ChatGPT

# SQL Interview Guide: Finance & Loan Use Cases

This guide provides a structured framework for intermediate-to-advanced SQL interview questions, focusing on finance and loan-related scenarios. It covers key SQL concepts commonly tested and includes example problems with solutions, tailored to typical use cases in banking/loan analytics. Each section below breaks down a concept with an explanation, followed by a sample problem (with tables and context), a commented SQL solution, and the expected output. This structured approach helps data analysts prepare effectively for SQL interviews in the finance industry.

## 1. SQL Concepts Commonly Tested (Finance Focus)

### Joins (INNER, LEFT, RIGHT, FULL OUTER, SELF)

**Explanation:** Joins combine rows from two or more tables based on a related column between them [1]. Key types include:

- **INNER JOIN:** returns only the rows where there is a match in **both** joined tables [1]. (Unmatched rows are excluded.)
- **LEFT JOIN (LEFT OUTER):** returns **all** rows from the left table and the matching rows from the right table; if no match, right-side columns come out as NULL [2] [3].
- **RIGHT JOIN (RIGHT OUTER):** returns all rows from the right table, plus matching rows from the left (NULL for missing matches) [2] [3].
- **FULL OUTER JOIN:** returns all rows when there is a match in **either** table; unmatched rows from either side are included with NULLs for missing values [4].
- **SELF JOIN:** joins a table to itself (treating one instance as "left" and another as "right") to compare rows within the same table [5]. This is useful for hierarchical data or comparing rows in one table.

Joins are fundamental in finance databases (e.g., linking customers to their loans, or loans to payments). Mastering join logic ensures you can retrieve integrated information from multiple tables.

### Aggregations (SUM, COUNT, AVG, GROUP BY, HAVING)

**Explanation:** Aggregation functions summarize or count data across multiple rows. Common functions include `SUM()` for totals, `COUNT()` for counts, `AVG()` for averages, `MIN()` / `MAX()` for extrema, etc. These are used alongside `GROUP BY` to aggregate rows sharing a common key (e.g., summing loan amounts per customer) [6]. The `GROUP BY` clause groups rows that have the same values in specified columns into summary rows (e.g., total loans per customer) [6].

A `HAVING` clause is often used with `GROUP BY` to filter groups based on aggregate criteria (because `WHERE` cannot be used with aggregate results) [7]. For example, you might use `HAVING COUNT(*) > 1` to find groups (e.g., customers) with more than one record (e.g., multiple loans). In finance data, aggregations help answer questions like total interest collected, average loan amount by type, count of loans per status, etc.

## Window Functions (ROW_NUMBER, RANK, DENSE_RANK, LAG, LEAD, PARTITION BY)

**Explanation:** Window functions perform calculations across a set of rows related to the current row, without collapsing those rows into a single output as aggregates do [8]. Using a `OVER ( ... )` clause, you can define a "window" (which may partition the data and order it) to compute running totals, rankings, or look at neighboring row values. Finance use cases include ranking customers by loan amounts, computing running balances, or comparing current and previous payments.

Key window functions:

- **ROW_NUMBER():** Assigns a sequential number to each row in a window (essentially, an ordered list of rows) [9]. This is often used for numbering or to pick top-N records in each group.
- **RANK():** Assigns a rank number to each row in an ordered partition, with ties receiving the same rank and leaving gaps in the ranking sequence [10] (e.g., if two loans tie for largest amount, both get rank 1 and the next rank is 3).
- **DENSE_RANK():** Similar to RANK, but without gaps in the sequence for ties [11] (if two loans tie for rank 1, the next rank is 2, not 3).
- **LAG()/LEAD():** These functions allow access to a prior or subsequent row's value from the current row's context [12]. For example, `LAG(payment_date)` can fetch the previous payment date of the same customer (when partitioned by customer and ordered by date), which helps in time-series analysis like finding the gap between payments. `LEAD` does the same for the next row.
- **PARTITION BY:** A sub-clause used within the `OVER` clause to break the data into partitions (groups) for the window calculation. For instance, `PARTITION BY customer_id ORDER BY loan_date` will perform the window function within each customer's partition, rather than over the whole table.

Window functions are powerful for finance analytics — e.g., cumulative interest calculations, ranking investment returns, or detecting trends in sequential loan payments.

## Subqueries and CTEs (Common Table Expressions)

**Explanation:** A **subquery** is a query nested inside another query (in `SELECT`, `FROM`, or `WHERE` clauses) to provide interim results for the outer query. For example, a subquery can fetch the average loan size which the outer query then uses to filter customers above that average. A **CTE** (Common Table Expression) is a named subquery defined using a WITH clause, which can be referenced like a temporary table within the main query [13] [14]. CTEs make complex queries more readable and can be reused multiple times in the query, whereas a subquery is typically written inline and used once [15].

Both subqueries and CTEs are useful for breaking down complex logic. In finance, you might use a subquery to filter customers who meet certain criteria (e.g., have a rejected loan application) or a CTE to stage a complicated calculation (like computing each loan's risk score) and then query from it. Being comfortable with when to use subqueries vs. CTEs (for clarity or performance) is often tested in advanced SQL interviews.

### CASE Statements

**Explanation:** The `CASE` expression allows conditional logic in SQL, similar to if/then/else in other languages. It evaluates a series of conditions and returns a value for the first condition that is true [16]. A typical syntax is:

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE default_result
END
```

You can use CASE in SELECT (to create categorized fields), in ORDER BY (for custom sorting), or even in WHERE/HAVING clauses. In a finance context, CASE is useful for classifying loans (e.g., flagging loans as *"Overdue"* vs *"On Time"*), bucketing customers by credit score ranges, or calculating fields like interest rate category based on conditions.

### String and Date Functions (CONCAT, FORMAT, DATEADD, DATEDIFF, etc.)

**Explanation:** Finance databases heavily use dates (for transactions, payments, etc.) and strings (customer names, loan IDs). SQL provides many functions to manipulate these:

- **String Functions:** e.g., `CONCAT(str1, str2, ...)` to merge strings, `SUBSTRING` to extract part of a string, `UPPER` / `LOWER` to change case, `TRIM` to remove whitespace. For instance, `CONCAT(first_name, ' ', last_name)` can produce a full name. These are useful for creating labels or merging fields (like combining address components or formatting IDs).
- **Date Functions:** e.g., `DATEADD(interval, value, date)` to add a time interval (days, months, etc.) to a date, `DATEDIFF(interval, start, end)` to calculate the difference between two dates in given units (e.g., days between payment date and due date), `DATEPART` or `EXTRACT` to get a part of a date (like year or month), and database-specific formatting functions (like `FORMAT(date, 'yyyy-MM-dd')` in T-SQL or `TO_CHAR` in Oracle) to display dates in a certain format. These functions help in calculating overdue days, scheduling due dates, or grouping events by month/quarter.

For example, `DATEDIFF(day, DueDate, PaymentDate)` could compute how many days late a payment was. Knowing these functions allows analysts to perform time calculations (interest accrual periods, age of accounts, etc.) and string manipulations (like anonymizing or standardizing data) directly in SQL.

### Performance Considerations (Indexes, Query Execution Plans)

**Explanation:** In high-stakes finance databases, performance matters. Interviewers might not expect deep DB administration knowledge, but understanding basics can set you apart. Key points include:

- **Indexes:** Indexes are data structures that improve data retrieval speed on columns used in joins, filters, or sorting. For example, indexing the `CustomerID` on a loans table can speed up join

operations with a customer table. The right indexes **can significantly improve query performance** by avoiding full table scans, though over-indexing can hurt write performance [17]. Be prepared to explain how an index on frequently queried fields (like a composite index on `(CustomerID, LoanDate)`) helps queries.

- **Query Execution Plans:** These show how the SQL engine executes your query (which indexes are used, join methods, etc.). An interviewer might ask how you would optimize a slow query. A good approach is: *check the execution plan for bottlenecks (like full scans or missing index suggestions), then add or adjust indexes, or refactor the query if needed* [18]. Demonstrating awareness of using `EXPLAIN` (in MySQL) or viewing the execution plan in SQL Server shows you understand performance tuning.
- **Query Optimization:** Writing efficient SQL includes selecting only needed columns, avoiding unnecessary subqueries or loops, and using appropriate clauses (e.g., using JOINs instead of subselects when possible, filtering early with WHERE). In finance data, where tables can be large (millions of transactions), these considerations ensure your queries run in reasonable time.

While pure SQL interviews might not dive deeply into indexing or plans, you should be ready to answer conceptual questions on how to improve a query or why a certain approach is more efficient.

---

## 2. Example Problems and Solutions (Finance/Loan Use Cases)

Below are example SQL problems typical in a finance/loan analytics context. Each problem is presented with a scenario and relevant tables, followed by a sample solution and expected result. These examples illustrate the concepts above in real-world tasks.

### Example 1: Finding Overdue Loan Payments

**Concepts:** Date functions, CASE/conditional logic, possibly JOINs.
**Scenario:** You have a `Loans` table tracking each loan's due date and payment status. You need to find loans that are overdue (the payment was received after the due date, or has not been received and is past due). This helps the finance team identify delinquent loans.

**Table Structure:** `Loans` table (simplified)
- **LoanID** (int) – unique loan identifier
- **CustomerID** (int) – reference to the borrower
- **LoanDate** (date) – date the loan was issued
- **DueDate** (date) – due date for full repayment
- **PaymentDate** (date) – date the loan was fully paid (NULL if not paid yet)
- **Amount** (decimal) – loan amount

**Sample Problem:** *"List all loans that are overdue, and how many days overdue they are. A loan is considered overdue if the PaymentDate is later than the DueDate, or if the loan is unpaid (PaymentDate is NULL) and the current date is past the DueDate."* We will assume "today" as a specific date for reproducibility (e.g., '2025-03-15').

**SQL Solution (with comments):**

```sql
SELECT
  LoanID,
  CustomerID,
  DueDate,
  PaymentDate,
  -- Calculate days overdue. If not paid, use current date for calculation.
  DATEDIFF(day, DueDate, COALESCE(PaymentDate, '2025-03-15')) AS DaysOverdue,
  -- Mark status as 'Overdue' or 'On Time' for clarity
  CASE
    WHEN PaymentDate IS NULL AND '2025-03-15' > DueDate THEN 'Overdue'
    WHEN PaymentDate > DueDate THEN 'Overdue'
    ELSE 'On Time'
  END AS PaymentStatus
FROM Loans
WHERE
  -- Overdue criteria: either unpaid past due, or paid after due date
  (PaymentDate IS NULL AND '2025-03-15' > DueDate)
   OR (PaymentDate IS NOT NULL AND PaymentDate > DueDate);
```

**Explanation:** We use `COALESCE(PaymentDate, '2025-03-15')` to treat an unpaid loan as if it were paid today, thus `DATEDIFF(day, DueDate, ...)` gives the number of days since due. The `WHERE` clause filters only loans that are actually overdue by either condition. The `CASE` in the SELECT isn't strictly necessary, but it explicitly labels the loans as "Overdue". (In an interview, you might explain that simply checking the conditions in WHERE already ensures all listed loans are overdue.) The query can be extended to join with a `Customers` table if we need customer names, but here we focus on identifying the loans.

**Expected Output:** (Example)

| LoanID | CustomerID | DueDate | PaymentDate | DaysOverdue | PaymentStatus |
|---|---|---|---|---|---|
| 1001 | 501 | 2025-01-15 | 2025-02-01 | 17 | Overdue |
| 1005 | 505 | 2025-02-20 | *NULL* | 23 | Overdue |

In this sample, Loan 1001 was paid 17 days late (due Jan 15, paid Feb 1), and Loan 1005 is unpaid for 23 days past its due date (due Feb 20, as of Mar 15 it's still not paid). Both are flagged as Overdue. (Loans paid on or before the due date would be excluded by the query.)

### Example 2: Identifying Customers with Multiple Active Loans

**Concepts:** Aggregation (`COUNT`), `GROUP BY`, `HAVING`, possibly self-join.
**Scenario:** A financial analyst wants to find customers who currently have more than one active loan. In many lending policies, having multiple concurrent loans might be unusual or risky. The `Loans` table has a status for each loan (e.g., 'Active' vs 'Closed'). We need to list customers with multiple active loans.

**Table Structure:** `Loans` table (relevant columns)
- **LoanID** – unique loan ID
- **CustomerID** – borrower's ID
- **Status** – status of loan ('Active', 'Closed', etc.)

**Sample Problem:** *"Find all customers who have more than one active loan. Return the customer ID and the number of active loans they have."*

**SQL Solution:**

```
SELECT
  CustomerID,
  COUNT(*) AS ActiveLoanCount
FROM Loans
WHERE Status = 'Active'
GROUP BY CustomerID
HAVING COUNT(*) > 1;
```

**Explanation:** We filter the `Loans` table to only include rows where `Status = 'Active'`. Then we group by `CustomerID` and count loans per customer. The `HAVING COUNT(*) > 1` ensures we only keep groups (customers) with more than one active loan. This directly gives the list of customer IDs with their active loan count. In an interview, you might also mention alternative approaches: e.g., a self-join of the `Loans` table with itself on `CustomerID` where both loans are active and different loan IDs (then deduplicate the customer), but the GROUP BY method is more straightforward and efficient.

**Expected Output:** (Example)

| CustomerID | ActiveLoanCount |
|---|---|
| 501 | 2 |
| 722 | 3 |

This result indicates Customer 501 has 2 active loans, and Customer 722 has 3 active loans, etc. (No other customers have more than one active loan.)

### Example 3: Calculating Total Interest Paid per Customer

**Concepts:** Aggregation (`SUM`), `GROUP BY`, arithmetic expressions, possibly JOIN if interest is in another table.
**Scenario:** We want to calculate how much interest each customer has paid on their loans in total. Suppose the `Loans` table contains an `InterestPaid` column (the total interest amount the customer paid for that loan). Alternatively, if we have principal and interest rate, we could calculate interest — but here we'll assume interest is tracked per loan for simplicity.

**Table Structure:** `Loans` table (relevant columns)
- **CustomerID** – borrower's ID
- **InterestPaid** (decimal) – total interest amount paid on that loan (could be calculated from rate and amount over time)

**Sample Problem:** *"For each customer, calculate the total interest they have paid across all their loans. Provide the customer ID and total interest paid."*

**SQL Solution:**

```
SELECT
  CustomerID,
  SUM(InterestPaid) AS TotalInterestPaid
FROM Loans
GROUP BY CustomerID;
```

**Explanation:** This query groups the loans by CustomerID and sums up the `InterestPaid` for each customer. It assumes all loan records (including closed loans) are in the `Loans` table with their interest. If interest needed to be derived (say from rate and amount), one could sum an expression like `SUM(Amount * InterestRate)` (assuming InterestRate is in a comparable unit or divided appropriately by 100 for percentage). The result gives the total interest revenue per customer. In a finance interview, this could lead to follow-up questions about interest calculation or whether to include only completed loans, etc., but the SQL pattern remains a GROUP BY and SUM.

**Expected Output:** (Example)

| CustomerID | TotalInterestPaid |
|---|---|
| 501 | 750.00 |
| 502 | 1200.50 |
| 503 | 300.00 |
| ... | ... |

Each row shows how much interest a customer paid in total. For instance, customer 502 paid \$1,200.50 in interest across all loans (perhaps they had multiple or large loans).

## Example 4: Ranking Customers by Loan Amount

**Concepts:** Window functions (`RANK` or `DENSE_RANK`), possibly subquery/CTE, `ORDER BY` in window.
**Scenario:** We want to rank customers based on the total amount they have borrowed, from highest to lowest. This can help identify top borrowers. We will compute each customer's total loan amount and then assign ranks (1 = highest total loan amount).

**Table Structure:** `Loans` (relevant columns)
- **CustomerID** – borrower's ID
- **Amount** (decimal) – loan principal amount

**Sample Problem:** *"Which customers have borrowed the most? Calculate the sum of loan amounts for each customer and rank the customers by this total loan amount (1 = highest total amount). Return the customer ID, total loan amount, and their rank."*

**SQL Solution:** Using a subquery to get totals, then a window function for rank:

```sql
-- First, get total loan amount per customer
WITH CustomerTotals AS (
  SELECT
    CustomerID,
    SUM(Amount) AS TotalLoanAmount
  FROM Loans
  GROUP BY CustomerID
)
SELECT
  CustomerID,
  TotalLoanAmount,
  RANK() OVER (ORDER BY TotalLoanAmount DESC) AS AmountRank
FROM CustomerTotals
ORDER BY AmountRank;
```

*(Alternatively, we could do this in one query by applying* `RANK() OVER (ORDER BY SUM(Amount) DESC)` *directly, but many SQL dialects require the aggregation to be done in a subquery or CTE before using the window function.)*

**Explanation:** The CTE `CustomerTotals` calculates each customer's total borrowed amount. The main query then selects from this CTE and uses `RANK()` as a window function to assign a rank based on `TotalLoanAmount` in descending order (highest amount gets rank 1). We choose `RANK()` over `DENSE_RANK()` here because if two customers have the exact same total, they will share the same rank and the next rank will be skipped (e.g., if two customers tie for rank 1, the next rank given will be 3). Depending on the requirement, `DENSE_RANK()` could also be used (which would make the next rank 2 in a tie scenario). The final `ORDER BY AmountRank` just sorts the output by rank for readability.

**Expected Output:** (Example)

| CustomerID | TotalLoanAmount | AmountRank |
|---|---|---|
| 722 | 500000 | 1 |
| 615 | 450000 | 2 |
| 501 | 450000 | 2 |

| CustomerID | TotalLoanAmount | AmountRank |
|---|---|---|
| 308 | 300000 | 4 |
| ... | ... | ... |

Here, Customer 722 has the highest total loans (\$500k) and is rank 1. Customers 615 and 501 each have \$450k, tied for second-highest total, so they both get rank 2, and the next rank is 4. This demonstrates the behavior of `RANK()` with ties. In an interview, make sure to explain if ties are possible and whether you used `RANK` or `DENSE_RANK` accordingly.

### Example 5: Finding Loan Applications Rejected Multiple Times

**Concepts:** Aggregation (`COUNT`), `GROUP BY`, `HAVING`, subquery alternative, filtering by a condition.
**Scenario:** Consider a `LoanApplications` table that logs each loan application attempt by customers and whether it was approved or rejected. We want to find customers who have had more than one rejected loan application — possibly indicating those who keep applying and getting denied.

**Table Structure:** `LoanApplications`
- **ApplicationID** – unique application record
- **CustomerID** – who applied
- **ApplicationDate** – date of application
- **Status** – status of application ('Approved', 'Rejected', etc.)

**Sample Problem:** *"List the customers who have at least two loan applications that were rejected. Provide the customer ID and the number of rejections."*

**SQL Solution:**

```sql
SELECT
  CustomerID,
  COUNT(*) AS RejectionCount
FROM LoanApplications
WHERE Status = 'Rejected'
GROUP BY CustomerID
HAVING COUNT(*) > 1;
```

**Explanation:** We filter the applications to only those with `Status = 'Rejected'`, then group by customer and count them. The `HAVING COUNT(*) > 1` ensures we only get customers with more than one rejection. This yields the list of customers and how many times they've been rejected. This pattern is straightforward for counting occurrences with a condition.

*Alternative approach:* If the question was phrased differently (e.g., "find customers who were rejected multiple times but eventually approved"), one might use a subquery or `EXISTS` to mix conditions (like one subquery checking for at least one 'Approved' status and the main query filtering those with rejects). But for the question as stated, the simple aggregation is enough.

**Expected Output:** (Example)

| CustomerID | RejectionCount |
|---|---|
| 210 | 3 |
| 502 | 2 |
| 718 | 4 |

This indicates, for example, customer 210 had 3 rejected applications, customer 502 had 2, etc. These are the customers who faced multiple rejections. (In a real scenario, one might then investigate if those customers eventually got approved or not, but that's beyond this query.)

### Example 6: Detecting Suspicious Loan Repayment Behavior

**Concepts:** Advanced window functions (`LAG`), self-join possibility, date difference, complex filtering.
**Scenario:** The risk/fraud team wants to flag unusual loan repayment patterns. One potential red flag is a customer repeatedly taking new loans shortly after closing previous ones (which might indicate using one loan to pay off another, or financial distress). Let's find customers who closed a loan and then opened a new one within 30 days. This could be considered "suspicious" rapid re-borrowing behavior.

**Table Structure:** `Loans` (assuming each record is a loan, and a customer can have multiple loans over time)
- **CustomerID** – borrower's ID
- **LoanID** – loan identifier
- **LoanDate** – date the loan was issued (start date)
- **PaymentDate** – date the loan was fully paid (end date; NULL if not paid off yet, but for this analysis we'll focus on loans that did close)

**Sample Problem:** *"Find instances where a customer took out a new loan within 30 days of closing a previous loan. Return the customer ID and details of the new loan (loan ID and start date), along with the previous loan's end date for reference."*

**SQL Solution:** We can use a window function to compare each loan's start date with the previous loan's end date for the same customer.

```
WITH LoanHistory AS (
  SELECT
    CustomerID,
    LoanID,
    LoanDate,
    PaymentDate,
    -- Use LAG to get the previous loan's PaymentDate for the same customer
    LAG(PaymentDate) OVER (
      PARTITION BY CustomerID
      ORDER BY LoanDate
```

```
    ) AS PrevLoanEndDate
  FROM Loans
  WHERE PaymentDate IS NOT NULL   -- consider only loans that ended (were paid
off)
)
SELECT
  CustomerID,
  LoanID AS NewLoanID,
  LoanDate AS NewLoanDate,
  PrevLoanEndDate AS PreviousLoanPaidDate,
  DATEDIFF(day, PrevLoanEndDate, LoanDate) AS DaysBetweenLoans
FROM LoanHistory
WHERE
  PrevLoanEndDate IS NOT NULL
  AND DATEDIFF(day, PrevLoanEndDate, LoanDate) <= 30;
```

**Explanation:** In the CTE `LoanHistory`, we partition the loans by CustomerID and order by LoanDate. The `LAG(PaymentDate)` gives us the PaymentDate of the previous loan for that customer. We filter to `PaymentDate IS NOT NULL` in the CTE because if a loan isn't paid off, it can't count as a "previous loan closed". In the main query, we then look for cases where `PrevLoanEndDate` is not null (meaning the customer had a prior loan) and the difference in days between that date and the current loan's start (`LoanDate`) is 30 days or less. We output the new loan's details and how many days after the previous loan it was taken.

If a customer had multiple sequential loans, each qualifying gap will appear. This query effectively flags rapid re-borrowing. A self-join solution is also possible: joining the `Loans` table to itself on matching CustomerID where one loan's LoanDate is within 30 days of the other's PaymentDate (with appropriate conditions to avoid false matches). However, the window function approach is cleaner for this sequential relationship.

**Expected Output:** (Example)

| CustomerID | NewLoanID | NewLoanDate | PreviousLoanPaidDate | DaysBetweenLoans |
|---|---|---|---|---|
| 305 | 9001 | 2024-11-15 | 2024-10-20 | 26 |
| 305 | 9100 | 2025-01-10 | 2024-12-20 | 21 |
| 422 | 8800 | 2025-03-01 | 2025-02-15 | 14 |

In this sample, Customer 305 had a loan (ID 9001) on Nov 15, 2024, which started 26 days after they paid off a previous loan on Oct 20, 2024. The same customer took another loan (ID 9100) on Jan 10, 2025, just 21 days after closing a loan on Dec 20, 2024. Customer 422 took out loan 8800 on Mar 1, 2025, two weeks after finishing a prior loan on Feb 15, 2025. These scenarios might be flagged for review as suspicious patterns.

---

**Note:** The above problems use simplified assumptions for demonstration. In a real interview, make sure to clarify the schema (table columns, meaning of fields) and any assumptions (like how interest is stored or what counts as "active" or "overdue"). Showing that you can ask the right clarifying questions is part of the interview. Each solution provided is one way to solve the problem; there can be other correct solutions (e.g., using different joins or subqueries). The key is to demonstrate understanding of the SQL concepts and produce correct results.

---


[1] [2] [3] [4] [5] mysql - What's the difference between INNER JOIN, LEFT JOIN, RIGHT JOIN and FULL JOIN? - Stack Overflow
https://stackoverflow.com/questions/5706437/whats-the-difference-between-inner-join-left-join-right-join-and-full-join

[6] SQL GROUP BY Statement - W3Schools
https://www.w3schools.com/sql/sql_groupby.asp

[7] SQL HAVING Clause - W3Schools
https://www.w3schools.com/sql/sql_having.asp

[8] SQL Window Functions | Advanced SQL - Mode
https://mode.com/sql-tutorial/sql-window-functions/

[9] [10] [11] Ranking with Window Functions | DataLemur
https://datalemur.com/sql-tutorial/sql-rank-dense_rank-row_number-window-function

[12] SQL Time-Series Window Functions: LEAD & LAG Tutorial
https://datalemur.com/sql-tutorial/sql-time-series-window-function-lead-lag

[13] [14] [15] CTE vs. Subquery | DataLemur
https://datalemur.com/sql-tutorial/sql-cte-subquery

[16] SQL CASE Expression - W3Schools
https://www.w3schools.com/sql/sql_case.asp

[17] MySQL Indexing Best Practices - GeeksforGeeks
https://www.geeksforgeeks.org/mysql-indexing-best-practices/

[18] 28 SQL interview questions and answers from beginner to senior level - CodeSignal
https://codesignal.com/blog/interview-prep/28-sql-interview-questions-and-answers-from-beginner-to-senior-level/