Your Intermediate Guide to SQL

As you get more comfortable with SQL, you will be able to take on even more advanced queries. This in-depth guide will give you a more detailed introduction to some of the SQL functions you have already learned, and give you some new tools to work with. Be sure to save this guide, so you can easily reference these helpful tips in the future.

SQL Structure for More Complex Queries

You will learn more about these clauses and expressions in the sections below, but first let's check out how you might structure a more involved SQL query:

```
SELECT         Column you want to look at

FROM           Table the data lives in

WHERE          Certain condition on the data

GROUP BY       Column you want to aggregate by

HAVING         Certain condition on the aggregation

ORDER BY       Column you want to order results by
               and in ASCending or DESCending order

LIMIT          The maximum number of rows you want
               your results to contain
```

Different types of JOINs

Most analysts will use either **INNER JOINs** or **LEFT JOINs** throughout their career. When you join tables, you are combining data from one table with data in another table connected by a common field. For example, let's say you have your friends' favorite colors in one table and your friends' favorite movies in another table. You can have both of their favorite colors and movies in one table by joining the two tables on your friends' names, which is the field they have in common. This is your JOIN field. In the workplace, a JOIN field is usually some sort of ID, like a customer_id or account_id.

Table view

`Favorite_Colors`	`Favorite_Movies`
friend (string)	friend (string)
color (string)	movie (string)

Data view

Favorite_Colors	
friend	**color**
Rachel DeSantos	blue
Sujin Lee	green
Najil Okoro	red
John Anderson	orange

Favorite_Movies	
friend	**movie**
Rachel DeSantos	Avengers
Sujin Lee	Despicable Me
Najil Okoro	Frozen

So, in this example, you would want to use an INNER JOIN if you only want to see information for friends who have both a favorite color and a favorite movie. That means if John Anderson has a favorite color but no favorite movie, John Anderson won't show up in your results. Friends have to be in both tables to show up in your results. So INNER JOINs are useful when you want to see data where the JOIN key exists in both tables, which is usually why you want to join datasets in the first place. Typically, analysts will use INNER JOINs most of the time.

```
SELECT
      friend,
      color,
      movie
FROM
      Favorite_Colors AS c
INNER JOIN
      Favorite_Movies AS m ON c.friend = m.friend
```

Results:

friend	color	movie
Rachel DeSantos	blue	Avengers
Sujin Lee	green	Despicable Me
Najil Okoro	red	Frozen

Since this query used an INNER JOIN, the results only have three out of the four friends. As a quick reminder, that is because INNER JOIN queries only return results where the JOIN field—in this case "friend"—exists in both tables. Since John Anderson doesn't exist in the favorite movies table, he is excluded from the query results.

Now, let's say you want to use a LEFT JOIN to get information for all of your friends into one table (e.g. favorite colors table) with data added from the other table (e.g. favorite movies table) if it exists. So, if John Anderson has a favorite color but no favorite movie, he will still show up in your results. He will just have an empty field (which is null) for his favorite movie. Most of the time LEFT JOINs are used if the data you are trying to pull in from another table is optional. This is a nice-to-have field but not necessary for your analysis since you may get nulls. On the job, you will find that analysts will use LEFT JOINs less often than INNER JOINs.

```
SELECT
      friend,
      color,
      movie
FROM
      Favorite_Colors AS c
LEFT JOIN
      Favorite_Movies AS m ON c.friend = m.friend
```

Results:

friend	color	movie
Rachel DeSantos	blue	Avengers
Sujin Lee	green	Despicable Me
Najil Okoro	red	Frozen
John Anderson	orange	null

So now you know the difference between INNER JOINs and LEFT JOINs. You know that INNER JOINs will be the most commonly used JOIN types because they usually align with business use cases. Another reason that INNER JOINs are used is because they result in less data since the JOIN key must exist in both tables. This means that queries with INNER JOINs tend to run faster and use less resources than queries with LEFT JOINs. This might not be a problem for most analysts, but if you are working with very large tables that have 1+ million rows and/or depending on the SQL dialect used, your query might take a lot longer to run if you use a LEFT JOIN instead of an INNER JOIN.

Basically, the takeaway here is to use INNER JOINs as much as you can.

Aggregators like SUM() and COUNT()

Aggregators summarize rows into a single value. The functions **SUM()** and **COUNT()** are examples of aggregators. The types of aggregators that will be available for you will depend on the SQL dialect that your company uses. But the most commonly used aggregators like SUM(), COUNT(), MAX(), and MIN() are available in all SQL dialects, even if there are some slight differences. It is easy to check how aggregators are formatted for whatever dialect you are working with. There are a lot of resources available online. Just open up your favorite search engine and search for the aggregate function that you want to use and your SQL dialect. For example, search for "SUM function in SQL Server." Aggregators all work the same way, so let's go over SUM() and COUNT(). The SUM() function takes a sum of whatever column you put inside the parentheses. The COUNT() function counts the number of entries in whatever column you put inside the parentheses. For example, let's say you have a purchase table with a list of people and the number of movie tickets they purchased.

The purchase table:

name	**tickets**
Rachel DeSantos	3
Sujin Lee	2
Najil Okoro	2
John Anderson	1

Query:

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

Result:

total_tickets	number_of_purchases
8	4

You can also add a **DISTINCT** clause inside the function. This will work for most SQL dialects but it is always good to first check and confirm that the function works with the dialect that your company uses. Adding a DISTINCT clause in your SUM() or COUNT() function lets you do an aggregation only on each distinct value of the field. Check it out in the example below:

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

Result:

total_tickets	**total_distinct_tickets**	**number_of_purchases**	**number_of_distinct_ purchases**
8	6	4	3

You might notice that the results contain smaller numbers for the columns with DISTINCT in them. That is because DISTINCT tells SQL to only aggregate unique values. To better understand this, check out the second column for total_distinct_tickets, which demonstrates how DISTINCT can be used with a SUM() function. But, in this example, it doesn't really make sense to do a sum of distinct values. You will probably never use DISTINCT with SUM() functions. Instead, you will more likely use DISTINCT with COUNT() functions since it is helpful in identifying unique cases.

Using GROUP BY with aggregators

Earlier, when you learned about SUM() and COUNT() with movie ticket purchases, you summarized the data to get the total number of tickets purchased and the total number of purchases made with SUM() and COUNT(). When you use aggregate functions like SUM() and COUNT() with a **GROUP BY** clause, the groups are summarized in slices specified by the GROUP BY clause.

For example, let's pretend that your purchase table is like the one below, where each person's transaction was for a particular occasion. You would want to use a GROUP BY clause if you want to get the total number of tickets sold and the total number of purchases made by the occasion type. You will notice that if you want to aggregate by something (e.g. occasion), you can use the GROUP BY clause. In this way, SQL is pretty intuitive.

The new purchase table:

occasion	name	tickets
fun	Rachel DeSantos	5
date	Sujin Lee	2
date	Najil Okoro	2
fun	John Anderson	3

Query:

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

Results:

occasion	total_tickets	number_of_purchases
fun	8	2
date	4	2

Awesome! Now you know how to use the GROUP BY clause and when you would want to use it. Here is another cool thing to know: you can use the column number in the GROUP BY clause to specify what you want to group by instead of using the column names. In the last example, you wanted to group by the occasion. Occasion is the first column written in the SQL query. That means it is possible to write GROUP BY 1 instead of GROUP BY occasion. If occasion was the second column in the SELECT clause, then you would write GROUP BY 2. See below:

Query:

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

Is the same as:

```
SELECT
      occasion,
      SUM(tickets) AS total_tickets,
      COUNT(tickets) AS number_of_purchases
FROM
      purchases
GROUP BY
      1
```

Knowing this shortcut can save you time when writing your SQL queries and when you are grouping by multiple fields. In that case, you just need to separate them with commas (e.g. GROUP BY 1, 2, 3, 4).

When to use HAVING

The HAVING clause is similar to the WHERE clause since it filters the data based on certain conditions. But these clauses are used in different situations. The **WHERE** clause is used to make filters on your table, like a filter for certain date ranges or specific countries. The **HAVING** clause is used to make filters on your aggregations and has to be paired with a GROUP BY clause.

The purchase table:

occasion	**name**	**tickets**
fun	Rachel DeSantos	5
date	Sujin Lee	2
date	Najil Okoro	2
fun	John Anderson	3

In this example, you will notice that you can layer on the HAVING clause if you want to set limits on your aggregation, or the sum and count in this case:

Query:

```
SELECT
      occasion,
      SUM(tickets) AS total_tickets,
      COUNT(tickets) AS number_of_purchases
FROM
      purchases
GROUP BY
      occasion
HAVING
      SUM(tickets) > 5
```

Results:

occasion	**total_tickets**	**number_of_purchases**
fun	8	2

It is important to note that your results don't contain the 'date' occasion anymore. That is because your HAVING clause filters for sums that are greater than 5. The 'date' occasion only had 4 total tickets, which is less than 5, so the 'date' occasion doesn't show up in your results.

Great work! Now you know how and when to use the HAVING clause. As a data analyst, you will use a lot of WHERE clauses and only a few HAVING clauses. That is because of the business use case, but also because of resources, just like INNER JOIN vs. LEFT JOIN. If your query contains a HAVING clause, it will take longer to run and will take more resources because SQL needs to filter after it runs the SUM() and COUNT() calculations. So, it is a good idea to try minimizing your usage of the HAVING clause whenever possible. But, if you do need to use HAVING, try using temporary tables.

Using ORDER BY to organize your results

The **ORDER BY** clause helps you organize your results. It goes at the end of the SQL query and it is the very last clause to use unless you have a LIMIT clause.

A slightly modified version of the purchase table:

name	**tickets**
Rachel DeSantos	3
Sujin Lee	5
Najil Okoro	2
John Anderson	4

Let’s say that we want everyone in this table organized by the number of tickets they purchased from biggest to smallest, or descending order.

```
SELECT
      name,
      tickets
FROM
      purchases
ORDER BY
      tickets DESC
```

Result:

name	tickets
Sujin Lee	5
John Anderson	4
Rachel DeSantos	3
Najil Okoro	2

If you want to show the person with the least amount of tickets first, you would order your results in ASCending order. To do that in SQL, you can either use ASC or leave it blank since SQL orders columns in ASCending order by default. But, best practice is to write ASC or DESC so this clause is clear to everyone reading your query.

When to use LIMIT

The **LIMIT** clause is helpful when you only want to work with a select number of rows. This is generally used in two situations.

In the first situation, let’s say that you want the top X number of cases. In the movie ticket example, let’s say you only want the top 3 biggest purchases. You could use a LIMIT clause like below.

Query:

```
SELECT
      name,
      tickets
FROM
      purchases
ORDER BY
      tickets DESC
LIMIT 3 --top 3 results only
```

Result:

name	tickets
Sujin Lee	5
John Anderson	4
Rachel DeSantos	3

In the second situation, let's say you want to work with your whole dataset before you write your query. In that case, you would use a LIMIT clause so you don't waste resources pulling in every single row.

Query:

```
SELECT
      name,
      tickets
FROM
      purchases
ORDER BY
      tickets DESC
LIMIT 20 --top 20 results only
```

Result:

name	tickets
Rachel DeSantos	3
Sujin Lee	5
Najil Okoro	2
John Anderson	4

You may have noticed that you only have four rows of data in our results even though you set a limit of 20. That is because the purchase table only contains four rows of data. The LIMIT clause is a maximum number of rows to show, And if the purchase table contained one million rows, only 20 rows would show. But, since the purchase table contains less than 20 rows, all of the data is shown.

Conditional expressions like CASE, IF, and COALESCE()

Conditional expressions like CASE statements and the COALESCE() function are used when you want to change how your results are presented. Though you are setting conditions in conditional expressions, conditional expressions differ from the type of conditions you put in a WHERE clause. The conditions put in WHERE clauses apply to the entire query, while conditional expressions apply to just that particular field. On top of that, you can change how your results are presented with conditional expressions, which is something you can't do with WHERE statements. Let's check out the three most common conditional expressions: CASE, IF, and COALESCE().

CASE statements

CASE statements are most commonly used as labels within your dataset. You can use CASE statements to label rows that meet a certain condition as X and rows that meet another condition as Y. This is why you will find it commonly used with aggregate functions when you want to group things by categories. Here is an example using a table of movies that are playing at the local movie theater:

The MovieTheater table:

genre	**movie_title**
horror	Silence of the Lambs
comedy	Jumanji
family	Frozen 2
documentary	13th

Let's say you want to group these movies into the movies that you will watch and movies that you will not watch, and count the number of movies that fall into each category. Your query would be:

```
SELECT
CASE
      WHEN genre = ‘horror’ THEN ‘will not watch’
      ELSE ‘will watch’
      END AS watch_category, --creating your own category
COUNT(movie_title) AS number_of_movies
FROM
      MovieTheater
GROUP BY
1 --when grouping by CASE, use position numbers or type
entire CASE statement here
```

Results:

watch_category	**number_of_movies**
Will not watch	1
Will watch	3

You may notice that you added your own labels to the dataset, which you can do with CASE statements. But keep in mind that this feature isn't in all SQL dialects, including BigQuery. If you would like to learn more, please review the documentation for COUNT() or SUM() for your particular SQL dialect and check out how CASE statements can be used.

There is also another way that you can use CASE statements within BigQuery (again, this might not apply to all SQL dialects). If your conditions are matches, like the example above, then you could write your CASE statement as (compare lines 2-3):

```
SELECT
      CASE genre
            WHEN ‘horror’ THEN ‘will not watch’
            ELSE ‘will watch’
            END AS watch_category
      COUNT(movie_title) AS number_of_movies
FROM
      MovieTheater
GROUP BY
      1
```

This will produce the same results, but it is not recommended because it is limited to just matching conditions (e.g. genre = ‘horror’). By comparison, the earlier version with WHEN genre = ‘horror’ is flexible and can take other types of conditions, like greater than (>), less than (<), not equal to (<> or !=), etc.

IF statements

Next up is IF statements. **IF statements** are similar to CASE statements, but they have one key difference: CASE statements can take multiple conditions, whereas IF statements can't. In the example above, you only had one condition (e.g. WHEN genre = ‘horror’), so you could have also used an IF statement such as:

```
SELECT
      IF(genre=’horror’, ‘will not watch’, ‘will watch’)
      AS watch_category,
      COUNT(movie_title) AS number_of_movies
FROM
      MovieTheater
GROUP BY
      1
```

But, if you have multiple conditions, you will want to use a CASE statement such as:

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

Results:

watch_category	number_of_movies
Will not watch	1
Will watch alone	1
Watch with others	2

COALESCE() function

Finally, there is the **COALESCE()** function. This function is used to return the first non-null expression in the order specified in the function. It is useful when your data is spread out in multiple columns. For example, let’s say you have a table of movies as rows, columns as months, and values as 1 if the movie launched in that month or null if it didn’t. See table MovieLaunches below:

movie_title	Jan_2030	Feb_2030	Mar_2030
Avengers X	1	*null*	*null*
Frozen V	*null*	1	*null*
Lion King IV	*null*	*null*	*null*

If you want to know if these movies have launched between Jan-Mar 2030, then you can use COALESCE. For example:

```
SELECT
      movie_title,
      COALESCE(Jan_2030, Feb_2030, Mar_2030) AS
      launched_indicator
FROM
      MovieLaunches
```

Results:

movie_title	**launched_indicator**
Avengers X	1
Frozen V	1
Lion King IV	*null*

You may notice that two out of the three movies had non-null values in the fields specified (Jan_2030, Feb_2030, Mar_2030). This example demonstrates how the COALESCE function works. It will search through each column you specify within the function and try to return a non-null value if it finds one.

In the workplace, COALESCE is often used to make sure that fields don't contain nulls. So a COALESCE statement could be: COALESCE(try_this_field, then_this_field, 0) to tell SQL to check the first two fields in order for a non-null value. If none exist in those fields, then assign a zero in place of a null. In BigQuery, this is the same as using the IFNULL() function (more about that here). Other SQL dialects may not have the IFNULL() function, and in that case, COALESCE() is then used instead.

Creating and deleting tables

Data in SQL is stored in tables. This guide has been referencing small tables like the purchase, MovieLaunches, and MovieTheater tables. These are tables that already exist because you didn't create them.

Creating tables

The ideal situation to create a table is if the following three conditions are met:

1. Complex query containing multiple JOINs
2. Result output is a table
3. You need to run the query frequently or on a regular basis

If these conditions are met, then it is a great idea to create a reporting table. But it is best practice to check with your manager or teammates before you do in case you need to access permissions to create a reporting table.

The syntax for creating tables will change depending on the SQL dialect and SQL platform you use. You will go over how to create tables in BigQuery here, but if your company uses a different SQL dialect, it is a good idea to search online for how to create tables in that SQL dialect (e.g. “Create tables in PostgreSQL”). Or better yet, ask your manager or teammate for help.

Generally speaking, when you create tables, you want to make sure the same table doesn’t already exist. This is because if you try to create a table that already exists, the query will return an error. Checking for existing tables will differ between SQL dialects, but it is always a good thing to check to avoid unnecessary errors in your work.

The way to check for pre-existing tables in BigQuery is:

```
CREATE TABLE IF NOT EXISTS mydataset.FavoriteColorAndMovie
AS
SELECT
      friend,
      color,
      movie
FROM
      Favorite_Colors AS c
INNER JOIN
      Favorite_Movies AS m ON c.friend = m.friend
```

You have already created a FavoriteColorAndMovie table! You can reference this single table to find the favorite color and/or favorite movie of each friend without having to join the two separate tables, Favorite_Colors and Favorite_Movies, each time.

The CREATE TABLE IF NOT EXISTS method is best if the tables in your query (e.g. Favorite_Colors and Favorite_Movies) are not being continuously updated. The CREATE TABLE IF NOT EXISTS won’t do anything if the table already exists because it won’t be updated. So, if the source tables are continuously being updated (e.g. new friends are continuously added with their favorite colors and movies), then it is better to go with a different method of creating tables.

This other method of creating tables is the CREATE OR REPLACE TABLE method. Here is how that works:

```
CREATE OR REPLACE TABLE mydataset.FavoriteColorAndMovie
AS
SELECT
      friend,
      color,
      movie
FROM
      Favorite_Colors AS c
INNER JOIN
```

```
      Favorite_Movies AS m ON c.friend = m.friend
```

You may notice how the only difference between the two ways of creating tables is the first line of the query. This tells SQL what to do if the table already exists. CREATE TABLE IF NOT EXISTS will only create a table if it doesn't already exist. If it exists, then the query will run but won't do anything. This is a failsafe so that you don't accidentally overwrite a potentially important table. Alternatively, if you do need to overwrite a table, you can use CREATE OR REPLACE TABLE.

In summary, you would use CREATE TABLE IF NOT EXISTS if you are creating a static table that doesn't need to be updated. If your table needs to be continuously updated, you would use CREATE OR REPLACE TABLE instead.

Deleting tables

Now let's talk about deleting tables. This rarely happens, but it is important to learn about because, once the tables are deleted, the data contained in them may be lost. And, since the data is owned by the company you are working for, deleting a table could mean getting rid of something that is not yours.

Think about how you would treat a social media account that didn't belong to you. For example, if your company gave you access to their account. You can look at the posts and maybe (with permission) create your own post for them, but you wouldn't delete a social media post because this is the owner's account, not yours.

If you ever find yourself contemplating hitting the delete button for a table that you didn't create, make sure you verify the reasons why you're hitting delete and double check with your manager just in case.

But, if you ever need to delete a table, especially if it is one that you created and don't need anymore, you can delete a table in BigQuery by using this query:

```
DROP TABLE IF EXISTS mydataset.FavoriteColorAndMovie
```

DROP TABLE tells SQL to delete the table, and the IF EXISTS part makes sure that you don't get an error if the table doesn't exist. This is a failsafe, so that if the table exists, the table will be dropped. If the table doesn't exist and you run this query, then nothing will happen. So, either way, the failsafe works in your favor. Best practice is to add on the IF EXISTS.

Temporary tables

So far, you have learned how to create tables and when you would want to create them. Feel free to review that in the above section if you ever need a refresher. The tables that you create with the CREATE TABLE IF NOT EXISTS method or CREATE OR REPLACE TABLE method are permanent tables. They can be shared and seen by others, and they can be accessed later.

But, there might be situations where you don't need to create permanent tables. Remember: storing data in SQL costs the company money and resources. If you don't need a permanent table, you can create temporary tables instead. Temporary tables only exist within your session (or up to 24 hours depending on your SQL platform) and aren't shareable or viewable by others. Temporary tables exist only for you during your session. Think of temporary tables like a scratch pad where you can scribble out your calculations before putting down your final answer.

Let's start by outlining when you would want to create a permanent table vs. a temporary table.

Here are the three conditions for when you would want to create a permanent table. All three conditions should be met.

1. Complex query containing multiple JOINs
2. Result output is a table
3. You need to run the query frequently or on a regular basis

Alternatively, **Temporary tables** are used to break down complex queries into smaller increments. These complex queries can contain multiple JOINs but don't necessarily have to. You might want to use temporary tables if one or more of the following conditions apply:

- Slowly running query with multiple JOINs and WHERE statements
- Slowly running query containing GROUP BY and HAVING
- Nested queries (i.e. query within a query)
- If you need to do a calculation on top of a calculation (e.g. take sum per day then average across the day sums)

If any of the above conditions are met, then using a temporary table may speed up your query, which will make it easier for you to write the query, as well as make it easier for you to troubleshoot your query if something goes wrong.

Here is how to create a temporary table:

```sql
CREATE TEMP TABLE ExampleTable
AS
SELECT
      colors
FROM
      Favorite_Colors
```

Now, let's go back to the permanent table you created earlier:

```
CREATE TABLE IF NOT EXISTS mydataset.FavoriteColorAndMovie
AS
SELECT
      friend,
      color,
      movie
FROM
      Favorite_Colors AS c
INNER JOIN
      Favorite_Movies AS m ON c.friend = m.friend
```

This query works for a permanent table because all three of the conditions that we previously mentioned were met. But, for temporary tables, this query isn't a good idea. You are working with multiple JOINs but there are no WHERE statements, and the query runs really quickly since the Favorite_Colors and Favorite_Movies tables are relatively small (<100k rows).

Let's consider a different scenario when you would want to use temporary tables. Earlier you learned about GROUP BY and HAVING. If your query contains both clauses, such as the one below, then you might want to use temporary tables if your query is running slowly.

Temporary table query (if your query is running slowly):

```
SELECT
      occasion,
      SUM(tickets) AS total_tickets,
      COUNT(tickets) AS number_of_purchases
FROM
      purchases
GROUP BY
      occasion
HAVING
      SUM(tickets) > 5
```

In the above query, SQL has to perform three actions. First, it will group your table by occasion. Second, it will take the SUM() and COUNT() of the tickets column. Third, it will only show occasions with a SUM() of tickets that are greater than five. If the purchases table were much larger (1+ million rows) and you also had JOINs in this table, then your query would most likely run slowly. But, you can avoid this using the HAVING clause and speed up your query by breaking down the query into two steps using temporary tables.

First, you can do the GROUP BY aggregations:

```
CREATE TEMP TABLE TicketsByOccasion
AS
SELECT
      occasion,
      SUM(tickets) AS total_tickets,
      COUNT(tickets) AS number_of_purchases
FROM
      purchases
GROUP BY
      occasion;
```

Then, you can do the HAVING limitation as a WHERE condition:

```
SELECT
      occasion,
      total_tickets,
      number_of_purchases
FROM
      TicketsByOccasion
WHERE
      total_tickets > 5
```

There are three key things to notice here:

1. If you are running two queries at the same time, which you have to do with temporary tables (only available during that session), then you need a semicolon separating each query.
2. The first query is where you created the temporary table. The second query references this temporary table and its fields. That is why you can access the sum of tickets as total_tickets in the second query.
3. When creating table names, don't use spaces or the query will return an error. Best practice is to use camelcase capitalization when naming the table you are building. **Camelcase capitalization** means that you capitalize the start of each word without any spaces in between, just like a camel's hump. For example, the table TicketsByOccasion uses camelcase capitalization.

In conclusion, you don't have to use temporary tables, but they can be a really useful tool to help break down complex or complicated queries into smaller and more manageable steps.

Conclusion

This guide covers a lot of concepts, but you can come back to it again and again as you continue to write SQL queries on your own. As you have learned throughout this course, practice is an important part of the learning process, and the more that you practice working in SQL, the more you will make new discoveries. You can save this guide so that you can review and reference these functions and concepts as needed.