#PostgreSQL#

Section 9. Modifying Data
In this section, you will learn how to insert data into a table with the INSERT statement, modify existing data with the UPDATE statement, and remove data with the DELETE statement. Additionally, you will learn how to use the UPSERT statement to merge data.

Insert – guide you on how to insert a single row into a table.
Insert multiple rows – show you how to insert multiple rows into a table.
Update – update existing data in a table.
Update join – update values in a table based on values in another table.
Delete – delete data in a table.
Upsert – insert or update data if the new row already exists in the table.

Can You Teach Me above all the topics with Realtime example (create a simple table and insert some values) and show output in Table Format and Also Explain Each topic why this needed, what happen, like syntax and so on.

Let's dive into each topic one by one with real-time examples and explanations.

---

1. INSERT (Single Row)

Why Needed:

The `INSERT` statement is used to add new data into a table. This is the fundamental step for populating tables with meaningful data.

Syntax:

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

Example:

Create a table and insert a single row.

```sql
-- Step 1: Create a simple table
CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
```

```
    position VARCHAR(100),
    salary NUMERIC
);

-- Step 2: Insert a single row
INSERT INTO employees (name, position, salary)
VALUES ('John Doe', 'Software Developer', 60000);
```

Output (Table Format):

```
plaintext


 id |    name     |       position       | salary
----+-------------+----------------------+--------
  1 | John Doe    | Software Developer   | 60000
```

---

2. INSERT (Multiple Rows)

Why Needed:

To quickly add multiple records into a table in a single statement, improving efficiency and reducing redundant code.

Syntax:

```
sql


INSERT INTO table_name (column1, column2, ...) VALUES (value1, value2, ...), (value1, value
2, ...), ...;
```

Example:

Insert multiple rows.

```
sql


INSERT INTO employees (name, position, salary)
VALUES
    ('Jane Smith', 'Product Manager', 75000),
    ('Alice Brown', 'Data Analyst', 50000),
    ('Bob Green', 'System Administrator', 55000);
```

Output (Table Format):

```
plaintext


 id |    name     |        position        | salary
----+-------------+------------------------+--------
  1 | John Doe    | Software Developer     | 60000
```

```
  2 | Jane Smith  | Product Manager        | 75000
  3 | Alice Brown | Data Analyst           | 50000
  4 | Bob Green   | System Administrator   | 55000
```

---

3. UPDATE

Why Needed:

The `UPDATE` statement modifies existing records in the table. It's useful for correcting or altering data.

Syntax:

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

Example:

Update the salary of an employee.

```sql
UPDATE employees
SET salary = 65000
WHERE name = 'Alice Brown';
```

Output (Table Format):

```plaintext
 id |    name     |        position        | salary
----+-------------+------------------------+--------
  1 | John Doe    | Software Developer     | 60000
  2 | Jane Smith  | Product Manager        | 75000
  3 | Alice Brown | Data Analyst           | 65000
  4 | Bob Green   | System Administrator   | 55000
```

---

4. UPDATE with JOIN

Why Needed:

To update a table based on related data in another table, useful in complex database systems.

Example:

Create another table and update using a join.

```sql
-- Create a new table for updates
CREATE TABLE bonuses (
    emp_name VARCHAR(100),
    bonus NUMERIC
);

-- Insert data into bonuses table
INSERT INTO bonuses (emp_name, bonus)
VALUES ('Alice Brown', 5000), ('Bob Green', 3000);

-- Update salary with bonuses
UPDATE employees
SET salary = salary + bonuses.bonus
FROM bonuses
WHERE employees.name = bonuses.emp_name;
```

Output (Table Format):

```plaintext
 id |    name     |        position         | salary
----+-------------+-------------------------+--------
  1 | John Doe    | Software Developer      | 60000
  2 | Jane Smith  | Product Manager         | 75000
  3 | Alice Brown | Data Analyst            | 70000
  4 | Bob Green   | System Administrator    | 58000
```

---

5. DELETE

Why Needed:

The `**DELETE**` statement removes unwanted or obsolete data from the table.

Syntax:

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

Example:

Remove an employee.

```sql
DELETE FROM employees WHERE name = 'Bob Green';
```

Output (Table Format):

```
plaintext


 id |    name     |        position        | salary
----+-------------+------------------------+--------
  1 | John Doe    | Software Developer     | 60000
  2 | Jane Smith  | Product Manager        | 75000
  3 | Alice Brown | Data Analyst           | 70000
```

---

6. UPSERT

Why Needed:

An `**UPSERT**` operation either inserts a new row or updates the existing row if a conflict occurs. This is helpful for maintaining unique constraints.

Syntax:

```
sql


  INSERT INTO table_name (column1, column2, ...)
  VALUES (value1, value2, ...)
  ON CONFLICT (column)
  DO UPDATE SET column1 = value1, column2 = value2, ...;
```

Example:

Insert or update data.

```
sql


  INSERT INTO employees (id, name, position, salary)
  VALUES (3, 'Alice Brown', 'Senior Data Analyst', 75000)
  ON CONFLICT (id)
  DO UPDATE SET position = 'Senior Data Analyst', salary = 75000;
```

Output (Table Format):

```
plaintext


 id |    name     |         position        | salary
----+-------------+-------------------------+--------
  1 | John Doe    | Software Developer      | 60000
  2 | Jane Smith  | Product Manager         | 75000
  3 | Alice Brown | Senior Data Analyst     | 75000
```

Would you like further explanations or additional examples for any of these topics?