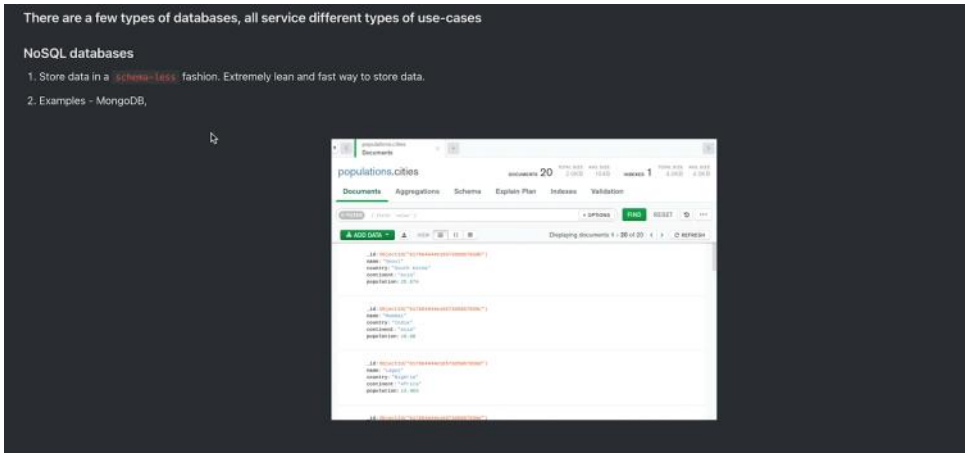
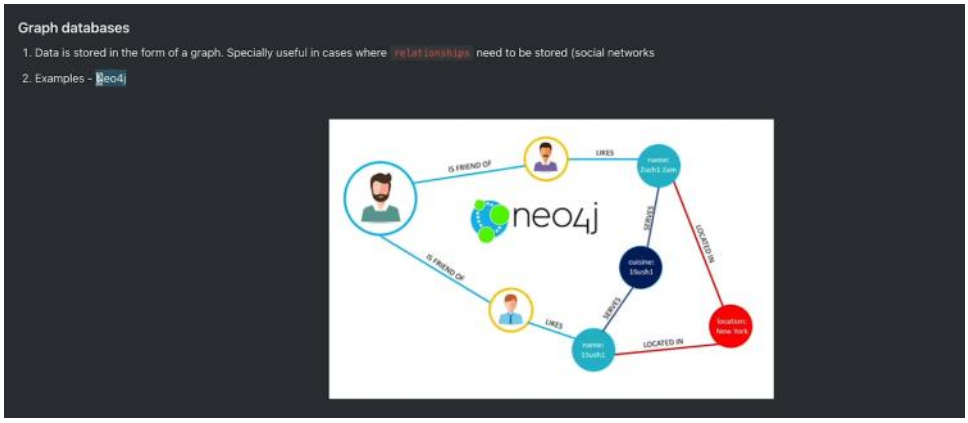
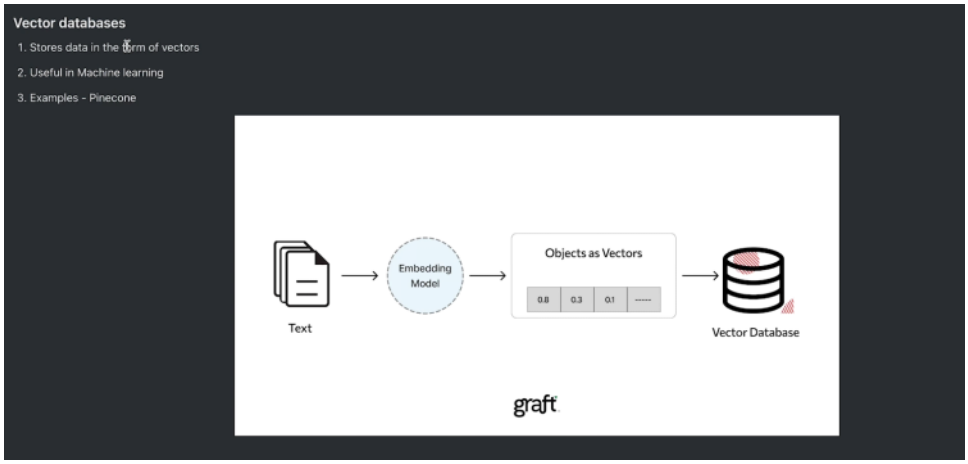
Postgres SQL

08 February 2025 17:00

Types of Databases







SQL databases

1. Stores data in the form of rows
2. Most full stack applications will use this
3. Examples - MySQL, Postgres

Table: customers

customer_id	first_name	last_name	phone	country
1	John	Doe	817-646-8833	USA
2	Robert	Luna	412-862-0502	USA
3	David	Robinson	208-340-7906	UK
4	John	Reinhardt	307-242-6285	UK
5	Betty	Taylor	806-749-2958	UAE

Query's

1. CREATE TABLE users

`CREATE TABLE users` : This command initiates the creation of a new table in the database named `users` .

2. id SERIAL PRIMARY KEY

- `id` : The name of the first column in the `users` table, typically used as a unique identifier for each row (user). Similar to `_id` in mongodb
- `SERIAL` : A PostgreSQL-specific data type for creating an auto-incrementing integer. Every time a new row is inserted, this value automatically increments, ensuring each user has a unique `id` .
- `PRIMARY KEY` : This constraint specifies that the `id` column is the primary key for the table, meaning it uniquely identifies each row. Values in this column must be unique and not null.

3. email VARCHAR(255) UNIQUE NOT NULL,

- `email` : The name of the second column, intended to store the user's username.
- `VARCHAR(50)` : A variable character string data type that can store up to 50 characters. It's used here to limit the length of the username.
- `UNIQUE` : This constraint ensures that all values in the `username` column are unique across the table. No two users can have the same username.
- `NOT NULL` : This constraint prevents null values from being inserted into the `username` column. Every row must have a username value.

4. password VARCHAR(255) NOT NUL

Same as above, can be non uniqye

5. created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP

- `created_at` : The name of the fifth column, intended to store the timestamp when the user was created.
- `TIMESTAMP WITH TIME ZONE` : This data type stores both a timestamp and a time zone, allowing for the precise tracking of when an event occurred, regardless of the user's or server's time zone.
- `DEFAULT CURRENT_TIMESTAMP` : This default value automatically sets the `created_at` column to the date and time at which the row is inserted into the table, using the current timestamp of the database server.

If you have access to a database right now, try running this command to create a simple table in there

```
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
);
```

Then try running
\dt;
to see if the table has been created or not

1. INSERT

```
INSERT INTO users (username, email, password)
VALUES ('username_here', 'user@example.com', 'user_password');
```

💡 Notice how you didn't have to specify the `id` because it auto increments

2. UPDATE

```
UPDATE users
SET password = 'new_password'
WHERE email = 'user@example.com';
```

3. DELETE

```
DELETE FROM users
WHERE id = 1;
```

4. Select

```
SELECT * FROM users
WHERE id = 1;
```

```
const result = await client.query('SELECT * FROM USERS;')
console.log(result)
```

```
// write a function to create a users table in your database.
import { Client } from 'pg'

const client = new Client({
  connectionString: "postgresql://postgres:mysecretpassword@localhost/postgres"
})

async function createUsersTable() {
    await client.connect()
    const result = await client.query(`
        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
        );
    `)
    console.log(result)
}

createUsersTable();
```

Transactions mostly used in paytm where the both query reached at the same time so even if the one of the side goes down both the query fails or goes at the same time

▼ SQL Query

```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;
```

▼ Node.js Code

```javascript
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();
```

```javascript
    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);
```

Join

```
SELECT users.id, users.username, users.email, addresses.city, addresses.country, addresses.
FROM users
JOIN addresses ON users.id = addresses.user_id
WHERE users.id = '1';
```

```
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 = YOUR_USER_ID;
```

```
// Async function to fetch user data and their address together
async function getUserDetailsWithAddress(userId: string) {
    const client = new Client({
        host: 'localhost',
        port: 5432,
        database: 'postgres',
        user: 'postgres',
        password: 'mysecretpassword',
    });

    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");
```

Types of joins

1. INNER JOIN

Returns rows when there is at least one match in both tables. If there is no match, the rows are not returned. It's the most common type of join.

Use Case: Find All Users With Their Addresses. If a user hasn't filled their address, that user shouldn't be returned

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

2. LEFT JOIN

Returns all rows from the left table, and the matched rows from the right table.

Use case - To list all users from your database along with their address information (if they've provided it), you'd use a LEFT JOIN. Users without an address will still appear in your query result, but the address fields will be NULL for them.

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

3. RIGHT JOIN

Returns all rows from the right table, and the matched rows from the left table.

Use case - Given the structure of the database, a RIGHT JOIN would be less common since the `addresses` table is unlikely to have entries not linked to a user due to the foreign key constraint. However, if you had a situation where you start with the `addresses` table and optionally include user information, this would be the theoretical use case.

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

4. FULL JOIN

Returns rows when there is a match in one of the tables. It effectively combines the results of both LEFT JOIN and RIGHT JOIN.

Use case - A FULL JOIN would combine all records from both `users` and `addresses`, showing the relationship where it exists. Given the constraints, this might not be as relevant because every address should be linked to a user, but if there were somehow orphaned records on either side, this query would reveal them.

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