Advance SQL

SQL mastery involves more than basic querying; it's about using advanced techniques to write efficient, readable, and scalable queries.

coding_knowledge

Today, we'll cover:

- Understanding Common Table Expressions (CTEs) to break down complex queries
- Using Subqueries for filtering, comparisons, and aggregations within queries
- Leveraging Self-Joins to compare rows within the same table
- Exploring various Window Functions to perform calculations across rows:
    - Aggregate Functions for cumulative calculations within partitions
    - Ranking Functions to assign ranks to rows within groups
    - Value Functions to access and compare values from different rows
- Data Modeling
- Key SQL Optimization Techniques to make your queries faster and more efficient

coding_knowladge

harry

Common Table Expressions (CTEs)

When to Use:

- CTEs simplify complex queries by breaking them into readable parts and allow for reusable temporary result sets.

Example: Using Two CTEs and Joining Them

Scenario: Find total sales per employee and compare it to their department's average sales.

```sql
WITH EmployeeSales AS (
    SELECT e.employee_id, e.name, e.department_id, SUM(s.sales_amount) AS total_sales
    FROM employees e
    INNER JOIN sales s ON e.employee_id = s.employee_id
    GROUP BY e.employee_id, e.name, e.department_id
),
DepartmentAvgSales AS (
    SELECT department_id, AVG(total_sales) AS avg_department_sales
    FROM (
        SELECT e.department_id, SUM(s.sales_amount) AS total_sales
        FROM employees e
        INNER JOIN sales s ON e.employee_id = s.employee_id
        GROUP BY e.department_id, e.employee_id
    ) AS DepartmentSales
    GROUP BY department_id
)
SELECT es.employee_id, es.name, es.total_sales, das.avg_department_sales
FROM EmployeeSales es
INNER JOIN DepartmentAvgSales das ON es.department_id = das.department_id
WHERE es.total_sales > das.avg_department_sales;
```

ChatGPT Prompt: Describe how to use multiple CTEs to join results and provide an example.

Subqueries

When to Use:

- Use subqueries when you need to pass the result of one query as input to another, often for filtering or comparisons.

Example: Find Employees Who Earn More Than Their Department's Average

```sql
SELECT employee_id, name, salary
FROM employees e
WHERE salary > (
    SELECT AVG(salary)
    FROM employees
    WHERE department_id = e.department_id);
```

ChatGPT Prompt: Explain how subqueries and correlated subqueries work.

Self-Joins

When to Use:

- Self-joins are used to compare rows within the same table, often for hierarchical data.

Example: Find Employees and Their Managers

```
SQL

SELECT e.employee_id, e.name AS employee_name,
m.name AS manager_name
FROM employees e
LEFT JOIN employees m ON e.manager_id =
m.employee_id;
```

ChatGPT Prompt: Explain self-joins and provide an example of a hierarchy.

coding_knowladge
harry

Window Functions

Aggregate Functions

(Used for calculations over a set of rows)

- SUM(): Calculates the sum of values over a window.
- AVG(): Calculates the average of values.
- MIN(): Finds the minimum value.
- MAX(): Finds the maximum value.
- COUNT(): Counts rows over a window.

Example: Calculate the Total Sales by Department for Each Employee

```sql
SELECT employee_id, department_id, salary,
       SUM(salary) OVER (PARTITION BY
department_id) AS total_department_salary
FROM employees;
```

Interview Question: How would you calculate the total salary for each department without grouping the entire dataset?

ChatGPT Prompt: Explain how aggregate window functions like SUM() and AVG() work in SQL, and provide examples.

Window Functions

Ranking Functions

coding_knowladge

(Used to rank rows within a partition)

- RANK(): Assigns a rank with gaps in case of ties.
- DENSE_RANK(): Assigns ranks without gaps.
- ROW_NUMBER(): Assigns a unique sequential number to each row.
- NTILE(): Divides rows into buckets and assigns bucket numbers.

Example: Rank Employees Based on Salary within Their Department

SQL

```sql
SELECT employee_id, department_id, salary,
       RANK() OVER (PARTITION BY department_id
ORDER BY salary DESC) AS salary_rank
FROM employees;
```

Interview Question: What's the difference between RANK() and DENSE_RANK()? When would you use one over the other?

ChatGPT Prompt: Describe how the RANK() and ROW_NUMBER() functions work, and provide examples for ranking rows within partitions.

Window Functions

Value Functions

(Used to retrieve values from different rows)

- LAG(): Returns the value from the previous row.
- LEAD(): Returns the value from the next row.
- FIRST_VALUE(): Returns the first value in the window.
- LAST_VALUE(): Returns the last value in the window.
- NTH_VALUE(): Returns the nth value from the window.

Example: Calculate the Difference in Salary Between Consecutive Employees

SQL

```sql
SELECT employee_id, salary,
       LAG(salary) OVER (ORDER BY salary) AS previous_salary,
       salary - LAG(salary) OVER (ORDER BY salary) AS
salary_difference
FROM employees;
```

Interview Question: How can you use the LAG() function to calculate the difference between consecutive rows in a time-series dataset?

ChatGPT Prompt: Explain how the LAG() and LEAD() functions work with examples for calculating the difference between consecutive rows.

coding_knowladge harry

Data Modeling

Data modeling is the process of structuring and organizing data in a database.

Star Schema

coding_knowladge

What It Is:

- The Star Schema is one of the most common data warehouse schema designs. It consists of a central fact table surrounded by dimension tables, resembling a star shape.

Key Features:

- Fact Table: Contains quantitative data, such as sales amounts, transaction counts, etc. It is at the center of the schema and stores the key performance metrics.
- Dimension Tables: Surround the fact table and store descriptive data that provides context to the facts. For example, dimensions could include time, customer, product, or geography.

When to Use:

- The star schema is ideal for simplifying queries. It provides fast performance for analytical queries because you don't need many joins between the fact and dimension tables.

Advantages:

- Easy to understand and query.
- Highly optimized for read-heavy workloads, like reporting and dashboards.

Disadvantages:

- Some level of data redundancy in dimension tables (e.g., repeating product categories across products), but it's acceptable in exchange for query performance.

Data Modeling

Snowflake Schema

coding_knowladge

What It Is:

- The Snowflake Schema is a more normalized version of the star schema. In this design, the dimension tables are further broken down into additional tables, creating a more complex structure that resembles a snowflake.

Key Features:

- Fact Table: Same as in the star schema, containing quantitative data.
- Normalized Dimension Tables: Dimension tables are broken into multiple related tables to minimize redundancy. For example, in a snowflake schema, a product dimension might be split into product, product category, and product subcategory tables.

When to Use:

- The snowflake schema is useful when you want to reduce redundancy in the dimension tables or when your data naturally fits a highly normalized structure.

Advantages:

- Reduces data redundancy, saving storage space.
- Can be useful when your database management system (DBMS) benefits from normalized structures.

Disadvantages:

- More complex queries because of the need for multiple joins.
- Slightly slower query performance compared to the star schema, as multiple joins are required..

Data Modeling

Choosing Between Star and Snowflake Schema

- Star Schema is typically preferred for data warehousing and reporting because it simplifies queries and improves performance. It's more user-friendly for non-technical users who may be writing queries.
- Snowflake Schema is more normalized, reducing redundancy but adding complexity. It’s better suited for systems where storage efficiency is more important, or when your data naturally fits into more normalized tables.

SQL Query Optimization Techniques

- Indexing:
    - Create indexes on columns frequently used in WHERE, JOIN, and ORDER BY clauses for faster lookups.
- Avoid SELECT *:
    - Select only the necessary columns instead of using SELECT * to reduce the amount of data retrieved.
- Query Execution Plan Analysis:
    - Use tools like EXPLAIN or EXPLAIN ANALYZE to understand how queries are being executed and identify bottlenecks.
- Use WHERE Clauses to Filter Early:
    - Apply WHERE clauses to filter data as early as possible, reducing the dataset size and improving performance.
- JOIN Optimization:
    - Ensure JOIN conditions use indexed columns and avoid unnecessary joins to prevent slowdowns.
- Partitioning:
    - Divide large tables into smaller, more manageable parts (horizontal/vertical partitioning) to improve query performance.

- Batch Updates and Inserts:
    - Group multiple updates or inserts into batches to reduce I/O operations and improve performance.

SQL Query Optimization Techniques

- Avoid Unnecessary Subqueries:
  - Replace subqueries with JOIN operations or CTEs where appropriate to reduce complexity and improve performance.
- Use EXISTS Instead of IN:
  - When checking for existence, EXISTS is typically more efficient than IN, especially with larger datasets.
- Denormalization:
  - In read-heavy environments, denormalizing data by combining tables can reduce the need for joins and improve performance.
- Materialized Views:
  - Use materialized views to store the results of expensive, frequently-run queries and retrieve them quickly.
- Optimizing GROUP BY and ORDER BY:
  - Avoid unnecessary GROUP BY and ORDER BY operations unless needed to reduce processing time.
- Use Proper Data Types:
  - Use the smallest appropriate data type for each column to minimize storage and improve performance.
- Avoid Functions on Indexed Columns in WHERE Clauses:
  - Applying functions to indexed columns negates the index, slowing down the query.
- Database Caching:
  - Use caching mechanisms to store frequently accessed data in memory, reducing the need for repeated database queries.