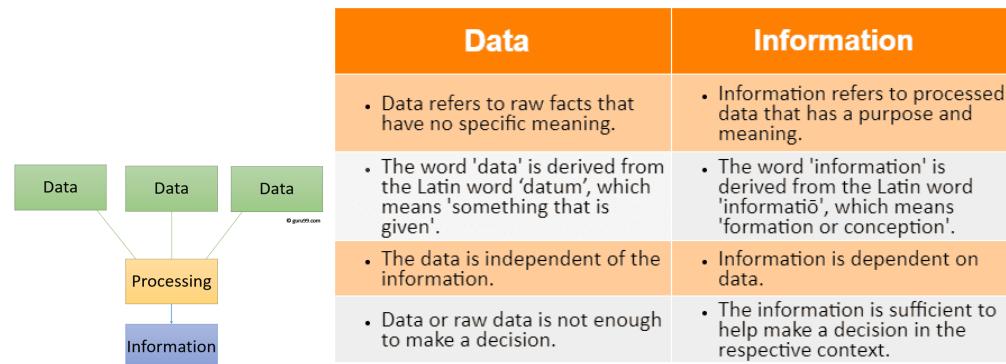


1. 데이터베이스 기본 이론

- 데이터베이스: 데이터 & 베이스의 합성어
- DBMS: Database Management System; 데이터베이스 관리 시스템
(예) 오라클, MySQL
- 데이터: 사전에서는 자료, 정보라는 의미를 가지고 있다. 하지만 데이터베이스 분야에서는 자료와 정보는 다른 의미를 가진다.

따라서, 우리는 데이터와 정보를 명확하게 구분해서 사용해야 한다.



(example)

A 카드사에서 발급한 카드를 사용한 커피 전문점 결제 내역 (자료 = 데이터)

→ 결제 내역이라는 즉, 평가하거나 정제하지 않은 날 것의 자료 그 자체

결제 분포의 최상위 순위를 30~40대 남성이 차지했다는 결과 (= 정보)

→ 어떠한 ‘목적’을 집어넣어서 분석, 가공을 하여 ‘가치를 추가’하여 새로운 의미를 도출하는 것을 바로 정보라고 한다.

즉, 데이터분석이란

정제되지 않은 데이터 그 자체를 토대로 분석과 가공을 통하여 새로운 가치가 담긴 ‘정보’를 얻어내는 과정을 의미한다.

커피 전문점 결제 내역 데이터를 저장한다고 가정해보자.

- 데이터 중 일부
 - 엑셀 같은 스프레드시트 파일에 나누어 저장
 - 어떤 특정 프로그램으로 관리하는 파일에 저장
 - 실제 종이 문서에 기록하여 캐비닛에 보관
- 데이터가 분산 관리되고 있음
 - 시간과 비용이 많이 들
 - 최신 데이터를 정확하게 찾기 어려움
 - 데이터 누락이나 중복 발생

∴ 결론: 데이터 분석을 위해서는 ‘통합 관리’가 필요 → DB의 출현배경

- 효율적 데이터 관리를 위한 조건 (= 통합 관리를 위한 조건)
 - 데이터 통합관리
 - 일관된 방법으로 관리 (= 데이터 입력 방식이 통일되어야 취합이 가능)

예를 들어 5/10일은 엑셀로, 5/11일은 워드로 데이터를 입력 X → 일관성 必
 - 데이터 누락 및 중복 제거

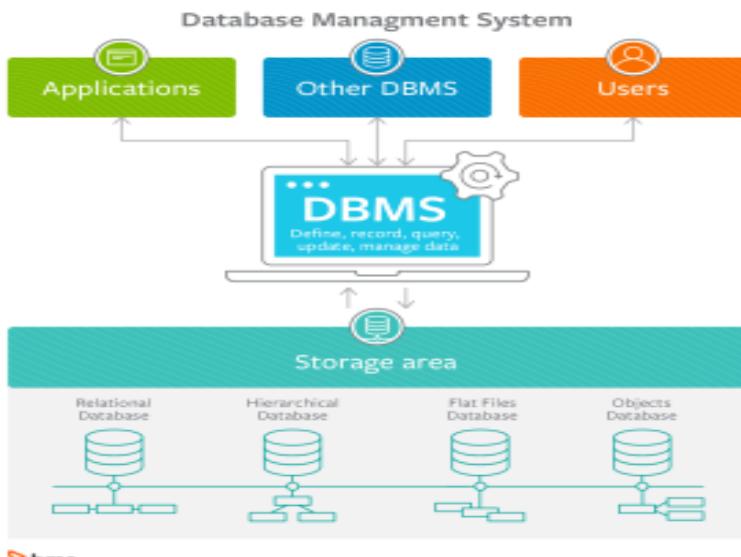
일관된 방법으로 관리할 경우 누락이나 중복된 데이터를 빠르게 검색 가능
 - 여러 사용자가 공동으로 실시간 사용 가능

DB 구축의 목적은 ‘다양한 사람들이 함께’ 사용하기 위함에 있다 (사유화 X)

- 파일 시스템 vs. 데이터베이스

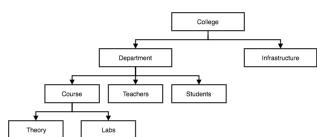
파일 시스템	데이터베이스
<p>서로 다른 여러 응용 프로그램이 제공하는 기능에 맞게 필요한 데이터를 ‘각각 저장’하고 관리</p> <p>문제점: 서로 연관이 없고 중복 및 누락이 발생</p>	<p>데이터 조작과 관리를 극대화한 시스템 소프트웨어</p> <p>keyword: 통합관리 실무에서는 DB와 DBMS의 구분 X</p> <p>www.educba.com</p>

- the role of DB and DBMS



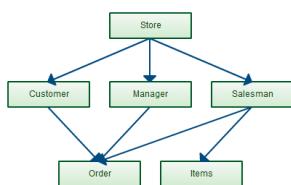
- 데이터 모델: 컴퓨터에 데이터를 저장하는 방식을 정의해 놓은 개념 모형

(A) 계층형



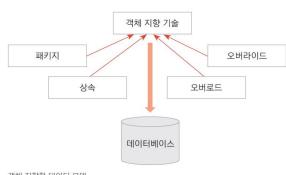
- 나뭇가지 형태의 트리 구조를 활용
- 데이터 관련성을 계층별로 나누어 부모-자식 같은 관계를 정의하고 데이터 관리
- 일대다 관계의 데이터 구조
- 부모가 여러 자식을 가질 순 있지만, 자식이 여러 부모를 가질 순 없다.

(B) 네트워크형



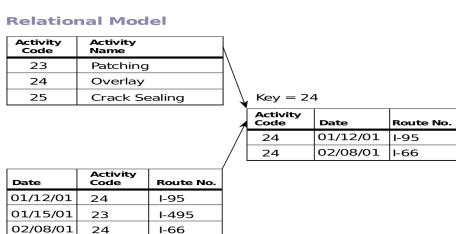
- 계층형 구조의 보완버전이다.
- 그래프 기반 구조를 기반으로 개체 간 관계를 그래프 구조로 연결하여 자식이 여러 부모를 가질 수 있다.

(C) 객체지향형



- 객체 지향 프로그래밍에 사용하는 객체 개념을 기반으로 한 모델
- 데이터를 독립된 객체로 구성, 관리, 상속, 오버라이드 등의 기능을 활용할 수 있다.
- DB에 적용이 쉽지 않다.

(D) 관계형

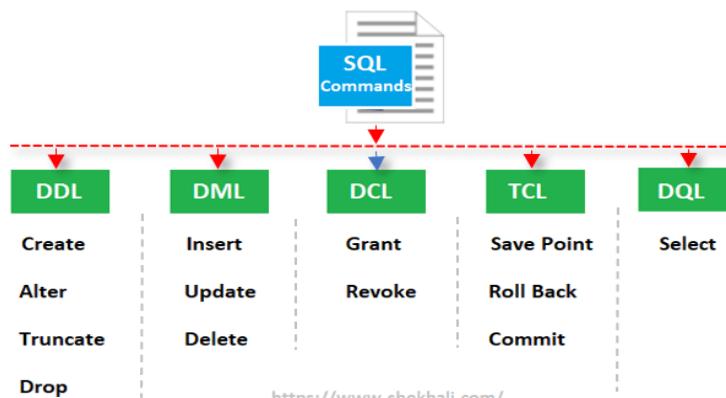


- 현재 가장 많이 사용하고 있는 DB 모델(예: 오라클)

- RDB: 관계형 데이터 모델 개념을 바탕으로 데이터를 저장 및 관리하는 데이터베이스로 1980년 후반부터 지금까지 가장 많이 사용하는 데이터베이스
(example) MS-SQL, MySQL, MariaDB, PostgreSQL, DB2, Oracle etc.
- SQL: Structured Query Language
데이터베이스를 다루고 관리하는데 사용하는 데이터베이스 질의 언어
즉, 데이터에 대해 물어보고 결과를 얻는다.

※ DQL은 데이터분석가로서 반드시 알아야 하는 언어! (DML, DDL도 아는 것이 좋음)

DQL(Data Query Language)	데이터를 원하는 방식으로 조회하는 명령어
DML(Data Manipulation Language)	데이터를 저장/수정/삭제하는 명령어
DDL(Data Definition Language)	여러 객체를 생성/수정/삭제하는 명령어
TCL(Transaction Control Language)	트랜잭션 데이터 영구 저장/취소 등 명령어
DCL(Data Control Language)	데이터의 사용 권한과 관련된 명령어



- 관계형 데이터베이스의 구성 요소
 - 테이블
: 2차원 표 형태로 저장 및 관리

가로줄: 행 = row = tuple = record

→ 저장하려는 하나의 개체를 구성하는 여러 값의 가로로 늘어뜨린 형태

(예) 학생 한 명의 데이터 : 학번, 이름 등

세로줄: 열 = column = attribute = field

→ 저장하려는 데이터를 대표하는 이름과 공통 특성을 정의

저장 정보의 종류, 저장 가능한 값의 최대 길이, 값의 중복을 허용하지 않음 등의 저장 조건과 범위를 지정할 수 있다.

(예) 학번: 숫자, 8자리, 중복 허용하지 않음

Table			Row Store	Column Store
Country	Product	Sales		
US	Alpha	3.000	US	US
US	Beta	1.250	US	JP
JP	Alpha	700	Beta	UK
UK	Alpha	450	Alpha	Alpha
			Beta	Beta
			Alpha	Alpha
			3.000	3.000
			1.250	1.250
			700	700
			450	450

∴ 행과 열의 특성에 맞춰 데이터를 저장한 테이블 하나하나가 **RDB의 관계**

여러 테이블의 구성과 관계를 잘 규정하고 관리하는 것이 데이터 관리의 핵심

- 키

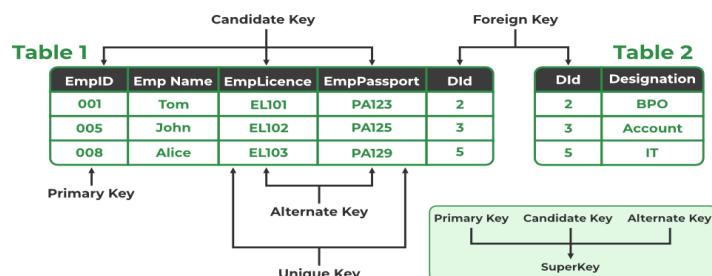
: 테이블을 열 수 있는 하나의 열쇠, 즉 수많은 데이터를 구별할 수 있는 유일한 값

(예) 학생을 구분할 수 있는 유일한 값 = 학번

환자를 구분할 수 있는 유일한 값 = 환자번호

데이터를 구별하거나 테이블 간의 연관관계를 표현할 때 키로 지정한 열 사용

종류: 기본키, 보조키, 외래키, 복합키



기본키(Primary Key): 가장 중요한 키로, 한 테이블 내에서 중복되지 않는 값만 가진다.

기본키의 속성: 유일한 값, 중복 없음, NULL 값 없음

(예) 학생 정보 테이블 : '학번'으로 각 학생을 구별 가능 (왜? 같은 학번은 존재 X)

후보키(Candidate Key; 보조 키): 대체 키로 기본 키가 될 수 있는 조건을 만족하는 열

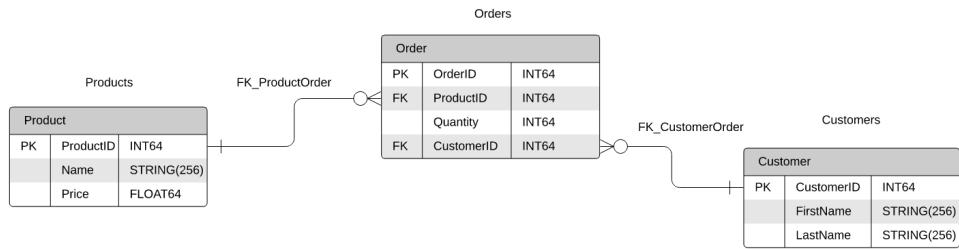
cf. 그렇다면 기본키와 후보키의 차이는 무엇인가?

사실 정말 텍스트 그대로 기본키이거나 아니었느냐의 차이일뿐, 기본적인 특징(중복되지 않는 고유의 값을 가져야 한다, null 값을 가져야 한다 등)은 동일하게 적용된다.

그렇다면 왜 기본키와 후보키로 나누었을까?

한 테이블 내에서 기본키는 하나만 있어도 충분하지만, 이러한 특성을 가진 것이 2개 이상인 경우에는 1개만을 기본키로 지정하고 나머지는 '후보키'로 둔다.

외래키(Foreign Key): 특정 테이블에 포함되어 있으면서 다른 테이블의 기본 키



다른 테이블에 있는 기본키를 끌어올 때 → 외래키 (두 테이블의 ‘연결’할 때 발생)

외래키의 개념이 존재하는 이유? 테이블이 분리되어 있기 때문

그렇다면, 왜 굳이 테이블을 분리할까?

데이터가 많아진다면 엄청난 양의 중복 데이터가 발생하며, 그로 인한 데이터 처리 비용 증가 및 저장 공간 크기와 관리의 문제가 빚어질 수 있기 때문이다.

엑셀로 관리한다면 ‘병합’을 할 수 있지만, DB에서는 병합이 불가능하여 테이블 분리

복합키(Composite Key): 여러 열을 조합하여 기본 키 역할을 할 수 있게 만든 키로 적개는 2~3개, 많게는 10개 이상인 경우도 있다.

복합키의 존재 이유? 하나의 열만으로 행을 식별하는 것이 불가능하기 때문

∴ 두 개 이상의 열을 조합하여 각 행이 유일한 데이터로서 가치를 가지는 것

cf. 복합키에 대한 궁금증

A. 한 테이블 안에서 복합키는 있을 수도 있고, 없을 수도 있다? YES!

B. 복합키가 있다면 그 테이블의 기본키는 없다? NO!

▶ 복합키와 기본키는 ‘별개’의 개념이다. 한 테이블 안에 기본키는 반드시 있으나, 때때로 복합키가 기본키가 될 수도 있고 아닐 수도 있다. 중요하게 짚고 넘어가야 할 점은! 복합키가 테이블에서 어떠한 역할을 하느냐이다. 복합키는 테이블 내에서 고유한 조합을 식별하거나 테이블 간의 관계를 정의하는데 사용되고, 여러 열을 함께 검색하거나 정렬하는 역할을 도맡는다.

Composite Key

Composite Key			
Roll_No	Name	Age	Phone
1	Arya	21	7491901521
2	Bran	19	8491901000
3	John	24	9291018403
4	Max	24	7903084562

- 오라클 데이터베이스: 대표적 상용 관계형 데이터베이스 제품

- 자료형

데이터는 다양한 형태를 가지고 있다.

가장 많이 사용하는 자료형: VARCHAR2, NUMBER, DATE

VARCHAR2(길이) : 4000byte 만큼의 가변 길이 문자열 데이터를 저장할 수 있다.

→ 불필요한 공간을 줄여서 자료를 효율적으로 관리하기 위한 자료형

NUMBER(전체 자릿수, 소수점 이하 자릿수): 총 38자릿수의 숫자를 저장할 수 있다.

DATE: 날짜 형식을 저장하기 위해 사용하는 자료형

- 객체: 데이터를 저장하고 관리하기 위한 논리 구조를 가진 구성 요소

객체	설명
테이블(table)	데이터를 저장하는 장소
인덱스(index)	테이블의 검색 효율을 높이기 위해 사용
뷰(view)	하나 또는 여러 개의 선별된 데이터를 논리적으로 연결하여 하나의 테이블처럼 사용하게 해줌 (SQL 문의 결과)
시퀀스(sequence)	일련 번호를 생성
시노ним(synonym)	오라클 객체의 별칭(다른 이름)을 지정 (공식별칭 != 일회성)
프로시저와 함수는 SQL문 순서를 나열하는 것을 의미	
<u>프로시저(procedure)</u>	프로그래밍 연산 및 기능 수행이 가능(반환 값 없음)
<u>함수(function)</u>	프로그래밍 연산 및 기능 수행이 가능(반환 값 있음)
패키지(package)	관련 있는 프로시저와 함수를 보관
트리거(trigger)	데이터 관련 작업의 연결 및 방지 관련 기능을 제공

- PL/SQL

데이터 관리를 하기 위해 복잡한 기능이 필요할 때 기존의 SQL만으로는 한계가 있다.

이때 데이터 관리를 위한 별도의 프로그래밍 언어가 PL/SQL

변수, 조건문, 반복문 등 프로그래밍 언어에서 제공하는 요소를 사용하여 데이터 관리

→ 이 부분은 파이썬으로 할 수 있으므로 다루지는 않는다.

2. 데이터 조회

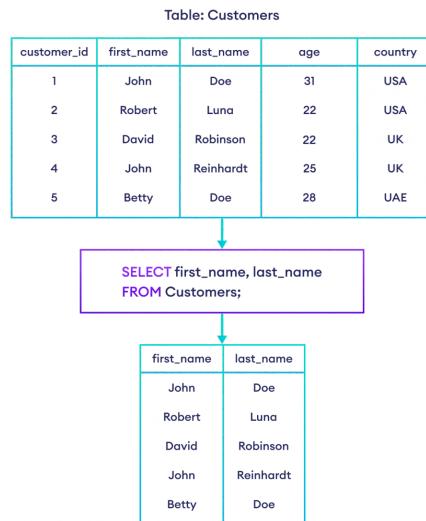
- SQL 작성 규칙: 대소문자를 구분하지 않기 때문에 대문자로 작성

WHY? 프로그래밍 언어는 소문자 vs. 쿼리는 대문자

if) Python 안에 SQL 언어가 들어갔을 경우의 구분은?

→ 두 언어를 동시에 사용할 경우 구분이 쉬워진다.

- SELECTION: 행 단위로 원하는 데이터를 조회



- PROJECTION: 열 단위로 데이터를 조회

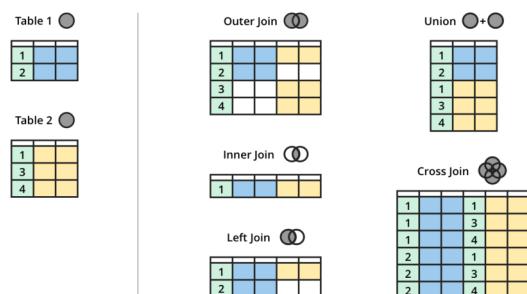


- SELECTION + PROJECTION

- JOIN: 두 개 이상의 테이블을 사용하여 하나의 테이블처럼 데이터 조회

Combining Data Tables – SQL Joins Explained

A JOIN clause in SQL is used to combine rows from two or more tables, based on a related column between them.



6. DQL(Data Query Language): 데이터를 조회하는 명령어

기본 문법: **SELECT FROM** → SQL은 여기서 살을 붙여나가는 것이다.

▶ **SELECT** 열 이름 **FROM** 테이블 이름(데이터)

(ex) **SELECT ENAME FROM EMP;**

▶ 여러 개의 열 이름을 사용할 때는 쉼표로 연결시킴

(ex) **SELECT ENAME, JOB FROM EMP;**

▶ 전체 열을 조회할 때는 별표(=모든 열을 보겠다)를 사용할 수 있음

(ex) **SELECT * FROM EMP;**

```
-- 급여 등급 정보  
DESC SALGRADE;  
  
--  
-- SQL 기본 문법  
-- SELECT ~ FROM ~  
-- SELECT 열 이름 FROM 테이블 이름  
-- *는 모든 열을 의미한다.  
SELECT * FROM EMP;  
-- 14개의 행과 8개의 열이 있는 2차원 구조의 테이블  
  
-- IF 사원 이름만 보고싶다면? (PROJECTION)  
SELECT ENAME FROM EMP;  
  
-- IF 사원 이름과 직책을 '동시에' 보고 싶다면?  
SELECT ENAME, JOB FROM EMP;
```

중복 데이터 제거: **DISTINCT** (개별 행이 아닌 전체 열에 적용 = **DISTINCT**를 사용할 땐 **SELECT**문 전체에 적용해야 하므로 가장 앞에 작성)

(ex) **SELECT DEPTNO FROM EMP;**

SELECT DISTINCT DEPTNO FROM EMP;

SELECT DISTINCT JOB, DEPTNO FROM EMP;

DISTINCT DEPTNO은 EMP 테이블 안에서 중복되는 DEPTNO 제거

DISTINCT JOB과 **DEPTNO** 두 개를 하나의 데이터로 보고,

그 중에서 중복되지 않는 데이터만 추출

(기본적으로 중복 결과 개수: 단일 < 다수)

※ **DISTINCT**를 사용할 때 **SELECT * FROM ~**은 사용 불가능

why? 중복 데이터를 제거할 열 데이터를 지정해야 사용 가능하므로

열 이름의 별칭: **AS** (별칭을 사용하는 방법은 네 가지이지만, AS를 사용하는 것만 기억!!!!)

(ex) **SELECT ENAME, SAL, SAL*12 + COMM FROM EMP;**

SELECT ENAME, SAL, SAL*12 + COMM AS ANNSAL FROM EMP;

SELECT ENAME, SAL, SAL+...+SAL(TOTAL 12)+COMM FROM EMP;

SELECT ENAME, SAL, SAL + SAL + ... + SAL(TOTAL 12) + COMM AS ANNSAL FROM EMP;

실무에서는 AS를 선호(알아보기 쉽고 복합 사용에서 에러 발생률 줄임

원하는 순서로 데이터 정렬: **ORDER BY ~**

SELECT ~ FROM ~ ORDER BY ~

ORDER BY 뒤에 정렬하려는 열 이름과 정렬 옵션을 작성

(ex) SELECT * FROM EMP ORDER BY SAL ASC;

SELECT * FROM EMP ORDER BY EMPNO;

ASC는 생략 가능(default value)

SELECT * FROM EMP ORDER BY ASL DESC;

SELECT * FROM EMP ORDER BY EMPNO DESC;

SELECT * FROM EMP ORDER BY DEPTNO ASC, SAL DESC;

EmployeeID	EmployeeLastName	EmployeeFirstName	EmailID
003	Jones	Amy	amy@gmail.com
006	Brown	Dan	dan@gmail.com
001	Donald	Jo	jo@gmail.com

SELECT *
FROM Employee
ORDER BY
EmployeeLastName;

Result

EmployeeID	EmployeeLastName	EmployeeFirstName	EmailID
006	Brown	Dan	dan@gmail.com
001	Donald	Jo	jo@gmail.com
003	Jones	Amy	amy@gmail.com

2개 이상의 정렬 진행 순서

	NationalIDNumber	SickLeaveHours	Vacationhours	BirthDate	MaritalStatus	Gender	ModifiedDate	SalariedFlag
1	674171828	20	1	1986-09-08	M	M	2014-06-30 00:00:00.000	0
2	872923042	20	1	1974-09-10	M	M	2014-06-30 00:00:00.000	0
3	18418830 ^{ASC}	20	0	1976-01-06	M	F	2014-06-30 00:00:00.000	1
4	153288994	21	3	1971-08-30	M	M	2014-06-30 00:00:00.000	0
5	509647174	21	2	1974-11-12	M	M	2014-06-30 00:00:00.000	1
6	695256908	22	5	1952-09-27	M	F	2014-06-30 00:00:00.000	1
7	912141525	22	5	1990-01-25	M	F	2014-06-30 00:00:00.000	0
8	339233463	22	5	1953-04-30	M	M	2014-06-30 00:00:00.000	0
9	152085091	22	4	1978-06-26	M	M	2014-06-30 00:00:00.000	0
10	701156975	22	4	1970-11-12	M	M	2014-06-30 00:00:00.000	0
11	56920285	22	4	1961-05-02	M	F	2014-06-30 00:00:00.000	1
12	25011600	23	7	1987-11-22	M	F	2014-06-30 00:00:00.000	0
13	204035155	23	7	1973-01-24	M	M	2014-06-30 00:00:00.000	0
14	886023130	23	6	1990-11-04	M	M	2014-06-30 00:00:00.000	0
15	718299860	23	6	1972-11-25	M	M	2014-06-30 00:00:00.000	0
16	998320692	23	6	1959-03-11	M	M	2014-06-30 00:00:00.000	1

※ ORDER BY 사용 시 유의사항

꼭 필요한 경우가 아니라면 사용하지 않는 것이 좋다.

why? 정렬하는 시간이 많이 걸리기 때문에 정렬하지 않을 때 더 빠르게 출력

→ SQL문 효율이 낮아진다는 점은 서비스 응답 시간이 느려진다는 뜻

필요한 데이터만 조회하는 절: **WHERE**

조건을 만족하는 행 단위 조회

SELECT ~ FROM ~ WHERE ~

(ex) SELECT * FROM EMP WHERE DEPTNO = 30;

SELECT * FROM EMP WHERE EMPNO = 7782;

여러 개의 조건식을 사용하는 **AND**, **OR** 연산자

AND = 교집합

(ex) SELECT * FROM EMP WHERE DEPTNO = 30 AND EMPNO = 7499;

SELECT * FROM EMP WHERE DEPTNO = 30 AND JOB = 'SALESMAN';

※ SQL문 문자열은 작은따옴표만 가능!

OR = 합집합

(ex) SELECT * FROM EMP WHERE DEPTNO = 20 OR EMPNO = 7499;

SELECT * FROM EMP WHERE DEPTNO = 30 OR JOB = 'SALESMAN';

조건식의 개수는 제한이 없으므로 AND, OR이 함께 사용 가능하다.

실무의 영역?!

AND 연산자를 많이 사용(why? 다양한 조건을 '한번에 만족'시키는 데이터를 추출해야 하는 경우가 많기 때문)

산술 연산자

: 사칙연산(+, -, *, /)

(ex) SELECT * FROM EMP WHERE SAL * 12 = 36000;

비교 연산자

(ex) SELECT * FROM EMP WHERE SAL >= 3000;

SELECT * FROM EMP WHERE SAL >= 3000 AND JOB = 'ANALYST';

SELECT * FROM EMP WHERE SAL >= 3000 AND MGR IS NOT NULL OR JOB = 'MANAGER';

등가 비교 연산자

연산자	사용법	의미
=	A = B	A 값이 B값과 같을 경우 true, 다를 경우 false
!=	A != B	A 값이 B값과 다를 경우 true, 같을 경우 false
<>	A <> B	
^=	A ^= B	

(ex) SELECT * FROM EMP WHERE SAL != 3000;

논리 부정 연산자

작성한 결과의 반대를 보고 싶다면 NOT을 범용적으로 사용 가능

SELECT * FROM EMP WHERE NOT SAL = 3000;

IN 연산자

SELECT ~ FROM ~ WHERE 열 이름 IN (값1, 값2, ..., 값n)

IN 연산자: OR 조건을 여러 개로 출력

SELECT * FROM EMP WHERE JOB IN ('MANAGER', 'SALESMAN', 'CLERK')

NOT IN 연산자

SELECT ~ FROM ~ WHERE 열 이름 NOT IN (값1, 값2, ..., 값n)

AND 조건을 여러 개로 반대의 결과를 출력

SELECT * FROM EMP WHERE JOB NOT IN ('MANAGER', 'SALESMAN', 'CLERK')

= MANAGER, SALESMAN, CLERK가 아닌 직업을 찾는다

```
-- NOT IN 연산자 = AND 조건
SELECT * FROM EMP;
SELECT * FROM EMP WHERE JOB NOT IN ('CLERK', 'MANAGER', 'SALESMAN');
```

	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
1	7788	SCOTT	ANALYST	7566	87/04/19	3000	(null)	20
2	7839	KING	PRESIDENT	(null)	81/11/17	5000	(null)	10
3	7902	FORD	ANALYST	7566	81/12/03	3000	(null)	20

BETWEEN

SELECT ~ FROM ~ WHERE 열 이름 BETWEEN A AND B

SELECT * FROM EMP WHERE SAL BETWEEN 200 AND 3000;

SELECT * FROM EMP WHERE SAL >= 2000 AND SAL <= 3000;

※ NOT BETWEEN A AND B = RANGE OF NOT A OR NOT B

UsuarioID	NombreUsuario	ApellidoP	ApellidoM	Edad	Sexo
1	Jose Francisco	Olivares	Martinez	22	M
2	Jorge	Castañeda	Morales	20	M
3	Rocio	Morales	Magallon	21	F
4	Jose Ramon	Tirado	Ramirez	30	M
5	Edgar	Perez	Puentes	40	M
6	Yolanda	Martinez	Estrada	33	F
7	Jose Ramon	Palacios	Doriga	34	M

```
-- BETWEEN : SELECT ~ FROM ~ WHERE 열 이름 BETWEEN A AND B
SELECT * FROM EMP;
SELECT * FROM EMP WHERE SAL BETWEEN 2000 AND 3000;
-- NOT BETWEEN
SELECT * FROM EMP WHERE SAL NOT BETWEEN 2000 AND 3000;
SELECT * FROM EMP WHERE SAL < 2000 OR SAL > 3000;
```

LIKE 연산자와 와일드 카드

: 일부 문자열이 포함된 데이터를 조회할 때

SELECT * FROM EMP WHERE ENAME LIKE 'S%'; ▶ S로 시작하는 모든 이름을 추출해라

SELECT * FROM EMP WHERE ENAME LIKE '_L%'; ▶ 두 번째 글자가 S로 시작하는 모든 이름을 추출해라

SELECT * FROM EMP WHERE ENAME LIKE '%S%'; ▶ S가 포함된 모든 이름을 추출해라

SELECT * FROM EMP WHERE ENAME LIKE '%AM%'; ▶ AM이 포함된 모든 이름을 추출해라

Wildcards

Wildcards are used to search for data within a table. These characters are used with the LIKE operator.

Wildcard	Description
%	Zero or more characters
_	One single character
[charlist]	Sets and ranges of characters to match
[^charlist] or [!charlist]	Matches only a character NOT specified within the brackets

※ 실무에서의 LIKE 연산자와 와일드 카드

사용하기 간편하고 기능 면에서 활용도가 높지만 데이터 조회 성능에 대한 의견은 분분하다. 와일드 카드를 어떻게 사용하느냐에 따라 조회 시간에 차이가 있으며, 조회 속도는 제공하는 서비스의 질과 직접적으로 연관되는 경우가 많으므로 중요하다.

따라서 실무에서는 여러가지 테스트하여 조회 시간이 빠른 쿼리 완성을 위해 노력해야 한다.

IS NULL 연산자

NULL은 데이터 값이 완전히 비어 있는 상태를 의미한다.

NULL은 무슨 짓을 해도 NULL!!

NULL의 의미	
값이 존재하지 않음	통장을 개설한 적 없는 은행 고객의 계좌번호
해당 사항 없음	미혼인 고객의 결혼기념일
노출할 수 없는 값	고객 비밀번호 찾기 같은 열람을 제한해야 하는 특정 개인정보
확인되지 않은 값	미성년자의 출신 대학

집합 연산자

데이터 조회한 결과를 하나의 집합과 같이 다룰 수 있다.

UNION: 연결된 SELECT문의 결과 값을 합집합으로 (중복은 제거)

UNION ALL: UNION과 동일하나 중복 값도 제거 없이 모두 출력

MINUS: 차집합

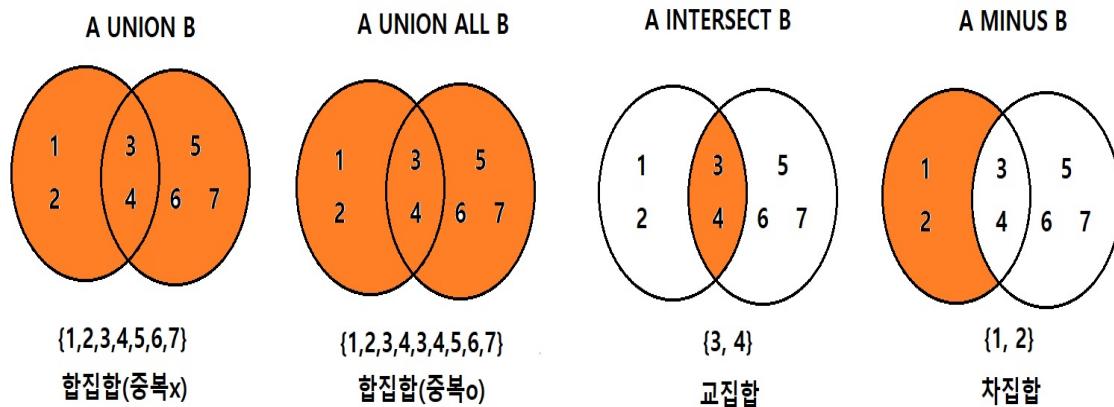
INTERSECT: 교집합

UNION

: 첫 번째 SELECT절과 두 번째 SELECT 절 뒤에 나오는 열 이름은 모두 동일

※ 잘못된 합집합이 만들어지는 경우

- A. 특정 열이 누락된 경우
- B. 열 순서가 다르고 데이터 유형이 다른 경우
- C. 열 순서가 다르지만 데이터 유형이 같은 경우



▶ 함수

- 입력값 : **input** → 데이터가 들어가는 부분
- 기능 : **function** → 함수의 기능, 실제로 작동하는 부분
- 출력값 : **output** → 반환값, 리턴값, 함수 적용의 결과
- 함수의 종류
 - 사용자 정의 함수: 사용자가 직접 만들어야 하는 함수
 - 내장함수: 이미 만들어져 있는 함수

내장 함수는 나오는 결과의 개수에 따라서 크게 두 가지로 나뉘어진다.

- **단일행** 함수: 행마다 결과가 나오는 함수 (집어넣은 개수 = 나오는 개수)
- **다중행** 함수: 하나의 행으로 결과가 나오는 함수 (주로 요약/기술통계)

1. 단일행함수

A. 문자 데이터를 가공하는 함수

▶ UPPER, LOWER, INITCAP

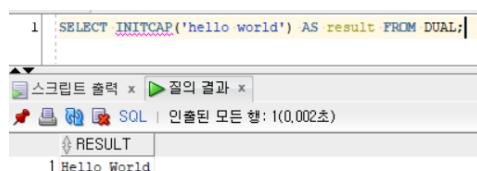
*INITCAP(): 첫글자는 대문자, 나머지는 소문자로 변환하는 함수

(예) hello world 소문자를 INITCAP 함수를 적용하였을 때

cf. DUAL: 관리자 계정 SYS 소유 테이블, 함수 기능이나 연산 등 테스트 목적으로 사용하는 가상 테이블

테스트할 때 불러오는 구색을 맞추기 위해 불러오는 테이블 why? SQL은 SELECT ~ FROM ~ 을 반드시 만족해야 하기 때문에 데이터를 불러올 '테이블'이 반드시 존재해야만 한다.

```
1 | SELECT INITCAP('hello world') AS result FROM DUAL;
```



▶ LENGTH, LENGTHB(= 문자열 바이트 수)

```
SELECT ENAME, LENGTH(ENAME) FROM EMP WHERE LENGTH(ENAME) >= 5;
```

LENGTHB: 한 글자당 영어는 1byte, 한글은 2byte로 계산되어 출력된다.

영어에서는 LENGTH와 LENGTHB가 같은 결과로 출력되어 차이를 구분하지 못할 수도 있다.

하지만 LENGTH는 글자 '길이'를 LENGTHB는 글자의 '바이트'를 구한다

```
SELECT LENGTH('HELLO'), LENGTHB('HELLO') FROM DUAL;
```



```
SELECT LENGTH('안녕하세요'), LENGTHB('안녕하세요') FROM DUAL;
```



▶ SUBSTR(= 문자열 일부추출)

첫 번째 숫자는 시작 위치, 두 번째 숫자는 끝 위치(생략한다면 끝까지)

(예)

```
SELECT JOB, SUBSTR(JOB, 1, 2) FROM EMP; JOB이라는 데이터에서 첫번째 글자부터 두번째 글자까지 출력!
```

	JOB	SUBSTR(JOB,1,2)
1	CLERK	CL
2	SALESMAN	SA
3	SALESMAN	SA
4	MANAGER	MA
5	SALESMAN	SA
6	MANAGER	MA

SELECT JOB, SUBSTR(JOB, 3 ,5) FROM EMP;

SELECT JOB, SUBSTR(JOB, 3) FROM EMP;

-- 응용) 글자수를 예측하기 어려울 때 **LENGTH()**을 사용하여 문자열을 일부 추출하자!

STEP 1. LENGTH('CLERK') : 1 ~ 5 (LENGTH 함수를 사용하여 글자수를 확인한다)

※ 글자수는 파이썬과 다르게 제로 인덱스가 아닌 원 인덱스! 1부터 시작한다.

マイナス는 뒤에서부터! 예) LENGTH('CLERK') : -5 ~ -1

CODE) SELECT JOB, LENGTH(JOB) FROM EMP; JOB의 글자수를 확인

STEP 2. SELECT JOB, SUBSTR(JOB, -LENGTH(JOB), 2) FROM EMP;

(a) SUBSTR(JOB → JOB 데이터의 글자에서 일부 글자만 추출할거야!

(b) LENGTH(JOB) → JOB 데이터 안에 있는 글자 수를 구해줘!

(c) SUBSTR(JOB, -LENGTH(JOB),2) → -LENGTH(JOB) = 처음 글자 수를 반환하게 된다.

(예) JOB에 CLERK가 있는 경우, LENGTH(JOB) = 5 거기에 -를 붙였다면 맨 처음 글자수!

=> CLERK을 기준으로 본다면: SUBSTR(JOB, -5, 2) = -5, -4의 글자를 출력하기

	JOB	SUBSTR(JOB,1,2)
1	CLERK	CL
2	SALESMAN	SA
3	SALESMAN	SA
4	MANAGER	MA
5	SALESMAN	SA
6	MANAGER	MA
7	MANAGER	MA
8	ANALYST	AN
9	PRESIDENT	PR
10	SALESMAN	SA
11	CLERK	CL
12	CLERK	CL
13	ANALYST	AN
14	CLERK	CL

Positive Indexing	1	2	3	4	5	6	7	8	9	10	11	12	13
String	S	Q	L	I	t	e		S	U	B	S	T	R
Negative Indexing	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

<-----

▶ INSTR (원본문자열, 찾을문자열, [시작위치], [반복횟수])

-- 특정 문자 위치

```
SELECT INSTR('HELLO, ORACLE!', 'L'), -- L 글자가 처음 나타나는 위치  
INSTR('HELLO, ORACLE!', 'L', 5), -- 5번째부터 L 글자가 처음 나타나는 위치  
INSTR('HELLO, ORACLE!', 'L', 2, 2) -- 2번째부터 L 글자가 두번째로 나타나는 위치  
FROM DUAL;
```

INSTR('HELLO,ORACLE!', 'L')	INSTR('HELLO,ORACLE!', 'L', 5)	INSTR('HELLO,ORACLE!', 'L', 2, 2)--2번째부터L글자가두번째로나타나는위치
3	12	4

▶ REPLACE (원본문자열, 찾을문자열, 대체할문자열)

-- 다른 문자로 대체

```
SELECT '010-1234-5678',  
REPLACE('010-1234-5678', '-', ''), -- 대시(-) 기호를 BLANK로 대체  
REPLACE('010-1234-5678', '-', '*'), -- 대시(-) 기호를 *로 대체  
REPLACE('010-1234-5678', '-') -- 대시(-) 기호를 삭제? NO! (기호가 사라졌기 때문에 '삭제'로 이해할 수  
있지만, 원론적인 개념은 삭제가 아닌 "" 공백으로 대체가 되었다는 것 = 실무에서 삭제를 시키고 싶을 때  
대체문자열을 비워두어 '삭제' 개념으로 사용해도 무방)
```

'010-1234-5678'	REPLACE('010-1234-5678', '-', '')	REPLACE('010-1234-5678', '-', '*')	REPLACE('010-1234-5678', '-')--대시(-)기호를삭제
010-1234-5678	010 1234 5678	010*1234*5678	01012345678

▶ LPAD/RPAD(= 왼쪽/오른쪽부터 채우기) - 남은 자리를 어디부터 채울 것인가?

기본 구조: **LPAD/RPAD**(원본문자열, 총길이, 패딩문자열)

```
SELECT 'Oracle',  
LPAD('Oracle', 10, '#'), -- 10자리, Oracle 글자를 오른쪽부터 넣고, 남은자리를 왼쪽에서 # 채움  
RPAD('Oracle', 10, '*'), -- 10자리, Oracle 글자를 왼쪽부터 넣고, 남은자리를 오른쪽에서 * 채움  
LPAD('Oracle', 10), -- 10자리, Oracle 글자를 오른쪽부터 넣고, 남은자리를 왼쪽에서 공백 채움  
RPAD('Oracle', 10) -- 10자리, Oracle 글자를 왼쪽부터 넣고, 남은자리를 오른쪽에서 공백 채움  
FROM DUAL;
```

'ORACLE'	LPAD('ORACLE',10,'#')	RPAD('ORACLE',10,'*')	LPAD('ORACLE',10)	RPAD('ORACLE',10)--10자리,ORACLE글자를왼쪽부터넣고,남은자리를오른쪽에서공백채움
Oracle	####Oracle	Oracle****	Oracle	Oracle

▶ CONCAT(= 문자열 합치기)

기본구조: CONCAT(문자열1, 문자열2, ...)

|| (vertical bar) 로도 같은 결과를 볼 수 있다.

SELECT * FROM EMP; → 전체 테이블 조회해서 어떤 값이 있는지 확인

SELECT EMPNO, ENAME FROM EMP WHERE ENAME = 'SCOTT';

→ 이름이 SCOTT인 사람의 EMPNO, ENAME 데이터를 뽑아내기

	EMPNO	ENAME
1	7788	SCOTT

SELECT CONCAT (EMPNO, ENAME) FROM EMP WHERE ENAME = 'SCOTT';

→ SCOTT의 이름을 가진 사람의 EMPNO와 ENAME 데이터를 합쳐라

	CONCAT(EMPNO,ENAME)
1	7788SCOTT

SELECT CONCAT (':', ENAME), CONCAT(EMPNO, CONCAT(':', ENAME)) FROM EMP WHERE ENAME = 'SCOTT';

→ 위의 데이터는 보기 어렵기 때문에 두 데이터 사이에 콜론을 넣어 구분할 수 있도록 할 수 있다.

즉 CONCAT 안에 또 다른 CONCAT 함수를 삽입할 수 있다.

SQL 인출된 모든 행: 1(0,002초)	
	CONCAT(':',ENAME)
1	:SCOTT
	CONCAT(EMPNO,CONCAT(':',ENAME))
1	7788:SCOTT

cf. 같은 표현 (버티칼바로도 표현할 수 있다.)

SELECT ':' || ENAME, EMPNO || ':' || ENAME FROM EMP WHERE ENAME = 'SCOTT';

* || ; vertical bar?

버티칼 바는 문자열 연결 연산자로 사용되는 기호로, 두 개의 문자열을 이어 붙여 하나의 문자열로 만들어 준다.

= **CONCAT()** 함수의 역할과 동일, 즉 **CONCAT** 함수에서 단어열을 구분하는 쉼표(,)에 ||를 사용하면 된다.

▶ TRIM / LTRIM / RTRIM (=특정 문자 지우기; 주로 공백 제거)

-- TRIM = 앞뒤 = TRIM BOTH FROM

-- LTRIM = 앞 = 왼쪽 = TRIM LEADING FROM

-- RTRIM = 뒤 = 오른쪽 = TRIM TRAILING FROM

-- 어떤 문자를 지울지 정하지 않을 경우 공백을 제거

```
SELECT '['||' Oracle '||']', '['||TRIM(' Oracle ')||']', '['||TRIM(BOTH FROM ' Oracle ')||']', '['||TRIM(LEADING FROM ' Oracle ')||']', '['||TRIM(TRAILING FROM ' Oracle ')||']' FROM DUAL;
```

```
SELECT '['||' Oracle '||']', '['||TRIM(' Oracle ')||']', '['||TRIM(BOTH FROM '__Oracle__')||']', '['||LTRIM(' Oracle ')||']', '['||RTRIM(' Oracle ')||']' FROM DUAL;
```

B. 숫자 데이터를 연산하고 수치를 조정하는 숫자 함수

ROUND : 반올림 (반올림, 표현할 자리수이자 반올림하는 자리수가 아니라는 점 유의하기)

TRUNC : 버리기

CEIL : 지정된 숫자의 가장 가까운 큰 정수값 → 높은 값을 추출

FLOOR : 지정된 숫자의 가장 가까운 작은 정수값 → 낮은 값을 추출

MOD : 나누기의 나머지값

CEIL(3,14)	FLOOR(3,14)	CEIL(-3,14)	FLOOR(-3,14)
1	4	3	-3

cf. CEIL과 FLOOR이 헷갈린다면?

-- 정수 반환 함수

-- **CEIL** = 천장 = 높은 정수 값 (예: $3 < 3.14 < 4$; 4가 출력)

-- **FLOOR** = 바닥 = 낮은 정수 값 (예: $3 < 3.14 < 4$; 3이 출력)

C. 날짜 데이터를 다루는 함수

▶ 날짜 데이터의 연산

더하기에서 ○ : 날짜 데이터 + 숫자, X : 날짜 데이터 + 날짜 데이터

빼기: 날짜 데이터 - 숫자, 날짜 데이터 - 날짜 데이터 (○)

SYSDATE : 현재 날짜와 시간 / ADD_MONTHS : 특정 개월 이후 날짜 / MONTH_BETWEEN : 특정 개월 수 차이

NEXT_DAY : 돌아오는 요일의 날짜 / LAST_DAY : 특정 월의 마지막 날짜

(예)

```
SELECT SYSDATE AS 오늘날짜, SYSDATE - 1, SYSDATE + 1 FROM DUAL;
```

```
SELECT SYSDATE, ADD_MONTHS(SYSDATE, 3) FROM DUAL;
```

```
SELECT ENAME, HIREDATE, ADD_MONTHS(HIREDATE, 120) FROM EMP;
```

-- 입사 41주년 미만 / 초과

```
SELECT ENAME, HIREDATE, ADD_MONTHS(HIREDATE, 492), SYSDATE FROM EMP WHERE  
ADD_MONTHS(HIREDATE, 492) > SYSDATE;
```

```
SELECT ENAME, HIREDATE, ADD_MONTHS(HIREDATE, 492), SYSDATE FROM EMP WHERE  
ADD_MONTHS(HIREDATE, 492) < SYSDATE;
```

	ENAME	HIREDATE	ADD_MONTHS(HIREDATE,492)	SYSDATE
1	SCOTT	1987/04/19	2028/04/19	2023/05/23
2	ADAMS	1987/05/23	2028/05/23	2023/05/23

-- 해당 월의 마지막 날짜

```
SELECT SYSDATE, NEXT_DAY(SYSDATE, '수요일'), LAST_DAY(SYSDATE) FROM DUAL;
```

	SYSDATE	NEXT_DAY(SYSDATE,'수요일')	LAST_DAY(SYSDATE)
1	2023/05/23	2023/05/24	2023/05/31

D. 형 변환 함수

▶ 자료형을 반환하는 함수

- 암시적 형 변환: 숫자를 문자데이터로 만들어서 숫자와 연산할 경우

- 형 변환 함수

-- 암시적 형 변환: 문자 (1000)을 숫자로 자동으로 변환

```
SELECT EMPNO, EMPNO + '1000' FROM EMP;
```

-- 1,000는 문자 = 숫자 변환 불가 = 에러 발생

```
SELECT EMPNO, EMPNO + '1,000' FROM EMP; → 에러문구: ORA-01722: 수치가 부적합합니다
```

결론: 콤마(,)가 있을 경우 프로그램이 숫자라는 것을 인식하지 못하기 때문에 암시적 형 변환이 불가능하다.

- 명시적 형 변환

- TO_CHAR(): 숫자 또는 날짜 데이터를 문자 데이터로 변환

(코드 예시) SELECT SAL, TO_CHAR(SAL, '\$999,999'), -- \$ & 천 단위, 구분

TO_CHAR(SAL, 'L999,999'), -- L: 로컬 통화량 = 한국 원 & 천 단위, 구분

TO_CHAR(SAL, '999,999.00'), -- 소수점 두자리 & 숫자로만 표현

TO_CHAR(SAL, '000,999,999.00'),

TO_CHAR(SAL, '000999999.99'), -- 총 11자리 = 정수 9자리수 & 소수점 2자리

TO_CHAR(SAL, '999,999,00')

FROM EMP;

	SAL	TO_CHAR(SAL, '\$999,999')	TO_CHAR(SAL, 'L999,999')	TO_CHAR(SAL, '999,999.00')	TO_CHAR(SAL, '000,999,999.00')	TO_CHAR(SAL, '000999999.99')	TO_CHAR(SAL, '999,999.00')
1	800	\$800	₩800	800.00	000,000,008,00	000000800.00	8,00
2	1600	\$1,600	₩1,600	1,600.00	000,000,016,00	000001600.00	16,00
3	1250	\$1,250	₩1,250	1,250.00	000,000,012,50	000001250.00	12,50
4	2975	\$2,975	₩2,975	2,975.00	000,000,029,75	000002975.00	29,75
5	1250	\$1,250	₩1,250	1,250.00	000,000,012,50	000001250.00	12,50
6	2850	\$2,850	₩2,850	2,850.00	000,000,028,50	000002850.00	28,50
7	2450	\$2,450	₩2,450	2,450.00	000,000,024,50	000002450.00	24,50
8	3000	\$3,000	₩3,000	3,000.00	000,000,030,00	000003000.00	30,00
9	5000	\$5,000	₩5,000	5,000.00	000,000,050,00	000005000.00	50,00
10	1500	\$1,500	₩1,500	1,500.00	000,000,015,00	000001500.00	15,00
11	1100	\$1,100	₩1,100	1,100.00	000,000,011,00	000001100.00	11,00
12	950	\$950	₩950	950.00	000,000,009,50	000000950.00	9,50
13	3000	\$3,000	₩3,000	3,000.00	000,000,030,00	000003000.00	30,00
14	1300	\$1,300	₩1,300	1,300.00	000,000,013,00	000001300.00	13,00

- TO_NUMBER(): 문자 데이터를 숫자 데이터로 변환

SELECT 1300 - TO_NUMBER('1500'), TO_NUMBER('1300') + 1500 FROM DUAL;

-- IF) SELECT 1300 - '1,500', '1,300' + 1500 FROM DUAL; 이렇게 천 단위 표시(,)를 하면 숫자로 인식하지 못해서 암시적 형 변환이 불가능! 이럴 땐 명시적 형 변환을 해야한다! 대신 뒤에 표시형식을 반드시 표기

SELECT 1300 - TO_NUMBER('1,500', '999,999'), TO_NUMBER('1,300', '999,999') + 1500 FROM DUAL;

- TO_DATE(): 문자 데이터를 날짜 데이터로 변환

SELECT TO_DATE('2023-05-19', 'YYYY/MM/DD') FROM DUAL;

SELECT TO_DATE('20230519', 'YYYY/MM/DD') FROM DUAL;

SELECT TO_DATE('20230519', 'YYYY-MM-DD') FROM DUAL;

(문제) 사원 데이터에서 81년 7월 1일 이후에 입사한 사람

SELECT * FROM EMP WHERE HIREDATE > TO_DATE('1981-07-01', 'YYYY-MM-DD');

	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
1	7654	MARTIN	SALESMAN	7698	1981/09/28	1250	1400	30
2	7788	SCOTT	ANALYST	7566	1987/04/19	3000	(null)	20
3	7839	KING	PRESIDENT	(null)	1981/11/17	5000	(null)	10
4	7844	TURNER	SALESMAN	7698	1981/09/08	1500	0	30
5	7876	ADAMS	CLERK	7788	1987/05/23	1100	(null)	20
6	7900	JAMES	CLERK	7698	1981/12/03	950	(null)	30
7	7902	FORD	ANALYST	7566	1981/12/03	3000	(null)	20
8	7934	MILLER	CLERK	7782	1982/01/23	1300	(null)	10

E. NULL 처리 함수

NULL은 기본적으로 모든 연산에 NULL을 출력하지만, IN (NOT) NULL에서만 TRUE/FALSE를 출력한다.

- 함수 NVL(): NULL값일 때 내가 원하는 값으로 변경, 나머지는 그대로(원래 가지고 있던 값 그대로 유지)

```
SELECT EMPNO, ENAME, COMM, SAL, SAL + COMM, NVL(COMM,0), SAL+NVL(COMM,0) FROM EMP;
```

- 함수 NVL2(): 첫 번째 매개변수의 경우에는 NULL값이 아닐 때 내가 원하는 값으로 변경, 두 번째 매개변수는 NULL값일 때 내가 원하는 값으로 변경

```
SELECT EMPNO, ENAME, COMM, SAL, NVL2(COMM,'O','X'), NVL2(COMM,SAL * 12 + COMM, SAL * 12)  
FROM EMP;
```

cf. NVL() vs. NVL2()?

NVL 함수는 두 개의 매개변수를 사용하여 첫 번째 매개변수가 널인 경우 두 번째 매개변수로 '대체'

NVL2 함수는 세 개의 매개변수를 사용하여 첫 번째 매개변수가 널이 아닌 경우 두 번째 매개변수, 널인 경우 세 번째 매개변수로 대체

※ 두 함수의 궁극적인 차이는 널이 아닌 값도 다른 매개변수로 대체할 수 있느냐 없느냐!

F. 상황에 따라 다른 데이터를 반환하는 함수

- DECODE()**: 같다는 조건만 사용 가능
- 기본 구조: **DECODE(표현식, 검색값1, 결과값1, 검색값2, 결과값2, ..., 디폴트값)**

```
SELECT EMPNO, ENAME, JOB, SAL, DECODE(JOB, 'MANAGER', SAL * 1.1, 'SALESMAN', SAL * 1.05,  
'ANALYST', SAL, SAL * 1.03) AS NEW_SAL FROM EMP;
```

	EMPNO	ENAME	JOB	SAL	NEW_SAL
1	7369	SMITH	CLERK	800	824
2	7499	ALLEN	SALESMAN	1600	1680
3	7521	WARD	SALESMAN	1250	1312.5
4	7566	JONES	MANAGER	2975	3272.5
5	7654	MARTIN	SALESMAN	1250	1312.5
6	7698	BLAKE	MANAGER	2850	3135
7	7782	CLARK	MANAGER	2450	2695
8	7788	SCOTT	ANALYST	3000	3000
9	7839	KING	PRESIDENT	5000	5150
10	7844	TURNER	SALESMAN	1500	1575
11	7876	ADAMS	CLERK	1100	1133
12	7900	JAMES	CLERK	950	978.5
13	7902	FORD	ANALYST	3000	3000
14	7934	MILLER	CLERK	1300	1339

- **CASE()**: 같다는 조건 말고 다른 조건도 사용 가능
- **기본 구조:** CASE WHEN 조건1 THEN 결과1 WHEN 조건2 THEN 결과2 ... ELSE 기본결과 END

WHEN과 **THEN** 사이에 조건을 넣고, **THEN** 뒤에는 결과값 넣기

1. 같다는 조건: SELECT EMPNO, ENAME, JOB, SAL, CASE JOB WHEN 'MANAGER' THEN SAL * 1.1 WHEN 'SALESMAN' THEN SAL * 1.05 WHEN 'ANALYST' THEN SAL ELSE SAL * 1.03 END AS NEW_SAL FROM EMP;
2. 다른 조건: SELECT EMPNO, ENAME, COMM, CASE WHEN COMM IS NULL THEN '해당 사항 없음' WHEN COMM = 0 THEN '수당없음' WHEN COMM > 0 THEN '수당:' || COMM END AS COMM_CMT FROM EMP;

2. 다중행 함수

- 다중행 함수는 여러 행에 대한 연산을 수행하고 결과를 반환하는 함수
- 특정한 기준에 따라 여러 행을 그룹화하거나 필터링
- 집계 함수를 사용하여 그룹의 통계 또는 계산된 결과를 생성
- 데이터베이스의 효율적인 데이터 분석 및 집계를 위해 사용
- 종류 : SUM, COUNT, MAX, MIN, AVG

하나의 열에 출력 결과를 담는 함수

-- 한 번에 출력하려면 나오는 행의 개수가 같아야 한다.

※ 예러 발생: SELECT ENAME, SUM(SAL) FROM EMP;

why? GROUP BY를 통한 그룹화 작업이 이루어지지 않은 상태에서 SAL의 집계를 구하고자 하였기 때문

→ 수정: SELECT ENAME, SUM(SAL) AS total_salary FROM EMP GROUP BY ENAME;

cf. 다중행 함수를 사용할 땐 반드시 GROUP BY가 동반되어야 하는가? NO!

간단하게 이야기하자면, SUM, MAX, MIN, AVG를 구하기 위해서(=COUNT를 제외한 함수 모두)는 GROUP BY가 동반되어야 하지만, COUNT는 COUNT(*)로도 가능하다.

함수 단어의 뜻으로 생각해본다면, 총합과 최대값, 최솟값 그리고 평균을 구하기 위해서는 '전체 데이터 중' 이라는 전제가 필요하기 때문이다. 그렇다면 이 **다중행 함수(SUM, MAX, MIN, AVG)**를 산출하기 위한 데이터 그룹이 필요하기 때문에 이 경우에는 **GROUP BY**를 사용한다.

-- NULL 값을 자동으로 '제외'하고 계산한다.

```
SELECT SUM(COMM) FROM EMP;
```

-- 행의 개수가 같으므로 동시 출력이 가능

```
SELECT SUM(DISTINCT SAL), SUM(ALL SAL) FROM EMP;
```

-- SUM(ALL SAL)은 SUM(SAL)과 동일한 결과이므로 ALL을 생략하고 쓰는 것이 편하다.

```
SELECT COUNT(COMM) FROM EMP;
```

```
SELECT COUNT(COMM) FROM EMP WHERE COMM IS NOT NULL;
```

-- COUNT 함수도 자동으로 NULL값을 제외하고 계산

cf. 다중행함수에서 NULL값의 처리?

기본원칙: NULL값은 다중행 함수에서 (a) 무시되거나 (b) 특정 규칙에 따라 처리된다.

1. NULL 값이 무시되는 경우: **SUM(), AVG(), COUNT()**

= NULL 값은 '제외'하고, 유효한 값들만 (= NULL값이 아닌 데이터) 고려하여 계산을 수행한다.

2. NULL값이 처리되는 경우: **MAX(), MIN()**

= NULL 값을 가진 행은 무시되지 않고 처리

HOW? 처리방식

- MAX(): NULL 값이 가장 작다고 간주한다.
- MIN(): NULL 값이 가장 크다고 간주한다.

결론: MAX와 MIN 함수에서 NULL 값은 무시되지 않으나 극단값(최소 OR 최대)으로 인식되므로 결과에 영향을 미치지 않는다고 보아도 무방하다.

EMPNO	ENAME	SAL
1	John	1000
2	Jane	NULL
3	Alice	1500
4	Bob	2000

- $\text{SUM}(\text{SAL}) = 1000 + \text{NULL} + 1500 + 2000 = 4500$ (이때, NULL 값은 무시)
- $\text{AVG}(\text{SAL}) = (1000 + \text{NULL} + 1500 + 2000) / 3 = 1500$ (이때, NULL 값은 무시)
- $\text{COUNT}(\text{SAL}) = 3$ (이때, NULL 값은 무시)
- $\text{MAX}(\text{SAL}) = 2000$ (이때, NULL 값은 무시되지 않으나, 최소값으로 인식된다.)
- $\text{MIN}(\text{SAL}) = 1000$ (이때, NULL 값은 무시되지 않으나, 최대값으로 인식된다.)

▶ 그룹화(GROUP BY 절)

: 결과 값을 원하는 열로 묶어 출력, 그룹 별로 하나씩 결과를 출력

GROUP BY를 사용하지 않고, 이전에 배운 UNION을 사용하여 결과물을 출력할 수 있지만, 상당히 번거롭고 처리의 효율성을 위해 그룹화된 집계를 보고싶을 땐 '**GROUP BY**'를 사용

QUESTION. 부서 번호 별로 평균 급여 값 출력

-- CASE 1. 그룹화를 이용하지 않는다면?

-- SOL 1. 부서 번호 확인

```
SELECT DISTINCT DEPTNO FROM EMP;
```

-- SOL 2. 각 부서 번호에서 근무하는 직원 출력

```
SELECT * FROM EMP WHERE DEPTNO = 10;
```

```
SELECT * FROM EMP WHERE DEPTNO = 20;
```

```
SELECT * FROM EMP WHERE DEPTNO = 30;
```

-- SOL 3. 각 부서 번호에서 근무하는 직원들의 평균 급여 출력

```
SELECT AVG(SAL) FROM EMP WHERE DEPTNO = 10;
```

```
SELECT AVG(SAL) FROM EMP WHERE DEPTNO = 20;
```

```
SELECT AVG(SAL) FROM EMP WHERE DEPTNO = 30;
```

-- SOL 4. 결과를 한번에 출력하기 위해서는 세 개의 테이블을 합치기

```
SELECT 10 AS DEPTNO, AVG(SAL) FROM EMP WHERE DEPTNO = 10
```

```
UNION
```

```
SELECT 20 AS DEPTNO, AVG(SAL) FROM EMP WHERE DEPTNO = 20
```

```
UNION
```

```
SELECT 30 AS DEPTNO, AVG(SAL) FROM EMP WHERE DEPTNO = 30;
```

cf. 별칭은 _____ AS 사용할 별칭의 순서로 알고 있는데 저건 반대로 입력했는데 왜 출력이 되는거지?

이해하기 쉽게 가상의 정보를 기입해서 입력할 수 있다. 그럼 이 코드에서는 **DEPTNO = 10**를 만족하는 열에 10을 넣는데, 그게 무엇인지 모르니까 **DEPTNO**라고 표기를 하자

※ UNION을 사용할 경우 12줄의 작성이 필요했던 것과 달리 GROUP BY를 사용하면 한 줄로 정리할 수 있다.

-- CASE 2. GROUP BY절을 활용한 문제 풀기

```
SELECT DEPTNO, AVG(SAL) FROM EMP GROUP BY DEPTNO;
```

▷ 결과값을 정리해서 보고 싶다면?

```
SELECT DEPTNO, ROUND(AVG(SAL),2) FROM EMP GROUP BY DEPTNO ORDER BY DEPTNO;
```

ORDER BY? 오름차순/내림차순 정렬에 대한 표기 방식

```
SELECT DEPTNO, ROUND(AVG(SAL),2) FROM EMP GROUP BY DEPTNO ORDER BY DEPTNO ASC;
```

(ORDER BY의 디폴트값은 오름차순이므로 생략하여 작성해도 된다.)

```
SELECT DEPTNO, AVG(SAL) FROM EMP GROUP BY DEPTNO ORDER BY DEPTNO DESC;
```

(단, 내림차순으로 정렬하고 싶을 경우 반드시 DESC를 표기해야만 한다)

-- 2개의 열을 이용하여 GROUP BY

```
SELECT DEPTNO, JOB, ROUND(AVG(SAL),2) FROM EMP GROUP BY DEPTNO, JOB;
```

```
SELECT DEPTNO, JOB, ROUND(AVG(SAL),2) FROM EMP GROUP BY DEPTNO, JOB ORDER BY DEPTNO ASC, JOB DESC;
```

※ GROUP BY 사용 시 유의사항



```
35: :--<
36: -- GROUP BY에 사용하는 열 이름을 SELECT 절에서도 동일하게 사용<
37: -- GROUP BY에 사용하지 않은 열 이름을 SELECT 절에서 사용하게 되면 오류가 발생한다.<
38: SELECT DEPTNO, ENAME, AVG(SAL) FROM EMP GROUP BY ENAME;
```

ORA-00979: GROUP BY 표현식이 아닙니다.
00979, 00000 - "not a GROUP BY expression"
*Cause:
*Action:
38행, 8열에서 오류 발생

※ GROUP BY 절에 입력한 컬럼을 반드시 SELECT 절에 입력해줄 필요는 없다.

하지만, SELECT 절에 이용할 컬럼은 반드시 GROUP BY에 입력해주어야만 한다. ※

※ 그렇다면 왜 집계함수는 별도로 작성하지 않아도 되는 것일까? (난 이게 너무 궁금했다구!)

GROUP BY 절에서 집계 함수를 별도로 지정하지 않아도 되는 이유는, GROUP BY의 목적이 계산을 지정하는 것이 아니라 그룹을 정의하는 것이기 때문이다. 집계 함수는 각 그룹에 개별적으로 적용되고 각 그룹에 대해 하나의 결과를 반환한다. 따라서 GROUP BY 절에서 집계 함수를 반복할 필요가 없다.

즉, 집계함수는 이미 SELECT문에 작성할 때부터 이미 '그룹화'로 집계되어있기 때문에 구태어 GROUP BY에 작성하지 않아도 된다. (SELECT에 작성할 때부터 이미 GROUP BY에서 할 일을 다 했기 때문!)

아! 그렇다면 SELECT 절에 있는 것들을 GROUP BY에 작성하는 이유가 바로 SELECT 처리를 통해 테이블로 표기되어야 할 것들이 그룹화가 되어야하기 때문이구나!

▶ HAVING 절 ≈ WHERE 조건절

GROUP BY 절에서 조건을 사용할 수 있다. (GROUP BY ~ HAVING)

WHERE	HAVING
전체 테이블에서 조건을 만족하는 행을 출력 조회하고자 하는 테이블의 데이터 '전체'에서 조건 (예) 고객 데이터(=전체)에서 연령대가 20대(=조건)인 사람을 찾아!	그룹화된 결과에서 조건을 만족하는 행을 출력 GROUP BY로 묶인 공간에서 조건을 찾기 (예) 연령별(= 그룹화된 조건) 고객별로 카페 결제내역(= 그룹화 조건 내의 조건)을 찾아!

<WHERE과 HAVING 비교하기>

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
1	7369 SMITH	CLERK	7902	1980/12/17	800	(null)	20
2	7499 ALLEN	SALESMAN	7698	1981/02/20	1600	300	30
3	7521 WARD	SALESMAN	7698	1981/02/22	1250	500	30
4	7566 JONES	MANAGER	7839	1981/04/02	2975	(null)	20
5	7654 MARTIN	SALESMAN	7698	1981/09/28	1250	1400	30
6	7698 BLAKE	MANAGER	7839	1981/05/01	2850	(null)	30
7	7782 CLARK	MANAGER	7839	1981/06/09	2450	(null)	10
8	7788 SCOTT	ANALYST	7566	1987/04/19	3000	(null)	20
9	7839 KING	PRESIDENT	(null)	1981/11/17	5000	(null)	10
10	7844 TURNER	SALESMAN	7698	1981/09/08	1500	0	30
11	7876 ADAMS	CLERK	7788	1987/05/23	1100	(null)	20
12	7900 JAMES	CLERK	7698	1981/12/03	950	(null)	30
13	7902 FORD	ANALYST	7566	1981/12/03	3000	(null)	20
14	7934 MILLER	CLERK	7782	1982/01/23	1300	(null)	10

SELECT * FROM EMP; (전체 테이블 조회)

CASE 1. WHERE 조건절 사용

SELECT DEPTNO, JOB FROM EMP WHERE SAL >= 2000; 로 코드를 작성할 경우 총 6명의 데이터가 출력

DEPTNO	JOB
1	20 MANAGER
2	30 MANAGER
3	10 MANAGER
4	20 ANALYST
5	10 PRESIDENT
6	20 ANALYST

CASE 2. HAVING 조건절 사용

SELECT DEPTNO, JOB, AVG(SAL) FROM EMP GROUP BY DEPTNO, JOB HAVING AVG(SAL) >= 2000 ORDER BY DEPTNO ASC, JOB ASC;

DEPTNO	JOB
1	10 CLERK
2	10 MANAGER
3	10 PRESIDENT
4	20 ANALYST
5	20 CLERK
6	20 MANAGER
7	30 CLERK
8	30 MANAGER
9	30 SALESMAN

이 데이터를 기준으로 HAVING의 조건이 처리된다 (# 전체 데이터)

AVG(SAL)	DEPTNO	JOB
1300	10	CLERK
2450	10	MANAGER
5000	10	PRESIDENT
3000	20	ANALYST
950	20	CLERK
2975	20	MANAGER
950	30	CLERK
2850	30	MANAGER
1400	30	SALESMAN

그렇다면 총 5개의 데이터가 출력된다.

<결과물 출력 비교: 좌(WHERE) vs. 우(HAVING)>

DEPTNO	JOB	AVG(SAL)
1	10 MANAGER	2450
2	10 PRESIDENT	5000
3	20 ANALYST	3000
4	20 MANAGER	2975
5	30 MANAGER	2850

DEPTNO	JOB
1	20 MANAGER
2	30 MANAGER
3	10 MANAGER
4	20 ANALYST
5	10 PRESIDENT
6	20 ANALYST

VS.

▶ 그룹화와 관련된 여러 함수

(1) ROLLUP : 계층적인 총계를 내기 위해 사용하는 함수이다. (부분 및 전체집계 모두 사용 가능)

-- **ROLLUP**: 열의 개수 + 1 개의 결과 출력

-- 1) A + B = 부서번호 + 직책으로 GROUP BY

-- 2) A = 부서번호로 GROUP BY

-- 3) 전체를 GROUP BY

```
SELECT DEPTNO, JOB, COUNT(*), MAX(SAL), SUM(SAL), AVG(SAL) FROM EMP GROUP BY
```

ROLLUP(DEPTNO, JOB);

-- ROLLUP을 일부 열만 가지고 사용

-- 전체 GROUP BY만 제외하고 결과 출력 (ROLLUP 일부 사용의 결과)

```
SELECT DEPTNO, JOB, COUNT(*), MAX(SAL), SUM(SAL), AVG(SAL) FROM EMP GROUP BY DEPTNO,
```

ROLLUP(JOB);

총계는 생략 : Why? ROLLUP 함수의 인수로 포함되지 않은 열은 전체 데이터에 대한 총계를 생성하지 않기 때문

GROUP BY(좌) vs. ROLLUP(우) ※ ROLLUP 함수는 GROUP BY에 비해 총계 값이 하나 더 출력되어 생긴다.

ROLLUP 함수에 대한 정리

1. **ROLLUP** 함수는 **SELECT ~ FROM ~ GROUP BY ROLLUP(A , B) -> GROUP BY** 함수 안에 총계를 보고 싶은 데이터 앞에 **ROLLUP()**으로 묶기
 2. **ROLLUP** 함수는 그룹화된 집계값에 대한 '총계값'을 보여주는 함수이다. -> 그렇기에 **GROUP BY** 함수 뒤에서 실행!

(2) CUBE: 그룹화된 데이터에 대한 다차원 집계를 수행하기 위해 사용하는 함수

그룹화된 결과를 여러 차원에서 집계할 수 있으며, 각 차원의 조합에 따른 집계 결과를 반환한다.

※ **ROLLUP** 함수는 **GROUP BY**에 대한 전체 혹은 일부 집계를 산출할 수 있지만, **CUBE**는 **GROUP BY**에 포함된 모든 데이터를 집계하기 위함을 목적으로 한다. (일부만 집계한다고 해서 오류가 발생하는 것은 아니다.)

-- 1) A + B = 부서번호 + 직책으로 GROUP BY

-- 2) A = 부서번호로 GROUP BY

-- 3) B = 직책으로 GROUP BY

-- 4) 전체

```
SELECT DEPTNO, JOB, COUNT(*), MAX(SAL), SUM(SAL), AVG(SAL) FROM EMP GROUP BY
```

CUBE(DEPTNO, JOB);

※ CUBE 함수는 ROLLUP 함수보다 기본적으로 출력되는 결과물이 많다 (WHY? 그룹한 '모든' 값에 대한 집계를 산출하기 때문이다.)

(3) GROUPING SETS

: 열을 따로 그룹화하여 합계를 출력한다

-- 1) A = 부서번호로 GROUP BY

-- 2) B = 직책으로 GROUP BY

```
SELECT DEPTNO, JOB, COUNT(*), MAX(SAL), SUM(SAL), AVG(SAL) FROM EMP GROUP BY GROUPING  
SETS(DEPTNO, JOB);
```

(4) 그룹화 함수 ※ 그룹화된 데이터에 대한 추가적인 정보 제공

: 데이터 가공이나 연산 기능을 수행하지는 않으며, 그룹화 데이터의 식별을 쉽고 가독성을 높이기 위한 목적으로 사용된다.

GROUPING (열) - 0: 해당 열이 그룹화되었다 OR 1: 해당 열이 그룹화 되지 않았다

▶ **NULL**값으로 처리된 것에 1이 출력된다고 이해하면 된다. 그룹화가 되었을 경우에는 그 이름(여기서는 부서별번호나 직업)으로 묶여서 셀에 반영될테지만, **NULL**이라는 것은 그룹화된 이름이 없다는 의미이므로!

※ GROUPING 함수는 주로 ROLLUP이나 CUBE와 함께 사용되어 NULL값을 가지는 열에 대한 식별을 한다.

-- GROUPING

-- 1) A + B = 부서번호 + 직책으로 GROUP BY -> 0 0

-- 2) A = 부서번호로 GROUP BY

-- 3) B = 직책으로 GROUP BY -> 1 0

-- 4) 전체 -> 1 1

```
SELECT DEPTNO, JOB, COUNT(*), MAX(SAL), SUM(SAL), AVG(SAL), GROUPING(DEPTNO),
```

GROUPING(JOB) FROM EMP GROUP BY CUBE(DEPTNO, JOB);

GROUPING ID(): 그룹화 여부의 검사를 열 하나씩 지정하는 **GROUPING** 함수와 달리 여러 열을 지정할 수 있다.

GROUPING ID(A, B): 0 0 = 0, 0 1 = 1, 1 0 = 2, 1 1 = 3

-- GROUPING ID

-- 1) A + B = 부서번호 + 직책으로 GROUP BY -> 0 0 = 0

-- 2) A = 부서번호로 GROUP BY

-- 3) B = 직책으로 GROUP BY $\Rightarrow 1 \ 0 = 2$

$$-4) \text{ 저체} \rightarrow 1 \ 1 = 3$$

```
SELECT DEPTNO ,JOB ,COUNT(*) ,MAX(SAL) ,SUM(SAL) ,AVG(SAL) ,GROUPING(DEPTNO)
```

GROUPING(JOB), GROUPING_ID(DEPTNO, JOB) FROM EMP GROUP BY CUBE(DEPTNO, JOB);

-- DECODE문으로 비어있는 값을 채워보자

-- 비어있는 값 = 그룹화에 사용되지 않음 = GROUPING 함수의 값이 1이라는 의미

```
SELECT DECODE(GROUPING(DEPTNO), 1,'ALL_DEPT',DEPTNO)AS DEPTNO, DECODE(GROUPING(JOB),  
1, 'ALL_JOB', JOB) AS JOB, COUNT(*), MAX(SAL), AVG(SAL) FROM EMP GROUP BY CUBE(DEPTNO, JOB);
```

※ 그룹화함수 정리

1. 그룹화 함수는 '이미' 그룹화된 데이터에서 작업을 진행한다.
 2. 각 열에 대한 결과값을 '각각' 보고 싶다면 GROUPING, 정한 데이터를 '하나의 열'로 축약하여 보고 싶다면 GROUPING

(5) `HSTAGG()`: 특정 컬럼의 값을 그룹화하여 문자열로 '결합'하는 기능 (= 그룹 내 데이터를 가로로 나열)

-- LISTAGG 형식: **LISTAGG(합칠 컬럼명, 구분자) WITHIN GROUP(ORDER BY 정렬 컬럼명)**

-- 불서별로 사원 이름을 알파벳 순서대로 나열한 다음 이름과 이름 사이에 ''로 연결시켜 출력

```
SELECT DEPTNO, LISTAGG(ENAME, ',') WITHIN GROUP(ORDER BY ENAME ASC) AS ENAME FROM EMP  
GROUP BY DEPTNO;
```

DEPTNO	ENAME
1	CLARK, KING, MILLER
2	ADAMS, FORD, JONES, SCOTT, SMITH
3	ALLEN, BLAKE, JAMES, MARTIN, TURNER, WARD

-- 부서별로 사원 이름을 글여가 높은 순서대로 나열한 다음 이름과 이름 사이에 ','로 연결시켜 출력

```
SELECT DEPTNO, LISTAGG(ENAME, ', ') WITHIN GROUP (ORDER BY SAL DESC) AS ENAME FROM EMP  
GROUP BY DEPTNO;
```

DEPTNO	ENAME
1	KING, CLARK, MILLER
2	FORD, SCOTT, JONES, ADAMS, SMITH
3	BLAKE, ALLEN, TURNER, MARTIN, WARD, JAMES

(6) PIVOT(): 행을 열로 바꿔서 출력하는 함수로 집계 연산을 수행하여 행을 열로 변환하는데 유용하다.

행 데이터를 열로 변환함으로써 요약된 결과를 쉽게 얻을 수 있기 때문이다.

-- PIVOT: 행을 열로 바꿈

```
기본구조: SELECT <기준 열>, [변환된 열1] AS <열 이름1>, [변환된 열2] AS <열 이름2>, ...
FROM (<원본 테이블>) PIVOT(<집계 함수>(<집계 대상 열>) FOR <변환할 열> IN ( <열 값들> ))
```

-- 부서번호를 행에서 열로 바꿈

```
SELECT * FROM (SELECT DEPTNO, JOB, SAL FROM EMP) PIVOT(MAX(SAL) FOR DEPTNO IN (10, 20, 30))
ORDER BY JOB;
```

-- 직책을 행에서 열로 바꿈

```
SELECT * FROM (SELECT DEPTNO, JOB, SAL FROM EMP) PIVOT(MAX(SAL) FOR JOB IN ('ANALYST' AS ANALYST , 'CLERK' AS CLERK , 'MANAGER' AS MANAGER , 'PRESIDENT' AS PRESIDENT , 'SALESMAN' AS SALESMAN)) ORDER BY DEPTNO;
```

-- DECODE 문으로 PIVOT 테이블 만들어보기

-- 직책을 행에서 열로 바꿈

```
SELECT DEPTNO, MAX(DECODE(JOB, 'ANALYST', SAL)) AS ANALYST , MAX(DECODE(JOB, 'CLERK', SAL))
AS CLERK , MAX(DECODE(JOB, 'MANAGER', SAL)) AS MANAGER , MAX(DECODE(JOB, 'PRESIDENT',
SAL)) AS PRESIDENT , MAX(DECODE(JOB, 'SALESMAN', SAL)) AS SALESMAN FROM EMP GROUP BY
DEPTNO ORDER BY DEPTNO;
```

	DEPTNO	ANALYST	CLERK	MANAGER	PRESIDENT	SALESMAN
1	10	(null)	1300	2450	5000	(null)
2	20	3000	1100	2975	(null)	(null)
3	30	(null)	950	2850	(null)	1600

(7) UNPIVOT(): 열을 행으로 바꿔서 출력

-- UNPIVOT

-- PIVOT 테이블을 다시 UNPIVOT

```
SELECT * FROM (SELECT DEPTNO, MAX(DECODE(JOB, 'ANALYST', SAL)) AS ANALYST ,
MAX(DECODE(JOB, 'CLERK', SAL)) AS CLERK , MAX(DECODE(JOB, 'MANAGER', SAL)) AS MANAGER ,
MAX(DECODE(JOB, 'PRESIDENT', SAL)) AS PRESIDENT , MAX(DECODE(JOB, 'SALESMAN', SAL)) AS
SALESMAN FROM EMP GROUP BY DEPTNO ORDER BY DEPTNO) UNPIVOT (SAL FOR JOB IN (ANALYST,
CLERK, MANAGER, PRESIDENT, SALESMAN)) ORDER BY DEPTNO, JOB;
```

DEPTNO	JOB	SAL
1	10 CLERK	1300
2	10 MANAGER	2450
3	10 PRESIDENT	5000
4	20 ANALYST	3000
5	20 CLERK	1100
6	20 MANAGER	2975
7	30 CLERK	950
8	30 MANAGER	2850
9	30 SALESMAN	1600

(8) 조인: 여러 테이블을 하나의 테이블처럼 사용

기본 구문

- (a) 교차조인 : `SELECT * FROM Table1 CROSS JOIN Table2;`
- (b) 등가조인 : `SELECT * FROM Table1 JOIN Table2 ON Table1.column = Table2.column;`
- (c) 비등가조인 : `SELECT * FROM Table1 JOIN Table2 ON Table1.column1 > Table2.column2;`
- (d) 자체조인 : `SELECT * FROM Table1 AS t1 JOIN Table1 AS t2 ON t1.column = t2.column`
- (e) 외부조인 : `SELECT * FROM Table1 FULL / LEFT / RIGHT OUTER JOIN Table2 ON Table1.column = Table2.column;`

집합연산자(UNION, UNION ALL)	조인(JOIN)
세로(= 위아래)로 합쳐진다. 여러 쿼리의 결과를 결합하여 원하는 집합을 생성하거나 분석하는 데 사용	가로(= 좌우)로 합쳐진다. 여러 테이블의 데이터를 연결하여 필요한 정보를 검색하거나 분석하는 데 사용

여러 테이블을 사용할 때 FROM 절: FROM 절 뒤에는 테이블, 뷰, 서브쿼리 등이 올 수 있다.

▶ 교차 조인 = 크로스조인 = 모든 행의 조합

- 교차조인(Cross Join): SQL에서 가장 기본적인 조인 유형으로 첫 번째 테이블의 '모든 행'을 두 번째 테이블의 '모든 행'과 결합하여 가능한 모든 조합을 생성한다. '가능한 모든' 조합을 만들어내므로 성능의 문제가 발생할 수 있으며 일반적으로 사용하지 않는 유형의 조인이다. (존재한다 정도로만 속지)
교차 조인의 결과 집합 = 첫 번째 테이블의 행 수와 X 두 번째 테이블의 행 수를 곱한 값

IF) 이 두개의 테이블을 합치게 된다면? = $14 * 4 = 56$

```

1 -- EMP 테이블은 행이 14개
2 SELECT * FROM EMP;
3
43 -- DEPT 테이블은 행이 4개
44 SELECT * FROM DEPT;
45 -- 교차 조인 = 크로스조인 = 모든 행
46 SELECT * FROM EMP, DEPT;

```

DEPTNO	DNAME	LOC
1	ACCOUNTING	NEW YORK
2	RESEARCH	DALLAS
3	SALES	CHICAGO
4	OPERATIONS	BOSTON

`SELECT * FROM EMP, DEPT;`

	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO	DEPTNO_1	DNAME	LOC
1	7369 SMITH	CLERK	7902 1980/12/17	800	(null)	20		10 ACCOUNTING	10 ACCOUNTING	NEW YORK	
2	7369 SMITH	CLERK	7902 1980/12/17	800	(null)	20		20 RESEARCH	20 RESEARCH	DALLAS	
3	7369 SMITH	CLERK	7902 1980/12/17	800	(null)	20		30 SALES	30 SALES	CHICAGO	
4	7369 SMITH	CLERK	7902 1980/12/17	800	(null)	20		40 OPERATIONS	40 OPERATIONS	BOSTON	
5	7499 ALLEN	SALESMAN	7698 1981/02/20	1600	300	30		10 ACCOUNTING	10 ACCOUNTING	NEW YORK	
6	7499 ALLEN	SALESMAN	7698 1981/02/20	1600	300	30		20 RESEARCH	20 RESEARCH	DALLAS	
7	7499 ALLEN	SALESMAN	7698 1981/02/20	1600	300	30		30 SALES	30 SALES	CHICAGO	
8	7499 ALLEN	SALESMAN	7698 1981/02/20	1600	300	30		40 OPERATIONS	40 OPERATIONS	BOSTON	
9	7521 WARD	SALESMAN	7698 1981/02/22	1250	500	30		10 ACCOUNTING	10 ACCOUNTING	NEW YORK	
10	7521 WARD	SALESMAN	7698 1981/02/22	1250	500	30		20 RESEARCH	20 RESEARCH	DALLAS	
11	7521 WARD	SALESMAN	7698 1981/02/22	1250	500	30		30 SALES	30 SALES	CHICAGO	
12	7521 WARD	SALESMAN	7698 1981/02/22	1250	500	30		40 OPERATIONS	40 OPERATIONS	BOSTON	
13	7566 JONES	MANAGER	7839 1981/04/02	2975	(null)	20		10 ACCOUNTING	10 ACCOUNTING	NEW YORK	
14	7566 JONES	MANAGER	7839 1981/04/02	2975	(null)	20		20 RESEARCH	20 RESEARCH	DALLAS	

▶ 등가조인 = 내부조인 = 단순조인

- 두 테이블 간에 ‘공통된 값을 기준’으로 조인하는 방법으로, 주로 두 테이블 사이의 관계를 구축하는 데 사용되며 ‘공통된 값’은 기본키와 외래키의 관계를 활용하는 경우가 대다수이다.
- 테이블의 별칭 설정 가능하며, 열 이름을 명시할 때 테이블 이름이나 별칭으로 명시할 수 있다.
(※ 단, 조인하려는 테이블 간 공통으로 가지고 있는 열의 경우 반드시 테이블 출처를 명시해야 한다)
- 조인에 사용하는 테이블의 개수와 조건의 관계 : 테이블이 2개이면 최소 조건은 1개 필요하다.
(why? 둘 사이의 공통점을 찾아 ‘잇기’ 위해서)
(2개의 테이블 1개의 조건) → SELECT * FROM EMP A, DEPT B WHERE A.DEPTNO = B.DEPTNO;
(2개의 테이블 2개의 조건 ‘AND’로 잇기) → SELECT * FROM EMP A, DEPT B WHERE A.DEPTNO = B.DEPTNO AND A.SAL >= 3000;

(예) SELECT * FROM EMP, DEPT WHERE EMP.DEPTNO = DEPT.DEPTNO;

-- 하나의 테이블로 연결시켜 보고싶을 땐 공통으로 가지고 있는 키를 중심으로 테이블을 연결하기

EMP 테이블에서 부서번호와 DEPT 테이블에서 부서번호가 같은 것끼리 연결하라!

IF) 일치하지 않는 값이 발생할 경우 해당 조건은 조인을 하지 않는다. 오직 조인하려는 테이블끼리의 조건이 ‘일치’하는 것에서만 값이 추출되어 출력된다!

	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO	DEPTNO_1	DNAME	LOC
1	7782 CLARK	MANAGER	7839 1981/06/09	2450	(null)	10		10 ACCOUNTING	10 ACCOUNTING	NEW YORK	
2	7839 KING	PRESIDENT	(null)	1981/11/17	5000	(null)		10	10 ACCOUNTING	10 ACCOUNTING	NEW YORK
3	7934 MILLER	CLERK	7782	1982/01/23	1300	(null)		10	10 ACCOUNTING	10 ACCOUNTING	NEW YORK
4	7566 JONES	MANAGER	7839	1981/04/02	2975	(null)		20	20 RESEARCH	20 RESEARCH	DALLAS
5	7902 FORD	ANALYST	7566	1981/12/03	3000	(null)		20	20 RESEARCH	20 RESEARCH	DALLAS
6	7876 ADAMS	CLERK	7788	1987/05/23	1100	(null)		20	20 RESEARCH	20 RESEARCH	DALLAS
7	7369 SMITH	CLERK	7902	1980/12/17	800	(null)		20	20 RESEARCH	20 RESEARCH	DALLAS
8	7788 SCOTT	ANALYST	7566	1987/04/19	3000	(null)		20	20 RESEARCH	20 RESEARCH	DALLAS
9	7521 WARD	SALESMAN	7698	1981/02/22	1250	500		30	30 SALES	30 SALES	CHICAGO
10	7844 TURNER	SALESMAN	7698	1981/09/08	1500	0		30	30 SALES	30 SALES	CHICAGO
11	7499 ALLEN	SALESMAN	7698	1981/02/20	1600	300		30	30 SALES	30 SALES	CHICAGO
12	7900 JAMES	CLERK	7698	1981/12/03	950	(null)		30	30 SALES	30 SALES	CHICAGO
13	7698 BLAKE	MANAGER	7839	1981/05/01	2850	(null)		30	30 SALES	30 SALES	CHICAGO
14	7654 MARTIN	SALESMAN	7698	1981/09/28	1250	1400		30	30 SALES	30 SALES	CHICAGO

-- 테이블 별칭 사용 (이때 AS는 생략해도 된다)

SELECT * FROM EMP **(AS)** A, DEPT **(AS)** B WHERE A.DEPTNO = B.DEPTNO;

-- 다만, 열 이름을 구체화하여 작성하는 것이 좋다 (WHY? 실무에서 에러가 발생할 위험이 있으므로)

```
SELECT A.EMPNO, A.ENAME, A.JOB, A.MGR, A.HIREDATE, A.SAL, A.COMM, B. DEPTNO FROM EMP A,  
DEPT B WHERE A.DEPTNO = B.DEPTNO;
```

▶ 비등가 조인 : 등가 조인 이외의 방식 (BETWEEN A AND B)

- 등호만 사용할 수 있는 등가 조인과 달리 비교 연산자(>, <, >=, <=, <>)를 사용하여 두 테이블 간의 조인을 수행하는 방법 (= 두 테이블 사이에 일치하지 않는 값을 비교 OR 특정 조건을 만족하는 값)
- 등가조인보다 더 유연한 조인 방법이며, 특정 조건을 만족하는 행을 선택하기 위해 사용된다.

<예> BETWEEN A AND B 문장으로 조인

(테이블 1) SELECT * FROM EMP; + (테이블 2) SELECT * FROM SALGRADE;

→ 비등가 조인 : SELECT * FROM EMP A, SALGRADE B WHERE A.SAL BETWEEN B.LOSAL AND B.HISAL;

	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO	GRADE	LOSAL	HISAL
1	7369	SMITH	CLERK	7902	1980/12/17	800	(null)	20	1	700	1200
2	7499	ALLEN	SALESMAN	7698	1981/02/20	1600	300	30	2	1201	1400
3	7521	WARD	SALESMAN	7698	1981/02/22	1250	500	30	3	1401	2000
4	7566	JONES	MANAGER	7839	1981/04/02	2975	(null)	20	4	2001	3000
5	7654	MARTIN	SALESMAN	7698	1981/09/28	1250	1400	30	5	3001	9999
6	7698	BLAKE	MANAGER	7839	1981/05/01	2850	(null)	30			
7	7782	CLARK	MANAGER	7839	1981/06/09	2450	(null)	10			
8	7788	SCOTT	ANALYST	7566	1987/04/19	3000	(null)	20			
9	7839	KING	PRESIDENT	(null)	1981/11/17	5000	(null)	10			
10	7844	TURNER	SALESMAN	7698	1981/09/08	1500	0	30			
11	7876	ADAMS	CLERK	7788	1987/05/23	1100	(null)	20			
12	7900	JAMES	CLERK	7698	1981/12/03	950	(null)	30			
13	7902	FORD	ANALYST	7566	1981/12/03	3000	(null)	20			
14	7934	MILLER	CLERK	7782	1982/01/23	1300	(null)	10			

	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO	GRADE	LOSAL	HISAL
1	7369	SMITH	CLERK	7902	1980/12/17	800	(null)	20	1	700	1200
2	7900	JAMES	CLERK	7698	1981/12/03	950	(null)	30	1	700	1200
3	7876	ADAMS	CLERK	7788	1987/05/23	1100	(null)	20	1	700	1200
4	7521	WARD	SALESMAN	7698	1981/02/22	1250	500	30	2	1201	1400
5	7654	MARTIN	SALESMAN	7698	1981/09/28	1250	1400	30	2	1201	1400
6	7934	MILLER	CLERK	7782	1982/01/23	1300	(null)	10	2	1201	1400
7	7844	TURNER	SALESMAN	7698	1981/09/08	1500	0	30	3	1401	2000
8	7499	ALLEN	SALESMAN	7698	1981/02/20	1600	300	30	3	1401	2000
9	7782	CLARK	MANAGER	7839	1981/06/09	2450	(null)	10	4	2001	3000
10	7698	BLAKE	MANAGER	7839	1981/05/01	2850	(null)	30	4	2001	3000
11	7566	JONES	MANAGER	7839	1981/04/02	2975	(null)	20	4	2001	3000
12	7788	SCOTT	ANALYST	7566	1987/04/19	3000	(null)	20	4	2001	3000
13	7902	FORD	ANALYST	7566	1981/12/03	3000	(null)	20	4	2001	3000
14	7839	KING	PRESIDENT	(null)	1981/11/17	5000	(null)	10	5	3001	9999

<예2>

```
SELECT e.EmployeeName, s.Salary FROM Employees e JOIN Salaries s ON e.EmployeeID = s.EmployeeID
```

WHERE s.Salary > 5500;

: Employees 테이블과 Salaries 테이블에서 EmployeeID가 동일하고, Salary가 5500을 초과하는 데이터 출력

※ 비등가 조인할 때 유의해야 할 점

- (a) 열의 데이터 유형과 값의 일치성 확인 (만약 데이터 유형이 다르다면 형 변환 함수를 사용하여 일치시키기)
- (b) 조인 조건의 정확성
- (c) 조인 방향: 어느 테이블을 기준으로 수행하는가에 따라 결과가 달라질 수 있다.

▶ 자체조인: 같은 테이블끼리 조인

- 동일한 테이블을 조인하는 것을 의미 (why? 같은 테이블을 '복사'해서 사용할 경우, 데이터 용량이 2배가 되며 특정 데이터 작업이 한 번씩 더 발생하여 작업 효율이 2배 떨어지므로 이를 해결하기 위해)
- 자체 조인을 사용하는 이유
 - (a) 계층 구조: 자체 조인은 계층 구조를 가진 데이터를 다룰 때 용이하다. 예를 들어, 조직도나 부모-자식 관계를 가진 데이터 모델에서 자식 항목과 그에 해당하는 부모 항목을 연결할 때 자체 조인을 사용할 수 있다.
 - (b) 관계 분석: 동일한 테이블 내에서 직원과 관리자 간의 관계를 파악하거나, 상위-하위 관계를 가진 항목들을 분석할 때 자체 조인을 사용할 수 있다.
 - (c) 복잡한 쿼리 작성: 여러 조건을 조합하거나 복잡한 계산을 수행해야 하는 경우, 동일한 테이블을 자체 조인함으로 필요 데이터에 맞춰 조작할 수 있다.
- 자체 조인을 사용할 때 유의해야 할 점
 - (a) 별칭 사용: 자체 조인을 사용할 때에는 테이블에 별칭을 지정하여 구분 (why? 동일한 테이블을 조인하기 때문에 테이블 이름을 구분하기 위해)
 - (b) 조인 조건 설정: 자체 조인에서는 올바른 조인 조건을 설정
 - (c) 데이터 중복 문제: 자체 조인으로 인해 조인된 결과가 원래 테이블과 비교해 중복되는 경우가 있을 수 있으므로, 중복을 제거하기 위해 DISTINCT를 사용하거나 적절한 조건을 추가

<예>

```
SELECT * FROM EMP A, EMP B WHERE A.EMPNO = B.MGR;
```

```
SELECT A.EMPNO, A.ENAME, A.MGR AS MGR_EMPNO, B.ENAME AS MGR_ENAME FROM EMP A, EMP B  
WHERE A.MGR = B.EMPNO;
```

▶ 외부조인: 기준이 어디인가?

- 조인하는 두 개의 테이블에서 일치하는 값을 가지지 않는 경우에 사용하며, 조인 조건을 만족하지 않는 행도 결과에 포함시키는 조인방법이다. (= 기준이 되는 테이블의 행은 모두 다 나와야하며 & 기준이 되지 않는 테이블은 교집합만 붙는다.)
- 조인하는 테이블 중 하나가 다른 테이블과의 일치되는 값이 없는 경우에 특히 유용
- 외부조인의 종류
 - (a) Left Outer Join: 왼쪽 테이블의 모든 행과 오른쪽 테이블에서 일치하는 행을 가져오며, 일치하는 값이 없는 경우 오른쪽 테이블의 열은 **NULL**로 처리

<코드 예시> : SELECT A.EMPNO, A.ENAME, A.MGR AS MGR_EMPNO, B.ENAME AS MGR_ENAME

FROM EMP A, EMP B WHERE A.MGR = B.EMPNO (+);

(b) Right Outer Join: 오른쪽 테이블의 모든 행과 왼쪽 테이블에서 일치하는 행을 가져오며, 일치하는 값이 없는 경우 왼쪽 테이블의 열은 **NULL**로 처리

<코드 예시> : SELECT A.EMPNO, A.ENAME, A.MGR AS MGR_EMPNO, B.ENAME AS MGR_ENAME
FROM EMP A, EMP B WHERE A.MGR(+) = B.EMPNO;

(c) Full Outer Join: 왼쪽 테이블과 오른쪽 테이블의 모든 행을 가져오며, 일치하는 값이 없는 경우 **NULL**로 채워진다. 왼쪽과 오른쪽 테이블의 조인 조건을 만족하는 행이 없을 때 사용된다.

(9) 서브쿼리 : 쿼리 안에 또 다른 쿼리가 존재한다! (= 가상의 테이블을 만든다)

- 서브쿼리는 SELECT, INSERT, UPDATE, DELETE 문장 안에서 사용된다.
- (예) SELECT : 주로 FROM, WHERE, HAVING 절에서 사용
- 서브쿼리를 사용하는 이유? 자동화가 가능해지기 때문
 - (A) 외부 쿼리의 결과를 기반으로 보다 복잡한 조건을 만들어야 할 때
 - (B) 여러 테이블 간의 관계를 활용하여 원하는 데이터를 추출해야 할 때
 - (C) 데이터의 부분 집합을 사용하여 계산하거나 필터링 해야 할 때



▶ 단일행 서브쿼리 = 결과가 하나 = 행이 1개

서브쿼리의 결과가 하나의 행만 반환하는 경우에 사용한다.

주로 비교 연산자와 함께 사용되어 단일 값을 가져오거나 조건을 만족하는지 확인하는 데 사용

(단일행 서브쿼리 출력의 예)

(a) 부서별로 평균 연봉이 가장 높은 사원 찾기

```
SELECT employee_name FROM employees WHERE salary = (SELECT MAX(avg_salary) FROM
( SELECT AVG(salary) as avg_salary FROM employees GROUP BY department_id) AS subquery)
```

(b) 특정 주문보다 많은 수량을 가진 모든 제품 조회하기

```
SELECT product_name FROM products WHERE quantity > (SELECT quantity FROM products
WHERE order_id = '123');
```

▶ 다중행 서브쿼리 = 결과가 2개 이상 = 행이 2개 이상

서브쿼리의 결과가 여러 개의 행을 반환하는 경우에 사용

주로 IN, EXISTS 또는 FROM 절에서 사용되어 여러 값을 가져오거나 조건을 만족하는지 확인하는 데 사용

(다중행 서브쿼리 출력의 예)

- (a) '특정' 부서의 '모든' 사원 출력하기 -> 조건에 따른 출력물이 n개

```
SELECT employee_name FROM employees WHERE department_id IN (SELECT department_id  
FROM departments WHERE department_name = 'IT');
```

- (b) 주문 수량이 평균 주문 수량보다 많은(조건) 모든 주문 조회하기

```
SELECT order_id FROM orders WHERE EXISTS (SELECT order_id FROM order_items WHERE  
order_items.order_id = orders.order_id GROUP BY order_id HAVING SUM(quantity) > (SELECT  
AVG(quantity) FROM order_items));
```

▶ 다중 열 서브쿼리 = 복수 열 서브쿼리 (실무 사용 多)

서브쿼리의 결과로 여러 개의 열을 반환

일반적으로 메인 쿼리의 SELECT 절에서 하나 이상의 열 값을 비교하는 데 사용

(ex) **SELECT * FROM EMP WHERE (DEPTNO, SAL) IN (SELECT DEPTNO, MIN(SAL) FROM EMP GROUP
BY DEPTNO);**

→ 결국 이 코드의 의미는 부서별로 최저 월급을 받는 사원의 정보를 출력하라는 의미

※ 다중 열 서브쿼리의 장점?

- (a) 복잡한 쿼리 작성을 단순화하고 가독성을 향상시킬 수 있다 (단, 쿼리 성능에 유의)
 - (i) 이로 인해 복잡한 조인을 피할 수 있다는 이점이 존재
- (b) 추가 데이터 제공 가능
- (c) 유연성과 확장성 및 간결성

▶ 스칼라 서브쿼리 : SELECT 절 뒤에서 사용하는 서브쿼리로 열처럼 사용한다.

코드 : SELECT 컬럼 ... FROM 테이블 WHERE 컬럼N = (SELECT 단일_값 FROM 서브쿼리_테이블 WHERE
조건)

▶ IN 연산자 = OR 조건 여러개 = 합집합

```
SELECT * FROM EMP WHERE DEPTNO IN (10, 20, 30);
```

```
SELECT * FROM EMP WHERE DEPTNO IN (SELECT DISTINCT DEPTNO FROM EMP);
```

▶ ANY/SOME = 여러 결과 중에 하나라도 만족하면 TRUE

```
SELECT * FROM EMP WHERE SAL = ANY(SELECT MAX(SAL) FROM EMP GROUP BY DEPTNO);
```

```
SELECT * FROM EMP WHERE SAL = SOME(SELECT MAX(SAL) FROM EMP GROUP BY DEPTNO);
```

▶ ALL = 여러 결과 중에서 모두 만족해야 TRUE

-- = 조건 -> AND 조건 여러개이므로 사용 불가, (예) 한 사람이 하나의 급여를 가지고 있음

-- 부서별로 최대 급여 3개 값이 한 사람의 급여와 모두 같다는 조건을 설립할 수 없음

-- 작다는 (<) 조건 = AND 조건 여러개 (950 < 1250 < 1500 < 1600 < 2850)

-- 최소값보다 작은 급여를 가진 직원

```
SELECT * FROM EMP WHERE SAL < ALL (SELECT SAL FROM EMP WHERE DEPTNO = 30);
```

-- 단일행 서브쿼리로 재현 = 결과 동일

```
SELECT * FROM EMP WHERE SAL < (SELECT MIN(SAL) FROM EMP WHERE DEPTNO = 30);
```

-- 크다는 (>) 조건 = AND 조건 여러 개

-- 최대값보다 큰 값에 가진 직원

```
SELECT * FROM EMP WHERE SAL > ALL (SELECT SAL FROM EMP WHERE DEPTNO = 30);
```

-- 단일행 서브쿼리로 재현 - 결과 동일

```
SELECT * FROM EMP WHERE SAL > (SELECT MAX(SAL) FROM EMP WHERE DEPTNO = 30);
```

▶ EXISTS 연산자

- 서브쿼리의 결과가 존재하는지 여부를 확인하기 위해 사용되며, 논리적으로 참 또는 거짓 값을 반환하며, 주로 WHERE 절이나 HAVING 절에서 사용된다.
- EXISTS 연산자는 서브쿼리의 결과에 상관없이 존재 여부만을 확인하므로 서브쿼리의 실제 결과 집합은 필요하지 않는다.
- EXISTS 연산자에서 TRUE로 사용이 되면 앞에 SQL문이 정상적으로 실행된다.
- EXISTS 연산자에서 FALSE로 사용이 되면 앞에 SQL문이 정상적으로 실행되지 않는다 = 데이터 출력 無

(ex) **SELECT * FROM EMP WHERE EXISTS (SELECT DNAME FROM DEPT WHERE DEPTNO = 10);**

※ 유의 : EXISTS를 사용한 건 WHERE과 같은 조건문으로 해당 조건에 '총족'하는 것만 출력되는 것이 아니다.

→ 조건을 만족할 경우, 그 테이블을 전체 움직이는 것!

∴ DEPTNO = 10인 사람만 출력이 아니라, DEPTNO = 10은 존재하므로 TRUE값으로써 메인쿼리가 실행된다.

결과는 EMP 테이블의 모든 데이터가 출력되는 형태 = SELECT * FROM EMP;와 같은 결과값이 도출

▶ 인라인 뷰

서브쿼리를 사용하여 메인 쿼리 내에서 가상의 테이블처럼 사용하는 기능

인라인뷰는 데이터의 규모가 너무 크거나 작업에 불필요한 행과 열이 많을 때 사용하며, 복잡한 쿼리를 단순화하고 가독성을 향상시킬 수 있다.

```
SELECT E30.EMPNO, E30.ENAME, D.DEPTNO, D.DNAME FROM (SELECT * FROM EMP WHERE DEPTNO = 30) E30, (SELECT * FROM DEPT) D WHERE E30.DEPTNO = D.DEPTNO;
```

E30과 D를 별칭으로 쓰는 곳은 실질적인 테이블이 아님에도 해당 값을 하나의 테이블로 가정하고 코딩

▶ WITH절

임시로 사용할 수 있는 하나 이상의 쿼리 블록을 정의하는 데 사용되며 메인 쿼리에서 참조되는 일시적인 테이블 또는 뷰의 역할을 도맡는다.

구문 : WITH 식별자 AS (SELECT 컬럼1, 컬럼2, ... FROM 테이블명 WHERE 조건) SELECT * FROM 식별자;

(with절을 사용한 코드 예시)

```
WITH E30 AS(SELECT * FROM EMP WHERE DEPTNO = 30), D AS (SELECT * FROM DEPT) SELECT  
E30.EMPNO, E30.ENAME, D.DEPTNO, D.DNAME FROM (SELECT * FROM EMP WHERE DEPTNO = 30)  
E30, (SELECT * FROM DEPT) D WHERE E30.DEPTNO = D.DEPTNO;
```

(10) 데이터 조작어(※ DML은 혼자 사용할 수 있지만, TCL과 DDL은 한 묶음으로 사용해야 한다)

▶ DDL(Data Definition Language)

DDL은 데이터베이스 구조를 정의하거나 수정하는 데 사용되며, 각각의 문장은 데이터베이스 객체를 생성, 변경, 삭제하는 데 사용한다.

- (a) CREATE : 새로운 데이터베이스 객체(=테이블, 뷰, 인덱스 등)를 만드는 데 사용된다.

테이블 생성 코드 : CREATE TABLE 테이블이름 (열1 데이터타입, 열2 데이터타입, ...);

뷰 생성 코드 : CREATE VIEW 뷰이름 AS SELECT 열1, 열2, ... FROM 테이블이름 WHERE 조건;

인덱스 생성 코드 : CREATE INDEX 인덱스이름 ON 테이블이름 (열1, 열2, ...);

시퀀스 생성 코드 : CREATE SEQUENCE 시퀀스이름 START WITH 시작값 INCREMENT BY 증가값
[MINVALUE 최소값] [MAXVALUE 최대값] [CYCLE | NOCYCLE];

- (b) DROP : 데이터베이스 객체(=테이블, 뷰, 인덱스 등)를 삭제하는 데 사용되는 구문

테이블 삭제 코드 : DROP TABLE 테이블이름;

뷰 삭제 코드 : DROP VIEW 뷰이름;

인덱스 삭제 코드 : DROP INDEX 인덱스이름;

시퀀스 삭제 코드 : DROP SEQUENCE 시퀀스이름;

※ DROP의 유의점

- (1) 객체가 삭제되면 해당 객체에 저장된 데이터와 구조가 ‘영구적으로 삭제’되므로 신중해야 함
- (2) 삭제하려는 객체가 다른 객체에 의존성을 가지고 있는 경우, 의존 객체를 먼저 삭제

- (c) TRUNCATE : 테이블의 모든 데이터를 삭제하는 데 사용되는 구문

코드 : TRUNCATE TABLE 테이블이름 [REUSE STORAGE];

cf) SELECT vs. TRUNCATE: TRUNCATE는 테이블의 데이터를 행 단위로 삭제하는 대신, 전체 테이블의 데이터를 한 번에 삭제

※ TRUNCATE 유의점

- (1) TRUNCATE 문은 테이블의 데이터를 완전히 삭제하므로 룰백이 불가능하다. 데이터 복구를 위해서는 미리 백업을 수행해야만 한다.
- (2) TRUNCATE 문은 테이블의 데이터만 삭제하고, 테이블의 구조는 그대로 유지된다. 즉, 테이블의 정의, 제약 조건, 인덱스 등은 삭제되지 않는다.
- (3) TRUNCATE 문은 DELETE 문에 비해 빠르게 데이터를 삭제할 수 있다.
(why? 테이블의 데이터를 행 단위로 삭제하는 DELETE 문과는 달리, TRUNCATE 문이 테이블의 데이터 블록을 직접 해제하기 때문)

(d) **ALTER** : DB에 이미 존재하는 객체의 구조를 변경(객체 수정/속성 추가/변경/삭제)하는데 사용된다.

테이블 열 추가 : ALTER TABLE 테이블이름 ADD (열이름 데이터타입);

테이블 열 변경 : ALTER TABLE 테이블이름 MODIFY (열이름 새로운데이터타입);

테이블 열 삭제 : ALTER TABLE 테이블이름 DROP COLUMN 열이름;

테이블 제약 조건 추가 : ALTER TABLE 테이블이름 ADD CONSTRAINT 제약조건이름 제약조건내용;

인덱스 생성 : CREATE INDEX 인덱스이름 ON 테이블이름 (열이름);

인덱스 삭제 : DROP INDEX 인덱스이름;

(e) **RENAME** : 데이터베이스에서 객체의 이름을 변경하는 데 사용

기본 코드 : RENAME [객체 종류] [현재 이름] TO [새 이름];

▶ TCL(Transaction Control Language)

데이터베이스 트랜잭션의 제어를 위해 사용되는 언어로, DB 작업의 원자성, 일관성, 격리성, 지속성 관리 및 상태 제어에 중요한 역할을 한다.

(a) **COMMIT** : 데이터베이스 트랜잭션 작업을 ‘영구적으로’ 저장하는 데 사용 (코드: COMMIT)

COMMIT문을 실행하면 현재 트랜잭션에서 수행한 DB작업의 변경 사항이 디스크에 영구적으로 저장되어 다른 사용자나 세션에서도 변경사항을 확인할 수 있다. (이때 COMMIT문이 실행되면 현재 트랜잭션은 종료되며, 변경사항이 저장되어 이전의 DB 사항을 복구할 수 없다)

(b) **ROLLBACK** : 데이터베이스 트랜잭션 작업을 취소하고 이전 상태로 돌리는 데 사용 (코드 : ROLLBACK)

ROLLBACK문을 실행하면 현재 트랜잭션에서 수행한 모든 DB 작업이 취소되고 이전 상태로 돌아간다.
(= 이전에 수행한 작업은 DB에 영향을 주지 않는다.)

COMMIT과 마찬가지로 ROLLBACK문이 실행되면 현재 트랜잭션은 종료 및 초기화된다.

→ 주로 예기치 못한 상황이 발생하였을 때 ROLLBACK을 사용하여 DB의 상태를 원상복구 시킨다.

(c) **SAVEPOINT** : 데이터베이스 트랜잭션 내에서 롤백할 지점을 설정하는 데 사용

코드 : SAVEPOINT savepoint_name; -> 예시 : ROLLBACK TO SAVEPOINT savepoint_name;

SAVEPOINT는 해당 지점 이전의 작업을 롤백하여 트랜잭션을 특정 지점으로 되돌린다.

SAVEPOINT를 사용하는 경우?

(1) 부분적인 롤백: 트랜잭션 도중에 오류가 발생하거나 원하지 않는 결과가 나타난 경우

(=SAVEPOINT를 통해 특정 지점 이전(NOT 전체!)의 작업만을 롤백할 수 있다.)

(2) 중첩된 트랜잭션 관리 : 여러 SAVEPOINT를 설정하여 트랜잭션의 다양한 지점에서 작업을 롤백 혹은 커밋함으로써 세밀한 트랜잭션 제어가 가능해진다.

▶ DML(Data Manipulation Language)

데이터베이스에서 데이터를 조작하거나 검색하기 위해 사용되는 데이터 조작언어 (= 주로 사용하는 SQL)

※ 모든 구문의 기본 전제

1. 수정하려는 값의 데이터 유형 = 열의 데이터 유형
2. WHERE 절을 사용하여 특정 행만 업데이트 가능(WHERE 절을 생략하면 모든 행이 업데이트)
3. UPDATE 문은 한 번에 하나의 테이블만 수정 가능, 여러 테이블을 동시에 수정 시 조인(JOIN)

(a) **INSERT** : 테이블에 새로운 데이터를 입력할 때 사용되는 구문 (하나, N개의 데이터 삽입 가능)

코드 : INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);

(b) UPDATE : 데이터베이스 테이블의 기존 데이터를 수정하는 데 사용

코드 : UPDATE table_name SET column1 = value1, column2 = value2, ... WHERE condition;

(c) DELETE : 데이터베이스 테이블에서 특정 행이나 모든 행을 삭제하는 데 사용 (데이터 영구 삭제)

코드 : DELETE FROM table_name WHERE condition;

(d) SELECT : 데이터베이스 테이블에서 특정 데이터를 조회 시 사용

코드 : SELECT first_name, last_name FROM employees;