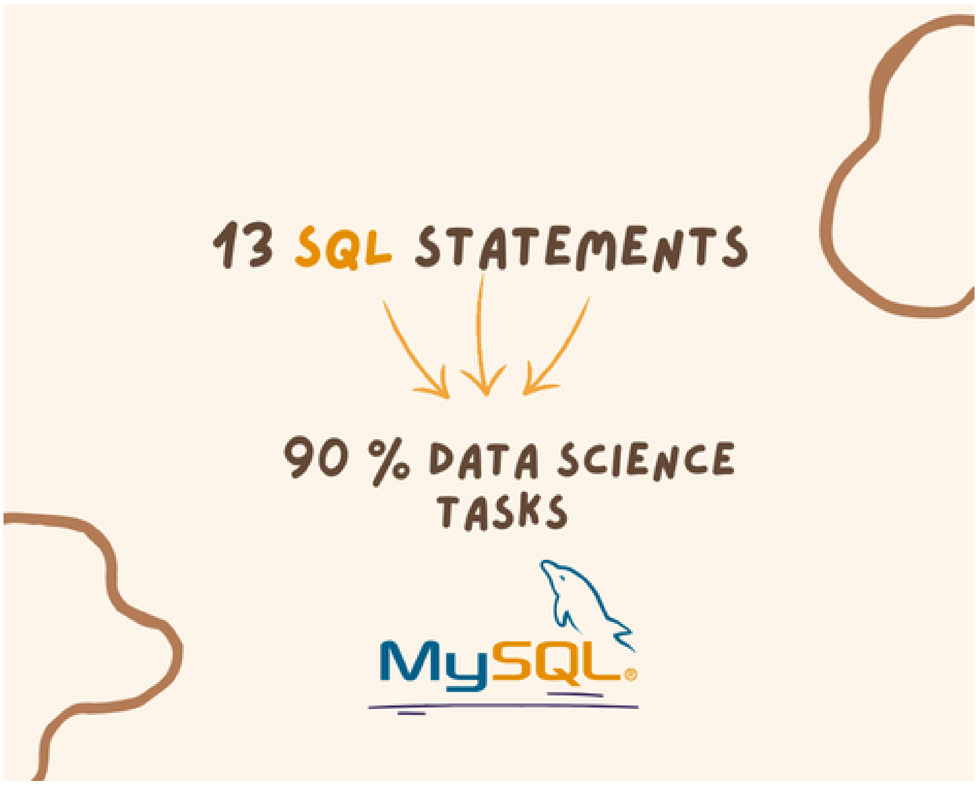
13 SQL Statements for 90% of Your Data Science Tasks

Top highlight



If you want to study Data Science and Machine Learning for free, check out these resources:

- Free interactive roadmaps to learn Data Science and Machine Learning by yourself. Start here: https://aigents.co/learn/roadmaps/intro
- The search engine for Data Science learning resources (FREE). Bookmark your favorite resources, mark articles as complete, and add study notes. https://aigents.co/learn
- Want to learn Data Science from scratch with the support of a mentor and a learning community? Join this Study Circle for free: https://community.aigents.co/spaces/9010170/

1. Select

The **SELECT** statement is used to retrieve data from one or more tables in a database. You should master using SELECT to filter, sort, and group data using different functions such as WHERE, ORDER BY, and GROUP BY. Here is an example of a SELECT statement:

```
SELECT column1, column2, column3
FROM table_name
WHERE condition;
```

In this example, `column1`, `column2`, and `column3` are the names of the columns that you want to retrieve data from, and `table_name` is the name of the table that contains the data. The `WHERE` clause is optional but is used to specify a condition that must be met in order for the query to retrieve data.

Here's an example that selects all records from a table called "customers" where the customer's age is greater than or equal to 18:

```
SELECT *
FROM customers
```

```
WHERE age >= 18;
```

2. JOIN

The JOIN statement is used to combine data from two or more tables in a database. You should master using JOIN to retrieve data from multiple tables and specify the type of join (e.g. INNER, LEFT, RIGHT, FULL OUTER) as appropriate.

Here are a few examples of JOIN statements:

INNER JOIN

An INNER JOIN returns only the rows where there is a match between the columns in both tables. Here is an example:

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

In this example, the `orders` table and the `customers` table are joined using the `customer_id` column. The resulting table will only include the `order_id` and `customer_name` columns where there is a match between the `customer_id` columns in both tables.

LEFT JOIN

A LEFT JOIN returns all the rows from the left table and the matching rows from the right table. If there is no match in the right table, the result will contain NULL values. Here is an example:

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

In this example, the `customers` table is the left table and the `orders` table is the right table. The `customer_id` column is used to join the tables. The resulting table will include all the rows from the `customers` table and the matching rows from the `orders` table. If there is no match in the `orders` table, the `order_id` column will contain NULL values.

RIGHT JOIN

A RIGHT JOIN returns all the rows from the right table and the matching rows from the left table. If there is no match in the left table, the result will contain NULL values. Here is an example:

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

In this example, the `orders` table is the left table and the `customers` table is the right table. The `customer_id` column is used to join the tables. The resulting table will include all the rows from the `orders` table and the matching rows from the `customers` table. If there is no match in the `customers` table, the `customer_name` column will contain NULL values.

OUTER JOIN

An OUTER JOIN in SQL is used to return all the rows from one or both tables, including the non-matching rows. There are two types of OUTER JOINs: LEFT OUTER JOIN and RIGHT OUTER JOIN.

Here's an example of a LEFT OUTER JOIN:

```
SELECT customers.customer_name, orders.order_id
FROM customers
```

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

In this example, the `customers` table is the left table and the `orders` table is the right table. The `customer_id` column is used to join the tables. The resulting table will include all the rows from the `customers` table and the matching rows from the `orders` table. If there is no match in the `orders` table, the `order_id` column will contain NULL values.

Here's an example of a RIGHT OUTER JOIN:

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

In this example, the `orders` table is the left table and the `customers` table is the right table. The `customer_id` column is used to join the tables. The resulting table will include all the rows from the `orders` table and the matching rows from the `customers` table. If there is no match in the `customers` table, the `customer_name` column will contain NULL values.

It's worth noting that some databases may not support RIGHT OUTER JOINs, but you can achieve the same result by using a LEFT OUTER JOIN and swapping the order of the tables.

3. WHERE

The WHERE statement is used to filter data based on a specified condition. You should master using WHERE to retrieve only the data that meets certain criteria.

Here's an example of using a "where" statement in SQL to filter data from a table:

Suppose we have a table named "employees" with columns for "name", "department", and "salary". We can use a "where" statement to select only those employees who work in the "Sales" department and have a salary greater than $50,000:

```
SELECT name, department, salary
FROM employees
WHERE department = 'Sales' AND salary > 50000;
```

This query would return a list of all employees who work in the "Sales" department and have a salary greater than $50,000, with their names, departments, and salaries displayed in the results.

4. GROUP BY

The GROUP BY statement is used to group data based on one or more columns, and aggregate functions (e.g. COUNT, SUM, AVG) can be used to calculate summaries of the grouped data. You should master using GROUP BY to analyze data by categories.

Suppose we have a table named "employees" with columns for "name", "department", and "salary". We can use a GROUP BY statement to group the employees by department and calculate the average salary for each department:

```
SELECT department, AVG(salary) as avg_salary
FROM employees
GROUP BY department;
```

This query would return a list of all departments and the average salary for each department, calculated by taking the sum of all salaries for employees in that department and dividing it by the number of employees in that department. The **GROUP BY** clause is used to group the employees by department, and the AVG function is used to calculate the average salary for each department.

```
department | avg_salary
-----------------------
```

```
Sales       | 65000
Marketing   | 55000
Engineering| 80000
```

In this example, we can see that the Sales department has an average salary of $65,000, the Marketing department has an average salary of $55,000, and the Engineering department has an average salary of $80,000.

5. HAVING

The **HAVING** statement is used to filter data after it has been grouped by the **GROUP BY** statement. You should master using **HAVING** to filter grouped data based on specific conditions.

Here's an example of using the **HAVING** clause in SQL:

Suppose we have a table named "orders" with columns for "order_id", "customer_id", "product_id", and "quantity". We want to find customers who have ordered a total quantity of at least 50 units of products. We can use the GROUP BY clause to group the orders by customer and calculate the total quantity of each product ordered by each customer. We can then use the HAVING clause to filter the results to only include customers who have ordered a total quantity of at least 50 units:

```
SELECT customer_id, SUM(quantity) AS total_quantity
FROM orders
GROUP BY customer_id
HAVING SUM(quantity) >= 50;
```

This query would return a list of all customers and their total quantity of products ordered, but only include customers who have ordered a total quantity of at least 50 units. The GROUP BY clause is used to group the orders by customer, the SUM function is used to calculate the total quantity of products ordered by each customer, and the HAVING clause is used to filter the results to only include customers who have ordered a total quantity of at least 50 units.

The output of the query would look something like this:

```
customer_id | total_quantity
---------------------------
123         | 60
456         | 70
```

In this example, we can see that customer 123 ordered a total of 60 units of products, and customer 456 ordered a total of 70 units of products. Both of these customers meet the condition specified in the HAVING clause, which requires a total quantity of at least 50 units.

6. Window Function

Window functions in SQL are used to perform calculations on a set of rows that are related to the current row. These functions are applied to a window, which is a subset of rows from a table based on a specified condition or partition. Here are some examples of window functions in SQL:

1. **ROW_NUMBER():** This function assigns a unique sequential number to each row within a partition. The syntax for the ROW_NUMBER() function is:

```
SELECT column1, column2, ..., ROW_NUMBER() OVER (ORDER BY column1) AS row_num
FROM table_name;
```

This query will return a result set with an additional column "row_num" that contains the sequential numbers assigned to each row based on the order of "column1".

2. **SUM():** This function calculates the sum of a column within a partition. The syntax for the SUM() function is:

```
SELECT column1, column2, ..., SUM(column3) OVER (PARTITION BY column1) AS column3_sum
FROM table_name;
```

This query will return a result set with an additional column “column3_sum” that contains the sum of “column3” for each partition based on the values of “column1”.

3. **RANK():** This function assigns a rank to each row within a partition based on the values of a specified column. The syntax for the RANK() function is:

```
SELECT column1, column2, ..., RANK() OVER (PARTITION BY column1 ORDER BY column3 DESC) AS rank_num
FROM table_name;
```

This query will return a result set with an additional column “rank_num” that contains the rank of each row within each partition based on the descending order of “column3”.

4. **AVG()**: This function calculates the average of a column within a partition. The syntax for the AVG() function is:

```
SELECT column1, column2, ..., AVG(column3) OVER (PARTITION BY column1) AS column3_avg
FROM table_name;
```

This query will return a result set with an additional column “column3_avg” that contains the average of “column3” for each partition based on the values of “column1”.

Note that the syntax for window functions may vary depending on the specific database management system (DBMS) being used.

7. UNION

In SQL, the **UNION** operator is used to combine the results of two or more SELECT statements into a single result set. The **SELECT** statements must have the same number of columns, and the columns must have compatible data types. Duplicate rows are automatically removed from the result set.

Here’s an example of using the UNION operator in SQL:

Suppose we have two tables named “customers” and “employees”, both with columns for “name” and “city”. We want to create a list of all people (both customers and employees) who live in New York City. We can use the UNION operator to combine the results of two SELECT statements, one for each table:

```
SELECT name, city
FROM customers
WHERE city = 'New York'
UNION
SELECT name, city
FROM employees
WHERE city = 'New York';
```

This query would return a list of all people who live in New York City, including both customers and employees. The first SELECT statement retrieves all customers who live in New York City, and the second SELECT statement retrieves all employees who live in New York City. The UNION operator combines the results of these two SELECT statements and removes any duplicate rows.

The output of the query would look something like this:

```
name        | city
-------------------
John Smith  | New York
Jane Doe    | New York
Bob Johnson | New York
Samantha Lee| New York
```

In this example, we can see that there are four people who live in New York City, two from the “customers” table and two from the “employees” table, and the UNION operator has combined the results of the two SELECT statements into a single result set.

8. CREATE

The CREATE statement is used to create a new database table, view, or other database objects. You should master using CREATE to create new tables, views, and other database objects. Here's an example of using the CREATE statement in SQL:

Suppose we want to create a new table called "customers" with columns for "id", "name", "email", and "phone". We can use the CREATE statement to do this:

```
CREATE TABLE customers (
  id INT PRIMARY KEY,
  name VARCHAR(50),
  email VARCHAR(100),
  phone VARCHAR(20)
);
```

This query would create a new table called "customers" with four columns: "id", "name", "email", and "phone". The "id" column is defined as an integer and is set as the primary key of the table. The "name" column is defined as a string with a maximum length of 50 characters, and the "email" and "phone" columns are also defined as strings with maximum lengths of 100 and 20 characters, respectively.

After the query is executed, we can insert new rows into the "customers" table and retrieve data from it:

```
INSERT INTO customers (id, name, email, phone)
VALUES (1, 'John Doe', 'johndoe@example.com', '555-555-1234');
SELECT * FROM customers;
```

This query would insert a new row into the "customers" table with an ID of 1, a name of "John Doe", an email of "johndoe@example.com", and a phone number of "555–555–1234". The second query would retrieve all rows from the "customers" table, which would include the new row we just inserted:

```
id | name     | email               | phone
--------------------------------------------
1  | John Doe | johndoe@example.com | 555-555-1234
```

In this example, we have used the CREATE statement to create a new table in a database and inserted a new row into the table.

9. INSERT

The INSERT statement is used to insert data into a database table. You should master using INSERT to add new data to a database table. Here's an example of using the INSERT statement in SQL:

Suppose we have a table named "students" with columns for "id", "name", "major", and "gpa". We want to insert a new row into the table for a student with an ID of 1234, a name of "John Doe", a major in "Computer Science", and a GPA of 3.5. We can use the INSERT statement to do this:

```
INSERT INTO students (id, name, major, gpa)
VALUES (1234, 'John Doe', 'Computer Science', 3.5);
```

This query would insert a new row into the "students" table with the specified values for the "id", "name", "major", and "gpa" columns. The INSERT statement specifies the name of the table we want to insert into, followed by the list of columns we want to insert values into. We then use the VALUES keyword to specify the values we want to insert into each column, in the order in which the columns were listed.

After the query is executed, the "students" table would have a new row with the following values:

```
id   | name     | major            | gpa
-----------------------------------------
1234 | John Doe | Computer Science | 3.5
```

In this example, we have inserted a new row into the “students” table using the INSERT statement.

10. UPDATE

The **UPDATE** statement is used to modify existing data in a database table. You should master using UPDATE to update the values of one or more columns in a table. Here’s an example of using the **UPDATE** statement in SQL:

Suppose we have a table named “students” with columns for “id”, “name”, “major”, and “gpa”. We want to update the major and GPA of a student with an ID of 1234. We can use the **UPDATE** statement to do this:

```
UPDATE students
SET major = 'Mathematics', gpa = 3.7
WHERE id = 1234;
```

This query would update the “major” and “gpa” columns of the row in the “students” table with an ID of 1234. The **UPDATE** statement specifies the name of the table we want to update, followed by the SET keyword and a list of column-value pairs that we want to update. We then use the WHERE clause to specify which rows we want to update. In this case, we want to update the row with an ID of 1234, so we specify “WHERE id = 1234”.

After the query is executed, the “students” table would have the updated values for the “major” and “gpa” columns in the row with an ID of 1234:

```
id   | name     | major       | gpa
--------------------------------------
1234 | John Doe | Mathematics | 3.7
```

In this example, we have updated the “major” and “gpa” columns of a row in the “students” table using the **UPDATE** statement.

11. DELETE

The DELETE statement is used to delete one or more rows from a database table. You should master using **DELETE** to remove data from a table. Here’s an example of using the **DELETE** statement in SQL:

Suppose we have a table named “students” with columns for “id”, “name”, “major”, and “gpa”. We want to delete a student with an ID of 1234 from the table. We can use the **DELETE** statement to do this:

```
DELETE FROM students
WHERE id = 1234;
```

This query would remove the row with an ID of 1234 from the “students” table. The **DELETE** statement specifies the name of the table we want to delete from, followed by the WHERE clause to specify which rows we want to delete. In this case, we want to delete the row with an ID of 1234, so we specify “WHERE id = 1234”.

After the query is executed, the “students” table would no longer have the row with an ID of 1234:

```
id   | name     | major       | gpa
--------------------------------------
5678 | Jane Doe | Computer Science | 3.5
```

In this example, we have used the **DELETE** statement to remove a row from the “students” table.

12. DROP

The DROP statement is used to delete a database table or other database object. You should master using DROP to remove unnecessary tables or other objects from a database. . The syntax for the DROP statement varies depending on the type of object being deleted, but some common examples are:

1. **DROP TABLE:** This statement is used to delete an existing table along with all its data and indexes. The syntax for the DROP TABLE statement is:

```
DROP TABLE table_name;
```

2. **DROP INDEX:** This statement is used to delete an existing index from a table. The syntax for the DROP INDEX statement is:

```
DROP INDEX index_name ON table_name;
```

3. DROP VIEW: This statement is used to delete an existing view. The syntax for the DROP VIEW statement is:

```
DROP VIEW view_name;
```

4. **DROP PROCEDURE:** This statement is used to delete an existing stored procedure. The syntax for the DROP PROCEDURE statement is:

```
DROP PROCEDURE procedure_name;
```

Note that the exact syntax for the DROP statement may vary depending on the specific database management system (DBMS) being used. Also, be careful when using the DROP statement, as it permanently deletes the specified object and all associated data and indexes. Make sure to back up your data before using the DROP statement.

13. ALTER

The **ALTER** statement is used to modify the structure of a database table or other database object. You should master using ALTER to add or remove columns, change data types, or modify other aspects of a table. The syntax for the ALTER statement varies depending on the type of object being modified, but some common examples are:

1. **ALTER TABLE:** This statement is used to modify the structure of an existing table, such as adding or deleting columns, changing data types, or setting constraints. The syntax for the ALTER TABLE statement is:

```
ALTER TABLE table_name
ADD column_name data_type [constraint],
MODIFY column_name data_type [constraint],
DROP column_name,
ADD CONSTRAINT constraint_name constraint_definition,
DROP CONSTRAINT constraint_name;
```

2. **ALTER INDEX:** This statement is used to modify the structure of an existing index, such as adding or removing columns or changing the index type. The syntax for the ALTER INDEX statement is:

```
ALTER INDEX index_name
ADD column_name,
DROP column_name;
```

3. **ALTER VIEW:** This statement is used to modify the definition of an existing view, such as changing the SELECT statement used to create it. The syntax for the ALTER VIEW statement is:

```
ALTER VIEW view_name
AS select_statement;
```

Note that the exact syntax for the ALTER statement may vary depending on the specific database management system (DBMS) being used.