https://classroom.udacity.com/courses/ud002-bert/lessons/8f23fc69-7c88-4a94-97a4-d5f6ef51cf7b/concepts/824ea43f-a26d-49d9-ad94-bc10ed3d758b

LESSON 29 : SQL JOINS (1-21)

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?

JOINS: Tells query an additional table from which you would like to pull data

ON : Specifies a logical statement to combine the table in FROM and JOIN statements.

```
SELECT orders.* , accounts.*  FROM demo.orders JOIN demo.accounts
ON orders.accounts_id = accounts.id
```

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;
```

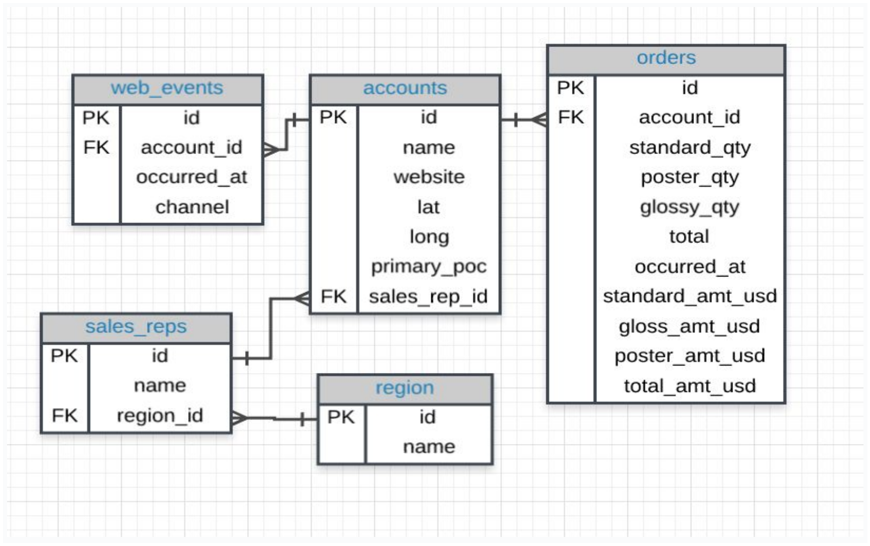
And the first query you ran pull all the information from only the orders table:

```
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.

Try pulling standard_qty, gloss_qty, and poster_qty from the orders table, and the website and the primary_poc from the accounts table?

```
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
```



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.

Keys

Primary Key (PK)

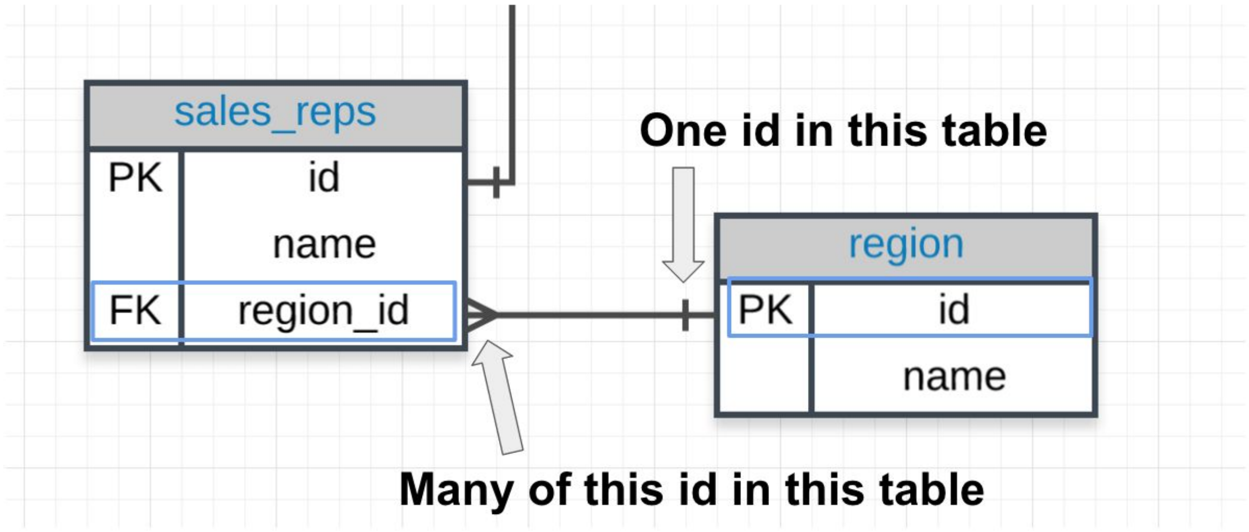
A primary key is a unique column in a particular table. This is the first column in each of our tables. Here, those columns are all called id, but that doesn't necessarily have to be the name. It is common that the primary key is the first column in our tables in most databases.

Foreign Key (FK)

A foreign key is when we see a primary key in another table. We can see these in the previous ERD the foreign keys are provided as:

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.**

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
```

Solutions

1.Provide a table for all the 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.

```
SELECT a.primary_poc, w.occurred_at, w.channel, a.name
FROM web_events w
JOIN accounts a
ON w.account_id = a.id
```

```
WHERE a.name = 'Walmart';
```

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.**

```
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
ORDER BY a.name;
```

3.**Provide the name for each region for every order, as well as the account name and the unit pricethey 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.**

```
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;
```

Traditional databases does not allow for many-to-many relationship

https://stackoverflow.com/questions/7339143/why-no-many-to-many-relationships

INNER JOIN : Only returns rows that appears in both tables

If we want to include data that doesn’t exist in both tables but only in one

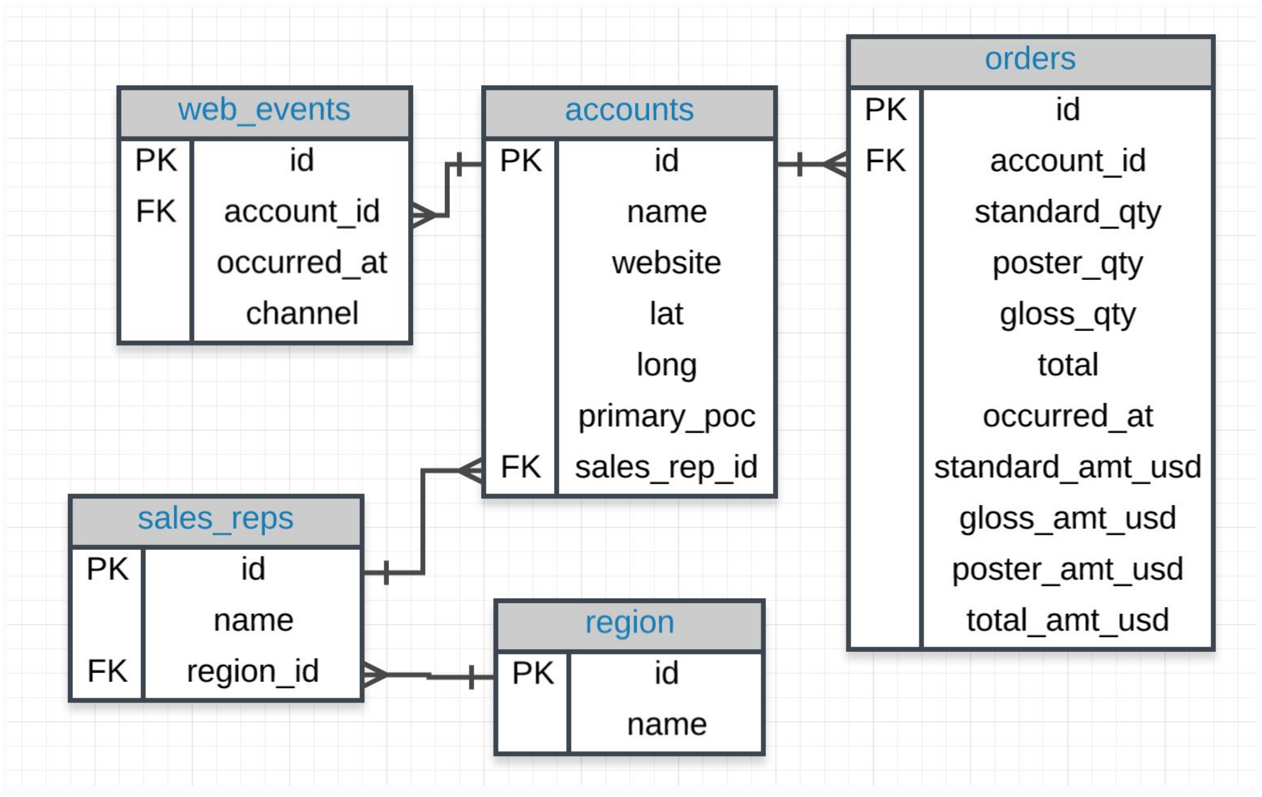
```
of the tables we are using our join statement, then three types of join we may
use they are :LEFT JOIN, RIGHT JOIN , FULL OUTER JOIN
LEFT JOIN:
SELECT
FROM left(lefttablename)
LEFT JOIN right(righttablename)
In this the we get results that match the right table and also the results
remaining in the left table eventhough they didnt match the right table.
```

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.



Solutions

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.

```
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;
```

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.

```
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;
```

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.

```
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;
```

4.Provide the name for each region for every order, as well as the account name and the unit pricethey 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 pricethey 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;
```

6.Provide the name for each region for every order, as well as the account name and the unit pricethey 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 pricefirst.

```
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;
```

7.What are the different channels used by account id `1001`? Your final table should have only 2 columns: account name and the different channels. 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';
```

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.id
WHERE o.occurred_at BETWEEN '01-01-2015' AND '01-01-2016'
ORDER BY o.occurred_at DESC;
```

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 JOINstatements you are most likely to use are:

1. JOIN - an INNER JOIN that only pulls data that exists in both tables.
2. LEFT JOIN - a way to pull all of the rows from the table in the FROM even if they do not exist in the JOIN statement.
3. RIGHT JOIN - a way to pull 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?**