Marketing Analytics - Week 2

Getting Started with SQL and BigQuery

In week 2, we will:

- Learn the workflow for handling data by SQL
- Make your first attempt at using BigQuery

Introduction to SQL

Structured Query Language, or **SQL**, is the programming language used with databases, and it is an important skill for any data analyst.
Last week, you learned how to access and explore a dataset in Google Cloud Platform (GCP), and you're ready to write your first SQL query! As you'll soon see, SQL queries will help you sort through a massive dataset to retrieve only the information that you need.

1. Basic Concepts
   a) Database: A database is a collection of related data that is organized and stored in a structured manner. For example, a company might have a database of employee information, including names, addresses, and salaries.
   b) Table: A table is a collection of data stored in rows and columns. Each column represents a field or attribute of the data, and each row represents a record. For example, a company might have a table of employee data with columns like "EmployeeID", "Name", "Department", and "Salary".
   c) Query: A query is a request for data from a database. It is used to retrieve, modify, or manipulate data in a database. For example, a query might be used to retrieve all employees who work in the IT department.

2. Basic SQL syntax
   - SELECT…FROM
   - WHERE…
   - COUNT()
   - GROUP BY
   - ORDER BY
   - DATE

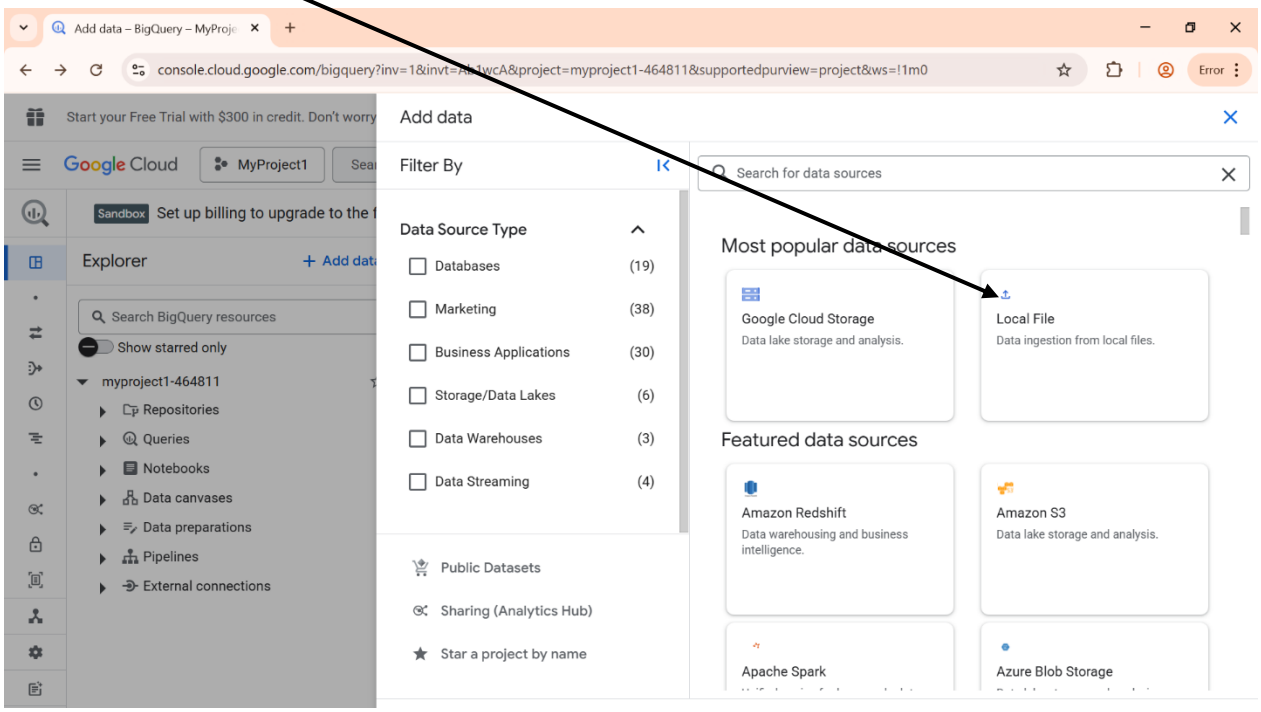
We'll begin by using the keywords **SELECT**, **FROM**, and **WHERE** to get data from specific columns based on conditions you specify.

For clarity, we'll work with a small imaginary dataset pet_records, which contains just one table, called pets.

Here is what the dataset looks like:

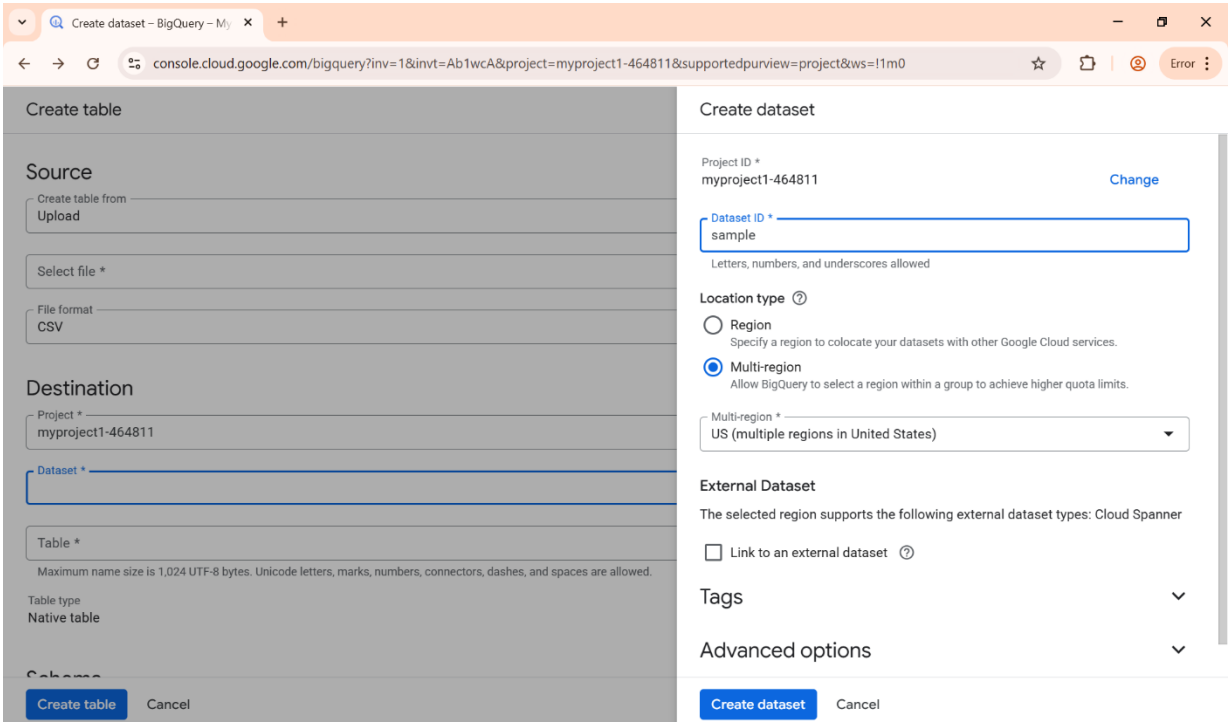
ID	Name	Animal
1	Dr. Harris Bonkers	Rabbit
2	Moon	Dog
3	Ripley	Cat
4	Tom	Cat

1. Download pet_records.csv file from the unit
2. Sign in to Google Cloud Console
   Navigate to: https://console.cloud.google.com/bigquery
3. Click + Add Data.
4. Click Local File and in the next step, click Local File again.

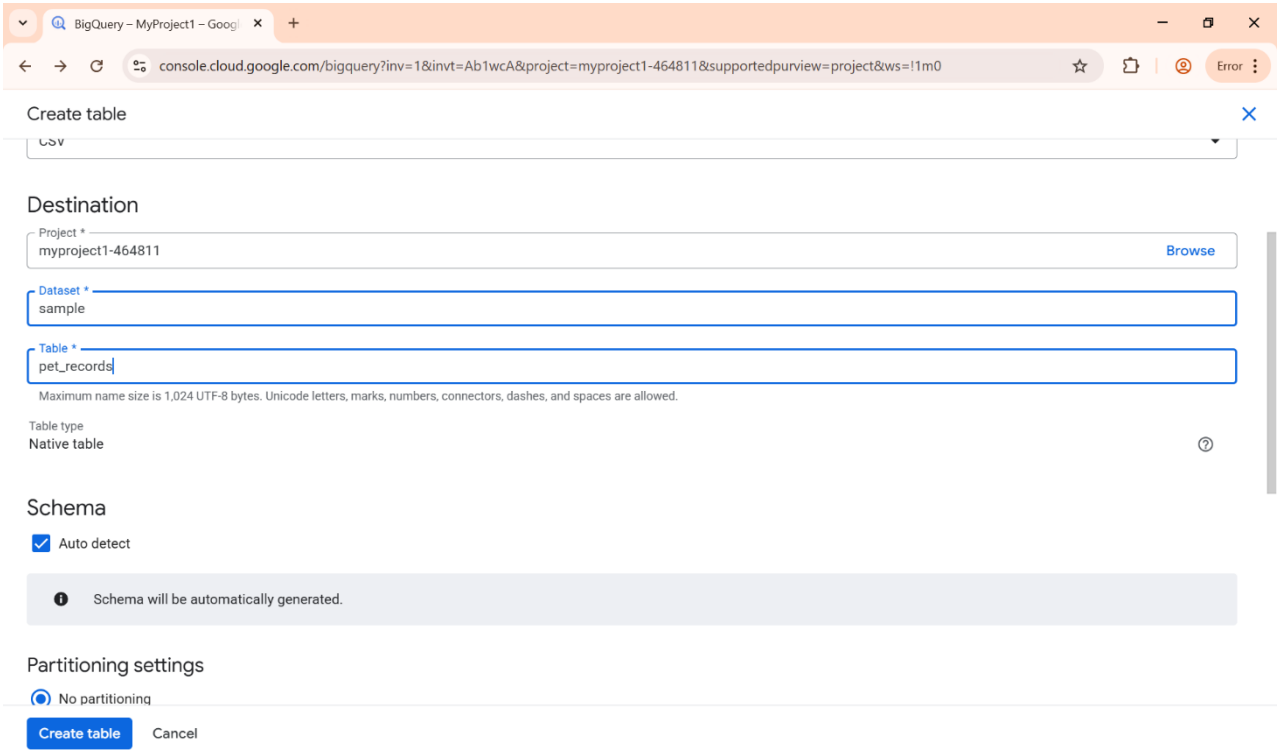


5. Select Create table > Create table from [Upload] > Select pet_records.csv file [your local path] > File format [CSV]

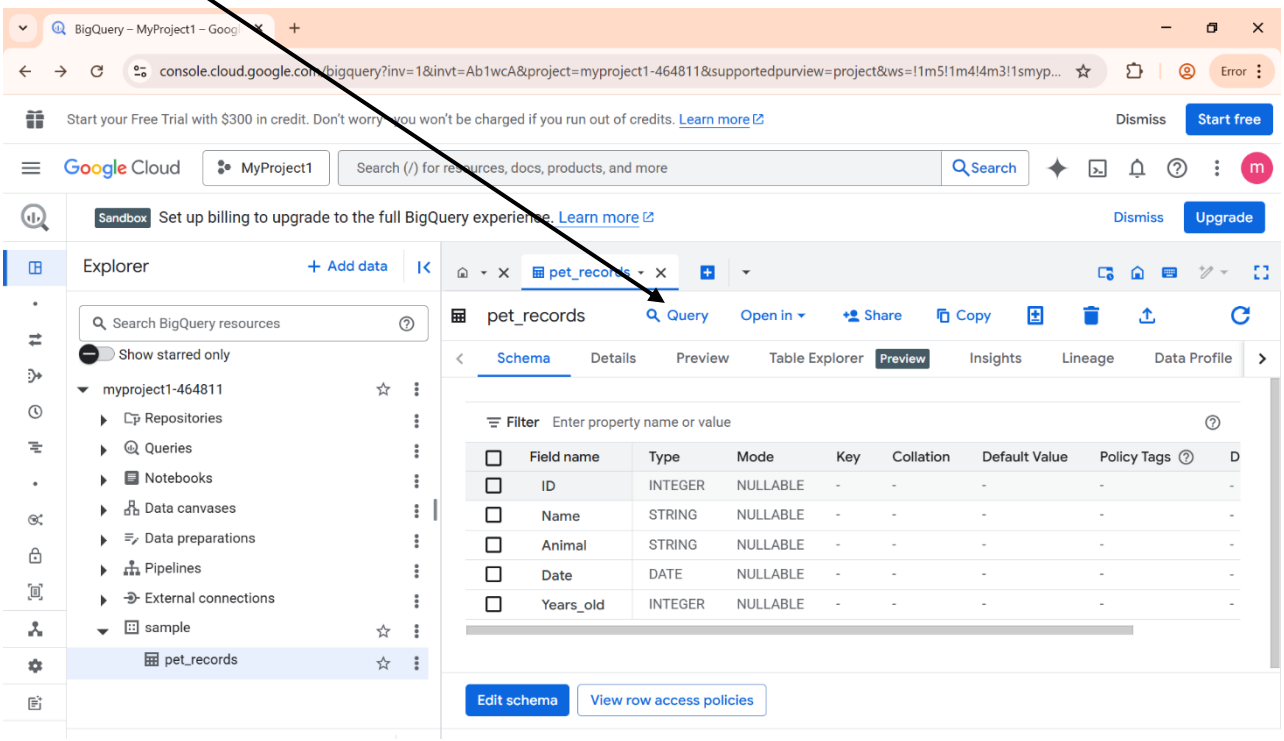
6. Click Dataset < CREATE NEW DATASET < Dataset ID [your preferred name] – (leave others as default) < click CREATE DATASET



7. Table [pet_records]. Schema, tick Auto detect, and click Create Table.



8. Click Query.



SELECT ... FROM

The most basic SQL query selects a single column from a single table. To do this,

- specify the column you want after the word **SELECT**, and then
- specify the table after the word **FROM**.

For instance, to select the Name column (from the pets table in the pet_records database in the bigquery-public-data project), our query would appear as follows:

```
query = """
        SELECT Name
        FROM `bigquery-public-data.pet_records.pets`
        """
```

ID	Name	Animal
1	Dr. Harris Bonkers	Rabbit
2	Moon	Dog
3	Ripley	Cat
4	Tom	Cat

Note that when writing an SQL query, the argument we pass to **FROM** is *not* in single or double quotation marks (' or "). It is in backticks (`).

Task: Type the query above.

WHERE ...

Some datasets are large, so you'll usually want to return only the rows meeting specific conditions. You can do this using the **WHERE** clause.

The query below returns the entries from the Name column that are in rows where the Animal column has the text 'Cat'.

```
query = """
        SELECT Name
        FROM `bigquery-public-data.pet_records.pets`
        WHERE Animal = 'Cat'
        """
```

ID	Name	Animal
1	Dr. Harris Bonkers	Rabbit
2	Moon	Dog
3	Ripley	Cat
4	Tom	Cat

Task: Type the query above.

COUNT ()

COUNT (), as you may have guessed from the name, returns a count of things. If you pass it the name of a column, it will return the number of entries in that column.

For instance, if we **SELECT** the **COUNT()** of the ID column in the pets table, it will return 4, because there are 4 ID's in the table.

```
query = """
        SELECT COUNT(ID)
        FROM `bigquery-public-data.pet_records.pets`
        """
```

f0_
4

COUNT(DISTINCT) allows you to count the number of distinct or unique values in a given column. The **DISTINCT** keyword can be used in conjunction with **COUNT** to count the number of unique values within a specific column or set of columns.

Task: type the query above and compare the COUNT() vs. COUNT(DISTINCT)

COUNT() is an example of an **aggregate function**, which takes many values and returns one. Other examples of aggregate functions include **SUM()**, **AVG()**, **MIN()**, and **MAX()**.

GROUP BY

GROUP BY takes the name of one or more columns, and treats all rows with the same value in that column as a single group when you apply aggregate functions like **COUNT()**.

For example, say we want to know how many of each type of animal we have in the pets table. We can use **GROUP BY** to group together rows that have the same value in the Animal column, while using **COUNT()** to find out how many ID's we have in each group.

```
query = """
        SELECT Animal, COUNT(ID)
        FROM `bigquery-public-data.pet_records.pets`
        GROUP BY Animal
        """
```

Animal	f0_
Rabbit	1
Dog	1
Cat	2

It returns a table with three rows (one for each distinct animal). We can see that the pets table contains 1 rabbit, 1 dog, and 2 cats.

Task: Type the query above.

So far, you've learned how to use several SQL clauses. For instance, you know how to use **SELECT** to pull specific columns from a table, along with **WHERE** to pull rows that meet specified criteria. You also know how to use aggregate functions like **COUNT()**, along with **GROUP BY** to treat multiple rows as a single group.

Now you'll learn how to change the order of your results using the **ORDER BY** clause, and you'll explore a popular use case by applying ordering to dates. To illustrate what you'll learn in this tutorial, we'll work with a slightly modified version of our familiar pet's table.

HAVING

HAVING is used in combination with GROUP BY to ignore groups that don't meet certain criteria. So this query, for example, will only include groups that have more than one ID in them.

```
query = """
        SELECT Animal, COUNT(ID)
        FROM `bigquery-public-data.pet_records.pets`
        GROUP BY Animal
        HAVING COUNT(ID) > 1
        """
```

Animal	f0_
Cat	2

ORDER BY

ORDER BY is usually the last clause in your query, and it sorts the results returned by the rest of your query. Notice that the rows are not ordered by the ID column. We can quickly remedy this with the query below.

```
query = """
        SELECT ID, Name, Animal
        FROM `bigquery-public-data.pet_records.pets`
        ORDER BY ID
        """
```

ID	Name	Animal
1	Dr. Harris Bonkers	Rabbit
2	Moon	Dog
3	Ripley	Cat
4	Tom	Cat

The **ORDER BY** clause also works for columns containing text, where the results show up in alphabetical order.

```
query = """
        SELECT ID, Name, Animal
        FROM `bigquery-public-data.pet_records.pets`
        ORDER BY Animal
        """
```

ID	Name	Animal
3	Ripley	Cat
4	Tom	Cat
2	Moon	Dog
1	Dr. Harris Bonkers	Rabbit

You can reverse the order using the **DESC** argument (short for 'descending'). The next query sorts the table by the Animal column, where the values that are last in alphabetical order are returned first.

```
query = """
        SELECT ID, Name, Animal
        FROM `bigquery-public-data.pet_records.pets`
        ORDER BY Animal DESC
        """
```

ID	Name	Animal
1	Dr. Harris Bonkers	Rabbit
2	Moon	Dog
3	Ripley	Cat
4	Tom	Cat

Task: Type the query above.

Date and EXTRACT

Next, we'll introduce dates, because they come up very frequently in real-world databases. There are two ways that dates can be stored in BigQuery: as a **DATE** or as a **DATETIME**.

The **DATE** format has the year first, then the month, and then the day. It looks like this:

YYYY-[M]M-[D]D

- YYYY: Four-digit year
- [M]M: One or two digit month
- [D]D: One or two digit day

So 2019-01-10 is interpreted as January 10, 2019.

The **DATETIME** format is like the date format, but with time added at the end.

Often, you'll want to look at part of a date, like the year or the day. You can do this with **EXTRACT**. We'll illustrate this with a slightly different table, called pets_with_date.

ID	Name	Animal	Date
1	Dr. Harris Bonkers	Rabbit	2019-04-18
4	Tom	Cat	2019-05-16
2	Moon	Dog	2019-01-07
3	Ripley	Cat	2019-02-23

The query below returns two columns, where column Day contains the day corresponding to each entry in the Date column from the pets_with_date table:

```
query = """
        SELECT Name, EXTRACT(DAY from Date) AS Day
        FROM `bigquery-public-data.pet_records.pets_with_date`
        """
```

Name	Day
Dr. Harris Bonkers	18
Moon	7
Ripley	23
Tom	16

SQL is very smart about dates, and we can ask for information beyond just extracting part of the cell. For example, this query returns one column with just the week in the year (between 1 and 53) for each date in the Date column:

```
query = """
        SELECT Name, EXTRACT(WEEK from Date) AS Week
        FROM `bigquery-public-data.pet_records.pets_with_date`
        """
```

Name	Week
Dr. Harris Bonkers	15
Moon	1
Ripley	7
Tom	19

You can find all the functions you can use with dates in BigQuery in this documentation under "Date and time functions".

Task: Type the query above.

Optional Tasks

With all that you've learned, your SQL queries are getting pretty long, which can make them hard understand (and debug).

You are about to learn how to use **AS** and **WITH** to tidy up your queries and make them easier to read.

Along the way, we'll use the familiar pets table, but now it includes the ages of the animals.

ID	Name	Animal	Years_old
1	Dr. Harris Bonkers	Rabbit	4.5
2	Moon	Dog	9.0
3	Ripley	Cat	1.5
4	Tom	Cat	7.8

AS

You learned in an earlier tutorial how to use **AS** to rename the columns generated by your queries, which is also known as **aliasing**. This is similar to how Python uses as for aliasing when doing imports like import pandas as pd or import seaborn as sns.

To use **AS** in SQL, insert it right after the column you select. Here's an example of a query *without* an **AS** clause:

```
query = """
        SELECT Animal, COUNT(ID)
        FROM `bigquery-public-data.pet_records.pets`
        GROUP BY Animal
        """
```

Animal	f0_
Rabbit	1
Dog	1
Cat	2

And here's an example of the same query, but *with* **AS**.

```
query = """
        SELECT Animal, COUNT(ID) AS Number
        FROM `bigquery-public-data.pet_records.pets`
        GROUP BY Animal
        """
```

Animal	Number
Rabbit	1
Dog	1
Cat	2

These queries return the same information, but in the second query the column returned by the **COUNT()** function will be called Number, rather than the default name of f0_.

WITH ... AS

On its own, **AS** is a convenient way to clean up the data returned by your query. It's even more powerful when combined with **WITH** in what's called a "common table expression".

A **common table expression** (or **CTE**) is a temporary table that you return within your query. CTEs are helpful for splitting your queries into readable chunks, and you can write queries against them.

For instance, you might want to use the pets table to ask questions about older animals in particular. So you can start by creating a CTE which only contains information about animals more than five years old like this:

```
query = """
        WITH Seniors AS
        (
            SELECT ID, Name
            FROM `bigquery-public-data.pet_records.pets`
            WHERE Years_old > 5
        )
        This query is incomplete. More coming soon!
        """
```

ID	Name
2	Moon
4	Tom

This is a **CTE** named `Seniors`.
(*It is not returned by the query.*)

While this incomplete query above won't return anything, it creates a CTE that we can then refer to (as Seniors) while writing the rest of the query.

We can finish the query by pulling the information that we want from the CTE. The complete query below first creates the CTE, and then returns all of the IDs from it.

```
query = """
        WITH Seniors AS
        (
            SELECT ID, Name
            FROM `bigquery-public-data.pet_records.pets`
            WHERE Years_old > 5
        )
        SELECT ID
        FROM Seniors
        """
```

ID
2
4

You could do this without a CTE, but if this were the first part of a very long query, removing the CTE would make it much harder to follow.

Also, it's important to note that CTEs only exist inside the query where you create them, and you can't reference them in later queries. So, any query that uses a CTE is always broken into two parts: (1) first, we create the CTE, and then (2) we write a query that uses the CTE.

Task: Type the query above.

JOIN

You have the tools to obtain data from a single table in whatever format you want it. But what if the data you want is spread across multiple tables?

That's where **JOIN** comes in! **JOIN** is incredibly important in practical SQL workflows. So, let's get started.

We'll use our imaginary pets table, which has three columns:

1. ID - ID number for the pet
2. Name - name of the pet
3. Animal - type of animal

We'll also add another table, called owners. This table also has three columns:

- ID - ID number for the owner (different from the ID number for the pet)
- Name - name of the owner
- Pet_ID - ID number for the pet that belongs to the owner (which matches the ID number for the pet in the pets table)

`owners` table

ID	Name	Pet_ID
1	Aubrey Little	1
2	Chett Crawfish	3
3	Jules Spinner	4
4	Magnus Burnsides	2

`pets` table

ID	Name	Animal
1	Dr. Harris Bonkers	Rabbit
2	Moon	Dog
3	Ripley	Cat
4	Tom	Cat

To get information that applies to a certain pet, we match the ID column in the pets table to the Pet_ID column in the owner's table.

For example,

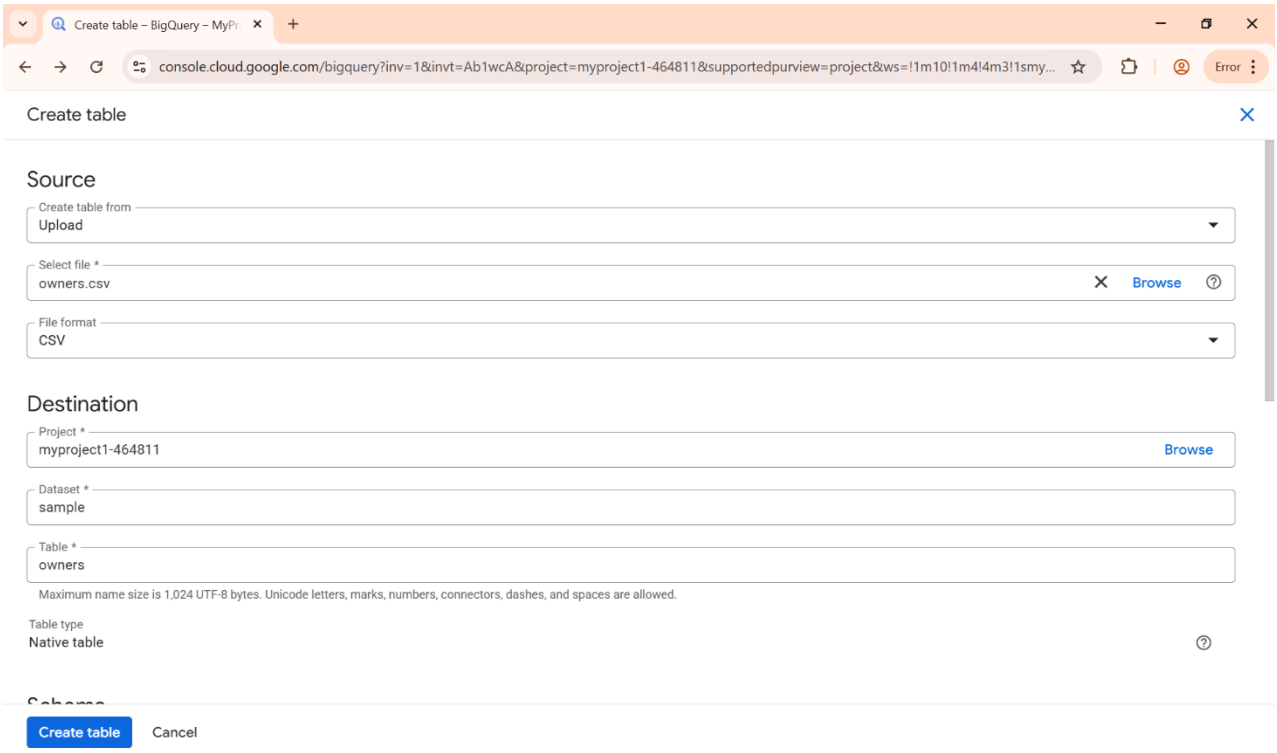
- the pets table shows that Dr. Harris Bonkers is the pet with ID 1.
- The owners table shows that Aubrey Little is the owner of the pet with ID 1.

Putting these two facts together, Dr. Harris Bonkers is owned by Aubrey Little.

Fortunately, we don't have to do this by hand to figure out which owner goes with which pet. In the next section, you'll learn how to use **JOIN** to create a new table combining information from the pets and owners' tables.

Using **JOIN**, we can write a query to create a table with just two columns: the name of the pet and the name of the owner.

Note: Upload the owners.csv file to the same Dataset following the previous steps.




```
query = """
        SELECT p.Name AS Pet_Name, o.Name AS Owner_Name
        FROM `bigquery-public-data.pet_records.pets` AS p
        INNER JOIN `bigquery-public-data.pet_records.owners` AS o
            ON p.ID = o.Pet_ID
        """
```

Pet_Name	Owner_Name
Dr. Harris Bonkers	Aubrey Little
Ripley	Chett Crawfish
Tom	Jules Spinner
Moon	Magnus Burnsides

We combine information from both tables by matching rows where the ID column in the pets table matches the Pet_ID column in the owner's table.

In the query, **ON** determines which column in each table to use to combine the tables. Notice that since the ID column exists in both tables, we have to clarify which one to use. We use p.ID to refer to the ID column from the pets table, and o.Pet_ID refers to the Pet_ID column from the owners table.

In general, when you're joining tables, it's a good habit to specify which table each of your columns comes from. That way, you don't have to pull up the schema every time you go back to read the query.

The type of **JOIN** we're using today is called an **INNER JOIN**. That means that a row will only be put in the final output table if the value in the columns you're using to combine them shows up in both the tables you're joining. For example, if Tom's ID number of 4 didn't exist in the pets table, we would only get 3 rows back from this query. There are other types of **JOIN**, but an **INNER JOIN** is very widely used, so it's a good one to start with.

Task: Type the query above.

Week 2 Task: Hands-on Experience

Now that you've got the basics down, let's work through an example with a real dataset.

The Iconic is a popular Australian online fashion and sports retailer. It was launched in 2011 by Adam Jacobs and Cameron Votan, with the aim of bringing a diverse range of international and local brands to the Australian market. Over the years, the company has become a leading player in the online fashion retail industry in Australia.

The Iconic has adopted various marketing strategies to gain a competitive edge in the Australian e-commerce market. Now you have access to its e-commerce database to evaluate the product-related questions.

You are a marketing analyst in an e-commerce company, Iconic. Your analyst team exported the Google Analytics logs for one of your ecommerce websites into BigQuery and created a new table of all the raw ecommerce visitor session data. You are going to present the following data analytical report to the team:

1. How many unique visitors (fullVisitorID) do we have to check 'Waterproof Backpack' (v2ProductName), with the GGOEGBRA037499(productSKU)? List all the channels (channelGrouping) they come from.
2. Which country has the highest average session duration (timeOnSite)? Which channel has the highest average session duration (timeOnSite)?
3. List the five visitors made the longest page views (pageviews) (include people who have viewed the same product more than once).
4. List the five products with the highest page views (pageviews).
5. What is the top landing page (pagePathLevel1) for each channel?
6. Propose questions of interest based on the online retail context and discuss with peers to develop possible answers.

Steps

Step 1: Download data from Cloud Deakin – Content – Topic 2 – For Lab (all_sessions_raws.csv file)

Step 2: Log in to GCP - BigQuery

1. Sign in to Google Cloud Console
    a. Navigate to: https://console.cloud.google.com/bigquery

Step 3: Upload CSV file to BigQuery

1. Click Navigation menu > BigQuery
2. Click + Add Data
3. Select Create table > Create table from [Upload] > Select file [your local path] > File format [CSV]

a. Click Dataset < CREATE NEW DATASET < Dataset ID [your preferred name] – (leave others as default) < click CREATE DATASET
b. Table [all_sessions_raw]
c. Schema, tick Auto detect
d. Click CREATE TABLE

Step 4: Answer the above questions by Queries

Step 5: Propose your own questions after understanding the data structure and write the queries accordingly

Step 6: Group discussion to propose the recommendations.