Comprehensive PostgreSQL SQL Guide

Table of Contents

SQL Command Types

DDL (Data Definition Language) Commands

These commands are used to define and modify the structure of database objects like tables, indexes, and schemas.

- `CREATE DATABASE` : Creates a new database
- `DROP DATABASE` : Deletes a database
- `CREATE TABLE` : Creates a new table
- `DROP TABLE` : Deletes a table
- `ALTER TABLE` : Modifies the structure of an existing table
- `CREATE INDEX` : Creates an index on a table column
- `DROP INDEX` : Deletes an index
- `CREATE SCHEMA` : Creates a new schema
- `DROP SCHEMA` : Deletes a schema

DML (Data Manipulation Language) Commands

These commands are used to manipulate data within the database.

- `INSERT INTO` : Inserts new rows into a table
- `UPDATE` : Updates existing rows in a table
- `DELETE` : Deletes rows from a table
- `SELECT` : Retrieves data from one or more tables
- `TRUNCATE` : Removes all rows from a table without logging individual row deletions

DCL (Data Control Language) Commands

These commands are used to control access to data.

- `GRANT` : Grants privileges to a user or role
- `REVOKE` : Revokes privileges from a user or role

TCL (Transaction Control Language) Commands

These commands are used to manage transactions in the database.

- `BEGIN` : Starts a new transaction
- `COMMIT` : Commits the current transaction
- `ROLLBACK` : Rolls back the current transaction to the last commit

---

SQL Query Execution Order

The execution order of SQL clauses in PostgreSQL is different from the written order:

1. **FROM** (and JOINs)
    - Identifies tables involved and performs necessary joins
    - Retrieves raw data from tables
2. **WHERE**
    - Filters rows based on specified conditions

    - Only rows satisfying the WHERE condition are included
3. **GROUP BY**
    - Groups filtered rows based on specified columns
    - Necessary for aggregate functions (COUNT, SUM, AVG, etc.)
4. **HAVING**
    - Filters groups based on conditions applied to aggregate values
    - Only groups satisfying the HAVING condition are included
5. **SELECT**
    - Determines which columns to include in the final result
    - Evaluates expressions or calculations specified
6. **DISTINCT**
    - Removes duplicate rows from the result set
7. **ORDER BY**
    - Sorts the result set based on specified columns
8. **LIMIT / OFFSET**
    - Restricts number of rows returned
    - Skips specified number of rows

Example Query

```
SELECT department_id, AVG(salary) AS avg_salary
FROM employees
WHERE salary > 40000
GROUP BY department_id
HAVING AVG(salary) > 50000
ORDER BY avg_salary DESC
LIMIT 2;
```

Common Mistakes

1. **Using SELECT aliases in WHERE**:

    ```
    SELECT employee_name AS name
    FROM employees
    WHERE name = 'Alice'; -- Error: "name" is not recognized in WHERE
    ```

    Fix: Use the original column name in the WHERE clause.
2. **Using HAVING without GROUP BY**: HAVING is used to filter groups, so it requires a GROUP BY clause (unless using it with aggregate functions on the entire table).

Database Management

Create a Database

```
CREATE DATABASE database_name;
```

Delete (Drop) a Database

```
DROP DATABASE database_name;
```

Alter a Database

Rename a Database:

```
ALTER DATABASE old_database_name RENAME TO new_database_name;
```

Change the Owner of a Database:

```
ALTER DATABASE old_database_owner OWNER TO new_database_owner;
```

Creating and Querying Tables

Create a New Table

```
CREATE TABLE users(
    id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    age INT,
    city VARCHAR(50)
);
```

Insert Data into the Table

```
INSERT INTO users(name, age, city) VALUES
('A', 30, 'Thanjavur'),
('B', 25, 'Kumbakonam'),
('C', 35, 'Orathanadu'),
('D', 45, 'Chennai'),
('E', 30, 'Madurai');
```

Retrieve Data Using SQL Commands

SELECT Statement

```
SELECT * FROM users;

SELECT id, name, age, city FROM users;
```

Column Aliases

```
SELECT name AS user_name, age AS user_age FROM users;
```

ORDER BY Clause

```
SELECT * FROM users ORDER BY age ASC;
SELECT * FROM users ORDER BY age DESC;
```

SELECT DISTINCT

```
SELECT DISTINCT age FROM users;        -- Works
SELECT DISTINCT age, name FROM users;  -- Works for unique combinations
```

Filtering Data

Sample Table Setup

```
CREATE TABLE employees(
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    department VARCHAR(50),
    salary NUMERIC(10,2),
    hire_date DATE
);

INSERT INTO employees(name, department, salary, hire_date) VALUES
('Alice', 'HR', 50000, '2020-06-15'),
('Bob', 'Engineering', 75000, '2019-03-22'),
('Charlie', 'Marketing', 60000, '2021-08-10'),
('David', 'Engineering', 80000, '2018-11-05'),
('Eve', 'HR', 55000, '2022-01-14'),
('Frank', 'Marketing', NULL, '2020-07-01');
```

Filtering Examples

WHERE Clause

Filter rows based on a specified condition:

```
SELECT * FROM employees WHERE department = 'Engineering';
```

AND Operator

Combines two boolean expressions and returns true if both are true:

```
SELECT * FROM employees WHERE department = 'Engineering' AND salary >= 75000;
```

OR Operator

Returns true if either expression is true:

```
SELECT * FROM employees WHERE department = 'HR' OR department = 'Marketing';
```

LIMIT Clause

Retrieves a subset of rows:

```
SELECT * FROM employees LIMIT 3;
SELECT * FROM employees ORDER BY id DESC LIMIT 3;
```

FETCH Clause

Similar to LIMIT:

```
SELECT * FROM employees FETCH FIRST 2 ROWS ONLY;
```

IN Operator

Selects data matching any value in a list:

```
SELECT * FROM employees WHERE department IN ('HR', 'Marketing');
```

BETWEEN Operator

Selects data within a range:

```
SELECT * FROM employees WHERE salary BETWEEN 50000 AND 70000;
```

LIKE Operator

Pattern matching for strings:

```
SELECT * FROM employees WHERE name LIKE 'A%';        -- Names starting with A
SELECT * FROM employees WHERE name LIKE '%e';        -- Names ending with e
SELECT * FROM employees WHERE name LIKE 'A____';     -- Names starting with A and exactly 5 characters long
SELECT * FROM employees WHERE name ILIKE 'a%';       -- Case-insensitive match for names starting with a or A
SELECT * FROM employees WHERE name LIKE '%e%';       -- Names containing e
SELECT * FROM employees WHERE name NOT LIKE 'A%';    -- Names not starting with A
```

IS NULL Operator

Checks for NULL values:

```
SELECT * FROM employees WHERE salary IS NULL;
SELECT * FROM employees WHERE salary IS NOT NULL;
```

OFFSET

Skips specified number of rows:

```
SELECT * FROM employees OFFSET 2 ROWS;
SELECT * FROM employees OFFSET 2 ROWS LIMIT 2;
SELECT * FROM employees OFFSET 2 ROWS FETCH FIRST 2 ROWS ONLY;
```

Joins

Sample Tables Setup

```
DROP TABLE IF EXISTS employees;

CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    employee_name VARCHAR(100) NOT NULL,
    department_id INT,
    salary NUMERIC(10, 2)
);

CREATE TABLE departments(
    department_id SERIAL PRIMARY KEY,
    department_name VARCHAR(50) NOT NULL
);

INSERT INTO employees (employee_name, department_id, salary) VALUES
('Alice', 1, 50000),
('Bob', 2, 60000),
('Charlie', 1, 55000),
('David', NULL, 70000),
('Eve', 3, 45000);

INSERT INTO departments (department_name) VALUES
('HR'),
('Engineering'),
('Marketing');
```

Table Aliases

```
SELECT e.employee_name, d.department_name
FROM employees AS e
JOIN departments AS d
ON e.department_id = d.department_id;
```

INNER JOIN

Returns only rows with matching values in both tables:

```
SELECT e.employee_name, d.department_name
FROM employees AS e
INNER JOIN departments AS d
ON e.department_id = d.department_id;
```

LEFT JOIN

Returns all rows from the left table and matching rows from the right table:

```
SELECT e.employee_name, d.department_name
FROM employees AS e
LEFT JOIN departments AS d
ON e.department_id = d.department_id;
```

Example with tables reversed:

```
SELECT e.employee_name, d.department_name
FROM departments AS d
LEFT JOIN employees AS e
ON d.department_id = e.department_id;
```

RIGHT JOIN

Returns all rows from the right table and matching rows from the left table:

```
SELECT e.employee_name, d.department_name
FROM employees AS e
RIGHT JOIN departments AS d
ON e.department_id = d.department_id;
```

Example with tables reversed:

```
SELECT e.employee_name, d.department_name
FROM departments AS d
RIGHT JOIN employees AS e
ON d.department_id = e.department_id;
```

Grouping Data

GROUP BY Clause

Groups rows that have the same values in specified columns into summary rows.

Syntax:

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1;
```

Example 1: Basic GROUP BY

Find the total salary for each department:

```
SELECT department_id, SUM(salary)
FROM employees
GROUP BY department_id;
```

Example 2: GROUP BY with JOIN

Find the total salary for each department with department name:

```
SELECT e.department_id, d.department_name, SUM(salary) AS TotalSalary
FROM employees AS e
INNER JOIN departments AS d ON e.department_id = d.department_id
GROUP BY e.department_id, d.department_name;
```

Example 3: COUNT

Count the number of employees in each department:

```
SELECT e.department_id, d.department_name, COUNT(salary) AS TotalEmployee
FROM employees AS e
INNER JOIN departments AS d ON e.department_id = d.department_id
GROUP BY e.department_id, d.department_name;
```

Example 4: AVG

Calculate the average salary for each department:

```
SELECT e.department_id, d.department_name, AVG(salary) as department_AvgSalary
FROM employees AS e
INNER JOIN departments AS d ON e.department_id = d.department_id
GROUP BY e.department_id, d.department_name;
```

Example 5: MIN

Find the minimum salary in each department:

```
SELECT e.department_id, d.department_name, MIN(salary) as department_MinSalary
FROM employees AS e
INNER JOIN departments AS d ON e.department_id = d.department_id
GROUP BY e.department_id, d.department_name;
```

Example 6: MAX

Find the maximum salary in each department:

```
SELECT e.department_id, d.department_name, MAX(salary) as department_MaxSalary
FROM employees AS e
INNER JOIN departments AS d ON e.department_id = d.department_id
GROUP BY e.department_id, d.department_name;
```

HAVING Clause

Filters groups based on a condition applied to aggregate function results.

Syntax:

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1
HAVING condition;
```

Example 1: HAVING with COUNT

Find departments with more than 1 employee:

```
SELECT d.department_id, department_name, COUNT(salary) AS total_employees
FROM employees AS e
JOIN departments AS d ON e.department_id = d.department_id
GROUP BY d.department_id, d.department_name
HAVING COUNT(e.department_id) >= 2;
```

Example 2: HAVING with SUM

Find departments where the total salary is greater than 100,000:

```
SELECT d.department_id, d.department_name, SUM(salary) AS TotalSalary
FROM employees AS e
JOIN departments AS d ON e.department_id = d.department_id
GROUP BY d.department_id, d.department_name
HAVING SUM(salary) >= 100000;
```

Example 3: HAVING with AVG

Find departments where the average salary is greater than 10,000:

```
SELECT d.department_id, d.department_name, AVG(salary) AS average_salary
FROM employees AS e
JOIN departments AS d ON e.department_id = d.department_id
GROUP BY d.department_id, d.department_name
HAVING AVG(salary) >= 10000;
```

Example 4: HAVING with MIN

Find departments where the minimum salary is less than 50,000:

```
SELECT d.department_id, d.department_name, MIN(salary) AS min_salary
FROM employees AS e
JOIN departments AS d ON e.department_id = d.department_id
GROUP BY d.department_id, d.department_name
HAVING MIN(salary) < 50000;
```

Example 5: HAVING with MAX

Find departments where the maximum salary is greater than 50,000:

```
SELECT d.department_id, d.department_name, MAX(salary) AS max_salary
FROM employees AS e
JOIN departments AS d ON e.department_id = d.department_id
GROUP BY d.department_id, d.department_name
HAVING MAX(salary) > 50000;
```

Set Operations

Sample Tables Setup

```
CREATE TABLE students (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL
);

CREATE TABLE teachers (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL
);

INSERT INTO teachers (name) VALUES
('Alice'), ('Bob'), ('Tamil'), ('Thillai');

INSERT INTO students (name) VALUES
('Alice'), ('Bob'), ('sabi'), ('sathish');
```

UNION

Combines results of two or more SELECT queries into a single result set, removing duplicates:

```
SELECT * FROM teachers
UNION
SELECT * FROM students;
```

With duplicates:

```
SELECT * FROM teachers
UNION ALL
SELECT * FROM students;
```

INTERSECT

Returns common rows that appear in both SELECT queries:

```
SELECT * FROM teachers
INTERSECT
SELECT * FROM students;
```

EXCEPT

Returns rows from the first SELECT query that are not present in the second:

```
SELECT * FROM teachers
EXCEPT
SELECT * FROM students;
```

Summary

- **UNION**: Combines results from multiple queries and removes duplicates
- **INTERSECT**: Returns common rows from multiple queries
- **EXCEPT**: Returns rows from the first query that are not in the second query

---

Modifying Data

Create a Table

```
CREATE TABLE worker
(
    id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    position VARCHAR(50)
);
```

INSERT

Insert a single row:

```
INSERT INTO worker (name, position) VALUES ('Abi', 'Software Engineer');
```

Insert multiple rows:

```
INSERT INTO worker (name, position) VALUES
('Dharshini', 'Data Scientist'),
('Thilai', 'Product Manager'),
('Tamizh', 'CEO');
```

UPDATE

Update existing rows:

```
UPDATE worker SET position = 'Senior Software Engineer' WHERE id = 1;
```

DELETE

Delete rows:

```
DELETE FROM worker WHERE id = 3;
```

UPSERT

Insert or update if conflict:

```
INSERT INTO worker (name, position)
VALUES ('Thillai', 'Team Leader')
ON CONFLICT(name)
DO UPDATE SET position = EXCLUDED.position;
```

Note: The ON CONFLICT clause requires a unique identifier to detect conflicts. If you try to use ON CONFLICT without a primary key or unique constraint, PostgreSQL will throw an error.