I. Join

1. 개념

두 개 이상의 테이블을 연결해서, 관련된 데이터를 하나로 합쳐서 조회하는 SQL 연산

- 여러 테이블에 나눠 저장된 정보를 연관된 값(주로 Foreign Key)을 기준으로 묶어 보여줌
- JOIN을 하면 마치 하나의 큰 표처럼 데이터를 다룰 수 있음
- 일반적으로 부모테이블과 자식테이블을 연결해서 조회하는 경우가 많음
- 부모테이블의 PK와 자식테이블의 FK의 컬럼의 값이 같은 행끼리 join

2. Source VS Target

source table	target table
Join시 조회 기준이 되는 테이블로서 main 정보를 가지고 있다. 기준 테이블, 먼저 읽는 테이블 (FROM에 위치)	Join시 source table의 추가정보를 가지고 있는 테이블로서 sub 정보를 가지고 있다. 연결 대상 테이블 (JOIN 뒤에 오는 테이블)

```
SELECT e.name, d.dept_name
FROM emp e                -- source
JOIN dept d               -- target
ON e.dept_id = d.dept_id;
```

- **비교 : 부모/자식 table**

구분	부모 테이블	자식 테이블
설명	참조되는 테이블 (Primary Key를 갖고 있음)	참조되는 테이블 (Primary Key를 갖고 있음)
예시	`dept` 테이블 (부서 정보)	`emp` 테이블 (직원이 부서를 참조함)

```
-- 예: emp 테이블의 dept_id가 dept 테이블의 dept_id를 참조함
CONSTRAINT fk_emp_dept
FOREIGN KEY (dept_id) REFERENCES dept(dept_id);
```

- **정리 비교표**

구분	부모/자식 테이블	소스/타겟 테이블

사용 맥락	데이터베이스 **설계**, 외래키 설정 등	**데이터 조회 (JOIN)**할 때
관계 기준	외래키(FK) → 기본키(PK)	`FROM` 기준 테이블 → `JOIN` 대상 테이블
예시	`emp(dept_id)` → `dept(dept_id)`	`FROM emp` → `JOIN dept`

- **한 줄 요약**

> **부모/자식**: 데이터 **구조 관계** (정의 시 사용)
> **소스/타겟**: 데이터 **조회 관계** (SELECT / JOIN 시 사용)

cf) Foreign key VS ERD VS Join

구분	항목	FOREIGN KEY	JOIN
개념 비교	개념	두 테이블의 **관계를 정의**하는 제약 조건	두 테이블의 **데이터를 함께 조회**하는 SQL 연산
	역할	참조 무결성 유지 (없는 부서를 참조하지 못하게 함)	연결된 테이블의 데이터를 가져오기
	사용 시점	**테이블 생성 시** 사용	**데이터 조회 시(SELECT)** 사용
	관계	FOREIGN KEY 없이도 JOIN 가능하지만, 있으면 더 안전함	FOREIGN KEY 없어도 JOIN은 가능
ERD와의 관계	역할	테이블 간 **관계 구조 정의**	관계 있는 테이블을 **데이터로 연결**
	기반	PK와 FK로 관계 설정	보통 FK로 연결된 테이블을 기준으로 함
	직접적 관계	❌ 직접 JOIN은 수행하지 않음	✅ ERD에 정의된 관계를 이용함
	간접적 힌트 제공	✅ JOIN이 어떻게 이루어져야 하는지 힌트를 줌	✅ ERD의 관계를 따라 작성함

3. 종류

(1) INNER JOIN

- **조인 연산 조건을 만족하는 행끼리만 합치는 JOIN**

```sql
SELECT e.emp_id, e.emp_name, e.hire_date, d.dept_name
FROM emp e INNER JOIN dept d ON e.dept_name = d.dept_name;

-- inner 생략 가능
-- inner join의 경우, source와 target의 순서를 바꿔도 출력되는 표는 같다.
-- select 뒤에 오는 컬럼의 순서대로 표가 출력되기 때문
```

(2) OUTER JOIN

- **조인 연산 조건을 만족하지 않는 행들도 합치는 JOIN**
- **조인조건을 만족하는 행이 없는 경우 NULL을 합친다.**
- **outer 생략 가능**
- **어떤 테이블에 있는 정보를 볼 것인지 명확하게 알려줘야한다.**
- **즉, 누가 source인지를 알려주어, source의 내용을 모두 조회하는 것.**
- **구문상 source가 어디인지(왼쪽/오른쪽/둘 다)에 따라 left/right/full outer join으로 나뉨**

```sql
-- 모든 직원의 id(emp.emp_id), 이름(emp.emp_name), 부서_id(emp.dept_id)를 조회하는
-- 부서_id가 80 인 직원들은 부서명(dept.dept_name)과 부서위치(dept.loc) 도 같이 출력한

SELCET  e.emp_id, e.emp_name, e.dept_id, d.dept_name, d.loc
FROM    emp e LEFT JOIN dept d ON e.dept_id = d.dept_id AND d.dept_id=80;

-- 부서_id가 80이 아닌 직원들도 조회는 되어야하므로 emp가 source table
-- 1st. source table로 부터 부른 컬럼은 다 보여준다.
-- 2nd. target table로 부터 부른 컬럼에 빈값은 null로 채워 맞춘다.
-- 3rd. d.dept_id = 80 조건에 맞는 값들만 살려두고, 나머지의 target table로 부터 부른

```

(3) SELF JOIN

- **물리적으로 하나의 테이블을 두개의 테이블처럼 조인하는 것**

```sql
-- 직원 ID가 101인 직원의 직원의 ID(emp.emp_id), 이름(emp.emp_name),
-- 상사이름(emp.emp_name)을 조회

SELECT  e.emp_id, e.emp_name, m.emp_name AS "manager name"
FROM    emp e JOIN emp m ON e.mgr_id = m.emp_id;

-- emp에는 상사 컬럼이 직접적으로 없지만, 두개의 emp를 보면 알 수 있다.
-- 두개의 emp 테이블을 각각 e, m 으로 별칭을 적어 어떤 테이블의 컬럼을 줄세우는지 봐야한다
```

II. SUBQUERY

1. 개념

쿼리 안에서 select 쿼리를 사용하는 것.

- main query & sub query
- select절, from절, where절, having절에서 사용

2. 구분

(1) Scalar Subquery & Inline View

어느 구절에서 사용되었는지에 따른 구분

1) Scalar Subquery

- SELECT절, WHERE절, HAVING절에 사용되는 서브쿼리로, 반드시 1행 1열(값 하나)만 반환해야 함.
- 0행이 조회되면 null을 반환

```
SELECT emp_name,
       (SELECT dept_name
        FROM dept
        WHERE dept_id = emp.dept_id) AS 부서이름
FROM emp;
-- 이때 (SELECT dept_name ...)은 스칼라 서브쿼리
-- 직원 한 명당 "부서 이름 하나(1행 1열)"이 출력됨
```

2) Inline View

- FROM 절에 들어가는 서브쿼리, 마치 가상의 테이블처럼 사용
- 결과는 여러 행과 여러 열 가능 (즉, 테이블처럼 사용)
- 반드시 as 별칭을 붙여야 함

```
-- 부서직원들의 평균이 전체 직원의 평균(emp.salary) 이상인 부서의 이름(dept.dept_name)
-- 평균급여는 소숫점 2자리까지 나오고 통화표시($)와 단위 구분자 출력
SELECT dept_id,
       dept_name,
       CONCAT('$', FORMAT(평균급여, 2)) AS "평균급여"
FROM(SELECT d.dept_id, d.dept_name, AVG(e.salary) AS "평균급여"
     FROM dept d JOIN emp e ON d.dept_id = e.dept_id
     GROUP BY d.dept_id, d.dept_name
     HAVING avg(salary) >= (SELECT avg(salary) FROM emp)
     ORDER BY 3 DESC
    ) AS t;

```

- 서브쿼리

	dept_id	dept_name	평균급여
▶	90	Executive	19333.333333
	110	Accounting	10154.000000
	70	Public Relations	10000.000000
	20	Marketing	9500.000000
	80	Sales	8960.606061
	100	Finance	8601.333333

- 메인쿼리

	dept_id	dept_name	평균급여
▶	90	Executive	$19,333.33
	110	Accounting	$10,154.00
	70	Public Relations	$10,000.00
	20	Marketing	$9,500.00
	80	Sales	$8,960.61
	100	Finance	$8,601.33

(2) 단일행 서브쿼리 & 다중행 서브쿼리

1) 단일행 서브쿼리

- 서브쿼리 조회 결과, 행이 하나인 것

2) 다중행 서브쿼리

- 서브쿼리 조회 결과, 행이 여러개인 것
- where절 에서의 연산자
  - `in`
  - `any` : 조회된 값들 중 하나만 참이면 참 (where 컬럼 > any(서브쿼리))
  - `all` : 조회된 값들 모두와 참이면 참 (where 컬럼 > all(서브쿼리))

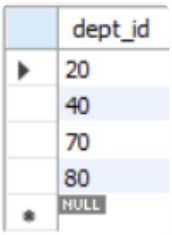
```
--  부서 위치(dept.loc) 가 'New York'인 부서에 소속된 직원의 ID(emp.emp_id),
-- 이름(emp.emp_name), 부서_id(emp.dept_id) 를 sub query를 이용해 조회.

SELECT emp_id, emp_name, dept_id
FROM emp
WHERE dept_id IN (SELECT dept_id FROM dept WHERE loc='New York');

-- 서브쿼리 where절의 비교 컬럼(loc)이 unique가 아닐 경우
-- 결과행이 여러개일 수 있다. 다중행 서브쿼리 연산!

```

- 서브쿼리



	dept_id
▶	20
	40
	70
	80
*	NULL

- 메인쿼리

	emp_id	emp_name	dept_id
▶	201	Michael	20
	202	Pat	20
	203	Susan	40
	204	Hermann	70
	145	John	80
	146	Karen	80

(3) 비상관 서브쿼리 & 상관 서브쿼리

1) 비상관 서브쿼리

- 서브쿼리에 메인쿼리의 컬럼이 사용되지 않는다.
- 메인쿼리에 사용할 값을 서브쿼리가 제공하는 역할을 한다.

2) 상관 서브쿼리

- 서브쿼리에서 메인쿼리의 컬럼을 사용한다.
- 메인쿼리가 먼저 수행되어 읽혀진 데이터를 서브쿼리에서 조건이 맞는지 확인하고자 할때 주로 사용한다.
- 메인 쿼리의 각 행마다 where의 subquery가 조회 대상인지 검사하면서 실행된다. 이때 현재 검사중인 그 행의 컬럼값을 subquery에서 사용한다.

3) `EXISTS`, `NOT EXISTS` 연산자 (상관쿼리와 같이 사용된다)

- 서브쿼리의 결과를 만족하는 값이 존재하는지 여부를 확인하는 조건.
- 조건을 만족하는 행이 여러개라도 한 행만 있으면 더이상 검색하지 않는다. 있는지 없는지만 알려주지, 몇 번 있는지는 안알려준다.
- 보통 데이터테이블의 값이 이력테이블(Transaction TB)에 있는지 여부를 조회할 때 사용된다.
    - 메인쿼리: 데이터테이블
    - 서브쿼리: 이력테이블
    - 메인쿼리에서 조회할 행이 서브쿼리의 테이블에 있는지(또는 없는지) 확인

```sql
-- 직원이 한명이상 있는 부서의 부서ID(dept.dept_id)와 이름
(dept.dept_name), 위치(dept.loc)를 조회

SELECT d.dept_id, d.dept_name, d.loc
FROM dept d
WHERE EXISTS (SELECT * FROM emp e WHERE e.dept_id = d.dept_id);

-- e.dept_id = d.dept_id 조건
-- 직원의 부서id가 있다는 건 그 부서에 직원이 있다는 의미
```

III. DML

DDL vs DML 차이 요약

항목	DDL (Data Definition Language)	DML (Data Manipulation Language)
목적	데이터 구조 정의	데이터 조작(삽입, 수정, 삭제 등)
대상	테이블, 뷰, 인덱스 등 스키마 객체	테이블 안의 행(레코드)
주요 명령어	`CREATE`, `ALTER`, `DROP`, `TRUNCATE`	`SELECT`, `INSERT`, `UPDATE`, `DELETE`
커밋 요부	일부는 자동 커밋됨	직접 `COMMIT` 또는 `ROLLBACK` 필요
원상복구	`ROLLBACK` 불가 (대부분 영구 반영됨)	`ROLLBACK` 가능 (트랜잭션 기반)

질문	DDL	DML
테이블 만들거나 구조 수정할 때?	✅ 사용	❌ 아님
데이터를 추가하거나 삭제할 때?	❌ 아님	✅ 사용
커밋 없이 실행 후 되돌릴 수 있어?	❌ (일반적으로 불가능)	✅ (트랜잭션 지원)

(1) INSERT

행을 넣는 것

- 한 행씩 처리가능. 한번에 여러행 못 넣는다.
- 문자열의 경우 삽입할 값을 ''로 감싸준다. 큰따옴표는 nope
- 날짜는 형태에 맞게 문자열로 넣어준다.(날짜 -, 시간: 로 구분)
- 결측치 값은 null 입력
- 테이블의 모든 컬럼에 데이터를 넣을 경우 컬럼 항목 생략 가능

```sql
-- [구문]
-- INSERT INTO 테이블명 (컬럼, 컬럼...) VALUES (값, 값...)

INSERT INTO DEPTARTMENT (DEPARTMENT_ID, DEPARTMENT_NAME, LOCATION)
VALUES (100, '기획부', '서울')
-- values 넣을 땐 대소문자 구별!
```

(2) UPDATE

테이블의 컬럼의 값을 수정

- UPDATE: 변경할 테이블 지정
- SET: 변경할 컬럼과 값을 지정
- WHERE: 변경할 행을 선택.

```sql
-- [구문]
-- UPDATE 테이블명
-- SET    변경할 컬럼 = 변경할 값, 변경할 컬럼 = 변경할 값...
-- WHERE 제약조건

--  IT 부서의 직원들의 급여를 3배 인상
UPDATE emp
SET    salary = salary * 3
WHERE  dept_id IN (SELECT dept_id FROM dept WHERE dept_name = 'IT');
```

(3) DELETE

테이블의 행을 삭제

- 전체 행 삭제
- WHERE: 삭제할 행을 선택

```sql
-- [구문 ]
-- DELETE FROM 테이블명 [WHERE 제약조건]

-- comm_pct 가 null이고 job_id 가 IT_PROG인 직원들을 삭제
DELETE FROM emp WHERE comm_pct IS NULL AND job_id = 'IT_PROG';
```