

# Working with JSON, Arrays, and Structs in BigQuery

GSP416



# 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.

In this lab you will work in-depth 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.

## Setup and Requirements

### Before you click the Start Lab button

Read these instructions. Labs are timed and you cannot pause them. The timer, which starts when you click **Start Lab**, shows how long Google Cloud resources will be made available to you.

This Qwiklabs hands-on lab lets you do the lab activities yourself in a real cloud environment, not in a simulation or demo environment. It does so by giving you new, temporary credentials that you use to sign in and access Google Cloud for the duration of the lab.

### What you need

To complete this lab, you need:

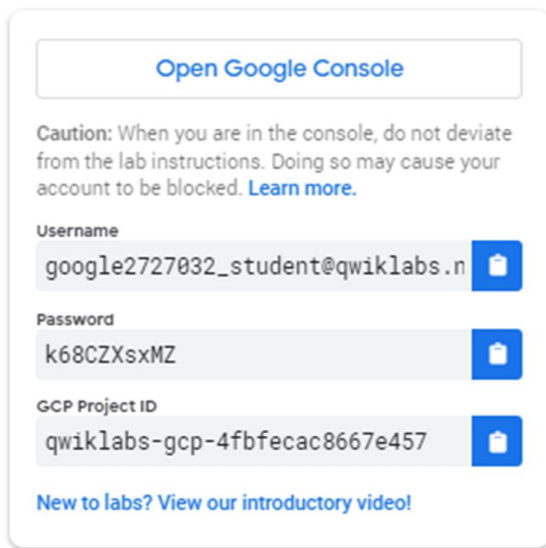
- Access to a standard internet browser (Chrome browser recommended).
- Time to complete the lab.

**Note:** If you already have your own personal Google Cloud account or project, do not use it for this lab.

**Note:** If you are using a Pixelbook, open an Incognito window to run this lab.


### How to start your lab and sign in to the Google Cloud Console


1. Click the **Start Lab** button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is a panel populated with the temporary credentials that you must use for this lab.




Open Google 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.](#)

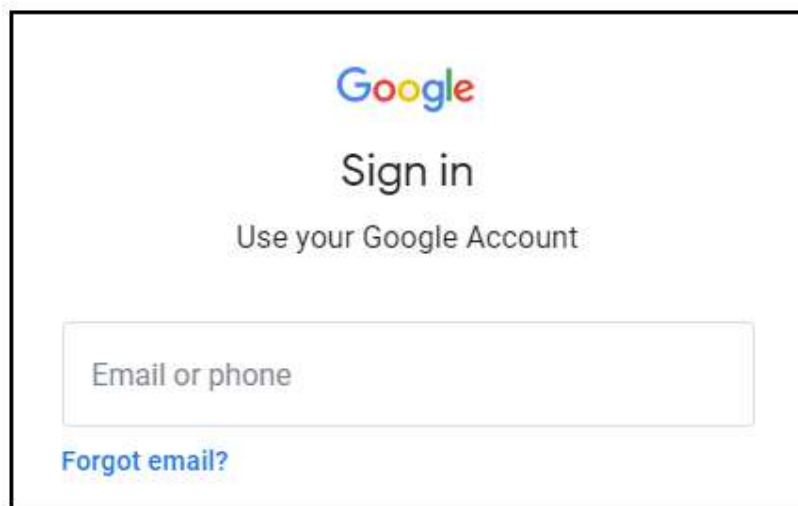
Username  
google2727032\_student@qwiklabs.n 

Password  
k68CZXsxMZ 

GCP Project ID  
qwiklabs-gcp-4fbfecac8667e457 

[New to labs? View our introductory video!](#)

2. Copy the username, and then click **Open Google Console**. The lab spins up resources, and then opens another tab that shows the **Sign in** page.



Google

Sign in

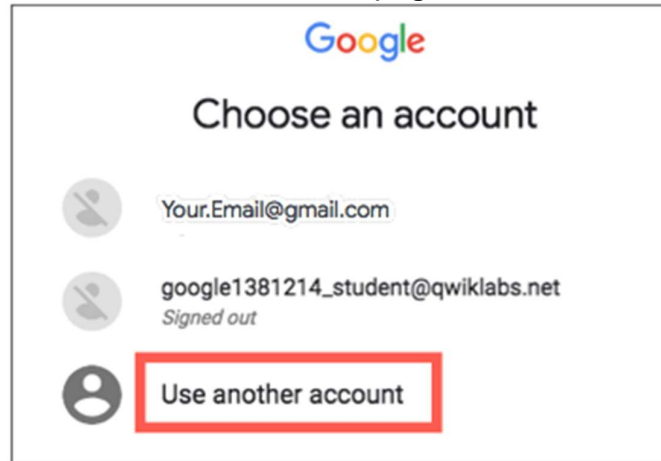
Use your Google Account

Email or phone

[Forgot email?](#)

**Tip:** Open the tabs in separate windows, side-by-side.

If you see the **Choose an account** page, click **Use Another**



**Account.**

3. In the **Sign in** page, paste the username that you copied from the Connection Details panel. Then copy and paste the password.

**Important:** You must use the credentials from the Connection Details panel. Do not use your Qwiklabs credentials. If you have your own Google Cloud account, do not use it for this lab (avoids incurring charges).

4. Click through the subsequent pages:

- Accept the terms and conditions.
- Do not add recovery options or two-factor authentication (because this is a temporary account).
- Do not sign up for free trials.

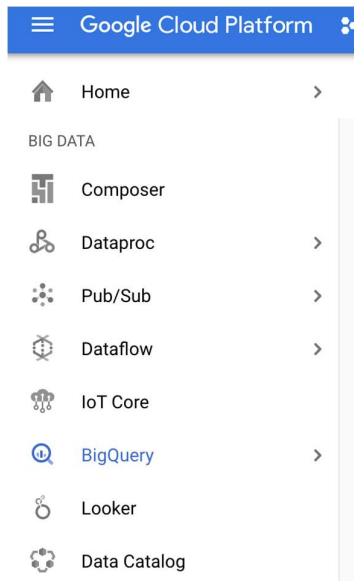
After a few moments, the Cloud Console opens in this tab.

**Note:** You can view the menu with a list of Google Cloud Products and Services by clicking the **Navigation menu** at the top-left.



# Open BigQuery Console

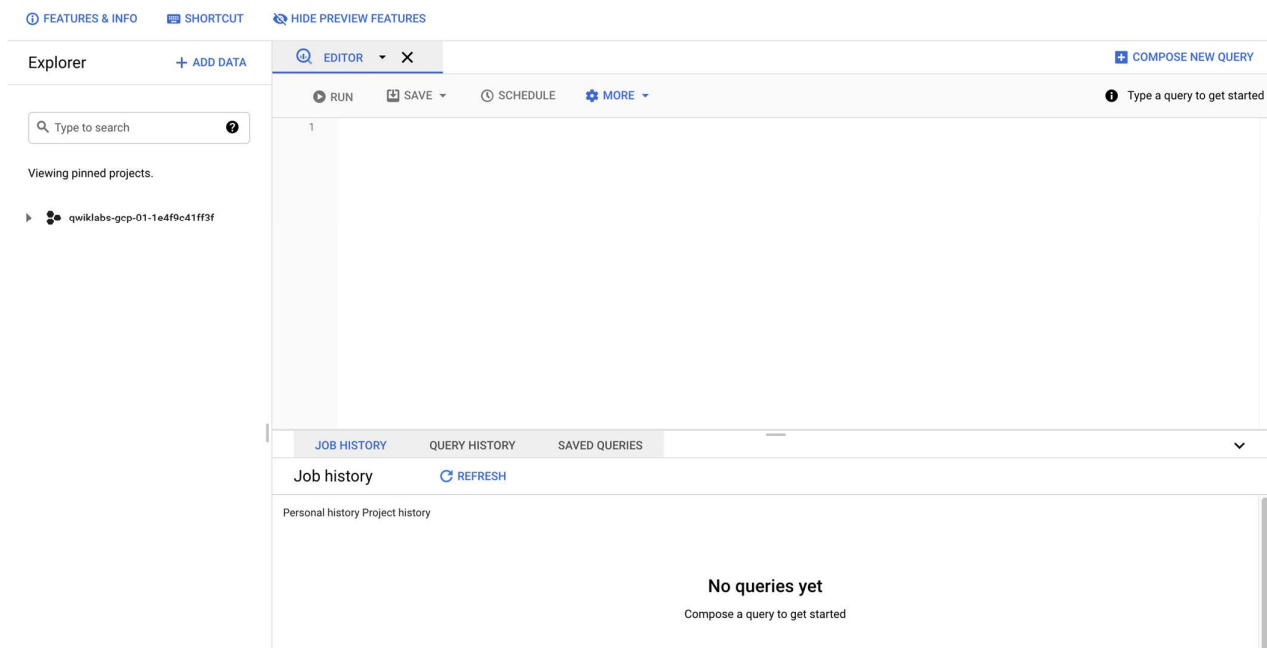
In the Google Cloud Console, select **Navigation menu** > **BigQuery**:



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.

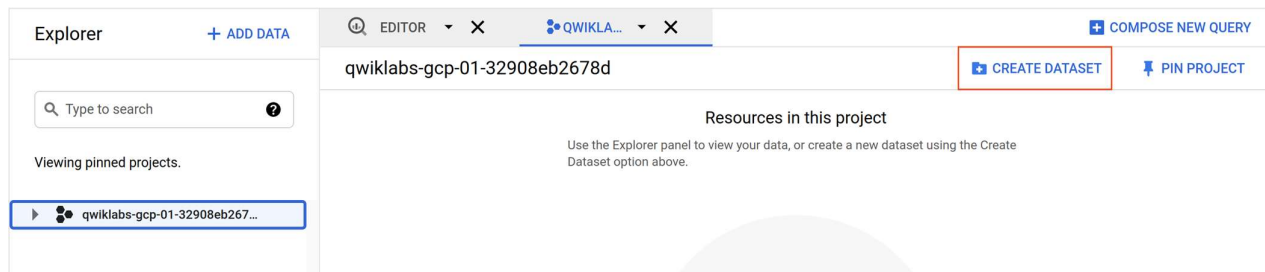
Click **Done**.

The BigQuery console opens.



# Create a new dataset to store the tables

In your BigQuery, click on your project name and then **Create Dataset**.



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.~~

check 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)

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

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

Click **Run**.

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 you get this error?



Data in an array cannot exceed 4 elements



Data in an array must only be strings  
checkData in an array [ ] must all be the same type  
Submit

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

Here's the final table to query against:

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

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

Query complete (0.952 sec elapsed, 92 B processed)

Job information   Results   **JSON**   Execution details

```
1  [
2    {
3      "person": [
4        "sally"
5      ],
6      "fruit_array": [
7        "raspberry",
8        "blackberry",
9        "strawberry",
10       "cherry"
11      ],
12      "total_cost": [
13        "10.99"
14      ]
15    },
16  ]
```



# 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. Click on `fruit_store` dataset and click Create table.

**Note:** You may have to widen your browser window to see the Create table option.  
Add the following details for the table:

- **Source:** Choose **Cloud Storage** in the **Create table from** dropdown.
- **Select file from Cloud Storage bucket:** `gs://cloud-training/gsp416/shopping_cart.json`
- **File format:** JSONL (Newline delimited JSON)  
Call the new table `fruit_details`.

Check the checkbox of **Schema and input parameters**.

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 table to store our data

Check my progress

# Creating your own arrays with ARRAY\_AGG()

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

**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
```

Click **Run** and view the results

How many rows are returned?

check111

☐

2

☐

100

☐

70

Submit

Now, 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
```

Click **Run** and view the results

How many rows are returned?

check2 - one for each day

☐

70 - one for each day

☐

63 - one for each day

☐

100 - one for each day

Submit

Next, 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_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?



101



8

check109



70

Submit

Next, deduplicate the pages and products so you can see how many unique products were viewed by adding `DISTINCT` to `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?

close101

close70

check8



109

Submit

Click *Check my progress* to verify the objective.

Execute the query to see how many unique products were viewed

Check my progress

## 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)`

# Querying datasets that already have 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.

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

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

Run the query.

**Scroll right** in the results until you see the `hits.product.v2ProductName` field (multiple field aliases are discussed shortly).

You'll note a lot of seemingly 'blank' values in the results as you scroll. Why do you think that is?  
checkThe fields that appear to be missing data are actually at a higher level of granularity than other fields



BigQuery is still loading the data for those fields



The dataset is missing data values for those fields

Submit

The amount of fields available in the Google Analytics schema can be overwhelming for analysis. Try to query just the visit and page name fields like before.

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

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

Before you 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 it needs to be

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

How do you 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.

Execute the query to use the UNNEST() on array field

Check my progress

# 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?

## Let's explore a dataset with STRUCTs

Under **Explorer** section, find the **bigquery-public-data** dataset (if it's not present already, use this [link](#) to pin the dataset)  
Click open **bigquery-public-data**

Find and open **google\_analytics\_sample**

Click the **ga\_sessions** table

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?

close4

close5

close44

check32

Submit

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

closeTotals

closeTrafficSource

closeTrafficSource.adwordsClickInfo

closedevice

checkAll of the above

Submit

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

checkA 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?

close4

close5

check11

32

Submit

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

# Practice with STRUCTs and ARRAYS

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



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.  
checkStore each split time as an element in an ARRAY of splits  
closeStore each split time in a separate STRING field with STRING\_AGG()

☐ Use a SQL UNION to join the race and split details

☐ Store each split time in a separate table called race\_splits  
Submit

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

Create a new dataset titled **racing**.

Create a new table titled **race\_results**. Click on `racing` dataset and click Create table.

**Note:** You may have to widen your browser window to see the Create table option.

Ingest this Cloud Storage JSON file:

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

- **Source:** select **Cloud Storage** under **Create table from** dropdown.
- **Select file from Cloud Storage bucket:** `gs://data-insights-course/labs/optimizing-for-performance/race_results.json`
- **File format:** JSONL (Newline delimited JSON)
- 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"
      }
    ]
  }
]
```

Click **Create table**.

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

## race\_results

[Schema](#) [Details](#) [Preview](#)

Field name	Type	Mode	Description
<b>race</b>	STRING	NULLABLE	
<b>participants</b>	RECORD	REPEATED	
participants. <b>name</b>	STRING	NULLABLE	
participants. <b>splits</b>	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.

Click *Check my progress* to verify the objective.

Create a dataset and a table to ingest JSON data

Check my progress

## Practice querying nested and repeated fields

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

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 [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

Great! Now try running this:

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

Add the dataset name to the query:

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

And click **Run**.

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

Row	race	name
1	800M	Rudisha
2	800M	Makhloufi
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](#)  
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 `racings.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.

Click *Check my progress* to verify the objective.

Execute the query to COUNT how many racers were there in total

Check my progress

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

Row	name	total_race_time
1	Rudisha	102.19999999999999
2	Rotich	103.6

Click *Check my progress* to verify the objective.

Execute the query that will list the total race time for racers whose names begin with R

Check my progress

# 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.

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

Click *Check my progress* to verify the objective.

Execute the query to see which runner ran fastest lap time

Check my progress



# Congratulations!

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



## Finish Your Quest

This self-paced lab is part of the Qwiklabs [BigQuery for Data Warehousing](#) Quest. A Quest is a series of related labs that form a learning path. Completing this Quest earns you the badge above, to recognize your achievement. You can make your badge (or badges) public and link to them in your online resume or social media account. Enroll in a Quest and get immediate completion credit if you've taken this lab. [See other available Qwiklabs Quests](#).

## Take Your Next Lab

Continue your Quest with [Creating Date-Partitioned Tables in BigQuery](#), or check out these suggestions:

- [Predict Taxi Fare with a BigQuery ML Forecasting Model](#)
- [Ingesting New datasets into BigQuery](#)

## Next Steps / Learn More

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

## Google Cloud Training & Certification

...helps you make the most of Google Cloud technologies. [Our classes](#) include technical skills and best practices to help you get up to speed quickly and continue your learning journey. We offer fundamental to advanced level training, with on-demand, live, and virtual

options to suit your busy schedule. [Certifications](#) help you validate and prove your skill and expertise in Google Cloud technologies.

Manual Last Updated: April 19, 2021

Lab Last Tested: April 19, 2021

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.