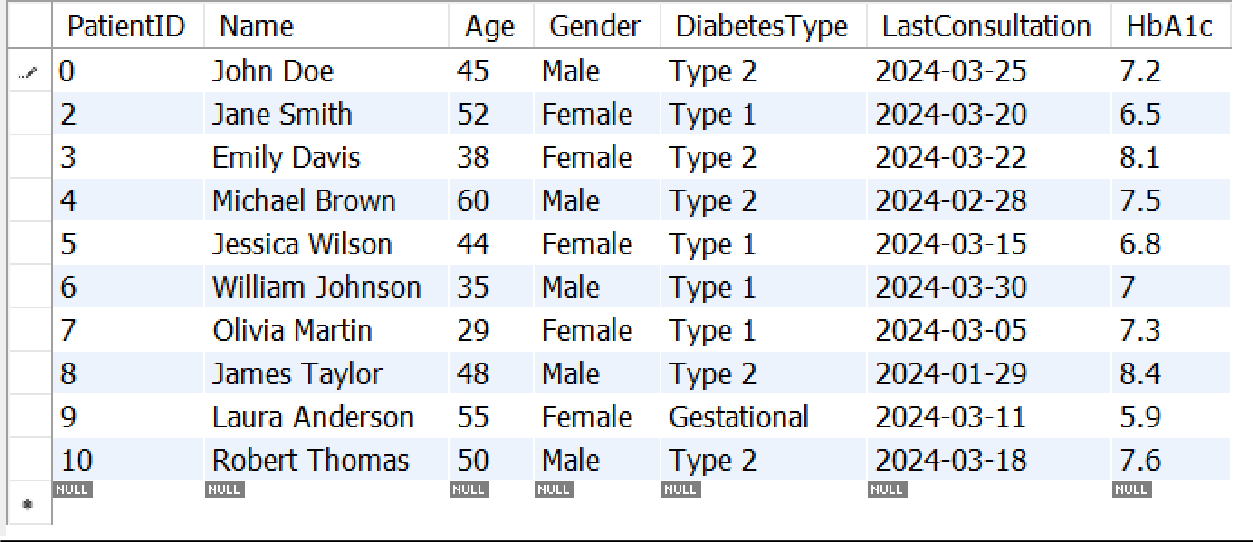
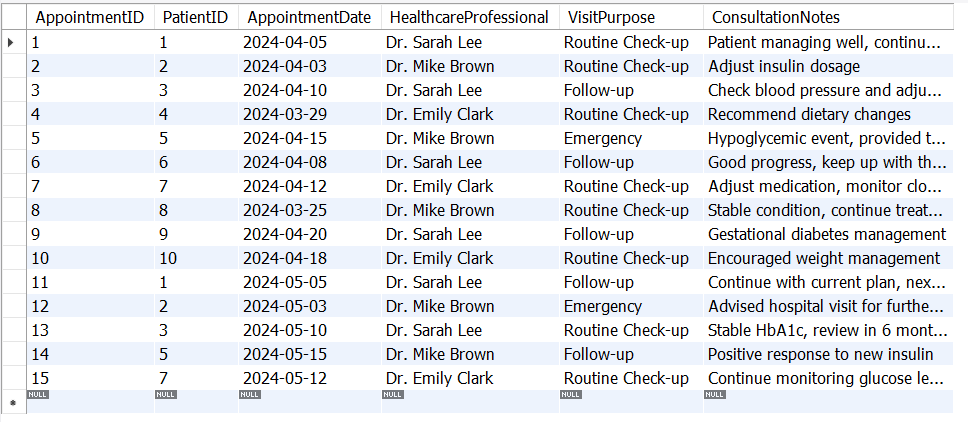
**Minor Project 2: Healthcare Data Insights Report**

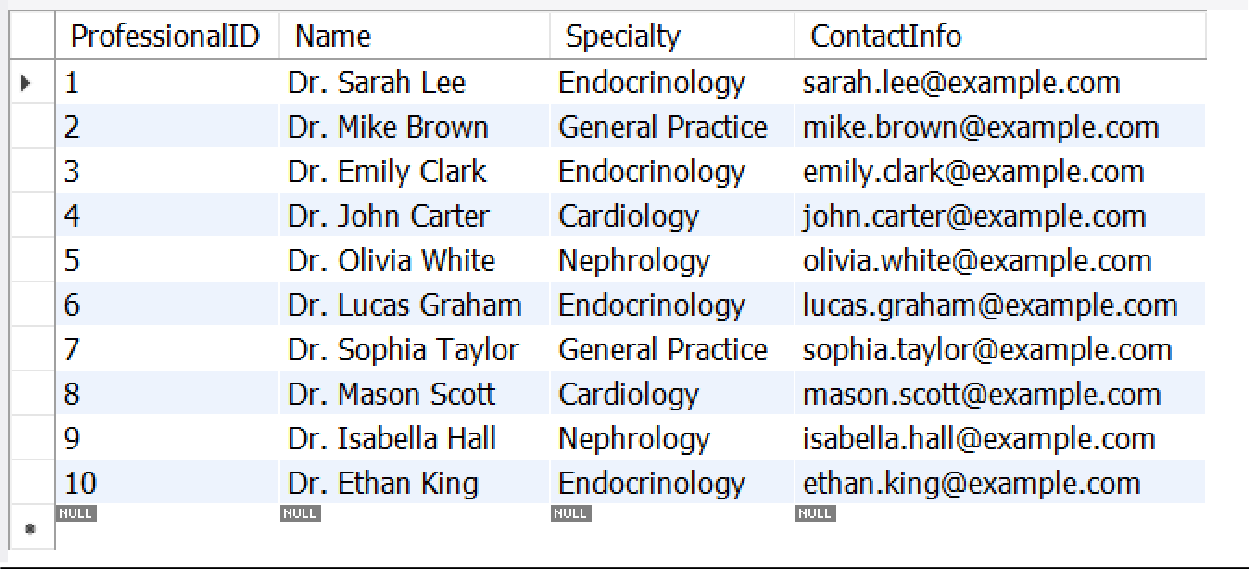
# Tables Present:

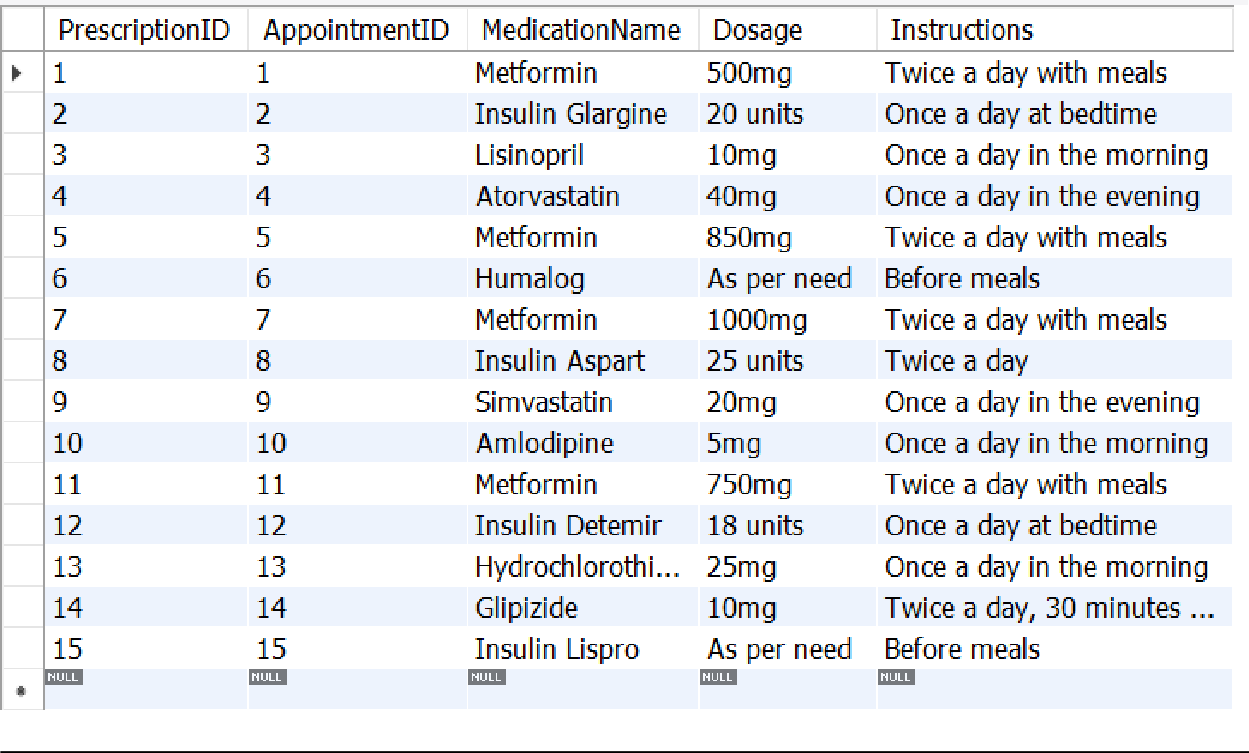
1. Patient Records:

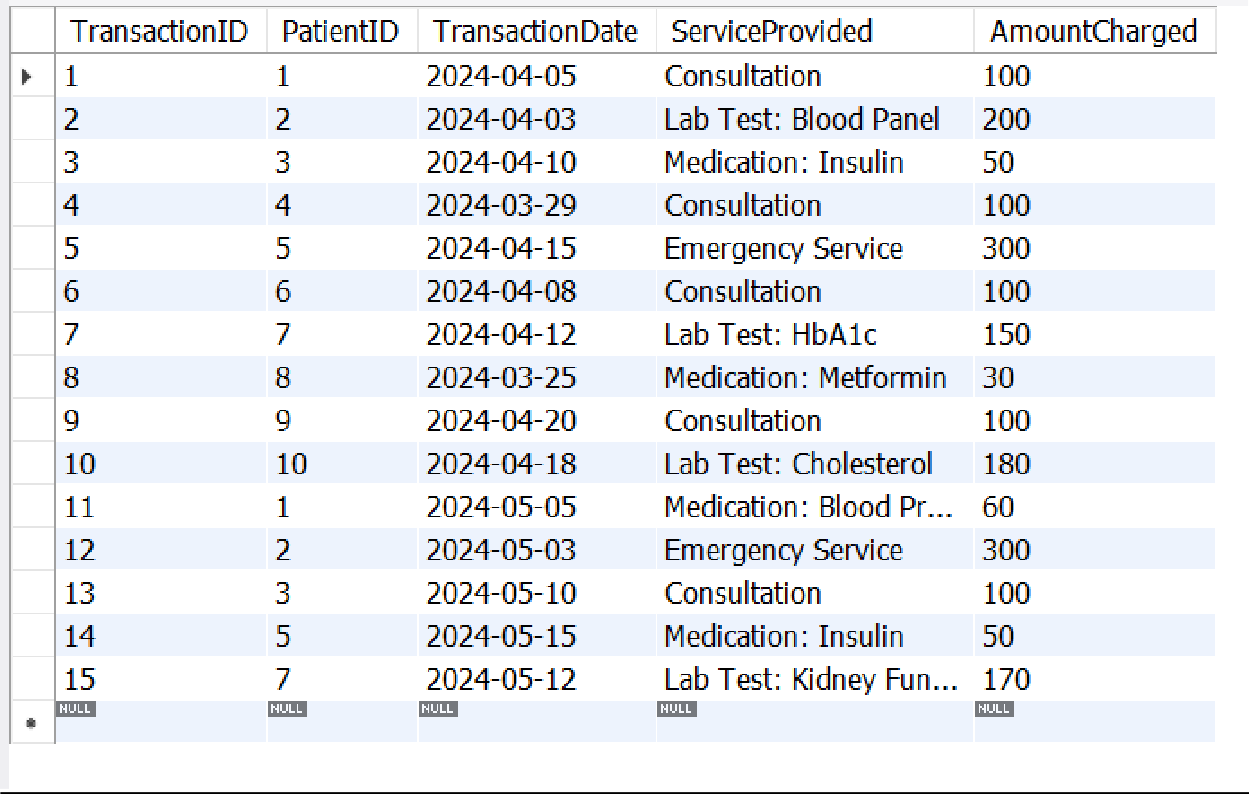


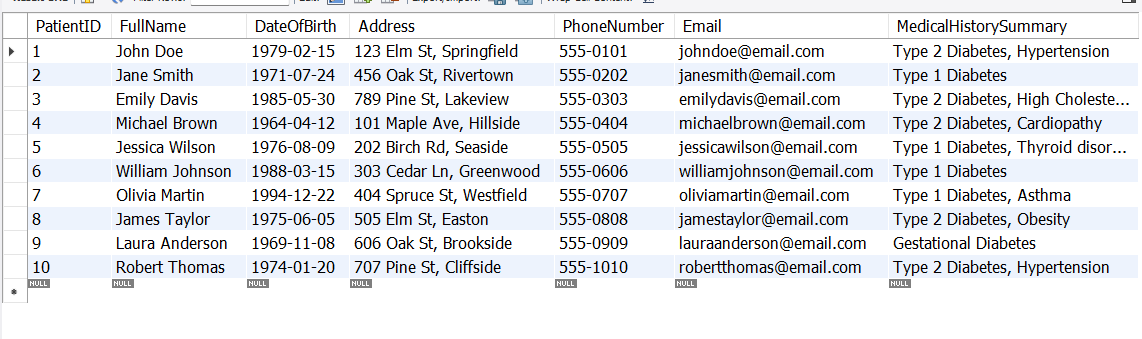
1. Appointment Details:



1. Healthcare Professionals:
2. Medication Prescribed:



1. Transactions:
2. Patients:



**Questions to Answer**

# Appointment and Patient Data

1. Can we see a list of all our patients along with the date of their last appointment?

Code:

SELECT

Patients.PatientID, Patients.FullName,

MAX(AppointmentDetails.AppointmentDate) AS LastAppointmentDate FROM

Patients LEFT JOIN

AppointmentDetails ON Patients.PatientID = AppointmentDetails.PatientID GROUP BY

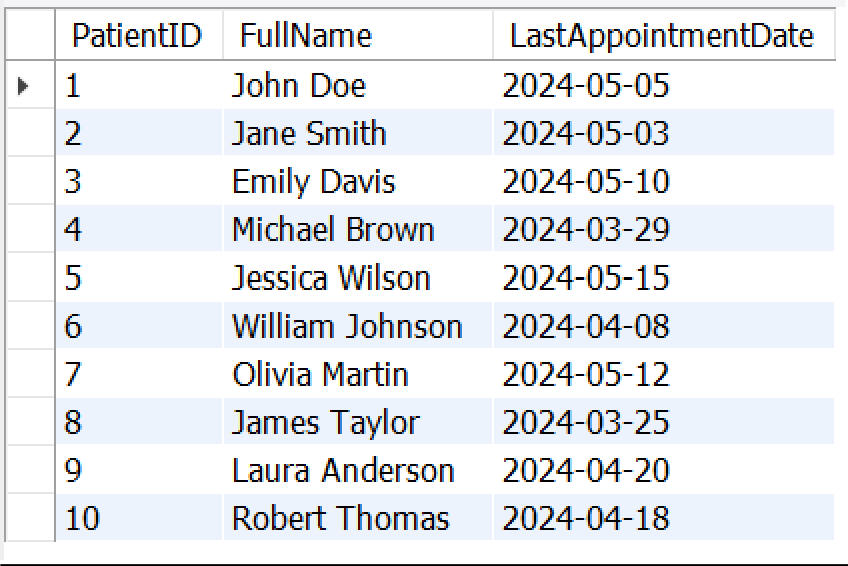
Patients.PatientID, Patients.FullName ORDER BY

Patients.PatientID;

Explanation:

* Patients.PatientID and Patients.FullName: Selects each patient's ID and full name.
* MAX(AppointmentDetails.AppointmentDate) AS LastAppointmentDate: Retrieves the latest appointment date for each patient.
* LEFT JOIN: Includes all patients, even those without any appointments, by joining the Patients table with AppointmentDetails.
* GROUP BY: Groups by each patient's ID and full name to aggregate appointment dates using MAX.
* ORDER BY: Sorts the result by PatientID in ascending order.

Output:



1. What's the total amount we've charged each patient?"

Code:

SELECT

Patients.PatientID, Patients.FullName,

SUM(Transactions.AmountCharged) AS TotalAmountCharged FROM

Patients INNER JOIN

Transactions ON Patients.PatientID = Transactions.PatientID GROUP BY

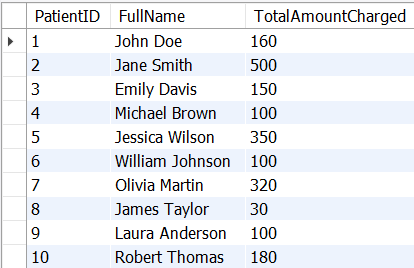
Patients.PatientID, Patients.FullName ORDER BY

Patients.PatientID;

Explanation:

* Patients.PatientID and Patients.FullName: Selects the patient ID and full name for identification.
* SUM(Transactions.AmountCharged) AS TotalAmountCharged: Sums the total charges for each patient.
* INNER JOIN: Ensures that only patients with transactions are included in the result.
* GROUP BY Patients.PatientID, Patients.FullName: Groups results by each patient to apply the SUM function on AmountCharged.
* ORDER BY Patients.PatientID: Sorts the output by PatientID for better readability.
* This query will give you the total amount charged for each patient based on their transaction history.

Output:



1. Which medication do we prescribe the most, and how often?

Code:

SELECT

MedicationName,

COUNT(\*) AS PrescriptionCount FROM

MedicationsPrescribed GROUP BY

MedicationName ORDER BY

PrescriptionCount DESC LIMIT 1;

Explanation:

* MedicationName: Selects each unique medication name.
* COUNT(\*) AS PrescriptionCount: Counts the number of times each medication has been prescribed.
* GROUP BY MedicationName: Groups results by medication to apply the COUNT function.
* ORDER BY PrescriptionCount DESC: Orders results by the highest count first.
* LIMIT 1: Restricts the output to the most frequently prescribed medication.
* This query will return the medication prescribed most often and the total count of its prescriptions.

Output:

A screenshot of a computer  Description automatically generated

1. How do we rank our patients by the number of their appointments?

Code:

SELECT

PatientID, FullName, AppointmentCount,

ROW\_NUMBER() OVER (ORDER BY AppointmentCount DESC) AS PatientRank

FROM (

SELECT

Patients.PatientID, Patients.FullName,

COUNT(AppointmentDetails.AppointmentID) AS AppointmentCount FROM

Patients LEFT JOIN

AppointmentDetails ON Patients.PatientID = AppointmentDetails.PatientID GROUP BY

Patients.PatientID, Patients.FullName

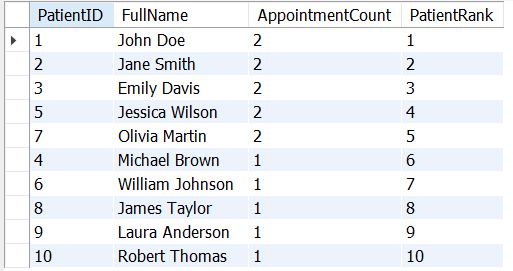
) AS PatientAppointments ORDER BY

PatientRank;

Explanation:

* **Inner query:**
* COUNT(AppointmentDetails.AppointmentID) AS AppointmentCount: Counts the number of appointments for each patient.
* LEFT JOIN: Ensures all patients are included, even those with no appointments.
* GROUP BY Patients.PatientID, Patients.FullName: Groups data by each patient to count their appointments.
* **Outer query:**
* ROW\_NUMBER() OVER (ORDER BY AppointmentCount DESC) AS PatientRank: Assigns a rank based on the number of appointments, with the highest number ranked first.
* ORDER BY PatientRank: Orders the results by the rank in ascending order.

Ouput:



1. Who are our patients that haven't booked any appointments yet?

Code:

SELECT

Patients.PatientID, Patients.FullName, Patients.DateOfBirth, Patients.Address, Patients.PhoneNumber, Patients.Email

FROM

Patients LEFT JOIN

AppointmentDetails ON Patients.PatientID = AppointmentDetails.PatientID WHERE

AppointmentDetails.AppointmentID IS NULL;

Explanation:

## LEFT JOIN:

* + The LEFT JOIN between Patients and AppointmentDetails includes all patients, even if they don’t have corresponding entries in AppointmentDetails.

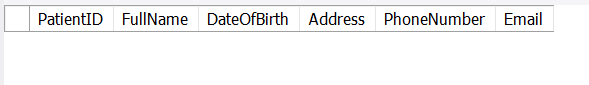
## Filtering with IS NULL:

* + WHERE AppointmentDetails.AppointmentID IS NULL filters for records where the AppointmentID from AppointmentDetails is NULL, meaning the patient has no recorded appointments.

## Result

This query will return a list of patients who have not booked any appointments, including their basic details like PatientID, FullName, DateOfBirth, Address, PhoneNumber, and Email.

Output:



1. Can we track the next appointment date for each patient?

Code:

SELECT

PatientID,

AppointmentDate AS CurrentAppointmentDate, LEAD(AppointmentDate) OVER (PARTITION BY PatientID ORDER BY

AppointmentDate) AS NextAppointmentDate FROM

AppointmentDetails ORDER BY

PatientID, AppointmentDate;

**Explanation:**

1. **LEAD() Function**:
   * LEAD(AppointmentDate) OVER (PARTITION BY PatientID ORDER BY AppointmentDate): This function looks at the next AppointmentDate for each patient within the partition (grouped by PatientID). The ORDER BY AppointmentDate clause within the OVER clause ensures that appointments are ordered chronologically for each patient.

## Columns:

* + AppointmentDate AS CurrentAppointmentDate: Renames the AppointmentDate column to indicate it’s the current appointment date.
  + LEAD(...) AS NextAppointmentDate: Returns the next appointment date for each row, if it exists. If there’s no future appointment, this column will display NULL.

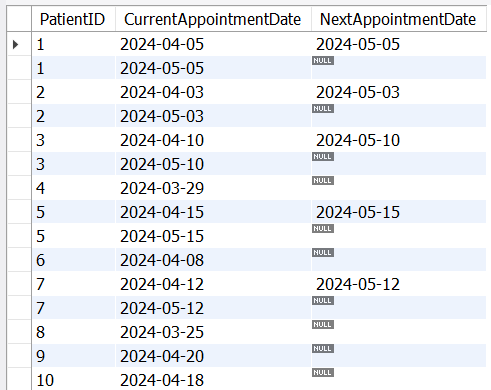
## Ordering:

* + ORDER BY PatientID, AppointmentDate: Orders the final output by PatientID and AppointmentDate, so you can see each patient’s current and next appointment in sequence.

## Result

The query returns a list with each patient’s current appointment date and the date of their next appointment, if scheduled. This structure makes it easy to track when each patient is due for a follow-up or their next scheduled appointment.

Output:



1. Which healthcare professionals haven't seen any patients?

Code:

SELECT

h.ProfessionalID, h.Name

FROM

HealthcareProfessionals h LEFT JOIN

AppointmentDetails a ON h.Name = a.HealthcareProfessional WHERE

a.AppointmentID IS NULL;

Explanation:

## SELECT Clause:

* h.ProfessionalID: This selects the ProfessionalID of the healthcare professional from the HealthcareProfessionals table (aliased as h).
* h.Name: This selects the name of the healthcare professional from the same table.

## FROM Clause:

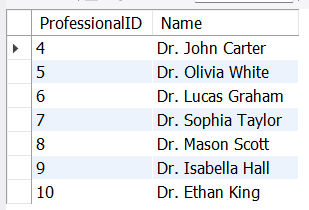
* HealthcareProfessionals h: This specifies that the main table from which to select data is HealthcareProfessionals. The alias h is used to reference this table in the rest of the query.

## LEFT JOIN Clause:

* LEFT JOIN AppointmentDetails a ON h.Name = a.HealthcareProfessional: This performs a left join between the HealthcareProfessionals table and the AppointmentDetails table (aliased as a). The join condition matches the Name column of the HealthcareProfessionals table with the HealthcareProfessional column in the AppointmentDetails table.
* A **left join** means that all records from the HealthcareProfessionals table will be included in the results, regardless of whether there are matching records in the AppointmentDetails table. If there are no matches, the result will include NULL values for columns from the AppointmentDetails table.

## WHERE Clause:

* a.AppointmentID IS NULL: This condition filters the results to include only those rows where there is no corresponding appointment in the AppointmentDetails table. Since a left join is used, if a healthcare professional has no appointments, the AppointmentID from AppointmentDetails will be NULL.

Output:

1. Can we identify patients who had back-to-back appointments within a 30-day period?

Code:

SELECT

a1.PatientID,

a1.AppointmentDate AS CurrentAppointmentDate, a2.AppointmentDate AS NextAppointmentDate

FROM

AppointmentDetails AS a1 JOIN

AppointmentDetails AS a2 ON

a1.PatientID = a2.PatientID

AND a2.AppointmentDate > a1.AppointmentDate

AND a2.AppointmentDate <= DATE\_ADD(a1.AppointmentDate, INTERVAL 30 DAY)

ORDER BY

a1.PatientID, a1.AppointmentDate;

Explanation:

## Self-Join:

* + **The query uses a self-join on the AppointmentDetails table, where a1 refers to the current appointment and a2 refers to the next appointment for the same patient.**

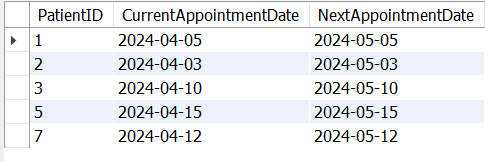
## Join Conditions:

* + **The join condition specifies that we are looking for appointments where a2.AppointmentDate is greater than a1.AppointmentDate and less than or equal to 30 days after a1.AppointmentDate.**

## Ordering:

* + **The results are ordered by PatientID and CurrentAppointmentDate to provide a clear view of back-to-back appointments for each patient.**

## This approach effectively identifies patients with appointments within a 30- day window without using window functions.

Output:

1. What's the average charge per appointment for each healthcare professional?

Code:

SELECT

HealthcareProfessionals.ProfessionalID, HealthcareProfessionals.Name AS HealthcareProfessionalName, AVG(Transactions.AmountCharged) AS AverageChargePerAppointment

FROM

HealthcareProfessionals LEFT JOIN

AppointmentDetails ON HealthcareProfessionals.Name = AppointmentDetails.HealthcareProfessional

LEFT JOIN

Transactions ON AppointmentDetails.AppointmentID = Transactions.TransactionID

GROUP BY

HealthcareProfessionals.ProfessionalID, HealthcareProfessionals.Name;

Explanation:

**SELECT Clause**: This selects the ProfessionalID, the Name of the healthcare professional, and calculates the average charge per appointment.



**FROM Clause**: The main table is HealthcareProfessionals.

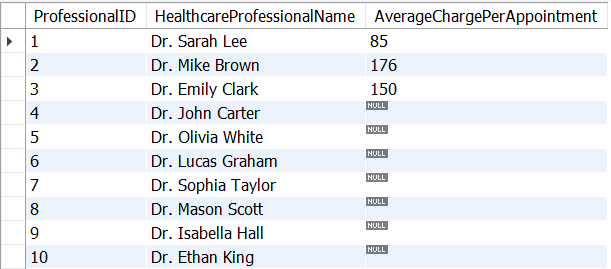
## LEFT JOIN:

* The first LEFT JOIN connects HealthcareProfessionals with AppointmentDetails based on the name of the healthcare professional. This retrieves all appointments for each professional, including those without appointments (resulting in NULL values for those professionals).
* The second LEFT JOIN connects AppointmentDetails to Transactions, matching the AppointmentID from AppointmentDetails to the TransactionID in Transactions. This allows us to access the charge amounts for each appointment.

**GROUP BY Clause**: This groups the results by the healthcare professional's ID and name so that the average charge can be calculated for each.

**AVG Function**: This calculates the average amount charged for appointments for each healthcare professional.

Output:



# Medication and Revenue Analysis

* 1. Who's the last patient each healthcare professional saw, and when?

Code:

WITH LastAppointment AS ( SELECT

HealthcareProfessional, PatientID, AppointmentDate,

DENSE\_RANK() OVER (PARTITION BY HealthcareProfessional ORDER BY AppointmentDate DESC) AS Rank

FROM

AppointmentDetails

) SELECT

HealthcareProfessional, PatientID, AppointmentDate

FROM

LastAppointment WHERE

Rank = 1; Explanation:

* Common **Table Expression (CTE)**:
* The WITH LastAppointment AS (...) creates a Common Table Expression that defines a temporary result set named LastAppointment.
* **Inner Query**:
* **Columns Selected**:
  + HealthcareProfessional: Identifies the healthcare professional.
  + PatientID: Identifies the patient.
  + AppointmentDate: The date of the appointment.

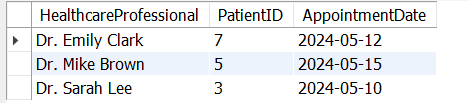
## DENSE\_RANK() Function:

* + This function assigns a rank to each appointment date for each healthcare professional, with the most recent appointment receiving the highest rank
    1. based on the descending order of AppointmentDate.
  + PARTITION BY HealthcareProfessional: The ranking resets for each healthcare professional.

## Outer Query:

* The SELECT statement retrieves HealthcareProfessional, PatientID, and AppointmentDate from the LastAppointment CTE.
* The WHERE Rank = 1 condition filters the results to only include the most recent appointment for each healthcare professional.

Output:



* 1. Which of our patients have been prescribed insulin?

Code :

SELECT

p.FullName, p.MedicalHistorySummary, mp.MedicationName

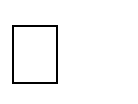
FROM

Patients p JOIN

AppointmentDetails ad ON p.PatientID = ad.PatientID JOIN

MedicationsPrescribed mp ON ad.AppointmentID = mp.AppointmentID WHERE

mp.MedicationName LIKE '%Insulin%';

Explanation: 

* **SELECT Statement**:
* p.FullName: Retrieves the full name of the patient from the Patients table.
* p.MedicalHistorySummary: Retrieves a summary of the patient's medical history from the Patients table.
* mp.MedicationName: Retrieves the name of the medication from the MedicationsPrescribed table.

## FROM Clause:

* Specifies the main table from which to retrieve data, which is the Patients table (aliased as p).

## JOIN Clauses:

* **First JOIN**:
  + JOIN AppointmentDetails ad ON p.PatientID = ad.PatientID: This joins the Patients table with the AppointmentDetails table based on the PatientID. This establishes a link between patients and their appointments.

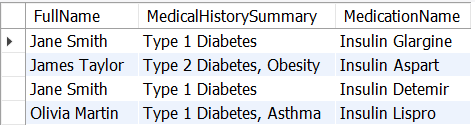
## Second JOIN:

* + JOIN MedicationsPrescribed mp ON ad.AppointmentID = mp.AppointmentID: This joins the AppointmentDetails table with the MedicationsPrescribed table based on the AppointmentID. This connects the appointments to the medications prescribed during those appointments.

## WHERE Clause:

* mp.MedicationName LIKE '%Insulin%': This filters the results to include only those records where the medication name contains the term "Insulin". The use of LIKE allows for flexible matching, accommodating variations in how insulin might be recorded (e.g., "Insulin", "Insulin Glargine").

Output:



* 1. How can we calculate the total amount charged and the number of appointments for each patient?"

Code:

SELECT

p.FullName,

COUNT(t.TransactionID) AS TotalAppointments, SUM(t.AmountCharged) AS TotalAmountCharged

FROM

Patients p LEFT JOIN

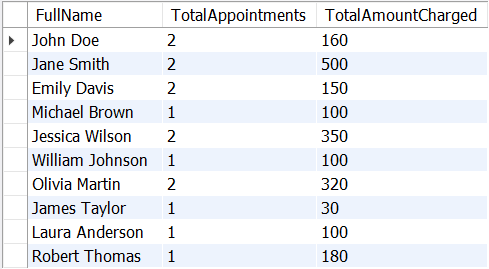
Transactions t ON p.PatientID = t.PatientID GROUP BY

p.FullName;

Explanation :

This SQL query retrieves a summary of patients' appointment information from the Patients and Transactions tables. By using a **LEFT JOIN**, it ensures that all patients are included in the results, even those without any transactions. The query selects each patient's full name and calculates two key metrics: the total number of appointments (TotalAppointments), represented by counting the TransactionID entries, and the total amount charged (TotalAmountCharged), calculated by summing the AmountCharged for each patient. The results are grouped by the patient's full name, allowing for a comprehensive view of appointment counts and charges for each individual, ensuring that patients with no appointments show up with zero counts and charges.

Output:



* 1. Can we rank our healthcare professionals by the number of unique patients they've seen?

Code:

WITH ProfessionalPatientCount AS ( SELECT

AppointmentDetails.HealthcareProfessional,

COUNT(DISTINCT AppointmentDetails.PatientID) AS UniquePatients FROM

AppointmentDetails GROUP BY

AppointmentDetails.HealthcareProfessional

) SELECT

HealthcareProfessional, UniquePatients,

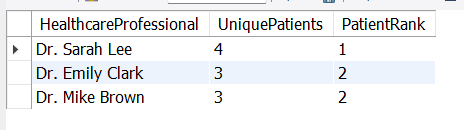
RANK() OVER (ORDER BY UniquePatients DESC) AS Rank FROM

ProfessionalPatientCount;

Explanation:

The provided SQL query uses a Common Table Expression (CTE) named ProfessionalPatientCount to calculate the number of unique patients each healthcare professional has seen by aggregating data from the AppointmentDetails table. Within this CTE, COUNT(DISTINCT AppointmentDetails.PatientID) is employed to ensure that each patient is counted only once per healthcare professional, effectively providing a tally of unique patients. The main query then selects each healthcare professional's identifier along with the count of unique patients, applying the RANK() function to assign

ranks based on the number of unique patients in descending order. This structure allows for a clear ranking of healthcare professionals, highlighting those with the highest patient interaction.

Output:

# Advanced Analysis with Subqueries and CTEs

1. How does each patient's appointment count compare to the clinic's average?

Code:

SELECT

Patients.PatientID, Patients.FullName,

COUNT(AppointmentDetails.AppointmentID) AS PatientAppointmentCount, (SELECT AVG(AppointmentCount)

FROM (SELECT COUNT(AppointmentID) AS AppointmentCount FROM AppointmentDetails

GROUP BY PatientID) AS PatientCounts) AS ClinicAverageAppointmentCount,

CASE

WHEN COUNT(AppointmentDetails.AppointmentID) > (SELECT AVG(AppointmentCount)

FROM (SELECT COUNT(AppointmentID) AS AppointmentCount FROM AppointmentDetails

GROUP BY PatientID) AS PatientCounts) THEN 'Above Average' WHEN COUNT(AppointmentDetails.AppointmentID) <

(SELECT AVG(AppointmentCount)

FROM (SELECT COUNT(AppointmentID) AS AppointmentCount FROM AppointmentDetails

GROUP BY PatientID) AS PatientCounts) THEN 'Below Average' ELSE 'At Par'

END AS AppointmentComparison FROM

Patients LEFT JOIN

AppointmentDetails ON Patients.PatientID = AppointmentDetails.PatientID GROUP BY

Patients.PatientID, Patients.FullName;

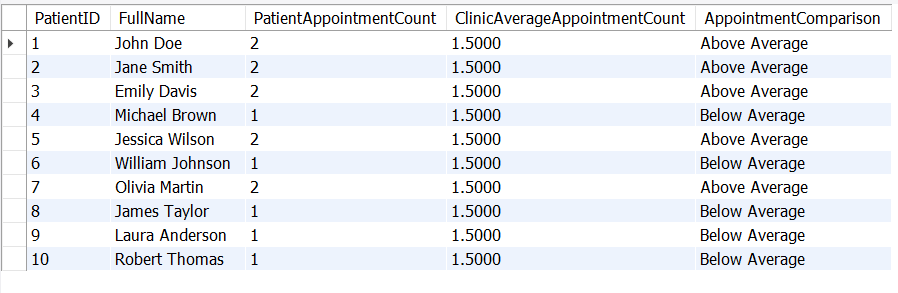
Explanation:

This query calculates each patient's total number of appointments, compares it with the clinic's average appointment count, and then categorizes each patient's count as "Above Average," "Below Average," or "At Par." The main structure involves calculating the average appointment count for the entire clinic within a subquery and using that average directly within the main query.

The query begins by selecting each patient’s ID and full name from the Patients table. It then calculates the PatientAppointmentCount using the COUNT function on AppointmentDetails.AppointmentID, joining AppointmentDetails to each Patients entry. To calculate the clinic-wide average appointment count, an inner subquery is used: it groups AppointmentDetails by each PatientID, counts the appointments per patient, and calculates the average from these counts. This calculated average, named ClinicAverageAppointmentCount, serves as the comparison point in the CASE statement.

In the CASE statement, each patient's appointment count is checked against the clinic average: if it is greater, it’s labeled as "Above Average"; if it is less, it’s labeled as "Below Average"; otherwise, it’s marked as "At Par." This single query approach allows for efficient categorization of each patient’s appointment activity in relation to the clinic's average, all within one structured query.

Output:



1. For patients without transactions, can we ensure their total charged amount shows up as zero instead of NULL?

Code:

SELECT

Patients.PatientID, Patients.FullName,

COALESCE(SUM(Transactions.AmountCharged), 0) AS TotalAmountCharged FROM

Patients LEFT JOIN

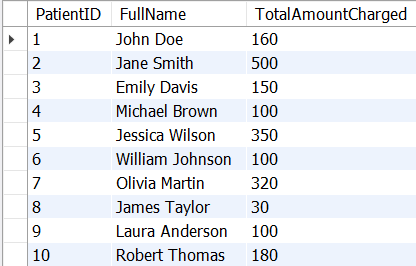
Transactions ON Patients.PatientID = Transactions.PatientID GROUP BY

Patients.PatientID, Patients.FullName;

Explanation:

* This query uses a LEFT JOIN to include all patients from the Patients table, even if they have no associated transactions in the Transactions table.
* The SUM(Transactions.AmountCharged) calculates the total amount charged for each patient.
* COALESCE(SUM(Transactions.AmountCharged), 0) replaces any NULL result from SUM with 0 for patients who have no transaction records.
* The result is grouped by PatientID and FullName to give a single row per patient, showing the total amount charged for each, with patients without transactions correctly showing a TotalAmountCharged of 0.

Output:



1. What's the most common medication for each type of diabetes we treat?

Code:

WITH MedicationCounts AS ( SELECT

pr.DiabetesType, mp.MedicationName,

COUNT(mp.MedicationName) AS PrescriptionCount FROM

PatientRecords pr JOIN

AppointmentDetails ad ON pr.PatientID = ad.PatientID JOIN

MedicationsPrescribed mp ON ad.AppointmentID = mp.AppointmentID GROUP BY

pr.DiabetesType, mp.MedicationName

),

MaxCount AS ( SELECT

DiabetesType,

MAX(PrescriptionCount) AS MaxPrescriptionCount FROM

MedicationCounts GROUP BY

DiabetesType

) SELECT

mc.DiabetesType, mc.MedicationName, mc.PrescriptionCount

FROM

MedicationCounts mc JOIN

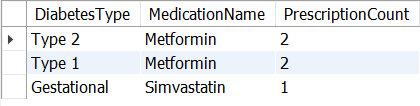
MaxCount m ON mc.DiabetesType = m.DiabetesType

AND mc.PrescriptionCount = m.MaxPrescriptionCount;

Explanation:

* MedicationCounts CTE: This part groups the data by DiabetesType and MedicationName to calculate how often each medication has been prescribed for each diabetes type.
* MaxCount CTE: We calculate the maximum prescription count for each DiabetesType.
* Final Query: By joining MedicationCounts and MaxCount on both DiabetesType and PrescriptionCount, we retrieve only those medications that have the highest prescription count per diabetes type.
* This query gives a result showing the most commonly prescribed medication for each type of diabetes in the dataset.

Output:



1. Can we see the growth in appointment numbers from month to month?

Code:

WITH MonthlyAppointments AS ( SELECT

DATE\_FORMAT(AppointmentDate, '%Y-%m') AS AppointmentMonth, COUNT(AppointmentID) AS AppointmentCount

FROM

AppointmentDetails GROUP BY

AppointmentMonth ORDER BY

AppointmentMonth

) SELECT

AppointmentMonth, AppointmentCount,

AppointmentCount - LAG(AppointmentCount, 1) OVER (ORDER BY AppointmentMonth) AS GrowthFromPreviousMonth

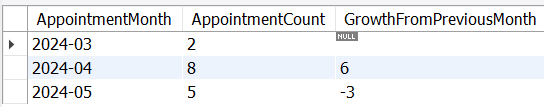
FROM

MonthlyAppointments;

Explanation:

* Date Extraction: Each variant uses the appropriate function to extract the year and month from the AppointmentDate in a format suitable for grouping.
* Counting Appointments: The query counts the total appointments for each month.
* Calculating Growth: The LAG() function is used to determine the change in appointment counts compared to the previous month.

Output:



1. How do healthcare professionals' appointments and revenue compare?

Code:

WITH AppointmentCounts AS ( SELECT

AppointmentDetails.HealthcareProfessional, COUNT(AppointmentID) AS TotalAppointments

FROM

AppointmentDetails GROUP BY

AppointmentDetails.HealthcareProfessional

),

RevenueTotals AS ( SELECT

AppointmentDetails.HealthcareProfessional, SUM(Transactions.AmountCharged) AS TotalRevenue

FROM

Transactions JOIN

AppointmentDetails ON Transactions.PatientID = AppointmentDetails.PatientID

GROUP BY

AppointmentDetails.HealthcareProfessional

) SELECT

hp.Name AS HealthcareProfessional,

COALESCE(ac.TotalAppointments, 0) AS TotalAppointments, COALESCE(rt.TotalRevenue, 0) AS TotalRevenue

FROM

HealthcareProfessionals hp LEFT JOIN

AppointmentCounts ac ON hp.Name = ac.HealthcareProfessional LEFT JOIN

RevenueTotals rt ON hp.Name = rt.HealthcareProfessional;

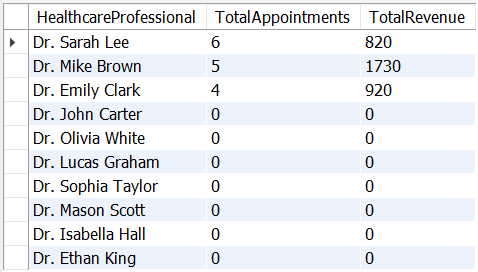
Explanation:

* • CTE: AppointmentCounts: This remains the same, counting the number of appointments for each healthcare professional.
* • CTE: RevenueTotals: This also remains the same, summing the revenue generated by each healthcare professional.

### • Final SELECT Statement:

* The main query now selects all healthcare professionals from the HealthcareProfessionals table (hp).
* It uses LEFT JOIN to join both the AppointmentCounts and RevenueTotals, ensuring that every healthcare professional is included in the results, even those without appointments or revenue.
* The COALESCE function is used to replace any NULL values for appointments and revenue with 0.

Output:



1. Which medications have seen a change in their prescribing rank from month to month?

Code:

WITH MonthlyMedicationRanks AS ( SELECT

DATE\_FORMAT(ad.AppointmentDate, '%Y-%m') AS Month, mp.MedicationName,

COUNT(mp.PrescriptionID) AS PrescriptionCount,

RANK() OVER (PARTITION BY DATE\_FORMAT(ad.AppointmentDate, '%Y-

%m')

ORDER BY COUNT(mp.PrescriptionID) DESC) AS MedicationRank

FROM

MedicationsPrescribed mp JOIN

AppointmentDetails ad ON mp.AppointmentID = ad.AppointmentID GROUP BY

Month, mp.MedicationName

)

SELECT

Current.Month, Current.MedicationName, Current.PrescriptionCount,

Current.MedicationRank AS CurrentRank, Previous.MedicationRank AS PreviousRank

FROM

MonthlyMedicationRanks AS Current LEFT JOIN

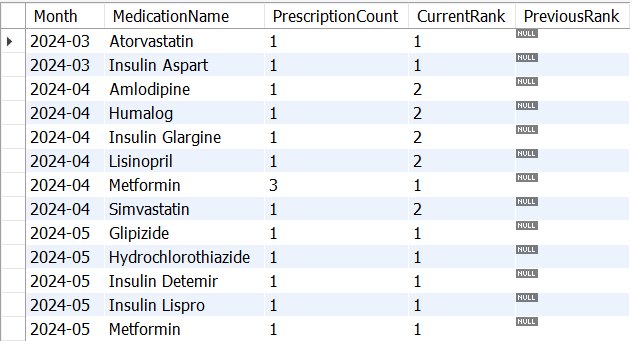
MonthlyMedicationRanks AS Previous ON

Current.MedicationName = Previous.MedicationName AND Current.Month =

DATE\_FORMAT(DATE\_SUB(STR\_TO\_DATE(CONCAT(Current.Month, '-01'), '%Y-%m-%d'), INTERVAL 1 MONTH), '%Y-%m')

ORDER BY

Current.Month, Current.MedicationName; Output:



1. Can we identify our top 3 most expensive services for each patient?

Code:

WITH ServiceRankings AS ( SELECT

t.PatientID,

p.FullName AS PatientName, t.ServiceProvided, t.AmountCharged,

DENSE\_RANK() OVER (PARTITION BY t.PatientID ORDER BY

t.AmountCharged DESC) AS ServiceRank FROM

Transactions t JOIN

Patients p ON t.PatientID = p.PatientID

),

NumberSeries AS (

SELECT 1 AS ServiceRank UNION ALL

SELECT 2 UNION ALL SELECT 3

),

CompleteServices AS ( SELECT

ps.PatientID, ps.PatientName, ns.ServiceRank, rs.ServiceProvided,

rs.AmountCharged FROM

(SELECT DISTINCT PatientID, PatientName FROM ServiceRankings) ps CROSS JOIN

NumberSeries ns LEFT JOIN

ServiceRankings rs ON ps.PatientID = rs.PatientID AND ns.ServiceRank = rs.ServiceRank

) SELECT

PatientID, PatientName,

COALESCE(ServiceProvided, 'No Service') AS ServiceProvided, COALESCE(AmountCharged, 0) AS AmountCharged, ServiceRank

FROM

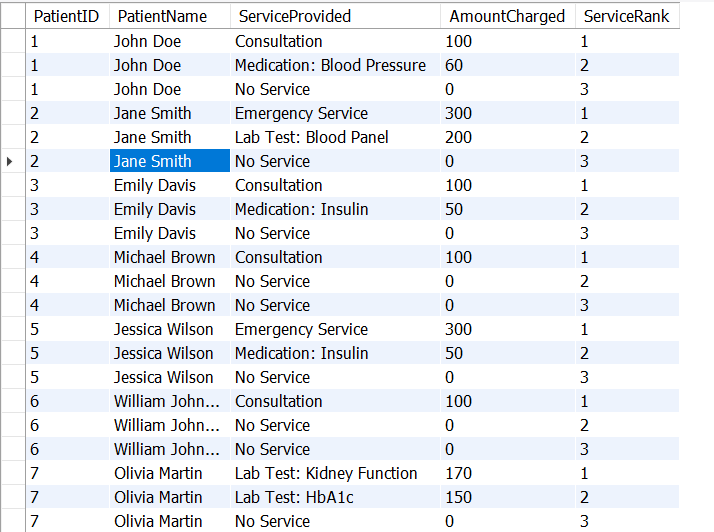
CompleteServices ORDER BY

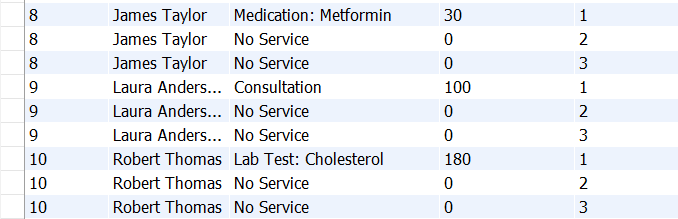
PatientID, ServiceRank;

Explanation:

The SQL query utilizes common table expressions (CTEs) to ensure that each patient has exactly three service entries in the output. It starts with the ServiceRankings CTE, which ranks the services provided to each patient based on the amount charged, using the DENSE\_RANK() function. The NumberSeries CTE generates a simple table with three rows, representing the service ranks (1, 2, and 3). The CompleteServices CTE combines unique patients with the number series using a CROSS JOIN, creating a Cartesian product that ensures each patient is paired with all three possible service ranks. A LEFT JOIN then connects back to the ServiceRankings CTE to pull in the relevant service details, filling in any gaps where a patient has fewer than three services. The final SELECT statement retrieves the patient information, replacing any NULL values for ServiceProvided and AmountCharged with default values ('No Service' and 0, respectively). This structure guarantees that the output displays exactly three rows for each patient, making it easier to analyze service provision consistency across the patient population.

Output:





1. Who is our most frequently seen patient in terms of prescriptions, and what medications have they been prescribed?

Code:

WITH PrescriptionCounts AS ( SELECT

pr.PatientID, pr.Name,

COUNT(mp.PrescriptionID) AS PrescriptionCount

FROM

PatientRecords pr JOIN

AppointmentDetails ad ON pr.PatientID = ad.PatientID JOIN

MedicationsPrescribed mp ON ad.AppointmentID = mp.AppointmentID GROUP BY

pr.PatientID, pr.Name

),

MostFrequentPatient AS ( SELECT

PatientID, Name

FROM

PrescriptionCounts ORDER BY

PrescriptionCount DESC LIMIT 1

) SELECT

mp.Name, mp.PatientID, pc.PrescriptionCount, mpd.MedicationName, mpd.Dosage, mpd.Instructions

FROM

MostFrequentPatient mp JOIN

PrescriptionCounts pc ON mp.PatientID = pc.PatientID JOIN

AppointmentDetails ad ON mp.PatientID = ad.PatientID JOIN

MedicationsPrescribed mpd ON ad.AppointmentID = mpd.AppointmentID; Explanation:

### Common Table Expressions (CTEs):

* **PrescriptionCounts**:
* Counts the total number of prescriptions per patient.
* Joins PatientRecords, AppointmentDetails, and MedicationsPrescribed tables.
* Groups results by PatientID and PatientName.

### MostFrequentPatient:

* Identifies the patient with the highest count of prescriptions from the PrescriptionCounts CTE.
* Uses ORDER BY PrescriptionCount DESC to sort and LIMIT 1 to select only the top patient.

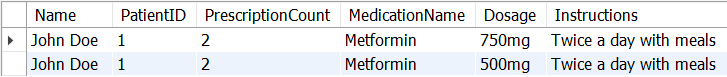
### Final Query:

* Retrieves details of the most frequently seen patient.
* Joins MostFrequentPatient with MedicationsPrescribed to list all medications prescribed to that patient.
* Outputs columns: PatientName, PatientID, PrescriptionCount, MedicationName, Dosage, and Instructions.

### Output:

* Displays the most frequently seen patient, John Doe (PatientID: 1), with a total of **2 prescriptions**.
* Lists the medications prescribed to him:
* **Metformin**: 500mg, taken twice a day with meals.
* **Metformin**: 750mg, taken twice a day with meals.

Output:



1. How does our monthly revenue compare to the previous month?

Code:

WITH MonthlyRevenue AS ( SELECT

DATE\_FORMAT(TransactionDate, '%Y-%m') AS RevenueMonth, -- Format date to year-month

SUM(AmountCharged) AS TotalRevenue -- Sum up revenue for each month

FROM

Transactions GROUP BY

RevenueMonth -- Group by month

),

RevenueComparison AS ( SELECT

RevenueMonth, TotalRevenue,

LAG(TotalRevenue) OVER (ORDER BY RevenueMonth) AS PreviousMonthRevenue -- Get revenue from the previous month

FROM

MonthlyRevenue

)

SELECT

RevenueMonth, TotalRevenue, PreviousMonthRevenue,

TotalRevenue - PreviousMonthRevenue AS RevenueChange, -- Calculate change in revenue

CASE

WHEN PreviousMonthRevenue IS NULL THEN NULL -- Handle cases with no previous month

WHEN TotalRevenue - PreviousMonthRevenue > 0 THEN 'Increased' WHEN TotalRevenue - PreviousMonthRevenue < 0 THEN 'Decreased' ELSE 'No Change'

END AS RevenueStatus FROM

RevenueComparison;

Explanation:

### Common Table Expression (CTE) - MonthlyRevenue:

* + - **Purpose**: This temporary result set calculates total revenue for each month.

### Components:

* + - strftime('%Y-%m', TransactionDate): This function formats the TransactionDate to the 'YYYY-MM' format, grouping all transactions by month and year.
    - SUM(AmountCharged): This aggregates the total revenue generated in that month by summing up the AmountCharged for each transaction.
    - FROM Transactions: Indicates that the data is sourced from the Transactions table.
    - GROUP BY RevenueMonth: This clause groups the results by the computed RevenueMonth, ensuring that revenue is calculated per month.

### Main Query:

* + - SELECT Statement: This part retrieves data from the MonthlyRevenue CTE.
    - Columns Selected:
    - mr.RevenueMonth: Displays the month and year for the total revenue calculated.
    - mr.TotalRevenue: Shows the total revenue for that month as calculated in the CTE.

### LAG Function:

* + - LAG(mr.TotalRevenue) OVER (ORDER BY mr.RevenueMonth): This function fetches the total revenue of the previous month based on the ordering of RevenueMonth.
    - This allows for comparing the current month's revenue against the previous month's revenue.

Output:

### Revenue Change Calculation:

* mr.TotalRevenue - LAG(mr.TotalRevenue) OVER (ORDER BY mr.RevenueMonth): This expression calculates the difference in revenue between the current and previous month, providing insight into whether revenue increased or decreased.

### Revenue Status:

* The CASE statement evaluates the RevenueChange:
* If the RevenueChange is greater than 0, it returns 'Increased'.
* If less than 0, it returns 'Decreased'.
* If equal to 0, it returns 'No Change'.
* This categorization helps in quickly assessing the trend of revenue over the months.

### Ordering the Results:

* ORDER BY mr.RevenueMonth: This ensures that the results are sorted by month in ascending order, making it easier to follow the monthly trends.

