Week 12.4

SQL Relationships & Joins

In today's lecture, Harkirat revisits the `Fundamentals of SQL`, with a particular focus on the crucial concepts of `relationships and transactions` in SQL databases. The session provided a deeper understanding of how to structure and query relational data using `joins`, and explored the various `types of joins` available in SQL.

Recap

Before diving into advanced topics such as "Relations, Transactions, and Join Queries in SQL Databases," it's crucial to have a solid foundation in the basics of SQL and database management. Here's a detailed recap of the previously covered topics:

Types of Databases

- **Relational (SQL) Databases**: Organize data into tables consisting of rows and columns. Each table represents a relation, and the rows hold individual records. Popular relational database management systems (RDBMS) include MySQL, PostgreSQL, Microsoft SQL Server, and Oracle Database[1][2].
- **Non-relational (NoSQL) Databases**: Use different data models for storing, managing, and accessing data. Common models include document-oriented, key-value, graph, and wide-column stores. NoSQL databases are designed for flexibility, scalability, and high performance with unstructured or semi-structured data[2].

Why Not NoSQL

NoSQL databases offer schema-less data storage, which is beneficial for rapid development and handling large volumes of unstructured data. However, the lack of a strict schema can lead to data inconsistency and challenges in enforcing data integrity as applications grow[1].

Why SQL?

SQL databases are preferred for applications requiring strict data integrity, complex transactions, and relationships between data entities. They support ACID (Atomicity, Consistency, Isolation, Durability)

properties, ensuring reliable transaction processing. SQL databases are ideal for applications like e-commerce platforms and financial systems where data consistency is critical[1][2].

Creating a PostgreSQL Database

- **Using `psql`**: A terminal-based front-end to PostgreSQL that allows for interactive command execution and database management.
- **Using Docker**: Running PostgreSQL in a Docker container for development purposes.

Using `psql`

`psql` is a powerful tool for interacting with PostgreSQL databases, allowing users to execute SQL queries, manage database objects, and view data directly from the terminal[1].

Creating a Table and Defining Its Schema

SQL databases require defining a schema before inserting data. This involves creating tables and specifying columns with data types and constraints. For example:

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

Interacting with the Database (CRUD Operations)

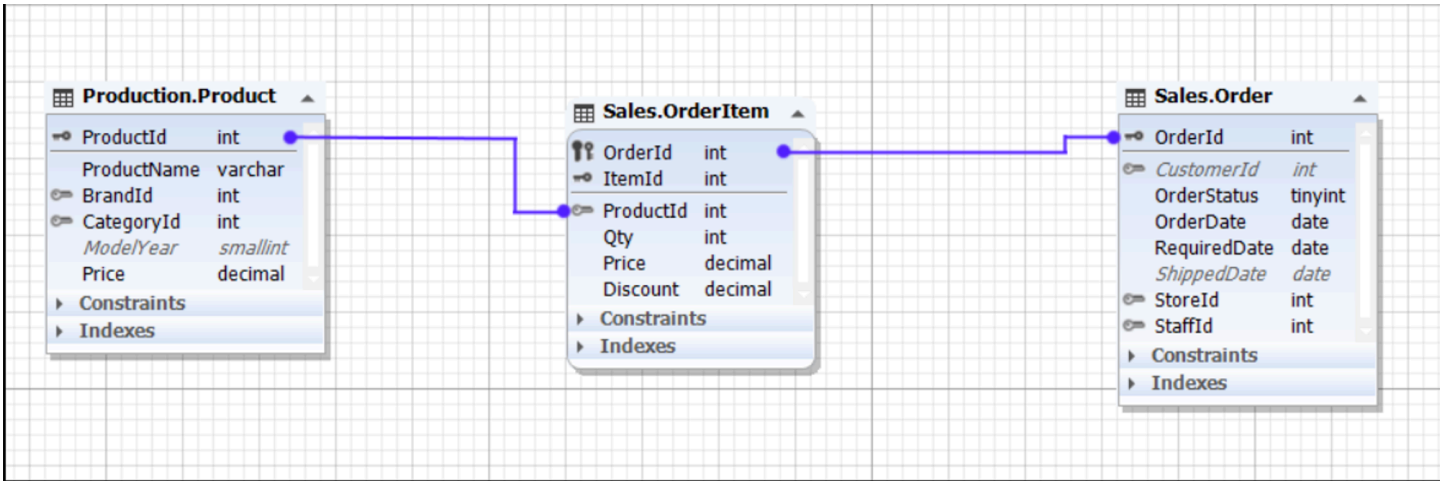
- **INSERT**: Adds new records to a table.
- **UPDATE**: Modifies existing records based on specific criteria.
- **DELETE**: Removes records from a table.
- **SELECT**: Retrieves data from one or more tables.

Using the `pg` Library

The `pg` library is a Node.js package for interfacing with PostgreSQL databases. It allows for executing SQL queries from within a Node.js application, leveraging JavaScript for database operations.

```
const { Client } = require('pg');
const client = new Client({connectionString: "your_connection_string"});
await client.connect();
// Perform database operations
await client.end();
```

> Now after this recap, we are all set to explore more advanced database concepts, including relationships between tables, transactions that ensure data integrity across multiple operations, and join queries that retrieve related data from multiple tables.



Relationships & Transactions

In database systems, relationships are fundamental for linking data stored across different tables. This section will elaborate on how relationships are handled in both NoSQL and SQL databases, and how transactions are used in SQL databases to ensure data integrity.

Relationships in MongoDB (NoSQL)

MongoDB, a NoSQL database, allows for flexible data modeling. You can store related data together in a single document, which can contain nested objects and arrays. This is useful for encapsulating all data about an entity in one place without the need for separate tables.

Example: Storing User Details with Address in MongoDB

```
{
  "username": "john_doe",
  "email": "john@example.com",
  "password": "securepassword",
  "address": {
    "city": "New York",
    "country": "USA",
    "street": "123 Liberty St",
    "pincode": "10005"
  }
}
```

This structure allows you to store a user's details along with their address in a single, nested object.

Relationships in SQL

SQL databases, on the other hand, require a more structured approach. Since SQL databases don't store objects directly, you need to create separate tables for different entities and establish relationships between them using keys.

Example: Defining Relationships in SQL

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE addresses (
    id SERIAL PRIMARY KEY,
    user_id INTEGER NOT NULL,
    city VARCHAR(100) NOT NULL,
    country VARCHAR(100) NOT NULL,
    street VARCHAR(255) NOT NULL,
    pincode VARCHAR(20),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);
```

In this SQL schema, the `addresses` table has a `user_id` column that serves as a foreign key, creating a relationship with the `users` table. The `FOREIGN KEY` constraint enforces the relationship and ensures referential integrity.

SQL Queries for Relationships

To insert an address for a user, you would use an `INSERT` statement:

```
INSERT INTO addresses (user_id, city, country, street, pincode)
VALUES (1, 'New York', 'USA', '123 Broadway St', '10001');
```

To retrieve the address of a user given their ID, you would use a `SELECT` statement:

```
SELECT city, country, street, pincode
FROM addresses
WHERE user_id = 1;
```

Transactions in SQL

Transactions are critical in SQL databases to ensure that a series of operations either all succeed or all fail. This is important when you have multiple related operations that must be treated as a single unit.

Example: Using Transactions in SQL

```
BEGIN; -- Start transaction

INSERT INTO users (username, email, password)
VALUES ('john_doe', 'john_doe1@example.com', 'securepassword123');

INSERT INTO addresses (user_id, city, country, street, pincode)
VALUES (currval('users_id_seq'), 'New York', 'USA', '123 Broadway St', '10001');

COMMIT;
```

In this example, both `INSERT` statements are wrapped in a transaction. If any statement fails, the entire transaction is rolled back, leaving the database in a consistent state.

Node.js Code for Transactions

In a Node.js application, you can use the `pg` library to manage transactions programmatically.

Example: Inserting User and Address with Transaction in Node.js

```
import { Client } from 'pg';

async function insertUserAndAddress(
    username: string,
    email: string,
    password: string,
    city: string,
    country: string,
    street: string,
    pincode: string
) {
    const client = new Client({
        host: 'localhost',
        port: 5432,
        database: 'postgres',
        user: 'postgres',
        password: 'mysecretpassword',
    });

    try {
        await client.connect();

        // Start transaction
        await client.query('BEGIN');

        // Insert user
```

```
        const insertUserText = `
            INSERT INTO users (username, email, password)
            VALUES ($1, $2, $3)
            RETURNING id;
        `;
        const userRes = await client.query(insertUserText, [username, email, password]);
        const userId = userRes.rows[0].id;

        // Insert address using the returned user ID
        const insertAddressText = `
            INSERT INTO addresses (user_id, city, country, street, pincode)
            VALUES ($1, $2, $3, $4, $5);
        `;
        await client.query(insertAddressText, [userId, city, country, street, pincode]);

        // Commit transaction
        await client.query('COMMIT');

        console.log('User and address inserted successfully');
    } catch (err) {
        await client.query('ROLLBACK'); // Roll back the transaction on error
        console.error('Error during transaction, rolled back.', err);
        throw err;
    } finally {
        await client.end(); // Close the client connection
    }
}

// Example usage
insertUserAndAddress(
    'johndoe',
    'john.doe@example.com',
    'securepassword123',
    'New York',
    'USA',
    '123 Broadway St',
    '10001'
);
```

In the provided Node.js function `insertUserAndAddress`, a transaction is started with `BEGIN`, and `COMMIT` is used to apply the changes. If an error occurs, `ROLLBACK` is called to undo the transaction. This ensures that either both the user information and address are inserted into the database, or neither is, maintaining data integrity.

> While NoSQL databases like MongoDB allow for flexible data storage, SQL databases require a structured approach with defined relationships and transactions to ensure data integrity. Using transactions in SQL databases,

> especially when performing multiple related operations, is crucial to prevent partial updates and maintain consistency. The `pg` library in Node.js provides the necessary tools to handle transactions effectively in your applications.

Joins

Joins in SQL are a powerful feature that allows you to combine rows from two or more tables based on a related column between them. This is essential when you want to fetch data that is distributed across multiple tables.

The Problem with Separate Queries

Fetching user details and their address could be done with two separate queries, but this approach has several drawbacks:

Approach 1: Separate Queries (Not Recommended)

```
-- Query 1: Fetch user's details
SELECT id, username, email
FROM users
WHERE id = YOUR_USER_ID;

-- Query 2: Fetch user's address
SELECT city, country, street, pincode
FROM addresses
WHERE user_id = YOUR_USER_ID;
```

This approach results in two round trips to the database, which can increase latency. It also complicates the application logic, as you need to manage two separate result sets and ensure they are related correctly in your application code.

The Power of Joins

Using joins, you can retrieve related data from multiple tables in a single query. This is more efficient and simplifies your application logic.

Approach 2: Using Joins (Recommended)

```
SELECT users.id, users.username, users.email, addresses.city, addresses.country, addresses.str
FROM users
JOIN addresses ON users.id = addresses.user_id
WHERE users.id = YOUR_USER_ID;
```

In this query, the `JOIN` clause is used to combine rows from `users` and `addresses` where the `id` of the user matches the `user_id` in the addresses table.

Implementing Joins in a Node.js Application

When implementing joins in a Node.js application using the `pg` library, you can use the same SQL join syntax within your application code.

Approach 2: Using Joins in Node.js

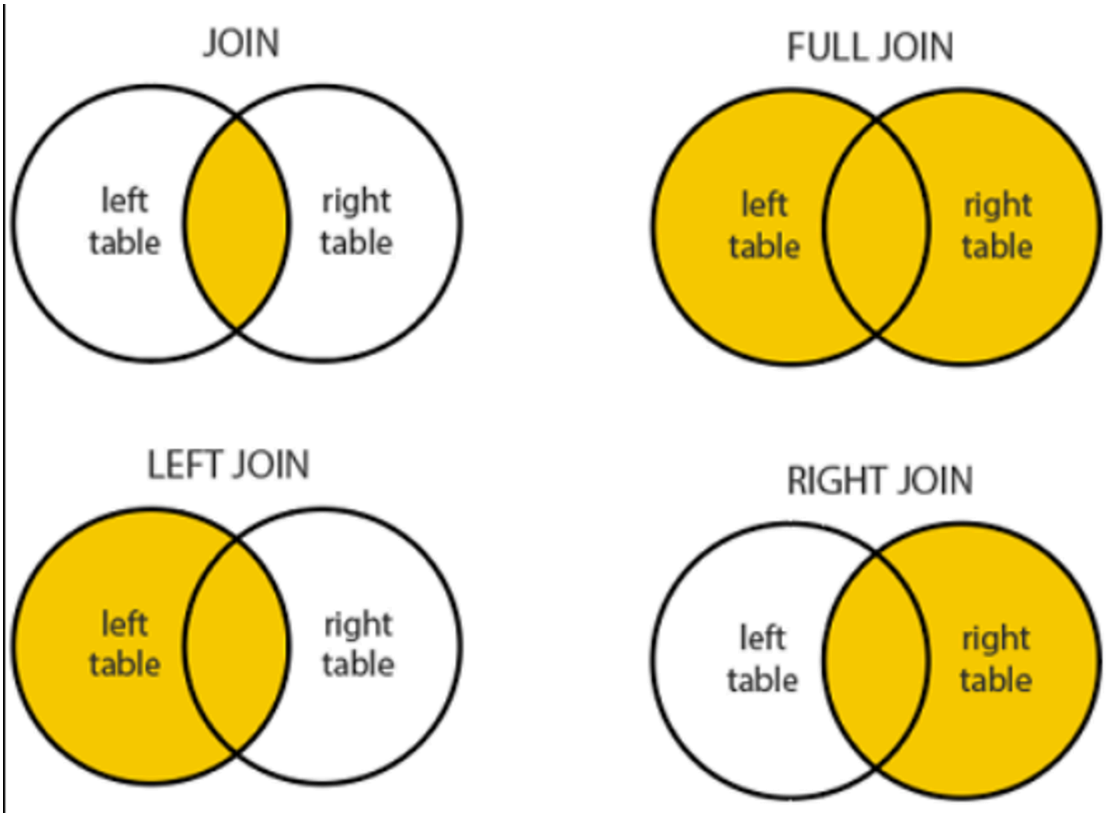
```
import { Client } from 'pg';

// Async function to fetch user data and their address together
async function getUserDetailsWithAddress(userId: string) {
    const client = new Client({
        // ...connection config
    });

    try {
        await client.connect();
        const query = `
            SELECT u.id, u.username, u.email, a.city, a.country, a.street, a.pincode
            FROM users u
            JOIN addresses a ON u.id = a.user_id
            WHERE u.id = $1
        `;
        const result = await client.query(query, [userId]);

        if (result.rows.length > 0) {
            console.log('User and address found:', result.rows[0]);
            return result.rows[0];
        } else {
            console.log('No user or address found with the given ID.');
            return null;
        }
    } catch (err) {
        console.error('Error during fetching user and address:', err);
        throw err;
    } finally {
        await client.end();
    }
}

getUserDetailsWithAddress("1");
```

In this function, `getUserDetailsWithAddress`, a single query with a join is used to fetch both the user details and their address. This reduces latency and simplifies the application logic, as you handle only one result set.

Benefits of Using a Join

- **Reduced Latency**: Fewer database calls mean less communication overhead and faster responses.
- **Simplified Application Logic**: Handling a single result set is easier than coordinating multiple queries and their results.
- **Transactional Integrity**: A join query ensures that the data retrieved is consistent and reflects the state of the database at the time of the query.

> Using joins is a best practice for fetching related data in SQL databases. It leverages the relational nature of SQL databases and provides a performant, reliable, and clean way to retrieve and work with data in your applications.



Types of Joins

SQL joins are used to combine rows from two or more tables based on a related column between them. There are several types of joins, each with its own use case depending on the nature of the data and the desired results. Here's a detailed explanation of the different types of joins:

1. INNER JOIN

Definition: The `INNER JOIN` keyword selects records that have matching values in both tables.

Use Case: You want to retrieve only the rows with matching keys in both tables. For example, if you want to find all users who have provided their address details, you would use an `INNER JOIN`.

SQL Example:

```
SELECT users.username, addresses.city, addresses.country, addresses.street, addresses.pincode
FROM users
INNER JOIN addresses ON users.id = addresses.user_id;
```

In this query, only users with corresponding entries in the `addresses` table will be returned.

2. LEFT JOIN (or LEFT OUTER JOIN)

Definition: The `LEFT JOIN` keyword returns all records from the left table (table1), and the matched records from the right table (table2). The result is `NULL` from the right side if there is no match.

Use Case: To list all users and their address information, regardless of whether they have provided an address. Users without an address will appear with `NULL` values for address fields.

SQL Example:

```
SELECT users.username, addresses.city, addresses.country, addresses.street, addresses.pincode
FROM users
LEFT JOIN addresses ON users.id = addresses.user_id;
```

This query includes all users, with address information where available.

3. RIGHT JOIN (or RIGHT OUTER JOIN)

Definition: The `RIGHT JOIN` keyword returns all records from the right table, and the matched records from the left table. The result is `NULL` from the left side if there is no match.

Use Case: While less common due to typical foreign key constraints, a `RIGHT JOIN` would be used if you want to start with the "addresses" table and optionally include user information.

SQL Example:

```
SELECT users.username, addresses.city, addresses.country, addresses.street, addresses.pincode
FROM users
RIGHT JOIN addresses ON users.id = addresses.user_id;
```

Given the usual foreign key constraints, every address should have a corresponding user, making `RIGHT JOIN` less likely to be used in this context.

4. FULL JOIN (or FULL OUTER JOIN)

Definition: The `FULL JOIN` keyword returns all records when there is a match in either left (table1) or right (table2) table records.

Use Case: A `FULL JOIN` would be used to combine all records from both "users" and "addresses" tables, showing all users and all addresses, with `NULL` values in columns from the table that lacks a matching row.

SQL Example:

```
SELECT users.username, addresses.city, addresses.country, addresses.street, addresses.pincode
FROM users
FULL JOIN addresses ON users.id = addresses.user_id;
```

This query would reveal all users and addresses, including any orphaned records that don't have a match in the other table.

> Each type of join serves a specific purpose and can be chosen based on the data relationships and the information you need to retrieve:

> Use `INNER JOIN` when you need to match rows from both tables. Use `LEFT JOIN` to include all rows from the left table, with matching rows from the right table if available. Use `RIGHT JOIN` to include all rows from the right table, with matching rows from the left table if available. Use `FULL JOIN` to include rows when there is a match in either table.