Udacity SQL Notes

This course will cover different uses of SQL but will focus on data analysis. SQL is used in virtually every industry so get read to superpower your skillset.

Parch & Posey Database

In this course, we will mostly be using the Parch & Posey database for our queries. Whenever we use a different database, we will let you know.

Parch & Posey (not a real company) is a paper company and the database includes sales data for their paper.

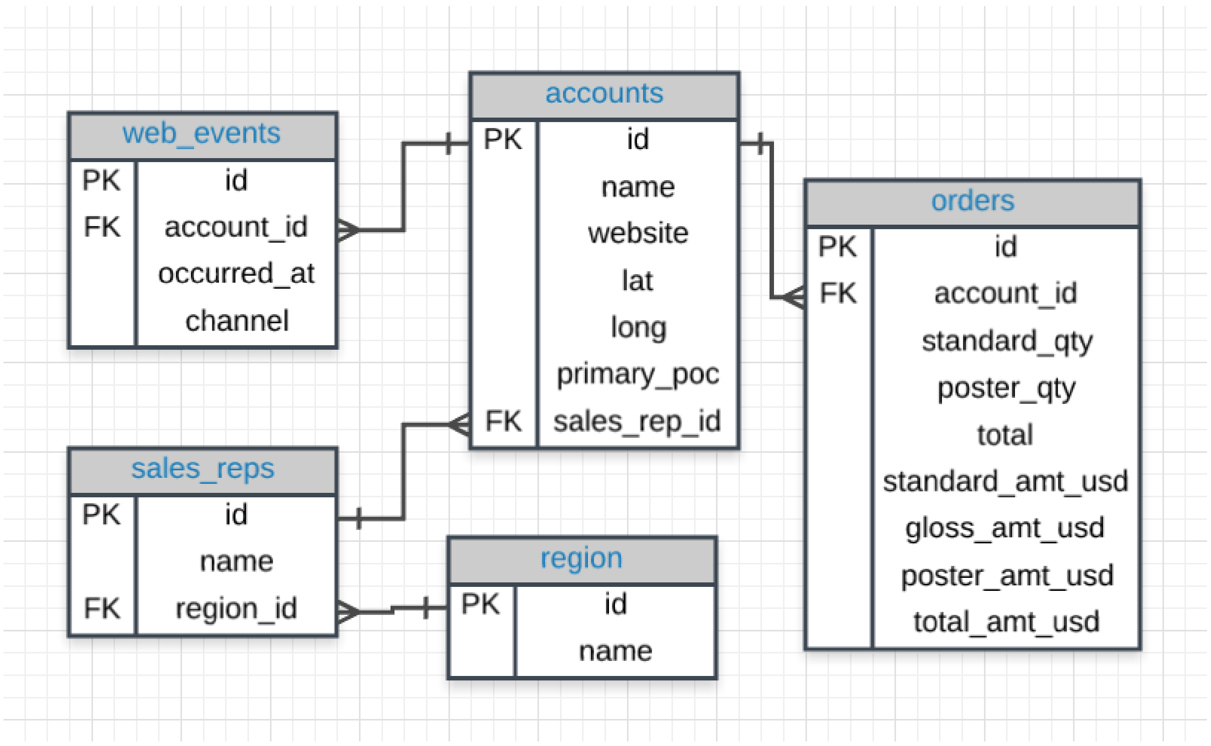
Using the sales data, you'll be able to put your SQL skills to work with data you would find in the real world.

Entity Relationship Diagrams

An **entity-relationship diagram** (ERD) is a common way to view data in a database. Below is the ERD for the database we will use from Parch & Posey. These diagrams help you visualize the data you are analyzing including:

1. The names of the tables.
2. The columns in each table.

You can think of each of the boxes below as a spreadsheet.



https://video.udacity-data.com/topher/2017/August/59821d7d_screen-shot-2017-08-02-at-11.14.25-am/screen-shot-2017-08-02-at-11.14.25-am.png

1. The way the tables work together.

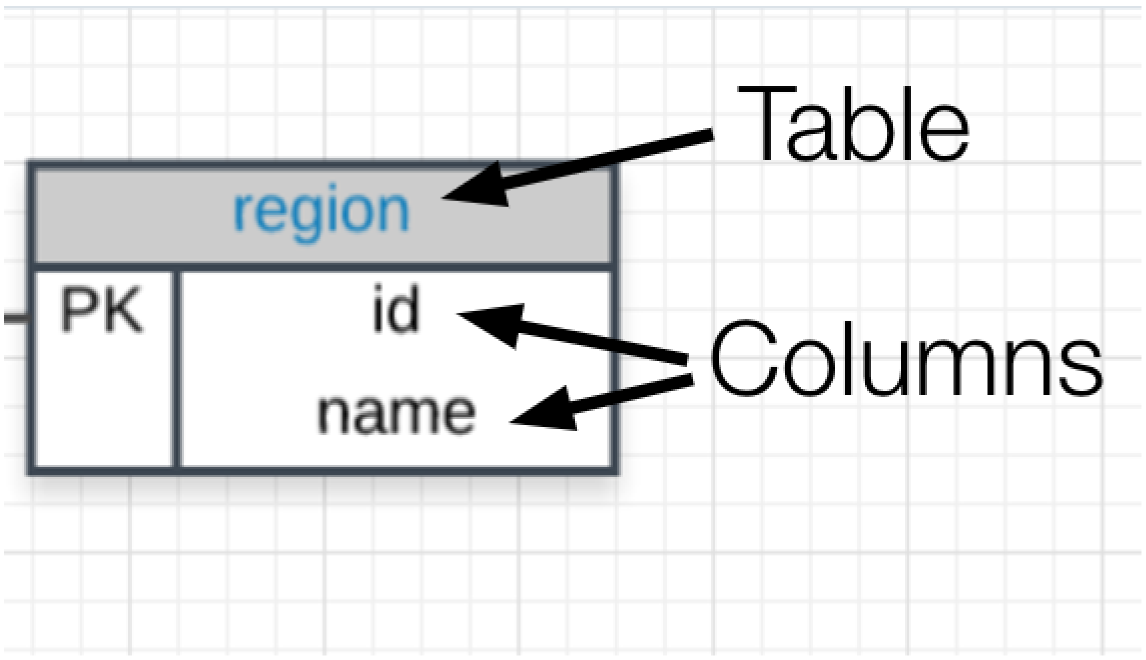
Parch & Posey Database ERD

What to Notice

In the Parch & Posey database there are five tables (essentially 5 spreadsheets):

1. **web_events**
2. **accounts**
3. **orders**
4. **sales_reps**
5. **region**

You can think of each of these tables as an individual spreadsheet. Then the columns in each spreadsheet are listed below the table name. For example, the **region** table has two columns: `id` and `name`. Alternatively, the **web_events** table has four columns.



https://video.udacity-data.com/topher/2017/August/59852269_screen-shot-2017-08-04-at-6.41.07-pm/screen-shot-2017-08-04-at-6.41.07-pm.png

Individual table diagram

The "**crow's foot**" that connects the tables together shows us how the columns in one table relate to the columns in another table. In this first lesson, you will be learning the basics of how to work with SQL to interact with a single table. In the next lesson, you will learn more about why these connections are so important for working with SQL and relational databases.

Introduction

Before we dive into writing Structured Query Language (SQL) queries, let's take a look at what makes SQL and the databases that utilize SQL so popular.

I think it is an important distinction to say that SQL is a **language**. Hence, the last word of SQ**L** being **language**. SQL is used all over the place beyond the databases we will utilize in this class. With that being said, SQL is most popular for its interaction with databases. For this class, you can think of a **database** as a bunch of excel spreadsheets all sitting in one place. Not all databases are a bunch of excel spreadsheets sitting in one place, but it is a reasonable idea for this class.

There are some major advantages to using **traditional relational databases,** which we interact with using SQL. The five most apparent are:

- SQL is easy to understand.

- Traditional databases allow us to access data directly.
- Traditional databases allow us to audit and replicate our data.
- SQL is a great tool for analyzing multiple tables at once.
- SQL allows you to analyze more complex questions than dashboard tools like Google Analytics.

You will experience these advantages first hand, as we learn to write SQL to interact with data.

Why Businesses Like Databases

1. **Data integrity is ensured -** only the data you want to be entered is entered, and only certain users are able to enter data into the database.
2. **Data can be accessed quickly -** SQL allows you to obtain results very quickly from the data stored in a database. Code can be optimized to quickly pull results.
3. **Data is easily shared -** multiple individuals can access data stored in a database, and the data is the same for all users allowing for consistent results for anyone with access to your database.

How Databases Store Data*

1. **Data in databases is stored in tables that can be thought of just like Excel spreadsheets.** For the most part, you can think of a database as a bunch of Excel spreadsheets. Each spreadsheet has rows and columns. Where each row holds data on a transaction, a person, a company, etc., while each column holds data pertaining to a particular aspect of one of the rows you care about like a name, location, a unique id, etc.
2. **All the data in the same column must match in terms of data type.** An entire column is considered quantitative, discrete, or as some sort of string. This means if you have one row with a string in a particular column, the entire column might change to a text data type. **This can be very bad if you want to do math with this column!**
3. **Consistent column types are one of the main reasons working with databases is fast.** Often databases hold **a LOT** of data. So, knowing that the columns are all of the same types of data means that obtaining data from a database can still be fast.

Types of Databases

SQL Databases

There are many different types of SQL databases designed for different purposes. In this course, we will use **Postgres** within the classroom, which is a popular open-source database with a very complete library of analytical functions. (Note: You do not need to install PostgreSQL on your computer unless you really want to. We provide SQL environments in the classroom for you to work in.)

Some of the most popular databases include:

1. MySQL
2. Access
3. Oracle
4. Microsoft SQL Server
5. Postgres

You can also write SQL within other programming frameworks like Python, Scala, and Hadoop.

Small Differences

Each of these SQL databases may have subtle differences in syntax and available functions – for example, MySQL doesn't have some of the functions for modifying dates as Postgres. **Most** of what you see with Postgres will be directly applicable to using SQL in other frameworks and database environments. For the differences that do exist, you should check the documentation. Most SQL environments have great documentation online that you can easily access with a quick Google search.

The article here compares three of the most common types of SQL: SQLite, PostgreSQL, and MySQL. Again, once you have learned how to write SQL in one environment, the skills are mostly transferable.

So with that, let's jump in!

Types of Statements

The key to SQL is understanding **statements**. A few statements include:

1. **CREATE TABLE** is a statement that creates a new table in a database.
2. **DROP TABLE** is a statement that removes a table in a database.

3. **SELECT** allows you to read data and display it. This is called a **query**.

The **SELECT** statement is the common statement used by analysts, and you will be learning all about them throughout this course!

SELECT & FROM

Here you were introduced to the SQL command that will be used in every query you write: SELECT … FROM ….

1. **SELECT** indicates which column(s) you want to be given the data for.
2. **FROM** specifies from which table(s) you want to select the columns. Notice the columns need to exist in this table.

If you want to be provided with the data from all columns in the table, you use "*", like so:

- SELECT * FROM orders

Note that using SELECT does not *create* a new table with these columns in the database, it just provides the data to you as the results, or output, of this command.

You will use this SQL SELECT statement in every query in this course, but you will be learning a few additional statements and operators that can be used along with them to ask more advanced questions of your data.

ORDER BY

The **ORDER BY** statement allows us to sort our results using the data in any column. If you are familiar with Excel or Google Sheets, using **ORDER BY** is similar to sorting a sheet using a column. A key difference, however, is that **using ORDER BY in a SQL query only has temporary effects, for the results of that query, unlike sorting a sheet by column in Excel or Sheets.**

In other words, when you use ORDER BY in a SQL query, your output will be sorted that way, but then the next query you run will encounter the unsorted data again. It's important to keep in mind that this is different than using common spreadsheet software, where sorting the spreadsheet by column actually alters the data in that sheet until you undo or change that sorting. This highlights the meaning and function of a SQL "query."

The **ORDER BY** statement always comes in a query after
the **SELECT** and **FROM** statements, but before the **LIMIT** statement. If you are

using the **LIMIT** statement, it will always appear last. As you learn additional commands, the order of these statements will matter more.

Pro-Tip

Remember `DESC` can be added after the column in your **ORDER BY** statement to sort in descending order, as the default is to sort in ascending order.

Solutions to previous ORDER BY questions

1. Write a query to return the 10 earliest orders in the **orders** table. Include the `id`, `occurred_at`, and `total_amt_usd`.

```
SELECT id, occurred_at, total_amt_usdFROM ordersORDER BY occurred_atLIMIT 10;
```

2. Write a query to return the top 5 **orders** in terms of the largest `total_amt_usd`. Include the `id`, `account_id`, and `total_amt_usd`.

```
SELECT id, account_id, total_amt_usd
FROM orders
ORDER BY total_amt_usd DESC
LIMIT 5;
```

3. Write a query to return the lowest 20 **orders** in terms of the smallest `total_amt_usd`. Include the `id`, `account_id`, and `total_amt_usd`.

```
SELECT id, account_id, total_amt_usd
FROM orders
ORDER BY total_amt_usd
LIMIT 20;
```

ORDER BY Part II

We can **ORDER BY** more than one column at a time. When you provide a list of columns in an **ORDER BY** command, the sorting occurs using the leftmost column in your list first, then the next column from the left, and so on. We still have the ability to flip the way we order using **DESC**.

ORDER BY Questions

1. Write a query that displays the order ID, account ID, and total dollar amount for all the orders, sorted first by the account ID (in ascending order), and then by the total dollar amount (in descending order).

   ```
   SELECT id, account_id, total_amt_usd
   FROM orders
   ORDER BY account_id, total_amt_usd DESC;
   ```

2. Now write a query that again displays order ID, account ID, and total dollar amount for each order, but this time sorted first by total dollar amount (in descending order), and then by account ID (in ascending order).

   ```
   SELECT id, account_id, total_amt_usd
   FROM orders
   ORDER BY total_amt_usd DESC, account_id;
   ```

   ## **WHERE Clause**

   Using the **WHERE** statement, we can display *subsets* of tables based on conditions that must be met. You can also think of the **WHERE** command as *filtering* the data.

   Common symbols used in **WHERE** statements include:

   1. `>` (greater than)
   2. `<` (less than)
   3. `>=` (greater than or equal to)
   4. `<=` (less than or equal to)
   5. `=` (equal to)
   6. `!=` (not equal to)

   ## **WHERE with Non-Numeric Data**

   The **WHERE** statement can also be used with non-numeric data. We can use the `=` and `!=` operators here. You need to be sure to use single quotes (just be careful if you have quotes in the original text) with the text data, not double quotes.

Commonly when we are using **WHERE** with non-numeric data fields, we use the **LIKE**, **NOT**, or **IN** operators. We will see those before the end of this lesson!

Arithmetic Operators

Derived Columns

Creating a new column that is a combination of existing columns is known as a **derived** column (or “calculated” or “computed” column). Usually, you want to give a name, or “alias,” to your new column using the **AS** keyword.

This derived column, and its alias, are generally only temporary, existing just for the duration of your query. The next time you run a query and access this table, the new column will not be there.

If you are deriving the new column from existing columns using a mathematical expression, then these familiar mathematical operators will be useful:

1. (Multiplication)
2. `+` (Addition)
3. `` (Subtraction)
4. `/` (Division)

Consider this example:

```
SELECT id, (standard_amt_usd/total_amt_usd)*100 AS std_percent, total_amt_usdFROM
 ordersLIMIT 10;
```

Here we divide the standard paper dollar amount by the total order amount to find the standard paper percent for the order, and use the **AS** keyword to name this new column “std_percent.” You can run this query on the next page if you’d like, to see the output.

Order of Operations

Remember PEMDAS from math class to help remember the order of operations? If not, check out this link as a reminder. The same order of operations applies when using arithmetic operators in SQL.

The following two statements have very different end results:

1. **Standard_qty / standard_qty + gloss_qty + poster_qty**
2. **standard_qty / (standard_qty + gloss_qty + poster_qty)**

It is likely that you mean to do the calculation as written in statement number 2!

Introduction to Logical Operators

In the next concepts, you will be learning about **Logical Operators**. **Logical Operators** include:

1. **LIKE** This allows you to perform operations similar to using **WHERE** and `=`, but for cases when you might **not** know **exactly** what you are looking for.
2. **IN** This allows you to perform operations similar to using **WHERE** and `=`, but for more than one condition.
3. **NOT** This is used with **IN** and **LIKE** to select all of the rows **NOT LIKE** or **NOT IN** a certain condition.
4. **AND & BETWEEN** These allow you to combine operations where all combined conditions must be true.
5. **OR** This allows you to combine operations where at least one of the combined conditions must be true.

LIKE

The **LIKE** operator is extremely useful for working with text. You will use **LIKE** within a **WHERE** clause. The **LIKE** operator is frequently used with `%`. The `%` tells us that we might want any number of characters leading up to a particular set of characters or following a certain set of characters. Remember you will need to use single quotes for the text you pass to the **LIKE** operator because these lower and uppercase letters are not the same within the string. Searching for **'T'** is not the same as searching for **'t'**. In other SQL environments (outside the classroom), you can use either single or double-quotes.

Hopefully, you are starting to get more comfortable with SQL, as we are starting to move toward operations that have more applications, but this also means we can't show you every use case. Hopefully, you can start to think about how you might use these types of applications to identify phone numbers from a certain region or an individual where you can't quite remember the full name.

Questions using the LIKE operator

Use the **accounts** table to find

1. All the companies whose names start with 'C'.

2. All companies whose names contain the string ‘one’ somewhere in the name.
3. All companies whose names end with ‘s’.

Solutions for LIKE operator

```
**SELECT** **name FROM** accounts**WHERE** **name** **LIKE** 'C%';
```

```
**SELECT** **name FROM** accounts**WHERE** **name** **LIKE** '%one%';
```

```
**SELECT** **name FROM** accounts**WHERE** **name** **LIKE** '%s';
```

IN

The **IN** operator is useful for working with both numeric and text columns. This operator allows you to use an `=`, but for more than one item of that particular column. We can check one, two, or many column values for which we want to pull data, but all within the same query. In the upcoming concepts, you will see the **OR** operator that would also allow us to perform these tasks, but the **IN** operator is a cleaner way to write these queries.

Expert Tip

In most SQL environments, although not in our Udacity’s classroom, you can use single or double quotation marks - and you may NEED to use double quotation marks if you have an apostrophe within the text you are attempting to pull.

Questions using IN operator

1. Use the **accounts** table to find the account `name`, `primary_poc`, and `sales_rep_id` for Walmart, Target, and Nordstrom.
2. Use the **web_events** table to find all information regarding individuals who were contacted via the **channel** of `organic` or `adwords`.

Solutions to IN Questions

```
**SELECT** **name**, primary_poc, sales_rep_id**FROM** accounts**WHERE** **name** **IN
** ('Walmart', 'Target', 'Nordstrom');
```

```
**SELECT** ***FROM** web_events**WHERE** channel **IN** ('organic', 'adwords');
```

NOT

The **NOT** operator is an extremely useful operator for working with the previous two operators we introduced: **IN** and **LIKE**. By specifying **NOT LIKE** or **NOT IN**, we can grab all of the rows that do not meet particular criteria.

Questions using the NOT operator

We can pull all of the rows that were excluded from the queries in the previous two concepts with our new operator.

1. Use the **accounts** table to find the account name, primary poc, and sales rep id for all stores except Walmart, Target, and Nordstrom.
2. Use the **web_events** table to find all information regarding individuals who were contacted via any method except using `organic` or `adwords` methods.

Use the **accounts** table to find:

1. All the companies whose names do not start with ‘C’.
2. All companies whose names do not contain the string ‘one’ somewhere in the name.
3. All companies whose names do not end with ‘s’.

Solutions to NOT IN Questions

```
**SELECT** **name**, primary_poc, sales_rep_id**FROM** accounts**WHERE** **name** **NO
T** **IN** ('Walmart', 'Target', 'Nordstrom');
```

```
**SELECT** ***FROM** web_events**WHERE** channel **NOT** **IN** ('organic', 'adword
s');
```

Solutions to NOT LIKE Questions

```
**SELECT** **name FROM** accounts**WHERE** **name** **NOT** **LIKE** 'C%';
```

```
**SELECT** **nameFROM** accounts**WHERE** **name** **NOT** **LIKE** '%one%';
```

```
**SELECT** **nameFROM** accounts**WHERE** **name** **NOT** **LIKE** '%s';
```

AND and BETWEEN

The **AND** operator is used within a **WHERE** statement to consider more than one logical clause at a time. Each time you link a new statement with an **AND**, you will need to specify the column you are interested in looking at. You may link as many statements as you would like to consider at the same time. This operator works with all of the operations we have seen so far including arithmetic operators (`+`, `*`, `-`, `/`). **LIKE**, **IN**, and **NOT** logic can also be linked together using the **AND** operator.

BETWEEN Operator

Sometimes we can make a cleaner statement using **BETWEEN** than we can use **AND**. Particularly this is true when we are using the same column for different parts of our **AND** statement. In the previous video, we probably should have used **BETWEEN**.

Instead of writing :

```
WHERE column >= 6 AND column <= 10
```

we can instead write, equivalently:

```
WHERE column BETWEEN 6 AND 10
```

1. Write a query that returns all the orders where the standard_qty is over 1000, the poster_qty is 0, and the gloss_qty is 0.

```
SELECT *FROM ordersWHERE standard_qty > 1000 AND poster_qty = 0 AND gloss_qty = 0;
```

2. Using the **accounts** table, find all the companies whose names do not start with 'C' and end with 's'.

```
SELECT nameFROM accountsWHERE name NOT LIKE 'C%' AND name LIKE '%s';
```

3. When you use the BETWEEN operator in SQL, do the results include the values of your endpoints, or not? Figure out the answer to this important question by writing a query that displays the order date and `gloss_qty` data for all orders where gloss_qty is between 24 and 29. Then look at your output to see if the BETWEEN operator included the begin and end values or not. **You should notice that there are a number of rows in the output of this query where the `gloss_qty`** values are 24 or 29. So the answer to the question is that yes, the BETWEEN operator in SQL is inclusive; that is, the endpoint values are included. So the BETWEEN statement in this query is equivalent to having written: "WHERE gloss_qty >= 24 AND gloss_qty <= 29."**

4. You will notice that using **BETWEEN** is tricky for dates! While **BETWEEN** is generally inclusive of endpoints, it assumes the time is at 00:00:00 (i.e. midnight) for dates. This is the reason why we set the right-side endpoint of the period at '2017-01-01'.

```
SELECT *FROM web_eventsWHERE channel IN ('organic', 'adwords') AND         occu
rred_at BETWEEN '2016-01-01'AND '2017-01-01'ORDER BY occurred_at DESC;
```

OR

Similar to the **AND** operator, the **OR** operator can combine multiple statements. Each time you link a new statement with an **OR**, you will need to specify the column you are interested in looking at. You may link as many statements as you would like to consider at the same time. This operator works with all of the operations we have seen so far including arithmetic operators (`+`, `*`, `-`, `/`), **LIKE**, **IN**, **NOT**, **AND**, and **BETWEEN** logic can all be linked together using the **OR** operator.

When combining multiple of these operations, we frequently might need to use parentheses to assure that logic we want to perform is being executed correctly.

Questions using the OR operator

1. Find list of **orders** ids where either `gloss_qty` or `poster_qty` is greater than 4000. Only include the `id` field in the resulting table.
2. Write a query that returns a list of **orders** where the `standard_qty` is zero and either the `gloss_qty` or `poster_qty` is over 1000.
3. Find all the company names that start with a ‘C’ or ‘W’, and the primary contact **contains** ‘ana’ or ‘Ana’, but it doesn’t contain ‘eana’.

```
SELECT id FROM orders WHERE gloss_qty>4000 OR poster_qty>4000;SELECT * FROM orders WHE
RE standard_qty=0 AND (gloss_qty>1000 OR poster_qty>1000);SELECT * FROM accountsWHERE
 (name LIKE 'C%' OR name LIKE 'W%') AND            ((primary_poc LIKE '%ana%' OR prima
ry_poc LIKE '%Ana%') AND                primary_poc NOT LIKE '%eana%');
```

Database Normalization

When creating a database, it is really important to think about how data will be stored. This is known as **normalization**, and it is a huge part of most SQL classes. If you are in charge of setting up a new database, it is important to have a thorough understanding of database **normalization**.

There are essentially three ideas that are aimed at database normalization:

1. Are the tables storing logical groupings of the data?
2. Can I make changes in a single location, rather than in many tables for the same information?
3. Can I access and manipulate data quickly and efficiently?

This is discussed in detail here.

However, most analysts are working with a database that was already set up with the necessary properties in place. As analysts of data, you don’t really need to think too much about data **normalization**. You just need to be able to pull the data from the database, so you can start making insights. This will be our focus in this lesson.

Introduction to JOINs

This entire lesson will be focused on **JOIN**s. The whole purpose of **JOIN** statements is to allow us to pull data from more than one table at a time.

Again - **JOIN**s are useful for allowing us to pull data from multiple tables. This is both simple and powerful all at the same time.

With the addition of the **JOIN** statement to our toolkit, we will also be adding the **ON** statement.

We use **ON** clause to specify a **JOIN** condition which is a logical statement to combine the table in `FROM` and `JOIN` statements.

```
SELECT orders.*,       accounts.*FROM ordersJOIN accountsON orders.account_id = accoun
ts.id;
```

Write Your First JOIN

Below we see an example of a query using a **JOIN** statement. Let's discuss what the different clauses of this query mean.

```
SELECT orders.*FROM ordersJOIN accountsON orders.account_id = accounts.id;
```

As we've learned, the **SELECT** clause indicates which column(s) of data you'd like to see in the output (For Example, orders.* gives us all the columns in the orders table in the output). The **FROM** clause indicates the first table from which we're pulling data, and the **JOIN** indicates the second table. The **ON** clause specifies the column on which you'd like to merge the two tables together.

What to Notice

We are able to pull data from two tables:

1. **orders**
2. **accounts**

Above, we are only pulling data from the **orders** table since in the SELECT statement we only reference columns from the **orders** table.

The **ON** statement holds the two columns that get linked across the two tables. This will be the focus of the next concepts.

Additional Information

If we wanted to only pull individual elements from either the **orders** or **accounts** table, we can do this by using the exact same information in the **FROM** and **ON** statements. However, in your **SELECT** statement, you will need to know how to specify tables and columns in the **SELECT** statement:

1. The table name is always **before** the period.
2. The column you want from that table is always **after** the period.

For example, if we want to pull only the **account name** and the **dates** in which that account placed an order, but none of the other columns, we can do this with the following query:

```
SELECT accounts.name, orders.occurred_at
FROM orders
JOIN accounts
ON orders.account_id = accounts.id;
```

This query only pulls two columns, not all the information in these two tables. Alternatively, the below query pulls all the columns from *both* the **accounts** and **orders** table.

```
SELECT *
FROM orders
JOIN accounts
ON orders.account_id = accounts.id;
```

And the first query you ran pull all the information from *only* the **orders** table:

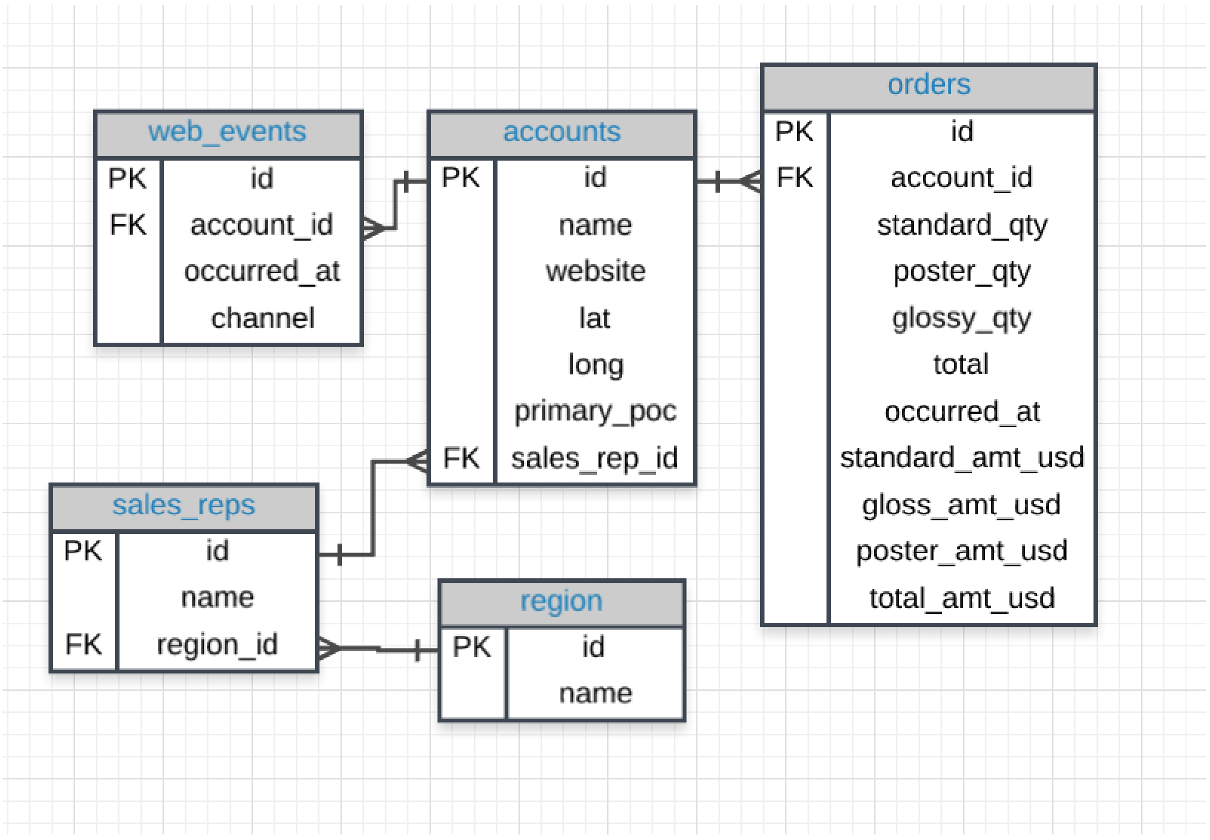
```
SELECT orders.*
FROM orders
JOIN accounts
ON orders.account_id = accounts.id;
```

Joining tables allows you access to each of the tables in the **SELECT** statement through the table name, and the columns will always follow a **.** after the table name.

Entity Relationship Diagrams

From the last lesson, you might remember that an **entity-relationship diagram** (ERD) is a common way to view data in a database. It is also a key element to understanding how we can pull data from multiple tables.

It will be beneficial to have an idea of what the ERD looks like for Parch & Posey handy.



https://video.udacity-data.com/topher/2017/October/59e946e7_erd/erd.png

Entity Relationship Diagram (ERD) for Parch & Posey Database

Tables & Columns

In the Parch & Posey database there are 5 tables:

1. **web_events**
2. **accounts**
3. **orders**
4. **sales_reps**
5. **region**

You will notice some of the columns in the tables have **PK** or **FK** next to the column name, while other columns don't have a label at all.

If you look a little closer, you might notice that the **PK** is associated with the first column in every table. The **PK** here stands for **the primary key**. **A primary key exists in every table, and it is a column that has a unique value for every row.**

If you look at the first few rows of any of the tables in our database, you will notice that this first, **PK**, column is always unique. For this database, it is always called `id`, but that is not true of all databases.

Keys

Primary Key (PK)

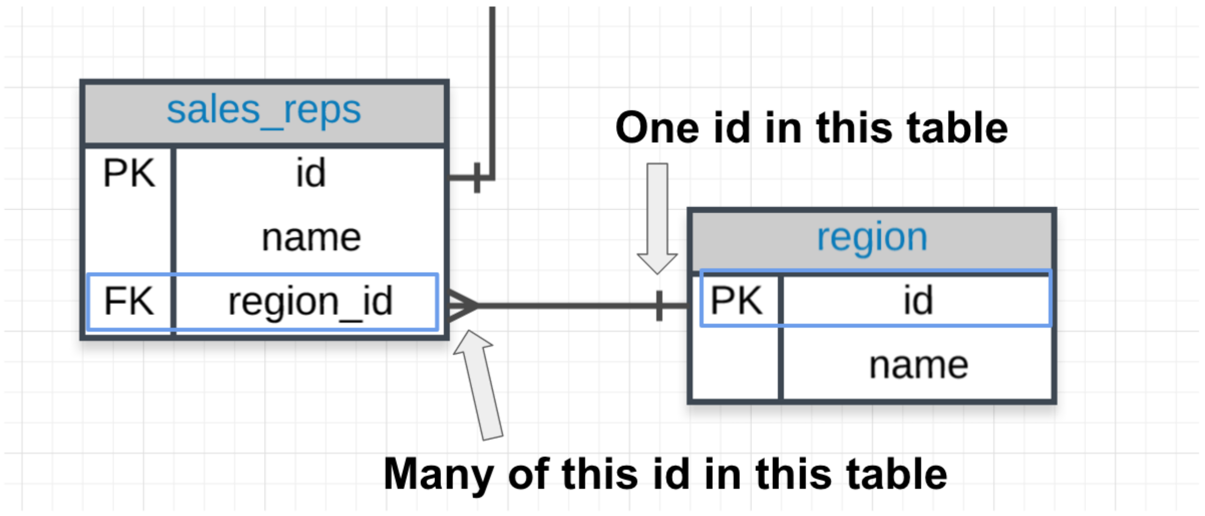
A **primary key** is a unique column in a particular table. This is the first column in each of our tables. Here, those columns are all called **id**, but that doesn't necessarily have to be the name. **It is common that the primary key is the first column in our tables in most databases.**

Foreign Key (FK)

A **foreign key** is a column in one table that is a primary key in a different table. We can see in the Parch & Posey ERD that the foreign keys are:

1. **region_id**
2. **account_id**
3. **sales_rep_id**

Each of these is linked to the **primary key** of another table. An example is shown in the image below:



https://video.udacity-data.com/topher/2017/August/598d2378_screen-shot-2017-08-10-at-8.23.48-pm/screen-shot-2017-08-10-at-8.23.48-pm.png

View of Primary Key & Foreign Key Relationship

Primary - Foreign Key Link

In the above image you can see that:

1. The **region_id** is the foreign key.
2. The region_id is **linked** to id **-** this is the primary-foreign key link that connects these two tables.
3. The crow's foot shows that the **FK** can actually appear in many rows in the **sales_reps** table.
4. While the single line is telling us that the **PK** shows that id appears only once per row in this table.

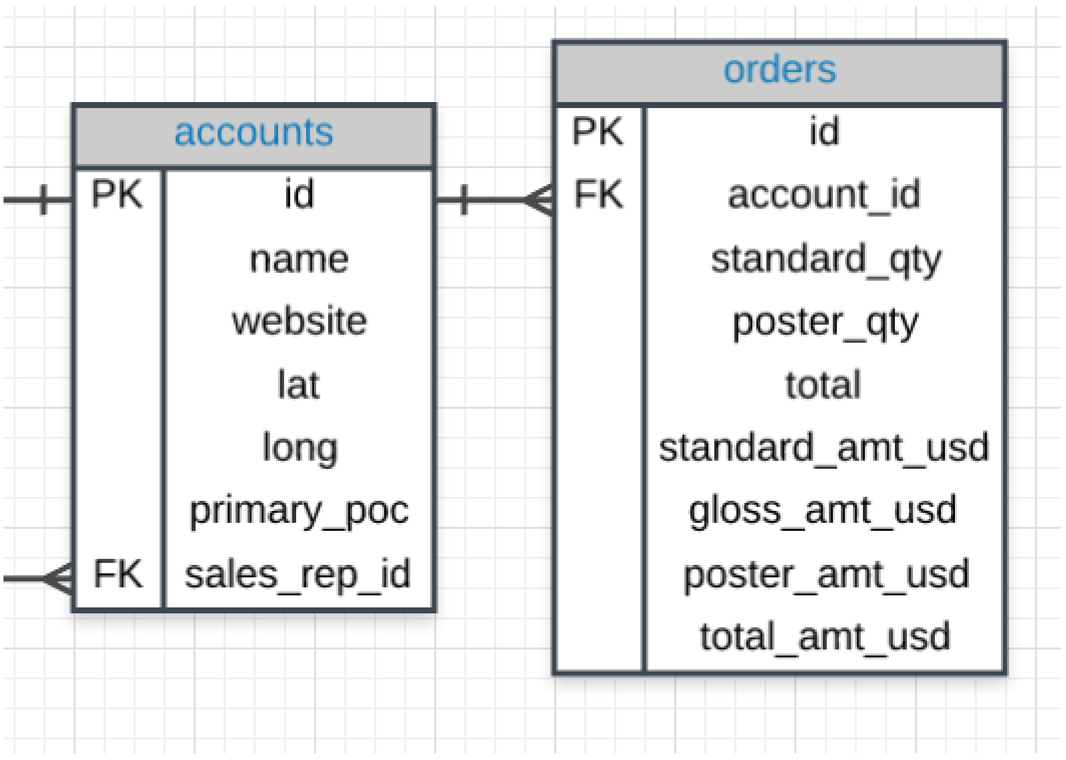
If you look through the rest of the database, you will notice this is always the case for a primary-foreign key relationship. In the next concept, you can make sure you have this down!

JOIN Revisited

Let's look back at the first JOIN you wrote.

```
SELECT orders.*
FROM orders
JOIN accounts
ON orders.account_id = accounts.id;
```

Here is the ERD for these two tables:



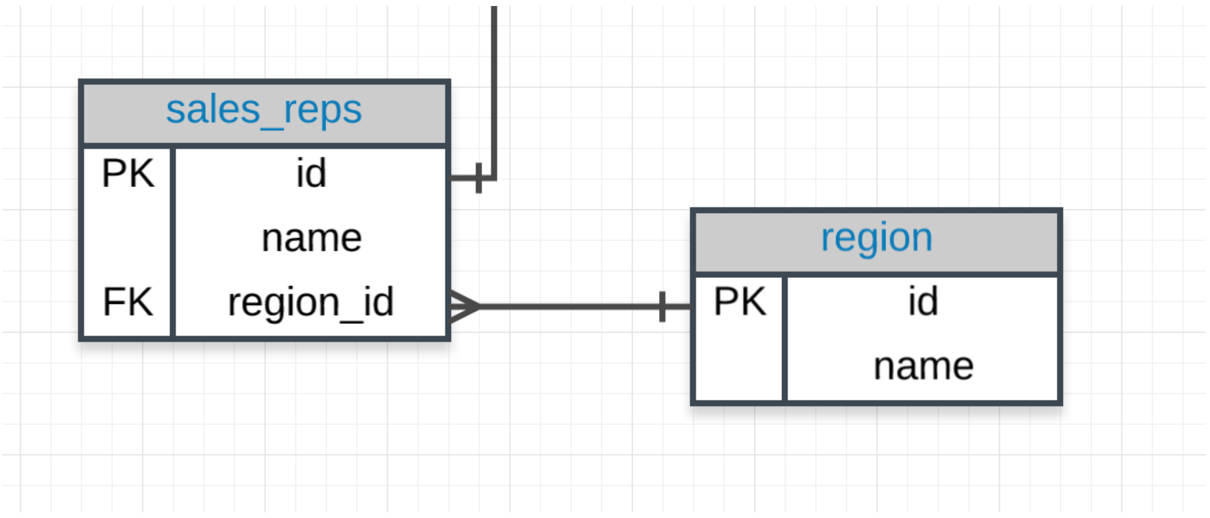
https://video.udacity-data.com/topher/2017/August/598dfda7_screen-shot-2017-08-11-at-11.54.30-am/screen-shot-2017-08-11-at-11.54.30-am.png

ERD for Accounts & Orders tables

Notice

Notice our SQL query has the two tables we would like to join - one in the **FROM** and the other in the **JOIN**. Then in the **ON**, we will **ALWAYs** have the **PK** equal to the **FK**:

The way we join any two tables is in this way: linking the **PK** and **FK** (generally in an **ON** statement).

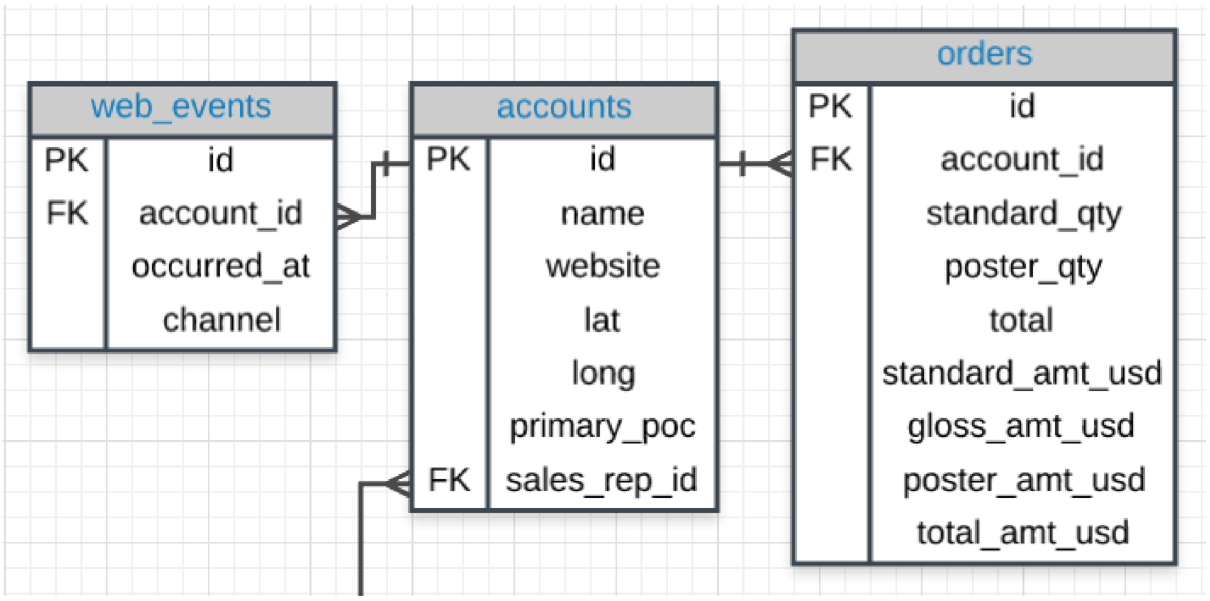


https://video.udacity-data.com/topher/2017/August/598e0b9e_screen-shot-2017-08-10-at-8.10.13-pm/screen-shot-2017-08-10-at-8.10.13-pm.png

ERD for Sales Reps & Region tables

JOIN More than Two Tables

This same logic can actually assist in joining more than two tables together. Look at the three tables below.



https://video.udacity-data.com/topher/2017/August/598e2e15_screen-shot-2017-08-11-at-3.21.34-pm/screen-shot-2017-08-11-at-3.21.34-pm.png

ERD for Web Events, Accounts & Orders Tables

The Code

If we wanted to join all three of these tables, we could use the same logic. The code below pulls all of the data from all of the joined tables.

```
SELECT *FROM web_eventsJOIN accountsON web_events.account_id = accounts.idJOIN ordersO
N accounts.id = orders.account_id
```

Alternatively, we can create a **SELECT** statement that could pull specific columns from any of the three tables. Again, our **JOIN** holds a table, and **ON** is a link for our **PK** to equal the **FK**.

To pull specific columns, the **SELECT** statement will need to specify the table that you are wishing to pull the column from, as well as the column name. We could pull only three columns in the above by changing the select statement to the below, but maintaining the rest of the JOIN information:

```
SELECT web_events.channel, accounts.name, orders.total;
```

We could continue this same process to link all of the tables if we wanted. For efficiency reasons, we probably don't want to do this unless we actually need information from all of the tables.

Alias

When we **JOIN** tables together, it is nice to give each table an **alias**. Frequently an alias is just the first letter of the table name. You actually saw something similar for column names in the **Arithmetic Operators** concept.

Example:

```
FROM tablename AS t1
JOIN tablename2 AS t2
```

Before, you saw something like:

```
SELECT col1 + col2 AS total, col3
```

Frequently, you might also see these statements without the **AS** statement. Each of the above could be written in the following way instead, and they would still produce the **exact same results**:

```
FROM tablename t1
JOIN tablename2 t2
```

and

```
SELECT col1 + col2 total, col3
```

Aliases for Columns in Resulting Table

While aliasing tables is the most common use case. It can also be used to alias the columns selected to have the resulting table reflect a more readable name.

Example:

```
Select t1.column1 aliasname, t2.column2 aliasname2
FROM tablenameAS t1
JOIN tablename2AS t2
```

The alias name fields will be what shows up in the returned table instead of t1.column1 and t2.column2

aliasname	aliasname2
example row	example row
example row	example row

💡 `If you have two or more columns in your SELECT that have the same name after the table name such as accounts.name and sales_reps.name you will need to alias them. Otherwise it will only show one of the columns. You can alias them like accounts.name AS AcountName, sales_rep.name AS SalesRepName`

Motivation for Other JOINs

Expert Tip

You have had a bit of an introduction to these **one-to-one** and **one-to-many** relationships when we introduced **PK**s and **FK**s. Notice, traditional databases do not allow for **many-to-many** relationships, as these break the schema down pretty quickly. A very good answer is provided here.

The types of relationships that exist in a database matter less to analysts, but you do need to understand why you would perform different types of **JOIN**s, and what data you are pulling from the database. These ideas will be expanded upon in the next concepts.

LEFT and RIGHT JOINs

LEFT JOIN

LEFT JOIN will pull the data that belongs to FROM table which might be or might not be present in the JOIN table.

```
SELECT a.id, a.name, o.totalFROM orders oLEFT JOIN accounts aON o.account_id = a.id
```

RIGHT JOIN

RIGHT JOIN will pull all the data from the table mentioned in the JOIN clause.

```
SELECT a.id, a.name, o.totalFROM orders oRIGHT JOIN accounts aON o.account_id = a.id
```

- ***Quick Note**** You might see the SQL syntax of `LEFT OUTER **JOIN**` OR `RIGHT OUTER **JOIN**` These are the exact same commands as the **LEFT JOIN** and **RIGHT JOIN**.

💡 LEFT JOIN and RIGHT JOIN will return the same results if we swap the tables in the FROM and JOIN clauses.

JOINs

JOIN statements pulls all the same rows as an **INNER JOIN**, which you saw by just using **JOIN**, but they also potentially pull some additional rows.

If there is not matching information in the **JOIN** ed table, then you will have columns with empty cells. These empty cells introduce a new data type called **NULL**. You will learn about **NULL**s in detail in the next lesson, but for now you have a quick introduction as you can consider any cell without data as **NULL**.

- ***Other JOIN Notes JOIN Check In INNER JOINs**** Notice **every** JOIN we have done up to this point has been an **INNER JOIN**. That is, we have always pulled rows only if they exist as a match across two tables. Our new **JOIN**s allow us to pull rows that might only exist in one of the two tables. This will introduce a new data type called **NULL**. This data type will be discussed in detail in the next lesson. ****OUTER JOINS**** The last type of join is a full outer join. This will return the inner join result set, as well as any unmatched rows from either of the two tables being joined. Again this returns rows that **do not match** one another from the two tables. The use cases for a full outer join are **very rare**. You can see examples of outer joins at the link here and a description of the rare use cases here. We will not spend time on these given the few instances you might need to use them. Similar to the above, you might see the language **FULL OUTER JOIN**, which is the same as **OUTER JOIN**.

Untitled

JOINs and Filtering

Query1

```
SELECT orders.*, accounts.*
FROM orders
LEFT JOIN accounts
ON orders.account_id = accounts.id
WHERE accounts.sales_rep_id = 321500
```

A simple rule to remember is that, when the database executes this query, it executes the join and everything in the **ON** clause first. Think of this as building the new result set. That result set is then filtered using the **WHERE** clause.

Query2

```
SELECT orders.*, accounts.*
FROM orders
LEFT JOIN accounts
        ON orders.account_id = accounts.id
        AND accounts.sales_rep_id = 321500
```

The fact that this example is a left join is important. Because inner joins only return the rows for which the two tables match, moving this filter to the **ON** clause of an inner join will produce the same result as keeping it in the **WHERE** clause.

💡 If you are LEFT/RIGHT/FULL OUTER joining two columns with a condition, do not use WHERE clause; because first the joined table is formed (result set) and then WHERE filters the data. Instead use AND along with ON. This makes a new combined joining condition and then forms the result set.

QUIZ

1. Provide a table that provides the **region** for each **sales_rep** along with their associated **accounts**. This time only for the `Midwest` region. Your final table should include three columns: the region **name**, the sales rep **name**, and the account **name**. Sort the accounts alphabetically (A-Z) according to the account name.
2. Provide a table that provides the **region** for each **sales_rep** along with their associated **accounts**. This time only for accounts where the sales rep has a first name starting with `S` and in the `Midwest` region. Your final table should include three columns: the region **name**, the sales rep **name**, and the account **name**. Sort the accounts alphabetically (A-Z) according to the account name.
3. Provide a table that provides the **region** for each **sales_rep** along with their associated **accounts**. This time only for accounts where the sales rep has a **last** name starting with `K` and in the `Midwest` region. Your final table should include three columns: the region **name**, the sales rep **name**, and the account **name**. Sort the accounts alphabetically (A-Z) according to the account name.
4. Provide the **name** for each region for every **order**, as well as the account **name** and the **unit price** they paid (total_amt_usd/total) for the order. However, you should only provide the results if the **standard order quantity** exceeds `100`. Your final table should have 3 columns: **region name**, **account name**, and **unit price**.

```
SELECT r.name region, a.name account, o.total_amt_usd/(o.total + 0.01) unit_price
FROM region r
JOIN sales_reps s ON s.region_id = r.id
JOIN accounts a ON a.sales_rep_id = s.id
JOIN orders o ON o.account_id = a.id
WHERE o.standard_qty > 100;
```

5. Provide the **name** for each region for every **order**, as well as the account **name** and the **unit price** they paid (total_amt_usd/total) for the order. However, you should only provide the results if the **standard order**

quantity exceeds `100` and the **poster order quantity** exceeds `50`. Your final table should have 3 columns: **region name**, **account name**, and **unit price**. Sort for the smallest **unit price** first.

```
SELECT r.name region, a.nameaccount, o.total_amt_usd/(o.total + 0.01) unit_price
FROM region r
JOIN sales_reps s
ON s.region_id = r.idJOIN accounts a
ON a.sales_rep_id = s.idJOIN orders o
ON o.account_id = a.idWHERE o.standard_qty > 100AND o.poster_qty > 50
ORDERBY unit_price;
```

6. Provide the **name** for each region for every **order**, as well as the account **name** and the **unit price** they paid (total_amt_usd/total) for the order. However, you should only provide the results if the **standard order quantity** exceeds `100` and the **poster order quantity** exceeds `50`. Your final table should have 3 columns: **region name**, **account name**, and **unit price**. Sort for the largest **unit price** first.

```
SELECT r.name region, a.nameaccount, o.total_amt_usd/(o.total + 0.01) unit_price
FROM region r
JOIN sales_reps s
ON s.region_id = r.idJOIN accounts a
ON a.sales_rep_id = s.idJOIN orders o
ON o.account_id = a.idWHERE o.standard_qty > 100AND o.poster_qty > 50
ORDERBY unit_priceDESC;
```

7. What are the different **channel**s used by **account id** `1001`? Your final table should have only 2 columns: **account name** and the different **channel**s. You can try **SELECT DISTINCT** to narrow down the results to only the unique values.

```
SELECTDISTINCT a.name, w.channel
FROM accounts a
JOIN web_events w
ON a.id = w.account_id
WHERE a.id = '1001';
```

8. Find all the orders that occurred in `2015`. Your final table should have 4 columns: **occurred_at**, **account name**, **order total**, and **order total_amt_usd**.

```
SELECT o.occurred_at, a.name, o.total, o.total_amt_usd
FROM accounts a
JOIN orders o
```

```
ON o.account_id = a.idWHERE o.occurred_atBETWEEN '01-01-2015'AND '01-01-2016'
ORDERBY o.occurred_atDESC;
```

Introduction to Aggregation

In the following concepts, you will be learning in detail about each of the aggregate functions mentioned as well as some additional aggregate functions that are used in SQL all the time.

Introduction to NULLs

NULLs are a datatype that specifies where no data exists in SQL. They are often ignored in our aggregation functions, which you will get a first look at in the next concept using **COUNT**.

NULLs and Aggregation

Notice that **NULL**s are different than a zero - they are cells where data does not exist.

When identifying **NULL**s in a **WHERE** clause, we write **IS NULL** or **IS NOT NULL**. We don't use `=`, because **NULL** isn't considered a value in SQL. Rather, it is a property of the data.

NULLs - Expert Tip

There are two common ways in which you are likely to encounter **NULL**s:

- **NULL**s frequently occur when performing a **LEFT** or **RIGHT JOIN**. You saw in the last lesson - when some rows in the left table of a left join are not matched with rows in the right table, those rows will contain some **NULL** values in the result set.
- **NULL**s can also occur from simply missing data in our database.

```
**SELECT** ***FROM** accounts**WHERE** primary_poc **IS** **NOT** NULL
```

First Aggregation - COUNT

COUNT the Number of Rows in a Table

Try your hand at finding the number of rows in each table. Here is an example of finding all the rows in the **accounts** table.

```
SELECT COUNT(*)
FROM accounts;
```

But we could have just as easily chosen a column to drop into the aggregation function:

```
SELECT COUNT(accounts.id)
FROM accounts;
```

These two statements are equivalent, but this isn't always the case. If the `account.id` contains NULLs, then COUNT(*) will return higher value than COUNT(account.id).

COUNT & NULLs

Notice that **COUNT** does not consider rows that have **NULL** values. Therefore, this can be useful for quickly identifying which rows have missing data. You will learn **GROUP BY** in an upcoming concept, and then each of these aggregators will become much more useful.

```
SELECT COUNT (*) AS account_count
FROM accounts
```

```
SELECT COUNT(id) AS account_id_count
FROM accounts
```

```
SELECT COUNT(primary_poc) AS account_primary_poc_count
FROM accounts
```

SUM

Unlike **COUNT**, you can only use **SUM** on numeric columns. However, **SUM** will ignore **NULL** values, as do the other aggregation functions you will see in the upcoming lessons.

Aggregation Reminder

An important thing to remember: **aggregators only aggregate vertically - the values of a column**. If you want to perform a calculation across rows, you would do this with simple arithmetic.

We saw this in the first lesson if you need a refresher, but the quiz in the next concept should assure you still remember how to aggregate across rows.

```
**SELECT** **SUM**(standard_qty) **AS** standard,       **SUM**(gloss_qty) **AS** glos
s,       **SUM**(poster_qty) **AS** poster**FROM** orders
```

MIN & MAX

Notice that **MIN** and **MAX** are aggregators that again ignore **NULL** values. Check the expert tip below for a cool trick with **MAX** & **MIN**.

Expert Tip

Functionally, **MIN** and **MAX** are 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 early in the alphabet as possible. As you might suspect, **MAX** does the opposite—it returns the highest number, the latest date, or the non-numerical value closest alphabetically to “Z.”

```
**SELECT** **MIN**(standard_qty) **AS** standard_min,       **MIN**(gloss_qty) **AS**
 gloss_min,       **MIN**(poster_qty) **AS** poster_min,       **MAX**(standard_qty) *
*AS** standard_max,       **MAX**(gloss_qty) **AS** gloss_max,       **MAX**(poster_qt
y) **AS** poster_max**FROM**   orders;
```

AVG

Similar to other software **AVG** returns the mean of the data - that is the sum of all of the values in the column divided by the number of values in a column. This aggregate function again ignores the **NULL** values in both the numerator and the denominator.

If you want to count **NULL**s as zero, you will need to use **SUM** and **COUNT**. However, this is probably not a good idea if the **NULL** values truly just represent unknown values for a cell.

💡 AVG(X) will return different value than SUM(X)/COUNT(*); and value equal to SUM(X)/COUNT(X).

MEDIAN - Expert Tip

One quick note that a median might be a more appropriate measure of center for this data, but finding the median happens to be a pretty difficult thing to get using SQL alone — so difficult that finding a median is occasionally asked as an interview question.

```
**SELECT** **AVG**(standard_qty) **AS** standard_avg,       **AVG**(gloss_qty) **AS**
 gloss_avg,       **AVG**(poster_qty) **AS** poster_avg**FROM** orders;
```

GROUP BY

```
**SELECT** account_id,       **SUM**(standard_qty) **AS** standard,       **SUM**(glos
s_qty) **AS** gloss,       **SUM**(poster_qty) **AS** poster**FROM** orders**GROUP** *
*BY** account_id**ORDER** **BY** account_id;
```

- **GROUP BY** can be used to aggregate data within subsets of the data. For example, grouping for different accounts, different regions, or different sales representatives.
- Any column in the **SELECT** statement that is not within an aggregator must be in the **GROUP BY** clause.
- The **GROUP BY** always goes between **WHERE** and **ORDER BY**.
- **ORDER BY** works like **SORT** in spreadsheet software.

GROUP BY - Expert Tip

Before we dive deeper into aggregations using **GROUP BY** statements, it is worth noting that SQL evaluates the aggregations before the **LIMIT** clause. If you don't group by any columns, you'll get a 1-row result—no problem there. If you group by a column with enough unique values that it exceeds the **LIMIT** number, the aggregates will be calculated, and then some rows will simply be omitted from the results.

This is actually a nice way to do things because you know you're going to get the correct aggregates. If SQL cuts the table down to 100 rows, then performed the aggregations, your results would be substantially different. The above query's results exceed 100 rows, so it’s a perfect example. In the next concept, use the SQL environment to try removing the **LIMIT** and running it again to see what changes.

GROUP BY Part II

- You can **GROUP BY** multiple columns at once. This is often useful to aggregate across a number of different segments.
- The order of columns listed in the **ORDER BY** clause does make a difference. You are ordering the columns from left to right.

GROUP BY - Expert Tips

```
**SELECT** account_id,       channel,       **COUNT**(**id**) **as** **eventsFROM** we
b_events**GROUP** **BY** account_id, channel**ORDER** **BY** account_id, channel;**SEL
ECT** account_id,       channel,       **COUNT**(**id**) **as** **eventsFROM** web_eve
nts**GROUP** **BY** account_id, channel**ORDER** **BY** account_id, channel **DESC;**
```

- The order of column names in your **GROUP BY** clause doesn’t matter—the results will be the same regardless. If we run the same query and reverse the order in the **GROUP BY** clause, you can see we get the same results.
- 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.

💡 A reminder here that any column that is not within an aggregation must show up in your GROUP BY statement. If you forget, you will likely get an error. However, in the off chance that your query does work, you might not like the results!

1. For each account, determine the average amount of each type of paper they purchased across their orders. Your result should have four columns - one for the account **name** and one for the average spent on each of the paper types.

```
SELECT a.name,
            AVG(o.standard_qty) avg_stand,
            AVG(o.gloss_qty) avg_gloss,
            AVG(o.poster_qty) avg_post
FROM accounts a
JOIN orders o
ON a.id = o.account_id
GROUP BY a.name;
```

QUIZ

1. For each account, determine the average amount spent per order on each paper type. Your result should have four columns - one for the account **name** and one for the average amount spent on each paper type.

```
SELECT a.name,AVG(o.standard_amt_usd) avg_stand,AVG(o.gloss_amt_usd) avg_gloss,AVG
(o.poster_amt_usd) avg_post
FROM accounts a
JOIN orders o
ON a.id = o.account_id
GROUP BY a.name;
```

2. Determine the number of times a particular **channel** was used in the **web_events** table for each **sales rep**. Your final table should have three columns - the **name of the sales rep**, the **channel**, and the number of occurrences. Order your table with the highest number of occurrences first.

```
SELECT s.name, w.channel,COUNT(*) num_events
FROM accounts a
JOIN web_events w
ON a.id = w.account_id
JOIN sales_reps s
ON s.id = a.sales_rep_id
GROUP BY s.name, w.channel
ORDER BY num_events DESC;
```

3. Determine the number of times a particular **channel** was used in the **web_events** table for each **region**. Your final table should have three columns - the **region name**, the **channel**, and the number of occurrences. Order your table with the highest number of occurrences first.

```
SELECT r.name, w.channel,COUNT(*) num_events
FROM accounts a
JOIN web_events w
ON a.id = w.account_id
JOIN sales_reps s
ON s.id = a.sales_rep_id
JOIN region r
ON r.id = s.region_id
GROUP BY r.name, w.channel
ORDER BY num_events DESC;
```

4. Determine the number of times a particular **channel** was used in the **web_events** table for each **region**. Your final table should have three columns - the **region name**, the **channel**, and the number of occurrences. Order your table with the highest number of occurrences first.

```
SELECT r.name, w.channel,COUNT(*) num_events
FROM accounts a
JOIN web_events w
ON a.id = w.account_id
JOIN sales_reps s
ON s.id = a.sales_rep_id
JOIN region r
ON r.id = s.region_id
GROUP BY r.name, w.channel
ORDER BY num_events DESC;
```

DISTINCT

DISTINCT is always used in **SELECT** statements, and it provides the unique rows for all columns written in the **SELECT** statement. Therefore, you only use **DISTINCT** once in any particular **SELECT** statement.

You could write:

```
SELECT DISTINCT column1, column2, column3
FROM table1;
```

which would return the unique (or **DISTINCT**) rows across all three columns.

You would **not** write:

```
SELECT DISTINCT column1,DISTINCT column2,DISTINCT column3
FROM table1;
```

You can think of **DISTINCT** the same way you might think of the statement "unique".

💡 DISTINCT checks the unique rows. The values in the particular column might be same, but 2 rows with same value in each column specified SELECT clause will be omitted.

DISTINCT - Expert Tip

It’s worth noting that using **DISTINCT**, particularly in aggregations, can slow your queries down quite a bit.

QUIZ

1. Use **DISTINCT** to test if there are any accounts associated with more than one region.

The below two queries have the same number of resulting rows (351), so we know that every account is associated with only one region. If each account was associated with more than one region, the first query should have returned more rows than the second query.

```
SELECT a.idas "account id", r.idas "region id",
a.nameas "account name", r.nameas "region name"
FROM accounts a
JOIN sales_reps s
ON s.id = a.sales_rep_id
JOIN region r
ON r.id = s.region_id;
```

and

```
SELECT DISTINCT id,name
FROM accounts;
```

1. Have any **sales reps** worked on more than one account?

Actually, all of the sales reps have worked on more than one account. The fewest number of accounts any sales rep works on is 3. There are 50 sales reps, and they all have more than one account. Using **DISTINCT** in the second query assures that all of the sales reps are accounted for in the first query.

```
SELECT s.id, s.name,COUNT(*) num_accounts
FROM accounts a
JOIN sales_reps s
ON s.id = a.sales_rep_id
GROUPBY s.id, s.nameORDERBY num_accounts;
```

and

```
SELECT DISTINCT id,name
FROM sales_reps;
```

HAVING

HAVING - Expert Tip

HAVING is the “clean” way to filter a query that has been aggregated, but this is also commonly done using a subquery. Essentially, any time you want to perform a **WHERE** on an element of your query that was created by an aggregate, you need to use **HAVING** instead.

```
**SELECT** account_id,       **SUM**(total_amt_usd) **AS** sum_total_amt_usd**FROM** o
rders**GROUP** **BY** 1**HAVING** **SUM**(total_amt_usd) >= 250000;
```

1. How many of the **sales reps** have more than 5 accounts that they manage?

   ```
   SELECT s.id, s.name,COUNT(*) num_accounts
   FROM accounts a
   JOIN sales_reps s
   ON s.id = a.sales_rep_id
   GROUPBY s.id, s.name
   HAVING COUNT(*) > 5
   ORDER BY num_accounts;
   ```

   and technically, we can get this using a **SUBQUERY** as shown below. This same logic can be used for the other queries, but this will not be shown.

   ```
   SELECT COUNT(*) num_reps_above5FROM (        SELECT s.id, s.name,COUNT(*) num_acco
   unts        FROM accounts a        JOIN sales_reps s        ON s.id = a.sales_rep_
   id        GROUP BY s.id, s.name        HAVING COUNT(*) > 5        ORDER BY num_acc
   ounts)AS Table1;
   ```

2. How many **accounts** have more than 20 orders?

   ```
   SELECT a.id, a.name,COUNT(*) num_orders
   FROM accounts a
   JOIN orders o
   ON a.id = o.account_id
   GROUP BY a.id, a.nameHAVINGCOUNT(*) > 20
   ORDER BY num_orders;
   ```

3. Which account has the most orders?

```
SELECT a.id, a.name,COUNT(*) num_orders
FROM accounts a
JOIN orders o
ON a.id = o.account_id
GROUP BY a.id, a.name
ORDER BY num_orders DESC
LIMIT 1;
```

4. How many accounts spent more than 30,000 usd total across all orders?

```
SELECT a.id, a.name,SUM(o.total_amt_usd) total_spent
FROM accounts a
JOIN orders o
ON a.id = o.account_id
GROUP BY a.id, a.name
HAVING SUM(o.total_amt_usd) > 30000
ORDER BY total_spent;
```

5. How many accounts spent less than 1,000 usd total across all orders?

```
SELECT a.id, a.name,SUM(o.total_amt_usd) total_spent
FROM accounts a
JOIN orders o
ON a.id = o.account_id
GROUP BY a.id, a.name
HAVING SUM(o.total_amt_usd) < 1000
ORDER BY total_spent;
```

6. Which account has spent the most with us?

```
SELECT a.id, a.name,SUM(o.total_amt_usd) total_spent
FROM accounts a
JOIN orders o
ON a.id = o.account_id
GROUPBY a.id, a.nameORDERBY total_spentDESCLIMIT 1;
```

7. Which account has spent the least with us?

```
SELECT a.id, a.name,SUM(o.total_amt_usd) total_spent
FROM accounts a
JOIN orders o
ON a.id = o.account_id
```

```
GROUPBY a.id, a.nameORDERBY total_spent
LIMIT 1;
```

8. Which accounts used `facebook` as a **channel** to contact customers more than 6 times?

```
SELECT a.id, a.name, w.channel,COUNT(*) use_of_channel
FROM accounts a
JOIN web_events w
ON a.id = w.account_id
GROUPBY a.id, a.name, w.channel
HAVINGCOUNT(*) > 6AND w.channel = 'facebook'
ORDERBY use_of_channel;
```

9. Which account used `facebook` most as a **channel**?

```
SELECT a.id, a.name, w.channel,COUNT(*) use_of_channel
FROM accounts a
JOIN web_events w
ON a.id = w.account_id
WHERE w.channel = 'facebook'
GROUPBY a.id, a.name, w.channel
ORDERBY use_of_channelDESCLIMIT 1;
```

Note: This query above only works if there are no ties for the account that used facebook the most. It is a best practice to use a larger limit number first such as 3 or 5 to see if there are ties before using LIMIT 1.

10. Which channel was most frequently used by most accounts?All of the top 10 are `direct`.

```
SELECT a.id, a.name, w.channel,COUNT(*) use_of_channel
FROM accounts a
JOIN web_events w
ON a.id = w.account_id
GROUPBY a.id, a.name, w.channel
ORDERBY use_of_channelDESCLIMIT 10;
```

DATE Functions

GROUPing **BY** a date column is not usually very useful in SQL, as these columns tend to have transaction data down to a second. Keeping date information at such granular levels is both a blessing and a curse, as it gives really precise information (a blessing), but it makes grouping information together directly difficult (a curse).

Lucky for us, there are a number of built-in SQL functions that are aimed at helping us improve our experience in working with dates.

Here we saw that dates are stored in the year, month, day, hour, minute, second, (YYYY-MM-DD HH:MM:SS) which helps us in truncating. In the next concept, you will see a number of functions we can use in SQL to take advantage of this functionality.

DATE Functions II

The first function you are introduced to in working with dates is **DATE_TRUNC**.

DATE_TRUNC allows you to truncate your date to a particular part of your date-time column. Common truncations are `day`, `month`, and `year`. Here is a great blog post by Mode Analytics on the power of this function.

DATE_PART can be useful for pulling a specific portion of a date, but notice pulling `month` or day of the week (`dow`) means that you are no longer keeping the years in order. Rather you are grouping for certain components regardless of which year they belonged in.

For additional functions you can use with dates, check out the documentation here, but the **DATE_TRUNC** and **DATE_PART** functions definitely give you a great start!

You can reference the columns in your select statement in **GROUP BY** and **ORDER BY** clauses with numbers that follow the order they appear in the select statement.

For example

```
SELECT standard_qty, COUNT(*)FROM ordersGROUP BY 1 #*(this 1 refers to standard_qty since it is the first of the columns included in the select statement)*ORDER BY 1 #*(this 1 refers to standard_qty since it is the first of the columns included in the select statement)*
```

```
**SELECT** DATE_PART('dow',occurred_at) **AS** day_of_week,       **SUM**(total) **AS** total_qty**FROM** orders**GROUP** **BY** 1**ORDER** **BY** 2
```

1. Find the sales in terms of total dollars for all orders in each `year`, ordered from greatest to least. Do you notice any trends in the yearly sales totals?

```
SELECT DATE_PART('year', occurred_at) ord_year,SUM(total_amt_usd) total_spent
FROM orders
GROUPBY 1
ORDERBY 2DESC;
```

When we look at the yearly totals, you might notice that 2013 and 2017 have much smaller totals than all other years. If we look further at the monthly data, we see that for `2013` and `2017` there is only one month of sales for each of these years (12 for 2013 and 1 for 2017). Therefore, neither of these is evenly represented. Sales have been increasing year over year, with 2016 being the largest sales to date. At this rate, we might expect 2017 to have the largest sales.

1. Which **month** did Parch & Posey have the greatest sales in terms of total dollars? Are all months evenly represented by the dataset?

In order for this to be 'fair', we should remove the sales from 2013 and 2017. For the same reasons as discussed above.

```
SELECT DATE_PART('month', occurred_at) ord_month,SUM(total_amt_usd) total_spent
FROM orders
WHERE occurred_atBETWEEN '2014-01-01'AND '2017-01-01'
GROUPBY 1
ORDERBY 2 DESC;
```

The greatest sales amounts occur in December (12).

1. Which **year** did Parch & Posey have the greatest sales in terms of the total number of orders? Are all years evenly represented by the dataset? Again, 2016 by far has the most amount of orders, but again 2013 and 2017 are not evenly represented to the other years in the dataset.

   ```
   SELECT DATE_PART('year', occurred_at) ord_year,COUNT(*) total_sales
   FROM orders
   GROUPBY 1
   ORDERBY 2DESC;
   ```

2. Which **month** did Parch & Posey have the greatest sales in terms of the total number of orders? Are all months evenly represented by the dataset? December still has the most sales, but interestingly, November has the second most sales (but not the most dollar sales. To make a fair comparison from one month to another 2017 and 2013 data were removed.

3. In which **month** of which **year** did `Walmart` spend the most on gloss paper in terms of dollars?

```
SELECT DATE_TRUNC('month', o.occurred_at) ord_date,SUM(o.gloss_amt_usd) tot_spent
FROM orders o
JOIN accounts a
ON a.id = o.account_id
WHERE a.name = 'Walmart'
GROUPBY 1
ORDERBY 2DESCLIMIT 1;
```

May 2016 was when Walmart spent the most on gloss paper.

CASE Statements

CASE - Expert Tip

- The CASE statement always goes in the SELECT clause.
- CASE must include the following components: WHEN, THEN, and END. ELSE is an optional component to catch cases that didn't meet any of the other previous CASE conditions.
- 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.
- You can include multiple WHEN statements, as well as an ELSE statement again, to deal with any unaddressed conditions.

Example

In a quiz question in the previous Basic SQL lesson, you saw this question:

1. Create a column that divides the `standard_amt_usd` by the `standard_qty` to find the unit price for standard paper for each order. Limit the results to the first 10 orders, and include the `id` and `account_id` fields. **NOTE - you will be thrown an error with the correct solution to this question. This is for a division by zero. You will learn how to get a solution without an error to this query when you learn about CASE statements in a later section.**

Let's see how we can use the **CASE** statement to get around this error.

```
SELECT id, account_id, standard_amt_usd/standard_qtyAS unit_price
FROM orders
LIMIT 10;
```

Now, let’s use a **CASE** statement. This way any time the **standard_qty** is zero, we will return 0, and otherwise, we will return the **unit_price**.

```
SELECT account_id,    CASE WHEN standard_qty = 0 OR standard_qty IS NULL THEN 0
ELSE standard_amt_usd/standard_qty    END AS unit_priceFROM ordersLIMIT 10;
```

Now the first part of the statement will catch any of those divisions by zero values that were causing the error, and the other components will compute the division as necessary. You will notice, we essentially charge all of our accounts 4.99 for standard paper. It makes sense this doesn’t fluctuate, and it is more accurate than adding 1 in the denominator like our quick fix might have been in the earlier lesson.

You can try it yourself using the environment below.

```
**SELECT** account_id,       occurred_at,       total,       **CASE** **WHEN** tot
al > 500 **THEN** 'Over 500'            **WHEN** total > 300 **THEN** '301 - 500'
**WHEN** total > 100 **THEN** '101 - 300'            **ELSE** '100 or under' **END
** **AS** total_group**FROM** orders*********SELECT** account_id,       occurred_
at,       total,       **CASE** **WHEN** total > 500 **THEN** 'Over 500'
**WHEN** total > 300 **AND** total <= 500 **THEN** '301 - 500'            **WHEN**
total > 100 **AND** total <=300 **THEN** '101 - 300'            **ELSE** '100 or u
nder'                        **END** **AS** total_group**FROM** orders;*********S
ELECT** **id**,       account_id,       occurred_at,       channel,       **CASE**
**WHEN** channel = 'facebook' **THEN** 'yes'                        **ELSE** 'no'
**END** **AS** is_facebook**FROM** web_events**ORDER** **BY** occurred_at
```

CASE & Aggregations

This one is pretty tricky. Try running the query yourself to make sure you understand what is happening. The next concept will give you some practice writing **CASE** statements on your own. In this video, we showed that getting the same information using a **WHERE** clause means only being able to get one set of data from the **CASE** at a time.

There are some advantages to separating data into separate columns like this depending on what you want to do, but often this level of separation might be easier to do in another programming language - rather than with SQL.

```
SELECT    CASE WHEN total > 500 THEN 'OVer 500'            ELSE '500 or under'    END
AS total_group,    COUNT(*)AS order_countFROM ordersGROUP BY 1;
```

1. Write a query to display for each order, the account ID, the total amount of the order, and the level of the order - ‘Large’ or ’Small’ - depending on if the order is $3000 or more, or less than $3000.
2. Write a query to display the number of orders in each of three categories, based on the `total` number of items in each order. The three categories are: ‘At Least 2000’, ‘Between 1000 and 2000’ and ‘Less than 1000’.

```
SELECT    CASE WHEN total >= 2000 THEN 'At Least 2000'             WHEN total >= 1
000 AND total < 2000 THEN 'Between 1000 and 2000'             ELSE 'Less than 100
0'END AS order_category,COUNT(*)AS order_countFROM ordersGROUP BY 1;
```

3. We would like to understand 3 different branches of customers based on the amount associated with their purchases. The top branch includes anyone with a Lifetime Value (total sales of all orders) `greater than 200,000` usd. The second branch is between `200,000 and 100,000` usd. The lowest branch is anyone `under 100,000` usd. Provide a table that includes the **level** associated with each **account**. You should provide the **account name**, the **total sales of all orders** for the customer, and the **level**. Order with the top spending customers listed first.

```
SELECT a.name,SUM(total_amt_usd) total_spent,    CASE WHEN SUM(total_amt_usd) > 20
0000 THEN 'top'             WHEN SUM(total_amt_usd) > 100000 THEN 'middle'
ELSE 'low' END AS customer_levelFROM orders oJOIN accounts aON o.account_id = a.id
GROUPBY a.nameORDERBY 2DESC;
```

4. We would now like to perform a similar calculation to the first, but we want to obtain the total amount spent by customers only in `2016` and `2017`. Keep the same **level**s as in the previous question. Order with the top spending customers listed first.

```
SELECT a.name,SUM(total_amt_usd) total_spent,    CASE WHEN SUM(total_amt_usd) > 20
0000 THEN 'top'             WHEN SUM(total_amt_usd) > 100000 THEN 'middle'
ELSE 'low'ENDAS customer_levelFROM orders oJOIN accounts aON o.account_id = a.idWH
ERE occurred_at > '2015-12-31'GROUP BY 1ORDER BY 2 DESC;
```

5. We would like to identify top-performing **sales reps**, which are sales reps associated with more than 200 orders. Create a table with the **sales rep name**, the total number of orders, and a column with `top` or `not` depending on if they have more than 200 orders. Place the top salespeople first in your final table. It

is worth mentioning that this assumes each name is unique - which has been done a few times. We otherwise would want to break by the name and the id of the table.

6. The previous didn't account for the middle, nor the dollar amount associated with the sales. Management decides they want to see these characteristics represented as well. We would like to identify top-performing **sales reps**, which are sales reps associated with more than `200` orders or more than `750000` in total sales. The `middle` group has any **rep** with more than 150 orders or `500000` in sales. Create a table with the **sales rep name**, the total number of orders, total sales across all orders, and a column with `top`, `middle`, or `low` depending on these criteria. Place the top salespeople based on the dollar amount of sales first in your final table. You might see a few upset salespeople by this criteria!

Introduction to Subqueries

Lesson Overview

Up to this point, you have learned a lot about working with data using SQL. You've covered a handful of basic SQL concepts, including aggregate functions and joins. In this lesson, we'll cover subqueries, a fundamental advanced SQL topic.

This lesson will focus on the following components of subqueries, and you will be able to:

- Create subqueries to solve real-world problems
- Differentiate between Subqueries and Joins
- Implement the best type of Subqueries
- Consider the tradeoffs to using subqueries
- Implement the best subquery strategy

Sometimes, the question you are trying to answer can't be solved with the set of tables in your database. Instead, there's a need to manipulate existing tables and join them to solve the problem at hand.

This is where subqueries come to the rescue!

If you can't think of a situation where this need exists, don't worry. We'll review a few real-world applications where existing tables need to be manipulated and joined. And how subqueries help us get there.

What exactly is a subquery?

A subquery is a query *within* a query.

- As a reminder, a query has both **SELECT** and **FROM** clauses to signify what you want to extract from a table and what table you'd like to pull data from. A query that includes subquery, as a result, has multiple **SELECT** and **FROM** clauses.
- The subquery that sits nested inside a larger query is called an **INNER QUERY**. This inner query can be fully executed on its own and often is run independently before when trying to troubleshoot bugs in your code.

```
SELECT product_id,              name,       priceFROM db.productWhere price > (SELECT A
VG(price)                                 FROM db.product);
```

Subqueries in Real-world Applications

When do you need to use a subquery?

You need to use a subquery when you have the need to manipulate an existing table to “pseudo-create” a table that is then used as a part of a larger query. In the examples below, existing tables cannot be joined together to solve the problem at hand. Instead, an existing table needs to be manipulated, massaged, or aggregated in some way to then join to another table in the dataset to answer the posed question.

Set of Problems:

1. Identify the top-selling Amazon products in months where sales have exceeded $1m
    - *Existing Table:* Amazon daily sales
    - *Subquery Aggregation:* Daily to Monthly
2. Examine the average price of a brand's products for the highest-grossing brands
    - *Existing Table:* Product pricing data across all retailers
    - *Subquery Aggregation:* Individual to Average
3. Order the annual salary of employees that are working less than 150 hours a month
    - *Existing Table:* Daily time-table of employees

- *Subquery Aggregation:* Daily to Monthly

Subqueries vs. Joins

Often, SQL users will question the differences between joins and subqueries. After all, they essentially do the same thing: join tables together to create a single output. However, there are times when subqueries are more appropriate than joins and vice versa. In this section of our lesson, we'll cover the differences between the two and when to use each.

Differences between Subqueries and Joins

Use Cases:

Subquery: When an existing table needs to be manipulated or aggregated to then be joined to a larger table.

Joins: A fully flexible and discretionary use case where a user wants to bring two or more tables together and select and filter as needed.

Syntax:

Subquery: A subquery is a query within a query. The syntax, as a result, has multiple **SELECT** and **FROM** clauses.

Joins: A join is simple stitching together multiple tables with a common key or column. A join clause cannot stand and be run independently.

Dependencies:

Subquery: A subquery clause can be run completely independently. When trying to debug code, subqueries are often run independently to pressure test results before running the larger query.

Joins: A join clause cannot stand and be run independently.

Similarities between Subqueries and Joins

Output:

Both subqueries and joins are essentially bringing multiple tables together (whether an existing table is first manipulated or not) to generate a single output.

Deep-dive topics:

What happens under the hood: Query plans are similar for both subqueries and joins. You can read more about how query plans are here. We will not be going in-depth for these in this lesson.

Subquery vs Joins Overview

Components	Subquery	JOINS
Combine data from multiple tables into a single result	YES	YES
Create a flexible view of tables stitched together using a “key”		YES
Build an output to use in a later part of the query	YES	
Subquery Plan: What happens under the hood	YES	YES

Subqueries and Joins Deep-dives

The following comparison will help iterate again the differences between subqueries and joins.

Subqueries:

Output: Either a scalar (a single value) or rows that have met a condition.

Use Case: Calculate a scalar value to use in a later part of the query (e.g., average price as a filter).

Dependencies: Stand independently and be run as complete queries themselves.

Joins:

Output: A joint view of multiple tables stitched together using a common “key”.

Use Case: Fully stitch tables together and have full flexibility on what to “select” and “filter from”.

Dependencies: Cannot stand independently.

Subquery Basics

Now, it’s time to review subquery fundamentals before writing our first subquery. Below are a set of rules to consider when building and executing subqueries.

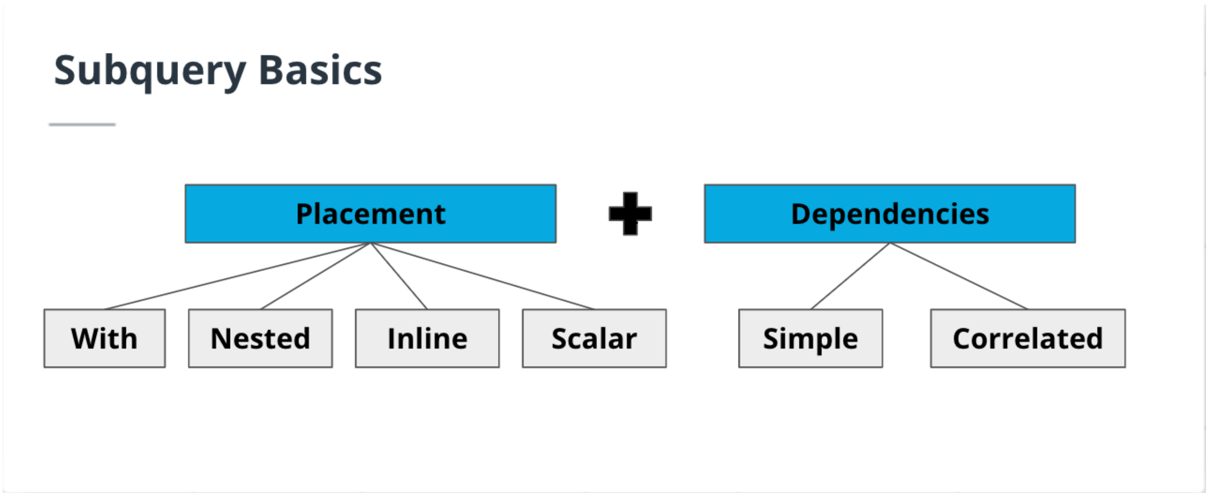
Fundamentals to Know about Subqueries:

- Subqueries must be fully placed inside parentheses.

- Subqueries must be fully independent and can be executed on their own
- Subqueries have two components to consider:
    - Where it's placed
    - Dependencies with the outer/larger query

A caveat with subqueries being independent:

In almost all cases, subqueries are fully independent. They are “interim”/temp tables that can be fully executed on their own. **However, there is an exception.** When a subquery, *typically in the form of a nested or inline subquery*, is correlated to its outer query, it cannot run independently. This is most certainly an edge case since correlated subqueries are rarely implemented compared to standalone, simple subqueries.



https://video.udacity-data.com/topher/2021/April/608b1e4a_screen-shot-2021-04-29-at-1.59.16-pm/screen-shot-2021-04-29-at-1.59.16-pm.png

Subquery Basics

Now, it's time to review subquery fundamentals before writing our first subquery. Below are a set of rules to consider when building and executing subqueries.

Placement:

There are four places where subqueries can be inserted within a larger query:

- With
- Nested
- Inline

- Scalar

Dependencies:

A subquery can be **dependent** on the outer query or **independent** of the outer query.

Resources:

One of my favorite resources on subqueries that covers use cases, syntax, and examples is from Microsoft and can be found here.

Subqueries: Placement

Before writing any code, a strong SQL user considers what problem he or she is trying to solve, where the subquery needs to be placed, and larger tradeoffs (e.g., readability).

The key concept of placement is where exactly the subquery is placed within the context of the larger query. There are four different places where a subquery can be inserted. From my experience, the decision of which placement to leverage stems from (1) the problem at hand and (2) the readability of the query.

Subquery Placement:

With: This subquery is used when you'd like to **"pseudo-create" a table** from an existing table and visually scope the temporary table at the top of the larger query.

Nested: This subquery is used when you'd like the **temporary table to act as a filter** within the larger query, which implies that it often sits within the **WHERE clause.**

Inline: This subquery is used in the same fashion as the WITH use case above. However, instead of the temporary table sitting on top of the larger query, it's embedded within the **FROM clause.**

Scalar: This subquery is used when you'd like to generate **a scalar value to be used as a benchmark** of some sort.

For example, when you'd like to calculate the average salary across an entire organization to compare to individual employee salaries. Because it's often a single value that is generated and used as a benchmark, the scalar subquery often sits within the **select clause.**

Advantages:

Readability: `With` and `Nested` subqueries are most advantageous for readability.

Performance: `Scalar` subqueries are advantageous for performance and are often used on smaller datasets.

Time for more Hands-on Practice

Let's go back to the Parch and Posey Database and put on our Marketing Manager hats back on.

We would like to know which channels send the most traffic per day on average to Parch and Posey. In order to do that, we'll need to aggregate events by channel by day, then we need to take those and average them.

Subqueries

In the video above, we review what it's really like to build and execute a subquery. You'll notice the following order of operations.

1. **Build the Subquery:** The aggregation of an existing table that you'd like to leverage as a part of the larger query.
2. **Run the Subquery:** Because a subquery can stand independently, it's important to run its content first to get a sense of whether this aggregation is the interim output you are expecting.
3. **Encapsulate and Name:** Close this subquery off with parentheses and call it something. In this case, we called the subquery table 'sub.'
4. **Test Again:** Run a `SELECT *` within the larger query to determine if all syntax of the subquery is good to go.
5. **Build Outer Query:** Develop the `SELECT *` clause as you see fit to solve the problem at hand, leveraging the subquery appropriately.

QUIZ

First, we needed to group by the day and channel. Then ordering by the number of events (the third column) gave us a quick way to answer the first question.

```
SELECT DATE_TRUNC('day',occurred_at) AS day,       channel,             COUNT(*) AS ev
ents             FROM web_eventsGROUP BY 1,2ORDER BY 3 DESC;
```

Here you can see that to get the entire table in question 1 back, we included an `*` in our **SELECT** statement. You will need to be sure to alias your table.

```
SELECT *FROM (SELECT DATE_TRUNC('day',occurred_at) AS day,             channel,
COUNT(*) AS events          FROM web_events          GROUP BY 1,2            ORDER BY
 3 DESC) sub;
```

Finally, here we are able to get a table that shows the average number of events a day for each channel.

```
SELECT channel,             AVG(events) AS average_eventsFROM (SELECT DATE_TRUNC('da
y',occurred_at) day,             channel,                         COUNT(*) events
FROM web_events            GROUP BY 1,2) subGROUP BY channelORDER BY 2 DESC;
```

Subquery Formatting

The first concept that helps when thinking about the format of a subquery is the placement of it: with, nested, inline, or scalar.

The second concept to consider is an indentation, which helps heighten readability for your future self or other users that want to leverage your code. The examples in this class are indented quite far—all the way to the parentheses. This isn't practical if you nest many subqueries, but in general, be thinking about how to write your queries in a readable way. Examples of the same query written in multiple different ways are provided below. You will see that some are much easier to read than others.

Badly Formatted Queries

Though these poorly formatted examples will execute the same way as the well-formatted examples, they just aren't very friendly for understanding what is happening!

Here is the first, where it is impossible to decipher what is going on:

```
SELECT * FROM (SELECT DATE_TRUNC('day',occurred_at)ASday, channel,COUNT(*) as eventsFR
OM web_eventsGROUPBY 1,2ORDERBY 3DESC) sub;
```

This second version, which includes some helpful line breaks, is easier to read than the previous version, but it is still not as easy to read as the queries in the **Well**

Formatted Query section.

```
SELECT *
FROM (
SELECT DATE_TRUNC('day',occurred_at)ASday,
channel,COUNT(*)aseventsFROM web_events
GROUPBY 1,2
ORDERBY 3DESC) sub;
```

Well Formatted Query

Now for a well-formatted example, you can see the table we are pulling from much easier than in the previous queries.

```
SELECT *
FROM (SELECT DATE_TRUNC('day',occurred_at)ASday,
                channel,COUNT(*)aseventsFROM web_events
GROUPBY 1,2
ORDERBY 3DESC) sub;
```

Additionally, if we have a **GROUP BY**, **ORDER BY**, **WHERE**, **HAVING**, or any other statement following our subquery, we would then indent it at the same level as our outer query.

The query below is similar to the above, but it is applying additional statements to the outer query, so you can see there are **GROUP BY** and **ORDER BY** statements used on the output and are not tabbed. The inner query **GROUP BY** and **ORDER BY** statements are indented to match the inner table.

```
SELECT *
FROM (SELECT DATE_TRUNC('day',occurred_at)ASday,
                channel,COUNT(*)aseventsFROM web_events
GROUPBY 1,2
ORDERBY 3DESC) sub
GROUPBYday, channel,eventsORDERBY 2DESC;
```

These final two queries are so much easier to read!

Key Details to Highlight

In the first subquery you wrote, you created a table that you could then query again in the **FROM** statement. This was an **inline subquery.**

In the video below, Derek reviews how to build a **nested subquery**. The subquery sits within the filter section of the larger query and leverages a logical operator (e.g., “=”).

Expert Tip

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. **Nested and Scalar subqueries often do not require aliases the way With and Inline subqueries do.**

Nested Subquery

QUIZ

1. Find the month when the first-ever order was placed. Then, use that query to find the average of all 3 paper quantities.
2. Use the last query to find the total amount in USD generated in that particular month.

```
SELECT AVG(standard_qty) avg_std,             AVG(gloss_qty) avg_gls,             AVG
(poster_qty) avg_pstFROM ordersWHERE DATE_TRUNC('month', occurred_at) =     (SELECT DA
TE_TRUNC('month',MIN(occurred_at))FROM orders);SELECT SUM(total_amt_usd)FROM ordersWHE
RE DATE_TRUNC('month', occurred_at) =      (SELECT DATE_TRUNC('month',MIN(occurred_a
t))FROM orders);
```

Subqueries: Dependencies

Simple Subquery

```
WITH dept_average AS
  (SELECT dept,AVG(salary)AS avg_dept_salary
    FROM employee
    GROUP BY employee.dept
  )
SELECT E.eid,E.ename,D.avg_dept_salary
FROM employee E
JOIN dept.average D
     ON E.dept =D.dept
WHERE E.salary>D.avg_dept_salary;
```

Correlated Subquery

```
SELECT employee_id,
nameFROM employees_db emp
WHERE salary >
      (SELECTAVG(salary)
FROM employees_db
WHERE department = emp.department
      );
```

The second concept to consider before writing any code is the dependency of your subquery to the larger query. A subquery can either be simple or correlated. In my experience, it’s better to keep subqueries simple to increase readability for other users that might leverage your code to run or adjust.

Simple Subquery: The inner subquery is completely independent of the larger query.

Correlated Subquery: The inner subquery is dependent on the larger query.

When to use Correlated Query

However, sometimes, it’s slick to include a correlated subquery, specifically when the value of the inner query is dependent on a value outputted from the main query (e.g., the filter statement *constantly changes*). In the example below, you’ll notice that the value of the inner query – average GPA – keeps adjusting depending on the university the student goes to. THAT is a great use case for the correlated subquery.

```
SELECT first_name, last_name, GPA, university
FROM student_db outer_db
WHERE GPA >
          (SELECT AVG(GPA)
                    FROM student_db
                    WHERE university = outer_db.university);
```

Views in SQL

Need for Views

Assume you run a complex query to fetch data from multiple tables. Now, you’d like to query again on the top of the result set. And later, you’d like to query more on the same result set returned earlier. So, there arises a need to store the result set of the original query, so that you can re-query it multiple times. This necessity can be fulfilled with the help of *views*.

Understanding Views

Tables in SQL reside in the database persistently. In contrast, **views** are the *virtual* tables that are derived from one or more base tables. The term *virtual* means that the views do not exist physically in a database, instead, they reside in the memory (not database), just like the result of any query is stored in the memory.

The syntax for creating a view is

```
CREATE VIEW <VIEW_NAME> AS
SELECT …
FROM …
WHERE …
```

The query above is called a view-definition. Once created, ***you can query a view just like you'd query a normal table***, by using its name. The tuples in a view are created as an outcome of a SQL query that selects the filtered data from one or more tables. Let's see a few examples below.

Examples

Example 1 - Consider the same **Parch & Posey** database schema again, where the `sales_reps` table contains details about sales representatives and the `region` table contains the list of regions.

> Suppose you are managing sales representatives who are looking after the accounts in the Northeast region only. The details of such a subset of sales representatives can be fetched from two tables, and stored as a view:

```
create view v1 as
select S.id, S.nameas Rep_Name, R.nameas Region_Name
from sales_reps S
join region R
on S.region_id = R.idand R.name = 'Northeast';
```

The query above will store the result as a view (virtual table) with the name "V1" that can be queried later.

Example 2 - Consider another example from **Parch & Posey** database schema again, where you have practiced the following query in the “Joins” lesson:

> Provide the name for each region for every order, as well as the account name and the unit price they paid (total_amt_usd/total) for the order. Your final result should have 3 columns: region name, account name, and unit price.

The query would be

```
CREATE VIEW V2 AS
SELECT r.name region, a.nameaccount,
       o.total_amt_usd/(o.total + 0.01) unit_price
FROM region r
JOIN sales_reps s
ON s.region_id = r.idJOIN accounts a
ON a.sales_rep_id = s.idJOIN orders o
ON o.account_id = a.id;
```

You can save the result set of the query as a view (virtual table) with the name “V2” that can be queried later.

Note - You can use any SELECT query in the CREATE VIEW query. The above two examples show a join query, whereas the next example shows a subquery used in creating a view.

Example 3 - The subquery you saw earlier, can be also stored as a view.

> Show the report which channels send the most traffic per day on average to Parch and Posey.

```
CREATE VIEW V3 AS
SELECT channel,AVG(events)AS average_events
FROM (SELECT DATE_TRUNC('day',occurred_at)ASday,
              channel,
                           COUNT(*) as events
            FROM web_events
            GROUP BY 1,2) sub
GROUP BY channel;
```

Now, this view can be queried for any information that it contains. For example, you can see the maximum value of `average_events` as:

```
select max(average_events)
from v3;
```

Review

We have already seen how we can use subqueries in the first couple of examples in this lesson. Let's try a tricky problem to see how far subqueries can extend to assist us.

Before we get started, remember:

Subquery: A SQL query where one SQL query is nested within another query

The Question

Now our questions to answer in this section are:

- **What is the top channel used by each account to market products?**
- **How often was that same channel used?**

However, we will need to do **two aggregations and two** subqueries to make this happen.

1. Let's find the number of times each channel is used by each account.
2. So we will need to count the number of rows by Account and Channel. This count will be our **first aggregation** needed.

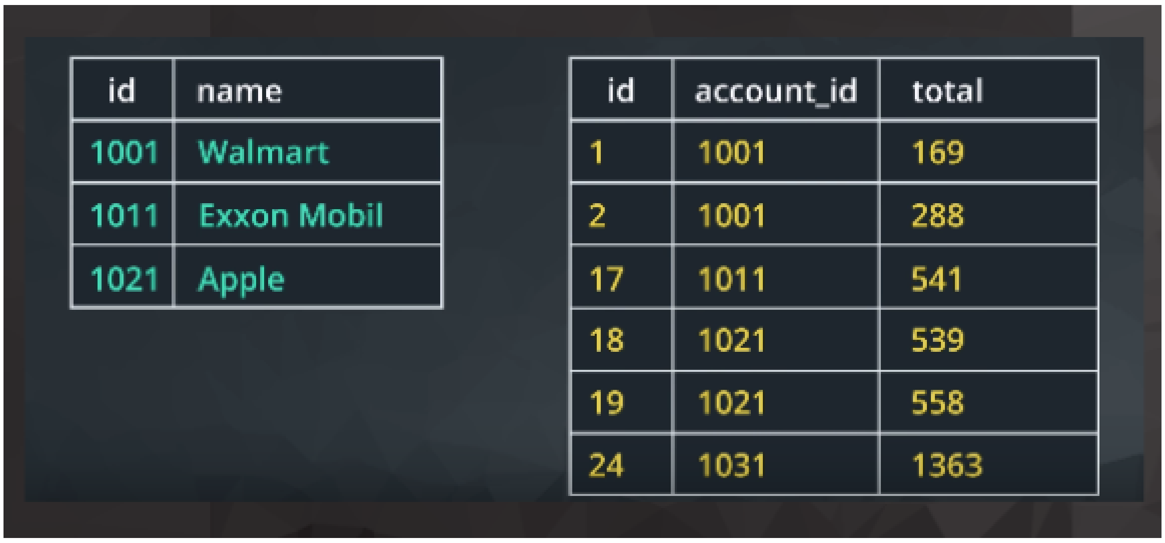
```
SELECT accounts.name, web_events.channel,Count(*)FROM accountsJOIN web_events ON accou
nts.id = Web_events.account_idGROUPBY 1, 2ORDERBY 1,3
```

1. Ok, now we have how often each channel was used by each account. *How do we only return the most used account (or accounts if multiple are tied for the most)?*

We need to see which usage of the channel in our first query is equal to the maximum usage channel for that account. So, a keyword should jump out to you - **maximum**. This will be our **second aggregation** and it utilizes the data from the first table we returned so this will be our **subquery**. Let's take the maximum count from each account to create a table with the maximum usage channel amount per account.

```
SELECT T1.name,Max(T1.count)
FROM (
SELECT accounts.nameasname,
                web_events.channelas channel,
                Count(*) as count
FROM accounts
JOIN web_events ON accounts.id = Web_events.account_id
GROUP BY 1, 2
ORDER BY 1,3
)as T1
GROUPBY 1
```

1. So now we have the MAX usage number for a channel for each account. Now we can use this to filter the original table to find channels for each account that match the MAX amount for their account.
2. We do this by putting this in the WHERE clause



id	name
1001	Walmart
1011	Exxon Mobil
1021	Apple

id	account_id	total
1	1001	169
2	1001	288
17	1011	541
18	1021	539
19	1021	558
24	1031	1363

https://video.udacity-data.com/topher/2018/March/5a9f0984_screen-shot-2018-03-06-at-1.34.13-pm/screen-shot-2018-03-06-at-1.34.13-pm.png

Example of what the two tables look might look like

As an intermediate table, I imagine a table that looks like the one below.

Intermediate Table Results

Account	Event	Count
Walmart	Direct	22
Walmart	Adwords	5

Walmart	Facebook	9
Walmart	Organic	22
Exxon Mobil	Direct	1
Exxon Mobil	Adwords	5
Exxon Mobil	Facebook	2
Exxon Mobil	Organic	4
Apple	Direct	7

Then obtaining a final table that looks like the following:

Final Table Results

```
SELECT t3.id, t3.name, t3.channel, t3.ctFROM (SELECT a.id, a.name, we.channel,COUNT(*)
ct            FROM accounts a            JOIN web_events we            ON a.id = we.ac
count_id            GROUP BY a.id, a.name, we.channel) T3JOIN (SELECT t1.id, t1.name,M
AX(ct) max_chan            FROM (SELECT a.id, a.name, we.channel,COUNT(*) ct
FROM accounts a                        JOIN web_events we                        ON a.
id = we.account_id                        GROUP BY a.id, a.name, we.channel) t1
GROUP BY t1.id, t1.name) t2ON t2.id = t3.id AND t2.max_chan = t3.ctORDER BY t3.id;
```

Account	Event	Count (MAX)
Walmart	Direct	22
Exxon Mobil	Adwords	5
Apple	Facebook	9
Tesla	Organic	15

```
SELECT t3.id, t3.name, t3.channel, t3.ct
FROM (SELECT a.id, a.name, we.channel,COUNT(*) ct
      FROM accounts a
      JOIN web_events we
      On a.id = we.account_id
      GROUP BY a.id, a.name, we.channel) T3
JOIN (SELECT t1.id, t1.name,MAX(ct) max_chan
      FROM (SELECT a.id, a.name, we.channel,COUNT(*) ct
            FROM accounts a
            JOIN web_events we
            ON a.id = we.account_id
            GROUP BY a.id, a.name, we.channel) t1
      GROUP BY t1.id, t1.name) t2
ON t2.id = t3.idAND t2.max_chan = t3.ct
ORDER BY t3.id;
```

QUIZ

1 Provide the **name** of the **sales_rep** in each **region** with the largest amount of **total_amt_usd** sales.

First, I wanted to find the **total_amt_usd** totals associated with each **sales rep**, and I also wanted the region in which they were located. The query below provided this information.

```
SELECT s.name rep_name, r.name region_name,SUM(o.total_amt_usd) total_amt
FROM sales_reps s
JOIN accounts a
ON a.sales_rep_id = s.idJOIN orders o
ON o.account_id = a.idJOIN region r
ON r.id = s.region_id
GROUPBY 1,2
ORDERBY 3DESC;
```

Next, I pulled the max for each region, and then we can use this to pull those rows in our final result.

```
SELECT region_name,MAX(total_amt) total_amt
FROM(SELECT s.name rep_name, r.name region_name,SUM(o.total_amt_usd) total_amt
FROM sales_reps s
JOIN accounts a
ON a.sales_rep_id = s.idJOIN orders o
ON o.account_id = a.idJOIN region r
ON r.id = s.region_id
GROUPBY 1, 2) t1
GROUPBY 1;
```

Essentially, this is a **JOIN** of these two tables, where the region and amount match.

```
SELECT t3.rep_name, t3.region_name, t3.total_amt
FROM(SELECT region_name,MAX(total_amt) total_amt
FROM(SELECT s.name rep_name, r.name region_name,SUM(o.total_amt_usd) total_amt
FROM sales_reps s
JOIN accounts a
ON a.sales_rep_id = s.idJOIN orders o
ON o.account_id = a.idJOIN region r
ON r.id = s.region_id
GROUPBY 1, 2) t1
GROUPBY 1) t2
JOIN (SELECT s.name rep_name, r.name region_name,SUM(o.total_amt_usd) total_amt
FROM sales_reps s
JOIN accounts a
ON a.sales_rep_id = s.idJOIN orders o
ON o.account_id = a.idJOIN region r
ON r.id = s.region_id
GROUPBY 1,2
ORDERBY 3DESC) t3
ON t3.region_name = t2.region_nameAND t3.total_amt = t2.total_amt;
```

2 For the region with the largest sales **total_amt_usd**, how many **total** orders were placed?

The first query I wrote was to pull the **total_amt_usd** for each **region**.

```
SELECT r.name region_name,SUM(o.total_amt_usd) total_amt
FROM sales_reps s
JOIN accounts a
ON a.sales_rep_id = s.idJOIN orders o
ON o.account_id = a.idJOIN region r
ON r.id = s.region_id
GROUPBY r.name;
```

Then we just want the region with the max amount from this table. There are two ways I considered getting this amount. One was to pull the max using a subquery. Another way is to order descending and just pull the top value.

```
SELECTMAX(total_amt)
FROM (SELECT r.name region_name,SUM(o.total_amt_usd) total_amt
FROM sales_reps s
JOIN accounts a
ON a.sales_rep_id = s.idJOIN orders o
ON o.account_id = a.idJOIN region r
ON r.id = s.region_id
GROUPBY r.name) sub;
```

Finally, we want to pull the total orders for the region with this amount:

```
SELECT r.name,COUNT(o.total) total_orders
FROM sales_reps s
JOIN accounts a
ON a.sales_rep_id = s.idJOIN orders o
ON o.account_id = a.idJOIN region r
ON r.id = s.region_id
GROUPBY r.nameHAVINGSUM(o.total_amt_usd) = (
SELECTMAX(total_amt)
FROM (SELECT r.name region_name,SUM(o.total_amt_usd) total_amt
FROM sales_reps s
JOIN accounts a
ON a.sales_rep_id = s.idJOIN orders o
ON o.account_id = a.idJOIN region r
ON r.id = s.region_id
GROUPBY r.name) sub);
```

This provides the **Northeast** with **2357** orders.

3 How many accounts had more **total** purchases than the account **name** which has bought the most **standard_qty** paper throughout their lifetime as a customer?

First, we want to find the account that had the most **standard_qty** paper. The query here pulls that account, as well as the total amount:

```
SELECT a.name account_name,SUM(o.standard_qty) total_std,SUM(o.total) total
FROM accounts a
JOIN orders o
ON o.account_id = a.idGROUPBY 1
ORDERBY 2DESCLIMIT 1;
```

Now, I want to use this to pull all the accounts with more total sales:

```
SELECT a.nameFROM orders o
JOIN accounts a
ON a.id = o.account_id
GROUPBY 1
HAVINGSUM(o.total) > (SELECT total
FROM (SELECT a.name act_name,SUM(o.standard_qty) tot_std,SUM(o.total) total
FROM accounts a
JOIN orders o
ON o.account_id = a.idGROUPBY 1
ORDERBY 2DESCLIMIT 1) sub);
```

This is now a list of all the accounts with more total orders. We can get the count with just another simple subquery.

```
SELECTCOUNT(*)
FROM (SELECT a.nameFROM orders o
JOIN accounts a
ON a.id = o.account_id
GROUPBY 1
HAVINGSUM(o.total) > (SELECT total
FROM (SELECT a.name act_name,SUM(o.standard_qty) tot_std,SUM(o.total) total
FROM accounts a
JOIN orders o
ON o.account_id = a.idGROUPBY 1
ORDERBY 2DESCLIMIT 1) inner_tab)
             ) counter_tab;
```

4 For the customer that spent the most (in total over their lifetime as a customer) **total_amt_usd**, how many **web_events** did they have for each channel?

Here, we first want to pull the customer with the most spent in lifetime value.

```
SELECT a.id, a.name,SUM(o.total_amt_usd) tot_spent
FROM orders o
JOIN accounts a
ON a.id = o.account_id
GROUPBY a.id, a.nameORDERBY 3DESCLIMIT 1;
```

Now, we want to look at the number of events on each channel this company had, which we can match with just the **id**.

```
SELECT a.name, w.channel,COUNT(*)
FROM accounts a
JOIN web_events w
ON a.id = w.account_idAND a.id =  (SELECTidFROM (SELECT a.id, a.name,SUM(o.total_amt_u
sd) tot_spent
FROM orders o
JOIN accounts a
ON a.id = o.account_id
GROUPBY a.id, a.nameORDERBY 3DESCLIMIT 1) inner_table)
GROUPBY 1, 2
ORDERBY 3DESC;
```

I added an **ORDER BY** for no real reason, and the account name to assure I was only pulling from one account.

5 What is the lifetime average amount spent in terms of **total_amt_usd** for the top 10 total spending **accounts**?

First, we just want to find the top 10 accounts in terms of highest **total_amt_usd**.

```
SELECT a.id, a.name,SUM(o.total_amt_usd) tot_spent
FROM orders o
JOIN accounts a
ON a.id = o.account_id
GROUPBY a.id, a.nameORDERBY 3DESCLIMIT 10;
```

Now, we just want the average of these 10 amounts.

```
SELECTAVG(tot_spent)
FROM (SELECT a.id, a.name,SUM(o.total_amt_usd) tot_spent
FROM orders o
JOIN accounts a
ON a.id = o.account_id
GROUPBY a.id, a.nameORDERBY 3DESCLIMIT 10) temp;
```

6 What is the lifetime average amount spent in terms of **total_amt_usd**, including only the companies that spent more per order, on average, than the average of all orders.

First, we want to pull the average of all accounts in terms of **total_amt_usd**:

```
SELECTAVG(o.total_amt_usd) avg_all
FROM orders o
```

Then, we want to only pull the accounts with more than this average amount.

```
SELECT o.account_id,AVG(o.total_amt_usd)
FROM orders o
GROUPBY 1
HAVINGAVG(o.total_amt_usd) > (SELECTAVG(o.total_amt_usd) avg_all
FROM orders o);
```

Finally, we just want the average of these values.

```
SELECTAVG(avg_amt)
FROM (SELECT o.account_id,AVG(o.total_amt_usd) avg_amt
FROM orders o
GROUPBY 1
HAVINGAVG(o.total_amt_usd) > (SELECTAVG(o.total_amt_usd) avg_all
FROM orders o)) temp_table;
```

Wow! That was intense. Nice job if you got these!

Subquery Tradeoffs

The same problem can be solved using multiple subquery techniques. A strong SQL user considers a set of tradeoffs before deciding which type of subquery to build and execute.

A few tradeoffs are listed below:

Readability: How easy it is to determine what the code is doing.

Performance: How quickly the code runs.

Query Plan: What happens under the hood.

We could spend an entire course walking through SQL performance optimizations. I've found this guide on optimization techniques helpful as you think through ways to increase performance.

Subquery Strategy

Before diving headfirst into building a subquery, consider the workflow below. Strong SQL users walk through the following **before ever writing a line of code**:

1. Determine if a subquery is needed (or a join/aggregation function will suffice).
2. If a subquery is needed, determine where you'll need to place it.
3. Run the subquery as an independent query first: is the output what you expect?
4. Call it something! If you are working with With or Inline subquery, you'll most certainly need to name it.
5. Run the entire query -- both the inner query and outer query.

Placement: WITH

Use Case for `With` subquery:

- When a user wants to **create a version** of an existing table **to be used in a larger query** (e.g., aggregate daily prices to an average price table).
- It is advantageous for readability purposes.

```
WITH average_price as
    (SELECT brand_id,AVG(product_price)as brand_avg_price
    FROM product_records),
SELECT a.brand_id, a.total_brand_sales, b.brand_avg_price
FROM brand_table a
JOIN average_price b
ON b.brand_id = a.brand_id
ORDERBY a.total_brand_salesdesc;
```

CTE stands for Common Table Expression. A Common Table Expression in SQL allows you to define a temporary result, such as a table, to then be referenced in a later part of the query.

Your First WITH (CTE)

Let's leverage the same question you saw 'In Your First Subquery' but this time try building a solution that uses the With Subquery.

QUESTION: You need to find the average number of events for each channel per day.

SOLUTION:

```
SELECT channel,AVG(events)AS average_events
FROM (SELECT DATE_TRUNC('day',occurred_at)ASday,
             channel,COUNT(*)aseventsFROM web_events
GROUPBY 1,2) sub
GROUPBY channel
ORDERBY 2DESC;
```

Let's try this again using a **WITH** statement.

Notice, you can pull the inner query:

```
SELECT DATE_TRUNC('day',occurred_at)ASday,
       channel,COUNT(*)aseventsFROM web_events
GROUPBY 1,2
```

This is the part we put in the **WITH** statement. Notice, we are aliasing the table as `events` below:

```
WITH events AS (
SELECT DATE_TRUNC('day',occurred_at)ASday,
                        channel,COUNT(*)aseventsFROM web_events
GROUPBY 1,2)
```

Now, we can use this newly created `events` table as if it is any other table in our database:

```
WITH events AS (
SELECT DATE_TRUNC('day',occurred_at)ASday,
                        channel,COUNT(*)aseventsFROM web_events
GROUPBY 1,2)

SELECT channel,AVG(events)AS average_events
FROMeventsGROUPBY channel
ORDERBY 2DESC;
```

For the above example, we don't need anymore than the one additional table, but imagine we needed to create a second table to pull from. We can create an additional table to pull from in the following way:

```
WITH table1 AS (
SELECT *
FROM web_events),

     table2AS (
SELECT *
FROM accounts)


SELECT *
FROM table1
JOIN table2
ON table1.account_id = table2.id;
```

You can add more and more tables using the **WITH** statement in the same way.

Below, you will see each of the previous solutions restructured using the **WITH** clause. This is often an easier way to read a query.

1. Provide the **name** of the **sales_rep** in each **region** with the largest amount of **total_amt_usd** sales.

```
WITH t1 AS (
  SELECT s.name rep_name, r.name region_name,
      SUM(o.total_amt_usd) total_amt
  FROM sales_reps s
  JOIN accounts a
  ON a.sales_rep_id = s.id
  JOIN orders o
  ON o.account_id = a.id
  JOIN region r
  ON r.id = s.region_id
  GROUP BY 1,2
  ORDER BY 3 DESC),
t2 AS (
  SELECT region_name,MAX(total_amt) total_amt
  FROM t1
  GROUP BY 1)
SELECT t1.rep_name, t1.region_name, t1.total_amt
FROM t1
JOIN t2 ON t1.region_name = t2.region_name
      AND t1.total_amt = t2.total_amt;
```

2. For the region with the largest sales **total_amt_usd**, how many **total** orders were placed?

```
WITH t1 AS (
SELECT r.name region_name,SUM(o.total_amt_usd) total_amt
FROM sales_reps s
JOIN accounts a
ON a.sales_rep_id = s.idJOIN orders o
ON o.account_id = a.idJOIN region r
ON r.id = s.region_id
GROUPBY r.name),
t2AS (
SELECTMAX(total_amt)
FROM t1)
SELECT r.name,COUNT(o.total) total_orders
FROM sales_reps s
JOIN accounts a
ON a.sales_rep_id = s.idJOIN orders o
ON o.account_id = a.idJOIN region r
ON r.id = s.region_id
GROUPBY r.nameHAVINGSUM(o.total_amt_usd) = (SELECT *FROM t2);
```

3. For the account that purchased the most (in total over their lifetime as a customer) **standard_qty** paper, **how many accounts** still had more in **total** purchases?

```
WITH t1 AS (
SELECT a.name account_name,SUM(o.standard_qty) total_std,SUM(o.total) total
FROM accounts a
JOIN orders o
ON o.account_id = a.idGROUPBY 1
ORDERBY 2DESCLIMIT 1),
t2AS (
SELECT a.nameFROM orders o
JOIN accounts a
ON a.id = o.account_id
GROUPBY 1
HAVINGSUM(o.total) > (SELECT totalFROM t1))
SELECTCOUNT(*)
FROM t2;
```

4. For the customer that spent the most (in total over their lifetime as a customer) **total_amt_usd**, how many **web_events** did they have for each channel?

```
WITH t1 AS (
SELECT a.id, a.name,SUM(o.total_amt_usd) tot_spent
FROM orders o
JOIN accounts a
ON a.id = o.account_id
GROUPBY a.id, a.nameORDERBY 3DESCLIMIT 1)
SELECT a.name, w.channel,COUNT(*)
FROM accounts a
JOIN web_events w
ON a.id = w.account_idAND a.id =  (SELECTidFROM t1)
```

```
GROUPBY 1, 2
ORDERBY 3DESC;
```

5. What is the lifetime average amount spent in terms of **total_amt_usd** for the top 10 total spending **accounts**?

```
WITH t1 AS (
SELECT a.id, a.name,SUM(o.total_amt_usd) tot_spent
FROM orders o
JOIN accounts a
ON a.id = o.account_id
GROUPBY a.id, a.nameORDERBY 3DESCLIMIT 10)
SELECTAVG(tot_spent)
FROM t1;
```

6. What is the lifetime average amount spent in terms of **total_amt_usd**, including only the companies that spent more per order, on average, than the average of all orders.

```
WITH t1 AS (
SELECTAVG(o.total_amt_usd) avg_all
FROM orders o
JOIN accounts a
ON a.id = o.account_id),
t2AS (
SELECT o.account_id,AVG(o.total_amt_usd) avg_amt
FROM orders o
GROUPBY 1
HAVINGAVG(o.total_amt_usd) > (SELECT *FROM t1))
SELECTAVG(avg_amt)
FROM t2;
```

Wow! That was intense. Nice job if you got these!

Placement: Nested

Use Case for a Nested Subquery

- When a user wants to **filter an output using a condition met from another table.**
- This type of placement also has advantages for making the code easy to read.

Nested Query

```
SELECT *
FROM students
WHERE student_id
IN (SELECT DISTINCT student_id
    FROM gpa_table
    WHERE gpa>3.5);
```

Placement: Inline

Use Case for Inline Subquery

- It is a very similar use case to ‘With’ subqueries.
    - Inline subqueries **create a “pseudo table”** that aggregates or **manipulates an existing table** to be **used in a larger query**.
- The disadvantage of the inline subquery is that it is not easy to read.

Inline Query

```
SELECT dept_name,
       max_gpa
FROM department_db x
     (SELECT dept_id
MAX(gpa)as max_gpa
FROM students
GROUPBY dept_id
      )y
WHERE x.dept_id = y.dept_id
ORDERBY dept_name;
```

Quiz Time

Let's go over what you have learned in the last few pages. Apply that learning to this use case described in the quiz below.

QUIZ QUESTION

You have 2 tables. The 1st table has house prices and zip codes. The 2nd table has user demographic information (e.g., county information, zipcodes). You need to write a query that identifies the counties with the houses above the median price per county. In order to answer this question, what type of subquery placement would be ideal and easy to read?

- With placement
- Inline placement
- Nested placement

▼ **SOLUTION**

WITH and NESTED.

Placement: Inline

Use Case for Scalar subquery placement

- It **selects only one column or expression and returns one row**, used in the select clause of the main query
- It has the advantage of performance or if the data set is small

Details:

- If a scalar subquery does not find a match, it returns a NULL.
- If a scalar subquery finds multiple matches, it returns an ERROR.

Scalar Query

```
SELECT
   (SELECT MAX(salary) FROM employees_db)AS top_salary,
   employee_name
FROM employees_db;
```

Subquery Conclusion

Recap

Subquery Facts to Know:

- Commonly used as a filter/aggregation tool
- Commonly used to create a “temporary” view that can be queried off
- Commonly used to increase readability

- Can stand independently

This lesson was the first of the more advanced topics in writing SQL. Being able to break a problem down into the necessary tables and finding a solution using the resulting table is very useful in practice.

If you didn't get the solutions to these queries on the first pass, don't be afraid to come back another time and give them another try. Additionally, you might try coming up with some questions of your own to see if you can find the solution.

Introduction to SQL Data Cleaning

Lesson Overview
In this lesson, you will learn a number of fundamental data cleaning SQL techniques that help prep raw data to make it usable for analyses.
At the end of this lesson, you will be able to:
• Clean and re-structure messy data.
• Convert columns to different data types.
• Tricks for manipulating **NULL**s.

Data Cleaning Real-world Applications

By definition, data cleaning is the task of cleaning up raw data to make it usable and ready for analysis. Almost always, your data will not be ready for you to run an analysis.

- Your data could all be lumped together in a single column, and you need to parse it to extract useful information.
- Your data could all default to string data types, and you need to cast each column appropriately to run computations.
- Your data could have un-standardized units of currency, and you need to normalize the column to ensure you are comparing equally across records.

In my experience, there hasn't been a day that goes by without some sort of data cleaning. In fact, data scientists often joke that 80% of their time is spent preparing the data and only 20% is spent building models.

Several articles and blog posts boast this claim, including this one by Forbes.

Relevant Definitions:

Normalization: Standardizing or "cleaning up a column" by transforming it in some way to make it ready for analysis. A few normalization techniques are below:

- Adjusting a column that includes multiple currencies to one common currency
- Adjusting the varied distribution of a column value by transforming it into a z-score
- Converting all price into a common metric (e.g., price per ounce)

Data Cleaning Strategy

It's inadvisable to dive into executing a problem immediately. In fact, I recommend taking a step back to think through a strategy before ever writing a line of code. This approach goes for data cleaning, as well.

The key steps to consider when going about your data cleaning task include the following:

1. **What data do you need?:** Review what data you need to run an analysis and solve the problem at hand.
2. **What data do you have?:** Take stock of not only the information you have in your dataset today but what data types those fields are. Do these align with your data needs?
3. **How will you clean your data?:** Build a game plan of how you'll convert the data you currently have to the data you need. What types of actions and data cleaning techniques will you have to apply? Do you have the skills you need to go from the current to future state?
4. **How will you analyze your data?:** Now, it's game time! How do you run an effective analysis? Build an approach for analysis, as well. And visualize your plan to solve the problem. Finally, remember to question "so what?" at the end of your results, which will help drive recommendations for your organization.

Methods to Cover

The following set of methods cover three types of data cleaning techniques: parsing information, returning where information lives, and changing the data type of the information.

- **Left:** Extracts a number of characters from a string starting from the left
- **Right:** Extracts a number of characters from a string starting from the right
- **Substr:** Extracts a substring from a string (starting at any position)
- **Position:** Returns the position of the first occurrence of a substring in a string
- **Strpos:** Returns the position of a substring within a string
- **Concat:** Adds two or more expressions together
- **Cast:** Converts a value of any type into a specific, different data type
- **Coalesce:** Returns the first non-null value in a list

A handful of these functions, as you'll quickly realize, are more commonly used than others.

Commonly-Used Data Functions

The following set of commonly-used data functions are handy when extracting and converting information in a raw dataset.

- **Left/Right/Substr:** Used to extract information typically when all information resides in a single column
- **Concat:** Used commonly when two or more pieces of information can be used as a unique identifier
- **Cast:** Used when data types default to a specific type (e.g., most commonly STRING) and need to be assigned to the appropriate data type to run computations

LEFT, RIGHT, SUBSTR

Syntax

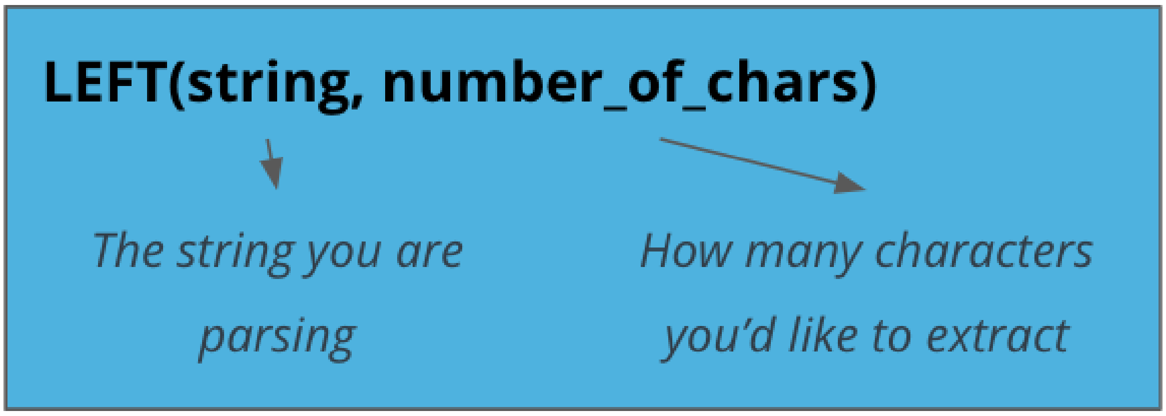
- **Left:** Extracts a # of characters from a string starting from the left
- **Right:** Extracts a # of characters from a string starting from the right

```
LEFT(student_information, 8) AS student_id
RIGHT(student_information, 6) AS salary
```

Use Case

Typically when a single column holds too much info from a raw data dump and needs to be parsed to make the data usable

---


LEFT(string, number_of_chars)
The string you are parsing
How many characters you'd like to extract

Syntax for LEFT, RIGHT

Syntax

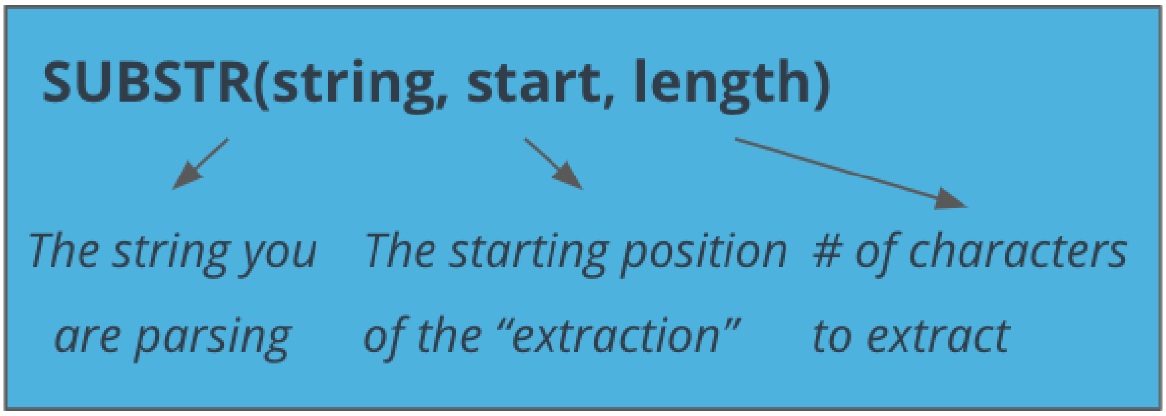
- **Substr:** Extracts a substring from a string (starting at any position)

```
SUBSTR(string,start,length)
```

```
SUBSTR(student_information, 11, 1) AS gender
```

Use Case

Typically when a single column holds too much info, needs to be parsed to make the data usable, and the info lies in the middle of the text


SUBSTR(string, start, length)
The string you are parsing
The starting position of the "extraction"
# of characters to extract

Syntax for SUBSTR

1. In the **accounts** table, there is a column holding the **website** for each company. The last three digits specify what type of web address they are using. A list of extensions (and pricing) is provided here. Pull these extensions and provide how many of each website type exist in the **accounts** table.
2. There is much debate about how much the name (or even the first letter of a company name) matters. Use the **accounts** table to pull the first letter of each company name to see the distribution of company names that begin with each letter (or number).
3. Use the **accounts** table and a **CASE** statement to create two groups: one group of company names that start with a number and the second group of those company names that start with a letter. What proportion of company names start with a letter?
4. Consider vowels as `a`, `e`, `i`, `o`, and `u`. What proportion of company names start with a vowel, and what percent start with anything else?

CONCAT

Syntax

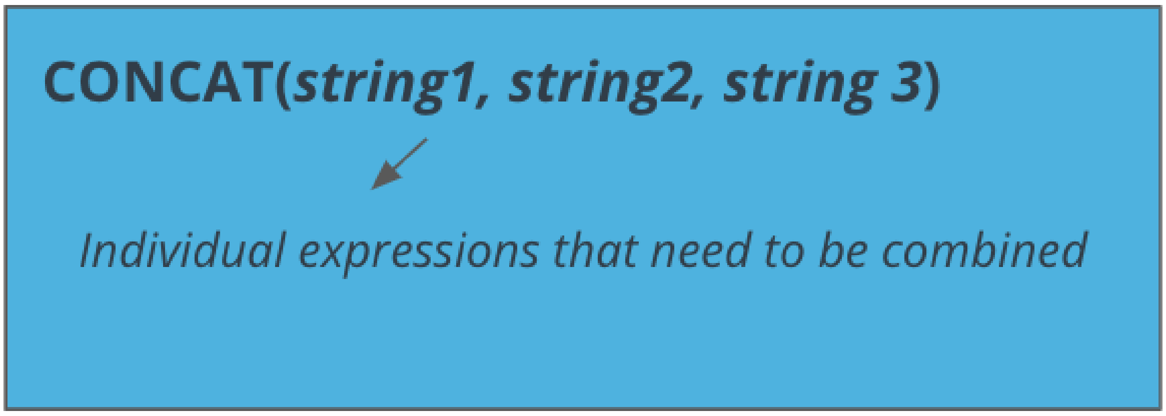
- **CONCAT:** Adds two or more expressions together

```
CONCAT(string1, string2, string3)
```

```
CONCAT(month, '-', day, '-', year) AS date
```

Use Case

When a unique identifier is split across multiple columns and the user has a need to combine them.


CONCAT(string1, string2, string 3)
Individual expressions that need to be combined

Syntax for CONCAT

Quiz: CONCAT, LEFT, RIGHT, and SUBSTR

1. Suppose the company wants to assess the performance of all the sales representatives. Each sales representative is assigned to work in a particular region. To make it easier to understand for the HR team, display the concatenated sales_reps.id, ‘_’ (underscore), and region.name as EMP_ID_REGION for each sales representative.
2. From the accounts table, display the name of the client, the coordinate as concatenated (latitude, longitude), email id of the primary point of contact as <first letter of the primary_poc><last letter of the primary_poc>@<extracted name and domain from the website>.
3. From the web_events table, display the concatenated value of account_id, '_' , channel, '_', count of web events of the particular channel.

Solution

```
#1.

SELECT CONCAT(SALES_REPS.ID, '_', REGION.NAME) EMP_ID_REGION, SALES_REPS.NAMEFROM SALE
S_REPS
JOIN REGION
ON SALES_REPS.REGION_ID = REGION_ID;

#2.

SELECT NAME, CONCAT(LAT, ', ', LONG) COORDINATE, CONCAT(LEFT(PRIMARY_POC, 1), RIGHT(PR
IMARY_POC, 1), '@', SUBSTR(WEBSITE, 5)) EMAIL
```

```
FROM ACCOUNTS;

#3.

WITH T1 AS (
 SELECT ACCOUNT_ID, CHANNEL, COUNT(*)
 FROM WEB_EVENTS
 GROUP BY ACCOUNT_ID, CHANNEL
 ORDER BY ACCOUNT_ID
)
SELECT CONCAT(T1.ACCOUNT_ID, '_', T1.CHANNEL, '_', COUNT)
FROM T1;
```

CAST

Syntax

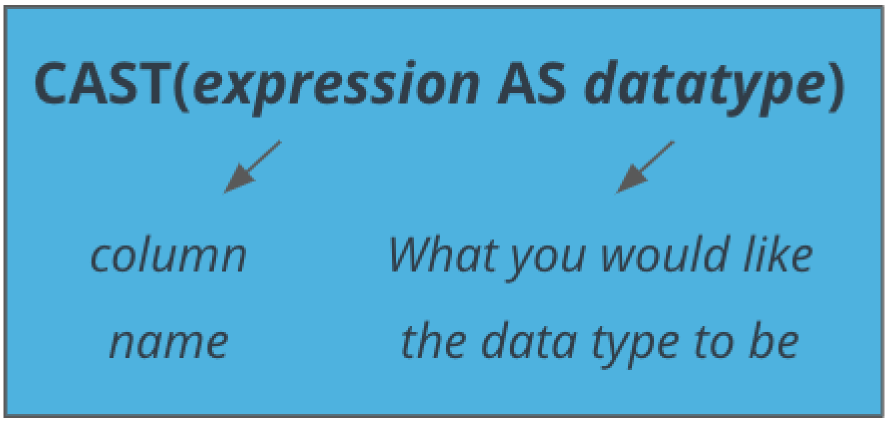
- **CAST:** Converts a value of any type into a specific, different data type

```
CAST(expression AS datatype)
```

```
CAST(salary AS int)
```

Use Case

When the raw data types are unsuitable for analyses. The most common occurrence is when the raw data types all default to strings, and the user has to cast each column to the appropriate data type.


CAST(expression AS datatype)
column name
What you would like the data type to be

Syntax for CAST

Advanced Cleaning Functions

The following advanced cleaning functions are used less often but are good to understand to complete your holistic understanding of data cleaning in SQL. For the most part, these functions are used to return the position of information and help you work with NULLs.

- **Position/Strpos:** Used to return the position of information to identify where relevant information is held in a string to then extract across all records
- **Coalesce:** Used to return the first non-null value that's commonly used for normalizing data that's stretched across multiple columns and includes NULLs.

POSITION, STRPOS

Syntax

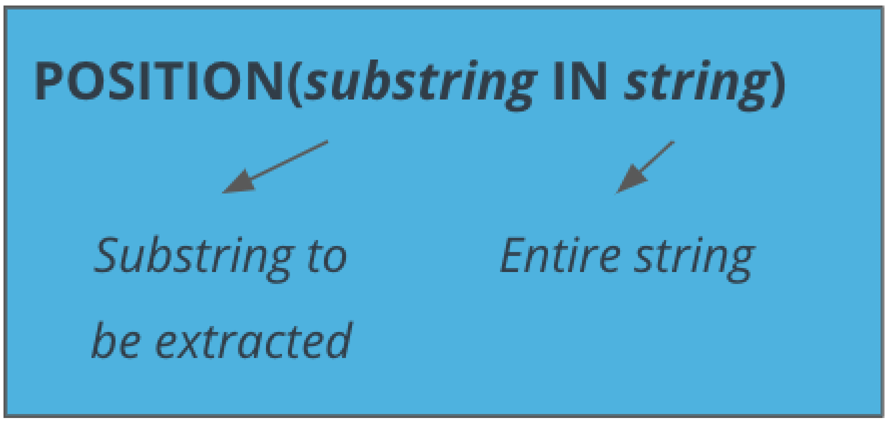
- **POSITION:** Returns the position of the first occurrence of a substring in a string

```
POSITION(substring IN string)
```

```
POSITION("$" IN student_information) as
salary_starting_position
```

Use Case

When there is a single column that holds so much information, and the user needs to identify where a piece of information is. The location of the information is typically then used to consistently extract this information across all records.



Syntax for POSITION

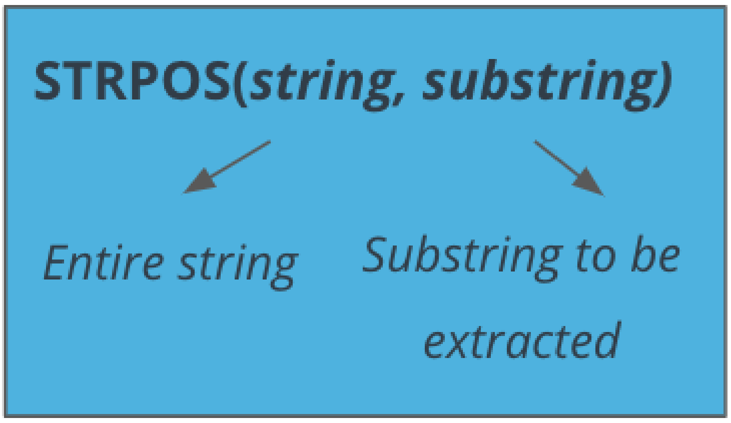
Syntax

- **STRPOS:** Converts a value of any type into a specific, different data type

```
STRPOS(string, substring)
```

Use Case

When the raw data types are unsuitable for analyses. The most common occurrence is when the raw data types all default to strings, and the user has to cast each column to the appropriate data type.



Syntax for STRPOS

Quizzes POSITION & STRPOS

You will need to use what you have learned about **LEFT** & **RIGHT**, as well as what you know about **POSITION** or **STRPOS** to do the following quizzes.

1. Use the `accounts` table to create **first** and **last** name columns that hold the first and last names for the `primary_poc`.
2. Now see if you can do the same thing for every rep `name` in the `sales_reps` table. Again provide **first** and **last** name columns.

```
SELECT LEFT(primary_poc, STRPOS(primary_poc, ' ') -1 ) first_name,
RIGHT(primary_poc, LENGTH(primary_poc) - STRPOS(primary_poc, ' ')) last_name
FROM accounts;
```

```
SELECT LEFT(name, STRPOS(name, ' ') -1 ) first_name, RIGHT(name, LENGTH(name) - STRPOS
(name, ' ')) last_name
FROM sales_reps;
```

Quizzes CONCAT

1. Each company in the `accounts` table wants to create an email address for each `primary_poc`. The email address should be the first name of the **primary_poc** `.` last name **primary_poc** `@` company name `.com`.
2. You may have noticed that in the previous solution some of the company names include spaces, which will certainly not work in an email address. See if you can create an email address that will work by removing all of the spaces in the account `name`, but otherwise, your solution should be just as in question `1`. Some helpful documentation is here.
3. We would also like to create an initial password, which they will change after their first log in. The first password will be the first letter of the `primary_poc`'s first name (lowercase), then the last letter of their first name (lowercase), the first letter of their last name (lowercase), the last letter of their last name (lowercase), the number of letters in their first name, the number of letters in their last name, and then the name of the company they are working with, all capitalized with no spaces.

CONCAT Solutions

```
WITH t1 AS ( SELECT LEFT(primary_poc, STRPOS(primary_poc, ' ') -1 ) first_name, RIGHT
(primary_poc, LENGTH(primary_poc) - STRPOS(primary_poc, ' ')) last_name, nameFROM acco
unts)
SELECT first_name, last_name, CONCAT(first_name, '.', last_name, '@', name, '.com')
FROM t1;
```

```
WITH t1 AS ( SELECT LEFT(primary_poc, STRPOS(primary_poc, ' ') -1 ) first_name, RIGHT
(primary_poc, LENGTH(primary_poc) - STRPOS(primary_poc, ' ')) last_name, nameFROM acco
unts)
SELECT first_name, last_name, CONCAT(first_name, '.', last_name, '@', REPLACE(name, '
 ', ''), '.com')
FROM t1;
```

```
WITH t1 AS ( SELECT LEFT(primary_poc, STRPOS(primary_poc, ' ') -1 ) first_name, RIGHT
(primary_poc, LENGTH(primary_poc) - STRPOS(primary_poc, ' ')) last_name, nameFROM acco
unts)
SELECT first_name, last_name, CONCAT(first_name, '.', last_name, '@', name, '.com'), L
EFT(LOWER(first_name), 1) || RIGHT(LOWER(first_name), 1) || LEFT(LOWER(last_name), 1)
 || RIGHT(LOWER(last_name), 1) || LENGTH(first_name) || LENGTH(last_name) || REPLACE(U
PPER(name), ' ', '')
FROM t1;
```

COALESCE

Syntax

- **COALESCE:** Returns the first non-null value in a list.

```
COALESCE(val1, val2, val3);
```

```
COALESCE(hourly_wage*40*52, salary, commission*sales) AS annual_income;
```

Use Case

If there are multiple columns that have a combination of null and non-null values and the user needs to extract the first non-null value, he/she can use the coalesce function.

COALESCE is a command that helps you deal with null values. Now before using COALESCE, take a step back and think through how'd you like to deal with missing values in the first place.

The three methods below are the most common ways to deal with null values in SQL:

1. *Coalesce:* Allows you to return the first non-null value across a set of columns in a slick, single command. This is a good approach only if a single column's value needs to be extracted whilst the rest are null.
2. *Drop records:* Sometimes, if there are null values in records at all, analysts can decide to drop the row entirely. This is not favorable, as it removes data. Data is precious. Think about the reason those values are null. Does it make sense to use COALESCE, drop records, and conduct an imputation.
3. *Imputation:* Outside of the COALESCE use case, you may want to impute missing values. If so, think about the problem you are trying to solve, and impute accordingly. Perhaps you'd like to be conversative so you take the MIN of that

column or the 25th percentile value. Classic imputation values are often the median or mean value of the column.

Recap

Remember to follow the steps outlined below when starting off with data cleaning in SQL.

1. Review the problem statement.
2. What data do you have? What data do you need?
3. How will you adjust existing data or create new columns?
4. Leverage cleaning techniques to manipulate data.
5. Leverage analysis techniques to determine the solution.

For a reminder on any of the data cleaning functionality, the concepts in this lesson are labeled according to the functions you learned. If you felt uncomfortable with any of these functions at first, that is normal - these take some getting used to. Don't be afraid to take a second pass through the material to sharpen your skills!

Memorizing all of this functionality isn't necessary, but you do need to be able to follow documentation and learn from what you have done in solving previous problems to solve new problems.

Normalization: Standardizing or “cleaning up a column” by transforming it in some way to make it ready for analysis. A few normalization techniques are below:

- Adjusting a column that includes multiple currencies to one common currency
- Adjusting the varied distribution of a column value by transforming it into a z-score
- Converting all price into a common metric (e.g., price per ounce)

WINDOW FUNCTIONS

Introduction to Window Functions

Lesson Overview

In this lesson, we will cover window functions. Window functions are primarily used in two ways:

1. To understand a running total or a running metric while maintaining individual records
2. To rank a dataset

At the end of this lesson, you will be able to create windows functions using:

- Core Functions
- Ranking Functions
- Advanced Functions

Window Functions in Real-world Applications

Window functions are sometimes confusing to figure out. When exactly do you use them? When are they appropriate over aggregate functions? There are *two use cases* that I've experienced where window functions are particularly helpful.

1. When you want to measure trends or changes over rows or records in your data.
2. When you want to rank a column for outreach or prioritization.

A few cases where you'd have these needs are described below:

- Measuring change over time:
- Has the average price of airline tickets gone up this year?
- What's the best way to keep the running total orders of customers?
- The ranking used for outreach prioritization:
    - Use a combination of factors to rank companies most likely to need a loan.

What is a Window Function?

What is a window function?

Window Function: A window function is a calculation across a set of rows in a table that are somehow related to the current row. This means we're typically:

1. Calculating running totals that incorporate the current row or,
2. Ranking records across rows, inclusive of the current one

A window function is similar to aggregate functions combined with group by clauses but have one key difference: **Window functions retain the total number of rows between the input table and the output table (or result).** Behind the scenes, the window function is able to access more than just the current row of the query result.

When window functions are used, you'll notice new column names like the following:

- Average running price
- Running total orders
- Running sum sales
- Rank
- Percentile

Resources

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

Code from the Video*

```
SELECT order_id,
       order_date,
       SUM(order_total) OVER
          (PARTITION BY month(order_date) ORDER BY order_date)
          AS running_monthly_sales
FROM amazon_db
WHERE order_date>'2017-01-01';
```

Terms to be Covered

The following terms will be covered over the course of this lesson broken into three types of window functions: core, ranking, and advanced. Note that the naming of these window functions are specific to this course. Most users call all of the functions below "window functions."

Core Window Functions

- **Partition by:** A subclause of the OVER clause. Similar to GROUP BY.
- **Over:** Typically precedes the partition by that signals what to "GROUP BY".

- **Aggregates:** Aggregate functions that are used in window functions, too (e.g., sum, count, avg).

Ranking Window Functions

- **Row_number():** Ranking function where each row gets a different number.
- **Rank():** Ranking function where a row could get the same rank if they have the same value.
- **Dense_rank():** Ranking function similar to rank() but ranks are not skipped with ties.

Advanced Window Functions

- **Aliases:** Shorthand that can be used if there are several window functions in one query.
- **Percentiles:** Defines what percentile a value falls into over the entire table.
- **Lag/Lead:** Calculating differences between rows' values.

Core Window Functions

```
#Syntax

AGGREGATE_FUNCTION (column_1) OVER
 (PARTITION BY column_2 ORDER BY column_3)
  AS new_column_name;
```

As a reminder, a window function allows users to compare one row to another without doing any joins. Window functions are effective when you want to measure trends over time or rank a specific column, and it retains the total number of records without collapsing or condensing any of the original datasets.

There are a few key terms to review as a part of understanding core window functions:

- **PARTITION BY:** A subclause of the OVER clause. I like to think of PARTITION BY as the GROUP BY equivalent in window functions. PARTITION BY allows you to determine what you'd like to "group by" within the window function. Most often, you are partitioning by a month, region, etc. as you are tracking changes over time.
- **OVER:** This syntax signals a window function and precedes the details of the window function itself.

The sequence of Code for Window Functions

Typically, when you are writing a window function that tracks changes or a metric over time, you are likely to structure your syntax with the following components:

1. An aggregation function (e.g., sum, count, or average) + the column you'd like to track
2. OVER
3. PARTITION BY + the column you'd like to "group by"
4. ORDER BY (optional and is often a date column)
5. AS + the new column name

We'll review the key differences between aggregate functions/group by queries and window functions in the next section of this lesson.

Resources

This resource from Pinal Dave is helpful.

Creating a Running Total Using Window Functions

Create a running total of `standard_amt_usd` (in the `orders` table) over order time with no date truncation. Your final table should have two columns: one with the amount being added for each new row, and a second with the running total.

```
SELECT standard_amt_usd,
       SUM(standard_amt_usd) OVER
       (ORDER BY occurred_at) running_total
FROM orders;
```

Creating a *Partitioned* Running Total Using Window Functions

Now, modify your query from the previous quiz to include partitions. Still create a running total of `standard_amt_usd` (in the `orders` table) over order time, but this time, date truncate `occurred_at` by year and partition by that same year-truncated `occurred_at` variable.

Your final table should have three columns:

- One with the amount being added for each row

- One for the truncated date,
- A final column with the running total within each year

```
SELECT standard_amt_usd,
        DATE_TRUNC('year',occurred_at),
        SUM(standard_amt_usd) OVER
        (PARTITION BY DATE_TRUNC('year',occurred_at)
            ORDER BY occurred_at)
FROM orders;
```

Group By vs. Window Functions

To emphasize the similarities and differences between group by/aggregation queries and window functions, let's review the details below.

Similarities

Both groups by/aggregation queries and window functions serve the same use case. Synthesizing information over time and often grouped by a column (e.g., a region, month, customer group, etc.)

Differences

The difference between group by/aggregation queries and window functions is simple. The output of window functions retains all individual records whereas the group by/aggregation queries condense or collapse information.

Key Notes

- 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**.
- Feel free to use as many window functions as you'd like in a single query. E.g., if you'd like to have an average, sum, and count aggregate function that captures three metrics' running totals, go for it.

```
SELECT order_id,
       order_total,
       order_price,
       SUM(order_total) OVER
         (PARTITION BY month(order_date) ORDER BY order_date)
              AS running_monthly_sales,
       COUNT(order_id) OVER
```

```
          (PARTITION BY month(order_date) ORDER BY order_date)
               AS running_monthly orders,
        AVG(order_price) OVER
          (PARTITION BY month(order_date) ORDER BY order_date)
               AS average_monthly_price
FROM  amazon_sales_db
WHERE order_date < '2017-01-01';
```

Now that you understand how window functions work, let's practice applying the same aggregates that you would under normal circumstances.

```
SELECT id,
       account_id,
       standard_qty,
       DATE_TRUNC('month', occurred_at) AS month,
       DENSE_RANK() OVER
        (PARTITION BY account_id
        ORDER BY DATE_TRUNC('month', occurred_at)) AS dense_rank,
       SUM(standard_qty) OVER
         (PARTITION BY account_id
         ORDER BY DATE_TRUNC('month', occurred_at)) sum_standard_qty,
       COUNT(standard_qty) OVER
        (PARTITION BY account_id
        ORDER BY DATE_TRUNC('month', occurred_at))
        count_standard_qty,
       AVG(standard_qty)OVER
        (PARTITION BY account_id
        ORDER BY DATE_TRUNC('month', occurred_at)) avg_standard_qty,
      MIN(standard_qty) OVER
       (PARTITION BY account_id
       ORDER BY DATE_TRUNC('month', occurred_at)) min_standard_qty,
      MAX(standard_qty) OVER
       (PARTITION BY account_id
        ORDER BY DATE_TRUNC('month', occurred_at)) max_standard_qty
FROM orders
```

Aggregates in Window Functions with and without ORDER BY

The `ORDER BY` clause is one of two clauses integral to window functions. The `ORDER` and `PARTITION` define what is referred to as the “window”—the ordered subset of data over which calculations are made. Removing `ORDER BY` just leaves an unordered partition; in our query's case, each column's value is simply an aggregation (e.g., sum, count, average, minimum, or maximum) of all the `standard_qty` values in its respective `account_id`.

As Stack Overflow user mathguy explains:

> The easiest way to think about this - leaving the ORDER BY out is equivalent to "ordering" in a way that all rows in the partition are "equal" to each other. Indeed, you can get the same effect by explicitly adding the ORDER BY clause like this: ORDER BY 0 (or "order by" any constant expression), or even, more emphatically, ORDER BY NULL.

Ranking Window Functions

There are three types of ranking functions that serve the same use case: how to take a column and rank its values. The choice of which ranking function to use is up to the SQL user, often created in conjunction with someone on a customer or business team.

- **Row_number():** Ranking is **distinct** amongst records even with ties in what the table is ranked against.
- **Rank():** Ranking is **the same amongst tied values** and ranks **skip** for subsequent values.
- **Dense_rank():** Ranking is **the same amongst tied values** and ranks **do not skip** for subsequent values.

```
SELECT ROW_NUMBER() OVER(ORDER BY date_time) AS rank,
       date_time
FROM db;
```

Query2

```
SELECT RANK() OVER(ORDER BY date_time) AS rank,
       date_time
FROM   db;
```

Query3

```
SELECT DENSE_RANK() OVER(ORDER BY date_time) AS rank,
       date_time
FROM   db;
```

Ranking Total Paper Ordered by Account

Select the `id`, `account_id`, and `total` variable from the `orders` table, then create a column called `total_rank` that ranks this total amount of paper ordered (from highest to lowest) *for each account* using a partition. Your final table should have these four columns.

```
SELECT id,
       account_id,
       total,
       RANK() OVER
         (PARTITION BY account_id
          ORDER BY total DESC) AS total_rank
FROM orders;
```

Advanced Functions: Aliases for Multiple Window Functions

Aliases Use Case

If you are planning to write multiple window functions that leverage the same PARTITION BY, OVER, and ORDER BY in a single query, leveraging aliases will help tighten your syntax.

Details of Aliases

- A **monthly_window** alias function is defined at the end of the query in the **WINDOW** clause.
- It is then called on **each time** an aggregate function is used within the SELECT clause.

Code from the Video*

```
SELECT order_id,
       order_total,
       order_price,
       SUM(order_total)OVER monthly_window AS running_monthly_sales,
       COUNT(order_id)OVER monthly_window AS running_monthly_orders,
       AVG(order_price)OVER monthly_window AS average_monthly_price
FROM amazon_sales_db
WHERE order_date < '2017-01-01'
WINDOW monthly_window AS
       (PARTITION BY month(order_date) ORDER BY order_date);
```

```
SELECT id,
       account_id,
       DATE_TRUNC('year',occurred_at) AS year,
       DENSE_RANK() OVER (PARTITION BY account_id ORDER BY DATE_TRUNC('year',occurred_
at)) AS dense_rank,
       total_amt_usd,
       SUM(total_amt_usd) OVER (PARTITION BY account_id ORDER BY DATE_TRUNC('year',occ
urred_at)) AS sum_total_amt_usd,
       COUNT(total_amt_usd) OVER (PARTITION BY account_id ORDER BY DATE_TRUNC('year',o
ccurred_at)) AS count_total_amt_usd,
       AVG(total_amt_usd) OVER (PARTITION BY account_id ORDER BY DATE_TRUNC('year',occ
urred_at)) AS avg_total_amt_usd,
       MIN(total_amt_usd) OVER (PARTITION BY account_id ORDER BY DATE_TRUNC('year',occ
urred_at)) AS min_total_amt_usd,
       MAX(total_amt_usd) OVER (PARTITION BY account_id ORDER BY DATE_TRUNC('year',occ
urred_at)) AS max_total_amt_usd
FROM orders
```

Now, create and use an alias to shorten the following query (which is ***different*** from the one in the Aggregates in Windows Functions video) that has multiple window functions. Name the alias `account_year_window`, which is more descriptive than `main_window` in the example above.

```
SELECT id,account_id,
DATE_TRUNC('year',occurred_at) _year,
DENSE_RANK() OVER account_year_window,
total_amt_usd AS t,
SUM(total_amt_usd) OVER account_year_window,
COUNT(total_amt_usd) OVER account_year_window,
AVG(total_amt_usd) OVER account_year_window,
MIN(total_amt_usd) OVER account_year_window,
MAX(total_amt_usd) OVER account_year_window
FROM orders
WINDOW account_year_window AS
(PARTITION BY account_id ORDER BY DATE_TRUNC('year',occurred_at));
```

Comparing a Row to Previous Row - LAG

LAG function

Purpose

It returns the value from a previous row to the current row in the table.

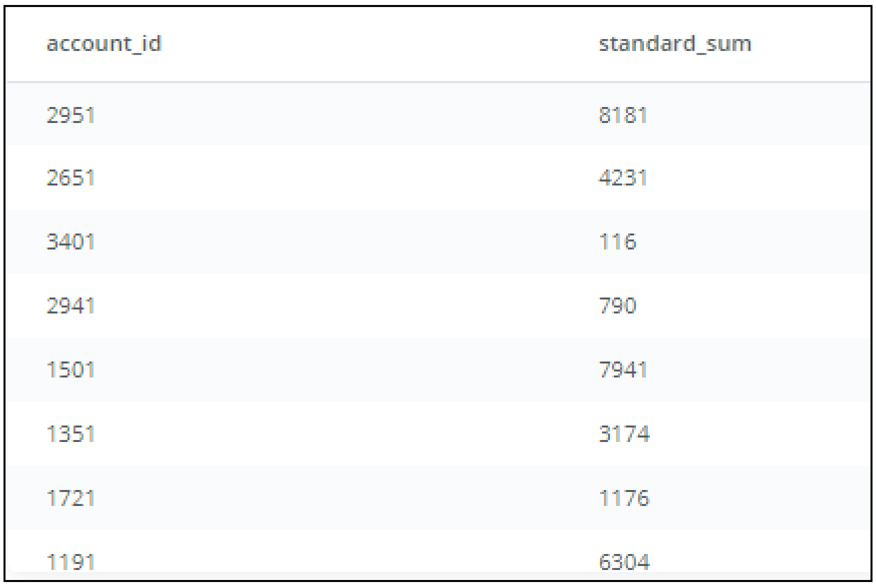
Step 1:

Let’s first look at the **inner query** and see what this creates.

```
SELECT     account_id, SUM(standard_qty)AS standard_sum
FROM       orders
GROUPBY    1
```

What you see after running this SQL code:

1. The query sums the standard_qty amounts for each account_id to give the standard paper each account has purchased overall time. E.g., account_id 2951 has purchased 8181 units of standard paper.
2. Notice that the results are not ordered by account_id or standard_qty.



account_id	standard_sum
2951	8181
2651	4231
3401	116
2941	790
1501	7941
1351	3174
1721	1176
1191	6304

Step 2:

We start building the **outer query**, and name the inner query as `sub`.

```
SELECT account_id, standard_sum
FROM   (
SELECT   account_id,SUM(standard_qty)AS standard_sum
FROM     orders
GROUPBY 1
       ) sub
```

This still returns the same table you see above, which is also shown below.

account_id	standard_sum
2951	8181
2651	4231
3401	116
2941	790
1501	7941

Step 3 (Part A): We add the Window Function `OVER (ORDER BY standard_sum)` in the outer query that will create a result set in ascending order based on the *standard_sum* column.

```
SELECT account_id,
       standard_sum,
       LAG(standard_sum)OVER (ORDERBY standard_sum)AS lag
FROM   (
SELECT   account_id,SUM(standard_qty)AS standard_sum
FROM     orders
GROUPBY 1
       ) sub
```

This ordered column will set us up for the other part of the Window Function (see below).

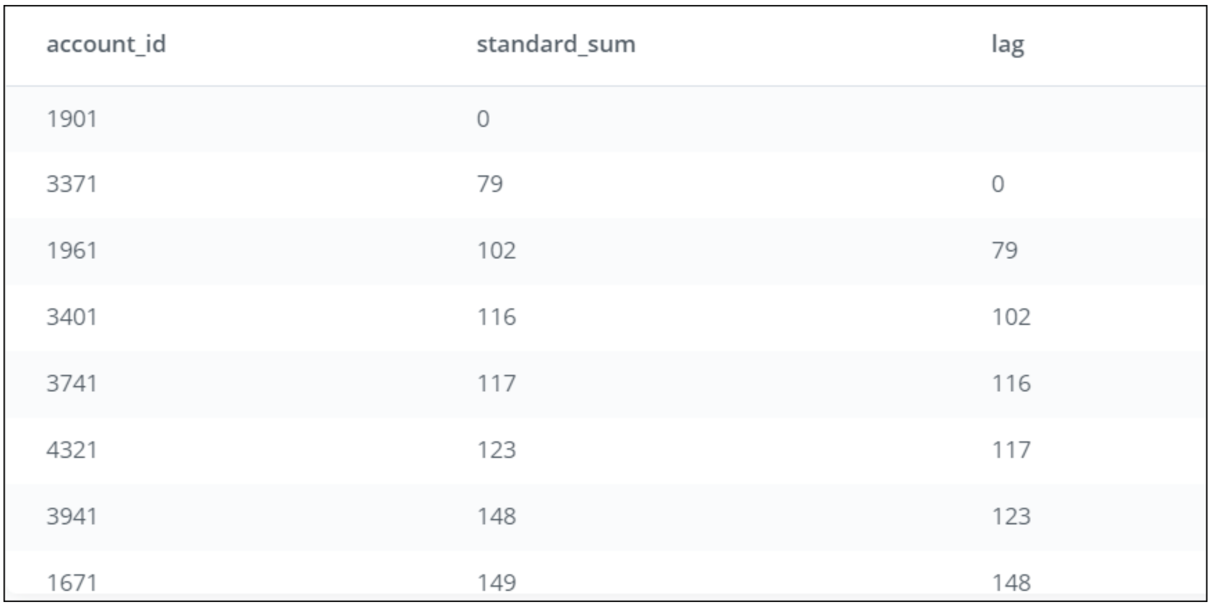
Step 3 (Part B):

The `LAG` function creates a new column called *lag* as part of the **outer query**: `LAG(standard_sum) OVER (ORDER BY standard_sum) AS lag`. This new column named *lag* uses the values from the ordered *standard_sum* (Part A within Step 3).

```
SELECT account_id,
       standard_sum,
       LAG(standard_sum)OVER (ORDERBY standard_sum)AS lag
FROM   (
SELECT   account_id,
SUM(standard_qty)AS standard_sum
FROM     orders
GROUPBY 1
       ) sub
```

Each row's value in *lag* is pulled from the previous row. E.g., for account_id 1901, the value in *lag* will come from the previous row. However, since there is no previous row to pull from, the value in *lag* for account_id 1901 will be NULL. For account_id 3371, the value in *lag* will be pulled from the previous row (i.e., account_id 1901), which will be 0. This goes on for each row in the table.

What you see after running this SQL code:



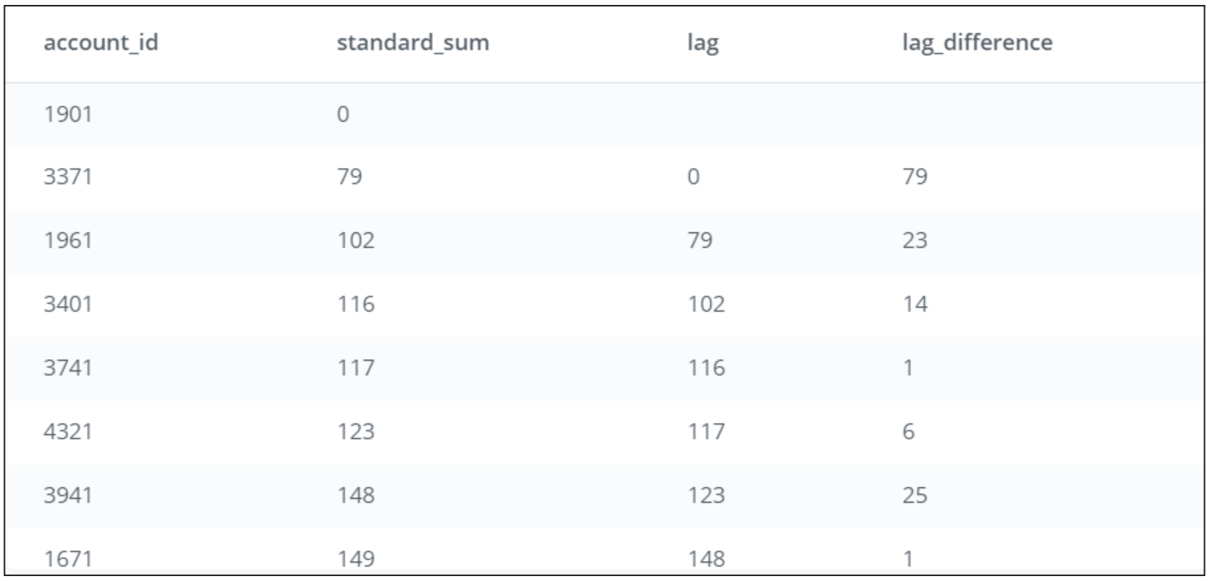
account_id	standard_sum	lag
1901	0	
3371	79	0
1961	102	79
3401	116	102
3741	117	116
4321	123	117
3941	148	123
1671	149	148

Step 4: To compare the values between the rows, we need to use both columns (*standard_sum* and *lag*). We add a new column named `lag_difference`, which subtracts the *lag* value from the value in *standard_sum* for each row in the table:`standard_sum - LAG(standard_sum) OVER (ORDER BY standard_sum) AS lag_difference`

```
SELECT account_id,
       standard_sum,
       LAG(standard_sum)OVER (ORDERBY standard_sum)AS lag,
       standard_sum - LAG(standard_sum)OVER (ORDERBY standard_sum)AS lag_difference
FROM (
SELECT account_id,
SUM(standard_qty)AS standard_sum
FROM orders
GROUPBY 1
      ) sub
```

Each value in *lag_difference* is comparing the row values between the 2 columns (*standard_sum* and *lag*). E.g., since the value for *lag* in the case of account_id 1901 is NULL, the value in *lag_difference* for account_id 1901 will be NULL. However, for account_id 3371, the value in *lag_difference* will compare the value 79 (*standard_sum* for account_id 3371) with 0 (*lag* for account_id 3371) resulting in 79. This goes on for each row in the table.

What you see after running this SQL code:



account_id	standard_sum	lag	lag_difference
1901	0		
3371	79	0	79
1961	102	79	23
3401	116	102	14
3741	117	116	1
4321	123	117	6
3941	148	123	25
1671	149	148	1

Comparing a Row to Previous Row - LEAD

Now let's look at the LEAD function.

LEAD function

Purpose: Return the value from the row following the current row in the table.

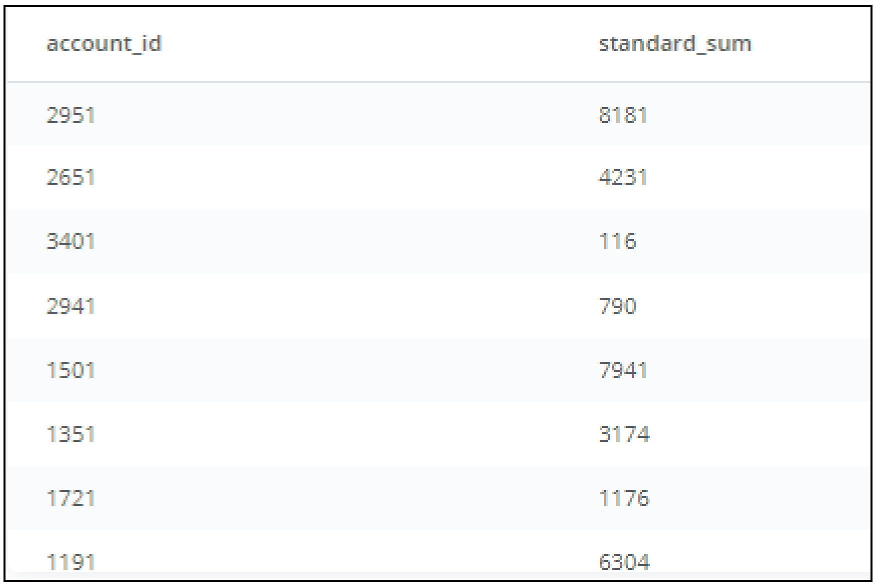
Step 1:

Let's first look at the **inner query** and see what this creates.

```
SELECT     account_id,
SUM(standard_qty)AS standard_sum
FROM       orders
GROUPBY   1
```

What you see after running this SQL code:

1. The query sums the standard_qty amounts for each account_id to give the standard paper each account has purchased over all time. E.g., account_id 2951 has purchased 8181 units of standard paper.
2. Notice that the results are not ordered by account_id or standard_qty.



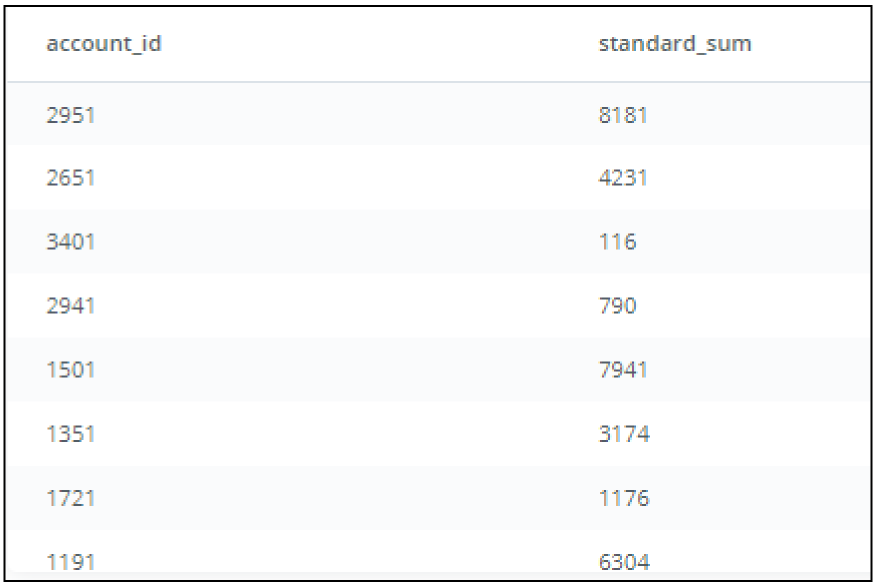
account_id	standard_sum
2951	8181
2651	4231
3401	116
2941	790
1501	7941
1351	3174
1721	1176
1191	6304

Step 2:

We start building the **outer query**, and name the inner query as `sub`.

```
SELECT account_id,
       standard_sum
FROM   (
SELECT   account_id,
SUM(standard_qty)AS standard_sum
FROM     orders
GROUPBY 1
       ) sub
```

This will produce the same table as above, but sets us up for the next part.



account_id	standard_sum
2951	8181
2651	4231
3401	116
2941	790
1501	7941
1351	3174
1721	1176
1191	6304

Step 3 (Part A):

We add the Window Function `(OVER BY standard_sum)` in the outer query that will create a result set ordered in ascending order of the *standard_sum* column.

```
SELECT account_id,
       standard_sum,
LEAD(standard_sum)OVER (ORDERBY standard_sum)ASleadFROM   (
SELECT   account_id,
SUM(standard_qty)AS standard_sum
FROM     orders
GROUPBY 1
       ) sub
```

This ordered column will set us up for the other part of the Window Function (see below).

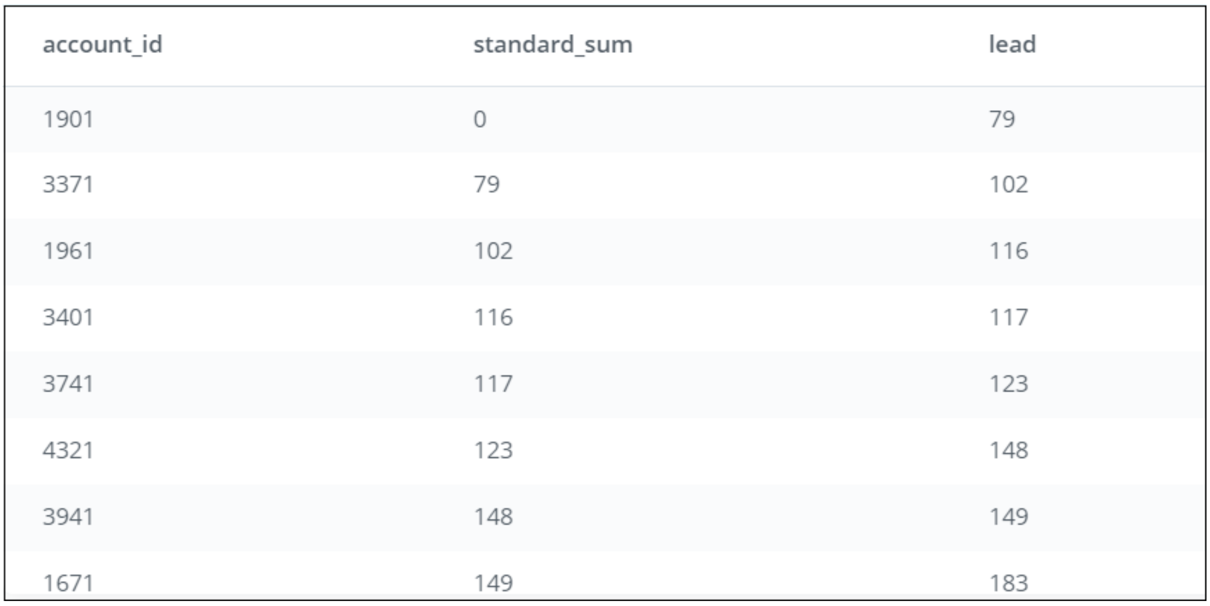
Step 3 (Part B):

The `LEAD` function in the Window Function statement creates a new column called *lead* as part of the outer query: `LEAD(standard_sum) OVER (ORDER BY standard_sum) AS lead`

This new column named *lead* uses the values from *standard_sum* (in the ordered table from Step 3 (Part A)). Each row's value in *lead* is pulled from the row after it. E.g., for account_id 1901, the value in *lead* will come from the row following it (i.e., for account_id 3371). Since the value is 79, the value in *lead* for account_id 1901 will be 79. For account_id 3371, the value in *lead* will be pulled from the following row (i.e., account_id 1961), which will be 102. This goes on for each row in the table.

```
SELECT account_id,
       standard_sum,
LEAD(standard_sum)OVER (ORDERBY standard_sum)ASleadFROM   (
SELECT   account_id,
SUM(standard_qty)AS standard_sum
FROM     orders
GROUPBY 1
       ) sub
```

What you see after running this SQL code:



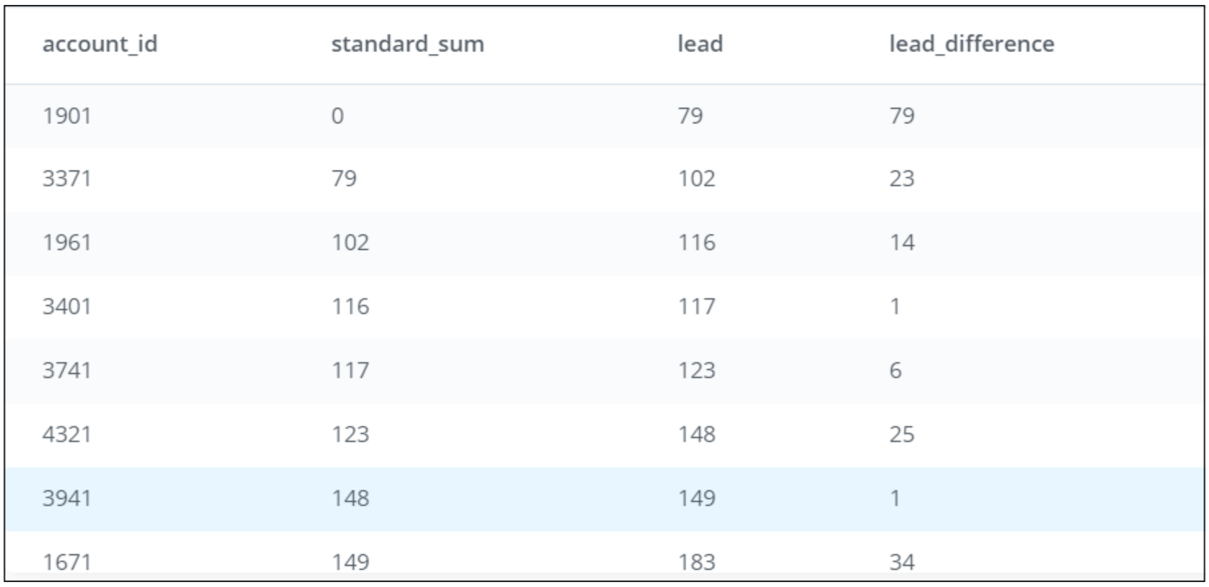
account_id	standard_sum	lead
1901	0	79
3371	79	102
1961	102	116
3401	116	117
3741	117	123
4321	123	148
3941	148	149
1671	149	183

Step 4: To compare the values between the rows, we need to use both columns (*standard_sum* and *lag*). We add a column named *lead_difference*, which subtracts the value in *standard_sum* from *lead* for each row in the table: `LEAD(standard_sum) OVER (ORDER BY standard_sum) - standard_sum AS lead_difference`

```
SELECT account_id,
       standard_sum,
LEAD(standard_sum)OVER (ORDERBY standard_sum)ASlead,
LEAD(standard_sum)OVER (ORDERBY standard_sum) - standard_sumAS lead_difference
FROM (
SELECT account_id,
SUM(standard_qty)AS standard_sum
FROM orders
GROUPBY 1
     ) sub
```

Each value in *lead_difference* is comparing the row values between the 2 columns (*standard_sum* and *lead*). E.g., for account_id 1901, the value in *lead_difference* will compare the value 0 (*standard_sum* for account_id 1901) with 79 (*lead* for account_id 1901) resulting in 79. This goes on for each row in the table.

What you see after running this SQL code:



account_id	standard_sum	lead	lead_difference
1901	0	79	79
3371	79	102	23
1961	102	116	14
3401	116	117	1
3741	117	123	6
4321	123	148	25
3941	148	149	1
1671	149	183	34

LAG and LEAD

Use Case

When you need to compare the values in adjacent rows or rows that are offset by a certain number, LAG and LEAD come in very handy.

Scenarios for using LAG and LEAD functions

You can use LAG and LEAD functions whenever you are trying to compare the values in adjacent rows or rows that are offset by a certain number.

Example 1: You have a sales dataset with the following data and need to compare how the market segments fare against each other on profits earned.

Market Segment	Profits earned by each market segment
A	$550
B	$500
C	$670
D	$730
E	$982

Example 2: You have an inventory dataset with the following data and need to compare the number of days elapsed between each subsequent order placed for Item A.

Inventory	Order_id	Dates when orders were placed

Item A	001	11/2/2017
Item A	002	11/5/2017
Item A	003	11/8/2017
Item A	004	11/15/2017
Item A	005	11/28/2017

As you can see, these are useful data analysis tools that you can use for more complex analysis!

Comparing a Row to the Previous Row

In the previous video, Derek outlines how to compare a row to a previous or subsequent row. This technique can be useful when analyzing time-based events. Imagine you're an analyst at Parch & Posey and you want to determine how the current order's total revenue ("total" meaning from sales of all types of paper) compares to the next order's total revenue.

Modify Derek's query from the previous video in the SQL Explorer below to perform this analysis. You'll need to use `occurred_at` and `total_amt_usd` in the `orders` table along with `LEAD` to do so. In your query results, there should be four columns: `occurred_at`, `total_amt_usd`, `lead`, and `lead_difference`.

```
SELECT occurred_at,
       total_amt_usd,
LEAD(total_amt_usd)OVER (ORDERBY occurred_at)ASlead,
LEAD(total_amt_usd)OVER (ORDERBY occurred_at) - total_amt_usdAS lead_difference
FROM (
SELECT occurred_at,
SUM(total_amt_usd)AS total_amt_usd
FROM orders
GROUPBY 1
) sub
```

Percentiles

Percentiles Use Case

When there are a large number of records that need to be ranked, individual ranks (e.g., 1, 2, 3, 4…) are ineffective in helping teams determine the best of the distribution from the rest. Percentiles help better describe large datasets. For example, a team might want to reach out to the Top 5% of customers.

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).

Percentiles Syntax

The following components are important to consider when building a query with percentiles:

1. NTILE + the number of buckets you’d like to create within a column (e.g., 100 buckets would create traditional percentiles, 4 buckets would create quartiles, etc.)
2. OVER
3. ORDER BY (optional, typically a date column)
4. AS + the new column name

Expert Tip

In cases with relatively few rows in a window, the `NTILE` function doesn’t calculate exactly as you might expect. For example, 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.

In other words, when you use an NTILE function but the number of rows in the partition is less than the NTILE(number of groups), then NTILE will divide the rows into as many groups as there are members (rows) in the set but then stop short of the requested number of groups. If you’re working with very small windows, keep this in mind and consider using quartiles or similarly small bands.

Finally, we're actually creating a new column and calling it percentile.

```
NTILE(# of buckets) OVER (ORDER BY ranking_column) AS new_column_name

SELECT  customer_id,
        composite_score,
        NTILE(100) OVER(ORDER BY composite_score) AS percentile
FROM    customer_lead_score;
```

In the SQL Explorer below, write three queries (separately) that reflect each of the following:

1. Use the `NTILE` functionality to divide the accounts into 4 levels in terms of the amount of `standard_qty` for their orders. Your resulting table should have the `account_id`, the `occurred_at` time for each order, the total amount of `standard_qty` paper purchased, and one of four levels in a `standard_quartile` column.
2. Use the `NTILE` functionality to divide the accounts into two levels in terms of the amount of `gloss_qty` for their orders. Your resulting table should have the `account_id`, the `occurred_at` time for each order, the total amount of `gloss_qty` paper purchased, and one of two levels in a `gloss_half` column.
3. Use the `NTILE` functionality to divide the orders for each account into 100 levels in terms of the amount of `total_amt_usd` for their orders. Your resulting table should have the `account_id`, the `occurred_at` time for each order, the total amount of `total_amt_usd` paper purchased, and one of 100 levels in a `total_percentile` column.

Note: To make it easier to interpret the results, order by the account_id in each of the queries.

Percentiles with Partitions

```
SELECT
       account_id,
       occurred_at,
       standard_qty,
       NTILE(4)OVER (PARTITIONBY account_idORDERBY standard_qty)AS standard_quartile
FROM orders
ORDERBY account_idDESC
```

```
SELECT
       account_id,
       occurred_at,
       gloss_qty,
       NTILE(2)OVER (PARTITIONBY account_idORDERBY gloss_qty)AS gloss_half
FROM orders
ORDERBY account_idDESC
```

```
SELECT
       account_id,
       occurred_at,
       total_amt_usd,
       NTILE(100)OVER (PARTITIONBY account_idORDERBY total_amt_usd)AS total_percentile
```

```
FROM orders
ORDERBY account_idDESC
```

Window Functions Conclusions

Lesson Overview

Great job for getting through the Window Functions lesson, an advanced SQL analysis topic. You can now create windows functions using:

- Core Functions
- Ranking Functions
- Advanced Functions

Key Takeaways:

- Window functions are similar to aggregate/group by functions.
- Window functions maintain the total number of rows from the original dataset.
- Window functions are typically used in the following ways:
  - Measure and/or track changes over time.
  - Rank a column to be used for outreach and/or prioritization.
- If you are planning to write multiple window functions that leverage the same PARTITION BY, OVER, and ORDER BY in a single query, leveraging aliases will help tighten your syntax.

Additional Readings:

This documentation on Window Functions is helpful to learn more.

Introduction to Advanced SQL

Lesson Overview

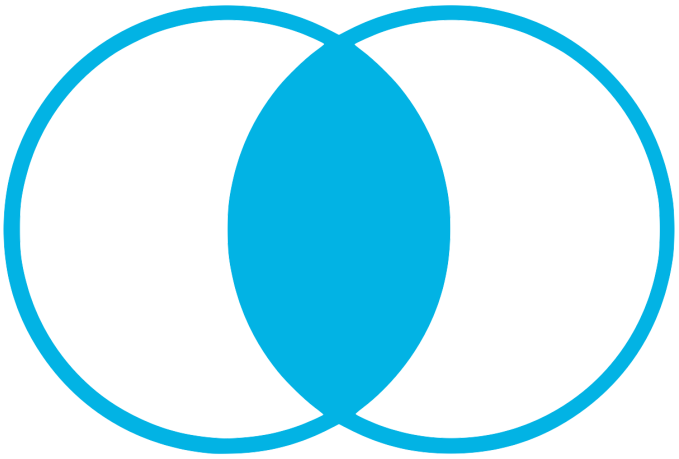
In this last lesson, you will:

- Create fast-running queries with advanced joins
- Evaluate business questions
- Tune the performance of queries
- Create solutions for edge-cases

FULL OUTER JOIN

In earlier lessons, we covered inner joins, which produce results for which the join condition is matched in both tables.

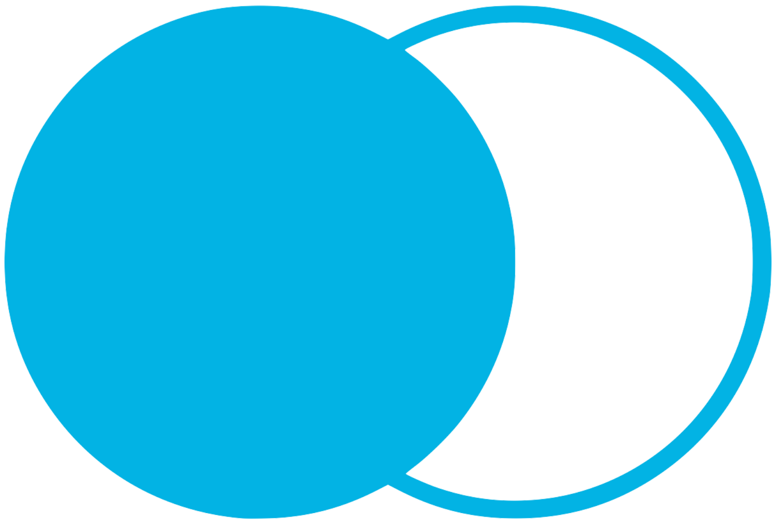
Venn diagrams, which are helpful for visualizing table joins, are provided below along with sample queries. Consider the circle on the left Table A and the circle on the right Table B.



`INNER JOIN` Venn Diagram

```
SELECT column_name(s)
FROM Table_A
INNERJOIN Table_B ON Table_A.column_name = Table_B.column_name;
```

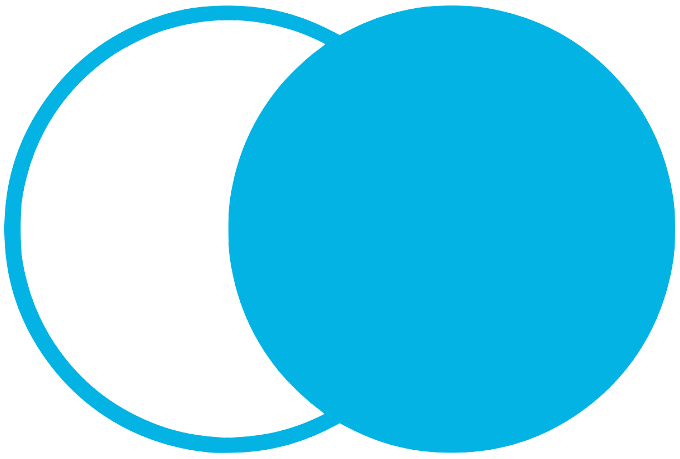
Left joins also include unmatched rows from the left table, which is indicated in the “FROM” clause.



`LEFT JOIN` Venn Diagram

```
SELECT column_name(s)
FROM Table_A
LEFT JOIN Table_B ON Table_A.column_name = Table_B.column_name;
```

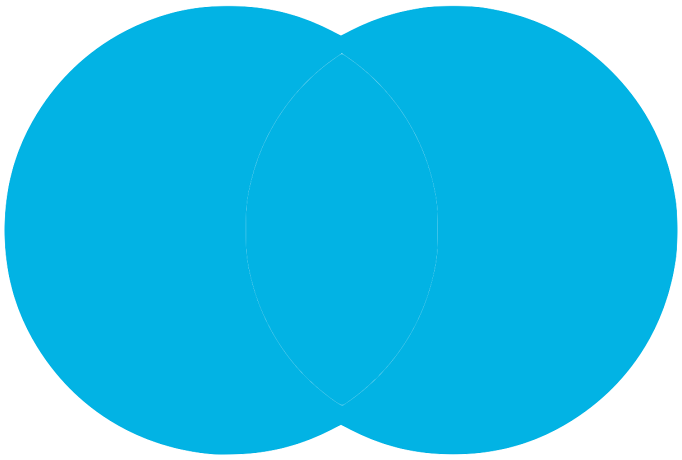
Right joins are similar to left joins, but include unmatched data from the right table -- the one that's indicated in the JOIN clause.



`RIGHT JOIN` Venn Diagram

```
SELECT column_name(s)
FROM Table_A
RIGHT JOIN Table_B ON Table_A.column_name = Table_B.column_name;
```

In some cases, you might want to include unmatched rows from *both* tables being joined. You can do this with a full outer join.



`FULL OUTER JOIN` Venn Diagram

```
SELECT column_name(s)
FROM Table_A
FULL OUTER JOIN Table_B ON Table_A.column_name = Table_B.column_name;
```

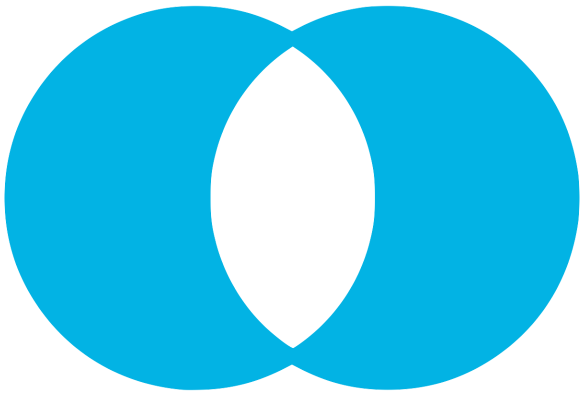
A common application of this is when joining two tables on a timestamp. Let's say you've got one table containing the number of *item 1* sold each day, and another containing the number of *item 2* sold. If a certain date, like January 1, 2018, exists in the left table but not the right, while another date, like January 2, 2018, exists in the right table but not the left:

- a left join would drop the row with January 2, 2018, from the result set
- a right join would drop January 1, 2018, from the result set

The only way to make sure both January 1, 2018, and January 2, 2018, make it into the results is to do a full outer join. A full outer join returns unmatched records in each table with null values for the columns that came from the opposite table.

If you wanted to return unmatched rows only, which is useful for some cases of data assessment, you can isolate them by adding the following line to the end of the query:

```
WHERE Table_A.column_name ISNULLOR Table_B.column_name IS NULL
```



`FULL OUTER JOIN` with `WHERE A.Key IS NULL OR B.Key IS NULL` Venn Diagram

Joining without an Equals Sign

Inequality JOINs

Expert Tip

If you recall from earlier lessons on joins, the join clause is evaluated before the where clause -- filtering in the join clause will eliminate rows before they are joined, while filtering in the WHERE clause will leave those rows in and produce some nulls.

```
SELECT orders.id,
       orders.occurred_at  AS order_date,
       events.*
FROM   orders
LEFT JOIN web_events events
       ON events.account_id = orders.account_id
      AND events.occurred_at < orders.occurred_at
WHERE  DATE_TRUNC('month', orders.occurred_at)=
       (SELECT DATE_TRUNC('month', MIN(orders.occurred_at))
        FROM orders)
ORDER BY orders.occurred_at, orders.occurred_at
```

Inequality operators (a.k.a. comparison operators) don't only need to be date times or numbers, they also work on strings! You'll see how this works by completing the following quiz, which will also reinforce the concept of joining with comparison operators.

In the following SQL Explorer, write a query that left joins the `accounts` table and the `sales_reps` tables on each sale rep's ID number *and* joins it using the `<` comparison operator on `accounts.primary_poc` and `sales_reps.name`, like so:

```
accounts.primary_poc < sales_reps.name
```

The query results should be a table with three columns: the account name (e.g. Johnson Controls), the primary contact name (e.g. Cammy Sosnowski), and the sales representative's name (e.g. Samuel Racine). Then answer the subsequent multiple-choice question.

```
SELECT accounts.nameas account_name,
       accounts.primary_pocas poc_name,
       sales_reps.nameas sales_rep_name
FROM accounts
LEFTJOIN sales_reps
ON accounts.sales_rep_id = sales_reps.idAND accounts.primary_poc < sales_reps.name
```

For more details on how string comparison with `<` and `>` work in SQL, check out this excellent answer on Stack Overflow: SQL string comparison, greater than and less than operators

SELF JOIN

```
SELECT o1.idAS o1_id,
       o1.account_idAS o1_account_id,
       o1.occurred_atAS o1_occurred_at,
       o2.idAS o2_id,
       o2.account_idAS o2_account_id,
       o2.occurred_atAS o2_occurred_at
FROM   orders o1
LEFT JOIN orders o2
  ON o1.account_id = o2.account_id
  AND o2.occurred_at > o1.occurred_at
  AND o2.occurred_at <= o1.occurred_at + INTERVAL '28 days'
ORDER BY o1.account_id, o1.occurred_at
```

Expert Tip

This comes up pretty commonly in job interviews. Self JOIN logic can be pretty tricky -- you can see here that our join has three conditional statements. It is important to pause and think through each step when joining a table to itself.

Self JOINs

One of the most common use cases for self JOINs is in cases where two events occurred, one after another. As you may have noticed in the previous video, using inequalities in conjunction with self JOINs is common.

Modify the query from the previous video, which is pre-populated in the SQL Explorer below, to perform the same interval analysis except for the `web_events` table. Also:

- change the interval to 1 day to find those web events that occurred after, but not more than 1 day after, another web event
- add a column for the `channel` variable in both instances of the table in your query

You can find more on the types of INTERVALS (and other date-related functionality) in the Postgres documentation here.

```
SELECT we1.idAS we_id,
       we1.account_idAS we1_account_id,
       we1.occurred_atAS we1_occurred_at,
       we1.channelAS we1_channel,
       we2.idAS we2_id,
       we2.account_idAS we2_account_id,
       we2.occurred_atAS we2_occurred_at,
       we2.channelAS we2_channel
FROM web_events we1
LEFT JOIN web_events we2
ON we1.account_id = we2.account_id
AND we1.occurred_at > we2.occurred_at
AND we1.occurred_at <= we2.occurred_at + INTERVAL '1 day'
ORDER BY we1.account_id, we2.occurred_at
```

UNION

Appending Data via UNION

UNION Use Case

- The UNION operator is used to combine the result sets of 2 or more SELECT statements. It removes duplicate rows between the various SELECT statements.
- Each SELECT statement within the UNION must have the same number of fields in the result sets with similar data types.
- Typically, the use case for leveraging the UNION command in SQL is when a user wants to pull together distinct values of specified columns that are spread across multiple tables. For example, a chef wants to pull together the ingredients

and respective aisle across three separate meals that are maintained within different tables.

Details of UNION

- There must be the same number of expressions in both SELECT statements.
- The corresponding expressions must have the same data type in the SELECT statements.
- For example:
    - Expression1 must be the same data type in both the first and second SELECT statement.

Expert Tip

- UNION removes duplicate rows.
- UNION ALL does not remove duplicate rows.

Resources

The resource here on SQL UNIONs is helpful in understanding syntax and examples.

Appending Data via UNION Demonstration

SQL's two strict rules for appending data:

1. Both tables must have the same number of columns.
2. Those columns must have the same data types in the same order as the first table.

A common misconception is that column names have to be the same. Column names, in fact, **don't** need to be the same to append two tables but you will find that they typically are.

Code from the Video Above

```
CREATEVIEW web_events_2
AS (SELECT *FROM web_events)
```

```
SELECT *
FROM web_events
```

```
UNIONSELECT *
FROM web_events_2
```

Pretreating Tables before doing a UNION

```
CREATEVIEW web_events_2
AS (SELECT *FROM web_events)

SELECT *
FROM web_events
WHERE channel = 'facebook'
UNION ALL
SELECT *
FROM web_events_2
```

Performing Operations on a Combined Dataset

```
CREATEVIEW web_events_2
AS (SELECT *FROM web_events)

SELECT channel,
COUNT(*)AS sessions
FROM (
SELECT *
FROM web_events
UNION ALL
SELECT *
FROM web_events_2
     ) web_events
GROUPBY 1
ORDERBY 2DESC
```

```
CREATEVIEW web_events_2
AS (SELECT *FROM web_events)

WITH web_eventsAS (
SELECT *
FROM web_events
UNION ALL
SELECT *
FROM web_events_2
     )
SELECT channel,
COUNT(*)AS sessions
FROM  web_events
GROUPBY 1
ORDERBY 2DESC
```

Appending Data via UNION

Write a query that uses `UNION ALL` on two instances (and selecting all columns) of the `accounts` table. Then inspect the results and answer the subsequent quiz.

Pretreating Tables before doing a UNION

Add a `WHERE` clause to each of the tables that you unioned in the query above, filtering the first table where `name` equals Walmart and filtering the second table where `name` equals Disney. Inspect the results then answer the subsequent quiz.

Performing Operations on a Combined Dataset

Perform the union in your first query (under the **Appending Data via UNION** header) in a common table expression and name it `double_accounts`. Then do a `COUNT` the number of times a `name` appears in the `double_accounts` table. If you do this correctly, your query results should have a count of 2 for each `name`.

Quiz 1:

```
SELECT *
FROM accounts

UNION ALL

SELECT *
FROM accounts
```

Quiz 2:

```
SELECT *
FROM accounts
WHEREname = 'Walmart'

UNION ALL

SELECT *
FROM accounts
WHEREname = 'Disney'
```

Quiz 3:

```
WITH double_accounts AS (
SELECT *
FROM accounts

UNION ALL

SELECT *
FROM accounts
)

SELECTname,
COUNT(*)AS name_count
FROM double_accounts
GROUPBY 1
ORDERBY 2DESC
```

Performance Tuning Motivation

Modern databases can scale to work with huge amounts of data and SQL excels in its ability to work with these massive datasets. In this section, you will learn to identify when your queries can be improved and how to improve them for faster query results.

How You Can and Can't Control Performance

One way to make a query run faster is to reduce the number of calculations that need to be performed. Some of the high-level things that will affect the number of calculations a given query will make include:

- Table size
- Joins
- Aggregations

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

- Other users running queries concurrently on the database
- Database software and optimization (e.g., Postgres is optimized differently than Redshift)

Query1

```
SELECT accounts.name,
  COUNT(*)AS web_events
FROM accounts
JOIN web_events events
  ON events.account_id = accounts.id
GROUP BY 1
ORDER BY 2 DESC

#More than 9,000 rows in web_events table. Pre-aggregating can help to reduce the numb
er of rows being returned.
```

Query2

```
SELECT a.name,
       sub.web_events
FROM
  (SELECT account.id,
          COUNT AS web_events
  FROM web_events
  GROUP BY 1) sub
JOIN accounts a
ON a.id = sub.account_id
ORDER BY 2 DESC

#Inner query pre-aggregates the data and reduces the number of rows
being returned.
```

The second thing you can do is to make joins less complicated, that is, reduce the number of rows that need to be evaluated. It is better to reduce table sizes before joining them. This can be done by creating subqueries and joining them to an outer query. Aggregating before joining will improve query speed; however, be sure that what you are doing is logically consistent. Accuracy is more important than run speed.

Adding the command EXPLAIN at the beginning of any query allows you to get a sense of how long it will take your query to run. This will output a Query Plan which outlines the execution order of the query. The query plan will attach a cost to the query and the higher the cost, the longer the runtime. EXPLAIN is most useful to identify and modify those steps that are expensive. Do this then run EXPLAIN again to see if the speed/cost has improved.

Query1

```
EXPLAIN
SELECT *
FROM   web_events
WHERE  occurred_at >='2016-01-01'
AND    occurred_at < '2016-02-01'
```

Query2

```
EXPLAIN
SELECT *
FROM   web_events
WHERE  occurred_at >='2016-01-01'
AND    occurred_at < '2016-02-01'
LIMIT 100
```

JOINing Subqueries

JOINing Subqueries to Improve Performance

Code along with Derek in the SQL Explorer below this next video. In the end, run each of the subqueries independently to get a better understanding of how they work.

Expert Tip

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.

Additional Practice Resources

If you would like to get more practice writing SQL queries, there are several great websites to practice writing SQL queries. Here are a couple we recommend: HackerRank and ModeAnalytics. We strongly recommend these. The skill test by AnalyticsVidhya is a fun test to take too.