**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.

**Instructions**

**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;

Hint

Make sure to select the tables in the order shown above.

**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.

**Instructions**

**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.

Hint

First, we look at the orders table.

The order with an order\_id of 1 is in the first row. Its subscription\_id is 3.

Then we look at the subscriptions table.

The subscription with a subscription\_id of 3 is in the third row. Its description is Sports Magazine.

**Answer:** Sports Magazine

Don’t write a query for this exercise!

**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.

Hint

**Answer:** 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;

**Instructions**

**1.**

Join orders table and subscriptions table and select all columns.

Make sure to join on the subscription\_id column.

Hint

Suppose we do:

SELECT \*

FROM orders

LIMIT 10;

SELECT \*

FROM subscriptions

LIMIT 10;

You will notice that both orders table and subscriptions table have a subscription\_id column.

And we want to match these two columns:

* orders.subscription\_id
* subscriptions.subscription\_id

SELECT \*

FROM orders

JOIN subscriptions

ON orders.subscription\_id = subscriptions.subscription\_id;

Notice that in the column subscriptions.subscription\_id, the table name has a ‘s’ in the end.

**2.**

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’.

Hint

The WHERE clause goes after the JOIN!

Answer:

-- Checkpoint 1

SELECT \*

FROM orders

JOIN subscriptions

ON orders.subscription\_id = subscriptions.subscription\_id;

-- Checkpoint 2

SELECT \*

FROM orders

JOIN subscriptions

ON orders.subscription\_id = subscriptions.subscription\_id

WHERE subscriptions.description = 'Fashion Magazine';

What is the difference between the two result tables?

**Inner Joins**

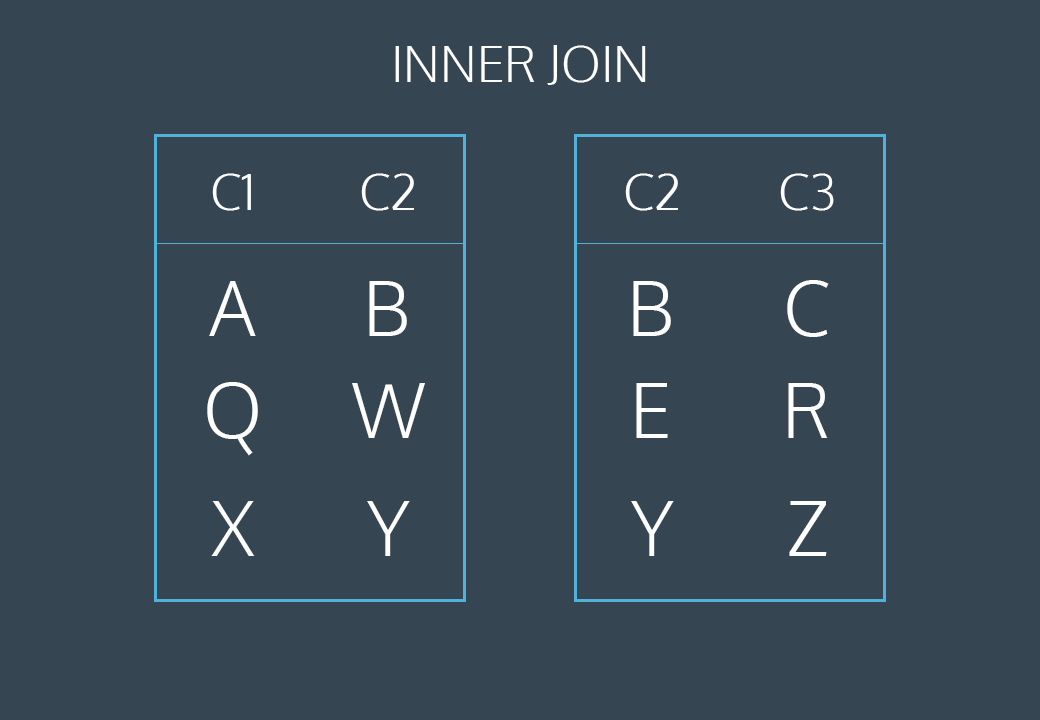
Let’s revisit how we joined orders and customers. For every possible value of customer\_id in orders, there was a corresponding row of customers with the same customer\_id.

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.

**Instructions**

**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().

Hint

Use COUNT(\*) to count all rows of a table:

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().

Hint

Use COUNT(\*) to count all rows of a table:

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?

Hint

Suppose we do:

SELECT \*

FROM newspaper

LIMIT 10;

SELECT \*

FROM online

LIMIT 10;

You will notice that both newspaper table and online table have an id column.

And we want to match these two columns:

* newspaper.id
* online.id

Your ON statement should look like this:

ON newspaper.id = online.id

Remember to use SELECT COUNT(\*) to count the rows:

SELECT COUNT(\*)

FROM newspaper

JOIN online

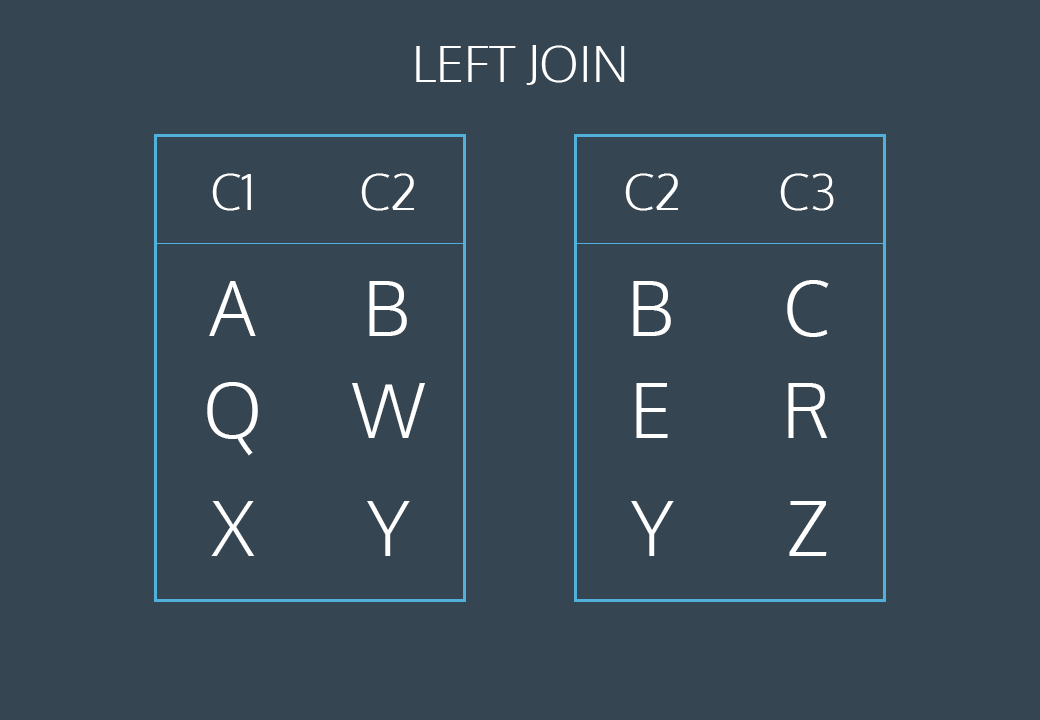
ON newspaper.id = online.id;

**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).

**Instructions**

**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.

Hint

Remember to put newspaper first to keep all rows of newspaper:

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.

Hint

Don’t remove your previous query.

Add the second query so you can compare the results!

The second query should look like:

SELECT \*

FROM newspaper

LEFT JOIN online

ON newspaper.id = online.id

WHERE online.id IS NULL;

Remember, WHERE clause goes after the LEFT JOIN.

**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.

**Instructions**

**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.

Hint

Your ON statement should include:

* classes.id (a primary key)
* students.class\_id (a foreign key)

It should look like:

SELECT \*

FROM classes

JOIN students

ON classes.id = students.class\_id;

You should already know how to do this join. But in this exercise, you learned that the matching column is usually a primary key of a table and foreign key of another!

**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.

**Instructions**

**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

Hint

“During March” means that the customer’s starting month was in or before March and final month was in or after March:

SELECT COUNT(\*)

FROM newspaper

WHERE start\_month <= 3

AND end\_month >= 3;

Answer: 13

**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.

Hint

SELECT \*

FROM newspaper

CROSS JOIN months;

When you get the result, make sure to scroll right to take a look at the rightmost column, month.

Each customer is CROSS JOIN‘ed with each month.

**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.

Hint

SELECT \*

FROM newspaper

CROSS JOIN months

WHERE \_\_\_\_\_\_\_\_ AND \_\_\_\_\_\_\_\_;

Scroll down to take a look at the result of this query.

Notice how it filtered from the previous CROSS JOIN.

Scroll right to look at the month column – some months are gone now.

**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 \_\_\_\_\_\_\_\_;

Hint

SELECT month,

COUNT(\*)

FROM newspaper

CROSS JOIN months

WHERE start\_month <= month

AND end\_month >= month

GROUP BY month;

While we are at it, let’s rename the second column using AS:

SELECT month,

COUNT(\*) AS 'subscribers'

FROM newspaper

CROSS JOIN months

WHERE start\_month <= month

AND end\_month >= month

GROUP BY month;

Scroll down to see the final result.

Which month has the highest subscribers? June!

**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.

**Instructions**

**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.

Hint

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.

**Instructions**

**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.

Hint

Remember to use the following ON statement as part of your JOIN:

ON previous\_query.customer\_id = customers.customer\_id

And for review, AS is how you give something an alias.

Here, we are essentially giving everything inside the parentheses (the sub-query) the name of previous\_query using AS.

Then, previous\_query is used as a temporary table that we will query from and also join with the customers table:

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;

Do *not* include ; inside of the () of your WITH statement.

# 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.

**Code Challenge 1**

For this challenge, you’ll use the following tables:

plans

|  |  |
| --- | --- |
| **Column** | **Description** |
| id | A unique identifier for the plan |
| price | The monthly cost of the plan |
| description | A description of the plan |

premium\_users

|  |  |
| --- | --- |
| **Column** | **Description** |
| user\_id | A unique identifier for the user |
| membership\_plan\_id | An ID for the user's payment plan (matches `plans.id`) |
| purchase\_date | Date when the user purchased their premium plan |
| cancel\_date | Date when the user canceled (NULL if they haven't) |

Click [here](https://s3.amazonaws.com/codecademy-content/courses/sql-intensive/SQL_diagram_v6.pdf) for the table descriptions.

**Instructions**

**1.**

Let’s see which plans are used by which premium members!

The column membership\_plan\_id in premium\_users should match the column id in plans.

Join plans and premium\_users and select:

* user\_id from premium\_users
* description from plans

(Be sure to select the columns in this order)

Hint

Your select statement should be:

SELECT premium\_users.user\_id,

plans.description

FROM premium\_users

JOIN plans

ON plans.id = premium\_users.membership\_plan\_id;

**Code Challenge 2**

For this exercise, you’ll use the following tables:

songs

| **Column** | **Description** |
| --- | --- |
| id | A unique identifier for each song |
| title | The title of the song |
| artist | The artist who recorded the song |
| year | The year that the song was released |

plays

| **Column** | **Description** |
| --- | --- |
| user\_id | A unique identifier for each user |
| song\_id | An ID for which song was played (matches songs.id) |
| play\_date | The date when the user played this song |
| play\_hour | The hour when the user played this song (0-23) |

Click [here](https://s3.amazonaws.com/codecademy-content/courses/sql-intensive/SQL_diagram_v6.pdf) for the table descriptions.

**Instructions**

**1.**

Let’s see the titles of songs that were played by each user!

The column song\_id in plays should match the column id in songs.

Join plays to songs and select:

* user\_id from plays
* play\_date from plays
* title from songs

(Be sure to select the columns in this order)

Hint

Your ON statement should be:

ON plays.song\_id = songs.id

Answer:

SELECT plays.user\_id,

plays.play\_date,

songs.title

FROM plays

JOIN songs

ON plays.song\_id = songs.id;

# Code Challenge 3

For this challenge, you’ll use the following tables:

users

| **Column** | **Description** |
| --- | --- |
| id | A unique identifier for each user |
| first\_name | The first name of the user |
| last\_name | The last name of the user |
| age | The age name of the user |
| gender | The gender name of the user |

premium\_users

| **Column** | **Description** |
| --- | --- |
| user\_id | A unique identifier for each user |
| membership\_plan\_id | An ID for the user’s payment plan (matches plans.id) |
| purchase\_date | Date when the user purchased their premium plan |
| cancel\_date | Date when the user canceled (NULL if they haven’t) |

Click [here](https://s3.amazonaws.com/codecademy-content/courses/sql-intensive/SQL_diagram_v6.pdf) for the table descriptions.

**Instructions**

**1.**

Which users aren’t premium users?

Use a LEFT JOIN to combine users and premium\_users and select id from users.

The column id in users should match the column user\_id in premium\_users.

Use a WHERE clause to limit the results to users where premium\_users.user\_id IS NULL. This will remove premium users and leave you with only free users.

 select users.id from users left join premium\_users on users.id=premium\_users.user\_id where premium\_users.user\_id IS NULL;

**Code Challenge 4**

We’ve used a WITH statement to create two temporary tables:

* january contains all song plays from January 2017
* february contains all song plays from February 2017

If you need help, check out this [Reference Guide](https://s3.amazonaws.com/codecademy-content/courses/learn-sql/SQL_Reference_sheet_v3.pdf) to multiple tables in SQL.

**Instructions**

**1.**

Use a left join to combine january and february on user\_id and select user\_id from january.

Add the following WHERE statement to find which users played songs in January, but not February:

WHERE february.user\_id IS NULL

Hint

Here’s some sample code to help you out:

SELECT january.user\_id

FROM \_\_\_\_\_\_

LEFT JOIN \_\_\_\_\_\_

ON \_\_\_\_\_\_\_\_\_\_ = \_\_\_\_\_\_\_\_\_\_

WHERE february.user\_id IS NULL;

WITH january AS (

  SELECT \*

  FROM plays

  WHERE strftime("%m", play\_date) = '01'

),

february AS (

  SELECT \*

  FROM plays

  WHERE strftime("%m", play\_date) = '02'

)

select january.user\_id from january left join february on january.user\_id = february.user\_id where february.user\_id IS NULL;

**CODE CHALLENGE: MULTIPLE TABLES**

**Code Challenge 5**

For this challenge, you’ll use the following tables:

months

| **Column** | **Description** |
| --- | --- |
| months | The first date of each month of the year |

premium\_users

| **Column** | **Description** |
| --- | --- |
| user\_id | A unique identifier for the user |
| plan\_id | An ID for the user’s payment plan (matches plans.id) |
| purchase\_date | Date when the user purchased their premium plan |
| cancel\_date | Date when the user canceled (NULL if they haven’t) |

Click [here](https://s3.amazonaws.com/codecademy-content/courses/sql-intensive/SQL_diagram_v6.pdf) for the table descriptions.

**Instructions**

**1.**

For each month in months, we want to know if each user in premium\_users was active or canceled. Cross join months and premium\_users and select:

* user\_id from premium\_users
* purchase\_date from premium\_users
* cancel\_date from premium\_users
* months from months

select premium\_users.user\_id, premium\_users.purchase\_date, premium\_users.cancel\_date, months.months from months cross join premium\_users;

# Code Challenge 6

If you need help, check out this [Reference Guide](https://s3.amazonaws.com/codecademy-content/courses/learn-sql/SQL_Reference_sheet_v3.pdf) to multiple tables in SQL.

**Instructions**

**1.**

Replace the SELECT statement in your CROSS JOIN with the following statement:

SELECT premium\_users.user\_id,

months.months,

CASE

WHEN (

premium\_users.purchase\_date <= months.months

)

AND

(

premium\_users.cancel\_date >= months.months

OR

premium\_users.cancel\_date IS NULL

)

THEN 'active'

ELSE 'not\_active'

END AS 'status'

This will tell us if a particular user is 'active' or 'not\_active' each month.

SELECT premium\_users.user\_id,

  premium\_users.purchase\_date,

  premium\_users.cancel\_date,

  months.months

FROM premium\_users

CROSS JOIN months;

SELECT premium\_users.user\_id,

  months.months,

  CASE

    WHEN (

      premium\_users.purchase\_date <=

             months.months

      )

      AND

      (

        (premium\_users.cancel\_date >=

                months.months)

        OR

        premium\_users.cancel\_date IS NULL

      )

    THEN 'active'

    ELSE 'not\_active'

  END as status

FROM premium\_users

CROSS JOIN months;

# Code Challenge 7

If you need help, check out this [Reference Guide](https://s3.amazonaws.com/codecademy-content/courses/learn-sql/SQL_Reference_sheet_v3.pdf) to multiple tables in SQL.

**Instructions**

**1.**

Songify has added some new songs to their catalog.

Combine songs and bonus\_songs using UNION and select all columns from the result.

Since the songs table is so big, just look at a sample by LIMITing the results to 10 rows.

Hint

Recall that syntax for combining two tables using UNION:

SELECT \*

FROM table1

UNION

SELECT \*

FROM table2;

And don’t forget to use LIMIT for this challenge.

select \* from songs union select \* from bonus\_songs limit 10;

# Code Challenge 8

Besides stacking one table on top of another, we can also use UNION to quickly make a “mini” dataset:

SELECT '2017-01-01' AS 'month'

UNION

SELECT '2017-02-01' AS 'month'

will produce:

| **month** |
| --- |
| 2017-01-01 |
| 2017-02-01 |

If you need help, check out this [Reference Guide](https://s3.amazonaws.com/codecademy-content/courses/learn-sql/SQL_Reference_sheet_v3.pdf) to multiple tables in SQL.

**Instructions**

**1.**

Modify the query in **test.sqlite**:

Add a third UNION/SELECT so that the result contains 2017-03-01.

Hint

The UNION operator is used to combine the result-set of two or more SELECT statements.

Here, we are using UNION to stack multiple times:

SELECT '2017-01-01' AS 'month'

UNION

SELECT '2017-02-01' AS 'month'

UNION

SELECT '2017-03-01' AS 'month';

**Code Challenge 9**

The following exercise uses the Songify tables explained before. You can look up the schema of those tables [here](https://s3.amazonaws.com/codecademy-content/courses/sql-intensive/SQL_diagram_v6.pdf).

If you need help, check out this [Reference Guide](https://s3.amazonaws.com/codecademy-content/courses/learn-sql/SQL_Reference_sheet_v3.pdf) to multiple tables in SQL.

**Instructions**

**1.**

The following query will give us the number of times that each song was played:

SELECT song\_id,

COUNT(\*) AS 'times\_played'

FROM plays

GROUP BY song\_id;

Use a WITH statement to alias this code as play\_count.

Join play\_count with songs and select (in this order):

* songs table’s title column
* songs table’s artist column
* play\_count‘s times\_played column

Remember that play\_count.song\_id will match songs.id.

Hint

Your WITH statement will look like this:

WITH play\_count AS (

SELECT song\_id,

COUNT(\*) AS 'times\_played'

FROM plays

GROUP BY song\_id)

We are putting the query inside the parentheses and giving it the result table a name of play\_count.

And now we can use the temporary table play\_count just like a normal table:

WITH play\_count AS (

SELECT song\_id,

COUNT(\*) AS 'times\_played'

FROM plays

GROUP BY song\_id)

SELECT songs.title,

songs.artist,

play\_count.times\_played

FROM play\_count

JOIN songs

ON play\_count.song\_id = songs.id;