP BY GROUPING SETS ( 인수, 인수, .... )**

**2-2-6. 윈도우 함수**

*** WINDOW FUNCTION 개요**

행과 행 간의 관계를 쉽게 정의하기 위해 만든 함수

**# WINDOW FUNCTION의 종류**

**순위**(RANK) : **RANK, DENSE_RANK, ROW_NUMBER**
**집계**(AGGREGATE) : **SUM, MAX, MIN, AVG, COUNT**
[ SQL Server 집계함수 OVER 절 내의 OREDER BY 구문 지원 X ]
**행 순서** [Oracle] : **FIRST_VALUE**, **LAST_VALUE**, **LAG**, **LEAD**
**비율** : **CUME_DIST**, **PERCENT_RANK**, **NTILE**, **RATIO_TO_REPORT**
[Oracle] [Oracle] [Oracle]

# WINDOW FUNCTION SYNTAX

윈도우 함수에는 OVER 문구가 필수 키워드로 포함

SELECT 칼럼명, 칼럼명, .... ,
**WINDOW_FUNCTION** (ARGUMENTS)
**OVER** ([PARTITION BY 칼럼] [ORDER BY 절] [WINDOWING 절]) **ALIAS**
FROM 테이블명;

PARTITION BY 절 조건과 ORDER BY 절 조건이 충돌 시 ORDER BY 절로 정렬

- **ARGUMENTS** (인수) : 함수에 따라 0~N개의 인수
- **PARTITION BY** 절 : 전체 집합을 기준에 의해 소그룹으로 나눌 수 있다.
- **ORDER BY** 절 : 어떤 항목에 대해 순위를 지정할지 ORDER BY절을 기술한다.
- **WINDOWING** 절 : 함수의 대상이 되는 행 기준의 범위를 강력히 지정 가능
→ ROWS 물리적인 결과 행 수 / RANGE 논리적인 값에 의한 범위
[ SQL Server WINDOWING 절 지원 X ]

**ROWS | RANGE [BETWEEN]**
**UNBOUNDED PRECEDING | CURRENT ROW | VALUE_EXPR PRECEDING/FOLLOWING**
**[AND**
**UNBOUNDED FOLLOWING | CURRENT ROW | VALUE_EXPR PRECEDING/FOLLOWING]**

**ROWS는 현재 행의 앞/뒤 건수, RANGE는 현재 행 값 기준 앞/뒤 값 범위**

현재부터 끝 → **[BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING]**

*** 그룹 내 순위 함수**

**# RANK** 함수 : 특정 항목(컬럼) / 특정 범위(파티션) / 전체 데이터에 대한 순위를 구하는 함수. 동일한 값에 대해서는 동일한 순위 (다음 순위 스킵)

**# DENSE_RANK** 함수 : RANK와 흡사하나, 동일한 순위를 하나의 건수로 취급 (스킵 X)

**# ROW_NUMBER** 함수 : 동일한 값이라도 고유한 순위를 부여

*** 일반 집계 함수**

**# SUM** 함수 : 파티션별 윈도우의 합 (같은 ORDER BY 순서 경우 합쳐서 계산)

**# MAX** 함수 : 파티션별 윈도우의 최대값

**# MIN** 함수 : 파티션별 윈도우의 최소값

**# AVG** 함수 : 파티션별 윈도우의 평균값

*** 그룹 내 행 순서 함수 [SQL Server X]**

**# FIRST_VALUE** 함수 : 파티션별 윈도우의 최초로 나온 값
MIN 이용하여 같은 결과 가능 (공동 등수 인정 X)

**# LAST_VALUE** 함수 : 파티션별 윈도우의 가장 마지막에 나온 값
MAX 이용하여 같은 결과 가능 (공동 등수 인정 X)

공동 등수 의도적으로 정렬하고 싶다면 인라인 뷰, ORDER BY조건 이용

**# LAG** 함수 : 파티션별 윈도우에서 이전 몇 번째 행의 값
**LAG (인수, 몇행 앞, 없는 경우 반환)**

**# LEAD** 함수 : 파티션별 윈도우에서 이후 몇 번째 행의 값
**LEAD (인수, 몇행 뒤, 없을 경우 반환)**

*** 그룹 내 비율 함수 [SQL Server X]**

**# RATIO_TO_REPORT** 함수 : 파티션 내 전체 SUM(칼럼) 값에 대한 행별 칼럼 값의
백분율을 소수점으로 반환 [ 0 < 결과 값 <= 1 ]
개별 RATIO 합을 구하면 1

**# PERCENT_RANK** 함수 : 파티션별 윈도우에서 제일 먼저 나오는 것 0, 제일 늦게
나오는 것을 1로 하여, 값이 아닌 행의 순서별 백분율 반환
[ 0 <= 결과 값 <= 1 ]
같은 ORDER 순위 인정 → 앞 행의 결과 값을 기준으로 삼음

**# CUME_DIST** 함수 : 파티션별 윈도우의 전체 건수에서 현재 행보다 작거나 같은
건수에 대한 누적 백분율. [ 0 < 결과 값 <= 1 ]
같은 ORDER 순위 인정 → 뒤 행의 결과 값을 기준으로 삼음

**# NTILE** 함수 : 파티션별 전체 건수를 ARGUMENT 값으로 N등분한 결과
나머지는 앞의 그룹부터 차례로 할당

**2-2-7. DCL**

*** DCL 개요**

유저를 생성하고 권한을 제어할 수 있는 명령어

*** 유저와 권한**

**# Oracle과 SQL Server의 사용자에 대한 차이점**

Oracle은 유저를 통해 데이터베이스에 접속하는 형태. 즉, 아이디와 비밀번호 방식으로
인스턴스에 접속을 하고 그에 해당하는 스키마에 오브젝트 생성 등의 권한을 부여받음

SQL Server는 인스턴스에 접속하기 위해 로그인이라는 것을 생성하게 되며, 인스턴스 내에
존재하는 다수의 데이터베이스에 연결하여 작업하기 위해 유저를 생성한 후, 로그인과
유저를 매핑해 주어야 함. 더 나아가 특정 유저는 특정 DB 내의 특정 스키마에 대해 권한을
부여받을 수 있다.

SQL Server 로그인 방식 → **Window 인증 방식 / 혼합 모드 방식**

*** 유저 생성과 시스템 권한 부여**

유저를 생성하고 데이터베이스에 접속한다. 하지만 데이터베이스에 접속했다고 해서
테이블, 뷰, 인덱스 등과 같은 오브젝트를 생성할 수는 없다.
사용자가 실행하는 **모든 DDL 문장은 그에 해당하는 적절한 권한**이 있어야만 가능
시스템 권한은 약 100개 이상의 종류가 있어 롤(ROLE)울 이용하여 권한을 부여한다
Oracle 기본 제공 유저 : SCOTT/TIGER (테스트용 샘플 유저)
SYS (DBA ROLE 부여받은 유저)
SYSTEM (모든 권한 부여받은 DBA 유저 - 설치시 패스워드 설정)

**# Oracle**

DBA 권한을 가지고 있는 SYSTEM 유저로 접속하여 다른 유저에게 생성 권한을 부여

GRANT CREATE USER TO SCOTT;
CONN SCOTT/TIGER
CREATE USER 유저명 IDENTIFIED BY 비밀번호;

유저가 생성되었지만 로그인을 하려면 CREATE SESSION 권한이 필요

CONN SCOTT/TIGER
GRANT CREATE SESSION TO 유저명;
CONN 유저명/패스워드

이후 테이블을 생성할 때에도 CREATE TABLE 권한을 부여 받아야 생성이 가능하다.

**# SQL Server**

유저 생성 전 먼저 로그인을 생성해야 함
로그인을 생성할 수 있는 권한을 가진 로그인은 기본적으로 sa이다

CREATE LOGIN 로그인명 WITH PASSWORD='비밀번호',
DEFAULT_DATABASE=로그인 후 최초 접속할 데이터베이스명;

SQL Server에서 유저는 데이터베이스마다 존재하기 때문에 유저를 생성하기 위해서는
생성하고자 하는 유저가 속할 DB로 이동한 후 처리한다.

USE 데이터베이스명;
GO
CREATE USER 유저명 FOR LOGIN 로그인명 WITH DEFAULT_SCHEMA=dbo;

*** OBJECT에 대한 권한 부여**

오브젝트 권한은 특정 오브젝트인 테이블, 뷰 등에 대한 SELECT, INSERT, DELETE, UPTATE 작업 명령어를 의미한다.

Oracle에서 자신이 생성한 테이블 외에 다른 유저의 테이블에 접근하려면 해당 테이블에 대한 오브젝트 권한을 소유자로부터 부여받아야 한다.

SQL Server도 같은 방식으로 동작한다.
한 가지 다른 점은 유저는 단지 스키마에 대한 권한만을 가진다. 다시 말하면 테이블과 같은 오브젝트는 유저가 소유하는 것이 아니고 스키마가 소유하게 되며 유저는 스키마에 대해 특정한 권한을 가지는 것이다.

SQL Server는 객체 앞에 소유한 유저의 이름을 붙이는 것이 아니고 객체가 속한 스키마 이름을 붙여야 한다.

**GRANT SELECT ON 테이블명 TO 유저명;**

*** ROLE을 이용한 권한 부여**

유저를 생성하게 되면 많은 권한을 부여해야 한다. DBA는 유저가 생성될 때마다 각각의 권한들을 유저에게 부여하는 작업을 수행해야 하며 간혹 권한을 빠뜨리는 등 실수를 할 수 있으므로 각 유저별로 어떤 권한이 부여되었는지를 관리해야 한다.

DBA는 ROLE을 생성하고, ROLE에 각종 권한들을 부여한 후 다른 ROLE이나 유저에게 부여할 수 있다. ROLE에 포함되어 있는 권한들이 필요한 유저에게는 해당 ROLE만 부여함으로써 빠르고 정확하게 필요한 권한을 부여할 수 있게 된다.

ROLE에는 시스템 권한과 오브젝트 권한을 모두 부여할 수 있으며, ROLE은 유저에게 직접 부여될 수도 있고, 다른 ROLE에 포함하여 유저에게 부여될 수도 있다.

[Oracle]
**CONN SYSTEM/MANAGER**
**CREATE ROLE 롤이름;**
**GRANT 부여권한, 부여권한, ..... TO 롤이름;**
**GRANT 롤이름 TO 유저명;**

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 ROLE**은 **CONNECT**와 **RESOURCE**이다.

CONNECT는 CREATE SESSION과 같은 **로그인 권한**, RESOURCE는 CREATE TABLE과 같은 **오브젝트 생성 권한**이 포함된 권한들을 가지고 있다. 일반적으로 유저를 생성할 때 이 두 가지를 사용하여 기본 권한을 부여한다.

USER를 삭제하는 명령어는 DROP USER이고, CASCADE 옵션을 주면 해당 유저가 생성한 오브젝트를 먼저 삭제한 후 유저를 삭제한다.

**DROP USER 유저명 CASCADE;**

SQL Server에서는 위와 같이 ROLE을 생성하여 사용하기 보다는 기본적으로 제공되는 ROLE에 멤버로 참여하는 방식으로 사용한다. (서버 수준 역할)

**# 특정 로그인이 멤버로 참여할 수 있는 서버 수준 역할**

**public** : 모든 SQL Server 로그인은 PUBLIC 서버 역할에 속한다. 특정 사용 권한이 부여되지 않았거나 거부된 경우 사용자는 해당 개체에 대해 public으로 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, 거부, 취소할 수 있다. 로그인 암호도 다시 설정 가능
**severadmin** : 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 sysadmin 고정 서버 역할의 멤버이다.

데이터베이스에 존재하는 **유저**에 대해서는 **데이터베이스 역할**의 멤버로 참여할 수 있다.

**# 데이터베이스 수준 역할명**

**db_accessadmin** : Window 로그인, Window 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 X.
**db_denydatawriter** : 데이터베이스 내의 모든 사용자 테이블에 있는 데이터 추가, 삭제, 변경할 수 X.
**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 데이터베이스 수준의 권한이 필요한 경우 데이터베이스 수준 역할을 부여하면 된다.

즉, 인스턴스 수준을 요구하는 로그인에는 서버 수준 역할을, 데이터베이스 수준을 요구하는 사용자에게는 데이터베이스 수준 역할을 부여한다.

## 2-2-8. 절차형 SQL

*** 절차형 SQL 개요**

일반적인 개발 언어처럼 SQL에도 절차 지향적인 프로그램이 가능하도록 DBMS 벤더별로 절차형 SQL을 제공 [ Oracle : **PL/SQL**, SQL Server : **T-SQL**, DB2 : SQL/PL ]

절차형 SQL을 이용하면 SQL문의 연속적인 실행이나 조건에 따른 분기처리를 이용하여 특정 기능을 수행하는 **저장 모듈을 생성**할 수 있다

절차형 SQL을 이용하여 **Procedure**, **User Defined Function**, **Trigger** 저장 모듈 생성 가능

*** PL/SQL 개요**

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 특징**

PL/SQL은 Block 구조로 되어있어 각 기능별로 모듈화가 가능하다.
변수, 상수 등을 선언하여 SQL 문장 간 값을 교환한다.
If, Loop 등의 절차형 언어를 사용하여 절차적인 프로그램이 가능하도록 한다.
DBMS 정의 에러나 사용자 정의 에러를 정의하여 사용할 수 있다.

PL/SQL은 Oracle에 내장되어 있으므로 Oracle과 PL/SQL을 지원하는 어떤 서버로도 프로그램을 옮길 수 있다.

PL/SQL은 응용프로그램의 성능을 향상시킨다. 여러 SQL 문장을 Block으로 묶고 한 번에 Block 전부를 서버로 보내기 때문에 통신량을 줄일 수 있다.

PL/SQL은 Block 프로그램을 입력받으면 SQL 문장과 프로그램 문장을 구분하여 처리한다.
**프로그램 문장 → PL/SQL 엔진**,
**SQL 문장 →** Oracle 서버의 **SQL Statement Executor**

*** PL/SQL 구조**

**DECLARE**
　　**선언부** (변수, 상수) → BEGIN~END에서 사용할 변수나 인수 정의 및 타입 선언
**BEGIN**
　　**실행부** → 개발자가 처리하고자 하는 SQL문, 필요한 로직 정의
**EXCEPTION**
　　**예외 처리부** → BEGIN~END에서 실행되는 SQL문에 발생된 에러 처리
**END**

*** PL./SQL 기본 문법 (Syntax)**

Stored Procedure 통해 기본적인 문법을 정리한다. (나머지도 비슷함)

**CREATE OR REPLACE Procedure 프로시저명 (argument1 mode data_type1, ... ) IS AS .....**
**BEGIN .....**
**EXCEPTION .....**
**END; /**

**DROP Procedure 프로시저명;**

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 절**은 데이터베이스 내에 같은 이름의 프로시저가 있을 경우, 기존의 프로시저를 무시하고 새로운 내용으로 덮어쓰기 하겠다는 의미이다.

**Argument**는 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.
[ **IN** - 운영체제→프로시저, **OUT** - 프로시저→운영체제, INOUT - 두 가지 기능 동시에 ]

마지막에 있는 **"/"**는 데이터베이스에게 프로시저를 **컴파일하라는 명령어**이다.

*** T-SQL 개요 및 특징**

T-SQL은 근본적으로 SQL Server를 제어하기 위한 언어로서, 엄격히 말하면 MS사에서 ANSI/ISO 표준의 SQL에 약간의 기능을 더 추가해 보완적으로 만든 것이다.
T-SQL을 이용하여 다양한 저장 모듈을 개발할 수 있다. 프로그래밍 기능은 다음과 같다.

- **변수 선언 기능** : @@ - 전역변수(시스템 함수), @ - 지역변수
  지역변수는 사용자가 자신의 연결 시간동안만 사용하기 위해 만들어지는 변수이며
  전역변수는 이미 SQL서버에 내장된 값이다.
- **데이터 타입을 제공**한다. 즉, INT, FLOAT, VARCHAR 등의 자료형을 의미한다.
- **산술연산자, 비교연산자, 논리연산자 사용이 가능**하다.
- **흐름 제어 기능** : IF-ELSE 와 WHILE, CASE-THEN 사용이 가능하다.
- **주석 기능** : 한줄 주석은 -- 뒤의 내용 주석, 범위 주석은 /* 내용 */ 형태를 사용하며 여러 줄도 가능하다.

*** T-SQL 구조**

PL/SQL과 유사하다.

**DECLARE**
**선언부 (변수, 상수)** → 사용할 변수나 인수에 대한 정의 및 데이터 타입 선언
**BEGIN**
**실행부** → 개발자가 처리하고자 하는 SQL문과 필요한 로직 정의
**ERROR 처리**
**예외 처리부** → 발생된 에러를 처리하는 에러 처리부
**END**

*** T-SQL 기본 문법 (Syntax)**

```
CREATE Procedure 스키마명.프로시저명 @parameter1 data_type1 mode, ......
WITH AS ......
BEGIN ......
ERROR 처리 ........
END;

DROP Procedure 스키마명.프로시저명;
```

프로시저의 변경이 필요할 경우 Oracle은 CREATE OR REPLACE와 같이 하나의 구문으로 처리하지만 SQL Server는 CREATE 구문을 ALTER 구문으로 변경하여야 한다.

**Mode** 부분에 지정할 수 있는 매개변수(@parameter)의 유형은 4가지가 있다.

**VARYING** : 결과 집합이 출력 매개변수로 사용되도록 지정 (CURSOR 매개변수에만 적용)

**DEFAULT** : 지정된 매개변수가 프로시저를 호출할 당시 지정되지 않을 경우 기본값으로 처리
즉, 기본값이 지정되어 있으면 해당 매개변수 지정하지 않아도 프로시저가 지정된 기본값으로 정상적으로 수행한다.

**OUT/OUTPUT** : 프로시저에서 처리된 결과 값을 EXECUTE 문 호출시 반환

**READONLY** : 프로시저 본문 내에서 매개변수를 업데이트하거나 수정할 수 없음을 나타냄
매개변수 유형이 사용자 정의 테이블 형식인 경우 READONLY 지정해야 함.

**WITH** 부분에 지정할 수 있는 옵션은 3가지가 있다.

**RECOMPILE** : 데이터베이스 엔진에서 현재 프로시저의 계획을 캐시하지 않고 프로시저가 런타임에 컴파일된다. 저장 프로시저 안에 있는 개별 쿼리에 대한 계획을 삭제하려 할 때 RECOMPILE 쿼리 힌트를 사용한다.

**ENCRYPTION** : CREATE PROCEDURE 문의 원본 텍스트가 알아보기 어려운 형식으로 변환됨. 변조된 출력은 SQL Server의 카탈로그 뷰 어디에서도 직접 표시되지 않는다 원본을 볼 수 있는 방법이 없기 때문에 반드시 백업을 해두어야 한다.

**EXECUTE AS** : 해당 저장 프로시저를 실행할 보안 컨텍스트를 지정한다.

*** Procedure의 생성과 활용**

**Scalar 변수** : 사용자의 임시 데이터를 하나만 저장할 수 있는 변수이며 거의 모든 형태의 데이터 유형을 지정할 수 있다.

PL/SQL에서 사용하는 SQL 구문은 대부분 동일하게 사용할 수 있지만 **SELECT 문장은 반드시 결과값**이 있어야 하며, 그 **결과 역시 반드시 하나**여야만 한다. 조회 결과가 없거나 하나 이상인 경우에는 에러를 발생시킨다. (T-SQL에서는 결과 없어도 에러 발생 X)

PL/SQL에서는 대입 연사자를 **":="**로 사용한다.

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 있다.

1. 프로시저를 실행한 결과 값을 받을 변수를 선언한다 (**BIND 변수**)
   → variable 변수명 varchar2(N);
2. 프로시저를 실행한다
   → EXECUTE 프로시저명(인수1, 인수2, ... , :BIND변수명)
3. BIND 변수를 출력하여 반환된 결과를 확인해 볼 수 있다.
   → print BIND변수명;

T-SQL로 작성한 프로시저 실행을 위해서는 일반적으로 SQL Server에서 제공하는 기본 클라이언트 프로그램인 **SQL Server Management Studio**를 사용한다.

1. DECLARE @v_result VARCHAR(100)
2. EXECUTE dbo.프로시저명 인수, 인수, @v_result=@v_result OUTPUT
3. SELECT @v_result AS RSLT

*** User Defined Function의 생성과 활용**

Procedure처럼 절차형 SQL을 로직과 함께 데이터베이스 내에 저장해 놓은 명령문의 집합을 의미한다. SUM, SUBSTR, NVL 등의 함수는 벤더에서 미리 만들어둔 내장 함수이고, 사용자가 별도의 함수를 만들 수도 있다. Procedure와 다른 점은 RETURN을 사용해서 하나의 값을 반드시 되돌려줘야 한다는 것이다. 즉, Function은 Procedure와 달리 SQL 문장에서 특정 작업을 수행하고 **반드시 수행 결과 값을 리턴**한다.

예제로 ABS 함수와 같은 기능을 하는 사용자 정의 함수를 만들어 본다
- INPUT 값으로 숫자만 들어온다고 가정.

**[ Oracle ]**

```
CREATE OR REPLACE Function UDF_ABS (v_input in number)
return Number IS v_return number := 0;
BEGIN
if v_input < 0 then
  v_return := v_input * -1;
else
  v_return := v_input;
end if;
RETURN v_return;
END; /
```

**[ SQL Server ]**

```
CREATE Function dbo.UDF_ABS (@v_input int)
RETURNS int AS
DECLARE @v_return int
BEGIN
SET @v_return=0
IF @v_input <0
  SET @v_return = @v_input * -1
ELSE
  SET @v_return = @v_input
RETURN @v_return;
END
```

*** 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.

예제로 주문한 건이 입력될 때마다, 일자별 상품별로 판매수량과 판매금액을 집계하여 집계자료를 보관하도록 하는 트리거를 생성해 본다.

트리거의 역할은 ORDER_LIST에 주문정보가 입력되면, 주문정보의 주문일자와 주문상품을 기준으로 판매 집계 테이블(SALES_PER_DATE)에 해당 주문 일자의 상품 레코드가 존재하면 판매 수량과 판매 금액을 더하고 존재하지 않으면 새로운 레코드를 입력한다.

**[ Oracle ]**

```
CREATE OR REPLACE Trigger SUMMARY_SALES   → SUMMARY_SALES 트리거 선언
AFTER INSERT ON ORDER_LIST FOR EACH ROW   →  레코드가 입력되면 트리거 발생.
DECLARE                                                ORDER_LIST에 설정. 각 ROW마다 적용
o_date ORDER_LIST.order_date%TYPE;   →  주문일자, 주문상품 값을 저장할 변수 선언
o_prod ORDER_LIST.product&TYPE;
BEGIN
o_date := NEW.order_date;   →  변수에 신규로 입력된 데이터를 저장한다 (:NEW 구조체)
o_prod := NEW.product;
UPDATE SALES_PER_DATE   →  해당 주문일자의 상품레코드 존재하면 더해서 업데이트
SET qty = qty + :NEW.qty, amount = amount + :NEW.amount
WHERE sale_data = o_date AND product = o_prod;
if SQL%NOTFOUND then   →  존재하지 않으면 기존에 없는 실적이기 때문에 새로 입력
  INSERT INTO SALES_PER_DATE VALUES(o_date, o_prod, :NEW.qty, :NEW.amount);
end if;
END; /
```

# Trigger에서 사용하는 레코드 구조체

**:OLD** - **INSERT**=NULL, **UPDATE**=업데이트되기 전 레코드 값, **DELETE**=삭제되기 전 값
**:NEW** - **INSERT**=입력된 값, **UPDATE**=업데이트된 후의 값, **DELETE**=NULL

( **:OLD = deleted / :NEW = inserted** ) - [ Oracle = SQL Server ]

**[ SQL Server ]**

```
CREATE Trigger dbo,SUMMARY_SALES
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
UPDATE SALES_PER_DATE
SET qty = qty + @qty, amount = amount + @amount
WHERE sale_date = @o_date AND product = @o_prod;
IF @@ROWCOUNT=0
   INSERT INTO SALES_PER_DATE VALUES(@o_date, @o_prod, @qty, @amount)
END
```

**ROLLBACK**을 하면 하나의 트랜잭션이 취소가 되어 **Trigger로 입력된 정보**까지 하나의 트랜잭션으로 인식하여 **모두 취소**된다.

Trigger는 데이터베이스에 의해 자동 호출되지만, 결국 INSERT, UPDATE, DELETE 문과 하나의 트랜잭션 안에서 일어나는 일련의 작업들이라 할 수 있다.

Trigger는 데이터베이스 보안의 적용이 유효하지 않은 트랜잭션의 예방,
업무규칙 자동 적용 등에 사용될 수 있다.

*** 프로시저와 트리거의 차이점**

프로시저는 BEGIN ~ END 절 내에 COMMIT, ROLLBACK과 같은 **트랜잭션 종료 명령어**를 사용할 수 있지만, **트리거는 사용할 수 없다**.

## 2-3. SQL 최적화 기본 원리

### 2-3-1. 옵티마이저와 실행계획

*** 옵티마이저(Optimizer)**

사용자가 질의한 SQL문에 대해 **최적의 실행 방법을 결정**하는 역할 수행
이러한 최적의 실행 방법을 **실행계획**(Execution Plan)이라 함
어떤 방법으로 처리하는 것이 동일한 일을 최소의 일량으로 처리할 수 있을지 결정

옵티마이저가 최적의 실행 방법을 결정하는 방식

- **규칙기반 옵티마이저** (RBO, Rule Based Optimizer)
- **비용기반 옵티마이저** (CBO, Cost Based Optimizer)

*** 규칙기반 옵티마이저**

규칙(**우선순위**)를 가지고 실행계획을 생성
실행계획을 생성하는 규칙을 이해하면 누구나 실행계획을 비교적 쉽게 예측 가능

인덱스를 이용한 엑세스 방식이 전체 테이블 액세스 방식보다 우선 순위가 높음
→ 이용 가능한 인덱스가 존재하면 전체 테이블 엑세스 방식보다 항상 인덱스를 사용하는 실행계획을 생성

조인 순서를 결정 시 조인 칼럼 인덱스의 존재 유무가 중요한 판단의 기준

- 조인 칼럼에 대한 인덱스가 양쪽에 존재 : 우선순위가 높은 테이블이 선행(Driving)
- 한쪽에만 인덱스 존재 : 인덱스 없는 테이블이 선행 ( NL Join 사용 )
- 모두 인덱스 존재 X : FROM 절의 뒤에 나열된 테이블이 선행 ( Sort Merge Join 사용 )
- 우선순위가 동일 : FROM절에 나열된 테이블의 역순으로 선행 테이블 선택

*** 비용기반 옵티마이저**

비용(**예상되는 소요시간, 자원 사용량**)이 가장 적은 실행계획을 선택하는 방식
규칙기반 옵티마이저의 단점을 극복하기 위해서 출현

다양한 객체 통계정보와 시스템 통계정보 등 이용
→ 통계정보 없을 경우 정확한 비용 예측이 불가능해 비효율적 실행계획 생성

( 정확한 통계정보 유지하는 것은 비용기반 최적화에 중요한 요소 )

**질의 변환기** : 사용자가 작성한 SQL문을 처리하기에 보다 용이한 형태로 변환하는 모듈

**대안 계획 생성기** : 동일한 결과를 생성하는 다양한 대안 계획을 생성하는 모듈
- 대안 계획은 연산의 적용 순서 변경, 연산 방법 변경, 조인 순서 변경 등을 통해 생성
- 대안 계획의 생성이 많아지면 최적화를 수행하는 시간이 그만큼 오래 걸린다.

대부분의 상용 옵티마이저들은 대안 계획의 수를 제약하는 다양한 방법을 사용한다.
대안 계획들 중에서 최적의 대안 계획이 포함되지 않을 수도 있다.

**비용 예측기** : 생성된 대안 계획의 비용을 예측하는 모듈
- 연산의 중간 집합의 크기 및 결과 집합의 크기, 분포도 등의 예측이 정확해야 한다.
- 정확한 통계정보, 대안 계획을 구성하는 각 연산에 대한 비용 계산식이 정확해야 함.

인덱스를 사용하는 비용이 전체 테이블 스캔 비용보다 크다고 판단되면 전체 테이블 스캔을 수행하는 방법으로 실행계획을 생성할 수도 있다.

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 어렵다.

*** 실행계획**

SQL에서 요구한 사항을 처리하기 위한 절차와 방법
다양한 실행계획(처리방법)마다 성능(실행시간)은 서로 다를 수 있다.
( 옵티마이저는 최적의 실행계획 생성 )

**# 실행계획 구성 요소**

- **조인 순서** (Join Order)
- **조인 기법** (Join Method)
- **액세스 기법** (Access Method)
- **최적화 정보** (Optimization Information) : 실행계획의 각 단계마다 예상 비용 표시
  Cost : 상대적인 비용 정보
  Card : Cardinality의 약자. (결과집합의 건수)
  Bytes : 결과 집합이 차지하는 메모리의 양 (바이트)
  ( 비용정보는 실제 SQL 실행하고 얻은 결과 X, 통계 정보를 바탕으로 계산한 예상치 )
- **연산** (Operation) : 여러 조작을 통해 원하는 결과를 얻어내는 일련의 작업

*** SQL 처리 흐름도**

SQL의 내부적인 처리 절차를 시각적으로 표현한 도표 (실행계획을 시각화)
조인 순서, 액세스 기법과 조인 기법 등을 표현 가능

액세스 건수, 조인 시도 건수, 테이블 액세스 건수, 성공 건수, 스캔 방식 등 표현

## 2-3-2. 인덱스 기본

*** 인덱스의 특징과 종류**

인덱스는 원하는 데이터를 쉽게 찾을 수 있도록 돕는 책의 찾아보기와 유사한 개념
Insert, Update, Delete 등과 같은 DML 작업은 테이블과 인덱스를 함께 변경해야 하기 때문에 오히려 느려질 수 있다는 단점이 존재한다.

*** 트리 기반 인덱스**

DBMS에서 가장 일반적인 인덱스는 B-트리 인덱스이다.

**# B-트리 인덱스** : 브랜치 블록과 리프 블록

- 브랜치 블록 중에서 가장 상위에서 있는 블록을 루트 블록이라 한다.
- 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(Record Identifier/Rowid)로 구성된다.
- 인덱스 데이터는 인덱스를 구성하는 칼럼의 값으로 정렬된다.
  만약 인덱스 데이터 값이 동일하면, 레코드 식별자의 순서로 저장된다.
- 리프 블록은 양방향 링크(Double Link)를 가지고 있다.
  이것을 통해서 오름차순과 내림차순 검색을 쉽게 할 수 있다.
- B-트리 인덱스는 '='로 검색하는 일치(Exact Match) 검색과 'BETWEEN', '>'과 같은 연산자로 검색하는 범위(Range) 검색 모두에 적합한 구조이다.

  1단계 : 브랜치 블록의 가장 왼쪽 값이 찾고자 하는 값보다 작거나 같으면 왼쪽 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.

Oracle의 트리 기반 인덱스에는 B-트리 인덱스 외에 비트맵, 리버스키, 함수기반 등 존재

*** SQL Server의 클러스터형 인덱스**

SQL Server의 인덱스는 저장 구조에 따라 클러스터형 인덱스와 비클러스터형 인덱스로 나뉨

**# 클러스터형 인덱스**

- 인덱스의 리프 페이지가 곧 데이터 페이지다. 따라서 테이블 탐색에 필요한 레코드 식별자가 리프 페이지에 없다.
  (인덱스 키 칼럼과 나머지 칼럼을 리프에 같이 저장, 테이블 랜덤 액세스 할 필요가 X)
  클러스터형 인덱스의 리프 페이지를 탐색하면 해당 테이블의 모든 칼럼 값 바로 얻음.

- 리프 페이지의 모든 로우(=데이터)는 인덱스 키 칼럼 순으로 물리적으로 정렬되어 저장.
  테이블 로우는 물리적으로 한 가지 순서로만 정렬될 수 있음
  → 클러스터형 인덱스는 테이블당 한 개만 생성할 수 있음

*** 전체 테이블 스캔과 인덱스 스캔**

**# 전체 테이블 스캔**

전체 테이블 스캔 방식으로 데이터를 검색한다는 것은 테이블에 존재하는 모든 데이터를 읽어 가면서 조건에 맞으면 결과로서 추출하고 아니면 버리는 방식으로 검색한다.

Oracle 에서는 테이블의 **고수위 마크(HWM)** 아래의 모든 블록을 읽는다.
고수위 마크는 테이블에 데이터가 쓰여졌던 블록 상의 최상의 위치를 의미.
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.
결과를 찾기 위해 꼭 필요해서 모든 블록을 읽은 것이기 때문에 이렇게 읽은 블록들은 재사용성이 떨어진다. 그래서 전체 테이블 스캔 방식으로 읽은 블록들은 메모리에서 곧 제거될 수 있도록 관리된다.

**옵티마이저가 연산으로써 전체 테이블 스캔 방식을 선택하는 일반적인 이유**

- SQL문에 조건이 존재하지 않는 경우

- SQL문의 주어진 조건에 사용 가능한 인덱스가 존재하지 않는 경우, 또한 주어진 조건에 사용 가능한 인덱스는 존재하나, 함수를 사용하여 인덱스 칼럼을 변형한 경우에도 인덱스를 사용할 수 없다.

- 옵티마이저의 취사 선택 : 조건을 만족하는 데이터가 多, 대부분 블록 액세스 판단

- 그 밖에 병렬처리 방식으로 처리하는 경우 또는 풀테이블 스캔 힌트를 사용한 경우

**# 인덱스 스캔**

인덱스의 리프 블록은 인덱스를 구성하는 칼럼과 레코드 식별자로 구성되어 있다. 따라서 검색을 위해 인덱스의 리프 블록을 읽으면 인덱스 구성 칼럼의 값과 테이블의 레코드 식별자를 알 수 있다.

인덱스는 인덱스 구성 칼럼의 순서로 정렬되어 있다. 인덱스의 구성 칼럼이 A+B라면, 먼저 칼럼 A로 정렬되고 칼럼 A의 값이 동일하면 칼럼 B로 정렬된다. 만약 칼럼 B도 같은 경우 레코드 식별자로 정렬된다.

인덱스가 구성 칼럼으로 정렬되어 있기 때문에 인덱스를 경유하여 데이터를 읽으면 그 결과 또한 정렬되어 반환된다.

따라서 인덱스의 순서와 동일한 정렬 순서를 원하는 경우 정렬 작업이 따로 필요 없다.

**인덱스 유일 스캔** : 유일(Unique) 인덱스를 사용하여 단 하나의 데이터를 추출하는 방식
유일 인덱스는 중복을 허락하지 않는 인덱스.
유일 인덱스 구성 칼럼에 모두 '='로 값이 주어지면 결과는 최대 1건

**인덱스 범위 스캔** : 인덱스를 이용하여 한 건 이상의 데이터를 추출하는 방식
유일 인덱스의 구성 칼럼 모두에 대해 '='로 값이 주어지지 않은 경우,
비유일 인덱스를 이용하는 모든 액세스 방식은 인덱스 범위 스캔 방식.

**인덱스 역순 범위 스캔** : 리프 블록의 양방향 링크를 이용, 내림차순으로 데이터를 읽음
이 방식으로 최대값을 쉽게 찾을 수 있다. 인덱스 범위 스캔의 일종

기타 방식 : **인덱스 전체 스캔, 인덱스 고속 전체 스캔, 인덱스 스킵 스캔**

**# 전체 테이블 스캔과 인덱스 스캔 방식의 비교**

- 데이터를 액세스하는 방법은 크게 두 가지로 나뉨. 인덱스를 경유하는 인덱스 스캔과 테이블 전체 데이터 모두 읽으며 추출하는 전체 테이블 스캔 방식.

- 인덱스 스캔 방식은 사용 가능한 적절한 인덱스가 존재할 때만 이용 가능한 방식
전체 테이블 스캔 방식은 인덱스 존재 유무와 상관없이 항상 이용 가능한 방식

- 옵티마이저는 인덱스가 존재하더라도 전체 테이블 스캔 방식을 취사 선택 가능.

- **인덱스 스캔** : 레코드 식별자 이용, 정확한 위치 알고 읽음. 한번의 I/O요청에 한 블록씩

- **전체 테이블 스캔** : 비효율적, 여러 블록씩. But 테이블 대부분 데이터 찾을 땐 유리.

**2-3-3. 조인 수행 원리**

*** 조인이란?**

두 개 이상의 테이블을 하나의 집합으로 만드는 연산.
SQL문에서 FROM 절에 두 개 이상의 테이블이 나열될 경우 조인이 수행.
조인 연산은 두 테이블 사이에서 수행된다.

조인의 종류 : **NL Join, Sort Merge Join, Hash Join**

*** NL Join**

두 개의 테이블을 중첩된 **반복문처럼 조인을 수행**한다.
반복문 외부(처음 테이블)에 있는 테이블을 선행 테이블 또는 외부 테이블
반복문 내부(두번째 테이블)에 있는 테이블을 후행 테이블 또는 내부 테이블이라고 부름

조인을 반복문으로 표현

```
for 선행테이블 읽음                    ( 외부 테이블 )
    for 후행테이블 읽음                ( 내부 테이블 )
        (선행테이블과 후행테이블 조인)
```

**결과 행의 수가 적은 테이블을 선행 테이블**로 선택한다.
→ 선행 결과값이 많을 경우 일량이 늘어날 수 있기 때문

1. 선행 테이블에서 조건에 맞는 값을 찾는다
2. 선행 테이블의 조인 키를 가지고 후행 테이블 조인 키 확인
3. 후행 테이블의 인덱스에 선행 테이블의 조인 키 존재 확인
4. 인덱스에서 추출한 레코드 식별자를 이용하여 후행 테이블 액세스하여 버퍼에 저장.
5. 앞의 작업을 선행 테이블에서 만족하는 키 값이 없을 때까지 반복하여 수행.

*** Sort Merge Join**

**조인 칼럼을 기준으로 데이터를 정렬**하여 조인한다.

**넓은 범위의 데이터를 처리할 때 주로 이용**.
정렬 데이터가 많을 경우 성능이 떨어질 수 있다. (디스크 I/O로 인한 부하 발생)
비동등 조인에 대해서도 조인이 가능하다
인덱스를 사용하지 않아 **인덱스가 존재하지 않을 경우에 사용**할 수 있다.

1. 선행 테이블에서 조건에 맞는 행을 찾는다
2. 선행 테이블의 조인 키를 기준으로 정렬 작업을 수행한다
3. 1~2번 작업을 반복 수행하여 모든 행을 찾아 정렬한다.
4. 후행테이블에서도 같은 작업을 진행한다
5. 정렬된 결과를 이용하여 조인을 수행하고 결과 값을 추출 버퍼에 저장한다.

*** Hash Join**

**해싱 기법을 이용**하여 조인을 수행한다

NL Join의 랜덤 액세스 문제점과 Sort Merge Join의 정렬 작업의 부담을 해결하기 위한 대안으로 사용한다.

조인 칼럼의 **인덱스가 존재하지 않을 경우에도 사용**할 수 있다.

"="로 수행하는 **동등 조인만 가능**하다 (해쉬 함수를 이용하기 때문에)

해쉬 테이블의 크기가 메모리에 적재할 수 있는 크기보다 커지면 디스크 사용 입출력에 따른 부하가 가중된다. 즉, **결과 행의 수가 적은 테이블을 선행 테이블로 사용**한다.

1. 선행테이블에서 조건에 만족하는 행을 찾음
2. **선행테이블의 조인 키를 기준으로 해쉬 함수를 적용하여 해쉬 테이블 생성**
3. 1~2번 작업을 반복하여 선행 테이블의 모든 조건에 맞는 행을 찾아 해쉬 테이블 완성
4. 후행테이블에서 조건에 만족하는 행을 찾음
5. **후행테이블의 조인 키를 기준으로 해쉬 함수를 적용하여 해당 버킷을 찾음**
6. 같은 버킷에 해당되면 조인에 성공하여 추출버퍼에 저장
7. 후행 테이블의 조건만큼 반복 수행하여 완료