Learning SQL

ZERO TO HERO

10 Days Of SQL

@talhakhan

But first, you need to understand

THE BASICS

Let’s start with databases

@talhakhan

1: DATABASES

A database is a collection of RELATED information

ANY collection of related information

- Phonebook
- Shopping List
- To do list
- LinkedIn User Base
- Your 5 favorite food

Database can be stored in different ways

- On Paper
- This PowerPoint
- Comments section
- On a computer
- Ledgers

2: DATABASE APPLICATIONS

Databases can be very large, and databases touch all aspects of our lives

Banking: transactions

Airlines: reservations, schedules

Universities: registration, grades

Sales: customers, products, purchases

Online retailers: order tracking, customized recommendations

Manufacturing: production, inventory, orders, supply chain

Human resources: employee records, salaries, tax deductions

3: WHAT IS DBMS?

DBMS is Database Management System (DUH!)

A Database Management System is a software system that allows access to data contained in a database. It is not an actual database.

The objective of the DBMS is to provide a convenient and effective method of defining, storing, and retrieving the information contained in the database.

- Makes it easy to manage large amounts of information
- Handles security
- Backup
- Access management
- Import/Export data

4: C.R.U.D

CREATE. READ. UPDATE. DELETE

- A good DBMS should have all these functionalities:
- The ability to create records / columns
- The ability to read what's already there in the database
- The ability to update the existing information
- The ability to delete existing information

5: TYPES OF DATABASES (Traditional)

Relational Databases (SQL) and Non-Relational Databases (NoSQL)

Relational Databases (SQL)

- Data is stored in one or more tables
- Each table has columns and a row
- A unique key identifies each row

Non-Relational Databases (NoSQL)

- Data is not stored in tables
- It can be stored in:
  - Key-value stores
  - Documents (JSON, XML etc.)
  - Graphs

6: Relational Databases

A relational database stores and organizes data into tables linked together based on common columns.

What is the relation here? What is common between both the tables?

Yes, it's Department ID. That's the 'relation.'

Employees Table

EmployeeID	Name	DepartmentID
1	John	1
2	Sara	2
3	Bob	3
4	Charlie	4
5	Alice	NULL

Department Table

DepartmentID	DepartmentName
1	**HR**
2	**Sales**
3	**IT**
6	**Finance**

7: RDBMS

Relational Database Management System (RDBMS)

Used to create and maintain a relational database

Some examples of RDBMS and vendors:

- MySQL
- Oracle
- PostgreSQL
- Microsoft SQL Server (MSS)

8: SQL

SQL (structured query language) is a standardized language for interacting with RDBMS

But what is a Query?

- A Google search is a query
- Query are requests made to the DBMS for specific information
- As a database's structure becomes more complex, it becomes more difficult to get the specific pieces of information we want.

SQL is used to:

- Perform C.R.U.D operations, as well as other administrative tasks such as security, backup and access management
- Create and manage databases
- Design and create database tables
- SQL code (syntax) used on one RDMBS is not always directly portable to another RDBMS without modifications – slight changes are required

9: Types of SQL

A hybrid language with 4 different types of language combined into one

1. DQL

- Data Query Language
- Used to query (request) the database for information
- Get information that is already there
- SELECT Statements

2. DDL

- Data Definition Language
- Used for defining database schemas
- CREATE, ALTER, DROP, TRUNCATE, RENAME

10: Types of SQL (continued)

A hybrid language with 4 different types of language combined into one

3. DCL

- Data Control Language
- Used for controlling access to the data in the database
- User access and permission management
- GRANT, REVOKE

4. DML

- Data Manipulation Language
- Used for inserting, updating, and deleting data from the database
- INSERT, DELETE, UPDATE

11: Common Data Types

A data type specifies the kind of data that can be stored in a column

Some examples:

- **INT** – Whole Numbers
- **DECIMAL(M,N) / FLOAT** – Decimal Numbers
- **VARCHAR(32)** – String of text of length 32
- **BLOB** – Binary Large Objects, Stores large data
- **DATE** – 'YYYY-MM-DD'
- **TIMESTAMP** – 'YYYY'MM'DD HH:MM:SS' used for recording different events, transactions, continuous values etc.

11: Where is data really stored?

Data is logically stored in rows and columns in a table – but where is it physically stored?

In SQL databases, data is physically stored in fixed-size storage units called data pages, often a few kilobytes.

These pages are organized in a file system managed by the database system.

There are typically two main types of data pages:

Data Pages (lead nodes): These contain the actual data of the tables, like the rows and columns of information (data).

Index Pages (root and intermediate nodes): These are used to speed up searches; they store index information that points to the actual data pages.

12: SQL Query Order of execution

SQL queries are executed in a specific logical order, which is particularly important to understand when dealing with complex queries, including those with **JOIN** operations.

SQL queries run in exactly this order 👇

1. FROM and JOINs
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. DISTINCT
7. ORDER BY
8. LIMIT / OFFSET

1 FROM and JOINs:

The query **starts** with the **FROM** clause, identifying the tables, and performs any **JOIN** operations.

Example: **FROM** Orders **INNER JOIN** Customers **ON** Orders.CustomerID = Customers.CustomerID joins the Orders table with the Customers table based on the customer ID.

2 WHERE:

Filters the rows based on a condition.

Example: **WHERE** Country = 'USA' filters the results to include only orders from customers in the USA. Country is a column in the Customers table.

13: SQL Query Order of execution

3 **GROUP BY:**

Groups rows with the same values in specified columns.

Example: **GROUP BY** Region groups the orders by the region of the customers.

4 **HAVING:** This clause is used to filter groups created by the **GROUP** BY clause. It is similar to the **WHERE** clause but operates on groups rather than rows.

Example: **HAVING COUNT**(Orders.OrderID) > 10 filters the groups to include only countries with more than 10 orders.

5 **SELECT:** The query now selects the columns that are to be displayed in the result set. This is where the actual data from the rows is retrieved, based on the previous steps.

6 **DISTINCT:** Removes duplicate rows.

Example: "**SELECT DISTINCT** Country **FROM** Customers" lists each country only once, regardless of how many orders they have

14: SQL Query Order of execution

7 **ORDER BY:**

This sorts the result set in ascending or descending order. The sorting is done after the data has been selected.

Example: **ORDER BY COUNT**(OrderID) DESC sorts the result by the number of orders in descending order.

8 **LIMIT / OFFSET:**

Finally, these clauses are used to limit the number of rows returned and to specify a starting point for the row count (common in pagination scenarios).

Example: **LIMIT** 5 **OFFSET** 10 returns 5 records starting from the 11th record.

```sql
SELECT Customers.Country, COUNT(Orders.OrderID) AS NumberOfOrders
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID
WHERE Customers.Country = 'USA'
GROUP BY Customers.Country
HAVING COUNT(Orders.OrderID) > 10
ORDER BY COUNT(Orders.OrderID) DESC
LIMIT 5;
```

RECAP

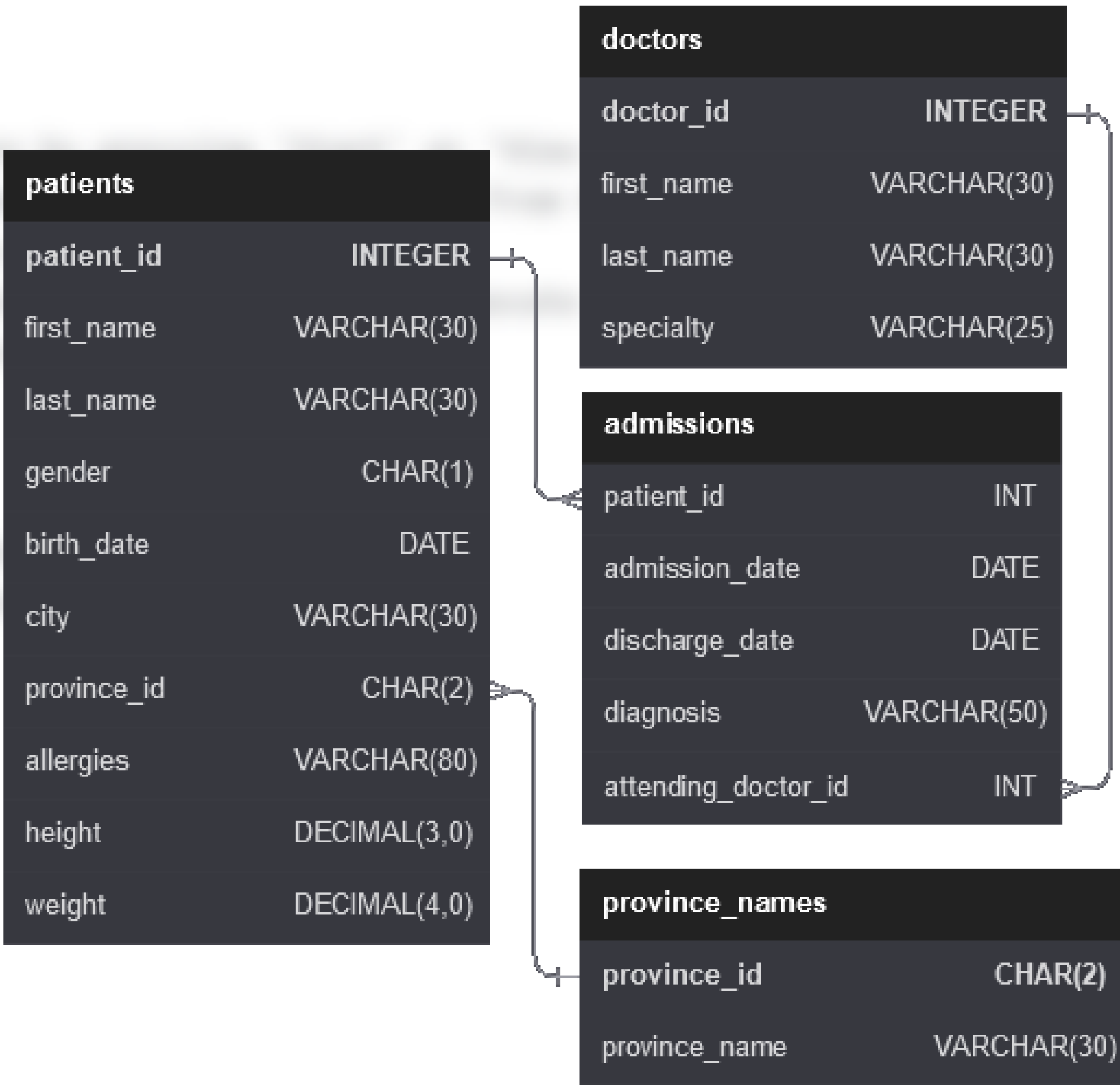
- Database is any collection of related information
- Computers are great for storing databases
- Database Management System (DBMS) makes it easy to create, maintain, and secure a database
- DBMS allow you to perform C.R.U.D. operations and other administrative tasks
- Two types of Databases: Relational vs Non-Relational
- Relation Databases use RDBMS, and RDBMS use SQL to store data in tables with rows and columns
- Non-Relational data is stored using other data structures like JSON, graphs, key-value stores etc.
- SQL data is physically stored in storage units called Data Pages

KEYS, C.R.U.D & JOINS

@talhakhan

Consider the schema of

A HOSPITAL DATABASE



sql-practice.com

But wait, what is a

SCHEMA?

A database schema is a structure that outlines how data is organized in a database, detailing tables, fields, and the relationships between them. It serves as the database's architectural blueprint.

1: PRIMARY KEY

A primary key is a unique identifier for each record in a database table.

Every table should have a primary key to ensure data integrity.

Example:

In the **patients** table**, patient_id** is the primary key, ensuring each patient is uniquely identified.

How do you create a PRIMARY KEY?

```
CREATE TABLE patients (
  patient_id INT PRIMARY KEY,
);
```

CREATE TABLE creates a new table called patients

patient_id is a column that has been created, **INT** defines the data type to be integer, and by using the clause **PRIMARY KEY**, you have made this column a primary key.

2: FOREIGN KEY (FK)

FK establishes a link between two tables.

A foreign key in one table is a primary key of another table (the parent table.)

We use **FKs** to enforce referential integrity between tables.

Example:

'attending_doctor_id' in **'admissions**' table references **'doctor_id**' in **'doctors**'.

You can add a FK by using the following query:

```
ALTER TABLE admissions
ADD FOREIGN KEY (attending_doctor_id) REFERENCES
doctors(doctor_id);
```

3: COMPOSITE KEY

A key composed of two or more columns that, together, uniquely identify a record.

When a single column does not uniquely identify records, you may need to use a combination of 2 or more columns that can uniquely identify a record (row.)

Example:

```
CREATE TABLE patient_allergies (
  patient_id INT,
  birth_date DATE,
  PRIMARY KEY (patient_id, birth_date),
  FOREIGN KEY (patient_id) REFERENCES
patients(patient_id)
);
```

4: CREATING TABLE

To define a new set of data with a specific structure for storage

When: Use when initializing a new database or adding a new category of data

Let us create a table called '**medications**', with fields '**medication_id**', '**name**' and '**prescribed date**.'

```
CREATE TABLE medications (
  medication_id INT PRIMARY KEY,
  name VARCHAR(100),
  prescribed_date DATE
);
```

Remember, there MUST be a primary key, which should be a unique identifier.

You can add a clause to auto increment the primary key.

```
  medication_id INT AUTO_INCREMENT PRIMARY KEY
```

5: ALTERING TABLE

To change the schema of an existing table

When: Use to add, delete, or modify **columns** in an existing table.

Now let's say we need to add a field called 'email' in the table 'patients', that already exists

```
ALTER TABLE patients
 ADD email VARCHAR(255);
```

6: UPDATING RECORDS

To modify existing data within the database

When: Use when data needs to be corrected, updated, or changed.

Let’s say a patient moved to a different city and you want to update their record:

```
UPDATE patients
SET city = 'New City'
WHERE patient_id = 123;
```

7: DELETING RECORDS OR TABLES

To remove data that is no longer needed or relevant

Use with caution when certain data needs to be permanently removed

```
-- Deleting records
DELETE FROM patients
WHERE patient_id = 123;

-- Deleting a table
DROP TABLE IF EXISTS old_patients;
```

You can add comments in your query using ‘--’

It’s always a great practice to add comments.

8: JOINS

To combine rows from two or more tables based on a related column.

Use when you need data from multiple tables to be presented in a single query result set.

There are 4 types of joins:

INNER JOIN: Selects records with matching values in both tables.

SELECT * FROM patients
INNER JOIN admissions **ON** patients.patient_id = admissions.patient_id;

LEFT JOIN: Selects all records from the left table, with matching records from the right table.

SELECT * **FROM** patients **LEFT JOIN** admissions **ON** patients.patient_id = admissions.patient_id;

RIGHT JOIN: Selects all records from the right table, with matching records from the left table.

SELECT * **FROM** patients **RIGHT JOIN** admissions **ON** patients.patient_id = admissions.patient_id;

FULL OUTER JOIN: Selects all records when there is a match in either left or right table.

SELECT * **FROM** patients **FULL OUTER JOIN** admissions **ON** patients.patient_id = admissions.patient_id;

9: Database Description

To understand the structure of a table and its columns

Use for database inspection, debugging, or when altering the database schema.

Whenever you start working on an existing database or schema, you definitely need to understand what the structure is like.

```
-- For MySQL
DESCRIBE patients;

-- For PostgreSQL and others
SELECT column_name, data_type
FROM information_schema.columns
WHERE table_name
```

Operators & Arithmetic functions

@talhakhan

Let’s take an example of

PATIENTS TABLE

```
SELECT * FROM patients
LIMIT 6;
```

patient_id	first_name	last_name	gender	birth_date	city	province_id	allergies	height	weight
1	Donald	Waterfield	M	12/02/1963	Barrie	ON	None	156	65
2	Mickey	Baasha	M	28/05/1981	Dundas	ON	Sulfa	185	76
3	Jiji	Sharma	M	05/09/1957	Hamilton	ON	Penicillin	194	106
4	Blair	Diaz	M	07/01/1967	Hamilton	ON	None	191	104
5	Charles	Wolfe	M	19/11/2017	Orillia	ON	Penicillin	47	10
6	Sue	Falcon	F	30/09/2017	Ajax	ON	Penicillin	43	5

1: WHERE Clause

Used to filter records and specify conditions for selecting rows from a table.

It restricts which rows are to be returned by a query by testing each row against a condition or a set of conditions. If the condition is true for a particular row, that row is included in the result set.

- It supports logical operators such as **=, !=, >, <, >=, <=, BETWEEN, LIKE, IN, AND, OR,** and **NOT**.
- Conditions can include arithmetic expressions, logical comparisons, or functions
- It is used before the aggregation in GROUP BY and cannot be used with aggregate functions directly
- **WHERE** is applied to individual rows before they are grouped or aggregated, which means it is processed before **GROUP BY**, **HAVING**, and **ORDER BY** clauses. (see Day 2OfSQL for details)

```
SELECT
        patient_id,
        first_name,
        last_name,
        weight
FROM patients
WHERE weight > 80;
```

The query above selects all columns for patients who weigh more than 80 kilograms.

2: COMPARISON OPERATORS

Comparison operators are used to compare values in different columns of a database.

Equal To (=): This operator checks if the values on either side are equal.

```
-- To select all male patients
SELECT *
FROM patients
WHERE gender = 'M';
```

Not Equal To (<> or !=): It tests if two values are not equal.

```
-- To select all patients who do not live in New York
SELECT *
FROM patients
WHERE city != 'New York';
```

Greater Than (>): This operator checks if the value on the left is greater than the one on the right

```
-- To select patients taller than 170cm
SELECT *
FROM patients
WHERE height > 170;
```

3: COMPARISON OPERATORS (cont'd)

Comparison operators are used to compare values in different columns of a database.

Less Than (<): It tests if the value on the left is less than the value on the right. SELECT * FROM table WHERE column < 10; would select rows where the column value is less than 10.

-- to select patients who weigh less than 80kg.

```
SELECT *FROM patients
WHERE weight < 80;
```

Greater Than or Equal To (>=): This operator checks if the left value is greater than or equal to the right value.

--to select cardiologists with a **doctor_id** of 10 or higher.

```
SELECT *FROM doctors
WHERE specialty = 'Cardiology' AND doctor_id >= 10;
```

Less Than or Equal To (<=): It checks if the left value is less than or equal to the right value.

-- to select admissions on or before the end of the year 2021

```
SELECT *FROM admissions
WHERE admission_date <= '2021-12-31';
```

4: LOGICAL OPERATORS

Logical operators are used to combine multiple conditions in a WHERE clause, enabling more complex and precise queries.

AND: This operator allows you to combine two or more conditions, and it returns true only if all conditions are true. It's used to narrow down the search results.

-- To select records where the patient was discharged and diagnosed with Appendicitis.

SELECT * **FROM** admissions **WHERE** discharge_date **IS NOT NULL AND** diagnosis = 'Appendicitis';

OR: Returns true if any of the combined conditions is true.

-- To select patients living in either Los Angeles or San Francisco.

SELECT * **FROM** patients **WHERE** city = 'Los Angeles' **OR** city = 'San Francisco';

NOT: Negates a condition, returning true if the condition is false.

-- To select admissions where the diagnosis is not the Flu.

SELECT * **FROM** admissions **WHERE NOT** diagnosis = 'Flu';

5: LOGICAL OPERATORS (cont'd)

Logical operators are used to combine multiple conditions in a WHERE clause, enabling more complex and precise queries.

BETWEEN: Used to filter the result set within a certain range. It is inclusive, meaning it includes the end values.

```
SELECT * FROM patients
WHERE birth_date BETWEEN '1960-01-01' AND '1980-12-31';
```

IN: This operator allows you to specify multiple values in a WHERE clause.

```
-- To select all the patients that reside in either Barrie or Dundas

SELECT * FROM patients
WHERE city IN ('Barrie', 'Dundas');
```

6: LOGICAL OPERATORS (cont'd)

Logical operators are used to combine multiple conditions in a WHERE clause, enabling more complex and precise queries.

LIKE: Used in a WHERE clause to search for a specified pattern in a column

-- To select all patients whose first name starts with the letter 'J'. The LIKE clause uses wildcards – more on this later.

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

IS NULL: It checks for NULL values.

-- To select all the patients where there is no record for allergies

```
SELECT * FROM patients
WHERE allergies IS NULL;
```

7: ARITHMETIC OPERATORS

These are used to perform mathematical calculations on numeric operands.

+ (Addition): Adds two or more numbers.

--To add together the height and weight for each patient. (this example is not practical, but just based on what we have in the patients table)

```
SELECT
    patient_id,
    first_name,
    height + weight AS combined_dimension
FROM patients;
```

- (Subtraction): Subtracts the second number from the first.

-- To subtract the weight from the height for each patient.

```
SELECT
    patient_id,
    first_name,
    height - weight AS dimension_difference
FROM patients;
```

8: ARITHMETIC OPERATORS (Cont'd)

These are used to perform mathematical calculations on numeric operands.

*** (Multiplication):** Multiplies two numbers.

/ (Division): Divides the numerator by the denominator.

% or MOD (Modulo): Returns the remainder of a division operation.

Try executing all of these on the patients table where the data types are INT, or FLOAT.

TEXT FUNCTIONS

@talhakhan

1: LEN

Counts the number of characters in a string, excluding trailing spaces.

LEN is handy for validating input or trimming output.

Examples

Assuming the 'first_name' is **'John**', 'city' is **'Dubai**', 'allergies' is **'Peanuts, Pollen'**

1. FirstNameLength

```
SELECT LEN('John') AS FirstNameLength;
-- Output: 4
```

2. CityLength

```
SELECT LEN('Dubai') AS CityLength;
-- Output: 5
```

3. AllergiesLength

```
SELECT LEN('Peanuts, Pollen') AS AllergiesLength;
-- Output: 15
```

2: Substring

Extracts characters from a string starting at a specified position. It's useful for parsing strings.

Syntax: **SUBSTRING**(string, start, length)

Examples ---

Assuming 'diagnosis' is **'Bronchitis**', 'first_name' is **'Alexander**', 'province_name' is **'Ontario**'

1. DiagnosisStart

```
SELECT SUBSTRING('Bronchitis', 1, 5) AS DiagnosisStart;
-- Output: 'Bronc'
```

2. FirstNamePart

```
SELECT SUBSTRING('Alexander', 3, 3) AS FirstNamePart;
-- Output: 'exa'
```

3. ProvinceEnd

```
SELECT SUBSTRING('Ontario', LEN('Ontario') - 4, 5) AS
ProvinceEnd;
-- Output: 'ario'
```

3: CONCAT

Joins two or more strings into one. It simplifies the concatenation of multiple string fields.

Syntax: ‘CONCAT(string1, string2, ...)

Examples

Assuming 'first_name' is 'John', 'last_name' is 'Doe'

1. FullName

```
SELECT CONCAT('John', ‘  ', 'Doe') AS FullName;
-- Output: 'John Doe'
```

2. DoctorFullName

```
SELECT CONCAT(‘Dr.  ', 'John', ‘  ', 'Doe') AS DoctorFullName;
-- Output: 'Dr. John Doe'
```

3. AdmissionDetails

```
SELECT CONCAT('Admitted:  ', '2023-01-01', ', Discharged: ', '2023-01-10') AS AdmissionDetails;
-- Output: 'Admitted: 2023-01-01, Discharged: 2023-01-10'
```

4: TRIM

Removes spaces from both ends of a string. This is crucial for cleaning data before processing.

Syntax: TRIM(string)

Examples

Assuming 'first_name' and 'province_name' have leading/trailing spaces

1. CleanFirstName

```
SELECT TRIM('  John  ') AS CleanFirstName;
-- Output: 'John'
```

2. CleanLastName

```
SELECT TRIM('  Doe  ') AS CleanLastName;
-- Output: 'Doe'
```

3. CleanProvinceName

```
SELECT TRIM('  Ontario  ') AS CleanProvinceName;
-- Output: 'Ontario'
```

5: REGEXP

Matches strings against regular expression patterns. It's powerful for pattern-based searching

Syntax (using **REGEXP_LIKE** for the example):
REGEXP_LIKE(source_string, pattern)

Example 1

Find doctors with a first name starting with 'Jo’

Assuming the 'first_name' column contains **'John**', **'Joanna**', **'Johnny**’

```
SELECT first_name
FROM doctors
WHERE first_name REGEXP '^Jo’;
```

```
+------------+
| first_name |
+------------+
| John       |
| Joanna     |
| Johnny     |
+------------+
```

6: REGEXP

Matches strings against regular expression patterns. It's powerful for pattern-based searching

Example 2

Identify entries with date format YYYY-MM-DD in diagnosis field

Assuming the 'diagnosis' column contains 'Diagnosed with Hypertension on 2023-01-10', 'Influenza', 'Routine check-up on 2024-03-15’

```
SELECT diagnosis
FROM admissions
WHERE diagnosis REGEXP '\\d{4}-\\d{2}-\\d{2}';
```

```
+----------------------------------------+
| diagnosis                              |
+----------------------------------------+
| Diagnosed with Hypertension on 2023-01-10  |
| Routine check-up on 2024-03-15             |
+----------------------------------------+
```

7: UPPER & LOWER

Convert all characters in a string to uppercase and lowercase respectively.

Syntax: LOWER(string), UPPER(string)

Examples

Assuming 'last_name' is **SMITH**’ and 'city' is **'dubai**’

1. LastNameUpper

```
SELECT UPPER('smith') AS LastNameUpper;
-- Output: 'SMITH’
```

2. LastNameLower

```
SELECT LOWER(‘SMITH') AS LastNameLower;
-- Output: ‘smith'
```

CASES & SUBQUERIES

@talhakhan

1: CASES

The CASE statement goes through conditions and returns a value when the first condition is met. If no conditions are met, it can return an ELSE value.

The **CASE** statement in SQL works like an IF-THEN-ELSE statement. It allows you to implement conditional logic in your SQL queries. You can use **CASE** in any statement or clause that accepts a valid expression, such as **SELECT**, **WHERE**, and **ORDER BY**.

Example 1

-- Assigning a classification based on age:

```
SELECT
        first_name,
        last_name,
CASE
        WHEN birth_date >= '2000-01-01' THEN 'Gen Z’
        WHEN birth_date < '2000-01-01' AND birth_date >=   '1980-01-01'
THEN 'Millennial’
         ELSE 'Other'
END AS generation
FROM patients;
```

The **CASE** statement ends with the keyword **END** to signify the end of the conditional logic. Following **END**, you can use **AS** to give a name (alias) to the new column generated by the **CASE** statement.

2: CASES (Cont'd)

The CASE statement goes through conditions and returns a value when the first condition is met. If no conditions are met, it can return an ELSE value.

Example 2

```
-- Creating a custom sorting order


SELECT
    patient_id,
    city,
CASE city
        WHEN 'Barrie' THEN 1
        WHEN 'Dundas' THEN 2
        WHEN 'Hamilton' THEN 3
        ELSE 4
END AS custom_order
FROM patients
ORDER BY custom_order;
```

3: Subqueries

A subquery is a query that is embedded within the WHERE clause of another query to help narrow down the data that you are looking to retrieve.

A subquery is a query nested within another SQL query.

It can be used in SELECT, INSERT, UPDATE, and DELETE statements, as well as in WHERE, IN, EXISTS, and JOIN clauses.

Example 1

-- Finding doctors who have treated patients with a specific allergy:

```
SELECT DISTINCT
        doctor_id,
        first_name,
        last_name
FROM doctors
WHERE doctor_id IN
       (
        SELECT attending_doctor_id
        FROM admissions
        JOIN patients ON admissions.patient_id = patients.patient_id
        WHERE allergies = 'Penicillin’
        );
```

4: Subqueries (Cont'd)

A subquery is a query that is embedded within the WHERE clause of another query to help narrow down the data that you are looking to retrieve.

Example 2

-- Updating records based on a condition from another table

```
UPDATE patients
SET city = 'Toronto'
WHERE patient_id IN
        (
         SELECT patient_id
         FROM admissions
         WHERE discharge_date < '2021-01-01'
        );
```

5: PRACTICE QUESTIONS

Try these out and write your queries in the comment section.

1. **Write a CASE statement that labels patients as 'Overweight', 'Normal', or 'Underweight' based on their BMI.**
2. **Create a subquery to select all patients who have been admitted more than three times.**
3. **Use a CASE statement within an ORDER BY clause to create a custom sort order based on the patients' cities.**

CTEs

@talhakhan

What are CTEs?

And why are they so important?

1: What are CTEs?

A Common Table Expression (CTE) is a temporary result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement.

CTEs are often used for simplifying complex queries, improving readability, and recursion.

2: And why are they so Important?

1. Readability and Maintenance:

   CTEs can make complex queries more understandable by breaking them down into simpler parts.

2. Reusability:

   They allow you to define a query once and then reference it multiple times in subsequent queries.

3. Recursive Queries:

   CTEs facilitate writing recursive queries which are useful in hierarchical or tree-structured data scenarios.

3: When to use CTEs?

1. To simplify complex joins and subqueries.
2. When working with hierarchical data.
3. For queries where you need to reference a derived table multiple times.
4. In situations where a query is divided into multiple, logical segments.

4: Syntax

The CTE query starts with a “WITH” and is followed by the Expression Name

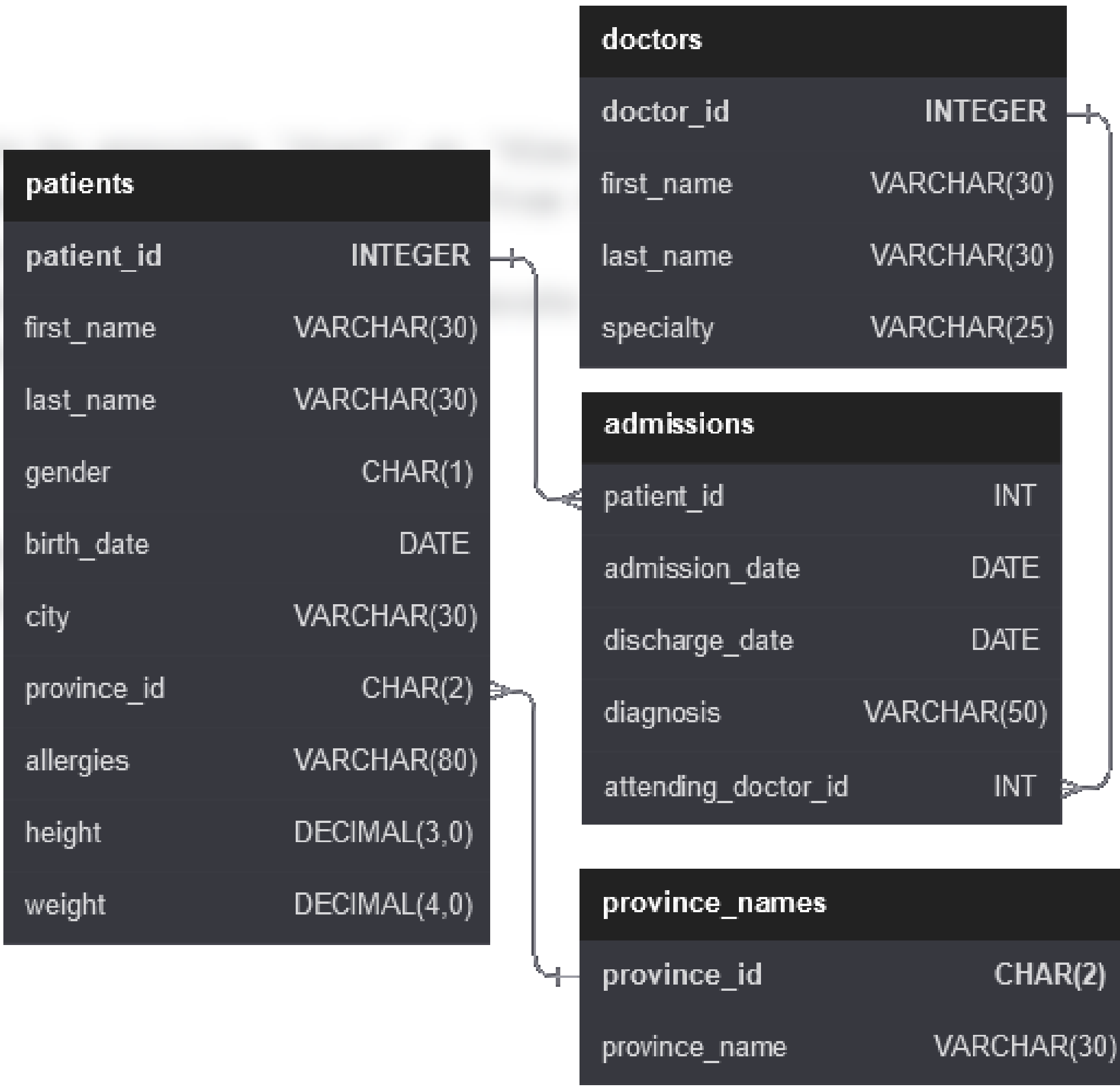
```
WITH CTE_Name AS
        (
   -- CTE query here
        )
```

To view the CTE result, we use a SELECT query with the CTE expression name.

```
SELECT * FROM CTE_Name;
```

Consider the schema of

A HOSPITAL DATABASE



sql-practice.com

5: EXAMPLE 1

List of doctors with count of their admissions

```
--First we define the CTE here

WITH DoctorAdmissions AS
        (
        SELECT attending_doctor_id,
        COUNT(*) AS AdmissionCount
        FROM admissions
        GROUP BY attending_doctor_id
        )
SELECT
    d.doctor_id,
    d.first_name,
    d.last_name,
    da.AdmissionCount
FROM doctors d
LEFT JOIN DoctorAdmissions da ON d.doctor_id =
da.attending_doctor_id;
```

6: EXAMPLE 2

Patients with Multiple Admissions

```
WITH MultipleAdmissions AS
(
    SELECT patient_id
    FROM admissions
    GROUP BY patient_id
    HAVING COUNT(admission_id) > 1
)
SELECT
    p.patient_id,
    p.first_name,
    p.last_name
FROM patients p
INNER JOIN MultipleAdmissions ma ON
p.patient_id = ma.patient_id;
```

7: RECURSIVE CTEs

A Recursive CTE is a special type of CTE in SQL that is used to perform recursive operations.

This means it can repeatedly execute a query and return subsets of data until it meets a specified condition, much like a loop in programming languages.

Recursive CTEs are particularly useful for dealing with hierarchical or tree-structured data, like organizational structures, folder directories, or graph data.

8: STRUCTURE OF RECURSIVE CTEs

A recursive CTE consists of two main parts:

1. Anchor Member:

This is the initial query that returns the base result set. It serves as the starting point of the recursion.

2. Recursive Member:

This is a query that references the CTE itself and is responsible for the recursive behavior. It's combined with the anchor member using a UNION ALL operator.

9: Syntax OF RECURSIVE CTEs

```
WITH RECURSIVE CTE_Name AS
(
   -- Anchor member
         SELECT ....
         FROM ....

         UNION ALL

   -- Recursive member
         SELECT ....
         FROM ....
         JOIN CTE_Name ON ...
         WHERE ....
)
SELECT * FROM CTE_Name;
```

10: When to Use Recursive CTEs

1. Traversing hierarchical or tree-structured data.
2. Generating series of numbers or dates.
3. Walking through graph-like structures.

11: PRACTICE QUESTIONS

Try these out and write your queries in the comment section.

1. **Recursive CTE for Treatment History: Write a recursive CTE that traces the treatment history for a patient, assuming a treatment_history table exists with patient_id, treatment_id, and previous_treatment_id.**
2. **CTE for Average Treatment Duration: Using a hypothetical treatment_durations table with treatment_id and duration, create a CTE to find the average duration of each treatment type.**
3. **Admission Trends CTE: Write a CTE that shows the number of admissions per month for the current year, assuming the admissions table has an admission_date field.**

Stored Procedures & Triggers

@talhakhan

What are Stored Procedures & Triggers?

And how can they make my life easier?

1: What are Stored Procedures?

Stored PROCEDURES are a set of SQL statements that are stored in the database and can be executed repeatedly.

They can accept parameters, perform complex operations, and return results.

2: And why are they used?

Code reuse and standardization.

Easier maintenance and debugging.

Improved performance.

- Stored Procedures can enhance performance as they are compiled once and stored in executable form.

Security:

- They help in controlling access to data (users can be granted permission to execute a procedure without having direct access to the underlying tables).

Reduced Network Traffic:

- Executing complex operations directly on the server side reduces the amount of data sent over the network.

3: Syntax

Creating a stored procedure:

```
CREATE PROCEDURE procedure_name
(parameter_list)
AS
BEGIN
    -- SQL statements
END;
```

Once you have created it, you need to call a stored procedure (DUH!!!)

4: Calling stored procedures

Here's how you generally call a stored procedure:

SQL SERVER:

```
EXEC procedure_name;
-- Or if the procedure takes parameters:
EXEC procedure_name @param1, @param2, ...;
```

MySQL:

```
CALL procedure_name();
-- Or if the procedure takes parameters:
CALL procedure_name(param1, param2, ...);
```

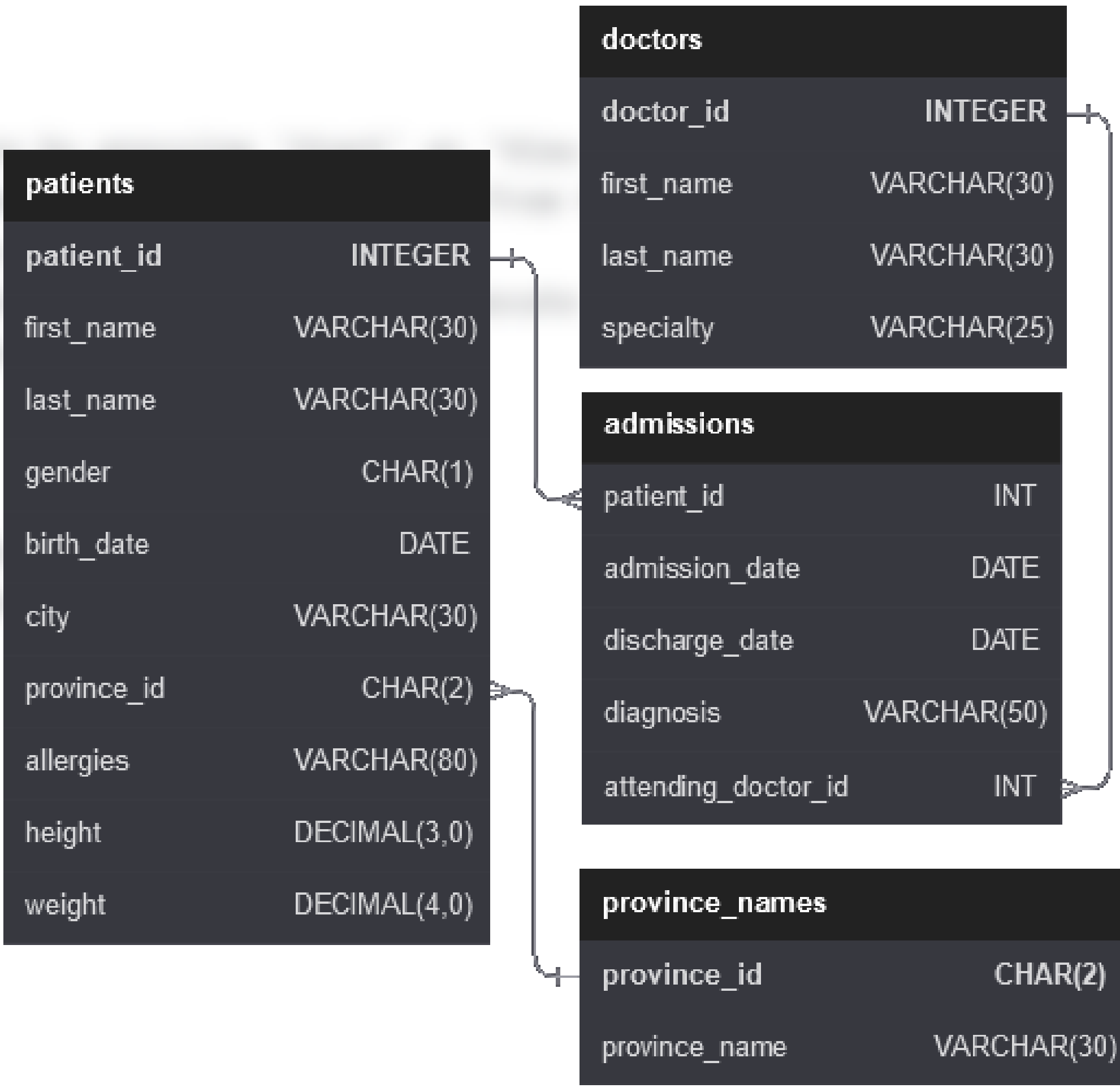
PostgreSQL:

PostgreSQL uses a SELECT statement to call functions (which are similar to stored procedures).

```
SELECT procedure_name();
-- Or if the procedure takes parameters:
SELECT procedure_name(param1, param2, ...);
```

Consider the schema of

A HOSPITAL DATABASE



sql-practice.com

5: EXAMPLE 1

Creating a Stored Procedure for Adding a New Patient

```
CREATE PROCEDURE AddPatient
(
    @FirstName VARCHAR(50),
    @LastName VARCHAR(50),
    @BirthDate DATE,
    @City VARCHAR(50)
)
AS
BEGIN
    INSERT INTO patients (first_name, last_name, birth_date, city)
    VALUES (@FirstName, @LastName, @BirthDate, @City);
END;
```

6: EXAMPLE 2

Stored Procedure to Calculate Age of Patients

```
CREATE PROCEDURE CalculatePatientAge
AS
BEGIN
    SELECT
     patient_id,
     DATEDIFF(year, birth_date, GETDATE()) AS Age
    FROM patients;
END;
```

7: Triggers

Triggers are a type of stored procedure that automatically execute or fire when certain events occur in the database, like **INSERT**, **UPDATE**, or **DELETE** operations.

8: Syntax of Triggers

```
CREATE TRIGGER trigger_name
ON table_name
AFTER | BEFORE INSERT, UPDATE, DELETE
AS
BEGIN
    -- SQL statements
END;
```

9: Why do we use Triggers?

Enforce business rules and data integrity.

Automatic execution ensures consistency.

Useful for auditing and monitoring changes.

10: EXAMPLE 3

Trigger to Audit Patient Admissions

```
CREATE TRIGGER AuditPatientAdmission
AFTER INSERT ON admissions
FOR EACH ROW
BEGIN
   INSERT INTO
         admission_audit
                (
                admission_id,
                audit_time
                )
   VALUES (
                NEW.admission_id,
                CURRENT_TIMESTAMP
                );
END;
```

11: EXAMPLE 4

Trigger to Update Patient Status on Discharge

```
CREATE TRIGGER UpdatePatientStatus
AFTER UPDATE ON admissions
FOR EACH ROW
BEGIN
    IF NEW.discharge_date IS NOT NULL THEN
        UPDATE patients
        SET status = 'Discharged'
        WHERE patient_id = NEW.patient_id;
    END IF;
END;
```

The END IF statement in SQL, particularly in the context of this trigger, is used to mark the end of an IF conditional block

12: PRACTICE QUESTIONS

Try these out and write your queries in the comment section.

1. **Create a Stored Procedure:**

Write a stored procedure to update the city of a specific patient based on the patient ID.

2. **Trigger for New Doctor:**

Create a trigger that inserts a welcome message into a doctor_logs table whenever a new doctor is added.

3. **Audit Trail Trigger:**

Develop a trigger to track changes in patient's weight, storing the old and new weights in an audit table whenever the weight is updated.

Best Practices & Query Optimization

@talhakhan

How to write SQL queries efficiently?

And make you stand out?

1: Query Design and Structure:

1. Select Only Required Columns: Specify only necessary columns instead of SELECT *.
2. Use Joins Wisely: Prefer joins over subqueries; use INNER JOIN over OUTER JOIN when only matching records are needed.
3. Filter Early with WHERE Clauses: Apply WHERE clauses early to reduce dataset size.
4. Limit the Use of Wildcards: Use wildcards cautiously, especially at the beginning of a string in a LIKE clause.
5. Avoid Unnecessary Columns in GROUP BY: Only include essential columns.
6. Use LIMIT or TOP: Restrict the number of rows returned for better performance.

2: Query Design and Structure (cont'd)

7. Group Data with GROUP BY: For grouping similar data together.
8. Employ CASE Statements: Implement conditional logic within queries.
9. Simplify Queries with Views: Use views to streamline complex queries.
10. Utilize Aggregate Functions: Like SUM, AVG, COUNT for large datasets.
11. Keep Queries Readable: Use table **aliases** for simplicity and clarity.
12. Test and Analyze: Experiment with different approaches and use tools like EXPLAIN to analyze performance.

3: Query Design and Structure (cont'd)

13. Format Queries Consistently: Consistent formatting aids in readability and maintenance. Write each field in a new row in SELECT statements
14. Avoid Using Functions on Indexed Columns in WHERE Clause: This can prevent the use of indexes.
15. While subqueries can slow down performance in certain scenarios, they're necessary in others. Test to find the best approach

4: Indexing and Data Retrieval

1. Use Indexes Effectively: Including columns in joins and WHERE clauses.
2. Optimize Data Types: Choose appropriate data types for optimal performance.
3. Use Temporary Tables: Simplify complex queries and improve performance.

4: Performance Tuning

1. Analyze Query Execution Plans: Understand and optimize query paths.
2. Use UNION ALL instead of UNION: When duplicates aren't an issue, for better performance.
3. Use EXISTS Over IN/NOT IN: Prefer EXISTS in certain scenarios but test for performance differences.
4. Optimize Subqueries: Ensure efficiency, especially in correlated subqueries.

6: Data Management

1. Normalize Data Where Appropriate: To reduce data redundancy and improve data integrity.
2. Denormalize if Necessary: In some scenarios, denormalization can speed up read-heavy operations.
3. Partition Large Tables: Helps in managing and querying large datasets more efficiently.

7: Server & Resource Management

1. Monitor Server Performance: Regularly check and optimize server settings.
2. Manage Transaction Logs: Keep an eye on log sizes in high-transaction environments.
3. Update Statistics and Rebuild Indexes: Maintain performance over time.
4. Use Stored Procedures: Reduce network traffic and enhance efficiency.
5. Maintain Database Statistics: Keep statistics up-to-date for optimal query planning.
6. Leverage Window Functions: For complex calculations over sets of row

TheDataDialogue

Mini Project

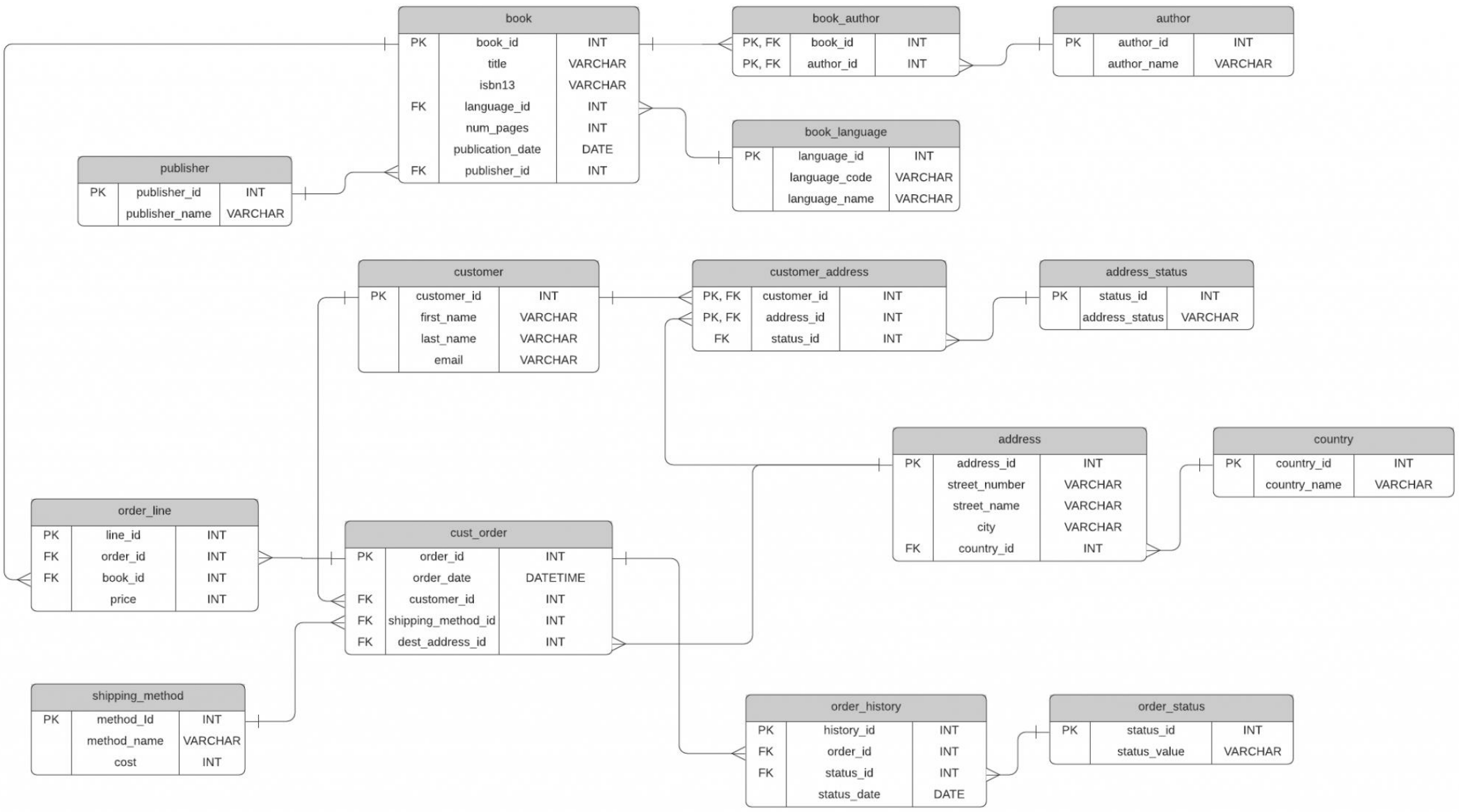
@talhakhan

If you have made so far

You already are a SQL intermediate user

Bookstore Inventory and Sales Analysis

The Database Schema



Courtesy of **databasestar**

GOAL

Enhance a bookstore's operations using SQL by analyzing sales, managing inventory, and understanding customer behavior.

Key Objectives

1. Analyze sales trends and inventory.
2. Automate data management tasks.
3. Generate actionable business insights.

Where is the data?

You can download it at

https://github.com/thedatadialogue/Databases

Project Tasks Breakdown

BASIC TASK: 1

List All Books with Authors and Categories

Task:

- Retrieve a list of all books, including their authors and categories.
- **Hint:** Join **books**, **book_authors, authors,** and **categories** tables.

BASIC TASK: 2

Total Sales for Each Book

Task:

- Calculate the total sales (quantity sold) for each book.
- **Hint:** Use order_items table and join with books

BASIC TASK: 3

Find the Most Popular Category

Task:

- Determine which book category is most popular based on the quantity of books sold.
- **Hint**: Aggregate sales data from **order_items** and join with **books** and **categories**

INTERMEDIATE TASK 1:

Monthly Sales Analysis

Task:

- Analyze the total sales amount for each month.
- **Hint**: Use **orders** and **order_items**. Extract month and year from **order_date**.

INTERMEDIATE TASK 2:

Top 5 Bestselling Authors

Task:

- Identify the top 5 authors based on the number of books sold.
- **Hint**: Join **books**, **book_authors**, **authors**, and **order_items**

INTERMEDIATE TASK 3:

Average Order Value Per Customer

Task:

- Calculate the average order value for each customer.
- **Hint**: Aggregate order totals in **orders** and average them for each customer in **customers**

ADVANCED TASK 1:

Sales Trend Over Years

Task:

- Determine how sales have trended over the years.
- **Hint**: Use window functions with **orders** and **order_items** to analyze year-over-year trends.

ADVANCED TASK 2:

Customer Loyalty Program

Task:

- Identify customers eligible for a loyalty program based on their order history.
- **Hint**: Use **orders** and **order_items** to calculate total purchases per customer, then set a threshold for loyalty eligibility.

CTEs, CASES, Stored Procedures and Subqueries

- **Task 1: Sales Category Analysis Using CASE**

Classify each order into a 'High', 'Medium', or 'Low' sales category based on the total amount

- **Task 2: Top Selling Books Using Subquery**

Find the top 3 selling books using a subquery.

- **Task 3: Customer Purchase History Using CTE**

Use a CTE to create a report of each customer's total purchases.

- **Task 4: Add New Book Using Stored Procedure**

Create a stored procedure to add a new book to the database.

KEEP PRACTICING
UPSKILL YOURSELF
AND GOOD THINGS AWAIT YOU

Stay tuned