✦ Member-only story

Must-do SQL topics for Data Engineers

Vishal Barvaliya · Follow

Published in Towards Data Engineering

13 min read · Feb 27, 2023

1. Data Modeling.

Data modeling is the process of creating a conceptual representation of data in a database. It involves designing the tables, columns, relationships, and constraints that define the data in a database. A good data model helps ensure data accuracy, consistency, and integrity. It also helps to ensure that the database is efficient, scalable, and easy to use.

There are several steps involved in data modeling, including:

1. Identifying the entities: Entities are the objects or concepts that are represented in a database. Examples of entities could include customers, orders, products, or employees.
2. Identifying the attributes: Attributes are the characteristics or properties of the entities. Examples of attributes could include name, address, date of birth, or salary.
3. Defining relationships: Relationships define how entities are related to each other. For example, a customer may place an order, and an order may contain multiple products.
4. Defining constraints: Constraints are rules that govern the data in a database. Examples of constraints could include ensuring that each customer has a unique ID, or that orders cannot be placed for out-of-stock products.

There are several types of data models, including:

1. Conceptual Data Model: A conceptual data model is a high-level representation of the data in a database. It is used to define the overall structure and organization of the data.
2. Logical Data Model: A logical data model is a detailed representation of the data in a database. It includes the entities, attributes, relationships, and constraints that define the data.
3. Physical Data Model: A physical data model is a representation of the data in a database that includes the specific details of how the data is stored and organized on a disk.

Data modeling is an essential skill for data engineers as it provides the foundation for building efficient and scalable databases. By understanding the principles of data modeling, data engineers can

ensure that databases are designed to meet the needs of the organization and can support efficient data processing and analysis.

2. Querying and manipulating data using SQL:

SQL is the primary language used for querying and manipulating data in relational databases. As a Data Engineer, it is important to have a strong foundation in SQL syntax, basic concepts, joins, subqueries, aggregation functions, and advanced concepts like window functions and common table expressions.

Here are some key concepts and techniques to master in SQL:

- Basic SQL syntax: Understanding the basic syntax of SQL is important, including keywords like SELECT, FROM, WHERE, and ORDER BY. A Data Engineer should be able to write SQL queries to filter, sort, and group data.
- Joins Understanding how to join tables using different types of joins (inner, outer, left, right) is essential for querying data from multiple tables.
- Subqueries: Subqueries allow you to write nested queries to filter data based on results from another query. A Data Engineer should be familiar with subqueries and know how to use them in conjunction with other SQL commands.
- Aggregation functions: Aggregation functions like COUNT, SUM, AVG, MAX, and MIN are used to summarize data in a table. A Data Engineer should be able to use these functions to perform data analysis.
- Window functions: Window functions allow you to perform calculations on a subset of rows in a table. A Data Engineer should be familiar with window functions and know how to use them to analyze data trends and patterns.

- Common table expressions (CTEs): CTEs are used to define temporary named result sets that can be referenced within a query. A Data Engineer should be able to use CTEs to simplify complex queries and improve query performance.

3. Indexing and performance tuning:

Indexing is the process of creating a data structure that improves the speed of data retrieval operations on a database table. In other words, indexing can help to speed up queries that search large amounts of data. A Data Engineer should understand the different types of indexes, how they work, and when to use them to optimize query performance.

There are several types of indexes, including clustered and non-clustered indexes. A clustered index is created on the primary key of a table, and it determines the physical order of the data in the table. A non-clustered index is created on a column other than the primary key, and it creates a separate data structure that allows for faster searching.

In addition to understanding indexing, a Data Engineer should be able to analyze query execution plans to identify performance bottlenecks. Query execution plans show how the database engine is executing a query and can help a Data Engineer to identify areas where performance can be improved. By analyzing query execution plans, a Data Engineer can optimize query performance by making changes to the database schema, indexing strategy, or query structure.

4. Query Optimization.

Query optimization is the process of improving the performance of SQL queries by minimizing the time it takes to retrieve data from a database. The goal of query optimization is to ensure that queries execute as quickly and efficiently as possible. This is particularly important for data engineers who work with large datasets and need to retrieve data quickly to perform data processing and analysis.

Several techniques can be used to optimize SQL queries, including:

1. **Use Indexes:** Indexes are data structures that enable fast retrieval of data. They work by storing a copy of the data in a separate data structure that can be searched more quickly than the main table. By using indexes, data engineers can minimize the time it takes to retrieve data from a table.

2. **Avoid Subqueries:** Subqueries are queries that are embedded within other queries. They can be useful for certain types of queries, but they can also be inefficient. In general, it is best to avoid subqueries whenever possible, as they can slow down the performance of a query.

3. **Optimize Joins:** Joins are used to combine data from multiple tables into a single result set. Data engineers can optimize joins by minimizing the number of tables involved in the query, using indexes on the join columns, and using the appropriate join type (inner, outer, left, or right) for the query.

4. **Use Views:** Views are virtual tables that can be used to simplify complex queries or to create reusable queries. They are particularly useful for queries that involve multiple tables or complex joins. By using views, data engineers can improve query performance and simplify query creation.

5. **Minimize Data Transfer:** Data transfer can be a bottleneck for query performance. To minimize data transfer, data engineers should only retrieve the data that is needed for the query and avoid transferring unnecessary data.

6. **Use Stored Procedures:** Stored procedures are precompiled SQL statements that can be reused across multiple queries. By using

stored procedures, data engineers can improve query performance and simplify query creation.

7. **Use Query Execution Plans:** Query execution plans are diagrams that show how a query is executed by the database engine. They can be used to identify performance bottlenecks and optimize queries.

In summary, query optimization is a critical skill for data engineers who need to work with large datasets and perform complex data processing and analysis. By using the techniques above, data engineers can improve query performance and minimize the time it takes to retrieve data from a database.

5. Aggregation Functions.

In SQL, aggregation functions are used to perform calculations on a set of values and return a single value. These functions are often used in combination with the GROUP BY clause to calculate summary statistics for groups of data. In this answer, I will explain the various aggregation functions in SQL and provide examples of how they can be used.

1. **COUNT:** The COUNT function is used to count the number of rows in a table or a group of rows based on a specified condition. For example, to count the number of rows in a table, you can use the following query:

```
SELECT COUNT(*)
FROM table_name;
```

To count the number of rows that meet a specific condition, you can use the following query:

```
SELECT COUNT(*)
FROM table_name
WHERE condition;
```

2. SUM: The SUM function is used to calculate the sum of a column or a group of rows based on a specified condition. For example, to calculate the total sales for a particular product, you can use the following query:

```
SELECT SUM(sales)
FROM sales_table
WHERE product_name = 'product_name';
```

3. AVG: The AVG function is used to calculate the average of a column or a group of rows based on a specified condition. For example, to calculate the average sales for a particular product, you can use the following query:

```
SELECT AVG(sales)
FROM sales_table
WHERE product_name = 'product_name';
```

4. MIN: The MIN function is used to find the minimum value in a column or a group of rows based on a specified condition. For example, to find the minimum sales for a particular product, you can use the following query:

```
SELECT MIN(sales)
FROM sales_table
WHERE product_name = 'product_name';
```

5. MAX: The MAX function is used to find the maximum value in a column or a group of rows based on a specified condition. For example, to find the maximum sales for a particular product, you can use the following query:

```
SELECT MAX(sales)
FROM sales_table
WHERE product_name = 'product_name';
```

6. GROUP BY: The GROUP BY clause is used to group rows based on one or more columns and apply an aggregation function to each group. For example, to calculate the total sales for each product, you can use the following query:

```
SELECT product_name, SUM(sales)
FROM sales_table
GROUP BY product_name;
```

This query will group the rows `sales_table` by `product_name` and calculate the total sales for each product.

In summary, aggregation functions are a powerful tool in SQL that can be used to perform calculations on large sets of data. By combining

these functions with the GROUP BY clause, you can quickly and easily generate summary statistics for groups of data in a table.

6. Joins

In SQL, a join is used to combine data from two or more tables into a single result set. Joins are a fundamental part of SQL and are used extensively in data processing and analysis. In this answer, I will explain the various types of joins in SQL and provide examples of how they can be used.

There are several types of joins in SQL, including:

1. **Inner Join:** An inner join returns only the rows that have matching values in both tables. The syntax for an inner join is as follows:

```
SELECT *
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

Here, `table1` and `table2` are the names of the tables being joined, and `column_name` is the name of the column that is used to join the tables.

For example, let's say we have two tables: `orders` and `customers`. The `orders` table contains information about orders, including the `order_id` and `customer_id`. The `customers` table contains information about customers, including the `customer_id` and `customer_name`. We can use an inner join to combine these tables and retrieve the customer name for each order:

```
SELECT *
FROM orders
INNER JOIN customers
ON orders.customer_id = customers.customer_id;
```

2. Left Join: A left join returns all the rows from the left table and the matching rows from the right table. If there are no matching rows in the right table, the result set will contain null values for the right table columns. The syntax for a left join is as follows:

```
SELECT *
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

For example, let's say we have two tables: `employees` and `departments`. The `employees` table contains information about employees, including the `employee_id` and `department_id`. The `departments` table contains information about departments, including the `department_id` and `department_name`. We can use a left join to retrieve all the employees and the department they belong to, even if they don't belong to any department:

```
SELECT *
FROM employees
LEFT JOIN departments
ON employees.department_id = departments.department_id;
```

3. **Right Join:** A right join returns all the rows from the right table and the matching rows from the left table. If there are no matching rows in the left table, the result set will contain null values for the left table columns. The syntax for a right join is as follows:

```
SELECT *
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

For example, let's say we have the same tables as in the previous example: `employees` and `departments`. We can use a right join to retrieve all the departments and the employees who belong to them, even if there are no employees in a department:

```
SELECT *
FROM employees
RIGHT JOIN departments
ON employees.department_id = departments.department_id;
```

4. **Full Outer Join:** A full outer join returns all the rows from both tables, including the rows that don't have matching values in the other table. If there are no matching rows in one table, the result set will contain null values for the columns of that table. The syntax for a full outer join is as follows:

```
SELECT *
FROM table1
```

```
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

For example, let's say we have two tables: `students` and `courses`. The `students` table contains information about students, including the `student_id` and `course_id`. The `courses` table contains information about courses

7. Windows Functions.

Window functions are a powerful feature in SQL that allows you to perform complex calculations across rows in a table. Window functions operate on a "window" of rows within the result set and allow you to perform calculations on that subset of data.

There are several types of window functions in SQL, including ranking functions, aggregate functions, and analytic functions. In this answer, we'll explain each of these types of functions in more detail.

1. **Ranking functions:** Ranking functions are used to assign a rank to each row within a result set based on the value of a particular column. The most common ranking functions are ROW_NUMBER(), RANK(), and DENSE_RANK().

- ROW_NUMBER(): This function assigns a unique number to each row in a result set, starting from 1. Example:

```
SELECT ROW_NUMBER() OVER (ORDER BY salary DESC) as rank, name, sala
FROM employees;
```

- RANK(): This function assigns the same rank to rows with the same values, and then skips the next rank number. For example, if two rows have the same value and are ranked 1, the next row will be ranked 3 (skipping rank 2). Example:

```
SELECT RANK() OVER (ORDER BY salary DESC) as rank, name, salary
FROM employees;
```

- DENSE_RANK(): This function assigns the same rank to rows with the same values, without skipping any rank numbers. Example:

```
SELECT DENSE_RANK() OVER (ORDER BY salary DESC) as rank, name, sala
FROM employees;
```

2. Aggregate functions: Aggregate functions can be used in conjunction with a window specification to calculate an aggregate value for a group of rows.

Example:

```
SELECT SUM(salary) OVER (PARTITION BY department) as department_sal
FROM employees;
```

3. Analytic functions: Analytic functions operate on a set of rows and return a single value for each row, based on the values in other rows in

the window. Analytic functions include functions like LAG(), LEAD(), FIRST_VALUE(), LAST_VALUE(), and others.

Example:

```
SELECT name, salary, AVG(salary) OVER (PARTITION BY department) as
FROM employees;
```

In summary, window functions in SQL are a powerful tool for performing complex calculations across rows in a table. By using ranking functions, aggregate functions, and analytic functions, you can perform a wide variety of calculations and analyses on your data.

8. Stored Procedures.

Stored procedures in SQL are a type of program that is stored in the database and can be executed by the database server. Stored procedures are typically used to perform complex operations on the database, such as inserting, updating, or deleting data, or performing calculations and other data transformations.

Stored procedures can be created using the CREATE PROCEDURE statement, which specifies the name of the stored procedure, the input parameters (if any), and the SQL statements that make up the procedure. Here is an example of a simple stored procedure that takes a customer ID as input and returns the customer's name and address:

```
CREATE PROCEDURE GetCustomerInfo
  @CustomerID int
AS
BEGIN
  SELECT Name, Address
```

```
    FROM Customers
    WHERE CustomerID = @CustomerID
  END
```

To execute a stored procedure, you can use the EXECUTE statement followed by the name of the stored procedure and any input parameters that it requires. For example:

```
EXECUTE GetCustomerInfo 1234
```

Stored procedures have several advantages over ad hoc SQL statements:

1. Code reusability: Since stored procedures are stored in the database, they can be reused by multiple applications or users.
2. Performance: Stored procedures can be compiled and optimized by the database server, which can improve performance compared to ad hoc SQL statements.
3. Security: Stored procedures can be used to restrict access to sensitive data or operations, as they can be granted permissions separately from the underlying tables.
4. Maintainability: Stored procedures can be modified and updated independently of the application code, which can make them easier to maintain and update over time.

Overall, stored procedures are a powerful tool in SQL that can help you to perform complex operations on your database more efficiently and securely. By creating and using stored procedures effectively, you can improve the performance and maintainability of your database applications.

9. Transactions.

In SQL, a transaction is a series of database operations that are executed as a single unit of work. Transactions are used to ensure the consistency and integrity of the data in the database, even in the presence of errors or failures.

The basic concept of a transaction is the ACID properties:

- Atomicity: A transaction is atomic, meaning that it is executed as a single, indivisible unit of work. Either all of the operations in the transaction are completed successfully, or none of them are.
- Consistency: A transaction ensures that the data in the database remains consistent throughout the transaction. Any changes made to the data must follow the rules and constraints of the database schema.
- Isolation: Transactions are executed in isolation from other transactions, meaning that the effects of one transaction do not affect other transactions until they are committed.
- Durability: Once a transaction is committed, its changes to the database are permanent and survive any subsequent system failures.

In SQL, transactions are managed using the BEGIN TRANSACTION, COMMIT, and ROLLBACK statements:

- BEGIN TRANSACTION: This statement begins a new transaction.
- COMMIT: This statement ends a transaction and makes its changes permanent.
- ROLLBACK: This statement cancels a transaction and rolls back any changes made during the transaction.

For example, consider a simple transaction that transfers money from one account to another:

```
BEGIN TRANSACTION

UPDATE Accounts SET Balance = Balance - 100 WHERE AccountNumber = '
UPDATE Accounts SET Balance = Balance + 100 WHERE AccountNumber = '

COMMIT
```

If either of the UPDATE statements fails for some reason (e.g., due to a network failure or a constraint violation), the transaction is rolled back and none of the changes are made permanent.

Overall, transactions are an important concept in SQL that help to ensure the consistency and integrity of the data in a database. By using transactions effectively, you can make your database applications more robust and reliable.

That's all you need to know about SQL as a data engineer.

Thanks for reading

. . .

> **Best of luck with your journey!!!**
>
> **Follow for more such content on Data Engineering and Data Science.**

Resources used to write this blog: