These best practices include guidelines for writing SQL queries, developing documentation, and examples that demonstrate these practices. This is a great resource to have handy when you are using SQL yourself; you can just go straight to the relevant section to review these practices. Think of it like a SQL field guide!

Capitalization and case sensitivity

With SQL, capitalization usually doesn't matter. You could write SELECT or select or SeLeCT. They all work! But if you use capitalization as part of a consistent style your queries will look more professional.

To write SQL queries like a pro, it is always a good idea to use all caps for clause starters (e.g., SELECT, FROM, WHERE, etc.). Functions should also be in all caps (e.g., SUM()). Column names should be all lowercase (refer to the section on snake_case later in this guide). Table names should be in CamelCase (refer to the section on CamelCase later in this guide). This helps keep your queries consistent and easier to read while not impacting the data that will be pulled when you run them. The only time that capitalization does matter is when it is inside quotes (more on quotes below).

Vendors of SQL databases may use slightly different variations of SQL. These variations are called **SQL dialects**. Some SQL dialects are case sensitive. BigQuery is one of them. Vertica is another. But most, like MySQL, PostgreSQL, and SQL Server, aren't case sensitive. This means if you searched for country_code = 'us', it will return all entries that have 'us', 'uS', 'Us', and 'US'. This isn't the case with BigQuery. BigQuery is case sensitive, so that same search would only return entries where the country_code is exactly 'us'. If the country_code is 'US', BigQuery wouldn't return those entries as part of your result.

Single or double quotes: '' or " "

For the most part, it also doesn't matter if you use single quotes ' ' or double quotes " " when referring to strings. For example, SELECT is a clause starter. If you put SELECT in quotes like 'SELECT' or "SELECT", then SQL will treat it as a text string. Your query will return an error because your query needs a SELECT clause.

But there are two situations where it does matter what kind of quotes you use:

1. When you want strings to be identifiable in *any* SQL dialect

2. When your string contains an apostrophe or quotation marks

Within each SQL dialect there are rules for what is accepted and what isn't. But a general rule across almost all SQL dialects is to use single quotes for strings. This helps get rid of a lot of confusion. So if we want to reference the country US in a WHERE clause (e.g., country_code = 'US'), then use single quotes around the string 'US'.

The second situation is when your string has quotes inside it. Suppose you have a column of favorite foods in a table called FavoriteFoods and the other column corresponds to each friend.

Friend	**Favorite_food**
Rachel DeSantos	Shepherd's pie
Sujin Lee	Tacos
Najil Okoro	Spanish paella

You might notice how Rachel's favorite food contains an apostrophe. If you were to use single quotes in a WHERE clause to find the friend who has this favorite food, it would look like this:

```
SELECT
      Friend
FROM
      FavoriteFoods
WHERE
      Favorite_food = 'Shepherd's pie'
```

This won't work. If you run this query, you will get an error in return. This is because SQL recognizes a text string as something that starts with a quote 'and ends with another quote '. So in the bad query above, SQL thinks that the Favorite_food you are looking for is 'Shepherd'. Just 'Shepherd' because the apostrophe in Shepherd's ends the string.

Generally speaking, this should be the only time you would use double quotes instead of single quotes. So your query would look like this instead:

```
SELECT
      Friend
FROM
      FavoriteFoods
WHERE
      Favorite_food = "Shepherd's pie"
```

SQL understands text strings as either starting with a single quote ' or double quote". Since this string starts with double quotes, SQL will expect another double quote to signal the end of the string. This keeps the apostrophe safe, so it will return "Shepherd's pie" and not 'Shepherd'.

Comments as reminders

As you get more comfortable with SQL, you will be able to read and understand queries at a glance. But it never hurts to have comments in the query to remind yourself of what you are trying to do. And if you share your query, it also helps others understand it.

For example:

```
--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 YYYY-MM-DD HH:MM:SS
      Info.code --e.g., 'pub-###'
FROM
      Publishers
```

You can use # in place of the two dashes, --, in the above query but keep in mind that # isn't recognized in all SQL dialects (MySQL doesn't recognize #). So it is best to use -- and be consistent with it. When you add a comment to a query using --, the database query engine will ignore everything in the same line after --. It will continue to process the query starting on the next line.

Snake_case names for columns

It is important to always make sure that the output of your query has easy-to-understand names. If you create a new column (say from a calculation or from concatenating new fields), the new column will receive a generic default name (e.g., f0). For example:

```
SELECT
      SUM(tickets),
      COUNT(tickets),
      SUM(tickets) AS total_tickets,
      COUNT(tickets) AS number_of_purchases
FROM
      purchases
```

The following table features the results of this query: f0: 8 f1: 4 total_tickets: 8 Number_of_purchases: 4

Results are:

f0	f1	total_tickets	number_of_purchases
8	4	8	4

The first two columns are named f0 and f1 because they weren't named in the above query. SQL defaults to f0, f1, f2, f3, and so on. We named the last two columns total_tickets and number_of_purchases so these column names show up in the query results. This is why it is always good to give your columns useful names, especially when using functions. After running your query, you want to be able to quickly understand your results, like the last two columns we described in the example.

On top of that, you might notice how the column names have an underscore between the words. Names should never have spaces in them. If 'total_tickets' had a space and looked like 'total tickets' then SQL would rename SUM(tickets) as just 'total'. Because of the space, SQL will use 'total' as the name and won't understand what you mean by 'tickets'. So, spaces are bad in SQL names. Never use spaces.

The best practice is to use snake_case. This means that 'total tickets', which has a space between the two words, should be written as 'total_tickets' with an underscore instead of a space.

CamelCase names for tables

You can also use CamelCase capitalization when naming your table. CamelCase capitalization means that you capitalize the start of each word, like a two-humped (Bactrian) camel. So the table TicketsByOccasion uses CamelCase capitalization. Please note that the capitalization of the first word in CamelCase is *optional;* camelCase is also used. Some people differentiate between the two styles by calling CamelCase, PascalCase, and reserving camelCase for when the first word isn't capitalized, like a one-humped (Dromedary) camel; for example, ticketsByOccasion.

At the end of the day, CamelCase is a style choice. There are other ways you can name your tables, including:

- All lower or upper case, like ticketsbyoccasion or TICKETSBYOCCASION
- With snake_case, like tickets_by_occasion

Keep in mind, the option with all lowercase or uppercase letters can make it difficult to read your table name, so it isn't recommended for professional use.

The second option, snake_case, is technically okay. With words separated by underscores, your table name is easy to read, but it can get very long because you are adding the underscores. It also takes more time to write. If you use this table a lot, it can become a chore.

In summary, it is up to you to use snake_case or CamelCase when creating table names. Just make sure your table name is easy to read and consistent. Also be sure to find out if your company has a preferred way of naming their tables. If they do, always go with their naming convention for consistency.

Indentation

As a general rule, you want to keep the length of each line in a query <= 100 characters. This makes your queries easy to read. For example, check out this query with a line with >100 characters:

```
SELECT
CASE WHEN genre = 'horror' THEN 'Will not watch' WHEN genre = 'documentary'
THEN 'Will watch alone' ELSE 'Watch with others' END AS
Watch_category, COUNT(movie_title) AS number_of_movies
  FROM
       MovieTheater
  GROUP BY
       1
```

SELECT CASE WHEN genre = 'horror' THEN 'Will not watch' WHEN genre = 'documentary' THEN 'Will watch alone' ELSE 'Watch with others' END AS Watch_category, COUNT(

This query is hard to read and just as hard to troubleshoot or edit. Now, here is a query where we stick to the <= 100 character rule:

```
SELECT
      CASE
            WHEN genre = 'horror' THEN 'Will not watch'
            WHEN genre = 'documentary' THEN 'Will watch alone'
            ELSE 'Watch with others'
            END AS watch_category, COUNT(movie_title) AS number_of_movies
FROM
      MovieTheater
GROUP BY
      1
```

Now it is much easier to understand what you are trying to do in the SELECT clause. Sure, both queries will run without a problem because indentation doesn't matter in SQL. But proper indentation is still important to keep lines short. And it will be valued by anyone reading your query, including yourself!

Multi-line comments

If you make comments that take up multiple lines, you can use -- for each line. Or, if you have more than two lines of comments, it might be cleaner and easier is to use /* to start the comment and */ to close the comment. For example, you can use the -- method like below:

```
-- Date: September 15, 2020
-- Analyst: Jazmin Cisneros
-- Goal: Count the number of rows in the table
SELECT
      COUNT(*) number of rows -- the * stands for all so count all
FROM
      table
```

-- Date: September 15, 2020 -- Analyst: Jazmin Cisneros -- Goal: Count the number of rows in the table SELECT COUNT(*) number of rows -- the * stands for all so count all FROM table

Or, you can use the /* */ method like below:

```
/*
Date: September 15, 2020
Analyst: Jazmin Cisneros
Goal: Count the number of rows in the table
*/
SELECT
      COUNT(*) AS number_of_rows -- the * stands for all so count all
FROM
      table
```

/* Date: September 15, 2020 Analyst: Jazmin Cisneros Goal: Count the number of rows in the table */ SELECT COUNT(*) AS number_of_rows -- the * stands for all so count all FROM
      table

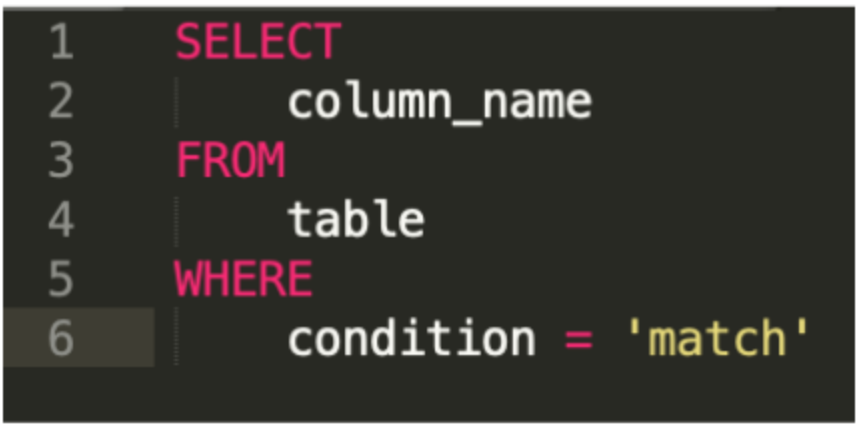
In SQL, it doesn't matter which method you use. SQL ignores comments regardless of what you use: #, --, or /* and */. So it is up to you and your personal preference. The /* and */ method for multi-line comments usually looks cleaner and helps separate the comments from the query. But there isn't one right or wrong method.

SQL text editors

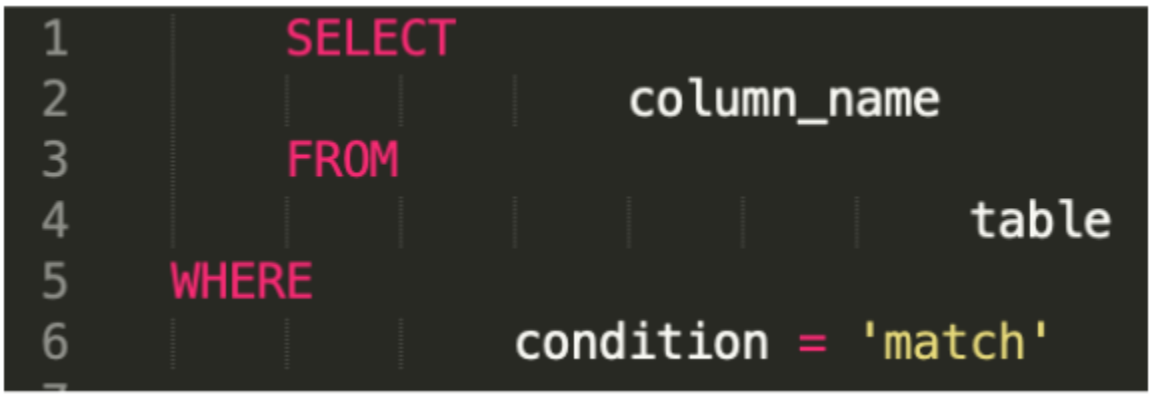
When you join a company, you can expect each company to use their own SQL platform and SQL dialect. The SQL platform they use (e.g., BigQuery, MySQL, or SQL Server) is where you will write and run your SQL queries. But keep in mind that not all SQL platforms provide native script editors to write SQL code. SQL text editors give you an interface where you can write your SQL queries in an easier and color-coded way. In fact, all of the code we have been working with so far was written with an SQL text editor!

Examples with Sublime Text

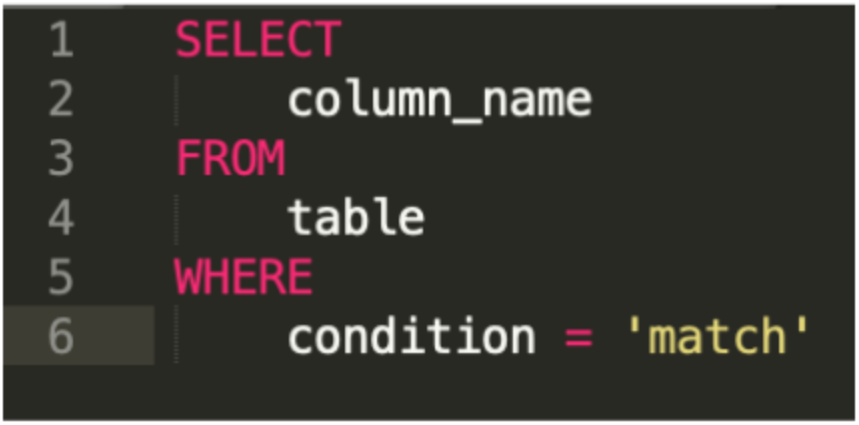
If your SQL platform doesn't have color coding, you might want to think about using a text editor like Sublime Text or Atom. This section shows how SQL is displayed in Sublime Text. Here is a query in Sublime Text:



With Sublime Text, you can also do advanced editing like deleting indents across multiple lines at the same time. For example, suppose your query somehow had indents in the wrong places and looked like this:



This is really hard to read, so you will want to eliminate those indents and start over. In a regular SQL platform, you would have to go into each line and press BACKSPACE to delete each indent per line. But in Sublime, you can get rid of all the indents at the same time by selecting all lines and pressing Command (or CTRL in Windows) + [. This eliminates indents from every line. Then you can select the lines that you want to indent (i.e., lines 2, 4, and 6) by pressing the Command key (or the CTRL key in Windows) and selecting those lines. Then while still holding down the Command key (or the CTRL key in Windows), press] to indent lines 2, 4, and 6 at the same time. This will clean up your query and make it look like this instead:



Sublime Text also supports regular expressions. **Regular expressions** (or **regex**) can be used to search for and replace string patterns in queries. We won't cover regular expressions here, but you might want to learn more about them on your own because they are a very powerful tool.

You can begin with these resources:

- Search and replace in Sublime Text
- Regex tutorial (if you don't know what regular expressions are)
- Regex cheat sheet