INZENT

IT Player, it's INZENT

PostgreSQL Basics

PostgreSQL Basics

2024.07

데이터플랫폼개발팀
신주한

Contents

Contents

Contents

PostgreSQL Basics

PostgreSQL

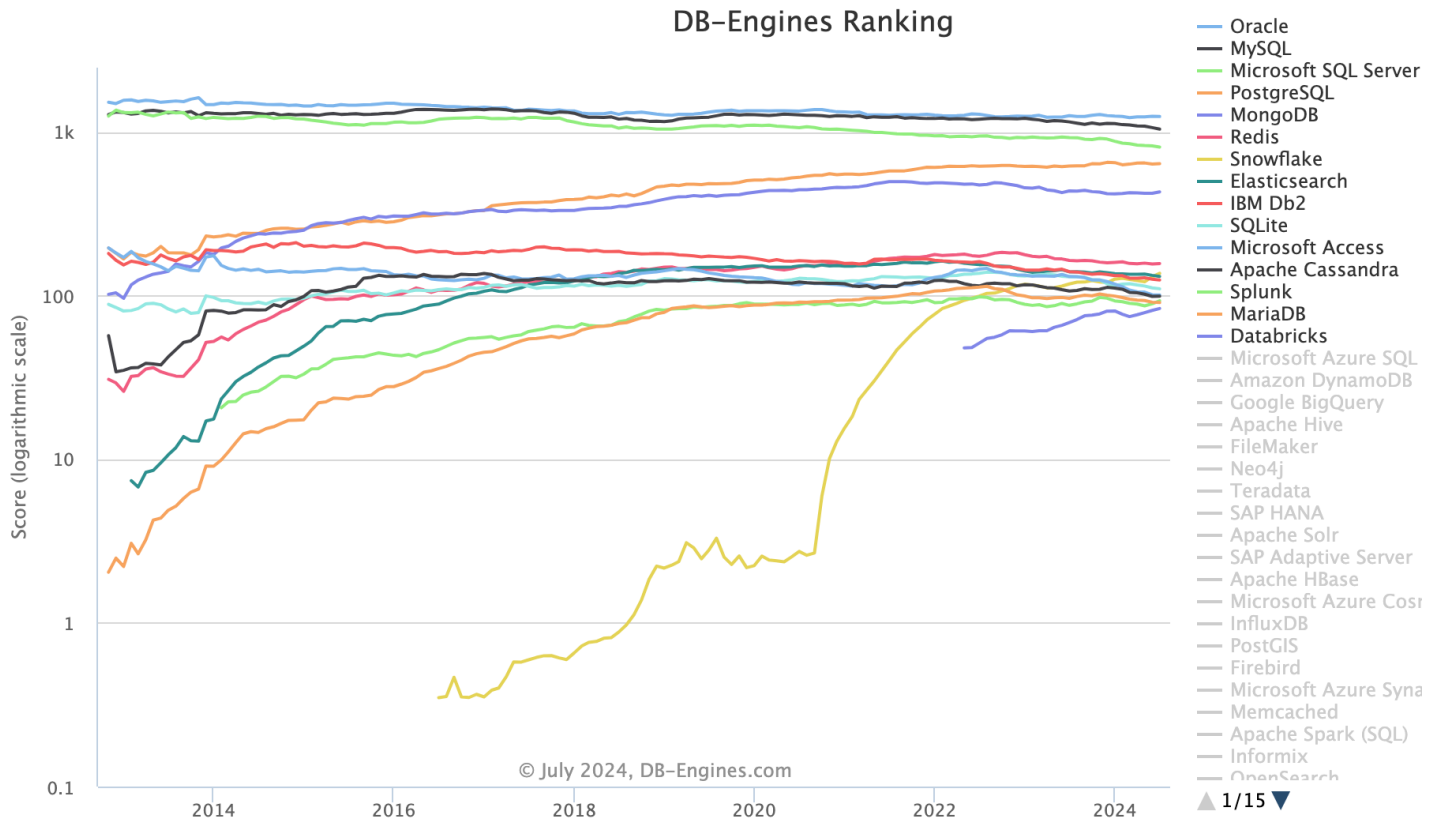
1.1. PostgreSQL

▶ Database

- 데이터베이스는 'data'와 'base'의 결합어로, 'data'는 정보나 자료를 의미하고 'base'는 이러한 자료를 조직적으로 저장하는 기반이나 장소를 의미함
- 체계적으로 조직된 데이터의 집합으로, 데이터의 효율적인 저장, 검색, 관리 및 수정이 가능하도록 설계된 시스템

▶ PostgreSQL

- PostgreSQL는 관계형 데이터베이스의 일종으로, 높은 확장성과 유연성을 제공
- 전 세계적으로 널리 사용되며, 데이터베이스 시장 점유율이 꾸준히 증가하고 있음
- 활발한 오픈소스 커뮤니티와 PostgreSQL 확장(Extension) 모듈로 인해 다양한 기능이 추가되고 있음



1.2. Table

▶ Table

- PostgreSQL은 관계형 데이터베이스로서 테이블 형태로 데이터를 저장
- 테이블은 관련된 데이터를 컬럼(Column)과 로우(Row)로 구성하여 저장

▶ Column

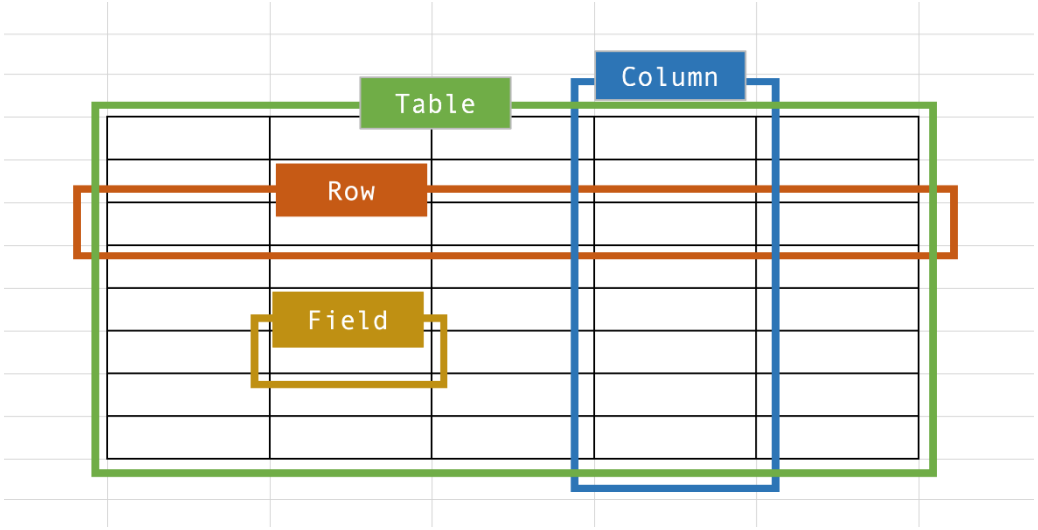
- 컬럼은 테이블의 세로 방향 구조로, 특정 데이터 타입을 가짐
- 동일한 종류의 데이터를 저장하며, '이름', '나이', '주소'와 같은 정보가 컬럼으로 정의됨

▶ Row

- 로우는 테이블의 가로 방향 구조로, 하나의 레코드를 나타냄
- 여러 컬럼에 대응하는 데이터를 포함하며, 한 로우에는 한 사람의 '이름', '나이', '주소' 등의 정보가 함께 저장됨

▶ Field

- 필드는 테이블의 특정 셀을 의미함
- 하나의 컬럼과 하나의 로우의 교차점에서 특정 데이터를 담고 있는 공간
- 필드에 채워진 값을 컬럼 값(Column value)라고도 부름



1.3. Data type

▶ Data type

- 데이터 타입은 각 컬럼에 저장될 데이터의 종류와 형식을 정의함
- 데이터의 유효성, 저장 효율성, 처리 성능을 고려해서 컬럼의 데이터 타입을 결정해야 함
- 데이터베이스마다 지원하는 데이터 타입이 조금씩 다르며, PostgreSQL은 아래와 같은 타입들을 지원함

Category	Type Name	Storage Size	Description
Numeric	smallint	2 bytes	작은 범위의 정수를 저장하는 타입, -32,768에서 32,767까지의 값을 저장
	integer	4 bytes	정수를 저장하는 타입, -2,147,483,648에서 2,147,483,647까지의 값을 저장
	bigint	8 bytes	큰 범위의 정수를 저장하는 타입, -9,223,372,036,854,775,808에서 9,223,372,036,854,775,807까지의 값을 저장
	decimal	Variable	소수점 앞 131072 자리, 소수점 이하 16383 자리까지 고정 소수점 숫자를 저장하는 타입, 정밀한 소수 계산이 필요한 경우 사용
	numeric	Variable	소수점 앞 131072 자리, 소수점 이하 16383 자리까지 고정 소수점 숫자를 저장하는 타입, 정밀한 소수 계산이 필요한 경우 사용
	real	4 bytes	10진수 6자리 정밀도의 부동 소수점 숫자를 저장하는 타입, 일반적인 소수 계산에 사용
	serial	4 bytes	자동 증가하는 정수를 저장하는 타입
Character	char(n)	n bytes	길이 제한이 있는 고정 길이 문자열을 저장하는 타입
	varchar(n)	n bytes	길이 제한이 있는 가변 길이 문자열을 저장하는 타입
	text	Variable	매우 긴 문자열을 저장하는 타입, 길이에 제한 없이 텍스트 데이터를 저장할 때 사용
Boolean	boolean	1 byte	참(true) 또는 거짓(false) 값을 저장하는 타입, 논리값을 저장할 때 사용
Date/Time	date	4 bytes	날짜를 저장하는 타입, 연/월/일의 값을 저장
	time	8 bytes	시간을 저장하는 타입, 시/분/초의 값을 저장
	timestamp	8 bytes	날짜와 시간을 함께 저장하는 타입
	interval	16 bytes	시간 간격을 저장하는 타입

1.3. Data type

▶ NULL value

- 데이터베이스에서 값이 존재하지 않음을 표현하기 위한 마커(Marker)
- 값이 아직 입력되지 않았거나, 해당 값이 존재하지 않는 경우 사용
- 0이나 empty string('')과는 다름

▶ Unknown

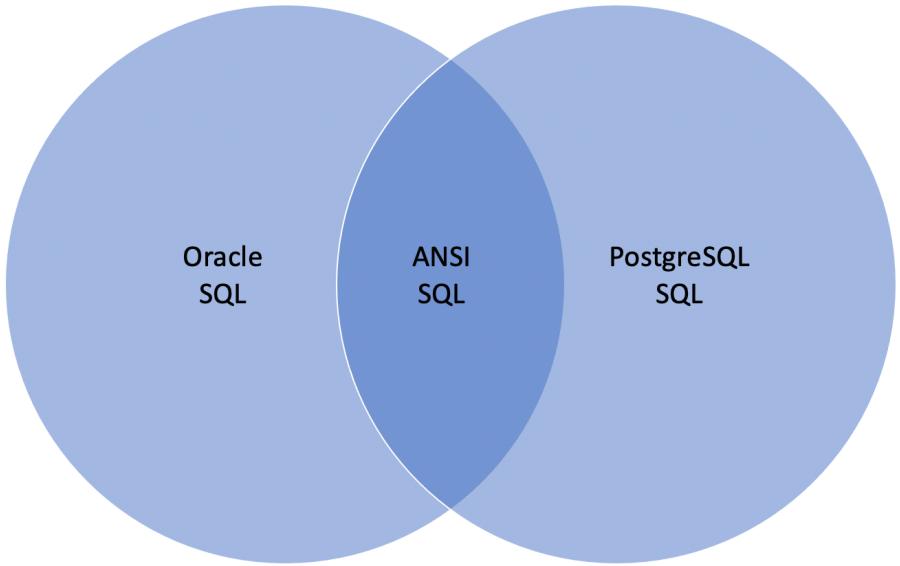
- 테이블의 필드에 NULL이 입력된 것은 값이 존재하지 않음을 나타내며, 이는 unknown을 의미함
- NULL은 값 자체가 아닌, 값의 부재를 나타내는 상태를 뜻함

Expression	Result
1 = 1	true
1 = 2	false
1 = NULL	NULL
NULL = NULL	NULL
1 + 1	2
1 + NULL	NULL

1.4. SQL (Structured Query Language)

▶ SQL

- 데이터의 저장, 수정, 삭제, 검색 등의 작업을 수행하기 위해 개발된 구조화된 질의 언어
- ANSI SQL을 기반으로 하며, Oracle, MS SQL Server, MySQL, PostgreSQL 등 대부분의 데이터베이스에서 사용됨
- 각 데이터베이스는 표준 SQL을 확장하여 고유한 기능을 제공함
- SQL 구문은 일반적으로 세미콜론(;)으로 종료되어 구문이 완료됨



1.5. PostgreSQL admin tools

INZENT

▶ PSQL

- 커맨드라인(CMD, Command line) 인터페이스 도구로, SQL 쿼리를 실행하고 데이터베이스 관리를 수행
- PostgreSQL 설치 시, 기본으로 포함되어 있음

▶ pgAdmin

- 웹 기반의 PostgreSQL 관리 도구로, 데이터베이스 구조를 시각화하고 관리 작업을 수행
- 다운로드 : https://www.pgadmin.org/download/

▶ DBeavor

- 데스크톱 전용 데이터베이스 관리 도구로, PostgreSQL을 비롯한 다양한 데이터베이스를 지원
- 다운로드 : https://dbeaver.io/download/

PostgreSQL Basics

02

SELECT 문

2.1. SELECT 정의

INZENT

▶ SELECT

- 데이터베이스에서 데이터를 조회하기 위한 SQL 문장

▶ Syntax

```
SELECT column1, column2, ...
FROM table_name
[WHERE condition]; SELECT expression;
```

▶ Example

```
SELECT name, salary
FROM employee
WHERE salary > 5000;

SELECT 1 + 1;
```

2.2. Projection

INZENT

▶ Projection

- 조회할 컬럼을 선택하는 작업
- 단순히 컬럼을 선택하는 것뿐만 아니라 선택된 컬럼에 대한 연산이나 변형도 포함
- SELECT 절에 작성

1. 특정 컬럼 조회

- 특정 컬럼을 선택해서 조회

```
SELECT name, salary
FROM employee;
```

2. 모든 컬럼 조회

- * (Asterisk)으로 모든 컬럼을 선택

```
SELECT *
FROM employee;
```

3. 컬럼 연산 (Column operation)

- 컬럼에 대한 연산이나 변형을 포함

```
SELECT name, salary, salary / 12
FROM employee;
```

4. DISTINCT

- * SELECT문 결과에서 중복된 로우를 제거

```
SELECT DISTINCT blood_type
FROM employee;
```

2.3. Selection

INZENT

▶ Selection

- 조회할 로우를 선택하는 작업
- 지정된 조건을 만족하는 로우만 필터링하여 반환
- WHERE 절에 작성

▶ Example

```
SELECT *
FROM employee
WHERE name = 'Alice';

SELECT name, salary
FROM employee
WHERE salary > 5000 AND blood_type = 'O';

SELECT name, salary, blood_type
FROM employee
WHERE blood_type = 'A' OR blood_type = 'O';
```

2.4. Practice

▶ **'Alice'와 'Bob'의 모든 정보를 조회하는 SELECT 문을 작성하세요.**

▶ **혈액형이 'AB' 형인 사람 중에 연봉이 7000 미만인 직원의 이름과 연봉을 조회하는 SELECT 문을 작성하세요.**

▶ **연봉이 3000에서 5000 사이인 직원의 모든 정보를 조회하는 SELECT 문을 작성하세요.**

▶ **연봉이 10000 이상이거나 4000 이하인 직원의 모든 정보를 조회하는 SELECT 문을 작성하세요.**

PostgreSQL Basics

연산자

3.1. 연산자 정의

▶ 연산자(Operator)

- SQL에서 다양한 작업을 수행하기 위해 사용되는 기호 또는 키워드
- 데이터 필터링, 비교, 계산, 문자열 조작, 비트 연산 등을 수행하는 데 사용
- 연산자는 WHERE 절과 SELECT 절 모두 사용 가능

▶ 연산자 종류

- 비교 연산자
- 논리 연산자
- 산술 연산자
- 문자열 연산자
- 집합 연산자
- 범위 연산자
- NULL 관련 연산자

3.2. 비교 연산자

INZENT

▶ 비교 연산자(Comparison operator)

- 비교 연산자는 두 값을 비교하여 조건의 참 또는 거짓을 결정하는 데 사용
- = : 값이 동일한지 비교
- > : Greater than
- < : Less than
- = : Greater than or equal
- <= : Less than or equal
- <>, != : Not equal

▶ Example

```
SELECT *
FROM employee
WHERE salary / 12 > 500;


SELECT *
FROM employee
WHERE name > 'Alice';
```

3.2. 비교 연산자

▶SQL에서의 대소문자 처리

- SELECT, FROM, WHERE 등의 키워드는 대소문자를 구분하지 않음
- 일반적으로 테이블, 컬럼 이름은 대소문자를 구분하지 않지 않음
- 문자 타입(CHAR, VARCHAR, TEXT)의 값은 따옴표(')로 묶어서 작성하며 대소문자를 구분함

▶ 아래 두 SQL 문은 결과가 다름

```
SELECT *
FROM employee
WHERE name = 'Alice';

SELECT *
FROM employee
WHERE name = 'aliCE';
```

▶대소문자 스타일 가이드

- SQL을 작성할 때 관습적으로 키워드들은 대문자로 작성하고, 테이블이나 컬럼 이름은 소문자로 작성 함

3.2. 비교 연산자

▶타입 호환성

- SQL에서는 기본적으로 같은 타입끼리만 연산을 처리할 수 있음
- PostgreSQL 내부에서형변환(Type conversion)을 통해 다른 종류의 타입끼리도 연산이 가능한 경우, 자동으로 형변환을 수행한 후 연산을 수행함

▶ 암묵적 형변환이 가능한 경우

```
-- 문자 > 숫자 (형변환이 자동으로 수행됨)
SELECT '123' + 100;  -- 결과: 223

-- 정수 > 실수 (형변환이 자동으로 수행됨)
SELECT 100 + 1.5;  -- 결과: 101.5
```

▶ 형변환이 불가능한 경우

```
-- 문자 > 숫자 (형변환을 명시하지 않으면 에러 발생)
SELECT '123a' + 100;  -- 에러: invalid input syntax for type integer: "123a"
```

▶명시적 형변환을 통해 연산이 가능한 경우

```
-- 날짜 문자열 > 간격 (명시적 형변환을 통해 연산)
SELECT '2023-01-01'::DATE + INTERVAL '1 day';  -- 결과: 2023-01-02

-- 실수 > 정수 (명시적 형변환을 통해 연산)
SELECT 1.99::INTEGER;  -- 결과: 2
```

3.3. 논리 연산자

INZENT

▶논리 연산자(Logical operator)

- 하나 이상의 조건을 조합하여 더 복잡한 조건식을 작성할 때 사용
- AND : 두 조건이 모두 참이면 참, 아니면 거짓
- OR : 두 조건 중 하나가 참이면 참, 둘 다 거짓이면 거짓
- NOT : 조건이 거짓이면 참, 조건이 참이면 거짓

▶ Example

```
SELECT *
FROM employee
WHERE (salary > 3000) AND (salary < 5000);

SELECT *
FROM employee
WHERE name = 'Alice' OR name = 'Bob' OR name = 'Charlie';

SELECT *
FROM employee
WHERE NOT (salary > 5000);
```

3.4. 산술 연산자

INZENT

▶산술 연산자(Arithmetic operator)

- • 숫자 타입에 대해 기본적인 수학 연산을 수행하는 데 사용
- + : 덧셈
- - : 뺄셈
- * : 곱셈
- / : 나눗셈
- % : 나머지

▶ Example

```
SELECT salary + 1000
FROM employee;

SELECT salary * 1.1
FROM employee;

SELECT salary / 2
FROM employee;

SELECT *
FROM employee
WHERE id % 2 = 0;
```

3.5. 문자열 연산자

INZENT

▶문자열 연산자(String operator)

- 문자열을 조작하거나 비교하는 데 사용되는 연산자
- || : 문자열 연결(Concatenation)
- LIKE : 패턴 매칭, 대소문자를 구분
- ILIKE : 패턴 매칭, 대소문자를 구분하지 않음
- Wildcard : 패턴에서 특정 문자나 문자열을 대체하는 데 사용
  - % : 0개 이상의 임의의 문자를 대체하는 와일드카드
  - _ : 정확히 한 개의 임의의 문자를 대체하는 와일드카드

▶ Example

```
SELECT id || '_' || name
FROM employee;

SELECT name
FROM employee
WHERE name LIKE 'A%';

SELECT name
FROM employee
WHERE name LIKE '_o%';

SELECT name
FROM employee
WHERE name ILIKE 'cHaRIe';
```

3.6. 집합 연산자

INZENT

▶ 집합 연산자(Set operator)

- IN : 값이 집합에 포함되는지 확인
- NOT IN : 값이 집합에 포함되지 않는지 확인

▶ Example

```
SELECT *
FROM employee
WHERE name IN ('Alice', 'Bob', 'Charlie');

SELECT *
FROM employee
WHERE name NOT IN ('Alice', 'Bob', 'Charlie');
```

3.7. 범위 연산자

INZENT

▶범위 연산자(Range operator)

- 값이 지정된 범위 내에 있는지를 확인
- BETWEEN start AND end : 지정된 범위 내에 있는지 확인
- 숫자, 문자, 날짜 타입 모두 사용 가능
- 시작과 종료 값 모두 포함하여 확인함

▶ Example

```
SELECT *
FROM employee
WHERE hire_date BETWEEN '2024-01-01' AND '2024-07-01';

SELECT *
FROM employee
WHERE salary BETWEEN 5000 AND 7000;
```

3.8. NULL 관련 연산자

INZENT

▶NULL 관련 연산자

- IS NULL : 값이 NULL인지 확인
- IS NOT NULL : 값이 NULL이 아닌지 확인
- IS DISTINCT FROM : 값이 다른지 확인 (NULL 값을 포함한 비교)
- IS NOT DISTINCT FROM : 값이 같은지 확인 (NULL 값을 포함한 비교)

▶ Example

```
SELECT *
FROM employee
WHERE mobile_number IS NULL;

SELECT *
FROM employee
WHERE office_number = mobile_number;

SELECT *
FROM employee
WHERE office_number IS NOT DISTINCT FROM mobile_number;
```

3.9. Practice

▶'A'형, 'B'형 직원을 조회하는 SQL을 작성하세요. IN 연산자를 이용하세요.

▶입사년도가 2020년 1월 1일 이후인 직원 중에 연봉이 6000을 넘는 직원을 조회하는 SQL을 작성하세요.

▶ 입사년도가 2020년인 직원을 조회하는 SQL을 작성하세요.

▶이름이 'J'로 시작하고, 알파벳 4글자인 직원의 이름을 조회하는 SQL을 작성하세요.

▶이름이 'k'로 끝나는 직원을 조회하는 SQL을 작성하세요.

▶ 사무실 번호를 입력하지 않은 직원의 정보를 조회하는 SQL을 작성하세요.

PostgreSQL Basics

JOIN 절

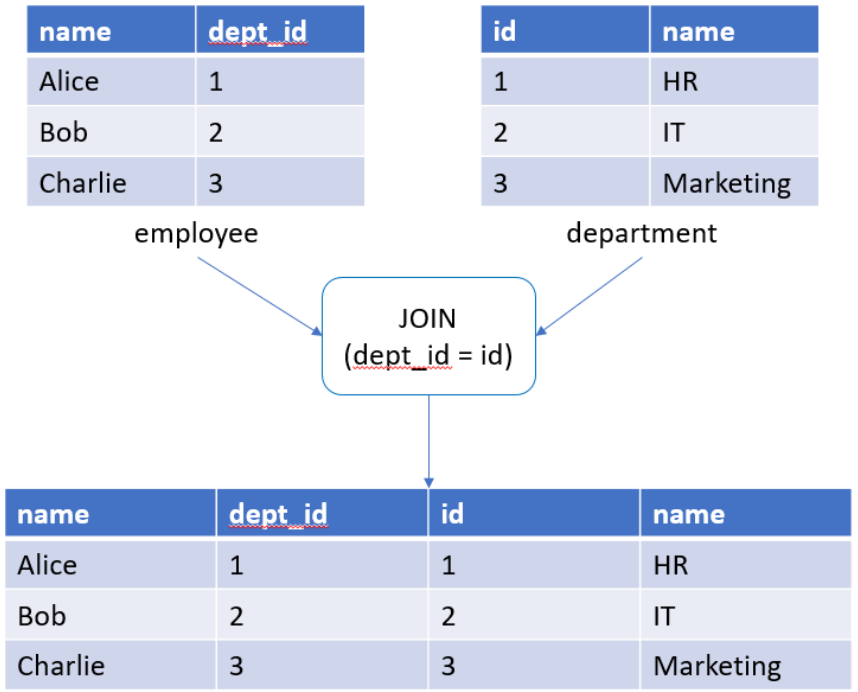
4.1. JOIN 정의

▶ JOIN

- 두 개의 테이블에서 관련된 컬럼 조건을 만족하는 로우를 결합하여 새로운 데이터 로우 집합을 생성
- 각 테이블의 공통된 컬럼 값을 기준으로 로우를 연결
- 일반적으로 고유한 값을 갖는 컬럼을 주로 사용함

▶ JOIN 종류

- Inner JOIN
- Outer JOIN
  - LEFT JOIN
  - RIGHT JOIN
  - FULL JOIN
- Cross JOIN
- Self JOIN



4.2. Inner JOIN

▶ Inner JOIN

- 두 테이블의 JOIN 조건이 충족되는 로우들을 결합
- 두 테이블에서 일치하는 로우만 포함
- 일반적으로 JOIN을 말할 때는 Inner JOIN을 뜻함

▶ Syntax

```
SELECT projection_list
FROM table_name1 INNER JOIN table_name2 ON join_condition;

SELECT projection_list
FROM table_name1, table_name2
WHERE join_condition;
```

▶ Example

```
SELECT *
FROM employee INNER JOIN department ON dept_id = id; -- ambiguois column r
eference

SELECT *
FROM employee INNER JOIN department ON dept_id = department.id;
```

4.2. Inner JOIN

▶ Qualified column name

- 테이블 이름을 포함하여 컬럼 이름을 참조하는 방식

```
table_name.column_name
```

- 테이블마다 겹치는 이름의 컬럼이 있는 경우에 이를 구분하기 위해서도 사용

```
SELECT employee .id, employee.name, dept_id, department.id, department.name
FROM employee INNER JOIN department ON dept_id = department.id;
```

▶ Alias

- SQL문 내에서 테이블/컬럼에 임시로 이름을 붙인 별칭
- 사용법
  - 테이블/컬럼의 이름 뒤에 space를 두고 별칭을 작성
  - (Optional) 테이블/컬럼의 이름 뒤에 AS 키워드를 두고 별칭을 작성

```
-- Table alias

SELECT *
FROM employee emp
WHERE emp.id = 1;

SELECT *
FROM employee AS emp
WHERE emp.id = 1;
```

```
-- Column alias

SELECT dept_id d_id
FROM employee;

SELECT dept_id as d_id
FROM employee;
```

4.2. Inner JOIN

▶ USING

- JOIN 조건에 사용된 컬럼을 하나만 projection하는 방법
- 두 테이블의 컬럼 이름이 동일해야 함

▶ Syntax

```
SELECT projection_list
FROM table_name1
INNER JOIN table_name2 USING (join_column);
```

4.2. Inner JOIN

INZENT

▶ NATURAL JOIN

- JOIN 조건에 사용된 컬럼을 하나만 projection하는 방법
- 두 테이블의 컬럼 이름이 동일해야 함
- 주의 : 동일한 이름의 컬럼이 모두 조인 조건에 포함되므로 두 테이블에 이름이 같은 컬럼이 두 개 이상일 경우 의도하지 않은 결과가 나올 수 있음

▶ Syntax

```
SELECT projection_list
FROM table_name1
NATURAL JOIN table_name2;
```

4.3. Outer JOIN

▶ LEFT JOIN

- 두 테이블의 JOIN 조건이 충족되는 로우들을 결합
- 왼쪽 테이블의 경우 JOIN 조건을 만족하지 않은 로우도 결과에 포함되며, 이 경우 오른쪽 테이블에서 채워져야 할 필드들은 NULL로 채워짐

▶ Syntax

```
SELECT projection_list
FROM table_name1 LEFT JOIN table_name2 ON join_condition;
```

▶ Example

```
-- 부서에 포함되지 않은 직원까지 포함
SELECT *
FROM employee LEFT JOIN department ON dept_id = department.id;
```

4.3. Outer JOIN

▶ RIGHT JOIN

- 두 테이블의 JOIN 조건이 충족되는 로우들을 결합
- 오른쪽 테이블의 경우 JOIN 조건을 만족하지 않은 로우도 결과에 포함되며, 이 경우 왼쪽 테이블에서 채워져야 할 필드들은 NULL로 채워짐

▶ Syntax

```
SELECT projection_list
FROM table_name1 RIGHT JOIN table_name2 ON join_condition;
```

▶ Example

```
-- 직원이 없는 부서까지 포함
SELECT *
FROM employee RIGHT JOIN department ON dept_id = department.id;
```

4.3. Outer JOIN

▶ FULL JOIN

- LEFT JOIN과 RIGHT JOIN의 합집합

▶ Syntax

```
SELECT projection_list
FROM table_name1 FULL JOIN table_name2 ON join_condition;
```

▶ Example

```
SELECT *
FROM employee FULL JOIN department ON dept_id = department.id;
```

4.4. Cross JOIN

INZENT

▶ Cross JOIN

- 두 테이블의 로우로 만들 수 있는 모든 조합을 결합
- 카티션/카테시안 곱(Cartesian product)이라고도 부름

▶ Syntax

```
SELECT projection_list
FROM table_name1, table_name2;
```

▶ Example

```
SELECT *
FROM employee, department;
```

4.5. Self JOIN

INZENT

▶ Self JOIN

- 한 테이블을 두 개의 테이블로 간주하여 JOIN 조건에 따라 서로 다른 로우들을 결합

▶ Syntax

```
SELECT projection_list
FROM table_name1 t1, table_name1 t2
WHERE join_condition;
```

▶ Example

```
SELECT *
FROM employee emp, employee mentor
WHERE emp.mentor_id = mentor.id;
```

4.6. Practice

▶ 직원의 정보와 부서 이름을 함께 조회하는 SQL을 작성하세요.

▶ 'HR' 부서의 직원 정보와 멘토의 이름을 함께 조회하는 SQL을 작성하세요.

▶멘토보다 연봉이 높은 직원의 이름과 연봉, 멘토의 이름과 연봉을 조회하는 SQL을 작성하세요.

▶ 멘토와 같은 부서에 속한 직원의 이름, 부서 이름, 멘토의 이름을 조회하는 SQL을 작성하세요.

▶ 'HR' 부서의 직원 정보와 멘토의 이름을 함께 조회하는 SQL을 작성하세요.

PostgreSQL Basics

05

집합 연산자

5.1. 집합 연산자 정의

INZENT

▶ 집합 연산자(Set operator)

- 두 SQL문의 결과를 결합하거나 비교하여 결과를 반환하는 연산자
- 각 SQL 결과의 projection의 컬럼 수와 타입이 동일해야 함

▶ 집합 연산자 종류

- UNION
- INTERSECT
- EXCEPT
- UNION ALL
- INTERSECT ALL
- EXCEPT ALL

5.2. UNION

▶ UNION

- 두 SQL 문의 결과를 합집합으로 결합하고, 중복된 로우를 제거한 후 반환

▶ Syntax

```
SELECT projection_list
FROM table_name
UNION
SELECT projection_list
FROM table_name;
```

▶ Example

```
SELECT name, blood_type
FROM employee
WHERE blood_type IN ('O', 'A')
UNION
SELECT name, blood_type
FROM employee
WHERE blood_type IN ('O');
```

5.3. INTERSECT

▶ INTERSECT

- 두 SQL 문의 결과의 교집합을 추출하고, 중복된 로우를 제거한 후 반환

▶ Syntax

```
SELECT projection_list
FROM table_name
INTERSECT
SELECT projection_list
FROM table_name;
```

▶ Example

```
SELECT name, blood_type
FROM employee
WHERE blood_type IN ('O', 'A')
INTERSECT
SELECT name, blood_type
FROM employee
WHERE blood_type IN ('O');
```

5.4. EXCEPT

INZENT

▶ EXCEPT

- 첫 번째 SQL 문의 결과에서 두 번째 SQL 문의 결과를 뺀 차집합을 구하고, 중복된 로우를 제거한 후 반환

▶ Syntax

```
SELECT projection_list
FROM table_name
EXCEPT
SELECT projection_list
FROM table_name;
```

▶ Example

```
SELECT name, blood_type
FROM employee
WHERE blood_type IN ('O', 'A')
EXCEPT
SELECT name, blood_type
FROM employee
WHERE blood_type IN ('O');
```

5.5. UNION/INTERSECT/EXCEPT ALL

▶ UNION/INTERSECT/EXCEPT ALL

- 각각 ALL이 없는 기존의 집합 연산자에서 중복된 로우를 제거하는 것과 달리 중복된 로우를 유지

▶ Syntax

```
SELECT projection_list
FROM table_name
UNION ALL
SELECT projection_list
FROM table_name;
```

```
SELECT projection_list
FROM table_name
INTERSECT ALL
SELECT projection_list
FROM table_name;
```

```
SELECT projection_list
FROM table_name
EXCEPT ALL
SELECT projection_list
FROM table_name;
```

▶ Example

```
SELECT name, blood_type
FROM employee
WHERE blood_type IN ('O', 'A')
UNION ALL
SELECT name, blood_type
FROM employee
WHERE blood_type IN ('O');
```

```
SELECT name, blood_type
FROM employee
WHERE blood_type IN ('O', 'A')
INTERSECT ALL
SELECT name, blood_type
FROM employee
WHERE blood_type IN ('O');
```

```
SELECT name, blood_type
FROM employee
WHERE blood_type IN ('O', 'A')
EXCEPT ALL
SELECT name, blood_type
FROM employee
WHERE blood_type IN ('O');
```

5.6. Practice

▶ **직원(employee)들의 이름과 퇴사자(resignee)들의 이름을 조회하는 SQL을 작성하세요. (UNION)**

▶ **연봉이 5000이 넘는 직원과 퇴사자의 이름, 연봉을 조회하는 SQL을 작성하세요.**

▶ **'HR' 부서의 직원 이름과, 'HR'에서 근무했던 퇴사자 이름을 조회하는 SQL을 작성하세요.**

PostgreSQL Basics

06

AGGREGATION 함수

6.1. Aggregation 정의

▶ Aggregation 함수

- 테이블 전체 또는 여러 로우를 대상으로 함수에 정의된 연산을 통해 하나의 값을 반환하는 함수

▶ Aggregation 함수 종류

- COUNT
- SUM
- MIN/MAX
- AVG

6.2. Aggregation 함수

INZENT

▶ COUNT

- SQL문의 결과에 해당하는 로우들의 수를 반환하는 함수
- 참고: COUNT()를 제외한 모든 집계 함수는 NULL 값을 무시함

▶ Syntax

```
SELECT COUNT(*)
FROM table_name;

SELECT COUNT(column_name)
FROM table_name;
```

▶ Example

```
SELECT COUNT(*)
FROM employee;
```

6.2. Aggregation 함수

▶ SUM

- SQL문의 결과에 해당하는 로우에 속한 컬럼 값을 인자로 받아 그 합을 반환하는 함수

▶ Syntax

```
SELECT SUM(column_name)
FROM table_name;
```

▶ Example

```
SELECT SUM(salary)
FROM employee;
```

6.2. Aggregation 함수

INZENT

▶ MIN/MAX

- SQL문의 결과에 해당하는 로우에 속한 컬럼 값을 인자로 받아 최소/최대 값을 반환하는 함수

▶ Syntax

```
SELECT MIN(column_name)
FROM table_name;

SELECT MAX(column_name)
FROM table_name;
```

▶ Example

```
SELECT MIN(salary)
FROM employee;

SELECT MAX(salary)
FROM employee;
```

6.2. Aggregation 함수

▶ AVG

- SQL문의 결과에 해당하는 로우에 속한 컬럼 값을 인자로 받아 평균 값을 반환하는 함수

▶ Syntax

```
SELECT AVG(column_name)
FROM table_name;
```

▶ Example

```
SELECT AVG(salary)
FROM employee;
```

6.2. Practice

▶ **혈액형이 'O'인 직원의 평균 연봉을 조회하는 SQL문을 작성하세요.**

▶ **'HR' 부서 직원들의 평균 연봉을 구하는 SQL문을 작성하세요. (JOIN)**

▶ **'HR' 부서 직원들의 멘토들의 평균 연봉을 조회하는 SQL문을 작성하세요. (JOIN)**

▶ **'O'형 직원의 평균 연봉과 'A'형 직원의 평균 연봉을 한 번에 조회하는 SQL문을 작성하세요. (UNION)**

PostgreSQL Basics

07

SUBQUERY

7.1. Subquery 정의

INZENT

▶ Subquery

- SQL 문 내에서 수행되는 별개의 쿼리(Query) 문
- 일반적으로 쿼리는 SQL 문 중에서 SELECT로 시작되는 구문을 의미함
- Subquery를 포함하는 SQL을 main query, outer query라고 부름
- 항상 SQL문 내에서 괄호로 묶여 있기 때문에 inner query, nested query라고도 불림

▶ Subquery 종류

- Scalar subquery
- Single-row subquery
- Multi-row subquery
- Correlated subquery

7.2. Scalar subquery

INZENT

▶ Scalar subquery

- 단일 값(컬럼 값)을 반환하는 subquery
- Outer query에서 단일 값이 필요한 경우 사용

▶ Syntax

```
SELECT projection_list
FROM table_name
WHERE column_name = (SELECT single_value FROM table_name WHERE condition);
```

▶ Example

```
-- 평균 연봉보다 연봉이 높은 직원을 조회
SELECT *
FROM employee
WHERE salary > (SELECT AVG(salary) FROM employee);
```

7.3. Single-row subquery

INZENT

▶ Single-row subquery

- 하나의 로우만 반환하는 subquery

▶ Single-row operator

- '=', '>', '<', '>=', '<=', '!='

▶ Syntax

```
SELECT projection_list
FROM table_name
WHERE column_list = (SELECT column_list FROM table_name WHERE single_row_condition);
```

▶ Example

```
-- 'Alice' 직원과 혈액형이 같은 직원
SELECT *
FROM employee
WHERE (blood_type) > (SELECT blood_type FROM employee WHERE name =
'Alice');
```

7.4. Multi-row subquery

INZENT

▶ Multi-row subquery

- 여러 로우를 반환하는 subquery

▶ Multi-row operator

- operator ANY : Subquery의 결과 로우와 하나라도 조건을 만족하면 true를 반환
- operator ALL : Subquery의 모든 결과 로우와 조건을 만족하면 true를 반환
- IN : Subquery의 결과 로우 중 하나와 일치하는 경우 true를 반환
- EXISTS : Subquery의 결과 로우가 하나라도 존재하면 true, 없으면 false를 반환

▶ Syntax

```
SELECT projection_list
FROM table_name
WHERE column_list = ANY (SELECT column_list FROM table_name WHERE multi_row_
condition);

SELECT projection_list
FROM table_name
WHERE column_list = ALL (SELECT column_list FROM table_name WHERE multi_row_
condition);

SELECT projection_list
FROM table_name
WHERE EXISTS (SELECT column_list FROM table_name WHERE multi_row_condition);
```

▶ Example

```
-- 모든 퇴사자들보다 연봉이 높은 직원 조회
SELECT *
FROM employee emp
WHERE emp.salary > ALL (SELECT res.salary from resignee res);

-- 퇴사자와 동명이인 직원 조회
SELECT *
FROM employee emp
WHERE emp.name IN (SELECT res.name from resignee res);

-- 퇴사자 중에 'Alice'가 존재하면 모든 퇴사자를 조회
SELECT *
FROM resignee
WHERE EXISTS (SELECT name from resignee WHERE name = 'Alice');
```

7.5. Correlated subquery

INZENT

▶ Correlated subquery

- Outer query의 컬럼을 참조하는 subquery
- 다른 종류의 subquery는 subquery 단독으로도 수행이 가능하지만, correlated query는 외부의 컬럼이 포함되어 있기 때문에 단독으로 수행할 수 없음

▶ Syntax

```
SELECT projection_list
FROM table_name1
WHERE column_name IN
                    (SELECT column_name
                     FROM table_name2
                     WHERE table_name1.column_name = table_name2.column_name);
```

▶ Example

```
-- 자신의 부서 평균 연봉보다 연봉이 높은 직원을 조회
SELECT *
FROM employee emp1
WHERE salary > (SELECT AVG(salary) FROM employee emp2 WHERE emp1.dept_id
= emp2.dept_id);
```

7.6. Practice

▶ 근무기간이 가장 긴 직원의 정보를 조회하는 SQL문을 작성하세요.

▶'Alice' 보다 연봉이 두 배 이상 높은 직원의 이름과 연봉을 조회하는 SQL을 작성하세요.

▶ 가장 최근에 입사한 직원보다 연봉이 낮은 직원의 정보를 조회하는 SQL문을 작성하세요.

▶ 연봉이 가장 높은 직원과 같은 부서의 직원들의 이름, 부서이름, 연봉을 조회하는 SQL을 작성하세요.

▶ 가장 최근에 입사한 직원과 혈액형이 같은 직원을 조회하는 SQL문을 작성하세요. (가장 최근에 입사한 직원의 정보는 제외하세요.)

PostgreSQL Basics

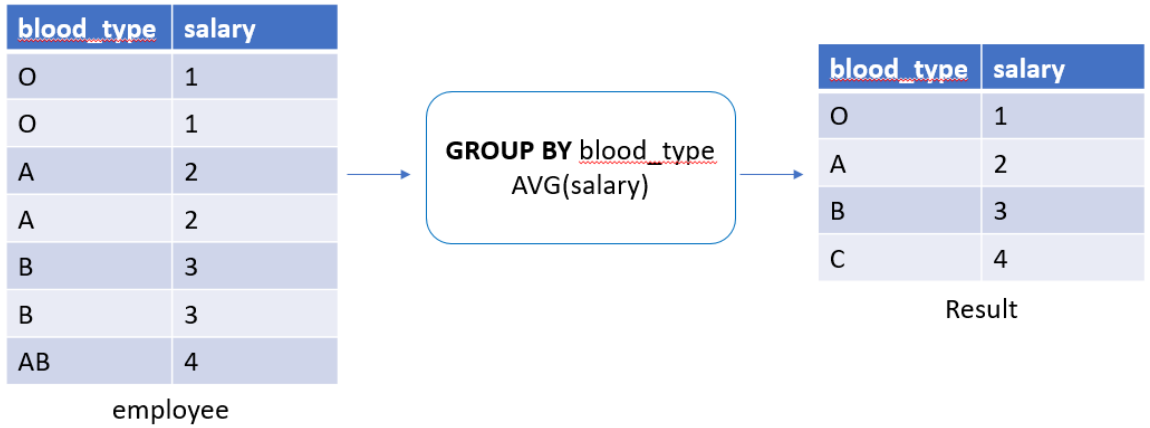
08

GROUP BY 절

8.1. Group by 정의

▶ Group by

- 로우 집합을 특정 조건에 의해 그룹으로 구분하고, 각 그룹에 대한 집계 결과를 제공함
- 주로 Aggregation 함수와 함께 사용되며, 각 그룹에 대한 Aggregation 결과를 반환함
- GROUP BY 절이 있는 경우 SELECT 절에는 다음 두 종류만 작성할 수 있음
  - GROUP BY 절에 사용된 컬럼
  - Aggregation 함수
- GROUP BY 절에는 여러 컬럼을 조합해서 사용할 수 있음
  - 첫 번째 컬럼으로 그룹화를 하고, 각 그룹 내에서 두 번째 컬럼으로 순서대로 그룹화 진행



▶ Syntax

```
SELECT projection_list
FROM table_name
GROUP BY group_by_expression;

SELECT projection_list
FROM table_name
WHERE condition
GROUP BY group_by_expression;
```

▶ Example

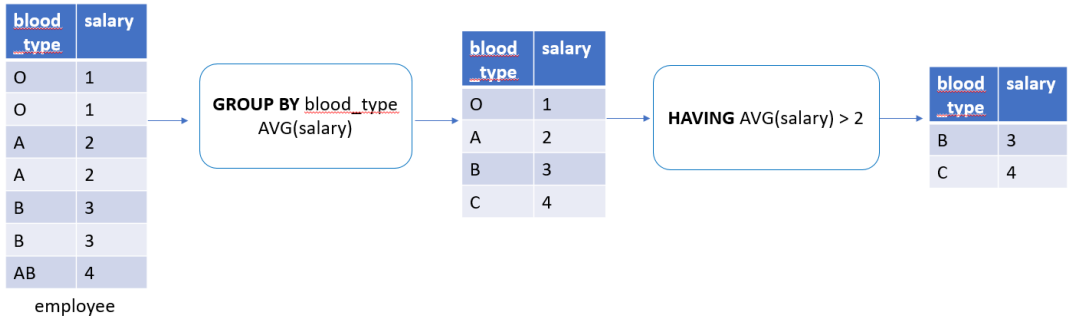
```
-- 부서별 평균 연봉을 조회
SELECT dept_id, AVG(salary)
FROM employee
GROUP BY dept_id;

-- 부서별, 혈액형별 평균 연봉을 조회
SELECT dept_id, blood_type, AVG(salary)
FROM employee
GROUP BY dept_id, blood_type;
```

8.2. HAVING 절

▶ HAVING

- GROUP BY 결과를 필터링하기 위한 조건을 작성
- WHERE 절과 달리 Aggregation 함수와 함께 사용되어 그룹화된 데이터에 조건을 적용
- GROUP BY 절에 사용된 컬럼과 Aggregation 함수만 필터링 조건에 사용할 수 있음



▶ Syntax

```
SELECT projection_list
FROM table_name
WHERE condition
GROUP BY group_by_expression
HAVING condition;
```

▶ Example

```
-- 평균 연봉이 7000 이상인 부서들의 평균연봉 조회
SELECT dept_id, AVG(salary)
FROM employee
GROUP BY dept_id
HAVING AVG(salary) >= 7000;

-- 평균 연봉이 7500에서 8000사이인 부서들의 평균연봉 조회
SELECT dept_id, AVG(salary)
FROM employee
GROUP BY dept_id
HAVING AVG(salary) BETWEEN 7500 AND 8000;
```

8.3. Practice

▶ **부서이름과 부서별 평균 연봉을 조회하는 SQL을 작성하세요**

▶ **혈액형별 평균 연봉과 해당 혈액형을 가진 직원 수를 조회하는 SQL을 작성하세요.**

▶ **부서별 평균 연봉과 각 부서의 직원 수를 조회하는 SQL을 작성하세요.**

▶ **부서별 평균 연봉이 8000 이상인 부서의 이름과 평균 연봉을 조회하는 SQL을 작성하세요.**

PostgreSQL Basics

09

ORDER BY 절

9.1. Order by 정의

INZENT

▶ Order by

- SQL 문의 결과를 오름차순 또는 내림차순으로 정렬
- 정렬 순서를 생략할 경우 기본적으로 오름차순으로 정렬
- 여러 컬럼을 기준으로 정렬할 수 있으며, 각 컬럼에 대해 개별적인 정렬 순서를 지정할 수 있음
- GROUP BY와 마찬가지로 첫 번째 컬럼으로 정렬을 마친 후, 그 다음 컬럼으로 정렬을 수행하여 보다 세부적인 정렬을 할 수 있음

▶ Syntax

```
SELECT projection_list
FROM table_name
ORDER BY order_by_expression [ASC | DESC];

SELECT projection_list
FROM table_name
WHERE condition
GROUP BY group_by_expression
HAVING having_condition
ORDER BY order_by_expression [ASC | DESC];
```

▶ Example

```
-- 직원 정보를 이름으로 내림차순하여 정렬
SELECT *
FROM employee
ORDER BY name desc;

-- 직원의 정보를 혈액형으로 정렬한 후, 혈액형 별로 직원의 이름으로 정렬
SELECT *
FROM employee
ORDER BY blood_type, name;
```

9.2. Practice

▶ 직원을 혈액형 별로 정렬하고, 같은 혈액형 내에서는 직원의 이름 순서로 정렬하는 SQL문을 작성하세요. (혈액형은 오름차순, 직원 이름은 내림차순으로 정렬하세요.)

▶ 부서 이름으로 직원을 정렬하고, 같은 부서 내에서는 직원의 이름 순서로 정렬하는 SQL문을 작성하세요. (부서 이름은 오름차순, 직원 이름은 내림차순으로 정렬하세요.)

▶ 각 부서별 이름과 평균 연봉을 구하고, 평균 연봉 순서로 정렬하는 SQL문을 작성하세요.

▶ 각 부서별, 혈액형별 평균 연봉을 구하고, 연봉 순서로 정렬하는 SQL문을 작성하세요.

PostgreSQL Basics

10

WINDOW 함수

10.1. Window 정의

▶ Window 함수

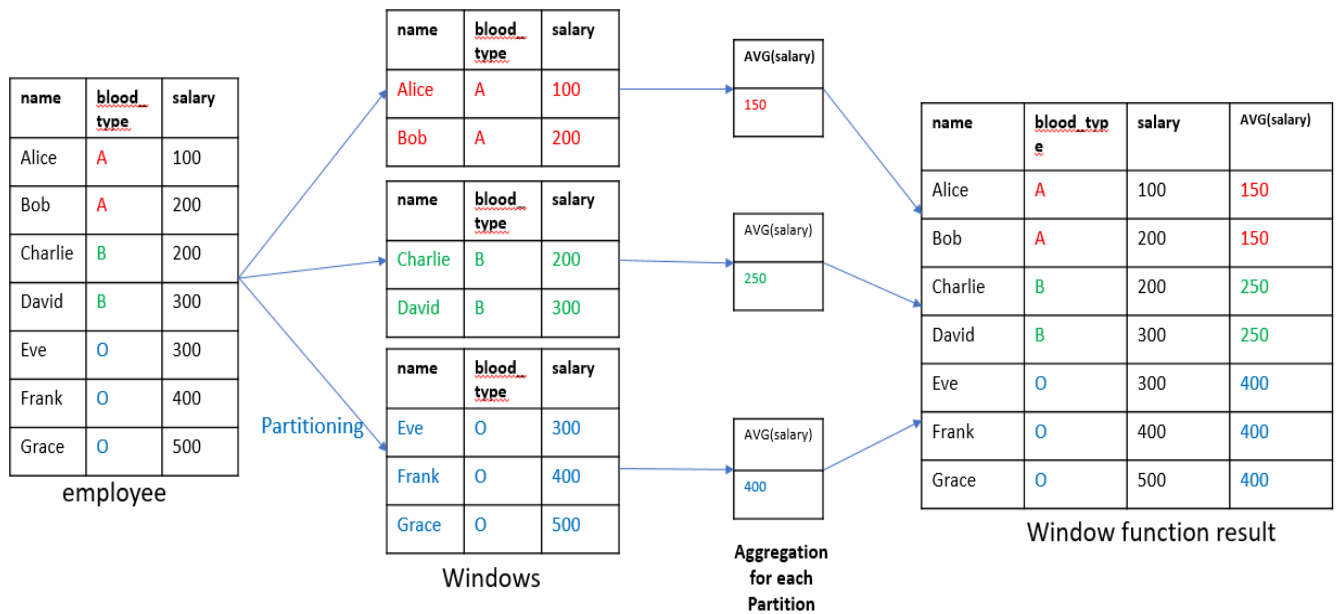
- 로우 집합을 그룹(윈도우)으로 나누고, 각 로우가 속한 그룹의 연산 결과를 로우와 함께 조회할 수 있도록 값을 반환해주는 함수
- GROUP BY 처럼 그룹화를 진행하지 않고 모든 로우에 연산 결과를 추가함
- 대부분의 Aggregation 함수가 Window 함수로 사용 가능
- Aggregation 함수는 WHERE 절의 결과인 로우 집합을 그룹화하는 과정에서 로우의 수를 줄이지만 Window 함수는 기존의 로우에 그 로우가 속한 그룹의 연산 결과를 projection 단계에서 추가하기만 함
- SELECT 절과 ORDER BY 절에서만 사용할 수 있는 제약이 있음

▶ Syntax

```
SELECT window_function OVER ([PARTITION BY condition] [ORDER BY [ASC | DESC ]] [ROWS BETWEEN range_keyword])
FROM table_name;
```

▶ Example

```
SELECT *, AVG(salary) OVER (PARTITION BY blood_type ORDER BY salary DESC) avg_salary
FROM employee;
```



10.1. Window 정의

INZENT

▶ Window 함수의 구성 요소

- **Window 함수**
  - 윈도우에서 수행할 연산의 종류를 정의하는 함수
  - Aggregation 함수 대부분 사용 가능
- **OVER 절**
  - Window 함수를 정의하는 기본 구문
  - Aggregation 함수와 Wndow 함수를 구분
- **PARTITION BY**
  - 로우 집합을 그룹(윈도우)로 나누는 기준을 정의
  - PARTITION BY로 나눈 그룹 내에서 독립적으로 연산이 수행됨
- **ORDER BY**
  - 각 윈도우 내에서 로우의 순서를 오름차순/내림차순으로 정의
  - 기존 ORDER BY 절과 동일하게 오름차순이 기본값
  - ORDER BY는 선택 사항이며 생략 가능
  - ORDER BY가 있는 경우 윈도우의 로우 범위는 윈도우의 시작부터 현재 처리중인 로우까지로 설정됨
  - 즉, ORDER BY가 있으면 윈도우의 시작부터 현재 처리중인 로우까지만 수행하고, ORDER BY가 없으면 윈도우에 속한 모든 로우에 대해 Window 함수을 수행
- **ROWS BETWEEN**
  - 윈도우 내에서 연산을 수행할 로우의 범위를 더 제한할 때 사용
  - UNBOUNDED PRECEDING : 윈도우 내에서 첫 번째 로우의 위치
  - UNBOUNDED FOLLOWING : 윈도우 내에서 마지막 로우의 위치
  - CURRENT ROW : 윈도우 내에서 현재 로우의 위치
  - n PRECEDING : 윈도우 내에서 CURRENT ROW부터 앞으로 n번째 위치
  - n FOLLOWING : 윈도우 내에서 CURRENT ROW로부터 뒤로 n번째 위치

10.2. Window 함수

INZENT

▶ ROW_NUMBER

- 각 윈도우 내의 로우에 고유한 순번을 할당
- 동일한 순번이 중복되지 않음

▶ Example

```
-- 각 부서별로 직원의 연봉 순위를 매겨서 직원 정보를 조회
SELECT id, name, salary, dept_id, ROW_NUMBER() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS row_num
FROM employee;
```

10.2. Window 함수

INZENT

▶ RANK, DENSE_RANK

- RANK
  - 각 윈도우 내의 로우에 순위를 할당하며, 동점이 있을 경우 순위는 건너뛰어 할당
- DENSE_RANK
  - 각 윈도우 내의 로우에 순위를 할당하며, 동점이 있어도 순위는 연속적으로 할당

▶ Example

```
-- 각 부서별로 직원의 연봉 순위(동점 포함)를 매겨서 직원 정보를 조회
SELECT
    id,
    name,
    salary,
    dept_id,
    RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS rank
FROM employee;

-- 각 부서별로 직원의 연봉 순위(동점 포함, 연속 순위)를 매겨서 직원 정보를 조회
SELECT
    id,
    name,
    salary,
    dept_id,
    DENSE_RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS rank
FROM employee;
```

10.2. Window 함수

▶ AVG, SUM, MIN, MAX

- Aggregation 함수와 동일한 연산을 각 윈도우 내에서 수행

▶ Example

```
-- 각 직원의 연봉과 직원이 속한 부서의 평균 연봉을 함께 조회
SELECT id, name, salary, dept_id, AVG(salary) OVER (PARTITION BY dept_id)
FROM employee;

-- 각 직원의 연봉과 직원이 속한 부서의 가장 높은 연봉을 함께 조회
SELECT id, name, salary, dept_id, MAX(salary) OVER (PARTITION BY dept_id)
FROM employee;
```

10.2. Window 함수

INZENT

▶ LEAD, LAG

- LEAD : 각 윈도우 내에서 현재 로우의 다음 위치의 로우 값을 참조
- LAG : 각 윈도우 내에서 현재 로우의 이전 위치의 로우 값을 참조

▶ Example

```
-- 부서별 각 직원의 자신보다 바로 다음으로 높은 연봉의 직원 급여를 조회
SELECT
    id,
    name,
    salary,
    dept_id,
    LEAD(salary, 1) OVER (PARTITION BY dept_id ORDER BY salary) AS previous_salary
FROM employee;

-- 부서별 각 직원의 자신보다 바로 다음으로 낮은 연봉의 직원 급여를 조회
SELECT
    id,
    name,
    salary,
    dept_id,
    LAG(salary, 1) OVER (PARTITION BY dept_id ORDER BY salary) AS previous_salary
FROM employee;
```

10.2. Window 함수

▶ OVER()

- OVER 절에 요소가 하나도 포함되지 않는 경우
- OVER 절 다음 괄호에 아무 요소도 추가하지 않아도 동작함
- 이런 경우 Aggregation 함수의 결과를 모든 로우에 추가할 수 있는 효과가 있음

▶ Example

```
-- 각 직원의 연봉과 전체 평균 연봉을 함께 조회
SELECT id, name, salary, AVG(salary) OVER ()
FROM employee;
```

10.2. Window 함수

▶ Window 함수 결과에 대한 필터링

- Window 함수 결과를 가지고 로우를 필터링하고 싶을 경우 window 함수를 사용한 SQL을 subquery를 사용하여 결과를 필터링 할 수 있음
- NOTE: Subquery를 FROM 절에서 사용할 경우 alias를 반드시 붙여야 함

▶ Example

```
-- 각 부서의 연봉 상위 1명만 남기고 나머지는 필터링
SELECT dept_id, name, salary, pos
FROM
  (SELECT dept_id, name, salary,
          RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS pos
     FROM employee
  ) AS emp
WHERE pos <= 1;
```

10.3. Practice

▶직원의 연봉과 그 직원과 동일한 혈액형인 직원들의 평균 연봉을 함께 조회하는 SQL을 작성하세요.

▶직원을 입사일 순으로 정렬하고, 자신보다 먼저 입사한 직원 한명의 연봉과 자신의 연봉을 함께 조회하는 SQL문을 작성하세요. (LAG)

▶직원을 연봉 순으로 정렬하고, 자신보다 연봉이 낮은 3명의 평균 연봉을 함께 조회하는 SQL문을 작성하세요. (AVG, ROWS BETWEEN)

PostgreSQL Basics

DDL

11.1. DDL

INZENT

▶ DDL (Database Definition Language)

- 데이터베이스 구조를 정의하고 관리하는 SQL
- 데이터베이스 객체(Database object)를 생성, 수정, 삭제
- 주요 명령어: CREATE, ALTER, DROP, TRUNCATE

11.1. DDL

▶ 데이터베이스 객체(Database object)

- **Table**
  - 데이터를 저장하고, 수정하고, 삭제할 수 있는 실제 데이터를 담고 있는 객체
- **View**
  - 하나 이상의 테이블을 기반으로 생성한 가상 테이블로, 실제 데이터를 저장하지 않으며 쿼리 결과를 재사용하기 위해 사용
  - 뷰를 조회하면 실제로는 뷰에 정의된 SQL을 수행하여 결과를 조회
  - 뷰를 사용하는 주요 목적 중 하나는 테이블에서 보안상 민감한 데이터를 감추고, 일부 데이터만 사용자에게 노출시키는 것
  - 테이블에 대한 직접적인 접근 권한을 제한하고, 뷰에 대한 접근 권한만 사용자에게 부여하여 민감한 정보를 보호할 수 있음
- **Index**
  - 테이블의 검색 속도를 향상시키기 위해 사용되는 데이터 구조로, 특정 컬럼에 대한 빠른 접근을 가능하게 함
- **Synonym**
  - 다른 데이터베이스 객체의 별칭을 제공하여 접근을 쉽게 하거나 명명 충돌을 방지
- **Sequence**
  - 고유한 숫자 값을 생성하는 객체로, 주로 기본 키 값을 자동으로 생성하기 위해 사용
- **Function**
  - 특정 작업을 수행하고 결과를 반환하는 코드 블록으로, 재사용 가능한 로직
  - User-Defined function
- **Trigger**
  - 데이터베이스나 데이터베이스 객체에 특정 이벤트가 발생할 때 자동으로 실행되는 코드 블록
  - 데이터 무결성 유지나 비즈니스 규칙을 구현하는 데 사용

11.2. TABLE DDL

INZENT

▶ CREATE TABLE

- 새로운 테이블을 생성

▶ Syntax

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    ...
);
```

▶ Example

```
CREATE TABLE employee (
    id INT,
    name VARCHAR(100),
    salary INT,
    blood_type CHAR(2)
);
```

11.2. TABLE DDL

INZENT

▶ ALTER TABLE

- 기존 테이블을 수정

▶ Syntax

```
ALTER TABLE table_name
ADD column_name datatype;

ALTER TABLE table_name
DROP COLUMN column_name;
```

▶ Example

```
ALTER TABLE employee
ADD hire_date DATE;

ALTER TABLE employee
DROP COLUMN hire_date;
```

11.2. TABLE DDL

INZENT

▶ DROP TABLE

- 기존 테이블을 제거
- CASCADE 키워드를 붙이면 DROP 대상 테이블과 연관이 있는 모든 데이터베이스 객체 또는 제약조건을 함께 제거함

▶ Syntax

```
DROP TABLE table_name [CASCADE];
```

▶ Example

```
DROP TABLE employee [CASCADE];
```

11.2. TABLE DDL

▶ CREATE TABLE AS (CTAS)

- SELECT 결과를 저장하는 새로운 테이블을 생성

▶ Syntax

```
CREATE TABLE new_table AS
SELECT column1, column2, ...
FROM existing_table
WHERE condition;
```

▶ Example

```
CREATE TABLE high_salary_employee AS
SELECT id, name, salary
FROM employee
WHERE salary > 70000000;
```

11.2. TABLE DDL

INZENT

▶ TRUNCATE TABLE

- 기존 테이블은 유지하며 테이블의 모든 로우를 제거

▶ Syntax

```
TRUNCATE TABLE table_name;
```

▶ Example

```
TRUNCATE TABLE employee;
```

11.2. VIEW DDL

INZENT

▶ CREATE VIEW

- 새로운 뷰를 생성

▶ Syntax

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

▶ Example

```
CREATE VIEW employee_view AS
SELECT id, name, salary
FROM employee
WHERE salary > 50000000;
```

11.2. VIEW DDL

INZENT

▶ ALTER VIEW

- 기존 뷰를 수정

▶ Syntax

```
ALTER VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

▶ Example

```
ALTER VIEW employee_view AS
SELECT id, name, salary, blood_type
FROM employee
WHERE salary > 50000000;
```

11.2. VIEW DDL

INZENT

▶ DROP VIEW

- 기존 뷰를 제거

▶ Syntax

```
DROP VIEW view_name;
```

▶ Example

```
DROP VIEW employee_view;
```

11.3. Practice

▶ 혈액형이 'O'형인 직원들의 정보만 모아놓은 employee_o 테이블을 생성하는 SQL문을 작성하세요. (CTAS)

▶ 위에서 생성한 employee_o를 제거하는 SQL문을 작성하세요.

▶ 직원의 정보 중 연봉을 제외한 employee_without_salary 뷰(View)를 생성하는 SQL을 작성하세요.

▶ 직원 정보에 부서 이름을 추가한 employee_with_dept_name 뷰를 생성하는 SQL을 작성하세요.

PostgreSQL Basics

DML

12.1. DML 정의

INZENT

▶ DML (Data Manipulation Language)

- 데이터베이스에서 데이터를 조작하기 위해 정의된 SQL 문
- 데이터에 대한 삽입, 수정, 삭제를 수행
- 대표적으로 INSERT, UPDATE, DELETE를 사용

12.2. INSERT 문

▶ INSERT VALUES 문

- 특정 컬럼을 지정하여 테이블에 로우를 추가
- 컬럼을 지정하지 않을 경우 컬럼 값은 테이블 컬럼 전체와 매칭이 되어야 함
- 지정되지 않은 컬럼은 NULL이 입력됨

▶ Syntax

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

▶ Example

```
INSERT INTO employee (id, name, salary, blood_type)
VALUES (1, 'Alice', 7800, 'A');

INSERT INTO employee
VALUES (1, 'Alice', 7800, 'A');

INSERT INTO employee (id, name)
VALUES (1, 'Alice');
```

12.2. INSERT 문

▶ INSERT SELECT 문

- SELECT 결과를 지정한 테이블에 삽입

▶ Syntax

```
INSERT INTO table_name1 (column1, column2, ...)
SELECT column1, column2, ...
FROM another_table2
WHERE condition;
```

▶ Example

```
INSERT INTO employee (id, name, salary, blood_type)
SELECT id, name, salary, blood_type
FROM applicant
WHERE salary < 5000;
```

12.2. INSERT 문

INZENT

▶ Multi-row INSERT 문

- 여러 행을 한 번의 명령으로 삽입
- INSERT를 여러 번 수행하는 것보다 효율적

▶ Syntax

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1a, value2a, ...),
       (value1b, value2b, ...),
       ...;
```

▶ Example

```
INSERT INTO employee (id, name, salary, blood_type)
VALUES
(1, 'Alice', 7800, 'A'),
(2, 'Bob', 5000, 'B'),
(3, 'Charlie', 7900, 'O');
```

12.3. UPDATE 문

INZENT

▶ UPDATE 문

- 테이블의 기존 로우를 수정할 때 사용
- WHERE 절의 조건을 만족하는 로우의 컬럼 값을 수정
- 조건이 없을 경우 테이블 전체 로우를 수정

▶ Syntax

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
[WHERE condition];
```

▶ Example

```
UPDATE employee
SET salary = 8000
WHERE name = 'Bob';
```

12.4. DELETE 문

INZENT

▶ UPDATE 문

- 테이블에서 기존 로우를 삭제할 때 사용
- WHERE 절의 조건을 만족하는 로우를 삭제
- 조건이 없을 경우 테이블 전체 로우를 삭제

▶ Syntax

```
DELETE FROM table_name
WHERE condition;
```

▶ Example

```
DELETE FROM employee
WHERE name = 'Bob';
```

12.5. Practice

▶ **'HR' 부서의 직원 정보를 직원(employee) 테이블에서 퇴직자(resignee) 테이블에 추가하는 SQL문을 작성하세요.**

▶ **'HR' 부서의 직원을 직원(employee) 테이블에서 제거하는 SQL문을 작성하세요.**

▶ **'IT' 부서의 모든 직원의 연봉을 10% 인상하는 SQL을 작성하세요.**

▶ **현재 연봉이 전체 평균 연봉보다 낮은 직원들의 연봉을 평균 연봉으로 변경하는 SQL을 작성하세요.**

▶ **'Bob'의 mentor_id를 연봉이 가장 높은 직원의 id로 변경하는 SQL을 작성하세요.**

PostgreSQL Basics

13

CONTRAINT

13.1. Constraint 정의

▶ Constraint

- 테이블 데이터에 대한 제약사항을 추가
- Constraint를 위반하는 DML/DDL은 수행이 불가능함

▶ Constraint 종류

- UNIQUE
- CHECK
- NOT NULL
- DEFAULT
- PRIMARY KEY
- FOREIGN KEY

13.2. UNIQUE

▶ UNIQUE

- 특정 컬럼의 값이 테이블 내에서 중복되지 않도록 보장
- 각 로우의 해당 컬럼이 고유한 값을 가져야 함
- NULL은 예외로 여러 개 존재할 수 있음

▶ Syntax

```
-- 컬럼 수준에서 UNIQUE 제약 조건을 설정할 때
CREATE TABLE table_name (
    column1 datatype UNIQUE,
    column2 datatype,
    ...
);

-- 테이블 수준에서 UNIQUE 제약 조건을 설정할 때
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    ...
    CONSTRAINT constraint_name UNIQUE (column1, column2, ...)
);

ALTER TABLE table_name
ADD CONSTRAINT constraint_name UNIQUE (column1, column2, ...);
```

▶ Example

```
CREATE TABLE employee (
    id INT,
    name VARCHAR(100),
    email VARCHAR(255) UNIQUE
);

CREATE TABLE employee (
    id INT,
    name VARCHAR(100),
    email VARCHAR(255),
    CONSTRAINT unique_email UNIQUE (email)
);

CREATE TABLE employee (
    id INT,
    name VARCHAR(100),
    email VARCHAR(255)
);
ALTER TABLE employee ADD CONSTRAINT unique_email UNIQUE (email);
```

13.3. NOT NULL

▶ NOT NULL

- 특정 컬럼이 NULL 값을 가질 수 없도록 보장
- 해당 컬럼에는 반드시 데이터가 입력되어야 함

▶ Syntax

```
-- 컬럼 수준에서 NOT NULL 제약 조건을 설정할 때
CREATE TABLE table_name (
    column1 datatype NOT NULL,
    column2 datatype,
    ...
);

-- 테이블 수준에서 NOT NULL 제약 조건을 설정할 때
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    ...
    CONSTRAINT constraint_name NOT NULL (column_name)
);

-- PostgreSQL은 NOT NULL에 MODIFY 사용하지 않음
ALTER TABLE table_name
ALTER COLUMN column_name SET NOT NULL;
```

▶ Example

```
CREATE TABLE employee (
    id INT,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(255)
);

CREATE TABLE employee (
    id INT,
    name VARCHAR(100),
    email VARCHAR(255),
    CONSTRAINT not_null_name CHECK (name IS NOT NULL)
);

CREATE TABLE employee (
    id INT,
    name VARCHAR(100),
    email VARCHAR(255)
);
ALTER TABLE employee MODIFY COLUMN name VARCHAR(100) NOT NULL;
```

13.4. CHECK

▶ CHECK

- 특정 컬럼의 값이 특정 조건을 만족해야 함
- NULL 값은 입력이 가능함

▶ Syntax

```
-- 컬럼 수준에서 CHECK 제약 조건을 설정할 때
CREATE TABLE table_name (
    column1 datatype CHECK (condition),
    column2 datatype,
    ...
);

-- 테이블 수준에서 CHECK 제약 조건을 설정할 때
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    ...
    CONSTRAINT constraint_name CHECK (condition)
);

ALTER TABLE table_name
ADD CONSTRAINT constraint_name CHECK (condition);
```

▶ Example

```
CREATE TABLE employee (
    id INT,
    name VARCHAR(100),
    salary INT CHECK (salary > 0)
);

CREATE TABLE employee (
    id INT,
    name VARCHAR(100),
    salary INT,
    CONSTRAINT check_salary CHECK (salary > 0)
);

CREATE TABLE employee (
    id INT,
    name VARCHAR(100),
    salary INT
);
ALTER TABLE employee ADD CONSTRAINT check_salary CHECK (salary > 0);
```

13.5. DEFAULT

▶ DEFAULT

- 특정 컬럼의 기본값을 설정
- DML 문에서 해당 컬럼의 값이 명시적으로 제공되지 않으면 기본값이 자동으로 입력

▶ Syntax

```
-- 컬럼 수준에서 DEFAULT 제약 조건을 설정할 때
CREATE TABLE table_name (
    column1 datatype DEFAULT default_value,
    column2 datatype,
    ...
);

-- 테이블 수준에서 DEFAULT 제약 조건을 설정할 때
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    ...
    CONSTRAINT constraint_name DEFAULT default_value FOR column_name
);

ALTER TABLE table_name
ALTER COLUMN column_name SET DEFAULT default_value;
```

▶ Example

```
DROP TABLE IF EXISTS employee;
CREATE TABLE employee (
    id INT,
    name VARCHAR(100),
    salary INT DEFAULT 3000
);

DROP TABLE IF EXISTS employee;
CREATE TABLE employee (
    id INT,
    name VARCHAR(100),
    salary INT,
    CONSTRAINT default_salary DEFAULT 3000 FOR salary
);

DROP TABLE IF EXISTS employee;
CREATE TABLE employee (
    id INT,
    name VARCHAR(100),
    salary INT
);
ALTER TABLE employee ALTER COLUMN salary SET DEFAULT 3000;
```

13.6. PRIMARY KEY

▶ PRIMARY KEY

- 테이블에서 각 로우를 고유하게 식별할 수 있는 하나의 컬럼 또는 컬럼의 조합을 지정
- 자동으로 NOT NULL과 UNIQUE 제약 조건이 포함됨
- 한 테이블에는 하나의 PRIMARY KEY만 존재할 수 있음

▶ Syntax

```
-- 컬럼 수준에서 PRIMARY KEY 제약 조건을 설정할 때
CREATE TABLE table_name (
    column1 datatype PRIMARY KEY,
    column2 datatype,
    ...
);

-- 테이블 수준에서 PRIMARY KEY 제약 조건을 설정할 때
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    ...
    CONSTRAINT constraint_name PRIMARY KEY (column1)
);

ALTER TABLE table_name
ADD CONSTRAINT constraint_name PRIMARY KEY (column1);
```

▶ Example

```
CREATE TABLE employee (
  id INT PRIMARY KEY,
  name VARCHAR(100),
  email VARCHAR(255)
);

CREATE TABLE employee (
  id INT,
  name VARCHAR(100),
  email VARCHAR(255),
  CONSTRAINT pk_emp PRIMARY KEY (id)
);

CREATE TABLE employee (
  id INT,
  name VARCHAR(100),
  email VARCHAR(255)
);
ALTER TABLE employee ADD CONSTRAINT pk_emp PRIMARY KEY (id);
```

13.7. FOREIGN KEY

▶ FOREIGN KEY

- 컬럼이 다른 테이블의 PRIMARY KEY나 UNIQUE 컬럼을 참조하도록 설정
- FOREIGN KEY 컬럼의 값은 참조한 컬럼의 값 중 하나이거나 NULL이어야 함
- PRIMARY KEY가 존재하는 테이블을 부모 테이블, PRIMARY KEY를 참조하는 FOREIGN KEY가 존재하는 테이블을 자식 테이블이라고 부름

▶ Syntax

```
-- 컬럼 수준에서 FOREIGN KEY 제약 조건을 설정할 때
CREATE TABLE table_name1 (
    column1 datatype,
    column2 datatype,
    column3 datatype REFERENCES table_name2 (other_column),
    ...
);

-- 테이블 수준에서 FOREIGN KEY 제약 조건을 설정할 때
CREATE TABLE table_name1 (
    column1 datatype,
    column2 datatype,
    column3 datatype,
    ...
    CONSTRAINT constraint_name FOREIGN KEY (column3) REFERENCES
table_name2 (other_column)
);

ALTER TABLE table_name
ADD CONSTRAINT constraint_name FOREIGN KEY (column_name) REFERENCES
table_name2 (other_column);
```

▶ Example

```
CREATE TABLE employee (
  id INT,
  name VARCHAR(100),
  dept_id INT REFERENCES department (id)
);

CREATE TABLE employee (
  id INT,
  name VARCHAR(100),
  dept_id INT,
  CONSTRAINT fk_dept FOREIGN KEY (dept_id) REFERENCES department (id)
);

CREATE TABLE employee (
  id INT,
  name VARCHAR(100),
  dept_id INT
);
ALTER TABLE employee
ADD CONSTRAINT fk_dept FOREIGN KEY (dept_id) REFERENCES department
(id);
```

13.7. FOREIGN KEY

▶ DROP TABLE CASCADE

- - 특정 테이블을 삭제할 때, 해당 테이블과 연관된 모든 객체를 함께 삭제
- - FK 관계에서의 부모 테이블을 먼저 제거하려고 할 경우 FK constraint 때문에 제거를 할 수 없음
- - FK constraint를 먼저 제거한 후, 부모 테이블을 제거하거나 아니면 CASCADE 옵션을 추가해서 제거

▶ Syntax

```
DROP TABLE table_name CASCADE;
```

▶ Example

```
DROP TABLE department CASCADE;
```

13.7. FOREIGN KEY

▶ ON DELETE

- ON DELETE 옵션은 부모 테이블의 로우가 삭제될 때 자식 테이블에서 해당 FOREIGN KEY 제약을 처리하는 방법을 정의
- 아래 세 가지 ON DELETE 옵션을 삭제된 로우에 대해 설정할 수 있음
- ON DELETE SET NULL
  - 부모 테이블의 로우가 삭제될 때, 자식 테이블의 FOREIGN KEY 값을 NULL로 설정
- ON DELETE SET DEFAULT
  - 부모 테이블의 로우가 삭제될 때, 자식 테이블의 FOREIGN KEY 값을 사전에 정의된 기본값으로 설정
  - FOREIGN KEY 컬럼에 DEFAULT 제약사항이 없으면 NULL 값으로 설정됨
- ON DELETE CASCADE
  - 부모 테이블의 로우가 삭제될 때, 자식 테이블의 관련 로우도 자동으로 삭제

▶ Syntax

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    ...
    FOREIGN KEY (column_name) REFERENCES other_table (other_column) ON
DELETE [SET NULL | SET DEFAULT | CASCADE]
);
```

▶ Example

```
CREATE TABLE employee (
    id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(255),
    dept_id INT DEFAULT 0,  -- 기본값 설정
    CONSTRAINT fk_ dept FOREIGN KEY (dept_id) REFERENCES dept (id) ON
DELETE SET DEFAULT
);
```

13.8. Practice

▶ **직원(employee) 테이블에서 혈액형(blood_type) 컬럼의 값을 'O', 'A', 'AB', 'B' 만 입력할 수 있도록 DDL문을 작성하세요**

PostgreSQL Basics

14

SQL 함수

14.1. SQL 함수 정의

INZENT

▶ SQL 함수

- 데이터베이스 내에서 특정 작업을 수행하도록 미리 정의되어 제공되는 코드 블록
- 데이터베이스 설치시부터 포함되어 있기 때문에 내장(Built-in)함수라고도 부름

14.2. SQL 함수

▶ generate_series

- 특정 범위의 숫자 또는 날짜의 값을 연속된 로우로 생성하는 함수

▶ Example

```
-- 숫자 시퀀스 생성
SELECT generate_series(1, 5);
SELECT generate_series(1, 5, 2);

-- 날짜 시퀀스 생성
SELECT generate_series('2023-01-01'::DATE, '2024-01-01'::DATE, '1 day');
SELECT generate_series('2023-01-01'::DATE, '2024-01-01'::DATE, '2 month');
```

▶ lpad, rpad

- 문자열을 지정된 길이로 채우고, 남은 부분을 특정 문자로 채워주는 함수

▶ Example

```
-- 좌측 채우기
SELECT lpad('ab', 5, 'c');  -- 결과: 'cccab'

-- 우측 채우기
SELECT rpad('ab', 5, 'c');  -- 결과: 'abccc'

-- 간단하게 큰 데이터를 담은 테이블을 생성할 때
CREATE TABLE t1 (c1 INT, c2 VARCHAR(8000));
INSERT INTO t1 SELECT generate_series(1, 10000), lpad('x', 8000, 'x');
```

▶ coalesce

- 주어진 표현식들 중 첫 번째로 NULL이 아닌 값을 반환하는 함수
- 컬럼 값이 NULL인 경우, NULL이 아닌 다른 값으로 반환하는데 사용

▶ Example

```
SELECT coalesce(NULL, 'default', 'value');  -- 결과: 'default'
SELECT coalesce(NULL, 'value');  -- 결과: 'value'

SELECT salary + 1000 FROM employee; // salary가 NULL인 로우가 있는 경우 unknown
SELECT coalesce(salary, 0) + 1000 FROM employee; // salary가 NULL인 로우가 있는 경우 100
```

▶ pg_sleep

- 지정된 시간(초) 동안 현재 세션을 일시 중지하는 함수

▶ Example

```
-- 2초 동안 일시 중지
SELECT pg_sleep(2);

-- WHERE 절에서 로우를 처리하는 과정에서 디버깅을 하고 싶은 경우
SELECT * FROM employee WHERE salary > 0 AND pg_sleep(10);
```

▶ json_build_object / jsonb_build_object

- 키-값 쌍을 받아서 JSON 객체를 생성하는 함수

▶ Example

```
SELECT jsonb_build_object('name', 'Alice', 'age', 30); -- 결과: {"name": "Alice", "age": 30}

SELECT json_build_object('name', 'Alice', 'age', 30); -- 결과: {"name": "Alice", "age": 30}

SELECT jsonb_build_object(id, name)
FROM employee; -- 결과: {"name": "Alice", "age": 30}

SELECT jsonb_build_object('emp', jsonb_build_object(id, name))
FROM employee;
```

▶ pg_typeof

- 해당 값의 데이터 타입을 반환
- 유추가 불가능한 경우는 unknown을 반환

▶ Example

```
SELECT pg_typeof(1); -- Default type : integer
SELECT pg_typeof('Hello'::VARCHAR);

SELECT pg_typeof('Hello'); -- unknown
SELECT pg_typeof(NULL); -- unknown

SELECT pg_typeof(now()); -- timestamp with time zone
```

14.2. SQL 함수

INZENT

▶ now

- 현재 날짜와 시간을 반환하는 함수

▶ Example

```
SELECT now();
```

▶ age

- 날짜 타입을 입력하면 현재와의 시간 차이를 INTERVAL 타입으로 반환

▶ Example

```
select age('2023-01-01'::DATE);
```

▶ extract

- 날짜 또는 시간 값에서 특정 필드를 추출하는 함수

▶ Syntax

```
extract (field FROM source)
```

- field
  - day : The day of the month (1-31)
  - dow : The day of the week as Sunday(0) to Saturday(6)
  - doy : The day of the year (1-366)
  - epoch : The time difference in seconds from '1970-01-01 00:00:00'
  - hour : The hour field
  - month : The number of the month within the year (1-12)
  - year : The year field
- source
  - timestamp, date, time, interval 타입의 값

▶ Example

```
-- 날짜에서 연도 추출
SELECT extract(year FROM '2023-07-16'::date);  -- 결과: 2023

-- 타임스탬프에서 월 추출
SELECT extract(month FROM '2023-07-16 12:34:56'::timestamp);  -- 결과: 7

SELECT extract(year FROM age('2021-01-01'::DATE));
```

14.2. SQL 함수

INZENT

▶ random

- 0.0 <= x < 1.0 범위의 난수를 반환

▶ Example

```
-- 0.0부터 1.0 사이의 실수
SELECT random();

-- 0.0부터 100.0 사이의 실수
SELECT random() * 100;

-- 0부터 100 사이의 정수
SELECT (random() * 100)::INT;
```

14.3. Practice

▶ 직원의 아이디, 나이, 연봉을 필드로 사용하는 JSON 객체를 생성하는 SQL 문을 작성하세요. (json_build_object())

▶ 1부터 10까지의 직원 아이디 중 랜덤으로 하나의 사원 아이디를 뽑아서 해당 사원의 정보를 조회하는 SQL 문을 작성하세요. SQL을 수행할 때마다 직원 아이디가 변경되도록 작성하세요. (random())

▶ 입사월 별로 직원의 평균 연봉을 구하세요. (extract())

▶ 직원의 이름과 나이를 구하고, 나이를 기준으로 정렬하는 SQL을 작성하세요. (extract(), age())

▶ 'Bob' 보다 근무 기간이 2년 이상 높은 직원을 조회하는 SQL 문을 작성하세요. (extract(), age(), age() + INTERVAL '2 years')

▶ 직원의 이름과 부서명, 그리고 직원이 속한 부서별 평균 근속년수를 함께 조회하는 SQL을 작성하세요. (extract(), age(), OVER)

Contact Us

(주)인젠트
서울 영등포구 국제금융로2길 36, 유화증권빌딩 8층, 9층

Tel : 070.8209.6189 | Fax : 02.787.3699
info@inzent.com

www.inzent.com