Day 2: Basic Database Concepts

Hour 1-2: Basic Concepts and SQL Commands

- Database, schema, tables, rows, and columns
- Data types and constraints
- Primary keys and foreign keys

Hour 3-4: Basic SQL Commands

- SELECT, INSERT, UPDATE, DELETE
- Filtering and sorting data
- Aggregate functions

Day 2: Basic Database Concepts

Hour 1-2: Basic Concepts and SQL Commands

- Database, schema, tables, rows, and columns
- Data types and constraints
- Primary keys and foreign keys

Basic Concepts and SQL Commands in PostgreSQL

Key Concepts

1. **Database:**
   - A PostgreSQL database is a collection of schemas. It is where you store data.
   - Example: A company database might store data related to employees, departments, products, etc.

2. **Schema:**
   - A schema is a logical container inside a database. It holds tables, views, indexes, sequences, functions, and other objects.
   - Example: In a company database, you might have schemas for public (default), hr (human resources), sales, etc.

3. **Table:**
   - A table is a collection of related data entries and consists of rows and columns.
   - Example: An employees table might store details like employee ID, name, position, and salary.

4. **Row:**
   - A row (or record) in a table represents a single, implicitly structured data item in a table.
   - Example: A row in the employees table might contain data about a single employee.

5. **Column:**
   - A column (or field) in a table is a set of data values of a particular type, one for each row of the table.
   - Example: The salary column in the employees table stores the salary of each employee.

Basic SQL Commands

1. **Creating a Database:**

```
CREATE DATABASE company_db;
```

2. **Creating a Schema:**

```
CREATE SCHEMA hr;
```

3. **Creating a Table:**

```
CREATE TABLE hr.employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    email VARCHAR(100) UNIQUE,
    hire_date DATE,
    salary NUMERIC(8, 2)
);
```

4. **Inserting Data into a Table:**

```
INSERT INTO hr.employees (first_name, last_name, email, hire_date, salary)
VALUES ('John', 'Doe', 'john.doe@example.com', '2023-07-01', 50000.00);
```

5. **Selecting Data from a Table:**

```
SELECT employee_id, first_name, last_name, email, hire_date, salary
FROM hr.employees;
```

6. **Updating Data in a Table:**

```
UPDATE hr.employees
SET salary = 55000.00
```

```
WHERE employee_id = 1;
```

7. **Deleting Data from a Table:**

```
DELETE FROM hr.employees
WHERE employee_id = 1;
```

8. **Adding a Column to a Table:**

```
ALTER TABLE hr.employees
ADD COLUMN phone_number VARCHAR(20);
```

9. **Dropping a Column from a Table:**

```
ALTER TABLE hr.employees
DROP COLUMN phone_number;
```

10. **Dropping a Table:**

```
DROP TABLE hr.employees;
```

11. **Dropping a Schema:**

```
DROP SCHEMA hr CASCADE;
```

12. **Dropping a Database:**

```
DROP DATABASE
```

Database Queries Explanations

1. **Creating a Database:**

```
CREATE DATABASE company_db;
```

- **Explanation:** This command creates a new database named company_db. A database is a container for schemas, tables, and other database objects.

Schema Queries

2. **Creating a Schema:**

```
CREATE SCHEMA hr;
```

- **Explanation:** This command creates a new schema named hr. A schema is a logical container within a database, used to organize and manage tables and other database objects.

Table Queries

3. **Creating a Table:**

```
CREATE TABLE hr.employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    email VARCHAR(100) UNIQUE,
    hire_date DATE,
    salary NUMERIC(8, 2)
);
```

- **Explanation:** This command creates a new table named employees within the hr schema. The table has the following columns:
  - employee_id: An integer column that automatically increments with each new row, serving as the primary key.
  - first_name: A string column with a maximum length of 50 characters.
  - last_name: A string column with a maximum length of 50 characters.
  - email: A string column with a maximum length of 100 characters, unique for each row.
  - hire_date: A date column.
  - salary: A numeric column that allows up to 8 digits, including 2 decimal places.

Data Manipulation Queries

4. **Inserting Data into a Table:**

```
INSERT INTO hr.employees (first_name, last_name, email, hire_date, salary)
VALUES ('John', 'Doe', 'john.doe@example.com', '2023-07-01', 50000.00);
```

- **Explanation:** This command inserts a new row into the employees table with the specified values for first_name, last_name, email, hire_date, and salary.

5. **Selecting Data from a Table:**

```
SELECT employee_id, first_name, last_name, email, hire_date, salary
FROM hr.employees;
```

- **Explanation:** This command retrieves all rows from the employees table, displaying the employee_id, first_name, last_name, email, hire_date, and salary columns.

6. **Updating Data in a Table:**

```
UPDATE hr.employees
```

```
SET salary = 55000.00
WHERE employee_id = 1;
```

- **Explanation:** This command updates the salary column to 55000.00 for the row where the employee_id is 1. The WHERE clause specifies which rows to update.

7. **Deleting Data from a Table:**

```
DELETE FROM hr.employees
WHERE employee_id = 1;
```

- **Explanation:** This command deletes the row from the employees table where the employee_id is 1. The WHERE clause specifies which rows to delete.

Altering Table Structure

8. **Adding a Column to a Table:**

```
ALTER TABLE hr.employees
ADD COLUMN phone_number VARCHAR(20);
```

- **Explanation:** This command adds a new column named phone_number to the employees table. The column is of type VARCHAR with a maximum length of 20 characters.

9. **Dropping a Column from a Table:**

```
ALTER TABLE hr.employees
DROP COLUMN phone_number;
```

- **Explanation:** This command removes the phone_number column from the employees table.

Dropping Objects

10. **Dropping a Table:**

```
DROP TABLE hr.employees;
```

- **Explanation:** This command deletes the employees table from the hr schema, along with all the data stored in it.

11. **Dropping a Schema:**

DROP SCHEMA hr CASCADE;

- **Explanation:** This command deletes the hr schema and all the objects contained within it. The CASCADE keyword ensures that all dependent objects (tables, functions, etc.) are also deleted.

12. **Dropping a Database:**

    DROP DATABASE company_db;

- **Explanation:** This command deletes the company_db database and all its contents. This operation cannot be undone, so use it with caution.

Summary

These SQL commands cover the basics of creating and managing databases, schemas, tables, and data within PostgreSQL. Each command is essential for performing typical database operations, such as setting up the database structure, adding and modifying data, and cleaning up resources. Understanding these commands is fundamental for working with PostgreSQL or any relational database management system.

Database, schema, tables, rows, and columns postgres examples for Healthcare domain

PostgreSQL Database Structure: Healthcare Domain Examples

In PostgreSQL, understanding the concepts of databases, schemas, tables, rows, and columns is fundamental for organizing and managing data efficiently. Below are examples demonstrating these concepts within the context of a healthcare domain.

1. Database

A database is a collection of schemas, tables, and other objects. In PostgreSQL, you can create a database specifically for your healthcare system.

Creating a Database

```
CREATE DATABASE healthcare_db;
```

2. Schema

A schema is a logical container within a database that groups together tables and other objects. Schemas help in organizing database objects and can also provide security boundaries.

Creating a Schema

```
CREATE SCHEMA healthcare;
```

3. Tables

Tables are the primary structures in a database where data is stored. In the healthcare domain, you might have tables such as patients, doctors, appointments, and departments.

Creating Tables

Patients Table

```
CREATE TABLE healthcare.patients (
    patient_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    date_of_birth DATE,
    gender VARCHAR(10),
    contact_number VARCHAR(15),
    address TEXT
);
```

Doctors Table

```
CREATE TABLE healthcare.doctors (
    doctor_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    speciality VARCHAR(50),
    contact_number VARCHAR(15),
    email VARCHAR(50)
);
```

Appointments Table

```
CREATE TABLE healthcare.appointments (
    appointment_id SERIAL PRIMARY KEY,
    patient_id INT REFERENCES healthcare.patients(patient_id),
    doctor_id INT REFERENCES healthcare.doctors(doctor_id),
    appointment_date TIMESTAMP,
    status VARCHAR(20)
);
```

Departments Table

```
CREATE TABLE healthcare.departments (
    department_id SERIAL PRIMARY KEY,
    department_name VARCHAR(50),
    location VARCHAR(100)
);
```

4. Rows

Rows represent individual records in a table. Each row in a table corresponds to a single record with values for each column.

Inserting Rows

Insert Data into Patients Table

```
INSERT INTO healthcare.patients (first_name, last_name, date_of_birth, gender, contact_number, address)
VALUES ('John', 'Doe', '1980-01-01', 'Male', '123-456-7890', '123 Main St, Anytown, USA');
```

Insert Data into Doctors Table

```
INSERT INTO healthcare.doctors (first_name, last_name, speciality, contact_number, email)
VALUES ('Alice', 'Smith', 'Cardiology', '234-567-8901', 'alice.smith@example.com');
```

Insert Data into Appointments Table

```
INSERT INTO healthcare.appointments (patient_id, doctor_id, appointment_date, status)
VALUES (1, 1, '2024-07-20 10:00:00', 'Scheduled');
```

Insert Data into Departments Table

```
INSERT INTO healthcare.departments (department_name, location)
VALUES ('Cardiology', 'Building A, Floor 3');
```

5. Columns

Columns define the structure of a table and store data for each attribute of the entity represented by the table. Columns have specific data types and constraints.

Adding a Column to an Existing Table

Add Email to Patients Table

```
ALTER TABLE healthcare.patients ADD COLUMN email VARCHAR(50);
```

Example Queries

Select All Patients

```
SELECT * FROM healthcare.patients;
```

Select Specific Columns from Doctors Table

```
SELECT first_name, last_name, specialty FROM healthcare.doctors;
```

Join Patients and Appointments

```
SELECT p.first_name, p.last_name, a.appointment_date, a.status
FROM healthcare.patients p
JOIN healthcare.appointments a ON p.patient_id = a.patient_id;
```

Count Appointments by Doctor

```
SELECT d.first_name, d.last_name, COUNT(a.appointment_id) AS appointment_count
FROM healthcare.doctors d
JOIN healthcare.appointments a ON d.doctor_id = a.doctor_id
GROUP BY d.first_name, d.last_name;
```

Summary

- **Database**: The healthcare_db database holds all healthcare-related data.
- **Schema**: The healthcare schema organizes the tables and objects within the database.
- **Tables**:
  - patients stores patient information.
  - doctors stores doctor information.
  - appointments stores appointment details.
  - departments stores department details.
- **Rows**: Individual records in each table represent entities like a single patient or appointment.
- **Columns**: Define the attributes of the entities, such as first_name, last_name, and appointment_date.

These examples demonstrate how to create and manage a PostgreSQL database for a healthcare domain, ensuring efficient organization and retrieval of critical data.

Data types and constraints postgres examples for Healthcare domain

Data Types and Constraints in PostgreSQL: Healthcare Domain Examples

In PostgreSQL, data types define the kind of data that can be stored in each column, and constraints enforce rules on the data. Below are examples of commonly used data types and constraints within the context of a healthcare domain.

1. Data Types

PostgreSQL supports a wide range of data types. Here are some commonly used data types in a healthcare database:

- **SERIAL**: Auto-incrementing integer, often used for primary keys.
- **VARCHAR(n)**: Variable-length character string.
- **TEXT**: Variable-length character string with no specific maximum length.
- **INTEGER**: Whole number.
- **DATE**: Calendar date (year, month, day).
- **TIMESTAMP**: Date and time.

- **BOOLEAN**: True or false.
- **NUMERIC(p, s)**: Exact numeric of selectable precision.

2. Constraints

Constraints are rules applied to columns to ensure data integrity. Common constraints include:

- **PRIMARY KEY**: Uniquely identifies each row in a table.
- **FOREIGN KEY**: Ensures referential integrity between tables.
- **NOT NULL**: Ensures that a column cannot have a NULL value.
- **UNIQUE**: Ensures that all values in a column are unique.
- **CHECK**: Ensures that values in a column satisfy a specific condition.

Data Types with Examples

PostgreSQL supports various data types to accommodate different kinds of data and use cases. Here is a detailed explanation of some commonly used PostgreSQL data types, including JSON and JSONB, with examples:

Numeric Types

1. **Integer Types:**
   - smallint: Stores 2-byte integer values. Range: -32768 to 32767.
   - integer: Stores 4-byte integer values. Range: -2147483648 to 2147483647.
   - bigint: Stores 8-byte integer values. Range: -9223372036854775808 to 9223372036854775807.

```
CREATE TABLE numeric_example (
    id serial PRIMARY KEY,
    small_number smallint,
    medium_number integer,
    large_number bigint
);

INSERT INTO numeric_example (small_number, medium_number, large_number) VALUES (100, 20000, 3000000000);
```

2. **Serial Types:**

   serial: Auto-incrementing 4-byte integer.

   bigserial: Auto-incrementing 8-byte integer.

```
CREATE TABLE serial_example (
    id serial PRIMARY KEY,
    big_id bigserial
```

```
);
```

3. **Decimal and Floating-Point Types:**
- decimal or numeric: Variable precision number.
- real: 4-byte floating point number.
- double precision: 8-byte floating point number.

```
CREATE TABLE decimal_example (
    decimal_value decimal(10, 2),
    real_value real,
    double_value double precision
);

INSERT INTO decimal_example (decimal_value, real_value, double_value) VALUES (12345.67,
1.23, 123456789.123456);
```

Character Types

1. **Fixed-Length Character:**
- char(n): Fixed length, blank-padded.

```
CREATE TABLE char_example (
    fixed_length char(10)
);

INSERT INTO char_example (fixed_length) VALUES ('abc');
```

2. **Variable-Length Character:**
- varchar(n): Variable length with a limit.
- text: Variable unlimited length.

```
CREATE TABLE varchar_example (
    variable_length varchar(50),
    unlimited_length text
);

INSERT INTO varchar_example (variable_length, unlimited_length) VALUES ('Hello, World!', 'This is
a long text field.');
```

Date/Time Types

1. **Date and Time:**
- date: Stores date values.
- time: Stores time values without time zone.
- timestamp: Stores both date and time values.
- timestamptz: Stores both date and time values with time zone.

```
CREATE TABLE datetime_example (
    event_date date,
    event_time time,
    event_timestamp timestamp,
    event_timestamptz timestamptz
);
```

```
INSERT INTO datetime_example (event_date, event_time, event_timestamp, event_timestamptz)
VALUES ('2024-07-17', '13:45:00', '2024-07-17 13:45:00', '2024-07-17 13:45:00+00');
```

Boolean Type

1. **Boolean:**
   - boolean: Stores true, false, or null.

```
CREATE TABLE boolean_example (
    is_active boolean
);

INSERT INTO boolean_example (is_active) VALUES (true), (false), (null);
```

JSON Types

1. **JSON:**
   - json: Stores JSON data as text.

```
CREATE TABLE json_example (
    json_data json
);

INSERT INTO json_example (json_data) VALUES ('{"name": "John", "age": 30, "city": "New York"}');
```

2. **JSONB:**
   - jsonb: Stores JSON data in a binary format for faster processing.

```
CREATE TABLE jsonb_example (
    jsonb_data jsonb
);

INSERT INTO jsonb_example (jsonb_data) VALUES ('{"name": "Jane", "age": 25, "city": "San
Francisco"}');
```

Array Types

1. **Array:**
   - array: Stores a list of values.

```
CREATE TABLE array_example (
    integer_array integer[],
    text_array text[]
);

INSERT INTO array_example (integer_array, text_array) VALUES ('{1, 2, 3}', '{"apple", "banana",
"cherry"}');
```

UUID Type

1. **UUID:**

- uuid: Stores universally unique identifiers.

```
CREATE TABLE uuid_example (
    id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
    name text
);

INSERT INTO uuid_example (name) VALUES ('Sample Entry');
```

Hstore Type

1. **Hstore:**
- hstore: Stores key-value pairs.

```
CREATE EXTENSION hstore;

CREATE TABLE hstore_example (
    attributes hstore
);

INSERT INTO hstore_example (attributes) VALUES ('color => "blue", size => "large"');
```

Composite Types

1. **Composite:**
- Composite types allow the creation of complex data types.

```
CREATE TYPE address AS (
    street text,
    city text,
    zip_code text
);

CREATE TABLE composite_example (
    id serial PRIMARY KEY,
    home_address address
);

INSERT INTO composite_example (home_address) VALUES (ROW('123 Main St', 'Springfield',
'12345'));
```

These examples illustrate the diverse range of data types available in PostgreSQL, catering to various data storage needs in a healthcare database. Understanding these data types is essential for efficient database design and optimal performance.

Examples of Tables with Data Types and Constraints

Patients Table

```
CREATE TABLE healthcare.patients (
   patient_id SERIAL PRIMARY KEY,
   first_name VARCHAR(50) NOT NULL,
   last_name VARCHAR(50) NOT NULL,
   date_of_birth DATE NOT NULL,
   gender VARCHAR(10) CHECK (gender IN ('Male', 'Female', 'Other')),
   contact_number VARCHAR(15) UNIQUE,
   email VARCHAR(50),
   address TEXT,
   registration_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

- **Data Types**: SERIAL, VARCHAR, DATE, TIMESTAMP, TEXT
- **Constraints**: PRIMARY KEY, NOT NULL, CHECK, UNIQUE, DEFAULT

Doctors Table

```
CREATE TABLE healthcare.doctors (
   doctor_id SERIAL PRIMARY KEY,
   first_name VARCHAR(50) NOT NULL,
   last_name VARCHAR(50) NOT NULL,
   specialty VARCHAR(50) NOT NULL,
   contact_number VARCHAR(15) UNIQUE,
   email VARCHAR(50) UNIQUE
);
```

- **Data Types**: SERIAL, VARCHAR
- **Constraints**: PRIMARY KEY, NOT NULL, UNIQUE

Appointments Table

```
CREATE TABLE healthcare.appointments (
   appointment_id SERIAL PRIMARY KEY,
   patient_id INT NOT NULL REFERENCES healthcare.patients(patient_id) ON DELETE CASCADE,
   doctor_id INT NOT NULL REFERENCES healthcare.doctors(doctor_id),
   appointment_date TIMESTAMP NOT NULL,
   status VARCHAR(20) CHECK (status IN ('Scheduled', 'Completed', 'Cancelled'))
);
```

- **Data Types**: SERIAL, INT, TIMESTAMP, VARCHAR
- **Constraints**: PRIMARY KEY, NOT NULL, FOREIGN KEY, CHECK

Departments Table

```
CREATE TABLE healthcare.departments (
   department_id SERIAL PRIMARY KEY,
   department_name VARCHAR(50) NOT NULL UNIQUE,
   location VARCHAR(100) NOT NULL
);
```

- **Data Types**: SERIAL, VARCHAR
- **Constraints**: PRIMARY KEY, NOT NULL, UNIQUE

Example Inserts with Data Types and Constraints

Insert Data into Patients Table

```
INSERT INTO healthcare.patients (first_name, last_name, date_of_birth, gender, contact_number, email, address)
VALUES ('John', 'Doe', '1980-01-01', 'Male', '123-456-7890', 'john.doe@example.com', '123 Main St, Anytown, USA');
```

Insert Data into Doctors Table

```
INSERT INTO healthcare.doctors (first_name, last_name, specialty, contact_number, email)
VALUES ('Alice', 'Smith', 'Cardiology', '234-567-8901', 'alice.smith@example.com');
```

Insert Data into Appointments Table

```
INSERT INTO healthcare.appointments (patient_id, doctor_id, appointment_date, status)
VALUES (1, 1, '2024-07-20 10:00:00', 'Scheduled');
```

Insert Data into Departments Table

```
INSERT INTO healthcare.departments (department_name, location)
VALUES ('Cardiology', 'Building A, Floor 3');
```

Summary

By using appropriate data types and constraints, you can ensure data integrity, enforce business rules, and improve the reliability of your healthcare database. This approach helps maintain high-quality data, which is critical for effective patient care and operational efficiency in the healthcare domain.

Primary Keys and Foreign Keys in PostgreSQL: Healthcare Domain Examples

In PostgreSQL, primary keys and foreign keys are essential for establishing relationships between tables and ensuring data integrity. Here are detailed examples of how to use primary keys and foreign keys in the context of a healthcare database.

1. Primary Keys

A primary key is a column (or a set of columns) that uniquely identifies each row in a table. It ensures that no two rows have the same primary key value.

Creating Tables with Primary Keys

Patients Table

```
CREATE TABLE healthcare.patients (
    patient_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    date_of_birth DATE NOT NULL,
    gender VARCHAR(10) CHECK (gender IN ('Male', 'Female', 'Other')),
    contact_number VARCHAR(15) UNIQUE,
    email VARCHAR(50),
    address TEXT,
    registration_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

- **Primary Key**: patient_id ensures each patient has a unique identifier.

Doctors Table

```
CREATE TABLE healthcare.doctors (
    doctor_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    specialty VARCHAR(50) NOT NULL,
    contact_number VARCHAR(15) UNIQUE,
    email VARCHAR(50) UNIQUE
);
```

- **Primary Key**: doctor_id ensures each doctor has a unique identifier.

2. Foreign Keys

A foreign key is a column (or a set of columns) that establishes a link between the data in two tables. It ensures referential integrity by making sure that the value in a foreign key column must match a value in the primary key column of the referenced table.

Creating Tables with Foreign Keys

Appointments Table

```
CREATE TABLE healthcare.appointments (
    appointment_id SERIAL PRIMARY KEY,
    patient_id INT NOT NULL REFERENCES healthcare.patients(patient_id) ON DELETE CASCADE,
    doctor_id INT NOT NULL REFERENCES healthcare.doctors(doctor_id),
    appointment_date TIMESTAMP NOT NULL,
    status VARCHAR(20) CHECK (status IN ('Scheduled', 'Completed', 'Cancelled'))
);
```

- **Foreign Keys**:

- patient_id references patients(patient_id) ensuring that each appointment is linked to a valid patient.
- doctor_id references doctors(doctor_id) ensuring that each appointment is linked to a valid doctor.

Departments Table

```
CREATE TABLE healthcare.departments (
    department_id SERIAL PRIMARY KEY,
    department_name VARCHAR(50) NOT NULL UNIQUE,
    location VARCHAR(100) NOT NULL
);
```

Doctor Departments Table (Associative Table)

```
CREATE TABLE healthcare.doctor_departments (
    doctor_id INT NOT NULL REFERENCES healthcare.doctors(doctor_id) ON DELETE CASCADE,
    department_id INT NOT NULL REFERENCES healthcare.departments(department_id) ON
DELETE CASCADE,
    PRIMARY KEY (doctor_id, department_id)
);
```

- **Primary Keys**: Composite primary key (doctor_id, department_id) ensures each doctor can be linked to multiple departments and vice versa.
- **Foreign Keys**:
- doctor_id references doctors(doctor_id) ensuring the doctor exists.
- department_id references departments(department_id) ensuring the department exists.

Example Inserts with Primary and Foreign Keys

Insert Data into Patients Table

```
INSERT INTO healthcare.patients (first_name, last_name, date_of_birth, gender, contact_number,
email, address)
VALUES ('John', 'Doe', '1980-01-01', 'Male', '123-456-7890', 'john.doe@example.com', '123 Main St,
Anytown, USA');
```

Insert Data into Doctors Table

```
INSERT INTO healthcare.doctors (first_name, last_name, specialty, contact_number, email)
VALUES ('Alice', 'Smith', 'Cardiology', '234-567-8901', 'alice.smith@example.com');
```

Insert Data into Appointments Table

```
INSERT INTO healthcare.appointments (patient_id, doctor_id, appointment_date, status)
VALUES (1, 1, '2024-07-20 10:00:00', 'Scheduled');
```

- **Explanation**: patient_id and doctor_id must refer to valid entries in the patients and doctors tables, respectively.

Insert Data into Departments Table

```
INSERT INTO healthcare.departments (department_name, location)
VALUES ('Cardiology', 'Building A, Floor 3');
```

Insert Data into Doctor Departments Table

```
INSERT INTO healthcare.doctor_departments (doctor_id, department_id)
VALUES (1, 1);
```

- **Explanation**: doctor_id and department_id must refer to valid entries in the doctors and departments tables, respectively.

Summary

- **Primary Keys**: Ensure each record in a table is unique and identifiable.
- **Foreign Keys**: Maintain referential integrity by ensuring that a value in one table must correspond to a valid record in another table.
- **Use Cases in Healthcare Domain**:
  - **Patients**: Uniquely identified by patient_id.
  - **Doctors**: Uniquely identified by doctor_id.
  - **Appointments**: Linked to patients and doctors using patient_id and doctor_id.
  - **Departments**: Uniquely identified by department_id.
  - **Doctor Departments**: Links doctors to departments using composite keys.

By using primary and foreign keys effectively, we can ensure the integrity and consistency of data within the healthcare database, supporting reliable and accurate patient care and management.

In PostgreSQL, enabling and disabling constraints directly is not supported as it is in some other database systems. However, you can achieve similar functionality by dropping and recreating the constraint when needed.

Here's how you can manage constraints by temporarily removing them and then adding them back:

Example: Temporarily Disabling a Constraint

Step 1: Drop the Constraint

Suppose you have a `patients` table with a `CHECK` constraint on the `age` column, and you want to temporarily disable it.

```
-- Drop the CHECK constraint
ALTER TABLE patients
DROP CONSTRAINT check_age;
```

Step 2: Perform Your Operations

Now you can perform any operations that would violate the constraint. For example, you might insert a patient with an invalid age for some specific reason.

```
-- Insert a patient with an invalid age
INSERT INTO patients (name, age, email) VALUES ('John Doe', -5,
'johndoe@example.com');
```

Example: Re-enabling the Constraint

Step 3: Recreate the Constraint

After performing the necessary operations, you can add the constraint back to the table.

```
-- Add the CHECK constraint back
ALTER TABLE patients
ADD CONSTRAINT check_age CHECK (age >= 0 AND age <= 120);
```

Here's the complete example:

```
-- Create the table with a CHECK constraint
CREATE TABLE patients (
    patient_id serial PRIMARY KEY,
    name varchar(100),
    age int,
    email varchar(100),
    CONSTRAINT check_age CHECK (age >= 0 AND age <= 120)
);

-- Insert a valid patient (for demonstration)
INSERT INTO patients (name, age, email) VALUES ('Alice', 30,
'alice@example.com');

-- Drop the CHECK constraint to disable it temporarily
ALTER TABLE patients
DROP CONSTRAINT check_age;

-- Perform operations that would violate the constraint
INSERT INTO patients (name, age, email) VALUES ('John Doe', -5,
'johndoe@example.com');

-- Add the CHECK constraint back to re-enable it
ALTER TABLE patients
ADD CONSTRAINT check_age CHECK (age >= 0 AND age <= 120);
```

Example: Temporarily Disabling a Foreign Key Constraint

Similarly, if you have a foreign key constraint in the `appointments` table referencing the `patients` table:

Step 1: Drop the Foreign Key Constraint

```
-- Drop the foreign key constraint
ALTER TABLE appointments
DROP CONSTRAINT fk_patient;
```

Step 2: Perform Your Operations

```
-- Insert an appointment with an invalid patient_id
INSERT INTO appointments (patient_id, doctor_id, appointment_date, status)
VALUES (999, 1, '2024-08-01', 'Scheduled'); -- Assuming 999 does not exist
in patients
```

Step 3: Recreate the Foreign Key Constraint

```
sql
Copy code
-- Add the foreign key constraint back
ALTER TABLE appointments
ADD CONSTRAINT fk_patient
FOREIGN KEY (patient_id) REFERENCES patients(patient_id);
```

Complete example:

```
sql
Copy code
-- Create the patients and appointments tables
CREATE TABLE patients (
    patient_id serial PRIMARY KEY,
    name varchar(100),
    age int,
    email varchar(100),
    CONSTRAINT check_age CHECK (age >= 0 AND age <= 120)
);

CREATE TABLE appointments (
    appointment_id serial PRIMARY KEY,
    patient_id int,
    doctor_id int,
    appointment_date date,
    status varchar(20),
    CONSTRAINT fk_patient FOREIGN KEY (patient_id) REFERENCES
patients(patient_id)
);

-- Insert a valid patient (for demonstration)
INSERT INTO patients (name, age, email) VALUES ('Alice', 30,
'alice@example.com');

-- Drop the foreign key constraint to disable it temporarily
ALTER TABLE appointments
DROP CONSTRAINT fk_patient;

-- Perform operations that would violate the constraint
INSERT INTO appointments (patient_id, doctor_id, appointment_date, status)
VALUES (999, 1, '2024-08-01', 'Scheduled'); -- Assuming 999 does not exist
in patients

-- Add the foreign key constraint back to re-enable it
ALTER TABLE appointments
ADD CONSTRAINT fk_patient
```

```
FOREIGN KEY (patient_id) REFERENCES patients(patient_id);
```

In PostgreSQL, this approach allows you to manage constraints flexibly, though you need to be careful to re-enable constraints as needed to maintain data integrity.

Data Types and Constraints in PostgreSQL

Data Types

PostgreSQL supports various data types for defining the nature of data stored in tables. Here are some common data types:

1. **Numeric Types:**
   - INTEGER or INT: Whole numbers.
   - SERIAL: Auto-incrementing integer (often used for primary keys).
   - NUMERIC(precision, scale): Exact numeric values with a specified precision and scale.
   - FLOAT, REAL, DOUBLE PRECISION: Floating-point numbers.

2. **Character Types:**
   - VARCHAR(n): Variable-length character string with a maximum length of n.
   - CHAR(n): Fixed-length character string.
   - TEXT: Variable-length character string with no specific length limit.

3. **Date/Time Types:**
   - DATE: Calendar date (year, month, day).
   - TIME [WITHOUT TIME ZONE]: Time of day (hours, minutes, seconds).
   - TIMESTAMP [WITHOUT TIME ZONE]: Date and time.
   - INTERVAL: Time interval.

4. **Boolean Type:**
   - BOOLEAN: Logical Boolean (TRUE or FALSE).

5. **Binary Data Types:**
   - BYTEA: Binary data ("byte array").
6. **Array Types:**
   - PostgreSQL supports arrays of any data type. For example, INTEGER[] or TEXT[].
7. **JSON Types:**
   - JSON: Stores JSON (JavaScript Object Notation) data.
   - JSONB: Stores JSON data in a decomposed binary format.

Constraints

Constraints are rules applied to table columns to enforce data integrity and ensure the accuracy and reliability of the data.

1. **NOT NULL Constraint:**
   - Ensures that a column cannot have a NULL value.

```
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL
);
```

2. **UNIQUE Constraint:**
   - Ensures that all values in a column are unique.

```
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    email VARCHAR(100) UNIQUE
);
```

3. **PRIMARY KEY Constraint:**
   - A combination of NOT NULL and UNIQUE. Uniquely identifies each row in a table.

```
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL
);
```

4. **FOREIGN KEY Constraint:**
   - Ensures the referential integrity of the data in one table to match values in another table.

```
CREATE TABLE departments (
    department_id SERIAL PRIMARY KEY,
    department_name VARCHAR(50) NOT NULL
);

CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    department_id INTEGER REFERENCES departments(department_id)
);
```

5. **CHECK Constraint:**
   - Ensures that all values in a column satisfy a specific condition.

```
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    salary NUMERIC(8, 2) CHECK (salary > 0)
);
```

6. **DEFAULT Constraint:**
   - Sets a default value for a column when no value is specified.

```
CREATE TABLE employees (
   employee_id SERIAL PRIMARY KEY,
   hire_date DATE DEFAULT CURRENT_DATE
);
```

Primary Keys and Foreign Keys

Primary Key

A primary key is a column (or a combination of columns) that uniquely identifies each row in a table. Each table can have only one primary key, which can consist of single or multiple columns.

- **Single Column Primary Key:**

```
CREATE TABLE employees (
   employee_id SERIAL PRIMARY KEY,
   first_name VARCHAR(50),
   last_name VARCHAR(50)
);
```

- **Composite Primary Key:**

```
CREATE TABLE project_assignments (
   employee_id INTEGER,
   project_id INTEGER,
   PRIMARY KEY (employee_id, project_id)
);
```

Foreign Key

A foreign key is a column (or a combination of columns) that establishes a link between data in two tables. It ensures that the value in the foreign key column(s) must match values in the referenced primary key column(s) of another table.

- **Foreign Key Example:**

```
CREATE TABLE departments (
   department_id SERIAL PRIMARY KEY,
   department_name VARCHAR(50)
);

CREATE TABLE employees (
   employee_id SERIAL PRIMARY KEY,
   first_name VARCHAR(50),
   last_name VARCHAR(50),
   department_id INTEGER,
   FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

- **Foreign Key with ON DELETE CASCADE:**
  - Ensures that when a row in the referenced table is deleted, all related rows in the referencing table are also deleted.

```
CREATE TABLE employees (
   employee_id SERIAL PRIMARY KEY,
   first_name VARCHAR(50),
   last_name VARCHAR(50),
   department_id INTEGER,
   FOREIGN KEY (department_id) REFERENCES departments(department_id) ON DELETE CASCADE
);
```

Summary

Understanding data types and constraints is essential for designing a robust and reliable database schema. Constraints like primary keys and foreign keys enforce data integrity and relationships between tables, ensuring that your database accurately represents your data model and maintains consistency.

Altering column constraints in PostgreSQL is a common task when you need to update your database schema. Here are examples of how to alter column constraints for a healthcare domain.

Example 1: Altering a Column to Add/Remove NOT NULL Constraint

Add NOT NULL Constraint:

Suppose you have a `patients` table, and you want to ensure that the `email` column cannot have null values.

```
-- Before alteration
CREATE TABLE patients (
    patient_id serial PRIMARY KEY,
    name varchar(100),
    email varchar(100) -- Currently, this column allows NULL values
);

-- Add NOT NULL constraint
ALTER TABLE patients
ALTER COLUMN email SET NOT NULL;
```

Remove NOT NULL Constraint:

If you later decide that the `email` column should allow null values, you can remove the NOT NULL constraint.

```
-- Remove NOT NULL constraint
ALTER TABLE patients
ALTER COLUMN email DROP NOT NULL;
```

Example 2: Adding a UNIQUE Constraint

Suppose you want to ensure that the `email` column in the `patients` table has unique values.

```
-- Add UNIQUE constraint
ALTER TABLE patients
ADD CONSTRAINT unique_email UNIQUE (email);
```

Example 3: Adding/Removing a CHECK Constraint

Add CHECK Constraint:

Suppose you want to add a constraint to the `age` column in the `patients` table to ensure that age is between 0 and 120.

```
-- Add CHECK constraint
ALTER TABLE patients
ADD CONSTRAINT check_age CHECK (age >= 0 AND age <= 120);
```

Remove CHECK Constraint:

If you need to remove the CHECK constraint:

```
-- Remove CHECK constraint
ALTER TABLE patients
DROP CONSTRAINT check_age;
```

Example 4: Modifying a Default Value

Suppose you want to set a default value for the `status` column in the `appointments` table.

```
-- Before alteration
CREATE TABLE appointments (
    appointment_id serial PRIMARY KEY,
    patient_id int REFERENCES patients(patient_id),
    appointment_date date,
    status varchar(20) -- No default value
);

-- Add default value
ALTER TABLE appointments
ALTER COLUMN status SET DEFAULT 'Scheduled';
```

If you later want to change the default value:

```
-- Change default value
ALTER TABLE appointments
ALTER COLUMN status SET DEFAULT 'Pending';
```

Or remove the default value:

```
-- Remove default value
ALTER TABLE appointments
ALTER COLUMN status DROP DEFAULT;
```

Example 5: Adding a Foreign Key Constraint

Suppose you have a `doctors` table, and you want to add a foreign key constraint to the `appointments` table to link it to the `doctors` table.

```
-- Before alteration
CREATE TABLE doctors (
    doctor_id serial PRIMARY KEY,
    name varchar(100)
);

CREATE TABLE appointments (
    appointment_id serial PRIMARY KEY,
    patient_id int REFERENCES patients(patient_id),
    doctor_id int, -- No foreign key constraint yet
    appointment_date date,
    status varchar(20) DEFAULT 'Scheduled'
);

-- Add foreign key constraint
ALTER TABLE appointments
ADD CONSTRAINT fk_doctor
FOREIGN KEY (doctor_id) REFERENCES doctors(doctor_id);
```

Example 6: Removing a Foreign Key Constraint

If you need to remove the foreign key constraint:

```
-- Remove foreign key constraint
ALTER TABLE appointments
DROP CONSTRAINT fk_doctor;
```

Example 7: Renaming a Constraint

If you need to rename a constraint for clarity:

```
-- Rename a constraint (e.g., the unique constraint on email)

ALTER TABLE patients
RENAME CONSTRAINT unique_email TO unique_patient_email;
```

These examples demonstrate how to alter column constraints in PostgreSQL to ensure data integrity and meet the evolving requirements of a healthcare database.

Hour 3-4: Basic SQL Commands

CREATE, INSERT, SELECT, UPDATE, DELETE postgres examples for Healthcare domain

PostgreSQL CRUD Operations: Healthcare Domain Examples

Here are examples of the basic SQL commands (CREATE, INSERT, SELECT, UPDATE, DELETE) within the context of a healthcare domain.

1. CREATE

Creating Tables

Patients Table

```
CREATE TABLE healthcare.patients (
    patient_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    date_of_birth DATE NOT NULL,
    gender VARCHAR(10) CHECK (gender IN ('Male', 'Female', 'Other')),
    contact_number VARCHAR(15) UNIQUE,
    email VARCHAR(50),
    address TEXT,
    registration_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Doctors Table

```
CREATE TABLE healthcare.doctors (
    doctor_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    specialty VARCHAR(50) NOT NULL,
    contact_number VARCHAR(15) UNIQUE,
    email VARCHAR(50) UNIQUE
);
```

Appointments Table

```
CREATE TABLE healthcare.appointments (
    appointment_id SERIAL PRIMARY KEY,
    patient_id INT NOT NULL REFERENCES healthcare.patients(patient_id) ON DELETE CASCADE,
    doctor_id INT NOT NULL REFERENCES healthcare.doctors(doctor_id),
    appointment_date TIMESTAMP NOT NULL,
    status VARCHAR(20) CHECK (status IN ('Scheduled', 'Completed', 'Cancelled'))
);
```

Departments Table

```
CREATE TABLE healthcare.departments (
```

```
    department_id SERIAL PRIMARY KEY,
    department_name VARCHAR(50) NOT NULL UNIQUE,
    location VARCHAR(100) NOT NULL
);
```

2. INSERT

Inserting Data into Tables

Insert Data into Patients Table

```
INSERT INTO healthcare.patients (first_name, last_name, date_of_birth, gender, contact_number,
email, address)
VALUES
('Dev', 'Daga', '1980-01-01', 'Male', '123-456-7890', 'john.doe@example.com', '123 Main St, Anytown,
USA'),
('Rani', 'Daga', '1990-02-02', 'Female', '234-567-8901', 'jane.roe@example.com', '456 Maple Ave,
Anytown, USA');
```

Insert Data into Doctors Table

```
INSERT INTO healthcare.doctors (first_name, last_name, specialty, contact_number, email)
VALUES
('Alice', 'Smith', 'Cardiology', '345-678-9012', 'alice.smith@example.com'),
('Bob', 'Johnson', 'Neurology', '456-789-0123', 'bob.johnson@example.com');
```

Insert Data into Appointments Table

```
INSERT INTO healthcare.appointments (patient_id, doctor_id, appointment_date, status)
VALUES
(1, 1, '2024-07-20 10:00:00', 'Scheduled'),
(2, 2, '2024-07-21 11:00:00', 'Completed');
```

Insert Data into Departments Table

```
INSERT INTO healthcare.departments (department_name, location)
VALUES
('Cardiology', 'Building A, Floor 3'),
('Neurology', 'Building B, Floor 2');
```

3. SELECT

Selecting Data from Tables

Select All Patients

```
SELECT * FROM healthcare.patients;
```

Select Specific Columns from Doctors Table

```
SELECT first_name, last_name, specialty FROM healthcare.doctors;
```

Join Patients and Appointments

```
SELECT p.first_name, p.last_name, a.appointment_date, a.status
FROM healthcare.patients p
JOIN healthcare.appointments a ON p.patient_id = a.patient_id;
```

Count Appointments by Doctor

```
SELECT d.first_name, d.last_name, COUNT(a.appointment_id) AS appointment_count
FROM healthcare.doctors d
JOIN healthcare.appointments a ON d.doctor_id = a.doctor_id
GROUP BY d.first_name, d.last_name;
```

4. UPDATE

Updating Data in Tables

Update Patient's Contact Number

```
UPDATE healthcare.patients
SET contact_number = '987-654-3210'
WHERE patient_id = 1;
```

Update Doctor's Specialty

```
UPDATE healthcare.doctors
SET specialty = 'Pediatrics'
WHERE doctor_id = 2;
```

Reschedule an Appointment

```
UPDATE healthcare.appointments
SET appointment_date = '2024-07-22 14:00:00', status = 'Rescheduled'
WHERE appointment_id = 1;
```

5. DELETE

Deleting Data from Tables

Delete a Patient Record

```
DELETE FROM healthcare.patients
WHERE patient_id = 2;
```

Delete a Doctor Record

```
DELETE FROM healthcare.doctors
WHERE doctor_id = 2;
```

Delete an Appointment

```
DELETE FROM healthcare.appointments
WHERE appointment_id = 1;
```

Summary

These examples demonstrate how to perform basic CRUD (Create, Read, Update, Delete) operations in PostgreSQL within a healthcare domain. By using these commands, you can efficiently manage and manipulate the data in your healthcare database, ensuring that information is accurately recorded, retrieved, and maintained.

Filtering and sorting data postgres examples for Healthcare domain

Filtering and sorting data are essential operations in PostgreSQL that allow you to retrieve specific subsets of data and arrange them in a desired order. Here are examples of filtering and sorting data within the context of a healthcare domain:

Filtering Data Examples

1. Filtering Patients by Gender and Age

Filter Patients by Gender

```
SELECT *
FROM healthcare.patients
WHERE gender = 'Male';
```

Filter Patients by Age Range

Assuming date_of_birth is stored in the patients table:

```
SELECT *
FROM healthcare.patients
WHERE date_of_birth <= '2000-01-01';
```

2. Filtering Doctors by Specialty

Filter Doctors by Specialty (Cardiology)

```
SELECT *
FROM healthcare.doctors
WHERE specialty = 'Cardiology';
```

3. Filtering Appointments by Status

Filter Appointments by Status (Scheduled)

```
SELECT *
FROM healthcare.appointments
WHERE status = 'Scheduled';
```

Sorting Data Examples

1. Sorting Patients by Last Name

Sort Patients by Last Name in Ascending Order

```
SELECT *
FROM healthcare.patients
ORDER BY last_name ASC;
```

Sort Patients by Last Name in Descending Order

```
SELECT *
FROM healthcare.patients
ORDER BY last_name DESC;
```

2. Sorting Doctors by Specialty and Last Name

Sort Doctors by Specialty and Last Name

```
SELECT *
FROM healthcare.doctors
ORDER BY specialty ASC, last_name ASC;
```

3. Sorting Appointments by Appointment Date

Sort Appointments by Appointment Date

```
SELECT *
FROM healthcare.appointments
ORDER BY appointment_date ASC;
```

Combined Filtering and Sorting Examples

1. Filter Patients by Gender and Sort by Registration Date

Filter Male Patients and Sort by Registration Date

```
SELECT *
FROM healthcare.patients
WHERE gender = 'Male'
ORDER BY registration_date DESC;
```

2. Filter Appointments by Doctor and Sort by Appointment Date

Filter Appointments for a Specific Doctor and Sort by Appointment Date

```
SELECT *
FROM healthcare.appointments a
JOIN healthcare.doctors d ON a.doctor_id = d.doctor_id
WHERE d.last_name = 'Smith'
ORDER BY a.appointment_date ASC;
```

Summary

Filtering and sorting data in PostgreSQL allows you to retrieve specific subsets of data based on criteria and arrange them in a desired order. These operations are crucial in querying healthcare databases to extract relevant information efficiently, whether for patient management, scheduling appointments, or analyzing medical records. Adjust the SQL queries according to your specific database schema and business requirements to effectively manage healthcare data.

Aggregate functions postgres examples for Healthcare domain

Aggregate functions in PostgreSQL allow you to perform calculations across sets of rows to produce summarized results. Here are examples of commonly used aggregate functions within the context of a healthcare domain:

Aggregate Functions Examples

1. Counting Patients

Count the Number of Patients

```
SELECT COUNT(*) AS total_patients
FROM healthcare.patients;
```

2. Calculating Average Age of Patients

Assuming date_of_birth is stored in the patients table:

Calculate Average Age of Patients

```
SELECT AVG(EXTRACT(YEAR FROM age(CURRENT_DATE, date_of_birth))) AS average_age
FROM healthcare.patients;
```

3. Finding the Oldest and Youngest Patients

Find the Oldest Patient

```
SELECT first_name, last_name, date_of_birth
FROM healthcare.patients
ORDER BY date_of_birth ASC
LIMIT 1;
```

Find the Youngest Patient

```
SELECT first_name, last_name, date_of_birth
FROM healthcare.patients
ORDER BY date_of_birth DESC
LIMIT 1;
```

4. Calculating Total Appointments per Doctor

Count Total Appointments per Doctor

```
SELECT d.first_name, d.last_name, COUNT(a.appointment_id) AS total_appointments
FROM healthcare.doctors d
LEFT JOIN healthcare.appointments a ON d.doctor_id = a.doctor_id
GROUP BY d.first_name, d.last_name
ORDER BY total_appointments DESC;
```

5. Summing Up Patients by Gender

Sum the Number of Patients by Gender

```
SELECT gender, COUNT(*) AS total_patients
FROM healthcare.patients
GROUP BY gender;
```

Summary

Aggregate functions in PostgreSQL allow for powerful data analysis and summarization within healthcare databases. Whether you need to count records, calculate averages, find extremum values, or summarize data based on specific criteria, aggregate functions provide essential tools for querying and analyzing healthcare data effectively. Adjust the SQL queries according to your specific database schema and business requirements to derive meaningful insights from your healthcare database.

Basic SQL Commands with Examples

SELECT

The SELECT statement is used to retrieve data from one or more tables.

Select All Columns from a Table:

```
SELECT * FROM employees;
```

Explanation: Retrieves all columns and all rows from the employees table.

Select Specific Columns from a Table:

```
SELECT first_name, last_name, email FROM employees;
```

Explanation: Retrieves only the first_name, last_name, and email columns from the employees table.

Select with a WHERE Clause:

```
SELECT * FROM employees WHERE department_id = 1;
```

Explanation: Retrieves all columns and rows from the employees table where the department_id is 1.

Select with an ORDER BY Clause:

```
SELECT * FROM employees ORDER BY last_name ASC;
```

Explanation: Retrieves all columns and rows from the employees table and orders the results by last_name in ascending order.

Select with a JOIN:

```
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id;
```

Explanation: Retrieves the first_name and last_name from the employees table and the department_name from the departments table where the department_id matches in both tables.

INSERT

The INSERT statement is used to add new rows to a table.

Insert a Single Row:

```
INSERT INTO employees (first_name, last_name, email, hire_date, salary, department_id)
VALUES ('Jane', 'Smith', 'jane.smith@example.com', '2024-01-01', 60000.00, 1);
```

Explanation: Inserts a new row into the employees table with the specified values for each column.

Insert Multiple Rows:

```
INSERT INTO employees (first_name, last_name, email, hire_date, salary, department_id)
VALUES
('John', 'Doe', 'john.doe@example.com', '2023-07-01', 50000.00, 1),
('Alice', 'Johnson', 'alice.johnson@example.com', '2023-08-15', 55000.00, 2);
```

Explanation: Inserts two new rows into the employees table with the specified values for each column.

UPDATE

The UPDATE statement is used to modify existing rows in a table.

Update a Single Row:

```
UPDATE employees
SET salary = 70000.00
WHERE employee_id = 1;
```

Explanation: Updates the salary column to 70000.00 for the row where employee_id is 1.

Update Multiple Rows:

```
UPDATE employees
SET department_id = 2
WHERE department_id = 1;
```

Explanation: Updates the department_id column to 2 for all rows where the department_id is 1.

DELETE

The DELETE statement is used to remove rows from a table.

Delete a Single Row:

```
DELETE FROM employees
WHERE employee_id = 1;
```

Explanation: Deletes the row from the employees table where the employee_id is 1.

Delete Multiple Rows:

```
DELETE FROM employees
WHERE department_id = 1;
```

Explanation: Deletes all rows from the employees table where the department_id is 1.

Delete All Rows:

```
DELETE FROM employees;
```

Explanation: Deletes all rows from the employees table. Note: The table structure remains intact.

Summary

These basic SQL commands are fundamental for interacting with any relational database, including PostgreSQL. They allow you to retrieve, add, modify, and delete data in a structured and efficient manner. Understanding these commands is crucial for database management and development tasks.

Examples for Filtering and Sorting Data in PostgreSQL

Filtering Data

Basic WHERE Clause:

```
SELECT * FROM employees
WHERE department_id = 1;
```

Explanation: Retrieves all columns and rows from the employees table where the department_id is 1.

Multiple Conditions with AND:

```
SELECT * FROM employees
WHERE department_id = 1 AND salary > 50000;
```

Explanation: Retrieves rows from the employees table where the department_id is 1 and the salary is greater than 50000.

Using OR in WHERE Clause:

```
SELECT * FROM employees
WHERE department_id = 1 OR department_id = 2;
```

Explanation: Retrieves rows from the employees table where the department_id is either 1 or 2.

Using IN Operator:

```
SELECT * FROM employees
WHERE department_id IN (1, 2);
```

Explanation: Retrieves rows from the employees table where the department_id is 1 or 2.

Using LIKE Operator for Pattern Matching:

```
SELECT * FROM employees
WHERE first_name LIKE 'J%';
```

Explanation: Retrieves rows from the employees table where the first_name starts with 'J'.

Sorting Data

Ordering Results with ORDER BY:

```
SELECT * FROM employees
ORDER BY last_name ASC;
```

Explanation: Retrieves all columns and rows from the employees table and sorts the results by last_name in ascending order (ASC).

Ordering Results with DESC (Descending Order):

```
SELECT * FROM employees
ORDER BY salary DESC;
```

Explanation: Retrieves all columns and rows from the employees table and sorts the results by salary in descending order (DESC).

Aggregate Functions

Aggregate functions perform calculations on multiple rows to return a single value.

COUNT:

```
SELECT COUNT(*) FROM employees;
```

Explanation: Returns the number of rows in the employees table.

SUM:

```
SELECT SUM(salary) FROM employees;
```

Explanation: Returns the sum of all salary values in the employees table.

AVG:

```
SELECT AVG(salary) FROM employees;
```

Explanation: Returns the average salary value in the employees table.

MIN:

```
SELECT MIN(salary) FROM employees;
```

Explanation: Returns the minimum salary value in the employees table.

MAX:

```
SELECT MAX(salary) FROM employees;
```

Explanation: Returns the maximum salary value in the employees table.

Summary

Filtering and sorting data are essential capabilities in SQL for retrieving specific subsets of data and arranging them in a desired order. Aggregate functions provide powerful tools for calculating summary statistics or performing calculations across multiple rows of data. These examples demonstrate common use cases for filtering, sorting, and using aggregate functions in PostgreSQL queries. Understanding these concepts is crucial for effective data manipulation and analysis in database operations

In PostgreSQL, you cannot directly enable or disable indexes like you might with constraints. However, you can achieve similar functionality by dropping the index when you want to "disable" it and recreating it when you want to "enable" it again. Here's how you can manage indexes for a table.

Example: Temporarily Disabling an Index

Step 1: Create a Table and Index

First, create a sample table and an index on one of its columns.

```
-- Create the table
CREATE TABLE patients (
    patient_id serial PRIMARY KEY,
    name varchar(100),
    age int,
    email varchar(100)
);

-- Create an index on the email column
CREATE INDEX idx_email ON patients(email);
```

Step 2: Drop the Index to Disable It

Dropping the index will effectively "disable" it.

```
-- Drop the index
DROP INDEX idx_email;
```

Step 3: Perform Operations

You can now perform any operations without the index. For example, inserting a large number of records or performing bulk updates.

```
-- Insert a large number of records (for demonstration)
INSERT INTO patients (name, age, email)
SELECT 'Patient ' || i, (random() * 100)::int, 'email' || i ||
'@example.com'
FROM generate_series(1, 100000) AS s(i);
```

Example: Re-enabling the Index

Step 4: Recreate the Index

After performing the necessary operations, you can recreate the index to "enable" it again.

```
-- Recreate the index
CREATE INDEX idx_email ON patients(email);
```

Here’s the complete example:

```
sql
Copy code
-- Step 1: Create the table and index
CREATE TABLE patients (
    patient_id serial PRIMARY KEY,
    name varchar(100),
    age int,
    email varchar(100)
);

CREATE INDEX idx_email ON patients(email);

-- Step 2: Drop the index to disable it
DROP INDEX idx_email;

-- Step 3: Perform operations without the index
INSERT INTO patients (name, age, email)
SELECT 'Patient ' || i, (random() * 100)::int, 'email' || i ||
'@example.com'
FROM generate_series(1, 100000) AS s(i);

-- Step 4: Recreate the index to enable it
CREATE INDEX idx_email ON patients(email);
```

Example: Temporarily Disabling and Re-enabling a Unique Index

Unique indexes are often used to enforce uniqueness constraints. Here’s how you can manage them.

Step 1: Create a Table and Unique Index

```
sql
Copy code
-- Create the table
CREATE TABLE doctors (
    doctor_id serial PRIMARY KEY,
    name varchar(100),
    email varchar(100)
);
```

```
-- Create a unique index on the email column
CREATE UNIQUE INDEX idx_unique_email ON doctors(email);
```

Step 2: Drop the Unique Index to Disable It

```
sql
Copy code
-- Drop the unique index
DROP INDEX idx_unique_email;
```

Step 3: Perform Operations

```
sql
Copy code
-- Insert records that might violate the unique constraint
INSERT INTO doctors (name, email) VALUES ('Dr. Smith',
'dr.smith@example.com');
INSERT INTO doctors (name, email) VALUES ('Dr. John',
'dr.smith@example.com');
```

Example: Re-enabling the Unique Index

Step 4: Recreate the Unique Index

```
sql
Copy code
-- Recreate the unique index
CREATE UNIQUE INDEX idx_unique_email ON doctors(email);
```

Complete example:

```
sql
Copy code
-- Step 1: Create the table and unique index
CREATE TABLE doctors (
    doctor_id serial PRIMARY KEY,
    name varchar(100),
    email varchar(100)
);

CREATE UNIQUE INDEX idx_unique_email ON doctors(email);

-- Step 2: Drop the unique index to disable it
DROP INDEX idx_unique_email;

-- Step 3: Perform operations without the unique index
INSERT INTO doctors (name, email) VALUES ('Dr. Smith',
'dr.smith@example.com');
INSERT INTO doctors (name, email) VALUES ('Dr. John',
'dr.smith@example.com');

-- Step 4: Recreate the unique index to enable it
CREATE UNIQUE INDEX idx_unique_email ON doctors(email);
```

Conclusion

While PostgreSQL does not have a built-in feature to enable or disable indexes directly, dropping and recreating indexes provides similar functionality. This approach is useful for bulk operations where indexes might degrade performance. Remember to recreate the indexes afterward to maintain query performance and constraints.

Sequences in PostgreSQL are used to generate unique identifiers, often for primary keys. Here are some examples of how to create, alter, and use sequences in PostgreSQL.

Example 1: Creating a Simple Sequence

Create a sequence named `patient_id_seq`.

```
sql
Copy code
CREATE SEQUENCE patient_id_seq
START WITH 1
INCREMENT BY 1
NO MINVALUE
NO MAXVALUE
CACHE 1;
```

This sequence starts at 1 and increments by 1 for each new value.

Example 2: Using a Sequence in a Table

Use the sequence in the `patients` table for the `patient_id` column.

```
sql
Copy code
CREATE TABLE patients (
    patient_id int DEFAULT nextval('patient_id_seq') PRIMARY KEY,
    name varchar(100),
    age int,
    email varchar(100)
);
```

Inserting a new row will automatically use the sequence to generate the `patient_id`.

```
sql
Copy code
INSERT INTO patients (name, age, email) VALUES ('Alice', 30,
'alice@example.com');
```

Example 3: Creating a Sequence with Additional Options

Create a sequence with custom options, such as minimum and maximum values, and cycle behavior.

```
sql
Copy code
CREATE SEQUENCE custom_seq
START WITH 1000
INCREMENT BY 5
MINVALUE 1000
MAXVALUE 2000
CYCLE
CACHE 10;
```

This sequence starts at 1000, increments by 5, and will restart from the minimum value when it reaches the maximum value due to the `CYCLE` option.

Example 4: Altering an Existing Sequence

Modify the parameters of an existing sequence.

```
sql
Copy code
ALTER SEQUENCE patient_id_seq
RESTART WITH 500
INCREMENT BY 2;
```

This changes the `patient_id_seq` to restart from 500 and increment by 2.

Example 5: Using a Sequence to Generate Values Manually

Retrieve the next value from a sequence manually.

```
sql
Copy code
SELECT nextval('patient_id_seq');
```

Set the current value of a sequence.

```
sql
Copy code
SELECT setval('patient_id_seq', 100);
```

Example 6: Using a Sequence in Multiple Tables

Create a sequence and use it in multiple tables.

```
sql
Copy code
CREATE SEQUENCE global_id_seq
START WITH 1
INCREMENT BY 1;

CREATE TABLE doctors (
    doctor_id int DEFAULT nextval('global_id_seq') PRIMARY KEY,
    name varchar(100),
    email varchar(100)
);

CREATE TABLE appointments (
    appointment_id int DEFAULT nextval('global_id_seq') PRIMARY KEY,
    patient_id int,
    doctor_id int,
    appointment_date date,
    status varchar(20)
);
```

Complete Example: Managing Sequences

Here's a complete example to demonstrate the lifecycle of sequences:

```
-- Step 1: Create a sequence


CREATE SEQUENCE patient_id_seq
START WITH 1
INCREMENT BY 1
NO MINVALUE
NO MAXVALUE
CACHE 1;

-- Step 2: Create a table using the sequence
CREATE TABLE patients (
    patient_id int DEFAULT nextval('patient_id_seq') PRIMARY KEY,
    name varchar(100),
    age int,
    email varchar(100)
);

-- Step 3: Insert rows
INSERT INTO patients (name, age, email) VALUES ('Alice', 30,
'alice@example.com');
INSERT INTO patients (name, age, email) VALUES ('Bob', 40,
'bob@example.com');

-- Step 4: Alter the sequence
ALTER SEQUENCE patient_id_seq
RESTART WITH 500
INCREMENT BY 2;

-- Step 5: Insert more rows
INSERT INTO patients (name, age, email) VALUES ('Carol', 35,
'carol@example.com');
INSERT INTO patients (name, age, email) VALUES ('Dave', 45,
'dave@example.com');

-- Step 6: Manually retrieve the next value
```

```
SELECT nextval('patient_id_seq');

-- Step 7: Manually set the sequence value
SELECT setval('patient_id_seq', 1000);

-- Step 8: Insert a row with the updated sequence value
INSERT INTO patients (name, age, email) VALUES ('Eve', 50,
'eve@example.com');
```

These examples demonstrate how to create, use, and manage sequences in PostgreSQL to generate unique identifiers and manage auto-incrementing values.

Renaming a schema in PostgreSQL is straightforward using the `ALTER SCHEMA` command. Here is the syntax and an example to demonstrate how to rename a schema:

Syntax

```
sql
Copy code
ALTER SCHEMA old_schema_name RENAME TO new_schema_name;
```

Example

Suppose you have a schema named `healthcare` that you want to rename to `medical`.

```
sql
Copy code
-- Rename the schema from 'healthcare' to 'medical'
ALTER SCHEMA healthcare RENAME TO medical;
```

Complete Example

Let's see a more comprehensive example where we create a schema, add a table to it, and then rename the schema.

Step 1: Create the Schema

```
sql
Copy code
CREATE SCHEMA healthcare;
```

Step 2: Create a Table in the Schema

```
sql
Copy code
CREATE TABLE healthcare.patients (
    patient_id serial PRIMARY KEY,
    name varchar(100),
    age int,
    email varchar(100)
);
```

Step 3: Insert Data into the Table

```
sql
Copy code
INSERT INTO healthcare.patients (name, age, email) VALUES ('Alice', 30,
'alice@example.com');
```

Step 4: Rename the Schema

```
sql
Copy code
ALTER SCHEMA healthcare RENAME TO medical;
```

Step 5: Query the Table in the Renamed Schema

```
sql
Copy code
SELECT * FROM medical.patients;
```

Complete SQL Script

Here's the complete SQL script for creating a schema, adding a table, inserting data, renaming the schema, and querying the table:

```
sql
Copy code
-- Step 1: Create the schema
CREATE SCHEMA healthcare;

-- Step 2: Create a table in the schema
CREATE TABLE healthcare.patients (
    patient_id serial PRIMARY KEY,
    name varchar(100),
    age int,
    email varchar(100)
);

-- Step 3: Insert data into the table

INSERT INTO healthcare.patients (name, age, email) VALUES ('Alice', 30,
'alice@example.com');

-- Step 4: Rename the schema
ALTER SCHEMA healthcare RENAME TO medical;

-- Step 5: Query the table in the renamed schema
SELECT * FROM medical.patients;
```

By following these steps, you can easily rename a schema in PostgreSQL while ensuring that all objects within the schema remain accessible under the new schema name.

`EXTRACT` function

The `EXTRACT` function in PostgreSQL is used to retrieve subfields such as year, month, day, etc., from date/time values. Here are some examples to demonstrate how to use the `EXTRACT` function in various scenarios.

Example 1: Extracting Year, Month, and Day from a Date

Assume you have a table `appointments` with a column `appointment_date` of type `date`.

```
CREATE TABLE appointments (
    appointment_id serial PRIMARY KEY,
    patient_id int,
    doctor_id int,
    appointment_date date,
    status varchar(20)
);

-- Insert some sample data
INSERT INTO appointments (patient_id, doctor_id, appointment_date, status)
VALUES
(1, 1, '2023-07-15', 'Scheduled'),
(2, 2, '2024-01-20', 'Completed');
```

Extracting year, month, and day from the `appointment_date`:

```
SELECT
    appointment_id,
    EXTRACT(YEAR FROM appointment_date) AS year,
    EXTRACT(MONTH FROM appointment_date) AS month,
    EXTRACT(DAY FROM appointment_date) AS day
FROM appointments;
```

Example 2: Extracting Hour, Minute, and Second from a Timestamp

Assume you have a table `events` with a column `event_timestamp` of type `timestamp`.

```
CREATE TABLE events (
    event_id serial PRIMARY KEY,
    event_name varchar(100),
    event_timestamp timestamp
);

-- Insert some sample data
INSERT INTO events (event_name, event_timestamp)
VALUES
('Event A', '2023-07-15 14:35:20'),
('Event B', '2024-01-20 09:15:45');
```

Extracting hour, minute, and second from the `event_timestamp`:

```
SELECT
    event_id,
    EXTRACT(HOUR FROM event_timestamp) AS hour,
    EXTRACT(MINUTE FROM event_timestamp) AS minute,
    EXTRACT(SECOND FROM event_timestamp) AS second
FROM events;
```

Example 3: Extracting Day of the Week and Day of the Year

You can extract the day of the week and the day of the year from a date or timestamp.

```
-- Extracting day of the week and day of the year from the appointment_date
SELECT
    appointment_id,
    EXTRACT(DOW FROM appointment_date) AS day_of_week, -- Sunday = 0,
Saturday = 6
    EXTRACT(DOY FROM appointment_date) AS day_of_year
FROM appointments;
```

Example 4: Extracting Quarter and Week from a Date

```
-- Extracting quarter and week from the appointment_date
SELECT
    appointment_id,
    EXTRACT(QUARTER FROM appointment_date) AS quarter,
    EXTRACT(WEEK FROM appointment_date) AS week
FROM appointments;
```

Example 5: Extracting Epoch Time

You can extract the epoch time (number of seconds since 1970-01-01 00:00:00 UTC) from a timestamp.

```
-- Extracting epoch time from the event_timestamp
SELECT
    event_id,
    EXTRACT(EPOCH FROM event_timestamp) AS epoch_time
FROM events;
```

Example 6: Using EXTRACT in WHERE Clause

You can use the `EXTRACT` function in the `WHERE` clause to filter results.

```
-- Find all appointments scheduled in July
SELECT
    appointment_id,
    patient_id,
    doctor_id,
    appointment_date,
    status
FROM appointments
WHERE EXTRACT(MONTH FROM appointment_date) = 7;
```

```
-- Find all events that occurred in the afternoon (12 PM to 5 PM)

SELECT
    event_id,
    event_name,
    event_timestamp
FROM events
WHERE EXTRACT(HOUR FROM event_timestamp) BETWEEN 12 AND 17;
```

Complete Example

Here is a complete example with the creation of tables, insertion of sample data, and various uses of the `EXTRACT` function:

```
-- Create tables
CREATE TABLE appointments (
    appointment_id serial PRIMARY KEY,
    patient_id int,
    doctor_id int,
    appointment_date date,
    status varchar(20)
);

CREATE TABLE events (
    event_id serial PRIMARY KEY,
    event_name varchar(100),
    event_timestamp timestamp
);

-- Insert sample data
INSERT INTO appointments (patient_id, doctor_id, appointment_date, status)
VALUES
(1, 1, '2023-07-15', 'Scheduled'),
(2, 2, '2024-01-20', 'Completed');

INSERT INTO events (event_name, event_timestamp)
VALUES
('Event A', '2023-07-15 14:35:20'),
('Event B', '2024-01-20 09:15:45');

-- Extract year, month, and day from appointment_date
SELECT
    appointment_id,
    EXTRACT(YEAR FROM appointment_date) AS year,
    EXTRACT(MONTH FROM appointment_date) AS month,
    EXTRACT(DAY FROM appointment_date) AS day
FROM appointments;

-- Extract hour, minute, and second from event_timestamp
SELECT
    event_id,
    EXTRACT(HOUR FROM event_timestamp) AS hour,
    EXTRACT(MINUTE FROM event_timestamp) AS minute,
    EXTRACT(SECOND FROM event_timestamp) AS second
FROM events;

-- Extract day of the week and day of the year from appointment_date
SELECT
    appointment_id,
```

```
    EXTRACT(DOW FROM appointment_date) AS day_of_week, -- Sunday = 0,
Saturday = 6
    EXTRACT(DOY FROM appointment_date) AS day_of_year
FROM appointments;

-- Extract quarter and week from appointment_date
SELECT
    appointment_id,
    EXTRACT(QUARTER FROM appointment_date) AS quarter,
    EXTRACT(WEEK FROM appointment_date) AS week
FROM appointments;

-- Extract epoch time from event_timestamp
SELECT
    event_id,
    EXTRACT(EPOCH FROM event_timestamp) AS epoch_time
FROM events;

-- Find all appointments scheduled in July
SELECT
    appointment_id,
    patient_id,
    doctor_id,
    appointment_date,
    status
FROM appointments
WHERE EXTRACT(MONTH FROM appointment_date) = 7;

-- Find all events that occurred in the afternoon (12 PM to 5 PM)
SELECT
    event_id,
    event_name,
    event_timestamp
FROM events
WHERE EXTRACT(HOUR FROM event_timestamp) BETWEEN 12 AND 17;
```

These examples show how to use the `EXTRACT` function in various contexts to retrieve specific parts of date and time values in PostgreSQL.