SQL 2025.09.10

챕터 1: SQL 기본 문법

1. 데이터 조회: `SELECT`

`SELECT` 는 데이터베이스에서 원하는 데이터를 조회할 때 사용하는 가장 기본적인 명령

- **모든 컬럼 조회**: `SELECT * FROM 테이블명;`
- **특정 컬럼 조회**: `SELECT 컬럼명1, 컬럼명2 FROM 테이블명;`
  - **Tip**: 모든 컬럼을 의미하는 `*` 대신, 필요한 컬럼명을 직접 나열하면 가독성이 높아지고 성능에도 유리함. 다른 개발자들이 열람 시 어떤 테이블에서 데이터를 조회하거나 옮기는걸 보고 뭐하는지 쉽게 파악함.

2. 조건 검색: `WHERE`

`WHERE` 절을 사용하여 특정 조건을 만족하는 데이터만 필터링

- **문자열, 숫자 비교**:
  - `select * from emp where deptno = 10;` (숫자)
  - `select * from emp where job = 'manager';` (문자열은 작은따옴표로 감싼다.)
- **비교 연산자**: `>`, `<`, `>=`, `<=`, `!=` 또는 `<>`
  - `select * from emp where sal > 3000;`
- **복합 조건** (**`AND`**, **`OR`**, **`NOT`**): 여러 조건을 결합할 때 사용
  - `AND` 는 모든 조건을 만족할 때 사용합니다. (`AND` 가 `OR` 보다 우선순위가 높다)
    - `select * from emp where sal > 2500 and job = 'manager';`
  - `OR` 는 둘 중 하나라도 만족할 때 사용
  - `NOT` 은 조건을 부정할 때 사용
- **우선순위 조정**: 괄호 `()` 를 사용하여 연산 순서를 임의로 조정 가능
  - 예시

    ```
    SELECT * FROM employees
    WHERE (department_id = 90 OR department_id = 100)
    AND salary >= 10000;
    ```

- **`IN` 연산자**: `OR` 조건을 간결하게 표현할 때 사용(효과가 같다)
  - `select * from emp where deptno in (10, 20, 30);`
  - `NOT IN` 은 해당 목록에 포함되지 않는 데이터를 찾음
    - `select * from emp where deptno not in (10, 40);`

챕터 2: 데이터 범위 및 패턴 검색

1. 범위 검색: `BETWEEN`

특정 범위 내의 데이터를 조회할 때 사용하며, 시작 값과 끝 값을 포함

- `select * from emp where sal between 2300 and 3000;`
- 날짜 범위도 지정
  - `select * from emp where hiredate between '1981-01-01' and '1981-12-31';`

2. 패턴 검색: `LIKE`

문자열 패턴을 기반으로 데이터를 검색

- **와일드카드 문자**:
  - `%` : 0개 이상의 모든 문자
  - `_` : 정확히 한 개의 문자
- **예시**:
  - `select * from emp where ename like 'F%';` (이름이 'F'로 시작하는 모든 사람)
  - `select * from emp where ename like '_o%';` (이름의 두 번째 글자가 'o'인 사람)
  - `select * from emp where hiredate like '1981%';` (1981년에 입사한 모든 사람)

챕터 3: NULL 값과 정렬

1. NULL 값 처리

`NULL` 은 '값이 없다'는 의미이며, 일반적인 연산자(`=`, `>`)로는 비교할 수 없다

- **NULL 여부 확인**:
  - `select * from emp where comm is null;` (커미션이 NULL인 데이터)
  - `select * from emp where comm is not null;` (커미션이 NULL이 아닌 데이터)
- **NULL 연산의 문제점**: `NULL` 과 다른 값을 연산하면 결과는 항상 `NULL`
  - `select ename, sal + comm from emp;` (커미션이 `NULL` 인 경우 결과도 `NULL`)
- **`NULL` 처리 함수**:
  - **`IFNULL(컬럼, 대체값)`** : 컬럼의 값이 `NULL` 이면 대체값으로 바꾸어 연산
    - `select ename, sal + IFNULL(comm, 0) from emp;`
  - **`COALESCE(컬럼1, 컬럼2, ...)`** : 여러 컬럼 중 **첫 번째로 `NULL` 이** 아닌 값을 반환
    - `select ename, coalesce(comm, job, '몰라') from emp;`

2. 데이터 정렬: `ORDER BY`

조회된 결과를 특정 컬럼을 기준으로 정렬

- **기본 구문**: `ORDER BY 컬럼 [ASC|DESC];`
  - **`ASC` (Ascending)**: 오름차순 (기본값)
  - **`DESC` (Descending)**: 내림차순
  - `order by ename desc;` (이름을 내림차순으로 정렬)
- **여러 컬럼 정렬**: 쉼표로 구분하여 정렬 순서를 지정
  - `order by deptno, sal desc;` (부서 번호로 1차 정렬, 동일 부서 내에서는 급여를 내림차순으로 2차 정렬)
- **컬럼 순서 번호로 정렬**: `SELECT` 절의 순서에 따라 컬럼 번호로 정렬
  - `order by 3;` (세 번째 컬럼을 기준으로 정렬)

챕터 4: 유용한 SQL 함수

1. 문자열 함수

- **`CONCAT(문자열1, 문자열2, ...)`** : 여러 문자열을 하나로 결합
  - `SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM employees;`
- **`UPPER()` / `LOWER()`** : 문자열을 대문자 또는 소문자로 변환
  - `select * from emp where upper(job) = 'MANAGER';`
- **`TRIM()`** : 문자열의 앞뒤 공백을 제거
- **`LPAD()`** : 특정 문자를 채워 문자열 길이를 맞춤

2. 데이터 제한: `LIMIT`

조회 결과의 개수를 제한하여 페이징 처리(많은 양 한번에 x , 정해진 개수 만큼 보여줘 성능향상 및 사용자경험에 도움) 에 유용

- `ORDER BY 연봉 LIMIT 5;` (연봉 순으로 정렬한 뒤 상위 5개만 조회)
- **오프셋 (`OFFSET`)**: 특정 위치부터 데이터를 가져옴
  - `SELECT * FROM employees LIMIT 10 OFFSET 20;` (21번째 데이터부터 10개를 가져옴)

정규화: 하나의 테이블에 데이터 중복이 발생하는 것을 방지하기 위해 여러 개의 테이블로 나누어 설계하는 과정을 의미

챕터 5: 날짜 함수와 그룹 함수

1. 날짜 함수

날짜 함수는 날짜 및 시간 데이터를 조작하고 포맷팅할 때 사용

- **`DATE_ADD()`** : 특정 날짜에 지정된 시간 간격을 더함
  - **구문**: `DATE_ADD(날짜, INTERVAL 값 단위)`
  - **예시**:

    ```
    SELECT DATE_ADD(NOW(), INTERVAL 1 DAY);
    ```

    👉 현재 날짜(`NOW()`)에 1일을 더한 결과를 반환
- **`DATE_FORMAT()`** : 날짜 데이터를 원하는 형식의 문자열로 변환
  - **구문**: `DATE_FORMAT(날짜, '형식')`
  - **예시**:

    ```
    SELECT DATE_FORMAT(hiredate, '%Y년 %m월 %d일') FROM emp;
    ```

    👉 `hiredate` 컬럼의 날짜를 '2025년 09월 10일' 형식

2. 그룹 함수와 단일행 함수

SQL 함수는 크게 두 가지

- **단일행 함수: 레코드(행)당 하나씩** 적용되어, 입력된 행의 수만큼 결과 나옴 (`CONCAT`, `UPPER`, `DATE_FORMAT` 등)
- **그룹 함수: 전체 레코드 또는 그룹별로** 하나의 통계값을 계산 (`COUNT`, `SUM`, `AVG` 등)
- ⚠ **주의: 그룹 함수와 일반 컬럼은 함께 사용할 수 없다**
  - **오류 예시**: `SELECT COUNT(ename), ename FROM emp;`
  - **이유**: `COUNT(ename)` 는 전체 행의 개수를 반환하는 반면, `ename` 은 각 행의 값을 반환하기 때문에 모순이 발생
- **`COUNT()`** : `NULL` 이 아닌 값의 개수를 센다
  - `select count(ename) from emp;`
  - `select count(*) from emp;`
  - **Tip**: `COUNT(컬럼명)` 은 해당 컬럼에 `NULL` 값이 있으면 제외하고 세지만, `COUNT(*)` 는 모든 행의 개수를 세므로 더 정확한 전체 행 수를 파악할 때 유용
- **`SUM()`** : 숫자 컬럼의 합계를 구함
  - `select sum(sal) from emp;`
- **`AVG()`** : 숫자 컬럼의 평균을 구함
  - `select avg(sal) from emp;`

3. 데이터 그룹화: `GROUP BY`

`GROUP BY` 를 사용하면 특정 컬럼의 값을 기준으로 데이터를 묶어서 그룹 함수를 적용 가능

- **구문**: `SELECT 컬럼, 그룹함수 FROM 테이블 GROUP BY 그룹화할_컬럼;`
- **예시**:
  - 부서별 평균 급여

    ```
    select deptno, avg(sal) from emp
    group by deptno;
    ```

  - 부서 및 직무별 평균 급여

    ```
    select deptno, job, avg(sal) from emp
    group by deptno, job
    order by 1, 2;
    ```

  - **Tip**: `GROUP BY` 절에 명시된 컬럼들은 `SELECT` 절에도 포함되어야 한다.

4. 그룹에 조건 적용: `HAVING`

`GROUP BY` 로 그룹화된 결과에 대해 조건을 적용할 때 `HAVING` 을 사용합니다. `WHERE` 는 전체 데이터에 조건을 적용하는 반면, `HAVING` 은 그룹화된 결과에 조건을 적용

- **구문**: `SELECT ... GROUP BY ... HAVING 조건;`
- **예시**: 10번 부서를 제외하고, 부서 및 직무별 평균 급여가 3000 미만인 데이터만 조회

  ```
  select deptno, job, avg(sal) 평균급여 from emp
  where deptno != 10
  group by deptno, job
  having 평균급여 < 3000
  order by 1, 2;
  ```

  - **`WHERE`** : 먼저 전체 데이터에서 `deptno` 가 10번이 아닌 데이터를 걸러낸다
    (전체 기준에서 걸러내므로 where가 쓰임)
  - **`GROUP BY`** : 필터링된 데이터를 부서와 직무별로 그룹화
  - **`HAVING`** : 그룹화된 결과 중에서 `평균급여` 가 3000 미만인 그룹만 최종적으로 선택
    (그룹핑 이후 그 안에서 조건으로 걸러내므로 having)

챕터 6: 테이블 관계와 JOIN의 개념

1. 테이블 관계의 필요성

데이터베이스를 설계할 때, 모든 데이터를 하나의 거대한 테이블에 저장하면 데이터 중복이 발생하고 관리하기 어렵다(정규화). 따라서 **관련 있는 데이터끼리 테이블을 나누고, 서로 관계를 맺어주는** 것이 효율적

- **외래 키 (Foreign Key): 한 테이블의 컬럼이 다른 테이블의 기본 키 (Primary Key)를 참조하여 테이블 간의 관계를 연결
  - 예시: `emp` 테이블의 `deptno` (외래 키)가 `dept` 테이블의 `deptno` (기본 키)를 참조. 이를 통해 사원 정보를 기반으로 부서 정보를 찾는다.
- **셀프 조인 (Self Join)**: 한 테이블이 자기 자신을 참조하는 관계
  - 예시: `emp` 테이블의 `mgr` (매니저) 컬럼은 다른 사원의 `empno` 를 참조. 이를 통해 사원의 정보를 이용해 해당 사원의 매니저 정보를 찾을 수 있다.

데이터가 여러 테이블에 분산되어 있을 때, **JOIN**을 사용하면 이 테이블들을 결합하여 원하는 정보를 한 번에 조회할 수 있다.

2. JOIN의 종류

JOIN은 여러 테이블을 결합하는 방법이며, 결합 조건에 따라 여러 종류

- **크로스 조인 (Cross Join)**: 조인 조건 없이 두 테이블의 모든 행을 1:1로 결합하여 가능한 모든 조합(경우의 수)을 만듦. 결과는 두 테이블의 행 수를 곱한 만큼 생성(OR조건 비슷)
  - `select ename, sal, dname from emp, dept;`
- **이너 조인 (Inner Join)**: 가장 흔히 사용되는 조인으로, **조인 조건이 일치하는 행들만** 결합하여 반환(AND조건 비슷)
  - `select e.ename, e.sal, d.deptno, d.dname from emp e, dept d where e.deptno = d.deptno;`
  - ⚠ 만약 40번 부서에 사원이 없거나, 사원에 부서 정보가 없는 경우 해당 행은 결과에 포함되지 않는다.
- **아우터 조인 (Outer Join)**: 조인 조건에 일치하지 않더라도 한쪽 테이블의 모든 행을 포함하여 결과를 보여줌
- **셀프 조인 (Self Join)**: 동일한 테이블을 마치 다른 테이블처럼 두 번 사용하여 자기 자신과 조인하는 방법

3. SQL JOIN 문법

조인 문법은 크게 **FROM 절에 조인 조건을 명시하는 ANSI 문법**과 **`WHERE` 절에 조건을 명시하는 비(非) ANSI** 문법으로 나뉨. 현대 SQL에서는 가독성이 좋은 ANSI 문법을 권장

- **테이블 별칭 (Alias)**
  - 테이블 이름이 길거나 여러 테이블을 조인할 때, 각 테이블에 짧은 별칭을 지정하여 가독성을 높임
  - `select e.ename, d.dname from emp e, dept d;` (`emp` 테이블을 `e` 로, `dept` 테이블을 `d` 로 별칭 지정)
  - ⚠ 조인하는 테이블에 **이름이 같은 컬럼**이 있을 경우, `테이블명.컬럼명` 또는 `별칭.컬럼명` 으로 명확히 구분해야 오류가 발생하지 않는다.
- **ANSI JOIN 문법**
  - **`JOIN ... ON ...`** : 가장 일반적인 조인 문법. `JOIN` 키워드 뒤에 테이블을 명시하고, `ON` 절에 조인 조건을 명시
    - 예시

      ```
      SELECT e.ename, e.sal, d.deptno, d.dname
      FROM emp e JOIN dept d
      ON e.deptno = d.deptno;
      ```

  - **`NATURAL JOIN`** : 두 테이블에서 **이름이 같고 데이터 타입이 같은 컬럼**을 찾아 자동으로 조인
    - 예시

      ```
      SELECT e.ename, e.sal, d.deptno, d.dname
      FROM emp e NATURAL JOIN dept d;
      ```

    - **Tip**: 편리하지만, 의도치 않은 컬럼으로 조인될 수 있어 권장하지 않는다.
  - **`JOIN ... USING (...)`** : 두 테이블 간에 이름이 같은 조인 컬럼을 명확하게 지정
    - 예시

      ```
      SELECT e.ename, e.sal, d.deptno, d.dname
      FROM emp e JOIN dept d
      USING(deptno);
      ```

4. JOIN과 기타 조건문

조인 조건 외에 추가적인 조건을 적용하려면 `WHERE` 절을 사용. `WHERE` 절은 `JOIN` 연산이 완료된 후에 필터링을 수행

- **예시**: 20번 부서에 속한 사원의 정보와 부서 정보를 조인하여 조회 SQL

```
SELECT e.ename, e.sal, d.deptno, d.dname
FROM emp e JOIN dept d
ON e.deptno = d.deptno
WHERE e.deptno = 20;
```