Endless SQL possibilities

- You have learned that SQL query uses **SELECT, FROM and WHERE** to specify the data to be returned from the query. This reading provides more detailed information about formatting queries, using the **WHERE** conditions, selecting all columns in a table, adding comments, and using aliases. All of these make it easier for you to understand (and write) queries to put SQL in action. The last section of this reading provides and example of what a data analyst would do to pull employee data for a project.

Capitalization, indentation, and semicolons

You can write your SQL queries in all lowercase without having to worry about extra spaces between words. However, using capitalization and indention can help you read the information more easily. Keep your queries neat, and they will be easier to review or troubleshoot if you need to check them later on.

```
SELECT
    field1
FROM
    table
WHERE
    field1 = condition;
```

Notice that the SQL statement show above has a semicolon at the end. The semicolon is a *statement terminator* and is part of the **American National Standards (ANSI) SQL-92 standard**, where is a recommended common syntax for adoption by all SQL databases. However, not all SQL databases have adopted or enforce the semicolon, so it's possible you may come across some SQL statements that aren't terminated with a semicolon. If a statement works without a semicolon, it's fine.

WHERE conditions

In the query shown above, the **SELECT** clause identifies the column you want to pull data from by name, **field1**, and the **FROM** clause identifies the table where the column is located by name, **table**. Finally, the **WHERE** clause narrows you query so that the database returns only the data with an exact value match or the data matches a certain condition that you want to satisfy.

For Example, if you are looking for a specific customer with the last name *Chavez*, the WHERE clause would be;

```
WHERE field1 = 'Chavez'
```

However, if you are looking for *all* customers with a last name that begins with the letters "Ch", the **WHERE** clause would be;

```
WHERE field1 = 'Ch%'
```

You can conclude that the *LIKE* clause is very powerful because it allows you to tell the database to look a certain pattern! **The Percent(%) sign is used as a *Wild Card* to match one or more characters.** In the example above, both **Chavez** and **Chen** would be returned. Note that in some databases an *asterisk()** is used as the *wildcard* instead of the percent sign (%).

SELECT all columns

Can you use **SELECT** *?

In the example, if you replace **SELECT field1** with *SELECT **, you would be selecting all of the columns in the table instead of the *field1* column only. From a syntax point of view, it is a correct SQL statement, but you should use the asterisk(*) Sparingly and with caution. Depending on how many columns a table has, you could selecting a tremendous amount of data. Selecting too much data can cause a query to run slowly.

Comments

Some tables aren't designed with descriptive enough naming conventions. In the example **field1** was the column for a customers last name, but you wouldn't know it by the name. A better name would have been something such as **last_name**. In these cases, you can place comments alongside your SQL to help you remember what the name represents. Comments are text placed between certain characters, */* and */*, or after two dashes (—) as shown below.

```
SELECT
    field1 /*This is the last name column*/
FROM
    table –this is the customer data table
WHERE
    field1 LIKE 'Ch%';
```

Comments can also be added outside of a statement as well as within a statement. You can use this flexibility to provide and overall description of what you are going to do, step-by-step notes about how you achieve it, and why you set different parameters/conditions.

—This is an important query used later to join with the accounts table

```
SELECT
    rowkey, –key used to join with account_id
    Info.date, –date is in string format of YYY-MM-DD HH:MM:SS
    Info.code, –e.g. 'pub-###'
FROM Publishers
```

The more comfortable you get with SQL, the easier it will be to read and understand queries at a glance. Still, it never hurts to have comments in a query for others to understand your query if your query is shared, As your queries become more and more complex, this practice will save you a lot of time and energy to understand complex queries you wrote months or years ago.

Example of a query with comments

Here is an example of how comments should be written in BigQuery

```
— Pull basic information from the customer table

SELECT
customer_id, —main ID used to join with customer_address
first_name, —customer’s first name from Loyalty program
last_name —customer’s last name
FROM
customer_data.customer_name
```

In the above example, a comment has been added before the SQL statement to explain what the query does, Additionally, a comment has been added next to each of the column name to describe the column and its use. Two dashes (—) are generally supported. So it is best to use — and be consistent with it. You can use # (pound sign) in place of — (double dash) in the above query, but # is not recognized in all SQL versions;for example, MySQL doesn’t recognize #. You can also place comments between */and/* if the database you are using supports it.

As you develop your skills professionally, depending on the SQL database you use, you can pick the appropriate comment delimiting symbols you prefer and stick with those as a consistent style. As your queries become more and more complex, the practice of adding helpful comment will save you a lot of time and energy to understand queries that you may have written months or years ago.

Aliases

You can also make it easier on yourself by assigning a new name or **alias** to the column or table names to make them easier to work with (and avoid the need for comments). This is done with a SQL *AS* clause. In the example below, the alias **last_name** has been assigned to *field1* and the alias *customers* assigned to table These aliases are good for the duration of that query only. An alias doesn’t change the actual name of the column or table in the database.

Example of a query with aliases

```
field1 AS last_name
table AS customers

SELECT
    last_name
FROM
    customers
WHERE
    last_name LIKE ‘Ch%’
```

Putting SQL to work as a data analyst

Imagine you are a data analyst for a small business and you'r manager asks you for some employee data. You decide to write a query with SQL to get what you need from the database.

You want to pull all the columns; **empID, firstName, lastName, jobCode, and salary**. Because you know the database isn't that big, instead of entering each column name in the **SELECT** clause, you use SELECT *. This will select all of the columns from the employee table in the **FROM** clause.

```
SELECT
    *
FROM
    Employee
```

Now, you can get more specific about the data you want from the Employee table. If you want all the data about employees working in the **SFI**n job code, you can use a **WHERE** clause to filter out the data based on this additional requirement.

Here, you use:

```
SELECT
    *
FROM
    Employee
WHERE
    jobCode = 'SFI'
```

A portion of the resulting data returned from the SQL query might look like this;

empID	firstName	lastName	jobCode	salary
0002	Homer	Simpson	SFI	15000
0003	Marge	Simpson	SFI	30000
0034	Bart	Simpson	SFI	25000
0067	Lisa	Simpson	SFI	38000
0088	Ned	Flanders	SFI	42000
0076	Barney	Gumble	SFI	32000

Suppose you notice a large salary range for the **SFI** job code. You might like to flag all employees in all departments with lower salaries for your manager. Because interns are also included in the table and they have salaries less then 30,000$, you want to make sure your results give you only the full time employees with salaries that are $30,000 or less. In other words, you want to exclude interns with the **INT** job code who also earn less then $30,000 or less. The AND clause enables you to test for both conditions.

You create a SQL query similar to below, where <> means *does not equal*;

```
SELECT
    *
FROM
    Employee
WHERE
    jobCode <> 'INT'
    AND salary <= 30000;
```

The resulting data from the SQL query might look like the following (interns with the job code **INT** are't returned):

empID	firstName	lastName	jobCode	salary
0002	Homer	Simpson	SFI	15000
0003	Marge	Simpson	SFI	30000
0034	Bart	Simpson	SFI	25000
0108	Edna	Krabappel	TUL	18000
0099	Moe	Szyslak	ANA	28000

With quick access to this kind of data using SQL, you can provide your manager with tons of different insights about employee data, including whether employee salaries across the business are equitable. Fortunately, the query shows only an additional two employees might need a salary adjustment and you share the results with your manager.

Pulling the data, analyzing it and implementing a solution might ultimately help improve employee satisfaction and loyalty. That makes SQL a pretty powerful tool.

Resources to learn more

Nonsubscriber's may access these resources for free, but if a site limits the number of free articles per month and you have already reached your limit, bookmark the resource and comeback to it later.

W3Schools SQL Tutorial

- If you would like to explore a detailed tutorial of SQL, this is the perfect place to start. This tutorial includes interactive examples you can edit, test, and recreate. Use it as a reference or complete the whole tutorial to practice using SQL. Click the green **Start learning SQL now** button or the **Next** button to begin the tutorial.

SQL Cheat Sheet

- For more advanced learners, go through this article for standard SQL syntax used PostgreSQL. By the time you are finished, you will know a lot more about SQL and will be prepared to use it for business analysis and other tasks.