Advanced Analytical Patterns in SQL: A Comprehensive Compendium of Window Functions and Practical Solved Problems

The modern data landscape necessitates a shift from traditional aggregate logic, which collapses datasets into summary rows, toward windowed analytical processing that preserves row-level granularity while performing multi-row computations.[1] Window functions, often categorized as the analytical Swiss army knife of SQL, allow engineers to address complex requirements such as running totals, moving averages, and ranking without the performance overhead associated with correlated subqueries or multiple self-joins.[1] By defining a specific "window" or partition of data, these functions provide a mechanism to analyze trends, sequences, and distributions with surgical precision, offering both the detail and the big picture simultaneously.[2]

Architectural Foundations of Windowing Functions

At the core of advanced SQL windowing is the OVER() clause, which distinguishes these functions from standard aggregates.[2] While an aggregate function like SUM() returns a single value for a group, the windowed version of SUM() computes a result for every individual row based on a defined frame.[8] The mechanism relies on three primary components: partitioning, ordering, and framing.[7]

Partitioning and Logical Grouping

The PARTITION BY clause divides the result set into logical groups, similar to a GROUP BY operation, but without merging rows.[1] For example, if an analyst requires the total sales per region alongside every individual order, SUM(amount) OVER (PARTITION BY region) calculates the total for each region and projects it back onto every relevant row.[1] This allows business users to view individual transactions in the context of broader regional performance, a common requirement in operational dashboards.[2]

Ordering and Sequential Calculation

The ORDER BY clause within the OVER() window determines the logical sequence in which the function is applied to the rows in each partition.[8] This is critical for functions where the order of operations changes the result, such as running totals or ranking.[1] When ORDER BY is specified without an explicit frame, the default window frame typically extends from the beginning of the partition to the current row, effectively creating a cumulative effect.[15]

The Frame Clause and Physical Offsets

The frame clause (e.g., ROWS BETWEEN) further refines the subset of rows within a partition that the function considers.[1] ROWS defines the frame based on physical offsets from the current row, whereas RANGE defines it based on logical value differences.[2] This distinction is vital in time-series data; a RANGE might include all rows within a 7-day interval regardless of how many rows exist, while ROWS would strictly count a fixed number of rows.[1]

Ranking Architecture and Ordinal Analysis

The ability to assign ranks and row numbers is fundamental to identifying top performers, detecting the "nth" event in a sequence, and filtering noise from datasets.[1]

Distinguishing Functional Behaviors: ROW_NUMBER, RANK, and DENSE_RANK

Selecting the appropriate ranking function depends on how business logic handles ties—rows sharing identical values in the sort criteria.[1]

- **ROW_NUMBER()**: Assigns a unique sequential integer starting at 1. It does not account for ties, meaning identical values will receive different numbers based on internal processing order.[1]
- **RANK()**: Assigns the same rank to tied values but creates gaps in the numbering sequence. If two items are tied for first place, the next item receives a rank of 3.[1]
- **DENSE_RANK()**: Similar to RANK(), it assigns the same rank to tied values but maintains a continuous sequence without gaps. The item following a tie for first place receives a rank of 2.[1]

The comparative application of these functions is illustrated in the following table using a sample of concert revenues.[12]

Artist Name	Concert Revenue	ROW_NUMBER()	RANK()	DENSE_RANK()
BTS	800,000	1	1	1
Beyonce	750,000	2	2	2
Bruno Mars	700,000	3	3	3
Taylor Swift	700,000	4	3	3
Justin Bieber	680,000	5	5	4
Ed Sheeran	650,000	6	6	5
Adele	650,000	7	6	5

Solved Problem 1: Identifying the Top N Rentals per Genre

In a movie store database context containing tables for movie, single_rental, and customer, a requirement exists to identify the top 3 rentals by price within each movie genre.[19]

Schema Context: The movie table contains id, title, and genre. The single_rental table tracks payment_amount and links to movie_id.[19]

SQL Solution:

```
SQL
```

```sql
SELECT
    rental_date,
    title,
    genre,
    payment_amount,
    RANK() OVER(
        PARTITION BY genre
        ORDER BY payment_amount DESC
    ) as price_rank
FROM movie
JOIN single_rental ON single_rental.movie_id = movie.id;
```

Logical Insight: The use of RANK() is preferred here because the business rule permits ties (multiple movies can share the same rank if they cost the same) and accepts gaps in the numbering sequence.[13]

Solved Problem 2: Finding the Second Most-Recent Purchase

Detecting repeat purchase behavior often involves finding a specific instance in a chronological sequence, such as the second-ever purchase made by a customer.[19]

Scenario: Identify the customer who bought the second most-recent gift card, including their full name and payment date.[19]

SQL Solution:

```
SQL
```

```sql
WITH GiftCardRanking AS (
    SELECT
        c.first_name,
        c.last_name,
        g.payment_date,
        ROW_NUMBER() OVER(ORDER BY g.payment_date DESC) AS rank
```

```
    FROM customer c
    JOIN giftcard g ON c.id = g.customer_id
)
SELECT first_name, last_name, payment_date
FROM GiftCardRanking
WHERE rank = 2;
```

Logical Insight: ROW_NUMBER() is utilized because the requirement assumes a unique rank for each purchase, even if timestamps are identical.[19] By encapsulating the window function in a CTE, the outer query can filter the results, as window functions are generally prohibited in WHERE clauses.[3]

Solved Problem 3: Multi-Dimensional Performance Scoring

A frequent requirement in advanced analytics is the creation of composite scores that normalize different performance metrics into a single rankable value.[13]

Scenario: Rank employees based on Q1-2024 performance using Sales (40%), Satisfaction (30%), and Projects (30%).[13]

Mathematical Formula:

The score S is calculated as:

$$S = (Sales/1000 \times 0.4) + (Satisfaction \times 20 \times 0.3) + (Projects \times 2 \times 0.3)$$

SQL Solution:

SQL

```
WITH performance_metrics AS (
  SELECT
    employee_id, employee_name, department,
    sales_amount, customer_satisfaction, projects_completed,
    (sales_amount / 1000) * 0.4 +
    (customer_satisfaction * 20) * 0.3 +
    (projects_completed * 2) * 0.3 as composite_score
  FROM employee_performance
)
SELECT
  employee_name, department,
  ROUND(composite_score, 2) as performance_score,
  RANK() OVER (ORDER BY composite_score DESC) as company_rank,
  RANK() OVER (PARTITION BY department ORDER BY composite_score DESC) as dept_rank,
  CASE
    WHEN RANK() OVER (PARTITION BY department ORDER BY composite_score DESC) = 1
```

```
    THEN 'Top Performer'
    WHEN RANK() OVER (PARTITION BY department ORDER BY composite_score DESC) <= 3
    THEN 'High Performer'
    ELSE 'Standard Performer'
  END as performance_tier
FROM performance_metrics
ORDER BY company_rank;
```

Logical Insight: This implementation allows a "Top Performer" to be identified within each department, even if their overall company-wide rank is lower. The redundancy in the CASE statement illustrates how window functions can be reused for categorization.[13]

Statistical Offsets: Time-Series and Sequence Analysis

Window functions such as LAG and LEAD allow the engine to access data from preceding or following rows without complex self-joins, making them the primary tools for time-series analysis.[1]

Core Offsets: LEAD and LAG

- **LAG(expression, offset)**: Retrieves data from a row at a specified offset before the current row within the partition.[1]
- **LEAD(expression, offset)**: Retrieves data from a row at a specified offset after the current row.[1]

Solved Problem 4: Month-over-Month (MoM) Growth Rates

Benchmarking business growth requires comparing revenue in month T against month $T-1$. The LAG() function facilitates this period-over-period comparison.[4]

Growth Formula:

$$Growth\% = \frac{Sales_{current} - Sales_{previous}}{Sales_{previous}} \times 100$$

SQL Solution:

SQL

```
SELECT
    Year,
    Month,
    Sales,
    LAG(Sales) OVER (ORDER BY Year, Month) AS Previous_Sales,
    CONVERT(NUMERIC(10,2),
```

```
      100.0 * (Sales - LAG(Sales) OVER (ORDER BY Year, Month)) /
      LAG(Sales) OVER (ORDER BY Year, Month)
   ) AS Index_Growth_Percent
FROM Example_Ranking;
```

Logical Insight: The query uses LAG(Sales) to bring the previous month's value into the current row context, enabling the percentage growth calculation in a single pass.[27]

Solved Problem 5: Accidental Repeated Payment Detection

Duplicate transactions are a significant source of operational friction in fintech. SQL windowing can detect "repeats" by looking at temporal gaps between identical transactions.[1]

Scenario: Identify payments made at the same merchant with the same credit card and amount within 10 minutes of each other.[1]

SQL Solution:

SQL

```
WITH payments AS (
  SELECT
    merchant_id,
    EXTRACT(EPOCH FROM transaction_timestamp -
          LAG(transaction_timestamp) OVER(
             PARTITION BY merchant_id, credit_card_id, amount
             ORDER BY transaction_timestamp
          )) / 60 AS minute_difference
  FROM transactions
)
SELECT COUNT(*) AS payment_count
FROM payments
WHERE minute_difference <= 10;
```

Logical Insight: By partitioning by merchant, card, and amount, the query isolates potential duplicates. The LAG() function then calculates the time difference in minutes from the previous instance. This pattern effectively filters out transactions that occur in isolation.[1]

Aggregates Over Frames: Moving Averages and Running Totals

Windowed versions of standard aggregate functions (SUM, AVG, MIN, MAX, COUNT) enable rolling calculations that adjust based on the current row's position.[1]

Solved Problem 6: 7-Day Moving Revenue Average

Moving averages are used to smooth out daily fluctuations and identify long-term trends.[1]

SQL Solution:

```
SQL
```

```sql
SELECT
    sale_date,
    sales,
    AVG(sales) OVER (
        ORDER BY sale_date
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) AS rolling_avg
FROM daily_sales;
```

Logical Insight: The frame ROWS BETWEEN 6 PRECEDING AND CURRENT ROW ensures that the average is always calculated over the most recent week of data.[2] This physical row definition is reliable if the dataset contains one entry per day with no gaps.[2]

Solved Problem 7: Cumulative Spend and Customer Lifetime Value

Tracking how much a customer has spent in total up to each transaction point is a cornerstone of cohort analysis.[21]

SQL Solution:

```
SQL
```

```sql
SELECT
    customer_id,
    purchase_date,
    amount,
    SUM(amount) OVER (
        PARTITION BY customer_id
        ORDER BY purchase_date
    ) AS cumulative_spend
FROM purchases;
```

Logical Insight: The omission of an explicit frame clause when an ORDER BY is present causes the window to default to RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.[15] This effectively creates a running total that resets for each customer.[17]

Advanced Logic: The Gaps and Islands Problem

The "Gaps and Islands" problem refers to identifying continuous sequences (islands) and missing values (gaps) in a numeric or date sequence.[29]

Solved Problem 8: Identifying Missing Transaction IDs (The Gaps)

Scenario: Identify ranges of missing values in a sequence of integers.[29]

SQL Solution:

SQL

```
WITH C AS (
  SELECT
    col1 AS cur,
    LEAD(col1) OVER(ORDER BY col1) AS nxt
  FROM dbo.T1
)
SELECT cur + 1 AS rangestart, nxt - 1 AS rangeend
FROM C
WHERE nxt - cur > 1;
```

Logical Insight: The LEAD() function allows each row to see the value of the next row. If the difference between nxt and cur is greater than 1, a gap exists.[29]

Solved Problem 9: Identifying Consecutive Login Streaks (The Islands)

Scenario: Group consecutive days of logins into distinct islands to measure user engagement.[29]

SQL Solution:

SQL

```
WITH C AS (
  SELECT
    col1,
    col1 - DENSE_RANK() OVER(ORDER BY col1) AS grp
  FROM dbo.T1
)
SELECT MIN(col1) AS rangestart, MAX(col1) AS rangeend
FROM C
GROUP BY grp;
```

Logical Insight: This utilizes a classic mathematical trick: within an island, both the sequence values (col1) and the DENSE_RANK() increment by 1. Therefore, the difference between them remains constant. When a gap occurs, the sequence value jumps while the rank only increments by 1, creating a new "group" ID.[29]

User Behavioral Analytics and Sessionization

Sessionization is the process of grouping individual events into sessions based on a timeout threshold, typically 30 minutes of inactivity.[10]

Solved Problem 10: Building a Unique Session ID

SQL Solution:

SQL

```
WITH Boundaries AS (
    SELECT
        uid,
        event_timestamp,
        CASE
            WHEN EXTRACT(EPOCH FROM event_timestamp) -
                LAG(EXTRACT(EPOCH FROM event_timestamp)) OVER (
                    PARTITION BY uid
                    ORDER BY event_timestamp
                ) > 30 * 60 THEN 1
            ELSE 0
        END AS is_new_session
    FROM clickstream
)
SELECT
    uid,
    uid |

| '-' |
| CAST(SUM(is_new_session) OVER (
        PARTITION BY uid
        ORDER BY event_timestamp
    ) AS VARCHAR) AS session_id,
    event_timestamp
FROM Boundaries;
```

Logical Insight: This is a multi-step windowing process. First, LAG() is used to compare the current timestamp to the previous one to determine if the 30-minute threshold was exceeded. If so, a 1 is assigned as a boundary marker. In the second step, a cumulative SUM() is applied to

these markers; the sum increments only at a boundary, effectively assigning the same session_id to all events between boundaries.[34]

Boundary Analysis and Value Extraction

Value functions like FIRST_VALUE, LAST_VALUE, and NTH_VALUE allow analysts to anchor reports to specific data points within a window.[16]

Solved Problem 11: Comparing Sales Against the Initial Benchmark

Scenario: Calculate the difference between every customer purchase and their first-ever purchase amount.[23]

SQL Solution:

SQL

```
SELECT
  CustomerID,
  OrderDate,
  Amount,
  FIRST_VALUE(Amount) OVER (
    PARTITION BY CustomerID
    ORDER BY OrderDate
  ) AS FirstAmt,
  Amount - FIRST_VALUE(Amount) OVER (
    PARTITION BY CustomerID
    ORDER BY OrderDate
  ) AS DiffFromFirst
FROM Orders;
```

Logical Insight: FIRST_VALUE() retrieves the very first amount for each CustomerID based on date, which is then projected onto every row for that customer, enabling a simple subtraction to show growth over time.[23]

Solved Problem 12: Identifying the Most Popular Item per Category

Scenario: Find the last (most frequently purchased) item in each product category.[8]

SQL Solution:

SQL

```sql
SELECT
    item,
    purchases,
    LAST_VALUE(item) OVER (
        PARTITION BY category
        ORDER BY purchases
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS most_popular
FROM Produce;
```

Logical Insight: Unlike FIRST_VALUE, LAST_VALUE requires an explicit frame of ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.[23] Without this frame, the default window only looks up to the current row, incorrectly returning the current row's value as the "last".[23]

Pareto Analysis and Cumulative Distribution

The 80/20 rule (Pareto principle) is implemented using cumulative percentage calculations.[36]

Solved Problem 13: Identifying Top 80% Revenue Customers

Scenario: Determine which subset of customers accounts for 80% of total revenue.[37]

SQL Solution:

SQL

```sql
WITH RevenueSummary AS (
    SELECT customer_id, SUM(amount) AS customer_revenue
    FROM sales
    GROUP BY customer_id
),
CumulativeRevenue AS (
    SELECT
        customer_id,
        customer_revenue,
        SUM(customer_revenue) OVER (ORDER BY customer_revenue DESC) AS running_total,
        SUM(customer_revenue) OVER () AS grand_total
    FROM RevenueSummary
)
SELECT
    customer_id,
    customer_revenue,
    100.0 * running_total / grand_total AS cumulative_pct
FROM CumulativeRevenue
WHERE cumulative_pct <= 80.0;
```

Logical Insight: This query demonstrates the power of having two window functions in the same SELECT list. One SUM() has an ORDER BY to create a running total, while the other is empty to provide the grand total as a constant denominator.[11]

Subscription Metrics: Churn and Retention Patterns

Measuring churn involves identifying users who were active in one period but absent in the next.[6]

Solved Problem 14: Cohort Retention Identification

Scenario: Track monthly usage and identify "Churn" (users who did not return next month) versus "Active" (users who returned).[6]

SQL Solution:

SQL

```
WITH monthly_usage AS (
    SELECT
        who_identifier,
        DATEDIFF(month, '1970-01-01', when_timestamp) AS time_period
    FROM events
    WHERE event = 'login'
    GROUP BY 1, 2
),
lag_lead AS (
    SELECT
        who_identifier,
        time_period,
        LAG(time_period) OVER (PARTITION BY who_identifier ORDER BY time_period) AS
prev_month,
        LEAD(time_period) OVER (PARTITION BY who_identifier ORDER BY time_period) AS
next_month
    FROM monthly_usage
)
SELECT
    time_period,
    CASE
        WHEN prev_month IS NULL THEN 'NEW'
        WHEN (time_period - prev_month) = 1 THEN 'ACTIVE'
        ELSE 'RETURN'
    END AS status,
    CASE
        WHEN (next_month - time_period) > 1 OR next_month IS NULL THEN 1
        ELSE 0
```

```
    END AS churned
FROM lag_lead;
```

Logical Insight: The LEAD() function is used to peek at the user's next login month. If that month is not consecutive or is missing, the user is flagged as having churned from that period.[6]

Solving Overlapping Date Range Gaps

Overlapping ranges (e.g., resource usage) require identifying "islands" of time where events overlap, then calculating the net duration.[30]

Solved Problem 15: Net Duration of Overlapping Events

Scenario: Calculate the total "active time spent" handling customer inquiries, where a single employee might handle multiple inquiries simultaneously.[30]

SQL Solution:

SQL

```
WITH Grouping AS (
    SELECT
        Name, StartDate, EndDate,
        MAX(EndDate) OVER (
            PARTITION BY Name
            ORDER BY StartDate, EndDate
            ROWS BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING
        ) AS PreviousMaxEndDate
    FROM OverlappingDateRanges
),
Islands AS (
    SELECT
        *,
        CASE WHEN PreviousMaxEndDate >= StartDate THEN 0 ELSE 1 END AS IsNewIsland,
        SUM(CASE WHEN PreviousMaxEndDate >= StartDate THEN 0 ELSE 1 END) OVER (
            ORDER BY Name, StartDate
        ) AS IslandID
    FROM Grouping
)
SELECT
    Name,
    IslandID,
    MIN(StartDate) AS IslandStart,
    MAX(EndDate) AS IslandEnd,
    DATEDIFF(minute, MIN(StartDate), MAX(EndDate)) AS ActualTimeSpent
FROM Islands
GROUP BY Name, IslandID;
```

Logical Insight: The MAX() window function is used with a frame of 1 PRECEDING to identify the latest end date seen so far for that user.[30] If the current task starts *before* the previous tasks ended, it is part of the same island of activity.[30]

Industrial Performance and Optimization

Window functions are powerful but can be resource-intensive on massive datasets.[1]

Optimization Strategies

Optimizing windowed queries involves the following architectural considerations [1]:

Optimization Strategy	Description	Impact
Indexing	Index columns used in PARTITION BY and ORDER BY.[1]	Reduces sort and grouping time.
Early Filtering	Apply WHERE filters before the window function to reduce input volume.[1]	Minimizes rows processed in memory.
Frame Specificity	Use ROWS instead of RANGE when possible.[1]	ROWS is physically faster as it avoids value-based logic.
Materialization	Use CTEs to calculate window values once and filter in the outer query.[19]	Prevents redundant calculations.

Conclusion: Strategic Application of Advanced Analytical Logic

The evolution of SQL from a retrieval tool to a complex analytical engine is best evidenced by the proliferation of window functions.[2] By mastering these functions, data professionals can perform sequential, comparative, and ordinal analysis within the database layer, ensuring high performance and clear logic.[1] The transition from traditional joins to windowed logic—as seen in the Gaps and Islands, Sessionization, and Pareto examples—represents a fundamental shift toward set-level calculations that respect row-level context.[1] Ultimately, the strategic use of PARTITION BY, ORDER BY, and specific frame clauses transforms raw data streams into sophisticated business intelligence assets.[13]

Works cited

1. 12 SQL Window Functions Interview Questions (With Answers!) - DataLemur, accessed February 12, 2026, https://datalemur.com/blog/sql-window-functions-interview-questions
2. SQL Window Functions Explained Clearly (With Real Examples) | by Mathur Danduprolu, accessed February 12, 2026, https://medium.com/@mathur.danduprolu/sql-window-functions-explained-clearly-with-real-examples-1514c67d8554
3. 30 SQL Window Functionion Questions for Interviews- Day 47 of 100 Days of Data Engineering, AI and Azure Challenge | by Karthik | Medium, accessed February 12, 2026, https://medium.com/@krthiak/30-sql-window-functionion-questions-for-interviews-day-47-of-100-days-of-data-engineering-ai-and-638bef1af0f6
4. LAG and LEAD Functions: SQL - The Data School, accessed February 12, 2026, https://www.thedataschool.co.uk/elizabeth-archer/over-and-aggregate-functions-sql/
5. Window Functions in SQL: A Complete Guide in 2025 - Interview Query, accessed February 12, 2026, https://www.interviewquery.com/p/window-functions-sql
6. SQL for calculating Churn, Retention & Re-Engagement - ForceRank, accessed February 12, 2026, https://blog.forcerank.it/sql-for-calculating-churn-retention-reengagement
7. Advanced SQL Concepts - Free Computer Books, accessed February 12, 2026, https://freecomputerbooks.com/books/Advanced-SQL-Concepts.pdf
8. Window function calls | BigQuery - Google Cloud Documentation, accessed February 12, 2026, https://docs.cloud.google.com/bigquery/docs/reference/standard-sql/window-function-calls
9. Analyzing data with window functions - Snowflake Documentation, accessed February 12, 2026, https://docs.snowflake.com/en/user-guide/functions-window-using
10. Adobe-Defined SQL Functions in Query Service - Experience League, accessed February 12, 2026, https://experienceleague.adobe.com/en/docs/experience-platform/query/sql/adobe-defined-functions
11. Percentage calculations using SQL Window Functions | developers - Oracle Blogs, accessed February 12, 2026, https://blogs.oracle.com/developers/percentage-calculations-using-sql-window-functions
12. SQL Ranking Window Functions With Examples - DataLemur, accessed February 12, 2026, https://datalemur.com/sql-tutorial/sql-rank-dense_rank-row_number-window-function
13. The RANK Window Function in SQL: A Complete Guide - DbVisualizer, accessed February 12, 2026, https://www.dbvis.com/thetable/the-rank-window-function-in-sql-a-complete-guide/
14. Window functions | Databricks on AWS, accessed February 12, 2026, https://docs.databricks.com/aws/en/sql/language-manual/sql-ref-window-functions

15. OVER Clause (Transact-SQL) - SQL Server | Microsoft Learn, accessed February 12, 2026, https://learn.microsoft.com/en-us/sql/t-sql/queries/select-over-clause-transact-sql?view=sql-server-ver17
16. FIRST_VALUE (Transact-SQL) - SQL Server - Microsoft Learn, accessed February 12, 2026, https://learn.microsoft.com/en-us/sql/t-sql/functions/first-value-transact-sql?view=sql-server-ver17
17. SQL Cumulative SUM: Window Functions, Rolling Totals & Best Practices - Interview Query, accessed February 12, 2026, https://www.interviewquery.com/p/sql-cumulative-sum-guide
18. How to get the cumulative running total of rows with SQL - Oracle Blogs, accessed February 12, 2026, https://blogs.oracle.com/sql/cumulative-running-total-of-previous-rows-with-sql
19. 11 SQL Window Functions Exercises with Solutions | LearnSQL.com, accessed February 12, 2026, https://learnsql.com/blog/sql-window-functions-practice-exercises/
20. How to Restrict Results to Top N Rows per Group in SQL - Baeldung, accessed February 12, 2026, https://www.baeldung.com/sql/top-n-rows-window-functions
21. Top 15 Advanced SQL Window Functions with Real Use Cases | by ..., accessed February 12, 2026, https://medium.com/@CodeWithHannan/top-15-advanced-sql-window-functions-with-real-use-cases-5cea87060d72
22. How to Solve the Top N Per Category Problem in SQL - Aman Kharwal, accessed February 12, 2026, https://amanxai.com/2025/10/10/how-to-solve-the-top-n-per-category-problem-in-sql/
23. SQL FIRST VALUE, LAST VALUE, and NTH_VALUE – Window ..., accessed February 12, 2026, https://www.practicewindowfunctions.com/learn/first_last_nth_value
24. MySQL LEAD and LAG Window Functions: Syntax and Examples - Devart, accessed February 12, 2026, https://www.devart.com/dbforge/mysql/studio/mysql-lead-and-lag-analytical-functions.html
25. SQL Time-Series Window Functions: LEAD & LAG Tutorial - DataLemur, accessed February 12, 2026, https://datalemur.com/sql-tutorial/sql-time-series-window-function-lead-lag
26. SQL Interview Questions - DBMS - GeeksforGeeks, accessed February 12, 2026, https://www.geeksforgeeks.org/sql/sql-interview-questions/
27. SQL OFFSET (Window) Functions - LAG, LEAD, FIRST_VALUE, LAST_VALUE | Jan Zedníček, accessed February 12, 2026, https://janzednicek.cz/en/sql-offset-functions-lag-lead-first_value-last_value/
28. Calculating Month-over-Month Growth Rate · Advanced SQL - SILOTA, accessed February 12, 2026, http://www.silota.com/docs/recipes/sql-mom-growth-rate.html
29. Solving Gaps and Islands with Enhanced Window Functions, accessed February 12, 2026, https://www.itprotoday.com/innovations-of-the-2010s/solving-gaps-and-islands-with-enhanced-window-functions
30. SQL Classic Problem: Identifying Gaps and Islands Across Overlapping Date Ranges | by Halim Fauzan Edher | Analytics Vidhya | Medium, accessed February 12, 2026, https://medium.com/analytics-vidhya/sql-classic-problem-identifying-gaps-and-islands-across-overlapping-date-ranges-5681b5fcdb8

31. Gaps and Islands Across Date Ranges – SQLServerCentral, accessed February 12, 2026, https://www.sqlservercentral.com/blogs/gaps-and-islands-across-date-ranges
32. Solving the Gaps and Islands Problem in SQL - VLDB Solutions, accessed February 12, 2026, https://vldbsolutions.com/blog/gaps-and-islands-sql-solution/
33. Session Window - Stream Analytics Query - Microsoft Learn, accessed February 12, 2026, https://learn.microsoft.com/en-us/stream-analytics-query/session-window-azure-stream-analytics
34. Sessionizing Log Data Using SQL - randyzwitch.com, accessed February 12, 2026, https://randyzwitch.com/sessionizing-log-data-sql/
35. FIRST_VALUE function and LAST_VALUE function examples - Sybase Infocenter, accessed February 12, 2026, https://infocenter.sybase.com/help/topic/com.sybase.help.sqlanywhere.12.0.1/dbusage/ug-olap-s-3858836.html
36. Dynamic Pareto analysis in Power BI - SQLBI, accessed February 12, 2026, https://www.sqlbi.com/articles/dynamic-pareto-analysis-in-power-bi/
37. Creating Pareto Charts to visualize the 80/20 principle · Advanced ..., accessed February 12, 2026, http://www.silota.com/docs/recipes/sql-pareto-analysis-80-20-principle.html
38. SQL and the 80/20 Rule | NHS Excel, accessed February 12, 2026, https://nhsexcel.com/sql-and-the-8020-rule
39. Replicate Excel Formula by using Domo to Calculate 80/20, accessed February 12, 2026, https://community-forums.domo.com/main/discussion/51533/replicate-excel-formula-by-using-domo-to-calculate-80-20
40. Calculate the Churn Rate - With SQL, accessed February 12, 2026, https://www.fightchurnwithdata.com/2019/01/01/calculate-churn-rate-sql/
41. How do I calculate user retention and churn rate? | The Mode Playbook, accessed February 12, 2026, https://mode.com/help/articles/cohort-analysis-for-customer-retention-and-churn-rate/
42. Product Analytics with MotherDuck & DuckDB: A Practical SQL Guide, accessed February 12, 2026, https://motherduck.com/learn-more/product-analytics-motherduck-duckdb/
43. “Ever hear of SQL 'Gaps & Islands'? They sound weird, but they show up in interviews a lot ” : r/learnSQL - Reddit, accessed February 12, 2026, https://www.reddit.com/r/learnSQL/comments/1nikstk/ever_hear_of_sql_gaps_islands_they_sound_weird/
44. 30 Advanced SQL Interview Questions & Answers - UPES Online, accessed February 12, 2026, https://upesonline.ac.in/blog/advanced-sql-interview-questions

◇ 1. Print Numbers from 1 to 100 (Without Using a Table)

❓ Question

Print numbers from 1 to 100 without using an existing numbers table.

☑ Solution (MySQL / PostgreSQL – Recursive CTE)

```
WITH RECURSIVE nums AS ( SELECT 1 AS n UNION ALL SELECT n + 1 FROM nums WHERE n < 100 ) SELECT * FROM nums;
```

◇ 2. Print Even Numbers from 1 to 100

❓ Question

Print only even numbers between 1 and 100.

☑ Solution

```
WITH RECURSIVE nums AS ( SELECT 1 AS n UNION ALL SELECT n + 1 FROM nums WHERE n < 100 ) SELECT n FROM nums WHERE n % 2 = 0;
```

◇ 3. Print Odd Numbers Without Using %

❓ Question

Print odd numbers between 1 and 100 without using modulus operator.

☑ Solution

```
WITH RECURSIVE nums AS ( SELECT 1 AS n UNION ALL SELECT n + 2 FROM nums WHERE n < 99 ) SELECT * FROM nums;
```

◇ 4. Find Second Highest Salary

❓ Question

Given table `employees(id, name, salary)`
Find the second highest salary.

☑ Solution 1 (Using LIMIT)

```sql
SELECT DISTINCT salary FROM employees ORDER BY salary DESC LIMIT 1 OFFSET 1;
```

☑ Solution 2 (Without LIMIT)

```sql
SELECT MAX(salary) FROM employees WHERE salary < (SELECT MAX(salary) FROM
employees);
```

◇ 5. Find Nth Highest Salary

❓ Question

Find the 5th highest salary.

☑ Solution (Using DENSE_RANK)

```sql
SELECT salary FROM ( SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC)
rnk FROM employees ) t WHERE rnk = 5;
```

◇ 6. Remove Duplicate Rows

❓ Question

Delete duplicate records but keep one.

☑ Solution (Using ROW_NUMBER)

```
DELETE FROM employees WHERE id NOT IN ( SELECT id FROM ( SELECT id,
ROW_NUMBER() OVER (PARTITION BY name, salary ORDER BY id) rn FROM employees )
t WHERE rn = 1 );
```

◇ 7. Find Employees Earning More Than Their Manager

? Question

Table: `employees(id, name, salary, manager_id)`

☑ Solution (Self Join)

```
SELECT e.name FROM employees e JOIN employees m ON e.manager_id = m.id WHERE
e.salary > m.salary;
```

◇ 8. Swap Values of Two Columns

? Question

Swap values of column A and B.

☑ Solution

```
UPDATE table_name SET A = B, B = A;
```

(Works in most SQL engines because assignment happens simultaneously.)

◇ 9. Find Consecutive Numbers

❓ Question

Find numbers appearing at least 3 times consecutively.

☑ Solution

```
SELECT DISTINCT num FROM ( SELECT num, LAG(num,1) OVER (ORDER BY id) prev1,
LAG(num,2) OVER (ORDER BY id) prev2 FROM logs ) t WHERE num = prev1 AND num =
prev2;
```

◇ 10. Running Total (Cumulative Sum)

❓ Question

Calculate running total of salary.

☑ Solution

```
SELECT name, salary, SUM(salary) OVER (ORDER BY id) AS running_total FROM
employees;
```

◇ 11. Find Duplicate Records

```
SELECT name, COUNT(*) FROM employees GROUP BY name HAVING COUNT(*) > 1;
```

◇ 12. Find Missing Numbers in Sequence

```
SELECT t1.id + 1 AS missing_number FROM numbers t1 LEFT JOIN numbers t2 ON
t1.id + 1 = t2.id WHERE t2.id IS NULL;
```

I'll assume a common table:

```
employees( id INT, name VARCHAR(100), salary INT, department_id INT,
manager_id INT, hire_date DATE ) departments( id INT, department_name
VARCHAR(100) ) orders( customer_id INT, product_id INT ) products( product_id
INT )
```

🔥 CORE ADVANCED SQL QUESTIONS (With Solutions)

1️⃣ Find Median Salary

☑ Solution (Works in most modern SQL – using window functions)

```
WITH ranked AS ( SELECT salary, ROW_NUMBER() OVER (ORDER BY salary) AS rn,
COUNT(*) OVER () AS total_count FROM employees ) SELECT AVG(salary) AS median
FROM ranked WHERE rn IN ((total_count + 1)/2, (total_count + 2)/2);
```

✔ Works for even and odd count.

2️⃣ Pivot Rows Into Columns

Example:

Convert:

Into:

| HR | IT | Sales |

☑ Solution (Conditional Aggregation)

```
SELECT SUM(CASE WHEN department_id = 1 THEN salary END) AS HR, SUM(CASE WHEN department_id = 2 THEN salary END) AS IT, SUM(CASE WHEN department_id = 3 THEN salary END) AS Sales FROM employees;
```

3️⃣ Top 3 Salaries Per Department

```
SELECT * FROM ( SELECT *, DENSE_RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) rnk FROM employees ) t WHERE rnk <= 3;
```

4️⃣ Employees Hired in Last 30 Days

```
SELECT * FROM employees WHERE hire_date >= CURRENT_DATE - INTERVAL '30 days';
```

(MySQL: `DATE_SUB(CURDATE(), INTERVAL 30 DAY)`)

5️⃣ Percentage Contribution Per Department

```
SELECT department_id, SUM(salary) AS dept_salary, ROUND( SUM(salary) * 100.0 / SUM(SUM(salary)) OVER (), 2 ) AS percentage FROM employees GROUP BY department_id;
```

6️⃣ Gaps and Islands Problem

Find consecutive salary sequences.

```
WITH numbered AS ( SELECT salary, salary - ROW_NUMBER() OVER (ORDER BY
salary) AS grp FROM employees ) SELECT MIN(salary) AS start_range,
MAX(salary) AS end_range FROM numbered GROUP BY grp;
```

---

7 Employees With Same Salary in Same Department

```
SELECT department_id, salary, COUNT(*) FROM employees GROUP BY department_id,
salary HAVING COUNT(*) > 1;
```

---

8 FizzBuzz in SQL

```
WITH RECURSIVE nums AS ( SELECT 1 AS n UNION ALL SELECT n+1 FROM nums WHERE n
< 100 ) SELECT CASE WHEN n % 15 = 0 THEN 'FizzBuzz' WHEN n % 3 = 0 THEN
'Fizz' WHEN n % 5 = 0 THEN 'Buzz' ELSE CAST(n AS VARCHAR) END FROM nums;
```

---

9 First Non-Repeated Value

```
SELECT name FROM employees GROUP BY name HAVING COUNT(*) = 1 LIMIT 1;
```

---

10 Cumulative Percentage

```
SELECT salary, SUM(salary) OVER (ORDER BY salary) * 100.0 / SUM(salary) OVER
() AS cumulative_percentage FROM employees;
```

🔥 REAL INTERVIEW-LEVEL TRICKY ONES

1️⃣ Print Pattern

Output:

```
1
12
123
1234
12345
```

```
WITH RECURSIVE nums AS ( SELECT 1 AS n UNION ALL SELECT n+1 FROM nums WHERE n < 5 ) SELECT STRING_AGG(n::text, '' ORDER BY n) FROM nums GROUP BY n;
```

2️⃣ Customers Who Bought All Products

```
SELECT customer_id FROM orders GROUP BY customer_id HAVING COUNT(DISTINCT product_id) = (SELECT COUNT(*) FROM products);
```

3️⃣ Department With Highest Average Salary

```
SELECT department_id FROM employees GROUP BY department_id ORDER BY AVG(salary) DESC LIMIT 1;
```

4️⃣ Employee With Same Salary as Manager

```
SELECT e.name FROM employees e JOIN employees m ON e.manager_id = m.id WHERE e.salary = m.salary;
```

5️⃣ Top 2 Salaries Per Department (Without Window Function)

```
SELECT e1.* FROM employees e1 WHERE 2 > ( SELECT COUNT(DISTINCT e2.salary) FROM employees e2 WHERE e2.department_id = e1.department_id AND e2.salary > e1.salary );
```

🔥 This is a classic Amazon-style question.

I'll assume common tables:

```
employees(id, name, salary, department_id, manager_id, hire_date)
logins(user_id, login_date) orders(order_id, user_id, product_id, order_date, amount) transactions(user_id, transaction_date, amount) products(product_id, category_id) invoices(store_id, invoice_no)
```

🔥 PART 1 — 20 HARDEST SQL QUESTIONS (WITH SOLUTIONS)

1️⃣ Median Salary Per Department

```sql
WITH ranked AS ( SELECT department_id, salary, ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary) rn, COUNT(*) OVER (PARTITION BY department_id) cnt FROM employees ) SELECT department_id, AVG(salary) AS median FROM ranked WHERE rn IN ((cnt+1)/2, (cnt+2)/2) GROUP BY department_id;
```

2️⃣ Salary > Dept Avg but < Company Avg

```sql
WITH dept_avg AS ( SELECT department_id, AVG(salary) d_avg FROM employees GROUP BY department_id ), company_avg AS ( SELECT AVG(salary) c_avg FROM employees ) SELECT e.* FROM employees e JOIN dept_avg d ON e.department_id = d.department_id CROSS JOIN company_avg c WHERE e.salary > d.d_avg AND e.salary < c.c_avg;
```

3️⃣ 3 Consecutive Login Days

```sql
WITH grp AS ( SELECT user_id, login_date, login_date - ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY login_date) AS grp FROM logins ) SELECT user_id FROM grp GROUP BY user_id, grp HAVING COUNT(*) >= 3;
```

4️⃣ Products Sold Every Day Last 30 Days

```sql
SELECT product_id FROM orders WHERE order_date >= CURRENT_DATE - INTERVAL '30 days' GROUP BY product_id HAVING COUNT(DISTINCT order_date) = 30;
```

5 Month-over-Month Growth %

```sql
WITH monthly AS ( SELECT DATE_TRUNC('month', order_date) m, SUM(amount)
revenue FROM orders GROUP BY m ) SELECT m, revenue, (revenue - LAG(revenue)
OVER (ORDER BY m)) * 100.0 / LAG(revenue) OVER (ORDER BY m) AS growth_pct
FROM monthly;
```

---

6 Detect Invoice Gaps Per Store

```sql
SELECT store_id, invoice_no + 1 AS missing_invoice FROM invoices i1 LEFT JOIN
invoices i2 ON i1.store_id = i2.store_id AND i1.invoice_no + 1 =
i2.invoice_no WHERE i2.invoice_no IS NULL;
```

---

7 Earn More Than Manager's Manager

```sql
SELECT e.name FROM employees e JOIN employees m ON e.manager_id = m.id JOIN
employees mm ON m.manager_id = mm.id WHERE e.salary > mm.salary;
```

---

8 Top 10% Customers by Revenue

```sql
WITH ranked AS ( SELECT user_id, SUM(amount) total, NTILE(10) OVER (ORDER BY
SUM(amount) DESC) bucket FROM orders GROUP BY user_id ) SELECT * FROM ranked
WHERE bucket = 1;
```

---

9 Longest Consecutive Transaction Streak

```sql
WITH grp AS ( SELECT user_id, transaction_date, transaction_date -
ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY transaction_date) grp FROM
transactions ) SELECT user_id, COUNT(*) longest_streak FROM grp GROUP BY
user_id, grp ORDER BY longest_streak DESC;
```

🔟 Rolling 7-Day Revenue Per Product

```sql
SELECT product_id, order_date, SUM(amount) OVER ( PARTITION BY product_id
ORDER BY order_date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW ) rolling_7_day
FROM orders;
```

1️⃣1️⃣ Remove Duplicates Keep Latest

```sql
DELETE FROM employees WHERE id NOT IN ( SELECT id FROM ( SELECT id,
ROW_NUMBER() OVER ( PARTITION BY name, salary ORDER BY hire_date DESC ) rn
FROM employees ) t WHERE rn = 1 );
```

1️⃣2️⃣ Churned Customers (No Purchase 90 Days)

```sql
SELECT user_id FROM orders GROUP BY user_id HAVING MAX(order_date) <
CURRENT_DATE - INTERVAL '90 days';
```

1️⃣3️⃣ Retention Rate MoM

```sql
WITH monthly_users AS ( SELECT DATE_TRUNC('month', order_date) m, user_id
FROM orders GROUP BY m, user_id ) SELECT m, COUNT(user_id) AS active_users,
COUNT(user_id) * 100.0 / LAG(COUNT(user_id)) OVER (ORDER BY m) retention_rate
FROM monthly_users GROUP BY m;
```

1️⃣4️⃣ Same Salary + Hire Date + Dept

```sql
SELECT department_id, salary, hire_date FROM employees GROUP BY
department_id, salary, hire_date HAVING COUNT(*) > 1;
```

1️⃣5️⃣ Salary Jump > 50%

```sql
SELECT id, salary, LAG(salary) OVER (PARTITION BY id ORDER BY hire_date)
prev_salary FROM employees WHERE salary > 1.5 * LAG(salary) OVER (PARTITION
BY id ORDER BY hire_date);
```

1️⃣6️⃣ Bought All Categories

```sql
SELECT user_id FROM orders o JOIN products p ON o.product_id = p.product_id
GROUP BY user_id HAVING COUNT(DISTINCT category_id) = (SELECT COUNT(DISTINCT
category_id) FROM products);
```

1️⃣7️⃣ First Product Per Customer

```sql
SELECT * FROM ( SELECT *, ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY
order_date) rn FROM orders ) t WHERE rn = 1;
```

1️⃣8️⃣ Second Highest Salary (No subquery, no window)

```sql
SELECT MAX(salary) FROM employees WHERE salary < (SELECT MAX(salary) FROM
employees);
```

(Practically impossible without subquery in pure SQL standard.)

1️⃣9️⃣ Dept Payroll > 40% of Company

```
SELECT department_id FROM employees GROUP BY department_id HAVING SUM(salary)
> 0.4 * (SELECT SUM(salary) FROM employees);
```

2️⃣0️⃣ Gaps & Islands (Transaction Streak)

```
WITH grp AS ( SELECT user_id, transaction_date, transaction_date -
ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY transaction_date) grp FROM
transactions ) SELECT user_id, MIN(transaction_date) start_date,
MAX(transaction_date) end_date FROM grp GROUP BY user_id, grp;
```

Assume common tables:

```
employees(id, name, salary, department_id, hire_date) orders(order_id,
user_id, product_id, order_date, amount) logins(user_id, login_date)
transactions(user_id, transaction_date, amount) prices(product_id,
price_date, price)
```

◇ ROW_NUMBER / RANK / DENSE_RANK (1–10)

---

1 Rank employees by salary

```
SELECT id, name, salary, RANK() OVER (ORDER BY salary DESC) AS salary_rank
FROM employees;
```

---

2 Top 3 salaries per department

```
SELECT * FROM ( SELECT *, DENSE_RANK() OVER ( PARTITION BY department_id
ORDER BY salary DESC ) rnk FROM employees ) t WHERE rnk <= 3;
```

---

3 Remove duplicates using row_number

```
DELETE FROM employees WHERE id NOT IN ( SELECT id FROM ( SELECT id,
ROW_NUMBER() OVER ( PARTITION BY name, salary ORDER BY hire_date DESC ) rn
FROM employees ) t WHERE rn = 1 );
```

---

4 2nd highest salary per department

```
SELECT * FROM ( SELECT *, DENSE_RANK() OVER ( PARTITION BY department_id
ORDER BY salary DESC ) rnk FROM employees ) t WHERE rnk = 2;
```

5️⃣ Rank customers by spending

```sql
SELECT user_id, SUM(amount) total, RANK() OVER (ORDER BY SUM(amount) DESC)
rnk FROM orders GROUP BY user_id;
```

6️⃣ Latest order per customer

```sql
SELECT * FROM ( SELECT *, ROW_NUMBER() OVER ( PARTITION BY user_id ORDER BY
order_date DESC ) rn FROM orders ) t WHERE rn = 1;
```

7️⃣ Earliest hire per department

```sql
SELECT * FROM ( SELECT *, ROW_NUMBER() OVER ( PARTITION BY department_id
ORDER BY hire_date ) rn FROM employees ) t WHERE rn = 1;
```

8️⃣ Nth order per user (e.g., 3rd)

```sql
SELECT * FROM ( SELECT *, ROW_NUMBER() OVER ( PARTITION BY user_id ORDER BY
order_date ) rn FROM orders ) t WHERE rn = 3;
```

9️⃣ Bottom 5% performers

```sql
SELECT * FROM ( SELECT *, NTILE(20) OVER (ORDER BY salary ASC) bucket FROM
employees ) t WHERE bucket = 1;
```

🔟 Identify ties in ranking

```sql
SELECT salary FROM employees GROUP BY salary HAVING COUNT(*) > 1;
```

◇ LAG / LEAD (11–20)

1️⃣1️⃣ Compare salary with previous employee

```sql
SELECT id, salary, LAG(salary) OVER (ORDER BY id) prev_salary FROM employees;
```

1️⃣2️⃣ Salary difference from previous year

```sql
SELECT id, hire_date, salary, salary - LAG(salary) OVER (PARTITION BY id
ORDER BY hire_date) diff FROM employees;
```

1️⃣3️⃣ Detect login gaps

```sql
SELECT user_id, login_date, login_date - LAG(login_date) OVER (PARTITION BY
user_id ORDER BY login_date) gap FROM logins;
```

1️⃣4️⃣ 3-day consecutive login

```sql
WITH grp AS ( SELECT user_id, login_date, login_date - ROW_NUMBER() OVER
(PARTITION BY user_id ORDER BY login_date) grp FROM logins ) SELECT user_id
FROM grp GROUP BY user_id, grp HAVING COUNT(*) >= 3;
```

---

1 5 Identify price changes

```sql
SELECT product_id, price_date, price, LAG(price) OVER (PARTITION BY
product_id ORDER BY price_date) prev_price FROM prices;
```

---

1 6 Day-over-day growth

```sql
SELECT order_date, SUM(amount) revenue, SUM(amount) - LAG(SUM(amount)) OVER
(ORDER BY order_date) growth FROM orders GROUP BY order_date;
```

---

1 7 Repeat purchase within 7 days

```sql
SELECT * FROM ( SELECT *, LEAD(order_date) OVER (PARTITION BY user_id ORDER
BY order_date) next_order FROM orders ) t WHERE next_order <= order_date +
INTERVAL '7 days';
```

---

1 8 Detect churn event

```sql
SELECT user_id, MAX(order_date) last_order FROM orders GROUP BY user_id
HAVING MAX(order_date) < CURRENT_DATE - INTERVAL '90 days';
```

1️⃣9️⃣ Compare order with next order

```sql
SELECT *, LEAD(amount) OVER (PARTITION BY user_id ORDER BY order_date)
next_amount FROM orders;
```

2️⃣0️⃣ Stock price drops > 5%

```sql
SELECT *, (price - LAG(price) OVER (PARTITION BY product_id ORDER BY
price_date)) / LAG(price) OVER (PARTITION BY product_id ORDER BY price_date)
* 100 AS change_pct FROM prices WHERE (price - LAG(price) OVER (PARTITION BY
product_id ORDER BY price_date)) / LAG(price) OVER (PARTITION BY product_id
ORDER BY price_date) < -0.05;
```

◇ Running Totals (21–30)

2️⃣1️⃣ Running total revenue

```sql
SELECT order_date, SUM(amount) OVER (ORDER BY order_date) running_total FROM
orders;
```

2️⃣2️⃣ Running total per department

```sql
SELECT department_id, hire_date, SUM(salary) OVER (PARTITION BY department_id
ORDER BY hire_date) FROM employees;
```

2️⃣3️⃣ Cumulative revenue %

```sql
SELECT order_date, SUM(amount) OVER (ORDER BY order_date) * 100.0 /
SUM(amount) OVER () AS cumulative_pct FROM orders;
```

2️⃣4️⃣ Cumulative orders per user

```sql
SELECT user_id, order_date, COUNT(*) OVER (PARTITION BY user_id ORDER BY
order_date) FROM orders;
```

2️⃣5️⃣ Running average salary

```sql
SELECT hire_date, AVG(salary) OVER (ORDER BY hire_date) FROM employees;
```

2️⃣6️⃣ Rolling 7-day average

```sql
SELECT order_date, AVG(amount) OVER (ORDER BY order_date ROWS BETWEEN 6
PRECEDING AND CURRENT ROW) FROM orders;
```

2️⃣7️⃣ Rolling 30-day sum

```sql
SELECT order_date, SUM(amount) OVER (ORDER BY order_date ROWS BETWEEN 29
PRECEDING AND CURRENT ROW) FROM orders;
```

2️⃣8️⃣ Rolling count per product

```sql
SELECT product_id, order_date, COUNT(*) OVER ( PARTITION BY product_id ORDER BY order_date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW ) FROM orders;
```

2️⃣9️⃣ Running profit per store

```sql
SELECT store_id, order_date, SUM(profit) OVER (PARTITION BY store_id ORDER BY order_date) FROM sales;
```

3️⃣0️⃣ Cumulative contribution %

```sql
SELECT department_id, SUM(salary) * 100.0 / SUM(SUM(salary)) OVER () FROM employees GROUP BY department_id;
```

◇ NTILE (31–35)

3️⃣1️⃣ Quartiles

```sql
SELECT *, NTILE(4) OVER (ORDER BY salary DESC) quartile FROM employees;
```

3️⃣2️⃣ Top decile

```sql
SELECT * FROM ( SELECT *, NTILE(10) OVER (ORDER BY salary DESC) bucket FROM
employees ) t WHERE bucket = 1;
```

---

3 3 5 revenue buckets

```sql
SELECT product_id, NTILE(5) OVER (ORDER BY SUM(amount) DESC) FROM orders
GROUP BY product_id;
```

---

3 4 Bottom 20%

```sql
SELECT * FROM ( SELECT *, NTILE(5) OVER (ORDER BY salary ASC) bucket FROM
employees ) t WHERE bucket = 1;
```

---

3 5 Assign risk categories

```sql
SELECT customer_id, CASE NTILE(3) OVER (ORDER BY credit_score) WHEN 1 THEN
'High Risk' WHEN 2 THEN 'Medium' ELSE 'Low' END FROM customers;
```

◇ FIRST_VALUE / LAST_VALUE (36–40)

---

3 6 First purchase

```sql
SELECT DISTINCT user_id, FIRST_VALUE(product_id) OVER (PARTITION BY user_id
ORDER BY order_date) FROM orders;
```

---

3 7 Last login

```sql
SELECT DISTINCT user_id, LAST_VALUE(login_date) OVER (PARTITION BY user_id
ORDER BY login_date ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
FROM logins;
```

---

3 8 First salary

```sql
SELECT id, FIRST_VALUE(salary) OVER (PARTITION BY id ORDER BY hire_date) FROM
employees;
```

---

3 9 First vs latest salary

```sql
SELECT id, FIRST_VALUE(salary) OVER (PARTITION BY id ORDER BY hire_date)
first_salary, LAST_VALUE(salary) OVER (PARTITION BY id ORDER BY hire_date
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) last_salary FROM
employees;
```

---

4 0 First churn event

```sql
SELECT user_id, MIN(order_date) FROM orders GROUP BY user_id;
```

◇ ADVANCED (41–50)

4️⃣1️⃣ Consecutive transaction streak

Use gaps & islands pattern (already shown above).

4️⃣2️⃣ Salary anomalies

```
SELECT *, salary / LAG(salary) OVER (PARTITION BY id ORDER BY hire_date)
ratio FROM employees WHERE salary > 1.5 * LAG(salary) OVER (PARTITION BY id
ORDER BY hire_date);
```

4️⃣3️⃣ Trend reversal

```
SELECT *, SIGN(amount - LAG(amount) OVER (ORDER BY order_date)) trend FROM
orders;
```

4️⃣4️⃣ Longest inactivity

```
SELECT user_id, MAX(login_date - LAG(login_date) OVER (PARTITION BY user_id
ORDER BY login_date)) FROM logins GROUP BY user_id;
```

---

4 5 Moving window difference

```sql
SELECT order_date, SUM(amount) OVER (ORDER BY order_date ROWS 6 PRECEDING) -
SUM(amount) OVER (ORDER BY order_date ROWS 13 PRECEDING) diff FROM orders;
```

---

4 6 Detect gaps invoice

Use LAG(invoice_no)

---

4 7 Retention cohort

```sql
SELECT DATE_TRUNC('month', first_order) cohort, COUNT(user_id) FROM ( SELECT
user_id, MIN(order_date) first_order FROM orders GROUP BY user_id ) t GROUP
BY cohort;
```

---

4 8 Time between events

```sql
SELECT user_id, order_date - LAG(order_date) OVER (PARTITION BY user_id ORDER
BY order_date) FROM orders;
```

---

4 9 Duplicate detection using lag

```sql
SELECT * FROM ( SELECT *, LAG(name) OVER (ORDER BY name) prev_name FROM
employees ) t WHERE name = prev_name;
```

5️⃣0️⃣ Cumulative growth rate

```
SELECT order_date, EXP(SUM(LN(1 + growth_rate)) OVER (ORDER BY order_date))
cumulative_growth FROM growth_table;
```