1과목 데이터 모델링의 이해 (요약본 1회독)

1-1. 데이터 모델링의 이해

1-1-1. 데이터 모델의 이해

* 모델링 특징 3가지 ★
 - **추**상화 : **일정한 형식**에 맞추어 표현
 - **단**순화 : **제한된 표기법, 언어**로 표현해서 **쉽게 이해하도록**
 - **명**확화(=**정**확화) : 누구나 **쉽게 이해하도록** 애매모호함을 제거해 정확하게 현상을 기술

* 정보시스템 구축에서 모델링 활용
 - 계획/ 분석/ 설계할 때 업무를 분석하고 설계하는데 이용
 - 구축/ 운영 단계에서는 변경과 관리의 목적으로 이용

* 모델링의 3가지 관점
 - 데이터 관점 (Data, what) : "업무"와 데이터 // 데이터와 데이터간의 관계
 - 프로세스 관점 (Process, How) : "업무 프로세스"가 실제 하는 일 or 무엇을 해야하는지
 - 상관 관점 (Interaction) : "업무"가 처리하는 일의 "방법"에 따른 "데이터가 받는 영향"

* 데이터 모델링 정의 ★
 - **정보시스템을 구축**하기 위한 데이터 관점의 **업무 분석 기법**
 - **현실 세계 데이터**를 -> **약속된 표기법**에 의해 표현
 - **DB 구축**을 위한 **분석/설계하는 과정**

* 데이터 모델링 기능
 - 시스템 가시화, 시스템 구조, 행동 명세화 시스템 구축의 구조화 틀 제공
 - 시스템 구축 과정을 문서화, 세부사항은 숨기는 다양한 관점 제공
 - 상세한 수준의 표현방법도 제공

* 데이터 모델링 필요한 이유 ★
 - 업무 정보를 **일정한 표기법**으로 표현 (=/=별도의 표기법 아님!)
 - 분석된 모델로 DB 생성 및 개발
 - **업무 흐름을 설명**

* 데이터 모델링의 중요성 => 간결 정확 신뢰
 - **파급효과 큼** : 일련의 변경 시 다른 것도 다 바꿔야
 - **복잡한걸 간결하게 표현** : 명확하고 간결하게 함
 - 데이터 **품질을 유지** : **오래된 데이터의 정확성과 신뢰성**

* 데이터 모델링의 유의점 (중-"비"유일) ★★★ => 중복없이 명확하게 분리
 - **중복** : 데이터베이스가 **여러 장소에 같은 정보 저장**되는 것을 주의
 - **비유연성★** : 데이터 정의를 **데이터 프로세스와 "분리★★"**해야함 **(분리해서 유연하게!!)**
 - **비일관성** : 데이터간 **"상호 연관관계★★**를 **명확★"**히 정의해야함 **(일관되고 명확하게!!)**

* 데이터 모델링 3단계
 - 개 : 추상화 수준이 높고, **업무중심적**, **전체에 대해 포괄적인 수준**의 모델링
 - 논 : { **키, 속성, 관계** }를 표현, **재사용성 높음**, **정규화를 수행**
 - 물 : 실제 DB에 이식, **물리적 성격**, **개념적보다 구체적**

* 프로젝트 생명주기
 - 폭포수 : 데이터 모델링이 명확히 구분됨 (분석 -> 개념, 논리 // **설계 -> 물리**)
 - 정보공학/구조적 방법론 : (분석 -> 논리) // (**설계 -> 물리**)
 - 나선형 : 업무 크기에 따라 논리, 물리적 설계가 분석, 설계 양쪽에서 수행

* 독립성 필요성
 - 유지보수 비용 절감
 - 중복된 데이터 줄이기
 - 데이터 복잡도 낮추기
 - 요구사항 대응을 높히고자

* **데이터베이스** 3단계 **구조** **(외개내)** ★
 - **외부 단계 / 개념적 단계 / 내부적 단계**

* 데이터 **독립성 요소** **(외개내)** ★
 - **외부 스키마 / 개념 스키마 / 내부 스키마**

* 두 영역의 데이터 독립성
 - **논리적 독립성 (외부<->개념)** / **물리적 독립성 (개념<->내부)**

외부단계	외부스키마 외부스키마 외부스키마	(=사용자와 가까운 단계)
	[논리적 데이터 독립성]	
개념적 단계	**개념스키마 ★(=DB에 저장되는 데이터와 사용자 관계 표현)**	
	모든 사용자 관점을 통합, 조직 전체 통합 <-┘	
	[물리적 데이터 독립성]	
내부단계	내부 스키마	

* **사상 (Mapping)** ★
 - **"상호 독립적인 두 개념을 연결시키는 다리" (= 매핑)**
 - **외부적/개념적 사상 + 개념적/"물리적" 사상**

* 데이터모델링 주요 세가지 개념
 - 업무가 관여하는 **어떤 것 (Things)**
 - 어떤 것이 가지는 **성격 (Attributes)**
 - 업무가 관여하는 어떤 것(Things) 간의 **관계 (Relationships)**

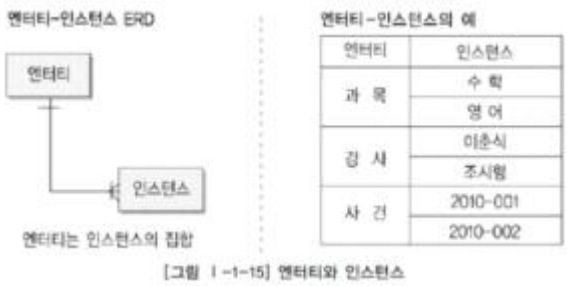
개념	복수/집합개념 & 타입/클래스	개별/단수개념 & 어커런스/인스턴트
어떤것(Thing)	엔터티타입(Entity Type)	엔터티(Entity)
	엔터티(Entity)	인스턴스(Instance) / 어커런스(Occurrence)
어떤것간의 연관(Relationships)	관계(Relationship)	페어링(Pairing)
어떤것의 성격(Attributes)	속성(Attribute)	속성값(Attribute Value)

* 데이터모델 표기법 ERD 이해
 - ERD 표기법 : 엔터티 = 사각형 // 관계 = 마름모 // 속성 = 타원형
 - ERD로 모델링하는 방법
   엔터티 그리기->엔터티 배치->엔터티간 관계 설정->관계의 참여도 설정-> 관계의 필수 여부 설정

* 좋은 데이터모델의 요소 ★
 - 완전성 / 중복배제 / 업무규칙 / 데이터 재사용 / 의사소통 / 통합성

1-1-2. 엔터티 *(이해 중요 : https://youtu.be/A2pXqyButgM)*

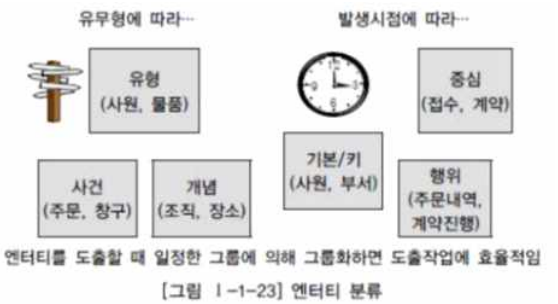
* **엔터티 ★**
 - 저장이 되기 위한 어떤 것, "실체, 객체"
 - **인스턴스 = 엔터티 안에 행 데이터** (ex. 환자 엔터티 -> 이춘식, 조시형의 데이터 인스턴스)



[그림 Ⅰ-1-15] 엔터티와 인스턴스

* **엔터티 특징 ★** (위 유튜브 영상을 생각하며 이해)
 - **"업무"에서 꼭 필요로 하는 정보 ★** (-> 책 9번문제와 같이, s병원만 있는 경우 엔터티 X)
 - **식별자에 의해 식별**이 가능해야 함 ★
 - **2개 이상의 인스턴스로 구성된 집합 ★**
 - 업무 프로세스에 의해 이용됨
 - **반드시 "속성"을 포함**해야 함 ★★
 - **다른 엔터티와 "관계"가 최소 1개 이상 존재 ★★**

* 엔터티 분류 (=> DB테이블에 들어가는 데이터 형태를 생각하면 이해됨)
 - **유형,무형**에 따른 : 유형 엔터티 / 개념 엔터티 / 사건 엔터티
 - **발생 시점**에 따른 : **기본 엔터티** / 중심 엔터티 / 행위 엔터티
   ㄴ>**기본 엔터티 : 다른 엔터티로 주식별자 상속X 자신 고유 주식별자 가짐**



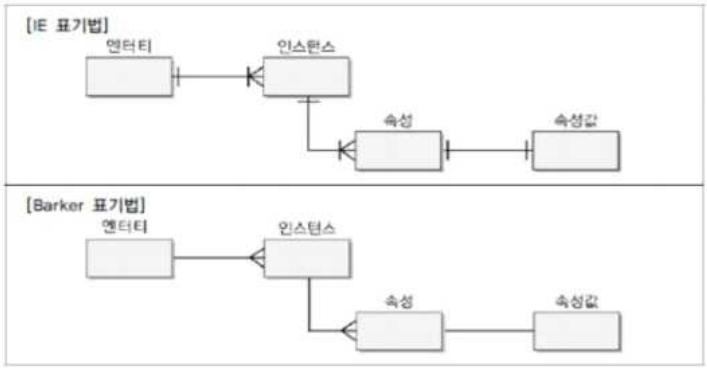
<== 이 그림을 외워야 (그림안에 예시들)★★
[그림 Ⅰ-1-23] 엔터티 분류

*** 엔터티 명명★★**
 - 약어 X
 - 현업에서 실제 사용하는 용어 사용
 - 단수명사 사용
 - 유일한 이름 부여
 - 생성 의미대로 이름 부여

1-1-3. 속성 *(이해: https://youtu.be/A2pXqyButgM)*

* 속성
 - 인스턴스로 관리하고자 하는 **의미상 더 이상 분리되지 않는 최소한의 단위**★★
 - **엔터티를 설명**하고, **인스턴스의 구성 요소**가 됨★★★★★★

* 엔터티<->속성<->인스턴스 관계 ★★★ 이해 중요 (유튜브 영상을 떠올리며 이해하는게 좋은 듯)
 - 1개의 엔터티 = 2개 이상의 인스턴스 집합 **(= 엔터티가 가장 큰 단위의 개념)**
 - 1개 인스턴스 = 2개 이상의 속성을 가짐
 - **1개 속성 = 1개의 속성값**을 가짐 **(하나이상 아니고 딱 하나!!)**



[그림 Ⅰ-1-25] 엔터티-속성의 관계

* 속성 분류
 - **속성의 특성에 따른 분류** { **기본속성, 설계속성, 파생속성** }★★★★★ (속성이라고 PK, FK이런거X)
  -> **기본** 속성 : 기본적인 모든 속성
  -> **설계** 속성 : **사용자에 의해 "새로 만들"**어지거나 정의되는 속성 ★★
  -> **파생** 속성 : 다른 속성의 "**영향을 받아**" 발생 ★★

[그림 Ⅰ-1-27] 속성의 분류

 - 엔터티 구성방식에 따른 분류
  -> PK : 엔터티 식별
  -> FK : 다른 엔터티와 관계에서 포함된 속성
  -> 일반 : 엔터티에 포함되어 있으며, PK도 FK도 아닌 속성

* 도메인
 - **각 속성(Attribute)이 가질 수 있는 범위**★★
 - { **데이터타입 / 크기 / 제약사항 - NOT NULL, Check 조건** }을 지정
  ㄴ> **테이블의 속성간 FK 제약조건 지정 X** 이런거 업음!!

* 속성 명명
 - 약어 X
 - 서술식의 속성명 사용 X
 - 현업에서 실제 사용하는 용어 사용
 - 유일한 이름 부여

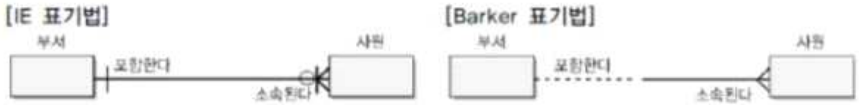
1-1-4. 관계 (이해: https://youtu.be/Pd2uJSvQVig)

* 관계 정의
 - 엔터티의 **인스턴스 사이 논리적 연관성**
 - 존재하는 형태나 행위로서 서로서로에게 연관성이 부여된 상태
 - **{ 관계명 / 차수(카디날리티) / 선택성(옵셔널리티) }** 로 구성 ★★

* 페어링
 - 엔터티 안에 인스턴스가 개별적으로 관계를 갖는 것
 - 관계 = 페어링의 집합

* 관계 분류 ★
 - **ERD** : **존재**에 의한 관계 / **행위**에 의한 관계 **=> 둘이 구분 없이 단일화된 표기법 사용** ★★★
 - **UML** : **연관** 관계 / **의존** 관계 **=> 실선과 점선 표기법으로 구분** ★★★

* 관계 표기법 (이해: https://youtu.be/eDjlQWzjTm0)
 - **관계명 // 관계차수(카디날리티) // 선택성(옵셔널리티)** 세 가지로 작성★



* 두 엔터티 사이에 정의한 관계를 체크하는 사항 ★★
 - **두 엔터티 사이 연관규칙** 존재?
 - 두 엔터티 **사이 정보의 조합**이 발생?
 - 업무기술서, 장표에 관계 **연결을 가능하게 하는 "동사(Verb)"**가 존재?
 - 업무기술서, 장표에 **관계 연결에 대한 "규칙"**이 존재?

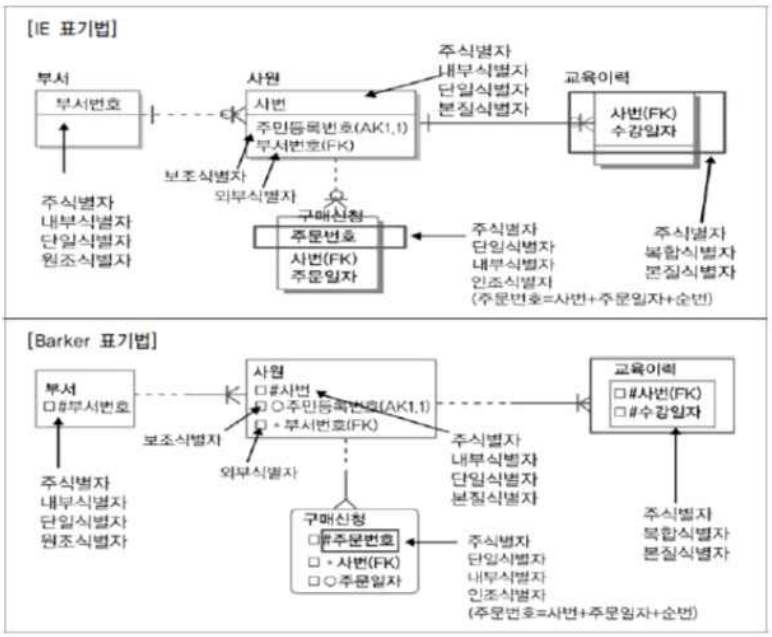
1-1-5. 식별자 (이해: https://youtu.be/A2pXqyButgM)

* 식별자 정의 (=키 정의)
 - **엔터티를 구분하는 논리적인 이름**
 - 즉, 엔터티를 **대표**할 수 있는 속성
 - **엔터티에는 반드시 하나의 유일한 식별자 존재**

* 식별자 특징 (유최불존)
 - 유일성 : 주 식별자에 의해 엔터티 내 모든 **인스턴스를 유일하게** 구분
 - 최소성 : 주 식별자를 구성하는 **속성 수는 유일성을 만족**하는 **최소의 수**여야 함
 - 불변성 : 주 식별자가 한 번 특정 엔터티에 지정되면, 그 **식별자 값은 변하지 X**
 - 존재성 : **주 식별자가 지정되면, 반드시 데이터값이 존재해야 함 (Null X)**

*** 식별자(키) 분류 ★★**
 - **대표성**을 가지는가 : 주 식별자(=PK) / 보조 식별자
 - **스스로 생성**될 수 있나: **내부 식별자 / 외부 식별자** ★★
 - **하나의 속성으로 식별**되나: **단일 식별자 / 복합 식별자**(=복합키) ★★
 - **본질 식별자 = 원래 있던 식별자**
   인조 식별자 = **기존에** 업무적으로 의미 있던 식별자 속성을 **대체해 새로 만든 식별자**★★

* 식별자 표기법



[그림 Ⅰ-1-42] 식별자의 분류-데이터 모델

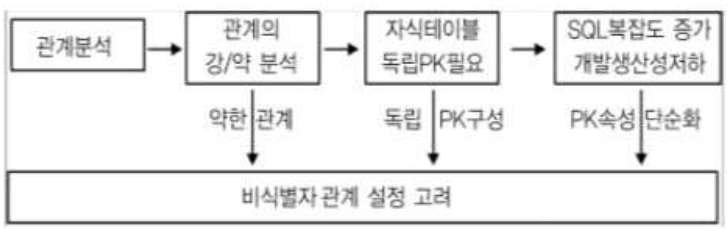
* 주 식별자 도출 기준 (= PK 도출 기준)
 - 해당 업무에서 자주 이용되는 속성으로 지정
 - 명칭, 내역처럼 이름으로 기술되는 것은 피함
   -> 구분자가 존재하지 않을 경우 새로운 식별자 생성
 - 복합키를 구성할 경우, 너무 많은 속성으로 구성하지 않도록 해야함

* 식별자와 비식별자 관계 ★★★★ (밑줄 내용을 이해)
 - **식별자 관계** : **자식이 부모의 기본키를 상속받아 기본키로 사용**할 경우임
   => Null 이면 안됨 => 반드시 부모 엔터티가 생성되어야 자신 엔터티 생성

   -> 문제점 : 자식의 주 식별자 속성이 지속적으로 증가할 수 있음 = 복잡해지고, 오류가능성 유발

 - **비식별자 관계** : **부모로의 속성**을 받았을 때, **자식의 기본키로 사용하지 않고 일반적인 속성으로만 사용**하는 경우

   -> 문제점 : 부모까지 조인해야되는 현상 발생 = SQL구문 길어져 성능 저하



[그림 Ⅰ-1-53] 비식별자관계 설정 고려사항

1-2. 데이터 모델과 성능

1-2-1. 성능 데이터 모델링의 개요

* 성능향상을 위한 데이터 모델링 수행 시점
  - **사전에 미리** 할수록 **비용 절감** 가능
  - **분석/설계 단계**에서 하는 것이 Best

* **성능 고려한 데이터 모델링 순서 ★★★**
  1- **정규화**
  2- **DB 용량 산정**
  3- **트랜잭션 유형** 파악
  4- DB 용량(2번), 트랜잭션 유형(3번)에 따라 **반정규화**
  5- **PK/FK "조정"**, **슈퍼타입/서브타입 "조정"**
  6- 성능관점에서 **데이터모델 "검증"**

* 성능 데이터모델링 고려사항 ★★★★ (문제나옴)
  - **정규화** -> 중복 제거를 통해 삽입/수정/삭제 성능 향상 (**조회 성능을 저하시키는건 아님!!!**)
  - **용량산정** -> **전체적인 DB의 트랜잭션 유형과 양을 분석하는 자료**가 됨
  - **물리적 데이터 모델링** -> **PK/FK 칼럼 순서 조정**, **FK 인덱스 생성** 수행 -> 성능향상
  - **이력데이터** -> 시간에 따라 **반복적으로 발생** -> **대량 데이터일 수 있다** -> **칼럼 추가하도록 설계**

1-2-2. 정규화와 성능 (정규화 이해 : https://mangkyu.tistory.com/110)

*** 정규화**
  - 데이터 모델을 좀 더 구조화하고 개선시키는 절차
  - **중복 제거**, **무결성을 지킴** ★★
  - 성능은 **" 조회 "** // **" 삽입, 수정, 삭제 "**의 두 가지 측면 **둘 다 고려**해야함
  - **정규화**가 잘 되어있으면, **"삽입,수정,삭제" 성능 향상(잘 쪼개서)** (**조회 성능을 저하시키는건 아님!!!**)
    반정규화가 잘되어 있으면, **"조회" 성능 향상(중복 만들어서->조인할필요X)**

  - 결정자에 의해 함수적 종속성이 있는 일반속성을 의존자로 하여 입력/수정/삭제 이상 제거
  - 중복속성 제거, 결정자에 의한 동일한 일반속성을 하나의 테이블로 합체
  - 한 테이블의 데이터 용량을 최소화

* 함수적 종속성에 근거한 정규화 수행 필요
  - 함수의 종속성(Dependency)는 데이터들이 어떤 기준값에 의해 종속되는 현상을 지칭
  - **결정자 ex) 주민등록번호 <-> 종속자 ex) 이름, 출생지, 주소**

1-2-3. 반정규화와 성능

*** 반정규화**

- **중복 생성!!! ★★**
- 정규화된 엔터티,속성,관계에 대해 성능향상, 단순화를 수행하기 위해
  중복, 통합, 분해 등을 수행
- **무결성이 깨질 수도 있**지만 ★★
  -> **Disk I/O를 감소**시키고, **긴 조인 쿼리문으로 인한 성능 저하 해결**함
  -> 중복성의 원리를 활용해 **데이터 조회 시 성능 향상** ★★

- 정규화도 일부 조회 성능을 향상시키지만
  -> **일부 여러 개의 조인이 필요할 때,**
  조회에 대한 처리 성능이 확실히 중요하다고 판단되면 부분적으로 반정규화

  -> **정규화의 종속 관계는 위반하지 않으면서**
  데이터의 중복성을 증가시켜 **조회 성능을 향상시킴**

* 반정규화 적용 방법

- 반정규화 대상 조사 : 범위 처리 빈도수, 대량 범위처리, 통계성 프로세스, 테이블 조인 수
  ↓
- 다른 방법 유도할 수 있는지 검토 : View테이블, Clustering, Index 조정, 응용애플리케이션
  ↓
- 반정규화 적용 : 테이블, 애트리뷰트, 관계를 반정규화

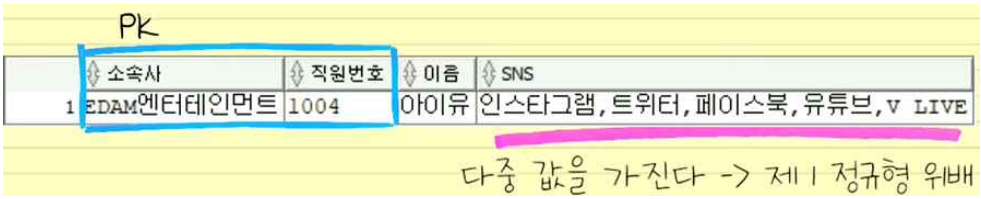
[정규화 - 중복 제거를 위해 쪼개는 것임을 인지하자 (무결성 O)] (링크 : https://youtu.be/pMcv0Zhh3J0)

* 1차 정규화

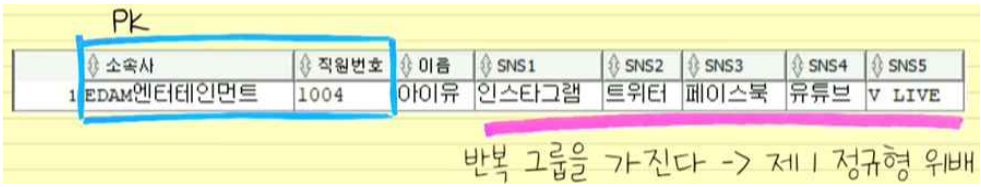
-> 중복은 데이터가 중복될 수도 있고

-> 데이터도! 암튼 **뭔가 중복된게 많아보이면 => 1차 정규화 대상!!**
중복된게 없다, 안보인다 => 1차 정규화 완료 => 2차 정규화 대상!!

1.

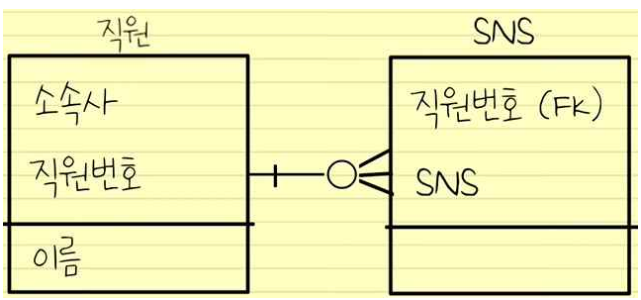


2.



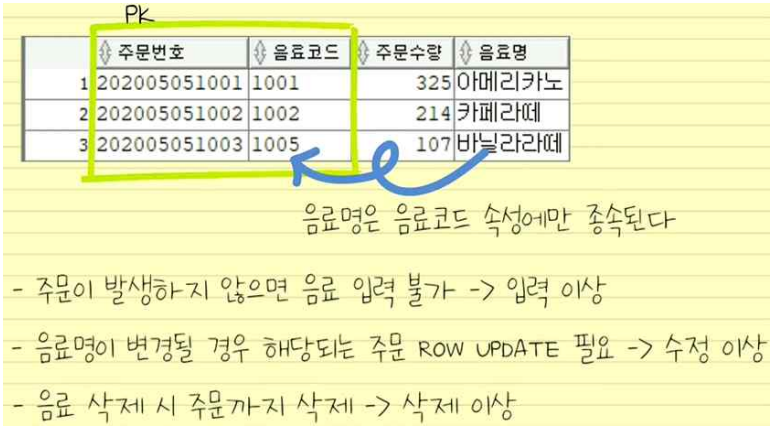
=> 반복그룹을 제거하기 위해 테이블을 쪼개자

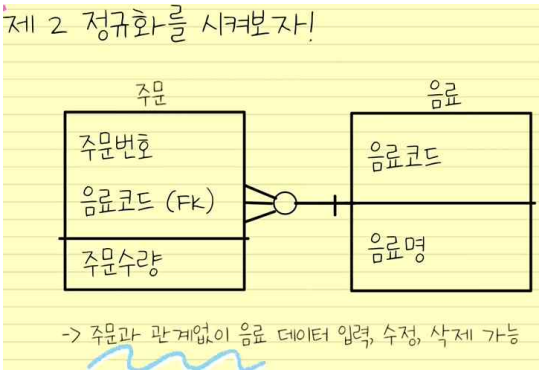
3.



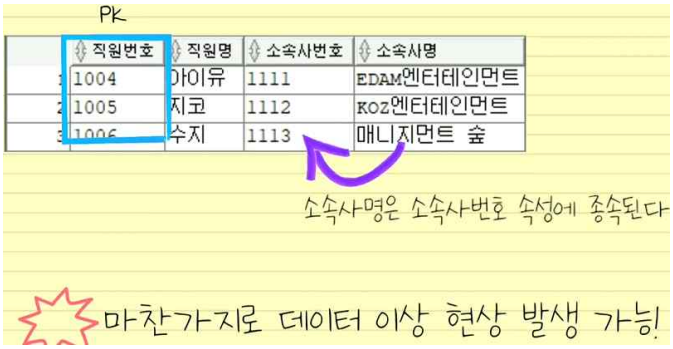
=> 최종 결과

* 2차 정규화 (주문 테이블 예)

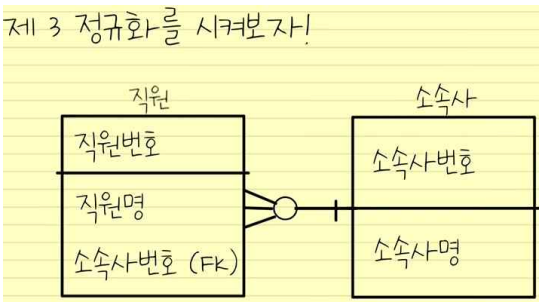




* 3차 정규화



(-> PK는 직원번호인데)



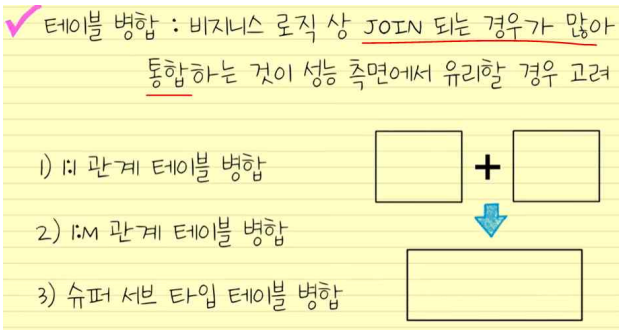
* 4차 정규화

1-2-3-1. **테이블 반정규화** **(=> 무결성 깨트릴 가능성)** (링크 : https://youtu.be/SS6H2whbfwc)

[반정규화 - 중복 추가하여 select-조인 성능 높이는 것을 인지 (무결성 X)]

* **테이블 병합**

- **1:1 관계 병합** // **1:M 관계 병합** // **슈퍼, 서브타입 병합**

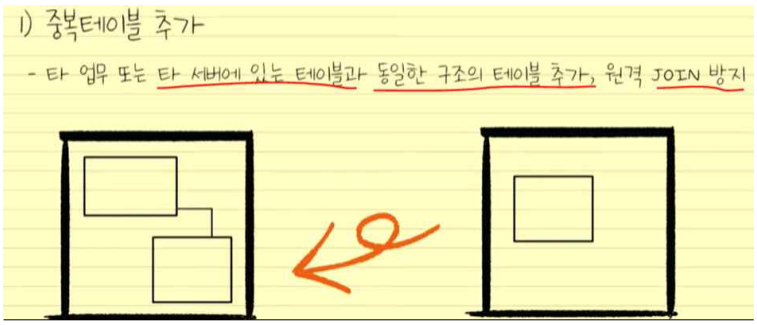


* **테이블 분할** **분할도 반정규화다 -> 테이블 큰 경우 처리범위 줄이기 위해 수행** ★★★

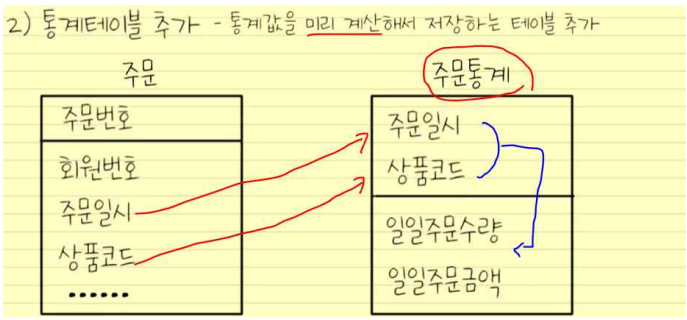
- **수직** 분할 (컬럼단위 or 테이블을 1:1로 분리)
- **수평** 분할 (행 단위)

* **테이블 추가**

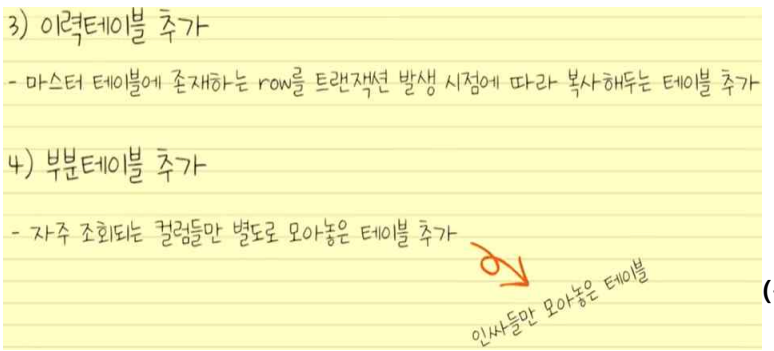
- **중복**테이블 추가: 동일한 테이블 구조를 추가하여 **중복 원격조인 "제거"** **(<->중복 컬럼 추가)**



- **통계**테이블 추가: SUM, AVG 등을 **미리 계산** (조회성능향상됨)



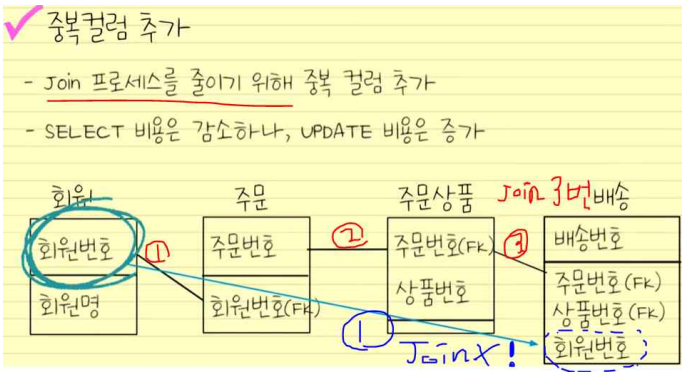
- **이력**테이블 추가: 마스터 테이블 **레코드(=이력)** 중복하여 이력테이블에 존재
- **부분테이블 추가**: 전체 컬럼 중 **자주 이용하는 인싸 "컬럼"만 모아 별도 테이블 형성**★★★



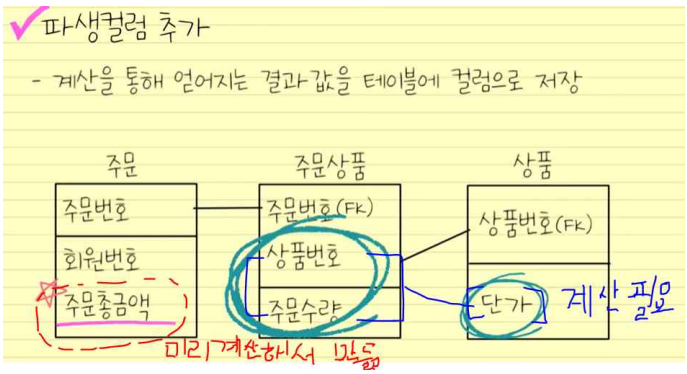
(<-->부분컬럼 추가는 없다!

1-2-3-2. **컬럼 반정규화 (=> 무결성 깨트릴 가능성)** (이해 링크 : https://opentutorials.org/module/4134/25352)

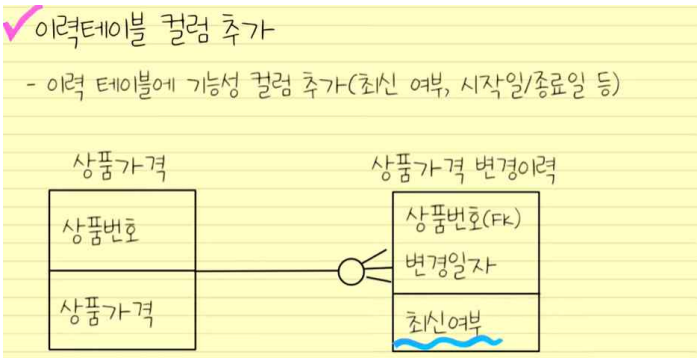
* **중복**컬럼 추가 : **조인을 감소시키기 위해** 중복된 칼럼을 추가 (=조인할 필요 줄어듦)



* **파생**컬럼 추가 : SUM처럼 계산하는 것들은 **미리 계산**하여 칼럼에 보관 (<->통계테이블 추가)



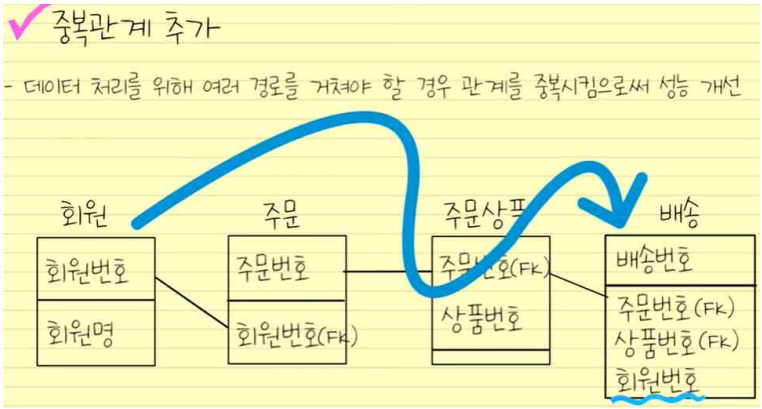
* **이력테이블** 칼럼 추가 : 기능성 칼럼 추가 (시작 및 종료일자 등)



* **기본키**에 의한 컬럼 추가 : **복합의 의미**를 갖는 PK 단일 구성 시 발생->**일반속성으로 PK 추가**
* **응용시스템 오작동**을 위한 컬럼 추가 : **이전 데이터를 임시적으로 중복 보관**

1-2-3-3. **관계 반정규화 (=> 무결성 깨트릴 가능성 X)**

* **중복**관계를 추가 : **여러 경로를 거친 조인을 방지**하기 위해 **추가적인 관계**를 맺음



1-2-4. 대량 데이터에 따른 성능

* 성능 저하 원인
  - A 한 테이블에 데이터 대량 집중 : 테이블 너무 커짐 -> 효율성 ↓ 디스크 I/O ↑
  - A 한 테이블에 여러 컬럼 존재 : 데스크 점유량 ↑ -> 디스크 읽기 I/O ↑
  - B 대량 데이터 처리되는 테이블 : SQL문에서 데이터 처리를 위한 I/O ↑ , 인덱스 구성
  - B 대량 데이터가 한 테이블에 존재: 인덱스 크기 ↑ 로 인한 성능 저하
  - 컬럼이 많아질 경우 : 로우체이닝, 로우마이그레이션 발생

* 해결 방안
  - **A 한 테이블에 많은 컬럼 => 수직분할**
    수직분할 : 컬럼 단위로 분할하여 I/O를 줄임
  - **B 대량 데이터 저장 문제 => 파티셔닝 or PK에 의한 테이블을 분할**
    수평분할 : 로우 단위로 분할하여 I/O를 줄임

* 로우 체이닝과 로우 마이그레이션
  - **로우 체이닝** : **행 길이가 길어서**, 데이터블록 하나에 데이터를 모두 저장하지 않고,
    두 개 이상의 블록에 걸쳐서 하나의 로우를 저장하는 형태 **(한 로우를 여러 체인처럼)**
    => **1:1로 테이블을 분리하면** => **디스크 I/O가 줄어** => **조회 성능 향상**★★★
  - **로우 마이그레이션** : **데이터블록에서 수정 발생 시**, 수정된 데이터를 해당 블록에 저장하지 못하고
    다른 블록에 빈 공간을 찾아 저장하는 방식
  - **파티셔닝** : **한 테이블에 많은 데이터가 저장되면**, 테이블 아무리 쪼개도 성능이 저하되기 때문에
    ★★ **논리적으로는 하나의 테이블**이지만, **물리적으로는 여러 개의 테이블로 분리**된 형태

* 수직/수평 분할 절차
  - **데이터 모델**을 먼저 완성
  - **DB 용량** 산정
  - **대량 데이터**가 처리되는 테이블에 대해 **트랜잭션 처리 패턴 분석**
  - **컬럼 or 로우 단위로 집중화된 처리**가 발생하는지 분석해 **테이블 분리**

1-2-5. 데이터베이스 구조와 성능 **(이해 링크 :** https://velog.io/@minddi/DB%EB%AA%A8%EB%8D%B8%EB%A7%81-%EA%B4%80%EA%B3%84%EC%8A%B8%ED%8D%BC-%EC%84%9C%EB%B8%8C%ED%83%80%EC%9E%85 **)**

*** 슈퍼/서브타입 데이터 모델**
  - **논리적 데이터 모델에서 주로** 이용(=분석 단계에서 많이 쓰임)
  - 물리적 데이터 모델로 설계 시 문제 발생!
  - **슈퍼타입** : **공통 부분을 슈퍼타입으로** 모델링
  - **서브타입** : 공통으로부터 **상속받아 다른 엔터티와 차이가 있는 속성만** 모델링

*** 데이터베이스 성능 저하 원인 3가지** ★★★ 문제나옴
  - **트랜잭션 : 전체를 일괄처리**, **테이블 : 개별로 유지 => Union 연산에 의해 성능 저하**
  - **트랜잭션 : 슈퍼+서브타입 공통 처리**, **테이블은 개별로 유지 => 조인에 의해 성능 저하**
  - **트랜잭션 : 서브타입만 개별로 처리**, **테이블은 하나로 통합 => 불필요하게 많은 데이터 집적**

* 슈퍼/서브 타입의 변환 기준
  - **데이터가 소량**일 경우 : 데이터 처리 **유연성 고려해 1:1 관계를 유지**
  - **데이터가 대량**일 경우 : 3가지 방법 -> **전부 개별 테이블**로 구성
    -> **슈퍼+서브타입 테이블**로 구성
    -> **하나의 테이블**로 구성

* 슈퍼/서브 타입의 데이터 모델 변환 기술
 - **개별로 발생**되는 트랜잭션 => **개별 테이블**로 구성 (One to One type)
 - **슈퍼+서브타입**으로 발생되는 트랜잭션 => **슈퍼+서브타입 테이블**로 구성 (Plus type)
 - **전체를 하나로 묶어** 발생하는 트랜잭션 => **하나의 테이블**로 구성 (Single type)

 - 쪼개질수록, { **확장성↑ // Disk I/O 성능↑ // 조인 성능↓, // 관리 용이성↓** }

* PK / FK 컬럼 순서 및 성능 ★
 - 일반적인 프로젝트에선 PK/FK 컬럼 순서의 중요성을 인지하지 못해서
   데이터 모델링된 그 상태대로 DDL을 생성하여 성능이 저하되는 경우가 빈번함!
 - **인덱스** 중요성 : **데이터 조작 시** 가장 효과적으로 처리될 수 있는 **접근 경로 제공** 오브젝트
   앞쪽에 위치한 속성값(컬럼)이 비교자로 있어야 인덱스가 좋은 효율을 냄
 - PK / FK **설계** 중요성 : **데이터 접근 시 접근경로** 제공, **설계단계 마지막에 컬럼 순서**를 조정
 - **PK 순서** 중요성 : **물리적 모델링 단계**에서 스스로 생성된 PK 외에 **상속되는 PK 순서도 중요**
 - **FK 순서** 중요성 : **조인을 할 수 있는 수단이 됨(=경로)** , 조회 조건 고려해서 반드시 인덱스 생성

* 인덱스 액세스 범위 좁히는 가장 좋은 방법 (49번)
 - PK가 여러 개일 때, Where절에서 사용하는 조건용 컬럼들이 우선순위가 되어야 함
 - **'=' EQUAL조건 (동등조건)에 있는 컬럼이 제일 앞으로**
 - **BETWEEN, IN (범위조건)에 있는 컬럼이 그 다음순위**
 - 나머지 PK는 그 뒤에 아무렇게나. (id값 이런거 상관없음)

* PK 순서를 조정하지 않으면 성능 저하되는 이유
 - **조회 조건(WHERE)에 따라** 인덱스를 처리하는 범위가 달라짐
 - **PK의 순서**를 인덱스 특징에 맞게 생성하지 않고 **자동으로 생성하면**,
   테이블에 접근하는 트랜잭션이 **인덱스 범위를 넓게 하거나 Full Scan**을 유발

* **물리적 테이블에 FK 제약이 걸려있지 않은 경우**, **인덱스 미생성**으로 생긴 성능 저하
 - 물리적으로 두 테이블 사이 FK 참조 무결성 관계를 걸어 상속받은 FK에 인덱스 생성

1-2-6. 분산 데이터베이스와 성능 (영상 : https://youtu.be/sznXWIS6Ej8)
* 분산 데이터베이스란
 - **빠른 네트워크 환경**을 이용해 DB를 여러지역, 여러노드로 위치시켜 **사용성, 성능을 극대화**시킨 DB
 - **분산된 DB**를 **하나의 가상 시스템**으로 사용할 수 있도록 한 DB
 - "**논리적"으로 동일한 하나의 시스템**이며, **네트워크를 통해** "**물리적"으로 분산**된 데이터들의 모임
 - 논리적으로 사용자 통합 및 공유 물리적 Site 분산

* 분산 DB의 투명성 6가지 (설명 영상 : https://youtu.be/sznXWIS6Ej8)
 - **분할** 투명성(단편화) : **하나**의 논리적 릴레이션을 **여러 단편으로 분할**, 그 **사본을 여러 Site에 저장**
 - **위치** 투명성 : 사용할 데이터의 **저장 장소를 알 필요가 없다**. (위치정보 시스템 카탈로그에 유지)
 - **중복** 투명성 : DB 객체가 여러 Site에 **중복 저장되었는지 알 필요**가 없다.
 - **장애** 투명성 : 구성요소의 **장애에 무관하게 트랜잭션 원자성 유지**
 - **병행** 투명성 : 다수 트랜잭션 **동시 수행 시 결과 일관성 유지**
 - **지역사상** 투명성 : **지역DBMS와 물리DB 간에 Mapping보장** (각 지역 이름과 **무관한 이름 사용O**)

* 분산 데이터베이스 적용 방법
 - 단순히 **분산 환경에서 DB를 구축하는게 목적이 아님!!!**
 - **업무의 특징에 따라** DB 분산구조를 **"선택적"으로 설계**

* 분산 데이터베이스 활용 방향성
 - 업무적 특징에 따라 **위치 중심 또는 업무 필요에 의한 분산 설계**
* 분산 데이터베이스 장단점

- 장점	- 단점
지역자치성, 점증적 시스템 용량 확장	개발비용 많이 든다
신뢰성, 가용성	잠재적 오류 증대
효용성, 융통성	처리 비용의 증대
빠른 응답 속도를 통한 통신비용 절감	설계와 관리 복잡
시스템 규모의 적절한 조절	불규칙한 응답 속도
각 지역 사용자 요구 수용 증대	데이터 무결성에 대한 위협

* 데이터베이스를 분산 구성했을 때의 가치
 - 통합된 DB에서 제공할 수 없는 **빠른 데이터 처리 성능**

1-2-7. 분산 데이터베이스 적용 기법
*** 테이블 위치 분산 (물리적인 분산인 듯)**
 - 테이블 **구조는 변경 X**
 - 테이블이 **다른 DB에 중복으로 생성되지도 X**
 - 정보를 이용하는 형태가 각 위치별로 차이가 있을 경우에만 사용(이때 위치=서버 컴퓨터)
 - **테이블 위치를 파악할 수 있는 도식화된 위치별 DB 문서 필요**

*** 테이블 분할 분산 - 수평분할**
 - 특정 **칼럼 값 기준**으로 **"행"단위**로 분리 **(열, 컬럼은 분리X)**
 - Primary Key에 의해 중복 발생 X
 - **데이터 수정** : **타 지사**에 있는 데이터를 **수정 X**, **자사의 데이터만 수정 O**
 - **각 지사 테이블 통합 처리** : 조인이 발생해 성능저하 예상됨
 통합 처리 프로세스가 많은지 검토 후 적으면 수평분할 ㄱ
 - **데이터 무결성 보장** : 데이터가 지사별로 별도로 존재하여 중복 발생 X
 - **지사별 DB를 운영하는 경우** : 어떤 경우든 간에 DB 테이블들은 수평 분할하여 존재함

*** 테이블 분할 분산 - 수직분할**
 - **컬럼 단위**로 분리 **(행 분리 X)**
 - 각 테이블은 동일한 기본키와 값을 가져야함
 - 데이터 중복이 발생 X

*** 테이블 복제 분산 - 부분복제**

- **마스터 DB에서** 테이블의 **일부 내용만 다른 서버에 위치**시킴
- **통합된 테이블은 본사에 저장**하고, **지사별로 각 지사에 해당하는 로우를** 가지는 형태
- **지사에 데이터 선 발생 후 본사는 지사 데이터 통합**! (<--> 광역복제)
- **여러 테이블을 조인하지 않고!** 빠른 작업 수행 가능!
  -> **각 지사별 처리도 가능, 전체 본사 통합 처리도 가능**
- 본사 데이터는 통계, 이동 등 수행 // 지사 데이터로는 지사별 빠른 업무!
- 다른 지역간 데이터 복제는 실시간 처리보다 배치 처리를 이용
- 데이터의 정합성 일치 어렵다

*** 테이블 복제 분산 - 광역복제**

- **통합된 테이블은 본사에 저장**하고, **각 지사에 동일한 데이터를** 저장!
- **본사에서 데이터 입력,수정,삭제** => **이를 지사에서 이용**! (<--> 부분복제)
- **본사 <--> 지사 간 특별한 제약 X**
- 다른 지역간 데이터 복제는 실시간 처리보다 배치처리를 이용

*** 테이블 요약 분산 - 분석요약**

- **각 지사별 존재하는 요약정보를 본사에 통합** 후 **전체에 대해 "다시 요약"**
- 동일한 테이블 구조를 가지고 분산된 동일 내용의 데이터를 이용해 통합된 데이터 산출
- **본사와 지사가 동일한 테이블을 갖지만**, **지사**는 동일 내용에 대해 **지사별 요약**정보 가짐
  본사는 지사의 요약정보 **통합해 "재산출"한 요약**정보 가짐

*** 테이블 요약 분산 - 통합요약**

- **각 지사별 존재하는 "다른 내용 정보"를 본사에 통합** 후 **전체에 대해 "다시 요약"**
- **각 지사**는 **타 지사와 다른 요약**정보를 가짐
  본사는 지사의 요약정보를 **통합해 "재산출"한 요약**정보 가짐

* 분산 데이터베이스를 적용하여 성능이 향상된 사례

- 분산 환경의 원리를 이해하지 않고 DB를 설계해 성능이 저하되는 경우가 빈번함
- 복제분산의 원리를 간단히 응용하면 성능향상해 설계가 가능하다~

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
 - DBMS : 데이터베이스 관리 소프트웨어

* 데이터베이스 발전
 - 플로우차트 -> 계층형/망형 -> 관계형 -> 객체관계형

* 관계형 데이터베이스
 - **파일 시스템 단점 : 동시에 삽입/수정/삭제가 불가능**하여 데이터 관리가 어렵다
  복사본 파일을 만들어 사용할 경우, 데이터의 불일치성이 발생한다
 - RDB의 장점 : 정규화를 통해 이상현상과 중복을 제거
  데이터 무결성 보장, 데이터 회복/복구 가능
  병행 제어, 동시성 관리를 통해 데이터를 공유
  데이터 표현 방법 등을 체계화 => 데이터 표준화, 품질을 확보

* SQL (Structured Query Language) ★★

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

* 테이블
 - 데이터를 저장하는 객체로서, **RDB의 기본 단위**
 - 테이블은 하나 이상의 칼럼을 가져야하며, 모든 데이터를 칼럼과 행의 2차원 구조로 나타냄

* 테이블의 분할
 - 데이터의 불필요한 중복을 줄이는 것이 정규화 -> 이상현상(Anomaly)를 방지함

*** DCL : 데이터 제어어 (<--> TCL 주의) ★★**
 - GRANT, REVOKE

* ERD (Entitiy Relationship Diagram)
 - **관계의 의미**를 직관적으로 표현할 수 있는 수단
 - 구성요소 : **엔티티(E), 관계(R), 속성**
 - 표기법 종류 : IE 표기법(Information Engineering) , Barker 표기법(Case Method)

2-1-2. DDL (Data Definition Language) (쏭즈 : https://youtu.be/Gx9dWxLXduA)

*** DDL**
 - 스키마, 도메인, 테이블, 뷰, 인덱스를 **정의/변경/제거**할 때 사용 ★★
 - **auto commit돼서 얘네 수행되면 ROLLBACK 있어도 데이터 COMMIT 된다** ★★★★★★★★

*** DDL - Create Table ★★★**

CREATE TABLE

```
CREAT TABLE 테이블 명 ( 칼럼1 칼럼 유형 <CONSTRAINT> ,
                       칼럼2 칼럼 유형 <CONSTRAINT> ,
                       칼럼3 칼럼 유형 <CONSTRAINT> ……);
```

CREATE TABLE AS (CTAS)

CRETAE TABLE AS문장을 사용하는 경우 테이블의 구조를 복사하므로 따로 작성할 필요가 없다.
NOT NULL 제약 조건만 복사가 된다. 다른 제약조건들은 복사가 되지 않아서 다시 적용해야 한다.
PK , FK, UNIQUE , CHECK 이런 것들 복사 X

```
CREAT TABLE 테이블 명 AS
SELECT 칼럼1, 칼럼2 ...
FROM 테이블 명
<WHER 이하 조건 절> ;
```

*** DDL - Select문 활용해서 테이블 복사 생성하기**
 - [Oracle] Create Table 테이블명B **AS Select *** From 테이블명A;
 - [SQL Server] **Select * Into** 테이블명B From 테이블명A;

CTAS 응용 : 테이블 구조만 복사하여 생성

WHERE 에 FALSE 조건을 입력하여 만든 CTAS 문이다. 테이블의 구조만 복사하고 로우는 복사하지 않는 방식이다. 테이블의 구조만 복사하고 싶은 경우 사용한다. INSERT INTO 를 할 때 편리하기 때문에

```
CREATE TABLE EMP3 AS
SELECT *
FROM EMP
WHERE 1 = 2 ;

SELECT EMP3;
```

*** 제약조건 (CONSTRAINT) ★★** (Alter Table에서도 설정 가능!)

- 데이터의 무결성을 유지하기 위한 방법, 사용자가 원하는 조건의 데이터만 유지하기 위한 방법
- **Primary Key** : 기본키, **한 테이블에 하나만 지정 가능★★** => 자동으로 UNIQUE한 인덱스 생성!
  NULL값 입력 불가 (=> 기본키 제약조건)
- **UNIQUE : NULL 가능★★★**, 행을 고유하게 식별하기 위한 고유 키**(=중복X)**
- **NOT NULL** : **명시적으로 NULL 입력 방지**
- **CHECK★**: **데이터 무결성을 유지**하기 위해 테이블의 **특정 컬럼에 설정**하는 제약
- **Foreign Key** : 외래키, 참조 무결성 옵션 선택 가능, **여러개 가능★★**

#. PK제약조건 생성하는 DDL★★★★★★★★★★

- CREATE문 안에 (두 가지 방법)
- 1번)

```
CREATE TABLE EMP
( EMP_NO VARCHAR2(10)  PRIMARY KEY
 ,EMP_NM   VARCHAR2(300)  NOT NULL
```

- 2번)

```
CREATE TABLE PRODUCT
( PROD_ID VARCHAR2(10) NOT NULL
 ,PROD_NM VARCHAR2(100) NOT NULL
 ,REG_DT DATE  NOT NULL
 ,REGR_NO NUMBER(10)
 ,CONSTRAINT PROCUT_PK  PRIMARY KEY
  (PROD_ID));
```

이름 혹 우는키인지

- CREATE문 밖에 (ALTER TABLE)

```
ALTER TABLE
    table_name
ADD
CONSTRAINT
  constraint_name
  PRIMARY KEY
  (col_1, col_2, ... )
```

이름 혹 우는키인지

*** NULL ★★★★**

- "알수없는값", **0도 아니고, 공백(' ')도 아님**
- NULL은 **"IS NULL", "IS NOT NULL"로만 비교 가능!!** <>NULL, !=NULL 이런거 안됨!!
- **[SQL Server]에서 a=''로 넣으면 -> NULL아니고, 공백값이 들어감!! ~> IS NULL 시 False 반환★**

*** 외래키(FK) 참고사항 ★★**

- 테이블 생성 시 설정할 수 있다. (Constraint play_FK Foreign Key (ID) References ~~~;)
- 외래키는 **NULL값 가질 수 있다!!!** ★
- 한 테이블에 **여러 개 존재** 가능 ★
- **참조 무결성 제약을 받는다** ★

*** DDL - Alter Table - Oracle ★**

- 컬럼 추가 : **Alter Table** 테이블명 **ADD** (추가칼럼명 Dtype**);**
- 컬럼 삭제 : **Alter Table** 테이블명 **DROP COLUMN** 삭제컬럼명**;**
- 컬럼 속성 변경 : Alter Table 테이블명 MODIFY (컬럼명 Dtype);

2. 테이블의 구조 변경

ALTER TABLE

ALTER TABLE	ADD COLUMN	칼럼 추가
ALTER TABLE	DROP COLUMN	칼럼 제거
ALTER TABLE	MODIFY COLUMN	칼럼 수정
ALTER TABLE	RENAME 기존 칼럼 명 TO 새로운 칼럼 명	칼럼 명 변경
ALTER TABLE	DROP CONSTRAINT	제약조건 삭제
ALTER TABLE	ADD CONSTRAINT	제약조건 추가

*** Alter Table + ADD COLUMN : 컬럼"을" 추가**

ALTER TABLE + ADD *COLUMN(S)*

기존 테이블에 새롭게 칼럼을 추가하고 싶은 경우 ALTER TABLE + ADD *COLUMNS* 을 사용하여 추가한다.

```
ALTER TABLE 테이블 명
  ADD ( 칼럼 이름  칼럼의 유형  제약조건 ,
        칼럼 이름2  칼럼의 유형  제약조건 .
        칼럼 이름3  칼럼의 유형  제약조건 ... ) ;
```

```
ALTER TABLE EMP2
 ADD (BONUS NUMBER(8) DEFAULT 200 );

SELECT *
FROM EMP2
WHERE ROWNUM <=5
;
```

ENAME	JOB	SAL
SMITH	CLERK	800
ALLEN	SALESMAN	1600
WARD	SALESMAN	1250
JONES	MANAGER	2975

(+)

BONUS
200
200
200
200

*** Alter Table + DROP COLUMN : 컬럼"을" 삭제**

ALTER TABLE + DROP COLUMN

칼럼을 삭제 할 때 사용한다. 한번 삭제된 칼럼은 복구 할 수 없다.

```
ALTER TABLE 테이블 명
   DROP COLUMN 삭제할 칼럼명1 < , 칼럼명2 , 칼럼명3 .... >
```

```
ALTER TABLE  EMP2
 DROP COLUMN  BONUS ;
```

*** Alter Table + MODIFY : 컬럼의 속성 변경**

ALTER TABLE + MODIFY

테이블에 이미 존재하는 칼럼에 대해서 칼럼의 유형 , 길이 (크기) , DEFAULT 값(=NULL 값이 입력 될 때 대체 될 값) , NOT NULL 제약 조건에 대한 변경이 가능하다.

```
DROP TABLE EMP2 ;

CREATE TABLE EMP2 AS
SELECT *
FROM EMP;

ALTER TABLE EMP2 MODIFY ( DEPTNO  NOT NULL ) ;
```

* Alter Table + RENAME COLUMN A TO B : 컬럼명 변경 ★★ (<--> 테이블명 변경)

RENAME COLUMN

테이블에 존재하는 칼럼의 이름을 변경할 때는 RENAME COLUMN 을 사용한다.

```
ALTER TABLE 테이블명 RENAME COLUMN 기존칼럼명 TO 새로운칼럼명;
```

```
DROP EMP2;
CREATE TABLE EMP2 AS
SELECT * FROM EMP;

ALTER TABLE EMP2 RENAME COLUMN COMM TO COMMISION;
```

테이블 EMP2 에서 기존에 존재하던 칼럼 COMM 을 COMMISION으로 변경한다.

* RENAME A TO B : 테이블명 변경 ★★ (<--> 컬럼명 변경)

RENAME TABLE

1. 기존에 존재하는 테이블 명을 변경하고 싶은 경우 RENAME TABLE TO 문장을 사용한다.

★ 빈칸

3. `RENAME 기존테이블명 TO 새로운테이블명 ; <오라클 기준>`
4. `SP_NAME '기존테이블명', '새로운테이블명' ; <SQL SERVER>`

둘 다 나옴!

* Alter Table + ADD CONSTRAINT : 제약조건 추가 (변경 희망 시 삭제 후 다시 ADD하면 됨)

ADD CONSTRAINT

CREATE TABLE 문장에서 보통 제약 조건을 입력한다. 그러나 이미 생성되어 있는 테이블에 추가적으로 제약조건을 입력하고 싶은 경우 ALTER TABLE + ADD CONSTRAINT 를 사용하여 제약조건을 칼럼에 추가 한다. CONSTRAINT 는 수정 하는 개념은 없으므로 수정을 원한다면 기존 CONTRAINT를 삭제 하고 새롭게 만들어야 한다.

2. `ALTER TABLE 테이블명 ADD CONSTRAINT 제약조건명 제약조건(칼럼명)`

```
DROP EMP2;
CREATE TABLE EMP2 AS
SELECT * FROM EMP;
```

1. `ALTER TABLE EMP2 ADD CONSTRAINT PK PRIMARY KEY (ENAME, EMPNO);`

Alter Table(로) 테이블명 ADD Constraint(정의하겠다) PK명(라는) PRIMARY KEY를(지정할컬럼명)★

* Alter Table + DROP CONSTRAINT : 제약조건 삭제

DROP CONSTRAINT

테이블에 존재하는 제약조건을 삭제 하는 경우 ALTER TABLE + DROP CONSTRAINT 를 사용한다.

`ALTER TABLE 테이블명 DROP CONSTRAINT 제약조건명 제약조건(칼럼명)`

```
ALTER TABLE EMP2 DROP CONSTRAINT PK PRIMARY KEY (ENAME, EMPNO);
```

*** ALTERT TABLE로 컬럼 변경 시 주의사항 ★★★**

- 컬럼의 길이(크기)를 늘리는 것은 자유롭지만, 값이 존재할 경우 크기 줄이는거 X
  -> NULL값만 가지거나, 아무 행도 존재하지 않으면 컬럼 길이 줄이기 O
- 해당 컬럼이 NULL값만 가지고 있으면 타입(숫자,문자) 변경 가능
- NULL이 없는 경우에만 NOT NULL 제약조건 추가 가능
  -> 이미 NULL값이 존재하는 경우 NOT NULL 제약 조건 추가 불가
- DEFAULT값을 설정하는 경우, 변경 작업 이후 발생하는 행 삽입에 대해서만 DEFAULT값이 영향

*** DDL - Drop Table - 테이블 제거**

DROP TABLE

더 이상 사용하지 않는 테이블을 삭제하는 경우 DROP TABLE 을 사용한다. DROP TABLE 은 테이블 구조 및 행까지 모두 "영구적으로 완전 완벽하게" 삭제 한다. 따라서 DROP TABLE 을 사용 한 이후 DESCRIBE 테이블명 문장을 사용할 경우 다시 확인이 불가능하다.

DROP TABLE 테이블명 ;

DROP TABLE EMP2 ;

- **Drop Table** 테이블명 **CASCADE CONSTRAINT**; ★★★★★ !주관식!
  -> 테이블을 삭제하되, **참조 제약에 걸린 것까지 연쇄적으로 제거해라 <해당 컬럼만 아니라 다>**
  -> **DROP TABLE 시 자식 테이블에 참조 제약이 걸려있는 경우 삭제 시 에러뜰 수 있으므로**
  -> SQL Server에는 Cascade 옵션이 없음!

테이블끼리는 행의 입력 및 삭제가 연동이 되도록 참조 제약이 걸려 있는 경우가 있다. 예를 들어서 A 테이블에 1행 입력하면 B 테이블에도 자동적으로 입력되는 것이다. 이런 제약이 걸려 있는 경우 테이블은 DROP TABLE 테이블명 CASCADE CONSTRAINT 를 입력해야 한다.

DROP TABLE EMP2 CASCADE CONSTRAINT ;

*** DDL - Truncate Table (자르다 테이블) - 테이블의 모든 행 제거 (<-->Cascade 착각 ㄴㄴ)**

- **Truncate Table** 테이블명 **Drop Column** 삭제할 컬럼명
- 테이블 삭제가 아닌, **해당 테이블의 모든 행만 제거 후 저장공간을 재사용하도록 해제**해줌★★★

1 4. 테이블 삭제
DROP TABLE 테이블 명

2 5. 테이블 비우기
TRUNCATE TABLE

*** Alter Table 예시 (Not Null 주의) ★**

Alter Table 테이블명 **Alter Column** 컬럼명 dtype **[Not NULL]**;

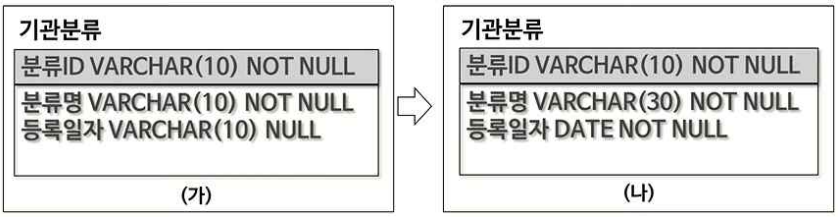
Alter Table 기관분류 Alter Column 분류명 VARCHAR(30) Not NULL;

*** DDL vs DML의 삭제 ★★★** (쏭즈 유튜브)

- **DDL은 반드시 AUTO COMMIT**이 일어남 => **DROP : 완전히 삭제, 원상복구 불가**
  TRUNCATE : 행만 완전히 삭제, 원상복구 불가
- **DML은 사용자가 COMMIT**해야함 => **Delete : 테이블 삭제하지만, ROLLBACK으로 복구 가능**

*** DDL - Alter Table - SQL server ★**

- 컬럼 추가 : Alter Table 테이블명 ADD 추가칼럼명 Dtype;
- 컬럼 삭제 : Alter Table 테이블명 DROP COLUMN 삭제컬럼명;
- 컬럼 속성 변경 : Alter Table 테이블명 ALTER COLUMN 컬럼명 Dtype;



=> **변경 사항만 적는게 아니라, 자료형과 제약조건(Constraint) 둘 다 완벽히 작성한다!**

*** DDL - 참고사항**

- DROP COLUMN: 데이터가 **있거나 없거나 모두 삭제 가능**. **한번에 하나의 칼럼만 삭제 가능**.
- DROP CONSTRAINT: 테이블 생성 시 부여했던 **제약조건을 삭제**하는 명령어
- ADD CONSTRAINT: 테이블 생성 이후에 **필요에 의해서 제약조건을 추가**

2-1-2-1. **참조동작 (Referential Action) ★★** !서술형!

*** Insert Action**

1) **Automatic** ★: **Child 삽입 시 Parent 테이블에 PK가 없으면, Parent PK 생성 후 Child에 삽입**
2) Set Null : child 삽입 시 Parent 테이블에 PK가 없으면, Child 외부키를 Null값으로 셋팅
3) Set Default : child 시 Parent테이블에 PK 없으면, Child 외부키를 지정된 Default값으로 셋팅
4) **Dependent**★: **Child 삽입 시 Parent 테이블에 PK가 존재할 때만, Child 삽입 허용 ★★**
5) No Action : 참조 무결성을 위반하는 삽입 액션은 취하지 않는다.

*** Delete / Modify Action**

1) **Cascade ★** : **Parent 삭제 시 Child 같이 삭제**
2) Set Null : Parent 삭제 시 Child의 해당 필드는 Null로 셋팅
3) Set Default : Parent 삭제 시 Child의 해당 필드를 Default값으로 셋팅
4) **Restrict ★** : **Parent 삭제 시 Child 테이블에 PK가 없는 경우**에만 **Parent 삭제 허용**
5) No Action : 참조 무결성을 위반하는 삭제/수정 액션은 취하지 않는다.

2-1-3. DML

* DML : **호스트 프로그램 속에 삽입되어 사용** => **데이터 부속어(Data Sub Language)**라고도 함 ★

*** Procedural DML** : **절차적 데이터 조작어** -> **초급언어** 니까 다 알려줘야함 ★

- 사용자가 **무슨 데이터(what)를 원하고, 어떻게(how) 접근해 처리할 것인지 명세**해야함

* **Nonprocedural DML : 비절차적 데이터 조작어 -> non초급 = 고급언어** 니까 어케하는지는 X ★
 - 사용자가 **무슨 데이터(what)를 원하는지만 명세**하고, **어떻게(how) 접근할 것인지는 X**
 - 사용자가 원하는 데이터만 선언하기 때문에, **선언적 언어(declarative language)**라고도 함

* **DML - Insert**
 - **Insert into** 테이블명 (컬럼명들 (생략 시 전체 컬럼))
  Values (위 컬럼명 순서에 맞춰 입력할 값 맵핑해 작성);
 - 주의사항 : 데이터가 문자형일 경우 ' ' (single quotation)으로 묶어서 입력, 숫자는 X

```
Insert Into Player (id, name, team_id, position, back_number)
Values (2002007, '박지성', 'K07', 'MF', 7);
```

* **DML - Update**
 - **Update** 테이블명
  Set 컬럼명 = 값 Where 조건;

```
Update Player
Set position = 'MF'
Where name = '박지성';
```

* **DML - Delete**
 - **Delete From** 테이블명 Where 조건;

```
Delete From Player;                          --> 조건절 없으면 전체 테이블 삭제

Delete From Player Where id = 2002007;       --> 해당 로우 제거
```

* **DML - Select**
 - Select [ALL | DISTINCT] 컬럼들 AS 엘리야스명
  From 테이블명;
 - ALL = 기본 옵션 (중복 데이터 모두 출력)
 - DISTINCT = 중복 제거 (중복인건 하나로 통일하여 출력)

```
Select * From Player;
Select Distinct position From Player;
Select name as 선수명 From player;
```

* **DML - 산술 연산자 우선순위 ★★**
 - **( )** -> ***** -> **/** -> **+** -> **-**

* **DML - 합성 연산자 ★★**
 - **문자와 문자를 연결** (이거였뜌!!! OR연산 아니다!!!!!!!!!)
 - [Oracle] **||** [SQL Server] **+**

```
[Oracle]
Select name || '선수', position || '포지션', back_number || '번' From Player Where id = 2002007;

>>> 박지성선수MF포지션7번
```

*** 삭제 SQL 비교 정리 ★★★★★★**

Drop Table 테이블명	Truncate Table 테이블명	Delete From 테이블명
DDL	DDL (일부 DML 성격 가짐)	DML
Auto Commit (DDL이니까)	**Auto Commit (DDL이니까)**	**사용자 Commit**
=> **RollBack 불가**	=> **RollBack 불가**	=>Commit 전에 **RollBack 가능**
테이블의 모든 데이터 삭제	테이블의 모든 데이터 삭제	테이블의 모든 데이터 삭제
디스크 초기화(=로그 제거)	**디스크 초기화(=로그 제거)**	**디스크 초기화 안함(=로그유지)**
스키마 정의까지 싹 다 삭제	**테이블 스키마 구조 유지**	**테이블 스키마 구조 유지**

2-1-4. TCL

* TCL ★★ (<-> DCL 주의)

- 논리적 작업 단위를 묶어 **DML에 의해 조작된 결과를 작업단위 별로 제어**
- 커밋, 롤백 등이 여기에 해당
- 일부에서는 **DCL로 분류하기도 한다** ★

*** 트랜잭션 ★★**

- 데이터베이스의 **논리적인 연산 단위** ★★
- 트랜잭션은 밀접히 관련되어 분리될 수 없는 한 개 이상의 DB조작을 가리킴
- 트랜잭션에는 **하나 이상의 SQL 문장이 포함됨**
- **분할할 수 없는 최소단위**★★이므로 -> **전부 적용하거나, 전부 취소해야 함**★★

* 트랜잭션 특성 ACID ★★★★

원자성(Atomicity)	트랜잭션의 연산은 **모두 적용되던지**, 아니면 **전혀 실행되지 않은 상태여야**
일관성(Consistency)	트랜잭션 **실행전 DB에 이상이 없다면**, **실행후에도 일관되게 이상이 없어야**
고립성(Isolation)	트랜잭션 **실행 중, 다른 트랜잭션의 영향을 받아선 안됨**
지속성(Durability)	트랜잭션이 성공적으로 수행되면, **영구적으로 반영되어 저장**된다

* DB트랜잭션에 대한 **격리성이 낮을 경우 발생하는 문제점 ★★**

- **Dirty Read**
  -> 다른 트랜잭션에 의해 수정되었지만, **아직 커밋되지 "않은" Dirty한 데이터를 읽는 것**
- **Non-Repeatable Read**
  -> **한 트랜잭션 내에서 같은 쿼리를 두 번 수행**했는데,
  그 사이 **다른 트랜잭션이 값을 수정/ 삭제**해서 **두 쿼리 결과가 다르게** 나타나는 현상
- **Phantom Read**
  -> **한 트랜잭션 내에서 같은 쿼리를 두 번 수행**했는데,
  첫 번째 쿼리에서 없던 유령 레코드(팬텀 레코드)가 두 번째 쿼리에서 나타나는 현상

* 트랜잭션을 컨트롤하는 TCL ★★

- **Commit** : **문제없이 처리된 트랜잭션**을 **전부** DB에 반영
- **RollBack** : 트랜잭션 **수행 이전 상태로 되돌림**
- 트랜잭션 대상 SQL -> DML : Insert, Update, Delete
  -> **Select : Select For Update 등 배타적 LOCK을 요구하는 것들**

* COMMIT
 - 삽입, 수정, 삭제한 이력이 문제가 없을 경우, COMMIT 명령어로 **변경 사항 적용**
 - **Commit 이전 상태** : 단지 **Memory Buffer에만 영향**을 주고, **이전 상태로 복구 가능**
   현재 사용자는 Select문으로 변경 결과를 확인 가능
   다른 사용자는 현재 사용자가 수행한 결과를 **확인도 불가능!!**
   변경된 행은 아직 잠금(Locking) 설정되어, **다른 사용자가 변경 불가능!!**

 - **Commit 이후 상태** : 데이터에 대한 변경사항을 DB에 **영구반영**
   이전 데이터는 영원히 잃어버림
   모든 사용자가 결과 조회 가능!!!
   변경된 행은 잠금이 해재되어, **다른 사용자가 변경 가능!!! 확인도 가능!!!**

* Auto Commit
 - **[Oracle] DDL 문장 수행 후 자동으로 COMMIT 수행 => 롤백해도 저장되어버림**★★★★★★★★
 - **[SQL Server] DDL 문장 수행 후 자동으로 COMMIT 수행하지 않음**

* RollBack
 - 테이블에 삽입, 수정, 삭제한 데이터에 대해 **COMMIT 이전에 변경사항을 취소(이전???????)**

 - **RollBack 후 상태** : 데이터에 대한 **변경사항 취소**됨
   이전 데이터가 다시 재저장됨
   관련 행에 대한 잠금이 풀리고, 다른 사용자들이 데이터 변경 가능

*** BEGIN TRANSACTION** ★★★ (이해 중요)
 - 트랜잭션을 시작하는 구문 -> **COMMIT이나 ROLLBACK으로 종료한다**
 - **ROLLBACK을 만나면 최초의 BEGIN시점까지 모두 ROLLBACK 수행됨**

품목ID	단가
001	1000
002	2000
003	1000
004	2000

Begin Transaction
Insert Into 품목(품목ID, 단가) Valuse ('005', 2000);
Commit --> 여기까지 OK : 종료된 것!!!
Begin Transaction --> 다시 시작
Delete 품목 Where 품목ID = '002'; --> 취소
Begin Transaction --> 아직 종료 안됨!
Update 품목 Set 단가=2000 Where 단가=1000; --> 취소
ROLLBACK --> **COMMIT되지 않은 트랜잭션 모두 취소**
Select Count(품목ID) From 품목 Where 단가=2000;
>>>결과 : 3

* Commit과 Rollback을 사용함으로써 얻을 수 있는 효과
 - 데이터 무결성 보장
 - 영구적인 변경을 하기 전에 데이터의 변경사항을 확인 가능
 - 논리적으로 연관된 작업을 그룹핑하여 처리 가능

*** SavePoint (=저장점)**

- savepoint를 지정하면 롤백할 경우, 전체 롤백이 아닌 **포인트까지의 일부만 롤백**할 수 있음
- savepoint는 **여러 개 지정할 수 있음**
- **동일 이름**으로 point 지정 시, **가장 나중에 정의한 point가 유효**함 (=덮어쓰기)

[Oracle] SavePoint 포인트이름; RollBack To 포인트이름;
[SQL Server] Save Transaction 포인트이름; RollBack Transaction 포인트이름;

* SavePoint 주의점

- **Point1로 되돌리고 나서 그보다 미래인 Point2로 되돌릴 수 없다**
- 특정 point까지 롤백하면, 그 이후에 설정한 point는 전부 무효가 된다
- **point없이 롤백하면 모든 병경사항을 취소**한다.

2-1-5. Where절

* WHERE 절

- 자신이 원하는 자료만을 검색하기 위해 이용
- Where절에 조건이 없는 **FTS (Full Table Scan) 문장은 SQL 튜닝 1차 검토 대상**임(=브루트폴스)
- FTS가 무조건 나쁜 것은 아님 -> 병렬 처리를 이용해 유용하게 사용하는 경우도 있음

* Where - 연산자 종류

- 비교 연산 : =, >, >=, <, <=
- 부정 비교 연산 : !=, ^=, <>, NOT 칼럼명 =, NOT 칼럼명 >
- **SQL 연산 : Between a AND b, IN (list), Like '비교문자열', IS NULL**
- 부정 SQL 연산 : NOT Between a AND b, NOT IN (list), IS NOT NULL
- 논리 연산 : AND, OR, NOT

* Where - 연산자 우선순위 : **부정=>비교,SQL=>논리** ★

- **( ) -> NOT(부정연산자) ->** 비교 연산자, SQL 연산자 -> AND -> OR

* Where - SQL 연산자

- Between a AND b : 사이 모든 값 **(a, b포함)**
- IN (list) : 리스트 안에 있는 값 중 하나라도 일치하면 True, 없으면 False
- LIKE '비교문자열' : 비교 문자열과 형태가 일치하면 됨
  -> **와일드카드 : % (=0개 이상 문자열), _ (=1개 단일 문자)**

* Where - IS NULL과 IS NOT NULL

- **IS NULL** : NULL값이면 True 아니면 False (=Null인 것만 찾기)

  -> **NULL값과 수치 연산은 NULL값을 리턴** ★ -> **NULL값과 비교 연산은 거짓(False)를 리턴** ★

- **IS NOT NULL : 널이 아닌 경우를 찾기 위해 사용** ★★★

```
SELECT PLAYER_NAME 선수이름, POSITION 포지션, TEAM_ID FROM PLAYER WHERE POSITION IS NULL;
결과>>> 선수이름 포지션 TEAM_ID ------ ----- ------- 정학범 K08 안익수 K08 차상광 K08
    (포지션 테이블에서 포지션이 NULL값을 갖는 선수이름, 포지션, 팀ID를 출력하라!)
```

* **(중요) NULL과의 모든 사칙연산 결과는 NULL이다** ★★★★★

- ex) 30+40+NULL = NULL 50/NULL = NULL 20*' ' = NULL

* Where - 논리 연산자

- 우선순위 : **( ) -> NOT -> AND -> OR** ★

```
"소속이 삼성블루윙즈이고 키가 170 센티미터 이상인 조건을 가진 선수들의 자료를 조회"
SQL>>
SELECT PLAYER_NAME 선수이름, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER
WHERE TEAM_ID = 'K02' AND HEIGHT >= 170;

"소속이 삼성블루윙즈이거나 전남드래곤즈인 선수들 중에서 포지션이 MF인 선수들 자료를 조회"
SQL>>
SELECT PLAYER_NAME 선수이름, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER
WHERE TEAM_ID IN ('K02','K07') AND POSITION = 'MF';

"소속팀이 삼성블루윙즈이거나 전남드래곤즈에 소속된 선수들이어야 하고, 포지션이 미드필더(MF)이며
키는 170 센티미터 이상이고 180 이하인 선수들 자료를 조회"
SQL>>
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;
```

* Where - 부정 연산자 <> != ^=

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.

```
"삼성블루윙즈 소속인 선수들 중에서 포지션이 미드필더(MF)가 아니고,
키가 175 센티미터 이상 185 센티미터 이하가 아닌 선수들의 자료를 찾아본다."
SQL>>
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;
```

*** RowNum (Oracle) ★★**

- 칼럼과 비슷한 성격의 Pseudo Column이다
- SQL 처리 결과 집합의 각 행에 대해 임시로 부여되는 일련번호이다
- 테이블이나 집합에서 원하는 만큼의 행만 가져올 때, **Where에서 행의 개수를 제한**하는 용도

```
"MY_TABLE 이라는 테이블의 첫번 째 칼럼을 '고유한 키'값 혹은 '인덱스 값'으로 설정하라!"
--> 새롭게 넘버링 칼럼을 설정하고, 그 값을 각 테이블의 '고유한 키'값 혹은 '인덱스 값'으로 설정
UPDATE MY_TABLE SET COLUMN1 = ROWNUM;

"PLAYER 테이블에서 PLAYER_NAME 번호가 3 이하인 선수 이름을 출력하라"
SELECT PLAYER_NAME FROM PLAYER WHERE ROWNUM <= 3;
```

-> 출력 결과의 3번째 행까지만 출력임

*** TOP (SQL server) ★**

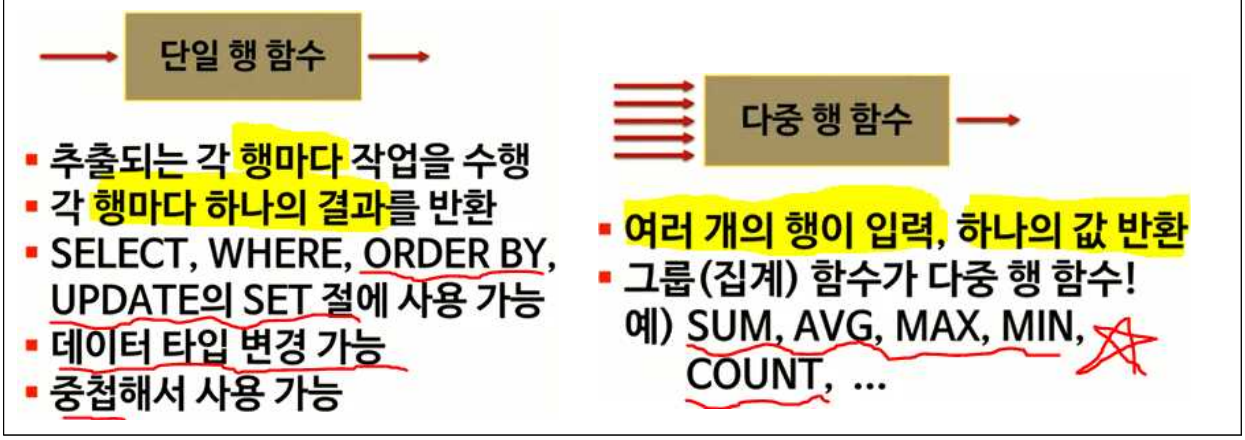
- select 결과물의 행의 수를 제한 (**맨 위에 n개만 출력하는 듯**)
- TOP (Expression) [PERCENT] [WITH TIES];
- Expression : 결과 리턴할 행의 수를 지정하는 int값
- PERCENT : 쿼리 결과 집합에서 처음 Expression%의 행만 반환됨을 나타냄
- **WITH TIES** : Order By절이 지정된 경우에만 사용 가능하며,
  동점은 추가 출력함 -> 밑에 RANK함수에서도 사용함

```
"PLAYER 테이블에서 1~5행까지의 PLAYER_NAME 을 출력하라"
SELECT TOP(5) PLAYER_NAME FROM PLAYER;
```

2-1-6. 함수

* **단일행 함수** : **내장함수 중 파라미터가 하나~여러 개인 함수** **(SUM, AVG 이런건 다중행임@!!!@)**
-> Select, Where, **ORDER BY, Update-Set 절에서 사용 가능하다!!!!** ★

* 단일 행 함수 VS 다중 행 함수 <이걸 이해!> ★



* 단일행 - 문자형 함수 **DB는 인덱스가 항상 1부터임**

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'

* 단일행 - 숫자형 함수 => **ROUND함수 중요**

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.

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)

* 단일행 - 날짜형 함수 ★★

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

연산	일수
DATE + NUMBER	
DATE - NUMBER	시간
DATE + NUMBER/24	
DATE1 - DATE2	

ORACLE은
세기, 년, 월, 일, 시, 분, 초
형식의 날짜를 내부적으로는
숫자 형태로 저장.

- 1 일
  SYSDATE + 1
- 1 시간
  SYSDATE + 1/24
- 10 분
  SYSDATE + 1/24/6
- 1 분
  SYSDATE + 1/24/60
- 매일 밤 11시
  TRUNCATE(SYSDATE) + 23/24

*** 날짜 연산 예제 (노랭이 42번) ★★★★**

날짜 자료형은 DATE임 => DATE = 하루 => +1 = 하루를 더해줌

2015.01.10 10 + 1 : 하루를 더해줌
+ 1/24 : 하루를 24로 나눔 = 한 시간을 더해줌
+ 1/24/(60/10) = 1/24/(6) : 한 시간을 6으로 나눔 = 10분을
+ 1/24/60 : 한 시간을 60으로 나눔 = 1분을 더해줌 더해줌

* 단일행 - 반환형 함수

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

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.

* **Case 절** ★★★
 - **IF-ELSE 문과 비슷**

```
Select   컬럼명,
         CASE When 조건절(Where 조건이랑 똑같음) Then True일 때 반환
            Else False일 때 반환
         END AS 컬럼명
From     테이블명;
```

*** SEARCHED_CASE_EXPRESSION => SIMPLE_CASE_EXPRESSION 표현 방법** ★★★

-> 똑같음 걍 '=' 기호 자리에 When 넣은것!!!!!!!!!!!

```
CASE WHEN LOC = 'NEW YORK' THEN 'EAST'
```

```
[SIMPLE_CASE_EXPRESSION 문장]
SELECT LOC,
   CASE LOC WHEN 'NEW YORK' THEN 'EAST'
      ELSE 'ETC'
   END as AREA
FROM DEPT;
```

* **NULL 관련 함수** ★★★★★ 문제 전부 중요

NVL(값1, 값2) : 값1 is null -> 값2 // is not null -> 값1
ISNULL(값1, 값2) : 값1 is null -> 값2 // is not null -> 값1

NVL2(값1, 값2, 값3) : 값1 is null -> 값3 // is not null -> 값2

NULLIF(값1, 값2) : 같으면 null // 다르면 값1

COALESCE(값1, 값2, 값3....) : 널 아닌 첫 번째 값
{ null, null, 2, 5, ... } 일 경우, 2를 반환

* **COALE SCE (코알러스)** ★★★★★★★★★★★★★

- **첫 번째 NULL이 아닌 값**을 반환 **(= NULL이 아닌 최초의 표현식을 나타낸다)**

TAB1

C1	C2	C3
①	2	3
	②	3
		③

```
SELECT SUM(COALESCE
   (C1, C2, C3))
FROM TAB1;
```

① 0 ② 1
✓③ 6 ④ 14

* 공집합
- 조건에 맞는 데이터가 한 건도 없는 경우를 공집합이라 함
- SELECT 1 FROM DUAL **WHERE 1 = 2**; 와 같은 조건이 대표적인 공집합을 발생시키는 쿼리
- **인수값이 공집합인 경우, NVL( )/ISNULL( ) 사용해도 공집합이 출력**

2-1-7. Group By Having 절

* **집계함수 (Aggregate Func)**
 - 여러 행들으로 구성된 그룹이 모여서 **그룹당 하나의 결과**를 돌려주는 **"다중행 함수"**
 - GROUP BY 절은 행들을 **소그룹화한다.**
 - **Select / Having / ORDER BY** 절에서 사용 가능

 - 집계함수명(ALL | Distinct 컬럼) - Default = ALL
 - 주로 숫자형에서 사용 // **MIN, MAX, COUNT는 문자, 날짜도 적용 가능**

집계 함수	사용 목적
COUNT(*)	NULL 값을 포함한 행의 수를 출력한다.
COUNT(표현식)	표현식의 값이 NULL 값인 것을 제외한 행의 수를 출력
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.

 => 집계함수 COUNT(*) 만 NULL포함해서 체크하고, ★★★★★★★★★
 => COUNT(컬럼명)과 SUM, AVG 그리고 IN, Between 등은 NULL 포함 안함!!! ★★★★★★★

* Group By 절
 - 데이터들을 작은 그룹으로 분류하여 소그룹에 대한 통계정보를 얻을 때 사용
 - ROLLUP이나 CUBE에 의한 소계가 계산된 결과에는 GROUPING(EXPR)=1이 표시됨
 - 그 외 결과에는 GROUPING(EXPR)=0이 표시

```
Select Distinct 컬럼명
From 테이블명
Where 조건식
Group By 컬럼/표현식
Having 그룹의 조건식;
```

"K-리그 선수들의 포지션별 평균키는 어떻게 되는가?"

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

결과 >>> 포지션 인원수 키대상 최대키 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

* Group By와 Having의 특징
 - **(중요) Group By에 의한 소그룹별 만들어진 집계 데이터 중, Having 조건을 만족하는 내용만 출력**
 - 가능하면 Group By하기 전에, Where절로 계산 대상을 줄이는게 효과적!!!!!
 - 즉, **Where절은 전체 데이터를 Group으로 나누기 전에 필요없는 조건을 미리 제거하는 역할**
 - ★**Having절은 Group By로 만들어진 소그룹에 대해서만 조건임을 명심하자!!!! ★★★★★★★★**

* Having 절
- Where절과의 차이는 1. Group By 뒤에 온다는 것 **(앞에 사용해도 에러는 안남!!)**
2. 집계함수를 사용 가능 (where은 사용불가)
- Where절 조건 변경은 출력되는 레코드 개수가 변경되고, 결과 데이터 값이 변경될 가능성 O
- Having절 조건 변경은 출력되는 레코드 개수는 변경되지만, 결과 데이터 값은 변경 X
- **Select-From-Having도 가능하다!! ★★**

2-1-8. ORDER BY 절

* Order By
- **(중요) ORDER BY에는 ★★★★GROUP BY의 컬럼이나 Select의 컬럼만★★★★이 올 수 있다**
- **숫자형 오름차순 -> 작은 값부터 출력** - **날짜형 오름차순 -> 빠른 날부터 출력**
- **[Oracle] Null값을 가장 큰 값으로 ★★★**, **[SQL Server] Null값을 가장 작은 값으로★★★** 간주

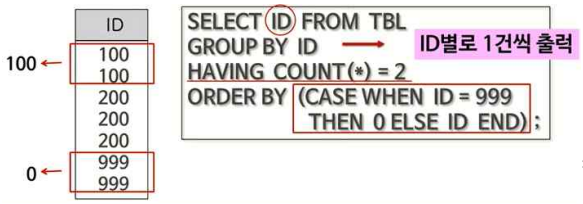
#. 만약 ORDER BY 1, 2 라면, 1번컬럼 기준으로 ASC한 후 2번컬럼 기준으로 ASC 한다 ★★★★

```
Select A, B, C
From table1
ORDER BY 2, 3
```

=> **Select문의 2번째로 먼저 정렬 후 3번째로 정렬**

=> **즉, B로 ASC 후 C로 ASC 함!!!** **=> Select에 A만 있는데 Order by 2 이러면 에러임★★**

*** ORDER BY에 CASE를 적용한 예시** (노랭이 55, 58번) ★★★★★★★★★



=>999의 인덱스를 0으로 바꿔 ASC 정렬

ID
999
100

* Select문의 실행 순서 ★★

실행 순서	설 명
5. Select 컬럼명	1. 테이블 참조 (From)
1. From 테이블명	2. 효율을 위해 조건에 안맞는 것 제거 (Where)
2. Where 조건식	3. 행들을 그룹화 (Group by)
3. Group By 컬럼/표현식	4. 그룹핑된 값의 조건에 맞는것만 도출(Having)
4. Having 그룹의 조건식	**5. 도출된 결과를 출력/계산 (Select)**
6. Order By 컬럼/표현식 [ASC | DESC];	**6. 출력 결과를 정렬한 후 출력함!! (Order by)**

*** TOP N 쿼리**
- SQL Server에서 ORDER BY 후에 TOP을 사용하면 상위/하위권 데이터를 추출 가능

* **WITH TIES 옵션**

팀별 성적 테이블에서 승리건수가 높은 순으로 3위까지 출력하되
3위의 승리건수가 동일한 팀이 있다면 함께 출력하기 위한 SQL로

```
SELECT TOP(3) WITH TIES 팀명, 승리건수
FROM 팀별성적
ORDER BY 승리건수 DESC;
```

=> 반드시 ORDER BY와 함께 사용한다!!

2-1-9. JOIN

- 두 개 이상의 테이블을 연결 또는 결합해 데이터를 출력
- **PK나 FK 값의 연관**에 의해 JOIN 성립
- **PK, FK 관계가 없어도 논리적인 값들의 연관으로 JOIN 성립 가능** ★★

- 한 SQL에서 여러 테이블을 조인할 수 있지만,
  SQL이 처리할 땐, **두 개의 집합 간에서만 JOIN을 수행함 => 그래서 AND로 계속 묶어야함** ★★

*** Equi JOIN (동등 조인, 이퀄 조인)**

- **'='연산자로 하는 조인**, "**그 외 비교연산자"를 사용할 경우는 모두 Non-Equi 다** ★★★
- 두 테이블간 **칼럼 값들이 서로 정확하게 일치하는 경우**에만 사용한다 ★★

[where절 방법]

```
Select ta1.컬럼명, ta2.컬럼명,
From table1 as ta1, table2 as ta2
Where ta1.컬럼A = ta2.컬럼B;
```

[ON절 방법]

```
Select ta1.컬럼명, ta2.컬럼명,
From table1 as ta1 INNER JOIN table2 as ta2
ON ta1.컬럼A = ta2.컬럼B;
```

*** Non Equi JOIN**

- 두 테이블간 **칼럼 값들이 서로 정확하게 일치하지 않는 경우**에만 사용됨
- "="연산자가 아닌 **Between, 부등호 등으로 하는 조인**
- **대부분 적용할 수 있지만**, 데이터 모델에 따라 **Non Equi JOIN이 불가한 경우도 있다**. ★★★

[where절 방법]

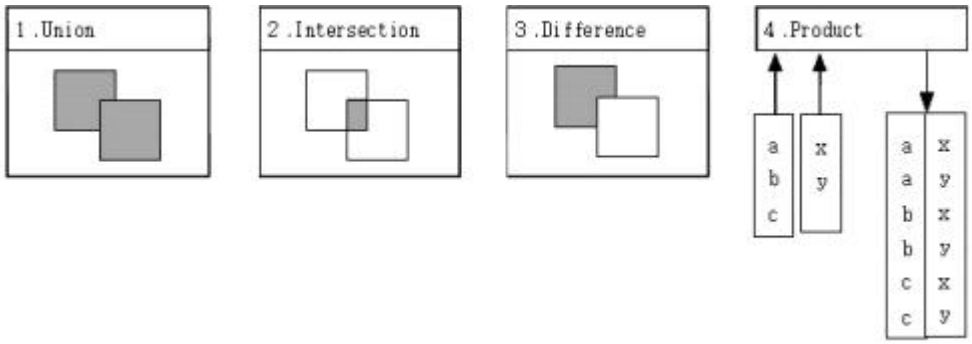
```
Select ta1.컬럼명, ta2.컬럼명,
From table1 as ta1, table2 as ta2
Where ta1.컬럼A Between ta2.컬럼A AND ta2.컬럼B;
```

2-2. SQL 활용

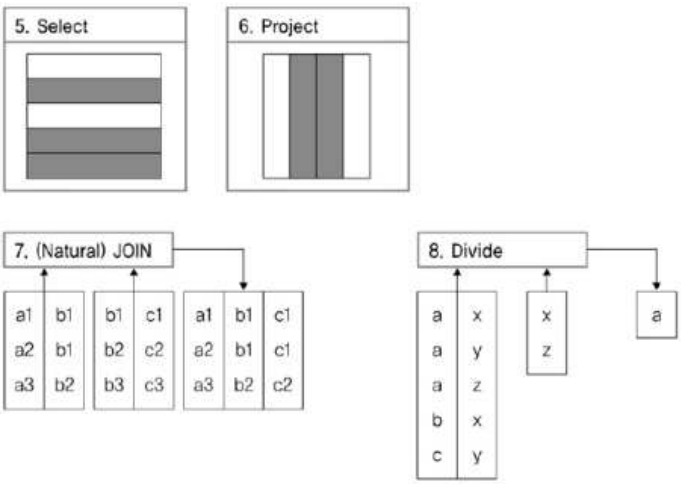
2-2-1. 표준 조인

* Standard SQL 개요

- **[일반 집합 연산자]** ★ => [SQL]
  - **Union** => Union
  - **Intersection** => Intersect
  - **Difference** => **EXCEPT (Oracle은 Minus)** ★★★★
    - ㄴ> **1:1 관계인 테이블에서 수행하면 공집합!!! (86번문제)**
  - **Product** => Cross Join



- **[순수 관계 연산자]** ★ => [SQL]
  - **Select** (σ) => Where
  - **Project** (π) => Select
  - (Natural) **JOIN** => 다양한 JOIN으로 기능 구현
  - **Divide** => 현재 사용하지 않음



*** 3개 이상 조인하기 ★★★★★★★★★★★★**

#. 방법 1

```
Select *
From A, B, C
Where A.id = B.id
AND B.id = C.id
AND D.id = ...
AND 조건절
```

#. 방법 2 (ANSI문법)

```
Select *
From A JOIN B
ON A.id = B.id JOIN C
ON C.id = B.id JOIN D ...
ON 조건절
Where 조건절
```

* From절에 Join 형태 (가능!!)
 - **Inner JOIN / Natural JOIN / USING 조건절 / ON 조건절 / Cross JOIN / Outer JOIN**
 - 기존 Where절 그대로 사용 가능
 - From절에서 JOIN 조건을 명시적으로 정의 가능

*** INNER JOIN**
 - **"내부 JOIN", "Equi JOIN", "동등 조인"**이라고도 함 -> **"=" 연산자**로 조인 (원래 다른건데 헷갈려서 통일함)
 - JOIN 조건에서 **동일한 값이 있는 행만** 반환
 - **Cross Join, Outer Join과 같이 사용할 수 없다!!**
 - **USING 조건절이나 ON 조건절을 필수적으로 사용**
 - 중복 테이블의 경우 별개의 컬럼으로 표시됨

```
# 내부 JOIN 기본 형태 ★
Select emp.depno, dname From emp, dept
Where emp.deptno = dept.deptno;

# 변환 주의! ★
Select emp.deptno, dname
FROM emp INNER JOIN dept ON emp.deptno = dept.deptno;

# 생략도 가능! ★
Select emp.deptno, dname
FROM emp JOIN dept ON emp.deptno = dept.deptno;
```

*** NATURAL JOIN**
 - 두 테이블간 **동일한 이름을 갖는 모든 컬럼에 대해 EQUI JOIN**을 수행
 - **Using이나 On이나 Where에서 JOIN조건을 정의할 수 X**
 - **Alias나 접두사 붙일 수 X**

```
Select deptno, empno
From emp NATURAL JOIN dept;
```

*** Using 조건절**
 - **From절에** 얘를 이용해서 같은 이름 컬럼 중 **원하는 컬럼만 EQUI JOIN할 수 있다**
 - SQL server는 지원X
 - **Alias나 접두사 붙일 수 X**
 - **JOIN에 사용되는 컬럼은 1개만 표시**★★★★★★

```
#. ON을 사용하지 않고 조인하는 방법!! => 상식적으로 컬럼명이 같아야 JOIN 가능!!!!!
Select *
From dept JOIN dept_temp
USING (deptno); ★★★★★★     (= ON (dept.deptno = dept_temp.deptno)★★★★★)
 --> USING (dept.deptno = dept_temp.deptno) 틀림!!!!!! ★★★★★★
```

*** ON 조건걸**
 - **컬럼명이 달라도 JOIN 사용 가능!!!!!!**(ON empid = empnum) **(<--> Equi JOIN, USING조건절)**
 - **Alias나 접두사 붙일 수 O, Alias나 접두사를 반드시 사용해야 함**

* **CROSS JOIN 또는 "카티시안 곱" ★★★**
  - **JOIN하려할 때 적절한 JOIN 조건 컬럼이 없는** 경우에 사용!!!!!★★★
    생길 수 있는 **모든 데이터 조합을 출력**
  - **Cartesian Product (카티시안 곱)★★★** / **Cross Product★★★**와 같은 표현
  - 결과는 양쪽 집합의 **M*N 건의 데이터 조합 발생**

#. Cartesian Product 활용 ★★★

```
Select ename, dname
From emp, dept        <-----카타시안 프로덕트 주의
ORDER BY ename
```

#. JOIN 활용 ★★★

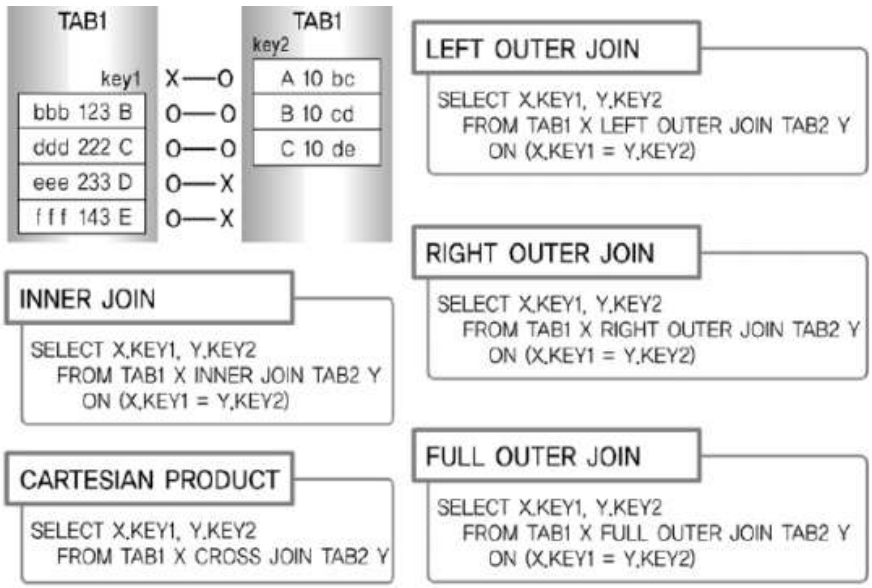
```
Select ename, dname
From emp CROSS JOIN dept
ORDER BY ename
```

* **OUTER JOIN ★**
  - **JOIN 조건에서 동일한 값이 없는 행도(=NULL값도)** 출력
  - Using 조건절이나 ON 조건절을 필수적으로 사용해야함
  - IN / ON 연산자 사용시 에러 발생
  - 표시가 누락된 컬럼이 있을 경우 OUTER JOIN 에러 발생
  - FULL OUTER JOIN 미지원으로 인해 STANDART JOIN을 주로 사용

* **Left OUTER JOIN (또는 Right OUTER JOIN)**
  - 좌측 테이블에서 먼저 데이터를 읽은 후, 우측 테이블에서 JOIN 대상을 읽음
  - 좌측 테이블 기준이며, 'OUTER' 키워드는 생략 가능
  - Right는 방향만 바꾸면됨

* **FULL OUTER JOIN**
  - 조인이 되는 모든 테이블의 데이터를 읽어 JOIN함
  - Left, Right 조인 결과의 합집합

* 각 JOIN 예시 쿼리



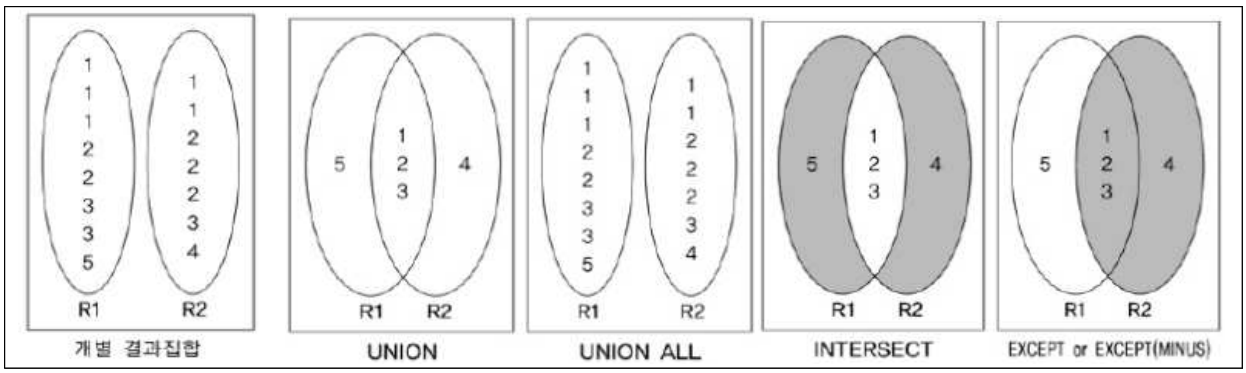
2-2-2. 집합 연산자

* 집합 연산자 (SET_OPERATOR)
 - 두 개이상의 테이블에서 JOIN을 사용하지 않고, 연관된 데이터를 조회하는 방법
 - 집합 연산자는 2개 이상의 질의 결과를 하나의 결과로 만듦

* 집합 연산자를 사용하기 위한 제약조건
 - Select절의 컬럼 **수가 동일**
 - Select절의 동일 위치 데이터 **타입이 상호 호환 가능**해야함 (동일할 필요는 x)

* 집합 연산자 종료 ★
 - **UNION** : 합집합, **중복된 행은 하나로 표시 (중복 제거)**
 - **UNION ALL** : 합집합, 중복된 행도 전부 표시 **(전부 표시)**
 - **INTERSECT** : 교집합, **중복된 행은 하나로 표시 (중복 제거)**
 - **EXCEPT** : 차집합, **중복된 행은 하나로 표시 (중복 제거)**

```
Select ---
From 000
[ Where ~~ ] [ Group By --- Having ~~~ ]
<집합연산자>
Select ---
[ Order By 000 ]
```



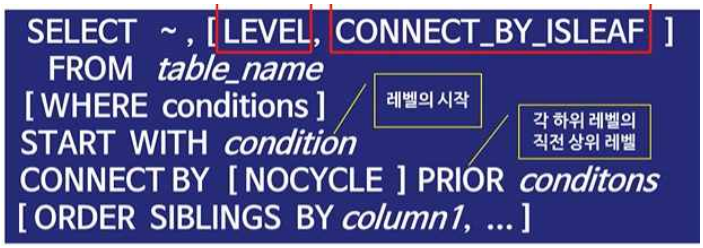
2-2-3. 계층형 질의와 셀프 조인

* 셀프 조인
 - 나 자신을 JOIN ~> **이때 두 컬럼 다 Alias 반드시 사용해야 식별 가능!**

* 계층형 질의 (Hierarchical Query)
 - 테이블에 계층형 데이터가 존재하는 경우의 데이터를 조회하기 위해 사용
  -> 계층형 데이터 : 동일 테이블에 계층적으로 상/하위 데이터가 포함된 데이터
  ex) 사원 테이블의 사원들 사이에 "하위 사원"과 "상위 사원(관리자)" 관계
 - 엔터티를 순환관계 데이터 모델로 설계할 경우 계층형 데이터가 됨

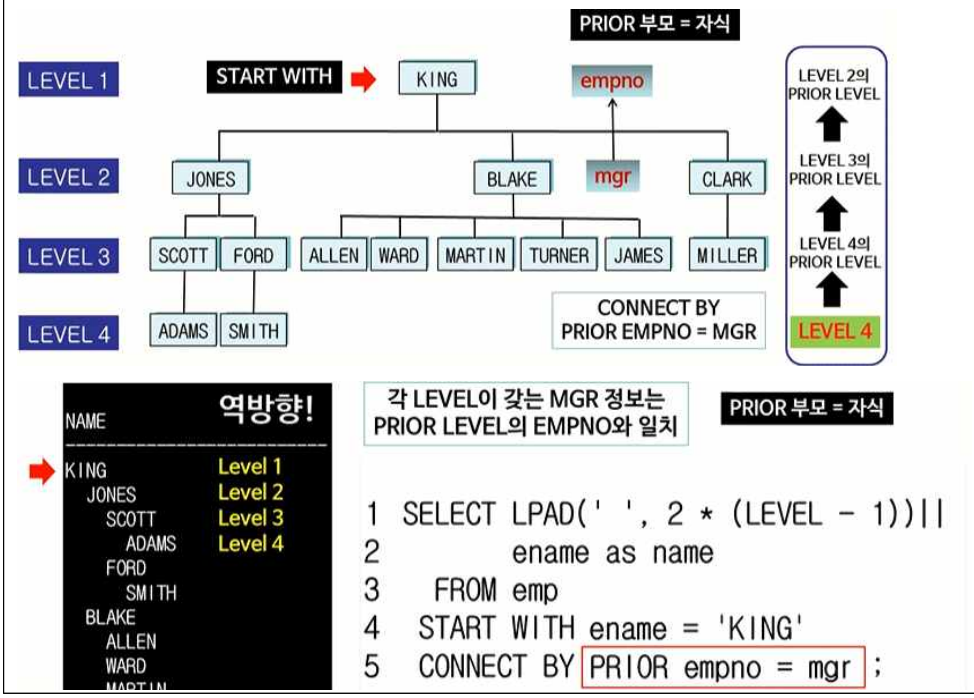
*** Oracle 계층형 질의** (쏭즈 : https://www.youtube.com/watch?v=JxyMJNX24LY)

- 계층형 질의에서 사용되는 가상 컬럼
  -> **LEVEL** : 루트 데이터를 1로 시작하여, Leaf까지 **하위로 갈수록 1씩 증가**
  -> **Connect_By_isLeaf** : **리프 데이터라면 1을 리턴**, 아니면 0
  -> **Connect_By_isCycle** : 자식을 갖는데, **해당 데이터가 조상 데이터면 1**, 아니면 0



- **START WITH 절** : 레벨의 시작
- **CONNECT BY 절** : 그 다음에 자식 레벨 지정 (이때 Connect By절의 조건 만족해야옴)
- **PRIOR** : Connect By절에 사용되며, 현재 읽은 칼럼을 지정
  PRIOR 자식 = 부모 ~> [부모 -> 자식]으로 **순방향 전개, 리프=1** (c언어 변수 할당처럼)
  PRIOR 부모 = 자식 ~> [자식 -> 부모]으로 **역방향 전개, 루트=1** (c언어 변수 할당처럼)

#. 역방향 예시



- **ORDER SIBLINGS BY**
  => 형제 Node 위치를 바꿈 ★★★

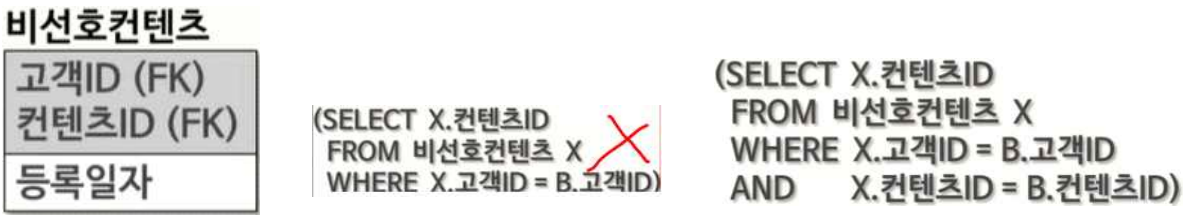
- **NoCycle** : 이미 나타난 동일한 데이터가 전개 중에 다시 나타나면, 이것을 Cycle 형성이라 함
  => 사이클 발생한 데이터는 런타임 오류 발생!!
  => 따라서 NOCycle 추가! ~> 사이클 발생 이후 데이터는 전개 X
- **SYS_Connect_By_Paht(컬럼명, 경로분리자)**
  => 루트 데이터부터 현재 전개할 데이터까지 경로를 표시
- **Connect_By_ROOT 컬럼명**
  => 현재 전개할 데이터의 루트 데이터 표시 (단항 연산자)

2-2-4. 서브쿼리

* 서브쿼리 주의사항
 - **단일행 또는 복수행 비교 연산자와 함께** 사용 가능 ★★★
 - **서브쿼리에선 ORDER BY를 사용 불가!!! => 메인 쿼리의 마지막 부분에만 위치 가능**
 - 서브쿼리는 메인쿼리의 테이블의 칼럼 사용 가능 **(메인쿼리에선 서브쿼리의 칼럼 사용 불가)**

* 서브쿼리가 SQL문에서 사용 가능한 곳 ★★★
 - **Select절, From절, Where절, Having절, Order By절**
 - Insert 문의 Values절, Update 문의 SET절 (delete만 안됨)

*** 서브쿼리 주의사항 ★★★**
 - 서브쿼리 **안에 있는 테이블(=조인할 테이블)의 기본키가 2개**면
 Where절에서 조인('=')도 기본키 **두 개 다 해줘야** 한다!



*** Un-Correlated 서브쿼리 (비연관 서브쿼리)**
 - **서브쿼리가 메인쿼리 테이블의 칼럼을 갖고 있지 않은 형태 (흔히 쓰는)**
 - 메인쿼리에 값만 제공하기 위한 목적으로 주로 사용

*** Correlated 서브쿼리 (연관 서브쿼리)**
 - **서브쿼리가 메인쿼리 테이블의 칼럼을 갖는 형태** ★★
 - 메인쿼리가 먼저 수행되어 읽은 데이터를 서브쿼리에서 조건이 맞는지 확인하려 할 때 사용

[반환되는 데이터 형태에 따른 서브쿼리 분류]

*** 단일행 서브쿼리**
 - 서브쿼리의 **결과가 항상 1건 이하**인 서브쿼리 (select문 출력결과 말하는 듯)
 - **단일행 비교 연산자(= , <, > 등) 함께 사용** ★★
 - **다중행 비교 연산자(IN, EXISTS, ALL, ANY 등)도 사용 가능!@@** ★★★★

*** 다중행 서브쿼리**
 - 실행 결과가 여러 개인 서브쿼리
 - **다중행 비교 연산자(IN★, ALL, ANY, SOME, EXISTS★ 등) 얘네만 함께 사용** ★★
 - **" Where ROWNUM = 1 "** ★★★
 => 서브쿼리 결과가 중복이 있어서 에러 뜰 경우 중복을 없애고 하나만 가져오게 하는 방법

*** Multi Column 서브쿼리 (다중 컬럼 서브쿼리)**
 - 실행 결과로 여러 컬럼을 반환
 - 조건절에 여러 컬럼을 동시에 비교 가능
 - 서브쿼리와 메인쿼리에서 비교하고자 하는 컬럼의 개수, 위치가 동일해야 함

[다양한 위치에서 사용하는 서브쿼리]

*** Select절에서 사용** (이해 https://youtu.be/_VY05qoz2eA)

- Scalar 서브쿼리 (스칼라 서브쿼리) => **단일행 연관 서브쿼리임** ★★
- 스칼라 서브커리 대신 **JOIN으로 동일한 결과를 추출 가능** ★★

```
SELECT FIRST_NAME,
       LAST_NAME,
       SALARY,
       (SELECT B.DEPARTMENT_NAME, B.LOCATION_ID
          FROM DEPARTMENTS B WHERE B.DEPARTMENT_ID = A.DEPARTMENT_ID)
  FROM EMPLOYEES A
 WHERE SALARY > 5000 ;
```

=>join과 결과 동일

*** From절에서 사용** (이해 https://youtu.be/efkQFP0wj08)

- 인라인 뷰 (Inline View) => **얘도 결국 서브쿼리다, 서브쿼리로 만든 가상 테이블** ★★
- SQL이 실행될 때만 **임시적으로 생성되는 동적 뷰 => DB에 해당 정보 저장X "일회성이다"**
- 일반 뷰 = 정적 뷰(Static View)라고 했을 때,
  인라인 뷰 = 동적 뷰(Dynamic View)라고 함

```
SELECT A.DEPARTMENT_NAME,
       B.AVG_SAL
  FROM DEPARTMENTS A,
       (SELECT DEPARTMENT_ID,
               ROUND(AVG(SALARY),2) AVG_SAL
          FROM EMPLOYEES
         GROUP BY DEPARTMENT_ID) B
 WHERE A.DEPARTMENT_ID = B.DEPARTMENT_ID;
```

=>join과 결과 동일

=> **안에 있는 서브쿼리로 원하는 내용만 추출해서 새로운 테이블을 만드는 개념** ★★

*** Where절에서 사용** (이해 https://youtu.be/oc-ya1MpK5c)

- 중첩 서브쿼리
- **단일행 중첩 서브쿼리 / 다중행 중첩 서브쿼리** 로 구분

```
SELECT * FROM HR.EMPLOYEES A
 WHERE A.DEPARTMENT_ID = (SELECT B.DEPARTMENT_ID
                            FROM HR.DEPARTMENTS B
                           WHERE B.LOCATION_ID=1700);
```

-> 이 서브쿼리의 결과가 1건 나오면 단일행 ~> ' =, >, < 등'으로 메인쿼리와 연동 ★★

-> 이 버스쿼리의 결과가 2건이상 다중행 ~> IN, ALL, ANY 등으로 메인쿼리와 연동 ★★

(단일행 비교 연산자로 연동하면 에러) ★★

```
SELECT *
  FROM HR.EMPLOYEES A,
       HR.DEPARTMENTS B
 WHERE A.DEPARTMENT_ID = B.DEPARTMENT_ID
   AND B.LOCATION_ID = 1700;
```

<= 조인일 경우

*** Having절에서 사용**

- **그룹함수**와 함께 사용될 때, **그룹핑된 결과에 대해 부가 조건을 걸기 위해** 사용

*** Update문의 SET절에서 사용**

- 서브쿼리를 사용한 변경 작업을 할 때, 서브쿼리 결과가 NULL을 반환하면
  해당 컬럼 결과가 NULL이 될 수 있기 때문에 주의

*** NOT EXISTS 와 서브쿼리 예시 ★★★★★★★★★★**

- where문에 NOT Exists는 **서브쿼리 테이블의 결과물을 제외한 나머지를 메인쿼리에 출력**
- 메인쿼리에서 **서브쿼리의 결과물이랑 겹치는 애들은 제외 (차집합)**

```
WHERE NOT EXISTS (SELECT 1 FROM TBL2 B
                  WHEREE A.ID = B.ID)
```

=> 서브쿼리 결과를 제외하고 출력

*** EXISTS 일 경우 예시 ★★★★★★★★★★**

- where문에 Exists는 **서브쿼리 테이블의 결과물과 겹치는 데이터만 메인쿼리에 출력**

```
EXISTS (SELECT 1 FROM TBL2 B
        WHEREE A.ID = B.ID)
```

=> 서브쿼리 결과랑 겹치는 애들만 출력

*** 뷰 (View) ★** (링크 : https://youtu.be/h-J4VtPP0bg)

- **가상테이블, VIEW, STORED QUERY 다 같은말**
- 테이블은 실제로 데이터를 가지고 있는 반면, **뷰는 실제로 데이터를 갖지 않음 ★★**
- 뷰는 단지 **뷰의 정의만을 가짐 ★★**

#. 뷰 생성

- (이해) 안에 들어가는 **이 작은 쿼리의 Select결과물로 가상 테이블을 만든다 ★★**

* 뷰 특징 ★ (103번 문제)

뷰 사용의 장점

뷰의 장점	설명
독립성	테이블의 구조가 변경돼도 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	상위 관리자가 뷰를 생성할 때 조회 할 수 있는 테이블과 칼럼을 지정할 수 있으므로 이용자가 접근 가능한 정보를 제한 할 수 있다.

2-2-5. 그룹함수 (링크 : https://youtu.be/pXq_iWt704c)

* 그룹함수
  - **전체 집계와 소계를 한번에 구하는 함수**
  - **Group by절 안에 다음 집계함수 사용**

*** ROLLUP 함수★★★**
- **A를 기준으로 소계 및 합계 생성**
- Group By Rollup(A) : 전체 합계, 컬럼 A소계
- Group By Rollup(A, B) : 전체 합계, 컬럼 A소계, 컬럼 {A, B} 조합 소계 **(B 소계 X)**
- Group By Rollup(A, B, C) : 전체 합계, 컬럼 A소계, 컬럼 {A,B} 조합 소계, {A, B, C}조합 소계
- Group By Rollup(A, (B,C)) : 전체 합계, 컬럼 A소계, 컬럼 {A, (B, C)} 조합 소계

- Group By A, Rollup(B) : A그룹별 집계, A그룹 내부에서 B 컬럼별 집계
- Group By A, Rollup(B, C) : A그룹별 집계, A그룹 내부에서 B 컬럼별 집계
  A그룹 내부에서 B, C 조합별 집계

=> **Rollup(A, B) =/= Rollup(B, A) 주의 (결과 다름)**★★★★★★★★★★★★

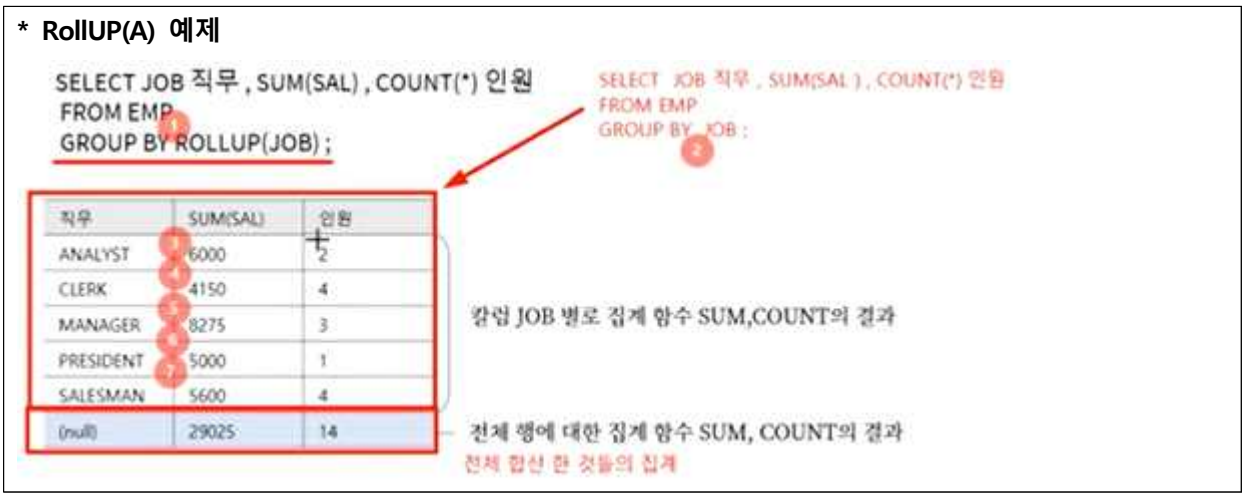
=> **롤업과 큐브는 시스템에 부담 딱히 안줌 둘다 동일한 부담임**★★★★★★★★★★

*** CUBE 함수★★★**
- **가능한 모든 조합의 소계 및 합계를 생성 => 시스템에 무리가 갈 수 있음!**
- GROUP BY CUBE(A) : 전체 합계, 컬럼 A소계
- GROUP BY CUBE(A, B) : 전체 합계, 컬럼 A소계, **컬럼 B 소계**, 컬럼 A, B 조합 소계

*** Grouping Sets 함수★★★** (<--> Grouping 함수)
- **내가 보고싶은 것만 소계를 생성**
- Group By Grouping Sets(A) : 컬럼 A 소계 **(전체 합계 X)**
- Group By Grouping Sets(A, B) : 컬럼 A 소계, 컬럼 B 소계 **(전체 합계 X)**
- Group By Grouping Sets((A, B) : 컬럼 (A, B)소계 **(전체 합계 X)**

=> **Grouping Sets(A, B) = Grouping Sets(B, A) (결과 같음)**★★★★★★★★★★★★

*** RollUP(A) 예제**



*** RollUP(A, B) 예제**

ROLLUP(A,B) 로 기술되어 있는 경우

1. 전체 합계 2. 칼럼 A 의 소계 3. 칼럼 A와 B 조합의 소계 를 구하게 된다.

```
SELECT B.DNAME 부서명 , A.JOB 직업 , COUNT(*) AS 인원 , SUM(A.SAL) AS 연봉합
 FROM EMP A , DEPT B
 WHERE A.DEPTNO = B.DEPTNO
 GROUP BY ROLLUP(B.DNAME , A.JOB) ;
```

부서명 (DNAME)	직업 (JOB)	인원	연봉합	
SALES	CLERK	1	950	3. 칼럼 A , B 조합의 소계
SALES	MANAGER	1	2850	
SALES	SALESMAN	4	5600	
SALES	(null)	6	9400	2. 칼럼 A 기준 소계
RESEARCH	CLERK	2	1900	3. 칼럼 A , B 조합의 소계
RESEARCH	ANALYST	2	6000	
RESEARCH	MANAGER	1	2975	
RESEARCH	(null)	5	10875	2. 칼럼 A 기준 소계
ACCOUNTING	CLERK	1	1300	3. 칼럼 A , B 조합의 소계
ACCOUNTING	MANAGER	1	2450	
ACCOUNTING	PRESIDENT	1	5000	
ACCOUNTING	(null)	3	8750	2. 칼럼 A 기준 소계
(null)	**(null)**	**14**	**29025**	1. 전체 합계

*** Group By A, ROLLUP(B) 형태 예제**

- A에 대해 먼저 쪼개서 개만 따로 하고 합쳐줌

group by a , rollup(b,c) 의 형태인 경우

GROUP BY 에 ROLLUP 함수 이외 추가적으로 칼럼을 추가 하는 경우
GROUP BY 절에 DNAME 을 ROLLUP 함수 전에 기입하는 경우 DNAME으로 전체 행을 분할한다.
기존 ROLLUP 함수를 사용하는 경우 전체 행에 대하여 집계를 구하지만 GROUP BY 에 칼럼을
기입하는 경우 전체 행이 기준이 아니라 그룹으로 나눠진 이후 상태에서 ROLLUP 함수가 작동한다.

```
SELECT JOB , DNAME , COUNT(*) AS 인원 , SUM(SAL)
FROM EMP, DEPT
WHERE EMP.DEPTNO = DEPT.DEPTNO
GROUP BY DNAME , ROLLUP(JOB) ;
```

DNAME	JOB	인원	SUM(SAL)
SALES	CLERK	1	950
SALES	MANAGER	1	2850
SALES	SALESMAN	4	5600
SALES	(NULL)	6	9400
RESEARCH	CLERK	2	1900
RESEARCH	ANALYST	2	6000
RESEARCH	MANAGER	1	2975
RESEARCH	(NULL)	5	10875
ACCOUNTING	CLERK	1	1300
ACCOUNTING	MANAGER	1	2450
ACCOUNTING	PRESIDENT	1	5000
ACCOUNTING	(NULL)	3	8750

칼럼 DNAME ; SALES 안에서 JOB 별로 집계 함수
+
칼럼 DNAME 별 합계 ; 전체 행의 집계는 실행하지 않는다.

*** Grouping 함수** (<--> Grouping Sets) ★★★

- **ROLLUP에 의해 "집계"가 일어나는 경우 111111, 일어나지 않는 경우 0을 표기**

```
SELECT DNAME, JOB , COUNT(*) AS 인원 , SUM(SAL) , GROUPING(DNAME) , GROUPING(JOB)
FROM EMP, DEPT
WHERE EMP.DEPTNO = DEPT.DEPTNO
GROUP BY ROLLUP(DNAME, JOB) ;
```

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

*** CUBE(A, B) 예시**

```
SELECT DNAME, JOB , COUNT(*) AS 인원 , SUM(SAL)
FROM EMP, DEPT
WHERE EMP.DEPTNO = DEPT.DEPTNO
GROUP BY CUBE(DNAME, JOB)
ORDER BY 1,2,3,4 ;
```

DNAME	JOB	인원	SUM(SAL)
ACCOUNTING	CLERK	1	1300
ACCOUNTING	MANAGER	1	2450
ACCOUNTING	PRESIDENT	1	5000
ACCOUNTING	(null)	3	8750
RESEARCH	ANALYST	2	6000
RESEARCH	CLERK	2	1900
RESEARCH	MANAGER	1	2975
RESEARCH	(null)	5	10875
SALES	CLERK	1	950
SALES	MANAGER	1	2850
SALES	SALESMAN	4	5600
SALES	(null)	6	9400
(null)	ANALYST	2	6000
(null)	CLERK	4	4150
(null)	MANAGER	3	8275
(null)	PRESIDENT	1	5000
(null)	SALESMAN	4	5600
(null)	(null)	14	29025

(DNAME , JOB 칼럼 조합별 집계) , DNAME 칼럼 집계

(DNAME , JOB 칼럼 조합별 집계) , DNAME 칼럼 집계

(DNAME , JOB 칼럼 조합별 집계) , DNAME 칼럼 집계

JOB 칼럼 별 집계

전체 총 집계 결과

*** Grouping Sets(A, B) 예시★★★**

=> 이때, **Grouping Sets(A, B) = Grouping Sets(B, A) 이라고 판단함(결과는 같은거임)★★★**

```
SELECT DNAME, JOB , COUNT(*) AS 인원 , SUM(SAL)
FROM EMP, DEPT
WHERE EMP.DEPTNO = DEPT.DEPTNO
GROUP BY GROUPING SETS(DNAME, JOB)
ORDER BY 1,2,3,4
;
```

DNAME	JOB	인원	SUM(SAL)	
ACCOUNTING	(null)	3	8750	
RESEARCH	(null)	5	10875	DNAME 칼럼 별 집계
SALES	(null)	6	9400	
(null)	ANALYST	2	6000	
(null)	CLERK	4	4150	
(null)	MANAGER	3	8275	JOB 칼럼 별 집계 ★★★
(null)	PRESIDENT	1	5000	
(null)	SALESMAN	4	5600	

*** grouping Sets() 주의사항 ★★★**

- GROUP BY GROUPING SETS (**A, B**, **( )**) 하면
  -> **A 소계, B 소계, 전체 총합** 이렇게 나옴!!!

2-2-6. 윈도우 함수

* Window 함수

- 전체 테이블 중에서 내가 원하는 일부만 작은 테이블로 만들어서 분석

*** Select에 Window계열(컬럼에적용한다) OVER (이러한 조건들을 활용해서)**

SELECT ④ WINDOW함수(⑤) OVER (<행을 분할> < 행을 정렬>< 대상 행 지정 >)
FROM *TABLE*; 윈도우 함수 ~에게 (윈도우 함수 지정 범위)

⑥ 행을 분할 : PARTITION BY (GROUP BY 같은 역할)
⑦ 행을 정렬 : ORDER BY (ORDER BY 역할)
⑧ 행을 지정 : ROWS 또는 RANGE (WHERE 의 역할)

WINDOW_FUNCTION OVER (<PARTITION BY 칼럼 > < ORDER BY 절 > <WINDOWING 절 >)

=> 순서 중요!!! 이중 하나 빠져도 상관은 없다!

=> Partition BY 대신 FROM절 이후에 GROUP BY로 묶어도 된다!! ★★

[윈도우 함수 종류 - 밑에 PPT로 설명 캡쳐해있음] ★

*** 순위 함수**

- Rank 함수 - Dense_Rank 함수 - Row_Number 함수

*** 윈도우 집계 함수 (흔히 아는 집계함수)**

- SUM, MAX, MIN, AVG, COUNT 등

*** 행 순서 함수**

- First_Value, Last_Value - LAG 함수 - LEAD 함수

*** 비율 함수**

- Ratio_To_Report 함수 - Percent_Rank 함수 - Cume_Dist 함수 - NTILE 함수

*** WINDOW함수의 Windowing절**

- **Rows** : 행의 수를 선택할 때 사용
- **Range** : 값의 범위를 선택할 때 사용

ROWS

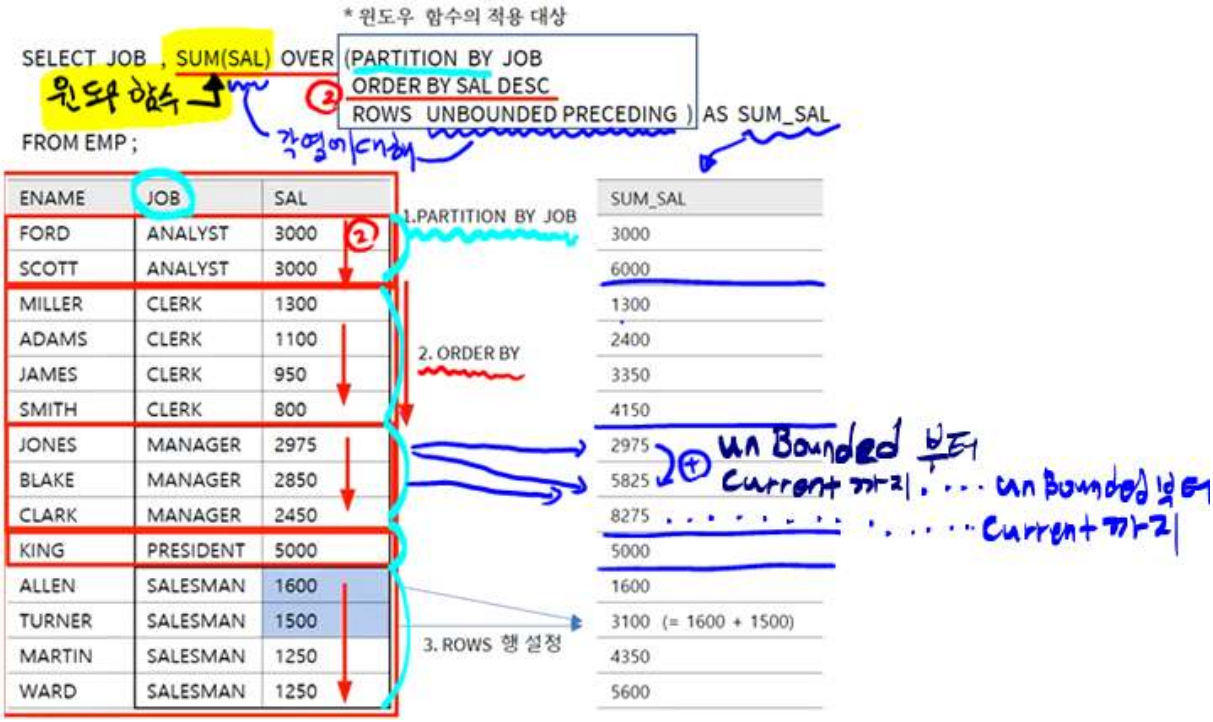
ROWS 의 경우 행을 호칭하는 방식을 다음과 같이 정리 할 수 있다.

JOB	ENAME	SAL	
CLERK	SMITH	800	← UNBOUNDED PRECEDING
CLERK	JAMES	950	↑ PRECEDING : 이전의 행
CLERK	ADAMS	1100	↑
CLERK	MILLER	1300	① CURRENT ROW 현재 행일 경우
MANAGER	CLARK	2450	↓ FOLLOWING : 다음 나오는 행
MANAGER	BLAKE	2850	↓
MANAGER	JONES	2975	← UNBOUNDED FOLLOWING

현재 연산 작업이 이루어지는 행 (CURRENT ROW) 을 기준으로 위에 위치한 행은 PRECEDING 이라 부르고 위치에 따라서 바로 한 칸 위는 1 PRECEDING 바로 한 칸 아래는 1 FOLLOWING 이라 부른다.

- 맨 위의 행은 UNBOUNDED PRECEDING (무한한 상위 행)이라 부른다.
- 맨 아래 행은 UNBOUNDED FOLLOWING (무한한 하위 행) 이라 부른다.

*** 윈도우 함수 - 집계함수와 ROWS 활용 예시**



#. ROWS의 범위 지정 예시 이해 ★★★★★★★

- 위와 같이 ROWS UnBounded Preceding이라고 시작점만 적으면, Current까지 자동으로 연산됨

- ROWS **UnBounded Preceding** : 맨 위에서부터 Current 까지만 계산
- ROWS **UnBounded Following** : Current부터 맨아래 행까지 포함해서 계산(총합에서 빼나간다고 생각)
- ROWS **1 Preceding** : 한칸 위 행부터 Current 까지 포함해서 계산
- ROWS **2 Following** : Current부터 2칸 아래 행까지 포함해서 계산

#. Rows - 시작 행과 끝 행을 정해주고 싶을 때 예시 이해 ★★★★★★

ROWS BETWEEN A AND B

연산의 시작하는 부분이나 끝나는 부분만 기입한 경우 현재 행 (CURRENT ROW)까지를 기준으로 연산을 실시 한다. 연산의 시작이나 끝이 CURRENT ROW가 아닌 경우가 있다.
예를 들어 SAL 칼럼을 합하는데 현재 행의 한 칸 위부터 아래 행까지 포함을 해서 계산을 하고 싶은 경우 BETWEEN A AND B 를 사용하여 연산의 시작점과 끝점을 지정하여 준다.

```
SELECT  JOB , ENAME , SAL ,
        SUM(SAL) OVER (ORDER BY SAL
                       ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING )
        AS CUME_SAL
FROM EMP;
```

JOB	ENAME	SAL	CUME_SAL
CLERK	SMITH	800	1750
CLERK	JAMES	950	2850
CLERK	ADAMS	1100	3300
SALESMAN	WARD	1250	3600
SALESMAN	MARTIN	1250	3800

#. Range 설명

RANGE

ROWS 는 행의 위치를 기준으로 행을 선택한다면 RANGE 는 칼럼의 '값'을 기준으로 연산에 참여할 행을 선택한다.

```
SELECT  JOB , ENAME , SAL ,
        SUM(SAL) OVER (ORDER BY SAL
                       RANGE 150 PRECEDING )
        AS CUME_SAL
FROM EMP
WHERE JOB = 'CLERK'  OR JOB = 'SALESMAN' ;
```

JOB	ENAME	SAL	CUME_SAL
CLERK	SMITH	800	800
CLERK	JAMES	950	1750
CLERK	ADAMS	1100	2050
SALESMAN	WARD	1250	3600
SALESMAN	MARTIN	1250	3600
CLERK	MILLER	1300	3800
SALESMAN	TURNER	1500	1500
SALESMAN	ALLEN	1600	3100

이때

950 보다 작기는 한데, 그 차이가 150 "이하" 인것들을 의미

={(950-150)<=x <=950}

1500-150 = 1350

#. Range의 범위 지정 예시 이해

- 위와 같이 Range 150 Preceding이라고 시작점만 적으면, 현재까지 **포함 해서 -150까지** 선택

- **Range UnBounded Preceding**
  : **현재 행(포함해서)**을 기준으로 **작은값 모두 선택**해 다 계산

- **Range Between 150 Preceding AND 150 Following**
  : 현재 행보다 -150 "이하" 차이나는 것들부터 +150 "이하" 차이나는 것들 전부 계산

#. Rank( ) 설명 ★★★★★

RANK( )

```
SELECT ENAME , SAL, RANK() OVER (ORDER BY SAL ) AS RANK
  FROM EMP ;
```

정렬

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

RANK() OVER (ORDER BY SAL)
순위출력함수 ~ 대상으로 (윈도우 지정 범위)

동차유지 ← 가독하다

특징!

#. Dense_Rank( ) 설명 ★★★★★

DENSE_RANK( )

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

ENAME	SAL	RANK
KING	5000	1
FORD	3000	2
SCOTT	3000	2
JONES	2975	3
BLAKE	2850	4
CLARK	2450	5
ALLEN	1600	6
TURNER	1500	7
MILLER	1300	8
WARD	1250	9
MARTIN	1250	9

동차유지 ↔ 그냥 Rank() 다음 등수 바로 채움

#. Row_Number( ) 설명 ★★

ROW_NUMBER( )

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

ENAME	SAL	RANK
KING	5000	1
FORD	3000	2
SCOTT	3000	3
JONES	2975	4
BLAKE	2850	5
CLARK	2450	6
ALLEN	1600	7
TURNER	1500	8
MILLER	1300	9
WARD	1250	10
MARTIN	1250	11

중복없이 연속등수로만 증가

#. SUM( ) - 파티션별로 그룹화해서 - ROWS 등 없는 경우 예시

```
SELECT ENAME, JOB, SAL, SUM(SAL) OVER (PARTITION BY JOB ) AS SUM_SAL
FROM EMP;
```

ENAME	JOB	SAL	SUM_SAL
SCOTT	ANALYST	3000	6000
FORD	ANALYST	3000	6000
MILLER	CLERK	1300	4150
JAMES	CLERK	950	4150
SMITH	CLERK	800	4150
ADAMS	CLERK	1100	4150

파티션을 JOB으로 나눠서

= 3000 + 3000 JOB별로 ↓ Sum

= 1300 + 950 + 800 + 1100

[Window 함수 종류 - 그룹 내 행 순서 함수] (이해 : https://youtu.be/4PH9UXVwzn8)

* First_Value 함수

- **파티션 내에서 파라미터의 컬럼에 맨 첫 번째 값으로** 모두 대체시킴

FIRST_VALUE

FIRST_VALUE 함수를 이용해 파티션별 윈도우에서 가장 먼저 나온 값을 구한다.

- SQL SERVER 에서는 지원하지 않는다.

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

맨 ① 2) ② 자기 수행

2) ORDER BY SAL DESC

DEPTNO	ENAME	SAL	ENAME_FV
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

1) PARTITION BY DEPTNO

3) ROWS UNBOUNDED PRECEDING

* Last_Value 함수

LAST_VALUE

LAST_VALUE 함수는 파티션에서 가장 마지막에 나온 값을 구한다.

SQL SERVER에서 지원하지 않는다.

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

① ② ④

이 파티션 안에서 ③ 이러한 각 범위 생성해서

DEPTNO	ENAME	SAL	ENAME_LV
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

그 범위 안에서 Last!

3) ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

* LAG 함수

- **Current의 바로 위에 있는 행을 가져오는 것이다 (<--> LEAD)**

LAG 함수

LAG란 '뒤쳐진다'는 의미를 가진 단어이다. 즉 현재 행으로 이전(위) 에 있는 행을 출력하는 함수이다.
SQL SERVER 에서는 지원하지 않는다.

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

ENAME	HIREDATE	SAL	LAG_SAL
ALLEN	1981-02-20	1600	(null)
WARD	1981-02-22	1250	1600
TURNER	1981-09-08	1500	1250
MARTIN	1981-09-28	1250	1500

- **2칸 위에 있는 애를 찾고싶으면 LAG(컬럼명, 2)**

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

ENAME	HIREDATE	SAL	LAG_SAL
ALLEN	1981-02-20	1600	(null)
WARD	1981-02-22	1250	(null)
TURNER	1981-09-08	1500	1600
MARTIN	1981-09-28	1250	1250

주의

- **NULL 없애고 싶으면 LAG(컬럼, 2, 대체값)**

```
SELECT ENAME , HIREDATE , SAL ,
       LAG(SAL , 2 , 1000 ) OVER (ORDER BY HIREDATE )
FROM EMP
WHERE JOB = 'SALESMAN' ;
```

null값 채워준다

ENAME	HIREDATE	SAL	LAG_SAL
ALLEN	1981-02-20	1600	1000
WARD	1981-02-22	1250	1000
TURNER	1981-09-08	1500	1600
MARTIN	1981-09-28	1250	1250

* LEAD 함수

- **Current의 바로 아래에 있는 행을 가져오는 것이다 (<--> LAG)**
- **LAG와 마찬가지로 파라미터 늘려서 쓸 수 있음!!!**

LEAD 함수

LEAD 란 앞장서다 , 이끌다 라는 뜻을 가지고 있다. LEAD 함수는 현재 행으로 부터 아래 행에 있는 값을 가져오는 함수이다.

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

ENAME	HIREDATE	SAL	LAG_SAL
ALLEN	1981-02-20	1600	1250
WARD	1981-02-22	1250	1500
TURNER	1981-09-08	1500	1250
MARTIN	1981-09-28	1250	(null)

[그룹 내 비율 함수] (https://www.youtube.com/watch?v=4PH9UXVwzn8)

#. Ratio_TO_Report 함수 (비율을 레포트해라)

- 총 합 중에서 현재 행의 비율을 계산 (= 현재 행 / 전체 합)

RATIO_TO_REPORT

RATIO 란 원래 값의 비율을 나타낼 때 사용하는 단어이다. RATIO_TO_REPORT 함수는 값의 비율을 나타내는 함수로 윈도우가 적용되는 공간에서 전체 값의 합 SUM 에서 해당 행이 가지고 있는 값의 비율을 나타낸다.

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

전체에서 현재 행의 SAL 칼럼 값이 어느 정도 비율을 차지하고 있느냐

ENAME	JOB	SAL	RATIO
ALLEN	SALESMAN	1600	0.29 --- 1600 / 5600
WARD	SALESMAN	1250	0.22
MARTIN	SALESMAN	1250	0.22
TURNER	SALESMAN	1500	0.27

#. PERCENT_RANK 함수

PERCENT_RANK 함수

PERCENT 는 파티션 별 윈도우에서 제일 먼저 나오는 것을 0 (상위 0%) , 제일 늦게 나오는 것을 1로 하여 값이 아닌 행의 순서 백분율을 구한다. SQL SERVER에서는 지원하지 않는다.

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

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

값이 같으면 0 %.
가운데 = 50 %
75%
100%

상위 0프로 부터 1(=100프로) 까지 행의 순서별 백분율을 구한다. DEPTNO가 10 인 경우 3개의 행이 있는데 처음 시작은 무조건 0 , 마지막은 1로 처리한다. 따라서 중간에 있는 CLARK는 0.5를 가지게 된다.

2-2-7. DCL

* **DCL과 유저의 권한 <--> (TCL 주의!!!)**
 - DCL : 유저를 생성하고 권한을 제어하는 명령어 (GRANT와 REVOKE)
 - 유저가 사용하는 **모든 DDL 문장은 그에 해당하는 적절한 권한**이 있어야만 가능

* Oracle 기본 유저 종류
 - **SCOTT / TIGER** : 테스트용 샘플 유저 (스콧 / 타이거)
 - **SYS** : DBA ROLE을 부여받은 유저
 - **SYSTEM** **: 모든 권한을 부여받은 DBA 유저** - 설치 시 패스워드를 설정

[Oracle - 권한 부여]

*** 권한 부여**
 - **DBA 권한을 가진 SYSTEM 유저로 먼저** => 다른 유저에게 권한을 주는 방식으로 진행

```
- GRANT Create User To SCOTT;                -- 샘플유저 하나 생성

-  CONN SCOTT
   CREATE USER 유저명 Identified By 비밀번호;

-- 유저가 생성되었으나, 로그인을 하려면 Create Session 권한이 필요

- CONN SCOTT
  GRANT Create Session To 유저명;          -- 세션 권한 부여 (로그인 권한임) ★

- CONN 유저명/패스워드
```

[SQL Server - 권한 부여]

*** 권한 부여**
 - 유저 생성 전, **먼저 로그인을 생성**해야함
 - **로그인을 생성할 수 있는 권한을 가진 로그인은 ' sa '**임

```
- CREATE LOGIN 로그인명 WITH PASSWORD = '비번',
  DEFAULT_DATABASE = 최초 접속할 데이터베이스명;

-- SQL Server에서 유저는 DB마다 존재하므로,
   유저를 생성하기 위해서 생성하고자 하는 DB에 이동 후 처리해야 함

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

[OBJECT에 대한 권한 부여]

*** 오브젝트 권한 ★★★ (<--> ROLE) (오브젝트는 그저 명령어일 뿐임)**

- 특정 오브젝트인 **"테이블, 뷰"등에 대한** Select, Insert, Delete, Update 작업 **명령어를 의미**함
- [Oracle] : 자신이 생성한 테이블 외 다른 유저의 테이블에 접근하려면,
  해당 오브젝트의 권한을 가진 소유자에게 권한을 부여받아야 함

- [SQL Server] :마찬가지임
  차이점은 유저는 **단지 "스키마"에 대한 권한만**을 가짐
  테이블과 같은 오브젝트는 유저가 소유하는게 아니고 **스키마가 소유**하는 것
  유저는 **그 스키마에 대해 권한**을 갖는 것

* A유저가 Tab1에 Select 권한을 갖고있는데, **A의 권한을 B에게도 부여**하고싶다. ★★★

=> GRANT Select ON 테이블명 TO 유저명;

=> GRANT Select ON Tab1 TO B;

[ROLE을 이용한 권한 부여 ★]

*** DBA의 역할**

- **각 유저별로 어떤 권한이 있는지 관리해야함**
- **DBA는 ROLE을 생성**하고, **ROLE에 각종 권한을 부여한 후**, **다른 ROLE이나 유저에게 부여**

*** ROLE (<-->오브젝트 주의) (ROLE은 권한 그룹임)**

- **다양한 권한을 그룹으로 묶어 관리, 사용자와 권한 사이에서 중개 역할 수행** ★★★★★★
- **시스템 권한, 오브젝트 권한 모두 부여 가능**
- ROLE은 유저에게 **직접 부여**될 수도 있고
  다른 ROLE에 포함되어 유저에게 부여될 수도 있다.

```
[Oracle]
CONN SYSTEM/MANAGER
CREATE ROLE 롤이름;                          -- 롤 생성

GRANT 부여권한, 부여권한, ... TO 롤이름;      -- 롤에 권한 부여
GRANT 롤이름 TO 유저명;                       -- 롤 권한 집합을 유저에게 부여
```

* Oracle에서 제공하는 ROLE 종류

- **CONNECT** : Create Session과 같은 "로그인 권한"
- **RESOURCE** : Create Table과 같은 "오브젝트(=리소스) 생성 권한"

[유저 삭제 명령어와 권한]

* 유저 삭제 명령어

- **DROP USER 유저명 CASCADE;**
  => **CASCADE 옵션** : **해당 유저가 생성한 오브젝트를 먼저 삭제** 후 유저를 삭제

2-2-8. 절차형 SQL

* 절차형 SQL
  - 분기, 반복이 가능한 모듈화된 프로그램, DBMS에서 직접 실행됨
  - **프로시저(Procedure), 사정함(User Defined Function), 트리거(Trigger)**
  - **[Oracle] PL/SQL**
  - **[SQL Server] T-SQL**

  - 절차형 SQL을 이용하면 SQL문의 연속적인 실행이나 조건에 따른 분기처리를 이용해 특정 기능을 수행하는 **"저장 모듈"을 생성**할 수 있음
  - PL/SQL은 여러 SQL문장을 Block으로 묶고 한번에 BLock 전부를 서버로 보내기 때문에 통신량을 줄여서 응용프로그램의 성능을 향상시킴

*** 프로시저(Procedure)**
  - 주로 DML을 사용해 주기적으로 진행해야되는 작업을 저장
  - 별도의 호출을 통해 실행

  - **CREATE OR REPLACE Procedure 문으로 프로시저를 생성**

    => OR Replace : 기존에 같은 이름의 프로시저 있으면 **무시하고 새로운 내용으로 덮어씀**

  - **작업의 결과를 DB에 저장 (=> 트랜잭션 처리)** ★★★ **(<--> 사용자 정의함수는 결과 리턴)**
  - BEGIN ~ END 문 사이에 작업 영역 생성 (1~3은 섞어서 사용하고, 4번만 마지막에 작성)
    -> 1. 조건/반복 영역
    -> 2. SQL을 사용해 데이터 관리하는 영역
    -> 3. 예외 처리 영역
    -> 4. 트랜잭션 영역 (작업 결과를 실제로 반영하거나 취소하는 영역)

  - **프로시저 내부의 절차적 코드는 PL/SQL 엔진이 처리** ★
    (<--> 일반적인 SQL 문장은 SQL실행기가 처리)

*** 프로시저 실행**
  - **EXECUTE** 프로시저명(파라미터);
  - **EXEC** 프로시저명(파라미터); **CALL** 프로시저명(파라미터);

*** 프로시저 삭제**
  - **DROP Procedure** 프로시저명;

* **[Oracle] 프로시저 PL/SQL 예시**

```
CREATE OR REPLACE PROCEDURE 매출마감 (마감일 IN CHAR(8))
IS
BEGIN
    매출총액 NUMBER;
    SELECT SUM(매출액) INTO 매출총액 FROM 판매내역 WHERE 판매일 = 마감일;

    EXCEPTION
        WHEN NO_DATA_FOUND THEN 매출총액 = 0;
    INSERT INTO 마감내역 (판매일, 판매총액) VALUES (마감일, 매출총액);

    COMMIT;
END;
```

```
EXECUTE 매출마감('20200908');

DROP PROCEDURE 매출마감;
```

<- 삭제의 경우 파라미터 없음

* **[SQL Server] 프로시저 T-SQL 예시**

```
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;
```

* 프로시저에서 DDL 사용 방법

- **execute immediate 'DDL' 문**을 사용 ★★ (실행한다 즉시 DDL문을)

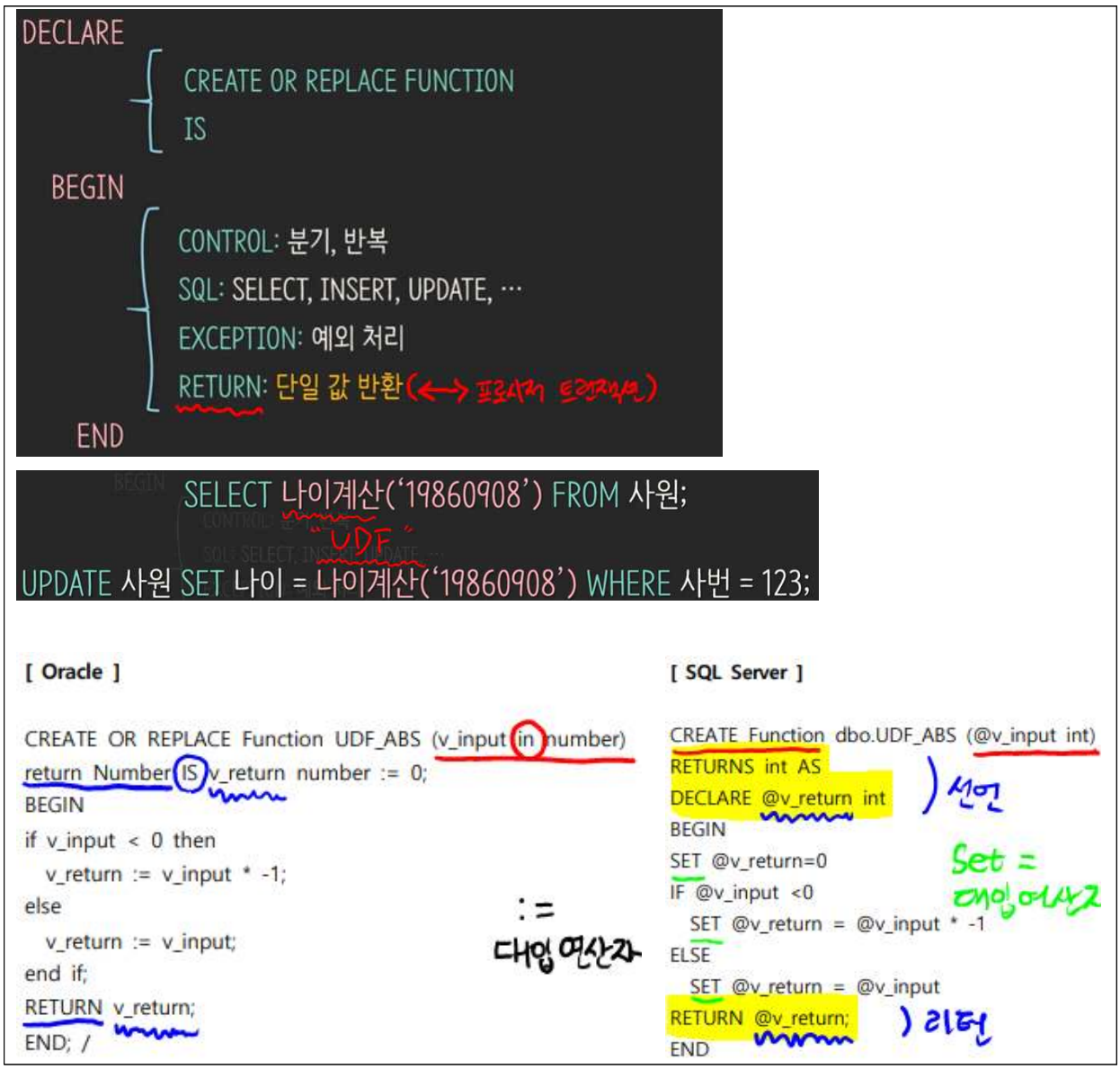
```
CREATE OR REPLACE PROCEDURE  insert_dept authid
AS
BEGIN
  EXECUTE IMMEDIATE  'TRUNCATE TABLE DEPT';
  INSERT /*+ APPEND */ INTO DEPT
          (DEPTNO, DNAME, LOC)
  SELECT  DEPTNO, DNAME, LOC
  FROM  TMP_DEPT;
  COMMIT;
END;
```

*** 사용자 정의 함수 (UDF)**

- 프로시저와 구조가 비슷
- **함수 호출 시 특정 값을 돌려받을 수 있음 => 리턴값(반환값)**
- **작업 결과를 호출한 쿼리문에 돌려줌★★★★ (<--> 프로시저는 DB에 저장)**

* 사용자 정의 함수 호출

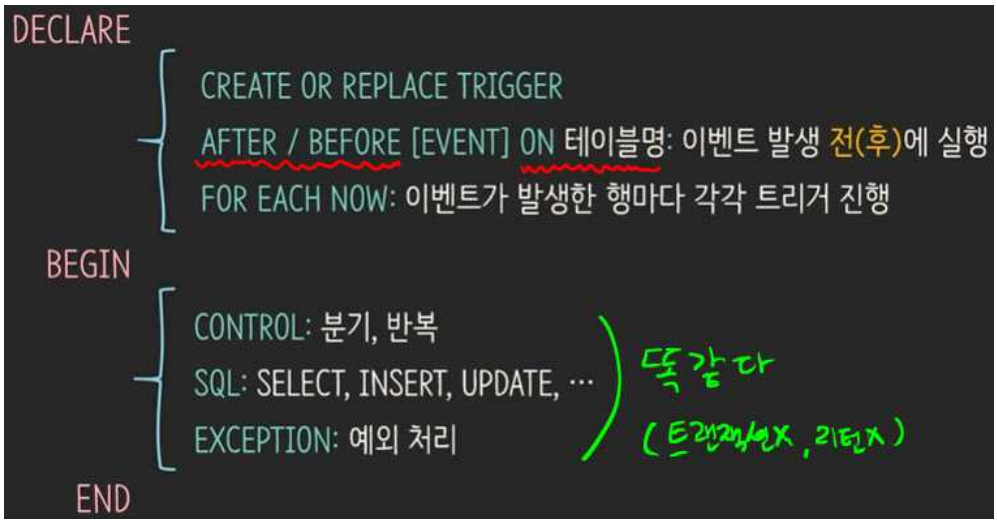
- 일반적인 집계함수처럼 호출 가능 => Select문, Update문 등



* 트리거 (Trigger)

- **삽입, 삭제, 수정 등으로 데이터베이스에 변화(=이벤트)가 발생하는 경우, 자동 호출**
- **INSERT, UPDATE, DELETE 등 "DML"문에 의해 자동 수행 ★★★★**
- **무결성을 유지**하거나, **로그(작업내용)을 저장**할 때 사용 **★★**
- 자동으로 실행되기 때문에 리턴값, **매개변수 없음**, **커밋 등도 없음 ★★ (TCL로 트랜잭션 제어 X)**

- Declare 영역을 **"특정 시점 전(Before)" / "특정 시점 후(After)"**로 지정
  변화가 있는 행들에 **각각 적용될 수 있도록 "Each For Row"**도 지정 가능



* 기존 데이터 사용 = OLD.나이 　　* 신규 데이터 사용 = NEW.나이

[Oracle]

```
CREATE OR REPLACE Trigger SUMMARY_SALSE      → 트리거 선언
AFTER INSERT ON ORDER_LIST FOR EACH ROW      → 레코드가 입력되면 트리거 발생
                                               ORDER_LIST에 이벤트 발생 시

DECLARE
o_date ORDER_LIST.order_date&TYPE;           → 변수 선언 : 주문일자, 주문상품 값을 저장
o_prod ORDER_LIST.product&TYPE;

BEGIN
o_date := NEW.order_date;                    → 변수에 신규로 입력된 데이터 저장
o_prod := NEW.product;                         ( :NEW 구조체 )

UPDATE SALES_PER_DATE                        → 주문일자의 상품레코드 존재하면 +후 업뎃
SET qty = qty + :NEW.qty,  amount = amount + :NEW.amount
Where sale_data = o_date AND product = o_prod;

if SQL % NOTFOUND then                       → 주문일자의 상품레코드 존재X면 새로 입력
  INSERT INTO SALES_PER_DATE VALUES(o_date, o_prod, :NEW.qty, :NEW.amount);
END;
```

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

*** 프로시저와 트리거 차이점**

프로시저	트리거
CREATE Procedure 문법 사용	CREATE Trigger 문법 사용
EXECUTE (EXEC, CALL) 로 호출	생성 후 자동 실행
COMMIT, ROLLBACK (O)	COMMIT, ROLLBACK (X)
매개변수 있음	매개변수 없음 (자동 호출)

2-3. SQL 최적화 기본 원리

2-3-1. 옵티마이저와 실행계획

* 옵티마이징과 옵티마이저
 - 옵티마이징 = 최적화 한다
 - 옵티마이저 = 옵티마이징을 수행하는 놈 -> 성능을 가장 유리한 방향으로 이끄는 역할 수행
 - 즉, **최적의 실행방법, 실행계획(Execution Plan)을 짠다**
 - **옵티마이저도 가끔 실수로 잘못된 실행계획을 짜기도 함**
   -> **'오라클 힌트'를 사용**하여 옵티마이저가 올바른 실행계획으로 도움줄 수 있도록 해야한다
 - **동일 SQL문에 대해 실행 계획이 달라도 쿼리의 실행 결과는 항상 같아야함!!!** ★★★★★

*** 실행 계획 구성 요소 ★★**
 - **조인 순서★** (Join Order)
 - **조인 기법(=조인 방법★)** (Join Method)
 - **액세스 기법(=액세스 방법★)** (Access Method)
 - **최적화 정보** (Optimization Information) : Cost, Card, Bytes
 - **연산** (Operator)
 - **질의 처리 예상 비용 ★** (Cost) **(--> 실행시간은 알 수 없음! ★)**

[옵티마이저 종류] (면접용)

*** 1. 로지컬 옵티마이저**
 -> Query Transformation을 수행하면서 우리가 짠 쿼리를 여러 형태로 변환(결과는 똑같은 쿼리)

*** 피지컬 옵티마이저**
 -> 로지컬 옵티마이저가 짠 쿼리의 비용을 비교해서 가장 저렴한 쿼리를 고름
 -> 이때, **2. Cost Estimator** = 비용 계산하는 장치
 3. Plan Generator = 비용 계산을 위해 실행계획을 도출하는 장치

[옵티마이저가 최적의 실행방법을 결정하는 방식]

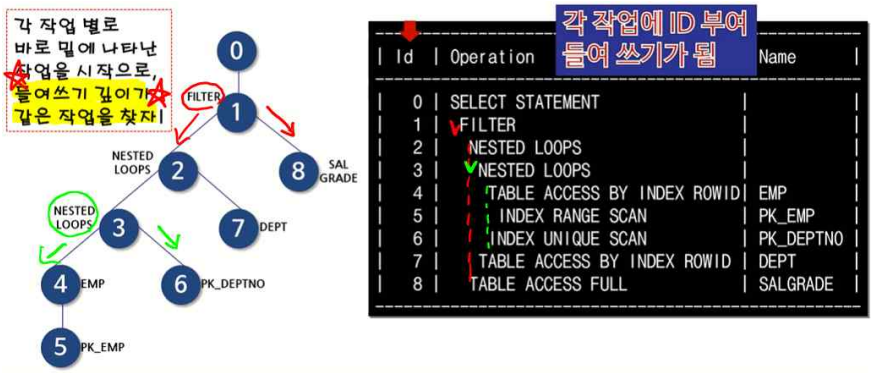
* **규칙 기반** 옵티마이저
 - **규칙(우선순위)을 기반으로 실행계획을 생성**
   -> 실행계획을 생성하는 규칙을 이해한다면 누구나 실행계획을 비교적 쉽게 예측 가능하다

 - **규칙 기반 우선순위 ★★★★★**
   -> **행에 대한 고유주소 액세스 방식 (=Single row by rowid★) > 인덱스를 이용한 엑세스 방식**
   인덱스를 이용한 엑세스 방식 > 전체 테이블 액세스 방식
   -> **이용가능한 인덱스가 있으면 항상 인덱스를 사용하는 규칙 계획 생성★★★**

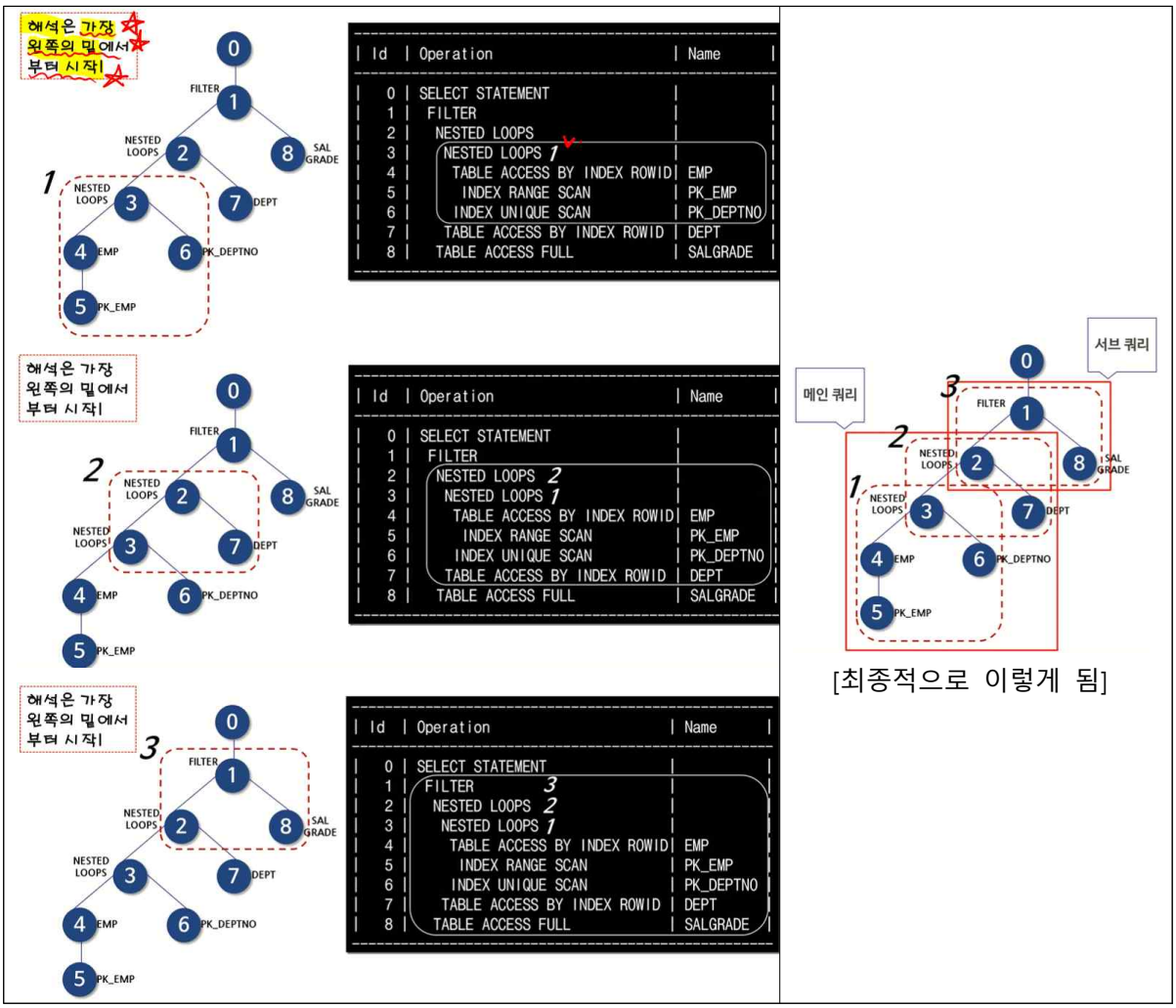
 - **조인 순서 결정 시** => **조인 컬럼 인덱스의 존재 유무**가 판단 기준 ★
   -> 조인 컬럼에 대한 인덱스가 양쪽에 존재 : 우선순위 높은 테이블이 먼저 수행(Driving)
   -> 한쪽만 인덱스 존재 : **인덱스 없는 테이블이 선행**
   -> 둘 다 인덱스 없음 : **FROM절의 뒤에 나열된 순서로** 테이블이 선행
   -> 우선순위가 동일 : FROM절에 나열된 테이블의 역순으로 선행 테이블 선택

* **비용 기반** 옵티마이저 ★★
  - **비용(예상되는 소요시간, 자원 사용량 ★★★)이 가장 적은 실행계획을 선택**하는 방식
    -> **비용에 따라 Full Scan이 유리하다고 판단할수도 있음!!!** ★★★
  - 규칙기반 옵티마이저 단점 극복을 위해 출현
  - 다양한 객체 통계정보와 시스템 통계정보 등을 활용
  - **질의 변환기** : 사용자가 작성한 SQL문을 처리하기에 보다 용이한 형태로 변환하는 모듈
  - **대안 계획 생성기** : 동일한 결과를 생성하는 다양한 대안 계획을 생성하는 모듈
    -> 대안계획 : 연산 적용순서 변경, 연산 방법변경, 조인순서 변경 등으로 생성
  - **비용 예측기** : 생성된 대안 계획의 비용을 예측하는 모듈
    -> 모든게 다 정확해야 함(계산식, 예측, 분포도 등)

#. 실행계획 분석



#. 실행계획 작업 순서 해석 - 큰 틀 (https://youtu.be/01VbLThCcm4)

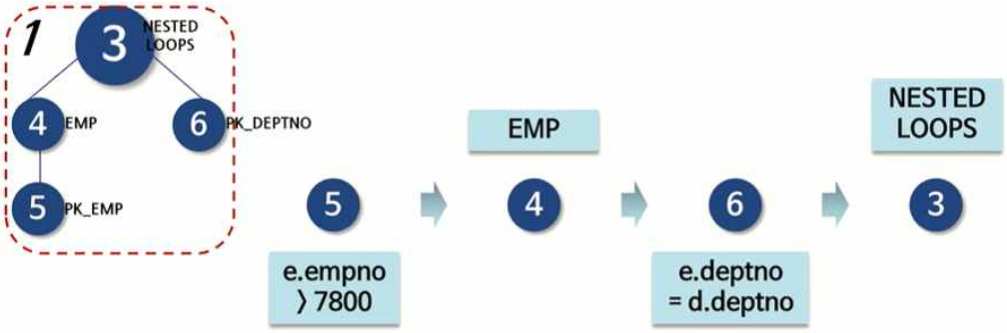


[최종적으로 이렇게 됨]

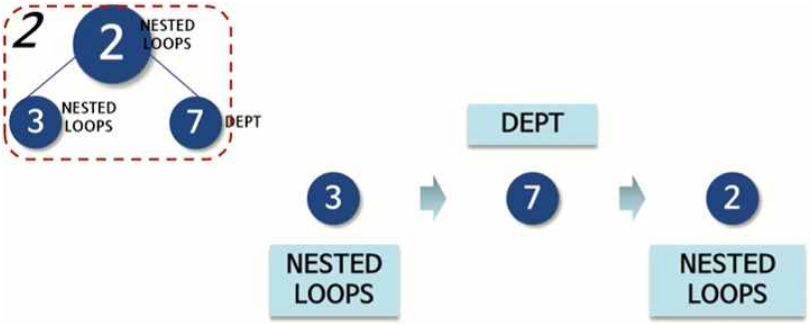
#. 실행계획 - 작업 순서 이어서 설명 (https://youtu.be/01VbLThCcm4)

```
-----------------------------------------------------
| Id  | Operation                    | Name         |
-----------------------------------------------------
|   0 | SELECT STATEMENT             |              |
|   1 |  FILTER                      |              |
|   2 |   NESTED LOOPS 2             |              |
|   3 |    NESTED LOOPS 1            |              |
|   4 |     TABLE ACCESS BY INDEX ROWID| EMP        |
|   5 |      INDEX RANGE SCAN        | PK_EMP       |
|   6 |     INDEX UNIQUE SCAN        | PK_DEPTNO    |
|   7 |    TABLE ACCESS BY INDEX ROWID | DEPT       |
|   8 |   TABLE ACCESS FULL          | SALGRADE     |
-----------------------------------------------------
```

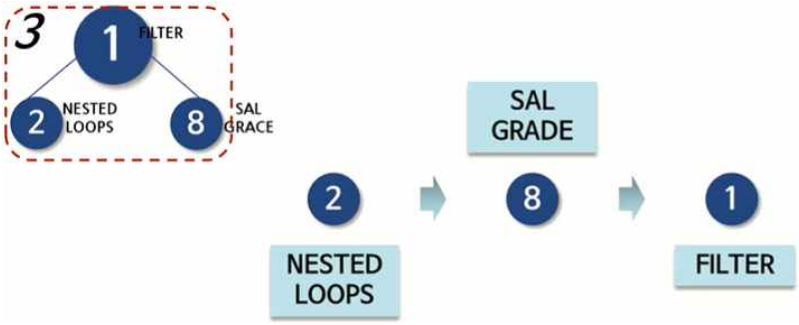
[1]의 경우, JOIN절인데, JOIN절의 최종 결과 = (3)번임



[2]의 경우, [1]의 결과인 (3)번이 먼저 실행



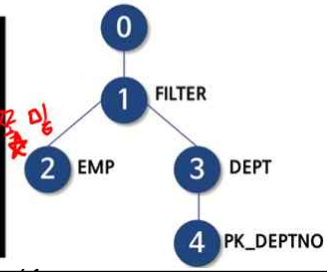
[3]의 경우, 서브쿼리인데, [2]의 결과인 (2)번이 먼저 실행



#. 실행 계획 주의사항 ★★★★ => <u>들여쓰기가 항상 우선순위가 아님을 주의!!!</u>

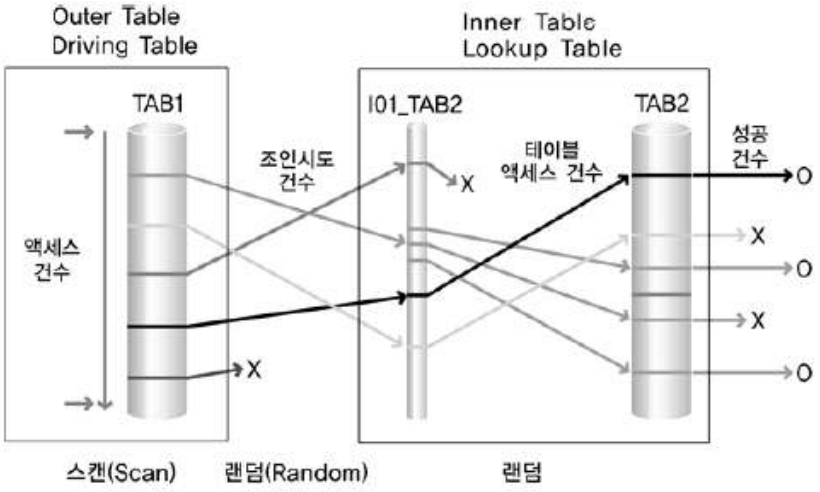
다음의 실행계획에서 가장 먼저 실행되는 작업은? ✗ 2

```
-----------------------------------------------------
| Id  | Operation                    | Name         |
-----------------------------------------------------
|   0 | SELECT STATEMENT             |              |
|   1 |  FILTER                      |              |
| ➡ 2 |   TABLE ACCESS FULL          | EMP          |
|   3 |   TABLE ACCESS BY INDEX ROWID| DEPT         |
|   4 |  → INDEX UNIQUE SCAN         | PK_DEPTNO    |
-----------------------------------------------------
```



#. SQL 처리 흐름도 (Access Flow Diagram)

- **SQL 내부적 처리 절차를 시각적으로 표현**한 것 ★★★ **(--> 실행시간은 알 수 없음! ★)**
- **인덱스 스캔, 테이블 전체 스캔** 등과 같은 **액세스 기법(액세스 방법)을 표현** ★★★
- **성능적인 측면도 표현**할 수 있다 ★★★



[그림 Ⅱ-3-5] SQL 처리 흐름도

2-3-2. 인덱스 기본

* 인덱스 (링크 : https://youtu.be/uO8tL0okg7Q)
  - 일종의 오브젝트(Object)임
  - 인덱스가 생성되면 테이블과 매핑된 또다른 테이블(=인덱스)가 생성된다 라고 생각하면 쉬움
    -> 그 테이블은 **인덱스 컬럼을 기준으로 Sorting**되어 저장되기 때문에 검색 시 **매우 빠름**
    -> 인덱스로 먼저 원하는 데이터를 찾은 후, 기존 테이블 매핑된 곳 가서 데이터를 꺼내는 방식

  - 주로 **Where절, Order By절에서 자주 쓰이는 컬럼을 인덱스로 지정**

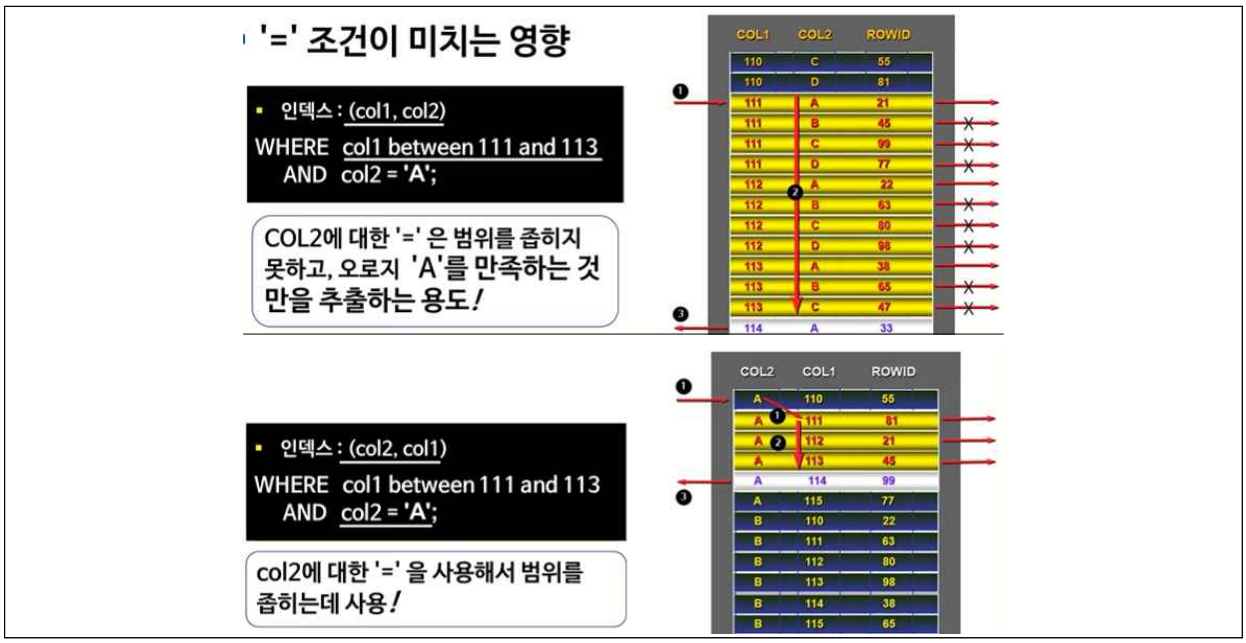
  - **Select 속도 증가**
  - **Insert, Update 속도 저하 (데이터 삽입 시 인덱스도 조정해야 하기 때문에!!)**
    -> INSERT, DELETE작업과 다르게 UPDATE 작업에선 부하가 없을 수도 있다. (몇개만 바꾸면 됨)★★★

* 블록(Block)의 개념
  - 인덱스는 해당 테이블의 블록에 주소를 가지고 있음
  - **블록(Block)** : 데이터가 저장되는 최소 단위
    테이블의 데이터들이 "Row 행"단위로 저장되어 있음

*** 랜덤 액세스**

- 인덱스를 스캔하여 테이블로 데이터를 찾아가는 방식 => 랜덤 액세스 부하가 큼
- **매우 많은 양의 데이터를 읽을 경우, 인덱스 스캔보다 Full 스캔이 더 효율적 (헷갈림주의)**★★★

* 인덱스 종류
 - **단일 인덱스 = 단일 컬럼으로 구성 O**
 - **결합 인덱스 = 여러 컬럼을 묶어 결합 인덱스로 구성 O**
  -> **결합하는 컬럼의 순서가 매우 중요 ★★★**
  => **인덱스 생성 시 아래 컬럼을 더 앞 순서에 배치 ★★★**
  1. **'=' 조건(동등조건)으로 많이 쓰는 컬럼이 앞에 오는게 좋음 ★★★**
  2. **'BETWEEN'처럼 범위 지정하는 컬럼이 그 다음으로 오는게 좋음 ★★★**
  3. **'id'처럼 분별력 높은 컬럼이 그 다음으로 오는게 좋음 ★★★**



* 인덱스 스캔 방식 종류
 - index full scan
 - index range scan : 특정 범위를 스캔
 - index skip scan, index fast full scan 등

#. 인덱스 생성 예제

CREATE INDEX 인덱스명 ON 테이블명 (인덱싱할 컬럼);

```
CREATE INDEX IDX_SB1 ON STARBUCKS_ORDER (REG_NAME);
```

INDEX_OWNER	INDEX_NAME	UNIQUENESS	STATUS	INDEX_TYPE	TEMPORARY	PARTITIONED	FUNCIDX_STATUS	JOIN_INDEX	COLUM
MINA	IDX_SB1	NONUNIQUE	VALID	NORMAL	N	NO	(null)	NO	REG_N

#. 인덱스 명시적으로 사용하기

 - 이렇게 안해도 기본적으로 인덱스 사용함, BUT **명시적으로** 해주는 방법은 아래와 같다
 - SELECT **/*+INDEX(테이블명 인덱스명) */** *
  FROM 테이블명
  select 할 때 이 인덱스를 사용해라!

```
SELECT /*+INDEX(STARBUCKS_ORDER IDX_SB1) */ * FROM STARBUCKS_ORDER WHERE REG_NAME = '봉';
```

#. 인덱스 삭제 예제

DROP INDEX 인덱스명;

```
DROP INDEX IDX_SB1;
```

[인덱스 종류]

*** 트리 기반 인덱스 (= B-tree 인덱스로 생각해도 될 듯) ★★★**

- [Oracle] [SQL Server]
- **RDBMS에서 가장 일반적인 인덱스는 B-tree 인덱스 ★**
- **'='**로 검색하는 **일치 검색(Exact Match)**, **'Between', '>'**로 검색하는 **범위 검색(Range) 둘 다 적합★**

- 각각의 노드 = 블록이라고 칭함 (루트 블록, 브랜치 블록, 리프 블록)
- **브랜치 블록(인터널 블록) ★**
  -> 하위 단계의 블록을 가리키는 **포인터**를 가짐
- **리프 블록 ★**
  -> 인덱스를 구성하는 컬럼의 데이터 + 해당 데이터에 대한 행의 위치를 가리키는 레코드 식별자
  -> **양방향 링크(Double Link)**를 가진다 (여기선 B+tree 구분 안함)

*** 클러스터형 인덱스**

- ~~[Oracle]~~ **[SQL Server]**
- 인덱스는 저장 구조에 따라 클러스터형 인덱스와 비클러스터형 인덱스로 구분
- **ORACLE의 IOT와 매우 유사 ★**

- 인덱스의 **리프 페이지 = 데이터 페이지 ★★★**
  -> 즉, 테이블 탐색에 필요한 레코드 식별자가 리프 페이지에 없음
  -> 인덱스 키 컬럼과 나머지 컬럼을 리프에 같이 저장 => 테이블 랜덤 액세스 할 필요가 X)
  -> **클러스터형 인덱스의 리프 페이지를 탐색**하면 **해당 테이블의 모든 칼럼 값 바로** 얻음!!

- **리프 페이지의 모든 로우(=데이터)는 인덱스 키 컬럼 순으로 물리적 정렬됨 ★★★**
  -> 테이블 로우는 물리적으로 한 가지 순서로만 정렬될 수 있음
  -> 즉, 클러스터형 인덱스는 테이블당 한 개만 생성할 수 있음

*** 비트맵 인덱스 (BITMAP Index)**

- 질의 시스템 구현 시에 모두 알 수 없는 경우인 **DW 및 AD-HOC 질의 환경을 위해 설계 ★★★**
- **하나의 인덱스 키 엔트리**가 **많은 행에 대한 포인터를 저장 ★★**

*** 전체 테이블 스캔 (Full Scan)**

- 테이블에 존재하는 모든 데이터를 읽으면서 조건에 맞는 결과면 추출, 아니면 버리는 방식
- 시간이 매우 오래걸림
- Orcle에선 테이블의 고수위 마크(HWM) 아래의 모든 블록을 읽는다.
  -> 고수위 마크 : 테이블 가장 위에 있는 데이터

* 옵티마이저가 연산으로써 Full Scan을 선택하는 이유

- SQL문에 조건이 존재하지 않는 경우
- SQL문의 주어진 조건에 사용 가능한 인덱스가 존재하지 않는 경우
- 옵티마이저의 취사 선택 (= 조건을 만족하는 데이터가 多 -> 대부분 블록 액세스해야됨을 판단)
- 그밖에 병렬처리 방식으로 처리하는 경우
- Full Table 스캔 힌트를 사용한 경우

* 인덱스 스캔
 - 인덱스의 리프 블록 = 인덱스를 구성하는 컬럼 + 레코드 식별자
   -> 검색 시 인덱스 리프블록을 읽으면 이 두 값을 알 수 있다

 - 인덱스는 인덱스 구성 컬럼의 순서로 정렬됨
   ex) 인덱스 구성 컬럼이 A+B라면 => A컬럼으로 먼저 정렬, A 값이 동일하면 B컬럼으로 정렬

 - **인덱스 유일 스캔**
   -> **유일(Unique) 인덱스**를 사용하여 **단 하나의 데이터를 추출**하는 방식
   -> 유일인덱스는 **중복을 허락하지 않는** 인덱스임!
   -> 유일 인덱스 구성 컬럼에 모두 '='로 값이 주어지면 **결과는 최대 1건**

 - **인덱스 범위 스캔**
   -> 인덱스를 이용해 **한 건 이상의 데이터 추출**하는 방식
   -> 유일 인덱스로 값을 못구하면 ->**비유일 인덱스를 이용**하는 모든 액세스 방식은 이것을 사용

 - **인덱스 역순 범위 스캔**
   -> 리프 블록의 **양방향 링크(double link)를 이용**
   -> **"내림차순"**으로 데이터를 읽음 => **최대값**을 쉽게 찾을 수 있음

*** Full 스캔 VS 인덱스 스캔 비교 ★★★**
 - FULL 스캔 : 인덱스 존재 유무와 상관없이 항상 이용 가능한 방식
   -> 비효율적, 여러 블록씩 읽음, "테이블 대부분/전체 데이터 찾을 땐 유리"

 - INDEX 스캔 : 사용 가능한 **적절한 인덱스가 존재할 때만** 이용 가능한 방식 ★★
   -> 레코드 식별자 이용, 정확한 위치를 알고 읽음 => **한번의 I/O요청에 한 블록씩**

 - 옵티마이저는 **인덱스가 존재하더라도, 경우에 따라 전체 테이블 스캔 방식을 선택 가능** ★★

2-3-3. 조인 수행 원리 <면접용 공부>

* 조인 종류 (이해 : https://youtu.be/SVD5ldwVYpo)

- NL Join, Sort Merge Join, Hash Join

*** Nested Loop JOIN (NL Join)**

- **2 중첩 for문과 같은 원리**
- ex) 두 개의 테이블이 있다고 가정해보자 (IDOL_GROUP , IDOL_MEMBER) <참고로 1:M관계임>
  -> **IDOL_GROUP을 Outer Table(=Driving Table?), IDOL_MEMBER를 Inner Table**이라고 함
  -> 각 그룹에 어떤 멤버가 있는지 알고싶다면
  1. IDOL_GROUP의 소시 선택 => 윤아, 태연, 티파니, 수영 ... 전부 꺼냄
  2. IDOL_GROUP의 2NE1 선택 => 박봄, 산다라, ... 전부 꺼냄
  3. 이런 식으로 두 테이블을 다 Full Scan하는 것
- 따라서 매우 비효율적, 오래걸림
- 조인 조건의 **인덱스 유무에 영향을 받음!!!** **(<--> Sort Merge) ★★★**
- **유니크 인덱스를 활용 ★**
- **OLTP 환경 (= 온라인 환경)의 쿼리에 적절 ★★**
- **조인 컬럼에 적당한 인덱스가 없어서 자연조인(Natural join)이 효율적일 때 유리 ★★**
- **Outer Table (=Driving Table?)이 성능에 매우 중요한 요인 ★★★**
- **예시가 1:M이라고 했는데, 1에 해당하는 테이블이 소량의 데이터를 가진 경우 적용하면 성능 향상**

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.

#. 참고사항

```
[SQL]
SELECT *
FROM DEPT D
WHERE D.DEPTNO = A001
AND EXISTS
  (SELECT X FROM EMP
   WHERE D.DEPTNO = E.DEPTNO);
```

[DEPT 테이블 INDEX 정보]
PK_DEPT : DEPTNO
[EMP 테이블 INDEX 정보]
PK_EMP : EMPNO
IDX_EMP_01 : DEPTNO

③ NESTED LOOPS ANTI JOIN	=> 서브쿼리 앞이 NOT EXISTS일 경우 "안티" ★★
NESTED LOOPS SEMI JOIN	=> 서브쿼리 앞이 EXISTS일 경우 "SEMI" ★★

★★★★★★★★

NL	SORT Merge	Hash
OLTP 환경	PGA영역	DW 환경
	Non-EQUI JOIN	EQUI JOIN
인덱스 유무에 영향 O	인덱스 유무에 영향 X	정렬 작업 필요X 대량 데이터 배치작업에 유리
Outer Table 중요	Outer Table 중요 X	Outer Table 중요

*** Sort Merge JOIN**

- NL Join과 비슷하게 2 중첩 for문과 같은 원리
- 차이점 : **두 테이블을 "조인 컬럼을 기준으로 데이터를 정렬시킨 후" JOIN시키는 방식**
- **넓은 범위의 데이터를 처리할 때 주로 이용**
- 조인 조건의 **인덱스 유무에 영향 받지 않음!!!** ★★★
- **Outer Table (Driving Table) 필요없음!!!!!! (<--> NL, Hash)** ★★★
- 비동등 조인에 대해서도 조인 가능

- Inner Table에 적절한 인덱스가 없어서 NL JOIN을 쓰기에 너무 비효율적일 경우 사용
- **Range Scan 쿼리에서 적절 (정렬되어 있으니까)**
- **Table Random Access 발생 X => 경합 발생 X => 성능 유리**
- **PGA영역에서 Sorting이 수행 => 경합 살생 X => 성능 유리**

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**

- NL Join의 랜덤 액세스 문제점 해결하기 위해 등장
- **정렬할 작업이 필요없어 정렬이 부담되는 대량 배치작업에 유리** ★
- **Outer Table (Driving Table)이 성능에 매우 중요한 요인** ★★★
- 배치에서 쓰면 좋은 수행 원리이다
- 대용량 테이블을 JOIN할 때 사용하면 좋다
- **인덱스가 존재하지 않는 경우에 사용 가능**
- **'='로 수행하는 Equal JOIN (동등조인)만 가능 (<--> NL, Sort Merge)** ★★★
- **DW 환경 등에서 데이터를 집계하는 업무에 많이 사용** ★★
- 해시 테이블의 key 컬럼에 중복값이 없을수록 성능에 유리
- PGA영역에서 수행되기 때문에 매우 빠름

- Table Random Access 발생 X
- Hash 영역에 들어갈 테이블 사이즈가 충분히 작아야 성능 유리
- 수행 빈도가 높은 OLTP 환경에서 수행하면, 오히려 CPU나 메모리 사용량 증가

- ex (IDOL_GROUP , IDOL_MEMBER) 에서 MEM테이블의 데이터가 너무 크다면
  GROUP 테이블을 Build Input으로 삼아서 Hash영역에 저장해둠
  -> GROUP이 Hash영역에 있으면서 MEM이 조인되는 원리

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