Week 3 Notes - DBMS

Prof. Partha Pratham Das, IIT KGP

Notes by Adarsh (23f2003570)

L3.1 SQL Samples (28:26)

Retrieve names of buildings that has classrooms with maximum capacity UTMOST 100

```
SELECT DISTINCT building_name
FROM classroom
WHERE capacity < 100
```

Cartesian Product example

```
SELECT name, budget
FROM student, department
WHERE student.dept_name = department.dept_name
        AND budget < 5000
```

This generates ALL combination of student department pairs and applies the 2 filters

Rename Feature (AS operator)

```
SELECT S.name AS student_name, budget AS dept_budget
FROM student AS S, department AS D
WHERE S.dept_name = D.dept_name
        AND budget < 5000
```

String operations

1. Starts with 3 alphabets

```
SELECT title
FROM course
WHERE course.id LIKE `___%`
```

Order By

```sql
SELECT name, dept_name, total_cred
FROM student
ORDER BY dept_name ASC, total_cred DESC;
```

First sort by dept_name and within those results sort by total_cred

Intersect and IN operator

```sql
SELECT name
FROM instructor
WHERE dept_name IN ('ComSci', 'Maths')
INTERSECT
SELECT name
FROM instructor
WHERE salary < 50000
```

- You can use AND clause to make this neater
- Selects all instructors from comp-sci and maths who have salary < 50K

Aggregate Functions

```sql
SELECT building, AVG(capacity)
FROM classroom
GROUP BY building
HAVING AVG(capacity) > 25
```

1. MAX
2. AVG
3. MIN
4. COUNT
5. SUM

Terminology

Pure and Multi Set

In **Database Management Systems (DBMS)**, a **pure set** is not a standard or widely used term, but it can be interpreted in the context of **set theory** as applied to databases.

In databases, sets are commonly used to describe collections of data, particularly when discussing relational databases. Here's how "pure set" might be understood:

1. Set Theory in DBMS:

- A **set** in DBMS is a collection of distinct elements (tuples or rows) where there are no duplicates. The set is **unordered**, and the elements in the set are unique.
- Operations like **Union**, **Intersection**, and **Difference** in relational algebra rely on set theory principles.

2. Pure Set:

- A **pure set** could be seen as a set that adheres strictly to the properties of a mathematical set. This would imply:
  - No duplicates (all elements or rows are unique).
  - The set is **unordered** (the order of rows does not matter).
  - All elements belong to a single, well-defined set (i.e., no nested or mixed sets).

3. Comparison with Bags (Multisets):

- In contrast to a **pure set**, a **bag** or **multiset** allows duplicates. In SQL, for example, the default result of a `SELECT` query is a bag (because duplicates are allowed unless explicitly removed using `DISTINCT`).

Example:

- If you have a table with rows `[1, 2, 3, 2]`, a **pure set** would contain only `[1, 2, 3]` (no duplicates), while a bag would allow `[1, 2, 3, 2]`.

In short, a **pure set** in DBMS would be a collection of distinct records, aligning with strict set theory principles.

Theta Join

A **theta join** is a type of join that uses a general condition to combine rows from two tables. Unlike an **equi-join**, which only uses equality (`=`) in its condition, a theta join can use any comparison operator like `<`, `>`, `<=`, `>=`, `!=`, or `=`, among others.

In SQL, this is often implemented using a `JOIN` clause with a specific condition in the `ON` clause.

Example of a Theta Join in PostgreSQL:

```
SELECT *
FROM table1
JOIN table2
ON table1.column1 < table2.column2;
```

In this example, the rows are joined based on the condition that `table1.column1` is less than `table2.column2`. This is a theta join because it uses the `<` operator instead of just equality.

Key Points:

- A **theta join** allows comparison with any relational operator, not just equality.
- It's flexible and can handle more complex relationships between tables.

L3.2 Intermediate SQL /1 (33:06)

Nested sub queries

1. Every output of a query is an relation
2. Every input of a query is one or more relations
3. An attribute in a query can be replaced with a query that generates a single value

```
SELECT a1, a2, a3...
FROM r1, r2,...
WHERE P
```

1. a1, a2... can be replaced by sub-query that generates single value
2. r1 can be replaced by any valid subquery
3. P can be replaced by B `<op>` sub-query
    i. op is like in, not in
    ii. B is the attribute/field name

```
SELECT DISTINCT course_id
FROM section
WHERE semester='Fall' AND year=2009
    AND course_id IN (SELECT course_id FROM section where
                        semester='Spring' AND year=2010);
```

Nested subqueries in SQL involve placing one query inside another query. They are used to perform operations that depend on the results of another query.

Example 1: Basic Usage

Find employees who earn more than the average salary in their department:

```
SELECT EmployeeID, Name
FROM Employees e
WHERE Salary > (SELECT AVG(Salary)
                FROM Employees
                WHERE Department = e.Department);
```

Here, the inner subquery calculates the average salary for each department, and the outer query retrieves employees who earn more than that average.

Example 2: Multiple Levels

Find employees who work in the same department as employees with the highest salary:

```
SELECT Name
FROM Employees
WHERE Department = (SELECT Department
                    FROM Employees
                    WHERE Salary = (SELECT MAX(Salary)
                                    FROM Employees));
```

The innermost subquery finds the highest salary, the middle subquery finds the department with that highest salary, and the outer query retrieves employees in that department.

SOME Operator

```
SELECT name
FROM instructor
WHERE salary > SOME (SELECT salary FROM instructor
                        WHERE dept_name = 'biology');
```

The `SOME` operator is used in a subquery to compare a value to any value returned by the subquery. It is functionally equivalent to `ANY`. The `SOME` operator allows you to check if a condition is true for at least one of the values returned by the subquery.

For example:

```
SELECT *
FROM Employees
WHERE Salary > SOME (SELECT Salary
                     FROM Employees
                     WHERE Department = 'Sales');
```

This query selects employees whose salary is greater than at least one salary from the Sales department.

Mathematically

1. F <comparison> *some* $r \iff \forall\, t \in r$
2. <comparison> are $<, \leq, >, \geq, =, \neq$

ALL Operator

ALL operators will match ALL results of a subquery

EXIST

1. exists means atleast one match exists
2. not exists

UNIQUE

Sir talks about unique construct. 1. returns TRUE is there are no dupes 2. FALSE otherwise

```
SELECT T.course_id
FROM course as T
WHERE UNIQUE (SELECT R.course_id
                FROM section as R
                WHERE T.course_id = R.course_id
                AND R.year = 2009);
```

WITH Clause

The `WITH` clause, also known as a Common Table Expression (CTE), allows you to define a temporary result set that can be referenced within a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement. CTEs improve query readability and can simplify complex queries by breaking them into manageable parts.

Syntax:

```
WITH CTE_Name AS (
    -- CTE query
    SELECT column1, column2
    FROM table_name
    WHERE condition
)
-- Main query using the CTE
SELECT column1
FROM CTE_Name
WHERE another_condition;
```

Example:

Find employees with salaries above the average salary of their department:

```
WITH AvgSalary AS (
    SELECT Department, AVG(Salary) AS AverageSalary
    FROM Employees
    GROUP BY Department
)
SELECT e.EmployeeID, e.Name, e.Salary
FROM Employees e
```

```
JOIN AvgSalary a
ON e.Department = a.Department
WHERE e.Salary > a.AverageSalary;
```

In this example, the `WITH` clause defines a CTE named `AvgSalary` to calculate the average salary per department. The main query then joins the `Employees` table with this CTE to find employees earning more than the average salary in their respective departments.

1. WITH defines a temporary relation available only in the query

Scalar Sub Query

A scalar subquery is a subquery that returns a single value (one row and one column). It can be used in contexts where a single value is expected, such as in comparisons or assignments.

Syntax:

```
SELECT column1
FROM table_name
WHERE column2 = (SELECT scalar_value
                 FROM another_table
                 WHERE condition);
```

Examples:

1. **Find Employees Earning More Than the Highest Salary in a Department:**

   Suppose you want to find employees who earn more than the highest salary in a particular department. The scalar subquery retrieves the highest salary for that department.

   ```
   SELECT EmployeeID, Name
   FROM Employees
   WHERE Salary > (SELECT MAX(Salary)
                   FROM Employees
                   WHERE Department = 'Sales');
   ```

   Here, the scalar subquery `(SELECT MAX(Salary) FROM Employees WHERE Department = 'Sales')` returns a single value, the maximum salary in the 'Sales' department. This value is then used to filter employees with higher salaries.

2. **Find Products More Expensive Than the Average Price:**

   To find products that cost more than the average price across all products:

   ```
   SELECT ProductName
   FROM Products
   ```

```
WHERE Price > (SELECT AVG(Price)
               FROM Products);
```

In this case, the scalar subquery `(SELECT AVG(Price) FROM Products)` returns the average price of all products. The main query uses this single value to filter products priced above this average.

Scalar subqueries are often used in `WHERE` clauses, but they can also appear in `SELECT` clauses, `HAVING` clauses, and `SET` clauses where a single value is needed.

Gives a runtime error if subquery returns more than 1 result

Sub Queries in PostgreSQL

In PostgreSQL, subqueries are queries nested inside other queries. They can be used in various places within the main query and offer flexibility to perform complex filtering, transformations, and operations on data. There are several types of subqueries, each serving a different purpose. Below are the main types of subqueries with examples:

1. Scalar Subqueries

A scalar subquery returns a single value (i.e., a single row with a single column). These can be used where a single value is expected, such as in the `SELECT` clause or a condition in the `WHERE` clause.

Example:

```
SELECT
    name,
    (SELECT AVG(salary) FROM employees) AS avg_salary
FROM employees;
```

Here, the subquery `(SELECT AVG(salary) FROM employees)` returns the average salary from the `employees` table, and this scalar value is used in the main query for each row.

2. Column Subqueries

A column subquery returns a single column of data, which can be used in the `WHERE` clause to filter rows based on the returned column.

Example:

```
SELECT name
FROM employees
WHERE department_id IN (SELECT id FROM departments WHERE location = 'New York');
```

This subquery `(SELECT id FROM departments WHERE location = 'New York')` returns a list of `department_id` s from departments located in New York. The main query then selects all employees who belong to any of those departments.

3. Row Subqueries

A row subquery returns a single row of data, which can be compared using a row constructor.

Example:

```
SELECT id, name
FROM employees
WHERE (department_id, salary) = (SELECT department_id, MAX(salary) FROM employees);
```

This query finds the employee with the highest salary in the entire company by comparing the `(department_id, salary)` pair with the subquery result, which returns a single row.

4. Table Subqueries

A table subquery returns a full table (multiple rows and columns) and is often used in the `FROM` clause. It essentially acts like a temporary table for the main query.

Example:

```
SELECT sub.department_id, AVG(sub.salary)
FROM (SELECT department_id, salary FROM employees WHERE salary > 50000) AS sub
GROUP BY sub.department_id;
```

In this example, the subquery `(SELECT department_id, salary FROM employees WHERE salary > 50000)` creates a temporary table of employees with salaries greater than 50,000. The main query then calculates the average salary by department.

5. Correlated Subqueries

A correlated subquery depends on the outer query for its values. For each row in the outer query, the subquery is executed.

Example:

```
SELECT e1.name, e1.salary
FROM employees e1
WHERE e1.salary > (SELECT AVG(e2.salary) FROM employees e2 WHERE e2.department_id = e1.depar
```

In this example, the subquery `(SELECT AVG(e2.salary) FROM employees e2 WHERE e2.department_id = e1.department_id)` calculates the average salary for each department, and the outer query compares each employee's salary to the average salary of their department. This is called a correlated subquery because the inner query depends on the `department_id` from the outer query (`e1.department_id`).

6. Exists Subqueries

An `EXISTS` subquery checks whether the subquery returns any rows. It returns `TRUE` if the subquery returns one or more rows and `FALSE` if no rows are returned. This is useful for checking conditions.

Example:

```
SELECT name
FROM employees e
WHERE EXISTS (SELECT 1 FROM departments d WHERE d.id = e.department_id AND d.location = 'New
```

Here, the `EXISTS` subquery checks if there are any departments in New York that match the employee's `department_id` . If such a department exists, the employee is selected.

7. NOT EXISTS Subqueries

A `NOT EXISTS` subquery is the opposite of `EXISTS` . It checks whether the subquery returns no rows, and the outer query will return rows if the subquery does not return any.

Example:

```
SELECT name
FROM employees e
WHERE NOT EXISTS (SELECT 1 FROM projects p WHERE p.employee_id = e.id);
```

This query returns employees who are not assigned to any projects. The subquery `(SELECT 1 FROM projects p WHERE p.employee_id = e.id)` returns rows if the employee is assigned to any project. The `NOT EXISTS` ensures that only employees without projects are selected.

8. WITH (Common Table Expression or CTE) Subqueries

A Common Table Expression (CTE) is essentially a named subquery that is written before the main query using the `WITH` keyword. CTEs are useful for improving query readability, especially for complex queries.

Example:

```sql
WITH department_avg AS (
    SELECT department_id, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY department_id
)
SELECT e.name, e.salary, da.avg_salary
FROM employees e
JOIN department_avg da ON e.department_id = da.department_id
WHERE e.salary > da.avg_salary;
```

In this example, the `WITH` clause creates a CTE named `department_avg`, which calculates the average salary by department. The main query then joins the `employees` table with the CTE and selects employees whose salary is higher than the department average.

Summary:

- **Scalar Subqueries**: Return a single value.
- **Column Subqueries**: Return a single column.
- **Row Subqueries**: Return a single row.
- **Table Subqueries**: Return multiple rows and columns (a table).
- **Correlated Subqueries**: Subquery depends on the outer query.
- **Exists Subqueries**: Check if subquery returns rows.
- **NOT EXISTS Subqueries**: Check if subquery does not return rows.
- **WITH (CTE) Subqueries**: Define reusable named subqueries for complex queries.

Modification of Databases

DELETE

```sql
DELETE FROM instructors

DELETE FROM instructors
    WHERE dept_name = 'Finance'

DELETE FROM instructors
    WHERE dept_name IN (SELECT dept_name FROM department
                            WHERE building_name = 'Watson`)
```

Delete Changes the state of the Table

```sql
```sql
DELETE FROM instructors
    WHERE salary < (SELECT AVG(salary) FROM instructors)
```
```

```
1. if delete happens, the average salary changes
```

INSERT

1. Always use named attributes/fields.. the order might change
2. Values in order of named attributes

Copy from one table to another

```
INSERT INTO table1
    SELECT * FROM table2
```

Basic Syntax:

1. Insert Data into All Columns:

```
INSERT INTO table_name (column1, column2, column3)
VALUES (value1, value2, value3);
```

2. Insert Data into Specific Columns:

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

Examples:

1. Insert Data into All Columns:

   Suppose you have a `Users` table with columns `UserID`, `UserName`, and `Email`.

   ```
   INSERT INTO Users (UserID, UserName, Email)
   VALUES (1, 'Alice', 'alice@example.com');
   ```

   This query inserts a new row into the `Users` table with `UserID` 1, `UserName` 'Alice', and `Email` 'alice@example.com'.

2. Insert Data into Specific Columns:

   If you only want to insert values into `UserName` and `Email`, assuming `UserID` is an auto-incrementing primary key:

   ```
   INSERT INTO Users (UserName, Email)
   ```

```
VALUES ('Bob', 'bob@example.com');
```

This query inserts a new row with `UserName` 'Bob' and `Email` 'bob@example.com', letting the database automatically assign a `UserID`.

3. **Insert Multiple Rows:**

   You can also insert multiple rows in a single query:

```
INSERT INTO Users (UserName, Email)
VALUES ('Charlie', 'charlie@example.com'),
       ('Diana', 'diana@example.com');
```

   This query inserts two new rows into the `Users` table.

4. **Insert Data from Another Table:**

   You can insert data into a table based on a `SELECT` statement from another table:

```
INSERT INTO ArchiveUsers (UserID, UserName, Email)
SELECT UserID, UserName, Email
FROM Users
WHERE RegistrationDate < '2023-01-01';
```

   This query copies rows from the `Users` table into the `ArchiveUsers` table where the `RegistrationDate` is before January 1, 2023.

UPDATE

Conditional Updates with case

```
UPDATE instructor
    SET salary = CASE
        WHEN salary <= 1000
        THEN salary * 1.05
        ELSE salary * 1.03
        END
```

Basic Syntax:

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

- **`table_name`** : The name of the table you want to update.

- `SET` : Specifies the columns to be updated and their new values.
- `WHERE` : Defines which rows should be updated. If omitted, all rows in the table will be updated.

Examples:

1. **Update a Single Record:**

   Suppose you have a `Users` table and you want to update the email address of a user with a specific `UserID` .

   ```
   UPDATE Users
   SET Email = 'newemail@example.com'
   WHERE UserID = 1;
   ```

   This query changes the `Email` of the user with `UserID` 1 to 'newemail@example.com'.

2. **Update Multiple Columns:**

   To update multiple columns for a specific user:

   ```
   UPDATE Users
   SET UserName = 'Alice Smith', Email = 'alice.smith@example.com'
   WHERE UserID = 1;
   ```

   This updates both the `UserName` and `Email` of the user with `UserID` 1.

3. **Update Multiple Records:**

   To increase the salary for all employees in a particular department:

   ```
   UPDATE Employees
   SET Salary = Salary * 1.10
   WHERE Department = 'Sales';
   ```

   This query gives a 10% raise to all employees in the 'Sales' department.

4. **Update Using a Subquery:**

   Suppose you want to update a table based on values from another table. For instance, update the `Salary` in the `Employees` table to match the average salary from a `Salaries` table:

   ```
   UPDATE Employees
   SET Salary = (SELECT AVG(Salary)
                 FROM Salaries
                 WHERE Department = Employees.Department)
   ```

```
WHERE Department IN (SELECT Department
                     FROM Salaries);
```

This query updates each employee's salary to the average salary for their department.

5. **Update All Records (Use with Caution):**

   To set a default value for a column in all records:

```
UPDATE Users
SET Status = 'Active';
```

   This query sets the `Status` column to 'Active' for all rows in the `Users` table.

Important Notes:

- Always use the `WHERE` clause to specify which rows to update. Without it, the update will apply to all rows in the table.
- Be cautious when performing updates, especially when updating multiple rows, to avoid unintended data changes.

SQL CASE STATEMENT

The `CASE` statement in SQL provides a way to perform conditional logic within a query. It allows you to return different values based on specified conditions, similar to if-else logic in programming.

Basic Syntax:

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE default_result
END
```

- **`WHEN condition`** : Defines a condition to be evaluated.
- **`THEN result`** : Specifies the result to return if the corresponding `WHEN` condition is true.
- **`ELSE default_result`** (optional): Provides a default result if none of the `WHEN` conditions are met.
- **`END`** : Marks the end of the `CASE` statement.

Examples:

1. **Simple Case Expression:**

   Categorize employees based on their salary:

```
SELECT EmployeeID, Salary,
       CASE
           WHEN Salary < 30000 THEN 'Low'
           WHEN Salary BETWEEN 30000 AND 60000 THEN 'Medium'
           ELSE 'High'
       END AS SalaryCategory
FROM Employees;
```

This query classifies salaries into 'Low', 'Medium', or 'High' categories and includes this classification as a new column `SalaryCategory` in the result set.

2. **Case in a `SELECT` Statement:**

   Display different messages based on order status:

```
SELECT OrderID, Status,
       CASE Status
           WHEN 'P' THEN 'Pending'
           WHEN 'S' THEN 'Shipped'
           WHEN 'D' THEN 'Delivered'
           ELSE 'Unknown'
       END AS StatusDescription
FROM Orders;
```

   This query translates order status codes into descriptive text.

3. **Case in an `ORDER BY` Clause:**

   Sort employees by a custom ranking based on job title:

```
SELECT EmployeeID, JobTitle
FROM Employees
ORDER BY CASE JobTitle
             WHEN 'Manager' THEN 1
             WHEN 'Team Lead' THEN 2
             WHEN 'Staff' THEN 3
             ELSE 4
         END;
```

   This query sorts employees by job title, giving priority to 'Manager', then 'Team Lead', and so on.

4. **Case in a `WHERE` Clause:**

   Apply different filters based on a condition:

```
SELECT ProductName, Price
FROM Products
WHERE CASE
          WHEN Category = 'Electronics' THEN Price > 100
          WHEN Category = 'Books' THEN Price < 50
          ELSE Price > 20
      END;
```

This query applies different price filters based on the product category.

Important Notes:

- **`CASE` in SQL** can be used in `SELECT`, `ORDER BY`, `WHERE`, and `HAVING` clauses.
- **`CASE` is evaluated in order**: Once a condition is true, subsequent `WHEN` clauses are not evaluated.
- **`ELSE` clause is optional**: If omitted, `CASE` returns `NULL` when no conditions are met.

Update with Scalar sub-queries

The `UPDATE` statement with a scalar subquery allows you to modify records in a table based on a single value returned by a subquery. A scalar subquery is a subquery that returns exactly one row and one column.

Basic Syntax:

```
UPDATE table_name
SET column_name = (SELECT scalar_value
                   FROM another_table
                   WHERE condition)
WHERE condition;
```

In this syntax:

- **`table_name`** : The table you want to update.
- **`column_name`** : The column to be updated.
- **`scalar_value`** : The value returned by the subquery.
- **`another_table`** : The table from which the subquery retrieves the scalar value.
- **`condition`** : The condition for filtering records to be updated.

Examples:

1. **Update Based on a Single Value from Another Table:**

   Suppose you have an `Employees` table and a `Departments` table. You want to update the `Salary` of employees to match the average salary of their department from the `Departments`

table.

```
UPDATE Employees
SET Salary = (SELECT AVG(Salary)
              FROM Employees e2
              WHERE e2.Department = Employees.Department)
WHERE Department IN (SELECT DISTINCT Department
                     FROM Employees);
```

Explanation:

- The scalar subquery `(SELECT AVG(Salary) FROM Employees e2 WHERE e2.Department = Employees.Department)` calculates the average salary for the department of each employee.
- The `UPDATE` statement then sets each employee's salary to this average value where the department matches.

2. **Update with a Scalar Subquery in the Same Table:**

Suppose you have a `Products` table with columns `ProductID`, `Price`, and `Category`. You want to set the `Price` of each product to the average price of products within its own category.

```
UPDATE Products
SET Price = (SELECT AVG(Price)
             FROM Products p2
             WHERE p2.Category = Products.Category)
WHERE Category IN (SELECT DISTINCT Category
                   FROM Products);
```

Explanation:

- The scalar subquery `(SELECT AVG(Price) FROM Products p2 WHERE p2.Category = Products.Category)` computes the average price for each product's category.
- The `UPDATE` query uses this average to set the `Price` for each product.

3. **Update Based on a Constant Value from a Subquery:**

Suppose you want to update a `Discount` column in the `Orders` table with the highest discount value available from a `Discounts` table.

```
UPDATE Orders
SET Discount = (SELECT MAX(DiscountValue)
                FROM Discounts)
WHERE OrderDate < '2024-01-01';
```

Explanation:

- The scalar subquery `(SELECT MAX(DiscountValue) FROM Discounts)` retrieves the highest discount value.
- The `UPDATE` statement applies this discount value to orders placed before January 1, 2024.

Important Points:

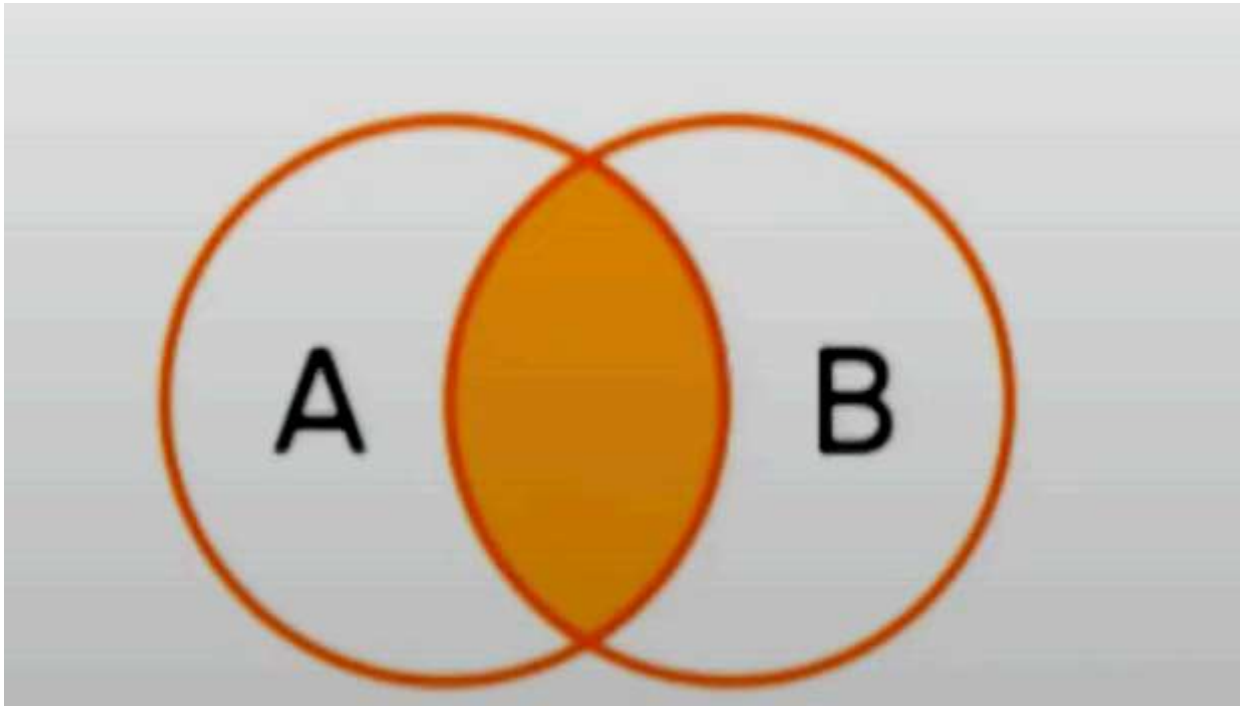
- The scalar subquery must return exactly one value; otherwise, the `UPDATE` statement will fail with an error.
- If the subquery returns more than one value or no value, you'll need to adjust the query to handle such cases, possibly by using aggregate functions or additional conditions.

L3.3 Intermediate SQL/2 (31:58)

Joins

In PostgreSQL (and SQL in general), **joins** are used to combine rows from two or more tables based on a related column between them. There are several types of joins, and each serves a different purpose when querying data. Let's explore the main types of joins with examples:

1. INNER JOIN ($R \bowtie S$)



An `INNER JOIN` returns only the rows where there is a match between the columns in both tables.

Syntax:

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

Example:

Suppose we have two tables:

- **employees**

id	name	department_id
1	Alice	101
2	Bob	102
3	Charlie	103

- **departments**

id	department_name
101	HR
102	Finance

To get employees along with their departments:

```
SELECT employees.name, departments.department_name
FROM employees
INNER JOIN departments
ON employees.department_id = departments.id;
```

Result:

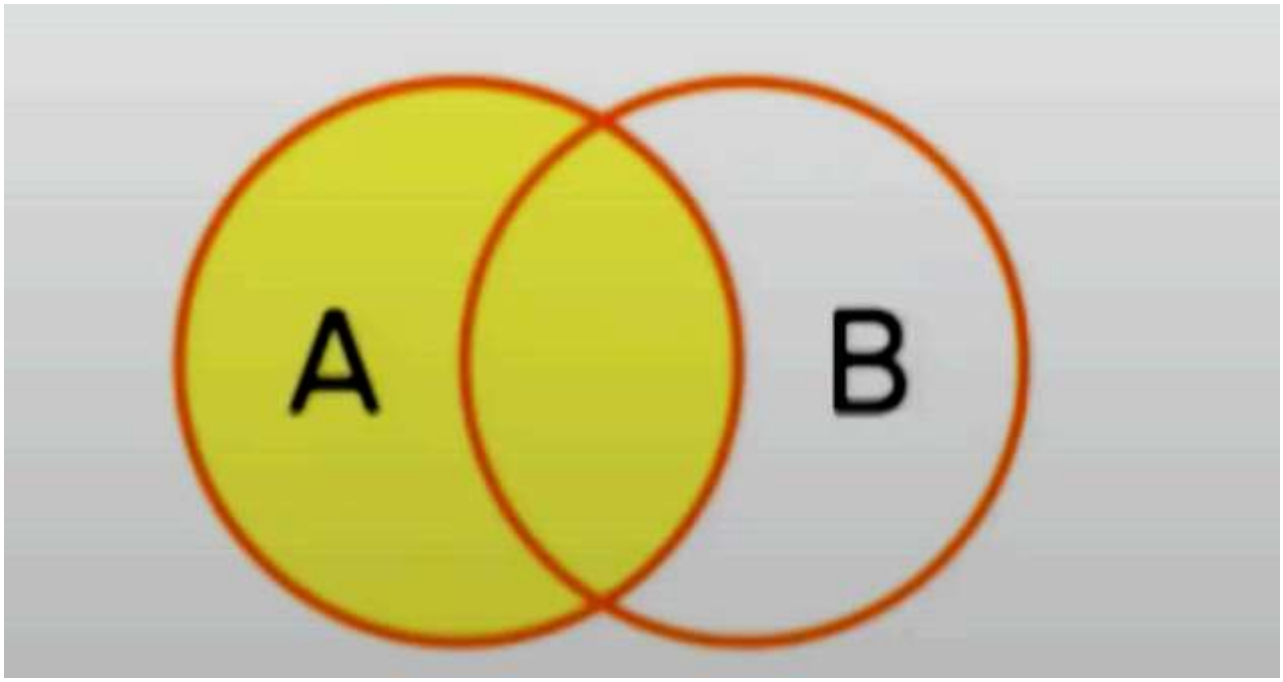
name	department_name
Alice	HR
Bob	Finance

Notice that Charlie, who belongs to department `103`, is excluded because there is no matching department in the `departments` table.

1. Inner Join is a SET INTERSECTION!
2. Equi Join

i. Natural Join skips common columns.

2. LEFT JOIN (LEFT OUTER JOIN) ($R \leftarrow S$)



A `LEFT JOIN` returns all rows from the left table (table1), and the matching rows from the right table (table2). If there is no match, NULL values are returned for columns from the right table.

Syntax:

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.column = table2.column;
```

Example:

Using the same tables as above:

```
SELECT employees.name, departments.department_name
FROM employees
LEFT JOIN departments
ON employees.department_id = departments.id;
```

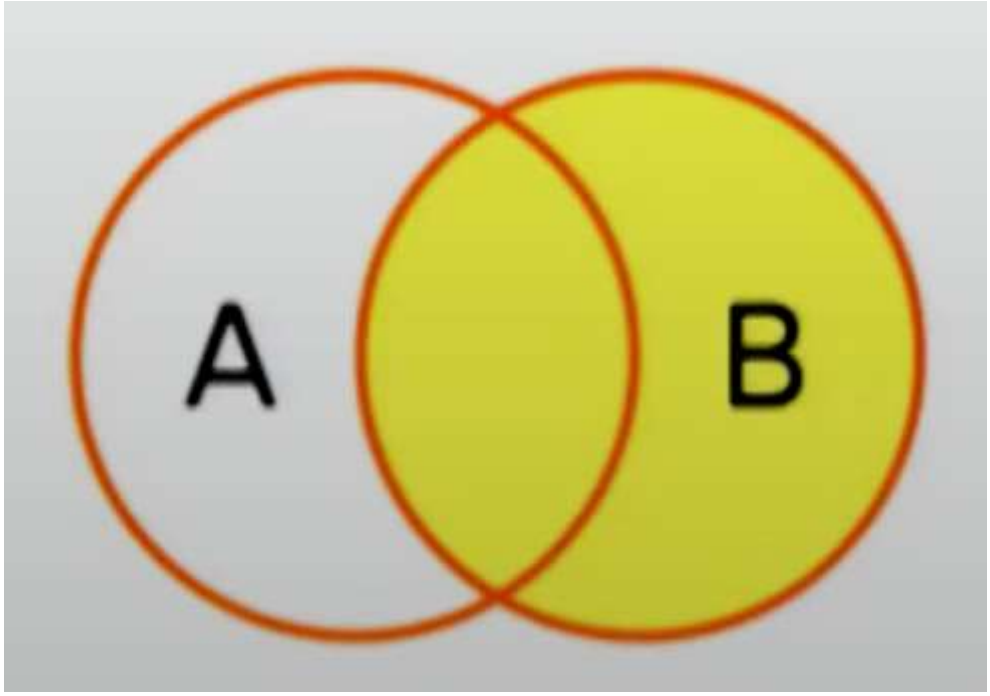
Result:

name	department_name
Alice	HR

name	department_name
Bob	Finance
Charlie	NULL

This time, Charlie is included, but with a `NULL` value for the department name because department `103` doesn't exist.

3. RIGHT JOIN (RIGHT OUTER JOIN) ($R \rightarrow S$)



A `RIGHT JOIN` is the opposite of a left join. It returns all rows from the right table (table2) and the matching rows from the left table (table1). If there is no match, NULL values are returned for columns from the left table.

Syntax:

```
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.column = table2.column;
```

Example:

Let's modify the tables slightly to add an extra department:

- **departments**

id	department_name
101	HR
102	Finance
104	IT

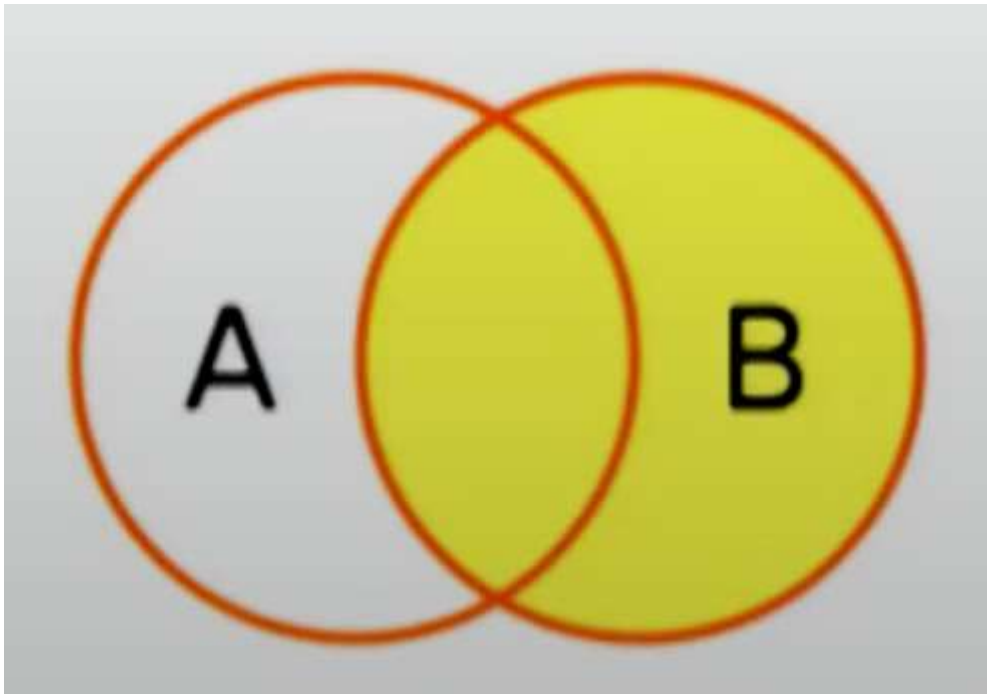
```
SELECT employees.name, departments.department_name
FROM employees
RIGHT JOIN departments
ON employees.department_id = departments.id;
```

Result:

name	department_name
Alice	HR
Bob	Finance
NULL	IT

The IT department appears, even though no employee belongs to it, with a `NULL` value for the employee name.

4. FULL OUTER JOIN ($R \Leftrightarrow S$)



A `FULL OUTER JOIN` returns all rows when there is a match in either table. If there is no match, NULL values will be used for non-matching columns from both tables.

Syntax:

```
SELECT columns
FROM table1
FULL OUTER JOIN table2
ON table1.column = table2.column;
```

Example:

```
SELECT employees.name, departments.department_name
FROM employees
FULL OUTER JOIN departments
ON employees.department_id = departments.id;
```

Result:

name	department_name
Alice	HR
Bob	Finance
Charlie	NULL
NULL	IT

This query returns all employees and all departments, even if there are no matches. Charlie has no matching department, and the IT department has no matching employee.

5. CROSS JOIN ($R \times S$)

A `CROSS JOIN` returns the Cartesian product of the two tables, meaning it combines all rows from the first table with all rows from the second table. Output Row size will be $m \times n$

Syntax:

```
SELECT columns
FROM table1
CROSS JOIN table2;
```

```
SELECT columns
FROM table1, table2;
```

Example:

```
SELECT employees.name, departments.department_name
FROM employees
CROSS JOIN departments;
```

Result:

name	department_name
Alice	HR
Alice	Finance
Alice	IT
Bob	HR
Bob	Finance
Bob	IT
Charlie	HR
Charlie	Finance
Charlie	IT

Each employee is paired with every department.

6. SELF JOIN ($R \bowtie S$)

A `SELF JOIN` is a regular join, but the table is joined with itself. This is useful when you need to compare rows within the same table.

Example:

Suppose you have an employees table where each employee reports to another employee:

- **employees**

id	name	manager_id
1	Alice	NULL

id	name	manager_id
2	Bob	1
3	Charlie	1

You can join the table with itself to find out who reports to whom:

```
SELECT e1.name AS employee, e2.name AS manager
FROM employees e1
LEFT JOIN employees e2
ON e1.manager_id = e2.id;
```

Result:

employee	manager
Alice	NULL
Bob	Alice
Charlie	Alice

Here, we see that Bob and Charlie report to Alice.

Examples of JOINS on the UNIVERSITY DATASET

EQUI-JOIN

This is an `INNER JOIN` NOTE: There are 2 `dept_name` columns

```
university=# SELECT * FROM department d INNER JOIN instructor AS i ON d.dept_name = i.dept_n
  dept_name  | building |  budget   |  id   |   name   |  dept_name  |  salary
-------------+----------+-----------+-------+----------+-------------+-----------
 Cybernetics | Mercer   | 794541.46 | 63395 | McKinnon | Cybernetics |  94333.99
 Statistics  | Taylor   | 395051.74 | 78699 | Pingr    | Statistics  |  59303.62
 Marketing   | Lambeau  | 210627.58 | 96895 | Mird     | Marketing   | 119921.41
(3 rows)
```

NATURAL JOIN ($R \bowtie S$)

Note there is only one `dept_name` column

```
university=# SELECT * FROM department d NATURAL JOIN instructor AS i LIMIT 3;
  dept_name  | building |  budget   |  id   |   name   |  salary
-------------+----------+-----------+-------+----------+-----------
 Cybernetics | Mercer   | 794541.46 | 63395 | McKinnon |  94333.99
 Statistics  | Taylor   | 395051.74 | 78699 | Pingr    |  59303.62
 Marketing   | Lambeau  | 210627.58 | 96895 | Mird     | 119921.41
(3 rows)
```

LEFT OUTER JOIN

```
university=# SELECT * FROM department d LEFT OUTER JOIN instructor AS i ON d.dept_name = i.d
 dept_name  |  building   |  budget   |  id   |   name   | dept_name |  salary
------------+-------------+-----------+-------+----------+-----------+----------
 Civil Eng. | Chandler    | 255041.46 |       |          |           |
 Biology    | Candlestick | 647610.55 | 80759 | Queiroz  | Biology   | 45538.32
 Biology    | Candlestick | 647610.55 | 81991 | Valtchev | Biology   | 77036.18
(3 rows)
```

Notes on Join using the University Dataset

- Rows in department: 20
- Rows in instructor: 50
- Rows in Inner Join: 50 (Note this is specific, if we set up tables where dupe-tupes are allowed what happens?)
- Rows in Left Outer Join: 53 (How did you get 53?)
- Rows in Right Outer Join: 50 (How did you get 50?)
- Rows in Full Outer Join: 53 (Why is it the same as a LEFT OUTER JOIN?)
- Rows in Natural Join: 50 Rows.
- Rows in Cross Join: 1000 Rows (50×20)

Views

In PostgreSQL, a **view** is a virtual table that is the result of a query. Unlike a table, a view does not store data itself. Instead, it provides a way to look at data from one or more tables through a predefined query. Views can be used to simplify complex queries, enhance security by limiting access to specific rows or columns, or present data in a specific format.

Key Characteristics of Views:

- **Virtual table**: A view acts like a table but does not store the data physically.

- **Reusable query**: The query behind the view is stored, and the view can be queried just like a regular table.
- **Updatable (in some cases)**: Some views allow data modification (inserts, updates, deletes), depending on the complexity of the view.

Benefits of Using Views:

1. **Simplicity**: You can hide the complexity of large queries by creating a view with a simple query interface.
2. **Security**: Views can restrict access to certain columns or rows of a table, enhancing security.
3. **Reusability**: Once a view is created, you can reuse it in multiple queries without rewriting the underlying logic.
4. **Logical Data Abstraction**: Views help abstract the complexity of your database schema by providing a logical interface.

Creating Views in PostgreSQL

1. Basic View Creation

To create a view, you use the `CREATE VIEW` statement followed by a query that defines the view. Here is an example:

```
CREATE VIEW employee_view AS
SELECT id, name, salary
FROM employees;
```

This creates a view named `employee_view` that selects the `id`, `name`, and `salary` columns from the `employees` table.

Usage:

You can query this view as if it were a regular table:

```
SELECT * FROM employee_view;
```

2. Creating Views with Joins

Views can combine data from multiple tables using `JOIN` operations. Here's an example of a more complex view:

```
CREATE VIEW employee_department_view AS
SELECT e.id, e.name, d.department_name
```

```
FROM employees e
JOIN departments d ON e.department_id = d.id;
```

In this case, `employee_department_view` is a view that joins the `employees` and `departments` tables and shows the employee name along with their respective department name.

3. Views with Filters

You can define views with filtering conditions as well:

```
CREATE VIEW high_salary_employees AS
SELECT id, name, salary
FROM employees
WHERE salary > 50000;
```

This creates a view called `high_salary_employees` that only returns employees with a salary greater than 50,000.

4. Updatable Views

Some views are **updatable**, meaning you can perform `INSERT`, `UPDATE`, or `DELETE` operations on them, and those changes will propagate to the underlying base tables.

Example:

If `employee_view` contains all the columns needed to update the `employees` table (e.g., primary key, necessary columns), then you can perform updates through the view.

```
UPDATE employee_view
SET salary = 60000
WHERE id = 3;
```

This will update the `salary` for the employee with `id = 3` in the base `employees` table.

However, not all views are updatable. Views involving `JOIN`, `GROUP BY`, `DISTINCT`, or aggregate functions are typically **not updatable** without additional configurations like triggers or `INSTEAD OF` rules.

Modifying Views

1. Modifying a View (ALTER VIEW)

If you need to change the view definition, you can drop and recreate it, or use `CREATE OR REPLACE VIEW` to modify the view without dropping it:

```
CREATE OR REPLACE VIEW employee_view AS
SELECT id, name, salary, hire_date
FROM employees;
```

This adds the `hire_date` column to the `employee_view`.

2. Dropping a View

To remove a view, use the `DROP VIEW` statement:

```
DROP VIEW employee_view;
```

This deletes the view `employee_view` but does not affect the underlying data in the `employees` table.

Example: Complex View with Aggregation

Let's create a view that calculates the average salary by department:

```
CREATE VIEW department_salary_view AS
SELECT department_id, AVG(salary) AS avg_salary
FROM employees
GROUP BY department_id;
```

This view groups employees by their department and calculates the average salary for each department.

Querying the View:

```
SELECT * FROM department_salary_view;
```

This will return the average salary for each department.

Materialized Views

Unlike regular views, **materialized views** store the result of the query physically. This can be useful for performance reasons, especially for complex or expensive queries, as the data is stored on disk rather than calculated on the fly.

Creating a Materialized View:

```
CREATE MATERIALIZED VIEW employee_materialized_view AS
SELECT id, name, salary
FROM employees;
```

This creates a materialized view that stores the query result on disk.

Refreshing a Materialized View:

Since materialized views store data physically, they can become stale if the underlying data changes. You need to refresh them manually or on a schedule:

```
REFRESH MATERIALIZED VIEW employee_materialized_view;
```

Dropping a Materialized View:

```
DROP MATERIALIZED VIEW employee_materialized_view;
```

Summary of Key Points:

1. **Views** are virtual tables that store query definitions but not data.
2. They help simplify complex queries, improve security, and make querying more efficient by abstracting logic.
3. Views can be based on simple `SELECT` queries, joins, or even filters.
4. **Updatable views** allow data manipulation, but more complex views (involving joins or aggregations) are usually **not updatable**.
5. **Materialized views** store data on disk and must be refreshed when the underlying data changes.

Views offer a powerful way to encapsulate and manage complex queries, making them reusable, modular, and more secure.

Materialized vs Normal View Example

```
CREATE VIEW top_paid AS  SELECT id, name, instructor.dept_name FROM instructor INNER JOIN de
> CREATE VIEW

CREATE MATERIALIZED VIEW top_paid_mat AS  SELECT id, name, instructor.dept_name FROM instruc
name = department.dept_name AND salary > 100000;
> SELECT 12

SELECT * FROM top_paid;
SELECT * FROM top_paid_mat;
```

```
  id   |   name    |  dept_name
-------+-----------+-------------
 96895 | Mird      | Marketing
 50330 | Shuming   | Physics
 74420 | Voronina  | Physics
 37687 | Arias     | Statistics
 6569  | Mingoz    | Finance
 74426 | Kenje     | Marketing
 63287 | Jaekel    | Athletics
 34175 | Bondi     | Comp. Sci.
 48507 | Lent      | Mech. Eng.
 95709 | Sakurai   | English
 90376 | Bietzk    | Cybernetics
 19368 | Wieland   | Pol. Sci.
(12 rows)

UPDATE instructor SET salary = 999 WHERE id = '19368';

select * from top_paid_mat ; -> (12 rows)
```

Materialized Views have to be updated.

Cascaded Views

In PostgreSQL, **nested views** (or **cascaded views**) refer to views that are built upon other views. This means you create a view, and then create another view that queries the first view, and so on. Cascaded views can be useful for simplifying complex queries by breaking them down into logical steps, or for abstracting layers of information.

Key Characteristics of Nested or Cascaded Views:

- **Layered abstraction**: Each view can build upon the previous one, adding more complexity or refining data.
- **Simplified queries**: Complex logic is broken into manageable layers, making queries easier to maintain.
- **Performance impact**: Cascading too many views can introduce performance overhead because each view needs to be evaluated at query time unless using materialized views.

How Nested Views Work:

- **View A** is created from a base query (e.g., from a table).
- **View B** is created from a query that selects from **View A**.
- **View C** can be created from a query that selects from **View B**, and so on.

Example of Nested (Cascaded) Views

Step 1: Create Base View

Suppose you have an `employees` table, and you want to create a base view that selects only employees with a salary greater than 50,000.

```
CREATE VIEW high_salary_employees AS
SELECT id, name, salary, department_id
FROM employees
WHERE salary > 50000;
```

This creates a view called `high_salary_employees` that selects employees with salaries above 50,000.

Step 2: Create a View Based on Another View

Now, you can create another view that adds more information, such as the department name by joining the `high_salary_employees` view with the `departments` table:

```
CREATE VIEW high_salary_employee_details AS
SELECT e.id, e.name, e.salary, d.department_name
FROM high_salary_employees e
JOIN departments d ON e.department_id = d.id;
```

This creates a view called `high_salary_employee_details` that includes the department name along with employee details, pulling data from the `high_salary_employees` view.

Step 3: Create Another Layer of View

You can further create a third view to calculate the average salary of high-paid employees by department:

```
CREATE VIEW avg_high_salary_by_department AS
SELECT department_name, AVG(salary) AS avg_salary
FROM high_salary_employee_details
GROUP BY department_name;
```

This view, `avg_high_salary_by_department`, calculates the average salary of high-paid employees (from the second view) by department.

Querying Cascaded Views

Once you've set up cascaded views, you can query the final view to get the desired results:

```
SELECT * FROM avg_high_salary_by_department;
```

This query will execute and retrieve the average salary of high-paid employees for each department. Even though you're querying the third view, PostgreSQL will evaluate all dependent views in the cascade, combining their logic.

Benefits of Cascaded Views

1. **Modularity and Reusability**: Each view encapsulates a piece of logic. You can reuse views in different queries, reducing duplication.
2. **Simplifying Complex Queries**: Instead of writing a long, complex query with many joins, subqueries, or aggregations, you can break it down into smaller steps (views), making each part easier to understand.
3. **Separation of Concerns**: Each view focuses on a specific task, allowing better organization of logic (e.g., one view handles filtering, another handles joining, and another handles aggregation).

Considerations and Performance Impact of Cascaded Views

- **Performance**: While cascaded views improve query readability and maintainability, they can introduce performance overhead. Every time you query a cascaded view, PostgreSQL has to process each underlying view recursively.
    - If performance becomes a concern, you may want to replace complex cascaded views with **materialized views**, which store the results of the query physically and don't need to be recomputed every time.
- **View Dependencies**: PostgreSQL manages view dependencies automatically. If you try to drop a base view that is used by other views, PostgreSQL will prevent the drop unless you cascade the operation (`DROP VIEW base_view CASCADE`), which also drops all dependent views.
- **Updatability**: Not all views are updatable, and cascading views that involve multiple tables, joins, or aggregate functions are usually not updatable without additional mechanisms like `INSTEAD OF` triggers.

Example of Performance Impact: Nested View vs. Materialized View

If you have deeply nested views that must be queried repeatedly, PostgreSQL has to execute all the layers each time.

Example of Nested View Query:

```
SELECT * FROM avg_high_salary_by_department;
```

This query will involve multiple layers:

1. The first view `high_salary_employees` is evaluated.
2. The second view `high_salary_employee_details` is then evaluated based on the first view.
3. Finally, the third view `avg_high_salary_by_department` computes the result.

Using Materialized Views:

To avoid repeated evaluation, you can convert the third view to a **materialized view**, storing the precomputed result on disk:

```
CREATE MATERIALIZED VIEW avg_high_salary_by_department_mat AS
SELECT department_name, AVG(salary) AS avg_salary
FROM high_salary_employee_details
GROUP BY department_name;
```

Now, when you query the materialized view:

```
SELECT * FROM avg_high_salary_by_department_mat;
```

The results are fetched directly from storage, which improves performance, especially for large datasets.

View Expansion

WIP:

1. Recursive View Expansion is forbidden. A view `View9` cannot refer to `View9` in any sub-view used (WIP: reframe this please)
2. C.R.U.D into a View??? Professor Partha Das says its possible if underlying database DDL schema is not violated. WIP: I will update this

Materialized Views

A **Materialized View** in PostgreSQL is a database object that stores the result of a query physically on disk, unlike a regular view which only stores the query and fetches data dynamically each time it is accessed. Materialized views are useful for improving performance, especially for complex queries that take a long time to compute, by storing precomputed results that can be queried quickly.

Key Characteristics of Materialized Views:

- **Stored results**: The result of the query is saved on disk, so accessing a materialized view is faster compared to a regular view.
- **Requires manual refresh**: Since materialized views store a snapshot of the data, they do not automatically reflect changes in the underlying tables. You need to refresh the view manually

to update the data.

- **Can be indexed**: Materialized views can be indexed, further improving query performance.
- **Improves query performance**: They are particularly useful when the underlying query is complex or involves large datasets.

Creating a Materialized View

To create a materialized view, use the `CREATE MATERIALIZED VIEW` statement.

Example:

Let's say you have a table `employees` and you want to create a materialized view to store employees with high salaries.

Base Table:

```
CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    name TEXT,
    salary NUMERIC,
    department_id INT
);
```

######3 Step 1: Create Materialized View You create a materialized view to store high-salary employees:

```
CREATE MATERIALIZED VIEW high_salary_employees_mat AS
SELECT id, name, salary, department_id
FROM employees
WHERE salary > 50000;
```

This creates a materialized view named `high_salary_employees_mat`, and the query result is stored physically on disk.

Step 2: Query the Materialized View

You can now query this materialized view just like a regular table:

```
SELECT * FROM high_salary_employees_mat;
```

This query will return the precomputed results of employees with a salary greater than 50,000.

Refreshing a Materialized View

Since materialized views store a snapshot of data, they do not automatically reflect updates made to the underlying tables. You need to refresh the materialized view to update its contents.

Example of Refreshing:

If the underlying `employees` table is updated (e.g., a new high-salary employee is added), the materialized view will not show this new data until it is refreshed.

```
REFRESH MATERIALIZED VIEW high_salary_employees_mat;
```

This statement updates the materialized view with the latest data from the `employees` table.

Creating Materialized Views with Data Refresh Options

PostgreSQL allows you to create materialized views with options to control when and how the data is refreshed.

Example: `WITH NO DATA` Option

You can create a materialized view without populating it immediately using the `WITH NO DATA` option. This is useful if you want to create the view but delay the actual data loading.

```
CREATE MATERIALIZED VIEW high_salary_employees_mat AS
SELECT id, name, salary, department_id
FROM employees
WHERE salary > 50000
WITH NO DATA;
```

Later, when you are ready to populate the materialized view, you can use the `REFRESH MATERIALIZED VIEW` command to load the data:

```
REFRESH MATERIALIZED VIEW high_salary_employees_mat;
```

Indexing a Materialized View

To improve query performance on a materialized view, you can create indexes, just like you would on a regular table.

Example:

Let's add an index on the `salary` column of the materialized view to optimize salary-based queries.

```
CREATE INDEX idx_high_salary ON high_salary_employees_mat (salary);
```

This index will improve the performance of queries that filter or sort by the `salary` column in the materialized view.

Dropping a Materialized View

If you no longer need a materialized view, you can drop it using the `DROP MATERIALIZED VIEW` command.

Example:

```
DROP MATERIALIZED VIEW high_salary_employees_mat;
```

This will delete the materialized view and free up the storage space used to store its results.

Differences Between Regular Views and Materialized Views:

Feature	Regular View	Materialized View
Data Storage	Does not store data, only the query	Stores the result of the query
Data Freshness	Always shows up-to-date data	Shows stale data until refreshed
Performance	Slower for complex queries	Faster for complex queries
Refresh	No need for refresh	Needs manual refresh
Indexing	Cannot be indexed	Can be indexed
Use Case	For frequently changing data	For rarely changing or large datasets

When to Use Materialized Views:

- **Complex Queries**: If your query is computationally expensive (e.g., involving multiple joins, aggregations, etc.), materialized views can significantly improve performance.
- **Static or Slowly Changing Data**: Materialized views are useful when the underlying data does not change frequently, or when real-time updates are not critical.
- **Reporting and Analytics**: Materialized views are often used in reporting and analytic systems where precomputed data can be queried quickly.

L3.4 Intermediate SQL/3 (30:46)

Transactions

Transactions in PostgreSQL are crucial for ensuring data integrity and consistency. A transaction is a sequence of one or more SQL operations that are executed as a single unit. The key properties of transactions are encapsulated in the ACID acronym:

- **Atomicity**: All operations in a transaction are completed successfully, or none are applied. If any operation fails, the entire transaction is rolled back.
- **Consistency**: Transactions bring the database from one valid state to another, maintaining integrity constraints.
- **Isolation**: Transactions are executed independently of one another, ensuring that concurrent transactions do not interfere.
- **Durability**: Once a transaction is committed, it will remain so, even in the event of a system failure.

Basic Transaction Commands

In PostgreSQL, you can manage transactions using the following commands:

1. **BEGIN**: Starts a new transaction.
2. **COMMIT**: Saves all changes made during the transaction.
3. **ROLLBACK**: Undoes all changes made during the transaction.

Example Scenario

Let's say we have a simple database with two tables: `accounts` and `transactions`.

```
CREATE TABLE accounts (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    balance NUMERIC
);

CREATE TABLE transactions (
    id SERIAL PRIMARY KEY,
    account_id INT REFERENCES accounts(id),
    amount NUMERIC,
    transaction_date TIMESTAMP DEFAULT NOW()
);
```

Initial Setup

First, we add some sample data to the `accounts` table:

```
INSERT INTO accounts (name, balance) VALUES
('Alice', 1000),
('Bob', 500);
```

Example of a Transaction

Suppose we want to transfer $200 from Alice to Bob. This involves two operations:

1. Deducting $200 from Alice's account.
2. Adding $200 to Bob's account.

Here's how you can perform this in a transaction:

```
BEGIN;

-- Step 1: Deduct from Alice's balance
UPDATE accounts
SET balance = balance - 200
WHERE name = 'Alice';

-- Step 2: Add to Bob's balance
UPDATE accounts
SET balance = balance + 200
WHERE name = 'Bob';

COMMIT;
```

Handling Errors with ROLLBACK

If any operation fails, we want to ensure that neither account is updated. For example, if Alice's balance is insufficient (less than $200), we would not want to deduct from her account. Here's how you can implement this:

```
BEGIN;

-- Check if Alice has enough balance
DO $$
DECLARE
    alice_balance NUMERIC;
BEGIN
    SELECT balance INTO alice_balance FROM accounts WHERE name = 'Alice';

    IF alice_balance < 200 THEN
        RAISE EXCEPTION 'Insufficient balance';
    END IF;

    -- Deduct from Alice's balance
    UPDATE accounts
```

```
    SET balance = balance - 200
    WHERE name = 'Alice';

    -- Add to Bob's balance
    UPDATE accounts
    SET balance = balance + 200
    WHERE name = 'Bob';

EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        RAISE NOTICE 'Transaction failed: %', SQLERRM;
        RETURN;
END $$;

COMMIT;
```

Nested Transactions

PostgreSQL does not support true nested transactions, but you can use savepoints to achieve a similar effect. A savepoint allows you to set a point within a transaction that you can roll back to if needed.

```
BEGIN;

SAVEPOINT before_transfer;

UPDATE accounts
SET balance = balance - 200
WHERE name = 'Alice';

-- Simulating an error condition
IF (SELECT balance FROM accounts WHERE name = 'Alice') < 0 THEN
    ROLLBACK TO SAVEPOINT before_transfer;
    RAISE NOTICE 'Rollback to before transfer due to insufficient balance';
END IF;

UPDATE accounts
SET balance = balance + 200
WHERE name = 'Bob';

COMMIT;
```

Integrity Constraints

Integrity constraints in PostgreSQL ensure the accuracy and consistency of data within a database. They are rules that the database management system enforces on the data in tables, preventing invalid data entry and maintaining relationships between tables.

1. NOT NULL Constraint

This constraint ensures that a column cannot have a NULL value. It's used to enforce that a certain field must always contain a value.

Example:

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL
);
```

In this example, both `username` and `email` cannot be NULL.

2. UNIQUE Constraint

The UNIQUE constraint ensures that all values in a column (or a group of columns) are different from one another. It prevents duplicate entries.

Example:

```
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    sku VARCHAR(50) UNIQUE
);
```

In this case, the `sku` column must contain unique values across all records in the `products` table.

3. PRIMARY KEY Constraint

A PRIMARY KEY is a combination of a NOT NULL and UNIQUE constraint. It uniquely identifies each record in a table.

Example:

```
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    user_id INT NOT NULL,
    order_date TIMESTAMP DEFAULT NOW()
);
```

Here, `order_id` is the primary key, ensuring that each order is uniquely identifiable and cannot be NULL.

4. FOREIGN KEY Constraint

A FOREIGN KEY constraint links two tables together. It enforces a relationship between the data in the two tables, ensuring that a value in one table must exist in another table.

NOTE: Foreign Keys has to may to a primary key!

Example:

```
CREATE TABLE order_items (
    item_id SERIAL PRIMARY KEY,
    order_id INT REFERENCES orders(order_id),
    product_id INT NOT NULL,
    quantity INT NOT NULL
);
```

In this example, the `order_id` column in the `order_items` table is a foreign key that references the `order_id` in the `orders` table. This ensures that each order item must be associated with a valid order.

The following is not possible.

```
CREATE TABLE customers (
    customer_id INT,
    name VARCHAR(100),
    email VARCHAR(100)
);
CREATE TABLE
CREATE TABLE orders (
    order_id INT,
    order_date DATE,
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

/*ERROR:  there is no unique constraint matching given keys for referenced table "customers"
```

5. CHECK Constraint

The CHECK constraint allows you to specify a condition that must be met for a value to be accepted in a column.

Example:

```
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
```

```
    age INT CHECK (age >= 18)
);
```

In this example, the `age` column has a CHECK constraint that ensures only values of 18 or older are allowed.

6. EXCLUSION Constraint

The EXCLUSION constraint is a unique constraint that ensures that if any two rows are compared on specified columns or expressions, at least one of these comparisons will return false or NULL.

Example:

```
CREATE TABLE reservations (
    id SERIAL PRIMARY KEY,
    room_number INT NOT NULL,
    start_time TIMESTAMP NOT NULL,
    end_time TIMESTAMP NOT NULL,
    EXCLUDE USING GIST (room_number WITH =,
                        tsrange(start_time, end_time) WITH &&)
);
```

In this case, the `EXCLUDE` constraint prevents overlapping time ranges for the same room number.

Cascading Integrity Constraints

Cascading integrity constraints in PostgreSQL refer to the behavior that occurs when a record is deleted or updated in a parent table that is referenced by foreign keys in a child table. When such changes occur, cascading actions can automatically propagate to related records in the child table, ensuring that referential integrity is maintained.

Types of Cascading Actions

When defining foreign keys in PostgreSQL, you can specify cascading actions for updates and deletions. The options available are:

1. **CASCADE**: Automatically perform the same action (delete or update) on the child records when the corresponding action is performed on the parent record.
2. **SET NULL**: Set the foreign key column in the child records to NULL when the corresponding parent record is deleted or updated.
3. **SET DEFAULT**: Set the foreign key column in the child records to a default value when the corresponding parent record is deleted or updated.
4. **RESTRICT**: Prevent the deletion or update of a parent record if any child records exist. This is the default behavior.

5. **NO ACTION**: Similar to RESTRICT, but it checks only at the end of the transaction to enforce the constraint.

Example Scenario

Let's look at a practical example with two tables: `authors` and `books`. Each book references an author, and we want to ensure that when an author is deleted, all their associated books are also deleted.

Step 1: Create the Tables

```
CREATE TABLE authors (
    author_id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL
);

CREATE TABLE books (
    book_id SERIAL PRIMARY KEY,
    title VARCHAR(100) NOT NULL,
    author_id INT REFERENCES authors(author_id) ON DELETE CASCADE
);
```

In this example, the `books` table has a foreign key reference to the `authors` table with the `ON DELETE CASCADE` action specified.

Step 2: Insert Sample Data

```
INSERT INTO authors (name) VALUES ('Jane Austen'), ('Mark Twain');

INSERT INTO books (title, author_id) VALUES
('Pride and Prejudice', 1),
('Emma', 1),
('Adventures of Huckleberry Finn', 2);
```

Step 3: Verify the Data

You can check the inserted data using:

```
SELECT * FROM authors;
SELECT * FROM books;
```

Step 4: Delete an Author

Now, let's delete an author and see what happens to their books:

```
DELETE FROM authors WHERE author_id = 1;
```

Since `author_id = 1` corresponds to Jane Austen, this command will automatically delete all books associated with her, thanks to the cascading action we defined.

Step 5: Verify the Deletion

You can check the books table again:

```
SELECT * FROM books;
```

You should see that all books by Jane Austen have been removed.

One more example..

```
CREATE TABLE multiple(
    first INTEGER,
    second INTEGER,
    PRIMARY KEY (first),
    FOREIGN KEY (second)
        REFERENCES multiple ON DELETE CASCADE);

work=# insert into multiple values (1,1);
INSERT 0 1
Time: 6.942 ms
work=# insert into multiple values (2,1);
INSERT 0 1
Time: 7.030 ms
work=# insert into multiple values (4,2);
INSERT 0 1
Time: 5.747 ms
work=# insert into multiple values (3,4);
INSERT 0 1
Time: 5.978 ms

work=# select * from multiple;
 first | second
-------+--------
     1 |      1
     2 |      1
     4 |      2
     3 |      4
(4 rows)

work=# DELETE FROM multiple WHERE first=1;
DELETE 1
Time: 18.484 ms
```

```
work=# select * from multiple;
 first | second
-------+--------
(0 rows)

Time: 0.648 ms

/* Notice how CASCADING deletes other relations ! */
```

Cascading Updates

You can also define cascading actions for updates. For example, if you wanted to change the `author_id` of all books when an author's ID changes, you could define it like this:

```
CREATE TABLE books (
    book_id SERIAL PRIMARY KEY,
    title VARCHAR(100) NOT NULL,
    author_id INT REFERENCES authors(author_id) ON UPDATE CASCADE
);
```

With this setup, if an author's ID is updated, all related books will also reflect this change automatically.

Indices

Indexes in PostgreSQL are special database objects that improve the speed of data retrieval operations on a table. They work like a book index, allowing the database to find rows faster without having to scan the entire table. Indexes can significantly enhance performance, especially for large tables and complex queries.

Types of Indexes in PostgreSQL

1. **B-tree Indexes**: The default type, optimized for equality and range queries.
2. **Hash Indexes**: Useful for equality comparisons but not for range queries.
3. **GIN (Generalized Inverted Index)**: Ideal for indexing array values, full-text search, and JSONB data types.
4. **GiST (Generalized Search Tree)**: Used for geometric data types and full-text search.
5. **BRIN (Block Range INdexes)**: Suitable for large tables with naturally ordered data, useful for large datasets where values are clustered.

Creating an Index

You can create an index using the `CREATE INDEX` statement. Let's illustrate this with examples.

Example Scenario

Consider a simple table `employees`:

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

Step 1: Inserting Sample Data

Let's insert some sample data into the `employees` table:

```
INSERT INTO employees (name, department, salary) VALUES
('Alice', 'HR', 60000),
('Bob', 'IT', 70000),
('Charlie', 'IT', 80000),
('Diana', 'Finance', 65000),
('Edward', 'HR', 72000);
```

Step 2: Creating a B-tree Index

If we frequently query the `department` column, we can create a B-tree index on it:

```
CREATE INDEX idx_department ON employees(department);
```

This index allows PostgreSQL to quickly locate employees based on their department.

Step 3: Querying with the Index

Now, when we run a query that filters by department, PostgreSQL can use the index:

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

With the index, this query will be faster than without it, especially if the table contains many rows.

Step 4: Checking Index Usage

You can use the `EXPLAIN` command to see if the index is being used:

```
EXPLAIN SELECT * FROM employees WHERE department = 'IT';
```

The output will show if the query planner used the `idx_department` index, typically indicated by a line containing `Index Scan`.

Other Index Types

Hash Index Example

If we primarily use equality comparisons (like searching for specific names), we can create a hash index:

```
CREATE INDEX idx_name_hash ON employees USING HASH (name);
```

However, note that hash indexes are not as commonly used because B-tree indexes also handle equality efficiently.

GIN Index Example (for JSONB)

For indexing JSONB data, you can use a GIN index. First, let's create a table with a JSONB column:

```
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    attributes JSONB
);
```

Now, we can create a GIN index:

```
CREATE INDEX idx_attributes_gin ON products USING GIN (attributes);
```

This index helps speed up searches for specific JSONB keys and values.

Dropping an Index

If you determine that an index is no longer needed, you can drop it:

```
DROP INDEX idx_department;
```

Data Types in PostgreSQL

Data Type Category	Data Type	Description
Numeric Types	`INTEGER`	A 4-byte integer, range: -2,147,483,648 to 2,147,483,647
	`BIGINT`	An 8-byte integer, range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
	`SMALLINT`	A 2-byte integer, range: -32,768 to 32,767

Data Type Category	Data Type	Description
	`DECIMAL(p, s)`	Exact numeric of selectable precision, where `p` is total digits and `s` is decimal places
	`NUMERIC(p, s)`	Synonymous with `DECIMAL`, used for exact numeric types
	`FLOAT`	A floating-point number (8-byte)
	`REAL`	A 4-byte floating-point number
Character Types	`CHAR(n)`	Fixed-length character string, padded with spaces to `n` length
	`VARCHAR(n)`	Variable-length character string with a limit of `n` characters
	`TEXT`	Variable-length character string with no specific limit
Date/Time Types	`DATE`	Date (year, month, day)
	`TIME`	Time (hours, minutes, seconds) without time zone
	`TIMESTAMP`	Date and time without time zone
	`TIMESTAMPTZ`	Date and time with time zone
	`INTERVAL`	Time interval (e.g., days, hours, minutes)
Boolean Type	`BOOLEAN`	Stores `TRUE`, `FALSE`, or `NULL`
Geometric Types	`POINT`	A point in 2D space (x, y)
	`LINE`	Infinite line in 2D space
	`LINESEG`	Line segment in 2D space (defined by two points)
	`BOX`	A rectangular box in 2D space (defined by two points)
	`PATH`	A geometric path in 2D space (open or closed)
	`POLYGON`	A polygon in 2D space
	`CIRCLE`	A circle in 2D space (defined by a center point and a radius)
Network Address Types	`CIDR`	IPv4 or IPv6 network address (Classless Inter-Domain Routing)
	`INET`	IPv4 or IPv6 host address

Data Type Category	Data Type	Description
	`MACADDR`	MAC address (6-byte identifier for network interface)
JSON Types	`JSON`	Text representation of JSON data
	`JSONB`	Binary representation of JSON data, supports indexing
XML Type	`XML`	Stores XML data
Composite Types	`ROW`	A row of fields with different data types
Array Types	`ARRAY`	Stores arrays of any base type (e.g., `INTEGER[]`, `TEXT[]`)
UUID Type	`UUID`	Universally Unique Identifier (128-bit number)
Money Type	`MONEY`	Currency amounts, can vary depending on locale

User Defined Data Types

In PostgreSQL, user-defined data types (UDTs) allow you to create custom data types that suit your specific application needs. This feature enhances flexibility and enables you to encapsulate complex data structures within a single type. UDTs can be defined in several ways, including composite types, enumerated types, range types, and more.

Types of User-Defined Data Types

Here are the main types of user-defined data types in PostgreSQL:

1. **Composite Types**
2. **Enumerated Types (ENUM)**
3. **Range Types**
4. **Domain Types**
5. **Array Types**
6. **User-Defined Functions**

1. Composite Types

Composite types allow you to define a structure composed of multiple fields, which can have different data types.

Example:

```
CREATE TYPE address AS (
    street VARCHAR(100),
    city VARCHAR(50),
```

```
    zip_code VARCHAR(10)
);
```

You can then use this type in a table:

```
CREATE TABLE customers (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    address_info address
);
```

2. Enumerated Types (ENUM)

Enumerated types allow you to define a data type that can take on a limited set of values.

Example:

```
CREATE TYPE mood AS (
    mood_name VARCHAR(20)
);

CREATE TABLE user_moods (
    user_id SERIAL PRIMARY KEY,
    mood mood NOT NULL
);
```

You can insert values like:

```
INSERT INTO user_moods (user_id, mood) VALUES (1, ROW('happy'));
```

3. Range Types

Range types represent a range of values of a particular data type. PostgreSQL has built-in range types for numeric, date, and timestamp types, but you can also create custom range types.

Example:

```
CREATE TYPE int4range AS RANGE (
    subtype = INTEGER
);
```

You can then use this type in a table:

```
CREATE TABLE time_slots (
    id SERIAL PRIMARY KEY,
```

```
    slot int4range
);
```

4. Domain Types

Domains allow you to create a new data type based on an existing data type with optional constraints.

Example:

```
CREATE DOMAIN us_phone AS VARCHAR(10)
    CHECK (VALUE ~ '^[0-9]{10}$');
```

You can use this domain in a table:

```
CREATE TABLE contacts (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    phone us_phone
);
```

5. Array Types

While PostgreSQL supports native array types, you can also create user-defined array types.

Example:

```
CREATE TYPE integer_array AS (
    ARRAY INTEGER[]
);

CREATE TABLE test_array (
    id SERIAL PRIMARY KEY,
    numbers integer_array
);
```

6. User-Defined Functions

User-defined functions can be created to operate on user-defined types. These functions can return UDTs or accept them as parameters.

Example:

```
CREATE FUNCTION get_full_address(address_info address) RETURNS TEXT AS $$
BEGIN
    RETURN address_info.street || ', ' || address_info.city || ' ' || address_info.zip_code;
```

```
END;
```

You can call this function with a composite type:

```
SELECT get_full_address(ROW('123 Main St', 'Springfield', '12345')::address);
```

Domains (User Defined Data Types)

In PostgreSQL, a **domain** is a user-defined data type that allows you to create a customized version of a base type with specific constraints. This is particularly useful when you want to enforce certain rules on data entries across different tables without having to redefine those rules each time.

Key Features of Domains:

1. **Base Type**: A domain is built on top of an existing data type (like `integer`, `text`, etc.), inheriting its properties.
2. **Constraints**: You can add constraints to a domain, such as `NOT NULL`, `CHECK`, or `UNIQUE`. These constraints apply whenever the domain is used, ensuring data integrity.
3. **Reusability**: Once a domain is defined, it can be used in multiple tables or columns. This promotes consistency in how data is defined and validated across your database.
4. **Simplicity**: Domains can simplify your schema by reducing redundancy. Instead of repeating the same constraints for multiple columns, you can define them once in the domain.

Creating a Domain

To create a domain, you use the `CREATE DOMAIN` command. Here's a basic example:

```
CREATE DOMAIN positive_integer AS integer
    CHECK (VALUE > 0);
```

In this example, `positive_integer` is a domain based on the `integer` type that only accepts positive values.

Using a Domain

Once you've created a domain, you can use it in table definitions like this:

```
CREATE TABLE orders (
    id serial PRIMARY KEY,
    quantity positive_integer NOT NULL,
```

```
    price numeric CHECK (price > 0)
);
```

Here, the `quantity` column is using the `positive_integer` domain, which means it must be a positive integer due to the constraint defined in the domain.

Modifying and Dropping Domains

You can modify a domain using the `ALTER DOMAIN` statement or drop it using `DROP DOMAIN` if it's no longer needed. Keep in mind that you must ensure that no existing tables or columns are using the domain before you can drop it.

Large Data Types in PostgreSQL

In PostgreSQL, large data types are designed to handle large volumes of data that exceed the limits of standard types like `VARCHAR` or `INTEGER`. While PostgreSQL doesn't explicitly use the terms BLOB (Binary Large Object) and CLOB (Character Large Object) as found in some other database systems, it offers similar functionality through types such as `BYTEA`, `TEXT`, and large objects (LOBs).

1. **TEXT**
    - **Description**: An unlimited-length string data type that can store large amounts of text.
    - **Use Case**: Suitable for storing large texts like articles, comments, or JSON data.
    - **Example**:

```
CREATE TABLE articles (
    id SERIAL PRIMARY KEY,
    title TEXT NOT NULL,
    content TEXT NOT NULL
);

INSERT INTO articles (title, content)
VALUES ('Large Article', 'This is a large piece of text that can go on for a long t:
```

2. **BYTEA**
    - **Description**: This type is used to store binary data. It allows for the storage of any kind of binary information, such as images, audio files, or other non-text data.
    - **Use Case**: Ideal for applications that need to store binary files directly in the database.
    - **Example**:

```
CREATE TABLE images (
    id SERIAL PRIMARY KEY,
    image_data BYTEA NOT NULL
);
```

```
INSERT INTO images (image_data)
VALUES (decode('your_base64_encoded_image_data', 'base64'));
```

3. **Large Objects (LOBs)**
    - **Description**: PostgreSQL provides a specific mechanism to handle large objects, which are stored in a separate system catalog. This approach is more suitable for very large files, such as videos or large datasets.
    - **Use Case**: Suitable for storing files that can be larger than 1 GB.
    - **Example**:

```
-- Create a large object
SELECT lo_create(0);  -- Creates a new large object and returns its OID

-- Open the large object for writing
SELECT lo_open(oid, 132);  -- 132 is the mode for writing

-- Write data to the large object
SELECT lo_write(oid, 'Your binary data goes here');

-- Close the large object
SELECT lo_close(oid);
```

While PostgreSQL does not use the BLOB and CLOB nomenclature directly, it provides equivalent functionality through `BYTEA`, `TEXT`, and large objects.

- **`TEXT`** serves the purpose of a CLOB for storing large text data.
- **`BYTEA`** acts similarly to a BLOB, allowing for the storage of binary data.
- **Large objects** offer a method to handle exceptionally large files efficiently.

Data Control Language (DCL)

Data Control Language (DCL) in PostgreSQL is a subset of SQL that focuses on controlling access to data within the database. DCL is used to manage permissions and security, ensuring that users have appropriate rights to perform actions on database objects. The primary DCL commands in PostgreSQL are `GRANT` and `REVOKE`.

Key Components of DCL

1. **GRANT**: This command is used to give specific privileges to users or roles. It allows administrators to define who can perform certain operations on database objects, such as tables, schemas, or functions.

2. **REVOKE**: This command is used to remove privileges from users or roles. It is important for managing and restricting access when necessary.

Detailed Explanation of DCL Commands

1. GRANT

The `GRANT` command is used to assign privileges on various database objects. The syntax for `GRANT` is:

```
GRANT privilege_type ON object_type object_name TO role_name;
```

- **privilege_type**: This specifies the type of privilege being granted (e.g., `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `USAGE`, etc.).
- **object_type**: This can be various database objects like tables, schemas, sequences, or functions.
- **object_name**: The name of the object to which the privilege applies.
- **role_name**: The user or role to whom the privilege is being granted.

Example:

```
-- Grant SELECT privilege on the 'employees' table to the 'hr' role
GRANT SELECT ON TABLE employees TO hr;

-- Grant INSERT privilege on the 'products' table to a specific user
GRANT INSERT ON TABLE products TO john_doe;

-- Grant EXECUTE privilege on a function to a role
GRANT EXECUTE ON FUNCTION calculate_bonus() TO finance;
```

2. REVOKE

The `REVOKE` command is used to withdraw privileges from users or roles. The syntax for `REVOKE` is:

```
REVOKE privilege_type ON object_type object_name FROM role_name;
```

Example:

```
-- Revoke SELECT privilege on the 'employees' table from the 'hr' role
REVOKE SELECT ON TABLE employees FROM hr;

-- Revoke INSERT privilege from a specific user on the 'products' table
REVOKE INSERT ON TABLE products FROM john_doe;

-- Revoke EXECUTE privilege on a function from a role
REVOKE EXECUTE ON FUNCTION calculate_bonus() FROM finance;
```

Special Considerations

- **Public Role**: In PostgreSQL, the `PUBLIC` role includes all users. You can grant or revoke privileges to/from `PUBLIC` to affect all users.

  Example:

```
-- Grant SELECT privilege on all tables to all users
GRANT SELECT ON ALL TABLES IN SCHEMA public TO PUBLIC;
```

- **Role Hierarchy**: Roles can be granted to other roles. This means that if a role has specific privileges, all roles that inherit from it also gain those privileges.
- **Default Privileges**: You can set default privileges for newly created objects using the `ALTER DEFAULT PRIVILEGES` command.

Data or Instance Authorization in PostgreSQL

Authorization in PostgreSQL refers to the mechanisms and policies that determine what users can and cannot do within the database. This is crucial for maintaining data security and integrity, ensuring that only authorized users can perform specific actions on database objects such as tables, views, and functions.

Key Concepts of Authorization

1. **Roles**: In PostgreSQL, a role can represent either a user or a group of users. Roles can own database objects and have specific privileges assigned to them.
2. **Privileges**: These are permissions granted to roles that determine what actions they can perform. Common privileges include:
   - `SELECT` : Read data from a table or view.
   - `INSERT` : Add new data to a table.
   - `UPDATE` : Modify existing data in a table.
   - `DELETE` : Remove data from a table.
   - `EXECUTE` : Run a stored procedure or function.
   - `USAGE` : Use a sequence or domain.
3. **Granting and Revoking Privileges**: Privileges can be assigned to roles using the `GRANT` command and removed using the `REVOKE` command.
4. **Schema and Object Ownership**: Each database object (like tables, views, and sequences) has an owner, typically the role that created it. The owner has full control over the object.

Example of Authorization in PostgreSQL

1. **Creating Roles**:

```
-- Create two roles
CREATE ROLE admin;
CREATE ROLE user;
```

2. **Granting Privileges**:
    - You can grant specific privileges to roles on tables or other database objects.

```
-- Create a new table
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    price NUMERIC NOT NULL
);

-- Grant SELECT and INSERT privileges to the user role
GRANT SELECT, INSERT ON products TO user;

-- Grant all privileges to the admin role
GRANT ALL PRIVILEGES ON products TO admin;
```

3. **Revoking Privileges**:
    - You can revoke privileges if necessary.

```
-- Revoke INSERT privilege from the user role
REVOKE INSERT ON products FROM user;
```

4. **Using Roles for Authentication**:
    - When a user connects to the database, they can do so as a specific role. For example:

```
-- Connect as the user role
SET ROLE user;

-- The user can now select data from the products table
SELECT * FROM products;  -- Allowed

-- The user cannot insert data into the products table
INSERT INTO products (name, price) VALUES ('Item A', 10.99);  -- Not allowed
```

5. **Creating a Schema**:
    - You can create schemas and control access to them.

```
-- Create a schema
CREATE SCHEMA sales AUTHORIZATION admin;
```

```
-- Grant usage on the schema to the user role
GRANT USAGE ON SCHEMA sales TO user;
```

Authorization in PostgreSQL is a vital part of database security, allowing for fine-grained control over who can access and manipulate data. By using roles and privileges, you can define clear policies that help protect sensitive information and ensure that users can only perform actions appropriate to their role within the organization. Through commands like `GRANT` and `REVOKE`, you can dynamically manage access as the needs of your application evolve.

Schema or DDL Authorization in PostgreSQL

Authorization in the context of database schemas in PostgreSQL involves managing access permissions related to the schema itself and the objects contained within it, such as tables, views, and functions. This ensures that only authorized users or roles can create, modify, or access schema-level objects.

Key Concepts of Schema Authorization

1. **Schema Ownership**: Each schema in PostgreSQL is owned by a specific role (typically the role that created it). The owner has full privileges over the schema and its objects.
2. **Privileges on Schemas**: Privileges can be granted or revoked on schemas, allowing control over various actions, such as:
    - `USAGE`: Allows the role to access objects within the schema.
    - `CREATE`: Allows the role to create new objects (like tables or views) within the schema.
    - `ALTER`: Allows the role to change the schema itself.
    - `DROP`: Allows the role to delete the schema and its contents.
3. **Inheritance of Privileges**: By default, privileges granted on a schema apply to all objects created within that schema by the roles that have the necessary privileges.

Examples of Schema Authorization

1. **Creating a Schema**:

```
-- Create a new schema
CREATE SCHEMA sales AUTHORIZATION admin;
```

2. **Granting Schema Privileges**:
    - You can grant privileges to roles on a schema, allowing them to perform specific actions.

```
-- Grant USAGE and CREATE privileges to the user role on the sales schema
GRANT USAGE, CREATE ON SCHEMA sales TO user;
```

3. **Revoking Schema Privileges**:
    - You can revoke privileges when needed.

```
-- Revoke CREATE privilege from the user role on the sales schema
REVOKE CREATE ON SCHEMA sales FROM user;
```

4. **Accessing Objects in a Schema**:
    - After granting the necessary privileges, users can access and manipulate objects within the schema.

```
-- Assuming the user role has been granted USAGE and CREATE privileges
SET ROLE user;

-- The user can create a table in the sales schema
CREATE TABLE sales.orders (
    id SERIAL PRIMARY KEY,
    product_name TEXT NOT NULL,
    quantity INTEGER NOT NULL
);  -- Allowed since the user has CREATE privilege
```

5. **Using Qualified Names**:
    - When referencing objects in a schema, use the qualified name (schema_name.object_name).

```
-- A user with USAGE privilege can query the table
SELECT * FROM sales.orders;  -- Allowed

-- If the user does not have USAGE privilege, this would fail
```

6. **Schema Privileges and Inheritance**:
    - If a role has privileges on a schema, those privileges will apply to all new objects created within that schema unless explicitly overridden.

```
-- Granting privileges on the schema also grants access to existing objects
GRANT SELECT ON ALL TABLES IN SCHEMA sales TO user;

-- Future tables created in the sales schema will also allow SELECT for the user role
```

Notes on Indices

While indexes in PostgreSQL can significantly improve query performance by speeding up data retrieval, they can also introduce performance issues in certain scenarios. Here are some of the ways indexes can negatively impact performance:

1. Increased Write Overhead

- **Impact**: Every time a row is inserted, updated, or deleted, PostgreSQL must also update any relevant indexes. This can slow down write operations because:
  - The database has to modify multiple structures (the table and its indexes).
  - For large tables with many indexes, the overhead can accumulate quickly.
- **Example**: If you have a table with multiple indexes and frequently perform bulk inserts or updates, the time taken for these operations can increase significantly.

2. Storage Costs

- **Impact**: Indexes consume additional disk space. As the size of the data grows, so does the size of the indexes, which can lead to:
  - Increased storage costs.
  - Potential for the database to run out of space.
- **Example**: Having several large indexes on a table can double or triple the amount of storage required for that table.

3. Cache Pollution

- **Impact**: When the database's shared buffers are filled with index data, it can push out other important data from memory, leading to increased disk I/O when that data needs to be accessed.
- **Example**: If a system has limited memory and many large indexes, they may displace frequently accessed table data, leading to more cache misses and slower performance.

4. Maintenance Overhead

- **Impact**: Indexes require maintenance. As data changes, indexes need to be re-evaluated, which can lead to performance degradation over time.
  - Tasks such as `VACUUM` and `ANALYZE` become more time-consuming with many indexes.
- **Example**: Regular maintenance jobs may take longer to complete, impacting overall database performance.

5. Query Planner Overhead

- **Impact**: The query planner has to consider all available indexes when generating an execution plan. A large number of indexes can lead to longer planning times.
- **Example**: Complex queries on tables with many indexes may experience increased planning time, leading to slower query response times.

6. Redundant or Unused Indexes

- **Impact**: Creating unnecessary or redundant indexes can waste resources. An index that is not used in queries can still incur the overhead of being maintained.
- **Example**: If you create multiple indexes on similar columns without any performance benefit, you increase both write overhead and storage costs without gaining any retrieval advantages.

7. Inefficient Index Usage

- **Impact**: If the query planner selects an inefficient index (e.g., one that doesn't filter results effectively), it can lead to suboptimal performance.
- **Example**: Using an index that provides minimal selectivity might still involve scanning a large portion of the table, negating the benefits of having an index.

While indexes are a powerful tool for optimizing read operations in PostgreSQL, they can also lead to performance issues, especially in write-heavy workloads or when poorly managed. It's essential to:

- Monitor the performance impact of indexes.
- Regularly review and remove unnecessary indexes.
- Consider the balance between read and write performance when designing your indexing strategy.

L3.5 Advanced SQL (33:17)

SQL Functions

SQL functions in PostgreSQL are powerful tools that allow you to encapsulate reusable code for performing calculations, data manipulation, or complex queries. Functions can take input parameters, perform operations, and return results

Types of Functions in PostgreSQL

1. **Built-in Functions**: These are predefined functions provided by PostgreSQL, such as mathematical, string, date/time, and aggregate functions.
2. **User-defined Functions (UDFs)**: These are functions created by users to encapsulate specific business logic or operations.

Creating User-Defined Functions

User-defined functions can be created using the `CREATE FUNCTION` statement. Here's the basic syntax:

```
CREATE FUNCTION function_name (parameters)
RETURNS return_type AS $$
BEGIN
    -- Function logic goes here
    RETURN value;
END;
$$ LANGUAGE plpgsql;
```

Examples of SQL Functions

1. Built-in Functions

String Functions:

```
SELECT UPPER('hello world');  -- Returns 'HELLO WORLD'
```

Mathematical Functions:

```
SELECT ROUND(123.4567, 2);  -- Returns 123.46
```

Date/Time Functions:

```
SELECT NOW();  -- Returns the current timestamp
```

Aggregate Functions:

```
SELECT AVG(salary) FROM employees;  -- Returns the average salary from the employees table
```

2. Creating a User-Defined Function

Example: Simple Function to Calculate the Area of a Circle

```
CREATE FUNCTION calculate_area(radius NUMERIC)
RETURNS NUMERIC AS $$
BEGIN
    RETURN PI() * radius * radius;
END;
$$ LANGUAGE plpgsql;
```

Usage:

```
SELECT calculate_area(5);  -- Returns 78.53981633974483
```

Example: Function with Conditional Logic

```
CREATE FUNCTION classify_employee(salary NUMERIC)
RETURNS TEXT AS $$
BEGIN
    IF salary < 30000 THEN
        RETURN 'Low';
    ELSIF salary < 60000 THEN
        RETURN 'Medium';
    ELSE
        RETURN 'High';
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Usage:

```
SELECT classify_employee(45000);  -- Returns 'Medium'
```

Functions with Multiple Parameters

Example: Concatenate First and Last Names

```
CREATE FUNCTION full_name(first_name TEXT, last_name TEXT)
RETURNS TEXT AS $$
BEGIN
    RETURN first_name || ' ' || last_name;
END;
$$ LANGUAGE plpgsql;
```

Usage:

```
SELECT full_name('John', 'Doe');  -- Returns 'John Doe'
```

Returning Table Rows

You can also create functions that return a set of rows, which can be useful for complex queries.

Example: Function Returning Employees in a Specific Department

```
CREATE FUNCTION get_employees_by_department(dept_id INT)
RETURNS TABLE(emp_id INT, emp_name TEXT) AS $$
```

```
BEGIN
    RETURN QUERY SELECT id, name FROM employees WHERE department_id = dept_id;
END;
$$ LANGUAGE plpgsql;
```

Usage:

```
SELECT * FROM get_employees_by_department(2);  -- Returns all employees in department 2
```

Table Functions

Table functions are a type of user-defined function that returns a set of rows (a table) instead of a single value. This makes them particularly useful for encapsulating complex queries or operations that need to produce multiple rows of results. Table functions can be called in the `FROM` clause of a query, just like a regular table.

Creating Table Functions

To create a table function, you define the function using the `CREATE FUNCTION` statement, specifying the return type as a table. Here's the basic syntax:

```
CREATE FUNCTION function_name(parameters)
RETURNS TABLE(column1 data_type, column2 data_type, ...)
AS $$
BEGIN
    -- Function logic goes here
    RETURN QUERY SELECT ...;
END;
$$ LANGUAGE plpgsql;
```

Examples of Table Functions

1. Simple Table Function

Example: Return All Employees with a Specific Salary Range

This function returns all employees whose salaries fall within a specified range.

```
CREATE FUNCTION get_employees_by_salary(min_salary NUMERIC, max_salary NUMERIC)
RETURNS TABLE(emp_id INT, emp_name TEXT, emp_salary NUMERIC) AS $$
BEGIN
    RETURN QUERY
    SELECT id, name, salary
    FROM employees
    WHERE salary BETWEEN min_salary AND max_salary;
```

```
END;
$$ LANGUAGE plpgsql;
```

Usage:

You can call this function in a query as follows:

```
SELECT * FROM get_employees_by_salary(30000, 60000);
```

This will return all employees with salaries between 30,000 and 60,000.

2. Table Function with Joins

Example: Get Employees with Department Information

This function retrieves employees along with their department names.

```
CREATE FUNCTION get_employees_with_departments()
RETURNS TABLE(emp_id INT, emp_name TEXT, dept_name TEXT) AS $$
BEGIN
    RETURN QUERY
    SELECT e.id, e.name, d.name
    FROM employees e
    JOIN departments d ON e.department_id = d.id;
END;
$$ LANGUAGE plpgsql;
```

Usage:

Call the function to get the list of employees along with their departments:

```
SELECT * FROM get_employees_with_departments();
```

3. Table Function with Conditional Logic

Example: Filter Employees by Job Title

This function returns employees filtered by a specific job title.

```
CREATE FUNCTION get_employees_by_title(job_title TEXT)
RETURNS TABLE(emp_id INT, emp_name TEXT, emp_salary NUMERIC) AS $$
BEGIN
    RETURN QUERY
    SELECT id, name, salary
    FROM employees
    WHERE title = job_title;
```

```
END;
$$ LANGUAGE plpgsql;
```

Usage:

To use this function, simply call it with the desired job title:

```
SELECT * FROM get_employees_by_title('Manager');
```

4. Using Table Functions with Additional Query Logic

You can also include additional logic or filters in your queries when using table functions.

Example: Get Employees with Average Salary Calculation

This function returns employees along with the average salary of their department.

```
CREATE FUNCTION get_employees_with_avg_salary()
RETURNS TABLE(emp_id INT, emp_name TEXT, avg_salary NUMERIC) AS $$
BEGIN
    RETURN QUERY
    SELECT e.id, e.name, AVG(e.salary) OVER (PARTITION BY e.department_id) AS avg_salary
    FROM employees e;
END;
$$ LANGUAGE plpgsql;
```

Usage:

This function can be called to retrieve employee data with average salaries calculated for their departments:

```
SELECT * FROM get_employees_with_avg_salary();
```

SQL Procedures

SQL procedures in PostgreSQL are similar to functions but are designed for executing a sequence of SQL commands and can include control-flow statements such as loops and conditionals. Procedures can be used to perform complex operations, manage transactions, and encapsulate business logic. Unlike functions, procedures do not return a value directly but can still output results through `OUT` parameters or through result sets.

Key Features of SQL Procedures

1. **Transaction Control**: Procedures can manage transactions (commit, rollback) within their execution context.

2. **Control Structures**: They can include various control structures like loops, conditionals, and exception handling.
3. **No Return Value**: Unlike functions, procedures do not return a value directly but can return multiple values through `OUT` parameters.

Creating a SQL Procedure

The syntax for creating a procedure in PostgreSQL is as follows:

```
CREATE PROCEDURE procedure_name (parameters)
LANGUAGE plpgsql AS $$
BEGIN
    -- Procedure logic goes here
END;
$$;
```

Examples of SQL Procedures

1. Simple Procedure to Insert a New Employee

This procedure inserts a new employee into the `employees` table.

```
CREATE PROCEDURE add_employee(
    emp_name TEXT,
    emp_salary NUMERIC,
    emp_department_id INT
)
LANGUAGE plpgsql AS $$
BEGIN
    INSERT INTO employees (name, salary, department_id)
    VALUES (emp_name, emp_salary, emp_department_id);
END;
$$;
```

Usage: To call this procedure, you can use the `CALL` statement:

```
CALL add_e`mployee('John Doe', 50000, 1);
```

2. Procedure with Output Parameters

This procedure retrieves an employee's information based on their ID and returns it via `OUT` parameters.

```
CREATE PROCEDURE get_employee_by_id(
    emp_id INT,
    OUT emp_name TEXT,
```

```
    OUT emp_salary NUMERIC,
    OUT emp_department_id INT
)
LANGUAGE plpgsql AS $$
BEGIN
    SELECT name, salary, department_id
    INTO emp_name, emp_salary, emp_department_id
    FROM employees
    WHERE id = emp_id;
END;
$$;
```

Usage: To call this procedure and retrieve the employee information:

```
CALL get_employee_by_id(1, emp_name, emp_salary, emp_department_id);
```

3. Procedure with Transaction Control

This procedure updates an employee's salary and uses transaction control to ensure data integrity.

```
CREATE PROCEDURE update_employee_salary(
    emp_id INT,
    new_salary NUMERIC
)
LANGUAGE plpgsql AS $$
BEGIN
    -- Start a transaction block
    BEGIN
        UPDATE employees
        SET salary = new_salary
        WHERE id = emp_id;

        -- Optionally include logic to check the update, then commit
        IF NOT FOUND THEN
            RAISE EXCEPTION 'Employee not found';
        END IF;

        -- Commit the transaction
        COMMIT;
    EXCEPTION
        WHEN OTHERS THEN
            -- Rollback in case of error
            ROLLBACK;
            RAISE;
    END;
END;
$$;
```

Usage: To call this procedure:

```
CALL update_employee_salary(1, 55000);
```

4. Procedure with Control Structures

This procedure uses a loop to process all employees and print their names and salaries.

```
CREATE PROCEDURE print_all_employees()
LANGUAGE plpgsql AS $$
DECLARE
    emp_record RECORD;
BEGIN
    FOR emp_record IN SELECT name, salary FROM employees LOOP
        RAISE NOTICE 'Employee Name: %, Salary: %', emp_record.name, emp_record.salary;
    END LOOP;
END;
$$;
```

Usage: To execute this procedure:

```
CALL print_all_employees();
```

Over loading of the function is allowed in SQL:1999

Language Constructs in PostgreSQL

PostgreSQL supports a variety of language constructs that can be used in its procedural language, PL/pgSQL, as well as in standard SQL queries. Here's a comprehensive list of key language constructs in PostgreSQL, along with examples:

1. Variables

You can declare variables to hold temporary data during the execution of a block.

Example:

```
DO $$
DECLARE
    employee_name TEXT;
BEGIN
    employee_name := 'Alice';
    RAISE NOTICE 'Employee Name: %', employee_name;
END;
$$;
```

2. Data Types

PostgreSQL supports various data types, including:

- **Scalar Types**: `INTEGER`, `TEXT`, `NUMERIC`, `BOOLEAN`, `DATE`, etc.
- **Composite Types**: Row types representing a collection of fields.
- **Array Types**: Arrays of any data type.

Example:

```
CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    name TEXT,
    salary NUMERIC,
    hire_date DATE
);
```

3. Control Structures

a. IF Statements

Used for conditional execution.

Example:

```
DO $$
DECLARE
    salary NUMERIC := 40000;
BEGIN
    IF salary < 50000 THEN
        RAISE NOTICE 'Low Salary';
    ELSIF salary < 70000 THEN
        RAISE NOTICE 'Medium Salary';
    ELSE
        RAISE NOTICE 'High Salary';
    END IF;
END;
$$;
```

b. CASE Statements

An alternative to `IF`, used for multi-way branching.

Example:

```
DO $$
DECLARE
    grade CHAR := 'B';
BEGIN
    CASE grade
```

```
        WHEN 'A' THEN RAISE NOTICE 'Excellent';
        WHEN 'B' THEN RAISE NOTICE 'Good';
        ELSE RAISE NOTICE 'Needs Improvement';
    END CASE;
END;
$$;
```

c. LOOP Statements

Used to create simple loops.

Example:

```
DO $$
DECLARE
    counter INT := 1;
BEGIN
    LOOP
        EXIT WHEN counter > 5;
        RAISE NOTICE 'Counter: %', counter;
        counter := counter + 1;
    END LOOP;
END;
$$;
```

d. WHILE Loops

Used to create loops that execute while a condition is true.

Example:

```
DO $$
DECLARE
    counter INT := 1;
BEGIN
    WHILE counter <= 5 LOOP
        RAISE NOTICE 'Counter: %', counter;
        counter := counter + 1;
    END LOOP;
END;
$$;
```

e. FOR Loops

Used for iterating over a range or a set of values.

Example:

```
DO $$
BEGIN
    FOR i IN 1..5 LOOP
        RAISE NOTICE 'Iteration: %', i;
    END LOOP;
END;
$$;
```

4. Error Handling

Using `EXCEPTION` to handle errors.

Example:

```
DO $$
BEGIN
    -- Attempt to divide by zero
    PERFORM 1 / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Caught division by zero error';
END;
$$;
```

5. Functions and Procedures

Creating reusable code blocks.

Example:

```
CREATE FUNCTION add_numbers(a INT, b INT)
RETURNS INT AS $$
BEGIN
    RETURN a + b;
END;
$$ LANGUAGE plpgsql;
```

6. Cursors

For iterating through a result set.

Example:

```
DO $$
DECLARE
    emp_cursor CURSOR FOR SELECT name FROM employees;
    emp_name TEXT;
```

```
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO emp_name;
        EXIT WHEN NOT FOUND;
        RAISE NOTICE 'Employee Name: %', emp_name;
    END LOOP;
    CLOSE emp_cursor;
END;
$$;
```

7. Triggers

Functions that automatically execute in response to certain events on a table.

Example:

```
CREATE FUNCTION log_employee_changes()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO employee_audit (emp_id, change_time)
    VALUES (NEW.id, NOW());
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER employee_update
AFTER UPDATE ON employees
FOR EACH ROW
EXECUTE FUNCTION log_employee_changes();
```

8. Dynamic SQL

Executing SQL commands dynamically.

Example:

```
DO $$
DECLARE
    table_name TEXT := 'employees';
    sql_command TEXT;
BEGIN
    sql_command := 'SELECT COUNT(*) FROM ' || table_name;
    EXECUTE sql_command;
END;
$$;
```

External Language Routines

PostgreSQL supports several external languages out of the box, including:

1. **PL/Python**: Allows you to write functions in Python.
2. **PL/Perl**: Enables the use of Perl for writing functions.
3. **PL/Tcl**: Supports Tcl for function creation.
4. **PL/Java**: Allows Java to be used for writing database routines.
5. **PL/pgSQL**: This is the default procedural language for writing stored procedures and functions.

You can also add additional languages through extensions, such as PL/R for R or PL/v8 for JavaScript.

Creating External Language Routines

To use an external language in PostgreSQL, you first need to ensure that the language is installed and available. You can then create functions in that language using the `CREATE FUNCTION` statement.

Examples of External Language Routines

PL/Python Example

First, you need to enable the PL/Python extension:

```
CREATE EXTENSION plpython3u;  -- Use plpythonu for untrusted mode
```

Example: A Python Function to Calculate Factorial

```
CREATE FUNCTION factorial(n INT)
RETURNS INT AS $$
    if n < 0:
        raise ValueError("Negative input not allowed")
    elif n == 0:
        return 1
    else:
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result
$$ LANGUAGE plpython3u;
```

Usage:

```
SELECT factorial(5);  -- Returns 120
```

PL/Java Example

Enable PL/Java (this requires additional setup beyond standard PostgreSQL installation):

```
CREATE EXTENSION pljava;
```

Example: A Java Function to Reverse a String

```
CREATE OR REPLACE FUNCTION reverse_string(s TEXT)
RETURNS TEXT AS $$
    return new StringBuilder(s).reverse().toString();
$$ LANGUAGE java;
```

Usage:

```
SELECT reverse_string('PostgreSQL');  -- Returns 'LQSERgtoP'
```

Considerations

1. **Performance**: While external language routines can be powerful, they may introduce overhead due to context switching between the database and the external language environment.
2. **Security**: Some external languages (like PL/Python in untrusted mode) can execute arbitrary code, which may pose security risks. Use untrusted languages cautiously.
3. **Dependencies**: Functions written in external languages may depend on external libraries. Ensure that the environment where PostgreSQL is running has all necessary dependencies installed.
4. **Error Handling**: Each external language has its own error handling mechanisms. Be sure to implement appropriate error handling within your routines.
5. **Installation**: Not all external languages are enabled by default. You may need to install additional packages or extensions based on your PostgreSQL setup.

Setting Up C Language Routines

To use C as an external language in PostgreSQL, follow these general steps:

1. **Install PostgreSQL Development Packages**: Ensure that you have the PostgreSQL development packages installed, which include the necessary headers and libraries to build C functions.
2. **Write Your C Function**: Create a C file with your function.
3. **Compile the C Function**: Use the `pg_config` tool to compile your C code into a shared library.
4. **Register the Function in PostgreSQL**: Use SQL commands to register the C function.

Example: A Simple C Function to Add Two Numbers

Step 1: Write the C Function

Create a file named `add.c` with the following content:

```c
#include "postgres.h"
#include "fmgr.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

// Function declaration
PG_FUNCTION_INFO_V1(add_numbers);

// Function definition
Datum add_numbers(PG_FUNCTION_ARGS) {
    int32 arg1 = PG_GETARG_INT32(0); // Get the first argument
    int32 arg2 = PG_GETARG_INT32(1); // Get the second argument
    int32 result = arg1 + arg2;      // Add the numbers
    PG_RETURN_INT32(result);          // Return the result
}
```

Step 2: Compile the C Function

You can compile this C code using `gcc`. Here's how you can do it from the command line:

```
gcc -fPIC -I /usr/include/postgresql/12/server -I /usr/include/postgresql/12/ \
    -shared -o add.so add.c
```

Make sure to replace `/usr/include/postgresql/12/` with the appropriate path for your PostgreSQL installation and version.

Step 3: Load the Shared Library in PostgreSQL

After compiling the shared library, you need to load it into PostgreSQL.

1. First, start your PostgreSQL session.
2. Use the following SQL commands to create the function:

```sql
CREATE FUNCTION add_numbers(INT, INT)
RETURNS INT AS '$libdir/add', 'add_numbers'
LANGUAGE C IMMUTABLE;
```

Step 4: Using the C Function

Now, you can use the `add_numbers` function in your SQL queries:

```
SELECT add_numbers(10, 20);  -- Returns 30
SELECT add_numbers(-5, 5);    -- Returns 0
```

Triggers in PostgreSQL

Triggers in PostgreSQL are special procedures that are automatically executed (or fired) in response to certain events on a particular table or view. They can help maintain data integrity, enforce business rules, and perform automatic actions in the database.

Types of Triggers in PostgreSQL

Triggers can be categorized based on various criteria, such as timing (before or after an event), the type of event (insert, update, delete), and whether they are row-level or statement-level.

1. Row-Level vs. Statement-Level Triggers

- **Row-Level Triggers**: These triggers are executed once for each row affected by the triggering event.
- **Statement-Level Triggers**: These triggers are executed once for the entire SQL statement, regardless of the number of rows affected.

2. Timing of Triggers

- **BEFORE Triggers**: Executed before the triggering event.
- **AFTER Triggers**: Executed after the triggering event.
- **INSTEAD OF Triggers**: Used mainly for views to replace the operation.

Examples of Each Trigger Type

Example 1: BEFORE Row-Level Trigger

This trigger logs information to an audit table before an employee record is inserted.

Step 1: Create an audit table.

```
CREATE TABLE employee_audit (
    emp_id INT,
    action TEXT,
    change_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 2: Create the trigger function.

```
CREATE OR REPLACE FUNCTION log_employee_insert()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO employee_audit (emp_id, action)
    VALUES (NEW.id, 'Inserted');
    RETURN NEW;  -- Return the new record
END;
$$ LANGUAGE plpgsql;
```

Step 3: Create the trigger.

```
CREATE TRIGGER before_employee_insert
BEFORE INSERT ON employees
FOR EACH ROW
EXECUTE FUNCTION log_employee_insert();
```

Usage:

```
INSERT INTO employees (name, salary) VALUES ('Alice', 50000);
```

This will insert a record into the `employee_audit` table each time a new employee is added.

Example 2: AFTER Statement-Level Trigger

This trigger updates a summary table after any update is made to the `employees` table.

Step 1: Create a summary table.

```
CREATE TABLE salary_summary (
    total_salary NUMERIC
);
```

Step 2: Create the trigger function.

```
CREATE OR REPLACE FUNCTION update_salary_summary()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE salary_summary
    SET total_salary = (SELECT SUM(salary) FROM employees);
    RETURN NULL;  -- No need to return anything for statement-level triggers
END;
$$ LANGUAGE plpgsql;
```

Step 3: Create the trigger.

```
CREATE TRIGGER after_employee_update
AFTER UPDATE ON employees
FOR EACH STATEMENT
EXECUTE FUNCTION update_salary_summary();
```

Usage:

```
UPDATE employees SET salary = salary * 1.10;  -- This will update the total salary in the su
```

Example 3: INSTEAD OF Trigger

This trigger can be used on a view to handle insertions instead of directly modifying the underlying table.

Step 1: Create a view.

```
CREATE VIEW employee_view AS
SELECT id, name FROM employees;
```

Step 2: Create the trigger function.

```
CREATE OR REPLACE FUNCTION handle_view_insert()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO employees (id, name, salary) VALUES (NEW.id, NEW.name, 50000);  -- Default s
    RETURN NULL;  -- No need to return anything
END;
$$ LANGUAGE plpgsql;
```

Step 3: Create the trigger.

```
CREATE TRIGGER instead_of_employee_insert
INSTEAD OF INSERT ON employee_view
FOR EACH ROW
EXECUTE FUNCTION handle_view_insert();
```

Usage:

```
INSERT INTO employee_view (id, name) VALUES (1, 'Bob');
```

This insertion will actually add a new employee to the `employees` table instead of the view.

Use cases for Triggers

Triggers in PostgreSQL are powerful tools that can automate tasks, enforce business rules, and maintain data integrity. Here are some common use cases for triggers:

1. Auditing and Logging Changes

Triggers can be used to maintain an audit trail of changes made to tables, capturing details about inserts, updates, or deletes.

Use Case:

- Log every modification to a sensitive table (e.g., employee data) by recording the change details in an audit table.

2. Enforcing Business Rules

You can implement complex business logic directly in the database by using triggers to enforce rules that may not be easily enforceable through constraints alone.

Use Case:

- Ensure that an employee's salary cannot be set below a minimum value or that the total salaries for a department do not exceed a certain budget.

3. Automatically Updating Related Tables

Triggers can synchronize changes across related tables by updating or inserting records in one table based on changes in another.

Use Case:

- When a product's price is updated, automatically adjust any related records in an order history table that track historical prices.

4. Data Validation

Triggers can perform complex validation checks before allowing data to be inserted or updated in a table.

Use Case:

- Ensure that no two employees have the same email address by checking for duplicates in a trigger before inserting a new record.

5. Maintaining Derived Data

Triggers can be used to maintain summary or derived data that needs to be updated whenever the underlying data changes.

Use Case:

- Keep a running total of sales in a summary table whenever a new sale is added or an existing sale is updated.

6. Implementing Cascading Actions

Triggers can help implement cascading actions that need to occur based on certain operations, similar to foreign key constraints with cascading options.

Use Case:

- When a record in a parent table is deleted, automatically delete all related records in child tables.

7. Complex Business Logic Execution

Triggers can encapsulate complex operations that should happen automatically based on certain conditions.

Use Case:

- When an order is created, automatically send a notification email or trigger an inventory update.

8. Data Transformation

You can use triggers to transform data before it is inserted or updated in the database.

Use Case:

- Automatically convert all text input to uppercase or format dates to a specific style before inserting them into the table.

9. Managing Referential Integrity

While foreign key constraints enforce basic referential integrity, triggers can handle more complex relationships and integrity checks.

Use Case:

- Ensure that an entry in one table is only valid if it exists in another table, even if that relationship is not defined through foreign keys.

Transition Tables

In PostgreSQL, transition tables are special constructs used within triggers to hold the old and new state of rows that are affected by the triggering event. They are especially useful in the context of `AFTER` and `BEFORE` triggers on row-level operations (INSERT, UPDATE, DELETE). Transition tables provide a way to reference the values of rows before and after modifications, making them crucial for implementing complex logic in triggers.

Types of Transition Tables

There are two types of transition tables in PostgreSQL:

1. **Transition Tables for `BEFORE` Triggers**: These tables hold the old values of the rows being modified before the change occurs.
2. **Transition Tables for `AFTER` Triggers**: These tables hold the new values of the rows after the change has occurred.

Naming of Transition Tables

- For `BEFORE` triggers, the transition table is often referred to as `OLD`.
- For `AFTER` triggers, the transition table is often referred to as `NEW`.

Example of Using Transition Tables

Let's illustrate the concept of transition tables with a practical example:

Use Case: Auditing Changes

Suppose you have an `employees` table, and you want to log changes whenever an employee's salary is updated. You can create a trigger that uses transition tables to capture the old and new salary values.

Step 1: Create the `employees` Table

```
CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    salary NUMERIC
);
```

Step 2: Create an Audit Table

Create an audit table to log the changes:

```
CREATE TABLE salary_audit (
    emp_id INT,
    old_salary NUMERIC,
    new_salary NUMERIC,
```

```
    change_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 3: Create the Trigger Function

Create a trigger function that uses transition tables:

```
CREATE OR REPLACE FUNCTION log_salary_change()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO salary_audit (emp_id, old_salary, new_salary)
    VALUES (OLD.id, OLD.salary, NEW.salary);
    RETURN NEW;  -- Return the new row
END;
$$ LANGUAGE plpgsql;
```

Step 4: Create the Trigger

Create a trigger that fires `BEFORE UPDATE` on the `employees` table:

```
CREATE TRIGGER before_salary_update
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
EXECUTE FUNCTION log_salary_change();
```

Usage

Now, when you update the salary of an employee, the old and new salaries will be logged in the `salary_audit` table.

Example Update:

```
UPDATE employees SET salary = 60000 WHERE id = 1;
```

After executing the update, the `salary_audit` table will contain a record of the old and new salary values for the employee with `id = 1`.

Best Practices when using Triggers

1. Keep Logic Simple

- **Avoid Complex Logic**: Triggers should perform straightforward tasks. If the logic becomes too complex, consider using a stored procedure instead.
- **Single Responsibility**: Each trigger should have a single purpose. This makes it easier to maintain and understand.

2. Limit the Number of Triggers

- **Minimize Trigger Usage**: Avoid overusing triggers on the same table. Too many triggers can complicate debugging and affect performance.
- **Combine Related Actions**: If multiple triggers can be logically combined, consider merging them to reduce complexity.

3. Optimize Performance

- **Use Efficient Queries**: Ensure that any SQL statements within triggers are optimized. Poorly performing queries can slow down data modifications.
- **Consider Trigger Timing**: Use `BEFORE` triggers when you need to validate or modify data before it is committed, and `AFTER` triggers for actions that don't affect the transaction's outcome.

4. Avoid Side Effects

- **No External Calls**: Avoid triggering external actions (e.g., sending emails, logging to external systems) directly from triggers, as these can introduce latency and complicate error handling.
- **Be Cautious with DML in Triggers**: Avoid performing DML (insert, update, delete) on the same table that the trigger is defined on to prevent recursive triggers unless absolutely necessary.

5. Use Proper Exception Handling

- **Handle Exceptions Gracefully**: Ensure that your triggers include exception handling to manage unexpected errors. Use the `EXCEPTION` block to log errors or take alternative actions.

6. Document Your Triggers

- **Provide Clear Documentation**: Document the purpose and behavior of each trigger, including when it fires and what actions it performs. This is crucial for maintenance and future development.

7. Test Thoroughly

- **Extensive Testing**: Test triggers in various scenarios, especially edge cases, to ensure they behave as expected. Consider performance testing to assess impact on large datasets.
- **Use Transactions for Testing**: When testing triggers, wrap operations in transactions so you can easily roll back and avoid unintended changes.

8. Monitor Performance

- **Analyze Trigger Impact**: Regularly monitor the performance impact of your triggers, especially as your data volume grows. Use tools like `EXPLAIN` to analyze performance.

9. Use Version Control

- **Manage Schema Changes**: Treat triggers like code and manage them using version control. This helps track changes and allows for collaborative development.

10. Limit Trigger Usage for High-Frequency Tables

- **Be Cautious with Frequent Operations**: If a table experiences high insert/update/delete rates, be cautious with triggers, as they can introduce overhead. Consider alternatives like periodic batch jobs for certain tasks.

Row Level Triggers

In PostgreSQL, the `NEW` keyword is used within row-level triggers and certain functions to reference the new state of a row being inserted or updated. Here's how it works in different contexts:

1. Row-Level Triggers

When you create a trigger for `INSERT` or `UPDATE`, you can use `NEW` to access the new values of the row:

- **INSERT Trigger:** `NEW` contains the values of the row that is being inserted.
- **UPDATE Trigger:** `NEW` contains the updated values of the row.

Example

Suppose you have a table `employees` and you want to log changes to a `salary` column:

```
CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    name TEXT,
    salary NUMERIC
);

CREATE TABLE salary_changes (
    employee_id INT,
    old_salary NUMERIC,
    new_salary NUMERIC,
    change_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE OR REPLACE FUNCTION log_salary_change()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'UPDATE' THEN
        INSERT INTO salary_changes (employee_id, old_salary, new_salary)
        VALUES (NEW.id, OLD.salary, NEW.salary);
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER trigger_salary_change
AFTER UPDATE ON employees
FOR EACH ROW EXECUTE FUNCTION log_salary_change();
```

In this example:

- `NEW.id` accesses the ID of the employee after the update.
- `OLD.salary` accesses the salary before the update.

2. Functions with `NEW`

In functions that are invoked by triggers, you can directly use the `NEW` keyword as described above. However, you can't use `NEW` in standalone SQL functions not associated with a trigger.