**SQL DIABETIES DATASET**

1. Retrieve the Patient\_id and ages of all patients

SELECT Patient\_id, age

FROM sugar;

1. Select all female patients who are older than 40.

SELECT \*

FROM sugar

WHERE gender = 'female' AND age > 40;

1. Calculate the average BMI of patients.

SELECT AVG(bmi) as average\_bmi

FROM sugar;

1. List patients in descending order of blood glucose levels.

SELECT \*

FROM sugar

ORDER BY blood\_glucose\_level DESC;

1. Find patients who have hypertension and diabetes.

SELECT \*

FROM sugar

WHERE hypertension = 1 AND diabetes = 1;

1. Determine the number of patients with heart disease.

SELECT COUNT(\*) as num\_patients\_with\_heart\_disease

FROM sugar

WHERE heart\_disease = 1;

1. Group patients by smoking history and count how many smokers and nonsmokers there are.

SELECT smoking\_history, COUNT(\*) as num\_patients

FROM sugar

GROUP BY smoking\_history;

1. Retrieve the Patient\_ids of patients who have a BMI greater than the average BMI.

SELECT Patient\_id

FROM sugar

WHERE bmi > (SELECT AVG(bmi) FROM sugar);

1. Find the patient with the highest HbA1c level and the patient with the lowest HbA1clevel.

SELECT \*

FROM sugar

WHERE HbA1c\_level = (SELECT MAX(HbA1c\_level) FROM sugar)

UNION ALL

SELECT \*

FROM sugar

WHERE HbA1c\_level = (SELECT MIN(HbA1c\_level) FROM sugar);

1. Calculate the age of patients in years (assuming the current date as of now).

SELECT Patient\_id,

FLOOR(DATEDIFF(CURRENT\_DATE, age) / 365.25) as age\_in\_years

FROM sugar;

1. Rank patients by blood glucose level within each gender group.

SELECT Patient\_id,

gender,

blood\_glucose\_level,

RANK() OVER (PARTITION BY gender ORDER BY blood\_glucose\_level DESC) as glucose\_level\_rank

FROM sugar;

1. Update the smoking history of patients who are older than 50 to "Ex-smoker."

UPDATE sugar

SET smoking\_history = 'Ex-smoker'

WHERE age > 50;

1. Insert a new patient into the database with sample data

INSERT INTO sugar (Patient\_id, EmployeeName, gender, age, hypertension, heart\_disease, smoking\_history, bmi, HbA1c\_level, blood\_glucose\_level, diabetes)

VALUES

(new\_patient\_id, 'John Doe', 'male', 55, 'no', 'no', 'non-smoker', 25.5, 5.6, 110, 'no');

1. Delete all patients with heart disease from the database

DELETE FROM sugar

WHERE heart\_disease =1;

1. Find patients who have hypertension but not diabetes using the EXCEPT operator.

SELECT Patient\_id

FROM sugar

WHERE hypertension = 1

MINUS

SELECT Patient\_id

FROM sugar

WHERE diabetes = 1;

1. Define a unique constraint on the "patient\_id" column to ensure its values are unique.

ALTER TABLE sugar

ADD CONSTRAINT unique\_patient\_id UNIQUE (patient\_id);

1. Create a view that displays the Patient\_ids, ages, and BMI of patients.

CREATE VIEW patient\_info\_view AS

SELECT Patient\_id, age, bmi

FROM sugar;

SELECT \* FROM patient\_info\_view;

1. Suggest improvements in the database schema to reduce data redundancy and improve data integrity.

**Normalization:**

Review the current schema for normalization opportunities. Break down tables into smaller, related tables to eliminate redundant data and reduce the likelihood of update anomalies.

**Primary Keys and Foreign Keys:**

Ensure that each table has a primary key to uniquely identify each record.

Use foreign keys to establish relationships between tables, enforcing referential integrity.

**Data Types:**

Choose appropriate data types for each column to optimize storage and ensure data accuracy.

For example, use DATE for date columns, NUMBER for numeric data, and VARCHAR2 for variable character data.

**Use Enumerations or Lookup Tables:**

Replace string-based fields with enumerations or use lookup tables for fields with a limited set of possible values. This improves consistency and reduces data entry errors.

Example: Create a lookup table for smoking history with entries like 'Smoker,' 'Non-smoker,' and 'Ex-smoker.'

**Composite Keys:**

Consider using composite keys when necessary to uniquely identify records based on multiple columns.

**Default Values and Constraints:**

Set appropriate default values for columns.

Add constraints such as NOT NULL, UNIQUE, and CHECK constraints to enforce data integrity rules.

**Indexes:**

Create indexes on columns frequently used in search conditions to improve query performance.

However, avoid excessive indexing, as it can impact write performance.

**Views:**

Use views to present data in a way that makes sense for certain use cases without duplicating the underlying data.

**Stored Procedures and Triggers:**

Implement stored procedures to encapsulate complex logic and promote code reusability.

Use triggers to enforce additional business rules or automatically update related data.

**Document Data Integrity Rules:**

Document and communicate data integrity rules to ensure consistency in data management practices.

**Regular Maintenance:**

Perform regular database maintenance, including index rebuilding and statistics gathering, to optimize performance.

**Consider Partitioning:**

For large tables, consider partitioning based on a key, such as date or region, to improve query performance and maintenance.

1. Explain how you can optimize the performance of SQL queries on this dataset

**Indexing:**

Properly index columns used in WHERE clauses and JOIN conditions.

Avoid over-indexing, as it can impact insert and update performance.

Regularly monitor and maintain indexes to ensure they remain effective.

**Query Optimization:**

Analyze and optimize the SQL queries, ensuring they are well-structured and utilize efficient joins.

Use the EXPLAIN PLAN statement to understand how the database processes the query, and adjust accordingly.

**Limiting the Result Set:**

Retrieve only the necessary columns using SELECT, and avoid using SELECT \* if unnecessary.

Use the LIMIT clause to restrict the number of rows returned, especially for large datasets.

**Normalization:**

Normalize the database schema to reduce redundancy and improve data integrity.

This can result in smaller tables, reducing the amount of data that needs to be processed in queries.

**Denormalization (Carefully):**

In some cases, denormalization may be beneficial to reduce the number of joins and improve query performance for read-heavy workloads.

However, denormalization comes with trade-offs and should be approached carefully.

**Caching:**

Implement caching mechanisms to store and reuse frequently accessed query results.

Use database caching solutions or application-level caching to reduce the load on the database.

**Partitioning:**

Consider partitioning large tables based on a key, such as date or region.

Partitioning can improve query performance by allowing the database to scan only relevant partitions.

**Materialized Views:**

Use materialized views to precompute and store the results of complex queries.

This can significantly reduce the processing time for frequently executed queries.

**Regular Maintenance:**

Schedule regular database maintenance tasks, such as updating statistics, rebuilding indexes, and optimizing storage.

**Connection Pooling:**

Implement connection pooling to reuse database connections and reduce the overhead of establishing new connections for each query.

**Avoid Using Functions in WHERE Clauses:**

Using functions in WHERE clauses can prevent the use of indexes. Where possible, avoid using functions directly on indexed columns in WHERE clauses.

**Batch Processing:**

For large data operations, use batch processing instead of individual row processing to minimize the number of transactions.