SQL Basic Tutorial

SELECT-

- indicates which columns you'd like to view,
- only one SELECT **statement** can be run at a time
- whenever you select multiple columns, they must be separated by commas, but you should **not** include a comma after the last column name.
- if you want to select every column in a table, you can use ***** instead of column names.
- Column name format- use underscores instead of spaces also the table name itself also uses underscores instead of spaces, most people avoid putting spaces in column name because it's annoying to deal with spaces in SQL – is you want to have space in column name **you need to always refer to those columns in double quotes.**

AS-

- If you'd like your results to look a bit more presentable, you can rename columns o include spaces. For example, if you eat the **west** column to appear as **West Region** in the results you would have to type:

```
SELECT west AS "West Region"
FROM table_name
```

- Without the double quote, the query would read 'West' and 'Region' and not "West Region" and would return an error.
- If we put column names in **double quotes** the results will be return **capital letter.**
- If we us **dash** bla_bla return results will be **lower-case column names**.

FROM- identifies the table that the live in.

LIMIT-

- Prevent you from accidentally returning millions of rows without mining

```
SELECT *
FROM table_name
LIMIT 100
```

WHERE-

- filtering the data.
- The clauses always need to be in this order: **SELECT->FROM-> WHERE**
- Where contain the column name that contains values -> the result will only include this row.
- Where filters based on values in one column, you'll limit the results in all columns to rows that satisfy the condition. The idea is that each row is one data point or observation, and all the information contained in that row belongs together.
- You can filter your results in a number if ways using comparison and logical operators.

```
SELECT *
 FROM table_name
WHERE col_name=value
```

Comparison operators on numerical data:

Equal to	=
Not equal to	<> or !=
Greater than	>
Less than	<
Greater than or equal to	>=
Less than or equal to	<=

Comparison operator on non-numerical data:

- All of the above operators work on non-numerical data as well.
- **=** and**!=** make perfect sense-they allow you to select rows that match or don't match any value, respectively.
- If you're using an operator with values that are non-numeric, you need to put the value in single quotes: **'value'**
- **SQL uses single quotes to reference column values.**
- If you're using **>, <,>= or <=,** operators on non-numerical columns as well -they filter based on alphabetical order.
- You don't necessarily need to be too specific about how you filter, if you are using one letter be carefully because SQL consider 'Ja' to be greater that 'J' because it has an extra letter

Arithmetic in SQL

- You can perform arithmetic in SQLS using the same operators like in Excel: **+, -, * and /**
- SQL can perform arithmetic only across columns on values in a given row, you can add values in multiple columns from the same row together using **+ or** - if you want add values across multiple rows, you'll need to **aggregate function**

```
SELECT Col1,
       Col2,
       Col3
       Col1+col3-4*col2 AS new_col_name
FROM  table_name
```

- **Derived columns-** columns that contain the arithmetic functions, because they are generated by modifying the information that exist in the underlying data.
- Like in Excel you can use **parentheses** to manage the order operations. You also can use parentheses even when it's not absolutely necessary just to make your query easier to read.

SQL Logical operators

You'll also won't to filter data using several conditions-possibly more often than you'll want to filter by only one condition. Logical operators in one query.

LIKE-

- Allow you to match similar values, instead of exact values.

```
SELECT *
FROM table_name
WHERE "col_name" LIKE 'values_in_the_col%"
```

- Include rows for which start with **'blabla%'** And is followed by any number and selection of characters.

ILAKE-

- To ignore case when you're matching values.
- ILIKE is similar to LIKE in all aspects except in one thing: it performs a case in-sensitive matching:

% -

- Represent any character or set of characters, is referred to as a 'wildcard'
- Like is case-sensitive.
- The percent sign represents zero, one, or multiple characters

_-

- Single underscore to substitute for an individual character

```
SELECT *
FROM table_name
WHERE artist ILIKE 'dr_ke'
```

LIKE OPERATOR	DESCRIPTION
WHERE CUSTOMERNAME LIKE 'A%'	Finds any values that start with "a"
WHERE CUSTOMERNAME LIKE '%A'	Finds any values that end with "a"
WHERE CUSTOMERNAME LIKE '%OR%'	Finds any values that have "or" in any position
WHERE CUSTOMERNAME LIKE '_R%'	Finds any values that have "r" in the second position
WHERE CUSTOMERNAME LIKE 'A_%'	Finds any values that start with "a" and are at least 2 characters in length
WHERE CUSTOMERNAME LIKE 'A__%'	Finds any values that start with "a" and are at least 3 characters in length
WHERE CONTACTNAME LIKE 'A%O'	Finds any values that start with "a" and ends with "o"

IN-

- Allows you to specify a list of values you'd like to include.
- Return results for which column is equal to one of the values in the **list=> (1,2,3)**
- If we use non-numerical values. You need to go inside single quotes.
- Regardless of the data type, the values in the list must be separated by commas.

```
SELECT *
FROM table_name
WHERE col_name IN (('bla1','bla2','bla3')
```

BETWEEN-

- allows you select only rows within a certain range.
- It has to be paired with the **AND** operator

```
SELECT *
FROM table_name
WHERE col_name BETWEEN a AND b
```

- Include the range bounds that you specify in the query, in addition to the values between them.

```
SELECT *
FROM table_name
WHERE col_name >= a AND col_name <=b
```

- some people prefer this because it more explicitly show what the query is doing (it's easy to forget whether or not BETWEEN includes the range bounds).

IS NULL-

- allows you to select only rows that contain no data in a given column.
- Is a logical operator in SQL that allows you to **exclude** rows with missing data from your results.
- Some tables contain null values-cell with no data in them at all.
- You can select rows that contain no data in a given column by using **IS NULL.**
- You can't perfume arithmetic on null values => **WHERE col_name=NULL** will not work.

```
SELECT *
FROM  table_name
WHERE col_name IS NULL
```

AND-

- allow you to select only rows that satisfy two conditions.
-

```
SELECT *
FROM table_name
WHERE col_name1 >= a AND col_name2 <=b
```

- You can use SQL's **AND** operator with additional **AND** statements or any comparison operator, as many times as you want.

```
SELECR*
FROM  table_name
WHERE year=2012
AND year_Rank<10
AND "group" ILIKE '%feat%'
```

- Example is spaced out onto multiple line-a good way to make long **WHERE** clauses more readable

OR-

- allows you to select rows that satisfy either of two conditions
- It works the same way as **AND** , which selects the rows that satisfy both of two conditions.
- Each row will satisfy one of the two conditions.
- You can combine **AND** with **OR** using parenthesis.

```
SELECT *
FROM tutorial.billboard_top_100_year_end
 WHERE "group" ILIKE '%DR. Dre%'
 AND (year <=2000 OR year >=2010)
```

NOT-

- allows you to select rows that do not match a certain condition.
- You can put before any conditions statement to select rows for which that statement is false.

```
 SELECT *
FROM table_name
WHERE year =2013
AND year_rank NOT BETWEEN 2 AND 3
```

- Using **NOT** with **> and <** usually doesn't make sense because you can simply use the opposite comparative operator instead.
- **NOT** is commonly used with **LIKE\ILIKE**
- **NOT** is also frequently used to identify non-null rows, but the syntax is somewhat special- you need to include **IS** beforehand.

```
SELECT *
FROM table_name
WHERE year=2013
AND artist IS NOT NULL
```

GROUP-

- is actually the name of a function in SQL. The **double quotes** are a way of indicating that you are referring to the column name, not the SQL function. In general, putting double quotes around a word or phrase will indicate that you are referring to that column name.

ORDER BY-

- Now you will learn how to sord data.
- The **ORDER BY** clause allow you to reorder your results based on the data in one or more columns.

```
SELECT *
FROM  table_name
ORDER BY col_name
```

- you'll notice that the results are now ordered alphabetically from a to z based on content in the artist column in **ascending order,** it will start with smaller (or most negative) numbers, with each successive row having a higher numerical value than the previous.

DESC –

- operator -results in the opposite order (referred to as descending order)
- Ordering data by multiple columns- you can also order by multiple columns. This is particularly useful if your data fall into categories and you'd like to organize rows by data

```
SELECT *
FROM table_name
WHERE year_rank<=3
ORDER BY year DESC, year_rank
```

- The columns in the **ORDER BY** clause must be separated by commas.
- The **DESC** operator is only applied to the column that precedes it.
- The result is sorted by the first column mentioned (year), then be (year_rank) afterward.
- You can make your life a little easier by **substituting** numbers for column names In the **ORDER BY** clause**.** The numbers will correspond to the order in which you list columns in the **SELECT** clause
- When using **ORDER BY** with a row limit (either through the check box on the query editor or by typing **LIMIT)**, the ordering clause is executing first. This means that the results are ordered **before** limiting to only a few rows.

Using comments

You can "comment out" pieces of code by adding combinations of characters. In other words, you can specify part of your query that will not actually be treated like SQL code. It can be helpful to include comments that explain your thinking so that you can easily remember what you intended to do if you ever want to revisit you work. Commenting can also be useful if you want to test variations on your query while keeping all of your code intact.

-- -

- Two dashes to comment out everything go the right of them on a given line.
-

```
SELECT * --this comment won't affect the way the code run
FROM  table_name
WHERE year=2013
```

/* blablabla blab
Blablblblabla
Blablabla. */ -

- You can also leave comments across multiple line using this