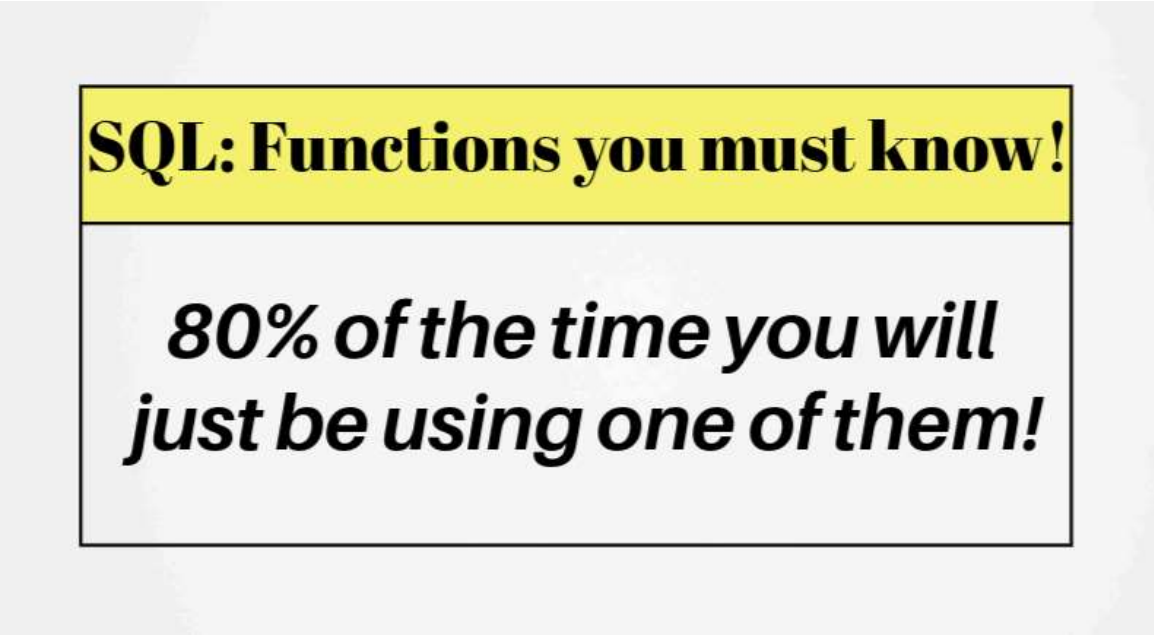
Minimal Devops

Summary

The provided content is a comprehensive guide to essential SQL



SQL Functions you must know for everyday use

80% of the time you will just be using one of them!

SQL (Structured Query Language) remains the backbone of data management in most organizations. As a data engineer or anyone else using SQL, mastering ***SQL*** is essential because it allows you to manipulate, query, and extract valuable insights from data efficiently. Read till the end to discover all ***SQL Functions you must know***!

In this blog, we'll explore the most commonly used ***SQL functions***, their importance, use cases, and examples to help you understand why these functions are indispensable in everyday data tasks.

By mastering these functions, you can handle around 80% of your daily SQL tasks efficiently, *showcasing the 80/20 rule in practice*. Start applying these functions in your SQL queries to become a more proficient **data engineer**!

1. SELECT

Importance: The `SELECT` statement is the foundation of SQL. It is used to retrieve data from one or more tables in a database.

Use Case: Retrieving customer details from a database for reporting purposes.

2. WHERE

Importance: The `WHERE` clause filters records that meet certain criteria, allowing you to refine your data queries.

Use Case: Finding customers from a specific city.

```
SELECT customer_name, customer_email FROM customers WHERE city = 'New York';
```

3. ORDER BY

Importance: `ORDER BY` is used to sort the result set in ascending or descending order.

Use Case: Sorting customers by their last purchase date.

4. JOINS (INNER, LEFT, FULL)

Importance: Joins are used to combine rows from two or more tables based on a related column between them.

INNER JOIN:

Use Case: Combining customer and order data to get a complete view of customer purchases.

```
SELECT customers.customer_name, orders.order_id
FROM customers
INNER JOIN orders ON customers.customer_id = orders.customer_id;
```

LEFT JOIN:

Use Case: You want to find all customers and their corresponding orders, including those customers who haven't made any orders yet.

```
FROM customers
LEFT JOIN orders ON customers.customer_id = orders.customer_id;
```

FULL JOIN:

Use Case: You want to list all customers and all orders, including customers without orders and orders without corresponding customer records (perhaps due to data issues).

```
SELECT customers.customer_id, customers.customer_name, orders.order_id
FROM customers
FULL JOIN orders ON customers.customer_id = orders.customer_id;
```

This query combines the results of a `LEFT JOIN` and a `RIGHT JOIN`, showing all records from both tables, with `NULLs` where there's no match.

5. Aggregation Functions (SUM, AVG, COUNT, MAX)

Importance: Aggregation functions perform calculations on a set of values and return a single value.

COUNT:

number of orders.

```
SELECT COUNT(order_id) AS total_orders, SUM(order_amount) AS total_sales FROM
```

SUM & AVG:

Use Case: You need to calculate the total revenue generated from all sales, as well as the average order value, for the last quarter.

```
SELECT SUM(order_amount) AS total_revenue, AVG(order_amount) AS average_order
FROM orders
WHERE order_date BETWEEN '2024-04-01' AND '2024-06-30';
```

MAX:

Use Case: You want to identify the highest purchase amount in your e-commerce platform to analyze the peak spending behavior of your customers.

```
FROM orders;
```

6. GROUP BY

Importance: `GROUP BY` groups rows that have the same values into summary rows, often used with aggregation functions.

Use Case: Summarizing total sales per customer.

```
SELECT customer_id, SUM(order_amount) AS total_sales
FROM orders
GROUP BY customer_id;
```

7. UNION

Importance: `UNION` combines the result sets of two or more `SELECT` statements.

Use Case: Combining customer data from two different regions.

```
UNION
SELECT customer_name FROM customers_region2;
```

8. CASE STATEMENT

Importance: The `CASE` statement is SQL's way of handling if/then logic.

Use Case: Categorizing customers based on their spending.

```
SELECT customer_name,
CASE
    WHEN order_amount > 1000 THEN 'High Spender'
    ELSE 'Regular Spender'
END AS spender_category
FROM customers;
```

9. CTEs (Common Table Expressions)

Importance: CTEs provide a way to create a temporary result set that can be referenced within a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement.

SELECT:

parts.

```
WITH Sales_CTE AS (
    SELECT customer_id, SUM(order_amount) AS total_sales
    FROM orders
    GROUP BY customer_id
)
SELECT * FROM Sales_CTE WHERE total_sales > 500;
```

INSERT:

Use Case: You've just onboarded a new customer, and you need to add their information to the `customers` table.

```
INSERT INTO customers (customer_name, email, city)
VALUES ('John Doe', 'johndoe@example.com', 'Los Angeles');
```

UPDATE:

Use Case: A customer has moved to a new city, and you need to update their record in the database.

```
SET city = 'San Francisco'
WHERE customer_id = 101;
```

DELETE:

Use Case: A customer has requested to have their account and all associated data removed from your system, so you need to delete their record from the database.

```
DELETE FROM customers
WHERE customer_id = 101;
```

10. Date and Time Operations (EXTRACT, DATE_ADD, DATE_SUB)

Importance: These functions help in manipulating and retrieving date and time values.

EXTRACT:

Use Case: Extracting year or month from a date for time-based analysis.

DATE_SUB:

Use Case: You need to create a report showing all orders made within the last week.

```
SELECT order_id, order_date
FROM orders
WHERE order_date >= DATE_SUB(CURRENT_DATE, INTERVAL 7 DAY);
```

DATE_ADD:

Use Case: You want to schedule a follow-up with each customer exactly 30 days after their order date.

```
SELECT customer_id, order_date, DATE_ADD(order_date, INTERVAL 30 DAY) AS foll
FROM orders;
```

11. COALESCE & IFNULL

NULL value in a list.

COALESCE:

Use Case: Providing a default value when data is missing.

```
SELECT COALESCE(customer_email, 'No Email Provided') FROM customers;
```

IFNULL:

Use Case: You're preparing a report where any missing sales amounts should be replaced with zero.

```
SELECT order_id, IFNULL(sales_amount, 0) AS sales_amount
FROM orders;
```

12. CAST & CONVERT

Importance: `CAST` and `CONVERT` are used to change the data type of a value or column.

Use Case: You need to ensure that a text-based date field is converted into an actual date type for accurate sorting and calculations.

```
SELECT order_id, CAST(order_date_text AS DATE) AS order_date
FROM orders;
```

CONVERT:

Use Case: You receive data where the `order_amount` is stored as text, but you need it as a numeric type for calculations.

```
SELECT order_id, CONVERT(DECIMAL(10, 2), order_amount_text) AS order_amount
FROM orders;
```

13. Ranking Functions (ROW_NUMBER, RANK, DENSE_RANK)

Importance: These functions are used to rank rows in a result set.

Use Case: Finding the top 5 customers based on total purchases.

```
SELECT customer_name,
ROW_NUMBER() OVER (ORDER BY total_purchase DESC) AS rank
FROM customers;
```

DENSE_RANK:

Use Case: You need to rank products within each category by sales, ensuring that ties get the same rank without gaps.

```
SELECT category, product_name, sales,
       DENSE_RANK() OVER (PARTITION BY category ORDER BY sales DESC) AS dense
FROM products;
```

14. QUALIFY

Importance: `QUALIFY` filters the result of a window function.

```
SELECT customer_name, RANK() OVER (ORDER BY total_purchase DESC) AS rank
FROM customers
QUALIFY rank <= 5;
```

15. String Functions (SUBSTRING, CONCAT)

Importance: String functions allow manipulation of text data.

CONCAT:

Use Case: Extracting specific parts of a text field or combining fields.

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM customers;
```

SUBSTRING:

Use Case: You need to extract the domain from a list of email addresses for analysis.

```
FROM customers;
```

16. LEAD & LAG

Importance: These window functions provide access to a row at a specific physical offset before or after the current row.

LAG:

Use Case: Analyzing changes in customer purchases over time.

```
SELECT customer_id, order_date, order_amount,
LAG(order_amount) OVER (PARTITION BY customer_id ORDER BY order_date) AS prev
FROM orders;
```

LEAD:

Use Case: You're analyzing customer order patterns and want to see the difference in order amounts between consecutive orders for each customer.

```
    LEAD(order_amount) OVER (PARTITION BY customer_id ORDER BY order_date) AS nex
    FROM orders;
```

This query uses `LEAD` to show the amount of the next order for each customer, allowing you to compare the current order with the subsequent one.

17. ARRAY_AGG

Importance: `ARRAY_AGG` is a function in PostgreSQL that aggregates multiple rows into an array, which can be extremely helpful when you want to compile a list of items from multiple rows into a single row.

Use Case: Suppose you have a list of car brands in a table, and you want to compile all the brands that a particular user has in their garage into a single row as an array.

```
SELECT user_id, ARRAY_AGG(car_brand) AS car_brands
FROM user_cars
GROUP BY user_id;
```

an array for each user.

18. STRING_AGG

Importance: `STRING_AGG` is another powerful PostgreSQL function that concatenates values from multiple rows into a single string, with a specified delimiter.

Use Case: Concatenating a list of employee names who belong to a specific department.

```
SELECT department_id, STRING_AGG(employee_name, ', ') AS employee_names
FROM employees
GROUP BY department_id;
```

This query groups employees by `department_id` and concatenates their names into a single string, separated by commas.

19. PARTITION BY

The `PARTITION BY` clause is a powerful feature in SQL, particularly when working with window functions. It allows you to divide the result set into partitions (or groups) and perform calculations across each partition

subsets of data rather than across the entire dataset.

Use Case: You work for a sales company, and you need to calculate the running total of sales for each salesperson, but you want the running total to reset for each new month. In this case, you want to partition the data by the salesperson and the month.

```
SELECT salesperson_id,
       order_date,
       SUM(sales_amount) OVER (PARTITION BY salesperson_id, EXTRACT(MONTH FRO
                               ORDER BY order_date) AS monthly_running_total
FROM sales_orders;
```

`PARTITION BY salesperson_id, EXTRACT(MONTH FROM order_date)` : This clause divides the dataset into partitions based on each `salesperson_id` and the month of the `order_date`.

Note: I will be covering a lot of interesting SQL content in future. Please share and support.

Recommended from ReadMedium

Amir

Selectivity in Databases: Are You Indexing Right? 📊

Ever struggled with slow database queries that bog down your application? The secret to lightning-fast performance often lies in...

4 min read

TechieTreeHugger

SQL Showdown: CTEs vs Subqueries vs Temp Tables vs Views

Boost your SQL Game !!

6 min read

Kamireddy Mahendra

Microsoft SQL (Hard Level) Interview Problem

A Detailed Step-by-Step Solution

4 min read

Data Engineer Interview : Deloitte

Questions that I was asked

3 min read

Muhammad Naveed Arshad,MSc |Writer|Editor| AI Engr

Python Tricks for Automating Your Everyday Tasks

A Comprehensive Guide by Muhammad Naveed Arshad,MSc |Writer|Editor| AI Engr

6 min read

Kuldeepkumawat

🌟🌦️ Discover Real-Time Weather Updates with This Amazing Dashboard

Ever wondered what the weather is like in your favourite city? Whether you're planning a trip, deciding what to wear, or just curious about...

4 min read