Pharmacy Store(CureCabin)

SIDHANT KR AGRAWAL(2021495) || PARAS DHIMAN (2021482)

SQL EMBEDDED:-

1.) Find the most commonly prescribed medication for each doctor

```
with connections['mydb'].cursor() as cursor:
cursor.execute('SELECT Doctors.DoctorID,
Doctors.Doctor_Name_Fname,\
Doctors.Doctor_Name_Lname, Medicines.Name, COUNT(*) AS
PrescriptionCount\
FROM Doctors\
JOIN Prescription ON Prescription.DoctorID = Doctors.DoctorID\
JOIN Medicines ON Medicines.MedicineID = Prescription.MedicineID\
GROUP BY Doctors.DoctorID, Medicines.MedicineID\
HAVING COUNT(*) = (\
SELECT MAX(Count) FROM (\
SELECT DoctorID, MedicineID, COUNT(*) AS Count\
FROM Prescription\
GROUP BY DoctorID, MedicineID\
) AS T\
WHERE T.DoctorID = Doctors.DoctorID\
);')
```

Explanation:-

This is a Python code snippet that uses a database connection object named mydb to execute a SQL query. The SQL query selects data from three tables named Doctors, Prescription, and Medicines.

The query joins the three tables using the JOIN clause on the DoctorID and MedicineID columns in the Prescription table and the corresponding columns in the Doctors and Medicines tables, respectively.

The SELECT clause in the query selects five columns: DoctorID, Doctor_Name_Fname, Doctor_Name_Lname, Name (of the medicine), and PrescriptionCount. The PrescriptionCount column is calculated using the COUNT function, which counts the number of times each combination of DoctorID and MedicineID occurs in the Prescription table.

The GROUP BY clause groups the results by DoctorID and MedicineID and calculates the PrescriptionCount for each group.

The HAVING clause filters the groups to include only those with the highest PrescriptionCount for each DoctorID. It uses a subquery to find the maximum PrescriptionCount for each DoctorID and includes only those groups in the final result set.

In summary, the SQL query selects the doctors who prescribe the most of a particular medicine, along with the number of prescriptions they have written for that medicine.

2.) **The number of prescriptions issued by each doctor for a particular medication,along with the total dosage prescribed**

```
with connections['mydb'].cursor() as cursor:
cursor.execute('SELECT Doctors.Doctor_Name_Fname,
Doctors.Doctor_Name_Mname,\
Doctors.Doctor_Name_Lname, COUNT(*) AS NumPrescriptions,\
SUM(Prescription.Dosage) AS TotalDosage\
FROM Doctors\
JOIN Prescription ON Prescription.DoctorID = Doctors.DoctorID\
JOIN Medicines ON Medicines.MedicineID = Prescription.MedicineID\
WHERE Medicines.Type = "Tablet"\
GROUP BY Doctors.Doctor_Name_Fname,
Doctors.Doctor_Name_Mname,\
Doctors.Doctor_Name_Lname\
ORDER BY NumPrescriptions DESC;')
row = cursor.fetchall()
```

Explanation:-

This is a Python code snippet that uses a database connection object named mydb to execute a SQL query. The SQL query selects data from three tables named Doctors, Prescription, and Medicines.
The query joins the three tables using the JOIN clause on the DoctorID and MedicineID columns in the Prescription table and the corresponding columns in the Doctors and Medicines tables, respectively.

The SELECT clause in the query selects four columns: Doctor_Name_Fname, Doctor_Name_Mname, Doctor_Name_Lname, NumPrescriptions, and TotalDosage. The NumPrescriptions column is calculated using the COUNT function, which counts the number of times each combination of DoctorID and MedicineID occurs in the Prescription table where the Type of medicine is "Tablet". The TotalDosage column is calculated using the SUM function, which calculates the total dosage for each doctor based on the prescription data.

The WHERE clause filters the data to include only those rows where the Type of medicine is "Tablet".

The GROUP BY clause groups the results by the Doctor_Name_Fname, Doctor_Name_Mname, and Doctor_Name_Lname columns and calculates the NumPrescriptions and TotalDosage for each group.

The ORDER BY clause orders the results by the NumPrescriptions column in descending order.

Finally, the code retrieves all rows from the result set using the fetchall method and assigns them to the variable row.

In summary, the SQL query selects the doctors who have prescribed the most number of tablets, along with the total dosage prescribed by each doctor. The Python code then retrieves all rows from the result set and assigns them to the variable row.

OLAP Queries:-

1.) **Query to retrieves the total sales, unique patients and pharmacists for each combination of state, doctor type and medicine name**

```
SELECT
   Patients.Address_city,
   Patients.Address_state,
   Patients.Address_country,
   SUM(Payment.Amount) AS TotalPayments,
   SUM(Prescription.Duration_Days) AS TotalPrescriptionDays,
   COUNT(DISTINCT Prescription.PatientID) AS UniquePatients,
   AVG(DATEDIFF(Prescription.EndDate, Prescription.StartDate)) AS
AvgPrescriptionDuration,
   (SUM(Stock.Quantity) * Medicines.Price) AS TotalStockValue
FROM Patients
JOIN Prescription ON Prescription.PatientID = Patients.PatientID
JOIN Payment ON Payment.OrderID = (SELECT Orders.OrderID FROM
Orders WHERE Orders.PatientID = Prescription.PatientID LIMIT 1)
JOIN Medicines ON Medicines.MedicineID = Prescription.MedicineID
JOIN Stock ON Stock.MedicineID = Medicines.MedicineID
GROUP BY Patients.Address_city, Patients.Address_state,
Patients.Address_country,Medicines.Price
HAVING TotalPayments > 10000 AND TotalPrescriptionDays > 365
ORDER BY TotalPayments DESC, TotalPrescriptionDays DESC;
```

Explanation:-
This SQL query retrieves data on the total payments, prescription days, unique patients, average prescription duration, and total stock value for patients based on their city, state, and country, while filtering the results by total payments and prescription days and ordering the output by total payments and prescription days in descending order

2.) **Query to Calculate the total sales made by each policy provider in 2022, and rank them in descending order**

```
SELECT PolicyProvider.PolicyProvider_Name_Fname,
SUM(Orders.Quantity * Medicines.Price) AS TotalSales
FROM Orders
INNER JOIN Medicines ON Orders.MedicineID = Medicines.MedicineID
INNER JOIN PolicyProvider ON Orders.PatientID =
PolicyProvider.PatientID
WHERE Orders.Date BETWEEN '2022-01-01' AND '2022-12-31'
GROUP BY PolicyProvider.PolicyProvider_Name_Fname
ORDER BY TotalSales DESC;
```

Explanation:-

This SQL query retrieves the total sales of medicines for each policy provider between a specified date range and orders the results by the total sales in descending order.

This query is using the JOIN clause to combine data from three tables: Orders, Medicines, and PolicyProvider. It uses the WHERE clause to filter the results by the date range specified. The query then groups the results by the policy provider name using the GROUP BY clause and calculates the total sales for each provider using the SUM function. Finally, the results are ordered by the total sales for each policy provider in descending order using the ORDER BY clause.

3.) **Query to Calculate the percentage of total sales contributed by each medicine for each branch, for a given date range.**

```
SELECT Medicines.Name, Branches.street_name,
  SUM(Orders.Quantity * Medicines.Price) / TotalSales.Total AS
SalesPercentage
FROM Orders
INNER JOIN Medicines ON Orders.MedicineID = Medicines.MedicineID
INNER JOIN Branches ON Orders.PharmacistID = Branches.BranchID
CROSS JOIN (SELECT SUM(Orders.Quantity * Medicines.Price) AS Total
        FROM Orders
        INNER JOIN Medicines ON Orders.MedicineID =
Medicines.MedicineID
```

```
          INNER JOIN Branches ON Orders.PharmacistID =
Branches.BranchID
          WHERE Orders.Date BETWEEN '2022-01-01' AND '2022-12-31')
AS TotalSales
WHERE Orders.Date BETWEEN '2022-01-01' AND '2022-12-31'
GROUP BY Medicines.Name, Branches.street_name,TotalSales.total;
```

Explanation:-

This SQL query retrieves the total sales of medicines for each policy provider between a specified date range and orders the results by the total sales in descending order.

This query is using the JOIN clause to combine data from three tables: Orders, Medicines, and PolicyProvider. It uses the WHERE clause to filter the results by the date range specified. The query then groups the results by the policy provider name using the GROUP BY clause and calculates the total sales for each provider using the SUM function. Finally, the results are ordered by the total sales for each policy provider in descending order using the ORDER BY clause.

4.) **Query to find the total revenue generated by each pharmacist in a particular branch**

```
SELECT p.Pharmacists_Name_Fname AS PharmacistName,
b.street_name AS BranchLocation, SUM(o.Quantity * m.price) AS
TotalRevenue
FROM Orders o
JOIN Medicines m ON o.MedicineID = m.MedicineID
JOIN Pharmacists p ON o.PharmacistID = p.PharmacistID
JOIN Stock s ON o.MedicineID = s.MedicineID AND o.PharmacistID =
s.PharmacistID
JOIN Branches b ON s.BranchID = b.BranchID
GROUP BY p.Pharmacists_Name_Fname, b.street_name
ORDER BY TotalRevenue DESC;
```

Explanation:-

This SQL query retrieves the total revenue generated by each pharmacist and branch location combination by joining data from five tables: Orders, Medicines, Pharmacists, Stock, and Branches.

The query uses the JOIN clause to join data from the Orders, Medicines, Pharmacists, Stock, and Branches tables. It then groups the results by the pharmacist name and branch location using the GROUP BY clause. The SUM function is used to calculate the total revenue generated by each combination of pharmacist and branch location. Finally, the results are ordered in descending order of total revenue using the ORDER BY clause.

The join conditions are as follows: the Orders table is joined with the Medicines table using the MedicineID, the Orders table is joined with the Pharmacists table using the PharmacistID, the Orders table is joined with the Stock table using both the MedicineID and the PharmacistID, and the Stock table is joined with the Branches table using the BranchID.

Overall, this query is performing data analysis on sales data to determine the total revenue generated by each combination of pharmacist and branch location. It uses SQL joins and aggregate functions to perform the necessary calculations and grouping.

5.) **Find the total sales amount for each medicine across all branches for the year 2022**

```
SELECT Medicines.Name, SUM(Orders.Quantity * Medicines.Price) AS
TotalSales
FROM Orders
INNER JOIN Medicines ON Orders.MedicineID = Medicines.MedicineID
INNER JOIN Branches ON Orders.PharmacistID = Branches.BranchID
WHERE Orders.Date BETWEEN '2022-01-01' AND '2022-12-31'
GROUP BY Medicines.Name;
```

Explanation:-
This SQL query retrieves the total sales of each medicine between a specified date range by joining data from three tables: Orders, Medicines, and Branches.

The query uses the JOIN clause to join data from the Orders table with the Medicines table using the MedicineID column and with the Branches table using the PharmacistID column. The WHERE clause is used to filter the results by the specified date range. The GROUP BY clause is used to group the results by the medicine name. The SUM function is used to calculate the total sales for each medicine by multiplying the quantity ordered with the price of the medicine.

Overall, this query is performing data analysis on sales data to determine the total sales of each medicine between a specified date range. It uses SQL joins and aggregate functions to perform the necessary calculations and grouping.

6.) Calculate the total sales amount of each medicine across all branches

```
SELECT Medicines.Name, SUM(Orders.Quantity * Medicines.Price) AS
TotalSales
FROM Orders
INNER JOIN Medicines ON Orders.MedicineID = Medicines.MedicineID
GROUP BY Medicines.Name;
```

Explanation:-
This SQL query retrieves the total sales of each medicine by joining data from two tables: Orders and Medicines.

The query uses the JOIN clause to join data from the Orders table with the Medicines table using the MedicineID column. The GROUP BY clause is used to group the results by the medicine name. The SUM function is used to calculate the total sales for each medicine by multiplying the quantity ordered with the price of the medicine.

Overall, this query is performing data analysis on sales data to determine the total sales of each medicine. It uses SQL joins and aggregate functions to perform the necessary calculations and grouping.

Triggers:-

1.) **Automatically update the stock table when a new order is placed**

```
delimiter |

CREATE TRIGGER update_stock
AFTER INSERT ON Orders
FOR EACH ROW
        BEGIN
                UPDATE Stock
                SET Quantity = Quantity - NEW.Quantity
                WHERE MedicineID = NEW.MedicineID AND
BranchID = NEW.PharmacistID;
        END;
|
delimiter ;
```

Explanation:-

This MySQL query creates a database trigger called update_stock. A trigger is a named database object that is associated with a table, and it is executed automatically in response to certain events or changes that occur to the associated table. In this case, the trigger is executed automatically after an insert operation is performed on the Orders table.

The FOR EACH ROW clause specifies that the trigger will be executed once for each row that is affected by the insert operation.

The BEGIN and END keywords enclose the code block that will be executed when the trigger is fired. The trigger code block performs an update operation on the Stock table, which updates the Quantity column by subtracting the quantity of the new order inserted in the Orders table. The WHERE clause ensures that only the stock record for the medicine and branch associated with the new order is updated.

The delimiter statement is used to change the delimiter to | so that the entire trigger code block can be executed as a single statement. The delimiter is then changed back to the default ; at the end of the trigger code block.

Overall, this trigger is designed to update the stock quantity for a medicine in a branch after a new order is inserted into the Orders table. It helps ensure that the stock quantity is always accurate and up-to-date.

2.) **update is made to the Stock table. It will retrieve the name of the branch and the name of the medicine associated with the updated record using SELECT statements, and then check if the updated quantity is less than or equal to the threshold. If so, it will insert a new record into the Notifications table with a message indicating that the stock is running low for that specific branch and medicine.**

```
delimiter |
CREATE TRIGGER update_stock_threshold
AFTER UPDATE ON Stock
FOR EACH ROW
BEGIN
   DECLARE v_branch_name VARCHAR(255);
   DECLARE v_medicine_name VARCHAR(255);
   SELECT Pharmacists.Pharmacists_Name_Fname INTO
v_branch_name
   FROM Pharmacists
   WHERE Pharmacists.PharmacistID = NEW.PharmacistID;
```

```
    SELECT Medicines.Name INTO v_medicine_name
    FROM Medicines
    WHERE Medicines.MedicineID = NEW.MedicineID;
    IF NEW.Quantity <= NEW.Threshold THEN
        INSERT INTO Notifications (NotificationType, Message)
        VALUES ('Stock Alert', CONCAT('Low stock alert: The
branch ', v_branch_name, ' has only ', NEW.Quantity, ' units of ',
v_medicine_name, ' remaining.'));
    END IF;
END;

|
delimiter ;
```

Explanation:-

This MySQL query creates a database trigger called update_stock_threshold. The trigger is executed automatically after an update operation is performed on the Stock table. The FOR EACH ROW clause specifies that the trigger will be executed once for each row that is affected by the update operation.

The trigger code block starts with the BEGIN keyword and ends with the END keyword. The code block first declares two variables v_branch_name and v_medicine_name to store the branch name and medicine name corresponding to the updated row.

The SELECT statements are used to retrieve the branch name and medicine name from the Pharmacists and Medicines tables, respectively, based on the PharmacistID and MedicineID values in the updated row.

The IF statement checks if the new Quantity value in the updated row is less than or equal to the new Threshold value. If this condition is true, then a new row is inserted into the Notifications table with a NotificationType of "Stock Alert" and a message that indicates that the branch has low stock for the specified medicine.

The delimiter statement is used to change the delimiter to | so that the entire trigger code block can be executed as a single statement. The delimiter is then changed back to the default ; at the end of the trigger code block.

Overall, this trigger is designed to generate a stock alert notification whenever the Quantity value in the Stock table is updated to a value less than or equal to the Threshold value. The trigger retrieves the corresponding branch and medicine names and inserts a new row into the Notifications table with a message that includes this information.