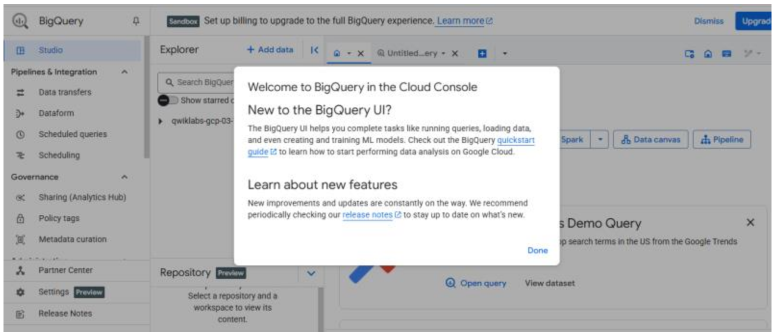
Working with JSON, Arrays, and Structs in BigQuery

Open the BigQuery console

1. In the Google Cloud Console, select **Navigation menu** > **BigQuery**.

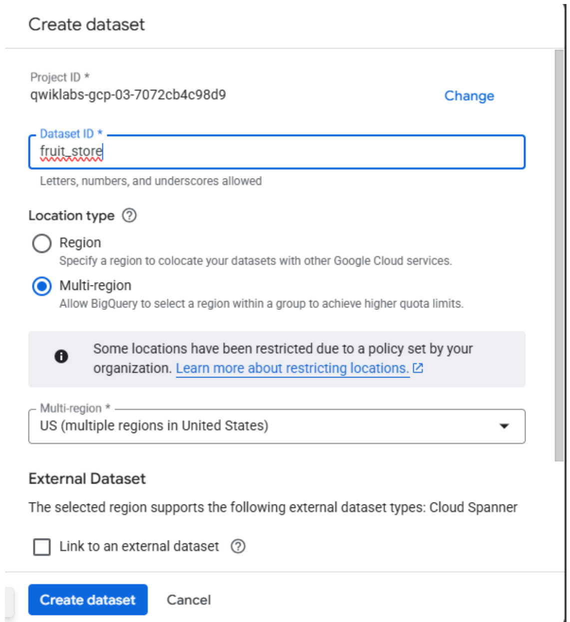
The **Welcome to BigQuery in the Cloud Console** message box opens. This message box provides a link to the quickstart guide and the release notes.

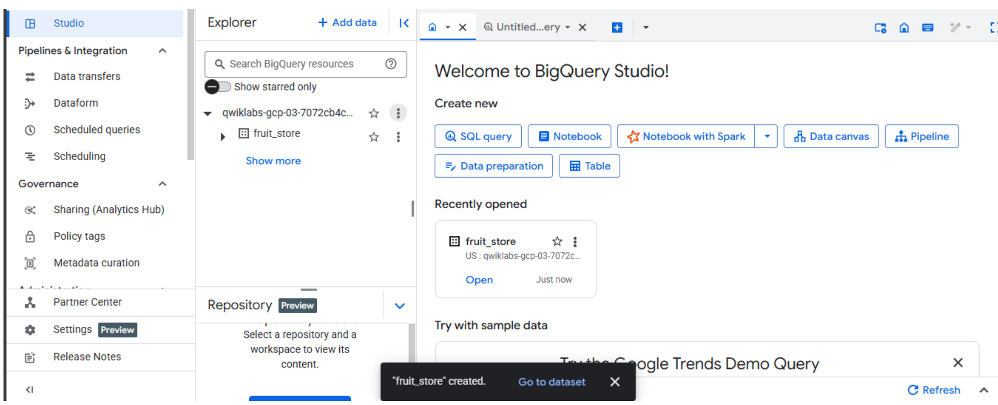
2. Click **Done**.



Task 1. Create a new dataset to store the tables

1. In your BigQuery, click the three dots next to your Project ID and select **Create dataset**:
2. Set the **Dataset ID** to fruit_store. Leave the other options at their default values (Data Location, Default Expiration).
3. Click **Create dataset**.





Task 2. Practice working with arrays in SQL

Normally in SQL you will have a single value for each row like this list of fruits below:

Row	Fruit
1	raspberry
2	blackberry
3	strawberry

4	cherry

What if you wanted a list of fruit items for each person at the store? It could look something like this:

Row	**Fruit**	**Person**
1	raspberry	sally
2	blackberry	sally
3	strawberry	sally
4	cherry	sally
5	orange	frederick
6	apple	frederick

In traditional relational database SQL, you would look at the repetition of names and immediately think of splitting the above table into two separate tables: Fruit Items and People. That process is called normalization (going from one table to many). This is a common approach for transactional databases like mySQL.

For data warehousing, data analysts often go the reverse direction (denormalization) and bring many separate tables into one large reporting table.

Now, you're going to learn a different approach that stores data at different levels of granularity all in one table using repeated fields:

Row	**Fruit (array)**	**Person**
1	raspberry	sally
	blackberry	

	strawberry	
	cherry	
2	orange	frederick
	apple	

What looks strange about the previous table?

- It's only two rows.
- There are multiple field values for Fruit in a single row.
- The people are associated with all of the field values.

What the key insight? The array data type!

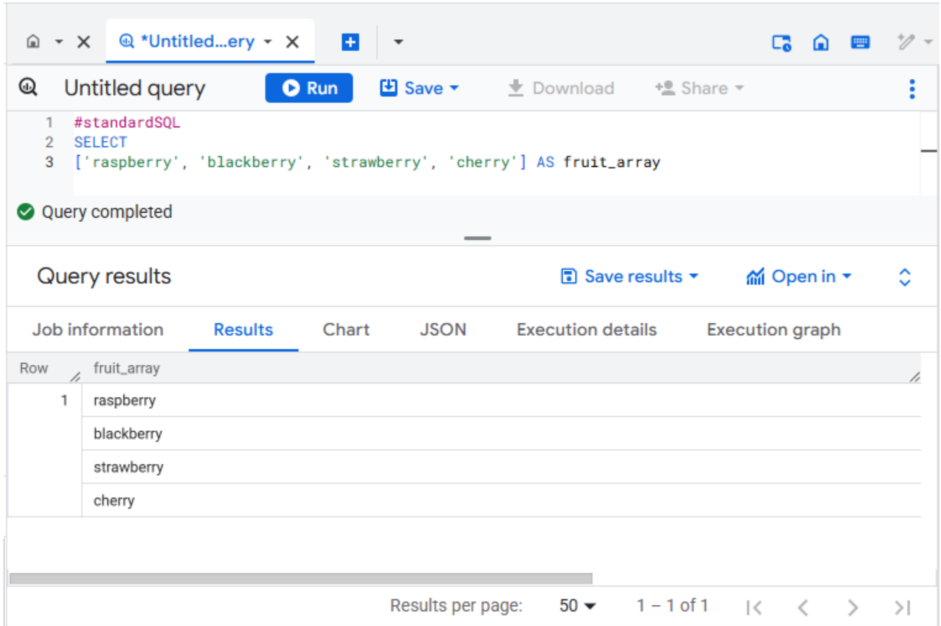
An easier way to interpret the Fruit array:

Row	**Fruit (array)**	**Person**
1	[raspberry, blackberry, strawberry, cherry]	sally
2	[orange, apple]	frederick

Both of these tables are exactly the same. There are two key learnings here:

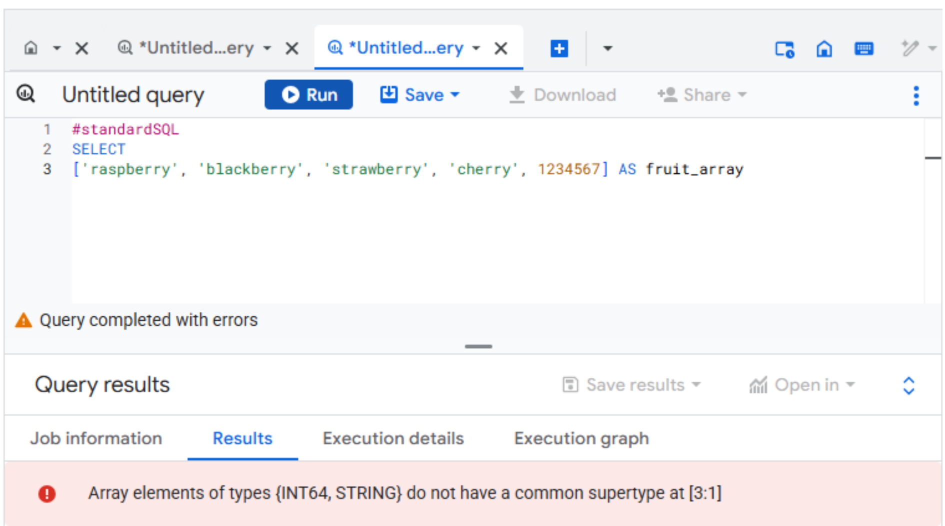
- An array is simply a list of items in brackets [ ]
- BigQuery visually displays arrays as *flattened*. It simply lists the value in the array vertically (note that all of those values still belong to a single row)

  1.Enter the following in the BigQuery Query Editor:



2. Click **Run**.

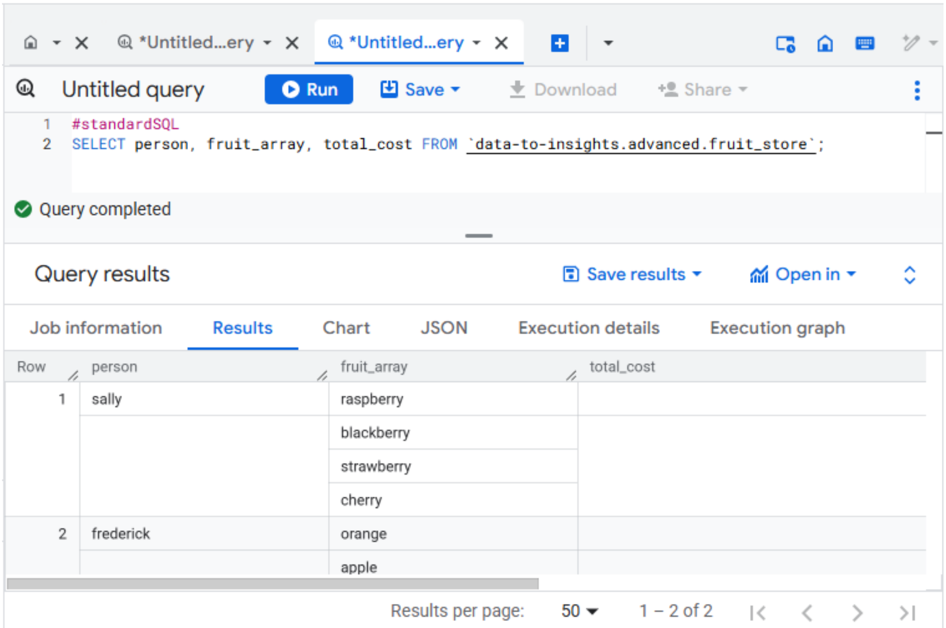
3. Now try executing this one:



You should get an error that looks like the following:

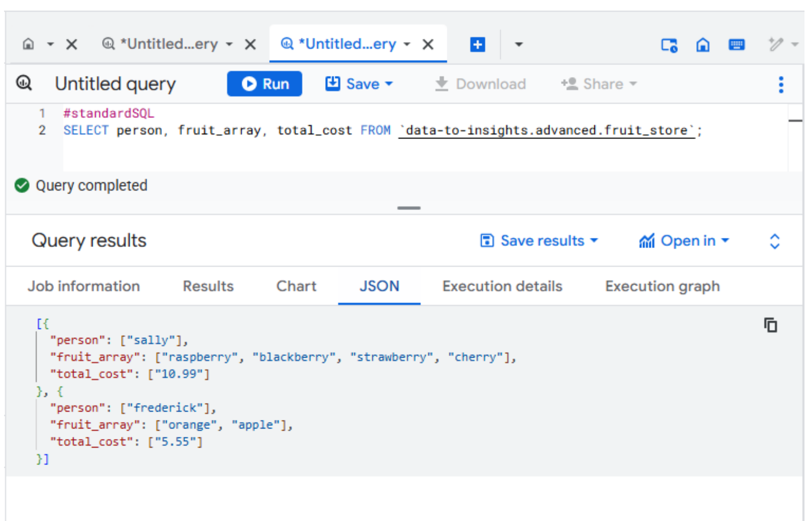
Error: Array elements of types {INT64, STRING} do not have a common supertype at [3:1]

4. Here's the final table to query against:



5. Click **Run**.

6. After viewing the results, click the **JSON** tab to view the nested structure of the results.



Loading semi-structured JSON into BigQuery

What if you had a JSON file that you needed to ingest into BigQuery?

Create a new table fruit_details in the dataset.

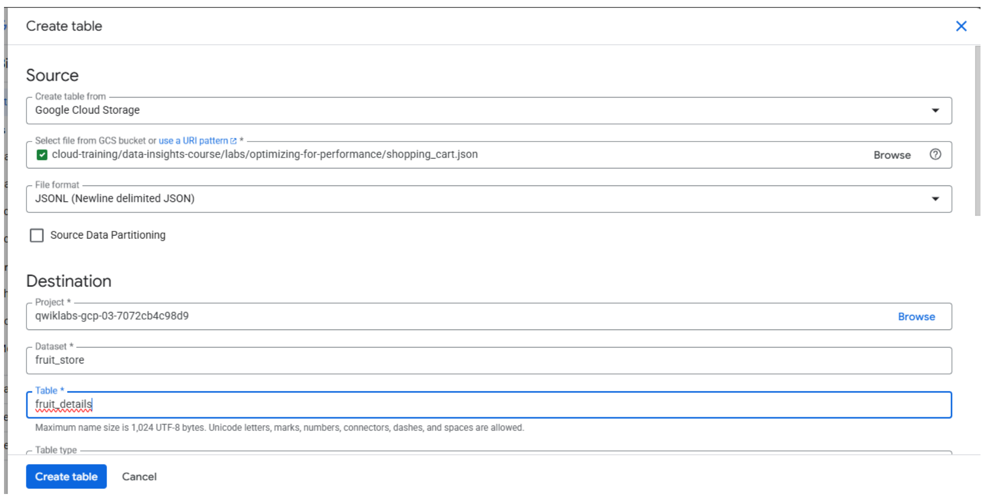
1. Click on fruit_store dataset.

Now you will see the **Create Table** option.

2. Add the following details for the table:

- **Source**: Choose **Google Cloud Storage** in the **Create table from** dropdown.
- **Select file from Cloud Storage bucket**: cloud-training/data-insights-course/labs/optimizing-for-performance/shopping_cart.json
- **File format**: JSONL (Newline delimited JSON)

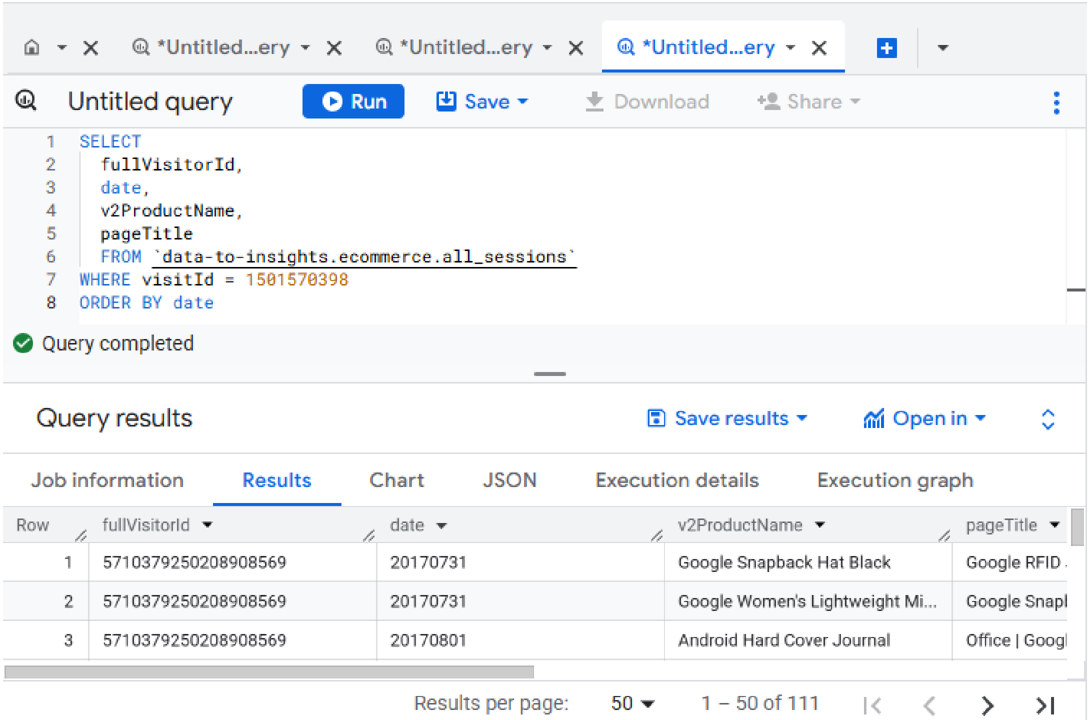
3. Call the new table fruit_details.
4. Check the checkbox of **Schema (Auto detect)**.
5. Click **Create table**.



Task 3. Create your own arrays with ARRAY_AGG()

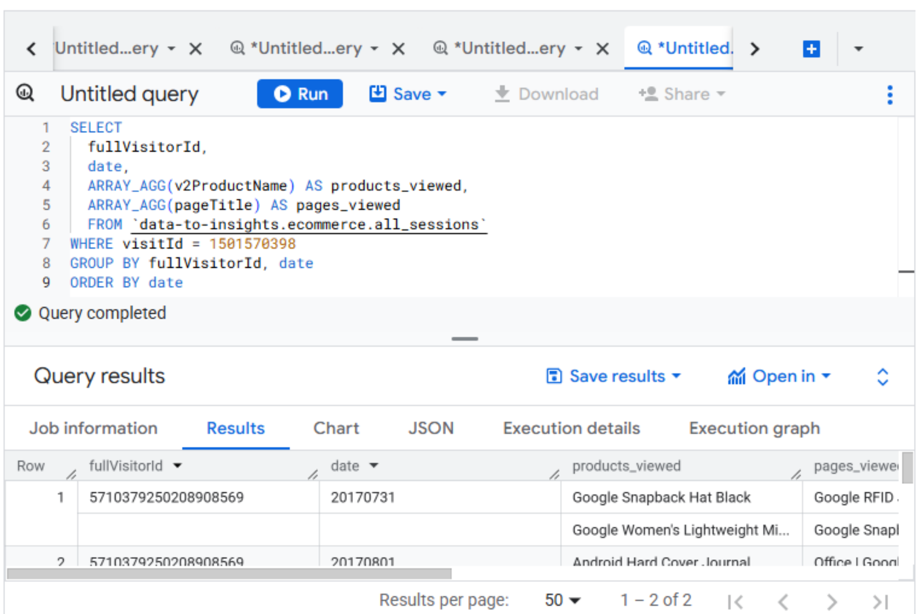
Don't have arrays in your tables already? You can create them!

1. **Copy and paste** the below query to explore this public dataset:



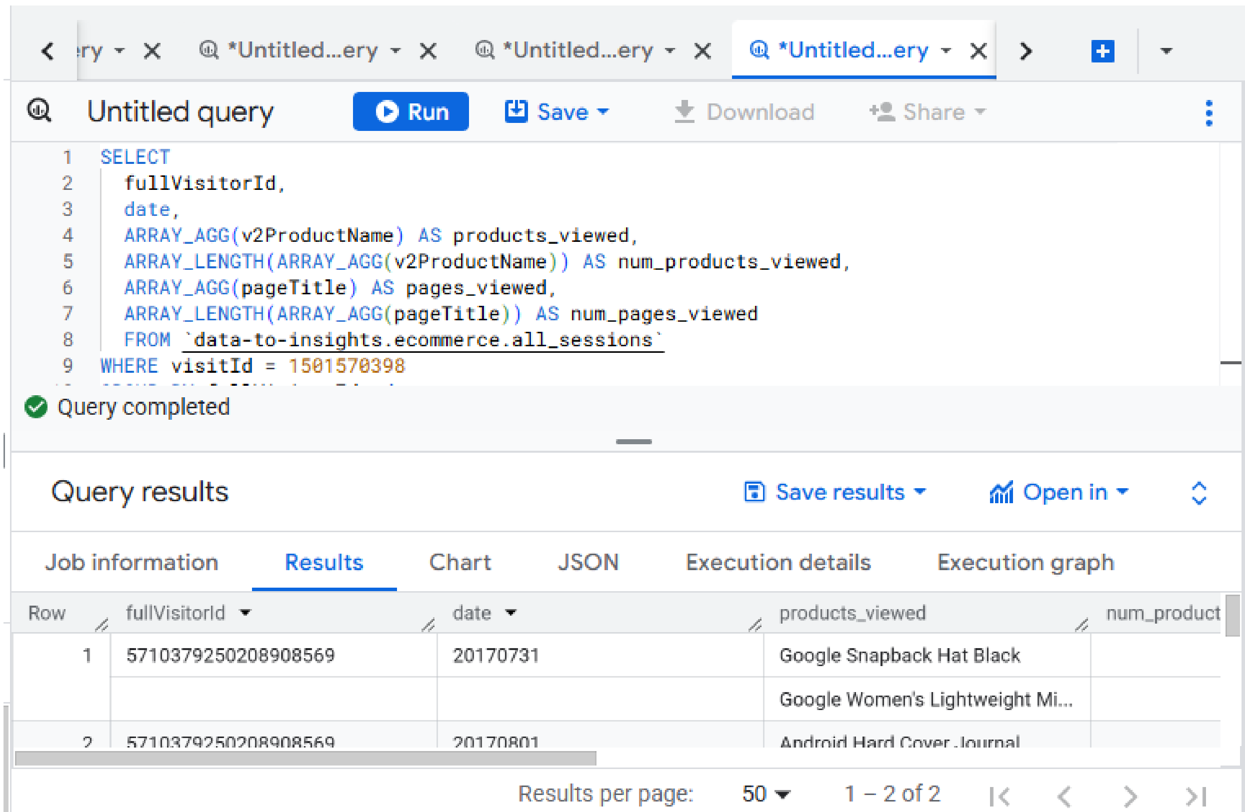
2. Click **Run** and view the results.

3. **Copy and paste** the below query to explore this public dataset:

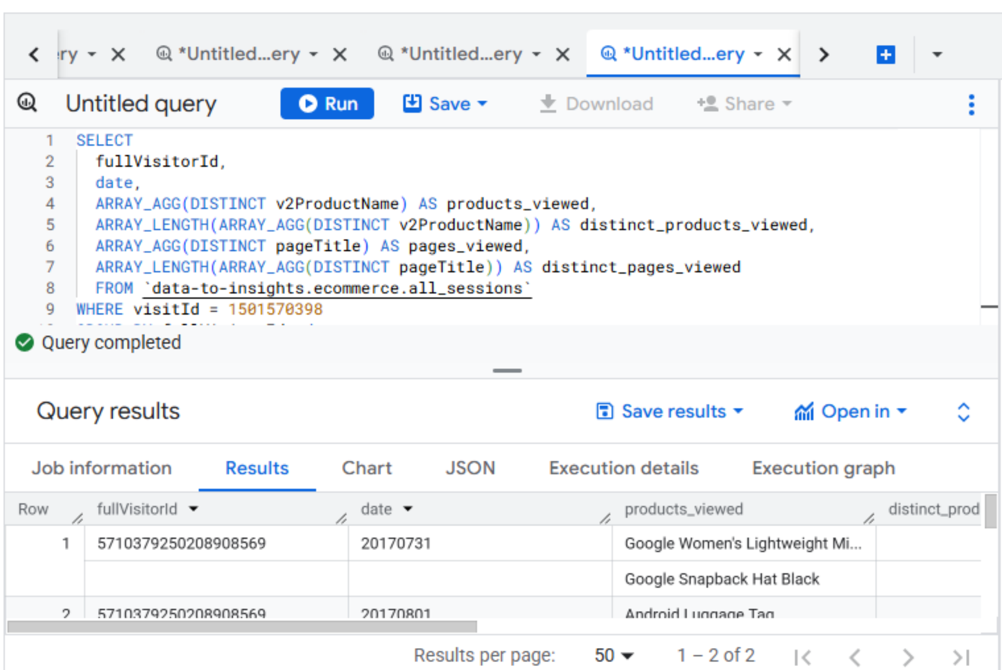


4. Click **Run** and view the results

5. Next, use the ARRAY_LENGTH() function to count the number of pages and products that were viewed:



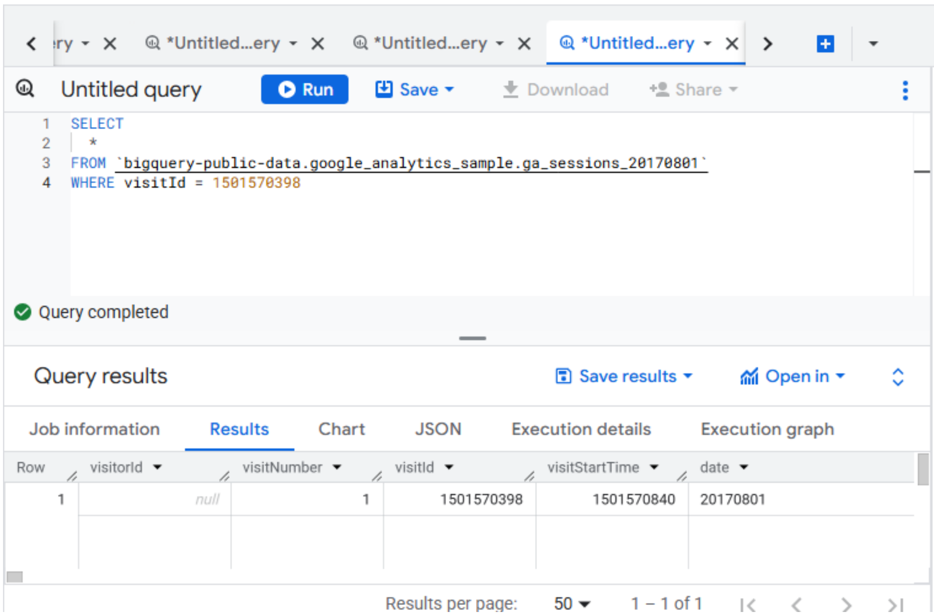
6. Next, deduplicate the pages and products so you can see how many unique products were viewed by adding DISTINCT to ARRAY_AGG():



Task 4. Query tables containing arrays

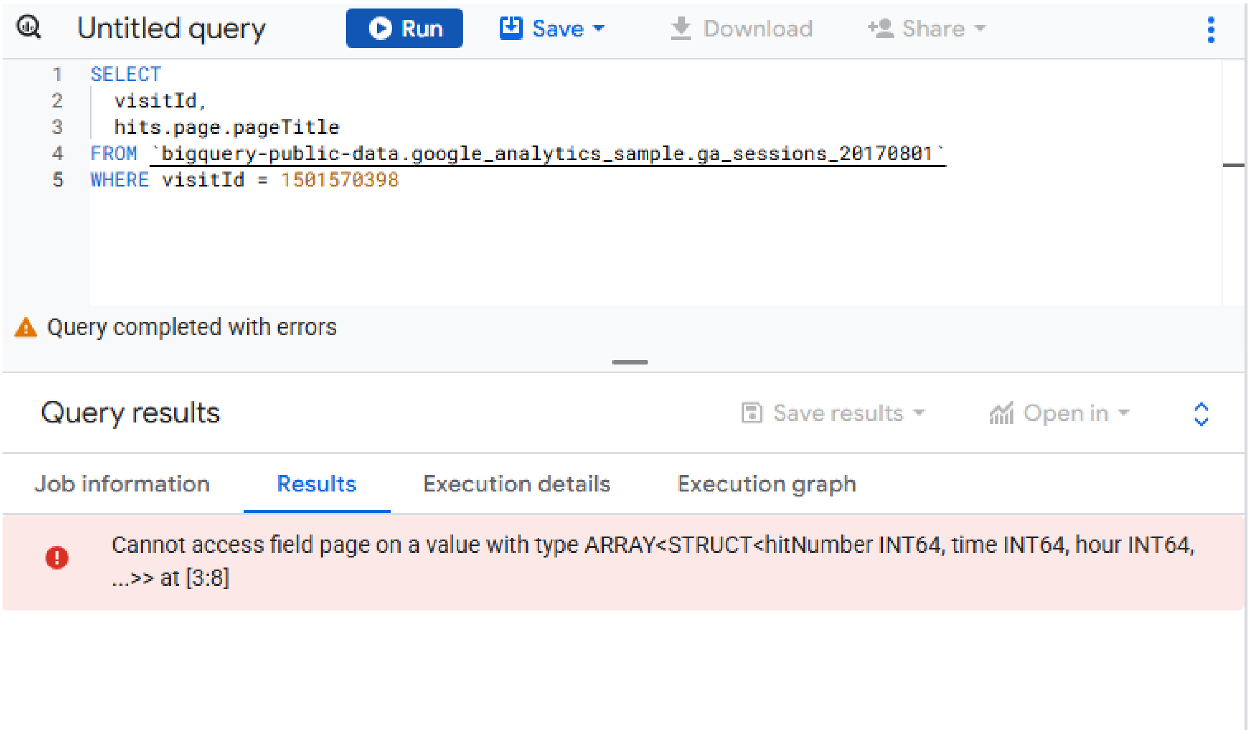
The BigQuery Public Dataset for Google Analytics bigquery-public-data.google_analytics_sample has many more fields and rows than our course dataset data-to-insights.ecommerce.all_sessions. More importantly, it already stores field values like products, pages, and transactions natively as ARRAYs.

1. **Copy and paste** the below query to explore the available data and see if you can find fields with repeated values (arrays):

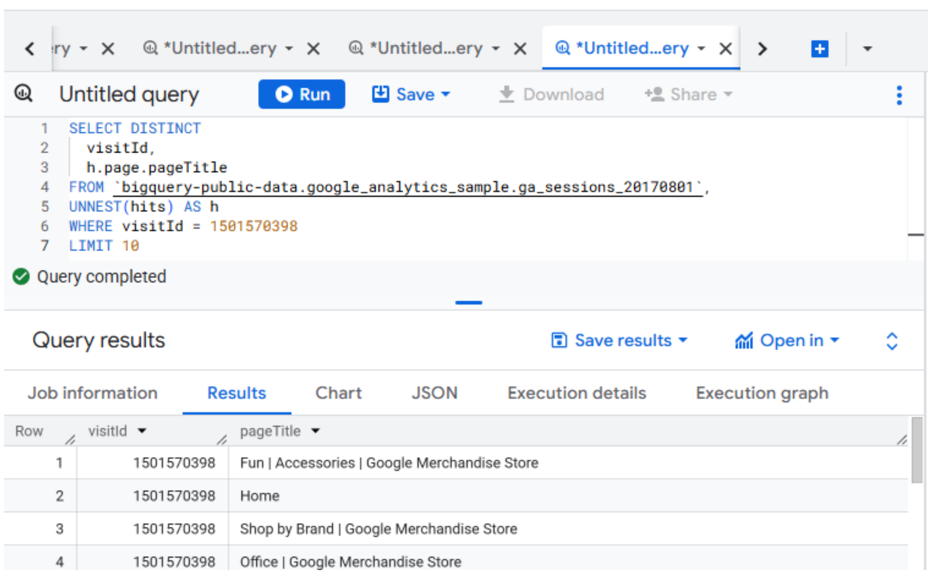


2. **Run** the query.

3. Try to query just the visit and page name fields like before:



You will get an error: **Error:**Cannot access field page on a value with type ARRAY<STRUCT<hitNumber INT64, time INT64, hour INT64, ...>> at [3:8]



We'll cover UNNEST() more in detail later but for now just know that:

- You need to UNNEST() arrays to bring the array elements back into rows
- UNNEST() always follows the table name in your FROM clause (think of it conceptually like a pre-joined table)

Task 5. Introduction to STRUCTs

You may have wondered why the field alias hit.page.pageTitle looks like three fields in one separated by periods. Just as ARRAY values give you the flexibility to *go deep* into the granularity of your fields, another data type allows you to *go wide* in your schema by grouping related fields together. That SQL data type is the STRUCT data type.

The easiest way to think about a STRUCT is to consider it conceptually like a separate table that is already pre-joined into your main table.

A STRUCT can have:

- One or many fields in it
- The same or different data types for each field
- It's own alias

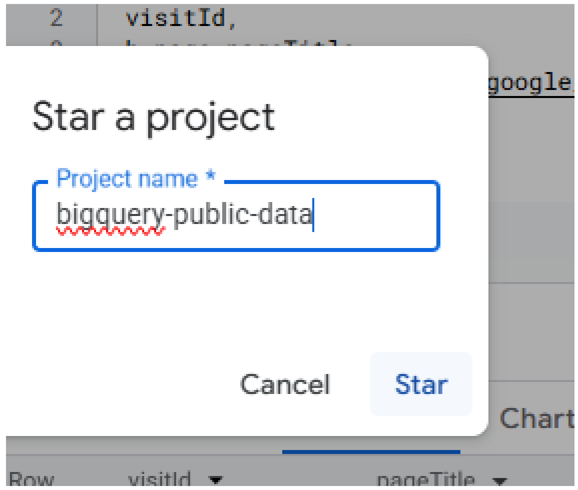
Sounds just like a table right?

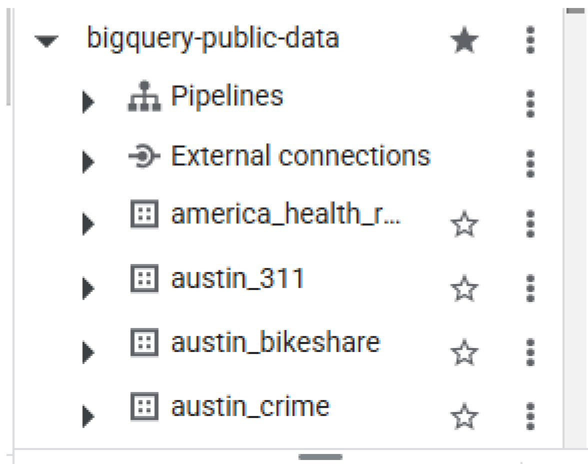
Explore a dataset with STRUCTs

1. To open the **bigquery-public-data** dataset, click **+Add Data** and then select **Star a project by name** and enter the name bigquery-public-data
2. Click **Star**.

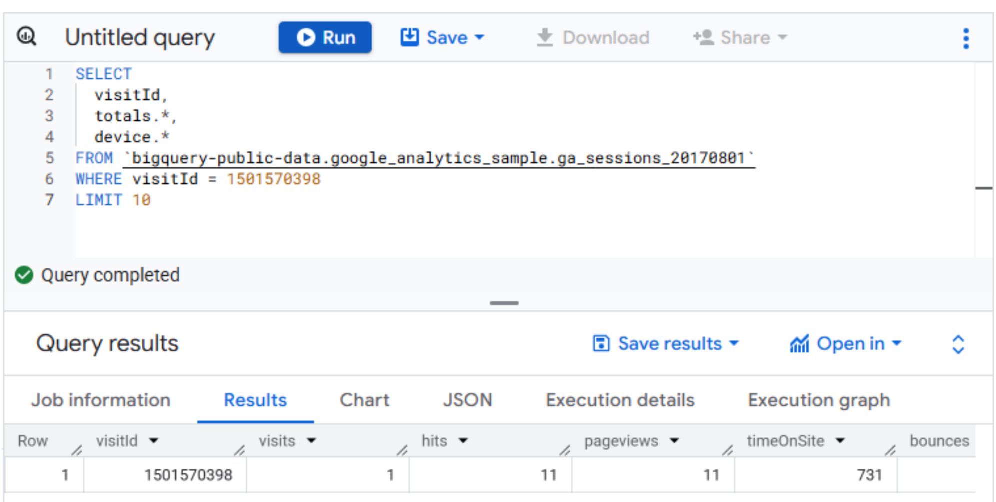
   The bigquery-public-data project is listed in the Explorer section.

3. Open **bigquery-public-data**.
4. Find and open **google_analytics_sample** dataset.
5. Click the **ga_sessions_ (366)** table.
6. Start scrolling through the schema and answer the following question by using the find feature of your browser.





The main advantage of having 32 STRUCTs in a single table is it allows you to run queries like this one without having to do any JOINs:

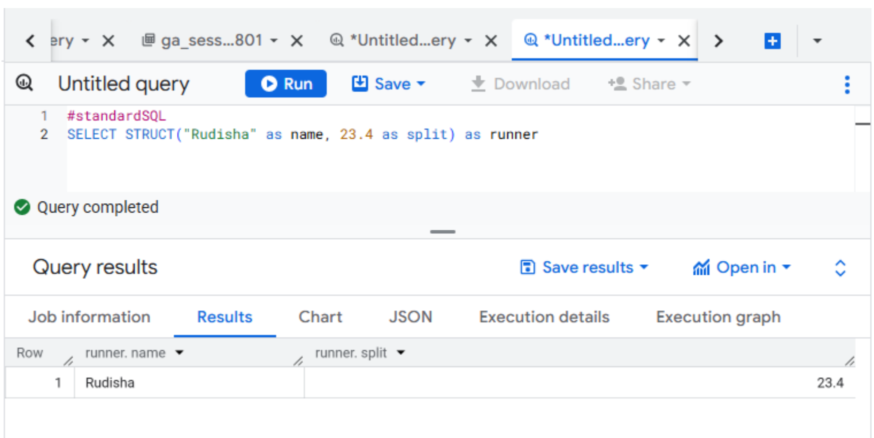


Storing your large reporting tables as STRUCTs (pre-joined "tables") and ARRAYs (deep granularity) allows you to:

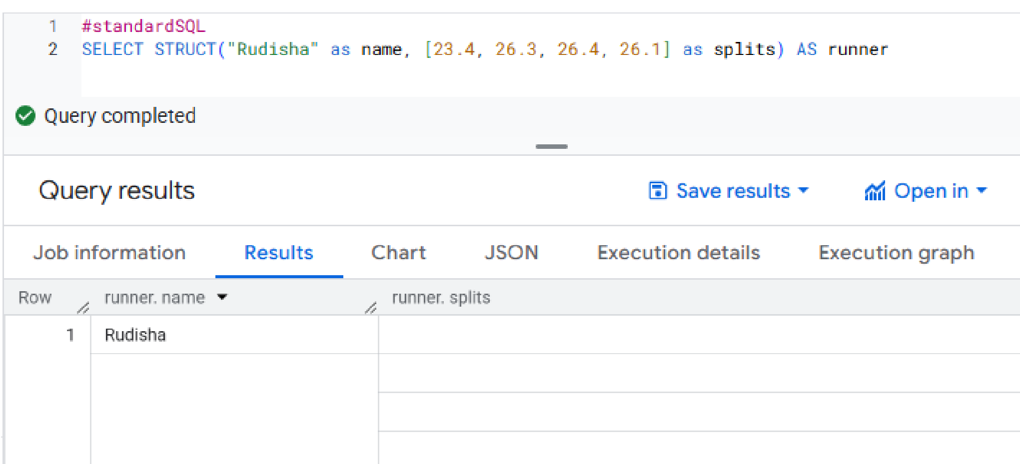
- Gain significant performance advantages by avoiding 32 table JOINs
- Get granular data from ARRAYs when you need it but not be punished if you don't (BigQuery stores each column individually on disk)
- Have all the business context in one table as opposed to worrying about JOIN keys and which tables have the data you need

Task 6. Practice with STRUCTs and arrays

1. With this query, try out the STRUCT syntax and note the different field types within the struct container:



2. Run the below query to confirm:



Practice ingesting JSON data

1. Create a new dataset titled racing.

2. Click on racing dataset and click **Create Table**.
3. **Source**: select **Google Cloud Storage** under **Create table from** dropdown.
4. **Select file from Cloud Storage bucket**: cloud-training/data-insights-course/labs/optimizing-for-performance/race_results.json
5. **File format**: JSONL (Newline delimited JSON)
6. In **Schema**, click on **Edit as text** slider and add the following:

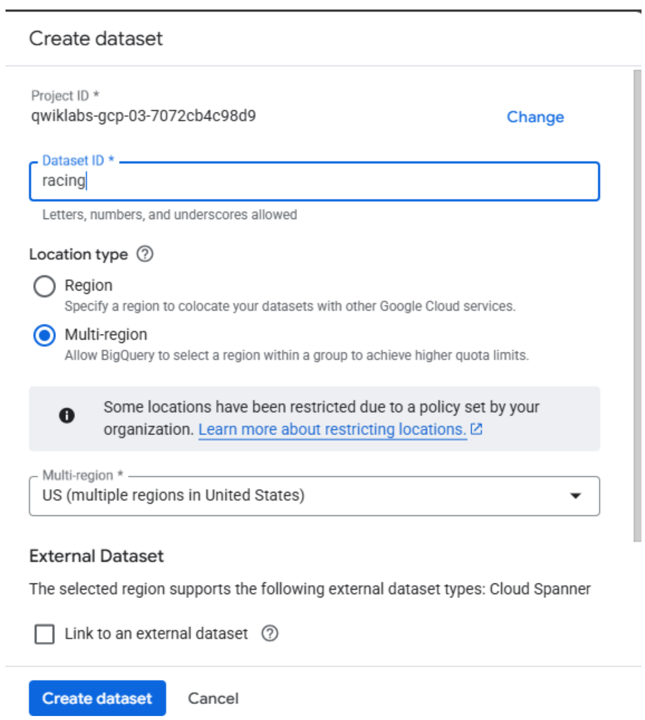
```
[
  {
    "name": "race",
    "type": "STRING",
    "mode": "NULLABLE"
  },
  {
    "name": "participants",
    "type": "RECORD",
    "mode": "REPEATED",
    "fields": [
      {
        "name": "name",
        "type": "STRING",
        "mode": "NULLABLE"
      },
      {
        "name": "splits",
        "type": "FLOAT",
        "mode": "REPEATED"
```

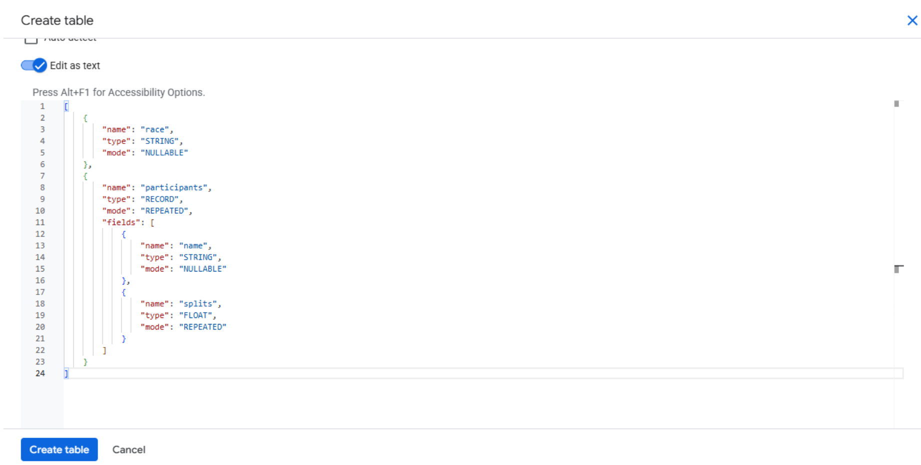
```
        }
      ]
    }
  ]
```

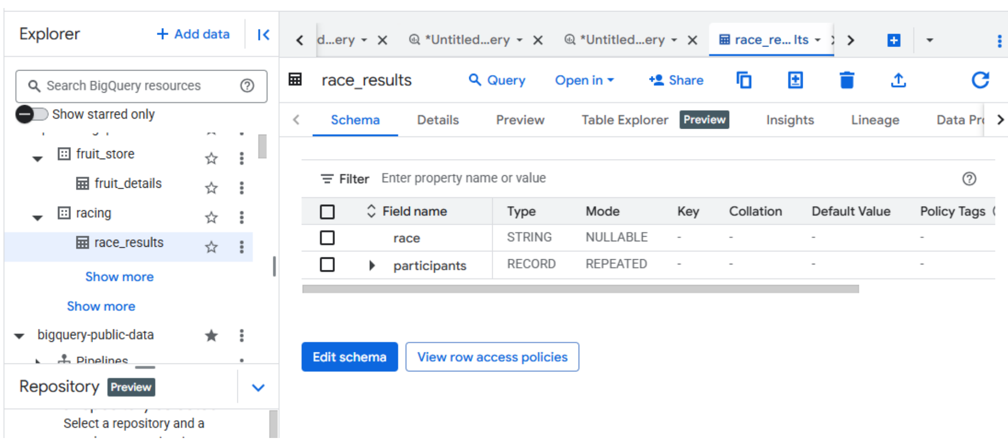
3. Call the new table race_results.
4. Click **Create table**.
5. After the load job is successful, preview the schema for the newly created table

   The **participants** field is the STRUCT because it is of type RECORD.

   The participants.splits field is an array of floats inside of the parent participants struct. It has a REPEATED Mode which indicates an array. Values of that array are called nested values since they are multiple values inside of a single field.

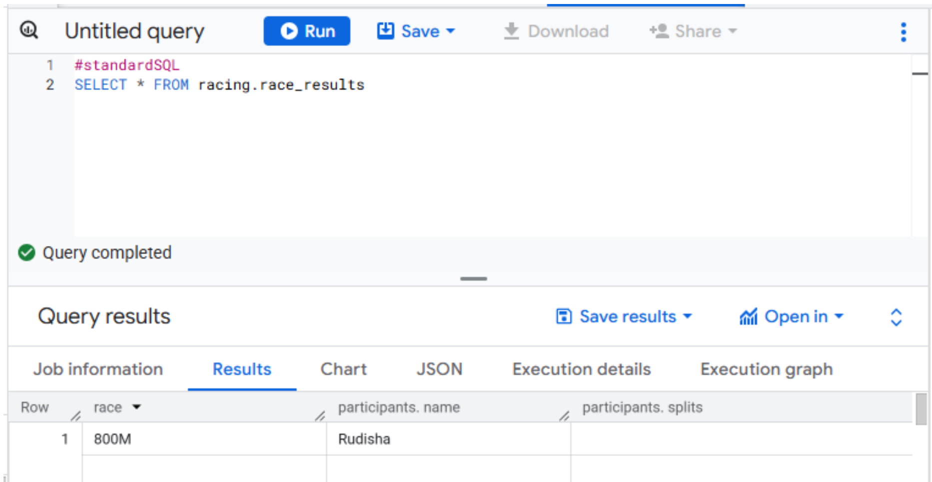




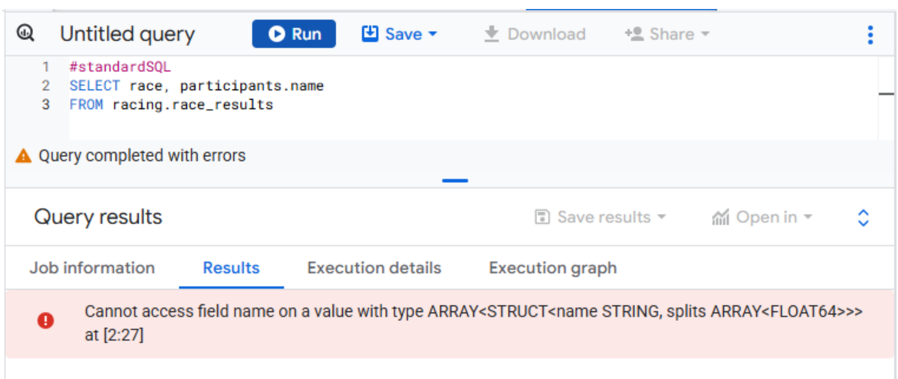


Practice querying nested and repeated fields

1. Let's see all of our racers for the 800 Meter race:



2. Run the below schema and see what happens:



Error: Cannot access field name on a value with type ARRAY<STRUCT<name STRING, splits ARRAY<FLOAT64>>>> at [2:27]

Much like forgetting to GROUP BY when you use aggregation functions, here there are two different levels of granularity. One row for the race and three rows for the participants names. So how do you change this...

Row	**race**	**participants.name**
1	800M	Rudisha
2	???	Makhloufi
3	???	Murphy

...to this:

Row	race	participants.name
1	800M	Rudisha
2	800M	Makhloufi
3	800M	Murphy

In traditional relational SQL, if you had a races table and a participants table what would you do to get information from both tables? You would JOIN them together. Here the participant STRUCT (which is conceptually very similar to a table) is already part of your races table but is not yet correlated correctly with your non-STRUCT field "race".

Can you think of what two word SQL command you would use to correlate the 800M race with each of the racers in the first table?

Answer: CROSS JOIN

3. Now try running this:

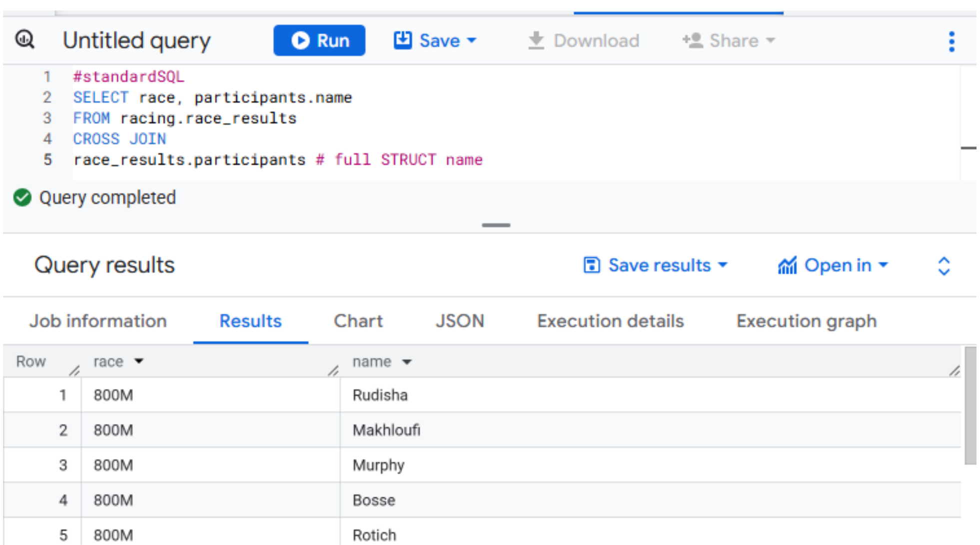
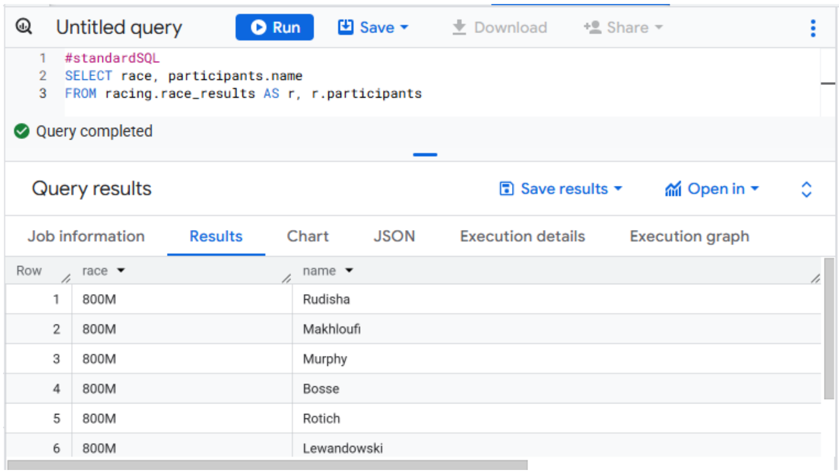


Table name "participants" missing dataset while no default dataset is set in the request.

Even though the participants STRUCT is like a table, it is still technically a field in the racing.race_results table.

4. Add the dataset name to the query:

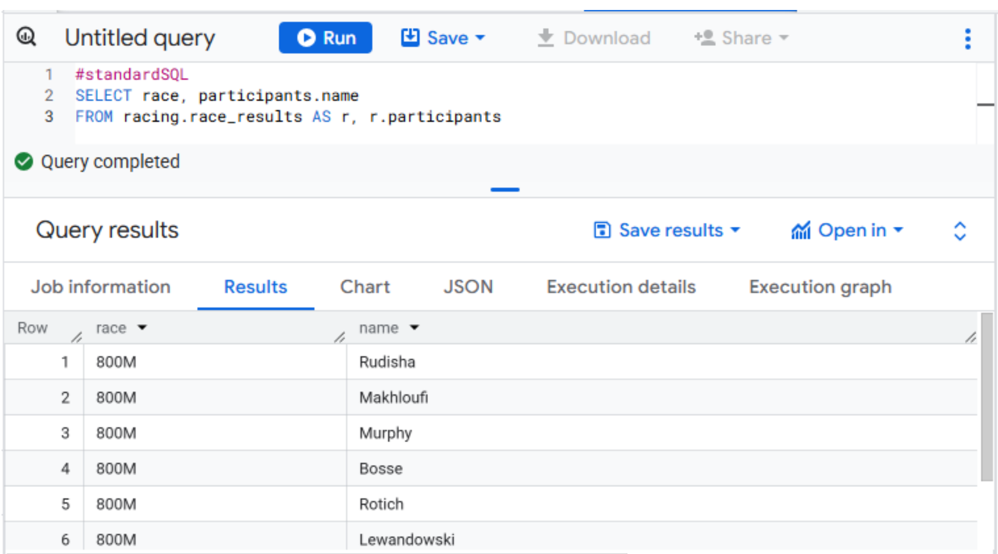


5. And click **Run**.

6. You can simplify the last query by:

- Adding an alias for the original table
- Replacing the words "CROSS JOIN" with a comma (a comma implicitly cross joins)

This will give you the same query result

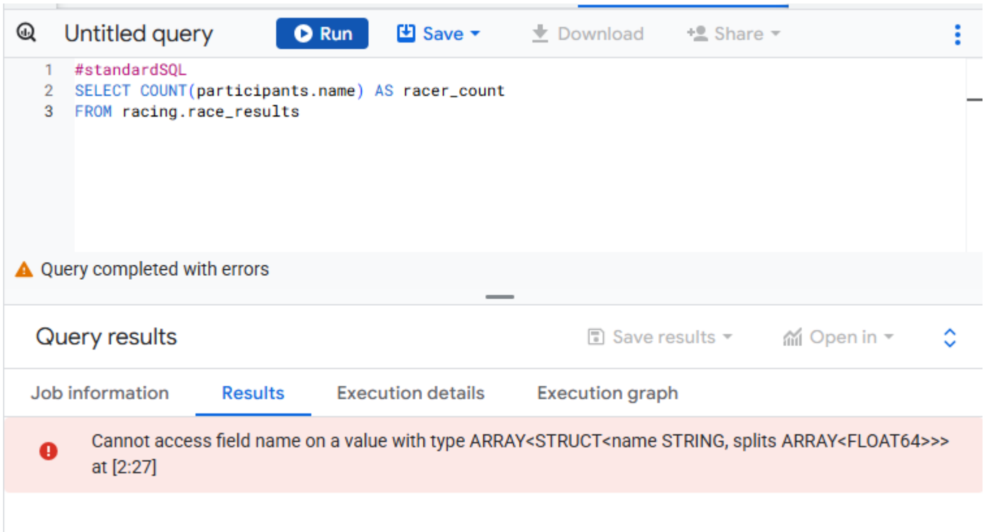


Task 7. Lab question: STRUCT()

Answer the below questions using the racing.race_results table you created previously.

Task: Write a query to COUNT how many racers were there in total.

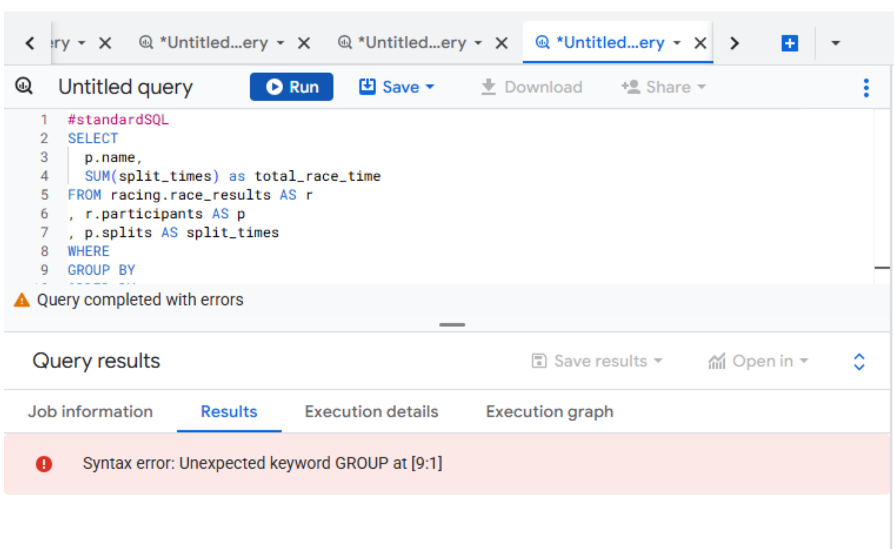
- To start, use the below partially written query:

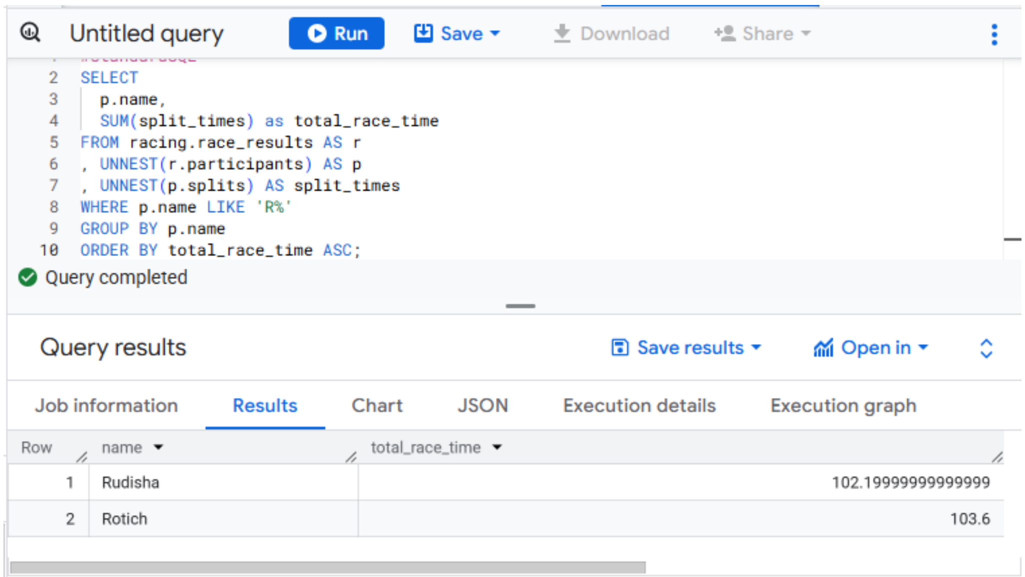


Task 8. Lab question: Unpacking arrays with UNNEST( )

Write a query that will list the total race time for racers whose names begin with R. Order the results with the fastest total time first. Use the UNNEST() operator and start with the partially written query below.

- Complete the query:





Task 9. Filter within array values

You happened to see that the fastest lap time recorded for the 800 M race was 23.2 seconds, but you did not see which runner ran that particular lap. Create a query that returns that result.

- Complete the partially written query: