# Contents

# 1 PostgreSQL

## 1.1 Introduction

PostgreSQL is a powerful open-source relational database management system (RDBMS) that is widely used for a variety of applications, from small projects to large-scale enterprise systems. It is known for its reliability, scalability, and robustness, and is often used for mission-critical applications where data integrity and availability are of utmost importance.

PostgreSQL supports a wide range of advanced features, including full ACID compliance, support for JSON and other NoSQL-style data types, and a powerful query optimizer that can handle complex queries with ease. It also has a large and active community of users and contributors, which means that there are many resources available for learning and troubleshooting.

1. Extensibility: As I mentioned earlier, PostgreSQL has a well-designed plugin architecture that allows developers to easily create extensions that add new functionality to the database. This means that PostgreSQL can be customized to meet the specific needs of a wide range of applications.
2. Full ACID Compliance: PostgreSQL is fully ACID compliant, which means that it guarantees data consistency and reliability even in the face of hardware failures, power outages, and other unexpected events. This level of data integrity is essential for mission-critical applications, and it sets PostgreSQL apart from many other databases.
3. Support for advanced data types: PostgreSQL supports a wide range of advanced data types, including arrays, hstore (a key-value store), and JSON. This makes it a great choice for applications that need to work with complex and heterogeneous data structures.
4. Powerful query optimizer: PostgreSQL has a powerful query optimizer that can handle complex queries with ease. This means that it can deliver fast query performance even on very large datasets.
5. Community support: PostgreSQL has a large and active community of users and contributors, which means that there are many resources available for learning and troubleshooting. The community is also committed to open-source principles, which means that PostgreSQL is constantly evolving and improving based on user feedback and contributions.

In PostgreSQL, tables are organized into a database, which is a container for related tables and other objects. Each table in PostgreSQL is defined by a set of columns, which specify the data type and constraints for each column, and a set of rows, which represent the actual data stored in the table.

```
CREATE TABLE customers (
  id SERIAL PRIMARY KEY,
  first_name VARCHAR(50),
  last_name VARCHAR(50),
  email VARCHAR(100) UNIQUE,
  created_at TIMESTAMP DEFAULT NOW()
);
```

Once the table is created, you can insert data into it using the INSERT statement:

```
INSERT INTO customers (first_name, last_name, email)
VALUES ('John', 'Doe', 'john.doe@example.com');
```

and select with:

```
SELECT * FROM customers;
```

### 1.1.1 CRUD Operations

CRUD stands for Create, Read, Update, and Delete. These operations are the four basic functions that can be performed on most database systems, including PostgreSQL.

1. Create: The Create operation is used to create new records in a database table. In PostgreSQL, this is usually done with the INSERT statement.
2. Read: The Read operation is used to retrieve records from a database table. In PostgreSQL, this is usually done with the SELECT statement.
3. Update: The Update operation is used to modify existing records in a database table. In PostgreSQL, this is usually done with the UPDATE statement.
4. Delete: The Delete operation is used to remove records from a database table. In PostgreSQL, this is usually done with the DELETE statement.

These operations are essential for managing and manipulating data within a database, and understanding how to perform CRUD operations is a fundamental skill for anyone working with databases.

### 1.1.2 Primary Key - Foreign Key

In a relational database, a primary key is a column or set of columns that uniquely identifies each row in a table. A foreign key is a column or set of columns that refers to the primary key of another table, creating a relationship between the two tables. For example, consider two tables: customers and orders. The customers table might have a primary key of customer_id, which uniquely identifies each customer, and the orders table might have a foreign key of customer_id, which refers to the customer_id column in the customers table. This establishes a relationship between the customers and orders tables such that each order is associated with a particular customer. Foreign keys can also be used to enforce referential integrity, which means that the values in the foreign key column must match the values in the corresponding primary key column in the referenced table. This ensures that there are no orphaned records in the database, where a foreign key references a non-existent primary key.

### 1.1.3 Datatypes

PostgreSQL supports a wide range of data types, including:

- Numeric types: INTEGER, BIGINT, SMALLINT, DECIMAL, NUMERIC, REAL, and DOUBLE PRECISION.
- Character types: CHAR, VARCHAR, and TEXT.
- Date/time types: DATE, TIME, TIMESTAMP, and INTERVAL.
- Boolean type: BOOLEAN.
- Enumerated types: ENUM.
- Geometric types: POINT, LINE, LSEG, BOX, PATH, POLYGON, and CIRCLE.
- Network address types: INET and CIDR.
- Bit string types: BIT and VARBIT.
- UUID type: UUID.

- XML type: XML.
- JSON and JSONB types: JSON and JSONB.
- Arrays: Arrays of any data type can be created by appending [] to the type name. For example, INTEGER[], TEXT[], or POINT[].
- Composite types: Composite types are user-defined types that combine multiple fields of different data types into a single type.
- Range types: Range types represent a range of values of a single data type, such as an integer range or a date range.
- Custom types: Custom types can be created by defining a new data type with a set of rules and behaviors.

These data types allow PostgreSQL to handle a wide variety of data and provide a lot of flexibility for working with different types of data.

### 1.1.4 Devs need to know

things that a developer should know when working with PostgreSQL:

- SQL syntax: Developers should be familiar with SQL syntax, as PostgreSQL uses SQL for creating tables, querying data, and managing the database.
- Data types: Developers should be familiar with the data types available in PostgreSQL, as this affects how data is stored and queried.
- Indexes: Developers should understand how to use indexes to optimize queries and improve performance.
- Constraints: Developers should know how to use constraints, such as primary keys and foreign keys, to enforce data integrity and ensure consistency.
- Transactions: Developers should be familiar with how transactions work in PostgreSQL, and how to use them to ensure that changes to the database are made in a consistent and reliable way.
- Backup and recovery: Developers should understand how to backup and recover the database in case of a disaster or data loss.
- Performance tuning: Developers should be able to tune the performance of the database by optimizing queries, adjusting settings, and using caching techniques.
- Security: Developers should be aware of security best practices and how to implement them in the context of PostgreSQL, such as setting up user accounts, roles, and permissions to restrict access to the database.

## 1.2 Entity-Relationship ER

ER stands for "Entity-Relationship" and it is a modeling technique used to represent the relationships between different entities in a database.

An entity can represent a physical object or a concept, and it is usually represented as a rectangle. Attributes of an entity, such as its name, age, or gender, are represented as ovals connected to the entity rectangle by lines. For example, in a database for a school, "students," "teachers," and "courses" would all be entities, with attributes like "name," "age," "address," and "subject."

A relationship represents the connection between entities. It is represented by a diamond shape with lines connecting it to the entities involved in the relationship. The lines connecting the entities to

the relationship indicate the cardinality of the relationship, which is the number of instances of one entity that can be associated with a single instance of another entity.

There are three main types of relationships in ER modeling:

### 1.2.1 One-to-One

One-to-One (1:1): Each instance of one entity is associated with only one instance of another entity. To handle a one-to-one relationship in PostgreSQL, you can simply add a foreign key to one of the tables that references the primary key of the other table. Here's an example:

Suppose you have two tables, "employees" and "employee_details". Each employee has exactly one set of employee details, and each set of employee details belongs to exactly one employee. This is a one-to-one relationship that can be represented by adding a foreign key to the "employee_details" table that references the primary key of the "employees" table.

The "employees" table has a primary key called "id", and the "employee_details" table has a foreign key called "employee_id" that references the "id" column of the "employees" table. Here's how you can create the tables and the foreign key constraint in PostgreSQL:

```
CREATE TABLE employees (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255)
);
CREATE TABLE employee_details (
  id SERIAL PRIMARY KEY,
  employee_id INTEGER UNIQUE REFERENCES employees(id),
  salary NUMERIC(10,2),
  hire_date DATE
);
```

In this example, the "employee_details" table has a foreign key called "employee_id" that references the "id" column of the "employees" table. The UNIQUE constraint on the "employee_id" column ensures that each set of employee details is unique and belongs to exactly one employee.

To query data from these tables, you can use JOIN statements to combine the data from the two tables based on the foreign key constraint. For example, to get the name and salary of a particular employee, you can use the following query:

```
SELECT employees.name, employee_details.salary
FROM employees
JOIN employee_details ON employees.id = employee_details.employee_id
WHERE employees.name = 'John Doe';
```

This query uses a JOIN statement to combine data from the "employees" and "employee_details" tables based on the foreign key constraint, filtering the results by the name of a particular employee. Similarly, to get the name and hire date of an employee given their salary, you can use the following query:

```
SELECT employees.name, employee_details.hire_date
FROM employees
```

```
JOIN employee_details ON employees.id = employee_details.employee_id
WHERE employee_details.salary = 50000;
```

This query uses a JOIN statement to combine data from the "employees" and "employee_details" tables based on the foreign key constraint, filtering the results by the salary of the employee.

### 1.2.2 One-to-Many

One-to-Many (1:N): Each instance of one entity is associated with one or more instances of another entity. you can handle a one-to-many relationship by creating a foreign key constraint on the "many" side of the relationship that references the primary key of the "one" side. Here's an example:

Suppose you have two tables: "students" and "courses". Each student can take many courses, but each course can be taken by many students. This is a many-to-many relationship that can be represented as a one-to-many relationship by introducing an intermediate table called "enrollments" that connects students to courses.

The "students" table has a primary key called "id", and the "courses" table also has a primary key called "id". The "enrollments" table has two foreign keys, one that references the "id" column of the "students" table, and another that references the "id" column of the "courses" table. Here's how you can create the tables and the foreign key constraint in PostgreSQL:

```
CREATE TABLE students (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255)
);

CREATE TABLE courses (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255)
);

CREATE TABLE enrollments (
  student_id INTEGER REFERENCES students(id),
  course_id INTEGER REFERENCES courses(id),
  PRIMARY KEY (student_id, course_id)
);
```

In this example, the foreign key constraint is created by adding the REFERENCES keyword to the student_id and course_id columns of the enrollments table, followed by the name of the referenced table and column. The PRIMARY KEY constraint on the enrollments table ensures that each combination of student and course is unique. To query data from these tables, you can use JOIN statements to combine the data from multiple tables based on the foreign key constraints. For example, to get the name of the courses that a particular student is enrolled in, you can use the following query:

```
SELECT courses.name
FROM courses
JOIN enrollments ON courses.id = enrollments.course_id
JOIN students ON enrollments.student_id = students.id
```

```
WHERE students.name = 'John Doe';
```

This query uses JOIN statements to combine data from the courses, enrollments, and students tables, filtering the results by the name of a particular student.

### 1.2.3 Many-to-Many

Many-to-Many (N:M): Each instance of one entity is associated with one or more instances of another entity, and each instance of the other entity is associated with one or more instances of the first entity. To handle a many-to-many relationship in PostgreSQL, you can use an intermediate table with foreign key constraints that reference the primary keys of the related tables. Here's an example:

Suppose you have two tables, "students" and "courses". Each student can take many courses, and each course can be taken by many students. This is a many-to-many relationship that can be represented by introducing an intermediate table called "enrollments" that connects students to courses.

The "students" table has a primary key called "id", the "courses" table has a primary key called "id", and the "enrollments" table has two foreign keys that reference the primary keys of the "students" and "courses" tables. Here's how you can create the tables and the foreign key constraints in PostgreSQL:

```
CREATE TABLE students (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255)
);

CREATE TABLE courses (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255)
);

CREATE TABLE enrollments (
  student_id INTEGER REFERENCES students(id),
  course_id INTEGER REFERENCES courses(id),
  PRIMARY KEY (student_id, course_id)
);
```

In this example, the "enrollments" table has two foreign keys, one that references the "id" column of the "students" table, and another that references the "id" column of the "courses" table. The PRIMARY KEY constraint on the "enrollments" table ensures that each combination of student and course is unique.

To query data from these tables, you can use JOIN statements to combine the data from multiple tables based on the foreign key constraints. For example, to get the name of the courses that a particular student is enrolled in, you can use the following query:

```
SELECT courses.name
FROM courses
JOIN enrollments ON courses.id = enrollments.course_id
```

```
JOIN students ON enrollments.student_id = students.id
WHERE students.name = 'John Doe';
```

This query uses JOIN statements to combine data from the "courses", "enrollments", and "students" tables, filtering the results by the name of a particular student.

Similarly, to get the names of the students who are enrolled in a particular course, you can use the following query:

```
SELECT students.name
FROM students
JOIN enrollments ON students.id = enrollments.student_id
JOIN courses ON enrollments.course_id = courses.id
WHERE courses.name = 'Calculus';
```

This query uses JOIN statements to combine data from the "students", "enrollments", and "courses" tables, filtering the results by the name of a particular course.

To create an ER model, you start by identifying the entities and their attributes, and then identifying the relationships between them. You can use various notations and symbols to represent the entities and relationships, but the basic concepts remain the same.

ER modeling is a useful tool for designing databases because it provides a clear visual representation of the relationships between different entities. It helps to ensure that the database is well-organized, with minimal redundancy and maximum efficiency, and can make it easier to modify and maintain the database over time.

### 1.2.4 How to create good ER?

Writing a good ER (Entity-Relationship) diagram is important for designing a well-organized database structure that can efficiently store and retrieve data. Here are some steps you can follow to create a good ER diagram:

1. Identify entities: Identify the main entities that will be included in your database, such as customers, products, orders, or employees. These entities should be relevant to the domain of the database.
2. Define relationships: Determine the relationships between the entities. For example, customers can place orders, so there is a relationship between the "customers" and "orders" entities. Relationships can be one-to-one, one-to-many, or many-to-many.
3. Add attributes: Add attributes to each entity, which are characteristics or properties of the entity. For example, the "customers" entity might have attributes such as name, address, and email.
4. Determine primary keys: Choose a primary key for each entity, which is a unique identifier that can be used to reference the entity. Primary keys can be generated automatically by the database, or they can be chosen based on the attributes of the entity.
5. Draw the diagram: Draw the ER diagram using standardized symbols and notation. Use boxes to represent entities, lines to represent relationships, and diamonds to represent cardinality.
6. Refine the diagram: Review the ER diagram and refine it as needed. Make sure that each entity has a primary key, each relationship is clearly defined, and each attribute is relevant to the entity.

7. Normalize the diagram: Normalize the ER diagram to ensure that it is structured in a way that avoids data redundancy and inconsistencies. This involves breaking down entities into smaller, more specialized entities, and creating separate tables for each entity.
8. Validate the diagram: Validate the ER diagram to ensure that it accurately represents the requirements of the database. You can do this by reviewing the diagram with stakeholders, testing the database using sample data, or using a modeling tool that checks for errors.

By following these steps, you can create a well-designed ER diagram that forms the basis for a reliable and efficient database structure.

## 1.3 Best practise for creating table

1. Use meaningful and consistent naming conventions for tables, columns, and other objects.
2. Create primary keys and unique constraints to ensure that each row in a table is unique and to help enforce data integrity.
3. Use foreign keys to create relationships between tables. This can help ensure data consistency and make it easier to retrieve related data.
4. Use indexes to optimize queries. Indexes can help improve the performance of search and sort operations on tables.
5. Normalize your data to reduce data redundancy and improve data integrity. Normalization involves breaking down a table into smaller tables to avoid repeating data.
6. Use appropriate data types for your columns. Choosing the right data type for a column can improve storage efficiency and help enforce data constraints.
7. Use default values and constraints to ensure that data is consistent and accurate.
8. Use comments and documentation to describe the purpose of each table and column.

### 1.3.1 Normalization

Normalization is the process of organizing and structuring a relational database to reduce redundancy and improve data integrity. In a normalized database, each table represents a single logical entity or concept, and each piece of information is stored in only one place. This helps to avoid data inconsistencies, reduce data redundancy, and make it easier to maintain and update the database.

Normalize your tables to avoid data redundancy and ensure that each piece of data is stored only once. This can help reduce the risk of data inconsistencies and improve the performance of your queries. Normalization is the process of organizing a database in a way that reduces data redundancy and eliminates data inconsistencies. It is an important part of designing a good table structure in PostgreSQL, as it can help improve the performance of your queries and reduce the risk of data inconsistencies. There are several levels of normalization, but the most common ones are:

2. First Normal Form (1NF): This level of normalization requires that each column in a table should contain only atomic values. In other words, each column should contain a single value, rather than a list or a group of values.
3. Second Normal Form (2NF): This level of normalization requires that a table should be in 1NF and that all non-key columns should be functionally dependent on the primary key. In other words, each non-key column should be dependent on the primary key and not on other non-key columns.
4. Third Normal Form (3NF): This level of normalization requires that a table should be in 2NF and that all non-key columns should be dependent only on the primary key, and not on other

non-key columns. In other words, there should be no transitive dependencies between non-key columns.

The process of normalization involves breaking down larger tables into smaller ones and creating relationships between them using foreign keys. This can result in a larger number of tables, but it also ensures that each table contains only related information and that each piece of information is stored in only one place. This can help to improve data quality and make the database more efficient to query and update. Normalization is an important aspect of database design and is often used in combination with other techniques, such as indexing and partitioning, to improve the performance and scalability of a database.

let's say we have a table called `orders` that looks like this:

| Order ID | Customer Name | Product Name | Product Price | Order Date |
|---|---|---|---|---|
| 1 | John Smith | iPhone | 1000 | 2022-01-01 |
| 2 | John Smith | MacBook | 2000 | 2022-01-02 |
| 3 | Jane Doe | iPad | 500 | 2022-01-03 |
| 4 | Jane Doe | iMac | 3000 | 2022-01-04 |

In this table, we can see that we have a few repeating values. For example, both John Smith's orders have the same customer name, and the product names are repeated as well. To achieve normalization, we would want to separate this table into multiple tables so that we're not repeating data unnecessarily. We could start by creating a table for `customers` that looks like this:

| Customer ID | Customer Name |
|---|---|
| 1 | John Smith |
| 2 | Jane Doe |

Next, we could create a table for `products` that looks like this:

Finally, we can create a table for `orders` that includes only the information that's specific to each order:

| Product ID | Product Name | Product Price |
|---|---|---|
| 1 | iPhone | 1000 |
| 2 | MacBook | 2000 |
| 3 | iPad | 500 |
| 4 | iMac | 3000 |

Now we can see that each table includes only the information that's specific to that entity. The `orders` table includes only the order ID, the IDs of the customer and product associated with the order, and the order date. The `customers` table includes only the customer ID and name, and the `products` table includes only the product ID, name, and price. This is an example of how we can use normalization to reduce data duplication and make our database more efficient and maintainable.

## 1.4 Indexes

In PostgreSQL, an index is a data structure that improves the speed of data retrieval operations on a database table. An index is similar to an index in a book, which provides a quick way to look up information in the book. Similarly, an index in a database provides a quick way to locate data in a table without having to scan the entire table. When you create an index on a table in PostgreSQL, it creates a separate data structure that stores a copy of the indexed data in a way that makes it easier to search. This structure is optimized for rapid access, and it allows PostgreSQL to quickly find the specific rows that match a query's search criteria. PostgreSQL supports several types of indexes, including B-tree, hash, GiST, GIN, and SP-GiST indexes. Each type of index is optimized for different types of queries, and choosing the right type of index for your use case can significantly improve query performance.

### 1.4.1 Indexes easier

In PostgreSQL, an index is like the index at the back of a book. Just like how the index helps you quickly find the page number of a specific topic in a book, an index in a database helps you quickly find the rows that match a specific search criteria.

When you create an index in PostgreSQL, it creates a separate data structure that stores a copy of the indexed data in a way that makes it easier to search. This data structure is optimized for fast searches and allows PostgreSQL to quickly locate the specific rows that match a query's search criteria.

By using indexes, PostgreSQL can significantly improve the speed of queries that search for specific data in a table.

## 1.5 Transactions

In PostgreSQL, a transaction is a sequence of SQL statements that are executed as a single unit of work. The main goal of a transaction is to ensure data consistency and integrity. A transaction can be used to combine multiple operations into a single logical operation that can be executed atomically.

When a transaction begins, it creates a new transaction ID, and any changes made to the database during the transaction are only visible to the transaction itself. If the transaction is successful, the changes are committed to the database, and if the transaction fails, the changes are rolled back, and the database returns to its previous state.

PostgreSQL provides the ACID properties for transactions:

1. Atomicity: All operations in a transaction are treated as a single unit, and they either all succeed or all fail.
2. Consistency: The database remains in a consistent state after a transaction. This means that the database's rules and constraints are not violated.
3. Isolation: Each transaction is isolated from other transactions, and changes made by one transaction are not visible to other transactions until they are committed.
4. Durability: Once a transaction is committed, the changes made during the transaction are permanent and survive even if there is a system failure.

In PostgreSQL, transactions are managed using the BEGIN, COMMIT, and ROLLBACK statements. The BEGIN statement marks the start of a new transaction, and the COMMIT statement is used to commit the transaction to the database. The ROLLBACK statement is used to roll back the transaction and undo any changes made during the transaction.

Here is an example of using transactions in PostgreSQL:

```
BEGIN;

UPDATE accounts SET balance = balance - 100 WHERE account_id = 123;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 456;

COMMIT;
```

In this example, we start a transaction with the BEGIN statement, and then update two rows in the accounts table. If both updates succeed, we commit the transaction with the COMMIT statement, which makes the changes permanent. If there is an error in one of the updates, we can roll back the entire transaction using the ROLLBACK statement, which undoes both updates.

## 1.6 Backups and persistency

PostgreSQL is known for its reliability, stability, and data integrity features. It has a reputation for being a highly reliable and robust database system, capable of handling large volumes of data with high concurrency and reliability.

PostgreSQL provides a number of features for ensuring data integrity and preventing data loss, such as:

1. ACID compliance: PostgreSQL is fully ACID-compliant, meaning that transactions are atomic, consistent, isolated, and durable.
2. Write-ahead logging: PostgreSQL uses write-ahead logging (WAL) to ensure that data changes are recorded to a log file before they are written to the database. This provides a high degree of protection against data loss, as it allows the database to recover quickly from crashes or other types of failures.
3. Point-in-time recovery: PostgreSQL's point-in-time recovery (PITR) feature allows you to recover a database to a specific point in time, which is useful for disaster recovery.
4. Replication: PostgreSQL supports a variety of replication options, including streaming replication, logical replication, and synchronous replication. These features provide high availability and load balancing capabilities.

In the context of databases, a replica refers to a copy of the primary database that is kept in sync with the primary database through some form of replication mechanism. Replicas are used for various purposes, such as improving the availability of the database, distributing the read load across multiple servers, and providing failover in case the primary database becomes unavailable.

In the case of PostgreSQL, replicas can be created using a feature called streaming replication. With streaming replication, changes made to the primary database are streamed to the replicas in near-real-time, ensuring that the replicas are always up-to-date. Replicas can be used for read-only queries, backup, and disaster recovery.

There are different types of replicas in PostgreSQL, including hot standby replicas, warm standby

replicas, and asynchronous replicas. Hot standby replicas are always available for read-only queries and can be promoted to become the new primary database in case the current primary fails. Warm standby replicas are similar to hot standby replicas but require some manual intervention to be promoted to primary. Asynchronous replicas, on the other hand, may have some lag in data replication and are used primarily for backup and disaster recovery purposes.

Regarding backups, PostgreSQL provides several mechanisms for creating backups, including:

1. pg_dump: This is a command-line tool that creates a text-based backup of a PostgreSQL database. It allows you to specify various options for customizing the backup, such as which tables to include, whether to include schema information, and whether to create a plain text or binary backup.
2. Continuous Archiving and Point-in-Time Recovery (PITR): In addition to the WAL, PostgreSQL also provides a continuous archiving feature which archives the WAL files in a separate location. These archived files can be used to recover the database to a specific point in time.
3. Third-party backup solutions: There are also third-party backup tools available for PostgreSQL, such as Barman and pgBackRest, which provide additional backup and recovery features.

### 1.6.1 How to backup a PostgreSQL database

Setting up a PostgreSQL replication system typically involves the following steps:

1. Choose the replication method: PostgreSQL supports several methods for replication, including streaming replication, logical replication, and trigger-based replication. Choose the method that best suits your requirements and constraints.
2. Configure the primary server: The primary server is the server that hosts the main database that will be replicated. You will need to configure the primary server to allow replication, which typically involves setting up the replication user, enabling WAL (Write-Ahead Logging), and specifying the replica servers.
3. Set up the replica servers: The replica servers are the servers that will host the replicated copies of the primary database. You will need to configure the replica servers to receive the replication data from the primary server. This typically involves setting up the replication user, enabling replication, and specifying the primary server.
4. Start the replication process: Once the primary and replica servers are configured, you can start the replication process. This typically involves taking a backup of the primary database, restoring the backup to the replica servers, and starting the replication process.
5. Test the replication system: Once the replication is set up, you should test the system to ensure that it is working correctly. This typically involves running some test queries on the replica servers to verify that they are returning the expected results.

## 1.7 Basics of the languages

Overview of the basic syntax of PostgreSQL:

1. Statements: In PostgreSQL, each command or instruction is called a statement. Statements are typically terminated with a semicolon (;). Examples of statements include CREATE TABLE, SELECT, INSERT, UPDATE, and DELETE.
2. Clauses: Statements are composed of one or more clauses. Clauses are used to specify various details and options for the statement. For example, the SELECT statement includes a FROM

clause to specify the table from which to retrieve data, and an optional WHERE clause to filter the rows based on certain criteria.
3. Keywords: Keywords are reserved words that have a specific meaning in PostgreSQL. Examples of keywords include SELECT, FROM, WHERE, AND, OR, INSERT, UPDATE, DELETE, CREATE, and TABLE. Keywords are not case-sensitive, so SELECT and select are equivalent.
4. Identifiers: Identifiers are used to name tables, columns, and other database objects. Identifiers are case-sensitive, so customer and Customer are considered two different identifiers. Identifiers that contain spaces or special characters must be enclosed in double quotes, such as "First Name".
5. Literals: Literals are values that are used in statements, such as string literals, numeric literals, and date/time literals. String literals are enclosed in single quotes, such as 'John', while numeric literals are just numeric values, such as 42. Date/time literals are enclosed in single quotes and follow a specific format, such as '2023-02-17' for a date or '2023-02-17 12:34:56' for a timestamp.
6. Comments: Comments are used to add notes and explanations to statements. Single-line comments begin with – and continue until the end of the line, while multi-line comments begin with /* and end with */.

### 1.7.1 Most common queries:

1. SELECT: The SELECT statement is used to retrieve data from one or more tables. It is one of the most commonly used statements in PostgreSQL.
2. INSERT: The INSERT statement is used to insert new rows into a table. It is typically used in combination with the SELECT statement to copy data from one table to another.
3. UPDATE: The UPDATE statement is used to update existing rows in a table. It is typically used in combination with the SELECT statement to copy data from one table to another.
4. DELETE: The DELETE statement is used to delete rows from a table. It is typically used in combination with the SELECT statement to copy data from one table to another.
5. CREATE TABLE: The CREATE TABLE statement is used to create a new table. It is typically used in combination with the SELECT statement to copy data from one table to another.
6. ALTER TABLE: The ALTER TABLE statement is used to modify an existing table. It is typically used in combination with the SELECT statement to copy data from one table to another.
7. DROP TABLE: The DROP TABLE statement is used to delete an existing table. It is typically used in combination with the SELECT statement to copy data from one table to another.

More useful way:

1. Select all columns from a table: SELECT * FROM table_name;
2. Select specific columns from a table: SELECT column1, column2, ... FROM table_name;
3. Filter results with WHERE clause: SELECT column1, column2, ... FROM table_name WHERE condition;
4. Order results with ORDER BY clause: SELECT column1, column2, ... FROM table_name ORDER BY column1 ASC/DESC;
5. Limit the number of results with LIMIT clause: SELECT column1, column2, ... FROM table_name LIMIT n;
6. Combine multiple tables with JOIN clause: SELECT column1, column2, ... FROM table1

JOIN table2 ON table1.column = table2.column;

7. Group results with GROUP BY clause: SELECT column1, SUM(column2) FROM table_name GROUP BY column1;
8. Filter grouped results with HAVING clause: SELECT column1, SUM(column2) FROM table_name GROUP BY column1 HAVING condition;
9. Calculate aggregate functions on column: SELECT AVG(column1), MAX(column2), MIN(column3) FROM table_name;
10. Use subqueries to get nested data: SELECT column1, column2, ... FROM table_name WHERE column1 IN (SELECT column1 FROM another_table WHERE condition);

### 1.7.2 Conditionals statement

**1.7.2.1 CASE statement** The CASE statement is a conditional expression that allows you to perform different actions based on different conditions. It can be used in the SELECT, WHERE, and ORDER BY clauses of a query. The basic syntax of the CASE statement is as follows:

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

The CASE statement starts with the keyword CASE, followed by one or more WHEN clauses. Each WHEN clause specifies a condition and a result to be returned if that condition is true. The final ELSE clause provides a default result to be returned if none of the conditions are true.

Here's an example of how to use the CASE statement in a SELECT clause:

```
SELECT product_name,
       CASE
           WHEN price > 50 THEN 'Expensive'
           WHEN price > 25 THEN 'Moderate'
           ELSE 'Inexpensive'
       END AS price_category
FROM products;
```

In this example, the query selects the product name and uses a CASE statement to categorize the product based on its price. If the price is greater than 50, the result will be 'Expensive'. If the price is between 25 and 50, the result will be 'Moderate'. Otherwise, the result will be 'Inexpensive'. The AS keyword is used to give a name to the new column created by the CASE statement.

You can also use the CASE statement in the WHERE clause of a query to filter data based on a condition. Here's an example:

```
SELECT product_name, price
FROM products
WHERE CASE
          WHEN price > 50 THEN 'Expensive'
          WHEN price > 25 THEN 'Moderate'
```

```
        ELSE 'Inexpensive'
    END = 'Moderate';
```

In this example, the query selects product name and price from the products table, but only returns products that fall into the 'Moderate' price category, as determined by the CASE statement in the WHERE clause.

The CASE statement is a powerful tool that allows you to customize the results of your queries based on specific conditions. It's especially useful for creating more complex reports and analyses.

#### 1.7.2.2 HAVING statement

The HAVING clause is used in conjunction with the GROUP BY clause to filter the results of a query based on a condition that applies to groups rather than individual rows. It is typically used to filter aggregated data.

The basic syntax of the HAVING statement is as follows:

```
SELECT column1, column2, aggregate_function(column3)
FROM table_name
GROUP BY column1, column2
HAVING condition;
```

In this syntax, the GROUP BY clause is used to group the rows of the table by the specified columns, and the HAVING clause is used to filter the groups based on the specified condition.

Here's an example to illustrate how the HAVING statement works. Suppose we have a table called orders that contains information about customer orders, and we want to find the total sales for each customer. We can use the following query:

```
SELECT customer_id, SUM(total)
FROM orders
GROUP BY customer_id
HAVING SUM(total) > 1000;
```

In this query, we are grouping the rows of the orders table by customer_id, and using the SUM aggregate function to calculate the total sales for each customer. The HAVING clause is used to filter the results to show only the customers whose total sales are greater than 1000.

The HAVING statement can be used with other aggregate functions, such as COUNT, MIN, MAX, and AVG, to filter the results of a query based on the results of the aggregate function. It can also be combined with the WHERE clause to provide more complex filtering of the data.

It's important to note that the HAVING clause is used to filter the groups, not the individual rows within each group. If you need to filter individual rows, you should use the WHERE clause instead.

### 1.7.3 Join tables

joining tables allows you to combine data from two or more tables based on a common column or set of columns. Joining tables is a fundamental operation in relational databases, and it allows you to create more complex and meaningful queries by retrieving data that is spread across multiple tables.

There are several types of joins in PostgreSQL, including:

- INNER JOIN: Returns only the rows that have matching values in both tables. To perform an inner join, you use the JOIN keyword followed by the name of the table you want to join, and then the ON keyword to specify the join condition. Here's an example:

```
SELECT *
FROM orders
JOIN customers ON orders.customer_id = customers.id;
```

This will return all rows from both tables where the customer_id in the orders table matches the id in the customers table.

- LEFT JOIN (or LEFT OUTER JOIN): Returns all rows from the left table and the matching rows from the right table. If there are no matching rows in the right table, the result will contain NULL values for the right table columns. To perform a left join, you use the LEFT JOIN keywords instead of JOIN. Here's an example:

```
SELECT *
FROM customers
LEFT JOIN orders ON customers.id = orders.customer_id;
```

This will return all rows from the customers table, along with any matching rows from the orders table. If there are no matching rows in the orders table, the result will contain NULL values for the orders table columns.

- RIGHT JOIN (or RIGHT OUTER JOIN): Returns all rows from the right table and the matching rows from the left table. If there are no matching rows in the left table, the result will contain NULL values for the left table columns. To perform a right join, you use the RIGHT JOIN keywords instead of JOIN. Here's an example:

```
SELECT *
FROM orders
RIGHT JOIN customers ON orders.customer_id = customers.id;
```

This will return all rows from the orders table, along with any matching rows from the customers table. If there are no matching rows in the customers table, the result will contain NULL values for the customers table columns.

- FULL JOIN (or FULL OUTER JOIN): Returns all rows from both tables, with NULL values for the columns that do not have a matching value in the other table. To perform a full join, you use the FULL JOIN keywords instead of JOIN. Here's an example:

```
SELECT *
FROM customers
FULL JOIN orders ON customers.id = orders.customer_id;
```

This will return all rows from both the customers and orders tables, with NULL values for the columns that do not have a matching value in the other table.

In addition to the basic join types, there are other features and functions you can use to customize your joins and make them more powerful, such as subqueries, aggregates, and window functions. By mastering the different types of joins and learning how to combine them with other features of PostgreSQL, you can create complex queries that retrieve and combine data from multiple tables in a variety of ways.

### 1.7.4 Aggregations functions

commonly used aggregate functions in PostgreSQL:

- COUNT: This function counts the number of rows in a table or a result set that meet a specific condition. For example, you can use COUNT to count the number of orders that a customer has made:

```
SELECT customer_id, COUNT(*) AS num_orders
FROM orders
WHERE customer_id = 123
GROUP BY customer_id;
```

In this example, we are counting the number of orders that were made by the customer with ID 123. The GROUP BY clause is used to group the results by customer_id, so we get the count for each customer.

- SUM: This function calculates the sum of a set of values. For example, you can use SUM to calculate the total revenue for a specific product:

```
SELECT product_id, SUM(price * quantity) AS total_revenue
FROM order_items
WHERE product_id = 'P001'
GROUP BY product_id;
```

In this example, we are calculating the total revenue for product P001. The GROUP BY clause is used to group the results by product_id, so we get the total revenue for each product.

- AVG: This function calculates the average of a set of values. For example, you can use AVG to calculate the average price of all products:

```
SELECT AVG(price) AS average_price
FROM products;
```

In this example, we are calculating the average price of all products. The result will be a single value that represents the average price.

- MAX/MIN: These functions return the maximum or minimum value in a set of values. For example, you can use MAX to find the highest price of any product:

```
SELECT MAX(price) AS highest_price
FROM products;
```

In this example, we are finding the highest price of any product. The result will be a single value that represents the highest price.

- GROUP_CONCAT: This function concatenates the values of a column for each group into a single string. For example, you can use GROUP_CONCAT to list all the products that a customer has ordered:

```
SELECT customer_id, GROUP_CONCAT(product_name) AS products_ordered
FROM orders
JOIN order_items ON orders.order_id = order_items.order_id
```

```
JOIN products ON order_items.product_id = products.product_id
GROUP BY customer_id;
```

In this example, we are listing all the products that each customer has ordered. The GROUP BY clause is used to group the results by customer_id, so we get the list of products for each customer.

### 1.7.5 Subqueries

A subquery, also known as a nested query, is a query that is embedded within another query. The subquery can be used to retrieve data that is used in the main query's WHERE clause or to create a derived table that is used in the main query's FROM clause. Subqueries can be correlated, meaning that they reference columns from the outer query, or uncorrelated, meaning that they do not reference columns from the outer query.

example Suppose we have two tables, orders and order_items, with a one-to-many relationship between them. We want to find the total quantity of items ordered for each order, and then find the average quantity of items ordered across all orders.

Here's how we can use a subquery to accomplish this:

```
SELECT AVG(order_total)
FROM (
  SELECT order_id, SUM(quantity) AS order_total
  FROM order_items
  GROUP BY order_id
) subquery
```

In this example, the inner subquery groups the order_items table by order_id and sums the quantity for each order. The outer query then takes the average of the order_total column from the subquery to get the average quantity of items ordered per order.

### 1.7.6 Window fuctions

A window function is a function that performs a calculation across a set of rows that are defined by a window specification. The window specification can be used to define a partition of rows, an ordering of rows, and an optional frame of rows to include or exclude from the calculation. Window functions are useful for calculating running totals, ranking data, and calculating moving averages or other statistical measures. Suppose we have a table sales with columns `date` and `amount`, representing the daily sales for a business. We want to find the total sales for each day, as well as the running total of sales up to that day.

Here's how we can use a window function to accomplish this:

```
SELECT date, amount, SUM(amount) OVER (ORDER BY date) AS running_total
FROM sales
```

In this example, the `SUM(amount) OVER (ORDER BY date)` expression is a window function that calculates the running total of sales up to each day. The `ORDER BY` clause in the window function specifies that the running total should be calculated based on the ordering of the `date` column. The result set includes the `date`, `amount`, and `running_total` columns for each row in the `sales` table.

## 1.8 General tips

additional best practices for managing databases:

1. Regularly backup your database: It is important to create regular backups of your database to ensure that you have a copy of your data in case of data loss or corruption.
2. Set up monitoring and alerting: Configure your database to monitor important metrics, such as CPU usage, memory usage, and disk usage, and set up alerts to notify you when these metrics exceed predefined thresholds.
3. Use stored procedures and functions: By using stored procedures and functions, you can reduce the amount of redundant code that you write, improve performance, and increase security.
4. Limit access to your database: Restrict access to your database to authorized users only, and grant the minimum level of permissions necessary for each user to perform their tasks.
5. Optimize your database performance: You can optimize your database performance by creating indexes, tuning queries, and avoiding long-running transactions.
6. Upgrade and maintain your database: Regularly upgrade your database to the latest version and apply any necessary patches or updates to ensure that it remains secure and up-to-date.
7. Document your database: Document your database schema, including the purpose of each table and column, and any relationships between tables. This will help other developers understand your database and make it easier to maintain over time.

## 1.9 How can i store . . . ?

### 1.9.1 array

you can store arrays in a table by using the array data type. The array data type allows you to store a variable-length array of values of the same data type. To store an array properly in PostgreSQL, follow these steps:

Choose the array data type for the column. PostgreSQL provides a range of built-in array data types, such as integer[], text[], timestamp[], and so on. Choose the array data type that best matches the data you want to store in the column.

Specify the dimensions of the array. When you define the column, you must specify the number of dimensions for the array. PostgreSQL supports arrays with one or more dimensions.

Insert data into the array column. To insert data into an array column, you can use the array constructor syntax, which is a comma-separated list of values enclosed in curly braces. For example, to insert an array of integers into a column named my_array, you can use the following SQL statement:

```
INSERT INTO my_table (my_array) VALUES ('{1, 2, 3, 4, 5}');
```

You can also use the ARRAY function to create an array from a list of values. For example, to insert an array of strings, you can use the following SQL statement:

```
INSERT INTO my_table (my_array) VALUES (ARRAY['foo', 'bar', 'baz']);
```

Query data from the array column. To query data from an array column, you can use the array access operator, which is a pair of square brackets enclosing the index or indices of the element or subarray you want to access. For example, to retrieve the second element of an array column named my_array, you can use the following SQL statement:

```
SELECT my_array[2] FROM my_table;
```

You can also use array functions to manipulate the array data, such as array_agg to aggregate the elements of an array column into a single array, or unnest to expand an array column into multiple rows.

By using the array data type in PostgreSQL, you can store and manipulate arrays of values in a table. This can be useful for storing data that naturally forms an array, such as a list of tags, a series of events, or a set of related items.

### 1.9.2 json

you can store JSON data in a table by using the json or jsonb data type. The json data type stores JSON data as text in a format that preserves the JSON structure, while the jsonb data type stores JSON data in a binary format that allows for more efficient storage and querying.

To store a JSON value in a json or jsonb column, you can use the ::json or ::jsonb cast operator to convert a text value to JSON format. For example, to insert a JSON value into a json column named my_json, you can use the following SQL statement:

```
INSERT INTO my_table (my_json) VALUES ('{"name": "John", "age": 30}'::json);
```

Or, if you want to use the jsonb data type, you can use the following SQL statement:

```
INSERT INTO my_table (my_json) VALUES ('{"name": "John", "age": 30}'::jsonb);
```

To query JSON data from a json or jsonb column, you can use the JSON operators and functions that PostgreSQL provides. For example, you can use the -> operator to extract a single field from a JSON object, or the -» operator to extract a single field as text. You can also use the #> operator to extract a nested field or a path of fields, or the #» operator to extract a nested field or a path of fields as text.

Here's an example query that retrieves the name field from a JSON object stored in a json column named my_json:

```
SELECT my_json->'name' FROM my_table;
```

Storing JSON data in a database can have disadvantages, including:

Limited query support: While PostgreSQL provides powerful JSON functions and operators for querying JSON data, they are still not as flexible or powerful as the SQL query language for structured data.

## 1.10 Object-relational mapping (ORM)

Relational mapping, also known as object-relational mapping (ORM), is the process of mapping database tables and their relationships to object-oriented programming constructs, such as classes and objects. Here are some general guidelines to follow when creating a good relational mapping:

1. Design your database schema carefully: A well-designed database schema can make it easier to map your data to objects. Make sure your tables are normalized and use appropriate data types and constraints.

2. Use an appropriate ORM tool: There are many ORM tools available for PostgreSQL, such as SQLAlchemy, Django ORM, and TypeORM. Choose a tool that is appropriate for your needs and that provides the functionality you need to map your data to objects.
3. Use consistent naming conventions: Use consistent naming conventions for your database tables and columns to make it easier to map them to objects. For example, if you have a table called customers, the corresponding object in your code should be called Customer.

Map tables to classes and columns to properties: Map each table in your database to a class in your code and each column to a property in the class. The name of the class and its properties should reflect the name of the corresponding table and columns.

Use relationships to map related data: Use relationships to map related data between tables. For example, if you have a customers table and an orders table, you can create a relationship between them so that you can retrieve a customer's orders easily.

Use lazy loading and caching: Use lazy loading and caching to optimize performance. Lazy loading means that related data is not loaded until it is actually needed, while caching means that data that has already been loaded is stored in memory to reduce the number of database queries.

Test your mapping thoroughly: Test your mapping thoroughly to ensure that it is working correctly and that you are retrieving and storing data properly. Make sure to test both normal and edge cases to ensure that your code is robust and reliable.

By following these guidelines, you can create a well-designed relational mapping that can make it easier to work with your PostgreSQL database in your object-oriented code.

## 1.11 Python integration

To use PostgreSQL with Python, you need to use a Python library that supports PostgreSQL connectivity. One popular library for this purpose is psycopg2.

Here's a brief example of how to use psycopg2 to connect to a PostgreSQL database and execute a query:

```
import psycopg2

# Connect to the PostgreSQL database
conn = psycopg2.connect(
    host="your_host",
    database="your_database",
    user="your_username",
    password="your_password"
)

# Create a cursor object to execute queries
cur = conn.cursor()

# Define the SQL statement to create the table
create_table_query = """
CREATE TABLE employees (
```

```
    id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    age INTEGER,
    job_title VARCHAR(50)
)
"""

# Execute the query to create the table
cur.execute(create_table_query)

# Define the SQL statement to insert data into the table
insert_query = """
INSERT INTO employees (name, age, job_title)
VALUES
    ('Alice', 28, 'Manager'),
    ('Bob', 35, 'Developer'),
    ('Charlie', 42, 'Sales Representative')
"""

# Execute the query to insert data into the table
cur.execute(insert_query)

# Define the SQL statement to select all data from the table
select_query = """
SELECT * FROM employees
"""

# Execute the query to select data from the table
cur.execute(select_query)

# Fetch all rows of data from the table
rows = cur.fetchall()

# Print the data
for row in rows:
    print(row)

# Commit the changes to the database
conn.commit()

# Close the cursor and connection
cur.close()
conn.close()
```

## 1.12 Difference with MySQL

PostgreSQL and MySQL are both popular relational database management systems, but there are some differences between the two in terms of functionality and syntax.

One key difference is that PostgreSQL is more feature-rich and is often considered more powerful than MySQL. For example, PostgreSQL has more advanced support for data types, arrays, and JSON data, and it includes a wider range of SQL functions and operators.

Another difference is in their approach to data integrity. PostgreSQL places a stronger emphasis on enforcing data integrity through the use of constraints and foreign keys, whereas MySQL has a more flexible approach to data integrity, which can sometimes lead to data inconsistencies.

In terms of syntax, PostgreSQL and MySQL have some differences in the way they handle SQL statements. For example, PostgreSQL uses the ILIKE operator for case-insensitive string comparisons, whereas MySQL uses the LIKE operator with the LOWER function.

Overall, both PostgreSQL and MySQL are powerful and capable database management systems, and the choice between them often comes down to the specific needs of the application being developed.

## 1.13 POSTGRESQL Interview

1. SQL basics, including data types, CRUD operations, and querying with SELECT, WHERE, and JOIN statements.
2. Relational database concepts, such as primary keys, foreign keys, and indexes.
3. Data normalization and database design.
4. Advanced SQL topics, such as subqueries, window functions, and aggregate functions.
5. Postgres-specific features, such as procedural languages (PL/pgSQL, PL/Python), stored procedures, and triggers.
6. Postgres data types and functions, including arrays, JSON, and full-text search.
7. High availability and data replication, including hot standby and streaming replication.
8. Backup and recovery strategies, such as pg_dump and pg_restore.
9. Postgres performance tuning and optimization, including query planning and execution, connection pooling, and caching.
10. Security and authentication, including user and role management, access control, and encryption.