EVERYTHING YOU WANTED TO KNOW ABOUT SQL BUT WERE TOO AFRAID TO ASK

A beginners guide to understand SQL

Santiago Cruz Todd

Exp: 260239

Professor: Edgar Ruiz

Index

INTRODUCTION

Programmings has always been the seed for the most innovative, (and actual), scientific and tecnological breakthroughs, and as the world has adopted these new technologies the need for a modern "*infraestructure*" to manage the type of data that is used has become fundamental.

Here is when we get to the most vital part of this handbook, what's SQL?

Well, it is a language designed for technical and non-technical users query, manipulate relational databases. It is one of the oldest programming languages, being invented around five decades ago it's still one of the most used in this day.

Let's go a little bit more technical about it, SQL is the acronym for *Structured Query Language*, and often its uses what we're going to call Relational Databases, this type of database differentiates from others because data is organized into tables containing rows and columns, these we are going to known them as records and fields, respectively. In many other materials you are going to find that a table is also known as a relation (inside the SQL terminology). This databases require to be structured in predefined *schemas*, this is also how the database organizes the information in a understandable form, this is useful when you try to relate or inter-connect two databases, having order in the way you arrange your databases – it will simply make your life easier!

Many systems from companies used this query-based tool as their main coprorate system, but you may be asking, what's a query? It's simply what you want to retrieve (as data) from your main base. You need to understand that under this query-based language you can do the following actions: Create, Read, Update and Delete.

How you do all this stuff?, simply by writing a statement. With a statement you have all the power to manage the information available in the company, but the trick is that this statement needs to be written in the way SQL planned it. Everything within the SQL language follows a hierarchical model, and it's just the best way to manage large quantities of information inside a company. One of the best qualities about SQL is that it's database-oriented system is safe and scalable for the storage.

We have simply detailed until now how SQL works in mere words, but here is where the fun begins, I plan to give you the guide for any non-technical person out there that has the hope to become a connoisseur in this programming language.

Before teaching you how to create a STATEMENT I'm going to give you some simple concepts for you to have in mind when creating databases.

BASIC KNOWLEDGE

Record: Is a row that holds data on a single entity.

Field: A column that stores specific piece of informtion for all records in the table.

These two form a two dimensional structure most known as a table, and there is no limit as how many data you can put in rows.

Some recommendations at the moment of creating your tables are these:

Avoid troubles when looking for a table, name them in lowercase and without spaces, use a underscore instead, like this: (books).

And for the best, use general-collective names when naming a table, then specify the rows, e.g., (books_fiction), (books_suspense).

For fields (which are columns!) always use the singular name, and never commit the mistake of using the same name as its prime table, and viceversa. Same as the previous recommendation, use lowercase.

TYPES OF DATA IN SQL

A basic principle is that each field must have a unique data type, this is important because depending on it SQL can store it in different ways.

NUMERIC DATA

The **first** type of data is the Numeric.

Numeric values can be in the detailed form of:

- Interger
- Decimal
- Floating

And these specific data types can be input in certain functions that work according the stated type of data, but we'll see that in a moment.

Int is for INTERGER: Numeric values without a decimal:

263 -457 +963

And each data type has its own variations, for numerica values we can find:

Signed range: This is for positive and begative intergers.

Unsigned range: For positive intergres ONLY

What is most common is the signed range, if you want to ask for a unsigned range you would have to declare it in the statement following this way:

```
CREATE TABLE number_range(

        Interger_column INT UNSIGNED

);
```

It's useful to also know how the range of values allowed for each interger type:

Data Type	Range Allowed	Storage
TINYINT	-128 to 127, 0 to 255 (unsgd.)	1 byte
SMALLINT	–32,768 to 32,767 0 to 65,535 (unsgd.)	2 byte
MEDIUMINT	–8,388,608 to 8,388,607 0 to 16,777,215 (unsgd.)	3 byte
INTERGER or INT	–2,147,483,648 to 2,147,483,647 0 to 4,294,967,295 (unsgd.)	4 byte
BIGINT	–263 to 263 – 1 0 to 264 – 1 (unsgd.)	8 byte

BIGINT is used for large numbers like card numbers or user ID's.

The second type of numeric data is DECIMAL:

Also known as fixed point numbers, this include a decimal point and are stored as an exact value, this is very useful when storing monetary data like $54.99, and you need to state it in the following way:

```
CREATE TABLE number_type (

        decimal_example DECIMAL(4,2)

);
```

When defining the data type (4,2) consider this:

- 4 is the maximum number of total digits that are stored, this goes for precision.
- 2 is the number of digits to the right of the decimal point, and goes for scale.

So, your number would end up looking like this:

89.40

Decimals are of extremely importance when managing financial data!

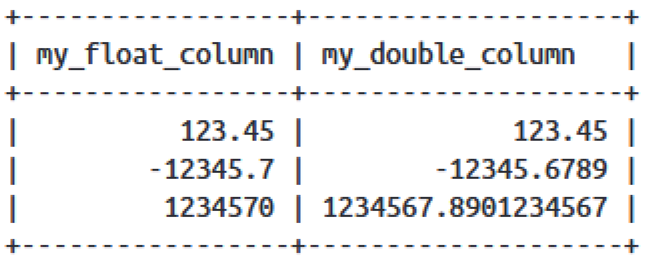
Our third type of numeric data is FLOAT or DOUBLE: This type stores a different type of numbers, when a numeric value has many digits, either before or after the decimal point what FLOAT does is that, instead of storing all the digits, floating point numbers only store a limited number of them to save space.

Consider using FLOAT values when you are trying to save the most of storage possible, but take note that this will decrease the precision of your management of data, often resulting in inaccuracies and some times rounding errors that could be fatal for a company's financial control.

Example of statement:

```
CREATE TABLE float_numbers (

float_column FLOAT,

double_column DOUBLE

);

INSERT INTO my_table VALUES

(123.45, 123.45),

(-12345.6789, -12345.6789),

(1234567.890123456789, 1234567.890123456789);

SELECT * FROM my_table;
```

And it would end up looking like this in a table:



```
+-----------------+--------------------+
| my_float_column | my_double_column   |
+-----------------+--------------------+
|          123.45 |             123.45 |
|        -12345.7 |        -12345.6789 |
|         1234570 | 1234567.8901234567 |
+-----------------+--------------------+
```

Our **second** category of data is the data type known as CHAR or Character data type.

CHAR is for CHARACTER and follows the directive that is the statement is CHAR(7), then each value in the column will have exacty 7 characters – in this case the string length is fixed.

But you may be asking, what is a string?

String values are sequences of characters including letters, numbers and special characters.

String basics

A sentence is a string: ‘a sentence is a string.’

Be careful when using two adjacent single quotes (‘’) in a string:

‘You’’re welcome.’

Because SQL will treat two adjacent single quotes as a single quote within the string and return:

‘You’re welcome.’

The best is to use the quotes to enclose string values, while double quotes (“”) should be for identifiers (like names of tables, columns, etc.).

One trait of the CHAR data type is that it will be right-padded with spaces to be exactly the length specified, se when writing CHAR(7) it would be like this:

CREATE TABLE my_table (

character_value VARCHAR(7)

);

Result: ‘SEVEN__’, adding spaces to fill the length.

The seconnd type for character data type is VARCHAR, that goes for variable character, and it’s the most popular string for data type. If the data type is VARCHAR(25), then the column will allow up to 30 characters, in this case the string length is variable; key difference with CHAR is that VARCHAR only uses the necessary space instead of filling the “blanks” with spaces. If you are going to be working in a project in which you need to optimize your storage, VARCHAR offers a slightly better performance.

CREATE TABLE my_table (

varcharacter_column VARCHAR(30)

);

Result: ‘here is the text’, ‘that include numbers 1 2 3 4’, ‘and puntuation marks!?’

But if you’re looking for an input of paragraphs or more text, then the best data type for you to use is TEXT.

Our third and final relevant data type is the Datetime Data.

This type of information can be input into datetime functions such as DATEDIFF() and EXTRACT(), that will be shown later!

Datetime vales can come in many shapes and forms, like dates, times or datetimes.

First, **date values** consist of a date-column that have the format: YYYY-MM-DD, in this case October 31st, 2000 is written as:

‘2000-10-31’

Time Values posses the format hh:mm:ss, and 12:45 pm is written as:

’12:45:00’

For more precision, you can also include granular seconds, up to six places.

More useful is adding a time zone.

More applied to real cases is a Date and Time values, like when the hour of a financial transaction is confirmed, this come in the format: YYYY-MM-DD hh:mm:ss. So this type combines both previous values:

October 31st, 2000 at 12:45 pm is written as:

‘2000-10-31   12:45’

When referencing a datetime value in query, you must preface the string with either a DATETIME or TIMESTAMP keyword to tell SQL it’s a datetime:

– SELECT TIMESTAMP '2021-02-25 10:30'

The last time data type is YEAR, that usually stores a year value either in a two-digit formar or four-digit format; you can state the different datetimes like this:

```
CREATE TABLE my_table (
dt DATE,
tm TIME,
dttm DATETIME,
ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
yr YEAR
);
```

And inster them thi way:

```
INSERT INTO my_table (dt, tm, dttm, yr)
VALUES ('21-7-4', '6:30',
2021, '2021-12-25 7:00:01');
```

That would end up looking like this:

```
+------------+----------+---------------------+
| dt         | tm       | dttm                |
+------------+----------+---------------------+

| 2021-07-04 | 06:30:00 | 2021-12-25 07:00:01 |
+------------+----------+---------------------+
```

```
+---------------------+------+
| ts                  | yr   |
+---------------------+------+
| 2021-01-29 12:56:20 | 2021 |
+---------------------+------+
```

This are the most common type of data type you're going to meet up with as you start in SQL, now we are going full steam ahead with how to make a statement.

First things first, SQL posses what is known as keywords, that is some sort of text that already has a meaning within its own logic. These keywords can be written weither in uppercase or lowercase, so do not give them much importance when writing your statements, but CAPITALIZING your code is more for the detail of readability.

Both ways SQL is going to run the code the same way.

Keywords are bolded in this text:

> **SELECT** e.name, **COUNT**(s.sale_id) **AS** num_sales
> **FROM** employee e
> **LEFT JOIN** sales s **ON** e.emp_id = s.emp_id
> **WHERE YEAR**(s.sale_date) = 2021
> **AND** s.closed **IS NOT NULL**
> **GROUP BY** e.name;

Keywords are already defined in SQL, and I will be telling you how to put them in the right order, but the next part of understanding what you're doing is the identifiers of your code, this are the terms you use and define, just mind the recommendations given before at the moment of naming them.

Identifiers are bolded in the text:

> SELECT **e.name**, COUNT**(s.sale_id)** AS **num_sales**
> FROM **employee e**
> LEFT JOIN **sales s** ON **e.emp_id = s.emp_id**
> WHERE YEAR**(s.sale_date)** = 2021
> AND **s.closed** IS NOT NULL
> GROUP BY **e.name**;

Identifiers are the name of a specific database object, can be a table or a column, you can use a combination of letters, numbers and underscores (watch out the spaces!).

Remember the readability thing? For the sake of it, identifiers are typically lowercase while keywords are uppercase, just...keep that in mind, the code will run anyway. One very important thing you need to consider is that identifiers should not be given the same name as an existing keyword.

Now a statement starts with a keyword and ends with a semicolon, the entire structure and funcionality depends on it, the code block is called a SELECT statement if it starts with the keyword SELECT, like the example below:

```
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
LEFT JOIN sales s ON e.emp_id = s.emp_id
WHERE YEAR(s.sale_date) = 2021
AND s.closed IS NOT NULL
GROUP BY e.name;
```

The SELECT statement is the most popular one due to its wide range of uses and is often called a query instead because it finds data in a database.

A statements basic content are four main clauses:

- The SELECT clause

  SELECT e.name, COUNT(s.sale_id) AS num_sales

- The FROM clause

  FROM employee e

- WHERE clause

  WHERE YEAR(s.sale_date) = 2021

- GROUP BY clause

  GROUP BY e.name**;**

And with this as a whole we can create a query.

As seen before, a query is just a nickname for the SELECT statement, its most complete form consists of six main clauses:

1. SELECT
2. FROM
3. WHERE
4. GROUP BY

5. HAVING
6. ORDER BY

First I'm going to tell you what each of the clauses do and then how does SQL understands the given query.

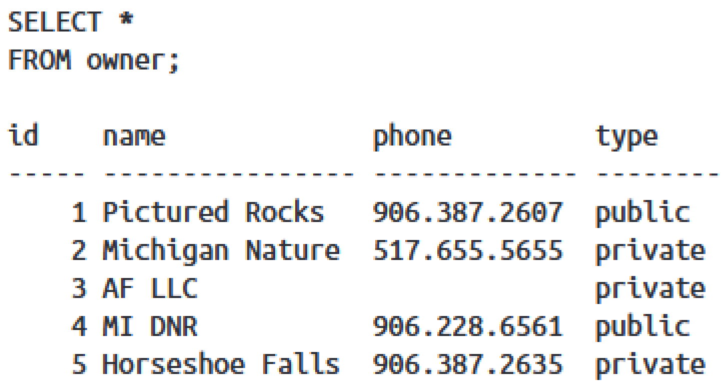
The Select Clause

This speciefies the columns you want a statement to return, SELECT is folowed by a list of column names and expressions that are separated by comas (this is very important, if it's not done right, your code won't run properly). Each of the column name or expression turns a column in the results.

```
SELECT id, name
FROM owner;
id name
----- ----------------
1 Pictured Rocks
2 Michigan Nature
3 AF LLC
4 MI DNR
5 Horseshoe Falls
```

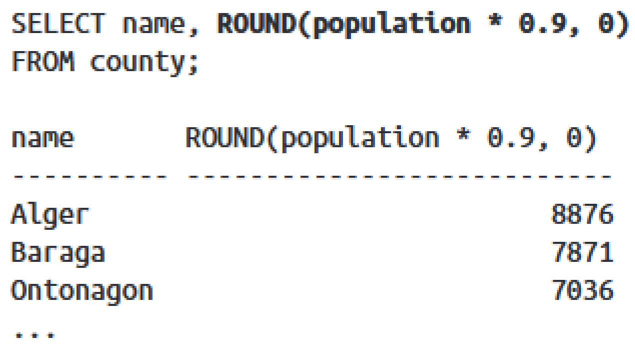
There's a trick if you need to extract all the columns and it's just by writing (*) in your statement.

```
SELECT *
FROM owner
```

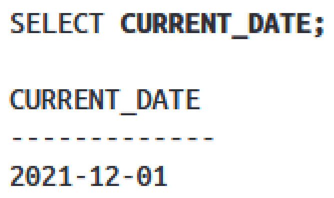


```
SELECT *
FROM owner;

id    name             phone         type
----- ---------------- ------------- --------
    1 Pictured Rocks   906.387.2607  public
    2 Michigan Nature  517.655.5655  private
    3 AF LLC                         private
    4 MI DNR           906.228.6561  public
    5 Horseshoe Falls  906.387.2635  private
```

Beginning wit the SELECT statement is helpful when extracting Expressions and Functions.



```
SELECT name, ROUND(population * 0.9, 0)
FROM county;

name       ROUND(population * 0.9, 0)
---------- --------------------------
Alger                            8876
Baraga                           7871
Ontonagon                        7036
...
```



```
SELECT CURRENT_DATE;

CURRENT_DATE
-------------
2021-12-01
```

In this clause it's important to remark the use of Aliases at the moment of calling-extracting information from columns. An alias renames a column or a table temporarily, only for the duration of the query. The new alias will be displayed in the results of the query, but the original name will remain unchanged in your querying tables. The process of how to apply an Alias is the following:

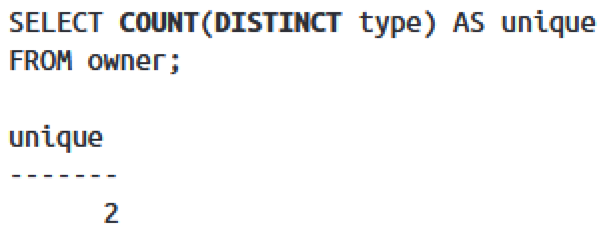
It's commonly done in the first line, using the standard **AS** in renaming columns. To create a column alias, you follow a column name or expression with either an alias name or the AS keyword and adding the alias name.

SELECT **id** **AS county_id**, name,
ROUND(population * 0.90, 0) AS estimated_pop
FROM county;

In this case the black-bolded id changes to country_id.

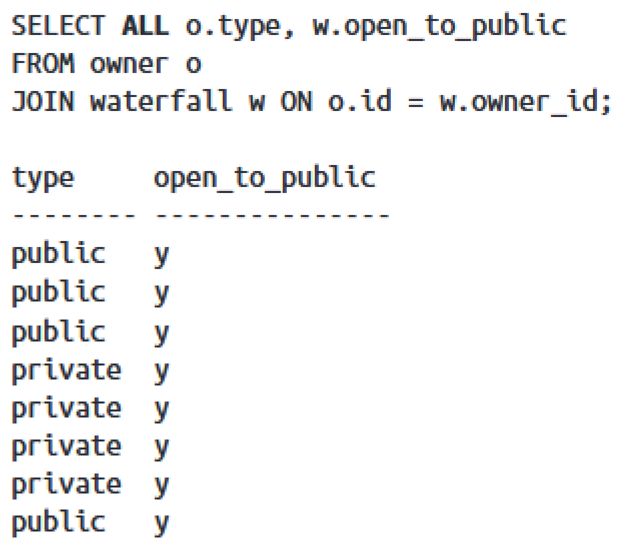
In this SELECT clause some other keywords can be called to extract ceratin information:

Keywords ALL and DISTINCT are used when you try to call all of the rows or if you want to remove duplicate rows from the results.
To count the number of unique values within a single column you can combine the COUNT and DISTINCT keywords within the SELECT clause, this query would return the number of unique type values.



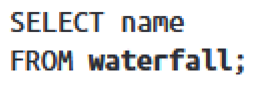
```
SELECT COUNT(DISTINCT type) AS unique
FROM owner;

unique
-------
      2
```

An example of ALL is:



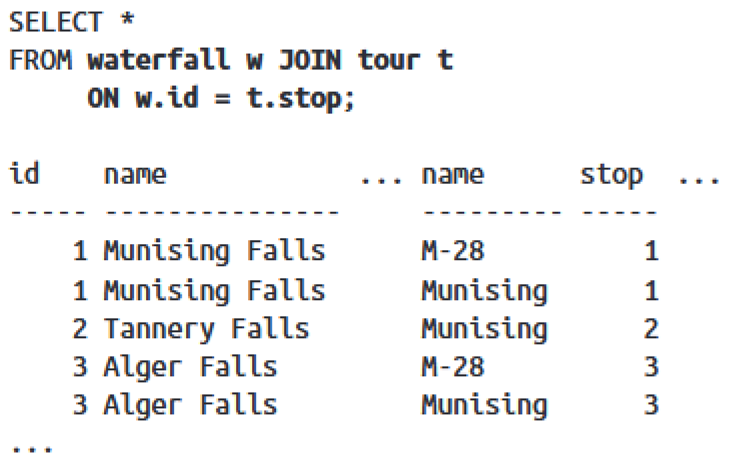
```
SELECT ALL o.type, w.open_to_public
FROM owner o
JOIN waterfall w ON o.id = w.owner_id;

type     open_to_public
-------- ---------------
public   y
public   y
public   y
private  y
private  y
private  y
private  y
public   y
```

Second is the FROM clause, this specifies the source of the data you want to retrieve, the simplest case is to name a single table or view in the FROM clause of query:



```
SELECT name
FROM waterfall;
```

But you can alaso extract as many information from multiple tables, by this the most common method is pulling a JOIN clause within the FROM. The next example retrieves data from the waterfall and tour tables and displays it as a single result table:



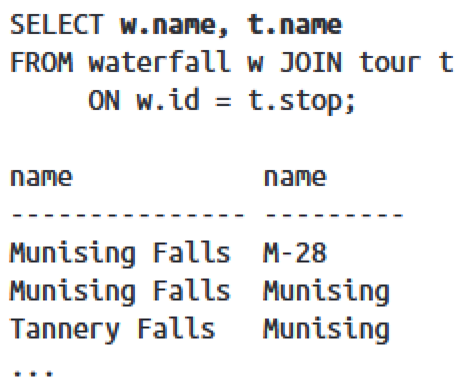
```
SELECT *
FROM waterfall w JOIN tour t
     ON w.id = t.stop;

id    name            ... name      stop  ...
----- ---------------     --------- -----
    1 Munising Falls      M-28          1
    1 Munising Falls      Munising      1
    2 Tannery Falls       Munising      2
    3 Alger Falls         M-28          3
    3 Alger Falls         Munising      3
...
```

Here we can see that there's an Aliases process by naming the tables waterfall (w) and tour (t). This helps shortening the process. Then, JOIN...ON, these two table w and t are pulled together with the JOIN keyword – which is always followed by an ON clause, this specifies how the tables should be linked together.

How does this…JOIN works?

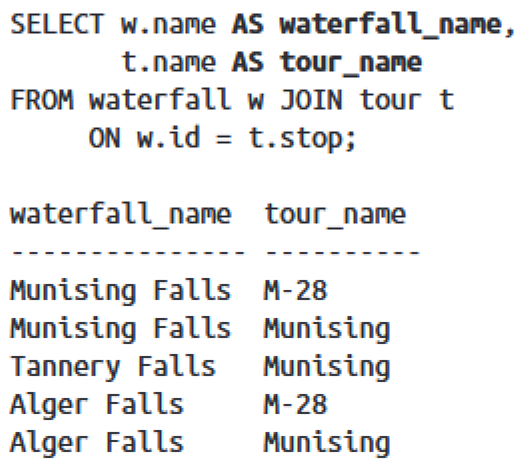
See the first column of the waterfall table, the id matches the stop column of the second table, the id must match the value of the stop in the tour table.

The result would be like this:



```
SELECT w.name, t.name
FROM waterfall w JOIN tour t
     ON w.id = t.stop;

name            name
--------------- ---------
Munising Falls  M-28
Munising Falls  Munising
Tannery Falls   Munising
...
```

But this seems kinda' ugly to be honest, so you'd like to alias the columns for **stethic** purposes (and to differentiate them):



```
SELECT w.name AS waterfall_name,
       t.name AS tour_name
FROM waterfall w JOIN tour t
     ON w.id = t.stop;

waterfall_name  tour_name
--------------- ----------
Munising Falls  M-28
Munising Falls  Munising
Tannery Falls   Munising
Alger Falls     M-28
Alger Falls     Munising
```

This type of JOIN clause it's the best in practice, but its own defaults to an INNER JOIN, meaning that only records that are in both tables are returned in the result.

For third, we have the WHERE clause.
WHERE clause functions as a filter, restricting query resuts to only rows of interest, as you will rarely want to display all rows from a table, but rather the rows that match a specific criteria.
This always follows the FROM clause and functions like this:
SELECT *
FROM annual_results ←example name
WHERE year_id = 1974

You can also use LIMIT clause or WHERE ROWNUM <=10, if you only want the first 10 rows.

SELECT *
FROM annual_results
LIMT 10

Fourth clause id the GROUP BY. The purpose of this clause is to collect rows into groups and summarize them within the groups, ultimately returning one row per group. This query counts the number of waterfalls along each of the tours:

```
SELECT   t.name AS tour_name,
         COUNT(*) AS num_waterfalls
FROM     waterfall w INNER JOIN tour t
         ON w.id = t.stop
GROUP BY t.name;

tour_name  num_waterfalls
```

```
---------- --------------
M-28                    6
Munising                6
US-2                    4
```

GROUP BY clause focuses on collecting rows, which is specified within the clause and summarizing the rows within groups, that is specified within the SELECT clause. We can see that in the following examples:

GROUP BY t.name

Collecting rows:

```
tour_name  waterfall_name
---------- ---------------
M-28       Munising Falls
M-28       Alger Falls
M-28       Scott Falls
M-28       Canyon Falls
M-28       Agate Falls
M-28       Bond Falls

Munising   Munising Falls
Munising   Tannery Falls
Munising   Alger Falls
Munising   Wagner Falls
Munising   Horseshoe Falls
Munising   Miners Falls
```

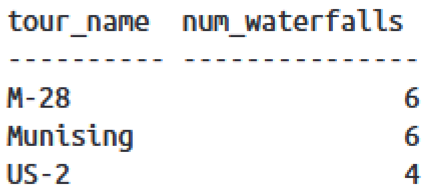
Summarizing rows:

```
tour_name  COUNT(*)
---------- ---------
M-28               6
M-28
M-28
M-28
M-28
M-28

Munising           6
Munising
Munising
Munising
Munising
Munising
```

Following, the fifth clause, HAVING places restrictions on the returned rows from the GROUP BY query, it is essential for the SQL “path-logic” for the HAVING to follow the previous query. It allows you to filter on the results after a GROUP BY has been applied.

This is a query that lists the number of waterfalls on each tour using a GROUP BY clause:

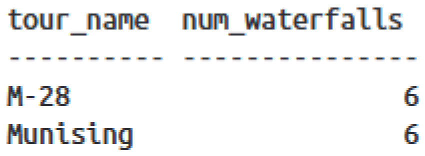
```
SELECT t.name AS tour_name,
COUNT(*) AS num_waterfalls
FROM waterfall w INNER JOIN tour t
ON w.id = t.stop
GROUP BY t.name;
```



```
tour_name   num_waterfalls
----------  ---------------
M-28                      6
Munising                  6
US-2                      4
```

But if you only wanted those who have six stops you would add then, a HAVING clause after the GROUP BY:

```
SELECT t.name AS tour_name,
COUNT(*) AS num_waterfalls
FROM waterfall w INNER JOIN tour t
ON w.id = t.stop
GROUP BY t.name
HAVING COUNT(*) = 6;
```

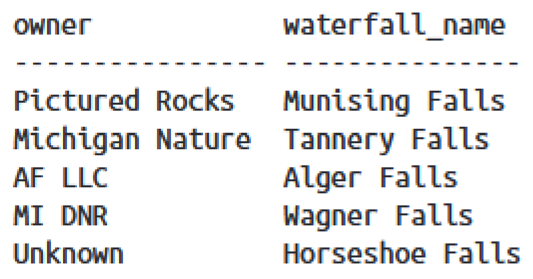


```
tour_name   num_waterfalls
----------  ---------------
M-28                      6
Munising                  6
```

To summarize all the points ...did we just use two filters for our query?

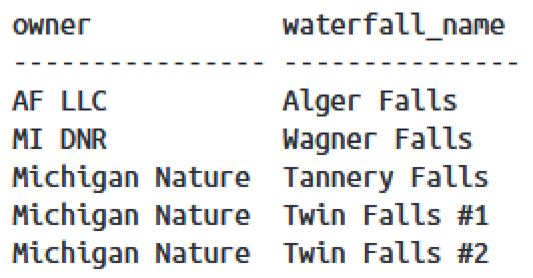
In essence, yes, previously on this explanation we used a WHERE, which is also a filter, but is one that only works on particular columns. If you wanted to filter on aggregations, write conditions within the HAVING clause. By aggregation we mean this: (*), never use it inside a WHERE.

Finally, we have the ORDER BY clause, which is used to specify how you want the results of a query to be sorted out.
Normally, writing this query you would have:
SELECT COALESCE(o.name, 'Unknown') AS owner,
w.name AS waterfall_name
FROM waterfall w
LEFT JOIN owner o ON w.owner_id = o.id;



```
owner            waterfall_name
---------------- ---------------
Pictured Rocks   Munising Falls
Michigan Nature  Tannery Falls
AF LLC           Alger Falls
MI DNR           Wagner Falls
Unknown          Horseshoe Falls
```

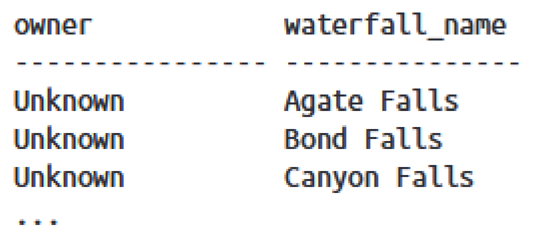
By applying an ORDER BY you can get them by name:



```
owner            waterfall_name
---------------- ---------------
AF LLC           Alger Falls
MI DNR           Wagner Falls
Michigan Nature  Tannery Falls
Michigan Nature  Twin Falls #1
Michigan Nature  Twin Falls #2
```

The default sort is in ascending order, from A to Z and number will go from lowest to highest, but you can apply the keywords ASCENDING and DESCENDING (abreviated ASC and DESC) depending on your preference.

SELECT COALESCE(o.name, 'Unknown') AS owner,
w.name AS waterfall_name
...
ORDER BY owner DESC, waterfall_name ASC;



```
owner            waterfall_name
---------------- ---------------
Unknown          Agate Falls
Unknown          Bond Falls
Unknown          Canyon Falls
...
```

The COALESCE is a function that replaces all NULL values ina column with one single value and it’s written like this:

ELECT COALESCE(**o.name, 'Unknown'**) AS owner,

The Order of execution is not the same as it has to be coded. It's useful to know this at the moment you're trying to reduce the amount of data processing and truest me, it's very useful when debugging and writing efficient queries.

First goes the FROM/JOINS, in this cases tables that are "joined" are processed here, values are matched according by the JOIN and On clauses, those search the data source. Following, the WHERE filters records, also using KEYWORDS like AND – OR, AND uses two conditions which are TRUE.

...WHERE condition_1 AND condition_2

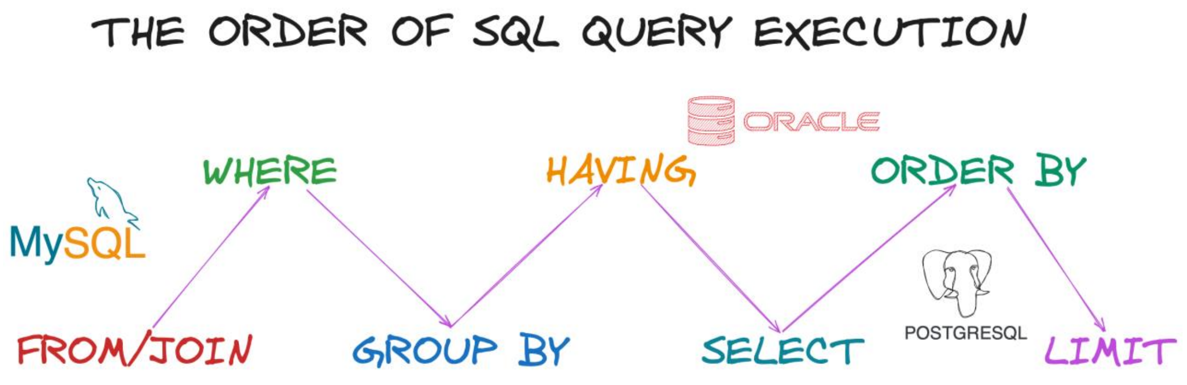
In the OR clause, one of the conditions given is true.

...WHERE NAME = 'condition_1' or NAME = 'condition_2'

Group by goes third, assempling groups with the resulted data, here we can use the aggregate functions, (that will be seen in the next section).

HAVING needs to follow a GROUP BY, filtering information based on conditions, operating with the groups determined by the previous instruction. From here SQL will go to the SELECT clause, executing all the desired columns and expressions. Once we get the extracted information the ORDER BY will sort it in our prefered way (ASC/DESC), and finally if used, LIMIT or OFFSET will display the specified quantity of numbers.

So, we have first a process that determines the relation of the tables we need to use, verifying any similarities in queries or single tables, then filters are applied and packed in groups and continuing with aggregate functions if they are specified, after all this process, the data is executed according to the conditions stablished, following then, the visualization.



Until here we still have to check on some extra-clauses that are useful when filtering information (like OFFSET, LIMIT, AND, OR), and something very important is also the subqueries, but it was fundamental to give you the basics of how SQL works.

We'll be checking this briefly to pass on the aggregate functions, that form part of the internal structure of most clauses checked.

Predicates

Among the most significant parts you can code on SQL is a logical comparison that could result in one of these three values: TRUE, FALSE, UNKNOWN. Commonly known as conditional statements, this predicates can compare expressions using operators like (=), (<>), and (<=, =>), this can be useful for you in example when you are looking for a determined group of workers that earn more of an exact figure:

```
SELECT*

FROM employees_board

WHERE salary > 50000;
```

Another common predicates in SQL are those that match and indicat existence, you can use the keyword LIKE to search for patterns within a text:

```
SELECT *

FROM customers

WHERE name LIKE ‘A%’;
```

This last string (‘A%’) posses a percent sign, and it's useful when you are looking for any string starting with ‘A’.

You can play with the logic of this keyword, as you can also look for determined positions too, like writing (‘_b%’) that matches any string where the second character is ‘b’.

The existence predicate goes by the keyword EXISTs and is used when checking for a row in a subquery:

```
SELECT *

FROM dprtmnts
```

WHERE EXISTS (SELECT * FROM employees WHERE Employees.DepartmentID = Departments.DepartmentID);

And believe it or not, the NULL predicate, that stands for no value, is very useful when you ar checking NULL values in a table, this keyword can be combined with logical operators AND, OR, NOT to form complex conditions that precisely will filter data:

SELECT *

FROM Employees WHERE ManagerID IS NOT NULL;

A reminder for you is that too look for no value is necessary to write IS NULL and not just = NULL.

To simply mention it, when you are using TRUE and FALSE, the rows that have the condition TRUE will be returned and the others, (FALSE), will be excluded.

This predicates can be helpful when working with boolean data type, that are those which can only assume two posible values, TRUE or FALSE more like 0 and 1.

CONDITIONALS

Beginning with the logical operators AND – OR, they are used to modify conditions, that will result in true or false.

AND returns TRUE if both conditions given are TRUE, returns FLASE if either is FALSE, if not it will return NULL.

OR returns TRUE if either condition is TRUE, returns FLASE if both are FALSE, returns NULL otherwise. This operator requires hat at least one of the conditions given is true. In difference to AND that is a more exact-type of data operator and restricts the data returned.

Remember that these operators are used within the WHERE clause and you can use both at the same time:

SELECT * FROM customers

WHERE (country = 'Mexico' AND city = 'Querétaro') OR city = 'Monterrey';

This query will give you the clients that are in Mexico city and in Queretaro or any costumer that could be in Monterrey.

COMMENTS, QUOTES AND WHITE SPACES

These punctuation marks are useful when you are writing a code and are used more for a format than for an obligatory use.

A comment is a text that is ignored when the code is run, but if you are teaching or passing the code to some other area in your company it can be used to specify what does the code line does or means:

```
-- Calculations of Sales closed in 2023
```

Think about others or even your future self when revisiting a code!

Another way to put a comment is by mutiple lines

```
/* This code will be the rpogressive calculation of sales
By department*\
```

Quotes

There are two types of quotes you can use in SQL, the single and double quote:

```
SELEC “this column”
FROM my_table
WHERE name = ‘Santiago’;
```

Single quotes go for string values, and double quotes go for Identifiers “This column”, remembering the first subjects treated on this handbook remember that an identifier is how you named a column. Using double quotes is obligatory for SQL for its content to be interpreted as a clumn name.

Whitespace

SQL does not care about the number of spaces between terms, whether it’s one space or tab or a new line, SQL will execute the query from the first keyword all the way to the semicolon at the end of the statement.

Extra-Clauses

LIMIT and OFFSET

They are clauses used to control the amount of rows that the query returns and from which point begins the extraction of data.

Limit restricts the number of rows that the query returns:

```
SELECT column_movie
FROM movies
LIMIT (5);
```

In this case the query obtains the first 5 rows of the movies table.

OFFSET is more specific for the rows that the query needs to return, some can be skipped before the extraction of data. It can be combined with LIMIT for more detailed results:

SELECT *

FROM customers

LIMIT 10 OFFSET 20;

This query will skip the first 20 rows and return the following 10 names in the customer table.

Making a subquery

A subquery is a type of query nested inside another. Subqueries can be located within various caluses, including the SELECT clause.

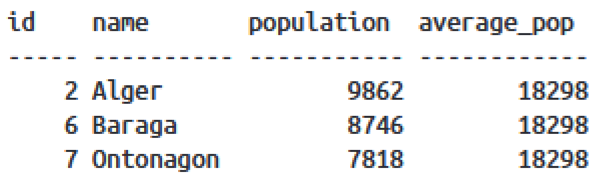
In the following example, in addition to the id, name and population we can see te average population of all counties, by including a subquery we are creating a new column in the results for the average population:

SELECT id, name, population,

(SELECT AVG(population) FROM county)

AS average_pop

FROM country



```
id    name        population  average_pop
----- ---------- ----------- ------------
    2 Alger             9862        18298
    6 Baraga            8746        18298
    7 Ontonagon         7818        18298
```

Few thing sto keep in mind is that a sbquery always needs to be surrounded by parentheses, and when writing a subquery within the SELECT clause, it is highly recommended that you specify a column alias, which in this case is average_pop, that way, the column has a simple name in the results.

A subquery can be located in any part of the code depending on the logic of what it returns but it’s poppular to find it within this clauses: SELECT, FROM, WHERE, HAVING. This makes easier the detailed data retrieval.

An alternative to writing a subquery is to write a common table expresion (CTE) using a WITH clause instead. The advantage of the with clause is that the subquery is named up front, which makes for cleaner code and also de ability to reference the subquery multiple times.

```
WITH o AS (SELECT * FROM owner

WHERE type = 'public')

SELECT w.name AS waterfall_name,

o.name AS owner_name

FROM o JOIN waterfall w

ON o.id = w.owner_id;
```

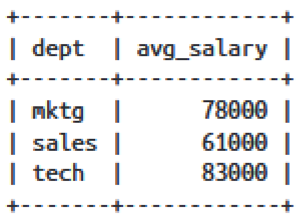
Here we first have the definition of the CTE, thi subquery selects all the columns from the owner table where type is public. Then in the main query the column retrieves the waterfall tables and puts an alias, o.name from the CTE is renamed as owner_name.

WHAT IS A CTE?

A common table expresion is a temporary result set, in other words it temporarily saves the output of a query for you to write other queries that reference it. Trhey are easy to spot for they have the WITH keyword, as said before, both CTE’s and subqueries allow you to write query, and then write another query that references the first query. Let’s put the case that your goal is to find the departement that has the largest average salary, this can be done in two steps:

– Write a **query** that returns the avergae salary for each department.

  ```
  SELECT dept, AVG(salary) AS avg_salary

  FROM employees

  GROUP BY dept;
  ```

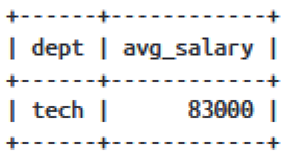


```
+-------+------------+
| dept  | avg_salary |
+-------+------------+
| mktg  |      78000 |
| sales |      61000 |
| tech  |      83000 |
+-------+------------+
```

– Step two is a CTE or subquery that finds the department with the largest average salary using the preceding query.

  ```
  -- CTE approach
  WITH avg_dept_salary AS (
  SELECT dept, AVG(salary) AS avg_salary
  ```

```
FROM employees
GROUP BY dept)
SELECT *
FROM avg_dept_salary
ORDER BY avg_salary DESC
LIMIT 1;
-- Equivalent subquery approach
SELECT *
FROM
(SELECT dept, AVG(salary) AS avg_salary
FROM employees
GROUP BY dept) avg_dept_salary
ORDER BY avg_salary DESC
LIMIT 1;
```



```
+------+------------+
| dept | avg_salary |
+------+------------+
| tech |      83000 |
+------+------------+
```

It's safe to say that we have covered all the topics we had left from our first section, so, the next part will contain the revision of the aggregate functions.

AGGREGATE FUNCTIONS

This type of functions perfoms a series of calculations on many rows of data and results in a single value, they are fundamental when you're summarizing and analizing data. You can locate them in the main SELECT clause and HAVING:

-
  ```
  SELECT COUNT(*) AS total_rows,
     AVG(age) AS average_age
     FROM my_table;
  ```

-
  ```
  SELECT region, MIN(age), MAX(age)
     FROM my_table
     GROUP BY region
     HAVING MIN(age) < 18;
  ```

Something that is really important to keep in mind when coding in SQL is that aggregate functions are not allowed in the WHERE clause, WHERE is a filtering clause, it isn't for aggergation, that's why we use it on HAVING, but you may wonder why, isn't HAVING a filter too? Yes, it is but HAVING filters data after aggregation, just like the code below:

```
SELECT city,
   AVG(temperature) AS average_highest_daily_temperature
FROM temperature_data
GROUP BY city
HAVING AVG(temperature) > 16;
```

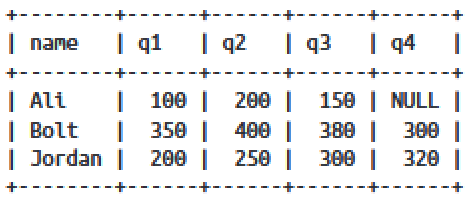
Now, after seeing the restrictions for its programming we have that the basic aggregate functions are:

1. COUNT(): Counts the number of values
2. SUM(): Calculates the sum of a column
3. AVG(): Calculates the average of a column
4. MIN(): Finds the minimum of a colum
5. MAX(): Finds the maximum of a column

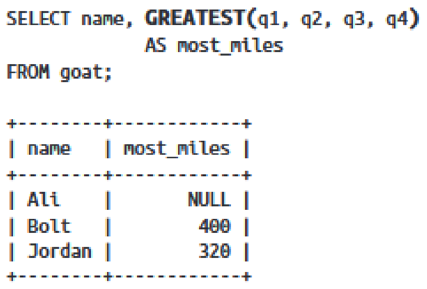
This functions apply calculations to all the non-values in a column, being the only exception is COUNT(*), which counts all rows including null values.

There are some other functions out there that can be useful too, like LEAST and GREATEST, this can be compared to MIN and MAX but they work different, min and max find values among a column, and this new two find the smallest and largest values within a row. Inputs can be numeric, string or datetime values. If one value is null, the function returns the null.

Here's the difference:



```
+--------+------+------+------+------+
| name   | q1   | q2   | q3   | q4   |
+--------+------+------+------+------+
| Ali    |  100 |  200 |  150 | NULL |
| Bolt   |  350 |  400 |  380 |  300 |
| Jordan |  200 |  250 |  300 |  320 |
+--------+------+------+------+------+
```



```
SELECT name, GREATEST(q1, q2, q3, q4)
             AS most_miles
FROM goat;

+--------+------------+
| name   | most_miles |
+--------+------------+
| Ali    |       NULL |
| Bolt   |        400 |
| Jordan |        320 |
+--------+------------+
```

There are some other numeric functions like ABS that returns the Absolute value of a number, the POWER function, that basically works like X raised to the power of Y and its written like this:

SELECT…

POWER (5,2);

RESULT: 25

Functions like SQRT, EXP, LOG, LN and LOG10 work like this:

Function	Description	Code	Result
SQRT	Square root	SELECT SQRT(25);	5
EXP	e(=2.71828) raised to the power of x	SELECT EXP(2);	7.389
LOG	Log of y base x	SELECT LOG(2,10); SELECT LOG(10,2);	3.322
LN	Natural log	SELECT LN(10); SELECT LOG(10);	2.303
LOG10	Log base 10	SELECT LOG10(100); SELECT LOG(10,100) FROM dual;	2

We can also use expressions in the SELECT and WHERE clauses to enable more simple calculations like arithmetic operations on numerical data:

+	**For addition**
-	For substraction
*****	For multiplication
/	For division
$	The modulus

For string values we also have functions like LENGTH, that finds the length of a string and it can be used in the SELECT clause or in the WHERE to filter the data:

SELECT LENGTH(name)

FROM my_table;

SELECT *

FROM my_table

WHERE LENGTH(name) < 10;

UPPER and LOWER functions change the case for s tring value:

SELECT UPPER('Hello World') AS uppercase_text;

Result: HELLO WORLD

Its purpose is that converts all chacters in a string to uppercase and follows the next format UPPER(string).

Same goes for LOWERCASE:

SELECT LOWER('Hello World') AS lowercase_text;

Results: hellow world

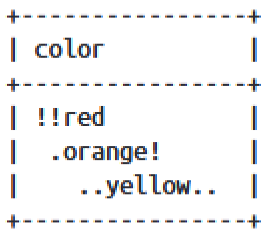
In some cases that are sensitive you need to apply the following line:

WHERE UPPER(username) = UPPER('john_doe');

To ensure that the comparison is case-sensitive, and matching ‘John_Doe’, ‘john_doe’ and ‘JOHN_DOE’

The following function works like its excel version, TRIM function removes unwanted characters around a string, first we have this:

SELECT * FROM my_table;

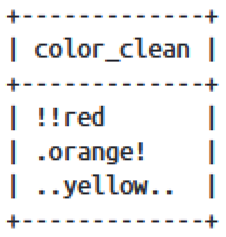


```
+----------------+
| color          |
+----------------+
| !!red          |
|  .orange!      |
|    ..yellow..  |
+----------------+
```

And to remove spaces you would have to write the following to correct the position of the text:

SELECT TRIM(color) AS color_clean

FROM my_table;



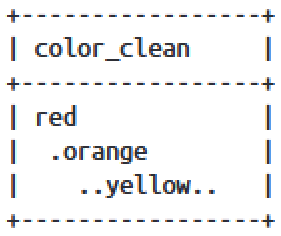
```
+------------+
| color_clean |
+------------+
| !!red       |
| .orange!    |
| ..yellow..  |
+------------+
```

That took out unnecessary spaces that are not pleaseant to the readability.

But this function can also trim characters inside the string:

SELECT TRIM('!' FROM color) AS color_clean

FROM my_table;



```
+----------------+
| color_clean     |
+----------------+
| red             |
|  .orange        |
|    ..yellow..   |
+----------------+
```

Until now we have seen the basics of how to formulate a statement, from functions to the basic clauses you need to know, but there are other commands you can do in SQL and are the most common to apply in practice, and I'm talking about the DQL or Data Query Language that supports a series of commands that are mainly used to retrieve data from databases. In some sort of way, those that we have seen, like SELECT, FROM, WHERE, GROUP BY, ORDERBY, LIMIT, etc. are primary DQL

COMMANDS

We'll now see what are the DML, Data Manipulation Language, these encompass commands that facilitate the retrieval, insertion, updating and deletion of data within a database, primarly DML include:

INSERT INTO is used to add new records (rows) to a table.

INSERT INTO employees (name, department, salary)

```
        VALUES ('John Doe', 'Marketing', 50000);
```

UPDATE is used to modify existing records in a table.

```
        UPDATE employees

        SET salary = 55000

        WHERE name = 'John Doe';
```

DELETE removes records from a table.

```
    DELETE FROM employees

    WHERE department = 'Marketing';
```

This statement deletes all records from the employees table where the department is marketing.

Needless to say these DML commands focus on manipulating data without altering the schema.

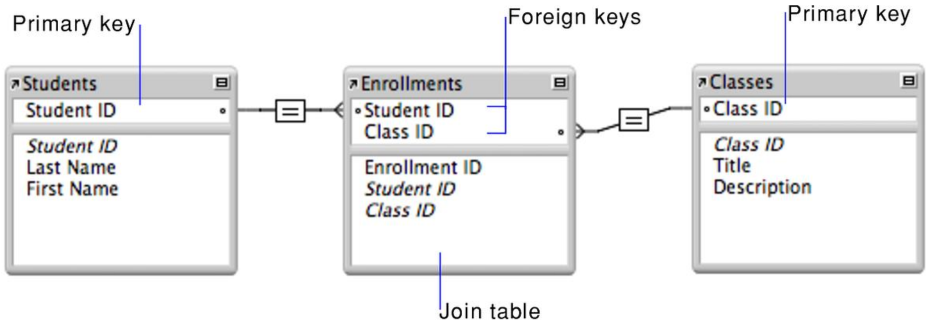
NORMALIZATION:

Our last topic will be the normalization of our databases, this is the process of organizing the data that's contained, it includes creating tables and establishing relationships between those tables acording to rules designed to protect the data and to make for efficent the retrieval exercise.

Normalization improves consistency, storage optimization, querying efficiency and accuracy and overall it maintains the state of the data to be adaptable without generating duplicates or redundancy, and its extremely useful when you use dependencies of tables.

We'll set a couple of definitiosn first for the whole understanding of this subject:

We have first what are the PRIMARY KEYS: that are fields that uniquely identifies the record of data in a table. And we also have the FOREIGN KEYS, that is a field that relates the primary key in another table. Much like the exercise we did when reviewing the JOIN, primery and foreign keys connect data from table to table.



Normalization is done through the normal forms, that are a series of steps to follow, each one dependent on the previous one and it’s not normalized until every table is in the formal form.

First Nomal Form (1NF)

The first form dedicates to elimante repeating groups, ensures the atomicity of data, with each column cell containing only a single value and each column having unique names.

Focus in this form when deleting duplicate fields or records, create separate tables for each set of relted data and very important, identify which are going to be the primary keys.

Second Normal Form (2NF)

Eliminates redundancy, meaning that eliminates partial dependancies by ensuring that non-key attributes depend only on the primary key, all this technically means that there should be a direct relationship between each column and the primary key, and not between other columns.

Third and Fourth Normal Forms, (3NF,4NF)

These are more strict versions of the previous ones, first the third form eliminates transitive partial dependancy, that means that there is no dependencies between non-key attributes themselves, everything goes towards the primary relation. And the fourth deals with muti.values dependencies, this occurs when one attribute can have multiple dependent attributes and this dependent attribtes are independent of the primary key.

CONCLUSION AND REFERENCES

Throughout the handbook we have reviewed some of the most basic ideas to begin in the world of SQL, there's still much out there you need to study if you want to become an expert in coding. This handbook was made more like a 'Back to Basics Guide' in case you forget some of the beginning concepts of SQL.

This is part of my final project for the subject of ***Selected Topics of Stadistics*** and none of this could have been possible without the lectures given by my professor that introduced me to this programming language.

Here are some of the references that I used along the investigation, materials and examples:

Segovia, J. (2017, 7 de noviembre). Diferencias entre DDL, DML y DCL. *Todo PostgreSQL*. https://www.todopostgresql.com/diferencias-entre-ddl-dml-y-dcl/

Zhao, A. (2021). *SQL pocket guide* (4ª ed.). O'Reilly Media.