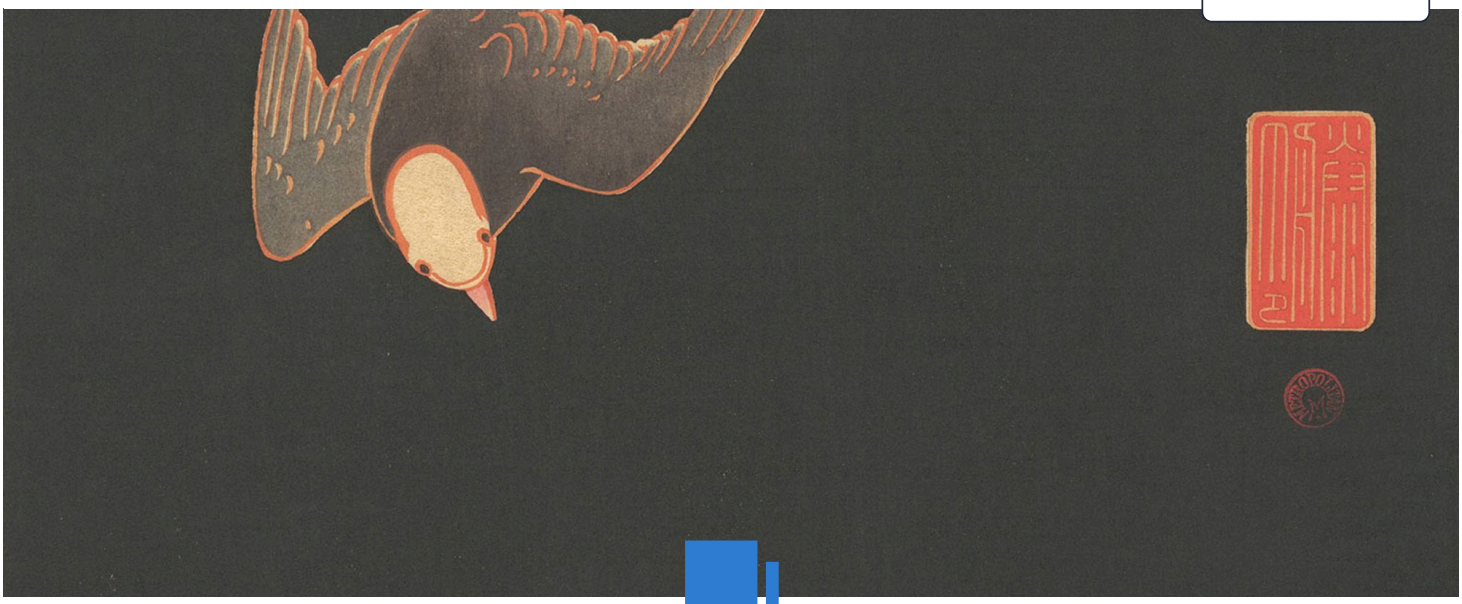


Week 10.1

Understanding Postgres

In today's lecture, Harkirat introduces the PostgreSQL database, beginning with an overview of `different database types` and the limitations of NoSQL databases. The discussion then pivots to the `necessity of SQL databases`, highlighting their advantages in certain scenarios.

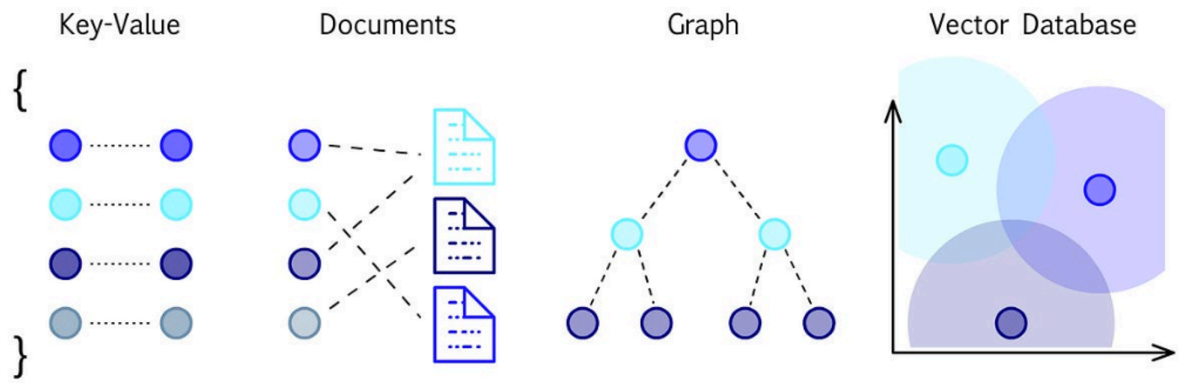
We further look into the process of `creating our first PostgreSQL database`, exploring how to `interact with it and the various operations` that can be executed. This provides a solid foundation for understanding the principles of relational database management systems.

What We'll Learn Today

- **Simple Topics**:
  - **SQL vs NoSQL**: Understanding the differences between structured SQL databases and flexible NoSQL databases.
  - **Creating Postgres Databases**: Learning how to set up and configure PostgreSQL databases.
  - **CRUD Operations**: Performing Create, Read, Update, and Delete operations on database records.

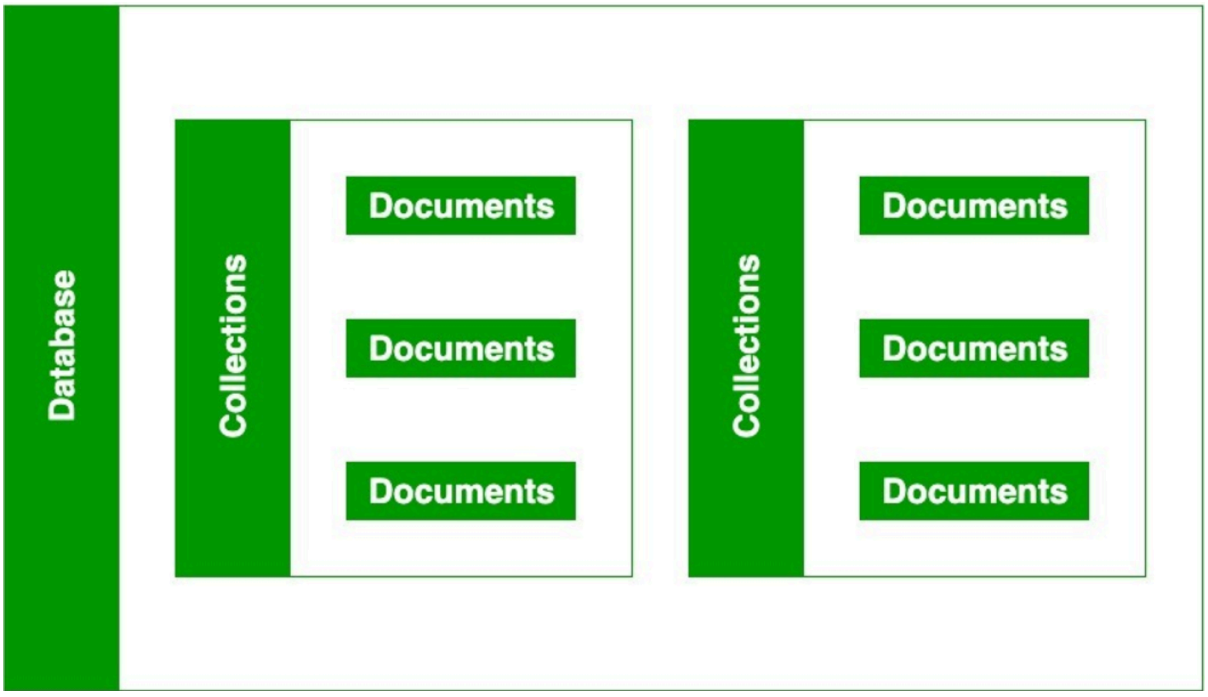
Types of Databases

Databases are an essential component of many applications, serving as the backbone for data storage and retrieval. There are several types of databases, each designed to serve specific use cases and data management needs. Below is an elaboration on the types of databases:



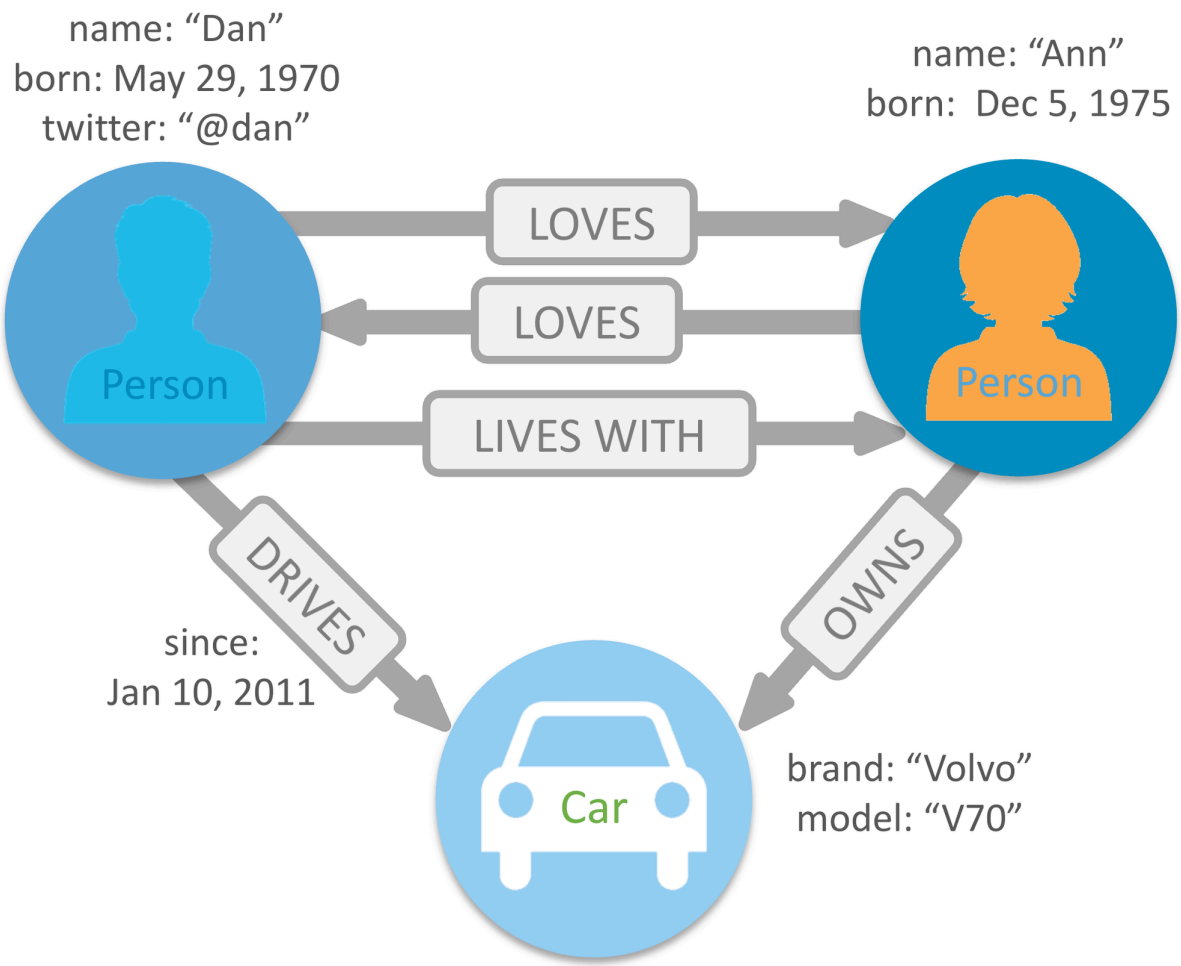
NoSQL Databases

- **Definition**: NoSQL databases are designed to store, retrieve, and manage large volumes of unstructured or semi-structured data. They are known for their flexibility, scalability, and high performance.
- **Schema-less**: Unlike SQL databases, NoSQL databases do not require a predefined schema, allowing for the storage of data in various formats.
- **Use Cases**: Ideal for big data applications, real-time web apps, and for handling large volumes of data that may not fit neatly into a relational model.
- **Examples**: MongoDB, Cassandra, Redis, Couchbase.



Graph Databases

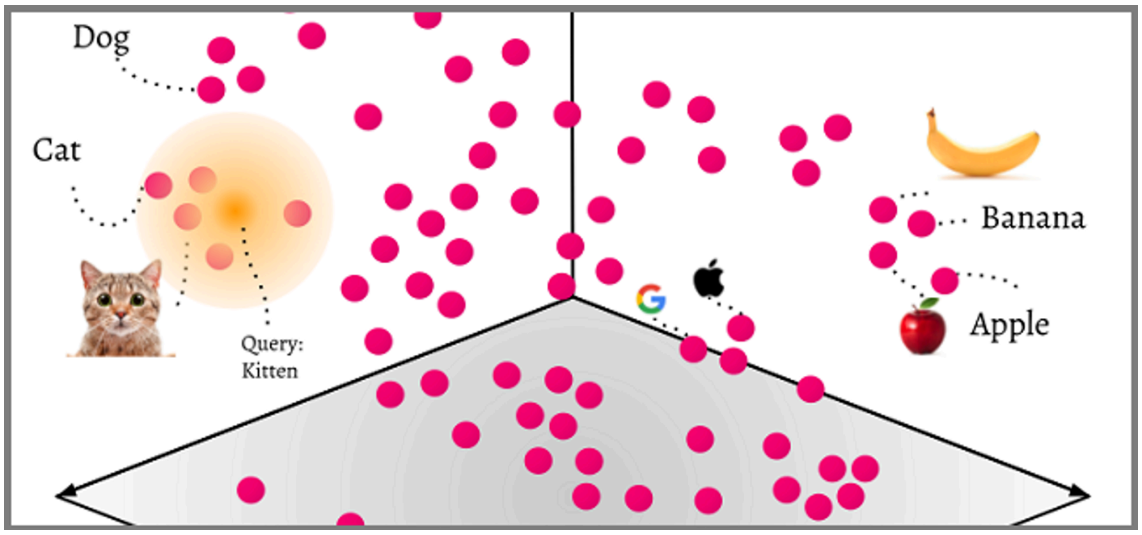
- **Definition**: Graph databases are designed to store and navigate relationships. They treat the relationships between data as equally important as the data itself.
- **Data Storage**: Data is stored in nodes (entities) and edges (relationships), which makes them highly efficient for traversing and querying complex relationships.
- **Use Cases**: Particularly useful for social networks, recommendation engines, fraud detection, and any domain where relationships are key.
- **Examples**: Neo4j, Amazon Neptune, OrientDB.



Vector Databases

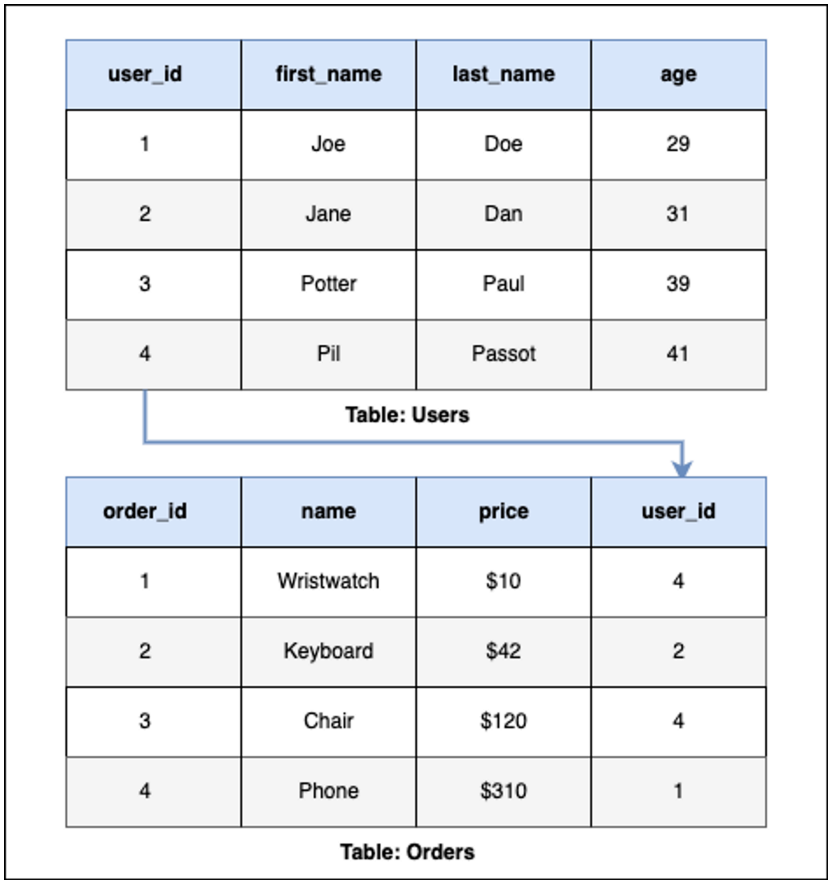
- **Definition**: Vector databases are specialized databases optimized for vector similarity searching. They are used to store and process vector embeddings typically generated by machine learning models.
- **Data Storage**: Data is stored in the form of vectors, which are arrays of numbers that represent data in a high-dimensional space.

- **Use Cases**: Useful in machine learning applications, such as image recognition, natural language processing, and recommendation systems.
- **Examples**: Pinecone, Milvus, Faiss.



SQL Databases

- **Definition**: SQL databases, also known as relational databases, store data in predefined schemas and tables with rows and columns.
- **Data Storage**: Data is organized into tables, and each row in a table represents a record with a unique identifier called the primary key.
- **Use Cases**: Most full-stack applications use SQL databases for their ability to maintain ACID (Atomicity, Consistency, Isolation, Durability) properties, ensuring reliable transaction processing.
- **Examples**: MySQL, PostgreSQL, Oracle, SQL Server.



user_id	first_name	last_name	age
1	Joe	Doe	29
2	Jane	Dan	31
3	Potter	Paul	39
4	Pil	Passot	41

Table: Users

order_id	name	price	user_id
1	Wristwatch	$10	4
2	Keyboard	$42	2
3	Chair	$120	4
4	Phone	$310	1

Table: Orders

Why Not NoSQL

While NoSQL databases like MongoDB offer significant advantages, particularly in terms of flexibility and speed of development, they also come with potential drawbacks that can become more pronounced as an application scales. Here's an elaboration on the points mentioned:

What is Schemaless?

- **Definition**: In a schemaless database, the structure of the data is not defined beforehand. This means that each 'row' or document can have a different set of fields (keys) and data types (values).
- **Flexibility**: This allows for the storage of heterogeneous data and can accommodate changes in the data model without requiring migrations or alterations to the existing data structure.

Problems with Schemaless Databases

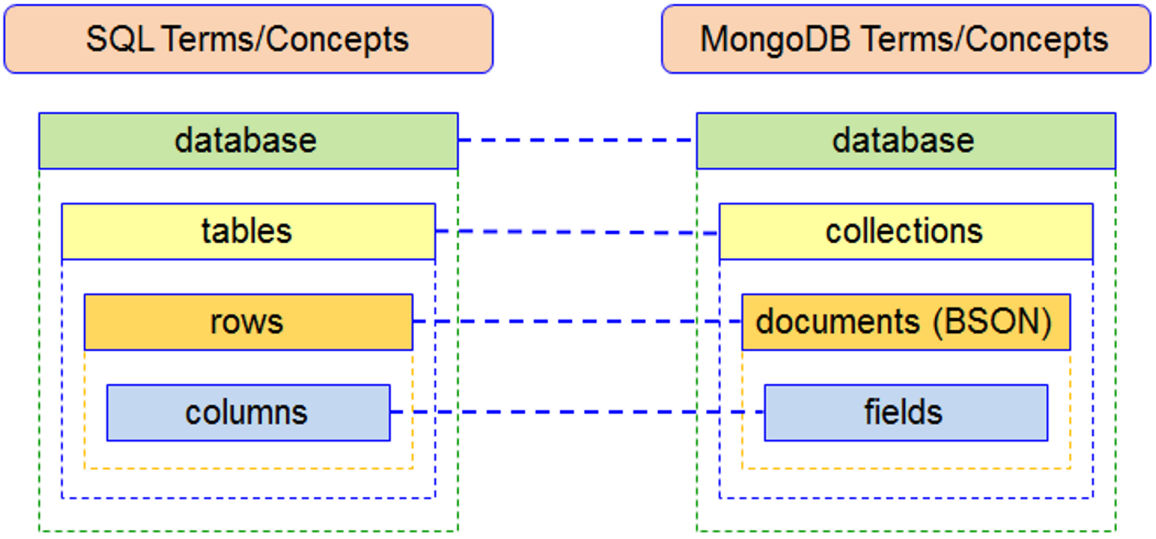
- **Inconsistent Database**: Without a uniform structure, data can become inconsistent. For example, one document might have a field that others don't, leading to unpredictable query results.
- **Runtime Errors**: Applications may expect certain fields or data types that are not present in all documents, leading to errors at runtime when the code tries to access or manipulate non-

existent fields.

- **Too Flexible**: The flexibility that makes NoSQL databases appealing for rapid development can become a liability for applications that require strict data integrity and consistency.

Upsides of Schemaless Databases

- **Speed**: Developers can iterate quickly without being constrained by a rigid database schema. This is particularly useful in the early stages of a project or when requirements are rapidly evolving.
- **Schema Evolution**: It's easier to adapt to changes in the application's data requirements, as there's no need to perform complex migrations or updates to the database schema.



Mongoose and Schema Enforcement

- **Mongoose Schemas**: Mongoose is an Object Data Modeling (ODM) library for MongoDB that allows developers to define schemas at the application level. It provides a layer of data validation and structure in a Node.js environment.
- **Application-Level Strictness**: While Mongoose enforces schema rules in the application code, it does not impose these constraints at the database level. MongoDB itself remains schemaless.
- **Potential for Erroneous Data**: Even with Mongoose schemas, it's still possible to insert data that doesn't conform to the defined schema directly into the database, bypassing the application's validation logic.

Why SQL?

SQL (Structured Query Language) databases, also known as relational databases, have been the cornerstone of data storage and management in software applications for decades. Their approach to data management offers several advantages, particularly in terms of data integrity, consistency, and reliability. Here's an elaboration on the structured approach of SQL databases and the four key aspects of using them:

1. Strict Schema

- **Define Your Schema**: Before inserting data into an SQL database, you must define a schema. This schema dictates the structure of the data, including the tables, columns, data types, and relationships between tables.
- **Data Integrity**: By requiring all data to adhere to the predefined schema, SQL databases ensure a high level of data integrity. Each column in a table is designed to hold data of a specific type, and relationships between tables are strictly enforced.
- **Schema Updates and Migrations**: As your application evolves, you may need to update the database schema. This typically involves performing migrations—carefully managed changes that may add, remove, or alter tables and columns without losing data.

2. Running the Database

- **Database Server**: An SQL database runs on a database server, which can be hosted locally on your development machine, on-premises in your data center, or in the cloud. Running the database involves setting it up, configuring it for performance and security, and ensuring it's accessible for data operations.

3. Connecting and Manipulating Data

- **Using a Library**: To interact with an SQL database, you typically use a library or an ORM (Object-Relational Mapping) tool that facilitates the connection and allows you to perform data operations in a more abstracted way, often using the programming language of your application.
- **Creating Tables and Defining Schemas**: Before you can store data, you need to create tables and define their schema. This includes specifying the columns, data types, primary keys, foreign keys, and any constraints to enforce data integrity.
- **Running Queries**: SQL databases are interacted with through SQL queries. These queries allow you to perform a variety of data operations, including:
  - **Insert**: Adding new records to a table.
  - **Update**: Modifying existing records based on specific criteria.
  - **Delete**: Removing records from a table.
  - **Select**: Retrieving data from one or more tables, often involving complex filtering, sorting, and joining operations.

Benefits of SQL Databases

- **Data Integrity and Consistency**: The strict schema and relational model of SQL databases ensure that data is stored in a consistent and reliable manner.
- **ACID Properties**: SQL databases are designed to guarantee ACID (Atomicity, Consistency, Isolation, Durability) properties, making them ideal for applications that require transactions to be processed reliably.
- **Complex Queries**: The SQL language provides powerful querying capabilities, allowing for complex data retrieval that can involve multiple tables and conditions.
- **Mature Ecosystem**: SQL databases have been around for a long time, resulting in a mature ecosystem of tools, libraries, and best practices.

Creating a PostgreSQL Database

Creating a PostgreSQL database can be done in several ways, depending on your environment and preferences. Here are a few methods:

1] Using Neon

- **Neon**: Neon is a cloud service that allows you to create and manage PostgreSQL databases without the need to handle the underlying infrastructure.
- **Steps**:
    1. Visit Neon's website and sign up for an account.
    2. Follow the instructions to create a new PostgreSQL server.
    3. Once created, you will be provided with a connection string that you can use to connect to your database from your application.
- **Connection String Example**:

```
postgresql://username:password@ep-broken-frost-69135494.us-east-2.aws.neon.tech/calm-gobbler-4
```

2] Using Docker Locally

- **Docker**: Docker allows you to run PostgreSQL in an isolated container on your local machine.
- **Steps**:
    1. Install Docker if you haven't already.
    2. Run the following command to start a PostgreSQL container:

```
docker run --name my-postgres -e POSTGRES_PASSWORD=mysecretpassword -d -p 5432:5432 pos
```

3. This command will download the PostgreSQL image if it's not already present, and start a new con named `my-postgres`.

- **Connection String Example**:

```
postgresql://postgres:mysecretpassword@localhost:5432/postgres?sslmode=disable
```

3] Using Docker on Windows

- **Windows Terminal with Docker**:
  1. Ensure Docker Desktop for Windows is installed and running.
  2. Open Windows Terminal or any command-line interface.
  3. To download and run the PostgreSQL image for the first time, use:

```
docker run --name my-postgres1 -e POSTGRES_PASSWORD=mysecretpassword -d -p 5432:5432 po
```

  4. If the image is already downloaded, you can start the container with:

```
docker start my-postgres1
```

  5. To connect to the PostgreSQL instance, use:

```
docker exec -it my-postgres1 psql -U postgres -d postgres
```

  6. Enter the password when prompted to access the PostgreSQL command line interface.
  7. Inside the PostgreSQL CLI, you can list all tables with:

```
\dt
```

Connection String

- The connection string is a vital piece of information that your application uses to connect to the database. It includes the username, password, host, port, and database name.
- **Format**:

```
postgresql://[user]:[password]@[host]:[port]/[database]?[options]
```

- This format is similar to what you might have seen with MongoDB and Mongoose, where the connection string is used to establish a connection to the database from your application code.

Understanding the Connection String Components

- `postgresql://` This is the protocol indicating that you are connecting to a PostgreSQL database.
- `username:password` Credentials for authenticating with the database.
- `host` The server where the database is hosted (e.g., localhost, a remote server, or a cloud service like Neon).
- `port` The port number on which the PostgreSQL server is listening (default is 5432).
- `database` The specific database you want to connect to.
- `options` Additional connection options such as SSL mode.

Understanding Vector Databases:

Let's take an example to understand vector databases more effectively, consider the following

```
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]
```

In the examples provided, the vectors for statements about "Harkirat" and "India" have similar coordinates because they contain similar words or concepts. The presence of identical numbers in different vectors indicates that those vectors represent statements with shared words or meanings. For instance, the repeated '2' in the vectors might indicate common words or a common structure in the statements, while unique identifiers like '3001' for "Harkirat" or '3' for "Chandigarh" show up in vectors representing statements about those specific entities.

Vector databases leverage this property to perform efficient similarity searches. When a query vector is provided, the database can quickly find other vectors with similar coordinates, which correspond to records containing similar words or concepts, thus retrieving relevant information based on semantic similarity.

Interact with PostgreSQL

When working with PostgreSQL databases, especially in the context of application development, it's common to use libraries or tools that facilitate connecting to, interacting with, and visualizing the data within these databases. Two such tools are `psql` and `pg`

Each serves different purposes and fits into different parts of the development workflow.

1. psql

- **What is psql?**
  - `psql` is a command-line interface (CLI) tool that allows you to interact with a PostgreSQL database server. It provides a terminal-based front-end to PostgreSQL, enabling users to execute SQL queries directly, inspect the database schema, and manage the database.

- **How to Connect to Your Database with psql?**
  - To connect to a PostgreSQL database using `psql`, you can use a command in the following format:

```
psql -h [host] -d [database] -U [user]
```

  - For example, based on the provided information:

```
psql -h p-broken-frost-69135494.us-east-2.aws.neon.tech -d database1 -U 100xdevs
```

  - Here, `h` specifies the host of the database server, `d` specifies the name of the database you want to connect to, and `U` specifies the username.

- **Installation and Usage**:
  - `psql` comes bundled with the PostgreSQL installation package. If you have PostgreSQL installed on your system, you likely already have `psql` available.
  - While not necessary for direct application development, `psql` is invaluable for database administration, debugging, and manual data inspection.

2. pg (node-postgres)

- **What is pg?**

- `pg`, also known as node-postgres, is a collection of Node.js modules for interfacing with your PostgreSQL database. It is non-blocking and designed specifically for use with Node.js. Similar to how `mongoose` is used for MongoDB, `pg` allows for interaction with PostgreSQL databases within a Node.js application.

- **How to Use pg in Your Application?**
  - To use `pg` in your Node.js application, you first need to install it via npm or yarn:

```
npm install pg
yarn add pg
```

  - After installing, you can connect to your PostgreSQL database using the `pg` library by creating a client and connecting it with your database's connection string:

```
javascriptconst { Client } = require('pg');

const client = new Client({
  connectionString: 'YourDatabaseConnectionStringHere'
});

client.connect();
```

  - You can then use this client to execute queries against your database.

- **Why Use pg?**
  - `pg` provides a programmatic way to connect to and interact with your PostgreSQL database directly from your Node.js application. It supports features like connection pooling, transactions, and streaming results. It's essential for building applications that require data persistence in a PostgreSQL database.

Creating a table schema

Creating a table and defining its schema is a fundamental step in working with SQL databases. This process involves specifying the structure of the data each table will hold. Let's break down the process and the components of a SQL statement used to create a table in PostgreSQL, using the table as an example.

```
users
```

Creating a Table in SQL

1. Initiate Table Creation

- Syntax: `CREATE TABLE users`
- **Description**: This command starts the creation of a new table named `users` in the database.

2. Define Columns and Constraints

- **Column Definition**: Each column in the table is defined with a name, data type, and possibly one or more constraints.
  - `id SERIAL PRIMARY KEY` :
    - `id` : Column name, typically used as a unique identifier for each row.
    - `SERIAL` : A PostgreSQL data type for auto-incrementing integers, ensuring each row has a unique ID.
    - `PRIMARY KEY` : A constraint that specifies the `id` column as the primary key, ensuring uniqueness and non-null values.
  - `username VARCHAR(50) UNIQUE NOT NULL` :
    - `username` : Column for storing the user's username.
    - `VARCHAR(50)` : Data type specifying a variable-length string of up to 50 characters.
    - `UNIQUE` : Ensures all values in this column are unique.
    - `NOT NULL` : Prevents null values, requiring every row to have a username.
  - `email VARCHAR(255) UNIQUE NOT NULL` :
    - Similar to `username` , but intended for the user's email address, allowing up to 255 characters.
  - `password VARCHAR(255) NOT NULL` :
    - Stores the user's password with the same data type as `email` , but without the `UNIQUE` constraint, as passwords can be non-unique.
  - `created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP` :
    - `created_at` : Column for storing the timestamp of when the user was created.
    - `TIMESTAMP WITH TIME ZONE` : Stores both a timestamp and a time zone.
    - `DEFAULT CURRENT_TIMESTAMP` : Automatically sets the value to the current timestamp when a new row is inserted.

Practical Steps

1. **Execute the CREATE TABLE Command**: Use the provided SQL statement to create the `users` table in your PostgreSQL database.

```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
);
```

2. **Verify Table Creation**: After running the command, you can verify the creation of the table by executing `\dt;` in the `psql` command-line interface. This command lists all tables in the current database, and you should see the `users` table listed among them.

Interacting with the database

Interacting with a database typically involves performing four fundamental operations, collectively known as CRUD operations: Create, Read, Update, and Delete. In the context of SQL databases, these operations are executed using SQL commands. Let's elaborate on how each of these operations is carried out in PostgreSQL, using the table as an example `users`

1. INSERT (Create)

- **Purpose**: To add new records to a table.
- **SQL Command**:

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

- **Explanation**: This command inserts a new row into the `users` table with values for `username`, `email`, and `password`. The `id` column is not specified because it's an auto-incrementing field (`SERIAL`), meaning PostgreSQL will automatically assign a unique `id` to each new row.

2. UPDATE

- **Purpose**: To modify existing records in a table.
- **SQL Command**:

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

- **Explanation**: This command updates the `password` for the user with the specified `email`. The `WHERE` clause is crucial as it determines which records are updated. Without it, all records

in the table would be updated.

3. DELETE

- **Purpose**: To remove records from a table.
- **SQL Command**:

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

- **Explanation**: This command deletes the row from the `users` table where the `id` is 1. Like with `UPDATE`, the `WHERE` clause specifies which records to delete. Omitting the `WHERE` clause would result in deleting all records in the table, which is rarely intended.

4. SELECT (Read)

- **Purpose**: To retrieve records from a table.
- **SQL Command**:

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

- **Explanation**: This command selects all columns ( ) for the row(s) in the `users` table where the `id` is 1. The `SELECT` statement is highly versatile, allowing for complex queries with various conditions, sorting, and joining multiple tables.

Practical Tips

- **Running Commands**: If you have `psql` installed locally, you can run these commands directly in your terminal to interact with your PostgreSQL database. This is a great way to practice and see the immediate effect of each operation.
- **Using pg Library**: For application development, especially in a Node.js environment, you'll eventually use the `pg` library to execute these operations programmatically. This allows your application to dynamically interact with the database based on user input or other logic.

> Understanding and being able to execute these four basic database operations is foundational for working with SQL databases. Whether you're manually testing commands in `psql` or integrating database operations into an application with the `pg` library, these CRUD operations form the basis of data manipulation and retrieval in relational databases.

Database Operations

To perform database operations from a Node.js application, you can use the `pg` library, a non-blocking PostgreSQL client for Node.js. This library allows you to connect to your PostgreSQL database and execute queries programmatically. Here's a step-by-step guide on how to connect to the database and perform basic operations using the library `pg`

Installing the `pg` Library

First, you need to install the `pg` library in your Node.js project. Run the following command in your project directory:

```
bashnpm install pg
```

Connecting to the Database

To connect to your PostgreSQL database, you need to create a `Client` instance with your database connection details and then call the `connect` method.

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

const client = new Client({
  host: 'my.database-server.com',
  port: 5334,
  database: 'database-name',
  user: 'database-user',
  password: 'secretpassword!!',
});

client.connect();
```

Querying the Database

Once connected, you can execute queries using the `query` method of the `Client` instance. Here's how to perform the four basic database operations:

INSERT

To insert data into your table:

```
javascriptconst insertResult = await client.query(
  "INSERT INTO users (username, email, password) VALUES ('username_here', 'user@example.co
```

```
);
console.log(insertResult);
```

UPDATE

To update existing data:

```
javascriptconst updateResult = await client.query(
  "UPDATE users SET password = 'new_password' WHERE email = 'user@example.com';"
);
console.log(updateResult);
```

DELETE

To delete data from your table:

```
javascriptconst deleteResult = await client.query(
  "DELETE FROM users WHERE id = 1;"
);
console.log(deleteResult);
```

SELECT

To retrieve data:

```
javascriptconst selectResult = await client.query(
  "SELECT * FROM users WHERE id = 1;"
);
console.log(selectResult.rows);
```

Creating a Table

You can also use the `pg` library to create tables. Here's an example function that creates a `users` table:

```
javascriptasync 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);
  await client.end();
}
```

```
createUsersTable();
```

Conclusion

The `pg` library provides a powerful interface for interacting with PostgreSQL databases from Node.js applications. By following the steps outlined above, you can connect to your database, execute queries, and manage your data effectively. For more advanced use cases, refer to the pg library documentation

Creating a Simple Node.js App

Creating a simple Node.js application that interacts with a PostgreSQL database using TypeScript involves several steps, from initializing the project to writing secure database interaction functions. Here's a concise guide to setting up your project and implementing basic database operations securely.

Step 1: Initialize a TypeScript Project

1. **Create a new directory** for your project and navigate into it.
2. **Initialize a new npm project**:

   ```
   npm init -y
   ```

3. **Initialize a TypeScript project**:

   ```
   npx tsc --init
   ```

4. **Configure TypeScript** by editing `tsconfig.json`:
   - Set `"rootDir": "./src"` to specify the source directory.
   - Set `"outDir": "./dist"` to specify the output directory for compiled JavaScript files.

Step 2: Install Dependencies

1. **Install the `pg` library** to interact with PostgreSQL:

   ```
   npm install pg
   ```

2. **Install TypeScript types for `pg`**:

   ```
   npm install @types/pg --save-dev
   ```

Step 3: Create a Simple Node.js App

Insert Data Function

Create a function to insert data into a table securely, using parameterized queries to prevent SQL injection.

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

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

  try {
    await client.connect();
    const insertQuery = "INSERT INTO users (username, email, password) VALUES ($1, $2, $3)
    const values = [username, email, password];
    const res = await client.query(insertQuery, values);
    console.log('Insertion success:', res);
  } catch (err) {
    console.error('Error during the insertion:', err);
  } finally {
    await client.end();
  }
}

// Example usageinsertData('username5', 'user5@example.com', 'user_password').catch(consol
```

Fetch Data Function

Implement a function to fetch user data from the database given an email, using parameterized queries for security.

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

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

  try {
    await client.connect();
```

1 co

Add a pu

```
    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]);
      return result.rows[0];
    } else {
      console.log('No user found with the given email.');
      return null;
    }
  } catch (err) {
    console.error('Error during fetching user:', err);
    throw err;
  } finally {
    await client.end();
```