So above, we understand that all of the information related to an account is not in the **orders** table, but why not? Watch the below video to find out!

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.

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.

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 orders
JOIN accounts
ON 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 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. Try running this query yourself below.

Menu

Expand

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 in 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.

Now it's your turn.

Quiz Questions

1. Try pulling all the data from the **accounts** table, and all the data from the **orders** table.

2. Try pulling **standard_qty**, **gloss_qty**, and **poster_qty** from the **orders** table, and the **website** and the **primary_poc** from the **accounts** table.

Another environment is below to practice these two questions, and you can check your solutions on the next concept.

Solutions

```
SELECT orders.*, accounts.*

FROM accounts

JOIN orders

ON accounts.id = orders.account_id;
```

1. Notice this result is the same as if you switched the tables in the **FROM** and **JOIN**. Additionally, which side of the `=` a column is listed doesn't matter.

```
SELECT orders.standard_qty, orders.gloss_qty,

       orders.poster_qty,  accounts.website,

       accounts.primary_poc

FROM orders

JOIN accounts

ON orders.account_id = accounts.id
```

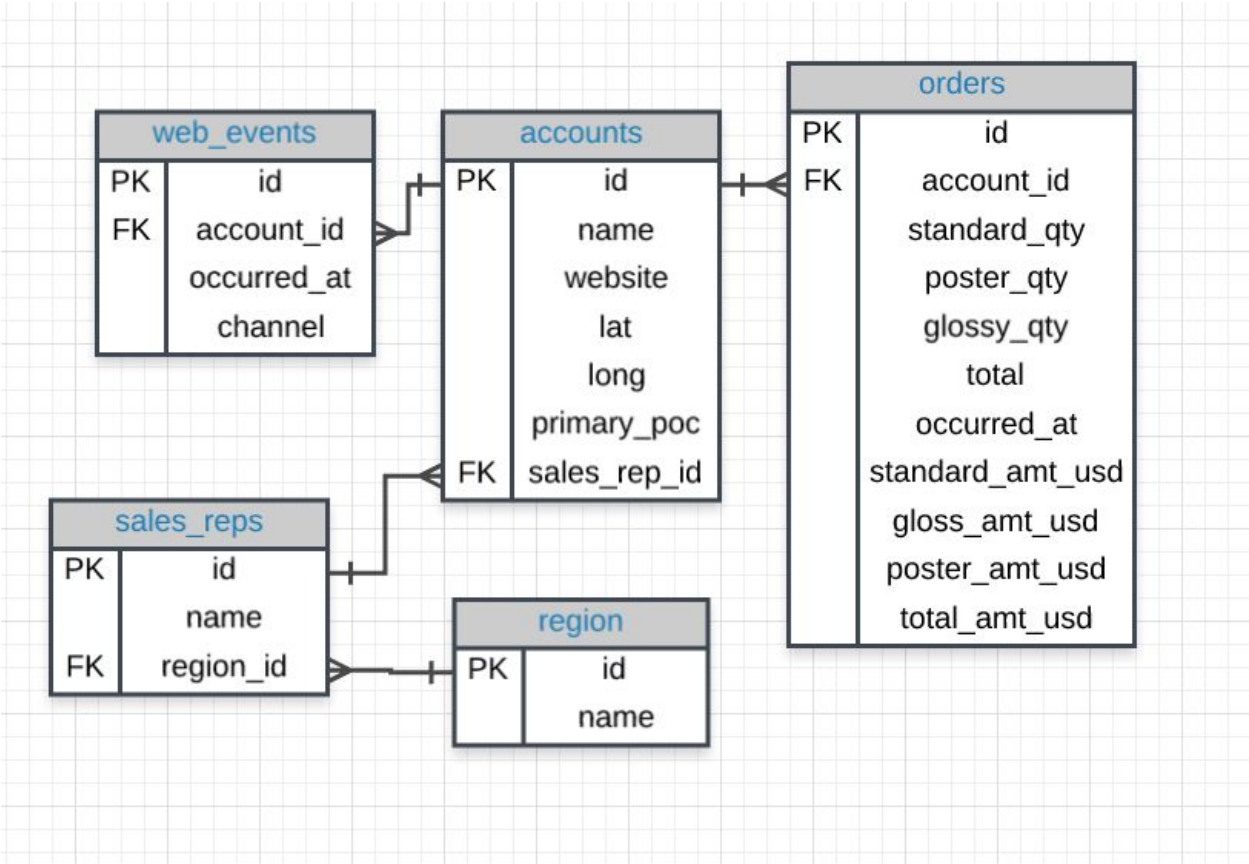
2.

    Notice that we need to specify every table a column comes from in the **SELECT** statement.

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, so I have posted it again below. **You might even print a copy to have with you as you work through the exercises in the remaining content.**



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 **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.

You can explore the database ERD below by clicking on the table names in the left menu of the workspace below. Or you can query the first row of these tables in the workspace to see some examples of the content inside of each field.

Keys

Primary Key (PK)

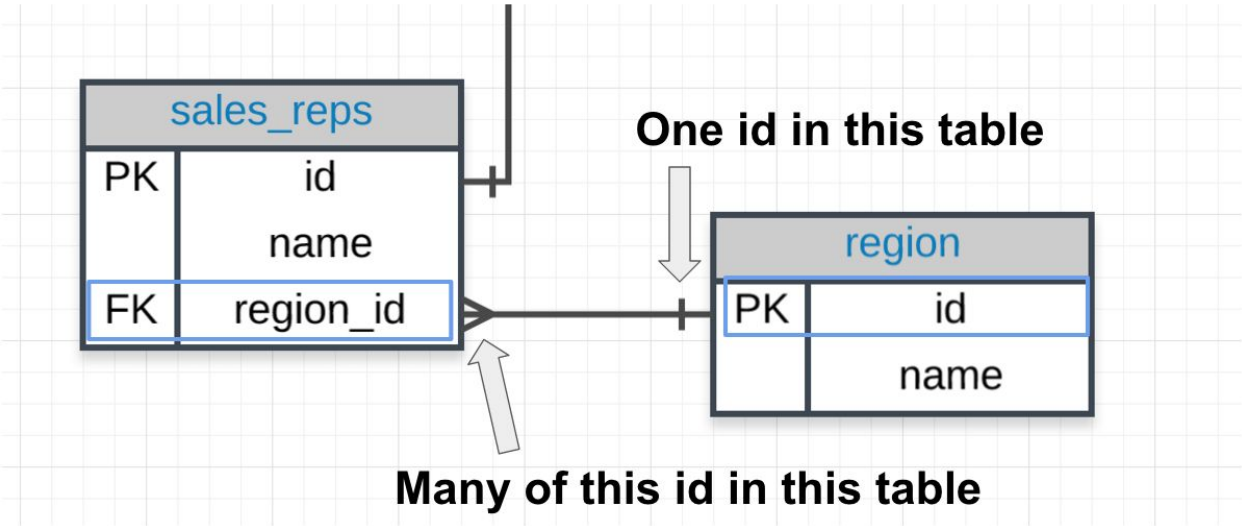
A **primary key** is a unique column in a particular table. This is the first column in each of our tables. Here, those columns are all called **id**, but that doesn't necessarily have to be the name. **It is common that the primary key is the first column in our tables in most databases.**

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:



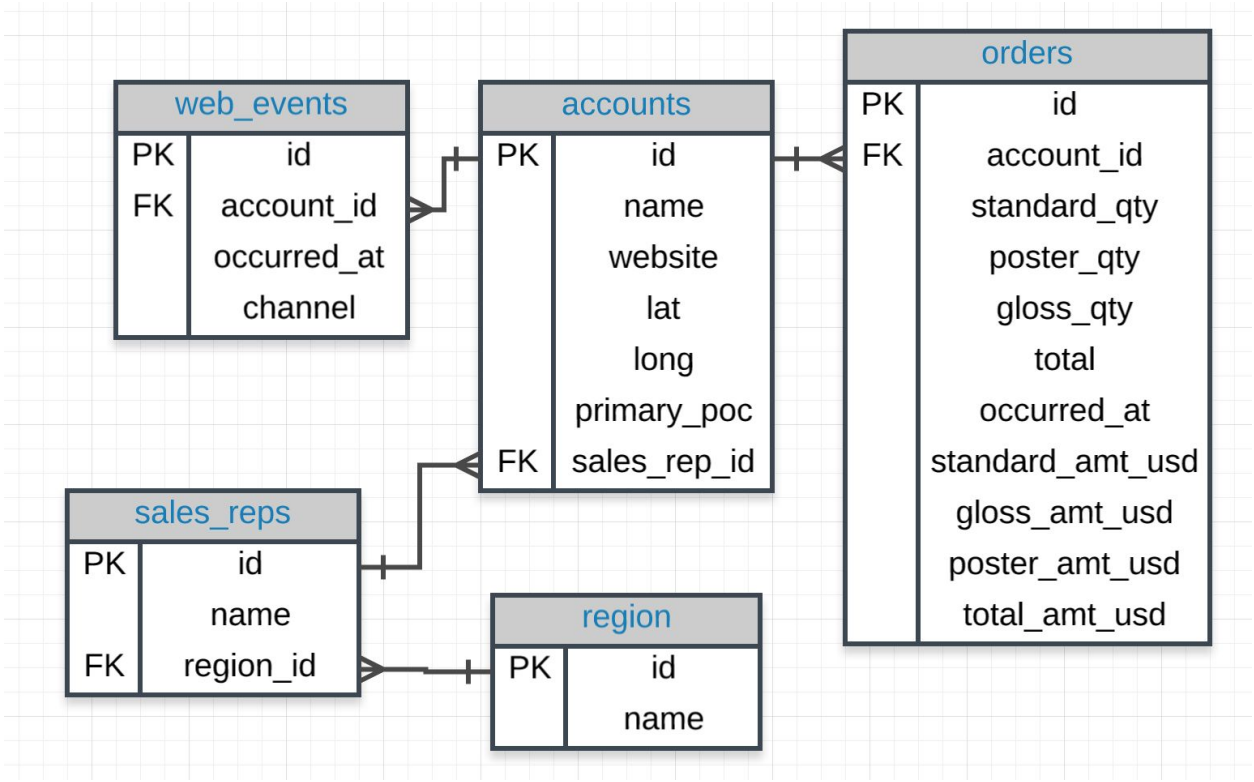
Primary - Foreign Key Link

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!

Helpful ERD For Answering the Below Questions



Match the appropriate definition or description to each term or column.

ON accounts.id = web_events.id

ON accounts.account_id = web_events.id

ON web_events.id = accounts.id

DEFINITION OR COLUMN DESCRIPTION

TERM OR COLUMN

Has a unique value for every row in that column. There is one in every table.

Primary Key

The **link** to the primary key that exists in another table.

Foreign Key

The primary key in every table of our example database.

id

A foreign key that exists in both the **web_events** and **orders** tables.

account_id

The **ON** statement associated with a **JOIN** of the **web_events** and **accounts** tables.

ON web_events.account_id = accounts.id

SUBMIT

QUESTION 2 OF 3

Select all that are true for primary keys.

- There is one and only one of these columns in every table.

- They are a column in a table.

- There might be more than one primary key for a table.
- They are a row in a table.
- Every database only has one primary key for the whole database.

SUBMIT

QUESTION 3 OF 3

Select all that are true of foreign keys.

- They are always linked to a primary key.

- They are unique for every row in a table.
- Every table must have a foreign key.
- A table can only have one foreign key.

- In the above database, every foreign key is associated with the crow-foot notation, which suggests it can appear multiple times in the column of a table.

JOIN Revisited

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

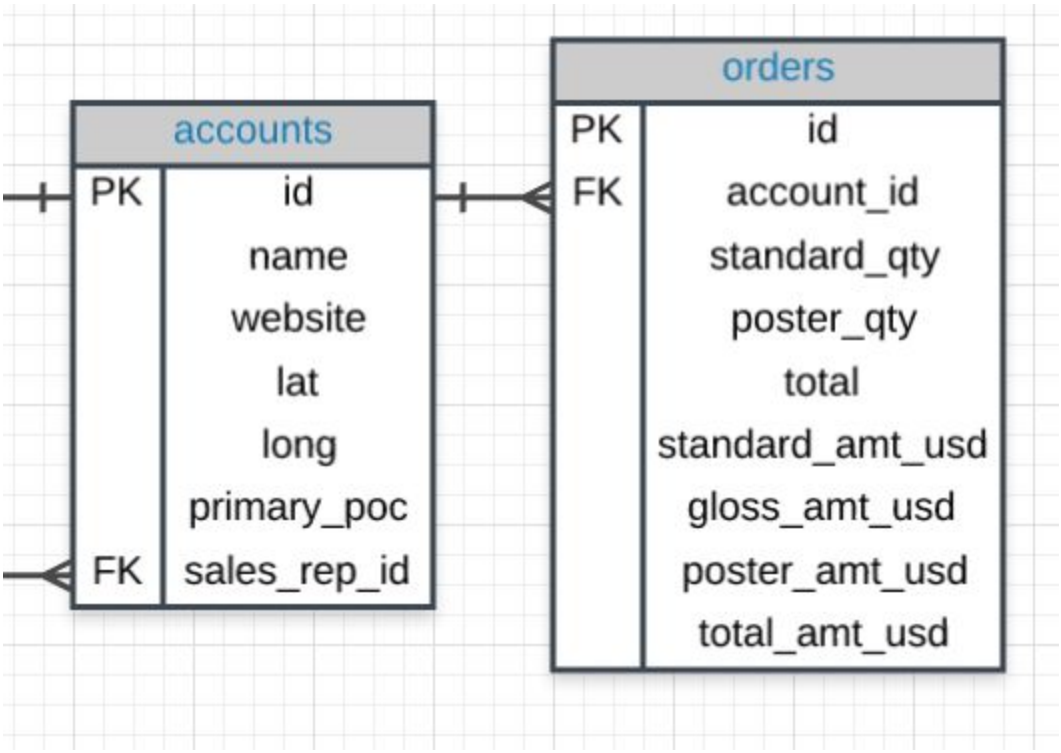
```
SELECT orders.*

FROM orders
```

```
JOIN accounts

ON orders.account_id = accounts.id;
```

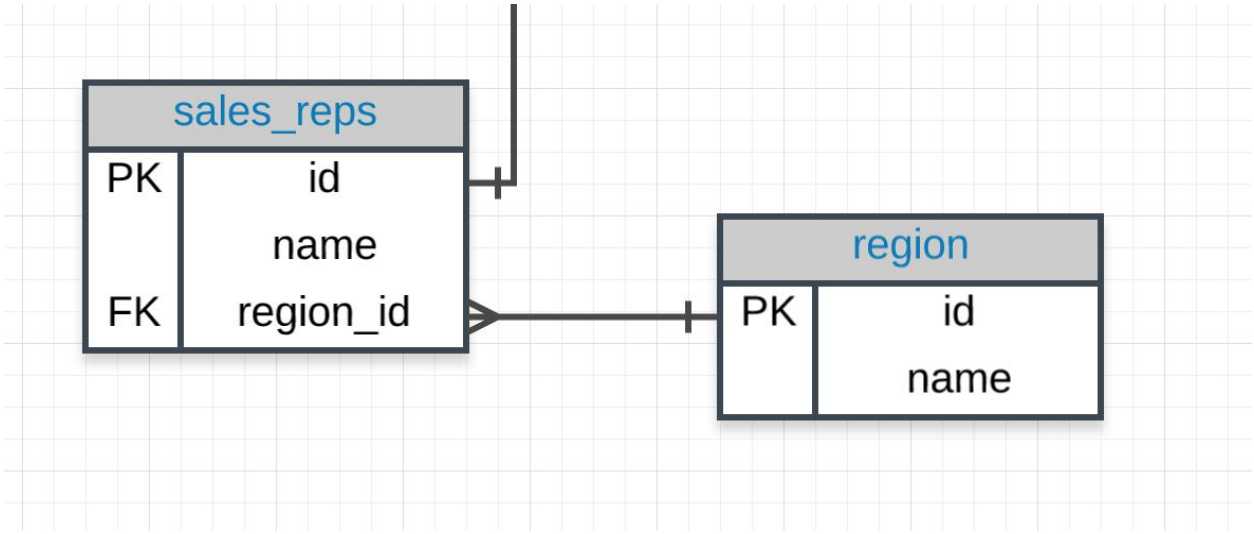
Here is the ERD for these two tables:



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).



QUIZ QUESTION

Practice

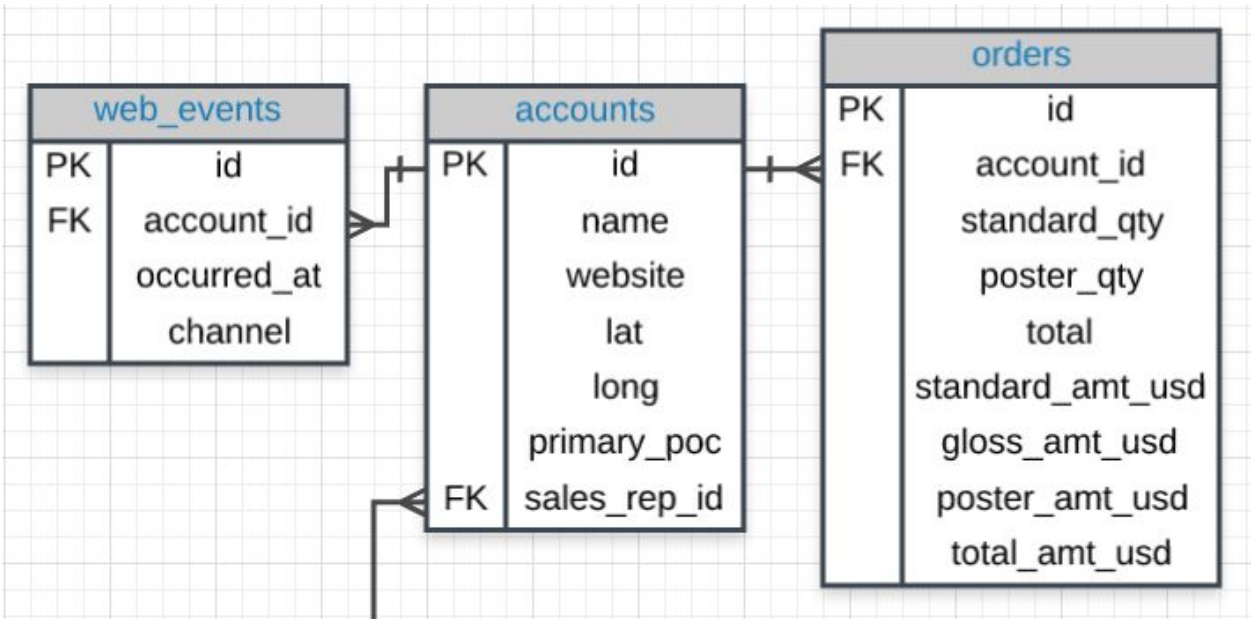
Use the image above to assist you. If we wanted to join the `sales_reps` and `region` tables together, how would you do it

- ON sales_reps.id = region.id
- ON sales_reps.id = region.name
- ON sales_reps.region_id = region.id

- ON region.id = sales_reps.id

JOIN More than Two Tables

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



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_events
JOIN accounts
```

```
ON web_events.account_id = accounts.id
JOIN orders
ON 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.

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 tablename AS t1
JOIN tablename2 AS t2
```

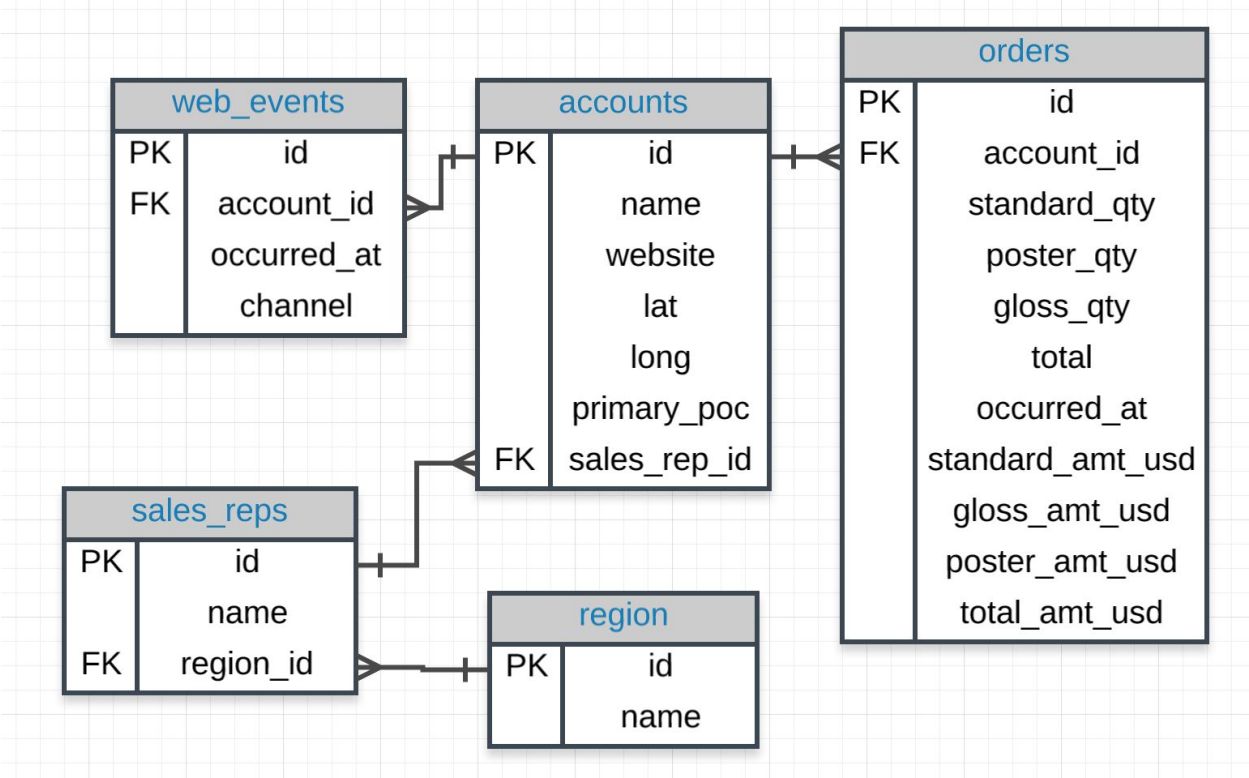
The alias name fields will be what shows up in the returned table instead of t1.column1 and t2.column2

aliasname	aliasname2
example row	example row

example row	example row

Question Mania

Now that you have been introduced to JOINs, let's practice to build your skills and comfort with this new tool. Below I have provided the **ERD** and a bunch of questions. The solutions for the questions can be found on the next concept for you to check your answers or just in case you get stuck!



I recommend testing your queries with the environment below, and then saving them to a file. Then compare your file to my solutions on the next concept!

Questions

1. Provide a table for all **web_events** associated with account **name** of `Walmart`. There should be three columns. Be sure to include the `primary_poc`, time of the event, and the `channel` for each event. Additionally, you might choose to add a fourth column to assure only `Walmart` events were chosen.

2. Provide a table that provides the **region** for each **sales_rep** along with their associated **accounts**. 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 account name.

3. 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 table should have 3 columns: **region name**, **account name**, and **unit price**. A few accounts have 0 for **total**, so I divided by (total + 0.01) to assure not dividing by zero.

Notice

There are two videos in this concept:

1. The first shows JOINs the way you have currently been working with data.

2. The second shows **LEFT** and **RIGHT** JOIN statements.

Above you learned about the JOINs you have been writing so far - that is an **INNER JOIN**. In the video below, you will learn about other ways that we might want to **JOIN** our data depending on the question we are asking.

JOINs

Notice each of these new **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**.

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.

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** we learned about in the previous video.

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**.

Above are two small tables for you to test your knowledge of **JOIN**s. You can click on the image to get a better view.

Country has 6 rows and 2 columns:

- **countryid** and **countryName**

State has 6 rows and 3 columns:

- **stateid**, **countryid**, and **stateName**

Use the above tables to determine the solution to the following questions.

QUESTION 2 OF 4

Match each statement to the item it describes.

Country.countryName

State.countryid

ITEM	DESCRIPTION
State.stateName	
State.stateid	The primary key of the **State** table.
Country.countryid	The primary key of the **Country** table.
	The foreign key that would be used in **JOIN**ing the tables.

LEFT and RIGHT JOIN Solutions

This section is a walkthrough of those final two problems in the previous concept. First, another look at the two tables we are working with:

Country

countryid	countryName
1	India
2	Nepal
3	United States
4	Canada
5	Sri Lanka
6	Brazil

State

stateid	countryid	stateName
1	1	Maharashtra
2	1	Punjab
3	2	Kathmandu
4	3	California
5	3	Texas
6	4	Alberta

INNER JOIN Question

The questions are aimed to assure you have a conceptual idea of what is happening with **LEFT** and **INNER JOIN**s before you need to use them for more difficult problems.

For an **INNER JOIN** like the one here:

```
SELECT c.countryid, c.countryName, s.stateName
FROM Country c
JOIN State s
ON c.countryid = s.countryid;
```

We are essentially **JOIN**ing the matching **PK**-**FK** links from the two tables, as shown in the below image.

INNER JOIN

Country

countryid	countryName
1	India
2	Nepal
3	United States
4	Canada
5	Sri Lanka
6	Brazil

State

stateid	countryid	stateName
1	1	Maharashtra
2	1	Punjab
3	2	Kathmandu
4	3	California
5	3	Texas
6	4	Alberta

The resulting table will look like:

countryid	countryName	stateName
1	India	Maharashtra
1	India	Punjab
2	Nepal	Kathmandu
3	United States	California

	3	United States	Texas
	4	Canada	Alberta

LEFT JOIN Question

The questions are aimed to assure you have a conceptual idea of what is happening with **LEFT** and **INNER JOIN**s before you need to use them for more difficult problems.

For a **LEFT JOIN** like the one here:

```
SELECT c.countryid, c.countryName, s.stateName
FROM Country c
LEFT JOIN State s
ON c.countryid = s.countryid;
```

We are essentially **JOIN**ing the matching **PK**-**FK** links from the two tables, as we did before, but we are also pulling all the additional rows from the **Country** table even if they don't have a match in the **State** table. Therefore, we obtain all the rows of the **INNER JOIN**, but we also get additional rows from the table in the **FROM**.

The resulting table will look like:

countryid	countryName	stateName
1	India	Maharashtra
1	India	Punjab
2	Nepal	Kathmandu
3	United States	California

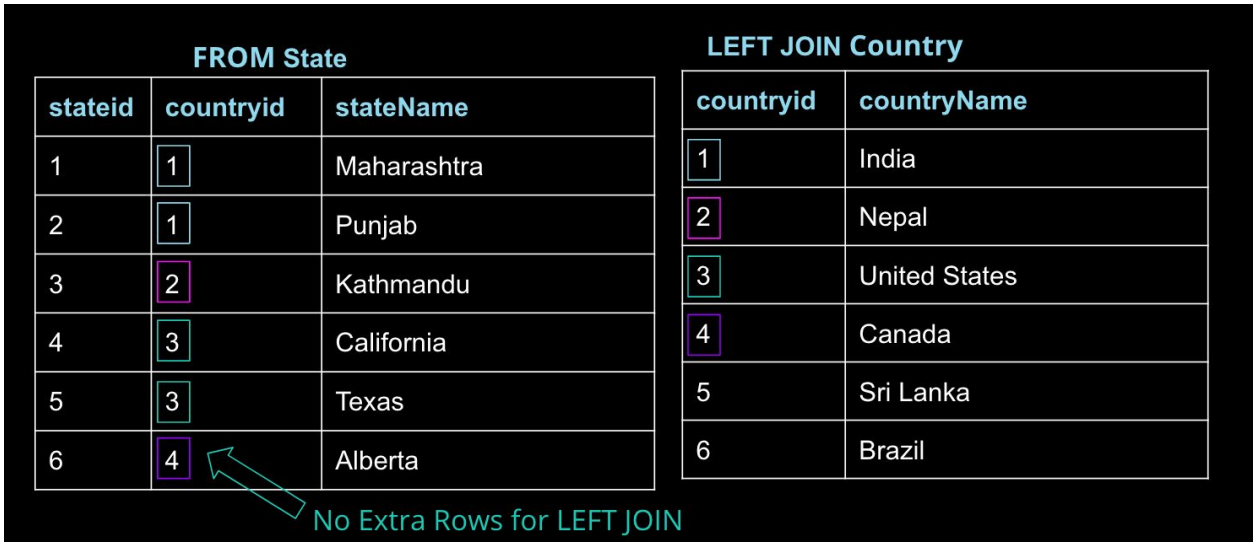
	3	United States	Texas
	4	Canada	Alberta
	5	Sri Lanka	NULL
	6	Brazil	NULL

FINAL LEFT JOIN Note

If we were to flip the tables, we would actually obtain the same exact result as the **JOIN** statement:

```
SELECT c.countryid, c.countryName, s.stateName
FROM State s
LEFT JOIN Country c
ON c.countryid = s.countryid;
```

This is because if **State** is on the **LEFT** table, all of the rows exist in the **RIGHT** table again.



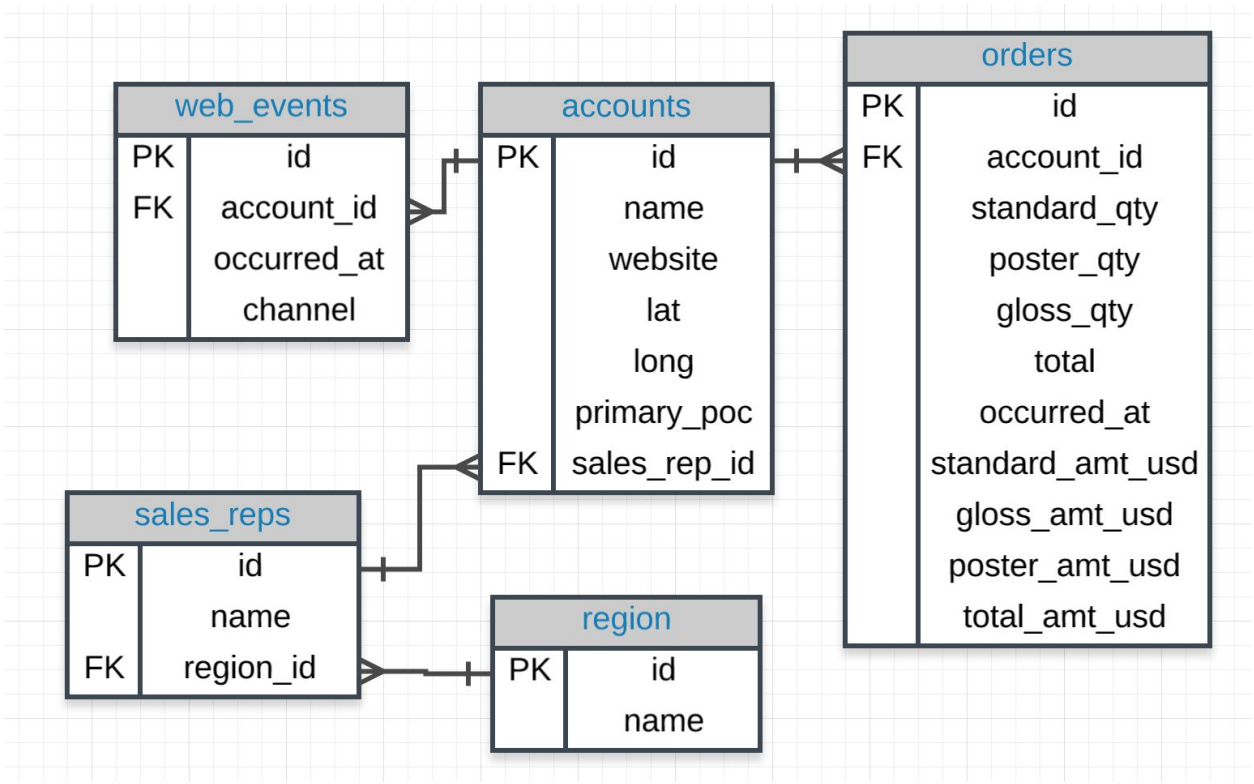
The resulting table will look like:

countryid	countryName	stateName
1	India	Maharashtra
1	India	Punjab
2	Nepal	Kathmandu
3	United States	California

	3	United States	Texas
	4	Canada	Alberta

A simple rule to remember this 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.

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.



I recommend testing your queries with the environment below, and then saving them to a file. Then compare your file to my solutions on the next concept!

`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`

Questions

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 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 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 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**. In order to avoid a division by zero error, adding .01 to the denominator here is helpful total_amt_usd/(total+0.01).

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. In order to avoid a division by zero error, adding .01 to the denominator here is helpful (total_amt_usd/(total+0.01).

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. In order to avoid a division by zero error, adding .01 to the denominator here is helpful (total_amt_usd/(total+0.01).

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.

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**.

Solutions

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 account name.

```
SELECT r.name region, s.name rep, a.name account
FROM sales_reps s
JOIN region r
ON s.region_id = r.id
JOIN accounts a
ON a.sales_rep_id = s.id
WHERE r.name = 'Midwest'
ORDER BY a.name;
```

1.

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 account name.

```
SELECT r.name region, s.name rep, a.name account
FROM sales_reps s
JOIN region r
ON s.region_id = r.id
JOIN accounts a
ON a.sales_rep_id = s.id
WHERE r.name = 'Midwest' AND s.name LIKE 'S%'
ORDER BY a.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 **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 account name.

```
SELECT r.name region, s.name rep, a.name account
FROM sales_reps s
JOIN region r
ON s.region_id = r.id
JOIN accounts a
ON a.sales_rep_id = s.id
WHERE r.name = 'Midwest' AND s.name LIKE '% K%'
ORDER BY a.name;
```

3.

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;
```

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` 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.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 AND o.poster_qty > 50
ORDER BY unit_price;
```

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 largest **unit price** first.

```
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 AND o.poster_qty > 50
ORDER BY unit_price DESC;
```

6.

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.

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

7.

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.id
```

```
WHERE o.occurred_at BETWEEN '01-01-2015' AND '01-01-2016'
```

Recap

Primary and Foreign Keys

You learned a key element for JOINing tables in a database has to do with primary and foreign keys:

- **primary keys - are unique for every row in a table. These are generally the first column in our database (like you saw with the id column for every table in the Parch & Posey database).**
- **foreign keys - are the primary key appearing in another table, which allows the rows to be non-unique.**

Choosing the set up of data in our database is very important, but not usually the job of a data analyst. This process is known as Database Normalization.

JOINs

In this lesson, you learned how to combine data from multiple tables using JOINs. The three JOIN statements you are most likely to use are:

1. **JOIN - an INNER JOIN that only pulls data that exists in both tables.**
2. **LEFT JOIN - pulls all the data that exists in both tables, as well as all of the rows from the table in the FROM even if they do not exist in the JOIN statement.**

3. **RIGHT JOIN - pulls all the data that exists in both tables, as well as all of the rows from the table in the JOIN even if they do not exist in the FROM statement.**

There are a few more advanced JOINs that we did not cover here, and they are used in very specific use cases. UNION and UNION ALL, CROSS JOIN, and the tricky SELF JOIN. These are more advanced than this course will cover, but it is useful to be aware that they exist, as they are useful in special cases.

Alias

You learned that you can alias tables and columns using AS or not using it. This allows you to be more efficient in the number of characters you need to write, while at the same time you can assure that your column headings are informative of the data in your table.

Looking Ahead

The next lesson is aimed at aggregating data. You have already learned a ton, but SQL might still feel a bit disconnected from statistics and using Excel like platforms. Aggregations will allow you to write SQL code that will allow for more complex queries, which assist in answering questions like:

- **Which channel generated more revenue?**
- **Which account had an order with the most items?**
- **Which sales_rep had the most orders? or least orders? How many orders did they have?**

8. `ORDER BY o.occurred_at DESC;`