PostgreSQL 문법 정리1 - 기초, SELECT, GROUP BY 정리

SQL 분류

- SQL문은 DDL, DML, DCL(TCL)로 분류한다.
- DDL(Data Definition Language, 데이터 정의어)
  - 데이터베이스를 정의하는 언어이며, 데이터리를 생성, 수정, 삭제하는 등의 데이터의 전체의 골격을 결정하는 역할을 하는 언어 이다. 데이터베이스, 테이블등을 생성하는 역할을 한다.
  - CREATE, ALTER, DROP, TRUNCATE 등이 있다.
- DML (Data Manipulation Language, 데이터 조작어)
  - 데이터베이스 사용자가 응용 프로그램이나 질의어를 통하여 저장된 데이터를 실질적으로 처리하는데 사용하는 언어 이다.
  - 데이터베이스 사용자와 데이터베이스 관리 시스템 간의 인터페이스를 제공한다.
  - SELECT, INSERT, UPDATE, DELETE 등이 있다.
- DCL(Data Control Language, 데이터 제어어)
  - 데이터를 제어하는 언어 이다.
  - 데이터의 보안, 무결성, 회복, 권한 등을 정의하는데 사용한다
  - TCL (Transaction Control Language, 트랜잭션 제어 언어)과 DCL로 구분하기도 한다.
  - GRANT, REVOKE, COMMIT, ROLLBACK 등이 있다
- SQL 예약어를 대문자로 표시하고 끝에 세미콜론(선택사항)을 붙이면 코드 가독성이 좋아진다.

SELECT

- 테이블의 데이터를 조회하는 명령어
- SELECT 컬럼명 FROM 테이블명;
- 컬럼명을 *으로 쓰면 모든 컬럼을 조회한다는 의미, 컬럼명들 사이에 쉼표를 넣으면 여러 개의 컬럼을 조회할 수 있음

- DISTINCT: 중복된 데이터를 제외하고 조회할 때 사용, 어떤 컬럼에서 고유한 값을 찾고 싶을 때 유용하다.
- SELECT DISTINCT 컬럼명 FROM 테이블명;
- SELECT DISTINCT(컬럼명) FROM 테이블명; DISTINCT 바로 뒤에 컬럼명에 괄호를 씌워서 사용할 수도 있다. 괄호를 쓰나 안쓰나 결과는 같다.
- COUNT 함수: 특정 쿼리 조건에 맞는 입력 행의 개수를 구하는데 사용한다.
  - 함수이기 때문에 괄호를 사용해야 한다.
  - 특정 컬럼을 지정하거나 COUNT(*)로 모든 컬럼을 지정할 수 있다. 둘 다 결과는 동일하다.
  - SELECT COUNT(컬럼) FROM 테이블명;
  - 단순히 테이블의 행 개수를 세어서 반환하기 때문에 다른 명령어와 함께 사용하는 경우가 많다. 특히 DISTINCT와 많이 사용된다.
  - SELECT COUNT(DISTINCT 컬럼명) FROM 테이블명;

- WHERE 문: 컬럼에 조건을 지정하여 그에 맞는 행이 반환 되도록 한다.
  - 조건에 맞는 필터 기능을 수행한다.
  - FROM 절 바로 뒤에 위치한다
  - SELECT column1, column2 FROM table WHERE conditions;
  - SELECT name, choice FROM table WHERE name = 'David' AND choice = 'RED'
- ORDER BY 문: 조회된 데이터를 기준에 맞게 정렬해준다.
  - SELECT column1, column2, ... FROM table ORDER BY column_1 ASC / DESC
  - ASC: 오름차순, DESC: 내림차순, 지정하지 않으면 보통 ASC를 사용한다
  - 보통 SQL문 가장 끝에 위치하지만 LIMIT 보다는 앞에 위치한다.
  - ORDER BY를 여러 컬럼으로 지정하여 사용할 수도 있다.
  - SELECT store_id, first_name, last_name FROM customer ORDER BY store_id ASC, first_name DESC;
- LIMIT: 쿼리로 반환 되는 행의 개수를 제한하는 명령어
  - 쿼리 요청의 가장 아래로 내려가며 가장 마지막에 실행된다.
  - SELECT * FROM table LIMIT 5;
  - SELECT customer_id FROM payment ORDER BY payment_date ASC limit 10;
- BETWEEN 연산자: 값을 값 범위와 비교할 때 사용
  - WHERE문에 조건을 더하기 위해 많이 사용된다.
  - value ≥ low AND value ≤ high
  - value BETWEEN low AND high 위와 같은 의미 이다. (low이상, high이하)
  - value < low or value > high
  - value NOT BETWEEN low AND high NOT을 이용한 반대 범위 지정(low 미달, high 초과)
  - SELECT COUNT(*) FROM film WHERE rating = 'R' AND replacement_cost BETWEEN 5 AND 15;
  - BETWEEN으로 date 범위를 필터링하는 경우 시간에 유의해야 한다.
  - BETWEEN '2007-02-01' AND '2007-02-14' 일 때 실제 시간 범위는 2007년 2월 1일 0시 ~ 2007년 2월 14일 0시 이다. 따라서 2월 14일 10시 같은 데이터는 조회 되지 않는다.
- IN 연산자: 어떤 값이 목록에 포함됐는지 확인할 때 사용한다.
  - SELECT color FROM table WHERE color IN ('red', 'blue');
  - SELECT color FROM table WHERE color NOT IN ('red', 'blue'); - NOT 연산자 사용
  - SELECT * FROM payment where amount not in (0.99, 1.98, 1.99);
- LIKE, ILIKE: 문자열 데이터에 대한 패턴 매칭을 수행하기 위한 명령어이다. LIKE는 대소문자를 구분하지만 ILIKE는 구분하지 않는다.
  - 와일드 카드 연산자를 사용할 수 있다.
  - 대문자 'A'로 시작하는 이름: WHERE name LIKE 'A%'
  - 소문자 'a'로 끝나는 이름: WHERE name LIKE '%a'
  - select * from customer where first_name like 'J%' and last_name ilike 's%';
  - 밑줄을 이용하면 문자 하나만 교체할 수 있다.
  - WHERE title LIKE 'Mission Impossible _'
  - 연속으로 쓸 수도 있다. WHERE value LIKE 'Version#__'
  - 조합해서 사용: WHERE name LIKE '_her%'

GROUP BY

- 데이터가 카테고리 별로 어떻게 분포되어 있는지 파악하기 위해 데이터를 집계하고 함수를 적용하는 SQL문이다.
- 밑의 표에서 A카테고리 값의 평균(9)이나 개수(2개)를 구해야 할 때 GROUP BY절이 유용하다.카테고리값

A	10
A	8
B	6
B	12

- 집계함수의 종류
  - AVG(): SELECT ROUND(AVG(replacement_cost), 2) FROM film;
  - COUNT(): SELECT COUNT(*) FROM film;
  - MAX(): SELECT MAX(replacement_cost) FROM film;
  - MIN(): SELECT MIN(replacement_cost) FROM film;
  - SUM(): SELECT SUM(replacement_cost) FROM film;
- SELECT category_col, AGG(data_col) FROM table GROUP BY category_col;
- GROUP BY절은 FROM문 바로 뒤 또는 WHERE문 바로 뒤에 위치해야 한다.
- SELECT문에서 특정 컬럼을 조회한다면 그 컬럼이 GROUP BY절에 포함되야 한다. 단 집계함수는 GROUP BY절에 포함되지 않아도 된다.
- SELECT company, division, SUM(sales) FROM finance_table GROUP BY company, division;
- WHERE절에는 집계함수를 쓸 수 없다. 대신 HAVING을 사용할 수 있다.
- 집계를 기반하여 정렬하려면 전체 함수를 참조해야 한다.
- SELECT company, SUM(sales) FROM finance_table GROUP BY company ORDER BY SUM(sales);
- 가장 많은 금액을 사용한 고객ID는?
- SELECT customer_id, SUM(amount) FROM payment GROUP BY customer_id ORDER BY SUM(amount) DESC;
- 결제 날짜 별(DATE함수 사용) 결제 금액 합계를 구하고, 합계 내림차순으로 정렬하기
- SELECT DATE(payment_date), SUM(amount) FROM payment GROUP BY DATE(payment_date) ORDER BY SUM(amount) DESC;
- 직원id별로 처리한 결제 건수는 몇 건인가?
- SELECT count(*), staff_id FROM payment group by staff_id;
- 영화 등급별 평균 교체 비용은?
- SELECT rating, ROUND(AVG(replacement_cost), 2) FROM film GROUP BY rating;
- 결제 금액 합계 기준 가장 많이 지출한 고객 id 5개 찾기
- SELECT customer_id, SUM(amount) FROM payment GROUP BY customer_id ORDER BY SUM(amount) DESC LIMIT 5;

Having 절

- HAVING절은 GROUP BY절로 선택된 그룹에 대한 탐색 조건을 지정한다.
- HAVING절은 집계가 이미 수행된 이후에 자료를 필터링하기 때문에 GROUP BY 호출 뒤에 위치한다.

- SELECT company, SUM(sales) FROM finance_table WHERE company != 'Google' GROUP BY company HAVING SUM(sales) > 1000;
- WHERE 필터를 적용하고 나서 GROUP BY를 호출한 후에 HAVING 절이 적용되어 판매액 총액이 1000보다 큰 값을 조건으로 다시 필터링 된다.
- 결제 거래 건수가 40건 이상인 고객 ID 찾기
- SELECT customer_id, COUNT() FROM payment GROUP BY customer_id HAVING COUNT() >= 40
- 직원 ID 2번과의 거래 중 100을 초과하여 사용한 고객의 ID 찾기
- SELECT customer_id, SUM(amount) FROM payment WHERE staff_id = 2 GROUP BY customer_id HAVING SUM(amount) > 100

SQL문 기초 연습문제

1. ID가 2인 직원에게서 최소 110달러를 쓴 고객의 ID는?

```
SELECT customer_id, SUM(amount)
FROM payment WHERE staff_id = 2
GROUP BY customer_id
HAVING SUM(amount) > 110;
```

2. J로 시작하는 영화는 몇 개인가?

```
SELECT COUNT(*)
FROM film
WHERE title LIKE 'J%';
```

3. 이름이 'E'로 시작하는 동시에 주소 ID가 500미만인 고객 중, ID 번호가 가장 높은 고객은?

```
SELECT first_name, last_name
FROM customer
WHERE first_name LIKE 'E%' AND address_id < 500
ORDER BY customer_id DESC
LIMIT 1;
```

4. 2012년 9월의 예약 건수 총합을 facid 별로 집계

```
SELECT facid, SUM(slots) AS total_slots
FROM cd.bookings
WHERE starttime BETWEEN '2012-09-01' AND '2012-10-01'
GROUP BY facid
ORDER BY SUM(slots)
```

5. facid 별로 예약 건수 총합이 1000건 이상인 것만 조회하고, facid 오름차순으로 정렬

```
SELECT facid, SUM(slots) AS total_slots
FROM cd.bookings
GROUP BY facid
HAVING SUM(slots) > 1000
ORDER BY facid
```

PostgreSQL 문법 정리2 - AS, JOIN, UNION

AS

- AS문은 열이나 결과에 별칭(alias)을 부여한다.
- SELECT column AS new_name FROM table
- SELECT SUM(column) AS new_name FROM table → 컬럼명이 new_name으로 출력된다
- AS연산자는 쿼리의 맨 마지막에 실행되기 때문에 WHERE, GROUP BY 호출, HAVING절 등에서는 쓸 수 없다.
- 정상 쿼리
- SELECT customer_id, sum(amount) as total_spent from payment group by customer_id having sum(amount) > 100
- 실행되지 않는 쿼리, 컬럼이 없다고 뜨면서 실행되지 않는다. AS는 맨 마지막에 실행되기 때문에 alias된 컬럼명을 인식하지 못한다.
- SELECT customer_id, sum(amount) as total_spent from payment group by customer_id having total_spent > 100
- 정상 쿼리
- SELECT customer_id, amount as new_name from payment where amount > 2
- 실행되지 않는 쿼리
- SELECT customer_id, amount as new_name from payment where new_name > 2

JOIN

- 여러 테이블의 레코드를 결합하여 하나의 열로 표현하는 것
- 결합된 테이블 중 하나에만 존재하는 정보를 처리하는 방식에 따라 분류한다.
- 하나의 쿼리에서 연달아 여러 개의 JOIN을 적용할 수 있다.
- ANSI 표준 SQL은 다음 다섯 가지 유형의 JOIN을 규정한다.
- INNER
- FULL OUTER
- LEFT OUTER
- RIGHT OUTER
- CROSS

※ 예제 데이터 테이블, 이름은 모두 고유한 이름이라고 가정.

- 테이블 명 : Registrations

reg_id	name
1	Andrew
2	Bob
3	Charlie
4	David

테이블명 : Logins

log_id	name
1	Xavier
2	Andrew
3	Yolanda
4	Bob

INNER JOIN

- 두 테이블 모두에서 일치하는 레코드를 조회하는 것
- 대칭적이기 때문에 테이블 순서를 바꿔도 결과는 같다.
- SELECT * FROM TableA INNER JOIN TableB ON TableA.col_match = TableB.col_match;
- SELECT * FROM TableB INNER JOIN TableA ON TableB.col_match = TableA.col_match;
- PostgreSQL에서는 그냥 JOIN이라고 쓰면 INNER JOIN으로 인식한다.

```
SELECT * FROM Registrations
INNER JOIN Logins
ON Registrations.name = Logins.name;
```

Results

reg_id	name	log_id	name
1	Andrew	2	Andrew
3	Bob	4	Bob

- 양쪽에서 조회하려는 컬럼명이 같은 경우 어떤 테이블에 있는 컬럼인지 명시해줘야 한다.

```
SELECT payment_id, payment.customer_id, first_name
FROM payment
INNER JOIN customer
ON payment.customer_id = customer.customer_id
```

FULL OUTER JOIN

- 테이블A와 B의 가능한 모든 레코드를 조합한다.
- 서로 일치하는 데이터가 없으면 null로 표시된다.
- 대칭적이기 때문에 테이블 순서를 바꿔도 결과는 같다.

```
SELECT * FROM TableA FULL OUTER JOIN TableB ON TableA.col_match = TableB.col_match;
SELECT * FROM Registrations
FULL OUTER JOIN Logins
ON Registrations.name = Logins.name;
```

Results

reg_id	name	log_id	name
1	Andrew	2	Andrew
2	Bob	4	Bob
3	Charlie	null	null

4 David null null
null null 1 Xavier
null null 3 Yolanda

- FULL OUTER JOIN에 WHERE문을 활용하여 양쪽 테이블에서 고유한 값을 조회할 수 있다.
- 대칭적이기 때문에 테이블 순서를 바꿔도 결과는 같다.
- INNER JOIN의 정반대인 결과가 나온다.

```
SELECT * FROM TableA FULL OUTER JOIN TableB ON TableA.col_match = TableB.col_match
WHERE TableA.id IS null OR TableB.id IS null;
SELECT * FROM Registrations
FULL OUTER JOIN Logins
ON Registrations.name = Logins.name
WHERE Registrations.reg_id IS null OR Logins.log_id IS null
```

Results

reg_id	name	log_id	name
3	Charlie	null	null
4	David	null	null
null	null	1	Xavier
null	null	3	Yolanda

LEFT OUTER JOIN (LEFT JOIN)

- 왼쪽 테이블에 있는 레코드 세트를 출력하고 오른쪽 테이블에 일치하는 데이터가 없으면 null로 출력한다.

```
SELECT * FROM TableA LEFT OUTER JOIN TableB ON TableA.col_match = TableB.col_match;
SELECT * FROM Registrations
LEFT OUTER JOIN Logins
ON Registrations.name = Logins.name
```

Results

reg_id	name	log_id	name
1	Andrew	2	Andrew
2	Bob	4	Bob
3	Charlie	null	null
4	David	null	null

- 테이블A에만 있고 B에는 없는 행을 구하기 위해 LEFT OUTER JOIN을 이용하면서 WHERE절 조건을 활용하는 방법

- 대칭적이지 않기 때문에 테이블 순서가 중요하다.

```
SELECT * FROM TableA LEFT OUTER JOIN TableB ON TableA.col_match = TableB.col_match
WHERE TableB.id IS NULL
SELECT * FROM Registrations
LEFT OUTER JOIN Logins
ON Registrations.name = Logins.name
WHERE Logins.log_id IS NULL
```

Results

reg_id	name	log_id	name
3	Charlie	null	null
4	David	null	null

RIGHT OUTER JOIN (RIGHT JOIN)

- 기본적으로 LEFT OUTER JOIN과 완전히 동일하지만 테이블이 서로 바뀐다는 점만 다르다.
- LEFT JOIN을 쓰면서 테이블 순서만 바꿔줘도 동일한 결과를 얻을 수 있다.
- SELECT * FROM TableA RIGHT OUTER JOIN TableB ON TableA.col_match = TableB.col_match;
- 테이블B에만 있는 데이터를 조회하기 위해 LEFT JOIN과 마찬가지로 WHERE를 활용할 수 있다.

```
SELECT * FROM TableA RIGHT OUTER JOIN TableB ON TableA.col_match = TableB.col_match
WHERE TableA.id IS null;
```

SELF JOIN

- 동일 테이블 사이의 조인, 테이블을 구별하기 위해 서로 다른 별칭을 사용해야 한다.

```
SELECT tableA.col, tableB.col
FROM table AS tableA
JOIN table AS tableB
ON tableA.som_col = tableB.other_col
```

```
-- 같은 영화 테이블 안에서, 영화 별로 상영시간이 같은 영화의 목록 구하기
SELECT f1.title, f2.title, f1.length
FROM film AS f1
INNER JOIN film AS f2
ON f1.film_id != f2.film_id AND f1.length = f2.length
```

JOIN 연습문제

1. 캘리포니아에 살고 있는 고객의 이메일은 무엇인가?

```
SELECT district, email
FROM address
INNER JOIN customer ON address.address_id = customer.address_id
WHERE district = 'California'
```

2. 닉 월버그(Nick Wahlberg)라는 배우가 출연한 모든 영화 목록 찾기

```
SELECT title, first_name, last_name
FROM film_actor
INNER JOIN actor ON film_actor.actor_id = actor.actor_id
INNER JOIN film ON film_actor.film_id = film.film_id
WHERE first_name = 'Nick' AND last_name = 'Wahlberg'
```

3. 테니스 코트(facid 0, 1)의 2012년 9월 21일의 예약 시작시간 목록을 구하고 시작시간 오름차순으로 정렬하라

```
SELECT cd.bookings.starttime, cd.facilities.name
FROM cd.facilities
INNER JOIN cd.bookings ON cd.facilities.facid = cd.bookings.facid
WHERE cd.facilities.facid IN (0, 1)
AND cd.bookings.starttime >= '2012-09-21'
AND cd.bookings.starttime <= '2012-09-22'
ORDER BY cd.bookings.starttime
```

4. David Farrell이란 회원의 예약 시작 시간을 구하라

```
SELECT cd.bookings.starttime
FROM cd.bookings
INNER JOIN cd.members
ON cd.bookings.memid = cd.members.memid
WHERE cd.members.firstname = 'David' AND surname = 'Farrell'
```

PostgreSQL 문법 정리3 - CREATE, INSERT, UPDATE, DELETE, ALTER

CREATE

- 테이블을 생성하는 명령어

```
-- 기본 문법
CREATE TABLE table_name(
   column_name TYPE column_constraint,
   column_name TYPE column_constraint,
   table_constraint table_cnostraint
) INHERITS existing_table_name;

-- 더 단순한 문법
CREATE TABLE table_name(
   column_name TYPE column_constraint,
   column_name TYPE column_constraint,
);
```

```
-- 예시: players 테이블 생성, player_id 컬럼은 serial pk
CREATE TABLE players(
   player_id SERIAL PRIMARY KEY,
   age SMALLINT NOT NULL
);
```

```
CREATE TABLE account(
      user_id SERIAl PRIMARY KEY,
      username VARCHAR(50) UNIQUE NOT NULL,
      password VARCHAR(50) NOT NULL,
      email VARCHAR(250) UNIQUE NOT NULL,
      created_on TIMESTAMP NOT NULL,
      last_login TIMESTAMP
)
```

```
CREATE TABLE job(
      job_id SERIAL PRIMARY KEY,
      job_name VARCHAR(200) UNIQUE NOT NULL
);
```

```
CREATE TABLE account_job(
      user_id INTEGER REFERENCES account(user_id),
      job_id INTEGER REFERENCES job(job_id),
      hired_date TIMESTAMP
)
```

INSERT

- 테이블에 데이터를 삽입하는 명령어

```
-- 기본 문법
INSERT INTO table (column1, column2, ...)
VALUES (value1, value2, ...),
          (value1, value2, ...),...;
```

- 다른 테이블의 값을 삽입하려면 SELECT ~ FROM ~ WHERE 활용

```
INSERT INTO table (column1, column2, ...)
SELECT column1, column2, ...
FROM another_table
WHERE condition;
```

- 예제

```
INSERT INTO account(username, password, email, created_on)
VALUES
('Jose', 'password', 'jose@mail.com', CURRENT_TIMESTAMP)

INSERT INTO job(job_name)
VALUES ('Astronaut')

INSERT INTO job(job_name)
VALUES ('President')

INSERT INTO account_job(user_id, job_id, hire_date)
VALUES (1, 1, CURRENT_TIMESTAMP)
```

- 10번 user_id와 job_id가 존재하지 않기 때문에 아래 쿼리는 제약 조건 위반 에러로 실행되지 않는다

```
INSERT INTO account_job(user_id, job_id, hire_date)
VALUES (10, 10, CURRENT_TIMESTAMP)
```

UPDATE

- 테이블의 데이터를 수정하는 명령어

```
-- 기본 문법
UPDATE table
SET column1 = value1,
        column2 = value2, ...
WHERE condition;
```

- account테이블에서 last_login이 NULL이면 현재시간(CURRENT_TIMESTAMP)으로 수정하는 쿼리

```
UPDATE account
SET last_login = CURRENT_TIMESTAMP
WHERE last_login IS NULL;
```

- 모든 last_login컬럼 값을 created_on의 값으로 수정

```
UPDATE account
SET last_login = created_on;
```

- 다른 테이블의 값 사용

```
UPDATE tableA
SET original_col = tableB.new_col
FROM tableB
WHERE tableA.id = tableB.id;
```

- RETURNING: RETURNING이 없으면 CRUD 쿼리를 실행한 후에 그냥 쿼리가 실행됐다고만 뜨는데 RETURNING을 사용하면 실행한 쿼리의 결과를 SELECT한 것처럼 보여준다.

```
UPDATE account
SET last_login = created_on
RETURNING account_id, last_login;
```

- 예제

```
UPDATE account
SET last_login = CURRENT_TIMESTAMP;

UPDATE account
SET last_login = created_on;

UPDATE account_job
SET hire_date = account.created_on
FROM account
WHERE account_job.user_id = account.user_id

UPDATE account
SET last_login = CURRENT_TIMESTAMP
RETURNING email, created_on, last_login
```

DELETE

- 테이블의 데이터를 삭제하는 명령어

```
-- 기본 문법
DELETE FROM table
WHERE row_id = 1
```

- 다른 테이블에 있는 행 지우기

```
DELETE FROM tableA
USING tableB
WHERE tableA.id = tableB.id
```

- 테이블의 모든 행 지우기

```
DELETE FROM table
```

- RETURNING사용 가능

```
DELETE FROM job
WHERE job_name = 'Cowboy'
RETURNING job_id, job_name
```

ALTER

- 이미 존재하는 테이블의 구조를 변경하는 명령어
- 컬럼 추가, 삭제, 변경, 기본값 지정, 제약조건 추가 등의 작업을 할 수 있다.

```
-- 기본 문법
ALTER TABLE table_name action

-- 컬럼 추가
ALTER TABLE table_name
ADD COLUMN new_col TYPE

-- 컬럼 삭제
ALTER TABLE table_name
DROP COLUMN col_name

-- 제약 조건 변경
ALTER TABLE table_name
ALTER COLUMN col_name
ADD CONSTRAINT constraint_name
```

- 예제

```
-- 테이블명 변경
ALTER TABLE information
RENAME TO new_info

-- 컬럼명 변경
ALTER TABLE new_info
RENAME COLUMN person TO people

-- 제약조건 변경: null 허용
ALTER TABLE new_info
ALTER COLUMN people DROP NOT NULL
```

DROP

- 테이블의 컬럼을 삭제하는 명령어
- PostgreSQL에서 DROP을 실행하면 삭제되는 컬럼에 관련된 모든 인덱스와 제약조건 또한 삭제한다.
- 단 뷰, 트리거, 저장 프로시저에 연관된 컬럼은 CASCADE조건이 없으면 삭제되지 않는다.

```
-- 기본 문법
ALTER TABLE table_name
DROP COLUMN col_name

-- 존재하지 않는 컬럼에 대한 에러 방지
ALTER TABLE table_name
DROP COLUMN IF EXISTS col_name
```

```
-- 여러 컬럼 삭제
ALTER TABLE table_name
DROP COLUMN col_one,
DROP COLUMN col_two
예제
ALTER TABLE new_info
DROP COLUMN people

ALTER TABLE new_info
DROP COLUMN IF EXISTS people
CHECK
```

- 특정 조건에 맞춘 제약조건을 쓸 수 있게 해주는 명령어

```
-- 기본 문법
CHECK (conditions)
```

```
CREATE TABLE example(
        ex_id SERIAl PRIMARY KEY,
    age SMALLINT CHECK (age > 21),
    parent_age SMALLINT CHECK(parent_age > age)
);
```

```
CREATE TABLE employees(
    emp_id SERIAl PRIMARY KEY,
```

```
    first_name VARCHAR(50) NOT NULL,
    last_nmae VARCHAR(50) NOT NULL,
    birthdate DATE CHECK (birthdate > '1900-01-01'),
    hire_date DATE CHECK (hire_date > birthdate),
    salary INTEGER CHECK (salary > 0)
)
```

```
INSERT INTO employees(
    first_name,
    last_name,
    birthdate,
    hire_date,
    salary
)
VALUES(
    'Jose',
    'Portilla',
    '1899-11-03', -- 제약 조건 위반으로 insert 안됨
    '2010-01-01',
    100
)
```

연습문제

```
CREATE TABLE students(
    student_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    homeroom_number INTEGER CHECK(homeroom_number > 0),
    phone VARCHAR(500) UNIQUE NOT NULL,
    email VARCHAR(250) UNIQUE,
    graduation_year INTEGER CHECK(graduation_year > 0)
);
```

```
CREATE TABLE teachers(
    teacher_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    homeroom_number INTEGER CHECK(homeroom_number > 0),
```

```
    department  VARCHAR(50),
    phone  VARCHAR(500)  UNIQUE,
    email  VARCHAR(250)  UNIQUE
);
```

PostgreSQL 문법 정리4 - 조건식과 view 정리

CASE

- 특정 조건이 충족되었을 때 SQL코드를 실행하기 위해 CASE를 사용한다.
- 프로그래밍 언어에서 흔히 쓰이는 IF/ELSE 문과 비슷하다.
- 일반적인 CASE문과 CASE 표현식으로 쓸 수 있다. 두 방법 모두 결과는 같다.

```
-- 기본 문법
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  ELSE some_other_result
END
```

```
SELECT a,
CASE
WHEN a = 1 THEN 'one'
WHEN a = 2 THEN 'two'
ELSE 'other' AS label
END
FROM test;
```

```
-- CASE 표현식
CASE expression
  WHEN value1 THEN result1
  WHEN value2 THEN result2
  ELSE some_other_result
END
```

```
SELECT a,
CASE a
WHEN 1 THEN 'one'
WHEN 2 THEN 'two'
ELSE 'other'
END
FROM test;
```

```
-- 일반 CASE 문
SELECT customer_id,
    CASE
        WHEN (CUSTOMER_ID <= 100) THEN 'Premium'
        WHEN (CUSTOMER_ID BETWEEN 100 AND 200) THEN 'Plus'
    ELSE 'Normal'
END AS customer_class
FROM customer
```

```
-- CASE 표현식
SELECT customer_id,
CASE customer_id
    WHEN 2 THEN 'Winner'
    WHEN 5 THEN 'Second Place'
    ELSE 'Normal'
END AS raffle_results
FROM customer
```

```
SELECT
SUM(CASE rental_rate
    WHEN 0.99 THEN 1
    ELSE 0
END) AS bargains,
SUM(CASE rental_rate
    WHEN 2.99 THEN 1
    ELSE 0
END) AS regular,
SUM(CASE rental_rate
    WHEN 4.99 THEN 1
    ELSE 0
END) AS premium
FROM film;
```

- CASE 연습문제

```
--- 영화 테이블에서 등급별 영화 개수 구하기
SELECT
SUM(CASE rating
    WHEN 'R' THEN 1
```

```
    ELSE 0
END) AS r,
SUM(CASE rating
    WHEN 'PG' THEN 1
    ELSE 0
END) AS pg,
SUM(CASE rating
    WHEN 'PG-13' THEN 1
    ELSE 0
END) AS pg13
FROM film
```

COALESCE

- 더 큰 덩어리로 합치다라는 뜻
- 처음으로 NULL이 아닌 값을 만나면 그 값을 리턴하는 함수
- 따라서 NULL값을 가질 수 있는 데이터를 쿼리할 때 유용하다.

```
SELECT COALESCE (1, 2) -> 1
SELECT COALESCE (NULL, 2, 3) -> 2

-- 할인율 컬럼이 NULL이 가능한 경우
SELECT item, (price - COALESCE(discount, 0))
AS final FROM table
```

CAST

- CAST 연산자는 데이터 유형을 바꿔준다.

```
-- CAST 함수 문법
SELECT CAST('5' AS INTEGER)
SELECT CAST(date AS TIMESTAMP) FROM table

-- PostgreSQL CAST 연산자 문법
SELECT '5'::INTEGER

-- inventory_id 컬럼을 정수에서 문자열로 바꾸고 길이 재기
SELECT CHAR_LENGTH(CAST(inventory_id AS VARCHAR))
FROM rental
```

NULLIF

- NULLIF 함수는 두 개의 인자를 받고 양쪽이 같으면 NULL을 반환하고 아니면 첫 번째 인자를 반환하는 함수이다.
- NULL 값이 에러의 원인이 되거나 원하지 않는 결과가 나오는 경우에 유용하게 사용될 수 있다.
- A부서 총합과 B부서 총합을 나눠서 부서별 비율을 구하는 쿼리인데, B부서 인원이 0명이 되면 0으로 나누기 에러가 난다.

```
SELECT (
SUM(CASE WHEN department = 'A' THEN 1 ELSE 0 END) /
SUM(CASE WHEN department = 'B' THEN 1 ELSE 0 END)
) AS department_ratio
FROM depts
```

- NULLIF를 사용 할 경우 B부서 인원이 0이 되면 NULL을 반환하기 때문에 NULL로 나누기를 하면 값이 NULL로 바뀌고 에러가 나지 않는다.

```
SELECT (
SUM(CASE WHEN department = 'A' THEN 1 ELSE 0 END) /
NULLIF(SUM(CASE WHEN department = 'B' THEN 1 ELSE 0 END), 0)
) AS department_ratio
FROM depts
```

VIEWS

- View는 SQL에서 하나 이상의 테이블(또는 다른 뷰)에서 원하는 모든 데이터를 선택하여, 그들을 사용자 정의하여 나타낸 것 이다.
- 뷰는 저장된 쿼리이며 가상의 테이블로 접근할 수 있다.
- 뷰는 단순히 쿼리를 저장한 것이고 실제로 데이터를 물리적으로 저장하는 것은 아니다.
- 뷰를 통해 SQL 이용의 편의성을 높일 수 있다.
- 뷰를 통해 원본 데이터를 조회하는 대신 데이터의 일부분만 보여주게 할 수 있기 때문에 권한관리와 보안에도 장점이 있다.

```
-- VIEW 생성
CREATE VIEW view_name AS SELECT clause;
```

```
-- VIEW 생성 또는 수정
-- view_name이란 뷰가 이미 있으면 수정하고 없으면 생성한다.
CREATE OR REPLACE VIEW view_name AS SELECT clause;
```

```
-- VIEW 이름 변경
ALTER VIEW view_name RENAME to renamed_view_name
```

```
-- VIEW 삭제
DROP VIEW view_name;
CREATE VIEW customer_info AS
SELECT first_name, last_name, address
FROM customer
INNER JOIN address
ON customer.address_id = address.address_id

SELECT * FROM customer_info
```

```
-- SELECT에 district 컬럼 추가
CREATE OR REPLACE VIEW customer_info AS
SELECT first_name, last_name, address, district
FROM customer
INNER JOIN address
ON customer.address_id = address.address_id
```

PostgreSQL 문법 정리5 - 내장함수

union / union all

- 두 집합을 중복을 제거하고 합집합 형태로 조회, union all은 중복제거 x

```
select
	A.*
from
	(select
		column1 ,
		column2
	from
		A_table
	union
	select
		column3 ,
		column4
	from
		B_table) A;
```

|| , concat

- 문자열 합치기

```
select 'My name is ' || name from name_table;
--Tom Holland--

select concat('Hi ', name, ' is my name') from name_table;
--Hi Tom Holland is my name--
```

substring

- 문자열 자르기 substring(문자열, 시작점, 시작점부터 개수)

```
select substring('Tom Holland', 1 ,3);
--Tom--
```

to_date

- 문자열을 date로 변환

```
select     to_date('2023-03-30',  'yyyy-mm-dd');
```

to_char

- 숫자나 date를 문자열로 변환

```
select  to_char(now(),  'yyyy-mm-dd');
--2023-03-31--
```

- 'FM999,999,999' 포멧을 사용하여 숫자를 천 단위마다 컴마를 넣어 조회할 수 있다.

* 포멧의 수치는 출력될 값의 예상 수치보다 높게 잡아야하며 .을 넣어서 소수점자리도 맞출 수 있다.

```
select  to_char(1000000,  'FM999,999,999');
--1,000,000--
```

coalesce

- null값 치환

```
select  coalesce(name,  '이름없음')
from  name_table
--name의 값이 null일 경우 '이름없음'으로 조회--
```

upper, lower

- 대소문자 변환

```
select  upper('tom');
--TOM--

select  lower('HOLLAND');
--holland-
```

집계함수 및 숫자 관련 함수

- count, sum, avg, round, max, min

```
select  name  ,  count(*)  from    name_table
group  by  name;
--name 컬럼을 그룹화하여 name별 개수 count--
```

```
select  sum(A.cnt)  from
(select  name  ,  count(*)  as  cnt  from    name_table
```

```
group by lcl.jimok)as A;
--총합계--
```

```
select round(avg(A.cnt)) from
          (select name , count(*) as cnt from name_table
           group by
          lcl.jimok)as A;
--평균 값을 반올림(2번째 인자로 소수점자리 지정 가능)--
```

```
select max(A.cnt), min(A.cnt)
from (select name , count(*) as cnt from  name_table
          group by lcl.jimok)as A;
--최댓값과 최솟값--
```

distinct

- 중복 제거

```
select distinct name from name_table
```

array_agg(배열로 반환), array_to_string(배열을 문자열로 반환)

```
select name from name_table
--Tom Holland--
--Petter Parker--
--Spider-Man--

위와 같이 조회되는 값을 아래와 같이 조회 가능

select array_to_string(array_agg(name), ',') from name_table
--Tom Holland,Petter Parker,Spider-Man--
```

split_part

- 구분자를 기준으로 문자열을 분할한 후 지정된 위치에 해당하는 부분을 반환

```
select
        split_part(to_char(now(), 'yyyy-mm-dd'), '-', 1),
        split_part(to_char(now(), 'yyyy-mm-dd'), '-', 2),
        split_part(to_char(now(), 'yyyy-mm-dd'), '-', 3)
```