# [BASIC SQL](https://community.modeanalytics.com/sql/tutorial/introduction-to-sql/)

## SQL SELECT

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

**SELECT** west **AS** "West Region"

**FROM** tutorial.us\_housing\_units

Note that the results will only return capital letters if you put column names in double quotes. The following query, for example, will return results with lower-case column names.

**SELECT** west **AS** West\_Region,

south **AS** South\_Region

**FROM** tutorial.us\_housing\_units

## SQL WHERE

**SELECT** \*

**FROM** tutorial.us\_housing\_units

**WHERE** **month** = 1

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

## SQL Comparison Operators

Equal to =

Not equal to <> or !=

Greater than >

Less than <

Greater than or equal to >=

Less than or equal to <=

**SELECT** \*

**FROM** tutorial.us\_housing\_units

**WHERE** west > 30

**SELECT** \*

**FROM** tutorial.us\_housing\_units

**WHERE** month\_name != 'January'

If you’re using an operator with values that are non-numeric, you need to put the value in single quotes: 'value'.

**Note:** SQL uses double quotes to reference [column names](https://community.modeanalytics.com/sql/tutorial/sql-select-statement/#column-names).

If you’re using >, <, >=, or <=, you don’t necessarily need to be too specific about how you filter. Try this:

**SELECT** \*

**FROM** tutorial.us\_housing\_units

**WHERE** month\_name > 'J'

The way SQL treats alphabetical ordering is a little bit tricky. You may have noticed in the above query that selecting month\_name > 'J' will yield only rows in which month\_name starts with “j” or later in the alphabet. “Wait a minute,” you might say. “January is included in the results—shouldn’t I have to use month\_name >= 'J to make that happen?” SQL considers ‘Ja’ to be greater than ‘J’ because it has an extra letter. It’s worth noting that most dictionaries would list ‘Ja’ after ‘J’ as well.

You can perform arithmetic in SQL using the same operators you would in Excel: +, -, \*, /.

**SELECT** **year**,

**month**,

west,

south,

west + south **AS** south\_plus\_west

**FROM** tutorial.us\_housing\_units

**SELECT** **year**,

**month**,

west,

south,

(west + south)/2 **AS** south\_west\_avg

**FROM** tutorial.us\_housing\_units

## SQL Logical Operators

* [LIKE](https://community.modeanalytics.com/sql/tutorial/sql-like/) allows you to match similar values, instead of exact values.
* [IN](https://community.modeanalytics.com/sql/tutorial/sql-in-operator/) allows you to specify a list of values you’d like to include.
* [BETWEEN](https://community.modeanalytics.com/sql/tutorial/sql-between/) allows you to select only rows within a certain range.
* [IS NULL](https://community.modeanalytics.com/sql/tutorial/sql-is-null/) allows you to select rows that contain no data in a given column.
* [AND](https://community.modeanalytics.com/sql/tutorial/sql-and-operator/) allows you to select only rows that satisfy two conditions.
* [OR](https://community.modeanalytics.com/sql/tutorial/sql-or-operator/) allows you to select rows that satisfy either of two conditions.
* [NOT](https://community.modeanalytics.com/sql/tutorial/sql-not-operator/) allows you to select rows that do not match a certain condition.

## SQL LIKE

**SELECT** \*

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** "group" **LIKE** 'Snoop%'

**Note:** "group" appears in quotations above because GROUP is actually the [name of a function in SQL](https://community.modeanalytics.com/sql/tutorial/sql-aggregate-functions/). The double quotes (as opposed to single: ') are a way of indicating that you are referring to the column name "group", not the SQL function.

The % used above represents any character or set of characters. In this case, % is referred to as a “wildcard.” In the type of SQL that Mode uses, LIKE is case-sensitive, meaning that the above query will only capture matches that start with a capital “S” and lower-case “noop.” To ignore case when you’re matching values, you can use the ILIKE command:

**SELECT** \*

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** "group" **ILIKE** 'snoop%'

You can also use \_ (a single underscore) to substitute for an individual character:

**SELECT** \*

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** artist **ILIKE** 'dr\_ke'

## SQL IN

**SELECT** \*

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** year\_rank **IN** (1, 2, 3)

You can use non-numerical values, but they need to go inside single quotes

**SELECT** \*

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** artist **IN** ('Taylor Swift', 'Usher', 'Ludacris')

## SQL BETWEEN

[dataset](https://community.modeanalytics.com/sql/tutorial/sql-logical-operators/#about-this-dataset):

**SELECT** \*

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** year\_rank **BETWEEN** 5 **AND** 10

Between includes the boundary points, same result as below:

**SELECT** \*

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** year\_rank >= 5 **AND** year\_rank <= 10

## SQL IS NULL

You can select rows that contain no data in a given column by using IS NULL.

**SELECT** \*

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** artist **IS** NULL

WHERE artist = NULL will not work—you can’t perform arithmetic on null values.

## SQL AND

the following query will return all rows for top-10 recordings in 2012.

**SELECT** \*

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** **year** = 2012 **AND** year\_rank <= 10

## SQL OR

You’ll notice that each row will satisfy one of the two conditions. You can combine AND with OR using parenthesis. The following query will return rows that satisfy both of the following conditions:

**SELECT** \*

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** **year** = 2013

**AND** ("group" **ILIKE** '%macklemore%' **OR** "group" **ILIKE** '%timberlake%')

## SQL NOT

NOT is a [logical operator](https://community.modeanalytics.com/sql/tutorial/sql-logical-operators/) in SQL that you can put before any conditional statement to select rows for which that statement is false.

Using NOT with < and > usually doesn’t make sense because you can simply use the opposite comparative operator instead. For example, this query will return an error:

**SELECT** \*

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** **year** = 2013

**AND** year\_rank **NOT** > 3

Instead, you would just write that as:

**SELECT** \*

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** **year** = 2013

**AND** year\_rank <= 3

NOT is commonly used with LIKE. Run this query and check out how Macklemore magically disappears!

**SELECT** \*

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** **year** = 2013

**AND** "group" **NOT** **ILIKE** '%macklemore%'

## SQL ORDER BY

The ORDER BY clause allows you to reorder your results based on the data in one or more columns.

You’ll notice that the results are now ordered alphabetically from a to z based on the content in the artist column. This is referred to as ascending order, and it’s SQL’s default. If you order a numerical column in ascending order, it will start with smaller (or most negative) numbers

## Ordering data by multiple columns

You can also order by mutiple columns. This example query makes the most recent years come first but orders top-ranks songs before lower-ranked songs:

**SELECT** \*

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** year\_rank <= 3

**ORDER** **BY** **year** **DESC**, year\_rank

First, columns in the ORDER BY clause must be separated by commas. Second, the DESC operator is only applied to the column that precedes it.

you can make your life a little easier by substituting numbers for column names in the ORDER BY clause. The numbers will correspond to the order in which you list columns in the SELECT clause.

**SELECT** \*

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** year\_rank <= 3

**ORDER** **BY** 2, 1 **DESC**

When using ORDER BY with a row limit (either through the check box on the query editor or by typing in LIMIT), the ordering clause is executed first. This means that the results are ordered before limiting to only a few rows.

## Using comments

You can use-- (two dashes) to comment out everything to the right of them on a given line:

**SELECT** \* --This comment won't affect the way the code runs

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** **year** = 2013

You can also leave comments across multiple lines using /\*to begin the comment and \*/ to close it:

/\* Here's a comment so long and descriptive that

it could only fit on multiple lines. Fortunately,

it, too, will not affect how this code runs. \*/

**SELECT** \*

**FROM** tutorial.billboard\_top\_100\_year\_end

**WHERE** **year** = 2013

# [INTERMEDIATE SQL](https://community.modeanalytics.com/sql/tutorial/sql-aggregate-functions/)

## SQL Aggregate Functions

## SQL COUNT

COUNT is a [SQL aggregate function](https://community.modeanalytics.com/sql/tutorial/sql-aggregate-functions/) for counting the number of rows in a particular column.

**SELECT** **COUNT**(\*)

**FROM** tutorial.aapl\_historical\_stock\_price

*Note: Typing* COUNT(1) *has the same effect as* COUNT(\*)*. Which one you use is a matter of personal preference.*

Things start to get a little bit tricky when you want to count individual columns. The following code will provide a count of all of rows in which the high column is not null.

**SELECT** **COUNT**(**high**)

**FROM** tutorial.aapl\_historical\_stock\_price

You can add column names (also called *aliases*) using AS:

**SELECT** **COUNT**(date) **AS** count\_of\_date

**FROM** tutorial.aapl\_historical\_stock\_price

## SQL SUM

SUM is a [SQL aggregate function](https://community.modeanalytics.com/sql/tutorial/sql-aggregate-functions/) that totals the values in a given column. Unlike [COUNT](https://community.modeanalytics.com/sql/tutorial/sql-count/), you can only use SUM on columns containing numerical values.

The query below selects the sum of the volume column from the [Apple stock prices dataset](https://community.modeanalytics.com/sql/tutorial/sql-aggregate-functions/#the-apple-stock-prices-dataset):

**SELECT** **SUM**(volume)

**FROM** tutorial.aapl\_historical\_stock\_price

An important thing to remember: **aggregators only aggregate vertically**. If you want to perform a calculation across rows, you would do this with [simple arithmetic](https://community.modeanalytics.com/sql/tutorial/sql-operators/#arithmetic-in-sql).

SELECT SUM(open)/COUNT(open) AS avg\_open\_price

FROM tutorial.aapl\_historical\_stock\_price

## SQL MIN/MAX

MIN and MAX are [SQL aggregation functions](https://community.modeanalytics.com/sql/tutorial/sql-aggregate-functions/) that return the lowest and highest values in a particular column.

They’re similar to [COUNT](https://community.modeanalytics.com/sql/tutorial/sql-count/) in that they can be used on non-numerical columns. Depending on the column type, MIN will return the lowest number, earliest date, or non-numerical value as close alphabetically to “A” as possible. As you might suspect, MAX does the opposite—it returns the highest number, the latest date, or the non-numerical value closest alphabetically to “Z.”

.

**SELECT** **MIN**(volume) **AS** min\_volume,

**MAX**(volume) **AS** max\_volume

**FROM** tutorial.aapl\_historical\_stock\_price

## SQL AVG

AVG is a [SQL aggregate function](https://community.modeanalytics.com/sql/tutorial/sql-aggregate-functions/) that calculates the average of a selected group of values. It’s very useful, but has some limitations. First, it can only be used on numerical columns. Second, it ignores nulls completely.

**SELECT** **AVG**(**high**)

**FROM** tutorial.aapl\_historical\_stock\_price

**WHERE** **high** **IS** **NOT** NULL

The above query produces the same result as the following query:

**SELECT** **AVG**(**high**)

**FROM** tutorial.aapl\_historical\_stock\_price

## SQL GROUP BY

[SQL aggregate functions](https://community.modeanalytics.com/sql/tutorial/sql-aggregate-functions/) like COUNT, AVG, and SUM have something in common: they all aggregate across the entire table.

GROUP BY allows you to separate data into groups, which can be aggregated independently of one another.

**SELECT** **year**,

**COUNT**(\*) **AS** **count**

**FROM** tutorial.aapl\_historical\_stock\_price

**GROUP** **BY** **year**

You can group by multiple columns, but you have to separate column names with commas—just as with [ORDER BY](https://community.modeanalytics.com/sql/tutorial/sql-order-by/):

**SELECT** **year**,

**month**,

**COUNT**(\*) **AS** **count**

**FROM** tutorial.aapl\_historical\_stock\_price

**GROUP** **BY** **year**, **month**

As with ORDER BY, you can substitute numbers for column names in the GROUP BY clause. It’s generally recommended to do this only when you’re grouping many columns, or if something else is causing the text in the GROUP BY clause to be excessively long:

**SELECT** **year**,

**month**,

**COUNT**(\*) **AS** **count**

**FROM** tutorial.aapl\_historical\_stock\_price

**GROUP** **BY** 1, 2

### Using GROUP BY with ORDER BY

The order of column names in your GROUP BY clause doesn’t matter—the results will be the same regardless. If you want to control how the aggregations are grouped together, use ORDER BY.

**SELECT** **year**,

**month**,

**COUNT**(\*) **AS** **count**

**FROM** tutorial.aapl\_historical\_stock\_price

**GROUP** **BY** **year**, **month**

**ORDER** **BY** **month**, **year**

### Using GROUP BY with LIMIT

There’s one thing to be aware of as you group by multiple columns: SQL evaluates the aggregations before the LIMIT clause. If you don’t group by any columns, you’ll get a 1-row result—no problem there. If you group by a column with enough unique values that it exceeds the LIMIT number, the aggregates will be calculated, and then some rows will simply be omitted from the results.

## SQL HAVING

You’ll often encounter datasets where GROUP BYisn’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. The WHERE clause won’t work for this because it doesn’t allow you to filter on aggregate columns—that’s where the HAVING clause comes in:

**SELECT** **year**,

**month**,

**MAX**(**high**) **AS** month\_high

**FROM** tutorial.aapl\_historical\_stock\_price

**GROUP** **BY** **year**, **month**

**HAVING** **MAX**(**high**) > 400

**ORDER** **BY** **year**, **month**

*Note:* HAVING *is the “clean” way to filter* a query that has been aggregated, but this is also commonly done using a subquery,

## 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

## SQL DISTINCT

### Using DISTINCT for viewing unique values

You’ll occasionally want to look at only the unique values in a particular column. You can do this using SELECT DISTINCT syntax. To select unique values from the month column in the [Apple stock prices dataset](https://community.modeanalytics.com/sql/tutorial/sql-aggregate-functions/#the-apple-stock-prices-dataset), you’d use the following query:

**SELECT** **DISTINCT** **month**

**FROM** tutorial.aapl\_historical\_stock\_price

If you include two (or more) columns in a SELECT DISTINCT clause, your results will contain all of the unique pairs of those two columns:

**SELECT** **DISTINCT** **year**, **month**

**FROM** tutorial.aapl\_historical\_stock\_price

*Note: You only need to include* DISTINCT *once in your* SELECT *clause—you do not need to add it for each column name.*

### Using DISTINCT in aggregations

You can use DISTINCT when performing an aggregation. You’ll probably use it most commonly with the [COUNT](https://community.modeanalytics.com/sql/tutorial/sql-count/) function.

**SELECT** **COUNT**(**DISTINCT** **month**) **AS** unique\_months

**FROM** tutorial.aapl\_historical\_stock\_price

The results show that there are 12 unique values (other examples may be less obvious).

You might follow this up by taking average trade volumes by month to get a sense of when Apple stock really moves:

**SELECT** **month**,

**AVG**(volume) **AS** avg\_trade\_volume

**FROM** tutorial.aapl\_historical\_stock\_price

**GROUP** **BY** **month**

**ORDER** **BY** 2 **DESC**

Okay, back to DISTINCT. You’ll notice that DISTINCT goes inside the [aggregate function](https://community.modeanalytics.com/sql/tutorial/sql-aggregate-functions/) rather than at the beginning of the SELECT clause. Of course, you can SUM or AVG the distinct values in a column, but there are fewer practical applications for them. For MAX and MIN, you probably shouldn’t ever use DISTINCT because the results will be the same as without DISTINCT, and the DISTINCT function will make your query substantially slower to return results.

## SQL 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. It 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:

**SELECT** player\_name,

**year**,

**CASE** **WHEN** **year** = 'SR' **THEN** 'yes'

**ELSE** NULL **END** **AS** is\_a\_senior

**FROM** benn.college\_football\_players

 What if you don’t want null values in the is\_a\_senior column? The following query replaces those nulls with “no”:

**SELECT** player\_name,

**year**,

**CASE** **WHEN** **year** = 'SR' **THEN** 'yes'

**ELSE** 'no' **END** **AS** is\_a\_senior

**FROM** benn.college\_football\_players

You can also define a number of outcomes in a CASE statement by including as many WHEN/THEN statements as you’d like:

**SELECT** player\_name,

weight,

**CASE** **WHEN** weight > 250 **THEN** 'over 250'

**WHEN** weight > 200 **THEN** '201-250'

**WHEN** weight > 175 **THEN** '176-200'

**ELSE** '175 or under' **END** **AS** weight\_group

**FROM** benn.college\_football\_players

 A better way to write the above would be:

**SELECT** player\_name,

weight,

**CASE** **WHEN** weight > 250 **THEN** 'over 250'

**WHEN** weight > 200 **AND** weight <= 250 **THEN** '201-250'

**WHEN** weight > 175 **AND** weight <= 200 **THEN** '176-200'

**ELSE** '175 or under' **END** **AS** weight\_group

**FROM** benn.college\_football\_players

You can also string together multiple conditional statements with AND and OR the same way you might in a WHERE clause:

**SELECT** player\_name,

**CASE** **WHEN** **year** = 'FR' **AND** **position** = 'WR' **THEN** 'frosh\_wr'

**ELSE** NULL **END** **AS** sample\_case\_statement

**FROM** benn.college\_football\_players

### A quick review of CASE basics:

1. The CASE statement always goes in the SELECT clause
2. CASE must include the following components: WHEN,T HEN, and END. ELSE is an optional component.
3. You can make any conditional statement using any conditional operator (like [WHERE](https://community.modeanalytics.com/sql/tutorial/sql-where/)) between WHEN and THEN. This includes stringing together multiple conditional statements using AND and OR.
4. You can include multiple WHEN statements, as well as anELSE statement to deal with any unaddressed conditions.

### Using CASE with aggregate functions

Since [COUNT](https://community.modeanalytics.com/sql/tutorial/sql-count/) ignores nulls, you could use a CASE statement to evaluate the condition and produce null or non-null values depending on the outcome:

**SELECT** **CASE** **WHEN** **year** = 'FR' **THEN** 'FR'

**ELSE** 'Not FR' **END** **AS** year\_group,

**COUNT**(1) **AS** **count**

**FROM** benn.college\_football\_players

**GROUP** **BY** **CASE** **WHEN** **year** = 'FR' **THEN** 'FR'

**ELSE** 'Not FR' **END**

Now, you might be thinking “why wouldn’t I just use a WHERE clause to filter out the rows I don’t want to count?” You could do that—it would look like this:

**SELECT** **COUNT**(1) **AS** fr\_count

**FROM** benn.college\_football\_players

**WHERE** **year** = 'FR'

But what if you also wanted to count a couple other conditions? Using the WHERE clause only allows you to count one condition. Here’s an example of counting multiple conditions in one query:

**SELECT** **CASE** **WHEN** **year** = 'FR' **THEN** 'FR'

**WHEN** **year** = 'SO' **THEN** 'SO'

**WHEN** **year** = 'JR' **THEN** 'JR'

**WHEN** **year** = 'SR' **THEN** 'SR'

**ELSE** 'No Year Data' **END** **AS** year\_group,

**COUNT**(1) **AS** **count**

**FROM** benn.college\_football\_players

**GROUP** **BY** 1

### Using CASE inside of aggregate functions

In the previous examples, data was displayed vertically, but in some instances, you might want to show data horizontally. This is known as “pivoting” (like a [pivot table](https://en.wikipedia.org/wiki/Pivot_table) in Excel). Let’s take the previous query and re-orient it horizontally:

**SELECT** **COUNT**(**CASE** **WHEN** **year** = 'FR' **THEN** 1 **ELSE** NULL **END**) **AS** fr\_count,

**COUNT**(**CASE** **WHEN** **year** = 'SO' **THEN** 1 **ELSE** NULL **END**) **AS** so\_count,

**COUNT**(**CASE** **WHEN** **year** = 'JR' **THEN** 1 **ELSE** NULL **END**) **AS** jr\_count,

**COUNT**(**CASE** **WHEN** **year** = 'SR' **THEN** 1 **ELSE** NULL **END**) **AS** sr\_count

**FROM** benn.college\_football\_players

## SQL Joins

### Aliases in SQL

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.

SELECT players.school\_name,

players.player\_name,

players.position,

players.weight

FROM benn.college\_football\_players players

WHERE players.state = 'GA'

ORDER BY players.weight DESC

### JOIN and ON

ON indicates how the two tables (the one after the FROM and the one after the JOIN) relate to each other. 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:

**ON** teams.school\_name = players.school\_name

**SELECT** \*

**FROM** benn.college\_football\_players players

**JOIN** benn.college\_football\_teams teams

**ON** teams.school\_name = players.school\_name

Note that SELECT \* returns all of the columns from both tables, not just from the table after FROM. If you want to only return columns from one table, you can write SELECT players.\* to return all the columns from the players table.

## INNER JOIN

We’ll start with inner joins, which can be written as eitherJOIN 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 i

### Joining tables with identical column names

In the below example, both tables have columns called school\_name:

**SELECT** players.\*,

teams.\*

**FROM** benn.college\_football\_players players

**JOIN** benn.college\_football\_teams teams

**ON** teams.school\_name = players.school\_name

The results can only support one column with a given name—when you include 2 columns of the same name, the results will simply show the exact same result set for both columns **even if the two columns should contain different data**. You can avoid this by naming the columns individually. It happens that these two columns will actually contain the same data because they are used for the join key, but the following query technically allows these columns to be independent:

**SELECT** players.school\_name **AS** players\_school\_name,

teams.school\_name **AS** teams\_school\_name

**FROM** benn.college\_football\_players players

**JOIN** benn.college\_football\_teams teams

**ON** teams.school\_name = players.school\_name

## SQL Outer Joins

There are a few types of outer joins:

* [LEFT JOIN](https://community.modeanalytics.com/sql/tutorial/sql-left-join/) returns only unmatched rows from the left table.
* [RIGHT JOIN](https://community.modeanalytics.com/sql/tutorial/sql-right-join/) returns only unmatched rows from the right table.
* [FULL OUTER JOIN](https://community.modeanalytics.com/sql/tutorial/sql-full-outer-join/) returns unmatched rows from both tables.
* visualization http://joins.spathon.com/

## SQL LEFT JOIN

**SELECT** companies.permalink **AS** companies\_permalink,

companies.name **AS** companies\_name,

acquisitions.company\_permalink **AS** acquisitions\_permalink,

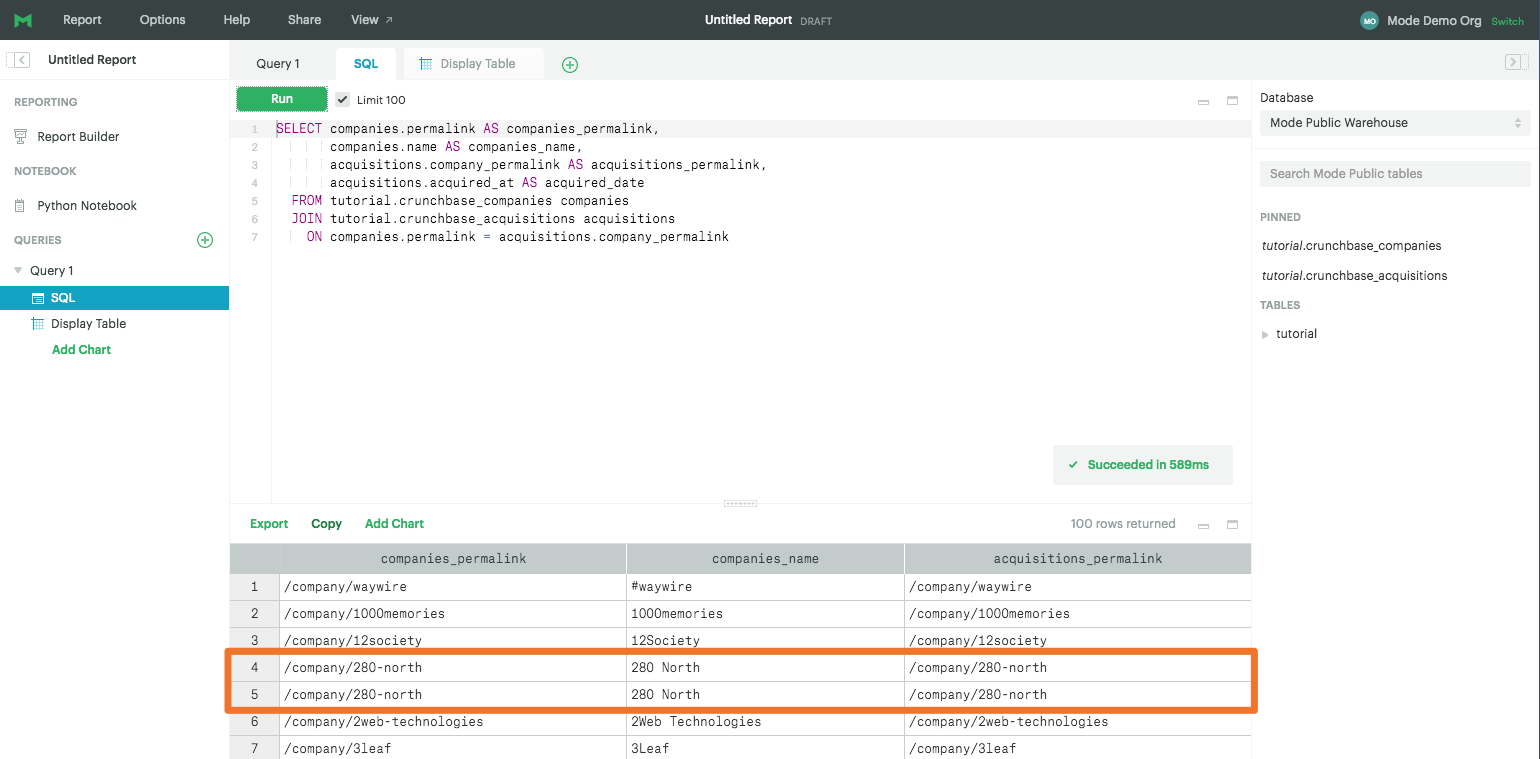
acquisitions.acquired\_at **AS** acquired\_date

**FROM** tutorial.crunchbase\_companies companies

**JOIN** tutorial.crunchbase\_acquisitions acquisitions

**ON** companies.permalink = acquisitions.company\_permalink

You may notice that “280 North” appears twice in this list. That is because it has two entries in the tutorial.crunchbase\_acquisitions table, both of which are being joined onto the tutorial.crunchbase\_companies table.



Now try running that query as a LEFT JOIN:

**SELECT** companies.permalink **AS** companies\_permalink,

companies.name **AS** companies\_name,

acquisitions.company\_permalink **AS** acquisitions\_permalink,

acquisitions.acquired\_at **AS** acquired\_date

**FROM** tutorial.crunchbase\_companies companies

**LEFT** **JOIN** tutorial.crunchbase\_acquisitions acquisitions

**ON** companies.permalink = acquisitions.company\_permalink

You can see that the first two companies from the previous result set, #waywire and 1000memories, are pushed down the page by a number of results that contain null values in the acquisitions\_permalink and acquired\_date fields.

This is because the LEFT JOIN command tells the database to return all rows in the table in the FROM clause, regardless of whether or not they have matches in the table in the LEFT JOIN clause.

## SQL RIGHT JOIN

RIGHT JOIN is rarely used because you can achieve the results of a RIGHT JOIN by simply switching the two joined table names in a LEFT JOIN.

The convention of always using LEFT JOIN probably exists to make queries easier to read and audit, but beyond that there isn’t necessarily a strong reason to avoid using RIGHT JOIN.

It’s worth noting that LEFT JOIN and RIGHT JOIN can be written as LEFT OUTER JOIN and RIGHT OUTER JOIN, respectively.

## SQL Joins Using WHERE or ON

Normally, filtering is processed in the [WHERE](https://community.modeanalytics.com/sql/tutorial/sql-where/) clause once the [two tables have already been joined](https://community.modeanalytics.com/sql/tutorial/sql-joins/). It’s possible, though that you might want to filter one or both of the tables before joining them

**SELECT** companies.permalink **AS** companies\_permalink,

companies.name **AS** companies\_name,

acquisitions.company\_permalink **AS** acquisitions\_permalink,

acquisitions.acquired\_at **AS** acquired\_date

**FROM** tutorial.crunchbase\_companies companies

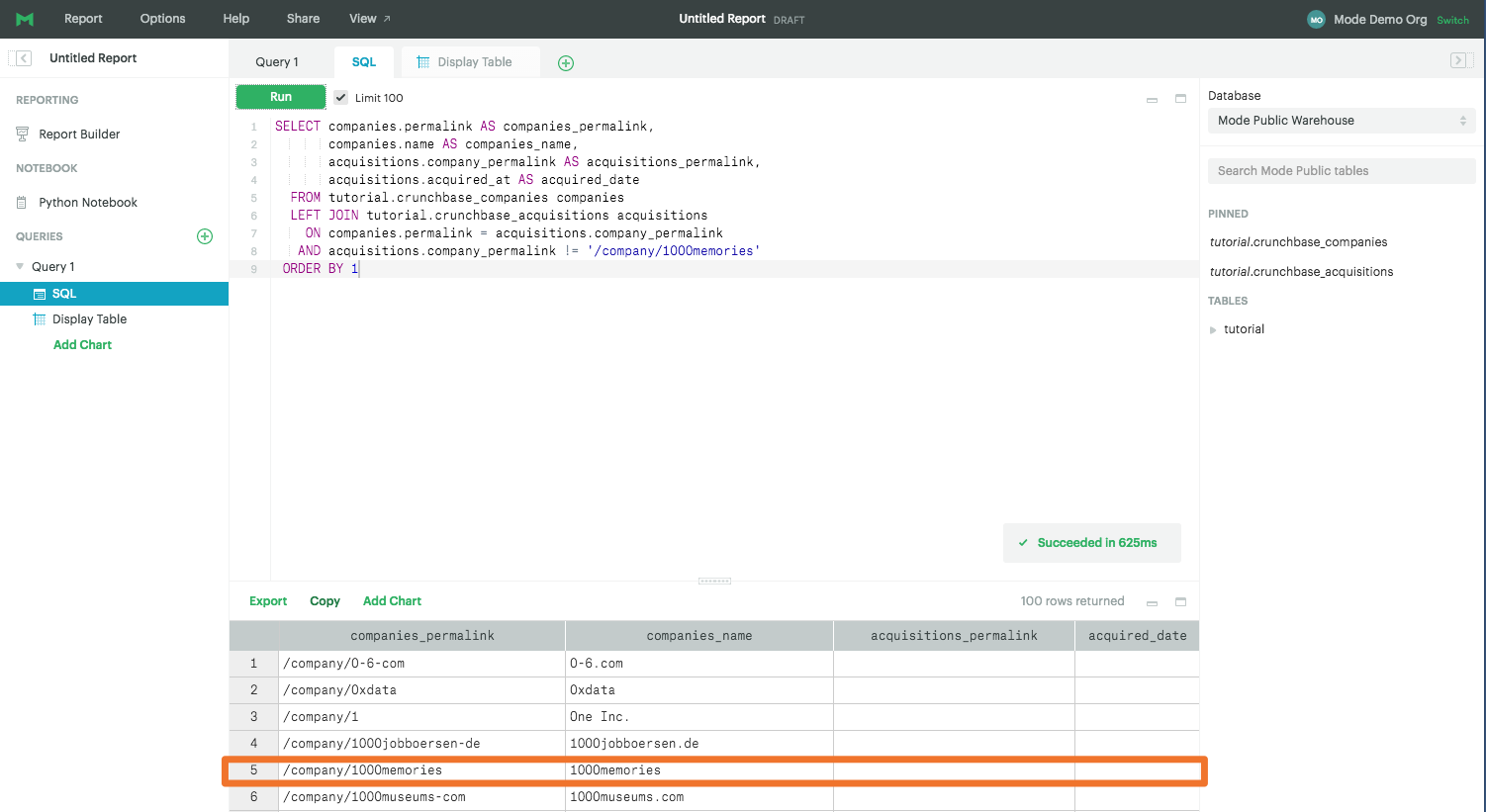
**LEFT** **JOIN** tutorial.crunchbase\_acquisitions acquisitions

**ON** companies.permalink = acquisitions.company\_permalink

**AND** acquisitions.company\_permalink != '/company/1000memories'

**ORDER** **BY** 1

What’s happening above is that the conditional statement AND... is evaluated before the join occurs. You can think of it as a WHERE clause that only applies to one of the tables. You can tell that this is only happening in one of the tables because the 1000memories permalink is still displayed in the column that pulls from the other table



## Filtering in the WHERE clause

The result is that the 1000memories row is joined onto the original table, but then it is filtered out entirely (in both tables) in the WHERE clause before displaying results.

**SELECT** companies.permalink **AS** companies\_permalink,

companies.name **AS** companies\_name,

acquisitions.company\_permalink **AS** acquisitions\_permalink,

acquisitions.acquired\_at **AS** acquired\_date

**FROM** tutorial.crunchbase\_companies companies

**LEFT** **JOIN** tutorial.crunchbase\_acquisitions acquisitions

**ON** companies.permalink = acquisitions.company\_permalink

**WHERE** acquisitions.company\_permalink != '/company/1000memories'

**OR** acquisitions.company\_permalink **IS** NULL

**ORDER** **BY** 1

You can see that the 1000memories line is not returned (it would have been between the two highlighted lines below). Also note that filtering in the WHERE clause can also filter null values, so we added an extra line to make sure to include the nulls.

## SQL FULL OUTER JOIN

 It is commonly used in conjunction with aggregations to understand the amount of overlap between two tables.

**SELECT** **COUNT**(**CASE** **WHEN** companies.permalink **IS** **NOT** NULL **AND** acquisitions.company\_permalink **IS** NULL

**THEN** companies.permalink **ELSE** NULL **END**) **AS** companies\_only,

**COUNT**(**CASE** **WHEN** companies.permalink **IS** **NOT** NULL **AND** acquisitions.company\_permalink **IS** **NOT** NULL

**THEN** companies.permalink **ELSE** NULL **END**) **AS** both\_tables,

**COUNT**(**CASE** **WHEN** companies.permalink **IS** NULL **AND** acquisitions.company\_permalink **IS** **NOT** NULL

**THEN** acquisitions.company\_permalink **ELSE** NULL **END**) **AS** acquisitions\_only

**FROM** tutorial.crunchbase\_companies companies

**FULL** **JOIN** tutorial.crunchbase\_acquisitions acquisitions

**ON** companies.permalink = acquisitions.company\_permalink

One important thing to keep in mind is that you must count from the crunchbase\_acquisitions table in order to get unmatched rows in that table—if you were to count companies.permalink as in the first two columns, you would get a result of 0 in the third column because it would be counting up a bunch of null values.

## SQL UNION

[SQL joins](https://community.modeanalytics.com/sql/tutorial/sql-joins/) allow you to combine two datasets side-by-side, but 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.

**SELECT** \*

**FROM** tutorial.crunchbase\_investments\_part1

**UNION**

**SELECT** \*

**FROM** tutorial.crunchbase\_investments\_part2

Note that UNION only appends distinct values. More specifically, when you use UNION, the dataset is appended, and any rows in the appended table that are exactly identical to rows in the first table are dropped. If you’d like to append all the values from the second table, use UNION ALL. You’ll likely use UNION ALL far more often than UNION. In this particular case, there are no duplicate rows, so UNION ALL will produce the same results

**SELECT** \*

**FROM** tutorial.crunchbase\_investments\_part1

**UNION** ALL

**SELECT** \*

**FROM** tutorial.crunchbase\_investments\_part2

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

While the column names don’t necessarily have to be the same, you will find that they typically are. This is because most of the instances in which you’d want to use UNIONinvolve stitching together different parts of the same dataset (as is the case here).

Since you are writing two separate SELECT statements, you can treat them differently before appending. For example, you can filter them differently using different WHERE clauses.

## SQL Joins with Comparison Operators

In the lessons so far, you’ve only [joined tables](https://community.modeanalytics.com/sql/tutorial/sql-joins/) by exactly matching values from both tables. However, you can enter any type of conditional statement into the ON clause. Here’s an example using > to join only investments that occurred more than 5 years after each company’s founding year:

**SELECT** companies.permalink,

companies.name,

companies.status,

**COUNT**(investments.investor\_permalink) **AS** investors

**FROM** tutorial.crunchbase\_companies companies

**LEFT** **JOIN** tutorial.crunchbase\_investments\_part1 investments

**ON** companies.permalink = investments.company\_permalink

**AND** investments.funded\_year > companies.founded\_year + 5

**GROUP** **BY** 1,2, 3

This technique is especially useful for creating date ranges as shown above. It’s important to note that this produces a different result than the following query because it only joins rows that fit the investments.funded\_year > companies.founded\_year + 5 condition rather than joining all rows and then filtering:

**SELECT** companies.permalink,

companies.name,

companies.status,

**COUNT**(investments.investor\_permalink) **AS** investors

**FROM** tutorial.crunchbase\_companies companies

**LEFT** **JOIN** tutorial.crunchbase\_investments\_part1 investments

**ON** companies.permalink = investments.company\_permalink

**WHERE** investments.funded\_year > companies.founded\_year + 5

**GROUP** **BY** 1,2, 3

## SQL Joins on Multiple Keys

There are couple reasons you might want to [join tables](https://community.modeanalytics.com/sql/tutorial/sql-joins/) on multiple foreign keys. The first has to do with accuracy.

The second reason has to do with performance. SQL uses “indexes” (essentially pre-defined joins) to speed up queries. This will be covered in greater detail the lesson on [making queries run faster](https://community.modeanalytics.com/sql/tutorial/sql-performance-tuning/), but for all you need to know is that it can occasionally make your query run faster to join on multiple fields, even when it does not add to the accuracy of the query. For example, the results of the following query will be the same with or without the last line. However, it is possible to optimize the database such that the query runs more quickly with the last line included:

**SELECT** companies.permalink,

companies.name,

investments.company\_name,

investments.company\_permalink

**FROM** tutorial.crunchbase\_companies companies

**LEFT** **JOIN** tutorial.crunchbase\_investments\_part1 investments

**ON** companies.permalink = investments.company\_permalink

**AND** companies.name = investments.company\_name

It’s worth noting that this will have relatively little effect on small datasets.

## SQL Self Joins

## Self joining tables

Sometimes it can be useful to join a table to itself. Let’s say you wanted to identify companies that received an investment from Great Britain following an investment from Japan.

**SELECT** **DISTINCT** japan\_investments.company\_name,

japan\_investments.company\_permalink

**FROM** tutorial.crunchbase\_investments\_part1 japan\_investments

**JOIN** tutorial.crunchbase\_investments\_part1 gb\_investments

**ON** japan\_investments.company\_name = gb\_investments.company\_name

**AND** gb\_investments.investor\_country\_code = 'GBR'

**AND** gb\_investments.funded\_at > japan\_investments.funded\_at

**WHERE** japan\_investments.investor\_country\_code = 'JPN'

**ORDER** **BY** 1

Note how the same table can easily be referenced multiple times using different aliases—in this case, japan\_investments and gb\_investments.