The SQL Tutorial for Data Analysis

This tutorial is designed for people who want to answer questions with data. For many, SQL is the “meat and potatoes” of data analysis—it’s used for accessing, cleaning, and analysing data that’s stored in databases. It’s very easy to learn, yet it’s employed by the world’s largest companies to solve incredibly challenging problems.

In particular, this tutorial is meant for aspiring analysts who have used Excel a little bit but have no coding experience.

Though some of the lessons may be useful for software developers using SQL in their applications, this tutorial doesn’t cover how to set up SQL databases or how to use them in software applications—it is not a comprehensive resource for aspiring software developers.

How the SQL Tutorial for Data Analysis works

The entire tutorial is meant to be completed using Mode, an analytics platform that brings together a SQL editor, Python notebook, and data visualization builder. You should open up another browser window to Mode. You’ll retain the most information if you run the example queries and try to understand results, and complete the practice exercises.

Note: You will need to have a Mode user account in order to start the tutorial. You can sign up for one at modeanalytics.com.

What is SQL?

SQL (Structured Query Language) is a programming language designed for managing data in a relational database. It’s been around since the 1970s and is the most common method of accessing data in databases today. SQL has a variety of functions that allow its users to read, manipulate, and change data. Though SQL is commonly used by engineers in software development, it’s also popular with data analysts for a few reasons:

- It’s semantically easy to understand and learn.
- Because it can be used to access large amounts of data directly where it’s stored, analysts don’t have to copy data into other applications.

- Compared to spreadsheet tools, data analysis done in SQL is easy to audit and replicate. For analysts, this means no more looking for the cell with the typo in the formula.

SQL is great for performing the types of aggregations that you might normally do in an Excel pivot table—sums, counts, minimums and maximums, etc.—but over much larger datasets and on multiple tables at the same time.

How do I pronounce SQL?

We have no idea.

What's a database?

From Wikipedia: A database is an organised collection of data.

There are many ways to organize a database and many different types of databases designed for different purposes. Mode's structure is fairly simple:

If you've used Excel, you should already be familiar with tables—they're similar to spreadsheets. Tables have rows and columns just like Excel, but are a little more rigid. Database tables, for instance, are always organised by column, and each column must have a unique name. To get a sense of this organization, the image below shows a sample table containing data from the 2010 Academy Awards:

Broadly, within databases, tables are organised in schemas. At Mode, we organize tables around the users who upload them, so each person has his or her own schema. Schemas are defined by usernames, so if your username is databass3000, all of the tables you upload will be stored under the databass3000 schema. For example, if databass3000 uploads a table on fish food sales called `fish_food_sales`, that table would be referenced as `databass3000.fish_food_sales`. You'll notice that all of the tables used in this tutorial series are prefixed with "tutorial." That's because they were uploaded by an account with that username.

You're on your way!

Now that you're familiar with the basics, it's time to dive in and learn some SQL.

SQL SELECT

Check out the beginning.

Basic syntax: SELECT and FROM

There are two required ingredients in any SQL query: `SELECT` and `FROM`—and they have to be in that order. `SELECT` indicates which columns you'd like to view, and `FROM` identifies the table that they live in.

Let's start by looking at a couple columns from the housing unit table:

```
SELECT year,
       month,
       west
  FROM tutorial.us_housing_units
```

To see the results yourself, copy and paste this query into Mode's Query Editor and run the code. If you already have SQL code in the Query Editor, you'll need to paste over or delete the query that was there previously. If you simply copy and paste this query below the previous one, you'll get an error—you can only run one `SELECT` statement at a time.

Try it out.

So what's happening in the above query? In this case, the query is telling the database to return the `year`, `month`, and `west` columns from the table `tutorial.us_housing_units`. (Remember that when referencing tables, the table names have to be preceded by the name of user who uploaded it.) When you run this query, you'll get back a set of results that shows values in each of these columns.

Note that the three column names were separated by a comma in the query. Whenever you select multiple columns, they must be separated by commas, but you should not include a comma after the last column name.

If you want to select every column in a table, you can use `*` instead of the column names:

```
SELECT *
  FROM tutorial.us_housing_units
```

Now try this practice problem for yourself:

Write a query to select all of the columns in the `tutorial.us_housing_units` table without using `*`.

Try it out | See the answer

Note: Practice problems will appear in boxes like the one above throughout this tutorial.

When you've completed the above practice problem, check your answer by clicking "See the answer." Following the link will show you our solution SQL query. To see the results produced by this solution query, click "Results" in the left sidebar:

This will show you a table of query results that should be the same as your query results (if your answer is correct):

To compare your query or results with our solution, jump back to the window where you're editing your practice solutions. There's lots to explore in the Editor (see "How to use the query editor" to learn more), but to start you might want to experiment with creating a chart using our drag-and-drop chart builder–just click on the green plus button next to the "Display Table" tab:

This will take you to Mode's drag-and-drop chart builder. For more about building charts in Mode, check out "How to build charts."

If you're feeling particularly proud of your work, you might want to explore how it looks in Mode's Report View—a cleaned-up view meant for sharing queries and results. Just click on "View" in the header:

Now you'll be looking at a cleaned-up version of your report fit for sharing. You can learn more about viewing and building reports on Mode's help site. For now, the most important thing to know is that you can share this report with anyone by clicking the "Share" menu in the Query Editor and selecting the channel you'd like to use for sharing:

Send to all your friends by email or Slack!

You can also share your work in progress from the Editor view, where you've been writing your queries. To get back to editing your query, click on "Edit" in the header bar:

You'll land back in the Query Editor, where you can edit your SQL, your charts, or your reports.

What actually happens when you run a query?

Let's get back to it! When you run a query, what do you get back? As you can see from running the queries above, you get a table. But that table isn't stored permanently in the database. It also doesn't change any tables in the database—`tutorial.us_housing_units` will contain the same data every time you query it, and the data will never change no matter how many times you query it. Mode does store all of your results for future access, but `SELECT` statements don't change anything in the underlying tables.

Formatting convention

You might have noticed that the `SELECT` and `FROM' commands are capitalised. This isn't actually necessary—SQL will understand these commands if you type them in lowercase. Capitalizing commands is simply a convention that makes queries easier to read. Similarly, SQL treats one space, multiple spaces, or a line break as being the same thing. For example, SQL treats this the same way it does the previous query:

```
SELECT *         FROM tutorial.us_housing_units
```

It also treats this the same way:

```
SELECT *
  FROM tutorial.us_housing_units
```

While most capitalization conventions are the same, there are several conventions for formatting line breaks. You'll pick up on several of these in this tutorial and in

other people's work on Mode. It's up to you to determine what formatting method is easiest for you to read and understand.

Column names

While we're on the topic of formatting, it's worth noting the format of column names. All of the columns in the `tutorial.us_housing_units` table are named in lower case, and use underscores instead of spaces. The table name itself also uses underscores instead of spaces. Most people avoid putting spaces in column names because it's annoying to deal with spaces in SQL—if you want to have spaces in column names, you need to always refer to those columns in double quotes.

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
```

Without the double quotes, that query would read 'West' and 'Region' as separate objects and would return an error. 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

Check out the beginning.

The SQL WHERE clause

Start by running a SELECT statement to re-familiarize yourself with the housing data used in this tutorial. Remember to switch over to Mode and run any of the code you see in the light blue boxes to get a sense of what the output will look like.

```
SELECT * FROM tutorial.us_housing_units
```

Once you know how to view some data using `SELECT` and `FROM`, the next step is filtering the data using the `WHERE` clause. Here's what it looks like:

```
SELECT *
  FROM tutorial.us_housing_units
 WHERE month = 1
```

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

How does WHERE work?

The SQL `WHERE` clause works in a plain-English way: the above query does the same thing as `SELECT * FROM tutorial.us_housing_units`, except that the results will only include rows where the `month` column contains the value `1`.

In Excel, it's possible to sort data in such a way that one column can be reordered without reordering any of the other columns—though that could badly scramble your data. When using SQL, entire rows of data are preserved together. If you write a `WHERE` clause that filters based on values in one column, you'll limit the results *in all columns* to rows that satisfy the condition. The idea is that each row is one data point or observation, and all the information contained in that row belongs together.

You can filter your results in a number of ways using comparison and logical operators, which you'll learn about in the next lesson.

SQL Comparison Operators

Check out the beginning.

Comparison operators on numerical data

The most basic way to filter data is using comparison operators. The easiest way to understand them is to start by looking at a list of them:

Equal to	`=`
Not equal to	`<>` or `!=`
Greater than	`>`
Less than	`<`
Greater than or equal to	`>=`
Less than or equal to	`<=`

These comparison operators make the most sense when applied to numerical columns. For example, let's use `>` to return only the rows where the West Region produced more than 30,000 housing units (remember, the units in this data table are already in thousands):

```
SELECT *
  FROM tutorial.us_housing_units
 WHERE west > 30
```

Try running that query with each of the operators in place of `>`. Try some values other than `30` to get a sense of how SQL operators work. When you're ready, try out the practice problems.

Did the West Region ever produce *more than 50,000* housing units in one month?

Try it out See the answer

Did the South Region ever produce *20,000 or fewer* housing units in one month?

Try it out See the answer

Comparison operators on non-numerical data

All of the above operators work on non-numerical data as well. `=` and `!=` make perfect sense—they allow you to select rows that match or don't match any value, respectively. For example, run the following query and you'll notice that none of the January rows show up:

```
SELECT *
  FROM tutorial.us_housing_units
 WHERE month_name != 'January'
```

There are some important rules when using these operators, though. 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 single quotes to reference column values.

You can use `>`, `<`, and the rest of the comparison operators on non-numeric columns as well—they filter based on alphabetical order. Try it out a couple times with different operators:

```
SELECT *
  FROM tutorial.us_housing_units
 WHERE month_name > 'January'
```

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.

Write a query that only shows rows for which the month name is February.

Try it out | See the answer

Write a query that only shows rows for which the `month_name` starts with the letter "N" or an earlier letter in the alphabet.

Try it out | See the answer

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, which are covered in the Intermediate SQL section of this tutorial.

The example below illustrates the use of `+`:

```
SELECT year,
       month,
       west,
       south,
       west + south AS south_plus_west
  FROM tutorial.us_housing_units
```

The above example produces a column showing the sum of whatever is in the `south`and `west` columns for each row. You can chain arithmetic functions, including both column names and actual numbers:

```
SELECT year,
       month,
       west,
       south,
       west + south - 4 * year AS nonsense_column
  FROM tutorial.us_housing_units
```

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.

Write a query that calculates the sum of all four regions in a separate column.

Try it out See the answer

As in Excel, you can use parentheses to manage the order of operations. For example, if you wanted to average the `west` and `south` columns, you could write something like this:

```
SELECT year,
       month,
       west,
       south,
       (west + south)/2 AS south_west_avg
  FROM tutorial.us_housing_units
```

It occasionally makes sense to use parentheses even when it's not absolutely necessary just to make your query easier to read.

Sharpen your SQL skills

Write a query that returns all rows for which more units were produced in the West region than in the Midwest and Northeast combined.

Try it out See the answer

Write a query that calculates the percentage of all houses completed in the United States represented by each region. Only return results from the year 2000 and later.

Hint: There should be four columns of percentages.

SQL Logical Operators

Check out the beginning.

SQL Logical operators

In the previous lesson, you played with some comparison operators to filter data. You'll likely also want to filter data using several conditions—possibly more often than you'll want to filter by only one condition. Logical operators allow you to use multiple comparison operators in one query.

Each logical operator is a special snowflake, so we'll go through them individually in the following lessons. Here's a quick preview:

- `LIKE` allows you to match similar values, instead of exact values.
- `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.

About this dataset

To practice logical operators in SQL, you'll be using data from Billboard Music Charts. It was collected in January 2014 and contains data from 1956 through 2013. The results in this table are the *year-end* results—the top 100 songs at the end of each year.

To access the dataset, use the following query:

```
SELECT * FROM tutorial.billboard_top_100_year_end
```

- `year_rank` is the rank of that song at the end of the listed year.
- `group` is the name of the entire group that won (this could be multiple artists if there was a collaboration).
- `artist` is an individual artist. This is a little complicated, as an artist can be an individual or group.

You can get a better sense of some of the nuances of this dataset by running the query below. It uses the `ORDER BY` clause, which you'll learn about in a later lesson. Don't worry about it for now:

```
SELECT *
  FROM tutorial.billboard_top_100_year_end
 ORDER BY year DESC, year_rank
```

You'll notice that Macklemore does a lot of collaborations. Since his songs are listed as featuring other artists like Ryan Lewis, there are multiple lines in the dataset for Ryan Lewis. Daft Punk and Pharrell Williams are also listed as two artists. Daft Punk is actually a duo, but since the album lists them together under the name Daft Punk, that's how Billboard treats them.

Now onto learning about each logical operator!

SQL LIKE

Check out the beginning.

The SQL LIKE operator

`LIKE` is a logical operator in SQL that allows you to match on similar values rather than exact ones.

In this example, the results from the Billboard Music Charts dataset will include rows for which `"group"` starts with “Snoop” and is followed by any number and selection of characters.

Run the code to see which results are returned.

```
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. 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. In general, putting double quotes around a word or phrase will indicate that you are referring to that column name.

Wildcards

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.”

```
SELECT *
  FROM tutorial.billboard_top_100_year_end
 WHERE "group" LIKE 'Snoop%'
```

Sharpen your SQL skills

Write a query that returns all rows for which Ludacris was a member of the group.

Try it out | See the answer

Write a query that returns all rows for which the first artist listed in the group has a name that begins with "DJ".

Try it out | See the answer

NEXT TUTORIAL

SQL IN

Check out the beginning.

The SQL IN operator

`IN` is a logical operator in SQL that allows you to specify a list of values that you'd like to include in the results. For example, the following query of data from the Billboard Music Charts will return results for which the `year_rank` column is equal to one of the values in the list:

```
SELECT *
  FROM tutorial.billboard_top_100_year_end
 WHERE year_rank IN (1, 2, 3)
```

As with comparison operators, you can use non-numerical values, but they need to go inside single quotes. Regardless of the data type, the values in the list must be separated by commas. Here's another example:

```
SELECT *
  FROM tutorial.billboard_top_100_year_end
 WHERE artist IN ('Taylor Swift', 'Usher', 'Ludacris')
```

Sharpen your SQL skills

Write a query that shows all of the entries for Elvis and M.C. Hammer.

Hint: M.C. Hammer is actually on the list under multiple names, so you may need to first write a query to figure out exactly how M.C. Hammer is listed. You're likely to face similar problems that require some exploration in many real-life scenarios.

Try it out | See the answer

SQL BETWEEN

Check out the beginning.

The SQL BETWEEN operator

`BETWEEN` is a logical operator in SQL that allows you to select only rows that are within a specific range. It has to be paired with the `AND` operator, which you'll learn about in a later lesson. Here's what `BETWEEN` looks like on a Billboard Music Charts dataset:

```
SELECT *
  FROM tutorial.billboard_top_100_year_end
 WHERE year_rank BETWEEN 5 AND 10
```

`BETWEEN` includes the range bounds (in this case, 5 and 10) that you specify in the query, in addition to the values between them. So the above query will return the exact same results as the following query:

```
SELECT *
  FROM tutorial.billboard_top_100_year_end
 WHERE year_rank >= 5 AND year_rank <= 10
```

Some people prefer the latter example because it more explicitly shows what the query is doing (it's easy to forget whether or not `BETWEEN` includes the range bounds).

Sharpen your SQL skills

Write a query that shows all top 100 songs from January 1, 1985 through December 31, 1990.

Try it out | See the answer

SQL IS NULL

Check out the beginning.

The IS NULL operator

`IS NULL` is a logical operator in SQL that allows you to exclude rows with missing data from your results.

Some tables contain null values—cells with no data in them at all. This can be confusing for heavy Excel users, because the difference between a cell having no data and a cell containing a space isn't meaningful in Excel. In SQL, the implications can be pretty serious. This is covered in greater detail in the intermediate tutorial, but for now, here's what you need to know:

You can select rows that contain no data in a given column by using `IS NULL`. Let's try it out using a dataset from the Billboard Music Charts.

```
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.

Sharpen your SQL skills

Write a query that shows all of the rows for which `song_name` is null.

Try it out | See the answer

SQL AND

Check out the beginning.

The SQL AND operator

`AND` is a logical operator in SQL that allows you to select only rows that satisfy two conditions. Using data from the Billboard Music Charts, 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
```

You can use SQL's `AND` operator with additional `AND` statements or any other comparison operator, as many times as you want. If you run this query, you'll notice that all of the requirements are satisfied.

```
SELECT *
  FROM tutorial.billboard_top_100_year_end
 WHERE year = 2012
   AND year_rank <= 10
   AND "group" ILIKE '%feat%'
```

You can see that this example is spaced out onto multiple lines—a good way to make long `WHERE` clauses more readable.

Sharpen your SQL skills

Write a query that surfaces all rows for top-10 hits for which Ludacris is part of the Group.

Try it out | See the answer

Write a query that surfaces the top-ranked records in 1990, 2000, and 2010

Try it out | See the answer

Write a query that lists all songs from the 1960s with "love" in the title.

Try it out | See the answer

SQL OR

Check out the beginning.

The SQL OR operator

`OR` is a logical operator in SQL that allows you to select rows that satisfy either of two conditions. It works the same way as `AND`, which selects the rows that satisfy both of two conditions. Try `OR` out by running this query against data from the Billboard Music Charts:

```
SELECT *
  FROM tutorial.billboard_top_100_year_end
 WHERE year_rank = 5 OR artist = 'Gotye'
```

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 bothof the following conditions:

```
SELECT *
  FROM tutorial.billboard_top_100_year_end
 WHERE year = 2013
   AND ("group" ILIKE '%macklemore%' OR "group" ILIKE '%timberlake%')
```

You will notice that the conditional statement `year = 2013` will be fulfilled for every row returned. In this case, `OR` is treated like one separate conditional statement because it's in parentheses, so it must be satisfied in addition to the first statement of `year = 2013`. You can think of the rows selected as being either of the following:

- Rows where `year = 2013` is true and `"group" ILIKE '%macklemore%'` is true
- Rows where `year = 2013` is true and `"group" ILIKE '%timberlake%'` is true
- Rows where `year = 2013` is true and `"group" ILIKE '%macklemore%'` is true and `"group" ILIKE '%timberlake%'` is true

Sharpen your SQL skills

Write a query that returns all rows for top-10 songs that featured either Katy Perry or Bon Jovi.

Try it out | See the answer

Write a query that returns all songs with titles that contain the word "California" in either the 1970s or 1990s.

Try it out | See the answer

Write a query that lists all top-100 recordings that feature Dr. Dre before 2001 or after 2009.

Try it out | See the answer

SQL NOT

Check out the beginning.

The SQL NOT operator

`NOT` is a logical operator in SQL that you can put before any conditional statement to select rows for which that statement is false.

Here's what `NOT` looks like in action in a query of Billboard Music Charts data:

```
SELECT *
  FROM tutorial.billboard_top_100_year_end
 WHERE year = 2013
   AND year_rank NOT BETWEEN 2 AND 3
```

In the above case, you can see that results for which `year_rank` is equal to 2 or 3 are not included.

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%'
```

`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:

```
SELECT *
  FROM tutorial.billboard_top_100_year_end
 WHERE year = 2013
   AND artist IS NOT NULL
```

Sharpen your SQL skills

Write a query that returns all rows for songs that were on the charts in 2013 and do not contain the letter "a".

Try it out | See the answer

SQL ORDER BY

Check out the beginning.

This lesson uses data from the Billboard Music Charts. Learn more about the dataset.

Sorting data with SQL ORDER BY

Once you've learned how to filter data, it's time to learn how to sort data. The `ORDER BY` clause allows you to reorder your results based on the data in one or more columns. First, take a look at how the table is ordered by default:

```
SELECT * FROM tutorial.billboard_top_100_year_end
```

Now let's see what happens when we order by one of the columns:

```
SELECT *
  FROM tutorial.billboard_top_100_year_end
 ORDER BY artist
```

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, with each successive row having a higher numerical value than the previous. Here's an example using a numerical column:

```
SELECT *
  FROM tutorial.billboard_top_100_year_end
 WHERE year = 2013
 ORDER BY year_rank
```

If you'd like your results in the opposite order (referred to as descending order), you need to add the `DESC` operator:

```
SELECT *
  FROM tutorial.billboard_top_100_year_end
 WHERE year = 2013
 ORDER BY year_rank DESC
```

Write a query that returns all rows from 2012, ordered by song title from Z to A.

Try it out | See the answer

Ordering data by multiple columns

You can also order by mutiple columns. This is particularly useful if your data falls into categories and you'd like to organize rows by date, for example, but keep all of the results within a given category together. 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
```

You can see a couple things from the above query: 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. Finally, the results are sorted by the first column mentioned (`year`), then by `year_rank` afterward. You can see the difference the order makes by running the following query:

```
SELECT *
  FROM tutorial.billboard_top_100_year_end
 WHERE year_rank <= 3
 ORDER BY year_rank, year DESC
```

Finally, 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. For example, the following query is exactly equivalent to the previous query:

```
SELECT *
  FROM tutorial.billboard_top_100_year_end
 WHERE year_rank <= 3
 ORDER BY 2, 1 DESC
```

Note: this functionality (numbering columns instead of using names) is supported by Mode, but not by every flavor of SQL, so if you're using another system or connected to certain types of databases, it may not work.

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, so if you were to order by `year_rank`, for example, you can be sure that you are getting the lowest values of `year_rank` in the entire table, not just in the first 100 rows of the table.

Write a query that returns all rows from 2010 ordered by rank, with artists ordered alphabetically for each song.

Try it out | See the answer

Using comments

You can “comment out” pieces of code by adding combinations of characters. In other words, you can specify parts of your query that will not actually be treated like SQL code. It can be helpful to include comments that explain your thinking so that you can easily remember what you intended to do if you ever want to revisit your work. Commenting can also be useful if you want to test variations on your query while keeping all of your code intact.

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
```

Sharpen your SQL skills

Write a query that shows all rows for which T-Pain was a group member, ordered by rank on the charts, from lowest to highest rank (from 100 to 1).

Try it out | See the answer

Write a query that returns songs that ranked between 10 and 20 (inclusive) in 1993, 2003, or 2013. Order the results by year and rank, and leave a comment on each line of the `WHERE` clause to indicate what that line does

Try it out | See the answer

What’s next?

Congrats on completing the Basic SQL tutorial!

You may find that your skills are limiting, though. If you want to do things like aggregate data across entire columns or merge multiple datasets together, check out the Intermediate SQL section of this tutorial.

SQL Aggregate Functions

The Intermediate SQL Tutorial

Welcome to the Intermediate SQL Tutorial! If you skipped the Basic SQL Tutorial, you should take a quick peek at this page to get an idea of how to use Mode's SQL editor to get the most out of this tutorial. For convenience, here's the gist:

- Open another window to Mode. Sign up for an account if you don't have one.
- For each lesson, start by running `SELECT *` on the relevant dataset so you get a sense of what the raw data looks like. Do this in that window you just opened to Mode.
- Run all of the code blocks in the lesson in Mode in the other window. You'll learn more if you really examine the results and understand what the code is doing.

In the previous tutorial, many of the practice problems could only be solved in one or two ways with the skills you learned. As you progress and problems get harder, there will be many ways of producing the correct results. Keep in mind that the answers to practice problems should be used as a reference, but are by no means the only ways of answering the questions.

The Apple stock prices dataset

For the first few lessons, you'll be working with Apple stock price data. The data was pulled from Google Finance in January 2014. There's one row for each day (indicated in the `date` field). `open` and `close` are the opening and closing prices of the stock on that day. `high` and `low` are the high and low prices for that day. `volume` is the number of shares traded on that day. Some data has been intentionally removed for the sake of this lesson. Check it out for yourself:

```
SELECT * FROM tutorial.aapl_historical_stock_price
```

Aggregate functions in SQL

As the Basic SQL Tutorial points out, SQL is excellent at aggregating data the way you might in a pivot table in Excel. You will use aggregate functions all the time, so it's important to get comfortable with them. The functions themselves are the same ones you will find in Excel or any other analytics program. We'll cover them individually in the next few lessons. Here's a quick preview:

- `COUNT` counts how many rows are in a particular column.
- `SUM` adds together all the values in a particular column.

- `MIN` and `MAX` return the lowest and highest values in a particular column, respectively.
- `AVG` calculates the average of a group of selected values.

The Basic SQL Tutorial also pointed out that arithmetic operators only perform operations across rows. Aggregate functions are used to perform operations across entire columns (which could include millions of rows of data or more).

NEXT TUTORIAL

SQL COUNT

Check out the beginning.

Counting all rows

`COUNT` is a SQL aggregate function for counting the number of rows in a particular column. `COUNT` is the easiest aggregate function to begin with because verifying your results is extremely simple. Let's begin by using `*` to select all rows from the Apple stock prices dataset:

```
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.

You can see that the result showed a count of all rows to be 3555. To make sure that's right, turn off Mode's automatic limit by unchecking the box next to "Limit 100" next to the "Run" button in Mode's SQL Editor. Then run the following query:

```
SELECT * FROM tutorial.aapl_historical_stock_price
```

Note that Mode actually provides a count of the total rows returned (above the results table), which should be the same as the result of using the `COUNT` function in the above query.

Counting individual columns

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'll notice that this result is lower than what you got with `COUNT(*)`. That's because `high` has some nulls. In this case, we've deleted some data to make the lesson interesting, but analysts often run into naturally-occurring null rows.

For example, imagine you've got a table with one column showing email addresses for everyone you sent a marketing email to, and another column showing the date and time that each person opened the email. If someone didn't open the email, the date/time field would likely be null.

Write a query to count the number of non-null rows in the `low` column.

Try it out | See the answer

Counting non-numerical columns

One nice thing about `COUNT` is that you can use it on non-numerical columns:

```
SELECT COUNT(date)
  FROM tutorial.aapl_historical_stock_price
```

The above query returns the same result as the previous: 3555. It's hard to tell because each row has a different `date` value, but `COUNT` simply counts the total number of non-null rows, not the distinct values. Counting the number of distinct values in a column is discussed in a later tutorial.

You might have also noticed that the column header in the results just reads "count." We recommend naming your columns so that they make a little more sense to anyone else who views your work. As mentioned in an earlier lesson, it's best to use lower case letters and underscores. You can add column names (also called *aliases*) using `AS`:

```
SELECT COUNT(date) AS count_of_date
  FROM tutorial.aapl_historical_stock_price
```

If you must use spaces, you will need to use double quotes.

```
SELECT COUNT(date) AS "Count Of Date"
  FROM tutorial.aapl_historical_stock_price
```

Note: This is really the only place in which you'll ever want to use double quotes in SQL. Single quotes for everything else.

Sharpen your SQL skills

Write a query that determines counts of every single column. Which column has the most null values?

Try it out | See the answer

SQL SUM

Check out the beginning.

The SQL SUM function

`SUM` is a SQL aggregate function that totals the values in a given column. Unlike `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:

```
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.

You don't need to worry as much about the presence of nulls with `SUM` as you would with `COUNT`, as `SUM` treats nulls as 0.

Sharpen your SQL skills

Write a query to calculate the average opening price (hint: you will need to use both `COUNT` and `SUM`, as well as some simple arithmetic.).

Try it out | See the answer

SQL MIN/MAX

Check out the beginning.

The SQL MIN and MAX functions

`MIN` and `MAX` are SQL aggregation functions that return the lowest and highest values in a particular column.

They're similar to `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."

For example, the following query selects the `MIN` and the `MAX` from the numerical `volume` column in the Apple stock prices dataset.

```
SELECT MIN(volume) AS min_volume,
       MAX(volume) AS max_volume
  FROM tutorial.aapl_historical_stock_price
```

Sharpen your SQL skills

What was Apple's lowest stock price (at the time of this data collection)?

Try it out | See the answer

What was the highest single-day increase in Apple's share value?

Try it out | See the answer

NEXT TUTORIAL

SQL AVG

Check out the beginning.

The SQL AVG function

`AVG` is a SQL aggregate function 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. You can see this by comparing these two queries of the Apple stock prices dataset:

```
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
```

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 (covered in a later lesson).

Sharpen your SQL skills

Write a query that calculates the average daily trade volume for Apple stock.

Try it out | See the answer

SQL GROUP BY

Check out the beginning.

The SQL GROUP BY clause

SQL aggregate functions 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. Here's an example using the Apple stock prices dataset:

```
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`:

```
SELECT year,
       month,
       COUNT(*) AS count
  FROM tutorial.aapl_historical_stock_price
 GROUP BY year, month
```

Calculate the total number of shares traded each month. Order your results chronologically.

Try it out | See the answer

GROUP BY column numbers

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
```

Note: this functionality (numbering columns instead of using names) is supported by Mode, but not by every flavor of SQL, so if you're using another system or connected to certain types of databases, it may not work.

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`. Try running the query below, then reverse the column names in the `ORDER BY` statement and see how it looks:

```
SELECT year,
       month,
       COUNT(*) AS count
  FROM tutorial.aapl_historical_stock_price
 GROUP BY year, month
 ORDER BY month, year
```

Sharpen your SQL skills

Write a query to calculate the average daily price change in Apple stock, grouped by year.

Try it out | See the answer

Write a query that calculates the lowest and highest prices that Apple stock achieved each month.

Try it out | See the answer

SQL HAVING

Check out the beginning.

The SQL HAVING clause

In the previous lesson, you learned how to use the `GROUP BY` clause to aggregate stats from the Apple stock prices dataset by month and year.

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.
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, which you will learn about in a later lesson.

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

Check out the beginning.

Using SQL 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, 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.

Write a query that returns the unique values in the `year` column, in chronological order.

Try it out | See the answer

`DISTINCT` can be particularly helpful when exploring a new data set. In many real-world scenarios, you will generally end up writing several preliminary queries in order to figure out the best approach to answering your initial question. Looking at the unique values on each column can help identify how you might want to group or filter the data.

Using DISTINCT in aggregations

You can use `DISTINCT` when performing an aggregation. You'll probably use it most commonly with the `COUNT` function.

In this case, you should run the query below that counts the unique values in the `month` column.

```
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). That's a small enough number that you might be able to aggregate by month and interpret the results fairly early. For example, 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 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.

DISTINCT performance

It's worth noting that using `DISTINCT`, particularly in aggregations, can slow your queries down quite a bit. We'll cover this in greater depth in a later lesson.

Sharpen your SQL skills

Write a query that counts the number of unique values in the `month` column for each year.

Try it out | See the answer

Write a query that separately counts the number of unique values in the `month` column and the number of unique values in the `year` column.

Try it out | See the answer

SQL CASE

Check out the beginning.

For the next few lessons, you'll work with data on College Football Players. This data was collected from ESPN on January 15, 2014 from the rosters listed on this pageusing a Python scraper available here. In this particular lesson, you'll stick to roster information. This table is pretty self-explanatory—one row per player, with columns that describe attributes for that player. Run this query to check out the raw data:

```
SELECT * FROM benn.college_football_players
```

The SQL CASE statement

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:

```
SELECT player_name,
       year,
       CASE WHEN year = 'SR' THEN 'yes'
            ELSE NULL END AS is_a_senior
  FROM benn.college_football_players
```

In plain English, here's what's happening:

1. The `CASE` statement checks each row to see if the conditional statement—`year = 'SR'` is true.
2. For any given row, if that conditional statement is true, the word "yes" gets printed in the column that we have named `is_a_senior`.
3. In any row for which the conditional statement is false, nothing happens in that row, leaving a null value in the `is_a_senior` column.
4. At the same time all this is happening, SQL is retrieving and displaying all the values in the `player_name` and `year` columns.

The above query makes it pretty easy to see what's happening because we've included the `CASE` statement along with the `year` column itself. You can check each

row to see whether `year` meets the condition `year = 'SR'` and then see the result in the column generated using the `CASE` statement.

But 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
```

Write a query that includes a column that is flagged "yes" when a player is from California, and sort the results with those players first.

Try it out | See the answer

Adding multiple conditions to a CASE statement

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
```

In the above example, the `WHEN`/`THEN` statements will get evaluated in the order that they're written. So if the value in the `weight` column of a given row is 300, it will produce a result of "over 250." Here's what happens if the value in the `weight` column is 180, SQL will do the following:

1. Check to see if `weight` is greater than 250. 180 is not greater than 250, so move on to the next `WHEN`/`THEN`
2. Check to see if `weight` is greater than 200. 180 is not greater than 200, so move on to the next `WHEN`/`THEN`
3. Check to see if `weight` is greater than 175. 180 is greater than 175, so record "175-200" in the `weight_group` column.

While the above works, it's really best practice to create statements that don't overlap. `WHEN weight > 250` and `WHEN weight > 200` overlap for every value greater than 250, which is a little confusing. 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
```

Write a query that includes players' names and a column that classifies them into four categories based on height. Keep in mind that the answer we provide is only one of many possible answers, since you could divide players' heights in many ways.

Try it out | See the answer

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`, `THEN`, and `END`. `ELSE` is an optional component.
3. You can make any conditional statement using any conditional operator (like `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 an `ELSE` statement to deal with any unaddressed conditions.

Write a query that selects all columns from `benn.college_football_players` and adds an additional column that displays the player's name if that player is a junior or senior.

Try it out | See the answer

Using CASE with aggregate functions

`CASE`'s slightly more complicated and substantially more useful functionality comes from pairing it with aggregate functions. For example, let's say you want to only count rows that fulfill a certain condition. Since `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
```

The above query is an excellent place to use numbers instead of columns in the `GROUP BY` clause because repeating the `CASE` statement in the `GROUP BY` clause would make the query obnoxiously long. Alternatively, you can use the column’s alias in the `GROUP BY` clause like this:

```
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 year_group
```

Note that if you do choose to repeat the entire `CASE` statement, you should remove the `AS year_group` column naming when you copy/paste into the `GROUP BY` clause:

```
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 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
```

Combining `CASE` statements with aggregations can be tricky at first. It's often helpful to write a query containing the `CASE` statement first and run it on its own. Using the previous example, you might first write:

```
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,
            *
  FROM benn.college_football_players
```

The above query will show all columns in the `benn.college_football_players` table, as well as a column showing the results of the `CASE` statement. From there, you can replace the `*` with an aggregation and add a `GROUP BY` clause. Try this process if you struggle with either of the following practice problems.

Write a query that counts the number of 300lb+ players for each of the following regions: West Coast (CA, OR, WA), Texas, and Other (Everywhere else).
Try it out See the answer

Write a query that calculates the combined weight of all underclass players (FR/SO) in California as well as the combined weight of all upperclass players (JR/SR) in California.
Try it out See the answer

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 in Excel). Let's take the following 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
```

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
```

It's worth noting that going from horizontal to vertical orientation can be a substantially more difficult problem depending on the circumstances, and is covered in greater depth in a later lesson.

Sharpen your SQL skills

Write a query that displays the number of players in each state, with FR, SO, JR, and SR players in separate columns and another column for the total number of players. Order results such that states with the most players come first.

Try it out | See the answer

Write a query that shows the number of players at schools with names that start with A through M, and the number at schools with names starting with N - Z.

Try it out | See the answer

SQL Joins

Check out the beginning.

Intro to SQL joins: relational concepts

Up to this point, we've only been working with one table at a time. The real power of SQL, however, comes from working with data from multiple tables at once. If you remember from a previous lesson, the tables you've been working with up to this point are all part of the same schema in a relational database. The term "relational database" refers to the fact that the tables within it "relate" to one another—they contain common identifiers that allow information from multiple tables to be combined easily.

To understand what joins are and why they are helpful, let's think about Twitter.

Twitter has to store a lot of data. Twitter could (hypothetically, of course) store its data in one big table in which each row represents one tweet. There could be one column for the content of each tweet, one for the time of the tweet, one for the person who tweeted it, and so on. It turns out, though, that identifying the person who tweeted is a little tricky. There's a lot to a person's Twitter identity—a username, a bio, followers, followees, and more. Twitter could store all of that data in a table like this:

Let's say, for the sake of argument, that Twitter did structure their data this way. Every time you tweet, Twitter creates a new row in its database, with information about you and the tweet.

But this creates a problem. When you update your bio, Twitter would have to change that information for every one of your tweets in this table. If you've tweeted 5,000 times, that means 5,000 changes. If many people on Twitter are making lots of changes at once, that's a lot of computation to support. Instead, it's much easier for Twitter to store everyone's profile information in a separate table. That way, whenever someone updates their bio, Twitter would only have to change one row of data instead of thousands.

In an organization like this, Twitter now has two tables. The first table—the users table—contains profile information, and has one row per user. The second table—the tweets table—contains tweet information, including the username of the person who sent the tweet. By matching—or *joining*—that username in the tweets table to the username in the users table, Twitter can still connect profile information to every tweet.

The anatomy of a join

Unfortunately, we can't use Twitter's data in any working examples (for that, we'll have to wait for the NSA's SQL Tutorial), but we can look at a similar problem.

In the previous lesson on conditional logic, we worked with a table of data on college football players—`benn.college_football_players`. This table included data on players, including each player's weight and the school that they played for. However, it didn't include much information on the school, such as the conference the school is in—that information is in a separate table, `benn.college_football_teams`.

Let's say we want to figure out which conference has the highest average weight. Given that information is in two separate tables, how do you do that? A join!

```
SELECT teams.conference AS conference,
       AVG(players.weight) AS average_weight
  FROM benn.college_football_players players
  JOIN benn.college_football_teams teams
    ON teams.school_name = players.school_name
 GROUP BY teams.conference
 ORDER BY AVG(players.weight) DESC
```

There's a lot of new stuff happening here, so we'll go step-by-step.

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 alias 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.

Write a query that selects the school name, player name, position, and weight for every player in Georgia, ordered by weight (heaviest to lightest). Be sure to make an alias for the table, and to reference all column names in relation to the alias.

Try it out | See the answer

JOIN and ON

After the `FROM` statement, we have two new statements: `JOIN`, which is followed by a table name, and `ON`, which is followed by a couple column names separated by an equals sign.

Though the `ON` statement comes after `JOIN`, it's a bit easier to explain it first. `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:

```
ON teams.school_name = players.school_name
```

In plain English, this means:

Join all rows from the `players` table on to rows in the `teams` table for which the `school_name` field in the `players` table is equal to the `school_name` field in the `teams` table.

What does this actually do? Let's take a look at one row to see what happens. This is the row in the players table for Wake Forest wide receiver Michael Campanaro:

During the join, SQL looks up the `school_name`—in this case, "Wake Forest"—in the `school_name` field of the `teams` table. If there's a match, SQL takes all five columns from the `teams` table and joins them to ten columns of the `players` table. The new result is a fifteen column table, and the row with Michael Campanaro looks like this:

When you run a query with a join, SQL performs the same operation as it did above to every row of the table after the `FROM` statement. To see the full table returned by the join, try running this query:

```
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.

Once you've generated this new table after the join, you can use the same aggregate functions from a previous lesson. By running an `AVG` function on player weights, and grouping by the `conference` field from the teams table, you can figure out each conference's average weight.

SQL INNER JOIN

Check out the beginning.

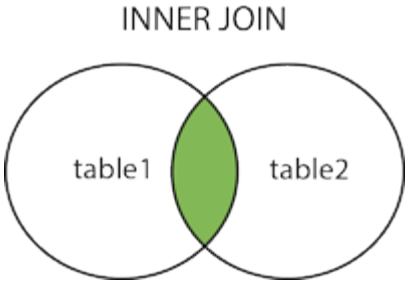
INNER JOIN

In the previous lesson, you learned the basics of SQL joins using a data about college football players. All of the players in the `players` table match to one school in the `teams` table. But what if the data isn't so clean? What if there are multiple schools in the `teams` table with the same name? Or if a player goes to a school that isn't in the `teams` table?

If there are multiple schools in the `teams` table with the same name, each one of those rows will get joined to matching rows in the `players` table. Returning to the previous example with Michael Campanaro, if there were three rows in the `teams`table where `school_name = 'Wake Forest'`, the join query above would return three rows with Michael Campanaro.

It's often the case that one or both tables being joined contain rows that don't have matches in the other table. The way this is handled depends on whether you're making an inner join or an outer 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.

Joining tables with identical column names

When you join two tables, it might be the case that both tables have columns with identical 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
```

Sharpen your SQL skills

Write a query that displays player names, school names and conferences for schools in the "FBS (Division I-A Teams)" division.

Try it out | See the answer

SQL Outer Joins

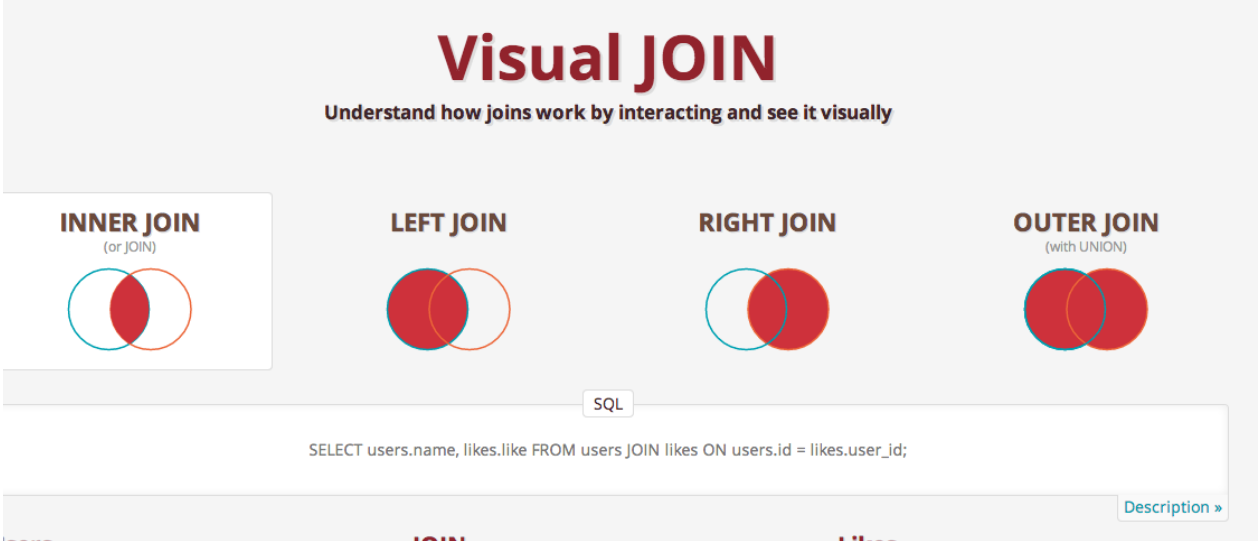
Check out the beginning.

Outer joins

When performing an inner join, rows from either table that are unmatched in the other table are not returned. In an outer join, unmatched rows in one or both tables can be returned. There are a few types of outer joins:

- `LEFT JOIN` returns only unmatched rows from the left table.
- `RIGHT JOIN` returns only unmatched rows from the right table.
- `FULL OUTER JOIN` returns unmatched rows from both tables.

As you work through the following lessons about outer joins, it might be helpful to refer to this JOIN visualization by Patrik Spathon.



The Crunchbase dataset

The data for the following lessons was pulled from Crunchbase, a crowdsourced index of startups, founders, investors, and the activities of all three. It was collected Feb. 5, 2014, and large portions of both tables were randomly dropped for the sake of this lesson. The first table lists a large portion of companies in the database; one row per company. The `permalink` field is a unique identifier for each row, and also shows the web address. For each company in the table, you can view its online Crunchbase profile by copying/pasting its permalink after Crunchbase's web domain. For example, the third company in the table, ".Club Domains," has the permalink "/company/club-domains," so its profile address would be http://www.crunchbase.com/company/club-domains. The fields with "funding" in the name have to do with how much outside investment (in USD) each company has taken on. The rest of the fields are self-explanatory.

```
SELECT *
  FROM tutorial.crunchbase_companies
```

The second table lists acquisitions—one row per acquisition. `company_permalink` in this table maps to the `permalink` field in `tutorial.crunchbase_companies` as described in the previous lesson. Joining these two fields will add information about the company being acquired.

You'll notice that there is a separate field called `acquirer_permalink` as well. This can also be mapped to the `permalink` field `tutorial.crunchbase_companies` to add additional information about the acquiring company.

```
SELECT *
  FROM tutorial.crunchbase_acquisitions
```

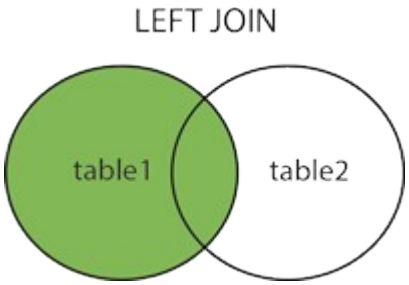
The foreign key you use to join these two tables will depend entirely on whether you're looking to add information about the acquiring company or the company that was acquired.

It's worth noting that this sort of structure is common. For example, a table showing a list of emails sent might include a `sender_email_address` and a `recipient_email_address`, both of which map to a table listing email addresses and the names of their owners.

SQL LEFT JOIN

Check out the beginning.

The LEFT JOIN command



Let's start by running an `INNER JOIN` on the Crunchbase dataset and taking a look at the results. We'll just look at `company-permalink` in each table, as well as a couple other fields, to get a sense of what's actually being joined.

```
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.

Sharpen your SQL skills

You can explore the differences between a `LEFT JOIN` and a `JOIN` by solving these practice problems:

Write a query that performs an inner join between the `tutorial.crunchbase_acquisitions` table and the `tutorial.crunchbase_companies` table, but instead of listing individual rows, count the number of non-null rows in each table.

Try it out | See the answer

Modify the query above to be a `LEFT JOIN`. Note the difference in results.

Try it out | See the answer

Now that you've got a sense of how left joins work, try this harder aggregation problem:

Count the number of unique companies (don't double-count companies) and unique *acquired* companies by state. Do not include results for which there is no state data, and order by the number of acquired companies from highest to lowest.

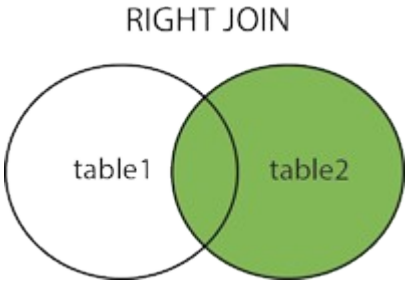
Try it out | See the answer

SQL RIGHT JOIN

Check out the beginning.

The RIGHT JOIN command

Right joins are similar to left joins except they return all rows from the table in the `RIGHT JOIN` clause and only matching rows from the table in the `FROM` clause.



`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`. For example, in this query of the Crunchbase dataset, the `LEFT JOIN` section:

```
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
```

produces the same results as this query:

```
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_acquisitions acquisitions
 RIGHT JOIN tutorial.crunchbase_companies companies
    ON companies.permalink = acquisitions.company_permalink
```

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.

Sharpen your SQL skills

Rewrite the previous practice query in which you counted total and acquired companies by state, but with a `RIGHT JOIN` instead of a `LEFT JOIN`. The goal is to produce the exact same results.

Try it out | See the answer

NEXT TUTORIAL

SQL Joins Using WHERE or ON

Check out the beginning.

Filtering in the ON clause

Normally, filtering is processed in the `WHERE` clause once the two tables have already been joined. It's possible, though that you might want to filter one or both of the tables *before* joining them. For example, you only want to create matches between the tables under certain circumstances.

Using Crunchbase data, let's take another look at the `LEFT JOIN` example from an earlier lesson (this time we'll add an `ORDER BY` clause):

```
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
 ORDER BY 1
```

Compare the following query to the previous one and you will see that everything in the `tutorial.crunchbase_acquisitions` table was joined on except for the row for which `company_permalink` is `'/company/1000memories'`:

```
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

If you move the same filter to the `WHERE` clause, you will notice that the filter happens after the tables are joined. 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.

Sharpen your SQL skills

For this set of practice problems, we're going to introduce a new dataset: `tutorial.crunchbase_investments`. This table is also sourced from Crunchbase and contains much of the same information as the `tutorial.crunchbase_companies` data. It it structured differently, though: it contains one row per *investment*. There can be multiple investments per company—it's even possible that one investor could invest in the same company multiple times. The column names are pretty self-explanatory. What's important is that `company_permalink` in the `tutorial.crunchbase_investments`table maps to `permalink` in the `tutorial.crunchbase_companies` table. Keep in mind that some random data has been removed from this table for the sake of this lesson.

It is very likely that you will need to do some exploratory analysis on this table to understand how you might solve the following problems.

Write a query that shows a company's name, "status" (found in the Companies table), and the number of unique investors in that company. Order by the number of investors from most to fewest. Limit to only companies in the state of New York.

Try it out | See the answer

Write a query that lists investors based on the number of companies in which they are invested. Include a row for companies with no investor, and order from most companies to least.

Try it out See the answer

SQL FULL OUTER JOIN

Check out the beginning.

The SQL FULL JOIN command

You're not likely to use `FULL JOIN` (which can also be written as `FULL OUTER JOIN`) too often, but it's worth covering anyway. `LEFT JOIN` and `RIGHT JOIN` each return unmatched rows from one of the tables—`FULL JOIN` returns unmatched rows from both tables. It is commonly used in conjunction with aggregations to understand the amount of overlap between two tables.

Here's an example using the Crunchbase companies and acquisitions 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.

You might also notice that surprisingly few rows in the `crunchbase_acquisitions` table were matched to the `crunchbase_companies` table. If this were a real assignment, you'd probably want to look at some individual rows to get a sense of why some of them weren't matched and whether or not you should consider finding more/better data.

Sharpen your SQL skills

This practice problem uses Crunchbase investment data described in a previous lesson. The Crunchbase Investments table has been split into two parts for the sake of this exercise.

Write a query that joins `tutorial.crunchbase_companies` and `tutorial.crunchbase_investments_part1` using a `FULL JOIN`. Count up the number of rows that are matched/unmatched as in the example above.

Try it out | See the answer

SQL UNION

Check out the beginning.

The SQL UNION operator

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.

Let's try it out with the Crunchbase investment data, which has been split into two tables for the purposes of this lesson. The following query will display all results from the first portion of the query, then all results from the second portion in the same table:

```
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 `UNION` involve 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.

Sharpen your SQL skills

Write a query that appends the two `crunchbase_investments` datasets above (including duplicate values). Filter the first dataset to only companies with names that start with the letter "T", and filter the second to companies with names starting with "M" (both not case-sensitive). Only include the `company_permalink`, `company_name`, and `investor_name` columns.

Try it out | See the answer

For a bit more of a challenge:

Write a query that shows 3 columns. The first indicates which dataset (part 1 or 2) the data comes from, the second shows company status, and the third is a count of the number of investors.

Hint: you will have to use the `tutorial.crunchbase_companies` table as well as the investments tables. And you'll want to group by status and dataset.

Try it out | See the answer

SQL Joins with Comparison Operators

Check out the beginning.

This lesson uses the same data from previous lessons, which was pulled from Crunchbase on Feb. 5, 2014. Learn more about this dataset.

Using comparison operators with joins

In the lessons so far, you've only joined tables 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
```

For more on these differences, revisit the lesson SQL Joins Using WHERE or ON.

SQL Joins on Multiple Keys

Check out the beginning.

This lesson uses the same data from previous lessons, which was pulled from Crunchbase on Feb. 5, 2014. Learn more about this dataset.

Joining on multiple keys

There are couple reasons you might want to join tables 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, 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

Check out the beginning.

This lesson uses the same data from previous lessons, which was pulled from Crunchbase on Feb. 5, 2014. Learn more about this dataset.

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`.

Also, keep in mind as you review the results from the above query that a large part of the data has been omitted for the sake of the lesson (much of it is in the `tutorial.crunchbase_investments_part2` table).

What's next?

Congratulations, you've learned most of the technical stuff you need to know to analyze data using SQL. The Advanced SQL Tutorial covers a few more necessities (an in-depth lesson on data types, for example), as well as some more technical features that will greatly extend the tools you've already learned.

SQL Data Types

Welcome to the Advanced SQL Tutorial! If you skipped the beginning tutorials, you should take a quick peek at this page to get an idea of how to get the most out of this tutorial. For convenience, here's the gist:

- Open another window to Mode. Sign up for an account if you don't have one.
- For each lesson, start by running `SELECT *` on the relevant dataset so you get a sense of what the raw data looks like. Do this in that window you just opened to Mode.
- Run all of the code blocks in the lesson in Mode in the other window. You'll learn more if you really examine the results and understand what the code is doing.

For this lesson, we'll use the same Crunchbase data from a previous lesson. It was collected on Feb 15, 2014, and large portions of the data were dropped for the sake of this tutorial. In this example, we'll also use a modified version of this data with date formats cleaned up to work better with SQL.

Data types

In previous lessons, you learned that certain functions work on some data types, but not others. For example, `COUNT` works with any data type, but `SUM` only works for numerical data (if this doesn't sound familiar, you should revisit this lesson). This is actually more complicated than it appears: in order to use `SUM`, the data must appear to be numeric, but it must also be stored in the database in a numeric form.

You might run into this, for example, if you have a column that appears to be entirely numeric, but happens to contain spaces or commas. Yes, it turns out that numeric columns cannot contain commas—If you upload data to Mode with commas in a column full of numbers, Mode will treat that column as non-numeric. Generally, numeric column types in various SQL databases *do not* support commas or currency symbols. To make things more complicated, SQL databases can store data in many different formats with different levels of precision.

The `INTEGER` data type, for example, only stores whole numbers—no decimals. The `DOUBLE PRECISION` data type, on the other hand, can store between 15 and 17 significant decimal digits (almost certainly more than you need unless you're a physicist). There are a lot of data types, so it doesn't make sense to list them all here. For the complete list, click here.

Here is the list of exact data types stored in Mode:

Imported as	Stored as	With these rules

String	VARCHAR(1024)	Any characters, with a maximum field length of 1024 characters.
Date/Time	TIMESTAMP	Stores year, month, day, hour, minute and second values as YYYY-MM-DD hh:mm:ss.
Number	DOUBLE PRECISION	Numerical, with up to 17 significant digits decimal precision.
Boolean	BOOLEAN	Only TRUE or FALSE values.

“Imported as” refers to the types that is selected in the import flow (see image below), “Stored as” refers to the official SQL data type, and the third column explains the rules associated with the SQL data type.

Changing a column’s data type

It’s certainly best for data to be stored in its optimal format from the beginning, but if it isn’t, you can always change it in your query. It’s particularly common for dates or numbers, for example, to be stored as strings. This becomes problematic when you want to sum a column and you get an error because SQL is reading numbers as strings. When this happens, you can use `CAST` or `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.

You could replace `integer` with any other data type that would make sense for that column—all values in a given column must fit with the new data types.

Mode Community (the site you’re using to complete this tutorial) performs implicit conversion in certain circumstances, so data types are rarely likely to be problematic. However, if you’re accessing an internal database (your employer’s, for example), you may need to be careful about managing data types for some functions.

Sharpen your SQL skills

Convert the `funding_total_usd` and `founded_at_clean` columns in the `tutorial.crunchbase_companies_clean_date` table to strings (varchar format) using a different formatting function for each one.

Try it out | See the answer

SQL Date Format

Check out the beginning.

This lesson uses the same data from previous lessons, which was pulled from Crunchbase on Feb. 5, 2014. Learn more about this dataset.

Why dates are formatted year-first

If you live in the United States, you're probably used to seeing dates formatted as MM-DD-YYYY or a similar, month-first format. It's an odd convention compared to the rest of the world's standards, but it's not necessarily any worse than DD-MM-YYYY. The problem with both of these formats is that when they are stored as strings, they don't sort in chronological order. For example, here's a date field stored as a string. Because the month is listed first, the `ORDER BY` statement doesn't produce a chronological list:

```
SELECT permalink,
       founded_at
  FROM tutorial.crunchbase_companies_clean_date
 ORDER BY founded_at
```

You might think that converting these values from `string` to `date` might solve the problem, but it's actually not quite so simple. Mode (and most relational databases) format dates as YYYY-MM-DD, a format that makes a lot of sense because it will sort in the same order whether it's stored as a date or as a string. Excel is notorious for producing date formats that don't play nicely with other systems, so if you're exporting Excel files to CSV and uploading them to Mode, you may run into this a lot.

Here's an example from the same table, but with a field that has a cleaned date. Note that the cleaned date field is actually stored as a string, but still sorts in chronological order anyway:

```
SELECT permalink,
       founded_at,
       founded_at_clean
  FROM tutorial.crunchbase_companies_clean_date
 ORDER BY founded_at_clean
```

The lesson on data cleaning provides some examples for converting poorly formatted dates into proper date-formatted fields.

Crazy rules for dates and times

Assuming you've got some dates properly stored as a `date` or `time` data type, you can do some pretty powerful things. Maybe you'd like to calculate a field of dates a week after an existing field. Or maybe you'd like to create a field that indicates how many days apart the values in two other date fields are. These are trivially simple, but it's important to keep in mind that the data type of your results will depend on exactly what you are doing to the dates.

When you perform arithmetic on dates (such as subtracting one date from another), the results are often stored as the `interval` data type—a series of integers that represent a period of time. The following query uses date subtraction to determine how long it took companies to be acquired (unacquired companies and those without dates entered were filtered out). Note that because the `companies.founded_at_clean` column is stored as a string, it must be cast as a timestamp before it can be subtracted from another timestamp.

```
SELECT companies.permalink,
       companies.founded_at_clean,
       acquisitions.acquired_at_cleaned,
       acquisitions.acquired_at_cleaned -
         companies.founded_at_clean::timestamp AS time_to_acquisition
  FROM tutorial.crunchbase_companies_clean_date companies
  JOIN tutorial.crunchbase_acquisitions_clean_date acquisitions
    ON acquisitions.company_permalink = companies.permalink
 WHERE founded_at_clean IS NOT NULL
```

In the example above, you can see that the `time_to_acquisition` column is an interval, not another date.

You can introduce intervals using the `INTERVAL` function as well:

```
SELECT companies.permalink,
       companies.founded_at_clean,
       companies.founded_at_clean::timestamp +
         INTERVAL '1 week' AS plus_one_week
  FROM tutorial.crunchbase_companies_clean_date companies
 WHERE founded_at_clean IS NOT NULL
```

The interval is defined using plain-English terms like '10 seconds' or '5 months'. Also note that adding or subtracting a `date` column and an `interval` column results in another `date` column as in the above query.

You can add the current time (at the time you run the query) into your code using the `NOW()` function:

```
SELECT companies.permalink,
```

```
       companies.founded_at_clean,
       NOW() - companies.founded_at_clean::timestamp AS founded_time_ago
  FROM tutorial.crunchbase_companies_clean_date companies
 WHERE founded_at_clean IS NOT NULL
```

Sharpen your SQL skills

Write a query that counts the number of companies acquired within 3 years, 5 years, and 10 years of being founded (in 3 separate columns). Include a column for total companies acquired as well. Group by category and limit to only rows with a founding date.

Try it out | See the answer

Using SQL String Functions to Clean Data

Check out the beginning.

This lesson features data on San Francisco Crime Incidents for the 3-month period beginning November 1, 2013 and ending January 31, 2014. It was collected from the SF Data website on February 16, 2014. There is one row for each incident reported. Some field definitions: `location` is the GPS location of the incident, listed in decimal degrees, latitude first, longitude second. The two coordinates are also broken out into the `lat` and `lon` fields, respectively.

Start by taking a look:

```
SELECT *
  FROM tutorial.sf_crime_incidents_2014_01
```

Cleaning strings

Most of the functions presented in this lesson are specific to certain data types. However, using a particular function will, in many cases, change the data to the appropriate type. `LEFT`, `RIGHT`, and `TRIM` are all used to select only certain elements of strings, but using them to select elements of a number or date will treat them as strings for the purpose of the function.

LEFT, RIGHT, and LENGTH

Let's start with `LEFT`. You can use `LEFT` to pull a certain number of characters from the left side of a string and present them as a separate string. The syntax is `LEFT(string, number of characters)`.

As a practical example, we can see that the `date` field in this dataset begins with a 10-digit date, and include the timestamp to the right of it. The following query pulls out only the ogimage: "/images/og-images/sql-facebook.png" date:

```
SELECT incidnt_num,
       date,
       LEFT(date, 10) AS cleaned_date
  FROM tutorial.sf_crime_incidents_2014_01
```

`RIGHT` does the same thing, but from the right side:

```
SELECT incidnt_num,
       date,
       LEFT(date, 10) AS cleaned_date,
```

```
       RIGHT(date, 17) AS cleaned_time
  FROM tutorial.sf_crime_incidents_2014_01
```

`RIGHT` works well in this case because we know that the number of characters will be consistent across the entire `date` field. If it wasn't consistent, it's still possible to pull a string from the right side in a way that makes sense. The `LENGTH` function returns the length of a string. So `LENGTH(date)` will always return `28` in this dataset. Since we know that the first 10 characters will be the date, and they will be followed by a space (total 11 characters), we could represent the `RIGHT` function like this:

```
SELECT incidnt_num,
       date,
       LEFT(date, 10) AS cleaned_date,
       RIGHT(date, LENGTH(date) - 11) AS cleaned_time
  FROM tutorial.sf_crime_incidents_2014_01
```

When using functions within other functions, it's important to remember that the innermost functions will be evaluated first, followed by the functions that encapsulate them.

TRIM

The `TRIM` function is used to remove characters from the beginning and end of a string. Here's an example:

```
SELECT location,
       TRIM(both '()' FROM location)
  FROM tutorial.sf_crime_incidents_2014_01
```

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`.

POSITION and STRPOS

`POSITION` allows you to specify a substring, then returns a numerical value equal to the character number (counting from left) where that substring first appears in the target string. For example, the following query will return the position of the character 'A' (case-sensitive) where it first appears in the `descript` field:

```
SELECT incidnt_num,
```

```
       descript,
       POSITION('A' IN descript) AS a_position
  FROM tutorial.sf_crime_incidents_2014_01
```

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:

```
SELECT incidnt_num,
       descript,
       STRPOS(descript, 'A') AS a_position
  FROM tutorial.sf_crime_incidents_2014_01
```

Importantly, both the `POSITION` and `STRPOS` functions are case-sensitive. If you want to look for a character regardless of its case, you can make your entire string a single by using the `UPPER` or `LOWER` functions described below.

SUBSTR

`LEFT` and `RIGHT` both create substrings of a specified length, but they only do so starting from the sides of an existing string. If you want to start in the middle of a string, you can use `SUBSTR`. The syntax is `SUBSTR(*string*, *starting character position*, *# of characters*)`:

```
SELECT incidnt_num,
       date,
       SUBSTR(date, 4, 2) AS day
  FROM tutorial.sf_crime_incidents_2014_01
```

Write a query that separates the `location` field into separate fields for latitude and longitude. You can compare your results against the actual `lat` and `lon` fields in the table.

Try it out | See the answer

CONCAT

You can combine strings from several columns together (and with hard-coded values) using `CONCAT`. 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:

```
SELECT incidnt_num,
       day_of_week,
       LEFT(date, 10) AS cleaned_date,
       CONCAT(day_of_week, ', ', LEFT(date, 10)) AS day_and_date
  FROM tutorial.sf_crime_incidents_2014_01
```

Concatenate the `lat` and `lon` fields to form a field that is equivalent to the `location` field. (Note that the answer will have a different decimal precision.)

Try it out | See the answer

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

```
SELECT incidnt_num,
       day_of_week,
       LEFT(date, 10) AS cleaned_date,
       day_of_week || ', ' || LEFT(date, 10) AS day_and_date
  FROM tutorial.sf_crime_incidents_2014_01
```

Create the same concatenated `location` field, but using the `||` syntax instead of `CONCAT`.

Try it out | See the answer

Write a query that creates a date column formatted YYYY-MM-DD.

Try it out | See the answer

Changing case with UPPER and 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:

```
SELECT incidnt_num,
       address,
       UPPER(address) AS address_upper,
       LOWER(address) AS address_lower
  FROM tutorial.sf_crime_incidents_2014_01
```

Write a query that returns the `category` field, but with the first letter capitalised and the rest of the letters in lower-case.

Try it out | See the answer

There are a number of variations of these functions, as well as several other string functions not covered here. Different databases use subtle variations on these functions, so be sure to look up the appropriate database's syntax if you're connected to a private database. If you're using Mode's public service as in this tutorial, the Postgres literature contains the related functions.

Turning strings into dates

Dates are some of the most commonly screwed-up formats in SQL. This can be the result of a few things:

- The data was manipulated in Excel at some point, and the dates were changed to MM/DD/YYYY format or another format that is not compliant with SQL's strict standards.
- The data was manually entered by someone who use whatever formatting convention he/she was most familiar with.
- The date uses text (Jan, Feb, etc.) intsead of numbers to record months.

In order to take advantage of all of the great date functionality (`INTERVAL`, as well as some others you will learn in the next section), you need to have your date field formatted appropriately. This often involves some text manipulation, followed by a `CAST`. Let's revisit the answer to one of the practice problems above:

```
SELECT incidnt_num,
       date,
       (SUBSTR(date, 7, 4) || '-' || LEFT(date, 2) ||
        '-' || SUBSTR(date, 4, 2))::date AS cleaned_date
  FROM tutorial.sf_crime_incidents_2014_01
```

This example is a little different from the answer above in that we've wrapped the entire set of concatenated substrings in parentheses and cast the result in the `date`format. We could also cast it as `timestamp`, which includes additional precision (hours, minutes, seconds). In this case, we're not pulling the hours out of the original field, so we'll just stick to `date`.

Write a query that creates an accurate timestamp using the `date` and `time`columns in `tutorial.sf_crime_incidents_2014_01`. Include a field that is exactly 1 week later as well.

Try it out | See the answer

Turning dates into more useful dates

Once you've got a well-formatted date field, you can manipulate in all sorts of interesting ways. To make the lesson a little cleaner, we'll use a different version of the crime incidents dataset that already has a nicely-formatted date field:

```
SELECT *
  FROM tutorial.sf_crime_incidents_cleandate
```

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:

```
SELECT cleaned_date,
       EXTRACT('year'   FROM cleaned_date) AS year,
       EXTRACT('month'  FROM cleaned_date) AS month,
       EXTRACT('day'    FROM cleaned_date) AS day,
       EXTRACT('hour'   FROM cleaned_date) AS hour,
       EXTRACT('minute' FROM cleaned_date) AS minute,
       EXTRACT('second' FROM cleaned_date) AS second,
       EXTRACT('decade' FROM cleaned_date) AS decade,
       EXTRACT('dow'    FROM cleaned_date) AS day_of_week
  FROM tutorial.sf_crime_incidents_cleandate
```

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:

```
SELECT cleaned_date,
       DATE_TRUNC('year'   , cleaned_date) AS year,
       DATE_TRUNC('month'  , cleaned_date) AS month,
       DATE_TRUNC('week'   , cleaned_date) AS week,
       DATE_TRUNC('day'    , cleaned_date) AS day,
       DATE_TRUNC('hour'   , cleaned_date) AS hour,
       DATE_TRUNC('minute' , cleaned_date) AS minute,
       DATE_TRUNC('second' , cleaned_date) AS second,
       DATE_TRUNC('decade' , cleaned_date) AS decade
  FROM tutorial.sf_crime_incidents_cleandate
```

Write a query that counts the number of incidents reported by week. Cast the week as a date to get rid of the hours/minutes/seconds.

Try it out | See the answer

What if you want to include today's date or time? You can instruct your query to pull the local date and time at the time the query is run using any number of functions. Interestingly, you can run them without a `FROM` clause:

```
SELECT CURRENT_DATE AS date,
       CURRENT_TIME AS time,
       CURRENT_TIMESTAMP AS timestamp,
       LOCALTIME AS localtime,
       LOCALTIMESTAMP AS localtimestamp,
       NOW() AS now
```

As you can see, the different options vary in precision. You might notice that these times probably aren't actually your local time. Mode's database is set to Coordinated

Universal Time (UTC), which is basically the same as GMT. If you run a current time function against a connected database, you might get a result in a different time zone.

You can make a time appear in a different time zone using `AT TIME ZONE`:

```
SELECT CURRENT_TIME AS time,
       CURRENT_TIME AT TIME ZONE 'PST' AS time_pst
```

For a complete list of timezones, look here. This functionality is pretty complex because timestamps can be stored with or without timezone metadata. For a better understanding of the exact syntax, we recommend checking out the Postgres documentation.

Write a query that shows exactly how long ago each indicent was reported. Assume that the dataset is in Pacific Standard Time (UTC - 8).

Try it out | See the answer

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:

```
SELECT incidnt_num,
       descript,
       COALESCE(descript, 'No Description')
  FROM tutorial.sf_crime_incidents_cleandate
 ORDER BY descript DESC
```

Writing Subqueries in SQL

Check out the beginning.

In this lesson, you will continue to work with the same San Francisco Crime data used in a previous lesson.

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:

```
SELECT sub.*
  FROM (
        SELECT *
          FROM tutorial.sf_crime_incidents_2014_01
         WHERE day_of_week = 'Friday'
       ) sub
 WHERE sub.resolution = 'NONE'
```

Let's break down what happens when you run the above query:

First, the database runs the "inner query"—the part between the parentheses:

```
SELECT *
  FROM tutorial.sf_crime_incidents_2014_01
 WHERE day_of_week = 'Friday'
```

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*:

```
SELECT sub.*
  FROM (
       <<results from inner query go here>>
       ) sub
 WHERE sub.resolution = 'NONE'
```

Subqueries are required to have names, which are added after parentheses the same way you would add an alias to a normal table. In this case, we've used the name "sub."

A quick note on formatting: The important thing to remember when using subqueries is to provide some way to for the reader to easily determine which parts of the query will be executed together. Most people do this by indenting the subquery in some way. The examples in this tutorial are indented quite far—all the way to the parentheses. This isn't practical if you nest many subqueries, so it's fairly common to only indent two spaces or so.

Write a query that selects all Warrant Arrests from the `tutorial.sf_crime_incidents_2014_01` dataset, then wrap it in an outer query that only displays unresolved incidents.

Try it out See the answer

The above examples, as well as the practice problem don't really require subqueries—they solve problems that could also be solved by adding multiple conditions to the `WHERE` clause. These next sections provide examples for which subqueries are the best or only way to solve their respective problems.

Using subqueries to aggregate in multiple stages

What if you wanted to figure out how many incidents get reported on each day of the week? Better yet, what if you wanted to know how many incidents happen, on average, on a Friday in December? In January? There are two steps to this process: counting the number of incidents each day (inner query), then determining the monthly average (outer query):

```
SELECT LEFT(sub.date, 2) AS cleaned_month,
       sub.day_of_week,
       AVG(sub.incidents) AS average_incidents
  FROM (
        SELECT day_of_week,
               date,
               COUNT(incidnt_num) AS incidents
          FROM tutorial.sf_crime_incidents_2014_01
         GROUP BY 1,2
       ) sub
 GROUP BY 1,2
 ORDER BY 1,2
```

If you're having trouble figuring out what's happening, try running the inner query individually to get a sense of what its results look like. In general, it's easiest to write

inner queries first and revise them until the results make sense to you, then to move on to the outer query.

Write a query that displays the average number of monthly incidents for each category. Hint: use `tutorial.sf_crime_incidents_cleandate` to make your life a little easier.

Try it out | See the answer

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):

```
SELECT *
  FROM tutorial.sf_crime_incidents_2014_01
 WHERE Date = (SELECT MIN(date)
                 FROM tutorial.sf_crime_incidents_2014_01
               )
```

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:

```
SELECT *
  FROM tutorial.sf_crime_incidents_2014_01
 WHERE Date IN (SELECT date
                  FROM tutorial.sf_crime_incidents_2014_01
                 ORDER BY date
                 LIMIT 5
               )
```

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:

```
SELECT *
  FROM tutorial.sf_crime_incidents_2014_01 incidents
  JOIN ( SELECT date
           FROM tutorial.sf_crime_incidents_2014_01
          ORDER BY date
          LIMIT 5
       ) sub
    ON incidents.date = sub.date
```

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:

```
SELECT incidents.*,
       sub.incidents AS incidents_that_day
  FROM tutorial.sf_crime_incidents_2014_01 incidents
  JOIN ( SELECT date,
          COUNT(incidnt_num) AS incidents
           FROM tutorial.sf_crime_incidents_2014_01
          GROUP BY 1
       ) sub
    ON incidents.date = sub.date
 ORDER BY sub.incidents DESC, time
```

Write a query that displays all rows from the three categories with the fewest incidents reported.

Try it out | See the answer

Subqueries can be very helpful in improving the performance of your queries. Let's revisit the Crunchbase Data briefly. Imagine you'd like to aggregate all of the companies receiving investment and companies acquired each month. You could do that without subqueries if you wanted to, but don't actually run this as it will take minutes to return:

```
SELECT COALESCE(acquisitions.acquired_month, investments.funded_month) AS
month,
       COUNT(DISTINCT acquisitions.company_permalink) AS
companies_acquired,
       COUNT(DISTINCT investments.company_permalink) AS investments
  FROM tutorial.crunchbase_acquisitions acquisitions
  FULL JOIN tutorial.crunchbase_investments investments
    ON acquisitions.acquired_month = investments.funded_month
 GROUP BY 1
```

Note that in order to do this properly, you must join on date fields, which causes a massive "data explosion." Basically, what happens is that you're joining every row in a given month from one table onto every month in a given row on the other table, so the number of rows returned is incredibly great. Because of this multiplicative effect, you must use `COUNT(DISTINCT)` instead of `COUNT` to get accurate counts. You can see this below:

The following query shows 7,414 rows:

```
SELECT COUNT(*) FROM tutorial.crunchbase_acquisitions
```

The following query shows 83,893 rows:

```
SELECT COUNT(*) FROM tutorial.crunchbase_investments
```

The following query shows 6,237,396 rows:

```
    SELECT COUNT(*)
      FROM tutorial.crunchbase_acquisitions acquisitions
      FULL JOIN tutorial.crunchbase_investments investments
        ON acquisitions.acquired_month = investments.funded_month
```

If you'd like to understand this a little better, you can do some extra research on cartesian products. It's also worth noting that the `FULL JOIN` and `COUNT` above actually runs pretty fast—it's the `COUNT(DISTINCT)` that takes forever. More on that in the lesson on optimizing queries.

Of course, you could solve this much more efficiently by aggregating the two tables separately, then joining them together so that the counts are performed across far smaller datasets:

```
SELECT COALESCE(acquisitions.month, investments.month) AS month,
       acquisitions.companies_acquired,
       investments.companies_rec_investment
  FROM (
        SELECT acquired_month AS month,
               COUNT(DISTINCT company_permalink) AS companies_acquired
          FROM tutorial.crunchbase_acquisitions
         GROUP BY 1
       ) acquisitions

  FULL JOIN (
        SELECT funded_month AS month,
               COUNT(DISTINCT company_permalink) AS
companies_rec_investment
```

```
          FROM tutorial.crunchbase_investments
         GROUP BY 1
       )investments

    ON acquisitions.month = investments.month
 ORDER BY 1 DESC
```

Note: We used a `FULL JOIN` above just in case one table had observations in a month that the other table didn't. We also used `COALESCE` to display months when the `acquisitions` subquery didn't have month entries (presumably no acquisitions occurred in those months). We strongly encourage you to re-run the query without some of these elements to better understand how they work. You can also run each of the subqueries independently to get a better understanding of them as well.

Write a query that counts the number of companies founded and acquired by quarter starting in Q1 2012. Create the aggregations in two separate queries, then join them.

Try it out See the answer

Subqueries and UNIONs

For this next section, we will borrow directly from the lesson on UNIONs—again using the Crunchbase data:

```
SELECT *
  FROM tutorial.crunchbase_investments_part1

 UNION ALL

 SELECT *
   FROM tutorial.crunchbase_investments_part2
```

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:

```
SELECT COUNT(*) AS total_rows
  FROM (
        SELECT *
          FROM tutorial.crunchbase_investments_part1

         UNION ALL
```

```
      SELECT *
        FROM tutorial.crunchbase_investments_part2
    ) sub
```

This is pretty straightforward. Try it for yourself:

Write a query that ranks investors from the combined dataset above by the total number of investments they have made.

Try it out | See the answer

Write a query that does the same thing as in the previous problem, except only for companies that are still operating. Hint: operating status is in `tutorial.crunchbase_companies`.

Try it out | See the answer

SQL Window Functions

Check out the beginning.

This lesson uses data from Washington DC's Capital Bikeshare Program, which publishes detailed trip-level historical data on their website. The data was downloaded in February, 2014, but is limited to data collected during the first quarter of 2012. Each row represents one ride. Most fields are self-explanatory, except `rider_type`: "Registered" indicates a monthly membership to the rideshare program, "Casual" incidates that the rider bought a 3-day pass.
The `start_time` and `end_time`fields were cleaned up from their original forms to suit SQL date formatting—they are stored in this table as timestamps.

Intro to 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:

```
SELECT duration_seconds,
       SUM(duration_seconds) OVER (ORDER BY start_time) AS running_total
  FROM tutorial.dc_bikeshare_q1_2012
```

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`."

If you'd like to narrow the window from the entire dataset to individual groups within the dataset, you can use `PARTITION BY` to do so:

```
SELECT start_terminal,
       duration_seconds,
       SUM(duration_seconds) OVER
         (PARTITION BY start_terminal ORDER BY start_time)
         AS running_total
  FROM tutorial.dc_bikeshare_q1_2012
 WHERE start_time < '2012-01-08'
```

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:

```
SELECT start_terminal,
       duration_seconds,
       SUM(duration_seconds) OVER
         (PARTITION BY start_terminal) AS start_terminal_total
  FROM tutorial.dc_bikeshare_q1_2012
 WHERE start_time < '2012-01-08'
```

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.

Write a query modification of the above example query that shows the duration of each ride as a percentage of the total time accrued by riders from each start_terminal

Try it out | See the answer

The usual suspects: SUM, COUNT, and AVG

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. Make

```
SELECT start_terminal,
       duration_seconds,
       SUM(duration_seconds) OVER
         (PARTITION BY start_terminal) AS running_total,
       COUNT(duration_seconds) OVER
         (PARTITION BY start_terminal) AS running_count,
       AVG(duration_seconds) OVER
         (PARTITION BY start_terminal) AS running_avg
  FROM tutorial.dc_bikeshare_q1_2012
 WHERE start_time < '2012-01-08'
```

Alternatively, the same functions with `ORDER BY`:

```
SELECT start_terminal,
       duration_seconds,
       SUM(duration_seconds) OVER
         (PARTITION BY start_terminal ORDER BY start_time)
         AS running_total,
       COUNT(duration_seconds) OVER
         (PARTITION BY start_terminal ORDER BY start_time)
         AS running_count,
       AVG(duration_seconds) OVER
         (PARTITION BY start_terminal ORDER BY start_time)
         AS running_avg
  FROM tutorial.dc_bikeshare_q1_2012
 WHERE start_time < '2012-01-08'
```

Make sure you plug those previous two queries into Mode and run them. This next practice problem is very similar to the examples, so try modifying the above code rather than starting from scratch.

Write a query that shows a running total of the duration of bike rides (similar to the last example), but grouped by `end_terminal`, and with ride duration sorted in descending order.

Try it out See the answer

ROW_NUMBER()

`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. `ROW_NUMBER()` does not require you to specify a variable within the parentheses:

```
SELECT start_terminal,
       start_time,
       duration_seconds,
```

```
       ROW_NUMBER() OVER (ORDER BY start_time)
                    AS row_number
  FROM tutorial.dc_bikeshare_q1_2012
 WHERE start_time < '2012-01-08'
```

Using the `PARTITION BY` clause will allow you to begin counting 1 again in each partition. The following query starts the count over again for each terminal:

```
SELECT start_terminal,
       start_time,
       duration_seconds,
       ROW_NUMBER() OVER (PARTITION BY start_terminal
                          ORDER BY start_time)
                    AS row_number
  FROM tutorial.dc_bikeshare_q1_2012
 WHERE start_time < '2012-01-08'
```

RANK() and DENSE_RANK()

`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:

```
SELECT start_terminal,
       duration_seconds,
       RANK() OVER (PARTITION BY start_terminal
                    ORDER BY start_time)
              AS rank
  FROM tutorial.dc_bikeshare_q1_2012
 WHERE start_time < '2012-01-08'
```

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.

Write a query that shows the 5 longest rides from each starting terminal, ordered by terminal, and longest to shortest rides within each terminal. Limit to rides that occurred before Jan. 8, 2012.

Try it out | See the answer

NTILE

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:

```
SELECT start_terminal,
       duration_seconds,
       NTILE(4) OVER
         (PARTITION BY start_terminal ORDER BY duration_seconds)
          AS quartile,
       NTILE(5) OVER
         (PARTITION BY start_terminal ORDER BY duration_seconds)
         AS quintile,
       NTILE(100) OVER
         (PARTITION BY start_terminal ORDER BY duration_seconds)
         AS percentile
  FROM tutorial.dc_bikeshare_q1_2012
 WHERE start_time < '2012-01-08'
 ORDER BY start_terminal, duration_seconds
```

Looking at the results from the query above, you can see that the `percentile` column doesn't calculate exactly as you might expect. If you only had two records and you were measuring percentiles, you'd expect one record to define the 1st percentile, and the other record to define the 100th percentile. Using the `NTILE` function, what you'd actually see is one record in the 1st percentile, and one in the 2nd percentile. You can see this in the results for `start_terminal` 31000—the `percentile` column just looks like a numerical ranking. If you scroll down to `start_terminal` 31007, you can see that it properly calculates percentiles because there are more than 100 records for that `start_terminal`. If you're working with very small windows, keep this in mind and consider using quartiles or similarly small bands.

Write a query that shows only the duration of the trip and the percentile into which that duration falls (across the entire dataset—not partitioned by terminal).

Try it out | See the answer

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:

```
SELECT start_terminal,
       duration_seconds,
       LAG(duration_seconds, 1) OVER
         (PARTITION BY start_terminal ORDER BY duration_seconds) AS lag,
       LEAD(duration_seconds, 1) OVER
         (PARTITION BY start_terminal ORDER BY duration_seconds) AS lead
  FROM tutorial.dc_bikeshare_q1_2012
 WHERE start_time < '2012-01-08'
 ORDER BY start_terminal, duration_seconds
```

This is especially useful if you want to calculate differences between rows:

```
SELECT start_terminal,
       duration_seconds,
       duration_seconds -LAG(duration_seconds, 1) OVER
         (PARTITION BY start_terminal ORDER BY duration_seconds)
        AS difference
  FROM tutorial.dc_bikeshare_q1_2012
 WHERE start_time < '2012-01-08'
 ORDER BY start_terminal, duration_seconds
```

The first row of the `difference` column is null because there is no previous row from which to pull. Similarly, using `LEAD` will create nulls at the end of the dataset. If you'd like to make the results a bit cleaner, you can wrap it in an outer query to remove nulls:

```
SELECT *
  FROM (
    SELECT start_terminal,
           duration_seconds,
           duration_seconds -LAG(duration_seconds, 1) OVER
             (PARTITION BY start_terminal ORDER BY duration_seconds)
            AS difference
      FROM tutorial.dc_bikeshare_q1_2012
     WHERE start_time < '2012-01-08'
     ORDER BY start_terminal, duration_seconds
       ) sub
 WHERE sub.difference IS NOT NULL
```

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:

```
SELECT start_terminal,
       duration_seconds,
       NTILE(4) OVER
         (PARTITION BY start_terminal ORDER BY duration_seconds)
         AS quartile,
       NTILE(5) OVER
         (PARTITION BY start_terminal ORDER BY duration_seconds)
         AS quintile,
       NTILE(100) OVER
         (PARTITION BY start_terminal ORDER BY duration_seconds)
         AS percentile
  FROM tutorial.dc_bikeshare_q1_2012
 WHERE start_time < '2012-01-08'
 ORDER BY start_terminal, duration_seconds
```

This can be rewritten as:

```
SELECT start_terminal,
       duration_seconds,
       NTILE(4) OVER ntile_window AS quartile,
       NTILE(5) OVER ntile_window AS quintile,
       NTILE(100) OVER ntile_window AS percentile
  FROM tutorial.dc_bikeshare_q1_2012
 WHERE start_time < '2012-01-08'
WINDOW ntile_window AS
         (PARTITION BY start_terminal ORDER BY duration_seconds)
 ORDER BY start_terminal, duration_seconds
```

The `WINDOW` clause, if included, should always come after the `WHERE` clause.

Advanced windowing techniques

You can check out a complete list of window functions in Postgres (the syntax Mode uses) in the Postgres documentation. If you're using window functions on a connected database, you should look at the appropriate syntax guide for your system.

Performance Tuning SQL Queries

Check out the beginning.

The lesson on subqueries introduced the idea that you can sometimes create the same desired result set with a faster-running query. In this lesson, you'll learn to identify when your queries can be improved, and how to improve them.

The theory behind query run time

A database is a piece of software that runs on a computer, and is subject to the same limitations as all software—it can only process as much information as its hardware is capable of handling. The way to make a query run faster is to reduce the number of calculations that the software (and therefore hardware) must perform. To do this, you'll need some understanding of how SQL actually makes calculations. First, let's address some of the high-level things that will affect the number of calculations you need to make, and therefore your querys runtime:

- Table size: If your query hits one or more tables with millions of rows or more, it could affect performance.
- Joins: If your query joins two tables in a way that substantially increases the row count of the result set, your query is likely to be slow. There's an example of this in the subqueries lesson.
- Aggregations: Combining multiple rows to produce a result requires more computation than simply retrieving those rows.

Query runtime is also dependent on some things that you can't really control related to the database itself:

- Other users running queries: The more queries running concurrently on a database, the more the database must process at a given time and the slower everything will run. It can be especially bad if others are running particularly resource-intensive queries that fulfill some of the above criteria.
- Database software and optimization: This is something you probably can't control, but if you know the system you're using, you can work within its bounds to make your queries more efficient.

For now, let's ignore the things you can't control and work on the things you can.

Reducing table size

Filtering the data to include only the observations you need can dramatically improve query speed. How you do this will depend entirely on the problem you're trying to

solve. For example, if you've got time series data, limiting to a small time window can make your queries run much more quickly:

```
SELECT *
  FROM benn.sample_event_table
 WHERE event_date >= '2014-03-01'
   AND event_date <  '2014-04-01'
```

Keep in mind that you can always perform exploratory analysis on a subset of data, refine your work into a final query, then remove the limitation and run your work across the entire dataset. The final query might take a long time to run, but at least you can run the intermediate steps quickly.

This is why Mode enforces a `LIMIT` clause by default—100 rows is often more than you need to determine the next step in your analysis, and it's a small enough dataset that it will return quickly.

It's worth noting that `LIMIT` doesn't quite work the same way with aggregations—the aggregation is performed, then the results are limited to the specified number of rows. So if you're aggregating into one row as below, `LIMIT 100` will do nothing to speed up your query:

```
SELECT COUNT(*)
  FROM benn.sample_event_table
 LIMIT 100
```

If you want to limit the dataset before performing the count (to speed things up), try doing it in a subquery:

```
SELECT COUNT(*)
  FROM (
    SELECT *
      FROM benn.sample_event_table
     LIMIT 100
       ) sub
```

Note: Using `LIMIT` this will dramatically alter your results, so you should use it to test query logic, but not to get actual results.

In general, when working with subqueries, you should make sure to limit the amount of data you're working with in the place where it will be executed first. This means putting the `LIMIT` in the subquery, not the outer query. Again, this is for making the query run fast so that you can test—*NOT* for producing good results.

Making joins less complicated

In a way, this is an extension of the previous tip. In the same way that it's better to reduce data at a point in the query that is executed early, it's better to reduce table sizes before joining them. Take this example, which joins information about college sports teams onto a list of players at various colleges:

```
SELECT teams.conference AS conference,
       players.school_name,
       COUNT(1) AS players
  FROM benn.college_football_players players
  JOIN benn.college_football_teams teams
    ON teams.school_name = players.school_name
 GROUP BY 1,2
```

There are 26,298 rows in `benn.college_football_players`. That means that 26,298 rows need to be evaluated for matches in the other table. But if the `benn.college_football_players` table was pre-aggregated, you could reduce the number of rows that need to be evaluated in the join. First, let's look at the aggregation:

```
SELECT players.school_name,
       COUNT(*) AS players
  FROM benn.college_football_players players
 GROUP BY 1
```

The above query returns 252 results. So dropping that in a subquery and then joining to it in the outer query will reduce the cost of the join substantially:

```
SELECT teams.conference,
       sub.*
  FROM (
        SELECT players.school_name,
               COUNT(*) AS players
          FROM benn.college_football_players players
         GROUP BY 1
       ) sub
  JOIN benn.college_football_teams teams
  ON teams.school_name = sub.school_name
```

In this particular case, you won't notice a huge difference because 30,000 rows isn't too hard for the database to process. But if you were talking about hundreds of thousands of rows or more, you'd see a noticeable improvement by aggregating before joining. When you do this, make sure that what you're doing is logically consistent—you should worry about the accuracy of your work before worrying about run speed.

Pivoting Data in SQL

Check out the beginning.

Pivoting rows to columns

This lesson will teach you how to take data that is formatted for analysis and pivot it for presentation or charting. We'll take a dataset that looks like this:

Table

conference	year
ACC	FR
ACC	JR
ACC	SO
ACC	SR
American Athletic	FR
American Athletic	JR
American Athletic	SO
American Athletic	SR
Big 12	FR
Big 12	JR
Big 12	SO
Big 12	SR
Big Sky	FR
Big Sky	JR
Big Sky	SO

And make it look like this:

Table

conference	total_players	fr
SEC	1650	659
ACC	1563	607
Conference USA	1495	519
Big Ten	1466	636
Mid-American	1392	551
Pac-12	1377	501
Mountain West	1285	458
Pioneer	1214	470
Big Sky	1198	442
Big 12	1190	456
American Athletic	1124	418
CAA	1046	335
MEAC	966	375
Missouri Valley	964	374
Southern	925	434
Ivy	871	214

For this example, we'll use the same dataset of College Football players used in the CASE lesson. You can view the data directly here.

Let's start by aggregating the data to show the number of players of each year in each conference, similar to the first example in the inner join lesson:

```
SELECT teams.conference AS conference,
       players.year,
       COUNT(1) AS players
  FROM benn.college_football_players players
  JOIN benn.college_football_teams teams
    ON teams.school_name = players.school_name
 GROUP BY 1,2
 ORDER BY 1,2
```

View this in Mode.

In order to transform the data, we'll need to put the above query into a subquery. It can be helpful to create the subquery and select all columns from it before starting to make transformations. Re-running the query at incremental steps like this makes it

easier to debug if your query doesn't run. Note that you can eliminate the `ORDER BY` clause from the subquery since we'll reorder the results in the outer query.

```
SELECT *
  FROM (
        SELECT teams.conference AS conference,
               players.year,
               COUNT(1) AS players
          FROM benn.college_football_players players
          JOIN benn.college_football_teams teams
            ON teams.school_name = players.school_name
         GROUP BY 1,2
       ) sub
```

Assuming that works as planned (results should look exactly the same as the first query), it's time to break the results out into different columns for various years. Each item in the `SELECT` statement creates a column, so you'll have to create a separate column for each year:

```
SELECT conference,
       SUM(CASE WHEN year = 'FR' THEN players ELSE NULL END) AS fr,
       SUM(CASE WHEN year = 'SO' THEN players ELSE NULL END) AS so,
       SUM(CASE WHEN year = 'JR' THEN players ELSE NULL END) AS jr,
       SUM(CASE WHEN year = 'SR' THEN players ELSE NULL END) AS sr
  FROM (
        SELECT teams.conference AS conference,
               players.year,
               COUNT(1) AS players
          FROM benn.college_football_players players
          JOIN benn.college_football_teams teams
            ON teams.school_name = players.school_name
         GROUP BY 1,2
       ) sub
 GROUP BY 1
 ORDER BY 1
```

Technically, you've now accomplished the goal of this tutorial. But this could still be made a little better. You'll notice that the above query produces a list that is ordered alphabetically by Conference. It might make more sense to add a "total players" column and order by that (largest to smallest):

```
SELECT conference,
       SUM(players) AS total_players,
       SUM(CASE WHEN year = 'FR' THEN players ELSE NULL END) AS fr,
       SUM(CASE WHEN year = 'SO' THEN players ELSE NULL END) AS so,
       SUM(CASE WHEN year = 'JR' THEN players ELSE NULL END) AS jr,
       SUM(CASE WHEN year = 'SR' THEN players ELSE NULL END) AS sr
  FROM (
```

```
        SELECT teams.conference AS conference,
               players.year,
               COUNT(1) AS players
          FROM benn.college_football_players players
          JOIN benn.college_football_teams teams
            ON teams.school_name = players.school_name
         GROUP BY 1,2
       ) sub
GROUP BY 1
ORDER BY 2 DESC
```

And you're done! View this in Mode.

Pivoting columns to rows

A lot of data you'll find out there on the internet is formatted for consumption, not analysis. Take, for example, this table showing the number of earthquakes worldwide from 2000-2012:

Magnitude	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012
8.0 to 9.9	1	1	0	1	2	1	2	4	0	1	1	1	2
7.0 to 7.9	14	15	13	14	14	10	9	14	12	16	23	19	12
6.0 to 6.9	146	121	127	140	141	140	142	178	168	144	150	185	108
5.0 to 5.9	1344	1224	1201	1203	1515	1693	1712	2074	1768	1896	2209	2276	1401
4.0 to 4.9	8008	7991	8541	8462	10888	13917	12838	12078	12291	6805	10164	13315	9534
3.0 to 3.9	4827	6266	7068	7624	7932	9191	9990	9889	11735	2905	4341	2791	2453
2.0 to 2.9	3765	4164	6419	7727	6316	4636	4027	3597	3860	3014	4626	3643	3111
1.0 to 1.9	1026	944	1137	2506	1344	26	18	42	21	26	39	47	43
0.1 to 0.9	5	1	10	134	103	0	2	2	0	1	0	1	0
No Magnitude	3120	2807	2938	3608	2939	864	828	1807	1922	17	24	11	3
Total	22256	23534	27454	31419	31194	30478	29568	29685	31777	14825	21577	* 22289	* 16667
Estimated Deaths	231	21357	1685	33819	228802	88003	6605	712	88011	1790	320120	21953	768

In this format it's challenging to answer questions like "what's the average magnitude of an earthquake?" It would be much easier if the data were displayed in 3 columns: "magnitude", "year", and "number of earthquakes." Here's how to transform the data into that form:

First, check out this data in Mode:

```
SELECT *
  FROM tutorial.worldwide_earthquakes
```

Note: column names begin with 'year_' because Mode requires column names to begin with letters.

The first thing to do here is to create a table that lists all of the columns from the original table as rows in a new table. Unless you have a ton of columns to transform, the easiest way is often just to list them out in a subquery:

```
SELECT year
  FROM (VALUES (2000),(2001),(2002),(2003),(2004),(2005),(2006),
               (2007),(2008),(2009),(2010),(2011),(2012)) v(year)
```

Once you've got this, you can cross join it with the `worldwide_earthquakes` table to create an expanded view:

```
SELECT years.*,
       earthquakes.*
  FROM tutorial.worldwide_earthquakes earthquakes
 CROSS JOIN (
       SELECT year
         FROM (VALUES (2000),(2001),(2002),(2003),(2004),(2005),(2006),
                      (2007),(2008),(2009),(2010),(2011),(2012)) v(year)
       ) years
```

Notice that each row in the `worldwide_earthquakes` is replicated 13 times. The last thing to do is to fix this using a `CASE` statement that pulls data from the correct column in the `worldwide_earthquakes` table given the value in the `year` column:

```
SELECT years.*,
       earthquakes.magnitude,
       CASE year
         WHEN 2000 THEN year_2000
         WHEN 2001 THEN year_2001
         WHEN 2002 THEN year_2002
         WHEN 2003 THEN year_2003
         WHEN 2004 THEN year_2004
         WHEN 2005 THEN year_2005
         WHEN 2006 THEN year_2006
         WHEN 2007 THEN year_2007
         WHEN 2008 THEN year_2008
         WHEN 2009 THEN year_2009
         WHEN 2010 THEN year_2010
         WHEN 2011 THEN year_2011
         WHEN 2012 THEN year_2012
         ELSE NULL END
         AS number_of_earthquakes
  FROM tutorial.worldwide_earthquakes earthquakes
 CROSS JOIN (
       SELECT year
```

```
        FROM (VALUES (2000),(2001),(2002),(2003),(2004),(2005),(2006),
                     (2007),(2008),(2009),(2010),(2011),(2012)) v(year)
    ) years
```

View the final product in Mode.

What's next?

Congrats on finishing the Advanced SQL Tutorial! Now that you've got a handle on SQL, the next step is to hone your analytical process.

We've built the SQL Analytics Training section for that very purpose. With fake datasets to mimic real-world situations, you can approach this section like on-the-job training. Check it out!