MULTIPLE TABLES

Introduction

In order to efficiently store data, we often spread related information across multiple tables. For instance, imagine that we're running a magazine company where users can have different types of subscriptions to different products. Different subscriptions might have many different properties. Each customer would also have lots of associated information.

We could have one table with all of the following information:

- `order_id`
- `customer_id`
- `customer_name`
- `customer_address`
- `subscription_id`
- `subscription_description`
- `subscription_monthly_price`
- `subscription_length`
- `purchase_date`
-

However, a lot of this information would be repeated. If the same customer has multiple subscriptions, that customer's name and address will be reported multiple times. If the same subscription type is ordered by multiple customers, then the subscription price and subscription description will be repeated. This will make our table big and unmanageable. So instead, we can split our data into three tables:

1. `orders` would contain just the information necessary to describe what was ordered:
   - `order_id`, `customer_id`, `subscription_id`, `purchase_date`
2. `subscriptions` would contain the information to describe each type of subscription:
   - `subscription_id`, `description`, `price_per_month`, `subscription_length`
3. `customers` would contain the information for each customer:
   - `customer_id`, `customer_name`, `address`

In this lesson, we'll learn the SQL commands that will help us work with data that is stored in multiple tables.

1. Examine these tables by pasting the following code into the editor:

```
SELECT *
FROM orders
LIMIT 5;

SELECT *
FROM subscriptions
LIMIT 5;
```

```
SELECT *
FROM customers
LIMIT 5;
```

Combining Tables Manually

Let's return to our magazine company. Suppose we have the three tables described in the previous exercise – shown in the browser on the right (we are going to try something new!):

- `orders`
- `subscriptions`
- `customers`

If we just look at the `orders` table, we can't really tell what's happened in each order. However, if we refer to the other tables, we can get a complete picture.

Let's examine the order with an `order_id` of 2. It was purchased by the customer with a `customer_id` of 2.

To find out the customer's name, we look at the `customers` table and look for the item with a `customer_id` value of 2. We can see that Customer 2's name is 'Jane Doe' and that she lives at '456 Park Ave'. Doing this kind of matching is called **joining** two tables.

1. Using the tables displayed, what is the `description` of the magazine ordered in `order_id` 1? Type your answer on line 1 of the code editor. Be sure to capitalize it the same as in the table.

2. Using the tables displayed, what is the `customer_name` of the customer in `order_id` 3? Type your answer on line 2 of the code editor. Be sure to capitalize it the same as in the table.

```
Sports Magazine
Joe Schmo
```

Combining Tables with SQL

Combining tables manually is time-consuming. Luckily, SQL gives us an easy sequence for this: it's called a `JOIN`. If we want to combine `orders` and `customers`, we would type:

```
SELECT *
FROM orders
JOIN customers
  ON orders.customer_id = customers.customer_id;
```

Let's break down this command:

1. The first line selects all columns from our combined table. If we only want to select certain columns, we can specify which ones we want.
2. The second line specifies the first table that we want to look in, `orders`

3. The third line uses `JOIN` to say that we want to combine information from `orders` with `customers`.
4. The fourth line tells us how to combine the two tables. We want to match `orders` table's `customer_id` column with `customers` table's `customer_id` column.

Because column names are often repeated across multiple tables, we use the syntax `table_name.column_name` to be sure that our requests for columns are unambiguous. In our example, we use this syntax in the `ON` statement, but we will also use it in the `SELECT` or any other statement where we refer to column names.

For example: Instead of selecting *all* the columns using `*`, if we only wanted to select `orders` table's `order_id` column and `customers` table's `customer_name` column, we could use the following query:

```
SELECT orders.order_id,
   customers.customer_name
FROM orders
JOIN customers
  ON orders.customer_id = customers.customer_id;
```

1. Join `orders` table and `subscriptions` table and select all columns. Make sure to join on the `subscription_id` column.

```
SELECT *
FROM orders
JOIN subscriptions
ON orders.subscription_id = subscriptions.subscription_id;
```

3. Don't remove the previous query. Add a second query after your first one that only selects rows from the join where `description` is equal to 'Fashion Magazine'.

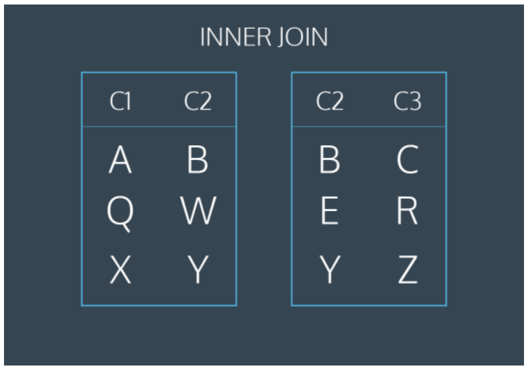
```
SELECT *
FROM orders
JOIN subscriptions
ON orders.subscription_id = subscriptions.subscription_id
WHERE description = 'Fashion Magazine';
```

Inner Joins

Let's revisit how we joined `orders` and `customers`. For every possible value of `customer_id` in `orders`, there was a corresponding row of `customers` with the same `customer_id`. What if that wasn't true?

For instance, imagine that our `customers` table was out of date, and was missing any information on customer 11. If that customer had an order in `orders`, what would happen when we joined the tables?

When we perform a simple `JOIN` (often called an *inner join*) our result only includes rows that match our `ON` condition. Consider the following animation, which illustrates an inner join of two tables on `table1.c2 = table2.c2`:



The first and last rows have matching values of `c2`. The middle rows do not match. The final result has all values from the first and last rows but does not include the non-matching middle row.

1. Suppose we are working for The Codecademy Times, a newspaper with two types of subscriptions:

- print newspaper
- online articles

Some users subscribe to just the newspaper, some subscribe to just the online edition, and some subscribe to both.There is a `newspaper` table that contains information about the newspaper subscribers. Count the number of subscribers who get a print newspaper using `COUNT()`.

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

2. Don't remove your previous query. There is also an `online` table that contains information about the online subscribers. Count the number of subscribers who get an online newspaper using `COUNT()`.

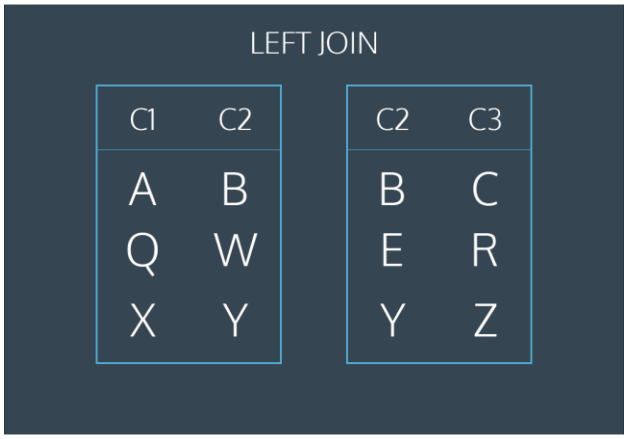
```
SELECT COUNT(*)
FROM online;
```

3. Don't remove your previous queries. Join `newspaper` table and `online` table on their `id` columns (the unique ID of the subscriber). How many rows are in this table?

```
SELECT COUNT(*)
FROM newspaper
JOIN online
  ON newspaper.id = online.id;
```

Left Joins

What if we want to combine two tables and keep some of the un-matched rows? SQL lets us do this through a command called `LEFT JOIN`. A *left join* will keep all rows from the first table, regardless of whether there is a matching row in the second table. Consider the following animation:



The first and last rows have matching values of `c2`. The middle rows do not match. The final result will keep all rows of the first table but will omit the un-matched row from the second table. This animation represents a table operation produced by the following command:

```
SELECT *
FROM table1
LEFT JOIN table2
  ON table1.c2 = table2.c2;
```

1. The first line selects all columns from both tables.
2. The second line selects `table1` (the "left" table).
3. The third line performs a `LEFT JOIN` on `table2` (the "right" table).
4. The fourth line tells SQL how to perform the join (by looking for matching values in column `c2`).

1. Let's return to our `newspaper` and `online` subscribers. Suppose we want to know how many users subscribe to the print newspaper, but not to the online. Start by performing a left join of `newspaper` table and `online` table on their `id` columns and selecting all columns.

```
SELECT *
FROM newspaper
LEFT JOIN online
  ON newspaper.id = online.id;
```

2. Don't remove your previous query. In order to find which users do *not* subscribe to the online edition, we need to add a `WHERE` clause. Add a second query after your first one that adds the following `WHERE` clause and condition:

```
WHERE online.id IS NULL
```

This will select rows where there was no corresponding row from the `online` table.

```
SELECT *
FROM newspaper
LEFT JOIN online
ON newspaper.id = online.id
WHERE online.id IS NULL;
```

Primary Key vs Foreign Key

Let's return to our example of the magazine subscriptions. Recall that we had three tables: `orders`, `subscriptions`, and `customers`. Each of these tables has a column that uniquely identifies each row of that table:

- `order_id` for `orders`
- `subscription_id` for `subscriptions`
- `customer_id` for `customers`

These special columns are called **primary keys**. Primary keys have a few requirements:

- None of the values can be `NULL`.
- Each value must be unique (i.e., you can't have two customers with the same `customer_id` in the `customers` table).
- A table can not have more than one primary key column.

Let's reexamine the `orders` table:

order_id	customer_id	subscription_id	purchase_date
1	2	3	2017-01-01
2	2	2	2017-01-01
3	3	1	2017-01-01

Note that `customer_id` (the primary key for `customers`) and `subscription_id` (the primary key for `subscriptions`) both appear in this. When the primary key for one table appears in a different table, it is called a **foreign key**. So `customer_id` is a primary key when it appears in `customers`, but a foreign key when it appears in `orders`.

In this example, our primary keys all had somewhat descriptive names. Generally, the primary key will just be called `id`. Foreign keys will have more descriptive names. *Why is this important?* The most common types of joins will be joining a foreign key from one table with the primary key from another table. For instance, when we join `orders` and `customers`, we join on `customer_id`, which is a foreign key in `orders` and the primary key in `customers`.

1. Suppose Columbia University has two tables in their database:

- The `classes` table contains information on the classes that the school offers. Its primary key is `id`.

- The students table contains information on all students in the school. Its primary key is id. It contains the foreign key class_id, which corresponds to the primary key of classes.

Perform an inner join of classes and students using the primary and foreign keys described above, and select all the columns.

```
SELECT *
FROM classes
JOIN students
  ON classes.id = students.class_id;
```

Cross Join

So far, we've focused on matching rows that have some information in common. Sometimes, we just want to combine all rows of one table with all rows of another table. For instance, if we had a table of shirts and a table of pants, we might want to know all the possible combinations to create different outfits. Our code might look like this:

```
SELECT shirts.shirt_color,
   pants.pants_color
FROM shirts
CROSS JOIN pants;
```

- The first two lines select the columns shirt_color and pants_color.
- The third line pulls data from the table shirts.
- The fourth line performs a CROSS JOIN with pants.

Notice that cross joins don't require an ON statement. You're not really joining on any columns! If we have 3 different shirts (white, grey, and olive) and 2 different pants (light denim and black), the results might look like this:

shirt_color	pants_color
white	light denim
white	black
grey	light denim
grey	black
olive	light denim
olive	black

3 shirts × 2 pants = 6 combinations! This clothing example is fun, but it's not very practically useful.
A more common usage of CROSS JOIN is when we need to compare each row of a table to a list of values. Let's return to our newspaper subscriptions. This table contains two columns that we haven't discussed yet:

- start_month: the first month where the customer subscribed to the print newspaper (i.e., 2 for February)

- `end_month`: the final month where the customer subscribed to the print newspaper

Suppose we wanted to know how many users were subscribed during each month of the year. For each month (`1`, `2`, `3`) we would need to know if a user was subscribed. Follow the steps below to see how we can use a `CROSS JOIN` to solve this problem.

1. Eventually, we'll use a cross join to help us, but first, let's try a simpler problem. Let's start by counting the number of customers who were subscribed to the `newspaper` during March.

Use `COUNT(*)` to count the number of rows and a `WHERE` clause to restrict to two conditions:

- `start_month <= 3`
- `end_month >= 3`

```
SELECT COUNT(*)
FROM newspaper
WHERE start_month <= 3
  AND end_month >= 3;
```

2. Don't remove the previous query. The previous query lets us investigate one month at a time. In order to check across all months, we're going to need to use a cross join.

Our database contains another table called `months` which contains the numbers between 1 and 12. Select all columns from the cross join of `newspaper` and `months`.

```
  SELECT *
FROM newspaper
CROSS JOIN months;
```

3. Don't remove your previous queries. Create a third query where you add a `WHERE` statement to your cross join to restrict to two conditions:

- `start_month <= month`
- `end_month >= month`

This will select all months where a user was subscribed.

```
SELECT *
FROM newspaper
CROSS JOIN months
WHERE start_month <= month
  AND end_month >= month;
```

4. Don't remove your previous queries. Create a final query where you aggregate over each month to count the number of subscribers. Fill in the blanks in the following query:

```
SELECT month,
  COUNT(*)
FROM ________
CROSS JOIN ________
WHERE ________ AND ________
GROUP BY ________;
```

```
SELECT month,
   COUNT(*) AS 'subscribers'
FROM newspaper
CROSS JOIN months
WHERE start_month <= month
   AND end_month >= month
GROUP BY month;
```

Union

Sometimes we just want to stack one dataset on top of the other. Well, the `UNION` operator allows us to do that.Suppose we have two tables and they have the same columns.

`table1`:

pokemon	type
Bulbasaur	Grass
Charmander	Fire
Squirtle	Water

`table2`:

pokemon	type
Snorlax	Normal

If we combine these two with `UNION`:

```
SELECT *
FROM table1
UNION
SELECT *
FROM table2;
```

The result would be:

pokemon	type
Bulbasaur	Grass
Charmander	Fire
Squirtle	Water
Snorlax	Normal

SQL has strict rules for appending data:

- Tables must have the same number of columns.
- The columns must have the same data types in the same order as the first table.

1. Let's return to our `newspaper` and `online` subscriptions. We'd like to create one big table with both sets of data. Use `UNION` to stack the `newspaper` table on top of the `online` table.

```
SELECT *
FROM newspaper
UNION
SELECT *
FROM online;
```

With

Often times, we want to combine two tables, but one of the tables is the result of another calculation. Let's return to our magazine order example. Our marketing department might want to know a bit more about our customers. For instance, they might want to know how many magazines each customer subscribes to. We can easily calculate this using our `orders` table:

```
SELECT customer_id,
   COUNT(subscription_id) AS 'subscriptions'
FROM orders
GROUP BY customer_id;
```

This query is good, but a `customer_id` isn't terribly useful for our marketing department, they probably want to know the customer's name. We want to be able to join the results of this query with our `customers` table, which will tell us the name of each customer. We can do this by using a `WITH` clause.

```
WITH previous_results AS (
   SELECT ...
   ...
   ...
   ...
)
SELECT *
FROM previous_results
JOIN customers
  ON _____ = _____;
```

- The `WITH` statement allows us to perform a separate query (such as aggregating customer's subscriptions)
- `previous_results` is the alias that we will use to reference any columns from the query inside of the `WITH` clause
- We can then go on to do whatever we want with this temporary table (such as join the temporary table with another table)

Essentially, we are putting a whole first query inside the parentheses `()` and giving it a name. After that, we can use this name as if it's a table and write a new query *using* the first query.

1. Place the whole query below into a `WITH` statement, inside parentheses `()`, and give it name `previous_query`:

```
SELECT customer_id,
   COUNT(subscription_id) AS 'subscriptions'
FROM orders
GROUP BY customer_id
```

Join the temporary table `previous_query` with `customers` table and select the following columns:

- `customers.customer_name`
- `previous_query.subscriptions`

Check the answer in the hint below.

```
WITH previous_query AS (
   SELECT customer_id,
      COUNT(subscription_id) AS 'subscriptions'
   FROM orders
   GROUP BY customer_id
)
SELECT customers.customer_name,
   previous_query.subscriptions
FROM previous_query
JOIN customers
  ON previous_query.customer_id = customers.customer_id;
```

Review

In this lesson, we learned about relationships between tables in relational databases and how to query information from multiple tables using SQL. Let's summarize what we've learned so far:

- `JOIN` will combine rows from different tables if the join condition is true.
- `LEFT JOIN` will return every row in the *left* table, and if the join condition is not met, `NULL` values are used to fill in the columns from the *right* table.
- *Primary key* is a column that serves a unique identifier for the rows in the table.
- *Foreign key* is a column that contains the primary key to another table.
- `CROSS JOIN` lets us combine all rows of one table with all rows of another table.
- `UNION` stacks one dataset on top of another.
- `WITH` allows us to define one or more temporary tables that can be used in the final query.

Download the Multiple Tables: Cheat Sheet to help you remember the content covered in this lesson.