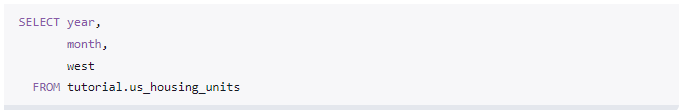
Basic SQL

SELECT indicates which columns you'd like to view.

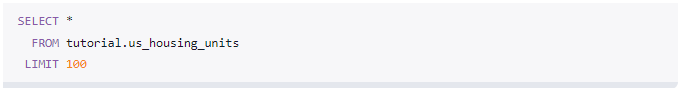
FROM identifies the table that they live in. You can only run one SELECT statement at a time.

SELECT \* to select every column in a table.

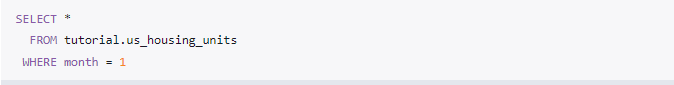
Whenever you select multiple columns, they must be separated by commas, but you should not include a comma after the last column name.



LIMIT restricts how many rows the SQL query returns.



WHERE to sort data.

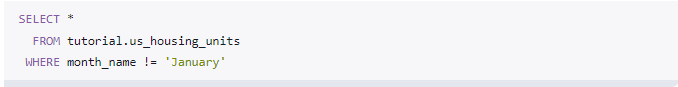


The clauses always need to be in this order: SELECT, FROM, WHERE.

Comparison Operators on numerical data:

= Equal to  
<> or != Not equal to  
> Grater then  
< Less then  
>= Grater then or equal to  
<= Less then or equal to

Comparison operators on non-numerical data: all of the above operators work on non-numerical data as well.  
Using an operator with values that are non-numeric, you need to put the value in single quotes: 'value'.



Note: SQL uses single quotes to reference column values.

Arithmetic in SQL

You can perform arithmetic in SQL using the same operators you would in Excel:

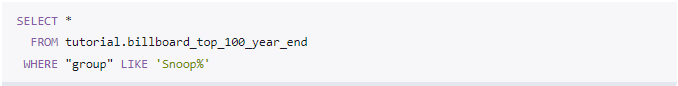
+, -, \*, /. However, in SQL you can only perform arithmetic across columns on values in a given row. To clarify, you can only add values in multiple columns from the same row together using +—if you want to add values across multiple rows, you'll need to use aggregate functions.

As in Excel, you can use parentheses to manage the order of operations.

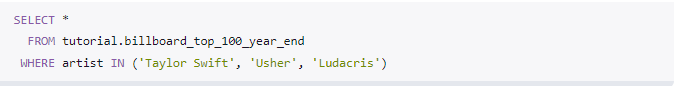
The columns that contain the arithmetic functions are called "derived columns" because they are generated by modifying the information that exists in the underlying data.

SQL Logical operators

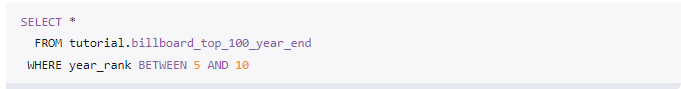
LIKE allows you to match similar values, instead of exact values (case-sensitive). ILIKE (not case-sensitive).



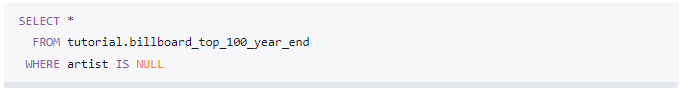
IN allows you to specify a list of values you'd like to include.



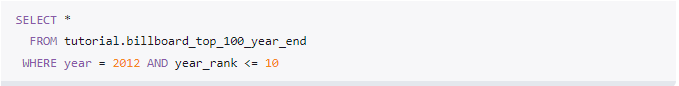
BETWEEN allows you to select only rows within a certain range.



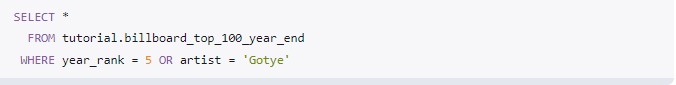
IS NULL allows you to select rows that contain no data in a given column.



AND allows you to select only rows that satisfy two conditions.

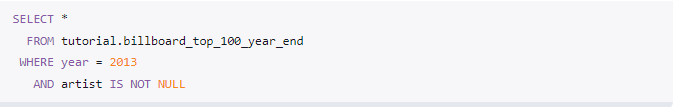


OR allows you to select rows that satisfy either of two conditions.



NOT allows you to select rows that do not match a certain condition.

NOT is also frequently used to identify non-null rows, but the syntax is somewhat special—you need to include IS beforehand. Here's how that looks:



The % used above represents any character or set of characters. In this case, % is referred to as a "wildcard."

ORDER BY Sorting data

This is referred to as ascending order from A to Z and it's SQL's default. If you order a numerical 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 If you'd like your results in the opposite order (referred to as descending order), you need to add the DESC operator:

A picture containing graphical user interface

Description automatically generated

Intermediate SQL

Aggregate functions in SQL

COUNT counts how many rows are in a particular column.

Counting all rows. Counting individual columns (is not null). Counting non-numerical columns



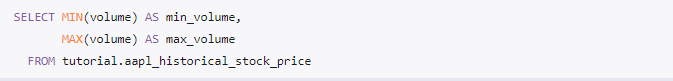
SUM adds together all the values in a particular column. A

Graphical user interface, text, application

Description automatically generated

Aggregators only aggregate vertically. If you want to perform a calculation across rows, you would do this with simple arithmetic.

MIN and MAX return the lowest and highest values in a particular column, respectively. Also can be used on non-numerical columns.



AVG calculates the average of a group of selected values. It can only be used on numerical columns and it ignores nulls completely. There are some cases in which you'll want to treat null values as 0. For these cases, you'll want to write a statement that changes the nulls to 0.

Graphical user interface, application

Description automatically generated

GROUP BY SQL aggregate function like COUNT, AVG, and SUM have something in common: they all aggregate across the entire table. But what if you want to aggregate only part of a table? For example, you might want to count the number of entries for each year. In situations like this, you'd need to use the GROUP BY clause. GROUP BY allows you to separate data into groups, which can be aggregated independently of one another.

Graphical user interface, text, application

Description automatically generated

HAVING However, you'll often encounter datasets where GROUP BY isn't enough to get what you're looking for. Let's say that it's not enough just to know aggregated stats by month. After all, there are a lot of months in this dataset. Instead, you might want to find every month during which AAPL stock worked its way over $400/share.

Text

Description automatically generated with low confidence

Query clause order

As mentioned in prior lessons, the order in which you write the clauses is important. Here's the order for everything you've learned so far: 1 SELECT, 2 FROM, 3 WHERE, 4 GROUP BY, 5 HAVING, 6 ORDER BY

CASE The CASE statement is SQL's way of handling if/then logic. The CASE statement is followed by at least one pair of WHEN and THEN statements—SQL's equivalent of IF/THEN in Excel. Because of this pairing, you might be tempted to call this SQL CASE WHEN, but CASE is the accepted term.

Every CASE statement must end with the END statement. The ELSE statement is optional, and provides a way to capture values not specified in the WHEN/THEN statements. CASE is easiest to understand in the context of an example:

Graphical user interface, application

Description automatically generated

DISTINCT You'll occasionally want to look at only the unique values in a particular column. To select unique values from the month and year column in the Apple stock prices dataset, you'd use the following query: Graphical user interface, application

Description automatically generated

SQL Joins for working with data from multiple tables at once. Graphical user interface, text, application

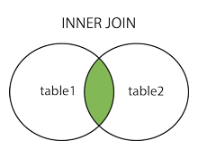
Description automatically generated

Aliases in SQL. When performing joins, it's easiest to give your table names aliases. benn.college\_football\_players is pretty long and annoying to type—players is much easier. You can give a table an alias by adding a space after the table name and typing the intended name of the alias. As with column names, best practice here is to use all lowercase letters and underscores instead of spaces.

Once you've given a table an alias, you can refer to columns in that table in the SELECT clause using the aliases name. For example, the first column selected in the above query is teams.conference. Because of the alias, this is equivalent to benn.college\_football\_teams.conference: we're selecting the conference column in the college\_football\_teams table in benn's schema.

JOIN and ON. ON indicates how the two tables (the one after the FROM and the one after the JOIN) relate to each other. You can see in the example above that both tables contain fields called school\_name. Sometimes relational fields are slightly less obvious. For example, you might have a table called schools with a field called id, which could be joined against school\_id in any other table. These relationships are sometimes called "mappings." teams.school\_name and players.school\_name, the two columns that map to one another, are referred to as "foreign keys" or "join keys." Their mapping is written as a conditional statement: 

INNER JOIN We'll start with inner joins, which can be written as either JOIN benn.college\_football\_teams teams or INNER JOIN benn.college\_football\_teams teams. Inner joins eliminate rows from both tables that do not satisfy the join condition set forth in the ON statement. In mathematical terms, an inner join is the intersection of the two tables.



Therefore, if a player goes to a school that isn't in the teams table, that player won't be included in the result from an inner join. Similarly, if there are schools in the teams table that don't match to any schools in the players table, those rows won't be included in the results either.

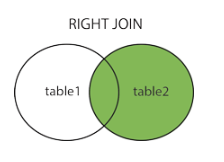
Outer JOIN Outer joins are joins that return matched values and unmatched values from either or both tables. There are a few types of outer joins:

LEFT JOIN returns only unmatched rows from the left table, as well as matched rows in both tables.

Diagram, venn diagram

Description automatically generated

RIGHT JOIN returns only unmatched rows from the right table , as well as matched rows in both tables.



FULL OUTER JOIN returns unmatched rows from both tables,as well as matched rows in both tables.

Note: LEFT JOIN is also refered to as OUTER LEFT JOIN. RIGHT JOIN is also refered to as OUTER RIGHT JOIN. FULL OUTER JOIN is also refered to as OUTER JOIN.

As you work through the following lessons about outer joins, it might be helpful to refer to [this JOIN visualization](https://joins.spathon.com/)

UNION allows you to stack one dataset on top of the other. Put differently, UNION allows you to write two separate SELECT statements, and to have the results of one statement display in the same table as the results from the other statement.

SQL has strict rules for appending data:   
1. Both tables must have the same number of columns  
2. The columns must have the same data types in the same order as the first table

Advanced SQL

Data type [complete list](https://www.w3schools.com/sql/sql_datatypes.asp).

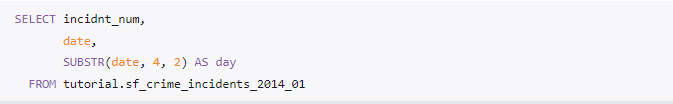
Graphical user interface, text, application, email

Description automatically generated

INTERVAL data type—a series of integers that represent a period of time.

CAST / CONVERT to change the data type to a numeric one that will allow you to perform the sum.   
You can actually achieve this with two different type of syntax. For example, CAST(column\_name AS integer) and column\_name::integer produce the same result.

LEFT, RIGHT like in Excel. You can use LEFT, RIGHT to pull a certain number of characters from the left, right side of a string and present them as a separate string. The syntax is LEFT(string, number of characters).

SUBSTR like middle in Excel. The syntax is SUBSTR(\*string\*, \*starting character position\*, \*# of characters\*): 

LENGTH like in Excel, returns the length of a string.

TRIM The TRIM function is used to remove characters from the beginning and end of a string. Here's an example: Graphical user interface, application, Teams

Description automatically generated  
The TRIM function takes 3 arguments. First, you have to specify whether you want to remove characters from the beginning ('leading'), the end ('trailing'), or both ('both', as used above). Next you must specify all characters to be trimmed. Any characters included in the single quotes will be removed from both beginning, end, or both sides of the string. Finally, you must specify the text you want to trim using FROM.

POSSITION Показывает на каком по счету месте (где) расположен нужный символ. A picture containing text

Description automatically generated

STRPOS You can also use the STRPOS function to achieve the same results—just replace IN with a comma and switch the order of the string and substring: Graphical user interface, application

Description automatically generated

CONCAT like in Excel. To combine strings from several columns together. Simply order the values you want to concatenate and separate them with commas. If you want to hard-code values, enclose them in single quotes. Here's an example:Text

Description automatically generated with low confidence

II Alternatively, you can use two pipe characters (||) to perform the same concatenation: Text

Description automatically generated with medium confidence

UPPER, LOWER Sometimes, you just don't want your data to look like it's screaming at you. You can use LOWER to force every character in a string to become lower-case. Similarly, you can use UPPER to make all the letters appear in upper-case: Graphical user interface, application, Teams

Description automatically generated

EXTRACT You've learned how to construct a date field, but what if you want to deconstruct one? You can use EXTRACT to pull the pieces apart one-by-one: Graphical user interface, text

Description automatically generated

DATE\_TRUNK You can also round dates to the nearest unit of measurement. This is particularly useful if you don't care about an individual date, but do care about the week (or month, or quarter) that it occurred in. The DATE\_TRUNC function rounds a date to whatever precision you specify. The value displayed is the first value in that period. So when you DATE\_TRUNC by year, any value in that year will be listed as January 1st of that year. A screenshot of a computer program

Description automatically generated with medium confidence

COALESCE Occasionally, you will end up with a dataset that has some nulls that you'd prefer to contain actual values. This happens frequently in numerical data (displaying nulls as 0 is often preferable), and when performing outer joins that result in some unmatched rows. In cases like this, you can use COALESCE to replace the null values: A picture containing text, font, white, algebra

Description automatically generated

Writing Subqueries in SQL.

Subquery basics .

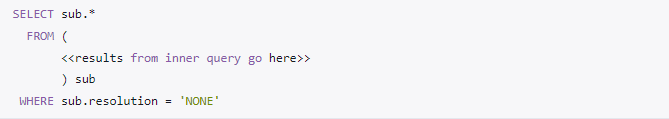
Subqueries (also known as inner queries or nested queries) are a tool for performing operations In multiple steps. For example, if you wanted to take the sums of several columns, then average all of those values, you'd need to do each aggregation in a distinct step.

Subqueries can be used in several places within a query, but it's easiest to start with the FROM statement. Here's an example of a basic subquery: A screenshot of a computer

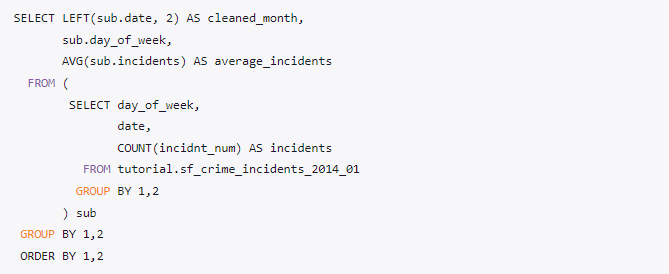
Description automatically generated with medium confidence

Let's break down what happens when you run the above query:   
First, the database runs the "inner query"—the part between the parentheses. A close-up of a computer screen

Description automatically generated with low confidence

If you were to run this on its own, it would produce a result set like any other query. It might sound like a no-brainer, but it's important: your inner query must actually run on its own, as the database will treat it as an independent query. Once the inner query runs, the outer query will run using the results from the inner query as its underlying table: 

Using subqueries to aggregate in multiple stages.



Subqueries in conditional logic.

You can use subqueries in conditional logic (in conjunction with WHERE, JOIN/ON, or CASE). The following query returns all of the entries from the earliest date in the dataset (theoretically—the poor formatting of the date column actually makes it return the value that sorts first alphabetically): A screenshot of a computer

Description automatically generated with low confidence

The above query works because the result of the subquery is only one cell. Most conditional logic will work with subqueries containing one-cell results. However, IN is the only type of conditional logic that will work when the inner query contains multiple results: A screenshot of a computer

Description automatically generated with low confidence

Note that you should not include an alias when you write a subquery in a conditional statement. This is because the subquery is treated as an individual value (or set of values in the IN case) rather than as a table.

Joining subqueries. You may remember that you can filter queries in joins. It's fairly common to join a subquery that hits the same table as the outer query rather than filtering in the WHERE clause. The following query produces the same results as the previous example: A screenshot of a computer

Description automatically generated with medium confidence

This can be particularly useful when combined with aggregations. When you join, the requirements for your subquery output aren't as stringent as when you use the WHERE clause. For example, your inner query can output multiple results. The following query ranks all of the results according to how many incidents were reported in a given day. It does this by aggregating the total number of incidents each day in the inner query, then using those values to sort the outer query: A screenshot of a computer

Description automatically generated with medium confidence

Subqueries and UNIONs.

It's certainly not uncommon for a dataset to come split into several parts, especially if the data passed through Excel at any point (Excel can only handle ~1M rows per spreadsheet). The two tables used above can be thought of as different parts of the same dataset—what you'd almost certainly like to do is perform operations on the entire combined dataset rather than on the individual parts. You can do this by using a subquery: A screenshot of a computer code

Description automatically generated with low confidence

SQL Window Functions.

PostgreSQL's documentation does an excellent job of introducing the concept of Window Functions:

„ A window function performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result. “

The most practical example of this is a running total: A picture containing text, font, line, white

Description automatically generated

You can see that the above query creates an aggregation (running\_total) without using GROUP BY. Let's break down the syntax and see how it works.

Basic windowing syntax

The first part of the above aggregation, SUM(duration\_seconds), looks a lot like any other aggregation. Adding OVER designates it as a window function. You could read the above aggregation as "take the sum of duration\_seconds over the entire result set, in order by start\_time."

PARTITION BY If you'd like to narrow the window from the entire dataset to individual groups within the dataset. A screenshot of a computer code

Description automatically generated with low confidence

The above query groups and orders the query by start\_terminal. Within each value of start\_terminal, it is ordered by start\_time, and the running total sums across the current row and all previous rows of duration\_seconds. Scroll down until the start\_terminal value changes and you will notice that running\_total starts over. That's what happens when you group using PARTITION BY. In case you're still stumped by ORDER BY, it simply orders by the designated column(s) the same way the ORDER BY clause would, except that it treats every partition as separate. It also creates the running total—without ORDER BY, each value will simply be a sum of all the duration\_seconds values in its respective start\_terminal. Try running the above query without ORDER BY to get an idea: A screen shot of a computer code

Description automatically generated with low confidence

The ORDER and PARTITION define what is referred to as the "window"—the ordered subset of data over which calculations are made.

Note: You can't use window functions and standard aggregations in the same query. More specifically, you can't include window functions in a GROUP BY clause.

When using window functions, you can apply the same aggregates that you would under normal circumstances—SUM, COUNT, and AVG. The easiest way to understand these is to re-run the previous example with some additional functions.

ROW\_NUMBER() does just what it sounds like—displays the number of a given row. It starts are 1 and numbers the rows according to the ORDER BY part of the window statement. A picture containing text, font, screenshot

Description automatically generated

RANK() is slightly different from ROW\_NUMBER(). If you order by start\_time, for example, it might be the case that some terminals have rides with two identical start times. In this case, they are given the same rank, whereas ROW\_NUMBER() gives them different numbers. In the following query, you notice the 4th and 5th observations for start\_terminal 31000—they are both given a rank of 4, and the following result receives a rank of 6: A picture containing text, screenshot, font

Description automatically generated

DENSE\_RANK() You can also use DENSE\_RANK() instead of RANK() depending on your application. Imagine a situation in which three entries have the same value. Using either command, they will all get the same rank. For the sake of this example, let's say it's "2." Here's how the two commands would evaluate the next results differently:   
RANK() would give the identical rows a rank of 2, then skip ranks 3 and 4, so the next result would be 5   
DENSE\_RANK() would still give all the identical rows a rank of 2, but the following row would be 3—no ranks would be skipped.

NTILE(\*# of buckets\*) You can use window functions to identify what percentile (or quartile, or any other subdivision) a given row falls into. The syntax is NTILE(\*# of buckets\*). In this case, ORDER BY determines which column to use to determine the quartiles (or whatever number of 'tiles you specify). For example: A screenshot of a computer program

Description automatically generated with low confidence

LAG and LEAD It can often be useful to compare rows to preceding or following rows, especially if you've got the data in an order that makes sense. You can use LAG or LEAD to create columns that pull values from other rows—all you need to do is enter which column to pull from and how many rows away you'd like to do the pull. LAG pulls from previous rows and LEAD pulls from following rows: A screenshot of a computer code

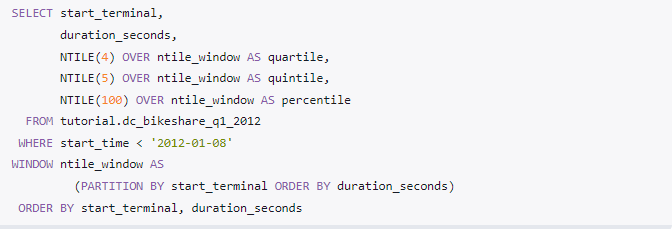
Description automatically generated with low confidence

This is especially useful if you want to calculate differences between rows: A screenshot of a computer code

Description automatically generated with low confidence

Defining a window alias. If you're planning to write several window functions in to the same query, using the same window, you can create an alias. Take the NTILE example above: A screenshot of a computer program

Description automatically generated with low confidence

This can be rewritten as: 

EXPLAIN You can add EXPLAIN at the beginning of any (working) query to get a sense of how long it will take. It's not perfectly accurate, but it's a useful tool. Try running this: A picture containing text, font, screenshot

Description automatically generated

You'll get this output. It's called the Query Plan, and it shows the order in which your query will be executed. The entry at the bottom of the list is executed first.