**SQL BASIC TRAINING GUIDE**

This document designed to give you a meaningful structure of SQL. It starts from the absolute basics (SELECT \* FROM table\_name;) and guides you to the intermediate level (JOIN, HAVING, CALCULATIONS). I added everything that you will need for data analysis.

The ideal use case of this doc is that you print it in color and keep it next to you while you are learning and practicing SQL on your computer.

# BASE QUERY

**SELECT \* FROM table\_name;**

This query returns every column and every row of the table called table\_name.

**SELECT \* FROM table\_name LIMIT 10;**

It returns every column and the first 10 rows from table\_name.

# SELECTING SPECIFIC COLUMNS

**SELECT column1, column2, column3 FROM table\_name;**

This query returns every row of column1, column2 and column3 from table\_name.

# CREATING NEW CALCULATIONS

**SELECT column1, column2, column3 , column1+column2 as column4 FROM table\_name;**

This query returns every row of column1, column2 ,column3 and a new column4 which is an addition of column1 and column2 from table\_name.

# FILTERING (the WHERE CLAUSE)

**SELECT \* FROM table\_name WHERE column1 = 'expression';**

"Horizontal filtering." This query returns every column from table\_name - but only those rows where the value in column1 is 'expression'. Obviously this can be something other than text: a number (integer or decimal), date or any other data format, too.

# ADVANCED FILTERING

Comparison operators help you compare two values. (Usually a value that you define in your query and values that exist in your SQL table.) Mostly, they are mathematical symbols, with a few exceptions:

|  |  |
| --- | --- |
| **Comparison operator** | **What does it mean?** |
| **=** | Equal to |
| **<>** | Not equal to |
| **!=** | Not equal to |
| **<** | Less than |
| **<=** | Less than or equal to |
| **>** | Greater than |
| **>=** | Greater than or equal to |
| **LIKE ‘%expression%’** | Contains ‘expression’ |
| **IN (‘exp1’, ‘exp2’, ‘exp3’)** | Contains any of ‘exp1’, ‘exp2’, or ‘exp3’ |

**A few examples:**

**SELECT \* FROM table\_name WHERE column1 != 'expression';**

This query returns every column from table\_name, but only those rows where the value in column1 is NOT 'expression'.

**SELECT \* FROM table\_name WHERE column2 >= 10;**

It returns every column from table\_name, but only those rows where the value in column2 is greater or equal to 10.

**SELECT \* FROM table\_name WHERE column3 LIKE ‘%xzy%’;**

It returns every column from table\_name, but only those rows where the value in column3 contains the 'xyz' string.

**SELECT \* FROM table\_name WHERE column4 in ( ‘abc’,’def’,’fgr’);**

It returns every column from table\_name, but only those rows where the value in column4 is either one of **‘abc’,’def’,’fgr’.**

# MULTIPLE CONDITIONS

You can use more than one condition to filter. For that, we have two logical operators: OR, AND.

**SELECT \* FROM table\_name WHERE column1 != ‘expression’ AND column3 LIKE**

## ‘%xzy%’;

This query returns every column from table\_name, but only those rows where the value in column1 is NOT ‘expression’ AND the value in column3 contains the 'xyz' string.

**SELECT \* FROM table\_name WHERE column1 != ‘expression’ OR column3 LIKE**

## ‘%xzy%’;

This query returns every column from table\_name, but only those rows where the value in column1 is NOT ‘expression’ OR the value in column3 contains the 'xyz' string.

# SORTING VALUES

**SELECT \* FROM table\_name ORDER BY column1;**

This query returns every row and column from table\_name, ordered by column1, in ascending order (by default).

**SELECT \* FROM table\_name ORDER BY column1 DESC;**

This query returns every row and column from table\_name, ordered by column1, in descending order.

# UNIQUE VALUES

**SELECT DISTINCT(column1) FROM table\_name;**

It returns every unique value from column1 from table\_name.

# CORRECT KEYWORD ORDER

SQL is extremely sensitive to keyword order.

So make sure you keep it right:

1. **SELECT**
2. **FROM**
3. **WHERE**
4. **ORDER BY**
5. **LIMIT**

# SQL FUNCTIONS FOR AGGREGATION

In SQL, there are five important aggregate functions for data analysts/scientists:

* **COUNT()**
* **SUM()**
* **AVG()**
* **MIN()**
* **MAX()**

A few examples:

**SELECT COUNT(\*) FROM table\_name WHERE column1 = 'something';** It counts the number of rows in the SQL table in which the value in column1 is 'something'.

**SELECT AVG(column1) FROM table\_name WHERE column2 > 1000;**

It calculates the average (mean) of the values in column1, only including rows in which the value in column2 is greater than 1000.

# SQL GROUP BY

The GROUP BY clause is usually used with an aggregate function (COUNT, SUM,

AVG, MIN, MAX). It groups the rows by a given column value (specified after GROUP BY) then calculates the aggregate for each group and returns that to the screen.

**SELECT column1, COUNT(column2) FROM table\_name GROUP BY column1;** This query counts the number of values in column2 - for each group of unique column1 values.

**SELECT column1, SUM(column2) FROM table\_name GROUP BY column1;** This query sums the number of values in column2 - for each group of unique column1 values.

**SELECT column1, MIN(column2) FROM table\_name GROUP BY column1;**

This query finds the minimum value in column2 - for each group of unique column1 values.

**SELECT column1, MAX(column2) FROM table\_name GROUP BY column1;**

This query finds the maximum value in column2 - for each group of unique column1 values.

# SQL HAVING

The execution order of the different SQL keywords doesn't allow you to filter with the WHERE clause on the result of an aggregate function (COUNT, SUM, etc.). This is because WHERE is executed before the aggregate functions. But that's what HAVING is for:

**SELECT column1, COUNT(column2)**

**FROM table\_name**

**GROUP BY column1**

**HAVING COUNT(column2) > 100;**

This query counts the number of values in column2 - for each group of unique column1 values. It returns only those results where the counted value is greater than 100.

# SQL ALIASES

You can rename columns, tables, subqueries, anything.

**SELECT column1, COUNT(column2) AS number\_of\_values FROM table\_name**

**GROUP BY column1;**

This query counts the number of values in column2 - for each group of unique column1 values. Then it renames the COUNT(column2) column to number\_of\_values.

# SQL JOIN

You can JOIN two (or more) SQL tables based on column values.

1. **INNER JOIN**

**SELECT \***

**FROM table1**

## INNER JOIN table2

**ON table1.column1 = table2.column1;**

This joins table1 and table2 values - for every row where the value of column1 from table1 equals the value of column1 from table2.

1. **LEFT JOIN**

**SELECT \***

**FROM table1**

## LEFT JOIN table2

**ON table1.column1 = table2.column1;**

This joins table1 and table2 values – It gives all the rows from left table **table1** and matching data from **table2** where the value of column1 from table1 equals the value of column1 from table2.

# CORRECT KEYWORD ORDER AGAIN

SQL is extremely sensitive to keyword order.

So make sure you keep it right:

1. **SELECT**
2. **FROM**
3. **JOIN (ON)**
4. **WHERE**
5. **GROUP BY**
6. **HAVING**
7. **ORDER BY**
8. **LIMIT**

# IMPORTANT STRING AND DATE FUNCTIONS

# LEN(string) :This gives the length of the string.

**SUBSTRING(string,2,3)** : Gives 3 character of a string starting from second position.

**LEFT( string, 4 )** : This gives left 4 characters of a string.

**DATEPART(year, date)** : This gives year part of the date.

**DATEDIFF(days, start\_date,end\_date)** : This gives no of days between 2 dates.

**DATEADD(days, 5,date)** : This will add 5 days to the given date

**Full list of functions:**

<https://docs.aws.amazon.com/redshift/latest/dg/String_functions_header.html>

# SQL CASE WHEN

You can create a new column based on a if else kinf of rules.

**SELECT column1 ,**

**CASE**

**WHEN column1>100 then ‘High’**

**WHEN column1 BETWEEN 50 AND 100 THEN ‘Medium’**

**ELSE ‘Low’ end as marks\_range**

**FROM table1**

This will give you a new column in output based on case when condition for every row from table1.