**Essential SQL Commands for Data Science.**

SQL in full stands for Structured Query Langauge. It is a widely known programming language mainly used for managing correlated databases. Such databases store data in form of a table, comprising rows and columns. Rows represent the data attribute columns indicating the numerous relationships between data values.

Being a programming language, SQL uses multiple functions to manipulate data as per the needs of the user in this case; the data scientist. Some of the common SQL commands are:

* SELECT
* UPDATE
* DELETE
* ALTER DATABASE
* CREATE DATABASE
* CREATE TABLE and so on.

Below are the essential SQL commands for data science:

**Essential SQL Commands for SQL.**

1. **Simple Data Extraction Commands.**

**SELECT FROM –** this helps in getting all the information on a particular table from a database. Example:

**SELECT \*** (this selects every column in the specified table below)

**FROM Employees** (employees being the name of the table)

**DISTINCT –** this helps in removing duplicates when selecting values. E.g:

**SELECT DISTINCT (name)**

**FROM employees**

1. **Data Retrieval with Simple Conditions.**

**WHERE –** this helps in filtering data according to stated criteria. E.g

**SELECT \***

**FROM employees**

**WHERE employee\_income >= 2000**

**ORDER BY –** this helps in further comprehending the queried results.

For example, we can order the length of the employee name to be in ascending order by writing the following query on the query editor

**SELECT \***

**FROM employees**

**WHERE employee\_income >= 2000**

**ORDER BY length ASC**

**LIMIT –** helps in returning a limited number of entries according to the specified criterion. E.g

**SELECT \***

**FROM employees**

**WHERE employee\_income >= 2000**

**ORDER BY length ASC**

**LIMIT 10**

In the above example, we are only interested in the first 10 longest employee names whose income is greater than or equal to 2000.

1. **Combinations.**

**GROUP BY & COUNT() –** aggregations or combinations are often used to get sum-ups of a data set to obtain various perceptions. Aggregations are commonly used with the **GROUP BY** clause. E.g, For example, if we need to know how many sales each worker has gained so far, we can count the number of sales and order the results with the most sales on top as shown below;

**SELECT employee\_name , COUNT(sales)**

**FROM employees**

**GROUP BY employee\_id**

**ORDER BY COUNT(sales) ASC**

**SUM –** this is used to find the total of a specified column as shown in the example below:

**SELECT employee\_name, SUM (sales)**

**FROM employees**

**GROUP BY employee\_id**

**ORDER BY COUNT(sales) ASC**

**AVG & HAVING() –** data scientists can include alternative conditions using **HAVING** after calculating a grouped aggregation. E.g:

**SELECT employee\_name, AVG (sales)**

**FROM employees**

**GROUP BY employee\_id**

**HAVING AVG (sales) > 120**

In general, the **HAVING** clause in SQL specifies that an SQL **SELECT** statement must only return rows where accumulated entries meet the stated criteria.

**MIN & Alias – MIN** function returns the shortest/ least/ smallest value of the selected column. E.g;

**SELECT MIN(sales) AS least\_sales\_revenue**

**FROM employees**

However, aggregation is not always a requirement in aggregation as in the example above.

1. **JOINS.**

**INNER JOIN (JOIN) –** Joins generally help data scientists in looking at data from multiple tables. Tables are usually joined using a similar key or column that is found in both tables e.g;

**SELECT \***

**FROM employees e**

**JOIN sales s**

**ON e.sales\_id = s.sales\_id**

**WHERE sales >=300**

Also, data scientists can use the inner join in reporting multiple tables e.g

**SELECT \***

**FROM employees e**

**JOIN sales s**

**ON e.sales\_id = s.sales\_id**

**JOIN clients c**

**ON e.customer\_id = c.customer\_id**

**GROUP BY employee\_id**

**ORDER BY employee\_name ASC**

1. **Changing Data Types.**

**CAST() –** this complex function that transforms one or more values from one data type into another. E.g; in the example below, the average sales amount is transformed from being just figures to actual currency when we execute the following code:

**SELECT employee\_name, CAST(SUM(sales) AS MONEY)**

**FROM employees**

**GROUP BY employee\_id**

**ORDER BY SUM(sales) ASC**

**ROUND() –** this function helps us round off numbers e.g;

**SELECT employee\_name , ROUND(AVG (sales), 2) AS average\_sales**

**FROM employees**

**GROUP BY employee\_id**

In the example above, the **ROUND** function rounds off the average sales value to two decimal places as indicated.

1. **Complex Conditions.**

**CASE Statement -**  this statement returns a value for the specified condition.

It evaluates a list of conditions and returns one of the multiple possible results. Data scientists frequently use the case expression in SQL stored procedures or as a formula for a particular column which optimizes the SQL statements. The keyword **CASE** indicates the beginning of the statement, while **END**  indicates the end of a statement. For a single condition, you can use **WHEN** then the accompanying condition. Afterward,  **THEN** precedes the value for the specified condition. (**WHEN {condition} THEN {value for the condition}).**  This can be accompanied by other **WHEN/THEN**  statements. In the end, the data scientist can add a value to use by default if none of the conditions are true using the **ELSE** keyword. Example;

**SELECT \***

**CASE**

**WHEN sales <= 100 THEN “below average”**

**WHEN sales BETWEEN 100 AND 400 THEN ‘average’**

**WHEN sales >= 400 THEN ‘above average’**

**ELSE ‘unrecorded’**

**END**

**FROM employees;**

**SUBQUERIES -**  subqueries are **SELECT**  statements that are part of another statement. E.g;

**SELECT employee\_name, SUM(sales)**

**FROM employees**

**WHERE sales > (**

**SELECT \***

**CASE**

**WHEN sales <= 100 THEN “below average”**

**WHEN sales BETWEEN 100 AND 400 THEN ‘average’**

**WHEN sales >= 400 THEN ‘above average’**

**ELSE ‘unrecorded’**

**END**

**FROM employees; )**

**COMMON TABLE EXPRESSIONS (CTEs) –** these function as virtual tables with records and columns, created during the execution of a query, used by the query, and eliminated after query execution. Regularly, they act as a link to change the data in source tables to the format the query anticipates. The CTE starts with the keyword **WITH**, followed by the CTE name in parenthesis (). The main query precedes the closing of the parenthesis.

Example:

**WITH sales\_performance AS (**

**SELECT \***

**CASE**

**WHEN sales <= 100 THEN “below average”**

**WHEN sales BETWEEN 100 AND 400 THEN ‘average’**

**WHEN sales >= 400 THEN ‘above average’**

**ELSE ‘unrecorded’**

**END**

**FROM employees; )**

**SELECT AVG(sales) AS average\_sales**

**FROM employees e**

**GROUP BY sales\_id**

**LIMIT 100**

**WINDOW FUNCTIONS :**

1. **OVER & PARTITION BY()** – data scientists use these to specify the column on which the aggregation is to be performed. E.g

**SELECT \*, SUM (sales) , OVER(PARTITION BY sales\_date) AS total\_daily\_sales**

**FROM employees**

In summary, data scientists occasionally utilize SQL in their daily operations in conjunction with other programming languages such as Python and R. SQL provides a convenient environment as it can be used across multiple platforms such as Oracle, PostgreSQL, and MySQL workbench to name a few.