

[End Lab](#)

00:34:33

Caution: When you are in the console, do not deviate from the lab instructions. Doing so may cause your account to be blocked.

[Learn more.](#)[Open Google Console](#)

Username

student-02-5a7887f0834de



Password

6ho9kG8jtn9z



GCP Project ID

qwiklabs-gcp-00-93900241



Working with JSON and Array data in BigQuery 2.5

1 hour Free ★★★★☆

[Overview](#)

25/5

[Set up and Requirements](#)[Create a new dataset to store our tables](#)[Practice working with Arrays in SQL](#)[Creating your own arrays with ARRAY_AGG\(\)](#)[Querying datasets that already have ARRAYS](#)[Introduction to STRUCTs](#)[Practice with STRUCTs and ARRAYS](#)[Lab Question: STRUCT\(\)](#)[Lab Question: Unpacking ARRAYS with UNNEST\(\)](#)[Lab Question: Filtering within ARRAY values](#)[Congratulations!](#)

Overview

[BigQuery](#) is Google's fully managed, NoOps, low cost analytics database. With BigQuery you can query terabytes and terabytes of data without having any infrastructure to manage or needing a database administrator. BigQuery uses SQL and can take advantage of the pay-as-you-go model. BigQuery allows you to focus on analyzing data to find meaningful insights.

This lab is an in-depth walkthrough of working with semi-structured data (ingesting JSON, Array data types) inside of BigQuery. Denormalizing your schema into a single table with nested and repeated fields can yield performance improvements, but the SQL syntax for working with array data can be tricky. You will practice loading, querying, troubleshooting, and unnesting various semi-structured datasets.

Objectives

In this lab, you learn about the following:

- Loading semi-structured JSON into BigQuery
- Creating and querying arrays
- Creating and querying structs
- Querying nested and repeated fields

Set up and Requirements

For each lab, you get a new GCP project and set of resources for a fixed time at no cost.

1. Make sure you signed into Qwiklabs using an **incognito window**.
2. Note the lab's access time (for example, **02:00:00**) and make sure you can finish in that time block.

There is no pause feature. You can restart if needed, but you have to start at the beginning.

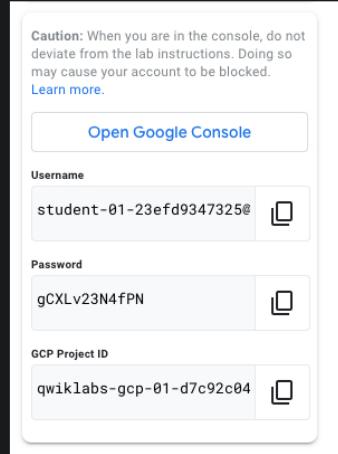
3. When ready, click **START LAB**.

4. Note your lab credentials. You will use them to sign in to Cloud Platform Console.

Caution: When you are in the console, do not deviate from the lab instructions. Doing so may cause your account to be blocked.
[Learn more.](#)

Open Google Console

Username
student-01-23efd9347325@
Password
gCXLv23N4fPN
GCP Project ID
qwiklabs-gcp-01-d7c92c04



5. Click **Open Google Console**.

6. Click **Use another account** and copy/paste credentials for **this** lab into the prompts.

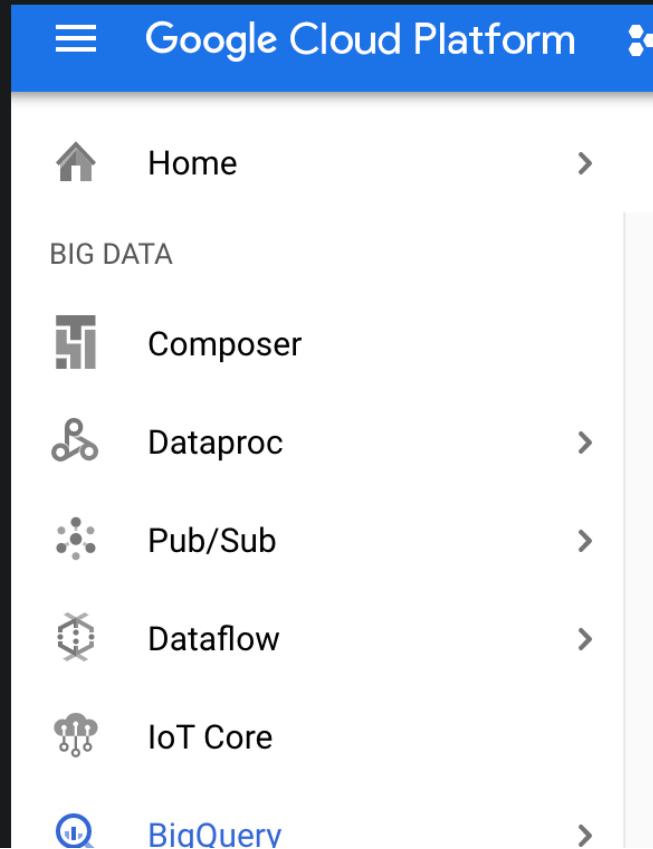
If you use other credentials, you'll get errors or incur charges.

7. Accept the terms and skip the recovery resource page.

Do not click **End Lab** unless you are finished with the lab or want to restart it. This clears your work and removes the project.

Open BigQuery Console

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





Looker



Data Catalog

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

Click **Done**.

Create a new dataset to store our tables

1. To create a dataset, click on the **View actions** icon next to your Project ID and then select **Create dataset**.

The screenshot shows the Google Cloud Platform Explorer interface. At the top, there's a search bar with placeholder text "Type to search" and a help icon. Below the search bar, it says "Viewing pinned projects." A list of pinned projects is shown, with the first item being "qwiklabs-gcp-01-fa810f3c...". To the right of this list is a vertical toolbar with a "⋮" icon. A context menu is open over the project name, with the "Create dataset" option highlighted by a red box. The "Open" option is also visible in the menu.

2. Name the new dataset **fruit_store**. Leave the other options at their default values (Data Location, Default Expiration). Click **Create dataset**.

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 to split 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.

What are some potential issues if you stored all your data in one giant table?

- The table row size could be too large for traditional reporting databases
- Any changes to a value (like customer email) could impact many other rows (like all their orders)
- Data at differing levels of granularity could lead to reporting issues because less granular fields would be repeated.
- All of the above

Submit

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. Try it yourself. Enter the following in the BigQuery Query Editor:

```
#standardSQL
SELECT
```



```
[ 'raspberry', 'blackberry', 'strawberry', 'cherry' ] AS  
fruit_array
```

2. Click **Run**.

3. Now try executing this one:

```
#standardSQL  
SELECT  
[ 'raspberry', 'blackberry', 'strawberry', 'cherry', 1234567 ] AS  
fruit_array
```

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]
```

Why did we get this error?

- Data in an array cannot exceed 4 elements
- Data in an array [] must all be the same type
- Data in an array must only be strings

Submit

Arrays can only share one data type (all strings, all numbers).

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

```
#standardSQL  
SELECT person, fruit_array, total_cost FROM `data-to-  
insights.advanced.fruit_store`;
```

5. Click **Run**.

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

The screenshot shows the BigQuery Query results interface. At the top, there are tabs for 'Job information', 'Results', 'JSON' (which is highlighted with a red box), and 'Execution details'. Below the tabs, the results are displayed as a JSON array. The first element of the array is a person object with properties: 'person' (containing 'sally'), 'fruit_array' (containing 'raspberry', 'blackberry', 'strawberry', 'cherry'), and 'total_cost' (containing '10.99').

```
[  
  {  
    "person": [  
      "sally"  
    ],  
    "fruit_array": [  
      "raspberry",  
      "blackberry",  
      "strawberry",  
      "cherry"  
    ],  
    "total_cost": [  
      "10.99"  
    ]  
  }  
]
```

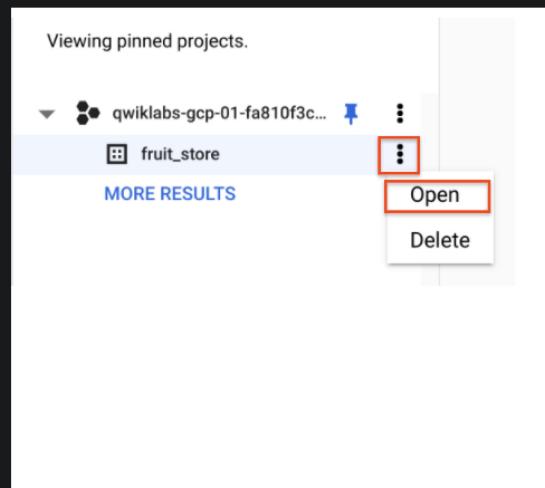
Loading semi-structured JSON into BigQuery

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

1. Create a new table in the `fruit_store` dataset.

2. To create a table, click on the **View actions** icon next to the `fruit_store` dataset and

select Open.



3. Now, click + CREATE TABLE.

4. Add the following details for the table:

- **Source:** Choose **Google Cloud Storage** in the **Create table from** dropdown.
- **Select file from GCS bucket** (type or paste the following): `data-insights-course/labs/optimizing-for-performance/shopping_cart.json`
- **File format:** JSONL (Newline delimited JSON) {This will be auto-populated}
- **Schema:** Check **Auto detect** (Schema and input parameters).

5. Call the new table "fruit_details".

6. Click **Create table**.

In the schema, note that `fruit_array` is marked as REPEATED which means it's an array.

Recap

- BigQuery natively supports arrays
- Array values must share a data type
- Arrays are called REPEATED fields in BigQuery

Click *Check my progress* to verify the objective.

Create a new dataset and load JSON data into the table



[Check my progress](#)

Assessment Completed!

Creating 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

```
SELECT
    fullVisitorId,
    date,
    v2ProductName,
    pageTitle
    FROM `data-to-insights.ecommerce.all_sessions`
WHERE visitId = 1501570398
ORDER BY date
```

2. Click Run and view the results

How many rows are returned?

- 70
- 111
- 2
- 100

Submit

3. Now, we will use the `ARRAY_AGG()` function to aggregate our string values into an array. Copy and paste the below query to explore this public dataset:

```
SELECT
    fullVisitorId,
    date,
    ARRAY_AGG(v2ProductName) AS products_viewed,
    ARRAY_AGG(pageTitle) AS pages_viewed
    FROM `data-to-insights.ecommerce.all_sessions`
WHERE visitId = 1501570398
GROUP BY fullVisitorId, date
ORDER BY date
```

4. Click Run and view the results

How many rows are returned?

- 100 - one for each day
- 2 - one for each day
- 70 - one for each day
- 63 - one for each day

Submit

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

```
SELECT
    fullVisitorId,
    date,
    ARRAY_AGG(v2ProductName) AS products_viewed,
    ARRAY_LENGTH(ARRAY_AGG(v2ProductName)) AS num_products_viewed,
    ARRAY_AGG(pageTitle) AS pages_viewed
```

```
ARRAY_AGG(pageTitle) AS pages_viewed,  
ARRAY_LENGTH(ARRAY_AGG(pageTitle)) AS num_pages_viewed  
FROM `data-to-insights.ecommerce.all_sessions`  
WHERE visitId = 1501570398  
GROUP BY fullVisitorId, date  
ORDER BY date
```

How many pages were visited by this user on 20170801?

- 70
- 109
- 101
- 8

Submit

6. Next, let's deduplicate the pages and products so we can see how many unique products were viewed. We'll simply add DISTINCT to our ARRAY_AGG()

```
SELECT  
    fullVisitorId,  
    date,  
    ARRAY_AGG(DISTINCT v2ProductName) AS products_viewed,  
    ARRAY_LENGTH(ARRAY_AGG(DISTINCT v2ProductName)) AS  
distinct_products_viewed,  
    ARRAY_AGG(DISTINCT pageTitle) AS pages_viewed,  
    ARRAY_LENGTH(ARRAY_AGG(DISTINCT pageTitle)) AS  
distinct_pages_viewed  
    FROM `data-to-insights.ecommerce.all_sessions`  
WHERE visitId = 1501570398  
GROUP BY fullVisitorId, date  
ORDER BY date
```

How many DISTINCT pages were visited by this user on 20170801?

- 101
- 8
- 70
- 109

Submit

Recap

You can do some pretty useful things with arrays like:

- finding the number of elements with `ARRAY_LENGTH(<array>)`
- deduplicating elements with `ARRAY_AGG(DISTINCT <field>)`
- ordering elements with `ARRAY_AGG(<field> ORDER BY <field>)`
- limiting `ARRAY_AGG(<field> LIMIT 5)`

Click *Check my progress* to verify the objective.

Creating arrays with `ARRAY_AGG()`

[Check my progress](#)

Assessment Completed!

Querying datasets that already have ARRAYS

The BigQuery Public Dataset for Google Analytics `bigrquery-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)

```
SELECT
  *
FROM `bigrquery-public-
data.google_analytics_sample.ga_sessions_20170801`
WHERE visitId = 1501570398
```

2. **Run** the query.
3. **Scroll right** in the results until you see the `hits.product.v2ProductName` field (we will discuss the multiple field aliases shortly).

You will notice a lot of seemingly 'empty' cells in the results as you scroll. These cells are grayed out and not marked as null. Why do you think that is?

- BigQuery is still in the process of loading values for the grayed out cells
- The grayed out cells are visual placeholders to make it possible to show each item in an array type column on its own row within the context of a row in the result set
- The entire dataset has no data values for the grayed out cells

Submit

4. The amount of fields available in the Google Analytics schema can be overwhelming for our analysis. Let's try to query just the visit and page name fields like we did before.

```
SELECT
  visitId,
  hits.page.pageTitle
FROM `bigrquery-public-
data.google_analytics_sample.ga_sessions_20170801`
WHERE visitId = 1501570398
```

You will get an error: Cannot access field `product` on a value with type `ARRAY> at [5:8]`

Before we can query REPEATED fields (arrays) normally, you must first break the arrays back into rows.

For example, the array for `hits.page.pageTitle` is stored currently as a single row like:

```
[ 'homepage', 'product page', 'checkout' ]
```

and we need it to be

```
[ 'homepage',
  'product page' ]
```

```
product_page',  
'checkout']
```

5. How do we do that with SQL? Answer: Use the UNNEST() function on your array field:

```
SELECT DISTINCT  
    visitId,  
    h.page.pageTitle  
FROM `bigquery-public-  
data.google_analytics_sample.ga_sessions_20170801`,  
UNNEST(hits) AS h  
WHERE visitId = 1501570398  
LIMIT 10
```



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)

Click *Check my progress* to verify the objective.

Querying datasets that already have ARRAYS



Check my progress

Assessment Completed!

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
- its own alias

Sounds just like a table right?

Let's explore a dataset with STRUCTs

1. Under **Explorer** find the **bigquery-public-data** dataset (if it's not present already, use this [link](#) to pin the dataset)
2. Click **bigquery-public-data**
3. Find and open **google_analytics_sample**
4. Click the **ga_sessions** table
5. Start scrolling through the schema and answer the following question by using the find feature of your browser (i.e. CTRL + F)

In a BigQuery schema, a STRUCT field is noted as a RECORD Type.
Search for RECORD in the Google Analytics schema. How many

STRUCTs are present in this dataset?

1

5

11

32

Submit

What are the names of some of the STRUCT (RECORD Type) fields?

Totals

TrafficSource

trafficSource.adwordsClickInfo

device

All of the above

Submit

How can both TrafficSource and trafficSource.adwordsClickInfo both be STRUCTs?

A STRUCT can have another STRUCT as one of its fields (you can nest STRUCTs)

They are not STRUCTs

Because they are all ARRAYS

This is an invalid data type

Submit

In a BigQuery schema, an ARRAY field is noted as a REPEATED Mode. Search for REPEATED in the Google Analytics schema. How many ARRAYS are present in this dataset?

1

5

11

32

Submit

6. As you can imagine, there is an incredible amount of website session data stored for a modern ecommerce website. 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:

```
SELECT  
  visitId,  
  totals.*,  
  device.*  
FROM `bigquery-public-
```



```
data.google_analytics_sample.ga_sessions_20170801`  
WHERE visitId = 1501570398  
LIMIT 10
```

Note: The `.*` syntax tells BigQuery to return all fields for that STRUCT (much like it would if `totals.*` was a separate table we joined against)

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

Click *Check my progress* to verify the objective.

Explore a dataset with STRUCTs

 Check my progress

Assessment Completed!

Practice with STRUCTs and ARRAYS

The next dataset will be lap times of runners around the track. Each lap will be called a "split".



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

```
#standardSQL  
SELECT STRUCT("Rudisha" as name, 23.4 as split) as runner
```



Row	runner.name	runner.split
1	Rudisha	23.4

What do you notice about the field aliases? Since there are fields nested within the struct (name and split are a subset of runner) you end up with a dot notation.

What if the runner has multiple split times for a single race (like time per lap)?

How could you have multiple split times within a single record? Hint: the splits all have the same numeric datatype.

 Store each split time in a separate STRING field with STRING_AGG()



- Store each split time in a separate table called race_splits
- Store each split time as an element in an ARRAY of splits
- Use a SQL UNION to join the race and split details

Submit

2. With an array of course! Run the below query to confirm:

```
#standardSQL
SELECT STRUCT("Rudisha" as name, [23.4, 26.3, 26.4, 26.1] as
splits) AS runner
```

Row	runner.name	runner.splits
1	Rudisha	23.4
		26.3
		26.4
		26.1

To recap:

- Structs are containers that can have multiple field names and data types nested inside.
- An arrays can be one of the field types inside of a Struct (as shown above with the splits field).

Practice ingesting JSON data

1. Create a new dataset titled **racing**.
2. Create a new table titled **race_results**.
3. Ingest this Google Cloud Storage JSON file:

```
data-insights-course/labs/optimizing-for-
performance/race_results.json
```

- **Source:** Google Cloud Storage under **Create table from** dropdown.
- **Select file from GCS bucket:** data-insights-course/labs/optimizing-for-performance/race_results.json
- **File format:** JSONL (Newline delimited JSON)
- In **Schema**, move the **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": "age",
        "type": "INT64",
        "mode": "NULLABLE"
      }
    ]
}
```

```

        "name": "splits",
        "type": "FLOAT",
        "mode": "REPEATED"
    }
]
}
]
```

4. Click **Create table**.

5. After the load job is successful, preview the schema for the newly created table:

Field name	Type	Mode	Policy Tags	Description
race	STRING	NULLABLE		
participants	RECORD	REPEATED		
name	STRING	NULLABLE		
splits	FLOAT	REPEATED		

Which field is the STRUCT? How do you know?

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

Which field is the ARRAY?

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.

```
#standardSQL
SELECT * FROM racing.race_results
```

How many rows were returned?

Answer: 1

Query complete (1.236 sec elapsed, 336 B processed)

Job information		Results	JSON	Execution details
Row	race	participants.name	participants.splits	
1	800M	Rudisha	23.4 26.3 26.4 26.1	
		Makhloufi	24.5	
			25.4	
			26.6	
			26.1	
		Murphy	23.9	

		26.0
		27.0
		26.0
Bosse		23.6

2. What if you wanted to list the name of each runner and the type of race?

Run the below schema and see what happens:

```
#standardSQL
SELECT race, participants.name
FROM racing.race_results
```

Error: Cannot access field name on a value with type ARRAY\<STRUCT\<name STRING, splits ARRAY\<FLOAT64\>>> at [1:21]

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. Great! Now try running this:

```
#standardSQL
SELECT race, participants.name
FROM racing.race_results
CROSS JOIN
participants # this is the STRUCT (it's like a table within a
table)
```

Error: Table name "participants" cannot be resolved: dataset name is missing.

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:

```
#standardSQL
SELECT race, participants.name
FROM racing.race_results
CROSS JOIN
```

```
race_results.participants # full STRUCT name
```

5. Click **Run**.

Wow! You've successfully listed all of the racers for each race!

Row	race	name
1	800M	Rudisha
2	800M	Makhloifi
3	800M	Murphy
4	800M	Bosse
5	800M	Rotich
6	800M	Lewandowski
7	800M	Kipketer
8	800M	Berian

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:

```
#standardSQL
SELECT race, participants.name
FROM racing.race_results AS r, r.participants
```



If you have more than one race type (800M, 100M, 200M), wouldn't a CROSS JOIN just associate every racer name with every possible race like a cartesian product?

Answer: No. This is a *correlated* cross join which only unpacks the elements associated with a single row. For a greater discussion, see [working with ARRAYS and STRUCTs](#)

Click *Check my progress* to verify the objective.

Practice with STRUCTs and ARRAYS

Check my progress

Assessment Completed!

Recap of STRUCTs:

- A SQL [STRUCT](#) is simply a container of other data fields which can be of different data types. The word struct means data structure. Recall the example from earlier:
- `__STRUCT(__ "Rudisha" as name, [23.4, 26.3, 26.4, 26.1] as splits __) __ AS runner`
- STRUCTs are given an alias (like runner above) and can conceptually be thought of as a table inside of your main table.
- STRUCTs (and ARRAYS) must be unpacked before you can operate over their elements. Wrap an UNNEST() around the name of the struct itself or the struct field that is an array in order to unpack and flatten it.

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:

```
#standardSQL
SELECT COUNT(participants.name) AS racer_count
FROM racing.race_results
```



Hint: Remember you will need to cross join in your struct name as an additional data source after the FROM.

Possible Solution:

```
#standardSQL
SELECT COUNT(p.name) AS racer_count
FROM racing.race_results AS r, UNNEST(r.participants) AS p
```



Row	racer_count
1	8

Answer: There were 8 racers who ran the race.

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:

```
#standardSQL
SELECT
  p.name,
  SUM(split_times) as total_race_time
FROM racing.race_results AS r
, r.participants AS p
, p.splits AS split_times
WHERE
GROUP BY
ORDER BY
;
```



Hint:

- You will need to unpack both the struct and the array within the struct as data sources after your FROM clause
- Be sure to use aliases where appropriate

Possible Solution:

```
#standardSQL
SELECT
  p.name,
  SUM(split_times) as total_race_time
FROM racing.race_results AS r
, UNNEST(r.participants) AS p
, UNNEST(p.splits) AS split_times
WHERE p.name LIKE 'R%'
GROUP BY
```



```
GROUP BY p.name  
ORDER BY total_race_time ASC;
```

Row	name	total_race_time
1	Rudisha	102.19999999999999
2	Rotich	103.6

Lab Question: Filtering 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.

Task: Complete the partially written query:

```
#standardSQL  
SELECT  
    p.name,  
    split_time  
FROM racing.race_results AS r  
, r.participants AS p  
, p.splits AS split_time  
WHERE split_time = ;
```

Possible Solution:

```
#standardSQL  
SELECT  
    p.name,  
    split_time  
FROM racing.race_results AS r  
, UNNEST(r.participants) AS p  
, UNNEST(p.splits) AS split_time  
WHERE split_time = 23.2;
```

Row	name	split_time
1	Kipketer	23.2

Congratulations!

You've successfully ingested JSON datasets, created ARRAYS and STRUCTs, and unnested semi-structured data for insights.

Next Steps / Learn More

- For additional reading, refer to [Working with Arrays](#).

Copyright 2021 Google LLC All rights reserved. Google and the Google logo are trademarks of Google LLC. All other company and product names may be trademarks of the respective companies with which they are associated.