SQLD

1️⃣ 데이터 모델링의 이해

제1장. 데이터 모델링의 이해

1) 데이터 모델링

데이터 모델링

- 정보시스템을 구축하기 위한 데이터 관점의 업무 분석 기법
- 현실세계의 데이터(what)에 대해 약속된 표기법에 의해 표현하는 과정
- DB를 구축하기 위한 분석/설계의 과정

데이터 모델링 3요소

1. 어떤 것 (Things) 2. 성격 (Attributes) 3. 관계 (Relationships)

데이터 모델링 특징

- (현실세계) => **추상화, 업무분석및 업무형성화, 단순화, 정확화** => (모델)

모델링의 관점

1. 데이터 관점 2. 프로세스 관점 3. 상관 관점

데이터 모델링 주요 이유

- 업무정보 구성하는 기초가 되는 정보들에 대해 **일정한** 표기법에 의해 표현

  => 정보시스템 구축의 대상이 되는 업무 내용을 정확하게 분석하는 것
- 분석된 모델을 가지고 실제 데이터베이스를 생성하여 개발 및 데이터관리에 사용하기 위한 것

  => 데이터모델링 **자체**로서 업무 설명, 분석하는 부분에서도 매우 중요한 의미 가짐7

데이터 모델링 유의점

1. **중복** (Duplication) 최소화

   여러 장소에 같은 정보 저장 X
2. **비유연성** (Inflexibility) => 유연성 높이기

   데이터 정의를 데이터의 사용 프로세스와 **분리**

   => 데이터/프로세스의 작은 변화가 애플리케이션과 데이터베이스에 중대한 **변화** 일으킬 가능성 줄임
3. **비일관성** (Inconsistency) => 일관성 유지, 연계성 낮추기

   데이터와데이터 간의 **상호 연관 관계** 명확하게 정의

   연계성 높이는 것은 데이터 모델이 업무 변경에 대해 취약하게 만드는 단점.

데이터 모델링 개념 => 개논물

- **개념적** 데이터 모델링

  추상화 수준 높음, 업무중심적, **포괄적인** 수준의 모델링 진행

전사적 데이터 모델링, **EA** 수립 시 많이 이용,

- **논리적** 데이터 모델링

  시스템으로 구축하고자 하는 업무에 대해 **Key, 속성, 관계** 등을 **정확**하게 표현

  재사용성 높음

- **물리적** 데이터 모델링

  실제로 데이터베이스에 이식할수 있도록 **성능, 저장** 등 **물리적인** 성격을 고려하여 설계

---

2) 스키마, 독립성, 사상

데이터베이스 스키마 구조 3단계 => 외개내, 개통물

- **외부 스키마** (External Schema) => **사용자 관점**

  개개 사용자가 보는 **개인적** DB 스키마

- **개념 스키마** (Conceptual Schema) => **통합 관점**

  모든 사용자 관점을 통합한 **전체** DB

- **내부 스키마** (Internal Schema) => **물리적 관점**

  물리적 장치에서 데이터가 **실제적** 저장

스키마 3단계 구조 나누는 이유

- DB에 대한 사용자들의 관점과 실제 표현되는 물리적인 방식 분리하여 **독립성을 보장**하기 위함

데이터 독립성

- **논리적** 독립성: **개념** 스키마 변경, 외부 스키마에 영향 X
- **물리적** 독립성: **내부** 스키마 변경, 외부/개념 스키마에 영향 X

Mapping (사상)

- 상호 독립적인 개념을 연결시켜주는 다리
- **논리적 사상**: **외부** 스키마 = 개념 스키마
- **물리적 사상**: 개념 스키마 = **내부** 스키마

---

3) ERD

데이터 모델 표기법

- 1976년 피터첸이 Entity Relationship Model(E-R 모델) 개발
- **IE**, Baker 기법 많이 쓰임
- 엔터티, 관계, 속성으로 이뤄짐

좋은 데이터 모델의 요소

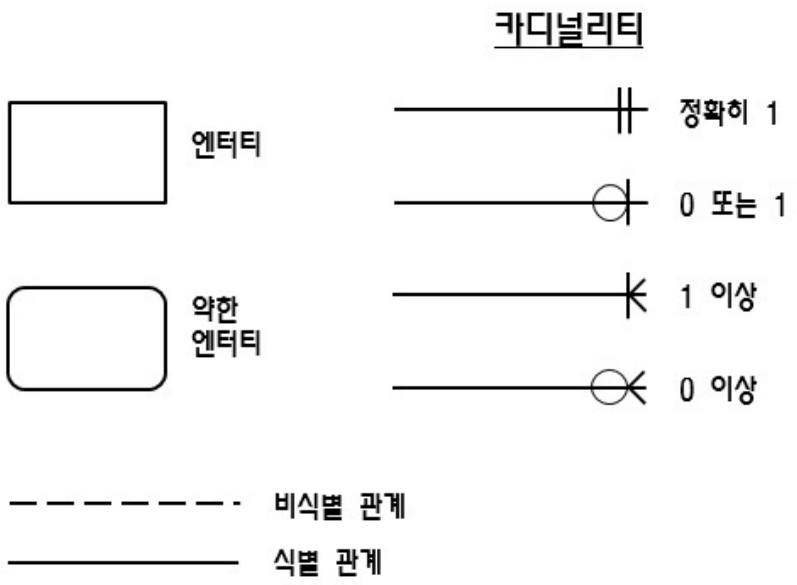
1. **완전성**: 업무에 필요한 모든 데이터가 모델에 정의
2. **중복배제**: 하나의 DB내에 동일한 사실은 한번만
3. **업무규칙**: 많은 규칙을 사용자가 공유하도록 제공
4. **데이터 재사용**: 데이터가 독립적으로 설계되어야 함

5. **의사소통**: 업무규칙은 엔터티, 서브타입, 속성, 관계 등의 형태로 최대한 자세히 표현
6. **통합성**: 동일한 데이터는 한번만 정의, 참조 활용

ERD 작성 순서

엔터티 그리기 => 엔터티 적절하게 배치 => 엔터티간 **관계** 설정 => 관계명 기술 => 관계의 **참여도** 기술 => 관계의 **필수여부** 기술

ER 모델 표기법



ERD 배치

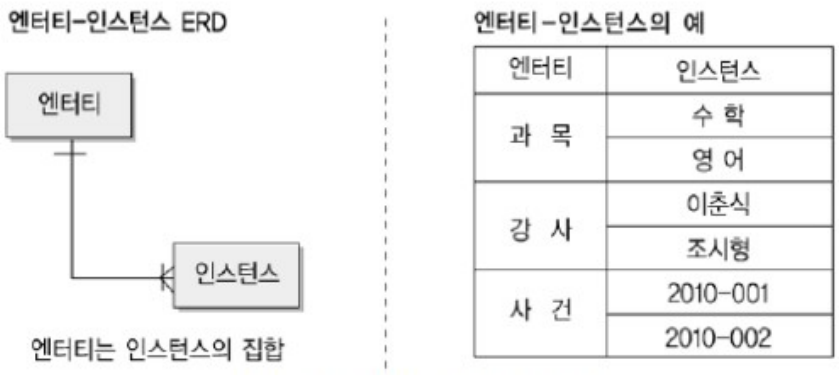
- 데이터 모델링 - 가장 중요한 엔터티: **왼쪽 상단**, 이를 중심 다른 엔터티 나열 전개
- ERD - 가장 중요한 엔터티: 왼쪽 상단에서 조금 아래쪽 중앙에 배치, 전체 엔터티와 어울릴 수 있도록

---

4) 엔터티

엔터티

- 업무에 필요하고 유용한 정보를 저장하고 관리하기 위한 집합적인 것
- 업무에서 관리되어야 하는 **데이터 집합**
- 보이지 않는 개념 표함



[그림 Ⅰ-1-15] 엔터티와 인스턴스

엔터티의 특징

1. 반드시 해당 업무에서 **필요하고 관리하고자하는 정보**이어야 한다
2. **유일한** 식별자에 의해 식별 가능
3. 영속적으로 존재하는 (1개 X, **2개 이상의**) 인스턴스의 집합
4. 업무 프로세스에 의해 **이용**되어야 한다
5. 반드시 **속성**이 있어야 한다
6. 다른 엔터티와 **최소 1개 이상의 관계**가 있어야 한다

   => 공통코드, 통계성 엔터티, 코드성 엔터티의 경우 관계를 생략할 수 있다

엔터티 분류

- **유무형**에 따른 분류 (물리적 형태 존재 여부)
  - **유형** 엔터티: 물리적 형태, 안정적, 지속적 활용 ex) 사원, 물품, 강사
  - **개념** 엔터티: 관리해야 할 개념적 정보로 구분 ex) 조직, 보험상품
  - **사건** 엔터티: 업무 수행 시 발생 ex) 주문, 청구, 미납
- **발생시점**에 따른 분류
  - **기본/키** 엔터티 (Fundamental Entity, **Key Entity**)

    그 업무에 **원래 존재**하는 정보, **독립적**으로 생성 가능

    타 엔터티의 **부모** 역할, 자신의 고유한 **주식별자**를 가짐

    ex) 사원, 부서, 고객, 상품, 자재
  - **중심** 엔터티 (Main Entity)

    기본 엔터티로부터 발생, **다른 엔터티와의 관계**로 많은 행위 엔터티 생성

    ex) 계약, 사고, 주문
  - **행위** 엔터티 (Active Entity)

    2개 이상의 부모 엔터티로부터 발생, **자주 바뀌거나 양 증가**

    ex) 주문목록, 사원변경이력
- **독립 엔터티**: 사람, 물건, 장소 등과 같이 현실세계에 존재하는 엔터티
- **업무중심 엔터티**: Transcation이 실행되면서 발생하는 엔터티
- **종속 엔터티**: 주로 1차 정규화로 인해 관련 중심 엔터티로부터 분리된 엔터티
- **교차 엔터티**: **M:M 관계 해소 목적** => N:M => 1:M

엔터티 명명

- 현업 업무에서 사용하는 용어 사용, **약어 사용 금지**
- 단수명사 사용, 고유한 이름 사용, 생성 의미대로 부여

---

5) 속성, 도메인

속성 (Attribute)

- 업무에서 필요로 하는 인스턴스에서 관리하고자 하는 의미상 더 이상 분리되지 않는 **최소의 데이터 단위**
- 엔터티를 설명하고, 인스턴스의 구성요소
- 1개의 엔터티는 **2개 이상의 인스턴스 집합**

- 1개의 엔터티는 2개 이상의 속성 / 1개의 속성은 1개의 값
- **1개의 속성은 1개의 속성값 가짐**
- 속성도 집합이다, 중복된 값 존재 가능

구성 방식의 분류

- **PK** (Primary Key) 속성: 엔터티 식별할 수 있는 속성
- **FK** (Foreign Key) 속성: 다른 엔터티와의 관계에서 포함된 속성
- **일반** 속성: 엔터티에 포함되고, PK, FK에 포함되지 않은 속성

> - **복합** 속성: 여러 세부 속성으로 구성되는 속성
> - **단순** 속성: 더 이상 다른 속성들로 구성될 수 없는 단순한 속성

속성의 특성에 따른 분류

1. **기본 속성** (Basic)

   업무로부터 추출한 **모든 일반적이고 많은 속성**

2. **설계 속성** (Designed)

   업무를 규칙화하기 위해 **새로 만들거나 변형, 정의**하는 속성

   ex) 일련번호

3. **파생 속성** (Derived)

   다른 속성에 **영향을 받아 발생**하는 속성

   빠른 성능을 낼 수 있도록 원래 속성의 **값을 계산**, 적을수록 좋음

   ex) 합

속성 명명

- 해당 업무에서 사용하는 이름 부여, 약어 사용 금지
- 서술식 속성명 사용 금지, 구체적으로 명명하여 데이터 모델에서 **유일성** 확보

선택도

- 특정 조건에 의해 선택될 것으로 예상되는 레코드의 비율
- 조건절에서 요청한 **값의 범위/전체 값**으로 계산

도메인 (Domain)

- 각 속성이 가질 수 있는 값의 범위
- 엔터티 내에서 속성에 대한 데이터 타입과 크기, 제약사항 지정

---

6) 관계, UML

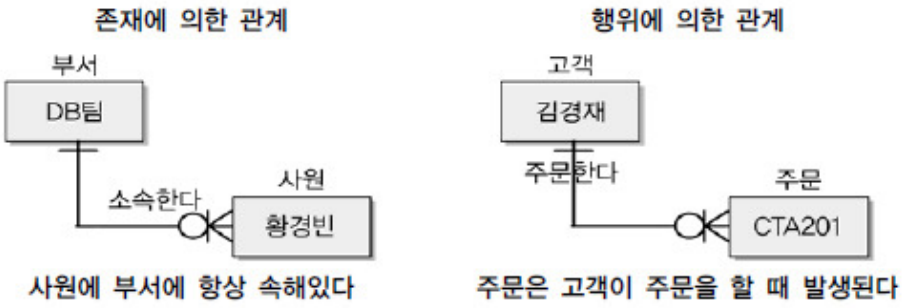
관계 => 관계명, 관계차수, 관계선택사양

- 엔터티의 인스턴스 사이의 **논리적인 연관성**으로서 존재의 형태로서나 행위로서 서로에게 **연관성이 부여**된 상태
- 관계 페어링의 집합
- ex) 강사 - 가르친다(관계) - 수강생

페어링

- 엔터티 안에 인스턴스가 개별적으로 관계를 가지는 것

UML (Unified Modeling Language, 통합모델링언어)에서의 관계



[그림 Ⅰ-1-32] 관계의 분류

- **연관관계** (실선) : **항상 이용**하는 관계, 존재적 관계 ex) 소속된다
- **의존관계** (점선) : **상대 행위**에 의해 발생하는 관계 ex) 주문한다
- **ERD**: 구분 X
- **클래스다이어그램**: 구분 O

관계 표기법

1. **관계명** (Membership): 관계의 이름
2. **관계차수** (Cardinality): 관계의 기수성, 1:1, 1:M, M:N
3. **관계선택성, 관계선택사양** (Optionality): 필수관계, 선택관계

관계 체크사항

1. 2개의 엔터티 상이에 관심 있는 연관 규칙 존재?
2. 2개의 엔터티 사이에 정보의 조합 발생 존재?
3. 업무기술서, 장표에 관계연결에 대한 규칙 서술 존재?
4. 업무기술서, 장표에 관계연결을 가능하게 하는 동사 존재?

관계 읽기

- 기준(Source) 엔터티: 한 개(One) 또는 각(Each)으로 읽기
- 대상(Target) 엔터티의 관계참여도 즉 개수(1개 이상) 읽기
- 관계선택사양과 관계명을 읽는다

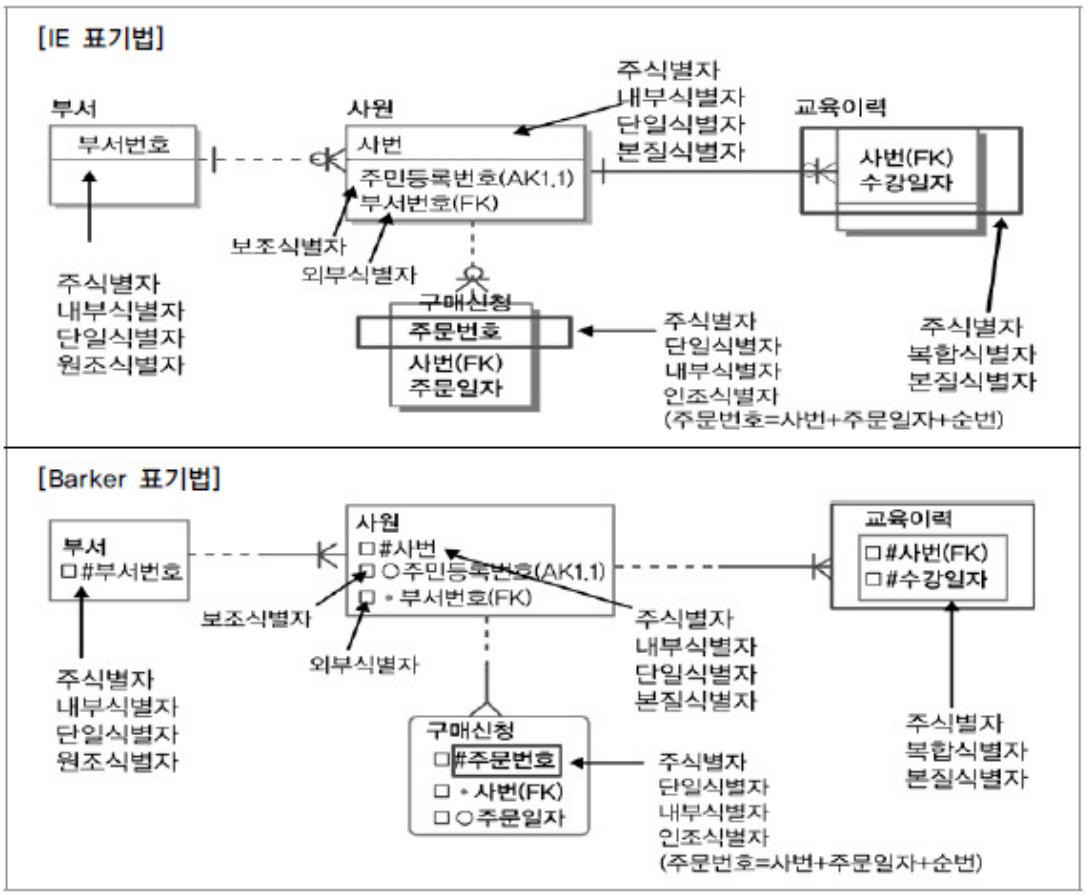
---

7) 식별자

식별자

- 엔터티 내에서 인스턴스를 구분하는 구분자
- 식별자는 **논리적**, Key는 **물리적** 데이터 모델링 단계에 사용

식별자 표기법



[그림 Ⅰ-1-42] 식별자의 분류-데이터 모델

식별자 종류

- **대표성 여부**
  - **주 식별자** (Primary Identifier)

    엔터티 내에서 각 어커런스를 구분할 수 있는 구분자, 타 엔터티와 참조 관계 연결 O
  - **보조 식별자** (Alternate Identifier)

    엔터티 내에서 각 어커런스를 구분할 수 있는 구분자, 대표성 X, 참조 관계 연결 X
- **스스로 생성 여부**
  - **내부 식별자**: **스스로 생성**되는 식별자
  - **외부 식별자**: 타 엔터티로부터 받아오는 식별자
- **속성의 수**
  - **단일 식별자**: 1개의 속성으로 구성
  - **복합 식별자**: 2개 이상의 속성으로 구성
- **대체 여부**
  - **본질 식별자**: 업무, **비즈니스 프로세스**에 의해 만들어지는 식별자
  - **인조 식별자**: 인위적으로 만든 식별자

주식별자 특징

- **엔터티를 대표**
- **유일성**: 주식별자에 의해 모든 인스터스들이 **유일하게 구분**
- **최소성**: 주식별자를 구성하는 속성의 수는 유일성을 만족하는 **최소의 수**가 되어야 함

- **불변성**: 지정된 주식별자의 값은 자주 변하지 않아야 함, 변하면 이전 기록 말소됨
- **존재성**: 주식별자가 지정되면 반드시 **값 존재** (Null 안됨)

주식별자 도출기준

1. 해당 업무에서 자주 이용되는 속성
2. 명칭, 내역 등과 같이 이름으로 기술되는 것 X
3. 복합으로 주식별자로 구성할 경우, 너무 많은 속성 X => 너무 많으면 인조식별자 생성
4. 자주 수정되는 속성 X

식별자 관계

- **식별자 관계**
  - **강한** 연결관계 표현, **실선** 표기 / 자식 주식별자의 구성에 포함됨
  - **반드시** 부모엔터티 종속
  - 자식 주식별자 구성에 부모 주식별자 포함 필요
  - 상속받은 주식별자 속성을 타 엔터티에 이전 필요
  - 부모로부터 받은 식별자를 자식 엔터티의 주식별자로 이용하는 경우
  - 자식의 주식별자로 부모의 주식별자 상속

  => 식별자 관계로만 설정 시 **주식별자 증가로 오류 유발**
- **비식별자 관계**
  - **약한** 연결관계 표현, **점선** 표기 / 자식 일반 속성에 포함됨
  - **약한 종속관계**, 자식 주식별자 구성을 **독립적**으로 구성
  - **대표성 X** => 참조 관계 X
  - 자식 주식별자 구성에 부모 주식별자 부분 필요
  - 상속받은 주식별자 속성을 타 엔터티에 차단 필요
  - 부모쪽의 관계 참여가 선택 관계
  - 부모 속성을 자식의 일반 속성으로 사용

    => 비식별자 관계로만 설정 시 **부모 엔터티와 조인하여 성능 저하**

> 1. 부모 없는 자식이 생성될 수 있는 경우
> 2. 부모와 자식의 생명주기가 다른 경우, 별도로 소멸
> 3. 여러 개의 엔터티를 하나로 통합하면서 각각의 엔터티가 갖고 있던 여러 개의 개별 관계가 통합되는 경우
> 4. 자식엔터티에 별도의 주식별자를 생성하는 것이 더 유리한 경우
> 5. SQL 문장이 길어져 복잡성 증가되는 것 방지 => 가장 마지막으로 고려

제2장. 데이터 모델과 성능

1) 성능 데이터 모델링

성능 데이터 모델링

- DB 성능 향상을 목적으로 **설계 단계**의 데이터 모델링 때부터 **성능과 관련된 사항**이

  (사항: 정규화, 반정규화, 테이블 통합, 테이블 분할, 조인 구조, PK, FK)

  데이터 모델링에 반영될 수 있도록 하는 것

- **분석/설계 단계**에서 데이터 모델에 **성능**을 고려한 데이터 모델링을 수행할 경우

  => 성능 저하에 따른 **재업무(Rework) 비용 최소화 가능**
- 데이터의 증가가 빠를수록

  => 성능 저하에 따른 **성능 개선 비용 기하급수적으로 증가**
- **데이터모델**: 성능을 튜닝하면서 **변경될 수 있다**

성능 데이터 모델링 고려사항 순서

1. 데이터 모델링을 할 때, **정규화** 정확하게 수행
2. **DB 용량 산정** 수행

   전체적인 데이터베이스에 발생되는 트랜잭션의 유형과 양 분석하는 자료가 됨
3. DB에 발생되는 **트랜잭션 유형** 파악 (애플리케이션의 트랜잭션의 유형 파악)
4. 용량과 트랜잭션 유형에 따라 **반정규화** 수행
5. **이력모델**의 조정, **PK/FK** 조정, **슈퍼/서브 타입** 조정

   성능 향상을 위한 모델링 작업의 중요한 요소
6. **성능 관점**에서 데이터 모델 **검증**

---

2) 정규화

함수적 종속성

- 데이터들이 어떤 기준 값에 의해 종속되는 현상

정규화 (-) => 입력/수정/삭제 성능 향상

- 반복적인 데이터를 분리하고 각 데이터가 종속된 테이블에 적절하게 배치되도록 하는 것
- 데이터의 일관성, 최소한의 데이터 중복, 최대한의 데이터 유연성을 위한 방법, 데이터 분해하는 과정
- 데이터의 **정합성 확보**와 데이터 **입력/수정/삭제** 시 발생할 수 있는 이상 현상을 방지하기 위해 **중복 제거, 데이터 독립성 확보**
- 비즈니스 변화 발생 => 데이터 모델의 변경 최소화 가능

1차 정규화

- 함수종속, 복수의 속성값을 갖는 속성 분리
- 같은 성격, 내용 컬럼이 연속, **중복** 속성, 로우/칼럼 단위의 중복 => 컬럼 제거, 테이블 생성, 속성의 원자성 확보

2차 정규화

- 함수 종속, 주식별자에 완전종속적이지 않는 속성 분리
- **PK 복합키 구성** => **부분적 함수 종속 관계** 테이블 분리 (부분종속 속성 분리)

3차 정규화

- 함수종속, 일반속성에 종속적인 속성 분리
- PK가 아닌(**주식별자를 제외한**) 일반 컬럼에 의존하는 컬럼 분리, **이행 함수 종속성** 제거

보이스-코드 정규화 (BCNF)

- 함수종속, 결정자안에 함수 종속을 가진 주식별자 분리
- 기본키 제외하고, 후보키가 있는 경우 => 후보키가 기본키를 종속시키면 분해, **다수의 주식별자** 분리

4차 정규화

- 다가 종속 속성 분리
- 하나의 릴레이션에서 한 속성에서 다른 속성에 대한 결과값이 여러 개 발생하는 **다치 종속** 분리

5차 정규화

- 결합 종속일 경우 => 2개 이상의 N개로 분리, **조인**에 의해 발생하는 이상현상 제거

---

3) 반정규화

반정규화 (+) => 조인 성능 향상

- 정규화된 엔터티, 속성, 관계에 대해 시스템의 **성능 향상**과 개발과 운영의 **단순화**를 위해

  중복, 통합, 분리 등을 수행하는 데이터 모델링의 기법
- 데이터 **무결성 위험 존재**, 데이터 중복하여 반정규화 적용 이유

  => 조회 시, 디스크 I/O가 많거나 경로가 멀어 **조인**에 의해 / 컬럼을 계산하여 읽을 때 => **성능 저하**
- 데이터 중복 허용 => 성능 향상, 데이터 독립성 낮아짐
- 트랜잭션의 유형, 데이터 용량 등 고려 => 특별히 성능 이슈 X => 반정규화 안 해도 됨
- **window function**: 이전/이후 위치의 레코드에 대한 탐색

반정규화 절차

- **반정규화 대상조사** => 빈도수, 범위, 통계성, 조인

  재현의 적시성으로 판단 => 빌링의 잔액은 다수 테이블에 대한 다량의 조인이 불가피, 데이터 제공의 **적시성 확보**를 위한 **필수 반정규화 대상 정보**

  1. **범위 처리 빈도수 조사**

     자주 사용되는 테이블에 접근하는 프로세스의 수가 **많고** 항상 **일정한 범위만을 조회**하는 경우
  2. **대량의 범위 처리 조사**

     테이블에 **대량**의 데이터가 있고 대량의 데이터 범위를 **자주 처리**하는 경우, **처리 범위를 일정하게 줄이지 않으면 성능을 보장할 수 없는 경우**
  3. **통계성 프로세스 조사**

     통계성 프로세스에 의해 **통계 정보를 필요**로 할 때 **별도의 통계 테이블 생성**
  4. **테이블 조인 개수**

     테이블에 지나치게 **많은 조인**이 걸려 데이터를 조회하는 작업이 기술적으로 어려울 경우
- **다른 방법 유도 검토** => 뷰, 클러스터링, 인덱스, 응용애플리케이션

  1. **뷰(VIEW) 테이블**

     지나치게 많은 조인이 걸려 데이터를 조회하는 작업이 기술적으로 어려울 경우, **뷰** 사용

     성능 향상 X
  2. **클러스터링 적용/인덱스의 조정**

     대량의 데이터 처리나 부분 처리 => 성능 저하 => **클러스터링** 적용/**인덱스** 조정

     조회가 대부분 => **클러스터링 적용**

3. **파티셔닝**
    - 논리적으로는 하나의 테이블이지만 **물리적으로는 여러 테이블로 분리**
    - **대량의 데이터**는 **PK의 성격**에 따라 **부분적인 테이블로 분리 가능**
    - 파티셔닝 키에 의해 **물리적 저장공간 분리** => 성능 저하 방지

      => 테이블 적용 기법) 데이터 엑세스 성능 향상, 데이터 관리방법 개선
    - 값 범위 기준: **Range** Partition / 특정 값 지정: **List** Partition
    - 해시 함수 적용: **Hash** Partition / 범위+해시 복합적: **Composite** Partition
4. **캐시**

   응용 애플리케이션에서 **로직 구사하는 방법 변경** => 성능 향상

테이블 반정규화

- **테이블 병합**
    1. **1:1 관계** 테이블 병합 / **1:M 관계** 테이블 병합
    2. **슈퍼/서브 관계** 테이블 병합
- **테이블 분할**
    1. **수직 분할**

       칼럼 단위 테이블을 디스크 I/O를 **분산 처리** => **테이블을 1:1로 분리**
    2. **수평 분할**

       로우 단위로 집중 발생되는 트랜잭션 분석

       => 디스크 I/O 및 데이터 접근의 **효율성**을 높여 성능 향상 => **로우 단위로 테이블 쪼갬**
- **테이블 추가**
    1. **중복**: 다른 업무이거나 서버가 **다른** 경우 => 동일한 테이블 구조 중복하여 원격 **조인** 제거
    2. **통계**: SUM, AVG 등을 미리 수행, 계산 => **조회** 시 성능 향상
    3. **이력**: 이력 테이블 중 마스터 테이블에 존재하는 레코드를 **중복** => **이력 테이블**에 존재시켜 성능 향상
    4. **부분**: 하나의 테이블의 전체 칼럼 중 자주 이용하는 **집중화**된 칼럼들이 있을 때 => 디스크 I/O를 줄이기 위해 해당 칼럼들을 모아놓은 별도의 반정규화된 테이블 생성

속성의 반정규화 (칼럼) => 무결성 깨트릴 가능성

1. **중복 칼럼 추가**

   조인에 의해 처리할 때 성능 저하 예방=> **중복된 칼럼**을 위치시킴
2. **파생 칼럼 추가**

   트랜잭션이 처리되는 시점에 계산에 의해 발생되는 성능 저하 예방 => **미리 값 계산**하여 칼럼에 보관
3. **이력 테이블 칼럼 추가**

   대량의 이력데이터를 처리할 때, **불특정 날/최근 값을 조회**할 때 => 나타날 수 있는 성능 저하 예방

   => 이력 테이블에 **기능성 칼럼**(최근값 여부, 시작.종료일자 등) 추가

   단점) 과도한 데이터가 한 테이블에 발생하게 되어 **용량이 너무 커짐**
4. **응용시스템 오작동을 위한 칼럼 추가**

   업무적으로 의미 없지만 **사용자의 실수**로 원래값으로 복구하기 원하는 경우

=> 이전 데이터를 **임시적으로 중복하여 보관**하는 기법

5. **PK에 의한 칼럼 추가**

   단일 PK 안에서 **특정 값을 별도로 조회**하는 경우 => 성능 저하 발생 가능, **일반 속성**으로 추가

관계의 반정규화 => 무결성 깨트릴 가능성 X

- **중복 관계 추가**

  데이터를 처리하기 위한 **여러 경로를 거쳐 조인 가능**

  => 이 때 성능 저하 발생 가능 예방 => **추가적인 관계** 맺는 방법

---

4) 로우 체이닝/마이그레이션

로우 체이닝

- 로우의 길이가 **너무 길어서** 데이터 블록 하나에 데이터가 모두 저장되지 않고

  2개 이상의 블록에 걸쳐 **하나의 로우**가 저장되어 있는 형태

로우 마이그레이션

- 데이터 블록에서 **수정 발생**

  => 수정된 데이터를 해당 데이터 블록에서 저장하지 못하고, **다른 블록의 빈 공간**을 찾아 저장하는 방식

로우 체이닝/로우 마이그레이션 발생

- 많은 블록에 데이터 저장 => DB 메모리에서 디스크 I/O가 발생할 때, **많은 I/O가 발생하여 성능저하 발생**
- **트랜잭션**을 분석하여 적절하게 **1:1 관계로 분리** => 성능 향상 가능하도록 해야 함

---

5) 슈퍼/서브 타입, PK/FK DB 성능 향상

슈퍼/서브 타입 모델

- 업무를 구성하는 데이터를 **공통/차이점의 특징을 고려**해 효과적으로 표현한 **논리적** 모델
- **슈퍼 타입**: **공통** 부분
- **서브 타입**: 공통으로부터 상속받아 다른 엔터티와 차이가 있는 속성, 서브타입끼리 **상호배타적 관계**
- **UNION**: 정렬작업 발생
- **UNION ALL**: 중복데이터가 없는 경우

슈퍼/서브 타입 데이터 모델의 변환 기술

OnoToOne Type	Plus Type	Single Type
1:1 타입	슈퍼+서브 타입	All in One 타입, 동시
개별로 발생되는 트랜잭션	**슈퍼+서브 타입**에 대해 발생되는 트랜잭션	**전체를 하나로 묶어** 트랜잭션이 발생할 때
개별 테이블로 구성	슈퍼+서브 타입 테이블로 구성	하나의 테이블로 구성

인덱스 특성 고려한 PK/FK DB 성능 향상

- 인덱스 특징) **여러 개의 속성이 하나의 인덱스**로 구성 => **앞쪽에 위치한 속성의 값**이 **비교자**로 있어야 좋은 효율 나타냄
- 앞쪽에 위치한 속성의 값 => 순서 **=, BETWEEN, <>, 기타**인 경우 효율적

---

6) 테이블 분할

PK에 의해 테이블을 분할하는 방법 => 파티셔닝

1. **RANGE PARTITION** ex) 요금_0401

   대상 테이블이 **날짜/숫자값으로 분리 가능**하고, **각 영역별**로 트랜잭션이 분리되는 경우

2. **LIST PARTITION** ex) 고객_서울

   지점, 사업소 등 **핵심적인 코드값으로 PK가 구성**되어 있고 **대량의 데이터**가 있는 테이블의 경우

3. **HASH PARTITION**

   지정된 HASH 조건에 따라 해시 알고리즘이 적용되어 테이블이 분리되는 경우

테이블에 대한 수평/수직 분할 절차

1. 데이터 모델링 완성
2. DB 용량산정
3. 대량 데이터가 처리되는 테이블에 대해 트랜잭션 처리 패턴 분석
4. 칼럼/로우 단위로 **집중화된 처리**가 발생하는 분석 => 집중화된 단위로 테이블 분리하는 것 검토
   - **칼럼** 많음 => **1:1 분리** (수직분할)
   - **데이터** 많음 => **파티셔닝** (수평분할)

---

7) 분산 DB

분산 DB

- **여러 곳으로 분산**되어있는 DB를 **하나의 가상 시스템**으로 사용할 수 있도록 한 DB
- **논리적으로 동일한 시스템에 속함**
- **컴퓨터 네트워크**를 통해 **물리적으로 분산**되어 있는 데이터 집합

분산 데이터베이스 장점

1. 지역 자치성, 점증적 시스템 용량 확장, 시스템 규모의 적절한 조절
2. 신뢰성, 가용성, 효용성, 융통성
3. 빠른 응답 속도, 통신비용 절감, 각 지역 사용자의 요구 수용 증대

분산 데이터베이스 단점

1. 소프트웨어 개발 비용 증가, 설계, 관리의 복잡성
2. 오류의 잠재성과 처리 비용의 증대, 불규칙한 응답 속도, 통제의 어려움
3. **데이터 무결성에 대한 위협**

분산 DB를 만족하기 위한 6가지 투명성

1. **분할 투명성 (단편화)**: 하나의 논리적 관계가 **여러 단편으로 분할**되어 각 사본이 **여러 사이트에 저장**
2. **위치 투명성**: 사용하려는 데이터의 **저장 장소 명시 불필요**, 위치 정보가 **시스템 카탈로그에 유지**

3. **지역사상 투명성**: **지역 DBMS와 물리적 DB** 사이의 Mapping 보장
4. **중복 투명성**: DB 객체가 여러 사이트에 **중복**되어 있는지 **알 필요가 없는 성질**
5. **장애 투명성**: **구성요소의 장애**에 무관한 트랜잭션의 **원자성 유지**
6. **병행 투명성**: 다수 트랜잭션 **동시 수행** 시 결과의 **일관성 유지**, **TimeStamp**, 분산 2단계 **Locking** 이용

   Lock/Unlock: 병행성 제어(동시성) 기법

분산 DB 적용 기법

1. **테이블 위치 분산**

   설계된 테이블을 **본사와 지사 단위**로 분산, 위치별 DB 문서 필요
2. **테이블 분할 분산**

   각각의 테이블을 쪼개어 분산

   - **수평 분할**: **로우** 단위로 분리, 지사별로 다를 때 => 중복 X
   - **수직 분할**: **칼럼** 단위로 분리, 각 테이블에 **동일 PK 존재**해야 함
3. **테이블 복제 분산**

   동일한 테이블을 다른 지역이나 서버에서 **동시에 생성, 관리**하는 유형

   - **부분 복제**: 마스터 DB에서 테이블의 일부의 내용만 다른 지역이나 서버에 위치
   - **광역 복제**: 마스터 DB 테이블의 내용을 각 지역이나 서버에 존재
4. **테이블 요약 분산**

   지역/서버 간에 데이터가 **비슷하지만 서로 다른 유형으로 존재**하는 경우

   - **분석 요약**: **동일한 테이블 구조**, 분산되어 있는 **동일한 내용**의 데이터 이용 => 통합된 데이터 산출

     ex) 판매실적 지사A, 지사B
   - **통합 요약**: 분산되어 있는 **다른 내용**의 데이터 이용 => 통합된 데이터 산출

     ex) 판매실적 지사 A: C 제품, 지사 B: D 제품

분산 DB 설계를 고려해야 하는 경우

1. **성능** 중요한 사이트
2. **공통 코드, 기준 코드, 마스터 데이터**의 성능 향상 => **복제** 분산
3. **실시간 동기화가 요구되지 않는 경우**, **Near Real Time**(거의 실시간) 특징을 가지고 있는 경우
4. 특정 서버에 부하가 집중되어 **부하 분산 필요**
5. **백업** 사이트 구성하는 경우

GSI (Global Single Instance)

- 통합된 1개의 인스턴스, **통합** 데이터베이스 구조 => 분산 데이터베이스와 **대치**되는 개념

좋은 모델링의 요건

- 중복배제, Business Rule, 완전성

2️⃣ SQL 기본 및 활용

제1장. SQL 기본

1) SQL 기본

DB

- 특정 기업/조직/개인이 필요에 의해 데이터를 **일정한 형태로 저장**해 놓은 것

DBMS

- **효율적인 데이터 관리**뿐만 아니라 예기치 못한 사건으로 인한 **데이터의 손상 피하고,**

  필요 시 필요한 데이터를 **복구**하기 위한 강력한 기능의 SW

RDBMS

- 관계형 데이터베이스 관리 시스템

테이블

- DB 기본 단위, 데이터를 저장하는 객체
- **가로** = 행 = **로우** = **튜플** = **인스턴스** / **세로** = 열 = **칼럼**

키 종류

- **기본키(Primary key)**: 후보지 중에서 엔터티를 **대표**할 수 있는 키
- **후보키(Candidate key)**: **유일성**과 **최소성**을 만족하는 키
- **슈퍼키(Super key)**: **유일성**은 만족하지만 최소성을 만족하지 않는 키
- **대체키(Alternate key)**: 여러 개의 후보키 중에서 **기본키를 선정하고 남은 키**
- **외래키(Foreign key)**: 하나 혹은 다수의 다른 테이블의 기본 키 필드를 가리키는 것

  참조 무결성을 확인하기 위해서 사용되는 키, **허용된 데이터 값만** 데이터베이스 저장하기 위해 사용

---

2) 명령어 기본

SQL

- 관계형 DB에서 데이터 **정의, 조작, 제어**를 위해 사용하는 언어
- **DML**: SELECT, INSERT, UPDATE, DELETE
- **DDL**: CREATE, ALTER, DROP, RENAME
- **DCL**: GRANT, REVOKE
- **TCL**: COMMIT, ROLLBACK

기본 규칙

- 테이블 명은 가능한 단수형 권고, 다른 테이블의 이름과 중복 X
- 테이블 내의 칼럼명은 중복 X
- 테이블 이름을 지정하고 각 칼럼들은 ()로 묶어 지정
- 각 칼럼들은 ,로 구분 ;로 끝난다
- 칼럼은 다른 테이블까지 고려해 DB 내에서는 일관성 있게 사용하는 것이 좋다 => 데이터 표준화 관점
- 칼럼 뒤에 데이터 유형 꼭 지정
- 테이블명과 컬럼명은 반드시 영문으로 시작, 벤더별로 길이에 대한 한계 존재
- A-Z, a-z, 0-9, _, $, #만 사용 가능

데이터 유형

- **CHAR(n)**: **고정** 길이 문자열 정보, 최대 길이만큼 **공간 채움** => char(3)='a' => 'a '
- **VARCHAR2(n)**: **가변** 길이 문자열 정보, 할당된 변수 값의 바이트만 적용

CHAR(n)	VARCHAR2(n)
'AA' == 'AA '	'AA' != 'AA '

- **NUMBER(n,m)**: 정수, 실수 등 숫자 정보, n길이의 숫자에 m길이만큼의 소수점 자리 포함
- **DATE**: 날짜와 시각 정보
- **TINYINY/SMALLINT 등**: 정수, 실수 등 숫자정보

제약조건: 데이터의 무결성 유지

1. **PRIMARY KEY** (기본키): UNIQUE & **NOT NULL**, 주키로 **테이블당 1개만 생성 가능**
2. **UNIQUE KEY** (고유키): 고유키 정의(중복 X), NULL 가능
3. **NOT NULL**: NULL 값 입력금지, 지정하지 않으면 기존의 NOT NULL 제약조건이 NULL로 변경됨
4. **CHECK**: 입력 값 범위 제한 => **데이터 무결성 유지**
5. **FOREIGN KEY** (외래키)

   테이블당 **여러 개 생성 가능**, NULL 가능

   입력하려는 값이 다른 테이블의 칼럼 값 참조, 참조 무결성 제약 받을 수 있음

와일드 카드

- *: 1개 이상 글자 %: 0개 이상 글자 -: 한 글자

합성 연산자, 문자열 연결

- ||: Oracle +: SQL Server

---

3) DDL (Data Definition Language)

- 데이터 정의어
- 테이블과 같은 데이터 구조를 정의하는데 사용되는 명령어
- 구조를 **생성/변경/삭제/이름 변경**하는 테이터 구조와 관련된 명령어

  => CREATE, ALTER, DROP, RENAME
- **Roll back 불가능, 삭제하면 복구 불가**

  => 모든 데이터 삭제 시, 로그를 남기는 것을 표준으로 한다는 원칙은 삭제 후 실수 발생 시 복구(Roll back)가 가능해야한다는 의미

```
# 테이블 생성
CREATE TABLE 테이블명(칼람명 데이터타입);
CREATE TABLE PLAYER
(PLAYER_ID CHAR(7) NOT NULL,
```

```
 PLAYER_NAME VARCHAR2(20) NOT NULL);

# 추가
ALTER TABLE 테이블명 ADD(칼람명 데이터타입);
# 삭제
ALTER TABLE 테이블명 DROP COLUMN 칼람명;
# 수정 - Oracle
ALTER TABLE 테이블명 MODIFY(칼람명 데이터타입 [NOT NULL]);
# 수정 - SQL Server: 여러개 칼럼 동시 수정 지원 X, 괄호 사용 X
ALTER TABLE 테이블명 ALTER COLUMN 칼람명 데이터타입 [NOT NULL];

# 제약조건 삭제
DROP CONSTRAINT 조건명;
# 제약조건 추가
ADD CONSTRAINT 조건명 조건(칼람명);
# 테이블명 변경
RENAME 테이블명 TO 바꾸려는 테이블명;
# 칼럼명 변경
RENAME COLUMN 열이름 TO 바꾸려는 칼람명;

# 테이블 삭제
DROP TABLE 테이블명 [CASCADE CONSTRAINT];
## 삭제하려는 테이블을 참조하고 있는 다른 테이블이 존재하는 경우,
## CASCADE 옵션을 명시하지 않으면 삭제되지 않음

# TRUNCATE: 테이블에 저장되어 있는 데이터 모두 제거, 테이블 자체 삭제 X
## DROP과 달리 디스크 사용량 초기화해서 재사용 가능, Roll back 불가능
TRUNCATE TABLE 테이블명;
```

```
# PK
## PRIMARY KEY 만들기
ALTER TABLE 테이블명 ADD CONSTRAINT 조건명 PRIMARY KEY(칼럼명);
## CREATE 구문에 포함 시
CONSTRAINT 조건명 PRIMARY KEY(칼럼명);

# NOTNULL: 공백불가 제약
ALTER TABLE 테이블명 MODIFY 칼럼명 NOT NULL;

# UNIQUE
ALTER TABLE 테이블명 ADD CONSTRAINT 조건명 UNIQUE(칼럼명);

# DEFAULT
ALTER TABLE 테이블명 MODIFY 컬럼명 DEFAULT 값 NOT NULL;
## 공백 시, DEFAULT로 가는 것이므로 NOT NULL 들어감

# CHECK: 범위를 정해놓은 제약
ALTER TABLE 테이블명 ADD CONSTRAINT 조건명 CHECK(범위지정);
```

- ADD CONSTRAINT: 컬럼 수정 => PK, UNIQUE, CHECK, FK
- MODIFY: 세부 항목 수정 => NOT NULL, DEFAULT

---

4) DML (Data Manipulation Language)

- 데이터 조작어
- **호스트 프로그램** 속에 삽입되어 사용되는 DML 명령어) **데이터 부속어**라고도 함
- DB에 들어 있는 데이터를 **조회하거나 검색**하기 위한 명령어 (RETRIEVE라고도 함)

  => SELECT
- DB의 **테이블에 들어 있는 데이터에 변형(삭제/수정)**을 가하는 종류의 명령어

  => INSERT, UPDATE, DELETE

```
# 데이터 삽입
## 삽입 칼럼을 명시하지 않으면, 모든 칼럼 삽입하기
INSERT INTO 테이블명 VALUES (데이터1, 데이터2);
INSERT INTO 테이블명(칼럼명1, 칼럼명2) VALUES (데이터1, 데이터2);
# 데이터 수정
## where절은 update 대상이 되는 데이터의 범위 결정
## where 없으면 모든 값 update 대상
UPDATE 테이블명 SET 칼럼명 = 변경값;
UPDATE 테이블명 SET 칼럼명 = 변경값 WHERE 조건문;
# 데이터 삭제
DELETE FROM 테이블명;
# 조회
SELECT 칼럼명 FROM 테이블명;
SELECT 칼럼명1 [AS] 별칭1, 칼럼명2 [AS] 별칭2 FROM 테이블명;

# 테이블 복사
CREATE TABLE 새로만들테이블명 AS
SELECT * FROM 복사할테이블명 [WHERE 절]
# 테이블 구조만 복사할 때
CREATE TABLE 새로만들테이블명 AS
SELECT * FROM 복사할테이블명 WHERE 1=2 (where에다가 참이 아닌 조건을 넣어줌)
# 테이블은 이미 생성되어 있고 데이터만 복사할 때
INSERT INTO 복사할테이블명 SELECT * FROM 복사할테이블명 [WHERE 절]
# 테이블 이름 변경
ALTER TABLE 구테이블명 RENAME TO 신테이블명

## SQL Server, 테이블 복사
SELECT * INTO 새로만들테이블명 FROM 복사할테이블명;


# ALL: 모두 출력, 별도로 표시 안 해도 Default
# DISTINCT: 중복 시 1번만 출력
SELECT DISTINCT 칼럼명 FROM 테이블명;
# 칼람 값 붙여서 출력
```

```
SELECT CONCAT(칼럼명1, 칼럼명2) FROM 테이블명;
SELECT 칼럼명1 || 칼럼명2 FROM 테이블명;
```

DELETE, DROP, TRUNCATE 비교

DROP	TRUNCATE	DELETE
테이블 자체 삭제	테이블 초기 상태로 만듦	데이터만 모두 지움
Auto commit	Auto commit	사용자 commit으로 수행
roll back 불가능 (복구 X)	roll back 불가능 (복구 X)	commit 이전 roll back 가능
테이블이 사용했던 Storage 모두 Release	최초 테이블 생성 시, 할당된 Storage만 남기고 Release	사용했던 Storage는 Release되지 않음

- **TRUNCATE**: UNDO를 위한 데이터 생성 X => 동일 데이터랑 삭제 시, **DELETE보다 빠름**

비절차적 데이터 조작어

- 사용자가 무슨(**WHAT**) 데이터를 원하는지만 명세
- 사용자가 **어떤 데이터를 원하지만 선언**
- **어떻게 데이터를 처리**할 지는 **DBMS에 맡김**

절차적 데이터 조작어

- 어떻게(**HOW**) 데이터를 접근해야하는지 명세
- **사용자가 어떤 데이터를 원하지는지, 그 데이터를 얻기 위한 처리 방법까지 설명**
- PL/SQL(오라클), T-SQL(SQL Serever)

---

5) DCL (Data Control Language)

- 데이터 제어어
- 데이터베이스에 접근하고 객체들을 사용하도록 **권한을 주고/회수**하는 명령어

  => GRANT, REVOKE

시스템 권한

- 사용자가 SQL 문을 실행하기 위해 필요한 적절한 권한
- 모든 유저는 각각 자신이 생성한 테이블 외에 다른 유저의 테이블에 접근하려면

  => 해당 테이블에 대한 오브젝트 권한을 소유자로부터 부여받아야 함

권한 명령어

- **GRANT**: 권한 부여 / **REVOKE**: 권한 취소

```
GRANT 권한 ON 테이블명 TO 유저명;
REVOKE 권한 ON 테이블명 FROM 유저명;
TO 유저 WITH GRANT OPTION; => 특정 사용자에게 권한 부여가능한 권한을 부여함, 부모 권한 회수, 자식도 회수
TO 유저 WITH ADMIN OPTION; => 테이블에 대한 모든 권한 부여, 부모 권한 회수 상관 X
```

- **DML 관련 권한**: SELECT, INSERT, UPDATE, DELETE, ALTER, ALL
- **REFERENCES**: 지정된 테이블을 참조하는 제약조건을 생성하는 권한
- **INDEX**: 지정된 테이블에서 인덱스를 생성하는 권한

```
CREATE INDEX 인덱스명 ON 테이블명 컬럼명;
```

- **CASCAED**: 해당 사용자 권한 및 트리구조로 **다른 사람에게 부여한 권한까지 모두 제거**
- **ROLE**: 유저에게 알맞은 권한들을 **한 번에 부여**하기 위해 사용하는 것

  다양한 권한을 그룹으로 묶어 관리할 수 있도록 사용자와 권한 사이에서 **중개** 역할 수행 => 번거로움, 어려움 해소

```
CREATE ROLE 유저명;
GRANT ROLE TO 유저명;
DROP USER PJS CASCADE;
```

Oracle의 유저

1. **SCOTT**: Oracle 테스트용 샘플 유저 / Default PW) TIGER
2. **SYS**: DBA 권한이 부여된 최상위 유저
3. **SYSTEM**: DB의 모든 시스템 권한을 부여받은 DBA 유저

---

6) TCL (Transaction Control Language)

- 트랜잭션 제어어
- 논리적인 작업의 단위를 묶어서 DML에 의해 조작된 결과를 **트랜잭션(작업단위) 별로 제어**하는 명령어

  => COMMIT, ROLLBACK

COMMIT

- 올바르게 반영된 데이터를 DB에 반영, 데이터에 대한 변경사항을 DB에 영구적으로 반영
- COMMIT 실행 전: 메모리에만 반영(**휘발성**), 나만 볼 수 있고, **다른 사용자가 데이터 못 고침**
- COMMIT 실행 후: 최종적으로 데이터 파일에 기록, 트랜잭션 완성
- DDL, DCL => **AUTO COMMIT** / DML => **COMMIT** (수동 commit/roll back)
- **AUTO COMMIT = false**
  - **Oracle**: DDL 수행 후 자동으로 commit 수행
  - **SQL Server**: 자동 commit 수행 X

ROLLBACK

- 트랜잭션 시작 이전의 상태로 되돌림, COMMIT되지 않은 모든 트랜잭션 롤백
- 관련된 행에 대한 잠금(LOCKING)이 풀리고 다른 사용자들이 데이터 변경을 할 수 있음

SAVEPOINT (저장 지점)

- 오브젝트를 현 시점에서 특정 SAVEPOINT까지 일부만 제거
- roll back을 수행할 때 => 전체 작업을 되돌리지 않고 일부만 되돌림

```
# 트랜잭션 시작
BEGIN TRANSACTION (== BEGIN TRAN)
# 트랜잭션 종료
COMMIT [TRANSACTION]
ROLLBACK [TRANSACTION] => 초최의 BEGIN TRANSACTION 시점까지 모두 ROLLBACK 수행

# Oracle
SAVEPOINT 저장지점;
ROLLBACK TO 저장지점;
# SQL Server
SAVE TRANSACTION 저장지점;
ROLLBACK TRANSACTION 지정지점;
```

---

7) 트랜잭션

트랜잭션 (Transasction)

- 데이터를 조작하기 위한 **하나의 논리적인 작업 단위**
- 밀접히 관련되어 분리될 수 없는 1개 이상의 DB 조작 말함
- **ROLLUP**
  - 트랜잭션이 항상 **전체**를 대상으로 **일괄 처리**하는 경우

    => 서브타입들을 슈퍼타입으로 **통합** => 전체를 **하나의 테이블** (슈퍼+서브타입) 구성해서 변환
- **ROLLDOWN**
  - 트랜잭션이 항상 **서브타입 개별**로 처리하는 경우

    => 슈퍼타입을 서브타입으로 통합 => **서브타입을 개별 테이블로 구성**
- **IDENTITY**
  - 트랜잭션이 항상 슈퍼/서브타입을 **각각 공동**으로 처리하는 경우

    => **슈퍼/서브타입 테이블**로 구성

트랜잭션 특징

- **원자성** (all or nothing)

  트랜잭션에 정의된 연산들은 **모두 성공적**으로 실행되던지, **전혀 실행**되지 않은 상태로 남아있어야 함
- **일관성**

  트랜잭션이 실행되기 전의 DB 내용이 잘못 되어 있지 않다면 트랜잭션이 실행된 이후에도 DB의 내용에 잘못이 있으면 안 됨
- **고립성/격리성**

트랜잭션이 실행되는 도중에 다른 트랜잭션의 영향을 받아 잘못된 결과를 만들어서는 안 됨

둘 이상의 트랜잭션이 동시에 실행되는 경우, **서로의 작업에 영향 X**

- **격리성이 낮은 경우, 발생할 수 있는 문제점**
  1. **Dirty read**

     다른 트랜잭션에 의해 **수정되었고 commit 전의 데이터를 읽는 것**
  2. **Non-repeatable read**

     한 트랜잭션 내에서 같은 쿼리를 2번 수행 => 그 사이에 다른 트랜잭션이 값을 **수정/삭제**하는 바람에 **두 쿼리 결과가 다르게 나타나는 현상**
  3. **Phantom read**

     한 트랜잭션 내에서 같은 쿼리를 2번 수행 => 1번째 쿼리에서 없던 **유령 레코드**가 2번째 쿼리에서 나타나는 현상

- **지속성/영속성**

  트랜잭션이 성공적으로 수행되면 그 트랜잭션이 갱신한 DB의 내용은 **영구적으로 저장됨**

---

8) NULL, 참조 무결성 규정 관련 옵션

NULL

- 값의 부재, 아직 정의되지 않은 미지의 값, 현재 데이터를 입력하지 못하는 경우
- IS NULL, IS NOT NULL만으로 비교 가능

  => NULL과의 모든 비교(IS NULL 제외)는 알 수 없음(Unknown) 반환
- **공백문자(Empty String) / 숫자 0 의미 X**
- NULL 값의 대상이 숫자 유형 데이터 경우 => 주로 0(zero) / 문자 유형 데이터 경우 => 'x'

  해당 시스템에서 의미 없는 문자로 바꾸는 경우가 많다
- **NULL 연산**
  - NULL 값과의 **사칙연산** => NULL 값 리턴
  - NULL 값과의 **비교연산** => 거짓(FALSE) 리턴
  - 숫자를 0으로 나누면 => 에러 발생 / 널로 나누면 => 널
  - 특정 값보다 크다, 적다라고 표현할 수 없음
- **Oracle**: 서비스명에 공백 입력 => **NULL**로 입력
- **SQL Server**: 서비스명에 공백 입력 => **공백**으로 입력

참조 무결성 규정 관련 옵션

- **ON DELETE**: 부모를 삭제할 때
  - **CASCADE**: 부모 값 삭제 시, 자식 값 같이 **삭제**
  - **SET NULL**: 부모 값 삭제 시, 자식 해당 칼럼 **NULL** 처리
  - **SET DEFAULT**: 부모 값 삭제 시, 자식 **기본값** 설정
  - **RESTRICT**: **자식의 PK 없는 경우만 부모 삭제**
- **ON INSERT**: 자식에 입력할 때
  - **AUTOMATIC**: 부모가 없을 때, **부모 입력 후 자식 입력**
  - **SET NULL**: 부모가 없는 경우, 자식의 FK를 **NULL**

- **SET DEFAULT**: 부모가 없는 경우, FK를 **기본값**으로
- **DEPENDENT**: **부모의 PK가 있는 경우만 자식 입력**

---

9) WHERE, 연산자

WHERE 절

- FROM 절 다음에 위치
- 집계함수는 WHERE 절에 올 수 없다

연산자의 우선순위

- () => NOT => 비교연산자 => AND => OR
- 비교 연산자 (=, >, >=, <, <=)
- SQL 비교 연산자 (BETWEEN a AND b, IN (list), LIKE, IS NULL)

부정 비교 연산자

- **!=, ^=, <>** : 같지 않다

  <>은 ISO 표준으로 모든 운영체제에서 사용 가능

비교 연산자

- **BETWEEEN a AND b**: a와 b의 값 사이에 있으면 된다, a,b 값 포함
- **IN (list)**: 리스트에 있는 값 중에서 어느 하나라도 일치하면 된다, IN(1, 2, NULL) => IN(1, 2)
- **IS NULL**: NULL 값인 경우 (Oracle은 VARCHAR2 빈 문자열을 NULL로 판단)
- **LIKE '비교문자열'**: 비교문자열과 형태 일치, _나 % 앞에 ESCAPE로 특수 문자 지정, 검색 가능

---

10) 내장 함수 (문자, 숫자, 날짜, 변환, NULL)

내장 함수 (Built-in Function)

- 함수 => 벤터에서 제공하는 함수 (내장 함수), 사용자가 정의할 수 있는 함수
- 내장 함수 => 단일행 함수, 다중행 함수

단일행 함수 (Single-Row Function)

1. SELECT, WHERE, ORDER BY, UPDATE의 SET 절에서 사용 가능
2. 행에 개별적 조작
3. 여러 인자가 있어도 결과는 1개만 출력 (다중행 함수도 단일 값만 반환)
4. 함수 인자에 상수, 변수, 표현식 사용 가능, 함수 중첩 가능

- **문자형 함수**: 문자 입력 => 문자나 숫자 값 반환

  > Oracle / SQL Server

  - **LOWER**(문자열): 소문자 / **UPPER**(문자열): 대문자
  - **ASCII**(문자): ASCII 값으로 반환 / **CHR / CHAR**(ascii 번호): 문자로 반환
  - **CONCAT**(문자열1, 문자열2): 문자열 연결
  - **SUBSTR / SUBSTRING**(문자열, m[, n]): 문자열 중 m위치에서 n개의 문자 반환

- **LENGTH / LEN**(문자열): 문자열 길이를 숫자 값으로 반환
- **LTRIM**(문자열, [, 지정문자]): **첫 문자**부터 확인, 지정문자 제거 (기본 지정문자: 공백)
- **RTRIM**(문자열, [, 지정문자]): **마지막 문자**부터 확인, 지정문자 제거 (기본 지정문자: 공백)
- **TRIM**([leading | trailing | both] 지정문자 **FROM** 문자열): 머리말, 꼬리말, 양쪽에 있는 지정 문자 제거 (생략: both, 기본 지정문자: 공백)

  > LTRIM, RTRIM, TRIM: SQL Server에서는 지정문자 사용 X => **공백**만 제거 가능

  ```
  CONCAT('RDBMS',' SQL') -> 'RDBMS SQL'
  SUBSTR('SQL Expert',5,3) -> 'Exp'
  LTRIM('xxxYYZZxYZ','x') -> 'YYZZxYZ'
  TRIM('x' FROM 'xxYYZZxYZxx') -> 'YYZZxYZ'
  ```

- **숫자형 함수**: 숫자 입력 => 숫자 값 반환
  - **ABS**(숫자): 절대값 반환
  - **SIGN**(숫자): 숫자가 양수 => 1, 음수 => -1, 0 => 0 반환
  - **MOD**(숫자1, 숫자2): 나머지 반환
  - **CEIL**(숫자): 올림 (정수) / **FLOOR**(숫자): 버림 (정수)
  - **ROUND**(숫자1, 숫자2): 반올림 / **TRUNC**(숫자1, 숫자2): 버림
- **날짜형 함수**: DATE 타입의 값 연산
  - **SYSDATE**: 현재날짜와 시각 출력
  - **EXTRACT**: 날짜에서 데이터 출력
  - **TO_NUMBER(TO_CHAR(d, 'YYYY')**: 연도를 데이트 타입 => 문자열 => 숫자

    > SELECT TO_CHAR(SYSDATE+1, 'YYYYMMDD') FROM DUAL;
    >
    > => 오라클 DB에서 내일 날짜 조회

  - 1 = 하루, 1/24 = 1시간, 1/24/60 = 1분
- **변환형 함수**: 문자, 숫자, 날짜형 값의 데이터 타입 변환
- **NULL 관련 함수**: NULL 처리하기 위한 함수
  - **NVL**(식1, 식2) / **ISNULL**(식1, 식2): 식1의 값이 NULL이면 식2 출력, 공집합을 바꿔주진 않음
  - **NULLIF**(식1, 식2): 식1이 식2와 같으면 **NULL**, 같지 않으면 식1 출력
  - **COALESCE**(식1, 식2): NULL이 아닌 **첫번째 값 반환**, 모두 NULL이면 NULL 반환

    ```
    COALESCE(NULL, NULL, 'abc', 12345) => 'abc'
    ```

---

11) 다중행 함수

다중행 함수 (Multi-Row Function)

1. 여러 행들의 그룹이 모여서 그룹당 단 하나의 결과를 돌려주는 함수
2. GROUP BY 절은 행들을 소그룹화 함
3. SELECT, WHERE, ORDER BY, UPDATE의 SET 절에서 사용 가능

- **집계 함수** (Aggregate Function)
  - **COUNT**(*): **NULL 포함** 행의 수 / **COUNT**(표현식): **NULL 제외** 행의 수
  - **SUM, AVG**([DISTINCT | ALL] 표현식): **NULL 제외** 합계, 평균 연산
  - **STDDEV**([DISTINCT | ALL] 표현식): 표준 편차 (루트 분산)
  - **VARIAN**([DISTINCT | ALL] 표현식): 분산 (편차(값-평균)제곱의 평균)
  - **MAX, MIN**([DISTINCT | ALL] 표현식): 최대값, 최소값
- **그룹 함수** (Group Function)

  합계 계산 함수, NULL을 빼고 집계, 결과 값 없는 행은 출력 X

  CUBE, GROUING SETS, ROLLUP 모두 일반 그룹 함수로 동일한 결과 추출 가능

  집계된 레코드에서 **집계 대상 컬럼 이외의 GROUP 대상 컬럼의 값 => NULL 반환**

  1. **ROLLUP**

     > ROLLUP(C1, C2) => (C1, C2), (C1, NULL), (NULL, NULL)
     >
     > ROLLUP(C1, C2, C3) => (C1, C2, C3), (C1, C2, NULL), (C1, NULL, NULL), (NULL, NULL, NULL)

     - GROUP BY로 묶인 칼럼의 소계 계산
     - **계층 구조** => GROUP BY의 **칼럼 순서가 바뀌면 결과 값 바뀜** => 인수 순서에 주의
     - Sub total을 생성하기 위해 사용
     - Grouping Columns의 수를 N이라고 가정하면, N+1 Level의 Subtotal 생성
  2. **CUBE** => 2^N

     > CUBE(A, B) => (A, NULL), (B, NULL), (A, B), (NULL, NULL)

     - 조합 가능한 모든 값에 대해 **다차원 집계 생성**, ROLLUP에 비해 **시스템 부하 심함**
     - 정렬 필요 시) **ORDER BY** 절에 명시적으로 정렬 칼럼 표시
     - **괄호 주의**

     ```
     CUBE(A, B) = GROUPING SETS((A), (B), (A, B), ())
     ```

  3. **GROUPING SETS** => 계층구조 X

     > GROUPING SETS(A, B) => (A, NULL), (NULL, B)

     - 인수들에 대한 개별 집계 구할 수 있음, 다양한 소계 집합 생성 가능
     - **인수 순서 바뀌어도 결과 같음** => 평등한 관계(계층 구조 X)
     - 정렬 필요 시) **ORDER BY** 절에 명시적으로 정렬 칼럼 표시
  4. **GROUPING**
     - GROUPING SETS로 CUBE 변환 가능
     - **추가 집계 => 1, 일반적인 컬럼 => 0**

     ```
     SELECT CASE WHEN GROUPING(자재번호) = 1 THEN '자재전체' ELSE 자재번호 END AS 자재번호
     ```

- **윈도우 함수** (Window Function)

  행과 행간의 관계 정의, 행과 행간 비교, 연산하는 함수

  PARTITION BY, ORDER BY => 유사

결과 건수 줄지 않음 / group by, window function 병행 X

- **순위 관련 함수**
  1. **RANK**: 동일한 값에 대해서는 동일한 순위 부여 **(1, 2, 2, 4)**
  2. **DENSE_RANK**: 동일한 순위를 하나의 등수로 간주 **(1, 2, 2, 3)**
  3. **ROW_NUMBER**: 동일한 값이라도 고유한 순위 부여 **(1, 2, 3, 4)**

```
SELECT RANK() OVER (PARTITION BY ~ ORDER BY ~)
```

- **집계 관련 함수**
  1. **SUM**: 파티션별 윈도우 **합**
  2. **MAX, MIN**: 파티션별 윈도우의 **최대/최소 값**
  3. **AVG**: 원하는 조건에 맞는 데이터에 대한 통계 값
  4. **COUNT**: 조건에 맞는 데이터에 대한 통계 값

```
# 현재 행을 기준으로 파티션 내에서 앞의 10건, 현재행, 뒤의 1건을 범위로 지정
ROWS BETWEEN 10 PRECEDING AND 1 FOLLING
```

  - 집계 윈도우 함수를 윈도우 절과 함께 사용 => 레코드 범위(집계대상) 지정 가능
- **행 순서 관련 함수**

  SQL Server 지원 X

  1. **FIRST_VALUE**: 파티션별 윈도우의 **처음** 값
  2. **LAST_VALUE**: 파티션별 윈도우의 **마지막** 값
  3. **LAG**: 파티션별 윈도우에서 **이전** 행의 값 (LAG(N) == 이전 N번째 값)
  4. **LEAD**: 파티션별 윈도우에서 **이후** 행의 값 (LAG(N) == 이후 N번째 값)
- **비율 관련 함수**
  1. **RATIO_TO_REPORT**: 파티션 내 칼럼의 **총합계에 대한 값의 백분율** (소수점) (>0, <=1)

     > **전체 급여**에서 사원의 급여 **차지 비율**

  2. **PERCENT_RANK**: 파티션별 윈도우에서 처음 값을 0, 마지막 값을 1로 해서 행의 순서별 백분율 (>0, <=1)

     > 본인의 급여가 **순서상** 몇 번째 위치인가

  3. **CUME_DIST**: 현재 행보다 작거나 같은 건수에 대한 **누적 백분율** (>0, <=1)

     > 본인의 급여가 **누적 순서상** 몇 번째 위치인가
     >
     > 3000 SAL 기준, 0.93 => 3000 이하 받는 사원이 모두를 기준으로 93%

  4. **NTILE**: 파티션별 **전체 데이터를 인수 값으로 N등분**

     > 급여를 기준으로 **4개의 그룹**으로 나눔
     >
     > 10명의 팀 4개조로 나눔 => 3, 3, 2, 2로 나눔 => 1, 1, 1, 2, 2, 2, 3, 3, 4, 4

---

12) GROUP BY, HAVING, ORDER BY, 연산순서

- 별칭이 없는 칼럼: 대문자
- 별칭 있는 칼럼: 그대로 사용

GROUP BY 절 특징

- GROUP BY 절을 통해 소그룹별 기준 정한 후, SELECT 절에 집계 함수 사용
- 집계 함수의 통계 정보는 **NULL 값을 가진 행을 제외하고 수행**
- GROUP BY 절에서는 **ALIAS(AS) 사용 불가**

HAVING 절 특징

- GROUP BY 절의 기준 항목이나 소그룹의 집계 함수를 이용한 조건 표시 가능
- HAVING 절에는 **집계함수**를 이용하여 조건 표시

ORDER BY 절 특징

- SQL 문장으로 조회된 데이터들을 다양한 목적에 맞게 특정한 칼럼을 기준으로 정렬 출력
- 칼럼명 대신 ALIAS 명이나 칼럼 순서를 나타내는 정수도 사용 가능, **혼용 가능**
- DEFAULT 값으로 ASC 적용, 오름차순, 작은 값부터, 생략 가능
- DESC 내림차순, 큰 값부터
- **SELECT 절에서 정의하지 않은 칼럼 사용 가능** => 메모리에 모든 칼럼을 올리므로
- GROUP BY 사용할 경우 => **SELECT에 없는 값 기술 X**, **집계함수 사용 가능**
- **Oracle**: NULL => 가장 큰 값
- **SQL Server**: NULL => 가장 작은 값

SQL 연산순서

- FROM > WHERE > GROUP BY > HAVING > **SELECT** > ORDER BY

```
SELECT [DISTINCT] 칼럼명 [[AS] ALIAS명] FROM 테이블명
[WHERE 조건식] [GROUP BY 칼럼/표현식] [HAVING 그룹조건식] [ORDER BY 칼람/표현식 [ASC /
DESC]];
```

---

13) DUAL 테이블, SEARCHED_CASE_EXPRESSION, SIMPLE_CASE_EXPRESSION, ROWNUM, TOP() WITH TIES

DUAL 테이블 특성

- 사용자 SYS가 소유하며 모든 사용자가 액세스 가능한 테이블
- SELECT ~ FROM ~의 형식을 갖추기 위한 일종의 **DUMMY** 테이블
- DUMMY라는 문자열 유형의 칼럼에 'X'라는 값이 들어 있는 행을 1건 포함하고 있다

SEARCHED_CASE_EXPRESSION

- 조건(**Condition**) 사용해 리턴값 결정
- CASE WHEN LOC = 'a' THEN 'b' => LOC의 **조건**에 따라 리턴값 정해짐

SIMPLE_CASE_EXPRESSION

- 표현(**Expression**) 사용해 리턴값 결정
- CASE LOC WHEN 'a' THEN 'b' => LOC가 **무슨 값인지의 표현**에 따라서 리턴값 정해짐

ROWNUM

- 원하는만큼의 행을 가져올 때 사용

```
SELECT * 칼럼명 FROM 테이블명 WHERE ROWNUM = 행번호;
```

SQL Server의 TOP() WITH TIES

- 사원 테이블에서 급여가 높은 2명을 내림차순으로 출력하는데 같은 급여를 받은 사원이 있으면 같이 출력

```
SELECT TOP(2) WITH TIES 사원이름, 급여 FROM 사원 ORDER BY 급여 DESC;
```

---

14) JOIN

JOIN

- 2개 이상의 테이블을 **연결/결합**하여 데이터 출력하는 것
- 일반적으로 행들은 **PK/FK 값의 연관**에 의해 JOIN 성립
- 어떤 경우에는 PK, FK 관계가 없어도 **논리적인 값들의 연관**만으로 JOIN 성립 가능
- **N-1**: 5가지 테이블을 JOIN하려면 최소 4번의 JOIN 과정 필요

```
# WHERE 절에 JOIN 조건 넣는다
SELECT 테이블1.칼럼명 테이블2.칼럼명 FROM 테이블1, 테이블2
WHERE 테이블1.칼럼명1 = 테이블2.테이블2;

# ANSI/ISO SQL 표준
# ON 절에 JOIN 조건 넣는다
SELECT 테이블1.칼럼명 테이블2.칼럼명 FROM 테이블1
INNER JOIN 테이블2 ON 테이블1.칼럼명1 = 테이블2.테이블2;
```

EQUI JOIN (등가 조인)

- **2개**의 테이블 간의 칼럼 값들이 **서로 정확하게 일치**하는 경우에 사용
- 대부분 PK, FK 관계 기반
- 컬럼명 SQL 앞에 테이블 명을 기술해줘야 함
- '**=**' 연산자에 의해서만 수행

```
SELECT PLAYER.PLAYER_NAME FROM PLAYER;
```

NON EQUI JOIN (비등가 조인)

- **2개**의 테이블 간에 칼럼 값들이 **서로 정확하게 일치하지 않는 경우** 사용
- '=' 연산자가 아닌 **BETWEEN, >, <=** 등 연산자 사용

- 대부분 NON EQUI JOIN 수행, 때로는 설계상의 이유로 수행이 불가능한 경우도 존재

```
SELECT E.ENAME, E.JOB, E.SAL, S.GRADE FROM EMP E, SALGRADE S
WHERE E.SAL BETWEEN S.LOSAL AND S.HSAL;
# E의 SAL의 값을 S의 LOSAL과 HSAL 범위에서 찾는 것이다.
```

제2장. SQL 활용

1) 연산자

집합 연산자

- **2개 이상**의 테이블에서 조인을 사용하지 않고 연관된 데이터를 조회할 때 사용
- **SELECT 절의 칼럼 수가 동일**
- SELECT 절의 **동일 위치에 존재**하는 칼럼의 데이터 타입이 **상호 호환**할 때 사용 가능

일반 집합 연산자

> **ALIAS**은 처음 테이블, **정렬**은 마지막 테이블 기준으로 표시한다

- **UNION**: 합집합 (중복 행 1개로), 정렬
- **UNION ALL**: 합집합 (중복 행도 표시), 정렬 X

  여러 질의 결과가 상호 배타적일 때 많이 사용

  개별 SQL문의 결과가 서로 중복되지 않는 경우, UNION과 결과 동일, 정렬 순서 차이 존재
- **INTERSECT**: 교집합 (중복 행 1개로)

  => **NOT IN 또는 NOT EXISTS**로 대체하여 처리 가능
- **EXCEPT**: 차집합 (중복 행 1개로), Oracle은 **MINUS** 사용

  => **NOT IN 또는 NOT EXISTS**로 대체하여 처리 가능
- **CROSS JOIN**: 곱집합(PRODUCT)

일반 집합 연산자를 SQL과 비교

- UNION 연산 => UNION 기능 (합집합)
- INTERSECTION 연산 => INTERSECT 기능 (교집합)
- DIFFERENCE 연산 => EXCEPT/MINUS 기능 (차집합)
- PRODECT 연산 => CROSS JON 기능 (곱집합)

순수 관계 연산자 => 관계형 DB를 새롭게 구현

- **SELECT, PROJECT, JOIN, DIVIDE**
- SELECT => WHERER 절로 구현
- PROJECT => SELECT 절로 구현
- NATRUAL JOIN => 다양한 JOIN으로 구현
- DIVIDE => 현재 사용 X

  {a,x}{a,y}{a,z} divdie {x,z} = {a}

---

2) FROM 절 JOIN 형태, USING, ON

FROM 절 JOIN 형태

1. **INNER JOIN**

   JOIN 조건에서 **동일한 값이 있는 행**만 반환, **USING이나 ON** 조건절 반드시 사용

2. **OUTER JOIN** => LEFT, RIGHT, FULL

   JOIN 조건에서 **동일한 값이 없는 행**도 반환 가능, **USING이나 ON** 조건절 반드시 사용

   ORACLE에서는 OUTER JOIN 구문을 **(+)**으로 처리 => ANSI 문장 변경 시, **(+) 안 붙은 쪽으로 JOIN한다**

   ```
   WHERE A.게시판ID = B.게시판ID(+) => FROM 게시판 A LEFT OUTER JOIN 게시글 B
   ```

   - **LEFT OUTER JOIN** (A, B => A 기준, B에 NULL)

     먼저 표기된 **좌측 테이블**에 해당하는 데이터를 읽은 후, 나중에 표기된 우측 테이블에서 JOIN 대상 데이터를 읽어 온다

     우측값에서 **같은 값이 없는 경우 => NULL 값**으로 채운다

   - **RIGHT OUTER JOIN** (A, B => B 기준, A에 NULL)

     LEFT OUTER JOIN의 반대

   - **FULL OUTER JOIN**

     좌우측 테이블의 모든 데이터를 읽어 JOIN하여 결과 생성, **중복 데이터 삭제**

3. **CROSS JOIN** => **카티시안 곱**(Cartesian Product)

   테이블 간 **JOIN 조건이 없는 경우** 발생하는 모든 데이터의 조합

   양쪽 집합의 **M*N**건의 **데이터 조합** 발생

   - **카티시안 곱**

     발생 가능한 모든 경우의 수의 행이 출력되는 것을 의미 => M*N

     조인 조건을 **생략**한 경우나 **부적합**할 경우 발생

   ```
   # table: 5건의 행 존재
   SELECT COUNT(*) FROM table CROSS JOIN table => 25
   ```

4. **NATURAL JOIN**

   두 테이블 간의 동일한 이름을 갖는 **모든** 칼럼들에 대해 **EQUI JOIN 수행**

   NATURAL JOIN이 명시되면 추가로 **USING, ON, WHERE 절에서 JOIN 조건 정의 X**

   SQL Serever 지원 X

USING 조건절

- **같은 이름을 가진 칼럼**들 중에서 **원하는** 칼럼에 대해서만 **선택적**으로 **EQUI JOIN 가능**
- JOIN 칼럼에 대해서 ALIAS나 테이블 이름과 같은 **접두사를 붙일 수 없다**

```
USING (T.STADUIM_ID S.STADIUM_ID) => USING (STADUIM_ID)
SELECT ... T.STADIUM_ID ... => SELECT ... STADUIM_ID ...
```

- SQL Server 지원 X

ON 조건절

- ON 조건절과 WHERE 조건절을 분리하면 이해가 쉬움, **ALIAS나 테이블명** 반드시 사용
- **칼럼명이 다르더라도 JOIN 조건을 사용 가능**

---

3) 계층형 질의

계층형 질의

- 테이블에 계층형 데이터가 존재하는 경우, 데이터를 조회하기 위해 사용
- **SQL Server계층형 질의문**

  CTE(Common Table Expression)를 **재귀** 호출, 계층 구조 전개

  앵커 멤버 실행 => 기본 결과 집합을 만들고, 이후 **재귀 멤버를 지속적으로 실행**
- **Oracle 계층형 질의문**

  WHERE 절은 모든 전개를 수행한 후, 지정된 조건 만족하는 데이터만 추출 (**필터링**)

- **PRIOR**: CONNECT BY 절에 사용, 현재 읽은 칼럼 지정
  - **순방향 전개** => 부모(상위코드) = PRIOR 자식(하위코드)

    계층 구조에서 부모 데이터에서 자식 데이터 (부모=>자식) 방향으로 전개
  - **역방향 전개** => PRIOR 부모(상위코드) = 자식(하위코드)

    자식 데이터에서 부모 데이터(자식 => 부모) 방향으로 전개
- **START WITH**: 계층 구조 전개의 **시작 위치 지정**, **루트 데이터 지정** (액세스)
- **CONNECT BY**: **다음**에 전재될 자식 데이터 지정, **규칙성 띄며 쭉 나감** (트리 형태의 구조)
- **ORDER SIBLINGS BY**: **같은 레벨 내에서 정렬**
- **LEVEL**: (Oracle) 루트 데이터 => **1**, 그 하위 데이터 => 2, 리프 데이터까지 +1

```
SELECT ~ FROM ~
START WITH ~ (시작 위치 조건/부모조건)
CONNECT BY PRIOR ~ (조인조건, 트리 연결 조건)
```

- **NOCYCLE**: 동일한 데이터가 전재되지 않음
- **CONNECT_BY_ISLEAF**: 해당 데이터가 리프 데이터 => 1, 아니면 => 0
- **CONNECT_BY_ISCYCLE**: 해당 데이터가 조상 => 1, 아니면 => 0 (**CYCLE 옵션 사용 시만 사용 가능**)
- **SYS_CONNECT_BY_PATH**: 루트 데이터부터 현재 전개할 데이터까지의 **경로** 표시
- **CONNECT_BY_ROOT**: 현재 전개할 데이터의 루트 데이터 표시, **단항 연산자**

---

4) 셀프 조인, 서브쿼리

셀프 조인 (Self Join)

- **한 테이블 내 두 칼럼이 연관 관계**가 있을 때 동일 테이블 사이의 조인
- **FROM** 절에 동일 테이블이 **2번** 이상 나타남
- 반드시 **테이블 별칭** 사용

```
SELECT 별칭1.칼럼명1, 별칭2.칼럼명1 FROM 테이블1 별칭1, 테이블2 별칭2
WHERE 별칭1.칼럼명2 = 별칭2.칼럼명3;
```

서브쿼리

- 하나의 SQL문 안에 포함되어 있는 또 다른 SQL문
- 알려지지 않은 기준을 이용한 검색에 사용, **괄호**로 감싸서 사용
- **ORDER BY**, 서브쿼리에 있는 **칼럼** 자유롭게 **사용 X**
- SELECT, FROM, WHERE, HAVING, ORDER BY, INSER-VALUES, UPDATE-SET 절에 사용 가능
- 메인 쿼리의 결과 서브쿼리로 제공 / 서브쿼리의 결과가 메인 쿼리로 제공 가능

  => **실행 순서 상황에 따라 달라짐**
- **단일/복수 행 비교 연산자**와 함께날 사용 가능
- **단일행 비교 연산자**: =, <, >, <> 등

  서브쿼리의 결과가 반드시 1건 이하
- **다중(복수)행 비교 연산자**: IN, ALL, ANY, SOME, EXISTS

  결과 건수 상관 X
  - **IN** (서브쿼리): 서브쿼리 결과 중 **하나만** 동일한 참
  - 비교연산자 **ALL** (서브쿼리): 서브쿼리 결과 **모두 동일**하면 참
    - < ALL: 최솟값 반환
    - '>' ALL: 최댓값 반환
  - 비교연산자 **ANY** (서브쿼리): 서브쿼리의 결과 중 **하나 이상 동일**하면 참
    - < ALL: 하나라도 크게 되면 참
    - '>' ALL: 하나라도 작게 되면 참
  - **EXISTS** (서브쿼리): 서브쿼리의 결과 하나라도 존재하면 참

    조건 만족하는 건이 여러 건이여서도 **1건만 찾으면 더 이상 검색 X**
- **스칼라 서브쿼리**: **한 행, 한 칼럼만 반환**하는 서브쿼리

  JOIN으로 동일한 결과 추출 가능

```
SELECT ENAME AS "이름", SAL AS "급여",
       (SELECT AVG(SAL) FROM EMP) AS "평균급여"
FROM EMP
WHERE EMPNO=1000;
```

서브쿼리 분류

- **반환 데이터에 따른 분류**
  1. **단일행 서브쿼리** (Single Row 서브쿼리)

     실행 결과 **1건 이하**, 단일행 비교 연산자와 함께 사용

     다중행 서브쿼리의 비교연산자 **사용 X**
  2. **다중행 서브쿼리** (Multi Row 서브쿼리)

     실행 결과 **여러 건**, 다중행 비교 연산자와 함께 사용

     단일행 서브쿼리의 비교연산자 **사용 가능**
  3. **다중 컬럼 서브쿼리** (Multi Column 서브쿼리)

     실행 결과 **컬럼 여러 개**

     메인쿼리의 조건절에 **여러 칼럼을 동시에 비교 가능**

     서브쿼리와 메인쿼리에서 비교하고자 하는 **칼럼 개수와 칼럼의 위치가 동일**해야 함

     SQL Server에서는 지원 X
- **동작 방식에 따른 분류**
  1. **비연관** 서브쿼리 (Un-Correlated sub query): 서브쿼리가 메인쿼리 칼럼 포함하지 **않는** 형태

     메인 쿼리에 값(서브쿼리가 실행된 결과) 제공 목적

     - **Access Subquery:** 제공자 역할
     - **Filter Subquery:** 확인자 역할
     - **Early Filter Subquery:** 데이터 필터링 역할
  2. **연관** 서브쿼리 (**Correlated** sub query): 서브쿼리가 메인쿼리 칼럼을 **포함하고 있는 형태**

     일반적으로 메인 쿼리 먼저 수행 => 읽혀진 데이터를 서브쿼리에서 조건이 맞는지 확인하고자 할 때 주로 사용

     => 서브쿼리에 ~값을 매번 가져와서 대입 => **성능 매우 좋지 않음**

---

5) 뷰

뷰 (가상 테이블)

- 테이블은 실제로 데이터를 가지고 있고, 뷰는 실제 데이터를 가지고 있지 않음
- 실행 시점에 SQL 재작성하여 수행

```
CREATE VIEW V_PLAYER_TEAM AS SELECT ~
DROP VIEW V_PLAYER_TEAM;
```

뷰 사용 장점

1. **독립성**: 테이블 구조가 변경되어도 뷰를 사용하는 응용 프로그램은 변경하지 않아도 됨
2. **편리성**: 복잡한 질의를 뷰로 생성 => 관련 질의를 단순하게 작성 가능
3. **보안성**: 직업의 급여정보와 같이 숨기고 싶은 정보가 존재할 때 사용

일반적인 뷰 => **정적 뷰** (Static View)

인라인 뷰 (Inline View) => **동적 뷰** (Dynamic View)

- SQL문이 실행될 때만 **임시적**으로 생성되는 **동적**인 뷰, DB에 해당 정보 저장 X
- **FROM** 절에서 사용되는 서브쿼리, ORDER BY를 사용 가능

```
SELECT *
FROM (SELECT ROWNUM NUM, ENAME FROM EMP) A
WHERE NUM < 5 ;
```

---

6) 절차형 SQL, 저장 모듈, PL/SQL, T-SQL

Oracle/SQL Server의 사용자 아키텍처 차이

- **Oracle**

  유저를 통해 DB에 접속하는 형태

  ID, PW 방식으로 인스턴스에 접속, 그에 해당하는 스키마에 오브젝트 생성 등의 **권한을 부여**받게 됩

- **SQL Serever**

  인스턴스에 접속하기 위해 **로그인 생성**

  인스턴스 내에 존재하는 다수의 DB에 연결 => 작업하기 위해 **유저를 생성**한 후, **로그인과 유저를 매핑**해 주어야 함

  - **Windows 인증 방식, 혼합 모드 방식**

절차형 SQL

- 일반적인 개발언어처럼 절차지향적인 프로그램을 작성할 수 있도록 제공하는 기능
- SQL문의 **연속적인** 실행이나 조건에 따른 **분기 처리** 이용 => 특정 기능 수행하는 **저장 모듈 생성**
- 선언부, 실행부, 예외 처리부로 이루어빔, **예외 처리부 생략 가능**
- Procedure, User Defined Function, Trigger

  => PL/SQL로 작성, PL/SQL엔진 처리, 작성자의 기준으로 **트랜잭션 분할 가능**

  => 프로시저 내에서 다른 프로시저 호출한 경우 => 호출 프로시저의 트랜잭션과는 별도로 **PRAGMA AUTONOMOUS_TRANSACTION** 선언하여 **자율 트랜젝션 처리 가능**

  - **Procedure**

    프로그램에서 특정 동작을 수행하는 이름이 주어진 일정 부분

  - **Usesr Defined Function** (사용자 정의 함수)

    절차형 SQL을 로직과 함께 DB 내에 저장해 놓은 명령문의 집합

    RETURN을 통해 반드시 하나의 값 반환

  - **Trigger**

    특정 테이블에 DML문 실행 => DB에서 자동 동작하도록 작성된 프로그램

    데이터의 **무결성**과 **일관성**을 위해 사용자 정의 함수 사용

    DB에 로그인하는 작업에도 정의 가능

Procedure	Trigger
EXECUTE 명령어로 실행	생성 후 자동 실행
TCL 사용 가능	TCL 사용 불가능

- **EXCUTE IMMEDIATE**: 오라클의 절차형 SQL로 DDL 또는 동적 실행 시 사용

저장 모듈 (Stored Module) => PL/SQL, LP/SQL, T-SQL

- PL/SQL 문장을 DB 서버에 저장 => 사용자와 애플리케이션 사이에서 공유 가능하게 만든 일종의 **SQL 컴포넌트 프로그램**
- **독립적으로 실**행되거나 다른 프로그램으로부터 실행될 수 있는 **완전한** 실행 프로그램

PL/SQL

- 오라클이 SQL을 절차적 프로그래밍적으로 확장시킨 언어
- PL/SQL 내부에서 테이블 생성 가능
- 임시 테이블로 잠깐 사용하기 위한 용도가 대다수
- 변수 대입 => **NAME:=aaa**
- 절차적 코드 => PL/SQL 엔진 처리 / 일반적인 SQL 문장 => SQL실행기 처리

PL/SQL 특징

- **Block 구조** => 각 기능별로 **모듈화** 기능, **통신량 줄일 수 있음**
- 변수, 상수 등 선언 => SQL 문장 간 값을 교환
- 일반 SQL 문장을 실행할 때 **WHERE절의 조건 등으로 대입 가능**
- IF, LOOP 등의 **절차형 언어** 사용 => 절차적인 프로그램 가능
- **DBMS 정의 에러/사용자 정의 에러** 정의 사용, **응용 프로그램 성능 향상**
- Oracle에 내장 => **호환성 좋음**

PL/SQL 명령어

- **DECLEAR**: BEGIN ~ END 절에서 사용될 변수와 인수에 대한 정의 및 데이터 타입 선언부
- **BEGIN ~ END**: 개발자가 처리하고자 하는 SQL 문과 여러 가지 비교문, 제어문이 필요한 로직 처리
- **EXCEPTION**: BEGIN ~ END 절에서 실행되는 SQL문이 실행될 때 에러가 발생하면 그 에러를 어떻게 처리할지 정의하는 예외 처리부

T-SQL

- 근본적으로 SQL Server를 제어하는 언어
- MS사에서 ANSI/ISO 표준의 SQL에 약간의 기능을 더 추가해 보완적으로 만든 것

```
CREATE Procedure schema_NAME.Procedure_name
```

제3장. SQL 최적화 기본 원리

1) 옵티마이저, 옵티마이저 엔진, SQL문 실행 순서, 실행 계획, SQL 처리 흐름도

옵티마이저

- SQL문에 대한 **최적의 실행 방법** 결정, 실행 계획 도출
- SQL문에 대한 **파싱 후 실행**
- **규칙/비용 기반 옵티마저**

1. **규칙 기반 옵티마이저** (RBO, Rule Based Optimizer)
    - 규칙(우선순위)에 따라 실행계획 생성
    - 제일 높은 우선순위: 위치를 가리키는 레코드 식별자(RID, Record Identifier/ROWID)(인덱스) 활용해 테이블 액세스
    - 제일 낮은 우선순위: 전체 테이블 스캔
    - 전체 테이블 액세스 방식보다는 **항상 인덱스를 사용하는 실행계획 생성**
    - **조인 순서 결정 시, 조인 칼럼 인덱스 존재 유무 판단**
        1. **조인 칼럼에 대한 인덱스가 양쪽에 존재하는 경우**: 우선 순위가 높은 테이블이 선행 테이블 (Driving Table)
        2. **한쪽에만 인덱스가 존재하는 경우**: 인덱스가 없는 테이블이 선행 테이블 (**NL Join 사용**)
        3. **모두 인덱스가 존재하지 않을 경우**: FROM 절의 뒤에 나열된 테이블이 선행 테이블로 선택 (**Sort Merge Join 사용**)
        4. **우선 순위가 동일한 경우**: FROM 절에 나열된 테이블의 **역순**으로 선행 테이블 선택
2. **비용 기반 옵티마이저** (CBO, Cost Based Optimizer)

- **처리비용**(예상되는 소요시간, 자원 사용량)이 **가장 적은 실행계획**을 선택하는 방식
- 규칙기반 옵티마이저의 단점 극복 위해 출현
- 인덱스를 사용하는 비용이 전체 테이블 스캔 비용보다 크다고 판단 시 => 전체 테이블 스캔을 수행하는 방법으로 실행 계획 생성 가능
- 통계정보, DBMS 버전/설정 정보 등의 차이 => 동일 SQL문도 서로 다른 실행계획이 생성 가능
- 그 밖의 다양한 한계들로 인해 실행계획의 예측 및 제어 어려움

옵티마이저 엔진

1. **질의 변환기**: 사용자가 작성한 SQL문을 처리하기에 보다 **용이한 형태 변환 모듈**
2. **비용 예측기**: 생성된 대안 계획의 **비용 예측 모듈**
3. **대안 계획 생성기**: 동일한 결과를 생성하는 다양한 **대안 계획 생성** 모듈

SQL문 실행 순서

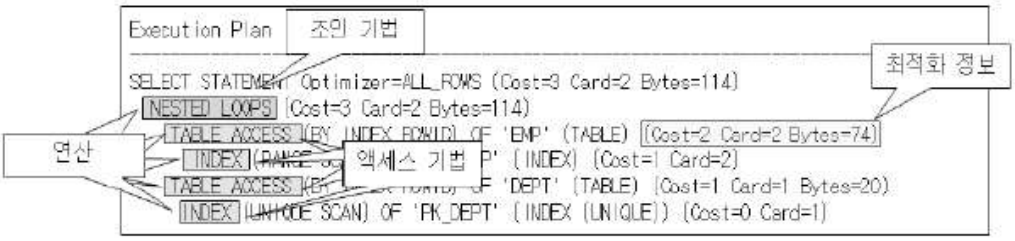
- 파싱(Parsing) > 실행(Execution) > 인출(Fetch)
- **파싱 (Parsing)**: SQL 문법 검사 및 구문 분석 작업
- **실행 (Execution)**: 옵티마이저의 실행 계획에 따라
- **인출 (Fetch)**: 데이터를 읽어 전송
- **CURSOR**(DB의 연결 포인트, 연결점) => **cursor 순서: 선언 > open > fetch > close**

  SQL 커서: Oracle 서버에서 할당한 전용 메모리 영역에 대한 포인터

  질의의 결과로 얻어진 여러 행이 저장된 메모리상의 위치, select문의 결과 집합 처리하는데 사용

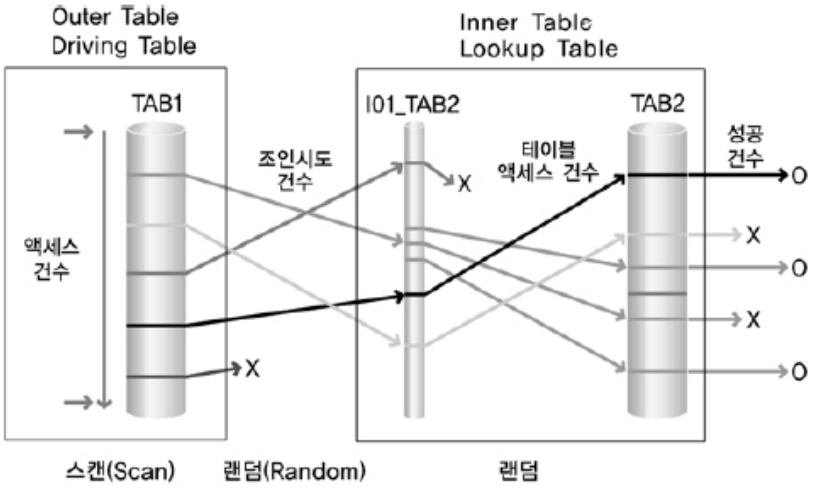
  모든 커서는 사용하기 전에 반드시 선언 필요

실행 계획



- SQL에서 요구한 사항을 처리하기 위한 절차와 방법
- **구성요소**: 조인 순서, 조인 기법, 액세스 기법, 최적화 정보(예상 비용)
- **실행 순서**: 위에서 아래로 (↓), 안에서 밖으로 (←) 읽음
- 실행 계획, 즉 실행 방법이 달라진다고 해서 결과 달라지지 않음

SQL 처리 흐름도



- SQL의 내부적인 처리 절차를 **시각적**으로 표현한 도표
- 조인순서, 액세스 기법, 조인 기법 표현 / 실행 시간 알 수 X
- 성능적인 측면도 표현 가능

---

2) 인덱스, 스캔 방법, IOT

인덱스

- 검색 조건에 부합하는 데이터를 효과적으로 검색할 수 있도록 돕는 기능
- 인덱스 키로 정렬, **조회** 속도 빠름
- **DML(INSERT, UPDATE, DELETE) 작업 효율 저하**
- 인덱스를 구성하는 칼럼 **이외의** 데이터가 **UPDATE**될 경우: 인덱스로 인한 부하 발생 X
- 테이블 전체 데이터 읽는 경우 => 인덱스 사용하지 않는 **FTS** 사용
- 인덱스 구성하는 칼럼들의 **순서** => 데이터 조회 시 **성능**적인 관점에서 매우 **중요한** 역할
- 인덱스 연산 => 인덱스 변형 => 인덱스 사용 X

1. **트리 기반 인덱스** (B-TREE 인덱스) => **기본**
    - DBMS에서 사용하는 가장 일반적인 인덱스, **OLTP 시스템 환경**에서 많이 사용
    - '='로 검색하는 **일치 검색**과 'BETWEEN'등의 **범위 검색** 모두 적합
    - 일반적) 테이블 내의 데이터 중 10% 이하의 데이터를 검색할 때 유리

    - **루트 블록, 브랜치 블록, 리프 블록** (Root Block, Branch Block, Leaf Block)으로 구성
    - **포인터** (Pointer): 루트 블록과 브랜치 블록의 키 값, 하위 블록 키 값의 범위 정보
    - **브랜치 블록** (Branch Block): 목적 => 분기
    - **리프 블록** (Leaf Block): 인덱스를 구성하는 칼럼의 값으로 정렬, 위치를 가리키는 레코드 식별자(RID, Record Identifier/ROWID)로 구성 (양방향 탐색 가능)
2. **비트맵 인덱스** (BTIMAP 인덱스)
    - 시스템에서 사용될 질의를 시스템 구현 시에 모두 알 수 없는 경우인 **DW 및 AD-HOC 질의 환경**을 위해 설계
    - 하나의 인덱스 키 엔트리가 많은 행에 대한 포인터를 저장하고 있는 구조
3. **클러스터형 인덱스** (CLUSTERED 인덱스)
    - **인덱스의 리프 페이지 == 데이터 페이지**

      => 테이블 탐색에 필요한 레코드 식별자가 리프 페이지에 없음
    - 클러스터형 인덱스의 리프 페이지를 탐색 => 해당 테이블의 모든 칼럼 값을 곧바로 얻을 수 있음
    - 리프 페이지의 모든 데이터는 인덱스 키 컬럼 순으로 물리적으로 정렬되어 저장

스캔 방법

1. **전체 테이블 스캔** (Full Table Scan)
    - 테이블의 **모든** 데이터를 읽으며 데이터 추출
    - 읽은 블록의 재사용성을 낮다고 판단 => 조건에 맞으면 추출, 아니면 버림
    - **많은 데이터 조회 시, 유리**
    - **사용되는 경우**
        - SQL문에 조건이나 조건 관련 인덱스가 없을 경우
        - 전체 테이블 스캔을 하도록 강제로 힌트를 지정하는 경우
        - 옵티마이저가 유리하다고 판단하는 경우
2. **인덱스 스캔** (Index Scan)
    - **인덱스를 구성하는 칼럼의 값을 기반**으로 데이터 추출
    - 인덱스를 읽어 ROWID를 찾고 해당 데이터를 찾기 위해 테이블 읽음
    - 일반적으로 인덱스 칼럼 순서로 정렬 출력
    - **적은 데이터 조회 시, 유리**
3. **인덱스 범위 스캔**: 인덱스를 이용해 **1건 이상의 데이터**를 추출하는 방식, 결과 없으면 1건도 반환 X
4. **인덱스 유일 스캔**: 인덱스가 **중복되지 않을 때 단 1건의 데이터**를 추출하는 방식, 검색 속도 가장 빠름
5. **인덱스 전체 스캔**: **리프 블록을 모두 읽으면서** 데이터를 추출하는 방식

IOT (Index-Organized Table)

- 인덱스키가 붙은 칼럼으로 구성된 테이블, **클러스터형 인덱스와 유사**
- 인덱스가 원래 테이블 참조 X

---

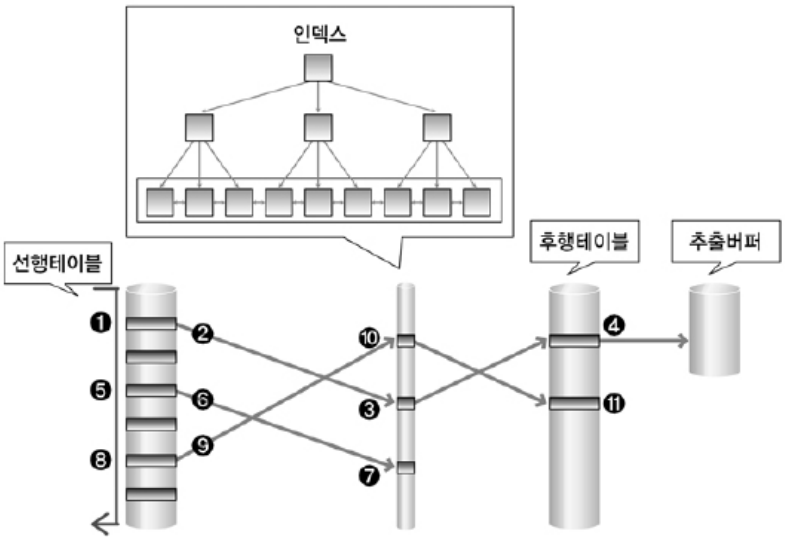
3) 조인, SEMI, ANTI

조인 (Join)

- 2개 이상의 테이블을 하나의 집합으로 만드는 연산
- SQL문에서 FROM 절에 2개 이상의 테이블이 나열될 경우 수행
- 선행 테이블로부터 **입력값**을 받아 처리
- 후행 테이블에 걸리는 **조인 조건**이 **성능**에 큰 영향을 미침

> 선행 테이블 (Driving Table / First Table) / 후행 테이블 (Driven Table / Second Table)

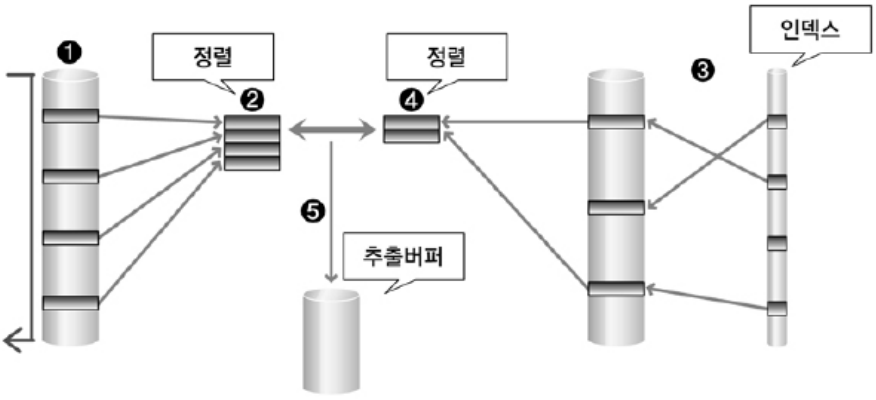
1. **NL Join** (Nested Loop Join) => 인덱스 사용



   - 2개의 테이블을 **중첩된 반복문**처럼 조인 수행, 선행 테이블의 데이터 하나씩 **순차적**으로 조인
   - **선행 테이블 처리 범위**(양) => **성능 결정**
   - 조인 칼럼에 **적당한 인덱스**가 있어서 **자연조인이 효율적**일 때 유용
   - **랜덤 엑세스** 위주 => **대용량 데이터 처리 시 불리**
   - 유니크 인덱스 이용 => **소량 테이블**을 **온라인** 조회하는 경우 유용
   - **OLTP의 목록 처리 업무**에서 많이 사용

> 반복문 **외부**(처음 테이블)에 있는 테이블 = 선행 테이블 또는 외부 테이블
>
> 반복문 **내부**(두번째 테이블)에 있는 테이블 = 후행 테이블 또는 내부 테이블

2. **소트 머지 조인** (Sort Merge Join)



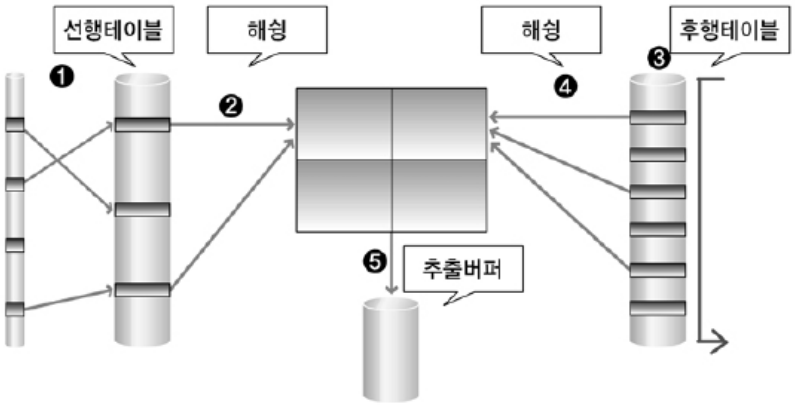
   - 조인 칼럼 기준 => 데이터를 정렬, 조인, 두 테이블을 **개별적**으로 **스캔** 후 조인

=> (**NL 조인**: 선행 테이블을 **랜덤 액세스 방식**으로 조회, 조인)

- **대용량 데이터 처리**할 때 주로 이용, 디스크에서 **정렬**(소트, Sort) 진행 => 성능상 불리함
- 인덱스 유무가 성능에 큰 영향을 주지 않음, **인덱스 존재하지 않은 경우에 사용 가능**

  => (**NL 조인**: 인덱스 구성에 크게 영향 받음)
- 선행 테이블의 개념이 중요하지 않음 **=>** (**NL 조인**: 중요함)
- **비동등** 조인에 대해서도 조인 가능 **=>** (**Hash 조인**: **동등** 조인(EQUI Join) 조건에서만 동작)
- **DW 등의 데이터 집계 업무**에서 많이 사용

3. **해시 조인** (Hash Join)



- NL Join의 **랜덤 액세스 문점**, Sort Merge Join의 **정렬 작업의 부담 문제** 해결 대안
- 조인 칼럼을 기준으로 동일한 해시 값을 갖는 데이터의 실제 값 비교 조인
- **두 테이블의 데이터 차이가 클 때 유리**, 테이블이 커서 **소트 부하가 심할 때 유리**
- **등가 조인**에서만 사용 가능
- 메모리에서 해시 테이블 생성 => **선행 테이블이 작을 때 유리** (**결과 행의 수가 적은** 테이블, 선택도가 **낮은** 테이블)
- 한쪽 테이블이 주 메모리의 가용 메모리에 담길 정도로 충분히 **작을** 때 => 효과적
- 해시 키 속성에 **중복 값이 적을 때** => 효과적
- Sort Merge Join보다 일반적으로 더 우수한 성능
- **Join Key 칼럼**으로 정렬 시 => Sort Merge Join이 더 우수한 성능
- **DW 등의 데이터 집계 업무**에서 많이 사용

SEMI, ANTI

- **SEMI**: 일부만 조인 (IN, EXISTS)
- **ANTI**: 부정 조인 (NOT IN, NOT EXISTS)