# Step 1 - Types of Databases

💡 What all we'll learn today -

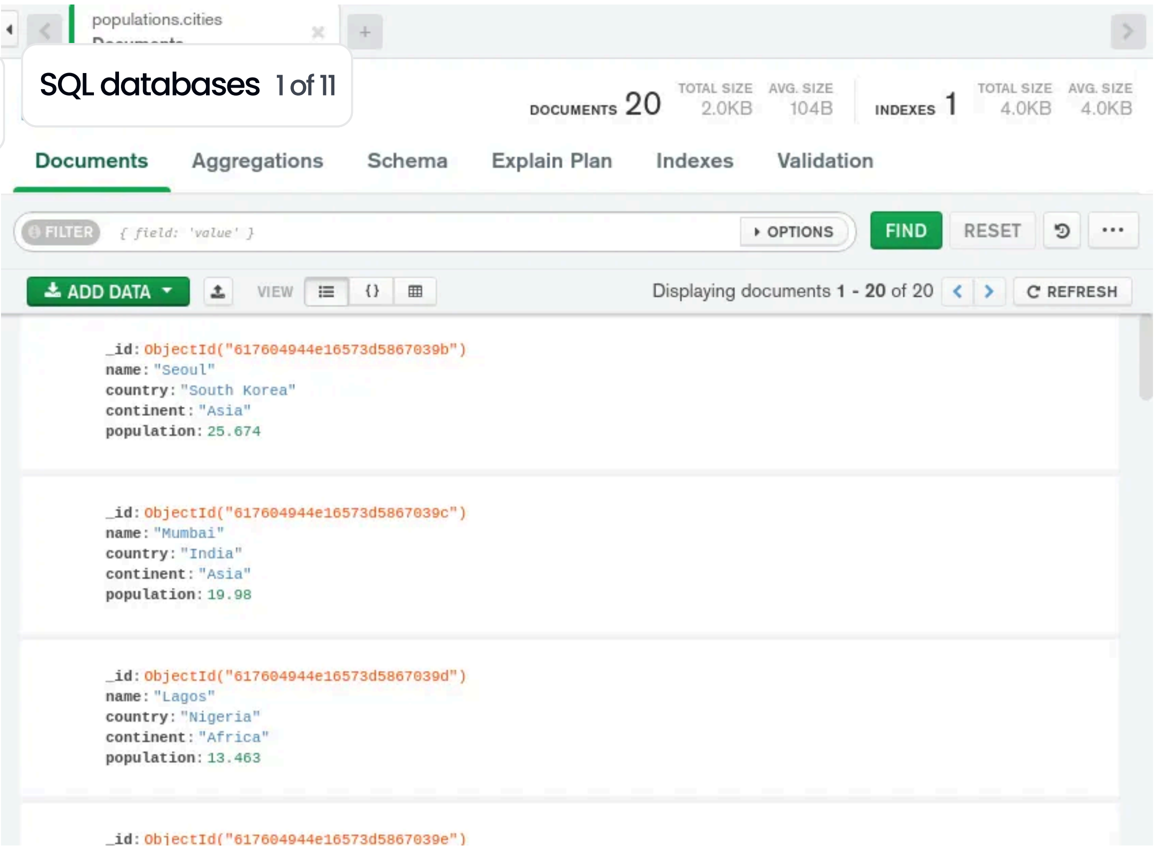
Simple - SQL vs NoSQL, how to create Postgres Databases, How to do CRUD on them

Advance - Relationships, Joins, Transactions

## There are a few types of databases, all service different types of use-cases

### NoSQL databases

1. Store data in a `schema-less` fashion. Extremely lean and fast way to store data.
2. Examples - MongoDB,



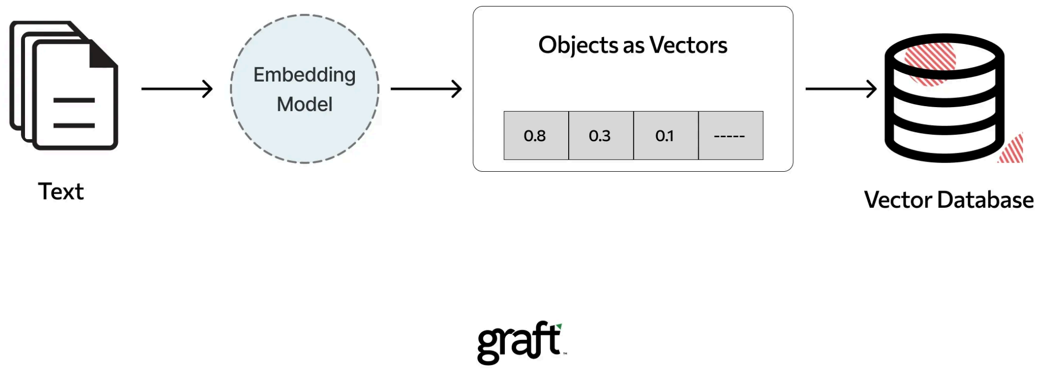

## Graph databases

1. Data is stored in the form of a graph. Specially useful in cases where relationships need to be stored (social networks

2. Examples - Neo4j

# Vector databases

1. Stores data in the form of vectors
2. Useful in Machine learning
3. Examples - Pinecone




# SQL databases

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

# Step 2 - Why not NoSQL

You might've used MongoDB

It's schemaless properties make it ideal to for bootstraping a project fast.

But as your app grows, this property makes it very easy for data to get currupted

## What is schemaless?

Different rows can have different schema (keys/types)

### Problems?

1. Can lead to inconsistent database
2. Can cause runtime errors
3. Is too flexible for an app that needs strictness

### Upsides?

1. Can move very fast
2. Can change schema very easily

k that mongoose does add strictness to the codebase because we used to define a schema there. That strictness is present at the Node.js level, not at the DB level. You can still put in erroneous data in the database that doesn't follow that schema.

# Step 3 - Why SQL?

SQL databases have a strict schema. They require you to

1. Define your schema
2. Put in data that follows that schema
3. Update the schema as your app changes and perform `migrations`

So there are 4 parts when using an SQL database (not connecting it to Node.js, just running it and putting data in it)

1. Running the database.
2. Using a library that let's you connect and put data in it.
3. Creating a table and defining it's `schema`.
4. Run queries on the database to interact with the data (Insert/Update/Delete)

# Step 4 - Creating a database

You can start a Potgres database in a few ways -

▼ Using neondb

https://neon.tech/ is a decent service that let's you create a server.

```
ername:password@ep-broken-frost-69135494.
```

▼ Using docker locally

```
docker run --name my-postgres -e POSTGRES_PASSWORD=m
```

## Connection String

```
postgresql://postgres:mysecretpassword@localhost:5432/pos
```

▼ Using docker on windows

How to run postgrSQL in windows terminal(if you have docker installed).
- fist run docker gui application that help in running commands in terminal.
- After that run it with the docker instance by the help of following command .
-- for the first time if the image is not downloaded .
-- `docker run --name my-postgres1 -e POSTGRES_PASSWORD=mysecretpassword -d -p 5432:5432 postgres` .
-- if the docker image is there, prior to use the it can simply be runned by `docker run <image name>` .
- After that ,
-- use `docker exec -it my-postgres1 psql -U postgres -d postgres` this command in terminal .
-- then enter the password and it will connect to localhost Postgress instance .
-- now you will be inside the postress command line that looks like `postgres-#` .
- U can check it by running `\\dt` , (the command to display all the tables.)

💡 The connection string is similar to the string we had in mongoose

## Connection String



where does harkirat live ⇒ [1, 2, 2, 2, 3001, 100]

Harkirat lives in INdia ⇒ [1, 2, 2, 2, 2,2 ]

Harkirat is from chandigarh ⇒ [1, 2, 2, 2, 3]

Harkirat has been living in india, chandigarh ⇒ [1, 2, 2, 2, 2, 3]

The world is round ⇒ [1, 2, 10001, 1001, 001001]

Pacman is such a good game ⇒ [100, 10001, 20020, 1-001, 100]

# Step 5 - Using a library that let's you connect and put data in it.

psql is a terminal-based front-end to PostgreSQL. It provides an n... d-line interface to the PostgreSQL (or TimescaleDB) database. With psql, you can type in queries interactively, issue them to PostgreSQL, and see the query results.

### How to connect to your database?

psql Comes bundled with postgresql. You don't need it for this tutorial. We will directly be communicating with the database from Node.js

```
psql -h p-broken-frost-69135494.us-east-2.aws.neon.tech -d dat
```

## 2. pg

pg is a Node.js library that you can use in your backend app to store data in the Postgres DB (similar to mongoose ). We will be installing this eventually in our app.

# Step 6 - Creating a table and defining it's schema .

A single database can have multiple tables inside. Think of them as ... ongoDB database.



Until now, we have a database that we can interact with. The next step in case of postgres is to define the `schema` of your tables.

SQL stands for `Structured query language`. It is a language in which you can describe what/how you want to put data in the database.

To create a table, the command to run is -

```
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_TIMES
);
```

There are a few parts of this SQL statement, let's decode them one by one

## 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 ... ach row (user). Similar to `_id` in mongodb



- ...eSQL-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



- **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

SQL databases 1 of 11

# Step 7 - Interacting with the database

There are 4 things you'd like to do with a database

## 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;
```

…our terminal if you have `psql` installed locally.

If not, that's fine we'll eventually be doing these through



# Step 8 - How to do queries from a Node.js app?

In the end, postgres exposes a protocol that someone needs to talk to be able to send these commands (update, delete) to the database.

`psql` is one such library that takes commands from your terminal and sends it over to the database.

To do the same in a Node.js , you can use one of many Postgres clients

## pg library

https://www.npmjs.com/package/pg

Non-blocking PostgreSQL client for Node.js.

Documentation - https://node-postgres.com/

Connecting -

```
const client = new Client({
```

```
  host: 'my.database-server.com',
  database: 'database-name',
  user: 'database-user',
  password: 'secretpassword!!',
})

client.connect()
```

Querying -

```
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@loc
})

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_T
        );
    `)
    console.log(result)
}

createUsersTable();
```

# Step 9 - Creating a simple Node.js app

1. Initialise an empty typescript project

```
npm init -y
npx tsc --init
```

1. Change the `rootDir` and `outDir` in `tsconfig.json`

```
"rootDir": "./src",
"outDir": "./dist",
```

1. Install the `pg` library and it's types (because we're using TS)

```
npm install pg
npm install @types/pg
```

1. Create a simple Node.js app that lets you put data

Create a function that let's you insert data into a table. Make it `async` , make sure `client.connect` resolves before u do the insert

▼ Answer

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

// Async function to insert data into a table
async function insertData() {
    host: 'localhost',
```

```
    port: 5432,
    ...ostgres',
    user: 'postgres',
    password: 'mysecretpassword',
  });

  try {
    await client.connect(); // Ensure client connection is establisl
    const insertQuery = "INSERT INTO users (username, email, pas
    const res = await client.query(insertQuery);
    console.log('Insertion success:', res); // Output insertion resul
  } catch (err) {
    console.error('Error during the insertion:', err);
  } finally {
    await client.end(); // Close the client connection
  }
}

insertData();
```

> 💡 This is an `insecure` way to store data in your tables.
> When you expose this functionality eventually via `HTTP`,
> someone can do an `SQL INJECTION` to get access to your data/delete your data.

1. More secure way to store data.
   Update the code so you don't put `user provided fields` in the `SQL string`

▼ What are user provided strings?

In your final app, this insert statement will be done when a user signs up on your app.
Email, username, password are all user provided strings

▼ What is the `SQL string` ?

```
const insertQuery = "INSERT INTO users (username, email, pass
```

```
ery = 'INSERT INTO example_table(column1, colu
ait client.query(insertQuery, [column1Value, colu
```

▼ Solution

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

// Async function to insert data into a table
async function insertData(username: string, email: string, passw
  const client = new Client({
    host: 'localhost',
    port: 5432,
    database: 'postgres',
    user: 'postgres',
    password: 'mysecretpassword',
  });

  try {
    await client.connect(); // Ensure client connection is establis
    // Use parameterized query to prevent SQL injection
    const insertQuery = "INSERT INTO users (username, email, pas
    const values = [username, email, password];
    const res = await client.query(insertQuery, values);
    console.log('Insertion success:', res); // Output insertion resul
  } catch (err) {
    console.error('Error during the insertion:', err);
  } finally {
    await client.end(); // Close the client connection
  }
}

// Example usage
insertData('username5', 'user5@example.com', 'user_passworc
```

1. Query data

   Write a function `getUser` that lets you fetch data from the database given a `email` as input.

SQL databases 1 of 11

```
om 'pg';

// Async function to fetch user data from the database given an e
async function getUser(email: string) {
  const client = new Client({
    host: 'localhost',
    port: 5432,
    database: 'postgres',
    user: 'postgres',
    password: 'mysecretpassword',
  });


 try {
  await client.connect(); // Ensure client connection is establishec
  const query = 'SELECT * FROM users WHERE email = $1';
  const values = [email];
  const result = await client.query(query, values);

  if (result.rows.length > 0) {
   console.log('User found:', result.rows[0]); // Output user data
   return result.rows[0]; // Return the user data
  } else {
   console.log('No user found with the given email.');
   return null; // Return null if no user was found
  }
 } catch (err) {
  console.error('Error during fetching user:', err);
  throw err; // Rethrow or handle error appropriately
 } finally {
  await client.end(); // Close the client connection
 }
}

// Example usage
getUser('user5@example.com').catch(console.error);
```

# Step 10 - Relationships and Transactions

Relationships let you store data in different tables and `relate` it with each other.

## Relationships in Mongodb

Since `mongodb` is a NoSQL database, you can store any shape of data in it.
If I ask you to store a users details along with their address, you can store it in an object that has the address details.

## Relationships in SQL

Since SQL can not store `objects` as such, we need to define two different tables to store this data in.

This is called a relationship , which means that the Address table is related to the Users table.

When defining the table, you need to define the relationship

```
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_TIMES
);

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_TIMES
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCAD
);
```

## SQL query

To insert the address of a user -

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

Now if you want to get the address of a user given an `id` , you

SQL databases 1 of 11

query -

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

## Extra - Transactions in SQL

> 💡 Good question to have at this point is what queries are run when the user signs up and sends both their information and their address in a single request.
> Do we send two SQL queries into the database? What if one of the queries (address query for example) fails?
> This would require `transactions` in SQL to ensure either both the user information and address goes in, or neither does

▼ SQL Query

```
BEGIN; -- Start transaction

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

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

COMMIT;
```

▼ Node.js Code

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

async function insertUserAndAddress(
    username: string,
    email: string,
    password: string,
    country: string,
```

```
    street: string,
SQL databases 1 of 11 ng
  ) {
    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, [usernar
      const userId = userRes.rows[0].id;

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

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

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

```
}
```



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

# Step 11 - Joins

Defining relationships is easy.

What's hard is `joining` data from two (or more) tables together.

For example, if I ask you to fetch me a users details `and` their address, what SQL would you run?

▼ Approach 1 (Bad)

```
-- 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;
```

▼ Approach 2 (using joins)

```
e, users.email, addresses.city, ac
```

```
JOIN addresses ON users.id = addresses.user_id
= '1';
```



```
SELECT u.id, u.username, u.email, a.city, a.country, a.street, a.pin
FROM users u
JOIN addresses a ON u.id = a.user_id
WHERE u.id = YOUR_USER_ID;
```

Now try converting the same to your node app

▼ Approach 1 (Bad)

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

// Async function to fetch user details and address separately
async function getUserDetailsAndAddressSeparately(userId: st
  const client = new Client({
    host: 'localhost',
    port: 5432,
    database: 'postgres',
    user: 'postgres',
    password: 'mysecretpassword',
  });

  try {
    await client.connect();

    // Fetch user details
    const userDetailsQuery = 'SELECT id, username, email FROM
    const userDetails = await client.query(userDetailsQuery, [u

    // Fetch user address
    const userAddressQuery = 'SELECT city, country, street, pinc
    const userAddress = await client.query(userAddressQuery,

    if (userDetails.rows.length > 0) {
      console.log('User found:', userDetails.rows[0]);
      console.log('Address:', userAddress.rows.length > 0 ? user
      return { user: userDetails.rows[0], address: userAddress.r

      console.log('No user found with the given ID.');
```

```
            return null;
SQL databases 1 of 11
        console.error('Error during fetching user and address:', err);
        throw err;
    } finally {
        await client.end();
    }
}
getUserDetailsAndAddressSeparately("1");
```

Approach 2 (using joins)

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

// 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.stree
            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;
```

```
        console.error('Error during fetching user and address:', err);
    } finally {
        await client.end();
    }
}
getUserDetailsWithAddress("1");
```



No

## Benefits of using a join -

1. **Reduced Latency**
2. Simplified Application Logic
3. **Transactional Integrity**

# 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, addres
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

Use case - To list all users from your database along with their address (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, addres
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, addres
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.

```
name, addresses.city, addresses.country, addres
FULL JOIN addresses ON users.id = addresses.user_id;
```