꿈꾸는 라이언

SQLD 핵심 요약 노트

SQLD 핵심 요약노트

발 행 일	2025년 7월 5일
저 자	꿈꾸는라이언
발 행 처	꿈꾸라 책방
출판 등록	2025년 7월 3일 (제 2025-000109호)
이 메 일	iuy64@naver.com
블 로 그	https://blog.naver.com/dreaming_ryan
판 매 가	7,900원

목 차

1. 데이터 모델링의 이해

1-1. 데이터 모델링의 이해 ····· 5

1-2. 데이터 모델과 성능 ····· 9

2. SQL 기본 및 활용

2-1. SQL 기본 ····· 12

2-2. SQL 활용 ····· 20

2-3. 관리 구문 활용 ····· 34

안녕하세요! **꿈꾸는라이언**입니다.
먼저 SQLD 요약본 구매를 감사드립니다! 😍

24년도에 SQLD 출제 항목에 개정이 되면서 개정판 **SQLD 요약 노트**를 준비했습니다.

SQLD는 객관식 50문항이 출제돼요. 총 1시간 30분 응시하게 되는데요. **총 점 60점 이상**이 되어야 합격이에요!

과목	문항 수	배점	과락기준	검정시간
데이터 모델링의 이해	10	20 (문항당 2점)	**8점 미만**	90분
SQL 기본 및 활용	40	80 (문항당 2점)	**32점 미만**	

1과목의 경우, 데이터 모델링 과목은 크게 변동 사항이 있지 않아요.
데이터 모델과 성능 과목이 바뀌었어요. 성능에 대한 내용을 많이 내려놓고 정규화에 초점을 두고 문제가 많이 출제 될 것이라고 예상돼요.

2 과목 SQL 기본 및 활용에서는 24년 개정이후에 변동사항이 꽤 있어요.
튜닝과 관련된 일부 요소들이 완전히 제외되고, 관리 구문만 남게 되었어요.
Top N 쿼리, PIVOT, UNPIVOT, 정규 표현식이 추가되었어요. 해당 부분은 어려울 수 있으나 어렵게 때문에 오히려 시험에서는 기본적인 것으로 시험에서 출제될 것으로 보여요. 그래서, 기본적인 부분만 잘 준비하면 문제 없을 거에요!

변경 전	
주요항목	**세부항목**
SQL 기본	• 관계형 데이터베이스 개요
	• DDL
	• DML
	• TCL
	• WHERE 절
	• FUNTION
	• GROUP BY, HAVING 절
	• ORDER BY 절
	• 조인
SQL 활용	• 표준조인
	• 집합연산자
	• 계층형 질의
	• 서브쿼리
	• 그룹 함수
	• 윈도우 함수
	• DCL
	• 절차형 SQL
SQL 최적화 기본 원리	• 옵티마이저와 실행계획
	• 인덱스 기본
	• 조인 수행 원리

⇒

변경 후	
주요항목	**세부항목**
SQL 기본	• 관계형 데이터베이스 개요
	• SELECT 문
	• 함수
	• WHERE 절
	• GROUP BY, HAVING 절
	• ORDER BY 절
	• 조인
	• 표준 조인
SQL 활용	• 서브 쿼리
	• 집합 연산자
	• 그룹 함수
	• 윈도우 함수
	• Top N 쿼리
	• 계층형 질의와 셀프 조인
	• PIVOT 절과 UNPIVOT 절
	• 정규 표현식
관리 구문	• DML
	• TCL
	• DDL
	• DCL

출처: 데이터자격시험 공식 웹사이트

① 데이터 모델의 이해

■ 데이터 모델링
- 고객의 비즈니스 프로세스를 이해하고, 규칙을 정의하여, 데이터 모델로 표현함

■ 데이터 모델링의 중요성 및 유의점

중복 (Duplication)	여러 장소에 같은 정보를 저장하지 않아야 함
비유연성 (Inflexibility)	작은 업무 변화에 의해 데이터 모델이 자주 변경되지 않아야 함 데이터 정의를 사용 프로세스와 분리
비일관성 (Inconsistency)	데이터 중복 없어도 비일관성이 발생 가능 데이터 간 상호 연관 관계에 대해 명확하게 정의 해야 함

※ 비일관성: 신용 상태에 대한 갱신 없이 고객의 납부 이력을 갱신하는 경우와 같이 서로 연관된 다른 데이터와 모순된다는 고려 없이 데이터를 수정할 수 있기 때문

■ 데이터 모델링의 3단계

추상화 높음 (진행 단계)

개념적 모델링	업무적 & 전사적 관점에서 모델링 복잡 X 기술적 용어 X 중요한 부분 위주로 **엔터티, 속성 도출**하여 ERD 작성
논리적 모델링	**식별자**를 도출, 정의하고 **릴레이션, 관계, 속성** 등을 표현 **정규화**를 통해, 모델의 독립성, 재사용성 높임 특정 데이터베이스 모델에 종속
물리적 모델링	**데이터베이스 실제 구축** DB 시스템에 테이블, 인덱스 함수 등 생성 성능, 보안, 가용성 등을 고려

■ 데이터 모델링 특징
- **추상화 (Abstraction):** 현실세계를 간략하게 표현 (공통적인 특징을 간략하게 표현)
- **단순화 (Simplification):** 누구나 쉽게 이해 가능 (복잡한 문제 X)
- **명확성 (Clarity):** 모호하지 않고, 명확한 의미 해석 (한 가지 의미를 가짐)

■ 데이터 모델링 요소 (3층 스키마)

3층 스키마
- 3단계 계층으로 분리하여 데이터베이스의 독립성을 확보하기 위한 방법
- 사용자 / 설계자 / 개발자가 DB를 보는 관점에 따라서, DB를 기술하고 관계를 정의

사용자 관점 외부 스키마	업무상 관련있는 접근 관련 DB의 View를 표시 여러 사용자가 보는 개인적 DB 스키마	논리적 독립성
설계자 관점 개념 스키마	통합된 모든 사용자 DB 구조 통합 데이터베이스 구조 전체 DB의 규칙, 구조 표현	
개발자 관점 내부 스키마	물리적 저장 구조 (실질적 데이터 저장 데이터 저장 구조 및 레코드 구조 필드, 인덱스 정의	물리적 독립성

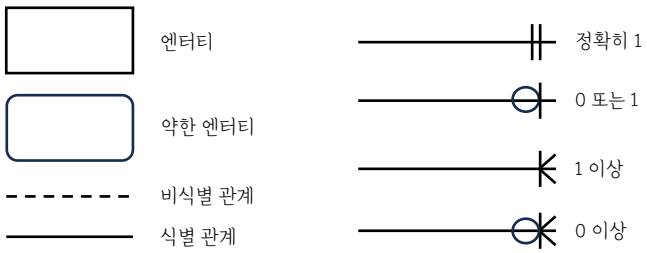
※ **논리적 독립성:** 개념스키마의 변경이 외부 스키마에 영향을 끼치지 않음
※ **물리적 독립성:** 내부스키마의 변경이 개념 스키마에 영향을 끼치지 않음

■ 데이터 모델링 3가지 관점

1. **데이터 관점**
   - 데이터가 어떻게 저장되고, 접근되고, 관리되는지를 정의하는 단계
2. **프로세스 관점**
   - 시스템이 어떤 작업을 수행하며, 이러한 작업들이 어떻게 조직되고 조정되는지를 정의하는 단계
   - 데이터가 시스템 내에서 어떻게 흐르고 변환되는지에 대한 확인
3. **데이터와 프로세스 관점**
   - 데이터 관점과 프로세스 관점을 결합하여 시스템의 전반적인 동작을 이해하는 단계
   - 특정 프로세스가 어떤 데이터를 사용하는지, 데이터가 어떻게 생성되고 변경되는지를 명확하게 정의

■ 데이터 모델링 3가지 요소
- **대상** (Entity) : 업무가 관리하고자 하는 대상(객체)
- **속성** (Attribute) : 대상들이 갖는 속성(하나의 특징으로 정의될 수 있는 것)
- **관계** (Relationship) : 대상들 간의 관계

■ ERD (Entity Relationship Diagram)
- 데이터 모델링 표준 (엔터티-엔터티간 관계 정의 방법)
- 중요한 엔터티는 왼쪽 상단에 배치
- 복잡하지 않고, 이해하기 쉬워야 함

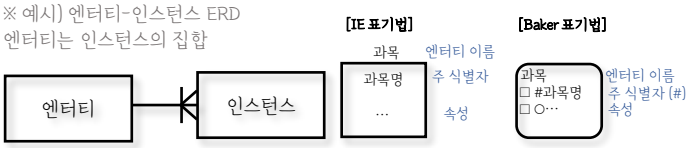


■ ERD 작성 절차

① 엔터티 도출 및 그림	업무에서 관리해야 하는 집합을 도출
② 엔터티 배치	중요한 엔터티는 왼쪽 상단에 배치
③ 엔터티 관계 설정	엔터티 간 관련성 파악
④ 관계명 서술	엔터티 간의 행위
⑤ 관계 참여도 표현	한 엔터티와 다른 엔터티 간의 참여하는 관계의 수
⑥ 관계 필수 여부 표현	반드시 존재해야 하는지 여부 표시

■ 엔터티
- 업무에 필요하고 유용한 정보를 저장하고 관리하기 위한 집합적인 것 (보이지 않는 개념 포함)
- 정보가 저장될 수 있는 장소, 사람, 사건, 개념, 물건 등
- 엔터티는 다른 개체와 확연히 구분되는 특성을 가짐 → 모델의 독립성 향상

※ 예시) 엔터티-인스턴스 ERD
엔터티는 인스턴스의 집합



엔터티	인스턴스
과목	수학
	과학
강사	홍길동
	김철수

■ 엔터티 특징

식별자	유일한 식별자가 있어야 함 ex) "고객"의 회원 ID, "계좌"의 계좌 번호
인스턴스 집합	2개 이상의 인스턴스가 있어야 함 ex) "계좌" 엔터티의 2개 이상의 계좌 정보 (인스턴스)
속성	반드시 속성을 가지고 있어야 함 ex) "고객"의 이름, 생년월일 등
관계	다른 엔터티와 최소 1개 이상의 관계가 있어야 함 ex) "고객"은 "계좌 " 를 개설
업무	엔터티는 업무에서 관리되어야 하는 집합이어야 함 ex) "고객", "계좌 "

※ 릴레이션은 DB 테이블을 뜻한다.
※ Relationship은 릴레이션 간의 관계
※ 인스턴스는 릴레이션이 가질 수 있는 값 (즉, 테이블 데이터의 행의 개수)

■ **엔터티 명명 유의사항**
- 현업 업무에서 사용하는 용어 사용
- 약어 사용 금지
- 단수명사 사용
- 모든 엔터티에서 유일한 이름 부여
- 생성되는 의미대로 자연스럽게 부여

■ **데이터 모델 표기법**
1976년 피터첸이 Entity Relationship Model 개발
- IE, Baker 기법이 많이 쓰임
- 엔터티, 관계, 속성으로 이뤄짐
- 엔터티와 엔터티 간의 관계를 시각적으로 표현한 다이어그램

■ **엔터티 종류**
- 두 가지 기준으로 엔터티들을 분류할 수 있음
- 물리적 형태의 존재 여부에 따라 "유형 엔터티" "개념 엔터티" "사건 엔터티"

유형 엔터티	물리적 형태 ex) 사원, 고객
개념 엔터티	개념적 형태 (물리적 형태 없음) ex) 조직, 보험 상품
사건 엔터티	실행하면서 생성됨 ex) 주문, 수수료 청구

- 발생 시점에 따라 "기본 엔터티" "중심 엔터티" "행위 엔터티"

기본 엔터티	다른 엔터티의 도움없이 독립적으로 생성 ex) 고객, 부서, 사원
중심 엔터티	키와 행위의 중간 (기본 엔터티 로부터 발생) ex) 계약, 주문
행위 엔터티	2개 이상의 엔터티 로부터 발생 양이 많거나 자주 변경되는 엔터티 ex) 주문 이력, 취소 이력

■ **속성 (Attribute)**
- 엔터티가 가지는 항목
- 인스턴스의 구성요소
- 중복된 값이 존재할 수 있음
- 업무에서 필요로 하는 인스턴스로 관리하고자 하는 의미상 분리되지 않는 최소의 데이터 단위

※ 한 개의 엔터티는 "2개 이상"의 인스턴스의 집합
※ 한 개의 엔터티는 "2개 이상"의 속성을 가짐

■ **속성 특징**
- 한 개의 속성은 1개의 속성 값을 가짐
- 업무에서 관리되는 정보
- **주식별자에게 함수적 종속성**을 가져야 됨

■ **함수적 종속성**
- 한 속성의 값이 다른 속성의 값에 종속적인 관계를 가지는 특징
- 어떤 A 속성의 값에 의해 B가 유일하게 결정된다면, B는 A에 함수적으로 종속되었다고 표현
  A → B라고 수식으로 표현

- **완전 함수적 종속**
- 특정 컬럼이 기본키에 대해 완전히 종속될 경우를 뜻함
- PK 를 구성하는 컬럼이 2개 이상일 경우, **PK 값 모두**(복합키 전체)에 의해 다른 속성이 결정

- **부분 함수적 종속**
- PK를 구성하는 컬럼이 2개 이상일 경우, **PK 값 일부**(복합키 일부 컬럼) 만으로 다른 속성이 결정

ex) 주문 상세 테이블에서 (ORDER_ID, PRODUCT_ID)를 합쳐서 복합 키(후보키)로 설정

QUANTITY는 복합키의 전부(ORDER_ID+PRODUCT_ID)가 있어야만 유일하게 결정 → 완전 함수적 종속

PRODUCT_NAME은 (ORDER_ID, PRODUCT_ID) 전체가 아니라, (PRODUCT_ID) 일부만으로도 알 수 있기에 부분 함수적 종속

ORDER_ID	PRODUCT_ID	QUANTITY	PRODUCT_NAME
1001	P001	3	노트북
1002	P003	2	이어폰

■ **속성 분류**
- 두 가지 기준으로 엔터티들을 분류할 수 있음
- 특성에 따라 "기본 속성" "설계 속성" "파생 속성"

기본 속성	업무로부터 추출된 일반적인 속성 ex) 이름, 회원ID
설계 속성	업무를 규칙화하기 위해 새로 만들거나 변형된 속성 유일한 값을 부여 ex) 일련번호, 제품번호
파생 속성	다른 속성에 의해 영향을 받아 만들어진 속성 빠른 성능을 낼 수 있도록 원래의 속성을 계산 ex) 합계, 평균

- 분해 여부에 따라 "단일 속성" "복합 속성" "다중값 속성"

단일 속성	하나의 의미로 구성 ex) 직원 테이블의 이름, 학생 테이블의 학번
복합 속성	여러 의미가 있음 ex) 주소 속성은 도시, 우편번호, 도로명으로 나뉠 수 있음
다중값 속성	속성에 여러 값을 가질 수 있음 별도의 엔터티로 분해됨 (별도의 테이블) ex) 전화번호 속성은 한 사람이 여러 개를 가질 수 있음

- 엔터티 구성방식에 따라 "PK" "FK" "일반 속성"

PK(Primary Key, 기본키)	인스턴스를 식별할 수 있는 속성
FK(Foreign Key, 외래키)	다른 엔터티와의 관계에서 포함된 속성
일반 속성	엔터티에 포함되어 있고 PK/FK에 포함되지 않는 속성

■ **도메인 (Domain)**
- 하나의 속성이 가질 수 있는 모든 원자들의 집합
- 속성에 대한 데이터 타입, 크기, 제한 사항

ex) 나이 INT CHECK (나이 >= 0 AND 나이 <= 150) -- 나이 속성의 도메인
정수이며, 0 이상 150 이하의 값을 가지도록 제한 → 데이터 무결성을 유지

■ **속성 명명 유의사항**
- 해당업무에서 사용하는 이름 부여
- 서술식 속성명은 사용 금지
- 약어 사용 금지
- 구체적으로 명명하여 데이터 모델에서 유일성 확보

■ 관계 (Relationship)

- **인스턴스 사이의 논리적인 연관성**

ex) 강사 – 가르친다(관계) – 수강생

존재 관계	한 개체가 다른 개체의 존재에 의존 ex) Class, Student – 반이라는 개체가 존재해야 학생이 존재
행위 관계	행위나 동작을 통해 두 개체가 연결 ex) 고객은 "주문한다"는 행위를 통해 주문을 생성

※ ERD에서는 존재 관계와 행위 관계를 구분하지 않음

- 필수적 관계 & 선택적 관계

필수적 관계	표현: | 반드시 하나가 있어야 함 ex) "고객" 엔터티가 있어야, "계좌" 엔터티 개설 가능
선택적 관계	표현: O 없을 수도 있는 관계 ex) "고객"이 있어도 , "계좌" 는 없을 수 있음

- 관계 차수: 2개 엔터티 간 관계에 참여하는 수
- 카디널리티 (Cardinality): 하나의 릴레이션에서 튜플의 전체 개수

1 대 1 관계	완전 1 대 1	1개 엔터티에 관계되는 엔터티 관계가 1개가 반드시 존재
	선택적 1 대 1	관계가 1개이거나 없을 수 있음
1 대 N 관계	엔터티에 하나의 행에 다른 엔터티에 관련 행이 여러 개 있음 ex) "고객" 테이블에 고객 한명은 "계좌" 여러 개 보유 가능	
M 대 N 관계	2개의 엔터티가 서로 여러 개의 관계를 가짐 이런 경우, 1 대 N 관계로 풀어야 함. ex) "학생"과 "강의"의 관계에서 학생은 여러 강의를 수강할 수 있고, 강의도 여러 학생이 들을 수 있음. 수강이라는 엔터티를 추가 도출하여 학생(1)-수강(n), 강의(1)-수강(n)으로 해소할 수 있음	

■ 관계의 구성

1. 관계 이름
2. 차수(Cardinality)
    - 1 대 1 인지, 1 대 N 인지 등
3. 선택성(Optionality)
    - 필요한 관계인지, 선택적인 관계인지

■ 관계의 페어링

페어링은 "두 엔터티(혹은 그 이상의 엔터티)가 실제로 연결된 구체적 한 인스턴스"를 뜻함

ex) 학생 엔터티와 강의 엔터티가 실제로 연결될 때, "**어느 학생이, 어느 강의를, 언제, 어떤 성적으로 수강했는지**"와 같은 개별적 기록(인스턴스)이 페어링이 됨

"학생 A(학번 2023001)가, 강의 B(강의코드 CS101)를, 2023년 1학기에 수강했고, 성적은 'A+' 받았다."

이게 한 개의 페어링(실제 연결 인스턴스)이고, 이런 "특정 학생 - 특정 강의"의 구체적 연결(수강기록)을 모아서 **관계**라고 함

결국, 관계는 **페어링의 집합**을 의미

■ 차수, 페어링 차이

- 차수(M:N, 1:N, 등)

"어떤 엔터티 A가 엔터티 B와 맺을 수 있는 연결의 **패턴이나 규칙**"
개념적으로 **"A는 B를 최대 몇 개 연결할 수 있는가?"**를 보여주는 것
예) 학생과 강의가 M:N 관계다 → "다수 학생 : 다수 강의" 가능

- 페어링(Pairing)

이 **차수(관계 규칙)**를 **실제 인스턴스 단위**로 구현한 "개별 연결" 정보
예) "학생 A가 강의 B를 2023-1학기에 수강하고 성적은 A+"라는 한 건의 수강기록

즉,
차수 = 엔터티 간 **연결 방식**을 추상적으로 표시 (1:1, 1:N, M:N 등)
페어링 = 해당 연결이 **구체적으로 어떻게 매핑**되었는지 나타내는 실질적 레코드(인스턴스)

■ 관계 체크사항

1. 두 개의 엔티티 사이에 관심있는 **연관규칙이 존재**하는가?
2. 두 개의 엔터티 사이에 **정보의 조합**이 발생되는가?
3. 관계연결에 대한 **규칙이 서술**되어 있는가?
4. 관계**연결을 가능하게 하는 동사**가 있는가?

■ 관계의 표기법

- 관계명(Membership): 관계의 이름
- 관계차수(Cardinality): 1:1, 1:M, M:N (2개 엔터티 간 관계에 참여하는 수)
- 관계선택사양(Optionality): 필수관계, 선택관계

■ 강한 개체 & 약한 개체

- **강한 개체:** 다른 개체에게 지배/종속 되지 않음 (독립적으로 존재하는 개체)
- **약한 개체:** 다른 개체에 의존함 (다른 개체의 존재에 약한 개체의 존재가 달려있음

■ UML 에서의 관계

연관관계 (Association)	존재적 관계에 해당 실선으로 표현 (강한 연결관계) 강한 개체의 기본키를 다른 개체의 기본키 중 하나로 공유 기본 키를 공유받은 다른 개체는 약한 개체가 됨 독립적으로 존재 불가능 ——— ex) 소속된다
의존관계 (Dependency)	행위적 관계에 해당 점선으로 표현 (약한 연결관계) 강한 개체의 기본키를 다른 개체의 일반키로 공유 기본 키를 공유받은 다른 개체는 약한 개체가 아님 독립적으로 존재 가능 ---------- ex) 주문한다

■ 'ERD'와 '클래스 다이어그램'의 관계차이

ERD에서는 존재적 관계와 행위적 관계를 구분하여 표현하지 않는다.
클래스 다이어그램에서는 연관관계와 의존관계로 구분하여 표현한다

■ 식별자 (Identifiers)

- 엔터터 내에 인스턴스들을 구분할 수 있는 구분자
- 엔터티를 대표하는 "속성"
- 하나의 엔터티는 하나의 식별자를 반드시 가져야 함
- 식별자는 논리 데이터 모델링 단계에서 사용하고 키는 물리 데이터 모델링 단계에서 사용

※ 하나의 식별자가 하나의 속성을 뜻하는 건 아님

■ 식별자의 4가지 분류

주 식별자 ⟷ 보조 식별자 (대표성 여부)

주 식별자
- 엔티티 내에서 각 인스턴스를 구분할 수 있는 구분자이며, 타 엔티티와 참조관계를 연결할 수 있는 식별자
- ex) 부서번호, 사번, 주문번호

보조 식별자
- 엔티티 내에서 각 인스턴스를 구분할 수 있는 구분자이나, 대표성을 갖지 못해 참조관계 연결 불가
- ex) user table의 email

내부 식별자 ⟷ 외부 식별자 (스스로 생성 여부)

내부 식별자
- 엔터티 내부에서 스스로 만들어지는 식별자
- ex) user table의 user_id

외부 식별자
- 타 엔터티와의 관계를 통해 생성 되는 식별자
- Foreign Key(FK) 역할을 함
- ex) order table의 user_id

단일 식별자 ⟷ 복합 식별자 (속성의 수)

단일 식별자
- 하나의 속성으로 구성된 식별자
- ex) 학생 테이블의 student_id

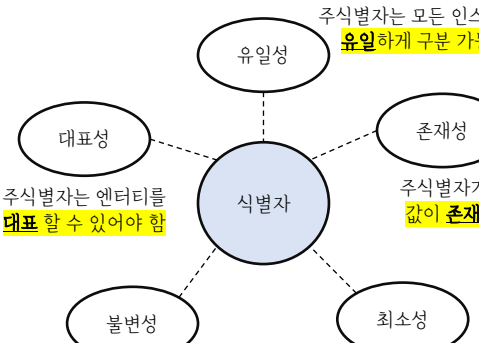
복합 식별자
- 둘 이상의 속성으로 구성된 식별자
- ex) 수강 관리 시스템의student_id, course_id 두 개 이상 컬럼으로 기본키가 됨

본질 식별자 ⟷ 인조 식별자 (대체 여부)

본질 식별자
- 업무로 인해 만들어지는 식별자
- ex) 현실에서 이미 부여된 주민번호

인조 식별자
- 인위적으로 만들어지는 식별자
- ex) 시스템에서 자동으로 생성된 식별자 (예: 자동 증가 ID 값)

■ 주 식별자 5가지 특징



■ 주 식별자 도출 기준

- 자주 이용하는 속성을 주식별자로 지정해야 함
- 이름이나 명명된 호칭, 길이가 일정하지 않은 내용 등은 주식별자로 지정하지 않아야 함
- 복합으로 주식별자를 구성할 경우 너무 많은 속성이 포함되지 않도록 함

■ 인조 식별자 특징

1. 최대한 범용적인 값을 사용해서 만든 식별자
2. 유일한 값을 만들기 위해 사용
3. 하나의 인조 식성으로 대체 불가능
4. 편의성 ·단순성 확보를 위해 사용
5. 의미의 체계화를 위해 사용
6. 내부적으로만 사용

■ Database 키 종류

기본키 (Primary Key)	후보키 중에서 엔터티를 대표할 수 있는 키
후보키 (Candidate Key)	유일성과 최소성을 만족하는 키
슈퍼키 (Super Key)	유일성은 만족하지만 최소성을 만족하지 않는 키
대체키 (Alternate Key)	여러 개의 후보키 중에서 기본키를 선정하고 남은 키
외래키 (Foreign Key)	타 테이블의 기본키 필드를 가리키는 것으로 참조무결성을 확인하기 위해 사용되는 키 (허용된 값만 저장하기 위해서 사용)

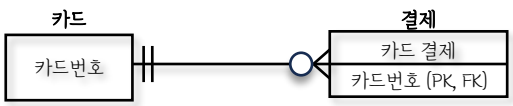
※ 참조무결성: 한 테이블의 외래 키가 참조하는 다른 테이블의 기본 키가 반드시 존재하도록 보장하는 규칙

■ 식별자 관계 (강한 연결 관계, 실선 표시)

부모의 주식별자를 자식 엔터티의 주식별자로 상속되는 경우를 식별자 관계라고 함

- Null값이 오면 안되므로 반드시 부모 엔터티가 생성되어야 자기 자신의 엔터티가 생성되는 경우

ex) 카드결제는 카드가 있어야 생기는 엔터티. 이런 경우를 부모관계라고 표현
카드의 기본 키(카드번호)를 결제 엔터티의 기본 키로 사용한 경우를 식별자 관계



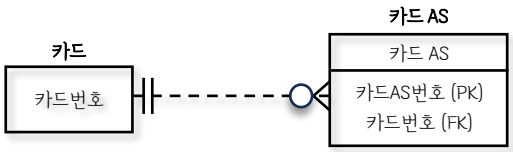
※ 부모 엔터티 주 식별자를 자식 엔터티 주 식별자로 쓸 때, 단일 식별자일 필요는 없음

■ 비 식별자 (약한 연결 관계, 점선 표시)

부모 엔터티로부터 속성을 받았지만, 자식 엔터티의 주식별자로 사용하지 않고 일반적인 속성으로 사용하는 경우를 비식별자 관계라고 함

- 부모 없는 자식이 생성될 수 있는 경우
- 엔터티 별로 데이터의 생명주기(Life Cycle)를 다르게 관리할 경우

ex) 카드AS는 카드가 있어야 생기는 엔터티
식별관계처럼 카드번호를 기본키로 사용하지 않고, 카드AS번호를 기본 키로 사용
부모 엔터티 주 식별자(카드번호)를 그냥 속성으로만 사용



■ 식별자 연결을 고려해야 하는 상황

- 자식 엔터티가 반드시 부모 엔터티에 종속되어야 할 경우
- 자식 주식별자 구성에 부모 주 식별자가 필요한 경우
- 상속받은 주 식별자 속성을 타 엔터티와의 관계에서도 사용해야 하는 경우
- 강한 연결관계를 표현해야 할 경우

■ 비식별자 관계를 고려해야 하는 상황

- 자식 주 식별자 구성을 독립적으로 구성해야 하는 경우
- 자식 주식별자 구성에 부모 주식별자가 부분적으로 필요한 경우
- 상속받은 주 식별자 속성을 타 엔터티에 차단 필요한 경우 (자식 엔터티 외 전파 X)
- 부모 쪽의 관계참여가 선택관계인 경우
- 약한 연결관계가 필요한 경우

② 데이터 모델과 성능

■ 정규화

엔터티를 분해(구조화)하는 과정

1. 데이터 **중복을 제거하고, 데이터 모델의 독립성 확보**
2. 데이터 **이상현상을 줄이기 위한** 데이터베이스 설계 기법

※ 이상현상

정규화를 하지 않아 발생하는 현상

삽입 이상	인스턴스가 삽입(추가)될 때, 정의되지 않아도 될 속성까지 입력해야 하는 현상 즉, 불필요한 정보까지 입력해야 하는 현상 ex) 사원+부서 엔터티가 합쳐져 있을 때 새로운 사원이 추가될 때, 정해지지 않은 부서 정보를 임의 값 혹은 null로 넣어야 함
갱신 이상	데이터를 수정할 때, 일부만 수정되어 데이터가 불일치 하는 현상 ex) 학생+수강 엔터티가 합쳐져 있을 때 학번 / 이름 / 학과 / 과목 1001 / 민수 / 경영 / 통계 1001 / 민수 / 경영 / 영어 이름이 변경되면 모든 이름이 수정되어야 함 첫번째 데이터의 이름만 철수로 수정된다면 데이터 불일치가 발생
삭제 이상	데이터베이스에서 특정 데이터를 삭제할 때 의도하지 않은 다른 정보까지 함께 삭제되는 현상 ex) 사원+부서 엔터티가 합쳐져 있을 때 특정 부서의 유일한 사원정보를 삭제하면 해당 부서정보도 함께 삭제됨. 즉, 사원을 삭제했을 뿐인데 부서 정보까지 손실되는 문제가 발생함

- 특징
  - 논리 데이터 모델링 수행 시점에 고려된다
  - 결과적으로, 데이터 일관성 · 최소한의 중복 · 데이터 유연성 확보가 목적
  - 제 1 정규화 ~ 제 5 정규화까지 존재 (보통, 제 3 정규화까지 진행)

■ 정규화 단계

제 1 정규화 (1NF)

테이블 컬럼이 **원자성**을 갖도록 테이블을 분해하는 단계

→ 하나의 속성이 하나의 값을 갖는 특징

ex) 구매 테이블

이름	품목
민수	샴푸, 린스

→

이름	품목
민수	샴푸
민수	린스

상품에 여러 값이 있으면 여러 인스턴스로 분리

제 2 정규화 (2NF)

제 1 정규화를 진행한 테이블에 **완전 함수 종속**을 만들도록 테이블을 분해하는 단계

※ 완전 함수 종속

기본키를 구성하는 모든 컬럼의 값이 다른 컬럼을 결정 짓는 상태

기본키의 부분 집합이 다른 컬럼과 1 대 1 대응 관계를 갖지 않는 상태

PK가 2개 이상 일 때 발생하며, PK 의 일부와 종속되는 관계가 있을 때 분리

ex) 학생 수업 등록 테이블 PK: (학번, 과목코드)

1. 학생이름은 학번에만 종속적
2. 과목명은 과목코드에만 종속적

학번	과목코드	학생이름	과목명
1	A123	김민수	경영학
1	B567	김민수	심리학

→

학번	과목코드
1	A123
1	B567

학번	학생이름
1	김민수

과목코드	과목명
A123	경영학
B567	심리학

- 학생이름이나 과목 정보를 한 번만 저장하므로 공간이 절약
- 이름 변경 시 한 곳만 수정하면 되므로 데이터 일관성이 유지
- 특정 과목이나 학생 정보를 쉽게 추가하거나 삭제

제 3 정규화 (3NF)

제 2 정규화를 진행한 테이블에 **이행적 종속**을 없애도록 테이블을 분리

A → B, B → C의 관계가 성립할 때,
A → C가 성립되는 것 (보통 A가 PK)
(A, B) (B, C)로 분리하는 것이 정규화

ex) 학생 테이블

즉, 기본키가 아닌 열이 다른 키가 아닌 열에 종속되는 상황

학번	학생이름	학과코드	학과이름	학과장
1	김철수	CS01	경영학	박교수
2	이영희	CS01	경영학	박교수
3	박지만	EE01	전자공학	김교수

- 학번 → 학과코드 → 학과이름
- 학번 → 학과코드 → 학과장

제 3 정규화 시 두개의 테이블로 분리

학번	학생이름	학과코드
1	김철수	CS01
2	이영희	CS01
3	박지만	EE01

학과코드	학과이름	학과장
CS01	경영학	박교수
EE01	전자공학	김교수

아래부터는 예제로는 잘 출제되지 않고,
문맥만 이해하시고 넘어가셔도 됩니다

BCNF (Boyce-Codd Normal Form) 정규화

- 제 3 정규화를 강화한 형태
- 모든 결정자가 후보 키여야 함
- 즉, 기본키가 아닌 속성이 다른 속성을 결정하지 않아야 함

ex) 학생 수강 테이블

학번	과목	교수
1	수학	김교수
2	영어	이교수
3	수학	김교수

→

학번	과목
1	수학
2	영어
3	수학

과목	교수
수학	김교수
영어	이교수
수학	김교수

(학번, 과목)이 기본 키이지만, 과목 → 교수의 종속성이 존재

제 4 정규화

- BCNF 조건을 만족하면서, **다치 종속성**을 제거해야 함
- 한 테이블에서 하나의 속성이 다른 속성과 다대다(N:M) 관계를 가질 때 발생하는 문제를 해결해야 함

※ 다치 종속성

A 값 하나에 대해 B의 값이 여러 개 존재할 수 있고, 이들이 서로 독립적일 때 발생

ex) 직원 기술 언어 테이블

직원	기술	언어
민수	전기	수학
민수	전기	영어

→

직원	언어
민수	수학
민수	영어

직원	기술
민수	전기

제 5 정규화

조인 종속성(테이블이 여러 개의 작은 프로젝션으로 무손실 분해 가능한 관계)을 제거

테이블을 더 작은 테이블로 **무손실 분해**할 수 있는 경우, 이를 분해합니다.

과목	선생	교실
수학	철수	A
수학	민수	B
영어	길동	A
과학	철수	C

→

과목	선생
수학	철수
수학	민수
영어	길동
과학	철수

과목	교실
수학	A
수학	B
영어	A
과학	C

선생	교실
철수	A
민수	B
길동	A
철수	C

- 특징
  - 이 세 테이블을 조인하면 원래의 테이블을 정확히 복원 가능
  - 각 테이블은 원래 테이블의 일부 정보만을 포함하지만, 전체적으로 모든 정보를 보존

■ 관계
- 서로 다른 엔터티들 간에 존재하는 연관성을 의미
- 관계를 맺는다는 의미는 부모 엔터티와 자식 엔터티가 연결된다는 의미이고, 부모의 식별자를 자식이 사용

ex) 하나의 학교에는 여러 명의 학생이 다닐 수 있고, 학생은 반드시 하나의 학교에 소속

학교

학교이름	주소
강남고	강남
양재고	양재

학생

학생번호	이름	학교이름
1	민수	강남고
2	철수	양재고

■ 관계의 분류
- 관계는 **존재**에 의한 관계와 **행위**에 의한 관계로 분류
- 4p에 설명

ex) 존재 관계는 두 엔터티 사이의 소유 ,분류 등 존재에 의해 발생
학교 – 학생: 학교가 없으면 학생도 존재할 수 없음
주문 – 주문 상세: 주문이 없으면 주문 상세 정보도 존재할 수 없음

행위 관계는 두 엔터티 사이의 행위나 사건에 의해 발생
고객 – 주문: 주문은 고객이 주문할 때 발생
학생 – 수강: 수강은 학생이 수업을 수강할 때 발생

■ 조인의 의미
- 데이터 중복을 최소화하기 위한 방향으로 테이블을 정규화를 하게 되는데 분리된 테이블들이 실제 사용하기 위해 데이터를 동시에 출력하는 과정에서 데이터를 연결 → 조인

학생

학번	이름	동아리 ID
100	민수	1
200	철수	2
300	영희	1

동아리

동아리 ID	동아리명	카테고리
1	영어 회화 스터디	스터디
2	영화	액티비티

↓

학번	이름	동아리 ID	동아리명	카테고리
100	민수	1	영어 회화 스터디	스터디
200	철수	2	영화	액티비티
300	영희	1	영어 회화 스터디	스터디

정규화된 학생, 동아리 테이블은 동아리 ID를 공유함
동아리 ID를 기준으로 조인하여 출력하게 되는데, 이때 두 데이터를 연결하는 키를 조인키라고 함

만약, 학번이 300인 학생의 동아리 명을 조회하는 SQL은
SELECT A.학번, B.동아리명
FROM 학생 A, 동아리 B
WHERE A.동아리 ID = B.동아리 ID AND A.학번 = 300

■ 계층형 데이터 모델
- **자기 자신끼리** 참조하는 관계가 발생
- 하나의 엔터티 내의 인스턴스끼리 계층 구조를 가지는 경우를 뜻함
- 계층 구조를 갖는 인스턴스끼리 연결하는 조인 → **셀프 조인**이라고 함

ex) 사원 테이블에서 사원 정보가 존재하고, 해당 사원의 매니저 직원 정보도 기록되어 있을 때, 사원의 매니저 이름을 알고 싶을 때 셀프 조인을 해야함

사원

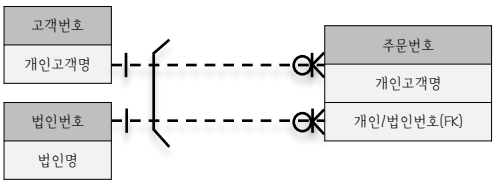
사번	이름	직책	부서	매니저 사번
100	광수	팀원	인사팀	300
200	길동	팀원	인사팀	300
300	민지	팀장	인사팀	400

SQL로는
SELECT A.이름 AS 사원이름, B.이름 AS 매니저이름
FROM 사원 A, 사원 B
WHERE A.매니저 사번 = B.사번 AND A.사번 = 100

사원이름	매니저이름
광수	민지

■ 상호 배타적 관계
- 두 테이블 중 하나만 가능한 관계를 뜻함 (둘 중 하나만 상속될 수 있음을 의미)

ex) 주문 엔터티에 주문한 사람은 개인 혹은 법인 둘 중 하나만 가능
개인이 주문했다면 법인 정보는 들어갈 수 없고,
법인이 주문했다면 개인이 들어갈 수 없음



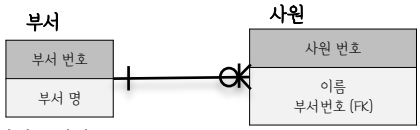
■ 트랜젝션
- **하나의 연속**적인 업무 단위
- 트랜젝션에 의한 관계는 **필수적인 관계** 형태를 가짐
- 하나의 트랜젝션에는 여러 DML(delete, update 등의 쿼리) 등이 포함될 수 있음

ex) 민수가 영희에게 100만원을 이체하는 경우
1. 민수 계좌에서 100만원이 존재하는지 확인
2. 존재한다면, 민수 계좌에 100만원을 출금(-)
3. 영희 계좌에 100만원을 입금(+)

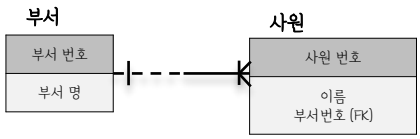
디비의 쿼리로는 3가지의 스텝이지만, 사용자 입장에서는 하나의 흐름으로 전부 성공하거나 전부 실패해야 한다.
이런 특성을 갖는 **연속적인 업무 단위를 트랜잭션**이라고 한다.

■ 필수적, 선택적 관계
- 두 엔터티의 관계가 서로 필수적일 때 하나의 트랜젝션을 형성
- 두 엔터티가 서로 독립적 수행이 가능하다면 선택적 관계로 표현
  - IE 표기법:
    - 필수: 원을 그리지 않음
    - 선택: 원을 그림

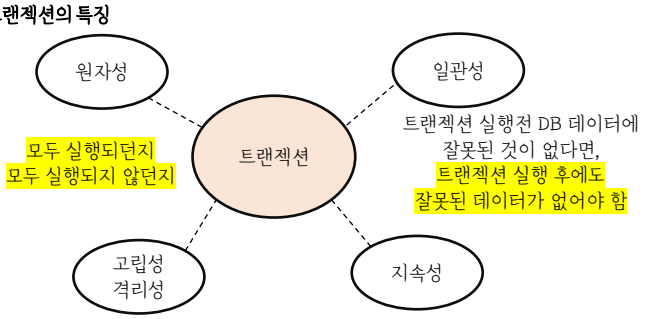
    ex) 신생 부서의 경우, 사원이 없을 수 있음



  - 바커 표기법
    - 필수: 실선으로 표기
    - 선택: 점선으로 표기



■ 트랜젝션의 특징



■ 고립성이 낮은 경우 문제점

Dirty Read	다른 트랜잭션에 의해 수정되었지만 아직 커밋되지 않는 데이터를 읽는 것
Non-Repeatable Read	한 트랜잭션 내에서 같은 쿼리를 두 번 수행했는데, 그 사이에 다른 트랜잭션이 값을 수정/삭제하여 두 쿼리의 결과가 다르게 제공되는 현상
Phantom Read	한 트랜잭션 내에서 같은 쿼리를 두 번 수행했는데, 그 사이에 다른 트랜잭션이 값을 추가하여 첫번째 쿼리 결과에 없던 레코드가 두번째 쿼리에서 제동되는 현상

■ Null 이란

- 아직 정해지지 않은 값을 의미
- 0과 빈 문자열("") 과는 다른 개념
- 각 컬럼 별 Null을 허용할지 말지 정해야 함

■ Null의 특성

- Null을 포함한 연산 결과는 항상 NULL
  - Null을 사전에 **치환**한 후 연산 해야함

이름	잔액	대출
김민수	10000	
최철수	500	-1000

← Null (빈 문자열일 수 있지만 여기서는 Null)

```
SELECT 이름, 잔액+대출
FROM 금융
```

이름	잔액+대출
김민수	
최철수	-500

← Null

```
SELECT 이름, 잔액+NVL(대출, 0)
FROM 금융
```

이름	잔액+대출
김민수	10000
최철수	-500

- 집계함수는 NULL을 제외한 연산 결과를 반환

※ sum, avg, min, max 등의 함수는 항상 NULL을 무시

이름	잔액	대출
김민수	10000	
최철수	500	-1000
이영희	5000	-500

```
SELECT COUNT(*), COUNT(잔액), COUNT(대출)
FROM 금융
```

COUNT(*)	COUNT(잔액)	COUNT(대출)
3	3	2

- **COUNT(*):** 모든 컬럼이 NULL이면 제외 (모든 컬럼이 NULL일 수 없음)
- **COUNT(잔액):** 잔액이 NULL이면 제외
- **COUNT(대출):** 대출이 NULL이면 제외

※ AVG(컬럼) 과 SUM(컬럼) / COUNT(*)는 다를 수 있음을 꼭 이해하고 넘어가자
위 예제라면 AVG(대출) = -750
SUM(대출) / COUNT(*)= -500
null을 제외하고 평균을 구하니 2명(최철수, 이영희)의 값만 평균값으로 계산
SUM(대출) / COUNT(*)은 2명의 대출이지만 COUNT(*)가 3 이므로 -500이 됨

■ Null의 ERD 표기

IE 표기법	NULL 허용 여부를 알 수 없음 **주문** 주문번호 주문금액 주문최소금액
바커 표기법	속성 앞 동그라미가 NULL 허용 속성을 의미 **주문** □ # 주문번호 □ ○ 주문금액 □ ○ 주문최소금액

■ 본질식별자 VS 인조식별자

본질식별자	업무에 의해 만들어지는 식별자
인조식별자	인위적으로 만들어지는 식별자 (**꼭 필요하지 않지만 관리 편이성 등의 이유로 인위적으로 생성**) 본질식별자가 복잡한 구성을 가질 때 인위적으로 생성 일반적으로, 자동으로 증가하는 일련번호(1,2,3…)의 형태로 기본키로 사용

ex) 주문, 주문 상세에 대한 엔터티 설계 시

주문

주문번호
고객번호

주문이력

주문번호(FK) 상품번호
주문수량 주문일자 배송지

1. PK: 주문번호 + 상품번호
   - 주문을 하게되면 **주문번호+상품번호**가 필요

본질식별자

- **하나의 주문번호로 같은 상품의 주문결과를 추가 저장 못함**

주문을 A상품 5개 하고, 추가적으로 3개를 주문할 경우
(물론, 8개로 row를 업데이트 할 수도 있지만 추가적으로 row를 쌓고 싶을 때)

2. PK: 주문번호 + 주문순번(새로운 컬럼)
   - 하나의 주문에 여러 상품에 대한 주문 결과 저장 가능 → 주문순번으로 구분가능
   - 주문순번 값을 위해 하나의 주문에 구매하는 상품의 Count를 매번 계산하여 입력해야 함.

주문이력

주문번호(FK) 주문순번
상품번호 주문일자 배송지

3. PK: 주문이력번호 (**인조식별자 생성**)
   - 주문이력번호로 각 주문이력을 구분하기 때문에 같은 주문의 상세 이력이 저장될 수 있음
   - 주문이력번호만 주식별자이므로 나머지 정보들이 중복 저장될 수 있음 (완전 같은 데이터지만 개발 실수 혹은 어떤 이유로든)
   - 실제 업무와 상관없는 주문상세번호를 주식별자로 생성하면 쓸모없는 index가 생성(PK는 자동 unique index 생성)

주문이력

주문이력번호
주문번호(FK) 상품번호 상품명 주문일자 배송지

주문이력번호	주문번호	상품번호	상품명	주문일자	배송지
1	000001	100	감귤 1KG	2024-01-01	서울
2	000001	100	감귤 1KG	2024-01-01	서울
3	000001	100	감귤 1KG	2024-01-01	부산

※ 인조 식별자는 편하긴 하지만 **다음의 단점**을 가짐

1. 중복 데이터 발생 가능 → 데이터 품질 저하
   - 물론, 주문번호 등에 unique 인덱스 설정이 가능하지만, 식별자 자체가 각 행을 구별할 수 있어야 하는데 그렇지 못하는 점
2. 불필요한 인덱스 생성 → 저장 공간 낭비 및 DML 성능 저하
   - 주문이력번호로 조회를 하지 않는 다는 점 (업무적 불필요)
   - 인덱스는 조회 성능을 향상 시킴
   - INSERT, UPDATE, DELETE 시, INDEX SPLIT 현상으로 성능 저하 됨

① SQL 기본

관계형 데이터베이스 개요

■ 데이터베이스와 DBMS(Data Management System)

- **데이터베이스: 데이터의 집합**. 일정한 형태로 데이터를 저장해 놓은 것으로 엑셀파일들을 모아둔다면 그것 또한 데이터베이스임
- **DBMS**: 데이터를 **효과적으로 관리하기 위한 시스템**. 데이터를 효율적으로 관리하고, 데이터의 손상을 피하며 필요시에 데이터를 복구하기 위함
  (위 요구사항을 시스템화 하기 위한 시스템을 DBMS라고 함(MYSQL, ORACLE))

■ 관계형 데이터베이스 구성 요소

계정	데이터 베이스 접근 제어를 하기 위한 업무별/시스템별 계정이 존재
테이블	DBMS의 DB안에 데이터가 저장되는 형식
스키마	테이블이 어떠한 구성으로 되어있는지, 어떤 정보를 가지고 있는지에 대한 구조를 정의

■ 테이블

데이터가 저장되는 단위

- 엑셀의 시트처럼 행(row)과 열(column)을 갖는 2차원 구조
- 데이터 저장하는 최소 단위
- 모델링 별 부르는 용어 다름 (성질은 같다고 생각하면 됨)

속성(개념모델링) → 컬럼(물리모델링)
엔터티(개념모델링) → 테이블(물리 모델링)
인스턴스(개념모델링) → 튜플 (하나의 행)

- 하나의 테이블은 **반드시 하나의 계정 소유**여야 함(여러 유저가 볼 수는 있음)
- 테이블간 관계는 1:1, 1:N, M:N의 관계를 가질 수 있음 (1 과목에서 설명했듯이)
- 테이블 명은 **중복 불가**. **소유자가 다른 경우 같은 이름으로 생성 가능**

- 행 단위로 데이터가 입력, 삭제 (삽입 시, 누락된 컬럼은 null 설정되어 삽입)
- 수정은 값(컬럼) 단위 가능 → ex) 사원의 직급만 수정 가능
- 특정 값만 삭제하고 싶어요 → **삭제가 아닌** 그 값을 null로 업데이트 하는 것

ex) 사원 테이블에 새로운 사원 입력 시, 사원번호 · 이름 등 테이블의 모든 컬럼을 **동시에 입력**하고, 삭제 시 해당 사원은 **모든 정보(row)가 삭제**됨.

■ 관계형 데이터베이스 (Relational DBMS, RDBMS)

테이블로 데이터를 관리하고 테이블간 관계를 이용해 데이터를 정의하는 방식
ex) Oracle도 RDBMS

■ 관계형 데이터베이스(RDBMS) 특징

1. 데이터 분류, 정렬, 탐색 속도가 빠름
2. 기존의 작성된 스키마 수정이 어려움
   ex) 기존 테이블에 컬럼을 추가할 때, 기존 존재하던 튜플들에도 영향을 줌
3. 데이터베이스의 부하를 분석하기 어려움
4. 신뢰성이 높고, 데이터의 무결성 보장

※ **신뢰성**: 안정적으로 작동하고 데이터를 정확하게 저장, 관리할 수 있는 능력
(안정적으로 동작한다, 예측 가능하게 동작한다 정도로 이해)

※ **데이터 무결성(integrity)**: 저장된 데이터의 정확성, 일관성, 데이터에 결손과 부정합이 없음을 보증하는 것 (데이터에 결점이 없다)
데이터베이스에 저장된 값과 그것이 표현하는 현실의 비즈니스 모델의 값이 일치하는 정확성을 의미함
데이터 무결성을 유지하는 것이 DBMS의 중요한 기능
(쉽게 표현하면, 테이블 · DB간 데이터가 일치한다, 틀어지지 않는다, 누락 없다)

■ SQL(Structured Query Language)

- **RDBMS**에서 존재하는 모든 언어 (데이터 조회 · 조작 · 시스템 관리를 명령하는 언어)
- SQL 용도 따라 구분

DDL(Data Definition Language) 데이터 정의어	CREATE, ALTER, DROP, RENAME, TRUNCATE *= 테이블 만들다 (구조 자체를 정의)*
DML(Data Manipulation Language) 데이터 조작어	INSERT, UPDATE, DELETE, MERGE *= 만든 테이블을 조회, 삽입, 수정, 삭제*
DCL(Data Control Language) 데이터 제어어	GRANT, REVOKE *= "조회, 삭제" 권한을 제어*
TCL(Transaction Control Language) 트랜젝션 제어어	COMMIT, ROLLBACK, SAVEPOINT *= 트랜젝션 제어*

(테이블의 데이터를 삭제하지만, 테이블의 구조(스키마)를 재설정하는 테이블 정의를 변경하는 작업이기에 DDL로 분류 (+ 자동으로 COMMIT. 롤백이 불가능한 DDL의 특성))

(확정 ex) 송금, 커밋 되지 않은 쿼리 취소)

■ 데이터 무결성 종류

개체 무결성	테이블의 기본 키(PK)가 각각의 행(row)에서 유일하게 식별되어야 함을 보장하는 것 (즉, 기본 키가 중복 X, NULL 가질 수 없음)
참조 무결성	다른 테이블과의 관계를 보장하는 것 외래 키는 NULL이거나 다른 테이블의 기본 키와 일치해야 함
도메인 무결성	각 속성이 해당 도메인 (속성(컬럼)이 갖는 범위) 내에 적합함 값을 가져야 함을 보장 (즉, 특정 열은 허용된 값의 범위 내에 존재 해야함)
NULL 무결성	특정 속성에 대해 NULL을 허용하지 않는 특징
고유 무결성	특정 속성에 대해 값이 중복되지 않는 특징
키 무결성	하나의 릴레이션 (테이블 간의 관계)에서 적어도 하나의 키가 존재해야 함

SELCT 문

SELECT 구문의 경우, SQL 종류 어디에도 속하지 않고, DQL (Data Query Language) 임
원하는 데이터를 조회할 때 사용하는 문법

■ SELECT 문 구조

```
1 SELECT * | 컬럼명 | 컬럼에서 파생된 표현
2 FROM 테이블명 또는 뷰명
3 WHERE 조회 조건
4 GROUP BY 그룹핑 할 컬럼명
5 HAVING 그룹핑 필터링 조건
6 ORDER BY 정렬할 컬럼명
```

(4행, 5행 순서를 바꿀 수 있지만 보통 사용되지 않음)

6가지 절로 구성
우리가 작성하는 순서와 DBMS가 해석하는 순서와는 다름

DBMS는 FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY 의 순서대로 파싱 후 해석 (FWGHS)

(각 절의 순서대로 작성해야 함)

SELECT	조회할 컬럼 (전체 컬럼, 일부 컬럼, 컬럼에 대한 연산) * ex) name, age ex) ROUND(price, 2)
FROM	조회할 데이터
WHERE	특정 행을 필터링 하기 위함
GROUP BY	특정 컬럼을 데이터 기준으로 그룹핑 ex) 성별의 COUNT, 성적 분포 등
HAVING	그룹핑한 데이터의 필터링 정보 (GROUP BY한 결과에 대한 조건) ex) 성적 그룹핑 후 B 이상 성적만 조회
ORDER BY	결과에 대한 정렬할 조건

■ SELECT 절

- **전체 컬럼 조회:** 테이블의 모든 컬럼을 조회

```
SELECT *
    FROM students;
```

id	name	gender
1	김철수	M
2	이영희	F

- **특정 컬럼 조회: 원하는 컬럼만 조회**

```
SELECT name
    FROM students;
```

name
김철수
이영희

- **별칭(Alias) 지정 가능:** 원래 정한 컬럼명 대신 출력할 임시 이름 지정 가능

(원래의 컬럼명이 변경되는 건 아님)

```
SELECT name as student_name
    FROM students;
```

student_name
김철수
이영희

- **표현식:** 조회한 컬럼에 대한 연산이 가능

```
SELECT CONCAT(name, "님") as student_name
    FROM students;
```

student_name
김철수님
이영희님

■ Alias 별칭 주의사항

- 순서상 SELECT에서 정의된 별칭은 뒤에 있는 ORDER BY만 사용 가능
  (그 외에서는 이미 DBMS 해석 순서상 먼저 수행되기 때문에 별칭을 모름)
- 이미 존재하는 예약어를 별칭으로 사용할 수 없음 (avg, count …)
- 문자로 시작해야 함 (사용하기로 약속한 약어)
- 가능한 특수문자: $ _ # (불가능한 특수문자도 쌍 따옴표 사용 시 가능)
- 띄어쓰기 불가 (쌍 따옴표 사용 시 가능)

```
SELECT S.name
    FROM students as S;
```

name
김철수
이영희

```
SELECT students.name
    FROM students as S;
```

Unknown column 'students.name'

■ FROM 절

- 테이블 별칭 선언 가능(ORACLE as 사용 불가, SQL Server는 사용/생략 가능)

※ 별칭 선언 시, 컬럼 구분자로 테이블 별칭으로만 가능 (테이블명으로 사용 시 에러)

- ORACLE에서는 FROM 절 생략 불가 – SQL Server에서는 불 필요시 생략 가능

ex) SELECT now(); 같이 FROM 절이 필요 없을 경우

함수

■ 함수 정의

- 다른 언어의 함수와 같이 input 이 존재할 경우, 그에 맞는 output을 출력 (Input이 여러 개 존재하더라도 output은 하나만 존재 (N:1))
- From 절을 제외한 모든 절에서 사용 가능

■ 함수의 종류

입력 값(input)과 리턴 값(output)에 따라 분류됨

- Input 1 → Output 1 : 단일행 함수
- Input N → Output 1: 복수(다중)행 함수

■ 문자함수 종류 (문자를 입력)

함수	기능	예시	출력	예시 설명
LOWER(대상)	문자열을 소문자로 변환	LOWER('aBcD')	abcd	
UPPER(대상)	문자열을 대문자로 변환	UPPER('aBcD')	ABCD	
SUBSTR(대상, 시작, n)	대상의 시작 위치에서 n개 추출	SUBSTR('1234', 2, 3)	234	시작 위치의 문자 포함
		SUBSTR('1234', -2, 2)	34	뒤에서 2번째에서 오른쪽방향으로 2개 추출
INSTR(대상, 찾을 문자열, 시작, n)	시작 위치에서부터 대상에서 n번째 찾을 문자열 위치 반환	INSTR('ABABAB', 'B')	2	시작, n 생략 시 각각 1로 동작
		INSTR('ABABAB', 'B', 3)	4	3 부터 시작해서 B 문자열 탐색
		INSTR('ABABAB', 'B', 3, 2)	6	3 위치에서부터 2번째로 찾은 B 문자
		INSTR('ABABAB', 'B', -1, 2)	4	뒤에서 1번째에서 왼쪽방향으로 2번째로 찾은 문자
LTRIM(대상, 삭제문자열)	대상에서 삭제문자열을 왼쪽에서부터 삭제	LTRIM('AAAB', 'A')	B	LTRIM('AAAB', 'B')였다면 AAAB 출력
RTIRM(대상, 삭제문자열)	대상에서 삭제문자열을 오른쪽에서부터 삭제	RTRIM('ABBB', 'B')	A	RTRIM('ABBB', 'A') 였다면 ABBB 출력
TRIM(대상, 삭제문자열)	대상의 양쪽에서 삭제문자열 삭제	TRIM(' A A ')	A A	생략 시 양쪽 공백 제거 (중간 공백은 제거 X)
		TRIM('ABA', 'A')	B	oracle TRIM은 삭제문자열 지정 불가 (공백만 삭제)
LPAD(대상, n, 추가문자열)	대상 왼쪽에 추가문자열을 추가하여 총 n개의 길이 반환	LPAD('CD', 4, 'A')	AACD	SELECT TRIM('ABA', 'A') missing right parenthesis
RPAD(대상, n, 추가문자열)	대상 오른쪽에 추가문자열을 추가하여 총 n개의 길이 반환	RPAD('CD', 4, 'A')	CDAA	
CONCAT(대상1, 대상2)	대상1과 대상2를 결합하여 반환	CONCAT('AB', 'CD')	ABCD	인자는 2개만 받음 (대상1, 대상2)
LENGTH(대상)	문자열의 길이 반환	LENGTH('ABC')	3	LENGTHB(대상)의 경우, 몇 바이트인지 반환
REPLACE (대상, 찾을문자열, 변경문자열)	문자열 치환 or 삭제	REPLACE('ABAB', 'A')	BB	변경문자열 생략 시, 빈문자열('')
		REPLACE('ABBA', 'AB','ab')	abBA	전체 문자열에서 찾을문자열을 변경문자열로 치환
TRANSLATE (대상, 찾을문자열, 변경문자열)	문자 1 대 1로 치환	TRANSLATE('CCB','BC','bc')	ccb	B는 b로 치환, C는 c로 각각 치환되어 ccb로 치환

TRIM 계열의 문자는 제거하는 기능을 함

둘 다 음수지만 스캔 방향이 다름

왼쪽에 삭제문자열 존재 시 제거

오른쪽에 삭제문자열 존재 시 제거

■ 숫자함수 종류 (숫자를 입력)

함수	기능	예시	출력	예시 설명
ABS(숫자)	숫자의 절대값 반환	ABS(-3)	3	양수도 양수로 출력 예를 들어, ABS(3) 도 3을 출력
ROUND(숫자, 자리수)	숫자의 특정 자리 반올림	ROUND(123.456, 2)	123.46	★소수점 2번째자리까지 노출하기 위해 소수점 3번째 반올림
		ROUND(123.456, -2)	100	정수 자리 2번째에서 반올림
TRUNC(숫자, 자리수)	숫자의 특정 자리 버림	TRUNC(123.456, 2)	123.45	소수점 2번째자리까지 노출하기 위해 소수점 3번째 버림
		TRUNC(123.456, -2)	100	정수 자리 2번째까지 버림
SIGN(숫자)	숫자 양수면 1, 음수면 -1 반환	SIGN(123.456)	1	
		SIGN(-123.456)	-1	
FLOOR(숫자)	숫자에 같거나 가까운 작은 정수 반환	FLOOR(4.9)	4	FLOOR(4)는 4를 반환
CEIL(숫자)	숫자에 같거나 가까운 큰 정수 반환	CEIL(5.1)	6	CEIL(5)는 5를 반환
MOD(숫자1, 숫자2)	숫자1을 숫자2로 나눈 나머지 반환	MOD(3,2)	1	
		MOD(6,3)	0	
		MOD(6,7)	6	
POWER(숫자, n)	숫자^n (숫자를 n번 제곱)	POWER(2,3)	8	2 * 2 * 2 = 8
SQRT(숫자)	루트 값 반환	SQRT(9)	3	

★ 둘의 차이를 이용한 문제가 가끔 출제

■ 날짜함수 종류

ORACLE과 SQL Server의 날짜함수가 많이 달라 시험에는 출제빈도가 낮을 것으로 **예상**됨

ORACLE	SQL Server	기능	출력 및 형식
SYSDATE	GETDATE()	현재 날짜와 시간 반환 (날짜만 출력될 수 있음)	03-SEP-24 2024-09-03 06:21:12
CURRENT_DATE	-	현재 날짜 반환	02-SEP-24
CURRENT_TIMESTAMP	CURRENT_TIMESTAMP	현재 타임스탬프 반환	02-SEP-24 11.03.21.773327 PM US/PACIFIC 2024-09-03 06:21:42
ADD_MONTHS(날짜, n)	DATEADD(MONTH, 3, 날짜) (월 이외의 단위 날짜로 연산 가능)	날짜에서 n개월 후 반환	03-DEC-24 2024-12-03 06:13:26
MONTHS_BETWEEN(날짜1, 날짜2)	DATEDIFF(MONTH, 날짜1, 날짜2)	날짜1과 날짜2의 개월 수 반환	11.967741… 11
LAST_DAY(날짜)	EOMONTH(날짜)	주어진 월을 마지막 날짜 반환	30-SEP-24 2024-09-30
NEXT_DAY(날짜, 'FRIDAY')	-	주어진 날짜 이후 지정된 요일의 첫 번째 날짜 반환	06-SEP-24
ROUND(SYSDATE, 'MONTH')	-	날짜 반올림	01-SEP-24
TRUNC(SYSDATE, 'YEAR')	-	날짜 버림	01-JAN-24

바로 매칭되는 것은 없고, DATEADD를 통해 동일한 효과를 구현하는 것은 가능

■ 변환함수 종류

- 값의 데이터 타입을 "변환" (문자 → 숫자, 숫자 → 문자 등)

기능	ORACLE	예제	SQL Server	예제
문자를 숫자로 변경	TO_NUMBER(문자)	TO_NUMBER('1234')	CAST(문자 AS INT)	CAST('1234' AS INT) → 1234
리턴 타입이 문자 타입 숫자 포맷을 변경	★ TO_CHAR(대상, 포맷)	천 단위 구분자 생성 TO_CHAR(1234, '9,999') → 1,234 출력 총 6자리로 리턴 (앞 자리수 0으로) TO_CHAR(1234, '000,999') → 001,234 출력	CONVERT(데이터타입, 날짜, 스타일)	스타일 코드 1 천 단위 구분자 X CONVERT(VARCHAR, 1234.56, 1) 스타일 코드 2 천 단위 구분자 O CONVERT(VARCHAR, 1234.56, 2)
리턴 타입이 문자 타입 날짜 포맷을 변경	TO_CHAR(대상, 포맷)	TO_CHAR(SYSDATE, 'YYYY-MM-DD')	FORMAT(날짜, 포맷)	FORMAT(GETDATE(), 'yyyy-MM-dd')
주어진 문자를 포맷 형식에 맞게 읽어서 날짜로 리턴	★ TO_DATE(대상, 포맷)	2023-05-15는 문자이고, 포맷대로 읽어서 날짜 타입으로 리턴 TO_DATE('2023-05-15', 'YYYY-MM-DD')	CONVERT(데이터타입, 날짜, 스타일)	스타일 코드 120은 'YYYY-MM-DD HH:MI:SS' 형식 CONVERT(DATE, '2023-05-15', 120)

- SQL Server의 경우, 포맷을 지정하여 CONVERT 를 사용하여 변환할 수 있고, 단순한 변환의 경우 CAST를 주로 사용

■ 그룹함수 종류

※ 모든 그룹함수는 NULL을 무시하고 연산 ex. 컬럼 값이 NULL, 1, 2 이렇게 존재하면 AVG의 값은 1 + 2 / 2 해서 1.5이 됨
AVG(NVL(컬럼, 0) 이렇게 표현한다면 0 + 1 + 2 / 3 해서 1이 출력됨

함수	기능	예시	특징
COUNT(대상)	행의 수 리턴	SELECT COUNT(TEST) FROM DEMO;	대상 데이터가 null 만 존재할 때, 0을 리턴
SUM(대상)	총 합 리턴	SELECT SUM(TEST) FROM DEMO;	대상 데이터가 null 만 존재할 때, 공집합을 리턴
AVG(대상)	평균 리턴	SELECT AVG(TEST) FROM DEMO;	대상 데이터가 null 만 존재할 때, 공집합을 리턴
MIN(대상)	최솟값 리턴	SELECT MIN(TEST) FROM DEMO;	대상 데이터가 null 만 존재할 때, 공집합을 리턴
MAX(대상)	최댓값 리턴	SELECT MAX(TEST) FROM DEMO;	대상 데이터가 null 만 존재할 때, 공집합을 리턴
VARIANCE(대상) – ORACLE VAR(대상) – SQL Server	분산 리턴 값들이 평균에서 얼마나 떨어져 있는지	SELECT VARIANCE(TEST) FROM DEMO; SELECT VAR(TEST) FROM DEMO;	대상 데이터가 null 만 존재할 때, 공집합을 리턴
STDDEV(대상) – ORACLE STDEV(대상) – SQL Server	표준편차 리턴 분산의 제곱근	SELECT STDDEV(TEST) FROM DEMO; SELECT STDEV(TEST) FROM DEMO;	대상 데이터가 null 만 존재할 때, 공집합을 리턴

```
SELECT
    COUNT(a) as cnt,
    SUM(a) as sum,
    AVG(a) as avg,
    MIN(a) as min,
    MAX(a) as max,
    VARIANCE(a) as var,
    STDDEV(a) as stdev
FROM demo;
```

CNT	SUM	AVG	MIN	MAX	VAR	STDEV
0	-	-	-	-	-	-

■ 일반함수 종류 ★★

함수	포맷	기능	예제
DECODE	DECODE(대상, 값1, 반환1, 값2, 반환2…, 그 외 반환)	대상 = 값1 이라면 반환1을 반환 아니라면 값2와 비교 대상 = 값2 라면 반환2를 반환 … 전부 다르다면, 그 외 반환을 반환	DECODE(score, 10, 'A', score가 10이면 A 9, 'B', score가 9이면 B 8, 'C', score가 8이면 C 7, 'D', score가 7이면 D 'F' 전부 불일치면 F) DECODE(gender, 반환 해야하는 값에 DECODE를 중첩해서 사용하는 것이 가능 'MALE', DECODE(score, A, 'A팀', 'B팀'), MALE이면, score를 확인 A이면 A팀, 그 외는 B팀 'FEMALE', DECODE(score, A, 'X팀', 'Y팀')) FEMALE이면, score를 확인 A이면 X팀, 그 외는 Y팀) MALE, FEMALE 둘 다 아닐 때 반환할 값을 지정하지 않을 수 있음. 이럴 경우, NULL 반환

■ 일반함수 종류

함수	포맷	기능	예제
NVL	NVL(대상, 치환값)	대상이 NULL 이 아닐 경우, 대상 그대로 반환 대상이 NULL 일 경우, 치환값 반환	NVL(title, '무제')
NVL2	NVL2(대상, 치환값1, 치환값2)	대상이 NULL 이 아닐 경우, 치환값1 반환 대상이 NULL 일 경우, 치환값2 반환	NVL2(bonus, salary + bonus, salary) AS total_compensation 보너스가 NULL이 아닐 경우, 급여와 보너스 합산 후 반환 보너스가 NULL일 경우, 급여만 반환
COALESCE	COALESCE(대상1, 대상2, …, 그 외 일 경우 반환)	대상1, 대상2… 순서대로 NULL인지 체크 대상1이 NULL이 아니면 바로 반환 대상1이 NULL이라면 대상2가 NULL인지 체크 대상2가 NULL이 아니면 바로 반환 순서대로 체크 후 모두 NULL인 경우 그 외 일 경우 반환를 반환	1. email check 2. phone check COALESCE(email, phone, '연락처 없음') AS primary_contact email 데이터가 NULL이라면, phone 데이터 확인 (email 데이터가 존재했다면, email 반환) phone 데이터도 NULL이라면 '연락처 없음' 노출 (phone 데이터가 존재했다면 phone 반환
NULLIF	NULLIF(대상1, 대상2)	대상1과 대상2가 같으면, NULL 반환 대상1과 대상2가 다르면, 대상1 반환	NULLIF(10, 20) 10 반환
CASE	CASE WHEN 조건1 THEN 반환1 WHEN 조건2 THEN 반환2 ⋮ ELSE 반환3 END	조건1에 해당된다면 반환1 반환 아니라면 조건2 체크 후 조건2에 해당된다면 반환2 반환 전부 해당되지 않는다면 반환3 반환 ※ DECODE와 다르게 여러 조건(대소 비교 등)에 대한 연산 가능	CASE WHEN score >= 90 THEN '우수' WHEN score >= 80 THEN '양호' WHEN score >= 70 THEN '보통' ELSE '노력 필요' END AS performance
	CASE 컬럼 WHEN 값1 THEN 반환1 WHEN 값2 THEN 반환2 ⋮ ELSE 반환3 END	조건식의 내용이 equals일 경우에만 해당 포맷으로 표현 가능 CASE WHEN code = 'HR' THEN '인사부' WHEN code = 'FIN' THEN '재무부' WHEN code = 'OPS' THEN '운영부' ELSE '알 수 없는 부서' END AS department_name ※ 동등 비교 시 비교대상(code)를 CASE와 WHEN 사이에 배치하면서 WHEN절 마다 반복하지 않아도 됨 (이 때 code. 데이터 타입과 WHEN절의 명시된 비교대상의 데이터타입 반드시 일치해야 함)	두 예제의 결과가 동일 CASE code WHEN 'HR' THEN '인사부' WHEN 'FIN' THEN '재무부' WHEN 'OPS' THEN '운영부' ELSE '알 수 없는 부서' END AS department_name CASE WHEN code='HR' THEN '인사부' WHEN code='FIN' THEN '재무부' WHEN code='OPS' THEN '운영부' ELSE '알 수 없는 부서' END AS department_name
ISNULL	ISNULL(대상, 치환값)	대상이 NULL이면 치환값 리턴	※ SQL Server 함수 ISNULL(Department, 'Unassigned') AS Department 부서 정보가 없으면 'Unassigned'으로 노출

WHERE 절

■ WHERE 절

- 데이터 중에서 원하는 조건에 맞는 데이터(행)만 조회하고 싶은 경우에 여러 조건을 전달해 조회
- 조건식에서 비교 대상의 타입과 데이터 타입이 일치하는 것이 성능상 좋음

■ WHERE 문법

```
SELECT * 또는 컬럼명 또는 표현식          SELECT *
    FROM 테이블명 또는 뷰명          -->      FROM student_scores
    WHERE 조회할 데이터 조건                  WHERE SCORE > 85;
```

연산자	설명	예제	예제 설명
=	동일한 데이터를 조회	name = '김민수'	name 컬럼 데이터 중 김민수 인 데이터만 조회
!=, <>	일치하지 않는 데이터만 조회	name != '김민수'	김민수가 아닌 데이터만 조회
>	큰 조건을 조회	age > 19	나이가 19살 초과인 데이터만 조회
>=	크거나 같은 조건을 조회	age >= 20	나이가 20살 이상인 데이터만 조회
<	작은 조건을 조회	score < 90	점수가 90점 미만인 데이터만 조회
<=	작거나 같은 조건을 조회	score <= 85	점수가 85점 이하인 데이터만 조회
조건1 OR 조건2	조건1 또는 조건2 조건 하나라도 만족하는 데이터를 조회	score = 'A' OR score = 'B'	학점이 A 이거나 B 인 학생들 조회
조건1 AND 조건2	조건1과 조건2 둘 다 모두 만족하는 데이터를 조회	name = '최철수' AND age = 20	이름이 최철수 이면서 20살인 데이터만 조회
BETWEEN a AND b	a 이상 ~ b 이하에 있는 범위 데이터를 전부 조회 컬럼 >= a AND 컬럼 <= b 와 동일 데이터 타입은 숫자 이외에 날짜, 문자(a, b, c…, z)들도 사용 가능 BETWEEN b AND a 로 입력하면 아무것도 조회되지 않음 (a에는 크거나 같은 데이터를 b에는 작거나 같은 데이터를 넣어야 함)	score BETWEEN 80 AND 90	점수가 80점 이상 90점 이하인 데이터 조회 score BETWEEN 90 AND 80으로 작성할 경우, 90점 이상이면서 80점 이하인 데이터를 조회 (모순적인 조건이므로 아무것도 조회되지 않음)
IN(a, b, c…)	a 이거나 b 이거나 c … 하나라도 해당하면 조회 OR 조건의 나열과 동일 (컬럼 = a OR 컬럼 = b OR 컬럼 = c …)	score IN ('A', 'B')	학점이 A 이거나 B 인 학생들 조회
NOT 조건1	조건1에 해당 되지 않은 데이터만 조회	NOT(score < 90)	90점 이상인 학생만 조회 (=90점 미만이 학생들을 제외한 데이터만 조회) (=score >= 90)

GROUP BY, HAVING 절

■ GROUP BY

- 각 행을 그룹으로 묶어서 조회하는 구문식
- GROUP BY 절에는 그룹으로 컬럼 여러 개 지정 가능
- GROUP BY는 WHERE 절 수행이후에 수행되므로, WHERE 절에는 GROUP BY에서 나온 결과에 대한 조건식을 넣을 수 없음 (수행 전이니 당연한 말이지만)
- 제거하고 싶은 그룹이 있다면 미리 WHERE 절에서 제거

ex) 사원 테이블에서 부서명으로 그룹핑 할 경우, 인사팀 정보는 제거하고 싶을 때 WHERE 절에서 department_name != '인사팀' 을 통해 인사팀 정보는 미리 제거를 하고 그룹핑 하는 게 성능상 좋음
물론, 그룹핑 후에 HAVING 절을 통해 제거하는 것도 가능하지만, 불필요한 데이터는 사전에 제거하면 묶는 게 더 좋음

```
SELECT department_name
FROM employees
WHERE department_name != '인사부'
GROUP BY department_name;
```

DEPARTMENT_NAME
영업부
IT부

- SELECT 절에 집계 함수를 사용하여 그룹 연산 결과를 표현할 수 있음

```
SELECT department_name, avg(salary)
FROM employees
GROUP BY department_name;
```

DEPARTMENT_NAME	AVG(SALARY)
영업부	5250000
IT부	6750000
인사부	4750000

- 그룹핑이 되면 그룹핑 되기 전 데이터는 출력할 수 없음 (GROUP BY 컬럼과 그룹함수를 사용한 결과 값만 표현 가능)

ex) 사원 테이블에 이름 정보가 있더라도, 부서명으로 그룹핑 을 했다면 정보가 요약됐기 때문에 이름 컬럼을 조회할 수 없음

```
SELECT department_name, salary
FROM employees
GROUP BY department_name;
```

→ GROUP BY 표현식이 아닙니다.

■ HAVING 절

- 그룹 함수 결과를 조건으로 사용할 때 사용하는 절 (WHERE에서는 순서상 GROUP BY의 결과값에 대한 조건식을 사용할 수 없기에 HAVING을 통해 조건식 사용)

```
SELECT department_name, avg(salary)
FROM employees
GROUP BY department_name
HAVING avg(salary) > 5000000;
```

부서 이름	평균(급여)
영업부	5250000
IT부	6750000

```
SELECT department_name, avg(salary)
FROM employees
WHERE avg(salary) > 5000000
GROUP BY department_name;
```

→ 그룹 기능은 여기에서 허용되지 않습니다.

- SELECT 절보다 먼저 호출되기 때문에 SELECT 절의 Alias(as)를 사용 불가
- WHERE 절과 HAVING 절 동시에 사용 가능

```
SELECT department_name, avg(salary)
FROM employees
WHERE department_name != 'IT부'
GROUP BY department_name
HAVING avg(salary) > 5000000;
```

부서 이름	평균(급여)
영업부	5250000

- WHERE 절과 HAING 절 둘 다 동일한 조건식이 사용가능한 경우엔 WHERE에서 표현하는 게 일반적 (그룹핑 수행을 다하고 조건식을 거르는 것보다는 불필요한 그룹핑 먼저 제외하는 게 성능상 유리)

```
SELECT department_name, avg(salary)
FROM employees
WHERE department_name != 'IT부'
GROUP BY department_name
HAVING department_name != 'IT부';
```

부서 이름	평균(급여)
영업부	5250000
인사부	4750000

ORDER BY 절

- 기본적으로 데이터는 입력한 순서대로 출력되지만, 출력되는 순서를 변경하고자 할 때 ORDER BY 절 사용
  - ASC: 오름차순 (생략 가능)
  - DESC: 내림차순

```
SELECT name, department_name, salary
FROM employees
WHERE department_name = '인사부'
ORDER BY salary;
```

이름	부서 이름	샐러리
송태호	인사부	4700000
임재현	인사부	4800000
한미영	인사부	5000000

```
SELECT name, department_name, salary
FROM employees
WHERE department_name = '인사부'
ORDER BY salary ASC;
```

이름	부서 이름	샐러리
송태호	인사부	4700000
임재현	인사부	4800000
한미영	인사부	5000000

```
SELECT name, department_name, salary
FROM employees
WHERE department_name = '인사부'
ORDER BY salary DESC;
```

이름	부서 이름	샐러리
한미영	인사부	5000000
임재현	인사부	4800000
송태호	인사부	4700000

- ORDER BY [컬럼1], [컬럼2] … 의 형태로 표현
  - 컬럼1 기준으로 정렬 후 동일한 데이터는 컬럼2까지 비교

```
SELECT name, bonus_percentage, salary
FROM employees
WHERE bonus_percentage <= 10
ORDER BY
    bonus_percentage DESC,
    salary ASC,
    name ASC;
```

먼저, 보너스 퍼센트를 내림차순 정렬
보너스 퍼센트의 데이터가 같다면 급여 오름차순 정렬
급여까지 데이터가 같은 경우, 이름 오름차순 정렬

이름	보너스_퍼센트	샐러리
김철수	10	5000000
한미영	10	5000000
송태호	8	4700000
박민수	8	5200000
임재현	5	4800000

- ORDER BY 뒤에 SELECT 절에 선언된 숫자로도 선언 가능
  - 컬럼명과 혼합해 사용 가능

```
SELECT name, bonus_percentage, salary
FROM employees
WHERE bonus_percentage <= 10
ORDER BY
    2 DESC,
    salary ASC,
    name ASC;
```

이름	보너스_퍼센트	샐러리
김철수	10	5000000
한미영	10	5000000
송태호	8	4700000
박민수	8	5200000
임재현	5	4800000

- 유일하게 SELECT 절에서 정의한 Alias(별칭) 사용 가능

```
SELECT name, bonus_percentage as bouns, salary
FROM employees
WHERE bonus_percentage <= 10
ORDER BY
    bouns DESC,
    salary ASC,
    name ASC;
```

NAME	BOUNS	SALARY
김철수	10	5000000
한미영	10	5000000
송태호	8	4700000
박민수	8	5200000
임재현	5	4800000

■ 데이터 타입 별 정렬 순서 (오름차순)

- 한글: 가 → 나 → 다 → 라…
- 영어: A → B → C → D…
- 숫자: 1 → 2 → 3 → 4…
- 날짜: 과거 → 최근

■문자 정렬

- 문자의 오름차순의 경우, 왼쪽부터 값이 작은 순으로 정렬
  - 만약, 값이 동일하다면 왼쪽에서 두 번째 값이 작은 순으로 정렬
  - ⋮

```
SELECT name
FROM employees
ORDER BY name;
```

→

NAME
김영희
김철수
박민수
정수연

- 숫자를 문자로 변경한 뒤에 정렬할 경우, 문자 방식으로 정렬 (왼쪽 글자부터 비교)

```
SELECT TO_CHAR(salary) as salary
FROM employees
ORDER BY salary;
```

→

SALARY
1000
1100
2000
500

■날짜 정렬

날짜 순으로 정렬

```
SELECT
    TO_CHAR(hire_date, 'YYYY-MM-DD') as format_date,
    hire_date
FROM employees
ORDER BY hire_date;
```

→

FORMAT_DATE	HIRE_DATE
2018-11-01	01-NOV-18
2019-05-20	20-MAY-19
2020-01-15	15-JAN-20
2021-03-10	10-MAR-21

■NULL 정렬

- NULL이 컬럼에 포함되어 있을 경우,
  - **ORACLE: 마지막으로 정렬**

```
SELECT name, bonus
FROM employees
ORDER BY bonus;
```

→

NAME	BONUS
박민수	300000
정수연	800000
이영희	-
강지원	-

  - SQL Server: 처음으로 정렬

```
SELECT name, bonus
FROM employees
ORDER BY bonus;
```

→

name	bonus
이영희	NULL
강지원	NULL
박민수	300000
정수연	800000

- ORACLE의 경우, NULL 정렬 순서 변경 가능
  - NULLS LAST: NULL 마지막으로 정렬 (기본)
  - NULLS FIRST: NULL 처음으로 정렬

```
SELECT name, bonus
FROM employees
ORDER BY bonus NULLS FIRST;
```

→

NAME	BONUS
강지원	-
이영희	-
박민수	300000
정수연	800000

- SQL Server의 경우, NULL 정렬 순서 변경 불가
- 정렬 순서를 DESC로 변경하면 NULL 정렬도 반대로 바뀜 (ORACLE, SQL Server)

```
SELECT name, bonus
FROM employees
ORDER BY bonus DESC;
```

→

NAME	BONUS
강지원	-
이영희	-
정수연	800000
박민수	300000

  - bonus는 내림차순, NULL은 마지막 정렬하고 싶다면, NULLS LAST 추가

```
SELECT name, bonus
FROM employees
ORDER BY bonus DESC NULLS LAST;
```

→

NAME	BONUS
박민수	300000
정수연	800000
이영희	-
강지원	-

조인

■조인(JOIN)

- 여러 테이블들을 겹치는 키(조인키)를 통해 동시에 출력하기 위해 사용
- NULL은 겹치는 데이터가 아님 (NULL인 컬럼끼리 연결 X)

ex) 사원 정보와 부서명 정보를 동시에 보고 싶을 때, 부서번호를 조인키로 동시 출력

사번	이름	부서번호
100	김민수	001
200	이영희	002

부서번호	부서명	위치
001	인사팀	10층
002	영업팀	14층

→

부서번호	이름	부서명
001	김민수	인사팀
002	이영희	영업팀

- FROM 절에 조인할 테이블 나열
  - ORACLE: 콤마(,)로 나열. WHERE 절에 조인 조건 작성.
    (테이블 나열 순서 영향 X)

```
SELECT
    e.id AS employee_id,
    e.name,
    d.id AS department_id,
    d.name AS department_name
FROM employees e, departments d
WHERE e.department_id = d.id;
```

  - ANSI 표준: 테이블 사이에 JOIN 종류 나열. OUTER JOIN 시 순서 중요

```
SELECT
    e.id AS employee_id,
    e.name,
    d.id AS department_id,
    d.name AS department_name
FROM
    employees e
JOIN
    departments d ON e.department_id = d.id;
```

- 각 테이블 컬럼명이 동일할 경우, 컬럼명 앞에 테이블명 혹은 Alias를 붙여야 함
- ORACLE과 ANSI 표준이 서로 다르기 때문에 각 문법으로 변환하는 문제가 출제될 수 있음

■조인 종류

EQUI JOIN (등가 JOIN) 조건 형태에 따라 분류	JOIN 조건이 동등 조건인 경우 조인 조건이 =(equal) 비교를 통해 같은 값을 가지는 행을 연결 FROM 절에 조인하고자 하는 테이블 명시 (별칭 사용 가능) WHERE 절에 두 테이블의 공통 컬럼에 대한 조인을 나열 SELECT 테이블1.컬럼, 테이블2.컬럼 (*는 전체 컬럼 출력 가능) FROM 테이블1, 테이블2 WHERE 테이블1.컬럼 = 테이블2.컬럼
NON EQUI JOIN	JOIN 조건이 동등 조건이 아닌 경우 조인 조건이 = 아닌 연산자 일 때 (<, BETWEEN a AND B 등) SELECT 테이블1.컬럼, 테이블2.컬럼 (*는 전체 컬럼 출력 가능) FROM 테이블1, 테이블2 WHERE 테이블1.컬럼 [비교조건] 테이블2.컬럼
INNER JOIN 조인 결과에 따라 분류	두 테이블에서 조인 조건을 만족하는 행만 반환하는 가장 일반적인 조인 방식
OUTER JOIN	JOIN 조건에 성립하지 않는 데이터도 출력하는 경우 (LEFT/RIGHT/FULL OUTER JOIN으로 나뉨)
NATURAL JOIN	두 테이블에서 이름이 같은 모든 컬럼을 기준으로 자동으로 조인하는 방식
CROSS JOIN	두 테이블의 모든 가능한 조합을 반환하는 카테시안 곱 (Cartesian Product)을 생성
SELF JOIN	하나의 테이블을 두 번 이상 참조하여 연결하는 조인 한 테이블을 참조할 때마다 명시해야 하며, 테이블명이 중복되므로 별칭 사용

■조인 예제

- EQUI JOIN
- student 테이블과 department 테이블을 사용하여 각 학생과 학과를 함께 출력

student

STUDENT_ID	STUDENT_NAME	DEPARTMENT_ID
1	김철수	10
2	이영희	20
3	박민수	10

department

DEPARTMENT_ID	DEPARTMENT_NAME
10	컴퓨터공학과
20	경영학과
30	물리학과

<정답>

```
SELECT
    s.STUDENT_ID,
    s.STUDENT_NAME,
    d.DEPARTMENT_NAME
FROM
    STUDENTS s
JOIN
    DEPARTMENTS d ON s.DEPARTMENT_ID = d.DEPARTMENT_ID;
```

STUDENT_ID	STUDENT_NAME	DEPARTMENT_NAME
1	김철수	컴퓨터공학과
2	이영희	경영학과
3	박민수	컴퓨터공학과

- NON-EQUI JOIN
- student_score 테이블과 grade_criteria 테이블을 사용하여, 학생의 학점을 출력

student

STUDENT_ID	STUDENT_NAME	SCORE
1	김철수	85
2	이영희	92
3	박민수	78

grade_criteria

GRADE	MIN_SCORE	MAX_SCORE
A	90	100
B	80	89
C	70	79

<정답>

```
SELECT
    s.STUDENT_ID,
    s.STUDENT_NAME,
    s.SCORE,
    g.GRADE
FROM
    STUDENT_SCORES s,
    GRADE_CRITERIA g
WHERE
    s.SCORE BETWEEN g.MIN_SCORE AND g.MAX_SCORE;
```

STUDENT_ID	STUDENT_NAME	SCORE	GRADE
1	김철수	85	B
2	이영희	92	A
3	박민수	78	C

- 조인이 여러 개 있는 경우
- students와 courses(강의), enrollments(수강 성적)을 이용하여, 학생 정보와 수강 내역, 학점을 출력

enrollments

ENROLLMENT_ID	STUDENT_ID	COURSE_ID	GRADE
1	1	101	A
2	1	102	B+
3	2	101	A-
4	2	103	B
5	3	102	A+

student

STUDENT_ID	STUDENT_NAME	MAJOR
1	김철수	컴퓨터공학
2	이영희	경영학
3	박민수	물리학

courses

COURSE_ID	COURSE_NAME
101	데이터베이스
102	프로그래밍
103	경영학원론

<정답>

```
SELECT
    s.STUDENT_ID,
    s.STUDENT_NAME,
    s.MAJOR,
    c.COURSE_NAME,
    e.GRADE
FROM
    STUDENTS s,
    COURSES c,
    ENROLLMENTS e
WHERE
    s.STUDENT_ID = e.STUDENT_ID
    AND c.COURSE_ID = e.COURSE_ID;
```

만약, 필수 조인 조건에서 하나라도 누락될 경우,
카티시안 곱 발생 (정상 조인보다 많은 데이터 반환)

STUDENT_ID	STUDENT_NAME	MAJOR	COURSE_NAME	GRADE
1	김철수	컴퓨터공학	데이터베이스	A
1	김철수	컴퓨터공학	프로그래밍	B+
2	이영희	경영학	경영학원론	B
2	이영희	경영학	데이터베이스	A-
3	박민수	물리학	프로그래밍	A+

- SELF JOIN
- employees(직원) 테이블에서 직원과 매니저 이름을 함께 출력

employees

EMP_ID	EMP_NAME	MANAGER_ID	POSITION
1	김철수	NULL	CEO
2	이영희	1	부장
3	박민수	2	과장
4	정미경	2	과장

<정답>

```
SELECT
    e.EMP_ID,
    e.EMP_NAME,
    e.POSITION,
    m.EMP_NAME AS MANAGER_NAME,
    m.POSITION AS MANAGER_POSITION
FROM
    EMPLOYEES e,
    EMPLOYEES m
WHERE
    e.MANAGER_ID = m.EMP_ID
ORDER BY
    e.EMP_ID;
```

★ 조인쿼리가 달라지면, 결과가 아예 달라지기 때문에 이 점을 유의
e.MANAGER_ID = m.EMP_ID는 "직원의 관리자를 찾아라"는 의미

e.EMP_ID = m.MANAGER_ID는 "직원의 부하를 찾아라"는 의미.
(해당 로우(직원)가 매니저인 직원들을 찾아라)

★ NULL은 조인에 반응하지 않음
EMP_ID 1은 노출 되지 않는 걸 확인 할 수 있음

EMP_ID	EMP_NAME	POSITION	MANAGER_NAME	MANAGER_POSITION
2	이영희	부장	김철수	CEO
3	박민수	과장	이영희	부장
4	정미경	과장	이영희	부장

- 위 employees에서 직원과 매니저 이름을 함께 출력하되, 매니저가 CEO인 걸 제외

<정답> ★ ORACLE의 경우, 조인과 일반 조건이 전부 WHERE 절에 포함

```
SELECT
    e.EMP_ID,
    e.EMP_NAME,
    e.POSITION,
    e.DEPARTMENT,
    m.EMP_NAME AS MANAGER_NAME,
    m.POSITION AS MANAGER_POSITION
FROM
    EMPLOYEES e,
    EMPLOYEES m
WHERE
    e.MANAGER_ID = m.EMP_ID
    AND m.POSITION != 'CEO'
ORDER BY
    e.EMP_ID;
```

ANSI 표준의 경우, 조인은 ON 일반 조건은 WHERE 절에 표현

```
SELECT
    e.EMP_ID,
    e.EMP_NAME,
    e.POSITION,
    e.DEPARTMENT,
    m.EMP_NAME AS MANAGER_NAME,
    m.POSITION AS MANAGER_POSITION
FROM
    EMPLOYEES e
JOIN
    EMPLOYEES m ON e.MANAGER_ID = m.EMP_ID
WHERE
    m.POSITION != 'CEO'
ORDER BY
    e.EMP_ID;
```

EMP_ID	EMP_NAME	POSITION	MANAGER_NAME	MANAGER_POSITION
3	박민수	과장	이영희	부장
4	정미경	과장	이영희	부장

표준 조인

■ 표준 조인

- ANSI 표준의 INNER JOIN, CROSS JOIN, NATURAL JOIN, OUTER JOIN을 뜻함
- ORACLE에서는 *INNER JOIN, CROSS JOIN, NATURAL JOIN, OUTER JOIN*를 사용하지 않음

■ INNER JOIN (내부 조인)

- 두 테이블 간에 일치하는 값이 있는 레코드만 반환
- ORACLE에서는 FROM 절에서 테이블을 콤마(,)로 나열하면 기본 조인이 INNER JOIN으로 동작 (별도의 문법 존재 X)
- ANSI 표준에서는 INNER JOIN 혹은 JOIN이라고 줄여서 테이블명 사이에 작성
- ANSI 표준에서, 어떤 조건인지 USING 혹은 ON 조건절을 필수적으로 명시해야 함

■ ON 절

- 조인할 컬럼명이 같든 다르든 사용 가능 ⟷
- 컬럼명이 같은 경우, ⟷
  테이블명, 별칭을 명확히 지정해줘야 함
- ON 절의 괄호 생략 가능 ⟷
- ORACLE은 WHERE 절에 조인,일반조건 다 나열하지만,
  ANSI 표준에서는 ON 조건절에서는 조인조건 명시, WHERE 절에서는 일반조건 명시

■ USING 조건절

- 조인 컬럼명이 같을 경우 사용
- 테이블명, 별칭을 따로 붙이지 않고 컬럼명만 명시
- 괄호 필수

```sql
SELECT
    e.EMP_ID,
    e.EMP_NAME,
    d.DEPT_ID,
    d.DEPT_NAME
FROM
    EMPLOYEES e
INNER JOIN
    DEPARTMENTS d
ON
    e.DEPT_ID = d.DEPT_ID;
```

```sql
SELECT
    e.EMP_ID,
    e.EMP_NAME,
    d.DEPT_NAME
FROM
    EMPLOYEES e
INNER JOIN
    DEPARTMENTS d
USING
    (DEPT_ID);
```

■ NATUAL JOIN

- 두 테이블 간 컬럼명이 동일한 모든 컬럼에 대해서 EQUI JOIN 을 수행
- ON, USING, WHERE 절을 사용 불가
- 동일한 두 컬럼들의 데이터 타입도 동일해야 함
- 접두사 사용 불가

```sql
SELECT *
FROM EMPLOYEES
NATURAL JOIN DEPARTMENTS;
```

EMPLOYEES

EMP_ID	DEPT_ID	MANAGER_ID	EMP_NAME
1	10	100	김철수
2	20	NULL	이영희
3	10	100	박민수

DEPARTMENTS

DEPT_ID	MANAGER_ID	DEPT_NAME
10	100	영업부
20	NULL	개발부
30	200	인사부

NULL 은 같은 값으로 보지 않음

<결과>

DEPT_ID	MANAGER_ID	EMP_ID	EMP_NAME	DEPT_NAME
10	100	1	김철수	영업부
10	100	3	박민수	영업부

<NATUAL JOIN 주의사항>

- NATUAL JOIN에서 두 테이블의 컬럼명은 같지만, 다른 정보일 경우 조회되는 데이터가 없을 수 있음

ex) 학생 테이블에도 이름, 전화번호가 있고, 교수 테이블에도 이름과 전화번호 가 있을 때 학생의 이름과 교수 의 이름 서로 컬럼명을 동일하지만 데이터가 같지 않기 때문에 정상적으로 데이터가 조회 되지 않음

■ CROSS JOIN

- 테이블 간 JOIN 조건이 없는 경우, 생성 가능한 모든 데이터들의 조합 출력 (카타시안 곱) A 테이블 데이터 N개 * B 테이블 M개 = 즉, M * N 개의 데이터 출력
- ORACLE: WHERE 절에 조인조건을 명시하지 않으면 자연스럽게 CROSS JOIN으로 출력
- ANSI 표준: CROSS JOIN이라고 적고, 조인 조건 명시 X

COLORS

COLOR_ID	COLOR_NAME
1	Red
2	NULL

SIZE_ID	SIZE_NAME
A	Small
C	NULL

```sql
SELECT
    c.COLOR_ID,
    c.COLOR_NAME,
    s.SIZE_ID,
    s.SIZE_NAME
FROM
    COLORS c
CROSS JOIN
    SIZES s;
```

<결과> 모든 데이터의 조합을 출력해야 하므로, NULL 존재해도 출력

COLOR_ID	COLOR_NAME	SIZE_ID	SIZE_NAME
1	Red	A	Small
1	Red	B	NULL
2	NULL	A	Small
2	NULL	B	NULL

■ OUTER JOIN

- INNER JOIN과 대비되는 조인
- JOIN 조건에 동일한 값이 없는 경우에도 행을 반환하고 싶을 때 사용
- INNER JOIN은 NULL 인 경우에 EQUI JOIN 시 출력되지 않았는데, OUTER JOIN 은 NULL 인 경우에도 출력 가능
- 어느 테이블을 기준으로 둘지에 따라 LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN으로 구분 (OUTER 생략 가능. LEFT JOIN, RIGHT JOIN …)
  - 즉, 테이블의 위치가 중요

LEFT OUTER JOIN	FROM 절에 나열된 왼쪽 테이블이 기준이 되어서 우측 테이블 데이터를 채움. (즉, 왼쪽 테이블 데이터는 다 노출) 왼쪽에만 존재하는 데이터일 경우, 우측 컬럼들은 NULL로 채워서 출력
RIGHT OUTER JOIN	FROM 절에 나열된 우측 테이블이 기준이 되어서 좌측 테이블 데이터를 채움. (즉, 우측 테이블 데이터는 다 노출) 우측에만 존재하는 데이터일 경우, 좌측 컬럼들은 NULL로 채워서 출력
FULL OUTER JOIN	두 테이블 모두 기준이 되어서 전체 데이터 기준으로 결과 생성 후 중복 데이터 제거하여 출력 ORACLE에서는 FULL OUTER JOIN이 없음 LEFT JOIN과 RIGHT JOIN 의 결과에 UNION 연산 한 것과 결과가 동일함

FROM a LEFT OUTER JOIN b = FROM b RIGHT OUTER JOIN a 은 동일

■ LEFT OUTER JOIN 예제

- 학생과 수업을 출력할 때, 수업을 안 듣고 있는 학생도 포함해 출력하고 싶을 경우

STUDENTS

STUDENT_ID	NAME	CLASS_ID
1	김철수	101
2	박민수	NULL

CLASSES

CLASS_ID	CLASS_NAME
101	수학
102	영어
103	과학

<결과>

STUDENT_ID	NAME	CLASS_ID	CLASS_NAME
1	김철수	101	수학
2	박민수	NULL	NULL

CLASSES에만 존재하는 정보는 노출되지 않음

- ANSI 표준 LEFT OUTER JOIN

```
SELECT
    s.STUDENT_ID,
    s.NAME,
    s.CLASS_ID,
    c.CLASS_NAME
FROM
    STUDENTS s
LEFT JOIN
    CLASSES c ON s.CLASS_ID = c.CLASS_ID
ORDER BY
    s.STUDENT_ID;
```

기준이 되는 테이블을 왼쪽에 둠 (STUDENT)

- ORACLE LEFT OUTER JOIN

```
SELECT
    s.STUDENT_ID,
    s.NAME,
    s.CLASS_ID,
    c.CLASS_NAME
FROM
    STUDENTS s, CLASSES c
WHERE
    s.CLASS_ID = c.CLASS_ID(+)
```

기준이 되는 테이블 반대편 컬럼에 (+)을 붙임

■ RIGHT OUTER JOIN

- LEFT OUTER JOIN과 기준이 되는 방향만 다를 뿐 동일하기 때문에 예제 생략

■ FULL OUTER JOIN

- LEFT OUTER JOIN의 결과와 RIGHT OUTER JOIN의 결과가 합쳐진 뒤 중복 제거되어 출력 (잘 사용되지는 않음)
- 수업을 듣는 학생 정보를 출력하고자 할 때,
  수업을 안 듣고 있는 학생도 포함하고,
  학생이 수강하지 않는 수업 정보도 포함해서 출력하고 싶은 경우

STUDENTS

STUDENT_ID	NAME	CLASS_ID
1	김철수	101
2	이영희	102
3	박민수	NULL

CLASSES

CLASS_ID	CLASS_NAME
101	수학
102	영어
103	과학

<결과>

STUDENT_ID	NAME	CLASS_ID	CLASS_NAME
1	김철수	101	수학
2	이영희	102	영어
3	박민수	NULL	NULL
NULL	NULL	103	과학

- ANSI 표준 FULL OUTER JOIN

```
SELECT
    s.STUDENT_ID,
    s.NAME,
    s.CLASS_ID,
    c.CLASS_NAME
FROM
    STUDENTS s FULL OUTER JOIN CLASSES c
    ON s.CLASS_ID = c.CLASS_ID
```

UNION은 두 개 이상의 SELECT 문의 결과를 하나로 합치는 연산자
이 때, 중복된 행은 자동으로 제거

- ORACLE LEFT OUTER JOIN

```
SELECT
    s.STUDENT_ID,
    s.NAME,
    s.CLASS_ID,
    c.CLASS_NAME
FROM
    STUDENTS s, CLASSES c
WHERE
    s.CLASS_ID = c.CLASS_ID(+)
UNION
SELECT
    s.STUDENT_ID,
    s.NAME,
    c.CLASS_ID,
    c.CLASS_NAME
FROM
    STUDENTS s, CLASSES c
WHERE
    s.CLASS_ID(+) = c.CLASS_ID
```

② SQL 활용

 # 서브 쿼리

■ 서브 쿼리

- 하나의 SQL 안에 또 다른 SQL 문이 있는 것을 뜻함
- 괄호로 묶어야 함

ex) 평균 급여보다 높은 급여를 받는 직원들의 이름과 급여를 조회

```
SELECT ename, sal
FROM emp
WHERE sal > (SELECT AVG(sal) FROM emp);
```

■ 서브 쿼리 사용 위치

- SELECT 절
- FROM 절
- WHERE 절
- HAVING 절
- ORDER BY 절
- 다른 DML(INSERT, DELETE, UPDATE …) 절
- GROUP BY 절에는 사용 불가

■ 서브 쿼리 종류

- **동작하는 방식**에 따라 분류 (출제 비중 ↓)

UN-CORRELATED 서브 쿼리	메인 쿼리와 독립적으로 실행되는 서브쿼리 (메인 쿼리의 컬럼을 가지고 있지 않음) 서브쿼리가 먼저 실행되고, 그 결과를 이용해 메인 쿼리 실행 독립적으로 실행 가능 `SELECT * FROM emp WHERE sal > (SELECT AVG(sal) FROM emp);`
CORRELATED 서브 쿼리	메인 쿼리의 칼럼을 참조하는 서브쿼리 메인 쿼리가 실행되면서 서브쿼리도 함께 실행 메인 쿼리에 의존적 `SELECT * FROM emp e WHERE sal > ( SELECT AVG(sal) FROM emp WHERE deptno = e.deptno );`

- **위치**에 따라 분류

스칼라 서브 쿼리	SELECT 절에 위치하여 단일 값(결과를 컬럼처럼)을 반환 조인 대체 용도로 자주 사용 `SELECT e.empno, e.ename, (SELECT d.dname FROM dept d WHERE d.deptno = e.deptno) AS dept_name FROM emp e;`
인라인 뷰	FROM 절에 위치하여 결과를 테이블처럼 사용 `SELECT e.ename, d.dname, e.sal FROM (SELECT * FROM emp WHERE sal > 2000) e, dept d WHERE e.deptno = d.deptno;`
WHERE 절 서브 쿼리	WHERE 절에 위치하여 조건으로 사용 단일 행 또는 다중 행 반환 가능 비교 상수 자리에 값을 전달하기 위한 목적으로 사용 리턴 데이터의 형태에 따라 아래처럼 구분 • 단일행 서브 쿼리 • 다중 행 서브 쿼리 • 다중 컬럼 서브 쿼리 • 상호 연관 서브 쿼리 `SELECT ename, sal FROM emp WHERE sal > (SELECT AVG(sal) FROM emp);`

■ 스칼라 서브 쿼리

- 각 행(컬럼)마다 서브 쿼리의 결과가 하나(단일행 서브 쿼리) 여야 함
- 서브 쿼리의 값이 없더라도 NULL로 노출됨 (즉, 기본적으로 OUTER JOIN의 형태로 출력)

[다음 장에서 스칼라 서브 쿼리 예제 계속]

- **스칼라 서브 쿼리 예제 – 각 직원의 매니저 정보 출력**

EMP

EMPNO	ENAME	DEPTNO	MGR	SAL
7839	KING	10	NULL	5000
7782	CLARK	10	7839	2450
7566	JONES	20	7839	2975

<정답>

```
SELECT e.empno, e.ename,
       (
            SELECT m.ename
            FROM emp m
            WHERE m.empno = e.mgr
       ) AS manager_name
FROM emp e;
```

결과

EMPNO	ENAME	MANAGER_NAME
7839	KING	NULL
7782	CLARK	KING
7566	JONES	KING

없는 데이터는 NULL 출력
OUTER JOIN의 효과가 남

- **스칼라 서브 쿼리 예제 – 서브 쿼리의 결과가 단일 행이 아니면 오류**

```
SELECT e.empno, e.ename,
       (
            SELECT m.ename
            FROM emp m
       ) AS manager_name
FROM emp e;
```

단일 행 하위 쿼리가 두 개 이상의 행을 반환합니다.

위 쿼리와 다르게
조인을 위한 WHERE 절(WHERE m.empno = e.egr)이
빠지자 여러 행이 return 되어 오류가 발생하는 것을
확인

- **스칼라 서브 쿼리 예제 – 서브 쿼리에 함수를 통해 단일 값으로 반환하면 사용 가능**

```
SELECT d.deptno, d.dname,
       (
            SELECT COUNT(*)
            FROM emp
            WHERE deptno = d.deptno
       ) AS emp_count,
       (
            SELECT AVG(sal)
            FROM emp
            WHERE deptno = d.deptno
       ) AS avg_salary
FROM dept d;
```

원래 였으면 다중 행 반환으로 실행이 불가능 하지만
함수를 통해 단일 행으로 반환하여 아래처럼 출력
가능

결과

DEPTNO	DNAME	EMP_COUNT	AVG_SALARY
10	ACCOUNTING	2	3725.00
20	RESEARCH	1	2975.00
30	SALES	0	NULL

■ 인라인 뷰 서브 쿼리

- 뷰 형태로 테이블 처럼 FROM 절에 조회할 데이터를 정의
- 테이블 명이 없어서 조인 위해서는 별칭 필요! (물론, 단독으로 사용되면 불필요)
- 순서상 제일 먼저 수행되기 때문에 서브 쿼리의 결과를 어느 절에서도 참조 가능
- 보통 메인 쿼리 테이블과 조인하기 위한 목적으로 사용

- **인라인 뷰 예제 – 각 부서별 직원 수와 평균 급여를 계산하여 평균 급여가 2000 이상인 부서 정보를 조회**

EMP (직원 급여)

EMPNO	ENAME	DEPTNO	SAL
7839	KING	10	5000
7782	CLARK	10	2450
7566	JONES	20	2975
7654	MARTIN	30	1250
7499	ALLEN	30	1600

DEPT (부서명)

DEPTNO	DNAME
10	ACCOUNTING
20	RESEARCH
30	SALES

```
SELECT d.dname, e.emp_count, e.avg_sal
FROM dept d,
     (SELECT deptno,
             COUNT(*) emp_count,
             AVG(sal) avg_sal
      FROM emp
      GROUP BY deptno) e
WHERE d.deptno = e.deptno
    AND e.avg_sal > 2000
ORDER BY e.avg_sal DESC;
```

1. 별칭 e를 통해 테이블 처럼 사용 하고 있음
2. 연산 함수로 계산 된 걸 밖에서 사용하려면 별칭 필요 (ex. avg_sal)
3. WHERE 절에서 일반조건으로 서브 쿼리 결과가 사용 가능

결과

DNAME	EMP_COUNT	AVG_SAL
ACCOUNTING	2	3725.00
RESEARCH	1	2975.00

■ WHERE 절 서브 쿼리 종류

- 단일행 서브 쿼리: 서브 쿼리의 결과가 1 개의 행이 리턴 되는 형태

연산자	의미
=	같다
>	크다
>=	크거나 같다
<	작다
<=	작거나 같다
<>	같지 않다

- **단일행 예제 – 평균 급여보다 많이 받는 모든 직원의 정보를 조회**

EMP (직원 정보)

EMPNO	ENAME	DEPTNO	SAL
7839	KING	10	5000
7782	CLARK	10	2450
7566	JONES	20	2975

```
SELECT empno, ename, deptno, sal
FROM emp
WHERE sal > (SELECT AVG(sal) FROM emp);
```

상수로 넣는 부분을 매번 따로 조회하지 않고,
조회 시점의 급여 평균을 계산해서 조회되도록 하는
효과

결과

EMPNO	ENAME	DEPTNO	SAL
7839	KING	10	5000
7566	JONES	20	2975

- 다중 행 서브 쿼리: 서브 쿼리의 결과가 여러 행이 리턴 되는 형태
  (단일 행이 아니다 보니 비교 연산(=, >, <…)을 사용할 수 없음)
- 서브 쿼리의 결과를 요약해 단일행 연산자 사용하거나 다중 행 연산자를 사용

연산자	의미
IN	서브 쿼리의 결과 중 하나라도 일치하면 참
ANY	서브 쿼리의 결과 중 하나라도 조건을 만족하면 참
ALL	서브 쿼리의 모든 결과가 조건을 만족하면 참
EXISTS	서브 쿼리의 결과가 존재하면(하나 이상이면) 참

연산자	의미
> ANY(1, 2)	최솟값인 1보다 큰 행들을 반환 1점 또는 2점보다 높은 점수 즉, 1 보다만 높으면 됨 a > 1 OR a > 2 => a > 1 이 됨
< ANY(1, 2)	최댓값인 2보다 작은 행들을 반환 1점 또는 2점보다 낮은 점수 즉, 2 보다만 낮으면 됨 a < 1 OR a < 2 => a < 2 가 됨
> ALL(1, 2)	최댓값인 2보다 큰 행들을 반환 1점과 2점 모두보다 높은 점수 즉, 2 를 넘어야 함 a > 1 AND a > 2 => a > 2가 됨
< ALL(1, 2)	최소값인 1보다 작은 행들을 반환 1점과 2점 모두보다 낮은 점수 즉, 1 미만이어야 함 a < 1 AND a < 2 => a < 1 이 됨

- **다중 행 예제 – 다중 행 연산자 없이 다중 행 반환 시 오류**

```
SELECT empno, ename, deptno, sal
FROM emp
WHERE sal > (SELECT sal FROM emp);
```

단일 행 하위 쿼리가 두 개 이상의 행을 반환합니다.

- **다중 행 예제 – 다중 행 서브 쿼리 연산자로 쓰지 않고 하나의 행으로 요약해서 사용**

```
SELECT ename, job, sal
FROM emp
WHERE sal > (SELECT MIN(sal) FROM emp);
```

두 쿼리의 결과가 동일

- **다중 행 예제 – 다중 행 서브 쿼리 연산자로 표현**

```
SELECT ename, job, sal, deptno
FROM emp
WHERE sal > ANY (SELECT sal FROM emp);
```

하나는 요약해 계산
하나는 다중 행 연산자로 계산

- **다중 행 예제 – 다중 컬럼 서브 쿼리**
- 각 부서에서 가장 높은 급여를 받는 직원의 부서번호와 급여를 조회

EMP (직원 정보)

EMPNO	ENAME	JOB	SAL	DEPTNO
7839	KING	PRESIDENT	5000	10
7782	CLARK	MANAGER	2450	10
7566	JONES	MANAGER	2975	20
7902	FORD	ANALYST	3000	20
7369	SMITH	CLERK	800	20

```
SELECT DEPTNO, SAL
FROM EMP
WHERE (DEPTNO, SAL) IN (
    SELECT DEPTNO, MAX(SAL)
    FROM EMP
    GROUP BY DEPTNO
);
```

1. 서브 쿼리 결과가 여러 컬럼이 리턴 되는 형태로 메인 쿼리와 비교 컬럼이 2개 이상
2. 반환 되는 컬럼을 대소 비교는 불가능 함

결과

EMPNO	ENAME	JOB	SAL	DEPTNO
7839	KING	PRESIDENT	5000	10
7902	FORD	ANALYST	3000	20

- **다중 행 예제 – 상호 연관 서브 쿼리**
- 각 직원의 급여가 자신이 속한 부서의 평균 급여보다 높은 경우, 해당 직원의 정보를 조회

EMP (직원 정보)

EMPNO	ENAME	DEPTNO	SAL
7839	KING	10	5000
7782	CLARK	10	2450
7566	JONES	20	2975
7369	SMITH	20	800
7499	ALLEN	30	1600

```
SELECT e.EMPNO, e.ENAME, e.DEPTNO, e.SAL
FROM EMP e
WHERE e.SAL > (
    SELECT AVG(SAL)
    FROM EMP
    WHERE DEPTNO = e.DEPTNO
);
```

비교할 컬럼은 SAL이기에 WHERE 절에서는 SAL만 반환
DEPTNO가 일치해야 하기 때문에 서브 쿼리에서 일치하는 조건만 반환

수행 순서가 메인 쿼리 수행 후 서브 쿼리를 수행하므로,
서브 쿼리의 WHERE 절이 메인 쿼리로 갈 수 는 없음

1. 메인 쿼리와 서브 쿼리의 비교를 수행하는 형태 다중 컬럼 형태로는 위에 언급했듯이 대소 비교 불가
2. 대소 비교할 컬럼을 메인 쿼리에 일치하게 서브 쿼리로 결과 전달

결과

EMPNO	ENAME	DEPTNO	SAL
7839	KING	10	5000
7566	JONES	20	2975

실행 순서

1. 메인 쿼리 FROM EMP
2. 메인 쿼리 WHERE e.SAL 확인
3. 서브 쿼리 FROM EMP
4. 서브 쿼리 WHERE DEPTNO 비교
5. 서브 쿼리 SELECT AVG(SAL) 반환
6. 메인 쿼리 WHERE 로 전달되어 e.SAL > 서브 쿼리 결과값과 비교
7. 메인 쿼리 SELECT 결과값 출력

■서브 쿼리 주의 사항

- 특별한 경우(TOP-N 분석 …)을 제외하고, 서브 쿼리절에 ORDER BY 사용 불가

```
SELECT EMP_NAME, SALARY
FROM EMPLOYEES
WHERE DEPT_ID IN (
    SELECT DEPT_ID
    FROM EMPLOYEES
    GROUP BY DEPT_ID
    ORDER BY AVG(SALARY) DESC
);
```

오른쪽 괄호가 없습니다.

```
SELECT EMP_NAME, SALARY
FROM EMPLOYEES
WHERE DEPT_ID IN (
    SELECT DEPT_ID
    FROM EMPLOYEES
    GROUP BY DEPT_ID
);
```

EMP_NAME	SALARY
John Doe	50000
Jane Smith	60000
Bob Johnson	55000
Alice Brown	52000
Charlie Davis	58000

- 단일 행 서브 쿼리와 다중 행 서브 쿼리에 따라 연산자 선택이 중요

집합 연산자

■집합 연산자

- SELECT의 결과를 하나의 집합이라고 했을 때, 그 집합에 대한 합집합, 교집합, 차집합 연산
- SELECT 문과 SELECT 문 사이에 집합 연산자 정의
- 연산 결과에 대한 데이터 타입, 컬럼 명을 첫 번째 집합에 의해 결정

■집합 연산자 주의사항

- 두 집합의 컬럼 수, 컬럼 순서, 데이터 타입 일치
- 컬럼의 사이즈 크기 까지 동일할 필요는 없음
- 개인 SELECT 절에는 ORDER BY 불가 (전체 집합에 대한 ORDER BY 가능)

■합집합

- 두 집합 결과의 전체를 출력

UNION	중복된 데이터를 제거하고 딱 한 번만 출력 제거하는 과정에서 내부적으로 정렬 수행 (그만큼 성능 하락)
UNION ALL	중복된 데이터 제거 없이 전체 출력 불필요한 정렬이 진행되지 않기 때문에 중복이 없는 상황에서는 UNION ALL을 사용

- **합집합 예제 – A 테이블과 B 테이블 UNION**

A

ID	NAME
1	John
2	Jane

B

ID	NAME
2	Jane
3	Mike

```
SELECT ID, NAME FROM A
UNION
SELECT ID, NAME FROM B;
```

결과

ID	NAME
1	John
2	Jane
3	Mike

- **합집합 예제 – A 테이블과 B 테이블 UNION ALL**

```
SELECT ID, NAME FROM A
UNION ALL
SELECT ID, NAME FROM B;
```

결과

1	John
2	Jane
2	Jane
3	Mike

■교집합

- 두 집합 결과의 공통 부분만 출력
- SELECT 문 사이에 INTERSECT 추가
- **교집합 예시 – A 테이블과 B 테이블 INTERSECT**

A

ID	NAME
1	John
2	Jane

B

ID	NAME
2	Jane
3	Mike

```
SELECT ID, NAME FROM A
INTERSECT
SELECT ID, NAME FROM B;
```

결과

ID	NAME
2	Jane

■ 차집합

- SELECT 문 사이에 MINUS 추가
- A 집합과 B 집합 사이에서 A 집합에만 존재하는 값 출력
- A – B 는 A 집합에만 존재하는 데이터, B – A 는 B 집합에만 존재하는 데이터 (순서 주의)₩

- **차집합 예제 – A 테이블과 B 테이블 MINUS**

A

ID	NAME
1	John
2	Jane

B

ID	NAME
2	Jane
3	Mike

```
SELECT ID, NAME FROM A
MINUS
SELECT ID, NAME FROM B;
```

결과

ID	NAME
1	John

- **차집합 예제 – B 테이블과 A 테이블 MINUS**

A

ID	NAME
1	John
2	Jane

B

ID	NAME
2	Jane
3	Mike

```
SELECT ID, NAME FROM B
MINUS
SELECT ID, NAME FROM A;
```

결과

ID	NAME
3	Mike

■ 집합 연산자 주의 사항 예제

- ORDER BY는 개별 SELECT 에는 전달 불가능 하지만 전체 결과에는 전달 가능

```
SELECT emp_id, emp_name, salary
FROM employees_dept1
ORDER BY salary DESC
UNION
SELECT emp_id, emp_name, salary
FROM employees_dept2;
```

SQL 명령이 제대로 종료되지 않았습니다.

개별 쿼리에 대한 ORDER BY 불가

```
SELECT emp_id, emp_name, salary
FROM employees_dept1
UNION
SELECT emp_id, emp_name, salary
FROM employees_dept2
ORDER BY salary DESC;
```

전체 결과에 대한 ORDER BY 가능

EMP_ID	EMP_NAME	SALARY
2	Bob	55000
6	Frank	53000
4	David	52000
1	Alice	50000
5	Eve	48000
3	Charlie	45000

 # 그룹 함수

■ 그룹 함수

- 숫자 함수 중 여러 입력 값을 하나의 요약 값으로 출력하는 다중 행 함수
- GROUP BY 절에 의해 그룹별 연산 결과 반환
- NULL은 무시하고 연산
- SUM, AVG, MIN, COUNT, MIN, MAX, VARIANCE, STDDEV… 등 존재
  (위의 함수 파트에서 설명하여 여기서는 생략)

■ GROUP BY 절에 사용하는 함수 ★★

- 여러 컬럼에 대한 GROUP BY 결과를 동시에 합집합하여 출력
- GROUP BY 함수로 표현하고 이를 대체하는 쿼리를 찾는 문제 출제 빈도 높음

GROUPING SETS(A, B,…)	A별, B별 … 그룹 연산 결과 출력
ROLLUP(A,B)	① A별, ② (A, B)별, ③ 전체 그룹 결과 출력 (3 종류)
CUBE(A,B)	① A별, ② B별, ③ (A, B)별, ④ 전체 그룹 연산 결과 출력 (4 종류)

■ GROUPING SET

- 표현식: GROUP BY GROUPING SET(컬럼1, 컬럼2)
- 컬럼1과 컬럼2 연산 결과를 출력
- NULL 이나 ()를 쓰면 모든 행에 대한 집계

- **GROUPING SET 예제 – 판매 기록에서 카테고리와 지역별 판매액과 총 판매액 출력**

판매 기록

product_id	category	region	sales_amount
1	김치	서울	1000
2	김치	부산	1500
3	된장	서울	800
4	된장	부산	1200

```
SELECT
    category,
    region,
    SUM(sales_amount) as total_sales
FROM
    sales
GROUP BY GROUPING SETS (
    (category),
    (region),
    ()
)
ORDER BY category, region;
```

결과

category	region	total_sales
김치	NULL	2500
된장	NULL	2000
NULL	부산	2700
NULL	서울	1800
NULL	NULL	4500

카테고리 GROUP BY 결과

지역(region) GROUP BY 결과

총 합 연산

- **위 쿼리와 동일한 출력을 쿼리** ★★

```
SELECT category, NULL as region, SUM(sales_amount) as total_sales
FROM sales                                  카테고리 GROUP BY 결과
GROUP BY category

UNION ALL                                   더하기

SELECT NULL as category, region, SUM(sales_amount) as total_sales
FROM sales                                  지역(region) GROUP BY 결과
GROUP BY region

UNION ALL                                   더하기

SELECT NULL as category, NULL as region, SUM(sales_amount) as total_sales
FROM sales                                  총 합 연산

ORDER BY category, region;
```

■ ROLLUP

- 표현식: ROLLUP(컬럼1, 컬럼2)
- 컬럼1, (컬럼1, 컬럼2), 전체 행에 대한 집계 출력
- 기본적으로, 전체 행에 대한 집계가 출력됨
- 컬럼1과 (컬럼1, 컬럼2) 로 출력되기 때문에 나열 순서가 중요함

- **ROLLUP 예제**

다음 페이지에 계속

- **ROLLUP 예제 – 카테고리 별, (지역, 카테고리)별, 전체 합계 출력**

판매 기록

product_id	category	region	sales_amount
1	김치	서울	1000
2	김치	부산	1500
3	된장	서울	800
4	된장	부산	1200

```
SELECT
    category,
    region,
    SUM(sales_amount) as total_sales
FROM
    sales
GROUP BY ROLLUP(category, region)
ORDER BY category, region;
```

결과

category	region	total_sales	
김치	부산	1500	카테고리, 지역 GROUP BY 결과 (김치, 지역)
김치	서울	1000	
김치	NULL	2500	카테고리 GROUP BY 결과 (김치)
된장	부산	1200	카테고리, 지역 GROUP BY 결과 (된장, 지역)
된장	서울	800	
된장	NULL	2000	카테고리 GROUP BY 결과 (된장)
NULL	NULL	4500	총 합 연산

- **위 쿼리와 동일한 출력을 쿼리** ★★

```
SELECT category, region, SUM(sales_amount) as total_sales
FROM sales                                  카테고리, 지역 GROUP BY 결과
GROUP BY category, region

UNION ALL                                   더하기

SELECT category, NULL as region, SUM(sales_amount) as total_sales
FROM sales                                  카테고리 GROUP BY 결과
GROUP BY category

UNION ALL                                   더하기

SELECT NULL as category, NULL as region, SUM(sales_amount) as total_sales
FROM sales                                  총 합 연산

ORDER BY category, region;
```

■ CUBE
- 표현식: CUBE(컬럼1, 컬럼2)
- 컬럼1과 컬럼2, (컬럼1, 컬럼2), 전체 행 집계 연산 결과를 출력
- 기본적으로 전체 행 집계 연산 결과를 출력

- **CUBE 예제 - 판매 기록에서 카테고리별 판매액, 지역별 판매액, (카테고리, 지역)별 판매액과 총 판매액 출력**

판매 기록

product_id	category	region	sales_amount
1	김치	서울	1000
2	김치	부산	1500
3	된장	서울	800
4	된장	부산	1200

```
SELECT
    category,
    region,
    SUM(sales_amount) as total_sales
FROM
    sales
GROUP BY CUBE(category, region)
ORDER BY category, region;
```

결과

category	region	total_sales	
김치	부산	1500	카테고리, 지역 GROUP BY 결과 (김치, 지역)
김치	서울	1000	
김치	NULL	2500	카테고리 GROUP BY 결과 (김치)
된장	부산	1200	카테고리, 지역 GROUP BY 결과 (된장, 지역)
된장	서울	800	
된장	NULL	2000	카테고리 GROUP BY 결과 (된장)
NULL	부산	2700	지역 GROUP BY 결과 (부산)
NULL	서울	1800	지역 GROUP BY 결과 (서울)
NULL	NULL	4500	총 합 연산

- **위 쿼리와 동일한 출력을 쿼리 – GROUPING SET** ★★

```
SELECT
    category,
    region,
    SUM(sales_amount) as total_sales
FROM
    sales
GROUP BY GROUPING SETS (
    (category, region),     카테고리와 지역별 판매액
    (category),             카테고리별 총 판매액
    (region),               지역별 총 판매액
    ()                      전체 총 판매액
)
ORDER BY category, region;
```

- **위 쿼리와 동일한 출력을 쿼리 – UNION ALL** ★★

```
SELECT category, region, SUM(sales_amount) as total_sales
FROM sales                                  카테고리와 지역별 판매액
GROUP BY category, region

UNION ALL                                   더하기

SELECT category, NULL as region, SUM(sales_amount) as total_sales
FROM sales                                  카테고리별 총 판매액
GROUP BY category

UNION ALL                                   더하기

SELECT NULL as category, region, SUM(sales_amount) as total_sales
FROM sales                                  지역별 총 판매액
GROUP BY region

UNION ALL                                   더하기

SELECT NULL as category, NULL as region, SUM(sales_amount) as total_sales
FROM sales                                  전체 총 판매액

ORDER BY category, region;
```

윈도우 함수

■ 윈도우 함수

- 서로 다른 행들의 비교 및 연산을 위해 사용
  (ex. 현재 행까지의 누적 판매량, 현재 행이 몇 번째 행인지 등)
- GROUP BY를 사용하지 않고, 그룹 연산을 사용 가능

■ 윈도우 함수 문법

```
SELECT
    윈도우함수(대상) OVER (
        [PARTITION BY 컬럼]
        [ORDER BY 컬럼1 [ASC | DESC]]
        [ROWS | RANGE 연산 범위 설정]
    ) AS [별칭]
FROM  테이블명;
```

그룹 연산을 수행할 그룹핑 컬럼 지정 (지정하지 않으면 전체 행)
컬럼 정렬 (만약, PARTITION BY 설정되어 있으면 같은 그룹핑 내에서 정
연산의 범위 지정, ORDER BY 절이 필수

PARTITION BY → ORDER BY → ROWS 순으로 순서 중요

■ 윈도우 함수 예제

판매 테이블

sale_date	product	amount
2024-01-01	김치	12000
2024-01-01	라면	3500
2024-01-02	김치	15000
2024-01-02	라면	3000
2024-01-03	김치	13000
2024-01-03	라면	4000

- **윈도우 함수 예제 – 날짜별 상품 판매액과 전체 상품 판매 합계 동시 출력**

```
SELECT
    sale_date,
    product,
    amount,
    SUM(amount) OVER () AS total_amount
FROM sales;
```

OVER에 아무것도 설정하지 않으면, 전체 행으로 그룹 함수 연산

결과

sale_date	product	amount	total_amount
2024-01-01	김치	12000	50500
2024-01-01	라면	3500	50500
2024-01-02	김치	15000	50500
2024-01-02	라면	3000	50500
2024-01-03	김치	13000	50500
2024-01-03	라면	4000	50500

전체 행 합산 결과 출력

```
SELECT
    sale_date,
    product,
    amount,
    (SELECT SUM(amount) FROM sales) AS total_amount
FROM sales;
```

스칼라 서브 쿼리로 동일한 결과 출력 가능

- **윈도우 함수 예제 – 날짜별 상품 판매액과 전체 행 개수 출력**

```
SELECT
    sale_date,
    product,
    amount,
    COUNT(*) OVER () AS total_count
FROM sales;
```

결과

sale_date	product	amount	total_count
2024-01-01	김치	12000	6
2024-01-01	라면	3500	6
2024-01-02	김치	15000	6
2024-01-02	라면	3000	6
2024-01-03	김치	13000	6
2024-01-03	라면	4000	6

전체 행 개수 결과 출력

- **윈도우 함수 예제 – 날짜별 상품 판매액과 전체 평균 판매액 출력**

```
SELECT
    sale_date,
    product,
    amount,
    AVG(amount) OVER () AS avg_amount
FROM sales;
```

결과

sale_date	product	amount	avg_amount
2024-01-01	김치	12000	8416.67
2024-01-01	라면	3500	8416.67
2024-01-02	김치	15000	8416.67
2024-01-02	라면	3000	8416.67
2024-01-03	김치	13000	8416.67
2024-01-03	라면	4000	8416.67

전체 행 평균 결과 출력

- **윈도우 함수 예제 – 날짜별 상품 판매액과 전체 행 중 최소 판매액 출력**

```
SELECT
    sale_date,
    product,
    amount,
    MIN(amount) OVER () AS min_amount
FROM sales;
```

결과

sale_date	product	amount	min_amount
2024-01-01	김치	12000	3000
2024-01-01	라면	3500	3000
2024-01-02	김치	15000	3000
2024-01-02	라면	3000	3000
2024-01-03	김치	13000	3000
2024-01-03	라면	4000	3000

전체 행 최소 행 결과 출력

- **윈도우 함수 예제 – 날짜별 상품 판매액과 전체 행 중 최대 판매액 출력**

```
SELECT
    sale_date,
    product,
    amount,
    MAX(amount) OVER () AS max_amount
FROM sales;
```

결과

sale_date	product	amount	max_amount
2024-01-01	김치	12000	15000
2024-01-01	라면	3500	15000
2024-01-02	김치	15000	15000
2024-01-02	라면	3000	15000
2024-01-03	김치	13000	15000
2024-01-03	라면	4000	15000

전체 행 최대 행 결과 출력

■ 윈도우 함수 ROWS, RANGE

- ROWS는 대상의 값이 같더라도 각 행씩 연산
- RANGW는 대상의 값이 동일할 경우, 하나의 RANGE로 묶어서 동시에 연산 (기본 설정)

- 연산 범위 설정: BETWEEN A AND B
- A는 시작점을 정의
  - CURRENT ROW: 현재 행부터
  - UNBOUNDED PRECEDING: 처음부터 (기본 설정)
  - N PRECEDING: N 행 이전 부터
- B는 마지막 지점을 정의
  - CURRENT ROW: 현재 행까지 (기본 설정)
  - UNBOUNDED FOLLOWING: 마지막 행까지
  - N FOLLOWING: N 행 이후 까지

■ 윈도우 함수 ROWS, RANGE 예제 - 각 판매일 별 누적 합계 출력

판매 테이블

sale_date	product	amount
2024-01-01	김치	12000
2024-01-02	김치	15000
2024-01-02	라면	3000
2024-01-03	라면	4000

- **RANGE 범위 설정 (DEFAULT 기본 설정 동작)**

```sql
SELECT
    sale_date,
    product,
    amount,
    SUM(amount) OVER (
        ORDER BY sale_date
    ) AS total_range
FROM sales
ORDER BY sale_date, product;
```

```sql
SELECT
    sale_date,
    product,
    amount,
    SUM(amount) OVER (
        ORDER BY sale_date
        RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS total_range
FROM sales
ORDER BY sale_date, product;
```

위 두 쿼리의 출력은 동일

결과

sale_date	product	amount	total_range
2024-01-01	김치	12000	12000
2024-01-02	김치	15000	30000
2024-01-02	라면	3000	30000
2024-01-03	라면	4000	34000

RANGE는 같은 대상 sale_date가 동일한 값에 대해서는 그 행을 묶어서 표현
이전 값 12,000 + (15,000+3,000) 하여 30,000으로 출력

- **ROWS 범위 설정**

```sql
SELECT
    sale_date,
    product,
    amount,
    SUM(amount) OVER (
        ORDER BY sale_date, product
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS total_rows
FROM sales
ORDER BY sale_date, product;
```

결과

sale_date	product	amount	total_rows
2024-01-01	김치	12000	12000
2024-01-02	김치	15000	27000
2024-01-02	라면	3000	30000
2024-01-03	라면	4000	34000

ROWS는 각 행 들을 개별적으로 순차 처리

- **ROWS 범위 설정 – BETWEEN 수정**

```sql
SELECT
    sale_date,
    product,
    amount,
    SUM(amount) OVER (
        ORDER BY sale_date, product
        ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING
    ) AS sum_current_and_next
FROM sales
ORDER BY sale_date, product;
```

현재 행(CURRENT ROW)부터 1 이후 행(1 FOLLOWING)까지 (다음 행)

결과

sale_date	product	amount	sum_current_and_next
2024-01-01	김치	12000	27000 (12,000(현재 행) + 15,000(다음 행))
2024-01-02	김치	15000	18000 (15,000(현재 행) + 3,000(다음 행))
2024-01-02	라면	3000	7000 (3,000(현재 행) + 4,000(다음 행))
2024-01-03	라면	4000	4000 (4,000(현재 행) + 없음(다음 행))

■ 순위 함수 ★★

RANK WITHIN GROUP

특정 값에 대한 순위를 확인 (윈도우 함수 아님)

```sql
SELECT RANK(대상) WITHIN GROUP(ORDER BY 컬럼 ASC|DESC);
```

ex) 직원 급여 순위에서 52,000원은 몇 등일까?

```sql
SELECT
    RANK(52000) WITHIN GROUP (ORDER BY salary DESC) as salary_rank
FROM employee_salaries;
```

결과

salary_rank
3

RANK OVER

윈도우 함수 (RANK 안에 대상 설정 필요하지 않음)
전체 행 기준, 특정 그룹 내 기준으로 모두 순위 확인 가능
특정 그룹 내 순위를 알고 싶다면 PARTITION BY 절 사용
ORDER BY 절이 필수 (여러 컬럼 나열 가능)

```sql
SELECT RANK() OVER([PARTITION BY 컬럼] ORDER BY 컬럼 ASC|DESC);
```

ex) 각 직원의 급여 전체 순위 (급여가 높은 순)

```sql
SELECT
    name,
    salary,
    RANK() OVER (ORDER BY salary DESC) as rank
FROM
    employee_salaries
ORDER BY
    rank;
```

결과

name	salary	rank
David	60000	1
Bob	60000	1
Eve	55000	3

값이 동일하면 동일한 순위 부여
엄격한 순위 매김 3등부터 시작

ex) 각 직원 부서별 급여 전체 순위 (부서 안에서 순위)

```sql
SELECT
    name,
    dept,
    salary,
    RANK() OVER (PARTITION BY dept ORDER BY salary DESC) as rank
FROM
    employee_salaries
ORDER BY
    dept, rank;
```

결과

name	dept	salary	rank
David	IT부	60000	1
Bob	IT부	60000	1
Grace	IT부	55000	3
Henry	영업부	51000	1
Frank	영업부	48000	2

값이 동일하면 동일한 순위 부여
2명의 1등 다음 3등부터 시작

DENSE_RANK

동일한 순위 부여 다음 순위가 이어지는 순위로 부여
ex) 부서별 급여 순위

```
SELECT
    name,
    dept,
    salary,
    DENSE_RANK() OVER (PARTITION BY dept ORDER BY salary DESC) as rank
FROM
    employee_salaries
ORDER BY
    dept, rank, name;
```

결과

name	dept	salary	rank
Bob	IT부	60000	1
David	IT부	60000	1
Grace	IT부	55000	2
Frank	영업부	51000	1
Henry	영업부	51000	1

1등 다음 2등

ROW_NUMBER

행 번호
동일한 순위 인정 X, 순서대로 나열했을 때의 순서 값 반환
ex) 부서별 급여 순의 행 번호

```
SELECT
    name,
    dept,
    salary,
    ROW_NUMBER() OVER (PARTITION BY dept ORDER BY salary DESC, name) as row_num
FROM
    employee_salaries
ORDER BY
    dept, row_num;
```

결과

name	dept	salary	rank
Bob	IT부	60000	1
David	IT부	60000	2
Grace	IT부	55000	3
Frank	영업부	51000	1
Henry	영업부	51000	2

1번째, 2번째, 3번째…

■ LAG, LEAD ★★

- ORDER BY 절 필수

LAG

N행의 이전 값을 가져옴 (N행 생략 시, 기본 설정 1)

```
SELECT
    LAG(컬럼, [N행]) OVER ([PARTITION BY 컬럼] ORDER BY 컬럼)
FROM 테이블;
```

ex) 이전 월 판매액, 현재 월 판매액에서 이전 월 판매액을 뺀 값 출력

```
SELECT
    month,
    sales,
    LAG(sales) OVER (ORDER BY month) as previous,
    sales - LAG(sales) OVER (ORDER BY month) as diff
FROM monthly_sales
ORDER BY month;
```

결과

month	sales	previous	diff
2024-01	50000	NULL	NULL
2024-02	52000	50000	2000
2024-03	48000	52000	-4000

LEAD

N행 이후의 값을 가져옴 (N행 생략 시, 기본 설정 1)

```
SELECT
    LEAD(컬럼, [N행]) OVER ([PARTITION BY 컬럼] ORDER BY 컬럼)
FROM 테이블;
```

LEAD

ex) 다음 월 판매액, 현재 월 판매액에서 다음 월 판매액을 뺀 값 출력

```
SELECT
    month,
    sales,
    LEAD(sales, 1) OVER (ORDER BY month) as next,
    LEAD(sales, 1) OVER (ORDER BY month) - sales as diff
FROM monthly_sales;
```

결과

month	sales	next	diff
2024-04	55000	53000	-2000
2024-05	53000	58000	5000
2024-06	58000	NULL	NULL

■ FIRST_VALUE, LAST_VALUE ★★

FIRST_VALUE

정해진 범위에서 정렬 순으로 봤을 때 첫 번째 값 반환

```
SELECT FIRST_VALUE(대상) OVER(
        [PARTITION BY 컬럼]
        [ORDER BY 컬럼]
        [RANGE|ROWS BETWEEN A AND B
    )
```

-- 컬럼 값이 동일한 행들끼리
-- 위에서 정한 범위 안에서 정렬
-- 어느 행부터 어느행까지

ex) 같은 부서의 가장 급여가 높은 사람과 가장 낮은 사람 출력

```
SELECT
    name,
    dept,
    sal,
    FIRST_VALUE(sal) OVER (PARTITION BY dept ORDER BY sal ASC) as low,
    FIRST_VALUE(sal) OVER (PARTITION BY dept ORDER BY sal DESC) as high
FROM employee_salaries;
```

결과

name	dept	sal	low	high
Charlie	HR	65000	65000	71000
Eve	HR	71000	65000	71000
Alice	IT	75000	75000	82000
David	IT	78000	75000	82000
Bob	IT	82000	75000	82000

LAST_VALUE

정해진 범위에서 정렬 순으로 봤을 때 마지막 값 반환

```
SELECT
    name,
    dept,
    sal,
    LAST_VALUE(sal) OVER (
        PARTITION BY dept
        ORDER BY sal
    ) as sal1,
    LAST_VALUE(sal) OVER (
        PARTITION BY dept
        ORDER BY sal ASC
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) as sal_a,
    LAST_VALUE(sal) OVER (
        PARTITION BY dept
        ORDER BY sal DESC
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) as sal_d
FROM employee_salaries
ORDER BY dept, sal;
```

★ RANGE 를 넣지 않으면 기본 설정이 첫번째 행부터 지금 행 까지 범위에서 마지막 값 (즉, 현재 행을 출력)

처음부터 마지막 행까지 범위 지정 및 salary로 오름차순 정렬의 마지막 값

처음부터 마지막 행까지 범위 지정 및 salary로 내림차순 정렬의 마지막 값

결과 FIRST_VALUE 와 비슷하지만 큰 차이가 있기에 이점을 노려서 출제 하기에 주의!!!

name	dept	sal	sal1	sal_a	sal_d
Charlie	HR	6500	6500	7100	6500
Eve	HR	7100	7100	7100	6500
Alice	IT	7500	7500	8200	7500
David	IT	7800	7800	8200	7500
Bob	IT	8200	8200	8200	7500

■ NTITLE

- N개의 그룹으로 나눠준 뒤, 그룹 번호를 붙임 (ORDER BY 필수)
- 정확하게 나눠지지 않는 경우, 앞 그룹이 큰 수를 가져감 (14행을 3으로 나눌 때, 5, 5, 4 로 나눔)

```
SELECT NTILE(N) OVER ([PARTITION BY 컬럼] ORDER BY 컬럼 ASC|DESC)
FROM 테이블;
```

```
SELECT
    student_name,
    score,
    NTILE(4) OVER (ORDER BY score DESC) as nt
FROM
    student_scores
ORDER BY
    score DESC;
```

결과

student_name	score	quartile
David	95	1
Bob	92	1
Jack	91	2
Henry	89	3
Eve	88	4

■ 비율 함수

RATIO_TO_REPORT	범위 내 합계에서 각 값의 비율 (ORDER BY 사용 불가) `SELECT RATIO_TO_REPORT(대상) OVER ([PARTITION BY 컬럼]) FROM 테이블`
CUME_DIST	나보다 같거나 작은 값이 전체의 몇 %인가? 쉽게 말해, "내가 이 그룹에서 상위 몇 % 안에 들어가는가?" ex) 시험 점수에서 CUME_DIST가 0.8이라면, "내 점수가 하위 80% 범위 안에 든다" 또는 "나는 상위 20% 안에 든다"로 해석 값의 범위는 0보다 크고 1 이하 ORDER BY 필수 `SELECT CUME_DIST() OVER ([PARTITION BY 컬럼] ORDER BY 컬럼) FROM 테이블`
PERCENT_RANK	순위에 기반한 "내가 이 그룹에서 몇 번째인가?" 쉽게 말해, "내 순위가 전체의 몇 % 지점에 있는가?" ex) PERCENT_RANK가 0.7이라면, "내 순위가 하위 30% 지점에 있다" 또는 "나는 상위 70% 지점에 있다"로 해석 값의 범위는 0 이상 1 이하 ORDER BY 필수 `SELECT PERCENT_RANK() OVER ([PARTITION BY 컬럼] ORDER BY 컬럼) FROM 테이블`

```
SELECT
    name,
    dept,
    sal,
    RATIO_TO_REPORT(sal) OVER (PARTITION BY dept) as sal_ratio,
    CUME_DIST() OVER (PARTITION BY dept ORDER BY sal) as cumulative_dist,
    PERCENT_RANK() OVER (PARTITION BY dept ORDER BY sal) as percent_rank
FROM
    employee_sal
ORDER BY
    dept, sal;
```

전체 대비 개별 항목의 비중
65000/(65000+71000)

65000 이하 개수 / 전체 개수
(65000) / (65000, 71000) = 1/2

(순위-1) / (전체 행-1)
(2-1) / (3-1) = 1/2

결과

name	dept	sal	sal_ratio	cumulative_dist	percent_rank
Charlie	HR	65000	0.4779	0.5000	0.0000
Eve	HR	71000	0.5221	1.0000	1.0000
Alice	IT	75000	0.3191	0.3333	0.0000
David	IT	78000	0.3319	0.6667	0.5000
Bob	IT	82000	0.3489	1.0000	1.0000
Frank	Sales	68000	0.4823	0.5000	0.0000
Grace	Sales	73000	0.5177	1.0000	1.0000

TOP N 쿼리

■ TOP N 쿼리

- 특정 조건에 맞는 상위 N개의 결과만 출력 (ex. 급여 상위 3명 등)

■ ROWNUM

- 쿼리 결과의 각 행에 자동으로 할당되는 번호 (1부터 시작하여 1씩 증가)
- 절대적인 번호가 아닌, 가상으로 부여되는 번호이기에 특정 행을 지정하는 것은 불가 (= 연산 사용 불가)
  - ex) ROWNUM = 2 를 조건에 넣으면 첫 번째 행이 검색 후 두 번째 결과는 다시 ROWNUM이 1이 되기 때문에 불가
- > 연산자(크다) 또한 위와 같은 사유로 사용 불가 (> 0 은 사용 가능)
- < 연산자(작다)는 ROWNUM 1을 포함해서 가능 (아래 예시 참조)
- 결론적으로, 조건에 ROWNUM 1이 포함되야 가능 (1 부터 순서대로 뽑을 수 있어야 하기 때문)
- 당연하지만, 정렬 순서에 따라서도 ROWNUM이 변경됨

직원 급여 테이블

emp_name	salary
Alice	50000
Bob	55000
Charlie	48000
David	60000
Eve	52000

```
SELECT ROWNUM, emp_name, salary
FROM employees
WHERE ROWNUM <= 3;
```

상위 행 3개 추출

결과

ROWNUM	emp_name	salary
1	Alice	50000
2	Bob	55000
3	Charlie	48000

급여가 가장 높은 3명의 직원 추출

```
SELECT ROWNUM, emp_name, salary
FROM (
    SELECT emp_name, salary
    FROM employees
    ORDER BY salary DESC
)
WHERE ROWNUM <= 3;
```

결과

ROWNUM	emp_name	salary
1	David	60000
2	Bob	55000
3	Eve	52000

■ ROWNUM 잘못된 사용

- > 연산자 사용 (크다 조건 불가)

```
SELECT ROWNUM, emp_name, salary
FROM employees
WHERE ROWNUM > 1;
```

- = 연산자 사용 불가

```
SELECT ROWNUM, emp_name, salary
FROM employees
WHERE ROWNUM = 3;
```

- 상위 급여 3명 직원 추출 오류

WHERE 절의 ROWNUM 조건은 ORDER BY 절보다 먼저 실행됨.
따라서 이 쿼리는 테이블에서 처음 3개의 행을 무작위로 선택한 후, 그 3개의 행만 급여 순으로 정렬되기 때문에 오류

```
SELECT emp_name, salary
FROM employees
WHERE ROWNUM <= 3
ORDER BY salary DESC;
```

- 해결 쿼리

먼저 모든 직원을 급여 순으로 정렬한 후, 상위 3명을 선택하면 됨

```
SELECT emp_name, salary
FROM (
    SELECT emp_name, salary
    FROM employees
    ORDER BY salary DESC
)
WHERE ROWNUM <= 3;
```

- 4~6번째 급여 순위 직원 출력 (ROWNUM)

서브 쿼리를 통해 정렬된 결과를 테이블처럼 전달
ROWNUM 을 부여 후 새로운 테이블 인 것처럼 전달
번호가 4 이상인 유저만 추출

```
SELECT emp_id, emp_name, salary
FROM (
    SELECT emp_id, emp_name, salary, ROWNUM as rn
    FROM (
        SELECT emp_id, emp_name, salary
        FROM employees
        ORDER BY salary DESC
    )
    WHERE ROWNUM <= 6
)
WHERE rn >= 4;
```

- 4~6번째 급여 순위 직원 출력 (RANK)

서브 쿼리에서 급여 순으로 RANK를 할당 한 뒤,
바깥쪽 쿼리에서 RANK가 4에서 6 사이인 행을 선택

```
SELECT emp_id, emp_name, salary
FROM (
    SELECT
        emp_id,
        emp_name,
        salary,
        RANK() OVER (ORDER BY salary DESC) as rnk
    FROM employees
)
WHERE rnk BETWEEN 4 AND 6;
```

■ FETCH 절

- 특정 위치에서부터 행을 출력
- Oracle 12c 이상 버전에서 사용되는 기능
  (ROWRUM보다 좀더 직관적이고 표준화됨. 이전 버전은 ROWNUM 을 사용)
- SQL Server 에서도 사용 가능
- 순서는 ORDER BY 절 뒤에 사용되며, 내부 파싱 순서도 ORDER BY 뒤임

```
SELECT column1, column2
FROM table_name
ORDER BY column3
OFFSET N { ROW | ROWS }N행 건너뜀 (행의 수가 하나이면 ROW, 복수이면 ROWS인데 크게 구분하지 않아도 됨)
FETCH { FIRST | NEXT } n { ROW | ROWS } ONLY;
```

OFFSET을 사용하지 않으면 FIRST 사용 (처음 행부터 n 행 출력)
OFFSET을 사용했으면, NEXT 사용 (N행 제외한 뒤부터 n행 출력)

- 급여 순위 상위 3명 출력

```
SELECT emp_id, emp_name, salary
FROM employees
ORDER BY salary DESC
FETCH FIRST 3 ROWS ONLY;
```

- 급여 순위 4등에서 6등만 출력

```
SELECT emp_id, emp_name, salary
FROM employees
ORDER BY salary DESC
OFFSET 3 ROWS FETCH NEXT 3 ROWS ONLY;
```

■ TOP N 쿼리

- SQL Server에서 상위 N개 행 출력 시 사용
- 정렬 된 순서에서 상위 N개 추출
- WITH TIES를 사용하면 동일한 순위도 함께 출력 가능

```
SELECT TOP n column1, column2
FROM table_name
ORDER BY column3 DESC;
```

- 상위 급여 3명 출력

```
SELECT TOP 3 emp_name, salary
FROM employees
ORDER BY salary DESC;
```

emp_name	salary
David	70000
Bob	65000
Grace	62000

- 상위 3명의 고연봉 직원 조회 (동일 급여 포함)

```
SELECT TOP 3 WITH TIES emp_name, salary
FROM employees
ORDER BY salary DESC;
```

emp_name	salary
David	70000
Bob	65000
Eve	65000
Grace	65000

3행만 출력되지 않고,
마지막 행과 동일한 값을 가진
추가 행들도 결과에 포함

계층형 질의

■ 계층형 질의

- 부모-자식 관계나 트리 구조와 같은 계층적 데이터를 쿼리

```
SELECT column1, column2
FROM table_name
START WITH 시작조건 (최상위)
CONNECT BY [NOCYCLE] PRIOR 연결조건 (부모-자식 관계)
```

- START WITH: 계층 구조에서 루트를 지정
  ex) START WITH parent_id IS NULL
- CONNECT BY: 부모-자식 관계를 정의 (PRIOR 키워드를 사용하여 부모 혹은 자식을 참조)
  ex) CONNECT BY PRIOR id = parent_id
- NOCYCLE: 생략 가능하며, 사용 시 순환 참조를 방지
- **계층형 질의 예제 – 사장부터 직원 방향으로 계층형 데이터 출력**

직원 테이블

emp_id	emp_name	manager_id	job_title
100	John Smith	NULL	CEO
201	Alice Johnson	100	CTO
202	Bob Williams	100	CFO
301	Carol Brown	201	Dev Manager
302	David Lee	201	QA Manager
303	Eve Taylor	202	Finance Manager

```
SELECT LEVEL, emp_name, emp_id, manager_id, job_title
FROM employees
START WITH manager_id IS NULL manager_id가 NULL인게 시작 조건 (최상위 루트)
CONNECT BY PRIOR emp_id = manager_id; PRIOR emp_id는 emp_id가 도착하는 곳
```

즉, 방향이 manager_id → emp_id (최상위 관리자 → 직원)

결과

LEVEL	emp_name	emp_id	manager_id	job_title
1	John Smith	100	NULL	CEO
2	Alice Johnson	201	100	CTO
3	Carol Brown	301	201	Dev Manager
3	David Lee	302	201	QA Manager
2	Bob Williams	202	100	CFO
3	Eve Taylor	303	202	Finance Manager

빨간색 박스 (manager_id)에서 시작 (NULL인 것 부터)
PRIOR인 emp_id로 도착 (101)
다시, manager_id에 101을 던져 자식 행들을 찾음

즉, 방향이 manager_id 에서 emp_id로 흘러감

- **계층형 질의 예제 – 특정 직원에서 그 위 매니저까지 계층 데이터를 출력**

```
SELECT LEVEL, emp_name, emp_id, manager_id, job_title
FROM employees
START WITH emp_id = 303  -- Eve Taylor부터 시작
CONNECT BY PRIOR manager_id = emp_id;직원 부터 시작해 매니저로 도착
```

결과

LEVEL	emp_name	emp_id	manager_id	job_title
1	Eve Taylor	303	202	Finance Manager
2	Bob Williams	202	100	CFO
3	John Smith	100	NULL	CEO

- **계층형 질의 예제 – CONNECT BY vs WHERE**

```
SELECT LEVEL, emp_name, emp_id, manager_id, job_title
FROM employees
START WITH manager_id IS NULL
CONNECT BY PRIOR emp_id = manager_id AND job_title != 'CFO'
```

결과

LEVEL	emp_name	emp_id	manager_id	job_title
1	John Smith	100	NULL	CEO
2	Alice Johnson	201	100	CTO
3	Carol Brown	301	201	Dev Manager
3	David Lee	302	201	QA Manager

계층 구조를 생성하는 과정에 적용되기 때문에 "가지치기 " 를 수행
즉, 조건에 맞지 않는 CFO와 그 하위 직원들은 출력하지 않음

```
SELECT LEVEL, emp_name, emp_id, manager_id, job_title
FROM employees
START WITH manager_id IS NULL
CONNECT BY PRIOR emp_id = manager_id
WHERE job_title != 'CFO';
```

결과

LEVEL	emp_name	emp_id	manager_id	job_title
1	John Smith	100	NULL	CEO
2	Alice Johnson	201	100	CTO
3	Carol Brown	301	201	Dev Manager
3	David Lee	302	201	QA Manager
3	Eve Taylor	303	202	Finance Manager

계층 구조는 유지되어 생성된 이후에, 결과에서 Bob Williams(CFO)만 제외되었고,
그의 부하 직원인 Eve Taylor는 여전히 결과에 포함됨

■ 계층형 질의 가상 컬럼

- LEVEL: 루트로부터 DEPTH를 나타냄 (루트는 1)
- CONNECT_BY_ISLEAF: 최하위 노트 여부를 반환 1(참) or 0(거짓)
- CONNECT_BY_ISCYCLE : 계층형 쿼리의 결과에서 순환이 발생했는지 여부

```
SELECT LEVEL, CONNECT_BY_ISLEAF AS is_leaf, emp_id, manager_id, job_title
FROM employees
START WITH manager_id IS NULL
CONNECT BY PRIOR emp_id = manager_id;
```

LEVEL	is_leaf	emp_id	manager_id	job_title
1	0	100	NULL	CEO
2	0	201	100	CTO
3	1	301	201	Dev Manager
3	1	302	201	QA Manager
2	0	202	100	CFO
3	1	303	202	Finance Manager

■ 계층형 질의 가상 함수

- CONNECT_BY_ROOT 컬럼명 : 루트 노드 컬럼 값
- SYS_CONNECT_BY_PATH(컬럼, 구분자) : 이어지는 경로 출력
- ORDER SIBLINGS BY 컬럼 : 같은 LEVEL일 경우 정렬 수행

```
SELECT
    emp_name,
    CONNECT_BY_ROOT emp_name AS root_employee,
    SYS_CONNECT_BY_PATH(emp_name, ' -> ') AS path,
FROM employees
START WITH manager_id IS NULL
CONNECT BY PRIOR emp_id = manager_id
ORDER SIBLINGS BY job_title;
```

결과

emp_name	root_employee	path
John Smith (루트)	John Smith	John Smith
Bob Williams	John Smith	John Smith -> Bob Williams
Eve Taylor	John Smith	John Smith -> Bob Williams -> Eve Taylor
Alice Johnson	John Smith	John Smith -> Alice Johnson
Carol Brown	John Smith	John Smith -> Alice Johnson -> Carol Brown
David Lee	John Smith	John Smith -> Alice Johnson -> David Lee

LEVEL이 3으로 같은 경우, job_title로 정렬

■ NOCYCLE

- 계층 구조에서 순환 참조를 할 때, 기본적으로 그냥 호출하면 오류 발생함

사이클 구조 테이블

id	name	manager_id
1	Alice	2
2	Bob	1

```
SELECT id, name, LEVEL
FROM emp_cycle
START WITH id = 1
CONNECT BY PRIOR id = manager_id;
```

오류 → 사용자 데이터의 CONNECT BY 루프

```
SELECT id, name, LEVEL
FROM emp_cycle
START WITH id = 1
CONNECT BY NOCYCLE PRIOR id = manager_id;
```

정상 출력

결과

ID	NAME	LEVEL
1	Alice	1
2	Bob	2

PIVOT 과 UNPIVOT

■ PIVOT

- 행 데이터를 열 데이터로 변환하는 기능
- 데이터를 더 읽기 쉽고 분석하기 좋은 형태로 재구성
- LONG DATA를 WIDE DATA로 변경

■ UNPIVOT

- 열 데이터를 행 데이터로 변환하는 기능
- 데이터를 더 정규화된 형태로 변환
- WIDE DATA를 LONG DATA 로 변경

■ LONG DATA (Tidy Data)

- 하나의 속성이 하나의 컬럼으로 정의되어 값들이 아래로 여러 행 쌓임
- RDBMS의 테이블 구조, 다른 테이블과 조인 연산이 가능한 구조를 뜻함

LONG DATA 예시

학생ID	과목	점수
1	수학	90
1	과학	85
1	영어	88
2	수학	78
2	과학	92
2	영어	95

아래로 여러 행이 LONG (길게 나열)

UNPIVOT ↔ PIVOT

■ WIDE DATA

- 여러 특성이나 변수가 각각의 열로 표현
- 새로운 변수가 추가될 때마다 새로운 열이 필요
- 행의 수가 상대적으로 적고, 열의 수가 많음 (데이터 요약이 주된 목적)

WIDE DATA 예시

학생ID	수학	과학	영어
1	90	85	88
2	78	92	95

개체가 넓게 나열

■ PIVOT 문법

- 교차표를 만드는 기능으로, STACK, UNSTACK, VALUE 컬럼만 정의

학생ID	수학	과학	영어
1	90	85	88
2	78	92	95

UNSTACK / VALUE / STACK

- FROM 절에 STACK, UNSTACK, VALUE 컬럼 정의 **(꼭 3 종류 컬럼 전부 명시)**
- PIVOT 절에 UNSTACK, VALUE를 정의 (FROM – PIVOT 컬럼이 STACK 됨)

```
SELECT *
FROM 테이블 | 서브 쿼리
PIVOT (VALUE의 컬럼명 FOR UNSTACK의 컬럼명 IN (값1, 값2, 값3));
```

- **PIVOT 예제 – 각 판매원의 제품별 판매 금액**

```
SELECT *
FROM (
    SELECT salesman, product, amount FROM sales
)
PIVOT (
    SUM(amount) FOR product IN ('Laptop', 'Desktop', 'Tablet')
);
```

결과

SALESMAN	'Laptop'	'Desktop'	'Tablet'
John	500	600	NULL
Jane	550	650	150
Bob	525	NULL	175

UNSTACK / VALUE / STACK

- **PIVOT 예제 – 제품별 판매 금액 (STACK이 누락된 경우)**

```
SELECT *
FROM (
    SELECT product, amount FROM sales
)
PIVOT (
    SUM(amount) FOR product IN ('Laptop', 'Desktop', 'Tablet')
);
```

결과

'Laptop'	'Desktop'	'Tablet'
1575	1250	325

- **PIVOT 예제 - 제품별 판매 금액 (서브 쿼리가 아닌 테이블 전체 컬럼을 넣은 경우)**

```
SELECT *
FROM sales
PIVOT (
    SUM(amount)
    FOR product IN ('Laptop', 'Desktop', 'Tablet')
);
```

필요한 컬럼만 정의하지 않으면 모든 STACK으로 UNSTACK, VALUE를 제외한 모든 컬럼이 STACK 설정됨

결과

SALESMAN	SALE_DATE	'Laptop'	'Desktop'	'Tablet'
John	2023-01-01	500	NULL	NULL
John	2023-01-02	NULL	600	NULL
Jane	2023-01-01	550	NULL	NULL
Jane	2023-01-02	NULL	NULL	150
Jane	2023-01-03	NULL	650	NULL
Bob	2023-01-02	NULL	NULL	175
Bob	2023-01-03	525	NULL	NULL

- **PIVOT 예제 – PIVOT을 통한 교차표를 보고 를 보고 쿼리를 작성 (fruit_data table)**

LONG DATA

FRUIT	COLOR	QUANTITY
Apple	Red	5
Apple	Green	3
Banana	Yellow	4
Grape	Purple	2
Grape	Green	1

WEIRD DATA

FRUIT	'Red'	'Green'	'Yellow'	'Purple'
Apple	5	3	NULL	NULL
Banana	NULL	NULL	4	NULL
Grape	NULL	1	NULL	2

정답

```
SELECT *
FROM (
    SELECT FRUIT, COLOR, QUANTITY
    FROM fruit_data
)
PIVOT (
    SUM(QUANTITY)
    FOR COLOR IN ('Red', 'Green', 'Yellow', 'Purple')
)
ORDER BY FRUIT;
```

★ 대괄호 하나당 하나의 문자를 표현

■ UNPIVOT 문법

- WIDE DATA를 LONG DATA로 변경하는 문법
- **기존 UNSTACK 된 컬럼을 STACK으로 나열 (이때, 컬럼명을 만들어줘야 함)**
- **기존 VALUE들을 STACK으로 나열 (이때, 컬럼명을 만들어줘야 함)**

```
SELECT *
FROM 테이블명 | 서브 쿼리
UNPIVOT (VALUE의 컬럼명 FOR UNSTACK의 컬럼명 IN (값1, 값2);
```

- **값1, 값2 는 UNSTACK 되어 있는 컬럼 이름들을 나열**
- **만약 기존 UNSTACK 되어 있는 컬럼 이름들이 숫자면 “”를 붙이고, 아니면 붙이지 않아도 됨**

- **UNPIVOT 예제 - 각 월 판매액을 UNPVIOT하는 쿼리를 작성**

WEIRD DATA

PRODUCT	JANUARY	FEBRUARY	MARCH
Laptop	1000	1200	1100
Phone	800	750	900

정답

```
SELECT *
FROM product_sales
UNPIVOT (
    sales_amount
    FOR sales_month IN (JANUARY,FEBRUARY,MARCH)
);
```

컬럼명을 지어줘야 함

컬럼명을 지어줘야 함

쌍 따옴표를 붙일 필요 없음 (만약 컬럼명이 숫자라면 전달 필요)

결과

PRODUCT	SALES_MONTH	SALES_AMOUNT
Laptop	January	1000
Laptop	February	1200
Laptop	March	1100
Phone	January	800
Phone	February	750
Phone	March	900

정규 표현식

- 문자열의 특정 패턴을 찾아 매칭하거나 변환하는 문법

ex) 컬럼 값이 숫자로 시작하는 4자리 행 조회

```
SELECT *
  FROM [테이블명]
 WHERE REGEXP_LIKE([컬럼명], '^[0-9]{4}$');
```

■ 정규 표현식 종류 ₩ 표시는 역슬래시 기호로 □와 동일

₩d	숫자 (0,1,2,3…9)	₩D	숫자가 아닌 것
₩s	공백	₩S	공백이 아닌 것
₩w	단어 (알파벳, 숫자, 언더스코어)	₩W	단어 아닌 것 (공백, 특수문자 등)
₩t	Tab	₩n	엔터 문자 (New Line)
^	시작 글자	$	마지막 글자
[ab]	a 또는 b 한 글자 (c[ab]t: cat, cbt만 매칭)	[^ab]	a, b 제외한 한 글자 ([^ab]t: cat, cbt 빼고 가능)
[0-9]	숫자	[A-z]	모든 영문자
[A-Z]	영어 대문자	[a-z]	영어 소문자
a+	a가 1회 이상 반복 (b 는 거짓, ab 는 참)	a*	a가 0회 이상 반복 (a, b, aa 전부 참)
a?	a가 0회 혹은 1회 반복 (a?b는 b, ab 매칭)	a{n}	a가 n회 반복 (a{3}b는 aaab 매칭)
a{n, m}	a가 n~m회 반복 (a{1,2}는 a, aa에는 매칭)	a{n,}	a가 n회 이상 (a{2,}는 aa 이상은 전부 매칭)
[:alpha:]	알파벳 문자	[:digit:]	숫자(0~9)
[:alnum:]	알파벳 문자([:alpha:]) + 숫자([:digit:])		
[:graph:]	눈에 보이는(그래픽) 문자 - 즉, 공백을 제외한 모든 출력 가능한 문자		
[:blank:]	공백 문자 (스페이스( ), 탭(₩t))	[:cntrl:]	제어 문자(ASCII 0~31, 127 등)
[:lower:]	소문자(a~z)	[:upper:]	대문자(A~Z)
[:xdigit:]	16진수 문자	[:punct:]	구두점(문장 부호)이나 특수문자
[:space:]	공백 문자 (탭, 줄바꿈(₩n), 폼 피드(₩f) 등을 포괄하는 “모든” 공백 문자에 해당)	[:print:]	출력 가능한(프린트 가능한) 모든 문자(공백 포함) ([:graph:] + 공백(space))
|	또는	.	한 글자 (엔터를 제외한)
()	그룹 지정	₩n	그룹번호 (재참조)
₩	뒤에 붙는 기호의 효력을 제거하여 기호 그 자체로 쓰일 수 있게 함		

- **예제 – 데이터를 통해 정규표현식 일반화 과정**

tel)02-1234-5678 의 경우, tel₩)[0-9]{2}-[0-9]{4}-[0-9]{4} 로 표현 가능
조금 더 단순화하면, tel₩)[0-9-]+ 로 표현 가능

tel02-1234-5678 의 경우, tel[0-9]{2}-[0-9]{4}-[0-9]{4} 로 표현 가능
조금 더 단순화하면, tel[0-9-]+ 로 표현 가능

위 두 데이터를 포함하는 정규표현식은 tel₩)?[0-9]{2}-[0-9]{4}-[0-9]{4} 로 표현
₩)?를 통해 0회 혹은 1회 가능함을 표현

■ REGEXP_REPLACE

- 정규식 표현을 사용하여, 문자열을 치환해 노출
- 문법
  - (컬럼, 찾을문자열, [바꿀문자열], [검색위치], [발견횟수], [옵션])
  - 바꿀문자열 생략 시, 문자열을 삭제
  - 검색위치 생략 시, 기본값 1
  - 발견횟수 생략 시, 기본값 0 (모든)
  - 옵션
    - c: 대문자, 소문자를 구분하여 검색
    - i: 대문자, 소문자를 구분하지 않고 검색
    - m: 패턴을 다중라인으로 선언 가능

- **예제 – 컬럼에서 숫자 삭제**

```
SELECT
    VAL, 원래 데이터
    REGEXP_REPLACE(VAL, '[0-9]', '') AS VAL2,
    REGEXP_REPLACE(VAL, '[[:digit:]]', '') AS VAL3,
    REGEXP_REPLACE(VAL, '₩d', '') AS VAL4
FROM TBL_EXAMPLE;
```

여러 정규표현식을 통한 숫자 제거

결과

VAL	VAL2	VAL3	VAL4
ABC123	ABC	ABC	ABC
Hello 456World	Hello World	Hello World	Hello World
Phone:010-1234-5678	Phone:--	Phone:--	Phone:--

빈문자열을 전달하여 숫자를 모두 삭제 처리

- **예제 – 컬럼에서 특수문자 삭제**

```
SELECT
    COL, 원래 데이터
    REGEXP_REPLACE(COL, '[^[:alnum:]가-힣]', '') AS COL2,
    REGEXP_REPLACE(COL, '[^0-9a-zA-Z가-힣]', '') AS COL3,
    REGEXP_REPLACE(COL, '₩₩', '') AS COL4
FROM TB_TEST;
```

알파벳, 숫자, 한글가 아닌 문자 제거

결과

COL	COL2	COL3	COL4
Hello@World!!!	HelloWorld	HelloWorld	HelloWorld
가나다!@#$123	가나다123	가나다123	가나다123
홍길?동##	홍길동	홍길동	홍길동

- **예제 – 컬럼에서 문자 바로 뒤에 오는 숫자 제거** 대괄호 하나당 하나의 문자를 뜻함

```
SELECT
    COL_VAL AS COL,                 문자와 연속오는 문자가 일치할 때 제거
    REGEXP_REPLACE(COL_VAL, '[A-Z][0-9]', '') AS COL2, 대문자 숫자 제거
    REGEXP_REPLACE(COL_VAL, '[A-Za-z][0-9]', '') AS COL3, 대소문자 숫자 제거
    REGEXP_REPLACE(COL_VAL, '[A-Z][0-9]', '', 1, 0, 'i') AS COL4, i 옵션으로 대소문자 숫자 제거
    REGEXP_REPLACE(COL_VAL, '[[:alpha:]][[:digit:]]', '') AS COL5, 대소문자 숫자 제거
    REGEXP_REPLACE(COL_VAL, '[[:alpha:]][[:digit:]]', '', 1, 0, 'i') AS COL6,
    REGEXP_REPLACE(COL_VAL, '[A-z0-9]', '') AS COL7 그냥 한 단어(대소문자숫자)에 해당하면 제거
FROM TBL_TEST;
```

결과

COL	COL2	COL3	COL4
a1B2c3	a1c3	-	-
Abc1Test2	Abc1Test2	AbTes	AbTes
HELL01world2	HELLworld2	HELLworl	HELLworl

COL5	COL6	COL7
-	-	-
AbTes	AbTes	-
HELLworl	HELLworl	-

주의

- **예제 – 문자 하이픈 숫자 구조일 때 제거 (ex. a-1)**

```
SELECT
    DATA,
    REGEXP_REPLACE(DATA, '[a-z]-[0-9]', '')     AS DATA2,
    REGEXP_REPLACE(DATA, '[a-z][-_][0-9]', '')  AS DATA3,
    REGEXP_REPLACE(DATA, '[a-z](-|_)[0-9]', '') AS DATA4
FROM TBL_SAMPLE;
```

소문자 - 숫자 형식 제거
소문자 -또는 _ 숫자 형식 제거
소문자 -또는 _ 숫자 형식 제거

결과

DATA	DATA2	DATA3	DATA4
abc-2	ab	ab	ab
cba_1	cba_1	cb	cb
1-abc	1-abc	1-abc	1-abc

- **예제 – 괄호와 괄호 안 내용 제거**

```
SELECT                              . (엔터를 제외한 모든 문자)
    DATA,                           * (0회 이상 반복)
    REGEXP_REPLACE(DATA, '₩(.*₩)', '') AS DATA2
FROM TBL_SAMPLE;
```

괄호는 표현식 기호이기에 백슬래시 필요 ₩(₩)

결과

DATA	DATA2
아이폰케이스(블랙)	아이폰케이스
삼성노트북(16GB/512GB)	삼성노트북
LG사운드바(우퍼포함)	LG사운드바

- **예제 – 이름 마스킹**

```
SELECT
    COL_TEXT AS DATA,       첫번째부터 검색해서 2번째 발견 문자 치환
    REGEXP_REPLACE(COL_TEXT, '₩w', '', 1, 2) AS DATA2,
    REGEXP_REPLACE(COL_TEXT, '₩w', '*', 1, 2) AS DATA3
FROM TBL_SAMPLE;
```

문자 제거
* 로 치환

결과

DATA	DATA2	DATA3
김민수	김수	김*수
박영희	박희	박*희
홍길동	홍동	홍*동

■ REGEXP_SUBSTR

- 정규표현식을 사용하여 문자열을 추출
- 문법
  - (대상, 패턴, [검색위치], [발견횟수], [옵션], [추출그룹])
  - 검색위치 생략 시, 기본값 1
  - 발견횟수 생략 시, 기본값 0 (모든)
  - 옵션
    - c: 대문자, 소문자를 구분하여 검색
    - i: 대문자, 소문자를 구분하지 않고 검색
    - m: 패턴을 다중라인으로 선언 가능
  - 추출그룹은 서브패턴을 추출 시, 추출할 서브패턴 번호 입력

```
SELECT
    FULL_NAME,
    REGEXP_SUBSTR(
        FULL_NAME, 대상 컬럼
        '([가-힣])([가-힣])([가-힣])', 패턴
        1, 검색 위치 (1은 처음부터 검색)
        1, 몇 번째 발견 된 것을 추출할 지
        null, 옵션
        1 지정된 그룹들 중 어느 그룹을 추출할 지
    ) AS SURNAME
FROM TEST_NAME;
```

파라미터 생략 기능이 없어서 추출그룹 지정해 주고 싶을 때, 옵션은 필요 없어도 null이라도 넣어야 함.

결과

FULL_NAME	SURNAME
김철수	김
이영희	이
박영수	박

■ REGEXP_INSTR

- 주어진 문자열에서 특정패턴의 시작 위치를 반환
- 문법
  - (원본, 찾을문자열, [시작위치], [발견횟수])
  - 시작위치 생략 시, 처음부터 확인 기본값 1
  - 발견횟수 생략 시, 처음 발견된 문자열 위치 리턴

```
SELECT
    USER_EMAIL,
    REGEXP_INSTR(USER_EMAIL, '@') AS AT, @가 나오는 첫 위치
    REGEXP_INSTR(USER_EMAIL, '₩.', 1, 1) AS DOT, . 이 나오는 첫 위치
    REGEXP_INSTR(USER_EMAIL, '₩.', 1, 2) AS DOT2 . 이 나오는 두 번째 위치
FROM TBL_USER;
```

결과

USER_EMAIL	AT	DOT	DOT2
kim.yuna@olympic.com	9	4	17
son.h@gmail.com	6	4	12
lee.kangin@la-liga.org	11	4	19

찾을 문자열에 걸리는 게 없다면 0 반환

- **조심해야 하는 예제** ★

```
SELECT
    ADDRESS,
    REGEXP_INSTR(ADDRESS, '₩d', 1, 2) AS WORD,
    REGEXP_INSTR(ADDRESS, '₩d+', 1, 2) AS WORD2,
    REGEXP_INSTR(ADDRESS, '[^ ]+', 1, 2) AS WORD3
FROM TBL_ADDRESS;
```

₩d 는 숫자지만, 뒤에 횟수 지정하지 않으면 한자리 숫자
₩d+는 숫자 한 덩이를 패턴으로 보기에 1600 이후 숫자 없어서 0
공백이 아닌 덩어리 패턴이기에 A인 6이 반환

결과

ADDRESS	WORD	WORD2	WORD3
1600 Amphitheatre Parkway, Mountain View	2	0	6

■ REGEXP_LIKE

- 주어진 문자열에서 특정 패턴을 가지는 경우 반환
- WHERE 절에서만 사용 가능
- 문법
  - (원본, 찾을문자열, [옵션])
  - 옵션
    - c: 대문자, 소문자를 구분하여 검색
    - i: 대문자, 소문자를 구분하지 않고 검색
    - m: 패턴을 다중라인으로 선언 가능

```
SELECT
    MEMBER_NAME,
    MEMBER_PHONE
FROM TBL_MEMBER
WHERE REGEXP_LIKE(
    MEMBER_PHONE,
    '^₩d{2,3}-₩d{3,4}-₩d{4}$'
);
```

원본

MEMBER_NAME	MEMBER_PHONE
Kim Chulsoo	010-1234-5678
Park Younghee	02-888-9999
Lee Hyojin	010-12-3456

결과

MEMBER_NAME	MEMBER_PHONE
Kim Chulsoo	010-1234-5678
Park Younghee	02-888-9999

■ REGEXP_COUNT

- 주어진 문자열에서 특정패턴의 횟수를 반환
- 문법
  - (원본, 찾을문자열, [시작위치], [옵션])
  - 시작위치 생략 시, 기본값 1
  - 옵션
    - c: 대문자, 소문자를 구분하여 검색
    - i: 대문자, 소문자를 구분하지 않고 검색
    - m: 패턴을 다중라인으로 선언 가능

```
SELECT
    MEMBER_PHONE,
    REGEXP_COUNT(MEMBER_PHONE, '₩d') AS RESULT,
    REGEXP_COUNT(MEMBER_PHONE, '₩d+') AS RESULT2
FROM TBL_MEMBER;
```

₩d는 한 자리수의 숫자를 의미하며
₩d+는 연속적인 숫자를 의미

결과

PHONE	R	R2
02-888-9999	9	3
010-1234-5678	11	3
010-12-3456	9	3

DML

■ DML

- 데이터 수정어 (INSERT, UPDATE, DELETE, MERGE)
- 반드시, commit 이나 rollback을 통한 transaction 제어가 필수

■ INSERT

- 테이블에 행을 삽입할 때 사용
- 한 번에 한 행만 입력 가능 SQL Server 의 경우, 여러 행 동시 삽입 가능
- 컬럼별 데이터타입과 사이즈에 맞게 삽입
- INTO 절에 컬럼명을 명시하여 일부 컬럼만 명시 가능 명시 되지 않은 컬럼은 기본값 NULL 단, NOT NULL 필드 시, 오류 발생
- 문법

```
INSERT INTO 테이블명 VALUES(값1, 값2, 값3 ...); 전체 컬럼의 모든 값을 입력 한 경우
INSERT INTO 테이블명(컬럼1, 컬럼2, 컬럼3 ...) VALUES(값1, 값2, 값3 ...);
                                                  선택한 컬럼만 값을 입력한 경우
                                                  (나머지는 NULL 로 입력)
```

- **예제 – 한 행씩 삽입**

테이블 구조

Column	Nullable	Type
STUDENT_ID	-	NUMBER
STUDENT_NAME	-	VARCHAR2(30)

테이블 컬럼의 타입과 실제 값이 같아야 함.

```
-- 1) 가장 기본적인 사용
INSERT INTO STUDENT (STUDENT_ID, STUDENT_NAME) VALUES (1, 'Alice');

-- 2) 컬럼 생략 방법
INSERT INTO STUDENT VALUES (2, 'Bob');

COMMIT;
```

결과

STUDENT_ID	STUDENT_NAME
1	Alice
2	Bob

숫자 타입에 '001' 이나 문자 타입에 100 처럼 입력해도 들어가지만, 성능상 좋지도 않고 권장하지 않음

```
INSERT INTO STUDENT (STUDENT_ID, STUDENT_NAME)
    VALUES ('3', 300);
```

- **예제 – 서브쿼리를 사용한 여러 행 INSERT**

```
INSERT INTO STUDENT (STUDENT_ID, STUDENT_NAME)
SELECT ID, NAME
  FROM STUDENT_TMP
 WHERE ID >= 200;

2개의 행이 삽입되었습니다.
```

- **예제 – 컬럼 명시 생략할 경우, 부분 데이터 삽입 시도 시 오류**

```
INSERT INTO STUDENT VALUES (3);

ORA-00947: 값이 충분하지 않습니다.
```

컬럼 명시 해주면 오류 해결

```
INSERT INTO STUDENT (STUDENT_ID) VALUES (3);

1개 행이 삽입되었습니다.
```

■ UPDATE

- 데이터 수정 시, 사용
- 컬럼 단위 수행 (다중 컬럼 수정 가능)
- 문법
  - 단일 컬럼 수정

```
UPDATE EMPLOYEE
   SET SALARY = 3200
 WHERE ID = 1;
```

수정할 내용 (컬럼 = 수정 값)
수정 대상에 대한 조건

  - 다중 컬럼 수정

```
UPDATE EMPLOYEE
   SET NAME = 'Bobby',
       SALARY   = 3600
 WHERE ID     = 2;
```

수정할 여러 컬럼들을 지정 가능

업데이트 대상과 서브쿼리 테이블을 매핑하면 여러 행도 가능은 함

서브 쿼리를 이용한 다중 컬럼 수정

```
UPDATE EMPLOYEE E
   SET SALARY = (
       SELECT S.ANNUAL_SALARY
         FROM SALARY_TABLE S
        WHERE S.EMPLOYEE_ID = E.EMP_ID
   )
 WHERE E.DEPT_ID = 10;
```

```
UPDATE EMPLOYEE
   SET (EMP_NAME, SALARY) = (
       SELECT NEW_NAME, NEW_SALARY
         FROM EMPLOYEE_NEW_INFO
        WHERE NEW_EMP_ID = 3
   )
 WHERE EMP_ID = 1;
```

SET 절에서 각 행마다 대입할 값이 정확히 하나여야 함

■ DELETE
- 데이터를 삭제할 때 사용 (행 단위로 삭제)

```
DELETE EMPLOYEE;
DELETE FROM EMPLOYEE;
DELETE EMPLOYEE WHERE ID = 1;
```
테이블 전체 row 삭제
FROM 생략 해도 되고 생략하지 않아도 동일
조건을 통해 특정 행만 삭제 가능

■ MERGE
- 데이터 병합
- 대상 테이블과 참조할 테이블을 동일하게 맞추는 작업 (참조 테이블과의 데이터 차이에 따라 INSERT, UPDATE, DELETE가 발생)

```
MERGE INTO STUDENT T
USING STUDENT_TMP S
  ON (T.ID = S.ID)
WHEN MATCHED THEN
  UPDATE
    SET T.NAME = S.NAME
  DELETE WHERE S.STATUS = 'D'
WHEN NOT MATCHED THEN
  INSERT (ID, NAME)
  VALUES (S.ID, S.NAME);
```
대상 테이블
STUDENT_TMP 테이블을 이용해 대상 테이블을 추가, 수정, 삭제 진행
→ 두 테이블 행을 매칭할 조건(ID가 같으면 "WHEN MATCHED")
괄호 필수
매칭되었다면 → 우선 UPDATE
DELETE는 "매칭된 행" 중에서 추가 조건을 만족할 때만 발생
매칭되지 않은 행(= 기존 STUDENT에 없는 ID)은 INSERT

STUDENT

ID	NAME
100	Alice
200	Bob
300	Charlie

STUDENT_TMP

ID	NAME	STATUS
200	Bob Jr.	U
300	Charlie Jr.	U
400	Daisy	U
500	Eve	U
100	Alice	D

수정
삽입
삭제

결과

ID	NAME
200	Bob Jr.
300	Charlie Jr.
400	Daisy
500	Eve

TCL
■ TCL
Transaction Control Language 의 약자로, 트랜젝션 제어어

■ 트랜젝션
- DB에서 트랜잭션은 논리적으로 하나의 작업 단위
DDL 의 경우, 기본적으로 AUTO COMMIT
SQL Server는 AUTO COMMIT 비활성화 가능
ORACLE은 23c 버전부터 비활성화 가능

COMMIT	트랜잭션으로 수행한 작업을 **영구적으로 저장**
ROLLBACK	트랜잭션 수행 내용을 **원래 상태로 되돌림** SAVEPOINT 혹은 최종 COMMIT 시점 이전까지 롤백 가능
SAVEPOINT	트랜잭션 내 **특정 지점을 저장**해두고, 그 지점으로 부분 롤백 가능

- **트랜젝션 특성 4가지**

원자성	트랜잭션의 작업들은 **모두 성공**하거나, **모두 실패**(롤백)해야 한다. ex) A계좌에서 B계좌 이체 시, 송금과 입금 중 하나만 성공하면 안 된다. (A 계좌 -10만원과 B계좌 +10만원이 동시에 성공하거나 둘 다 실패 해야함)
일관성	트랜잭션 시작 전과 끝난 후의 **데이터 무결성**이 유지되어야 함. ex) 재고가 10개에서 9개로 바뀌면, 실제 판매 기록도 1개 증가해야 한다.
고립성	동시에 실행되는 트랜잭션들이 서로의 작업에 **직접적으로 간섭**하지 않아야 함. ex) 두 사람이 동시에 같은 티켓을 사도, 결국 중복 예매는 없어야 한다.
영속성	트랜잭션이 정상적으로 끝난(COMMIT) 이후의 결과는 **영구적으로 보존**되어야 함. ex) 전기가 나가도, 이미 결제 완료된 주문 정보는 사라지지 않는다.

- **트랜젝션 예제**

```
INSERT INTO STUDENT (STUDENT_ID, STUDENT_NAME) VALUES (1, 'Alice');
INSERT INTO STUDENT (STUDENT_ID, STUDENT_NAME) VALUES (2, 'Bob');
COMMIT;

INSERT INTO STUDENT (STUDENT_ID, STUDENT_NAME) VALUES (3, 'Charlie');
ROLLBACK;

INSERT INTO STUDENT (STUDENT_ID, STUDENT_NAME) VALUES (4, 'Daisy');
SAVEPOINT SP1;

INSERT INTO STUDENT (STUDENT_ID, STUDENT_NAME) VALUES (5, 'Emily');
ROLLBACK TO SP1;
COMMIT;
```
초반에 COMMIT되어 유지
ROLLBACK으로 취소
마지막에 COMMIT되어 유지
Emily는 SP1 지점까지 롤백되어 취소
부분 롤백(ROLLBACK TO SP1)으로 취소

결과

STUDENT_ID	STUDENT_NAME
1	Bob Jr.
2	Bob
4	Daisy

③ 관리 구문

DDL
■ DDL
- **데이터 구조를 정의**하는 언어
- **CREATE** (객체 생성), **ALTER** (객체 변경), **DROP** (객체 삭제), **TRUNCATE** (객체 구조는 그대로 두고 데이터만 삭제)

※ TRUNCATE가 데이터만 날리는 점에서 DML처럼 보일 수 있지만, AUTO COMMIT이라는 점에서 DDL (데이터 정의어) 임

■ CREATE
CREATE 문은 새로운 오브젝트(개체)를 생성할 때 사용
※ [ ] 괄호로 묶은 것은 생략 가능

```
CREATE TABLE [Owner.]테이블명
(
  컬럼명1 데이터타입 [DEFAULT 기본값] [제약조건],
  컬럼명2 데이터타입 [DEFAULT 기본값] [제약조건],
  ...
)
```
Owner 생략 시, 현재 접속한 사용자로 설정
DEFAULT 기본값을 생략하면, 해당 컬럼에 기본값이 설정되지 않음
제약조건(예: NOT NULL, PRIMARY KEY 등)도 상황에 따라 생략 가능

- **데이터타입**

NUMBER	숫자형 SQL Server 의 경우, NUMERIC 으로 표현 ex) NUMBER, NUMBER(10), NUMBER(10, 2) 등 [최대] 총 10 자리, 소수점 자리 제한 2
VARCHAR2(길이)	가변 길이 문자열 SQL Server 의 경우도, VARCHAR 사용 ex) VARCHAR2(50) → 최대 50자 저장 (사이즈 생략 가능. 생략 시, 1)
CHAR(길이)	고정 길이 문자열 ex) CHAR(10) → 항상 10자 공간을 차지
DATE	날짜와 시간을 모두 저장 (오라클은 DATE에도 시분초 포함)
TIMESTAMP	소수점 이하 초 단위까지 더 정밀한 시각 저장

```
INSERT INTO TEST_TABLE(COLUMN_NAME)
VALUES (123456.78);

INSERT INTO TEST_TABLE(COLUMN_NAME)
VALUES (123456789.01);
```
NUMBER (10, 2)의 경우, 최대 10자리 넘지 않았으므로, 가능
최대 10자리가 넘었기 때문에, 저장 불가

- **테이블 복제** ※ 복제 시, NULL 속성도 같이 복제 되지만, 제약조건, INDEX 등은 복제되지 않음

```
CREATE TABLE STUDENT_COPY
AS
SELECT *
  FROM STUDENT;
```
STUDENT_COPY 테이블이 새로 만들어지고, STUDENT의 데이터까지 그대로 들어갑니다.

```
CREATE TABLE STUDENT_EMPTY
AS
SELECT *
  FROM STUDENT
 WHERE 1=0;
```
WHERE 절에 항상 거짓인 조건 사용 시, STUDENT_EMPTY 테이블은 컬럼 구조만 동일, 실제 행(데이터)은 0건.

```
CREATE TABLE STUDENT_PART (A, B)
AS
SELECT STUDENT_ID, STUDENT_NAME
  FROM STUDENT
 WHERE STUDENT_ID < 1000;
```
복제 시, 컬럼명 변경 가능
특정 컬럼만 선택해서 새 테이블을 만들 수 있고, WHERE 조건도 걸 수 있어서, 부분 데이터만 복제할 수도 있습니다.

■ ALTER
테이블의 구조를 변경할 때 사용

- **컬럼 추가 (ADD)**

```
ALTER TABLE 테이블명
  ADD ( 새컬럼명 자료형 [DEFAULT 기본값] [제약조건] );
```

```
ALTER TABLE STUDENT
  ADD ( EMAIL VARCHAR2(50) DEFAULT 'NONE' );
```
컬럼 하나만 추가할 경우, 괄호 생략 가능
마지막 컬럼으로 추가

```
ALTER TABLE STUDENT
  ADD (
    PHONE VARCHAR2(20),
    ADDRESS VARCHAR2(100) DEFAULT 'UNKNOWN'
  );
```
여러 컬럼을 한 번에 추가할 때는 괄호 안에 쉼표로 구분
여러 컬럼을 추가할 경우, 괄호 누락 시 에러 발생

```
ALTER TABLE STUDENT ADD ( EMAIL VARCHAR2(50) NOT NULL );
ORA-01758: 필수(NULL이 아님) 열을 추가하려면 테이블이 비어 있어야 합니다.
```
테이블에 **데이터가 존재**하는 경우, NOT NULL 설정하면 에러 발생
기존에 존재하던 데이터에 추가되는 컬럼은 NULL로 설정되기 때문

※ 순서 주의(NOT NULL은 DEFAULT값 선언 뒤)

```
ALTER TABLE STUDENT ADD ( EMAIL VARCHAR2(50) DEFAULT 'NONE' NOT NULL );
```
DEFAULT 설정해주면 기존 데이터에 디폴트 값이 들어가므로 오류 발생하지 않음

- **컬럼 속성 변경 (MODIFY)** 데이터 타입이나 사이즈, DEFAULT 값, NOT NULL 여부 등을 수정할 때 사용

```
ALTER TABLE 테이블명
  MODIFY ( 컬럼명 새자료형 [DEFAULT 새기본값] [NOT NULL] ... );
```

- **데이터 사이즈 변경 및 데이터 타입 변경 예제**

```
ALTER TABLE STUDENT
  MODIFY ( STUDENT_NAME VARCHAR2(100) );
```

※ 컬럼 길이를 줄일 때는 기존 데이터 때문에 오류가 발생

ex) 150자 이름이 들어 있는 컬럼을 100자로 줄이면 에러
위 예시에서는 150자 까지만 축소 가능
그 테이블에서 **최대 사이즈 까지만 축소** 가능
컬럼 사이즈 증가는 항상 가능

```
ALTER TABLE STUDENT
  MODIFY ( STUDENT_NAME DATE );
```

※ 컬럼 타입을 호환되지 않는 경우 기존 데이터 때문에 오류가 발생

ex) 문자열을 날짜 형식으로 수정 시, 에러
테이블에 **데이터가 없는 경우 가능**
단, NUMBER → VARCHAR2 (숫자가 너무 큰 경우나, 문자 제한길이를 넘는 경우 오류), CHAR → VARCHAR2, VARCHAR2 → CHAR 로의 변환은 가능

- **여러 컬럼 변경 예제**

```
ALTER TABLE STUDENT
  MODIFY (
    STUDENT_NAME VARCHAR2(120),
    EMAIL        VARCHAR2(80) DEFAULT 'NO_EMAIL'
  );
```

여러 컬럼을 동시에 수정하려면
괄호 안에 각 컬럼 변경 사항을 쉼표로 구분

- **DEFAULT 값 변경 예제**

※ 기존 DEFAULT 값 변경하더라도 기존 데이터에는 영향을 주지 않음

※ DEFAULT 란?
특정 컬럼 값이 생략될 경우, 자동으로 부여되는 값
명시적으로 NULL을 입력할 경우에는 NULL이 들어감

```
ALTER TABLE STUDENT
  MODIFY ( EMAIL VARCHAR2(50) DEFAULT 'NO_EMAIL' );
```

DEFAULT 값 변경
ex) EMAIL 컬럼에 DEFAULT 값을 변경('NONE' → 'NO_EMAIL')

- **DEFAULT 값 해제 예제**

```
ALTER TABLE STUDENT
  MODIFY (EMAIL VARCHAR (5) DEFAULT NULL);
```

DEFAULT값 해제 시 **DEFAULT값을 NULL로 선언**

- **컬럼 이름 변경 (RENAME COLUMN)**

COLUMN 으로 끝나는 명령어는 한번에 두개 이상 시도 불가

```
ALTER TABLE 테이블명
  RENAME COLUMN 기존컬럼명 TO 새컬럼명;
```

※ 컬럼 이름 변경은 한 번에 하나씩 가능

```
ALTER TABLE STUDENT
  RENAME COLUMN STUDENT_NAME TO NAME;
```

ex) STUDENT_NAME 컬럼명을 NAME으로 변경

- **컬럼 삭제 (DROP COLUMN)**

```
ALTER TABLE 테이블명
  DROP COLUMN 컬럼명;
```

※ 한 번 컬럼이 삭제되면 롤백으로 되돌릴 수 없음

```
ALTER TABLE STUDENT
  DROP COLUMN EMAIL;
```

ex) EMAIL 컬럼 삭제
데이터 존재 여부와 상관없이 언제나 삭제

■ **DROP**

- 데이터베이스 객체(테이블, 뷰, 인덱스, 사용자 등)를 삭제하는 명령어
- 한 번 DROP하면 해당 객체 및 그 안의 데이터가 모두 영구적으로 삭제
  (단, **Recycle Bin**이 활성화된 환경에서는 일시적으로 휴지통에 들어갈 수 있으나, 일반적으로 '완전히 삭제된 것'으로 간주)

```
DROP TABLE 테이블명
[PURGE];
```

※ PURGE 란?
휴지통(Recycle Bin)을 사용하지 않고 바로 영구 삭제
Recycle Bin을 사용 중이라면 DROP 후에도 FLASHBACK TABLE 기능으로 테이블을 복구할 수 있는 경우가 있지만, PURGE까지 하면 복구가 불가능

```
DROP TABLE STUDENT;
```
→ `Table dropped.`

테이블 전체를 지웠기 때문에 빈 테이블이 아닌 테이블 또는 뷰가 존재하지 않는다는 에러 발생

```
SELECT * FROM STUDENT;
```
→ `ORA-00942: table or view does not exist`

■ **TRUNCATE**

- 모든 행 삭제, 테이블 구조는 유지
- DDL로 취급 (자동 커밋)
- DELETE와 달리 트랜잭션 제어(COMMIT, ROLLBACK) 대상이 아님
  RECYCLEBIN에 남지 않음

데이터 만 없는 상태기에 에러 문구가 아닌 데이터가 없다는 메시지 출력

```
TRUNCATE TABLE STUDENT;
```
→
```
SELECT * FROM STUDENT;
```
→ `no data found`

■ **DELETE vs DROP vs TRUNCATE**

DELETE	DML이라서 롤백 가능 (한 행, 전체 행 삭제 가능)
DROP	DDL로서 테이블 객체 자체를 제거 (롤백 불가) 데이터와 구조(스키마 정의), 인덱스, 제약 조건 등 모두 사라짐
TRUNCATE	DDL이라 실행 즉시 COMMIT, 부분 롤백 불가 전체 데이터를 한꺼번에 삭제

■ **제약조건**

데이터베이스가 스스로 데이터 무결성을 지키도록 도와주는 규칙
ex) "이 컬럼에는 절대 중복값이 들어가면 안 돼" 또는 "이 컬럼에는 절대 빈 값(NULL)이 들어가면 안 돼" 등 어떤 컬럼이나 컬럼의 조합에 대한 규칙을 DB가 강제하는 것

■ **제약조건 - PRIMARY KEY (기본키)**

- **행을 유일하게 식별**하기 위한 컬럼
- 중복을 허용하지 않고(Unique), Null을 허용하지 않음(Not Null)
- PK는 **하나의 테이블에 하나만 존재**할 수 있음
  - PK는 **하나의 컬럼 혹은 여러 개의 컬럼으로 결합**하여 지정 가능
- 테이블 복제 (CREATE TABLE 테이블 AS SELCT) 시, PK는 복사되지 않음.
  - NOT NULL을 제외한 제약조건은 복사되지 않는데, PK는 NOT NULL을 기능 포함하고 있지만 PK가 복사되지 않기 때문에 복사된 컬럼은 NOT NULL로 동작하지 않음

- **테이블 생성 시, 제약조건 함께 정의**

```
CREATE TABLE [스키마명.]테이블명 (
    컬럼명1 데이터타입
        [DEFAULT 기본값]
        [CONSTRAINT 제약조건이름] [제약조건종류
    컬럼명2 데이터타입
        [DEFAULT 기본값]
        [CONSTRAINT 제약조건이름] [제약조건종류],
    ...
    [CONSTRAINT 제약조건이름] PRIMARY KEY (컬럼명1[, 컬럼명2, ...])
);
```

CONSTRAINT 이름을 생략하면, 자동으로 시스템 이름이 부여

```
CREATE TABLE STUDENT ( 단일 컬럼 PK
    STUDENT_ID    NUMBER        CONSTRAINT PK_STUDENT PRIMARY KEY,
    STUDENT_NAME  VARCHAR2(50) DEFAULT 'UNKNOWN',
    REG_DATE      DATE
);

CREATE TABLE ORDER_DETAIL   복합 컬럼 PK (ORDER_ID, PRODUCT_ID) 한 쌍이 유일
    ORDER_ID   NUMBER,
    PRODUCT_ID NUMBER,
    QUANTITY   NUMBER,
    CONSTRAINT PK_ORDER_DETAIL
       PRIMARY KEY (ORDER_ID, PRODUCT_ID)
);
```

- **컬럼 추가 시, 제약조건 정의'**

```
ALTER TABLE 테이블명
  ADD (
      새컬럼명 데이터타입
        [DEFAULT 기본값]
        [CONSTRAINT 제약조건이름] [제약조건종류],
      ...
      [CONSTRAINT 제약조건이름] PRIMARY KEY (컬럼명1[, 컬럼명2, ...])
  );
```

```
ALTER TABLE STUDENT
  ADD (
    STUDENT_ID NUMBER
      CONSTRAINT PK_STUDENT PRIMARY KEY
  );
```

STUDENT 테이블에 STUDENT_ID 컬럼 추가하면서 PK로 지정

- **이미 존재하는 컬럼에 "PK 제약조건"만 추가**

```
ALTER TABLE 테이블명
  ADD CONSTRAINT 제약조건이름
  PRIMARY KEY (기존컬럼명1[, 기존컬럼명2, ...]);
```

```
ALTER TABLE STUDENT
  ADD CONSTRAINT PK_STUDENT
  PRIMARY KEY (STUDENT_ID);
```

STUDENT 테이블의 STUDENT_ID 컬럼을 PK로 설정

- **제약조건(PK) 삭제**

```
ALTER TABLE 테이블명
  DROP CONSTRAINT 제약조건이름;
```

```
ALTER TABLE STUDENT
  DROP CONSTRAINT PK_STUDENT;
```

관련 인덱스도 자동 삭제 (PK 는 Unique Index를 자동 생성함)

■ **제약조건 - UNIQUE**

- 중복 값을 허용하지 않음
  - NULL의 중복은 허용 (값이 없음 취급이기에 중복으로 인정하지 않음)
- 인덱스 자동 생성
- 한 테이블에 여러 Unique 컬럼 가능 (PK는 한 테이블에 하나만 가능하지만)
- 복합 Unique 가능
  - 하나의 컬럼을 유니크로 설정 가능하지만 여러 컬럼 조합도 가능
  - 여러 컬럼(예: (COL1, COL2) 조합)이 중복되면 안 되게 설정 가능

- **유니크 생성 예제**

```
CREATE TABLE MEMBER (
    MEMBER_ID   NUMBER PRIMARY KEY,        테이블 생성 시 (컬럼 레벨)
    EMAIL       VARCHAR2(100) CONSTRAINT UQ_MEMBER_EMAIL UNIQUE,
    NAME        VARCHAR2(50)
);

CREATE TABLE MEMBER (
    MEMBER_ID   NUMBER,
    EMAIL       VARCHAR2(100),
    NAME        VARCHAR2(50),          테이블 생성 시 (테이블 레벨)
    CONSTRAINT PK_MEMBER_ID PRIMARY KEY (MEMBER_ID),
    CONSTRAINT UQ_MEMBER_EMAIL UNIQUE (EMAIL)
);

ALTER TABLE MEMBER
  ADD CONSTRAINT UQ_MEMBER_EMAI    이미 만들어진 테이블에 나중에 UNIQUE 추가
  UNIQUE (EMAIL);
```

```
INSERT INTO MEMBER (MEMBER_ID, EMAIL, NAME) VALUES (1, 'alpha@demo.com', 'Alpha');
INSERT INTO MEMBER (MEMBER_ID, EMAIL, NAME) VALUES (2, 'alpha@demo.com', 'Delta');
INSERT INTO MEMBER (MEMBER_ID, EMAIL, NAME) VALUES (3, NULL, 'Beta');
INSERT INTO MEMBER (MEMBER_ID, EMAIL, NAME) VALUES (4, NULL, 'Gamma');
```

에러 발생! 'alpha@demo.com' 이미 존재.

UNIQUE 제약이 걸린 EMAIL 컬럼이지만, NULL은 중복 가능.

■ 제약조건 - NOT NULL

- **컬럼에 NULL(빈 값)이 들어가는 것을 방지**
- 다른 제약조건과 다르게 컬럼 생성 시, 제약조건을 설정하지 않으면 **Nullable 상태로 생성**
  - 즉, 추후에 NOT NULL 선언 시, 제약조건 생성이 아닌 수정 쿼리
- 다른 제약조건과 다르게 테이블 복제 시, **NOT NULL 조건도 같이 복제**

```
ALTER TABLE MYTABLE
  MODIFY COL1 NOT NULL;
```

ALTER … ADD 가 아닌 ALTER … MODIFY 를 통한 변경

■ 제약조건 - FOREIGN KEY

- **자식(Child) 테이블에서 특정 컬럼**이, 부모(Parent) 테이블의 **PRIMARY KEY(또는 UNIQUE) 컬럼을 참조**하도록 설정
  - 이를 통해 **참조 무결성을 보장**

ex) 자식 테이블에 존재하는 학번(Student ID)는 반드시 부모 테이블에 학번 정보가 있어야 함

```
CREATE TABLE 자식테이블명
(
    자식컬럼1     데이터타입,
    자식컬럼2     데이터타입,
    ...
    [CONSTRAINT 제약이름]
       FOREIGN KEY (자식컬럼명)
       REFERENCES 부모테이블명(부모컬럼명)
);
```

```
CREATE TABLE DEPT (부모 테이블
    DEPT_ID     NUMBER          CONSTRAINT PK_DEPT PRIMARY KEY,
    DEPT_NAME VARCHAR2(50)
);

CREATE TABLE EMP (  자식 테이블
    EMP_ID      NUMBER          CONSTRAINT PK_EMP PRIMARY KEY,
    EMP_NAME    VARCHAR2(50),
    DEPT_ID     NUMBER,
    CONSTRAINT FK_EMP_DEPT
       FOREIGN KEY (DEPT_ID)
       REFERENCES DEPT(DEPT_ID)
);
```

외래 키 제약만 나중에 추가하는 방법

```
ALTER TABLE EMP
  ADD CONSTRAINT FK_EMP_DEPT
  FOREIGN KEY (DEPT_ID)
  REFERENCES DEPT(DEPT_ID);
```

- **부모 테이블에 존재하지 않는 값을 자식의 FK 컬럼에 삽입/수정하면 에러**

```
INSERT INTO EMP (EMP_ID, EMP_NAME, DEPT_ID)
VALUES (1, 'Alice', 999);
```

```
UPDATE EMP
   SET DEPT_ID = 999
 WHERE EMP_ID = 101;
```

DEPT_ID = 999인 부서가 없으면, INSERT / UPDATE 모두 오류
즉, 자식 레코드가 참조하려는 부모 레코드가 미리 존재해야 하며,
없는 부모를 가리키는 FK 값으로는 수정(INSERT/UPDATE)할 수 없음

```
ORA-02291: 무결성 제약 조건(SQL_SAANNTKJLJQSDFBGHZBIUORET.FK_EMP_DEPT) 위반 - 부모 키를 찾을 수 없음
```

- **자식 테이블 데이터 삭제**

```
DELETE FROM EMP
 WHERE EMP_ID = 101;
```

자식 테이블 행을 지우는 것은 자유롭게 **가능**
부모 테이블에 대한 참조 무결성은 "부모가 없어지면 자식은 어떡하나" 를 걱정하는 제약이므로, 자식이 스스로 없어지는 것은 문제되지 않음

★ ※ 제약에 대해서 조금 헷갈린다면, 아래의 예시로 생각해볼 수 있습니다.
부모 입장에서는 자식이 있을 수도, 없을 수도 있지만
부모 없이 자식은 태어날 수 없다고 생각하면 쉽습니다.

- **자식 테이블에서 값을 참조 중인 데이터는 부모 테이블에서 값을 변경/삭제할 수 없음** 해당 쿼리를 통해 부모, 자식 데이터 세팅

```
INSERT INTO DEPT (DEPT_ID, DEPT_NAME) VALUES (10, 'SALES');
                                                  참조
INSERT INTO EMP (EMP_ID, EMP_NAME, DEPT_ID) VALUES (101, 'Alice', 10);
```

```
UPDATE DEPT
   SET DEPT_ID = 999
 WHERE DEPT_ID = 10;
```

```
DELETE FROM DEPT
 WHERE DEPT_ID = 10;
```

자식(EMP) 테이블에서 아직 DEPT_ID=10을 참조하는 행이 있으므로, 기본적으로 제약조건 위배로 에러 발생.

```
ORA-02292: 무결성 제약 조건(SQL_SAANNTKJLJQSDFBGHZBIUORET.FK_EMP_DEPT) 위반 - 자식 레코드 발견
```

■ FOREIGN KEY 옵션 ★★

옵션을 통해 **부모(Parent) 테이블의 행이 삭제될 때 자식(Child) 테이블의 행을 어떻게 처리할 지 결정**할 수 있음

ON DELETE CASCADE	부모 행 삭제 시, 해당 부모를 참조하고 있는 자식 행도 **자동 삭제**
ON DELETE SET NULL	부모 행 삭제 시, 자식에서 해당 FK 컬럼을 **NULL로 설정**

```
CREATE TABLE EMP (
    EMP_ID   NUMBER CONSTRAINT PK_EMP PRIMARY KEY,
    EMP_NAME VARCHAR2(50),
    DEPT_ID  NUMBER,
    CONSTRAINT FK_EMP_DEPT
        FOREIGN KEY (DEPT_ID)
        REFERENCES DEPT (DEPT_ID)
        ON DELETE CASCAD
);
```

부모 행 삭제 시, 해당 부모를 참조하고 있는 자식 행도 자동 삭제.

```
CREATE TABLE EMP (
    EMP_ID   NUMBER CONSTRAINT PK_EMP PRIMARY KEY,
    EMP_NAME VARCHAR2(50),
    DEPT_ID  NUMBER,
    CONSTRAINT FK_EMP_DEPT
        FOREIGN KEY (DEPT_ID)
        REFERENCES DEPT (DEPT_ID)
        ON DELETE SET NULL
);
```

■ CHECK

컬럼(또는 컬럼들의 조합)의 값이 특정 조건(논리식)을 만족하도록 강제

```
CREATE TABLE EMPLOYEE (
    EMP_ID   NUMBER PRIMARY KEY,
    EMP_NAME VARCHAR2(50) NOT NULL,
    SALARY   NUMBER
        CONSTRAINT CK_EMP_SALARY
        CHECK (SALARY >= 0)
);
```

SALARY가 음수일 경우, INSERT or UPDATE 시 오류 발생
NULL은 특별히 막지 않았으므로,
SALARY가 NULL이 들어오면 제약을 위배하지는 않음

■ 오브젝트 종류 오브젝트의 경우, 학습량에 비해 시험 출제 빈도가 낮음

- **View (뷰)**

실제로 데이터를 저장하는 물리적 테이블이 아니라, 기존 테이블(또는 다른 뷰)에 대한 SELECT 쿼리 결과를 가상 테이블처럼 사용하는 객체
즉, "**SELECT문을 이름 붙여 재사용**"한다고 볼 수 있음

- **View의 특징**

1. 실제 **데이터 보관 X**
   - 데이터는 원본 테이블에 있고, 뷰는 SQL 문장을 통해 **실시간으로 원본 데이터**를 보여줌
2. 특정 유저에게 뷰에만 접근하게 하여, **보안 용도로도 활용** 가능 (데이터 보호)
3. 테이블에서 유도되었기 때문에, 구조가 기존 테이블과 같으며 조작 또한 기본 테이블과 유사함
4. 이미 정의되어 있는 뷰는 **다른 뷰의 기초**가 될 수 있음
5. 기존 테이블 삭제 시, 해당 테이블을 **참조한 뷰 역시 삭제**됨

- **View 종류**

단순 뷰	한 개 테이블에서 일부 컬럼만 가져온 뷰
복합 뷰	여러 테이블을 조인

- **View 문법**
  - **생성**

```
CREATE [OR REPLACE] VIEW 뷰이름
AS
SELECT ...
```

이미 생성된 뷰에 대해서 SELECT 쿼리 바꾸고 싶을 때, 대체하는 쿼리

  - **삭제**

```
DROP VIEW 뷰이름;
```

- **View 예제**

```
CREATE VIEW VW_EMP_BASIC
AS
SELECT EMP_ID, EMP_NAME
  FROM EMP
 WHERE DEPT_ID = 10;
```
뷰 생성

```
SELECT * FROM VW_EMP_BASIC;
```
뷰 조회

결과

EMP_ID	EMP_NAME
101	Alice
102	Bob

- **View 장점**

1. 논리적 독립성
   - 원본 테이블 구조가 바뀌어도, 뷰를 통해 접근하는 쿼리는 크게 변경 없이 유지할 수 있음
2. 보안 향상
   - 원본 테이블 전체를 공개하지 않고, 필요한 컬럼/행만 노출 가능
3. 복잡한 쿼리 단순화
   - 자주 쓰는 SELECT 문을 하나의 이름(뷰)으로 저장 → 사용 편의성, 쿼리 간결화
4. 다양한 데이터 지원 가능

- **View 단점**

1. 갱신 제약
   - 복잡 뷰(조인, 그룹함수)는 INSERT/UPDATE/DELETE가 제한되어, 실제 데이터를 변경하기 어려움.
2. 성능 저하 우려
   - 뷰 자체는 실시간으로 SELECT 문을 실행하므로, 큰 테이블을 조인/집계하는 뷰는 자주 부를 때 오버헤드가 발생할 수 있음
3. 인덱스 구성 불가

- **시퀀스(SEQUENCE)**

자동으로 순차적인 숫자(일반적으로 1씩 증가)를 생성해 주는 DB 객체
주로 기본 키(ID 컬럼) 값 등 유일한 숫자를 쉽게 부여할 때 사용
ex) 회원 가입할 때 MEMBER_ID 자동 증가 주문 번호, 송장 번호 등을 중복 없는 순서로 생성

```
CREATE SEQUENCE [스키마명.]시퀀스이름
[START WITH 시작번호] 시작 번호 (기본값 1)
[INCREMENT BY 증가값] 증가 값 (기본값 1, 음수 가능)
[MINVALUE 숫자 | NOMINVALUE] 최소값/최대값 지정
[MAXVALUE 숫자 | NOMAXVALUE] (기본값: NOMINVALUE, NOMAXVALUE)
[CYCLE | NOCYCLE] 최대값 도달 시 다시 최소값으로 순환할지 여부 (기본: NOCYCLE)
[CACHE 숫자 | NOCACHE]; 미리 숫자를 캐싱해서 성능을 높임 (기본 20 정도), NOCACHE면 매번 디스크 액세스
```

- **시퀀스 예제**

```
CREATE SEQUENCE SEQ_STUDENT_ID
    START WITH 1
    INCREMENT BY 1
    NOMAXVALUE
    NOCYCLE
    CACHE 20;
```
1부터 시작, 1씩 증가, 최대값 없이 끝없이 증가
20개씩 캐시 → 성능 향상

```
CREATE SEQUENCE SEQ_EXAMPLE
    START WITH 100
    INCREMENT BY -1
    MINVALUE 1
    CYCLE;
```
100부터 시작, 1씩 감소
1까지 내려가면 다시 100(최대값)으로 순환

```
CREATE SEQUENCE SEQ_EXAMPLE
    MINVALUE 50
    MAXVALUE 200
    START WITH 100
    INCREMENT BY 10
    CYCLE
    CACHE 10;
```
100부터 시작, 10씩 증가
100 → 110 → ⋯200 → 50 → 60 → ⋯

- **시노님(SYNONYM)**

시노님은 DB 객체(테이블, 뷰, 시퀀스 등)에 대한 별칭을 부여하는 것
ex) USER1.EMPLOYEE_TBL 이라는 테이블에 SYN_EMP라는 별칭(시노님)을 만들어두면, 다른 사용자가 SYN_EMP라고 간단하게 접근할 수 있음.

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM 시노님이름
FOR [스키마명.]객체이름;
```

기존 시노님이 이미 존재할 때, **CREATE SYNONYM**으로 만들면 보통 **이미 존재한다**는 에러 발생
이때 **OR REPLACE 옵션**을 쓰면, 동일한 시노님 이름이 이미 있어도 **덮어쓰기(재정의)**

PUBLIC: 생략 시 Private Synonym, 명시 시 Public Synonym
Private 시노님은 시노님을 생성한 유저만 사용 가능
Public은 공용 시노님

```
CREATE SYNONYM EMP_ALIAS
FOR SCOTT.EMP;

SELECT * FROM EMP_ALIAS;
```
생성
조회

DCL

■ DCL (Data Control Language)

객체에 대한 권한을 부여하거나 회수하는 기능
ex) 인사 관련 테이블은 인사팀만 접근 가능하도록 접근 제어

시스템 권한	**DB 전반적인 작업**을 할 수 있는 권한 예) CREATE SESSION (DB 접속), CREATE TABLE, CREATE ANY TABLE, DROP ANY TABLE, ALTER ANY TABLE 등 관리자 권한을 가진 사람만 권한 부여 및 회수 가능
객체 권한	특정 **스키마 객체**(테이블, 뷰, 시퀀스 등)에 대해 "SELECT, INSERT, UPDATE, DELETE, REFERENCES" 등을 수행할 수 있는 권한 테이블 소유자는 타계정에 소유 테이블에 대한 조회 및 수정 권한 부여 및 회수 가능

■ GRANT

권한 부여

- **시스템 권한**

```
GRANT <시스템권한> [, <시스템권한> ...]
TO <사용자 or ROLE> [, <사용자 or ROLE> ...];
```

```
GRANT CREATE TABLE TO HR;
```
HR 사용자에게 '본인 스키마에 테이블 생성' 권한 부여

- **객체 권한**

```
GRANT <객체권한> [, <객체권한> ...]
ON <스키마.객체>
TO <사용자 or ROLE> [, <사용자 or ROLE> ...];
```

```
GRANT SELECT, INSERT, UPDATE
ON SCOTT.EMP
TO HR, DEVELOP;
```
객체권한 (종류)은 한번에 여러 권한 부여 가능
객체는 한번에 하나만 부여 가능
사용자 or ROLE은 동시에 여러 유저에게 부여 가능

■ REVOKE

권한 회수
이미 회수한 권한을 **재회수 시, 오류 발생**

객체권한 (종류)은 한번에 여러 권한 회수 가능
사용자 or ROLE은 동시에 여러 유저에게 회수 가능

```
REVOKE <객체권한> [, <객체권한> ...]
ON <스키마.객체>
FROM <사용자 or ROLE> [, <사용자 or ROLE> ...];
```

```
REVOKE SELECT, INSERT
ON SCOTT.EMP
FROM HR;
```
이제 HR은 SCOTT.EMP를 조회/삽입 할 수 없음

■ ROLE

권한의 집합 (여러 권한들을 한번에 부여 및 회수)
System 계정을 통한 Role 생성 가능

```
CREATE ROLE <role이름>;

GRANT <시스템권한 or 객체권한> [, ...]
TO <role이름>;
```
롤 생성
생성된 롤에 대해서 권한 설정

```
REVOKE <시스템권한 or 객체권한> [, ...]
FROM <role이름>;
```
롤에 존재하는 권한 회수
해당 롤을 부여 중인 유저는 회수당한 권한을 사용할 수 없음

- **롤에 권한 담기 예제**

```
GRANT SELECT, INSERT, UPDATE
ON SCOTT.EMP
TO MANAGER_ROLE;

GRANT CREATE SESSION, CREATE TABLE
TO MANAGER_ROLE;
```
MANAGER_ROLE은 SCOTT.EMP에 대한 **SELECT/INSERT/UPDATE 가능** + **시스템 권한(세션 생성, 테이블 생성) 보유**

```
GRANT MANAGER_ROLE TO HR;
```
HR이 **MANAGER_ROLE**을 사용 가능해짐
결과적으로 HR은 **MANAGER_ROLE 안에 정의된 모든 권한을 상속**

※ 롤을 부여 후, 롤 안에 있는 권한을 직접 회수하는 것은 불가
롤 회수를 통한 회수만 가능

■ 권한 부여 옵션 (중간 관리자 권한)

- **WITH GRANT OPTION**

대상: **객체 권한**(SELECT, INSERT, UPDATE, DELETE, EXECUTE 등)
이 옵션을 달아서 **권한을 부여**하면, **해당 권한을 받은 사용자가 또 다른 사용자에게 동일한 객체 권한을 재부여** 가능

- **WITH ADMIN OPTION**

대상: **시스템 권한**(CREATE TABLE, DROP ANY TABLE, CREATE SESSION 등) 또는 ROLE
이 옵션을 붙이면, 해당 권한(또는 ROLE)을 부여받은 사람이 **다른 사용자에게도** 그 권한(또는 ROLE)을 부여 가능

- **시험 출제 포인트** ★★

1. 중간 관리자가 제 3 계정에게 부여한 권한을 최상위 관리자가 회수할 수 있는가?
2. 중간 관리자의 권한이 최상위 관리자에 의해 회수되었을 경우, 중간 관리자가 부여한 제 3계정의 권한도 회수 되는가?

WITH GRANT OPTION: 중간 관리자가 부여한 권한은 최상위 관리자가 회수할 수 없음 (중간관리자만 회수 가능)
WITH ADMIN OPTION: 중간 관리자가 부여한 권한도 최상위 관리자가 회수 가능

WITH GRANT OPTION: 중간 권한이 회수될 경우, 중간관리자에 의해 부여된 제 3계정들의 권한도 같이 회수됨
WITH ADMIN OPTION: 중간 권한이 회수될 경우, 중간관리자에 의해 부여된 제 3계정들의 권한은 남아 있음

- **예제**

```
GRANT SELECT ON SCOTT.EMP
TO HR
WITH GRANT OPTION;
```

SCOTT 테이블 EMP에 대해, HR이 SELECT할 수 있도록 권한 부여
HR이 다른 사용자에게도 **SELECT 권한을 넘겨줄 수 있게 함**
SYS 계정이 HR이 부여한 **제 3 계정에 대해서 권한 회수 불가**
HR의 권한이 회수 될 때, **제 3 계정의 권한도 같이 회수**

```
GRANT MANAGER_ROLE
TO HR
WITH ADMIN OPTION;
```

HR이 또 다른 사용자에게 ROLE을 넘길 수 있음
SYS 계정이 HR이 부여한 **제 3 계정에 대한 권한 회수 가능**
HR이 권한 회수 되더라도, **제 3 계정에 권한은 유지됨**