DATABRICKS Data Analyst SQL Interview Questions
0-3 YOE
12+ LPA

Question 1: Cumulative Revenue per Day

Concept: This question tests your ability to use window functions, specifically SUM() OVER(), to calculate running totals.

Input Table: Sales

sale_date	revenue
2025-01-01	100
2025-01-01	50
2025-01-02	75
2025-01-03	120
2025-01-03	30
2025-01-04	90

Required SQL Query:

```
SELECT
  sale_date,
  SUM(daily_revenue) OVER (ORDER BY sale_date) AS cumulative_revenue
FROM (
  SELECT
    sale_date,
    SUM(revenue) AS daily_revenue
  FROM
    Sales
  GROUP BY
    sale_date
) AS daily_sales
ORDER BY
  sale_date;
```

Explanation:

1. **Inner Query (daily_sales):**
   - SELECT sale_date, SUM(revenue) AS daily_revenue FROM Sales GROUP BY sale_date
   - This subquery first calculates the total revenue for each sale_date.
2. **Outer Query:**
   - SUM(daily_revenue) OVER (ORDER BY sale_date): This is the window function. It calculates a running sum of daily_revenue.
     - OVER (ORDER BY sale_date) specifies that the sum should be calculated based on the sale_date in ascending order. For each row, it sums all daily_revenue values from the beginning up to and including the current sale_date.

Output Table:

sale_date	cumulative_revenue
2025-01-01	150
2025-01-02	225
2025-01-03	375
2025-01-04	465

Question 2: First and Last Login Per User

Concept: This tests your ability to find minimum and maximum values grouped by an identifier, often using GROUP BY with aggregate functions, or sometimes window functions for more complex scenarios.

Input Table: UserLogins

user_id	login_timestamp
101	2025-03-01 08:00:00
102	2025-03-01 09:15:00
101	2025-03-01 10:30:00
103	2025-03-02 11:00:00
102	2025-03-02 14:45:00
101	2025-03-03 07:00:00
103	2025-03-03 16:30:00

Required SQL Query:

```
SELECT
  user_id,
  MIN(login_timestamp) AS first_login,
  MAX(login_timestamp) AS last_login
```

```
FROM
  UserLogins
GROUP BY
  user_id
ORDER BY
  user_id;
```

Explanation:

- MIN(login_timestamp): Finds the earliest login_timestamp for each group.
- MAX(login_timestamp): Finds the latest login_timestamp for each group.
- GROUP BY user_id: Groups the rows so that MIN and MAX are applied separately for each unique user_id.

Output Table:

user_id	first_login	last_login
101	2025-03-01 08:00:00	2025-03-03 07:00:00
102	2025-03-01 09:15:00	2025-03-02 14:45:00
103	2025-03-02 11:00:00	2025-03-03 16:30:00

Question 3: Top 3 Customers by Spend in Each Region

Concept: This is a classic "top N per group" problem, best solved using window functions, specifically ROW_NUMBER(), RANK(), or DENSE_RANK() with PARTITION BY.

Input Table: Orders

order_id	customer_id	region	spend
1	101	North	150
2	102	South	200

3	103	North	100
4	101	North	50
5	104	West	300
6	102	South	75
7	105	North	250
8	106	West	180
9	107	South	120
10	103	North	80
11	108	East	400
12	109	East	150
13	110	East	220
14	111	East	90

Required SQL Query:

```
WITH CustomerTotalSpend AS (
  SELECT
    region,
    customer_id,
    SUM(spend) AS total_spend
  FROM
    Orders
  GROUP BY
    region,
    customer_id
),
RankedCustomers AS (
    SELECT
        region,
        customer_id,
        total_spend,
        ROW_NUMBER() OVER (PARTITION BY region ORDER BY total_spend DESC) AS rn
    FROM
        CustomerTotalSpend
)
SELECT
    region,
    customer_id,
    total_spend
FROM
    RankedCustomers
WHERE
    rn <= 3
ORDER BY
    region,
    total_spend DESC;
```

Explanation:

1. **CustomerTotalSpend CTE (Common Table Expression):**
   - SELECT region, customer_id, SUM(spend) AS total_spend FROM Orders GROUP BY region, customer_id
   - This CTE first calculates the total spend for each customer_id within each region.
2. **RankedCustomers CTE:**

- ROW_NUMBER() OVER (PARTITION BY region ORDER BY total_spend DESC) AS rn
- This is the core of the solution.
    - PARTITION BY region: Divides the data into separate partitions for each region. The ranking will reset for each new region.
    - ORDER BY total_spend DESC: Within each region, customers are ordered by their total_spend in descending order (highest spend first).
    - ROW_NUMBER(): Assigns a unique, sequential rank (1, 2, 3, ...) to each row within its partition. If two customers have the same total_spend in a region, ROW_NUMBER() will assign them different, arbitrary ranks. (If you need to handle ties differently, consider RANK() or DENSE_RANK()).

3. **Final SELECT Statement:**
    - SELECT region, customer_id, total_spend FROM RankedCustomers WHERE rn <= 3: Filters the results to include only those customers whose rn (rank) is 3 or less within their respective regions.

Output Table:

region	customer_id	total_spend
East	108	400
East	110	220
East	109	150
North	105	250
North	101	200
North	103	180
South	102	275
South	107	120
West	104	300
West	106	180

Question 4: Detect and Delete Duplicate Transactions

Concept: This question tests your ability to identify duplicate rows based on a set of columns and then use various strategies to remove them, typically keeping one unique record. The "delete" part implies either physically removing rows or selecting only unique ones for further processing.

Input Table: Transactions

Assume a composite primary key or a unique transaction is defined by transaction_id (though often, duplicates arise when multiple columns combined should be unique, e.g., transaction_date, user_id, amount). For this example, let's assume transaction_id is the unique identifier, but due to some ETL issue, duplicates exist.

transaction_id	transaction_date	user_id	amount	
T001	2025-05-01	101	100	
T002	2025-05-01	102	150	
T001	2025-05-01	101	100	-- Duplicate of T001
T003	2025-05-02	103	200	
T002	2025-05-01	102	150	-- Duplicate of T002
T004	2025-05-02	101	50	
T001	2025-05-01	101	100	-- Another duplicate of T001

Required SQL Query (Detecting Duplicates):

To simply detect duplicates (show all instances of duplicated transactions):

```
SELECT
  transaction_id,
  transaction_date,
  user_id,
  amount,
  COUNT(*) AS duplicate_count
FROM
  Transactions
GROUP BY
  transaction_id,
  transaction_date,
  user_id,
  amount  -- Group by all columns that define a unique transaction
HAVING
  COUNT(*) > 1;
```

Output Table (Detecting Duplicates):

transaction_id	transaction_date	user_id	amount	duplicate_count
T001	2025-05-01	101	100	3
T002	2025-05-01	102	150	2

Required SQL Query (Deleting Duplicates - keeping one instance):

There are multiple ways to delete duplicates. Here's a common and efficient method using a CTE with ROW_NUMBER(). This approach is safe as it doesn't delete *all* duplicates, only the "extra" ones.

SQL

```
-- For Databricks/Spark SQL, you often create a new table/view with distinct records
-- or use DELETE if it's a Delta table (which supports MERGE/DELETE).


-- Option 1: Create a new table/view with distinct records (common in data pipelines)
CREATE OR REPLACE TABLE Transactions_Deduped AS
```

```
WITH RankedTransactions AS (
  SELECT
    transaction_id,
    transaction_date,
    user_id,
    amount,
    ROW_NUMBER() OVER (PARTITION BY transaction_id, transaction_date, user_id,
amount ORDER BY transaction_date) AS rn
    -- The ORDER BY in ROW_NUMBER() is arbitrary if all duplicate columns are identical.
    -- If there are subtle differences (e.g., different load timestamps), you might order by
that.
  FROM
    Transactions
)
SELECT
  transaction_id,
  transaction_date,
  user_id,
  amount
FROM
  RankedTransactions
WHERE
  rn = 1;


-- Option 2: Directly DELETE duplicates from a Delta table (more advanced, requires Delta
Lake)
-- This approach is more direct for modifying an existing table in Databricks.
-- ONLY RUN IF YOU UNDERSTAND THE IMPLICATIONS OF DELETING DATA!
/*
DELETE FROM Transactions
WHERE (transaction_id, transaction_date, user_id, amount) IN (
  SELECT
    t.transaction_id, t.transaction_date, t.user_id, t.amount
  FROM (
    SELECT
      transaction_id,
      transaction_date,
      user_id,
      amount,
      ROW_NUMBER() OVER (PARTITION BY transaction_id, transaction_date, user_id, amount ORDER BY transaction_date) AS rn
    FROM
      Transactions
  ) AS t
  WHERE t.rn > 1
);
*/

-- For MySQL, a common way to delete duplicates while keeping one (if you have an auto-incrementing ID for example):
-- DELETE t1 FROM Transactions t1, Transactions t2 WHERE t1.transaction_id = t2.transaction_id AND t1.row_id > t2.row_id;
-- This assumes a unique row identifier exists. Without one, the ROW_NUMBER() approach is safer.
```

Output Table (After Deletion/Creation of Deduped Table):

transaction_id	transaction_date	user_id	amount
T001	2025-05-01	101	100
T002	2025-05-01	102	150
T003	2025-05-02	103	200
T004	2025-05-02	101	50

Question 5: Users with Purchases in 3 Consecutive Months

Concept: This question requires identifying patterns over time. It typically involves date functions (extracting year/month), self-joins, or more elegantly, window functions (LAG, LEAD) combined with date arithmetic.

Input Table: Purchases

purchase_id	user_id	purchase_date	amount
P001	101	2025-01-15	25
P002	102	2025-01-20	30
P003	101	2025-02-10	40
P004	103	2025-02-25	50
P005	101	2025-03-05	60
P006	102	2025-03-18	35
P007	104	2025-03-22	100
P008	101	2025-04-01	70
P009	103	2025-04-10	55
P010	102	2025-05-01	45

| P011 | 101 | 2025-06-01 | 80 |

Required SQL Query:

This problem is best solved by first identifying the unique months each user made a purchase and then checking for a sequence of 3.

```
WITH UserMonthlyPurchases AS (
  SELECT DISTINCT
    user_id,
    DATE_TRUNC('month', purchase_date) AS purchase_month -- Extracts the first day of
the month
  FROM
    Purchases
),
RankedMonths AS (
  SELECT
    user_id,
    purchase_month,
    ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY purchase_month) AS rn
  FROM
    UserMonthlyPurchases
),
ConsecutiveMonths AS (
  SELECT
    user_id,
    purchase_month,
    -- Calculate the difference in months from a fixed point (e.g., year * 12 + month_num)
    -- and subtract the row number. If a user has consecutive months, this 'group_id' will be
constant.
    EXTRACT(YEAR FROM purchase_month) * 12 + EXTRACT(MONTH FROM
purchase_month) - rn AS month_group_id
  FROM
    RankedMonths
)
SELECT DISTINCT
  user_id
FROM
  ConsecutiveMonths
GROUP BY
  user_id,
  month_group_id
HAVING
  COUNT(*) >= 3;
```

Explanation:

1. **UserMonthlyPurchases CTE:**
   - DATE_TRUNC('month', purchase_date): This function (common in PostgreSQL, Databricks, Spark SQL) truncates the date to the beginning of its month. DISTINCT ensures we only get one entry per user per month they purchased.
2. **RankedMonths CTE:**
   - ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY purchase_month) AS rn: Assigns a sequential number to each purchase month for a given user. This is crucial for detecting gaps.
3. **ConsecutiveMonths CTE:**

    - EXTRACT(YEAR FROM purchase_month) * 12 + EXTRACT(MONTH FROM purchase_month): Converts the month to a continuous integer (e.g., Jan 2025 = 2025*12+1, Feb 2025 = 2025*12+2).
    - - rn AS month_group_id: This is the clever trick. If months are consecutive, the (year*12+month) - rn value will be constant.
        - Example:
            - Jan 2025 (rn=1): (2025*12+1) - 1 = 24301
            - Feb 2025 (rn=2): (2025*12+2) - 2 = 24302 - 2 = 24300 -- Error in logic. Let's re-evaluate.
    - **Correction to ConsecutiveMonths CTE logic (The "Islands and Gaps" method):** The formula for month_group_id should be (year * 12 + month) - rn. If the months are consecutive, this value *will be the same*.

Let's re-trace with the correct formula:

- User 101:
    - Jan 2025 (rn=1): (2025*12 + 1) - 1 = 24301 - 1 = 24300
    - Feb 2025 (rn=2): (2025*12 + 2) - 2 = 24302 - 2 = 24300
    - Mar 2025 (rn=3): (2025*12 + 3) - 3 = 24303 - 3 = 24300
    - Apr 2025 (rn=4): (2025*12 + 4) - 4 = 24304 - 4 = 24300

This month_group_id successfully identifies consecutive sequences.

4. **Final SELECT Statement:**
    - GROUP BY user_id, month_group_id: Groups the data by each user and their consecutive month sequences.
    - HAVING COUNT(*) >= 3: Filters these groups to only include those where the count of months in a consecutive sequence is 3 or more.
    - SELECT DISTINCT user_id: Selects the unique user IDs that meet this criterion.

Output Table:

| user_id |

|---------|

101

(User 101 has purchases in Jan, Feb, Mar, Apr 2025 which includes multiple sets of 3 consecutive months: JFM, FMA.)

Question 6: Identify Skewed Joins and Fix with Broadcast Hint

Concept: This question is highly specific to distributed SQL engines like Databricks (Spark SQL). It tests your understanding of data distribution, join performance, and how to use hints to optimize query execution.

Understanding Skewed Joins: A skewed join occurs when the data for one or more join keys is unevenly distributed across the partitions of a distributed data processing system. For example, if you're joining orders and customers on customer_id, and 80% of your orders are for customer_id = 1, then one executor/task will be responsible for processing 80% of the data during the shuffle phase of the join, leading to that executor becoming a bottleneck.

Identifying Skewed Joins:

- **Slow Query Execution:** The most obvious sign.
- **Spark UI (Databricks Jobs):** Look for stages that take a very long time, especially shuffle reads. Within a stage, look at the "Tasks" tab and observe the "Duration" and "Input Size" for individual tasks. If a few tasks are significantly larger/slower than others, it's a strong indicator of skew.
- **Error Messages:** Sometimes, you might see "OutOfMemoryError" on an executor if the skewed partition is too large to fit in memory.

Input Tables:

Let's imagine we have a Customers table (relatively small) and a Orders table (very large and potentially skewed on customer_id).

Customers Table (Small/Reference Table)

customer_id	customer_name	region
1	Alice	North

2	Bob	South
3	Charlie	East
... (millions)		

Orders Table (Large/Fact Table, potentially skewed)

order_id	customer_id	order_date	amount
O001	1	2025-01-01	100
O002	1	2025-01-02	150
O003	2	2025-01-02	200
O004	1	2025-01-03	50
O005	1	2025-01-03	250
O006	3	2025-01-04	120
... (billions)			

Self-correction: Customer_id 1 having disproportionately many orders would be the skew.

Required SQL Query (Original Query - potentially skewed):

```
SELECT
  o.order_id,
  c.customer_name,
  o.amount
FROM
  Orders o
JOIN
  Customers c ON o.customer_id = c.customer_id;
```

Required SQL Query (Fix with Broadcast Hint):

A broadcast join works by sending the *entire small table* to *all* nodes/executors in the cluster. This avoids the shuffle phase for the large table, as the small table's data is already available locally on each node. This is highly efficient when one side of the join is small enough to fit into memory on all executors.

In Databricks/Spark SQL, you use the BROADCAST hint.

SQL

```
SELECT /*+ BROADCAST(c) */
  o.order_id,
  c.customer_name,
  o.amount
FROM
  Orders o
JOIN
  Customers c ON o.customer_id = c.customer_id;
```

Explanation of the fix:

- /*+ BROADCAST(c) */: This is a Spark SQL (and by extension, Databricks SQL) hint. It tells the optimizer to broadcast the Customers table (c) to all executor nodes before performing the join.
- **When to use:** Use this when one of the join tables is significantly smaller than the other (typically, fits entirely in the memory of each executor, or a few hundred MBs to a few GBs).
- **Benefits:**
  - Eliminates the shuffle of the larger table.
  - Significantly reduces network I/O.
  - Often leads to much faster join performance, especially for skewed joins, as the "skewed" key from the large table now has the entire small table locally to join against.
- **Drawbacks:**

- If the broadcasted table is too large, it can lead to OutOfMemoryError on executors.
- Network overhead of broadcasting a very large table.

Output Table (Same as original join, but performed more efficiently):

order_id	customer_name	amount
O001	Alice	100
O002	Alice	150
O003	Bob	200
O004	Alice	50
O005	Alice	250
O006	Charlie	120
...		

Important Note for Interviews: For Question 6, the interviewer isn't just looking for the syntax. They want to know:

- How you identify skew (monitoring Spark UI, understanding job stages).
- Your understanding of *why* broadcasting helps (reduces shuffle, brings data to computation).
- The limitations and considerations for using broadcast joins (memory, table size).
- Other strategies for handling skew (salting, repartitioning, AQE - Adaptive Query Execution in Spark 3+ which can handle some skew automatically).

7) 7 day moving average of Product sales

8) Pivot daily sales into month - wise columns

9) customers who brought products every month in a year .

No problem, let's break down the next set of SQL questions. These involve more window functions, pivoting data, and advanced temporal analysis, all highly relevant for data engineering and analytics roles.

Question 7: 7-Day Moving Average of Product Sales

Concept: This question tests your understanding of window functions, specifically calculating a moving (or rolling) average. This is a common requirement in time-series analysis for smoothing out daily fluctuations and observing trends.

Input Table: ProductSales

sale_date	product_id	sales_amount
2025-01-01	P001	100
2025-01-01	P002	50
2025-01-02	P001	120
2025-01-02	P002	60
2025-01-03	P001	110
2025-01-03	P002	55
2025-01-04	P001	130
2025-01-04	P002	70
2025-01-05	P001	90
2025-01-05	P002	45
2025-01-06	P001	140
2025-01-06	P002	75
2025-01-07	P001	105
2025-01-07	P002	52
2025-01-08	P001	115
2025-01-08	P002	58
2025-01-09	P001	135
2025-01-09	P002	65

Required SQL Query:

First, we need to aggregate daily sales per product. Then, apply the moving average.

```
WITH DailyProductSales AS (
  SELECT
    sale_date,
    product_id,
    SUM(sales_amount) AS daily_total_sales
  FROM
    ProductSales
  GROUP BY
    sale_date,
    product_id
)
SELECT
  sale_date,
  product_id,
  daily_total_sales,
  AVG(daily_total_sales) OVER (
    PARTITION BY product_id
    ORDER BY sale_date
    ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
  ) AS seven_day_moving_avg
FROM
  DailyProductSales
ORDER BY
  product_id,
  sale_date;
```

Explanation:

1. **DailyProductSales CTE:**
   - This CTE aggregates the sales_amount for each product_id on each sale_date, ensuring we have a single daily sales figure per product.
2. **Outer Query with Window Function:**
   - AVG(daily_total_sales) OVER (...): This calculates the average of daily_total_sales over a defined window.
   - PARTITION BY product_id: This is crucial. It tells the window function to calculate the moving average separately for each unique product_id. Without this, it would calculate a global moving average.
   - ORDER BY sale_date: Specifies the order within each partition, which is necessary for a time-based window.
   - ROWS BETWEEN 6 PRECEDING AND CURRENT ROW: This defines the sliding window. It means for the current row's sale_date, the average should include the daily_total_sales from the current day and the 6 days immediately preceding it, summing up to a 7-day window.

Output Table (Partial, for P001 example):

sale_date	product_id	daily_total_sales	seven_day_moving_avg
2025-01-01	P001	100	100.00
2025-01-02	P001	120	110.00
2025-01-03	P001	110	110.00
2025-01-04	P001	130	115.00
2025-01-05	P001	90	110.00
2025-01-06	P001	140	116.67
2025-01-07	P001	105	116.43
2025-01-08	P001	115	118.57

sale_date	product_id	daily_total_sales	seven_day_moving_avg
2025-01-09	P001	135	120.71

(Values are rounded for display. For the first few days, the average is based on fewer than 7 days as there isn't enough preceding data.)

Question 8: Pivot Daily Sales into Month-Wise Columns

Concept: This question requires "pivoting" data, transforming rows (daily sales) into columns (monthly sales). This is often done for reporting and analysis to get a summary view. SQL implementations for pivoting vary, but the most common ways are using conditional aggregation (CASE WHEN with SUM) or a specific PIVOT clause if the SQL dialect supports it (like Oracle, SQL Server, and Spark SQL has PIVOT functionality).

Input Table: DailySales

sale_date	sales_amount
2025-01-05	100
2025-01-15	120
2025-02-01	80
2025-02-10	90
2025-02-20	110
2025-03-03	150
2025-03-12	130
2025-03-25	160
2025-04-01	70
2025-04-10	85

Required SQL Query (Using Conditional Aggregation - most portable):

```
SELECT
  EXTRACT(YEAR FROM sale_date) AS sales_year,
  SUM(CASE WHEN EXTRACT(MONTH FROM sale_date) = 1 THEN sales_amount ELSE 0 END) AS Jan_Sales,
  SUM(CASE WHEN EXTRACT(MONTH FROM sale_date) = 2 THEN sales_amount ELSE 0 END) AS Feb_Sales,
  SUM(CASE WHEN EXTRACT(MONTH FROM sale_date) = 3 THEN sales_amount ELSE 0 END) AS Mar_Sales,
  SUM(CASE WHEN EXTRACT(MONTH FROM sale_date) = 4 THEN sales_amount ELSE 0 END) AS Apr_Sales,
  -- ... and so on for all 12 months
  SUM(CASE WHEN EXTRACT(MONTH FROM sale_date) = 12 THEN sales_amount ELSE 0 END) AS Dec_Sales
FROM
  DailySales
GROUP BY
  EXTRACT(YEAR FROM sale_date)
ORDER BY
  sales_year;
```

Explanation:

- EXTRACT(YEAR FROM sale_date) AS sales_year: Groups the sales by year.
- SUM(CASE WHEN EXTRACT(MONTH FROM sale_date) = N THEN sales_amount ELSE 0 END) AS Month_Name_Sales: This is the pivoting logic.
  - For each row, it checks the month of sale_date.
  - If it matches the specified month (N), it includes the sales_amount in that month's sum.
  - Otherwise, it contributes 0 to that month's sum.
  - The SUM() then aggregates these conditional values for each sales_year.

Required SQL Query (Using Spark SQL PIVOT function - Databricks specific):

Databricks (Spark SQL) has a PIVOT clause which is more concise for this type of operation.

```
SELECT *
FROM (
  SELECT
    EXTRACT(YEAR FROM sale_date) AS sales_year,
    EXTRACT(MONTH FROM sale_date) AS sales_month_num,
    sales_amount
  FROM
    DailySales
)
PIVOT (
  SUM(sales_amount)
  FOR sales_month_num IN (
    1 AS Jan_Sales,
    2 AS Feb_Sales,
    3 AS Mar_Sales,
    4 AS Apr_Sales,
    5 AS May_Sales,
    6 AS Jun_Sales,
    7 AS Jul_Sales,
    8 AS Aug_Sales,
    9 AS Sep_Sales,
    10 AS Oct_Sales,
        11 AS Nov_Sales,
        12 AS Dec_Sales
    )
)
ORDER BY sales_year;
```

Output Table (for 2025):

sales_year	Jan_Sales	Feb_Sales	Mar_Sales	Apr_Sales	...	Dec_Sales
2025	220	280	440	155	...	0

Question 9: Customers Who Bought Products Every Month in a Year

Concept: This is another temporal analysis problem, similar to the consecutive months but focusing on *all* months within a given year. It requires identifying distinct active months for each customer and checking if that count matches the total number of months in a year (12).

Input Table: CustomerPurchases

purchase_id	customer_id	purchase_date	amount
P001	101	2024-01-05	25
P002	102	2024-01-10	30
P003	101	2024-02-15	40
P004	103	2024-02-20	50
P005	101	2024-03-01	60
P006	102	2024-03-18	35
P007	101	2024-04-05	70

P008	103	2024-04-10	55
P009	101	2024-05-01	80
P010	102	2024-05-10	45
P011	101	2024-06-01	90
P012	103	2024-06-15	65
P013	101	2024-07-01	100
P014	102	2024-07-20	50
P015	101	2024-08-01	110
P016	103	2024-08-05	75
P017	101	2024-09-01	120
P018	102	2024-09-10	60
P019	101	2024-10-01	130
P020	103	2024-10-20	80
P021	101	2024-11-01	140
P022	102	2024-11-15	65
P023	101	2024-12-01	150
P024	103	2024-12-10	90

Required SQL Query:

```sql
WITH CustomerMonthlyActivity AS (
  SELECT DISTINCT
    EXTRACT(YEAR FROM purchase_date) AS purchase_year,
    EXTRACT(MONTH FROM purchase_date) AS purchase_month,
    customer_id
  FROM
    CustomerPurchases
)
SELECT
   customer_id,
   purchase_year
FROM
   CustomerMonthlyActivity
GROUP BY
   customer_id,
   purchase_year
HAVING
   COUNT(DISTINCT purchase_month) = 12; -- Check if they had distinct purchases in all 12 months
```

Explanation:

1. **CustomerMonthlyActivity CTE:**
   - EXTRACT(YEAR FROM purchase_date) AS purchase_year: Extracts the year of the purchase.
   - EXTRACT(MONTH FROM purchase_date) AS purchase_month: Extracts the month of the purchase.
   - DISTINCT is crucial here. It ensures that if a customer makes multiple purchases in the same month, that month is only counted once for their activity.
2. **Outer Query:**
   - GROUP BY customer_id, purchase_year: Groups the distinct monthly activities by each customer and the year.
   - HAVING COUNT(DISTINCT purchase_month) = 12: This is the filtering condition. It checks if the count of *distinct* months a customer was active in a given year is exactly 12. If it is, they purchased in every month of that year.

Output Table (for 2024):

customer_id	purchase_year
101	2024

Question 10: Rank Products by Sales Per Year

Concept: This question tests your ability to use window functions, specifically ranking functions (RANK(), DENSE_RANK(), ROW_NUMBER()), to assign ranks to products based on their sales within each year.

Input Table: ProductSales

sale_date	product_id	sales_amount
2024-01-01	P001	100
2024-01-05	P002	150
2024-02-10	P001	200
2024-03-01	P003	120
2024-04-15	P002	180
2024-05-20	P001	250
2024-06-01	P003	90
2025-01-01	P001	300
2025-01-10	P002	220
2025-02-15	P003	180
2025-03-01	P001	350
2025-04-05	P002	280
2025-05-10	P003	200

Required SQL Query:

We need to first calculate the total sales for each product per year, then apply the ranking. Let's use RANK() here, which assigns the same rank to ties and leaves gaps in the sequence. DENSE_RANK() would assign sequential ranks without gaps for ties, and ROW_NUMBER() assigns unique ranks even for ties. Choose based on specific requirements for tie-breaking.

```
WITH YearlyProductSales AS (
  SELECT
    EXTRACT(YEAR FROM sale_date) AS sales_year,
    product_id,
    SUM(sales_amount) AS total_sales
  FROM
    ProductSales
  GROUP BY
    EXTRACT(YEAR FROM sale_date),
    product_id
)
SELECT
  sales_year,
  product_id,
  total_sales,
  RANK() OVER (PARTITION BY sales_year ORDER BY total_sales DESC) AS sales_rank
FROM
  YearlyProductSales
ORDER BY
  sales_year,
  sales_rank;
```

Explanation:

1. **YearlyProductSales CTE:**
   - This CTE aggregates sales_amount by sales_year and product_id to get the total sales for each product in a given year.
2. **Outer Query with Window Function:**
   - RANK() OVER (PARTITION BY sales_year ORDER BY total_sales DESC):
     - PARTITION BY sales_year: This is crucial. It tells the ranking function to restart the rank calculation for each new year.
     - ORDER BY total_sales DESC: Within each year, products are ordered by their total_sales in descending order (highest sales get rank 1).
     - RANK(): Assigns a rank. If two products have the same total sales, they get the same rank, and the next rank skips a number (e.g., 1, 2, 2, 4).

Output Table:

sales_year	product_id	total_sales	sales_rank
2024	P001	550	1
2024	P002	330	2
2024	P003	210	3
2025	P001	650	1
2025	P002	500	2
2025	P003	380	3

Question 11: Employees Earning More Than Department Average

Concept: This is a common pattern for comparing individual records to an aggregate value within their group. It can be solved using subqueries or window functions. Window functions are generally preferred for performance and readability in modern SQL.

Input Table: Employees

employee_id	employee_name	department	salary
101	Alice	Sales	60000
102	Bob	Sales	75000
103	Charlie	Sales	50000
104	David	Marketing	80000
105	Eve	Marketing	70000
106	Frank	Marketing	90000
107	Grace	HR	65000
108	Heidi	HR	60000

Required SQL Query (Using Window Function - recommended):

SQL

```
WITH DepartmentAvgSalary AS (
  SELECT
    employee_id,
    employee_name,
    department,
    salary,
    AVG(salary) OVER (PARTITION BY department) AS dept_avg_salary
  FROM
    Employees
)
SELECT
  employee_id,
  employee_name,
  department,
  salary,
  dept_avg_salary
FROM
  DepartmentAvgSalary
WHERE
  salary > dept_avg_salary
ORDER BY
  department,
  employee_id;
```

Explanation:

1. **DepartmentAvgSalary CTE:**
    - AVG(salary) OVER (PARTITION BY department): This calculates the average salary for each department. The PARTITION BY department ensures that the average is computed separately for each department, and this average is then "attached" to every employee in that department in the result set.
2. **Outer Query:**
    - WHERE salary > dept_avg_salary: Filters the results to include only those employees whose individual salary is greater than their department's average salary.

Output Table:

```
| employee_id | employee_name | department | salary | dept_avg_salary |
|-------------|---------------|------------|--------|-----------------|
| 107         | Grace         | HR         | 65000  | 62500.00        |
| 106         | Frank         | Marketing  | 90000  | 80000.00        |
| 102         | Bob           | Sales      | 75000  | 61666.67        |
```

Alternative (Using Subquery):

```
SELECT
  e.employee_id,
  e.employee_name,
  e.department,
  e.salary,
  da.avg_salary AS dept_avg_salary
FROM
  Employees e
JOIN (
  SELECT
    department,
    AVG(salary) AS avg_salary
  FROM
    Employees
  GROUP BY
    department
) da ON e.department = da.department
WHERE
  e.salary > da.avg_salary
ORDER BY
  e.department,
  e.employee_id;
```

This approach first calculates the average salary per department in a subquery, then joins it back to the main Employees table to perform the comparison.

Question 12: Find Median Transaction Amount

Concept: Finding the median is tricky in SQL because it's a positional measure, not a straightforward aggregate like AVG or SUM.

- **Built-in:** Some SQL databases (like SQL Server 2012+, Oracle, PostgreSQL 9.4+) have PERCENTILE_CONT or MEDIAN aggregate functions. Spark SQL/Databricks also provides PERCENTILE_CONT and PERCENTILE_APPROX.
- **Manual Method:** For databases without built-in median, you typically use window functions (ROW_NUMBER()) to assign ranks and then select the middle value(s).

Input Table: Transactions

transaction_id	transaction_amount
T001	100
T002	250
T003	50
T004	300
T005	120
T006	180
T007	200
T008	70
T009	160

Sorted amounts: 50, 70, 100, 120, **160**, 180, 200, 250, 300 (Median is 160)

Required SQL Query (Using Built-in Function - Spark SQL/Databricks):

```
SELECT
   PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY transaction_amount) AS
median_transaction_amount
FROM
```

```
  Transactions;


-- OR (for approximate median, which can be faster for large datasets)
-- SELECT PERCENTILE_APPROX(transaction_amount, 0.5) AS approximate_median_transaction_amount
-- FROM Transactions;
```

Explanation (Built-in):

- PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY transaction_amount): This function calculates the 50th percentile (which is the median) of transaction_amount.
    - PERCENTILE_CONT (continuous) interpolates values if the median falls between two numbers (e.g., for an even number of rows).
    - WITHIN GROUP (ORDER BY transaction_amount) specifies the column to order by for the percentile calculation.

Output Table (Built-in):

```
| median_transaction_amount |
|---------------------------|
| 160.0                     |
```

Required SQL Query (Manual Method - more portable across SQL dialects):

This method handles both odd and even numbers of rows.

```
WITH RankedTransactions AS (
  SELECT
    transaction_amount,
    ROW_NUMBER() OVER (ORDER BY transaction_amount) AS rn,
    COUNT(*) OVER () AS total_rows
  FROM
    Transactions
)
SELECT
  AVG(transaction_amount) AS median_transaction_amount
FROM
  RankedTransactions
WHERE
  rn = CEIL(total_rows / 2.0) OR rn = FLOOR(total_rows / 2.0) + 1;
  -- For odd total_rows, both conditions point to the same middle row.
  -- For even total_rows, it selects the two middle rows to average.
```

Explanation (Manual Method):

1. **RankedTransactions CTE:**
   - ROW_NUMBER() OVER (ORDER BY transaction_amount) AS rn: Assigns a sequential rank to each transaction amount after ordering them.
   - COUNT(*) OVER () AS total_rows: This is a window function that counts the total number of rows in the entire result set (the entire table in this case), and this count is added to every row.
2. **Outer Query:**
   - WHERE rn = CEIL(total_rows / 2.0) OR rn = FLOOR(total_rows / 2.0) + 1: This is the core logic to select the middle row(s).
     - If total_rows is odd (e.g., 9): CEIL(9/2.0) = 5, FLOOR(9/2.0)+1 = 4+1=5. It selects the 5th row.
     - If total_rows is even (e.g., 8): CEIL(8/2.0) = 4, FLOOR(8/2.0)+1 = 4+1=5. It selects the 4th and 5th rows.
   - AVG(transaction_amount): Averages the selected transaction amount(s). If only one row is selected (odd count), it's just that value. If two rows are selected (even count), it averages them.

Output Table (Manual Method):

median_transaction_amount
160.0

Question 13: Users Who Placed Their First Order in the Last 30 Days

Concept: This question involves identifying a user's *first* activity and then filtering based on a time window. It combines aggregation (MIN), date functions, and filtering.

Input Table: Orders

order_id	user_id	order_date	amount
O001	101	2025-05-20	100
O002	102	2025-05-25	150
O003	101	2025-05-28	50
O004	103	2025-06-01	200
O005	104	2025-06-05	120
O006	102	2025-06-08	75
O007	105	2025-06-15	300
O008	103	2025-06-18	90
O009	101	2025-06-20	110
O010	106	2025-06-25	180
O011	107	2025-06-28	220

Given today's date is 2025-06-29 (as per your prompt's current time). Last 30 days would be from 2025-05-30 to 2025-06-29.

Required SQL Query:

```
SELECT
  user_id,
  MIN(order_date) AS first_order_date
FROM
  Orders
GROUP BY
  user_id
HAVING
  MIN(order_date) >= DATE('2025-06-29') - INTERVAL '30' DAY AND MIN(order_date) <=
DATE('2025-06-29');
  -- In Databricks/Spark SQL, you might use:
  -- MIN(order_date) >= date_sub(current_date(), 30) AND MIN(order_date) <=
current_date();
  -- Or for literal date:
  -- MIN(order_date) >= date_sub('2025-06-29', 30) AND MIN(order_date) <= '2025-06-29';
```

Explanation:

1. SELECT user_id, MIN(order_date) AS first_order_date FROM Orders GROUP BY user_id: This part aggregates the data to find the earliest order_date for each user_id. This effectively identifies their *first* order.
2. HAVING MIN(order_date) >= DATE('2025-06-29') - INTERVAL '30' DAY AND MIN(order_date) <= DATE('2025-06-29'): This HAVING clause filters the grouped results. It keeps only those users whose first_order_date falls within the last 30 days from the current date (2025-06-29).
   - DATE('YYYY-MM-DD') - INTERVAL 'X' DAY is common syntax (e.g., in MySQL, PostgreSQL).
   - For Databricks/Spark SQL, date_sub(current_date(), 30) is the typical way to get a date 30 days ago. current_date() would be used if the query runs on the current day, otherwise, a specific date literal like '2025-06-29' is used.

Output Table:

user_id	first_order_date
103	2025-06-01
104	2025-06-05
105	2025-06-15
106	2025-06-25
107	2025-06-28

Question 14: Compare Price Change Between Two Dates Per Product

Concept: This requires comparing a product's price at a start date versus an end date. This can be done by joining the table to itself or, more elegantly, using window functions like LAG or by filtering and joining. Let's assume price is recorded daily for simplicity.

Input Table: ProductPrices

product_id	price_date	price
P001	2025-01-01	10.00
P002	2025-01-01	20.00
P001	2025-01-15	10.50
P002	2025-01-15	21.00
P001	2025-01-31	11.00
P002	2025-01-31	20.50
P003	2025-01-01	5.00
P003	2025-01-31	5.50

Required SQL Query (Let's compare between '2025-01-01' and '2025-01-31'):

```
WITH StartPrices AS (
  SELECT
    product_id,
    price AS start_price
  FROM
    ProductPrices
  WHERE
    price_date = '2025-01-01'
),
EndPrices AS (
  SELECT
    product_id,
    price AS end_price
  FROM
    ProductPrices
  WHERE
    price_date = '2025-01-31'
)
SELECT
  s.product_id,
  s.start_price,
  e.end_price,
  e.end_price - s.start_price AS price_change,
  ROUND((e.end_price - s.start_price) / s.start_price * 100, 2) AS percentage_change
FROM
```

```
  StartPrices s
JOIN
  EndPrices e ON s.product_id = e.product_id
ORDER BY
  s.product_id;
```

Explanation:

1. **StartPrices CTE:** Filters ProductPrices to get prices on the specified start_date.
2. **EndPrices CTE:** Filters ProductPrices to get prices on the specified end_date.
3. **Outer Query:**
   - JOIN StartPrices s JOIN EndPrices e ON s.product_id = e.product_id: Joins the two CTEs on product_id to bring the start and end prices onto the same row.
   - e.end_price - s.start_price AS price_change: Calculates the absolute change.
   - ROUND((e.end_price - s.start_price) / s.start_price * 100, 2) AS percentage_change: Calculates the percentage change. ROUND is used for cleaner output.

Output Table:

product_id	start_price	end_price	price_change	percentage_change
P001	10.00	11.00	1.00	10.00
P002	20.00	20.50	0.50	2.50
P003	5.00	5.50	0.50	10.00

Question 15: Customers Whose First and Last Transaction on Same Day

Concept: This requires finding the minimum and maximum transaction dates for each customer and then comparing them.

Input Table: CustomerTransactions

transaction_id	customer_id	transaction_date	amount	
T001	101	2025-04-01	100	
T002	102	2025-04-05	150	
T003	101	2025-04-01	50	-- Same day as T001
T004	103	2025-04-10	200	
T005	102	2025-04-05	75	-- Same day as T002
T006	104	2025-04-12	300	
T007	103	2025-04-11	90	
T008	105	2025-04-15	180	

Required SQL Query:

```
SELECT
  customer_id,
  MIN(transaction_date) AS first_transaction_date,
  MAX(transaction_date) AS last_transaction_date
FROM
  CustomerTransactions
GROUP BY
  customer_id
HAVING
  MIN(transaction_date) = MAX(transaction_date)
ORDER BY
  customer_id;
```

Explanation:

1. SELECT customer_id, MIN(transaction_date) AS first_transaction_date, MAX(transaction_date) AS last_transaction_date FROM CustomerTransactions GROUP BY customer_id: This aggregates the data to find the earliest (MIN) and latest (MAX) transaction_date for each customer_id.
2. HAVING MIN(transaction_date) = MAX(transaction_date): This HAVING clause filters the grouped results, keeping only those customers where their first and last transaction dates are identical.

Output Table:

customer_id	first_transaction_date	last_transaction_date
101	2025-04-01	2025-04-01
102	2025-04-05	2025-04-05
104	2025-04-12	2025-04-12
105	2025-04-15	2025-04-15

Question 16: Percentage of Returning Users Each Month

Concept: This is a classic cohort analysis or user retention problem. It requires identifying first-time users, then identifying returning users in subsequent months, and finally calculating the percentage.

Input Table: UserLogins (or UserPurchases, assuming any activity counts as "usage")

user_id	activity_date	
101	2024-12-01	-- First activity (Dec 2024)
102	2024-12-05	-- First activity (Dec 2024)
103	2025-01-01	-- First activity (Jan 2025)
101	2025-01-10	-- Returning in Jan
102	2025-01-15	-- Returning in Jan
104	2025-01-20	-- First activity (Jan 2025)

| 101 | 2025-02-01 | -- Returning in Feb

| 103 | 2025-02-05 | -- Returning in Feb

| 105 | 2025-02-10 | -- First activity (Feb 2025)

| 102 | 2025-03-01 | -- Returning in Mar

| 101 | 2025-03-05 | -- Returning in Mar

| 104 | 2025-03-10 | -- Returning in Mar

Required SQL Query:

This is a multi-step process:

1. Find each user's first activity month.
2. Find distinct active users per month.
3. For each month, count users who are *not* first-time users in that month (returning users).
4. Calculate the percentage.

```
WITH UserFirstActivity AS (
  SELECT
    user_id,
    DATE_TRUNC('month', MIN(activity_date)) AS first_activity_month
  FROM
    UserLogins
  GROUP BY
    user_id
),
MonthlyActiveUsers AS (
  SELECT DISTINCT
    DATE_TRUNC('month', ul.activity_date) AS activity_month,
        ul.user_id,
        ufa.first_activity_month
    FROM
        UserLogins ul
    JOIN
        UserFirstActivity ufa ON ul.user_id = ufa.user_id
),
MonthlyUserCounts AS (
    SELECT
        activity_month,
        COUNT(user_id) AS total_active_users,
        COUNT(CASE WHEN activity_month > first_activity_month THEN user_id ELSE NULL
END) AS returning_users,
        COUNT(CASE WHEN activity_month = first_activity_month THEN user_id ELSE NULL
END) AS new_users
    FROM
        MonthlyActiveUsers
    GROUP BY
        activity_month
)
SELECT
    activity_month,
    total_active_users,
    new_users,
    returning_users,
    CASE
    WHEN total_active_users > 0 THEN ROUND((CAST(returning_users AS DECIMAL) / total_active_users) * 100, 2)
    ELSE 0
  END AS percentage_returning_users
FROM
  MonthlyUserCounts
ORDER BY
  activity_month;
```

Explanation:

1. **UserFirstActivity CTE:** Determines the first_activity_month for each user.
2. **MonthlyActiveUsers CTE:** Gets all distinct user_id and their activity_month (current month of activity) along with their first_activity_month.
3. **MonthlyUserCounts CTE:**
   - COUNT(user_id) AS total_active_users: Counts all unique users active in a given activity_month.
   - COUNT(CASE WHEN activity_month > first_activity_month THEN user_id ELSE NULL END) AS returning_users: This is the key for returning users. A user is "returning" in a given activity_month if their activity_month is later than their first_activity_month. COUNT(CASE WHEN ... THEN ... ELSE NULL END) effectively counts only those where the condition is true.
   - COUNT(CASE WHEN activity_month = first_activity_month THEN user_id ELSE NULL END) AS new_users: Counts users whose activity_month is their first_activity_month (i.e., new users for that month).
4. **Final SELECT:**
   - Calculates the percentage_returning_users by dividing returning_users by total_active_users and multiplying by 100.
   - CAST(... AS DECIMAL) ensures floating-point division.
   - CASE WHEN total_active_users > 0 THEN ... ELSE 0 END handles division by zero.

Output Table:

activity_month	total_active_users	new_users	returning_users	percentage_returning_users
2024-12-01	2	2	0	0.00
2025-01-01	4	2	2	50.00
2025-02-01	3	1	2	66.67
2025-03-01	3	0	3	100.00

These are excellent, practical SQL problems often encountered in data analysis and engineering