

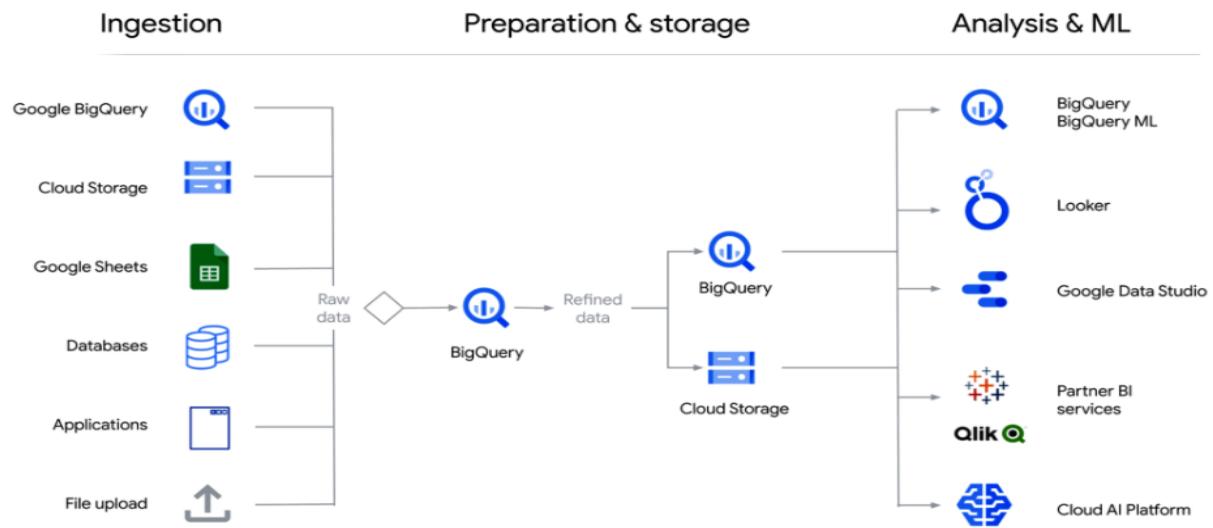
S.NO	INDEX	PAGE NO.
1.	What is Google BigQuery?	2
2.	BigQuery Architecture	3
3.	Features of Google BigQuery	4
4.	Geospatial Analysis	5
5.	BigQuery ML	18
6.	Arrays and Struct in Google BigQuery - Arrays	26
7.	Struct	36
8.	BigQuery Partitioning and Clustering – Partitioning	40
9.	Clustering	45

Google BigQuery

What is Google BigQuery?

BigQuery is a fully managed enterprise data warehouse that helps manage and analyse data with built-in features like machine learning, geospatial analysis and business intelligence. In layman terms, it is just a **BIG** database

BigQuery's serverless architecture lets us use SQL queries to answer organisation's biggest questions with zero infrastructure management. Its scalable, cost effective, distributed analysis engine lets us query terabytes in seconds and petabytes in minutes.



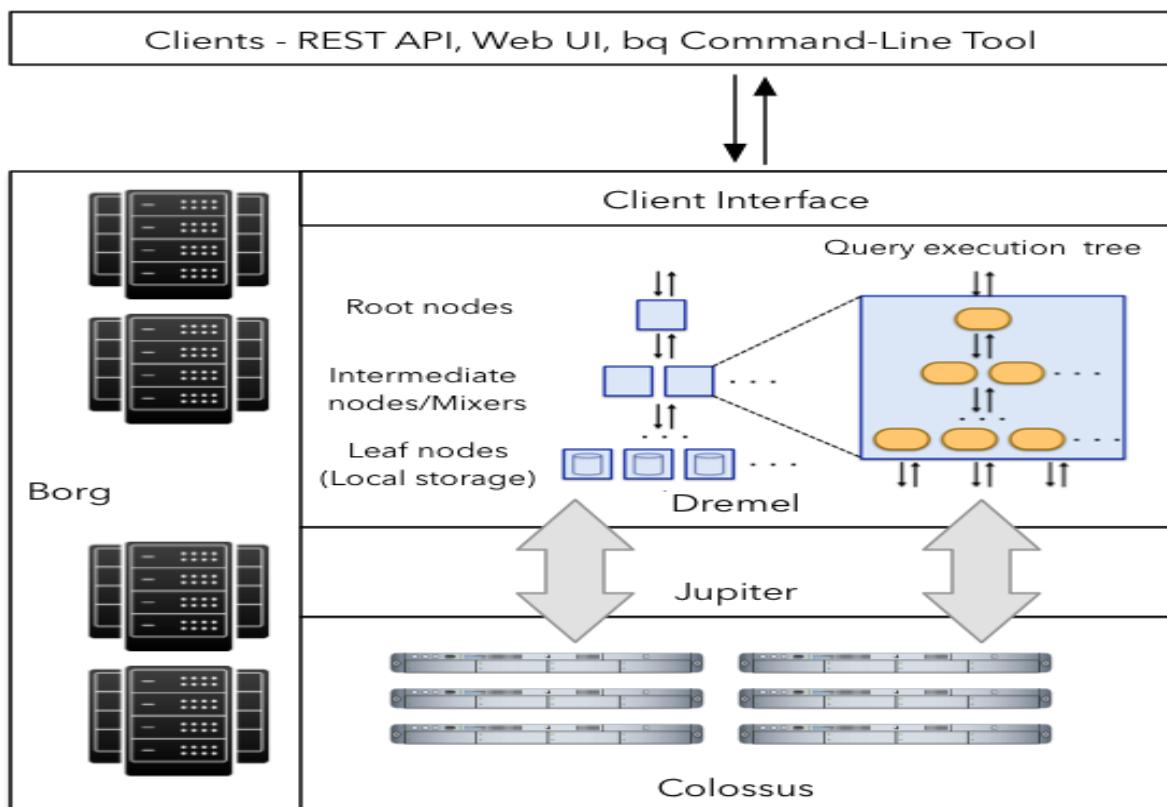
Working of BigQuery

BigQuery Architecture

Bigquery is built on Dremel technology (an interactive ad-hoc query system for analysing nested read only data). But bigquery is much more than that. Bigquery also has a serverless architecture that decouples storage and allows us to scale independently.

This structure offers both flexibility and cost control.

It is different from traditional cloud data warehouse solutions or on-premise massively parallel processing systems. This approach allows customers of any size to bring their data into the data warehouse and start analysing their data using Standard SQL without worrying about database operations and system engineering. Dremel is just an execution engine for BigQuery. In fact, BigQuery service leverages Google's innovative technologies like Borg, Colossus, Capacitor, and Jupiter. As illustrated below, a BigQuery client (typically BigQuery Web UI or bq command-line tool or REST APIs) interacts with the Dremel engine via a client interface.



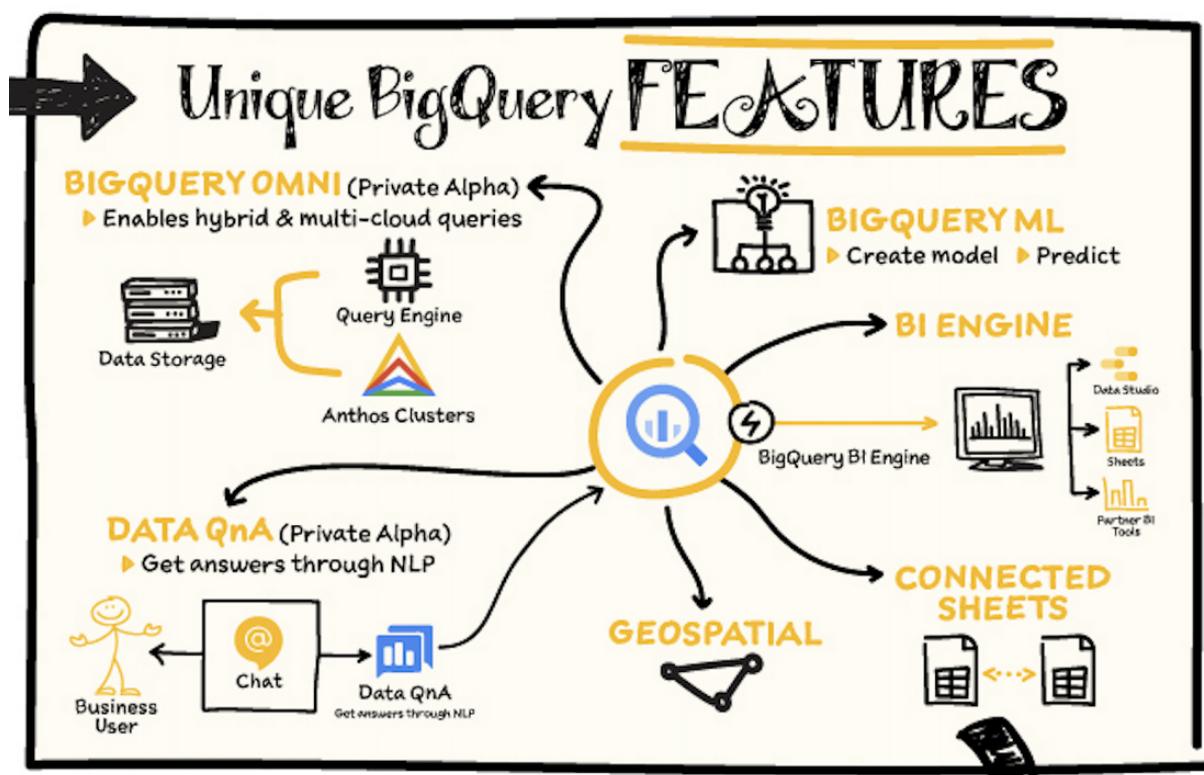
Architecture of BigQuery

Features of BigQuery

Bigquery offers a wide range of features that helps us get detailed insights from data in a very cost efficient manner.

Here is a list of bigquery features:

- Multi Cloud Functionality
- Built-in ML Integration
- Foundation for BI
- Geospatial Analysis
- Automated Data Transfer
- Free Access
- Partitioning and Clustering



Geospatial Analysis

What is Geospatial Analysis ?

Geospatial Analytics highlights historical changes and current shifts by collecting, displaying, and manipulating Imagery and Geographic Information System (GIS) Data related to a specific location. GIS provides information about location and mapping. It functions by converting latitudes and longitudes columns into geographical points. The gathered information helps create Data Visualisations, such as Graphs, Maps, Stats, and Cartograms. These reports help the human brain to understand the distance, proximity, and contiguity not visible in large datasets.

Tools Supported by BigQuery for Geospatial Analytics - BigQuery GeoViz, Google Data Studio, Google Earth Engine and Jupyter Notebooks.

GoogleCloudPlatform/ **bigquery-geo-viz**

Visualize Google BigQuery geospatial data using
Google Maps Platform APIs



[BigQuery Geospatial analysis](#)

Now, let's try to explore and understand the covid-19 dataset available on bigquery and later visualise that on a graph.

The screenshot shows the Google Cloud BigQuery Explorer interface. On the left, the 'Explorer' sidebar lists pinned projects: 'bigqueryindium' and 'bigquery-public-data'. In the center, the 'usa_names' dataset is selected. The 'Data set info' panel on the right displays details like Data set ID, Created, Default table expiry, Last modified, Data location, and Description. Below it, the 'Default collation' is listed. The main content area is titled 'About COVID-19 Public Datasets' with a 'VIEW DATA SET' button. It includes tabs for 'OVERVIEW' (selected) and 'SAMPLES'. The 'OVERVIEW' section contains a brief description of the COVID-19 Public Datasets program, its purpose, and the datasets it contains.

The screenshot shows the Google Cloud BigQuery Explorer interface with a query results page. The search bar at the top contains the query: 'jhu'. The results table shows data from the 'covid19_jhu_csse' table, specifically the 'summary' view. The table has columns: Row, province_state, country_region, date, latitude, longitude, location_geom, confirmed, deaths, recovered, active, fips, and admin2. The results are ordered by date desc. The first 10 rows are displayed, showing data for various US states like Virginia, Louisiana, Vermont, West Virginia, North Carolina, Wisconsin, Texas, Connecticut, and Massachusetts. The table includes geographical coordinates and administrative details for each entry.

This query selects only rows that are in US country_region and orders it by most recent data.

NOTE :- fips is an id made with first 2 digits of province state code and last 3 digits with counties code.

This query shows the confirmed cases, deaths and recovered cases only on **1st of March, 2020** and in the US only.

This query shows the country, province_state, confirmed cases, deaths and recovered cases only in the **country US** and in **Vermont** only.

The screenshot shows the Google Cloud BigQuery interface. In the top navigation bar, it says "Google Cloud" and "BigQueryIndium". The search bar contains "Search Products, resources, docs (/)". Below the navigation, there's a "Sandbox" message: "Set up billing to upgrade to the full BigQuery experience. [Learn more](#)". On the right, there are "DISMISS" and "UPGRADE" buttons.

The main area shows an "Explorer" view with a sidebar containing a tree structure of datasets and tables. One table, "recovered_cases", is selected and expanded. The schema for "recovered_cases" is displayed in a table:

Field name	Type	Mode	Collation	Policy tags	Description
province_state	STRING	NULLABLE			
country_region	STRING	NULLABLE			
latitude	FLOAT	NULLABLE			
longitude	FLOAT	NULLABLE			
location_geom	GEOGRAPHY	NULLABLE			
_1_22_20	INTEGER	NULLABLE			
_1_23_20	INTEGER	NULLABLE			
_1_24_20	INTEGER	NULLABLE			
_1_25_20	INTEGER	NULLABLE			
1_26_20	INTEGER	NULLABLE			

Below the schema table are buttons for "EDIT SCHEMA" and "VIEW ROW ACCESS POLICIES". To the right of the schema table, there are several options: "Explore with Sheets", "Explore with Data Studio", "Explore with GeoViz", "Export to GCS", and "Scan with DLP".

It can be noticed that since there is a data type **GEOGRAPHY**, bigquery allows us to explore this dataset on a map. To do this, we have to click on **EXPORT** and then **Explore with GeoViz**.

The screenshot shows the "BigQuery Geo Viz" interface. At the top, there's a blue header with the title "BigQuery Geo Viz" and links for "Feedback", "Source", "Terms & privacy", and "Sign out". The user is signed in as "k4balaji@gmail.com".

The main area has a left sidebar labeled "Query" with a "Project ID" section showing "bigqueryindium-355006". Below it is a code editor with a single line of SQL:

```
1. SELECT * FROM `bigqueryindium-355006.covid_19_usa.recovered_cases`
```

At the bottom of the code editor are "Run" and "Show results (270)" buttons. A note below the code editor says "Estimated query size: 1.9 MB".

The right side of the screen is a world map showing red dots representing the query results. The map includes labels for continents, countries, and oceans. The red dots are concentrated in North America, Europe, and parts of Africa and Asia.

The tiny red dots indicate the query results on the map.

BigQuery Geo Viz

Feedback Source Terms & privacy

1 Query

Project ID
bigqueryindium-355006

```
1 SELECT* FROM `bigqueryindium-355006.covid_19_usa.summary`  
2 where country_region = 'US'  
3 order by date desc
```

Run Show results (269,230 of 2,686,970)

Estimated query size: 474.3 MB

Now let's look at more queries.

As we can see the red dots are presented only within the US.

1 Query

Project ID
bigqueryindium-355006

```
1 SELECT *
2 FROM `bigqueryindium-355006.covid_19_usa.summary`
3 where country_region = 'US' and province_state = 'Vermont'
4
```

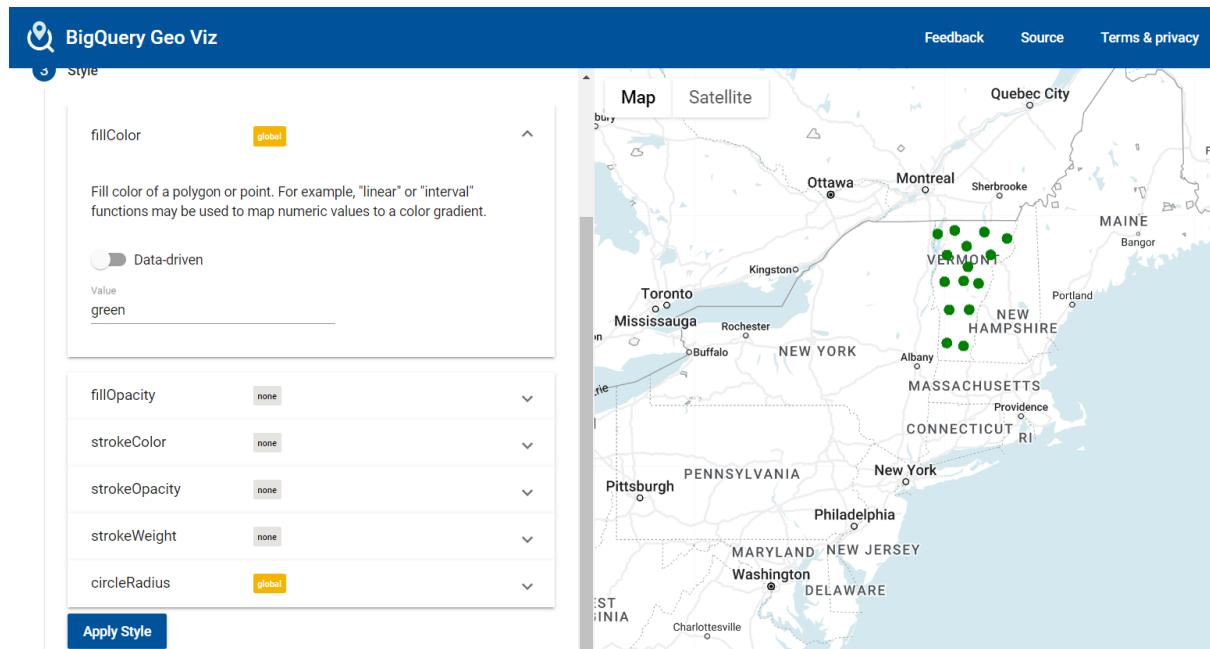
Run Show results (12,460)

A map of the New England region focusing on Vermont. The state of Vermont is highlighted with a red bounding box and numerous red dots representing data points. Other states shown include Quebec, Maine, New Hampshire, Massachusetts, Connecticut, Rhode Island, and parts of New York, Pennsylvania, and New Jersey. Major cities like Montreal, Ottawa, Quebec City, Bangor, Portland, Albany, Albany, Boston, Providence, Hartford, and New York City are labeled. The map also shows Lake Ontario and Lake Erie.

The red dots are only within Vermont.

Note that we can also change the colour of the dots. Now, let's change it to green and have a look.

We can also increase or decrease the radius of the points.



GEOSPATIAL ANALYSIS WITH STARBUCKS DATABASE

Let us explore a few problem statements with the starbucks database. I have downloaded the dataset from [Kaggle](#). The said database has the information on the location of its stores worldwide.

For example,

As an individual, I am interested in investing in a starbucks store in my hometown of Regina, Canada. Here are a few problem statements that I would need to solve.

1. How many countries in the world have starbucks?
2. How many stores does Canada have?
3. Find the number of distinct cities.
4. Figure out the type of ownership.
5. No. of chains in a city (In this case, City of Regina)
6. What is the distance between each store?

Starbucks started as a roaster and retailer of whole bean and ground coffee, tea and spices with a single store in Seattle's Pike Place Market in 1971. The company now operates more than 24,000 retail stores in 70 countries.

Now, let's find the answers!

Based on our query below, we can see that there are 24,079 stores in the world.

Unsaved ... ery

RUN **SAVE** **SHARE** **SCHEDULE** **MORE**

```
1 SELECT
2   | count(*) Country
3   | FROM
4   | `myfirstbigquery356706.starbucksdata`
```

Query results

RESULTS

Row	Country
1	24079

SAVE RESULTS

JOB INFORMATION EXECUTION DETAILS

One more interesting thing about bigquery is that we can explore the dataset on maps using GeoViz. Let's take a look below!

BigQuery Geo Viz

Feedback Source Terms & privacy | sfredrick35@gmail.com Sign out

```
1 SELECT
2   ST_GeogPoint(longitude, latitude) AS WKT,
3   FROM `myfirstbigquery-356706.myfirstbigquery356706.starbucksdata`
```

Run Show results (24,079)

Estimated query size: 376.2 KB
Processing location Auto-select

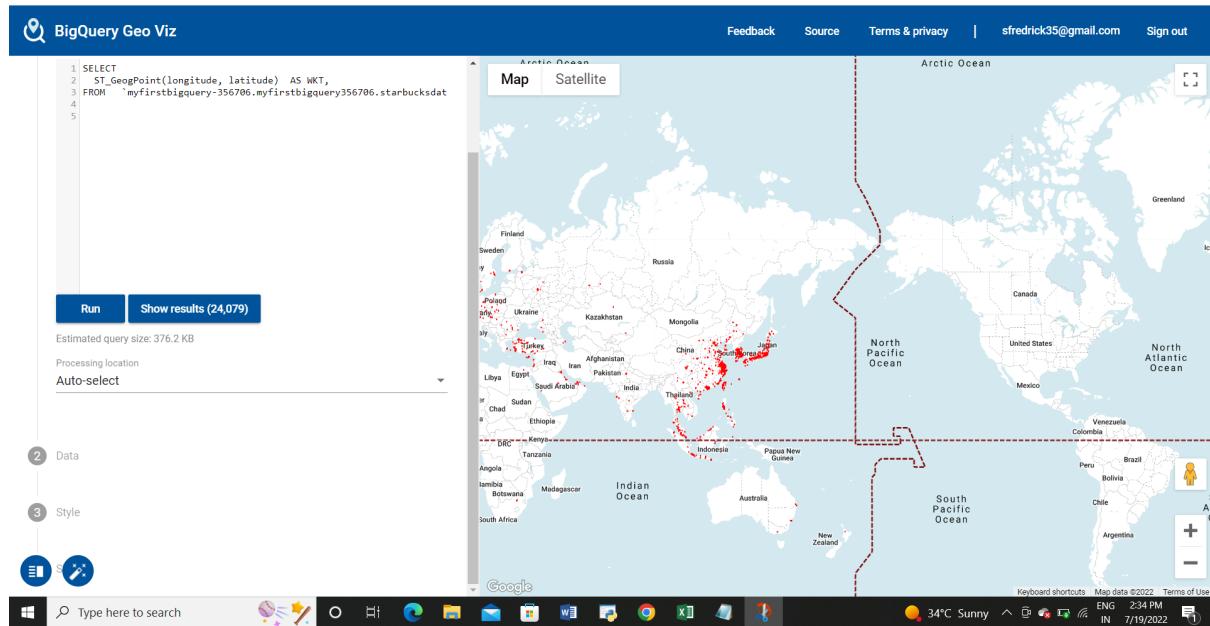
Map Satellite

② Data ③ Style

Type here to search

Keyboard shortcuts Map data ©2022 Terms of Use

34°C Sunny ENG 233 PM IN 7/19/2022



I need to know the number of stores in Canada as I am looking to invest in a store here. Here we can see that 1464 stores are operating in Canada.

```

1 SELECT
2   | count(*) Country
3   | FROM
4   |   `myfirstbigquery356706.starbucksdata`
5 WHERE
6   |   | Country = 'CA'
7
  
```

Query results				
JOB INFORMATION		RESULTS	JSON	EXECUTION DETAILS
Row	Country			
1	1464			

Search Products, resources, docs (/)

*Unsaved query +>

RUN SAVE SHARE SCHEDULE MORE This query will be deleted at 10:00 AM on 11/19/2022

```

1 SELECT
2   COUNT(DISTINCT City) AS City
3 FROM `myfirstbigquery356706.starbucksdata`
4 WHERE
5   Country = 'CA'

```

Pr

Query results SAVE RESULTS

Row	City
1	221

These 1464 stores are present in 221 cities. This can be plotted on a map as well.

BigQuery Geo Viz

Feedback Source Terms & privacy | sfredrick35@gmail.com Sign out

```

1 SELECT
2   ST_GeogPoint(longitude, latitude) AS WKT,
3   FROM `myfirstbigquery-356706.myfirstbigquery356706.starbucksdata`
4 WHERE
5   Country = 'CA'
6

```

Run Show results (1,464)

Estimated query size: 470.3 KB
Processing location: Auto-select

② Data ③ Style

Google 34°C Sunny 2:26 PM IN 11/19/2022

Canada has 6% of the stores in its country. This can be calculated.

The screenshot shows a BigQuery query interface. At the top, there is a toolbar with buttons for RUN, SAVE, SHARE, SCHEDULE, MORE, and a note indicating the query will process 94.06 KB. Below the toolbar is the SQL query:

```
1 SELECT
2   COUNTRY
3   ,count(*) / (sum(count(*)) OVER()) as pct
4 FROM
5   `myfirstbigquery356706.starbucksdata`
6 GROUP BY
7   Country
```

The results section shows a table with columns: Row, COUNTRY, and pct. The row for Canada (Row 10) has a value of 0.060799867104115619, which is circled in red. A red arrow points from this circled value to the text "PERCENTAGE IS 6%".

Row	COUNTRY	pct
8	DK	0.00083059927737862865
9	RO	0.00091365920511649154
10	CA	0.060799867104115619
11	MA	4.1529963868931435e-05
12	FI	0.00033223971095145148
13	VN	0.0010382490967232858
14	AD	4.1529963868931435e-05
15	EG	4.1529963868931435e-05
16	TD	0.012625100016155156

Results per page: 50 ▾ 1 – 50 of 66 | < >

When you explore the dataset, you might find an interesting factor. Not all stores are company owned. Which is not surprising since Starbucks is a multinational chain of coffeehouses.

The types of ownership are Licensed, company owned, franchise and joint venture. Let's take a look at the types of ownership in Canada. (Which is Licensed and Company owned. Since it is a small number, we can easily filter it on excel).

There are about 1093 company owned stores and 371 licensed stores in Canada.

The image shows two separate BigQuery query interfaces. Both queries are run against the dataset 'myfirstbigquery356706.starbucksdata' and filter for Canada ('CA').

Query 1: Company Owned Stores

```
1 SELECT
2   COUNT (*) Ownership_Type
3 FROM `myfirstbigquery356706.starbucksdata`
4 WHERE
5   Ownership_Type = 'Company Owned'
6 AND
7   COUNTRY = 'CA'
```

Query Results:

Row	Ownership_Type
1	1093

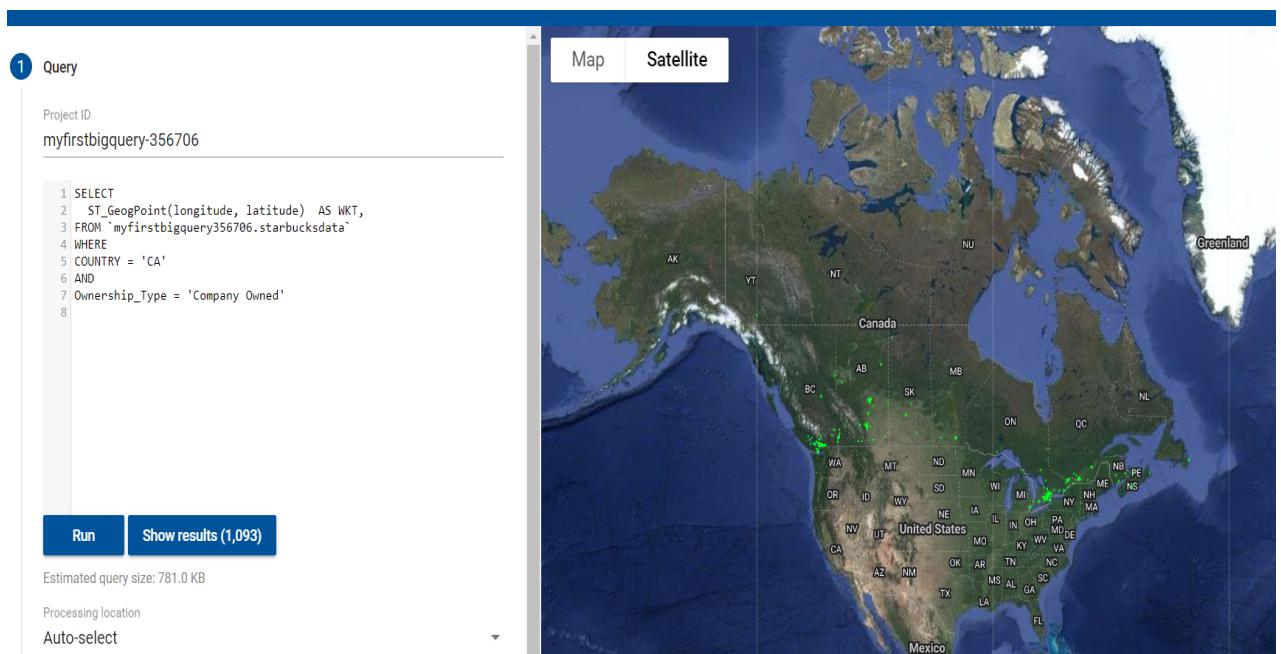
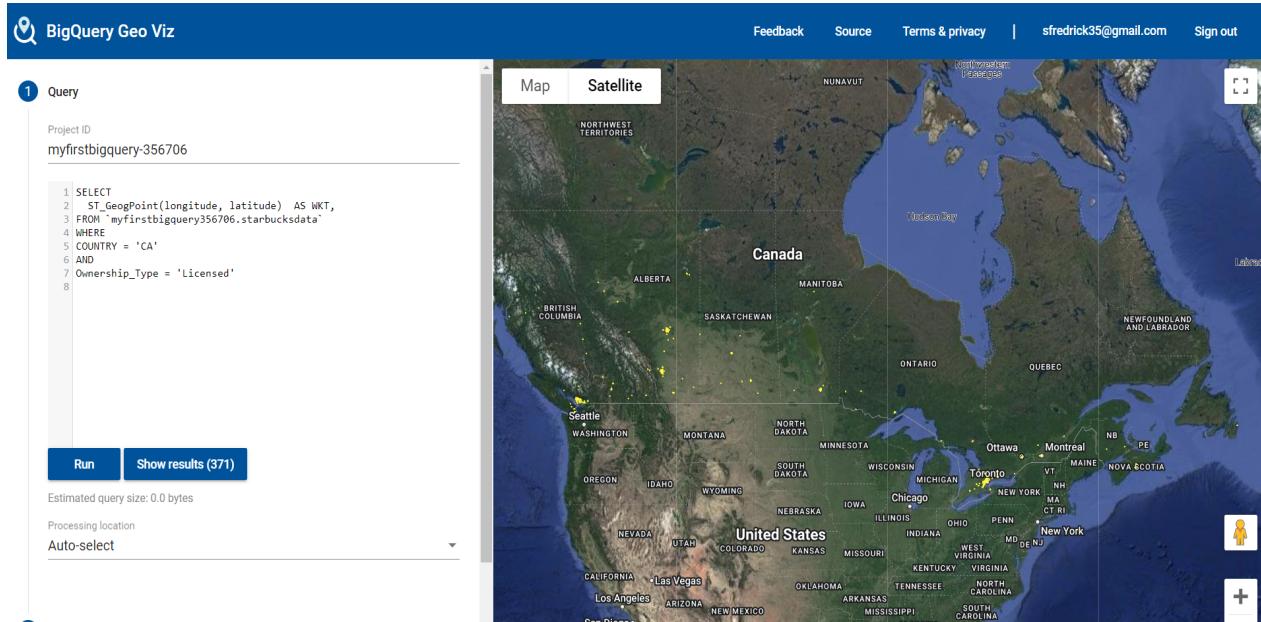
Query 2: Licensed Stores

```
1 SELECT
2   COUNT (*) Ownership_Type
3 FROM `myfirstbigquery356706.starbucksdata`
4 WHERE
5   Ownership_Type = 'Licensed'
6 AND
7   COUNTRY = 'CA'
```

Query Results:

Row	Ownership_Type
1	371

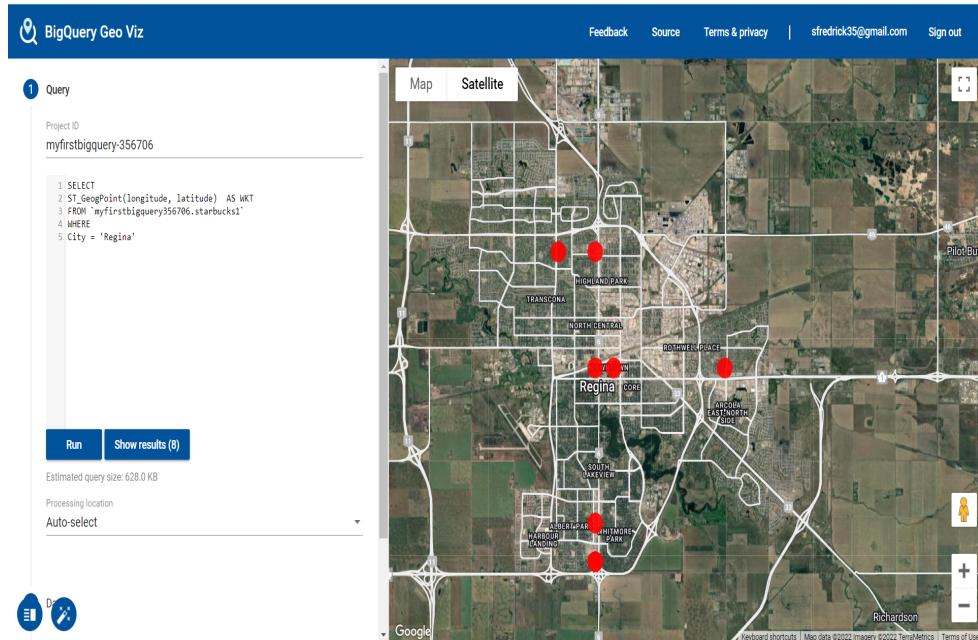
The yellow dots in the map represent the licensed stores and the green dots represent company owned stores.



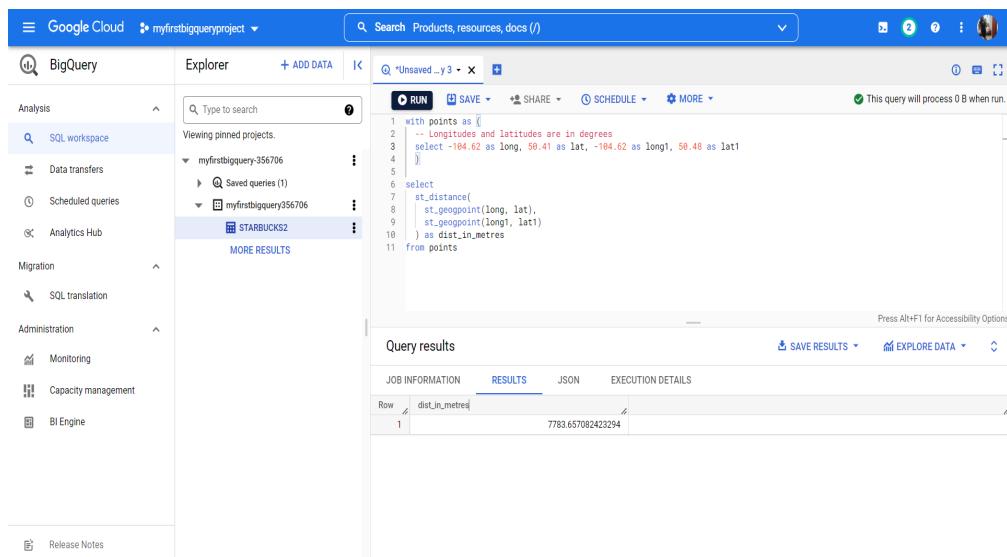
In this example, I want to open a store in the city of Regina.

It is the capital of the province of Saskatchewan. It is the second largest city in the province with a growing population. With an area of 12,931km, it has about 9 stores in the city.

Below is the map view of Regina with the store's location.



Let us find the distance between the stores.



The distance between the stores is approximately 7800 metres.

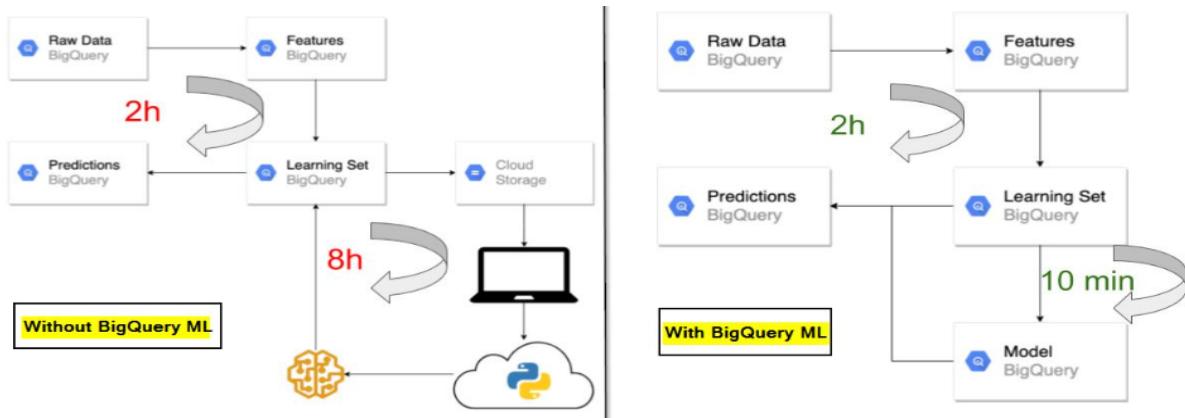
Since the stores come under the same coordinates, the distance between them will be more or less similar.

Et Voila! We have found out the basic answers that we need using geospatial analysis and Bigquery.

BigQuery ML

Google BigQuery ML is a set of tools and extensions that enables users to create, train and execute Machine Learning models in Google BigQuery using standard SQL queries. It eliminates the need for data movement by allowing users to directly create Machine Learning models into the Data Warehouse.

Key Features of Google BigQuery ML - Automatically Generate ML Models, Eliminates Data Transfer and Encrypted Models, supports linear and logistic regression, k-means, matrix factorization, time series, XGBoost, AutoML tables, etc.



Working of Machine learning With and without Bigquery

Different Models Supported in Google BigQuery ML

Various Machine Learning models are used to train predict data. All the Machine Learning models supported by Google BigQuery ML are listed below:

- **Linear Regression:** Linear Regression is the most basic model that is used for forecasting. It uses statistical calculations to make predictions. Google BigQuery ML provides this Machine Learning model to forecast trends based on data. Linear Regression is used for real-valued label prediction.
- **Logistic Regression:** Logistic Regression is used for the classification. When the required output is in YES or NO form, Logistic Regression delivers better results than Linear Regression because it converts every result in 1 or 0 i.e., YES or NO.
 - **Binary Logistic Regression:** It is used when the required output only has 2 outcomes meaning YES or NO. Binary-class classification can have only two possible values, 1 and 0.
 - **Multiclass Logistic Regression:** Classification can have multiple outcomes. Google BigQuery ML supports Multiclass Logistic Regression to allow users to solve real-world problems based on classification. It uses a multinomial classifier with a cross-entropy loss function to train Multiclass Logistic Regression models.
- **K-Means Clustering:** K-means clustering model is used for data segmentation. It uses non labeled data and groups similar data points in one group. Users can use the K-means clustering model with the CREATE MODEL statement with the KMEANS model type.

- **Matrix Factorization:** Google BigQuery ML allows users to quickly create and train models. The matrix Factorization model is used to create product recommendation systems. Users can use past customer behavior data with their product purchase data to build a robust recommendation system using SQL.
- **Time Series:** It is used to perform forecasts on time-series data. Google BigQuery ML eliminates the tedious task to handle anomalies in data like holidays, seasonality, etc. With the help of the Time Series model, users can create many time series models, test them and use them for forecasting.

Let us look at a simple Linear Regression model.

Problem statement 1 - Bank marketing (Dataset from [Kaggle](#))

The screenshot shows the Google BigQuery interface. On the left, there's a sidebar with various icons and a search bar. The main area is titled "bank_marketing" and shows the schema. The "SCHEMA" tab is selected, displaying a table with columns: Field name, Type, Mode, Collation, Policy tags, and Description. The columns listed are age (INTEGER, NULLABLE), job (STRING, NULLABLE), marital (STRING, NULLABLE), education (STRING, NULLABLE), default (BOOLEAN, NULLABLE), balance (INTEGER, NULLABLE), housing (BOOLEAN, NULLABLE), loan (BOOLEAN, NULLABLE), contact (STRING, NULLABLE), and day (INTEGER, NULLABLE).

Field names - age, job, marital, education, default, balance, housing, loan, contact, day, month, duration, campaign, pdays, previous, outcome, deposit

AIM - Predict whether a customer will subscribe to a term deposit or not based on the provided information

Preview of the dataset.

The screenshot shows the results of a query on the bank_marketing dataset. At the top, there are buttons for RUN, SAVE, SHARE, SCHEDULE, and MORE. A note says "This query will process 1.22 MB when run". Below that is a "Query results" section with tabs for JOB INFORMATION, RESULTS (which is selected), JSON, and EXECUTION DETAILS. The results table has columns: Row, age, job, marital, education, default, balance, housing, loan, contact, day, month, duration, campaign, pdays, previous, outcome, and deposit. The data shows 10 rows of customer information, such as age 26, job admin., marital single, education secondary, and so on. At the bottom, there are pagination controls for Results per page (50), page number (1 – 50 of 1000), and navigation arrows.

Now, let's check out what's the distribution of customers subscribing and those not to the deposit.

RUN SAVE ▾ SHARE ▾ SCHEDULE ▾ MORE ▾

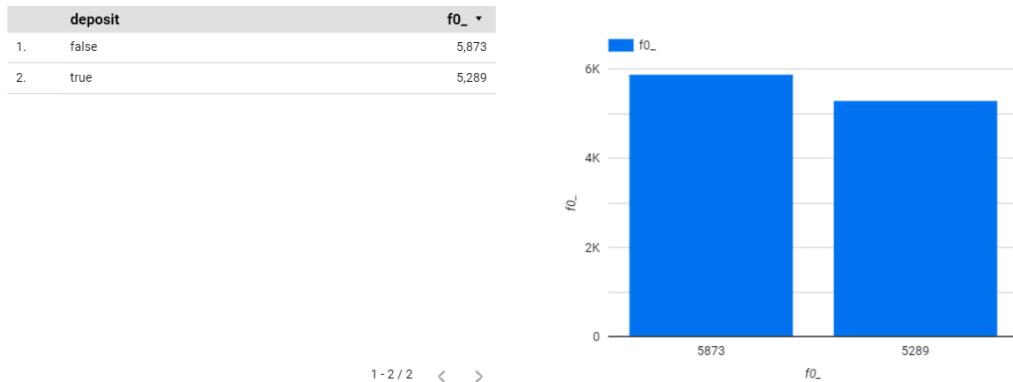
```
1 select count(*), deposit
2 FROM `bigqueryindium-355006.bank_ML.bank_marketing`
3 group by deposit
4
```

Query results

JOB INFORMATION			RESULTS	JSON	EXECUTION DETAILS
row	f0_	deposit			
1	5289	true			
2	5873	false			

Now, on exploring the obtained result on Data Studio,

bank_marketing



Now, let's split the data into training, validation and prediction data, in the ratio of 80-10-10.

```

1 select age, job, marital, education, 'default', balance, housing,
2 loan, contact, day, month, campaign, pdays, previous, poutcome, deposit,
3 CASE
4   WHEN split_field < 0.8 THEN 'train'
5   WHEN split_field = 0.8 THEN 'valid'
6   WHEN split_field > 0.8 THEN 'pred'
7 end as dataframe
8 from (
9 select age, job, marital, education, 'default', balance, housing,
10 loan, contact, day, month, campaign, pdays, previous, poutcome, deposit,
11 ROUND(ABS(RAND()),1) as split_field
12 FROM `bigqueryindium-355006.bank_ML.bank_marketing` )
13

```

The inner clause (line 9-12) helps in random sampling by assigning random values within 0 to 1 to split_field. CASE helps in splitting the data into 3 groups.

job	marital	education	f0_	balance	housing	loan	contact	day	month	campaign	pdays	previous	poutcome	deposit	dataframe
admin.	single	secondary	default	274	false	false	cellular	1	apr	3	-1	0	unknown	true	train
admin.	married	secondary	default	1474	false	false	cellular	2	apr	2	-1	0	unknown	true	train
admin.	single	secondary	default	471	false	true	cellular	2	apr	3	-1	0	unknown	true	train
admin.	divorced	secondary	default	4099	false	false	cellular	2	apr	2	-1	0	unknown	true	valid
admin.	married	secondary	default	1207	true	false	telephone	6	apr	2	138	2	success	true	train
admin.	single	secondary	default	82	false	true	cellular	7	apr	2	-1	0	unknown	true	pred
admin.	divorced	secondary	default	12039	false	false	telephone	8	apr	1	-1	0	unknown	true	valid

Let's copy this to a new table **marketing_pred** to prevent the change of values in the dataframe column .

Now, let's check the distribution into the 3 groups.

Row	dataframe	deposit	f0_
1	pred	false	872
2	pred	true	760
3	train	true	4002
4	train	false	4430
5	valid	true	527
6	valid	false	571

Now, let's train the model using logistic regression.

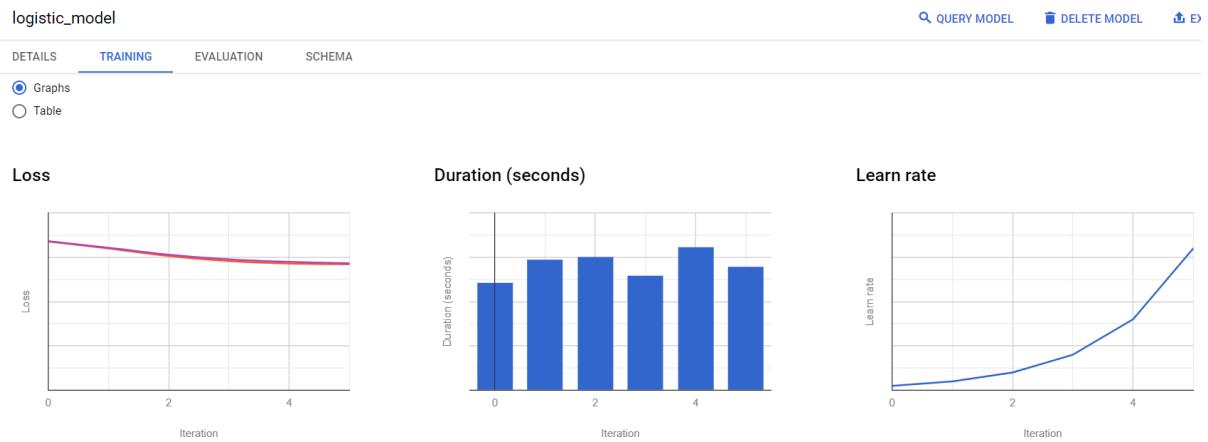
RUN **SAVE** **SHARE** **SCHEDULE** **MORE**

```

1 CREATE or REPLACE MODEL
2 `bank_ML.logistic_model`
3 OPTIONS
4   ( model_type="logistic_reg",
5     input_label_cols=['deposit'] ) AS
6 SELECT* EXCEPT (dataframe)
7 FROM `bigqueryindium-355006.bank_ML.marketing_pred`
8 WHERE dataframe = 'train'

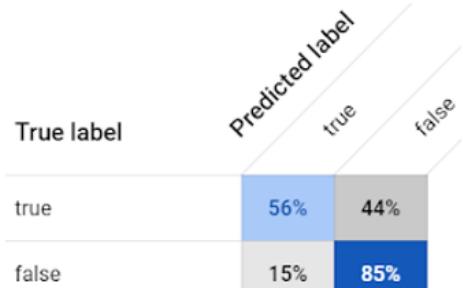
```

Output we get as,



Score threshold

Positive class threshold	?	<input type="range" value="0.5126"/>
Positive class	true	
Negative class	false	
Precision	?	0.7667
Recall	?	0.5560
Accuracy	?	0.7124
F1 score	?	0.6445



Below query gives the weights/coefficients of each feature in the model.

```
1 SELECT *  
2 FROM ML.WEIGHTS(MODEL `bigqueryindium-355006.bank_ML.logistic_model`)
```

Query results

JOB INFORMATION		RESULTS	JSON	EXECUTION DETAILS	
Row	processed_input	weight	catego....category	catego....weight	
1	age	0.00096588255658649016	null	null	
2	job		blue-collar	-0.11093851040324444	
			retired	0.58413191977710488	
			management	-0.12791035822067576	
			services	-0.067945921343715632	
			entrepreneur	-0.18184123491478263	
			technician	-0.039505260047204144	
			housemaid	-0.56428551280472339	
			unknown	-0.014622892775512451	
			admin.	0.0027200460901776577	

Below query gives the details about each feature.

```
1 SELECT *
2 FROM ML.FEATURE_INFO(MODEL `bigqueryindium-355006.bank_ML.logistic_model` )
```

Query results

 SAVE R

JOB INFORMATION		RESULTS		JSON	EXECUTION DETAILS				
Row	input	min	max	mean	median	stddev	category_count	null_count	dimension
1	age	18.0	95.0	41.31764357608057	38.0	11.941879586814858	null	0	null
2	job	null	null	null	null	null	12	0	null
3	marital	null	null	null	null	null	3	0	null
4	education	null	null	null	null	null	4	0	null
5	derog	null	null	null	null	null	1	0	null
6	balance	-6847.0	81204.0	1500.466844286557	555.0	3115.4530967678393	null	0	null
7	housing	null	null	null	null	null	2	0	null
8	loan	null	null	null	null	null	2	0	null
9	contact	null	null	null	null	null	3	0	null
10	day	1.0	31.0	15.587773830669061	15.0	8.45900945005697	null	0	null

On predicting values for valid dataset,

```
RUN SAVE ▾ SHARE ▾ SCHEDULE ▾ MORE ▾
1 SELECT *
2 FROM ML.EVALUATE(MODEL `bigqueryindium-355006.bank_ML.logistic_model`,
3 (SELECT*
4 FROM `bigqueryindium-355006.bank_ML.marketing_pred`
5 WHERE dataframe='valid'
6 )
7 )
```

Query results

 SAVE RESULT

JOB INFORMATION		RESULTS	JSON	EXECUTION DETAILS			
Row	precision	recall	accuracy	f1_score	log_loss	roc_auc	
1	0.73366834170854267	0.5540796963946869	0.68943533697632053	0.63135135135135134	0.60041486294969	0.74416383616383619	

Note that we want to know customers who will be interested in deposit, so we have to improve our recall.

Now, let's do some feature engineering so as to improve recall using the **TRANSFORM** function.

```
RUN SAVE SHARE SCHEDULE MORE
1 CREATE OR REPLACE MODEL `bigqueryindium-355006.bank_ML.logistic_model_2`
2 TRANSFORM(
3   ML.QUANTILE_BUCKETIZE(age,5) OVER() AS bucketized_age,
4   ML.FEATURE_CROSS(STRUCT(job,education)) job_education,
5   marital,balance,housing,loan,contact,day,month,pdays,previous,poutcome,deposit)
6 OPTIONS(
7   model_type='LOGISTIC_REG',
8   input_label_cols=['deposit']
9 ) AS
10 SELECT * EXCEPT(dataframe,campaign, derog)
11 FROM `bigqueryindium-355006.bank_ML.marketing_pred`
12 WHERE dataframe='train'
```

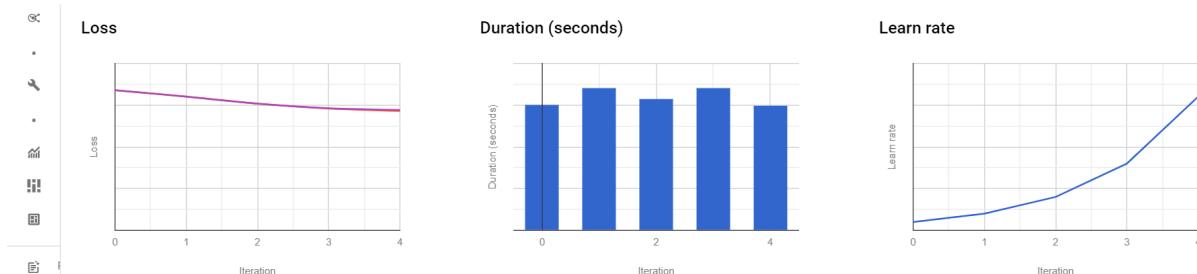
In line 3, 'age', which is a continuous numerical feature, is bucketized into a STRING based on quantiles. Here, we are making it into 5 buckets.

The 'job' and 'education' columns which have categorical values are kind of related, so we can do a feature cross on them.

Note that we have also dropped campaign and derog columns.

We'll now apply the logistic regression model on the train part of the dataset.

Now, click on '**go to model**' and let's check out its details.



It can be noticed that the number of iterations have reduced.

Score threshold

Positive class threshold	?	<input type="range" value="0.5192"/>	0.5192
Positive class	true		
Negative class	false		
Precision	?	0.7475	True label
Recall	?	0.5616	predicted label
Accuracy	?	0.7069	true false
F1 score	?	0.6414	60% 40% 20% 80%

We can see that the true positive has increased thus causing a higher recall.

```
1 SELECT *
2 FROM ML.WEIGHTS(MODEL `bigqueryindium-355006.bank_ML.logistic_model_2`)
```

Query results

JOB INFORMATION		RESULTS	JSON	EXECUTION DETAILS	
Row	processed_input	weight	catego....category	catego....weight	
1	bucketized_age	null	bin_2	-0.079846096895319491	
			bin_4	-0.12447591816604348	
			bin_5	0.059261741101308038	
			bin_1	0.20544639114220695	
			bin_3	-0.087251293314640072	
2	job_education_job_education	null	services_secondary	-0.11958930699687889	
			services_tertiary	0.2178799635747094	
			self-employed_tertiary	0.32073672221621924	
			retired_unknown	0.34745278405240387	

We can now see the changed weights/coefficients of each feature in the new model.

Arrays and Struct in Google BigQuery

ARRAYS

What are arrays?

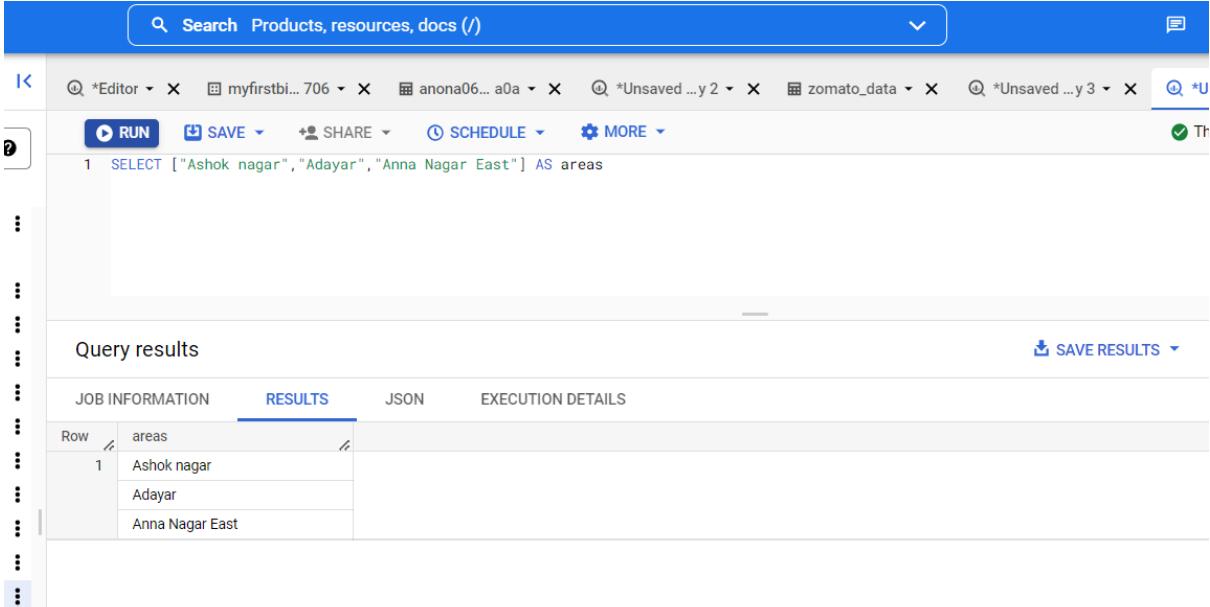
Arrays are basically an ordered list with zero or more values of the same data type. It is used when we want to store repeated values in a single row. They can be very powerful once we learn how to use them. For example, they can be used to:

- Transforming non-normalized data
- Advanced string manipulation
- Optimising storage and performance, etc.,

FUNCTIONS OF ARRAY

There are several features of array:

- Array_Length
- Array_concat
- Array_generate
- Generate_date_array
- Array_reverse



The screenshot shows the Google BigQuery web interface. At the top, there is a search bar and a navigation bar with various project and dataset names. Below the search bar, there are buttons for RUN, SAVE, SHARE, SCHEDULE, and MORE. A dropdown menu is open, showing a checked item. The main area displays a query result. The query is:

```
1 SELECT ["Ashok nagar", "Adayar", "Anna Nagar East"] AS areas
```

The results section shows the following table:

Row	areas
1	Ashok nagar
	Adayar
	Anna Nagar East

At the bottom right of the results table, there is a 'SAVE RESULTS' button.

A simple example of an array

Arrays Length in BigQuery:

As the name suggests, an array length tells us the length of an array. This may come in handy when we want to know the length of an array while indexing.

We can use the `array_length` to do this.

The screenshot shows a BigQuery query results interface. At the top, there are buttons for RUN, SAVE, SHARE, SCHEDULE, and MORE. Below that, the query code is displayed:

```
1 SELECT ARRAY_LENGTH(['SALIGRAMAM', 'ADAYAR', 'KODAMBAKKAM']) AS LENGTH
```

Below the code, the results section is titled "Query results". It has tabs for JOB INFORMATION, RESULTS (which is selected), JSON, and EXECUTION DETAILS. The results table has two columns: Row and LENGTH. There is one row with the value 1 in the Row column and 3 in the LENGTH column.

Row	LENGTH
1	3

A simple example of an array length

Arrays Reverse in BigQuery:

The order of array elements can be reversed. If we want to access the elements at the bottom of an array, we can use this function.

We can use the `array_reverse` to do this.

The screenshot shows the BigQuery web interface. At the top, there is a search bar and several project selector dropdowns. Below the header, there are buttons for RUN, SAVE, SHARE, SCHEDULE, and MORE. A code editor window contains the following SQL query:

```
1 SELECT ARRAY_REVERSE(['SALIGRAMAM', 'ADAYAR', 'KODAMBAKKAM']) AS LENGTH
```

Below the code editor is a section titled "Query results". It includes tabs for JOB INFORMATION, RESULTS (which is selected), JSON, and EXECUTION DETAILS. The results table has a single row with one column labeled "LENGTH". The values in the column are KODAMBAKKAM, ADAYAR, and SALIGRAMAM, listed from bottom to top, demonstrating the reversal of the array.

A simple example of an array reverse

Array Aggregate in BigQuery:

This functions **returns an ARRAY of expression values**. It is basically the opposite of UNNEST (used to flatten an array into its components to make it usable for analysis and database operations).

We use the `array_agg` to do this.

The screenshot shows the BigQuery web interface. At the top, there's a search bar and a navigation bar with tabs like 'myfirstbi...', 'anona06...', 'zomato_data', and 'zomato_data'. Below the search bar are buttons for 'RUN', 'SAVE', 'SHARE', 'SCHEDULE', and 'MORE'. The main area contains a SQL query:

```
1 WITH Restaurants as (
2   SELECT "That Madras Place" AS name, "Adayar" AS location UNION ALL
3   SELECT "Haunted" AS name, "Anna Nagar East" AS location UNION ALL
4   SELECT "Pantry d'or" AS name, "Anna Nagar East" AS location UNION ALL
5   SELECT "Palmshore" AS name, "Ashok Nagar" AS location
6 )
7 SELECT location, ARRAY_AGG(name) as arrayAgg
8 FROM Restaurants
9 GROUP BY Restaurants.location
```

Below the query is a 'Query results' section with a 'RESULTS' tab selected. The results table has columns 'JOB INFORMATION', 'RESULTS', 'JSON', and 'EXECUTION DETAILS'. The 'RESULTS' table looks like this:

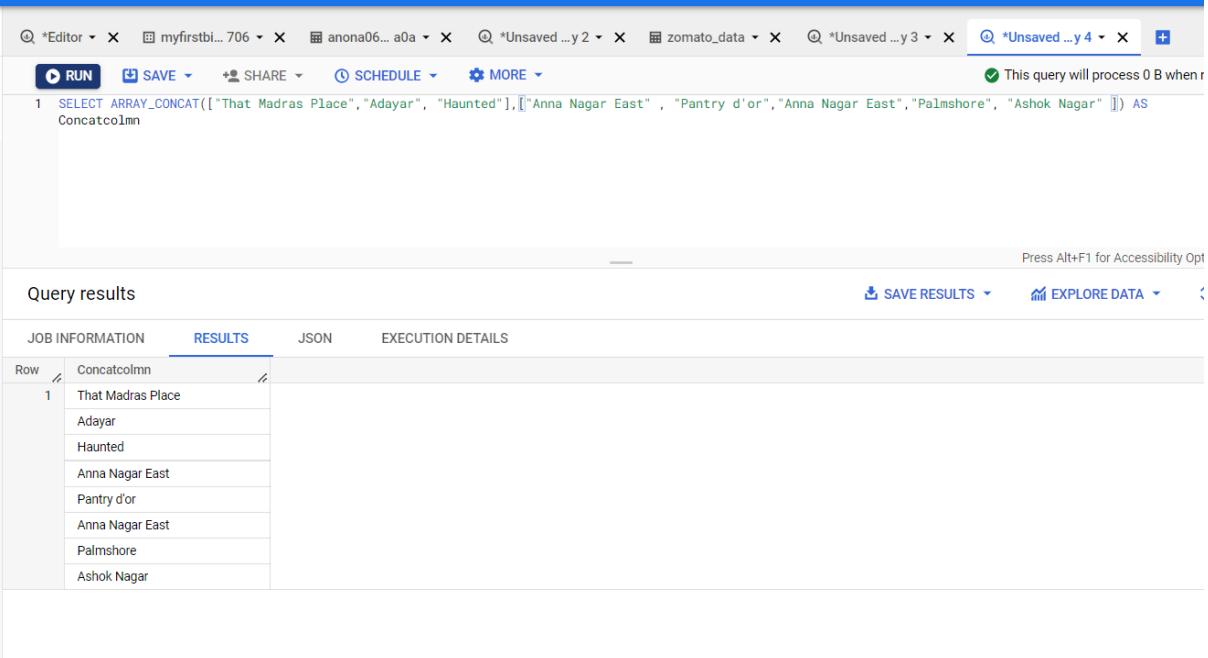
Row	location	arrayAgg
1	Adayar	That Madras Place
2	Anna Nagar East	Haunted
		Pantry d'or
3	Ashok Nagar	Palmshore

A simple example of an array_agg

Array Concat in BigQuery:

This function concatenates one or more arrays with the same element type into a single array.

We use the `array_concat` to do this.



The screenshot shows the BigQuery web interface. At the top, there are several tabs and buttons: RUN, SAVE, SHARE, SCHEDULE, MORE, and a note indicating "This query will process 0 B when run". Below the toolbar, a query is displayed:

```
1 SELECT ARRAY_CONCAT(["That Madras Place", "Adayar", "Haunted"], ["Anna Nagar East", "Pantry d'or", "Anna Nagar East", "Palmshore", "Ashok Nagar"]) AS Concatcolmn
```

Below the query, the results are shown in a table titled "Query results". The table has three columns: JOB INFORMATION, RESULTS, and EXECUTION DETAILS. The RESULTS column is selected and contains the following data:

Row	Concatcolmn
1	That Madras Place Adayar Haunted Anna Nagar East Pantry d'or Anna Nagar East Palmshore Ashok Nagar

A simple example of an array concat

Array Generate in BigQuery:

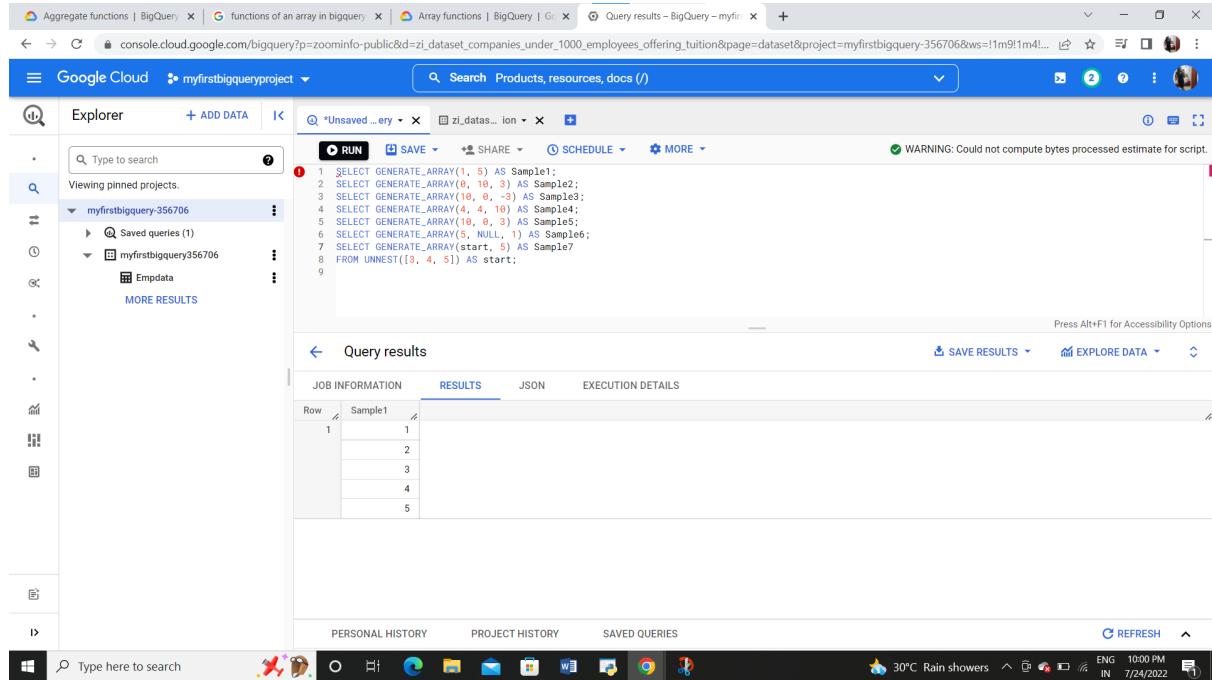
This returns an array of values. We need to define the start and end expressions in order to implement this function. We also need to define a step expression in order to determine the increment value.

We use the `array_generate` to do this.

Now let us take a look at a few examples:

1. The defined start and end expression for sample 1 is 1 and 5 respectively.

Hence, it will return the value up to 5 with a default step of 1.



The screenshot shows the Google Cloud BigQuery interface. On the left, the 'Explorer' sidebar lists projects, with 'myfirstbigquery-356706' selected. Under this project, 'Saved queries (1)' and 'myfirstbigquery356706' (containing 'Emptdata') are visible. The main area displays a query editor with the following SQL code:

```
1 SELECT GENERATE_ARRAY(1, 5) AS Sample1;
2 SELECT GENERATE_ARRAY(8, 10, 3) AS Sample2;
3 SELECT GENERATE_ARRAY(10, 0, -3) AS Sample3;
4 SELECT GENERATE_ARRAY(4, 4, 10) AS Sample4;
5 SELECT GENERATE_ARRAY(10, 0, 3) AS Sample5;
6 SELECT GENERATE_ARRAY(5, NULL, 1) AS Sample6;
7 SELECT GENERATE_ARRAY(3, start, 5) AS Sample7;
8 FROM UNNEST([3, 4, 5]) AS start;
```

The 'RESULTS' tab is selected in the query results interface, showing the output:

Row	Sample1
1	1
	2
	3
	4
	5

A simple example of an array_Generate

2. In this sample, I have mentioned the start expression as 0 and end expression as 10 with a step expression as 3. Hence the values will increase by 3 until it reaches the end expression value.

The screenshot shows the Google Cloud BigQuery interface. The left sidebar displays the 'Explorer' section with a pinned project 'myfirstbigquery-356706'. The main area shows a query editor with the following SQL code:

```

1 SELECT GENERATE_ARRAY(1, 5) AS Sample1;
2 SELECT GENERATE_ARRAY(0, 10, 3) AS Sample2;
3 SELECT GENERATE_ARRAY(10, 0, -3) AS Sample3;
4 SELECT GENERATE_ARRAY(4, 4, 10) AS Sample4;
5 SELECT GENERATE_ARRAY(10, 0, 3) AS Sample5;
6 SELECT GENERATE_ARRAY(5, NULL, 1) AS Sample6;
7 SELECT GENERATE_ARRAY(start, 5) AS Sample7;
8 FROM UNNEST([3, 4, 5]) AS start;
9

```

The results tab shows the output for 'Sample2':

Row	Sample2
1	0
	3
	6
	9

A simple example of an array Generate

3. In this sample, I have mentioned the step value as -3, which would mean the start expression would decrease in value until it reaches the end expression.

The screenshot shows the Google Cloud BigQuery interface. The left sidebar displays the 'Explorer' section with a pinned project 'myfirstbigquery-356706'. The main area shows a query editor with the same SQL code as the previous example, but with a different step value:

```

1 SELECT GENERATE_ARRAY(1, 5) AS Sample1;
2 SELECT GENERATE_ARRAY(10, 0, 3) AS Sample2;
3 SELECT GENERATE_ARRAY(10, 0, -3) AS Sample3;
4 SELECT GENERATE_ARRAY(4, 4, 10) AS Sample4;
5 SELECT GENERATE_ARRAY(10, 0, 3) AS Sample5;
6 SELECT GENERATE_ARRAY(5, NULL, 1) AS Sample6;
7 SELECT GENERATE_ARRAY(start, 5) AS Sample7;
8 FROM UNNEST([3, 4, 5]) AS start;
9

```

The results tab shows the output for 'Sample3':

Row	Sample3
1	10
	7
	4
	1

A simple example of an array Generate

4. The following returns an array using the same value for the start and end expression.

The screenshot shows the Google Cloud BigQuery interface. In the top navigation bar, there are tabs for 'Aggregate functions | BigQuery', 'functions of an array in bigquery', 'Array functions | BigQuery', and 'Query results - BigQuery - myfirstbigquery'. The main area displays a query editor with the following SQL code:

```

1 SELECT GENERATE_ARRAY(1, 5) AS Sample1;
2 SELECT GENERATE_ARRAY(9, 10, 3) AS Sample2;
3 SELECT GENERATE_ARRAY(10, 0, -3) AS Sample3;
4 SELECT GENERATE_ARRAY(4, 4, 10) AS Sample4;
5 SELECT GENERATE_ARRAY(10, 0, 3) AS Sample5;
6 SELECT GENERATE_ARRAY(5, NULL, 1) AS Sample6;
7 SELECT GENERATE_ARRAY(start, 5) AS Sample7;
8 FROM UNNEST([3, 4, 5]) AS start;
9

```

The 'RUN' button is highlighted. Below the code, the 'Query results' section shows a table with one row labeled 'Sample4' containing the value '4'.

A simple example of an array_Generate

5.The following returns an empty array as the start expression is higher than the end expression. And we cannot decrease the value either because the step expression is a positive Integer.

The screenshot shows the Google Cloud BigQuery interface. In the top navigation bar, there are tabs for 'Aggregate functions | BigQuery', 'functions of an array in bigquery', 'Array functions | BigQuery', and 'Query results - BigQuery - myfirstbigquery'. The main area displays a query editor with the same SQL code as the previous screenshot:

```

1 SELECT GENERATE_ARRAY(1, 5) AS Sample1;
2 SELECT GENERATE_ARRAY(9, 10, 3) AS Sample2;
3 SELECT GENERATE_ARRAY(10, 0, -3) AS Sample3;
4 SELECT GENERATE_ARRAY(4, 4, 10) AS Sample4;
5 SELECT GENERATE_ARRAY(10, 0, 3) AS Sample5;
6 SELECT GENERATE_ARRAY(5, NULL, 1) AS Sample6;
7 SELECT GENERATE_ARRAY(start, 5) AS Sample7;
8 FROM UNNEST([3, 4, 5]) AS start;
9

```

An error message 'Only saved queries can be shared.' is displayed above the code. Below the code, the 'Query results' section shows a table with one row labeled 'Sample5' containing the value 'nuli'.

A simple example of an array_Generate

6. The following sample returns a null value because the end expression is a null value.

The screenshot shows the Google Cloud BigQuery interface. In the top navigation bar, there are several tabs: "Aggregate functions | BigQuery", "functions of an array in bigquery", "Array functions | BigQuery", and "Query results - BigQuery - myfirstbigquery". The main area is titled "Query results" and shows a single row of results. The query itself is:

```
1 SELECT GENERATE_ARRAY(1, 10) AS Sample1;
2 SELECT GENERATE_ARRAY(1, 10, -3) AS Sample2;
3 SELECT GENERATE_ARRAY(10, 0, -3) AS Sample3;
4 SELECT GENERATE_ARRAY(4, 4, 10) AS Sample4;
5 SELECT GENERATE_ARRAY(10, 0, 3) AS Sample5;
6 SELECT GENERATE_ARRAY(5, NULL, 1) AS Sample6;
7 SELECT GENERATE_ARRAY(start, 5) AS Sample7
8 FROM UNNEST([3, 4, 5]) AS start;
```

The results table has three columns: Row, Sample6, and Sample7. The first row shows "1" in the Row column and "null" in the Sample6 column. The Sample7 column is empty. The status bar at the bottom right indicates "30°C Rain showers" and the date "7/24/2022".

A simple example of an array_Generate

7. The following sample returns multiple.

The screenshot shows the Google Cloud BigQuery interface. The layout is identical to the previous one, with the same tabs and interface elements. The query is the same as in question 6:

```
1 SELECT GENERATE_ARRAY(1, 10) AS Sample1;
2 SELECT GENERATE_ARRAY(1, 10, -3) AS Sample2;
3 SELECT GENERATE_ARRAY(10, 0, -3) AS Sample3;
4 SELECT GENERATE_ARRAY(4, 4, 10) AS Sample4;
5 SELECT GENERATE_ARRAY(10, 0, 3) AS Sample5;
6 SELECT GENERATE_ARRAY(5, NULL, 1) AS Sample6;
7 SELECT GENERATE_ARRAY(start, 5) AS Sample7
8 FROM UNNEST([3, 4, 5]) AS start;
```

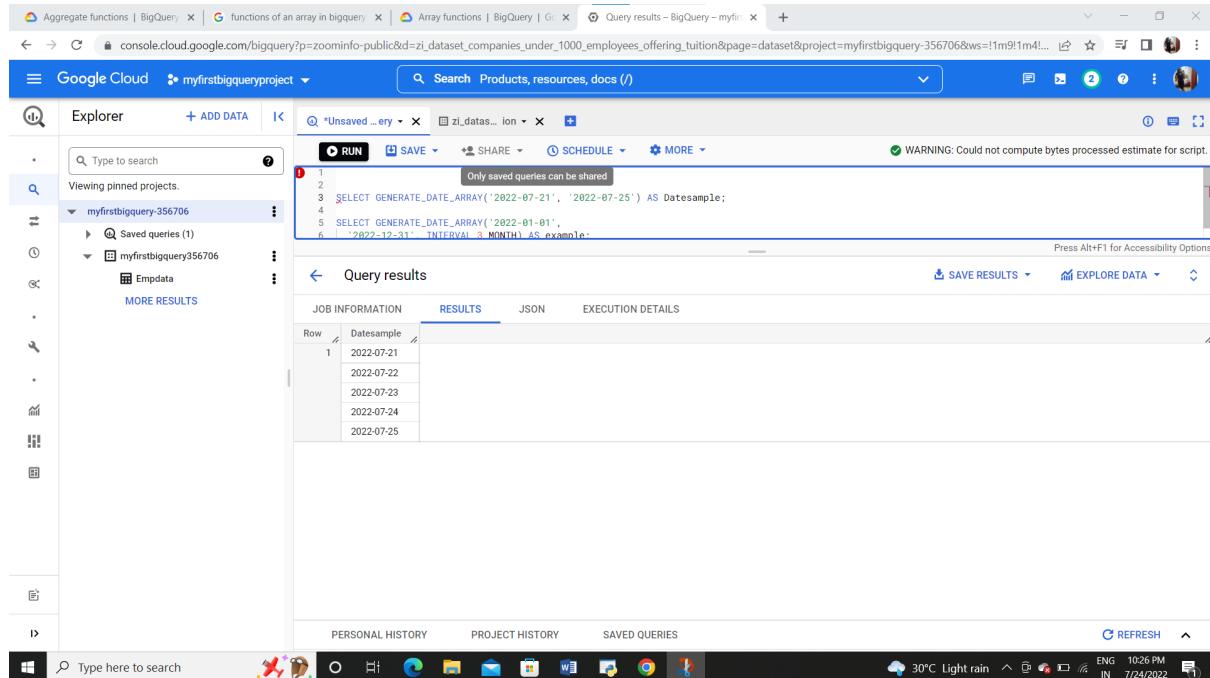
The results table now shows multiple rows. The first row has "1" in the Row column, "3" in Sample6, and "1" in Sample7. The second row has "2" in the Row column, "4" in Sample6, and "4" in Sample7. The third row has "3" in the Row column, "5" in Sample6, and "5" in Sample7. The Sample7 column is empty for the first two rows. The status bar at the bottom right indicates "30°C Rain showers" and the date "7/24/2022".

A simple example of an array_Generate

Generate Date Array in BigQuery:

This function returns an array of dates. Just like the generate function, we need a start and end expression for generating dates.

1. The following example returns the dates with an interval of 1.



The screenshot shows the Google Cloud BigQuery interface. On the left, the Explorer sidebar displays the project 'myfirstbigquery-356706' and a dataset 'myfirstbigquery356706' containing a single table 'Empdata'. In the center, the 'Query results' pane shows the output of a query. The query is:

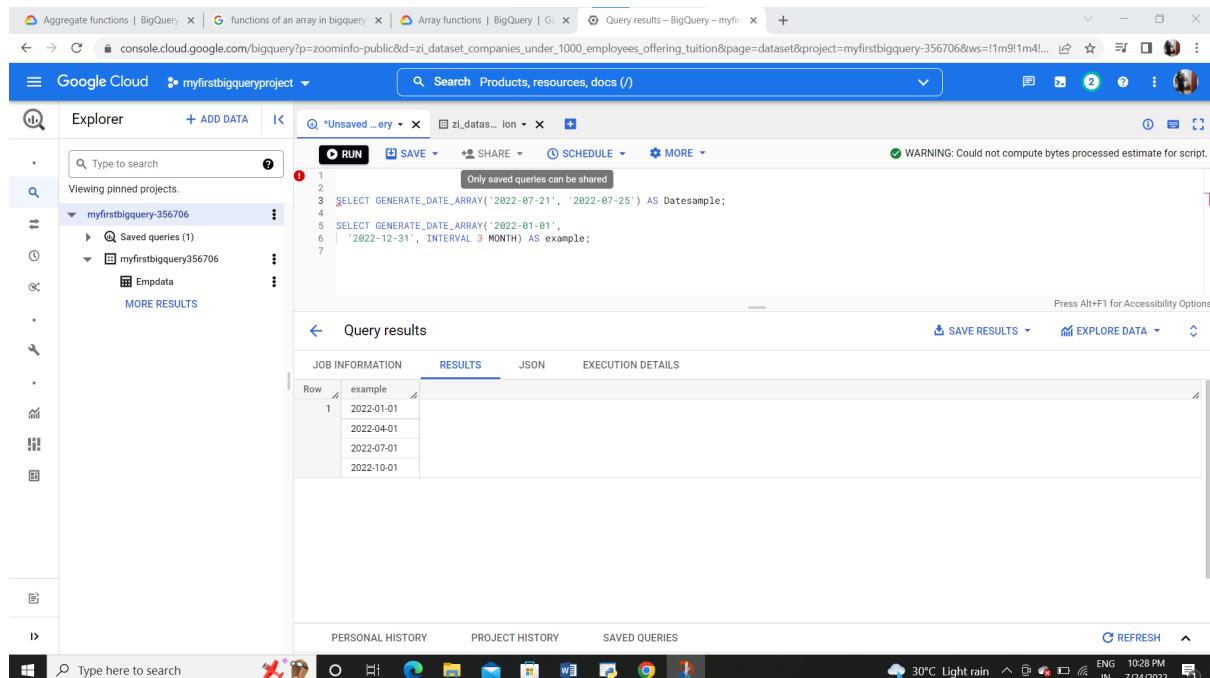
```
1 3 SELECT GENERATE_DATE_ARRAY('2022-07-21', '2022-07-25') AS Datesample;
2
3
4
5 SELECT GENERATE_DATE_ARRAY('2022-01-01',
6     '2022-12-31', INTERVAL 3 MONTH) AS example;
```

The 'RESULTS' tab is selected, displaying the following data:

Row	Datesample
1	2022-07-21
	2022-07-22
	2022-07-23
	2022-07-24
	2022-07-25

A simple example of an Generate_date_array

2. Let us find out the dates for a year with 3 month intervals.



The screenshot shows the Google Cloud BigQuery interface. The setup is identical to the previous one, with the project 'myfirstbigquery-356706' and dataset 'myfirstbigquery356706'. The 'Query results' pane shows the output of a query. The query is:

```
1
2
3 SELECT GENERATE_DATE_ARRAY('2022-07-21', '2022-07-25') AS Datesample;
4
5 SELECT GENERATE_DATE_ARRAY('2022-01-01',
6     '2022-12-31', INTERVAL 3 MONTH) AS example;
7
```

The 'RESULTS' tab is selected, displaying the following data:

Row	example
1	2022-01-01
	2022-04-01
	2022-07-01
	2022-10-01

A simple example of an Generate_date_array

STRUCTS

Structs are flexible containers of ordered fields each with a type (required) and a name (optional) – From Google cloud.

Unlike arrays, you can store multiple data types in structs and by that even arrays can be stored in structs. To put it simply, the struct is the root or the parent column with multiple child columns. For example, an employee has a qualification associated with different fields such as university, degree, start date and end date.

We use the keyword **STRUCT** to create the table.

Let's take a look at a simple example below:

The screenshot shows the Google Cloud BigQuery interface. On the left, the sidebar displays the project 'myfirstbigqueryproject' and datasets like 'myfirstbigquery-356706' and 'myfirstbigquery356706'. In the main area, a query is running, showing the following code:

```
1 SELECT
2   | STRUCT(430021 AS EmpID, 'Sasha Maria' AS Empname, 32 AS Empage)
```

The results table shows one row with the following data:

Row	f0_EmpID	f0_Empname	f0_Empage
1	430021	Sasha Maria	32

A simple example of struct

The best way to understand structs is to create one. Let us look at the restaurant database from Zomato. (Source: Kaggle)

The database has details about the restaurants location, address, rating, cuisines available, table booking, online delivery, etc., etc.,

I have filtered out the restaurants in Chennai and selected a few.

The screenshot shows the Google Cloud BigQuery interface with a query editor and results viewer. The query is as follows:

```
1 (
2   SELECT "That Madras Place" AS name, "Adayar" AS location,
3   | STRUCT(["European", "Italian", "Desserts"] AS cuisine, 500 AS price_range, FALSE AS delivery) AS details
4 UNION ALL
5   SELECT "Haunted" AS name, "Anna Nagar East" AS location,
6   | STRUCT(["North Indian", "Chinese", "Arabian"] AS cuisine, 1500 AS price_range, FALSE AS delivery) AS details
7 UNION ALL
8   SELECT "Pantry d'or" AS name, "Anna Nagar East" AS location,
9   | STRUCT(["Continental", "Cafe", "Italian", "Desserts"] AS cuisine, 2000 AS price_range, FALSE AS delivery) AS details
10 UNION ALL
11  SELECT "Palmshore" AS name, "Ashok Nagar" AS location,
12  | STRUCT(["North Indian", "Mughlai", "Chinese", "South Indian"] AS cuisine, 2000 AS price_range, FALSE AS delivery) AS details);
```

The results table shows the selected restaurants:

name	location	cuisine	price_range	delivery
"That Madras Place"	"Adayar"	["European", "Italian", "Desserts"]	500	False
"Haunted"	"Anna Nagar East"	["North Indian", "Chinese", "Arabian"]	1500	False
"Pantry d'or"	"Anna Nagar East"	["Continental", "Cafe", "Italian", "Desserts"]	2000	False
"Palmshore"	"Ashok Nagar"	["North Indian", "Mughlai", "Chinese", "South Indian"]	2000	False

Sample query for Struct creation

The screenshot shows the BigQuery results interface. At the top, there are buttons for RUN, SAVE, SHARE, SCHEDULE, and MORE. A note says "This query will process 0 B when run". Below the query editor, the results table has columns: JOB INFORMATION, RESULTS, JSON, and EXECUTION DETAILS. The RESULTS column displays the data, which includes a STRUCT field named 'details'. The 'details' field contains three nested fields: 'cuisine', 'price_range', and 'delivery'. Arrows point from the 'details' label to the 'cuisine' field, from 'cuisine' to 'price_range', and from 'price_range' to 'delivery', illustrating the structure of the data.

Row	name	location	detail...cuisine	detail...price...	detail...delivery
1	That Madras Place	Adyar	European	500	false
			Italian		
			Desserts		
2	Haunted	Anna Nagar East	North Indian	1500	false
			Chinese		
			Arabian		
3	Pantry d'or	Anna Nagar East	Continental	2000	false
			Cafe		
			Italian		
			Desserts		
4	Palmshore	Ashok Nagar	North Indian	2000	false
			Mughlai		
			Chinese		

Output showing struct containing different data types.

The screenshot shows the BigQuery schema editor for a table named 'anona06046d...'. The schema is defined as follows:

```

name: STRING NULLABLE
location: STRING NULLABLE
details: RECORD NULLABLE
  cuisine: STRING REPEATED
  price_range: INTEGER NULLABLE
  delivery: BOOLEAN NULLABLE

```

The 'details' field is highlighted with a green circle and labeled 'RECORD'. The 'cuisine' field under 'details' is highlighted with a blue circle and labeled 'REPEATED'. A callout box points to the 'REPEATED' label with the text 'STRUCT : NESTED FIELDS' and 'ARRAY: REPEATED FIELDS'. At the bottom, there are buttons for EDIT SCHEMA and VIEW ROW ACCESS POLICIES.

Let us check the schema of the table we just created. If the mode is record (which is nested fields) it means it is a Struct. But if the mode is Record/repeated fields, then it is an array.

Schema of the zomato_data table

Now, in the given database, I want to see which restaurant accepts online delivery. This is where the dot notation comes in handy. (WHERE = details.delivery = true). This will return all the restaurants that have online delivery. Below is the code for the dot notation.

The screenshot shows the BigQuery web interface. In the top navigation bar, there are tabs for 'myfirstbigquery356706' and 'zomato_data'. A search bar at the top right contains the placeholder 'Search Products, resources, docs (/)'. Below the search bar, there are buttons for 'RUN', 'SAVE', 'SHARE', 'SCHEDULE', and 'MORE'. A checkbox labeled 'This c' is checked. The main area displays a query in the code editor:

```
1 WITH online_delivery AS (
2   SELECT name, location, details.price_range AS price_range, cuisine
3   FROM `myfirstbigquery356706.zomato_data`, UNNEST(details.cuisine) AS cuisine
4   WHERE details.delivery = true
5 )
6 SELECT name, location, price_range, ARRAY_AGG(cuisine) AS cuisine
7 FROM online_delivery
8 GROUP BY name, location, price_range;
```

Below the code editor, the title 'Query results' is followed by a 'SAVE RESULTS' button. The results table has three columns: 'JOB INFORMATION', 'RESULTS', and 'EXECUTION DETAILS'. The 'RESULTS' column is currently selected. It shows one row for 'Pantry d'or' with the following details:

Row	name	location	price_range	cuisine
1	Pantry d'or	Anna Nagar East	2000	Continental Cafe Italian Desserts

Struct with Dot notation.

NESTED RECORDS:

We have created nested records that denormalized multiple levels without joins. As we can see in the picture below, all records exist in one table. Each record is nested to save storage but we can use the UNNEST function for expansion if needed. Maintaining nested records will eliminate the need for repeating data, using join or creating subtables.

Row	city	restau..name	restau..location	restau..cuisine	restau..price	restau..delivery
1	Chennai	That Madras Place	Adayar	European	900	false
				Italian		
				Desserts		
		Haunted	Anna Nagar East	North Indian	1000	true
				Chinese		
				Arabian		
		Pantry d'or	Anna Nagar East	Continental	1600	true
				Cafe		
				Italian		
				Desserts		
		Palmshore	Ashok Nagar	North Indian	2000	false
				Mughlai		
				Chinese		
				South Indian		
2	Coimbatore	Kuchi n Kream	Gandhipuram	Cafe	800	false
				Continental		
				Fast Food		
		Debutant	Gandhipuram	South Indian	1000	false
				Chettinad		
				North Indian		

Nested records.

Query for nested records.

```

1 CREATE OR REPLACE TABLE `myfirstbigquery356706.zomato2` AS (
2   SELECT "Chennai" AS city, [
3     STRUCT("That Madras Place" AS name, "Adayar" AS location, STRUCT(["European", "Italian", "Desserts"] AS cuisine, 900 AS price_range, False AS delivery) AS Details),
4     STRUCT("Haunted" AS name, "Anna Nagar East" AS location, STRUCT(["North Indian", "Chinese", "Arabian"] AS cuisine, 1000 AS price_range, True AS delivery) AS Details),
5     STRUCT("Pantry d'or" AS name, "Anna Nagar East" AS location, STRUCT(["Continental", "Cafe", "Italian", "Desserts"] AS cuisine, 1600 AS price_range, True AS delivery) AS Details),
6     STRUCT("Palmshore" AS name, "Ashok Nagar" AS location, STRUCT(["North Indian", "Mughlai", "Chinese", "South Indian"] AS cuisine, 2000 AS price_range, False AS delivery) AS Details)
7   UNION ALL
8   SELECT "Coimbatore" AS city, [
9     STRUCT("Kuchi n Kream" AS name, "Gandhipuram" AS location, STRUCT(["Cafe", "Continental", "Fast Food"] AS cuisine, 800 AS price_range, False AS delivery) AS Details),
10    STRUCT("Debutant" AS name, "Gandhipuram" AS location, STRUCT(["South Indian", "Chettinad", "North Indian"] AS cuisine, 1000 AS price_range, False AS delivery)) AS Details)
11  ]
12 )

```

We have looked at arrays and structs in BigQuery. We've explored the definitions and examples. We can sum up these use cases under two import points.

- Arrays support elements of the same data types.
- Structs support elements of different data types which also include arrays. Nested records in Bigquery are structs that help us save storage and eliminates the risk of repeated data. This will enhance the performance in return and also reduce the storage cost.
- Nested records in Bigquery are structs that helps us save storage and eliminates the risk of repeated data. This will enhance the performance in return and also reduces the storage cost

BigQuery Partitioning and Clustering

Partitioning

A partitioned table is a special table that is divided into segments, called partitions, that make it easier to manage and query your data. Dividing a large table into smaller partitions, improves query performance and control costs by reducing the number of bytes read by a query.

Partitioning is used under the following circumstances:

You want to know query costs before a query runs. Partition pruning is done before the query runs, so you can get the query cost after partitioning pruning through a [dry run](#). Cluster pruning is done when the query runs, so the cost is known only after the query finishes.

You need partition-level management. For example, you want to set a partition expiration time, load data to a specific partition, or delete partitions.

You want to specify how the data is partitioned and what data is in each partition. For example, you want to define time granularity or define the ranges used to partition the table for integer range partitioning.

We can partition BigQuery tables by 3 ways -:

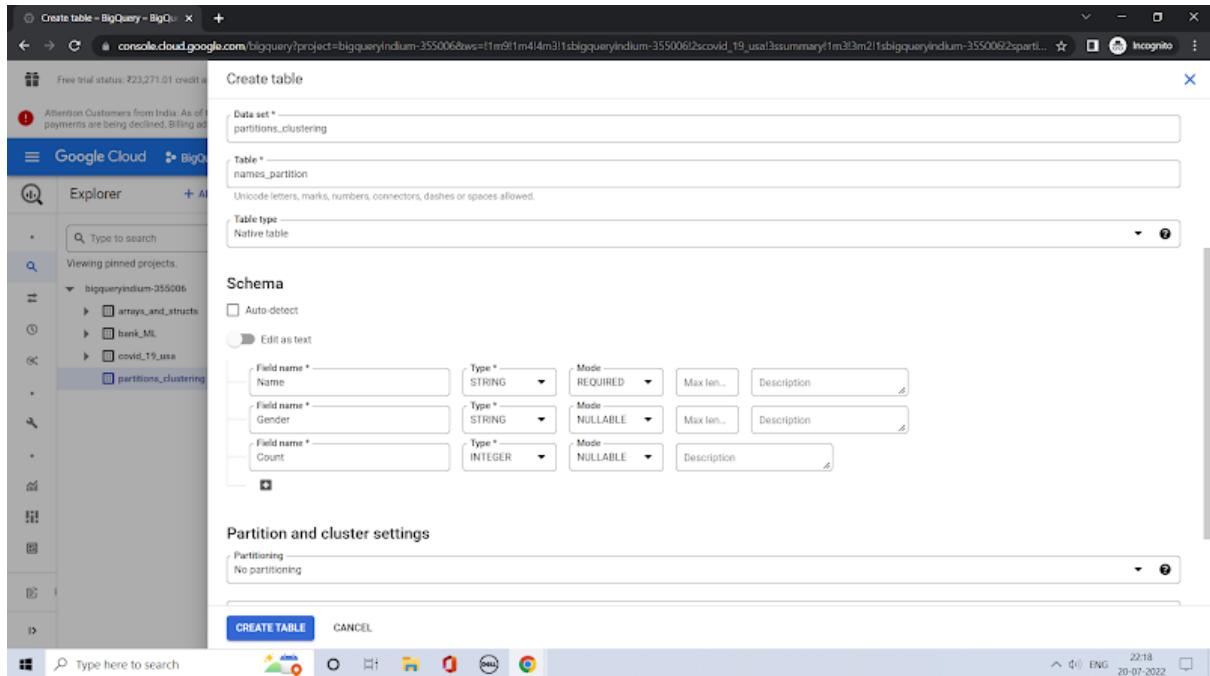
Time-unit column - Tables are partitioned based on a [TIMESTAMP](#), [DATE](#), or [DATETIME](#) column in the table.

Ingestion time - Tables are partitioned based on the timestamp when BigQuery ingests the data.

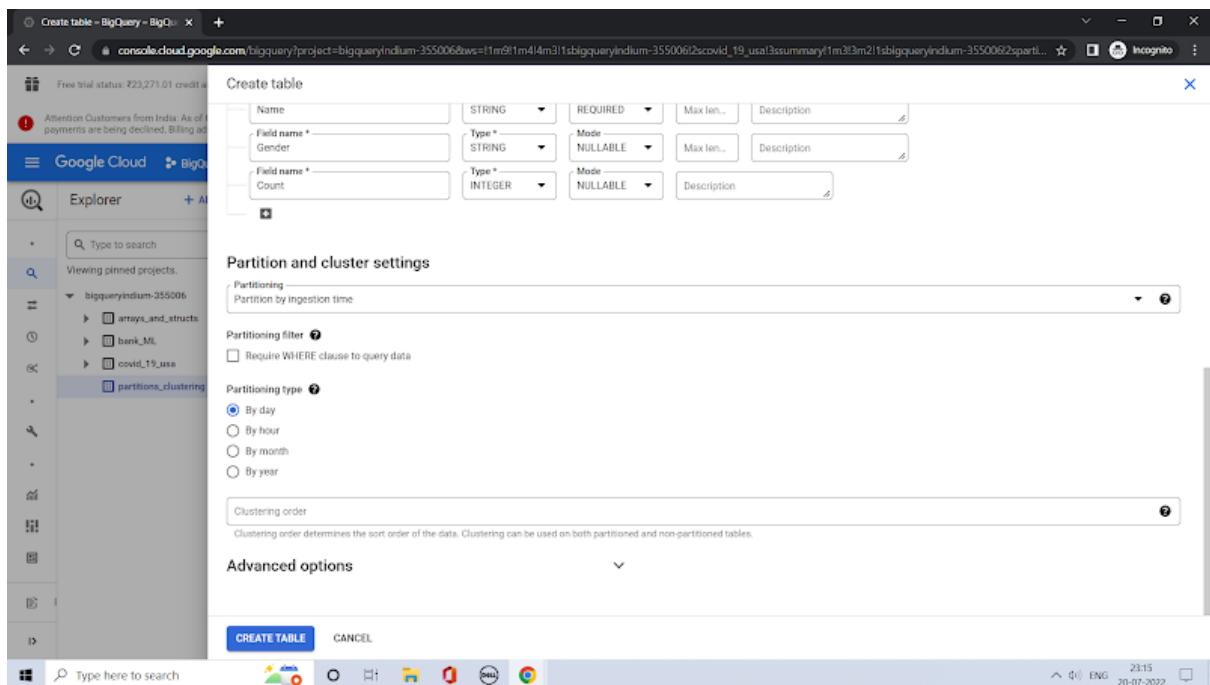
Integer range - Tables are partitioned based on an integer column.

Now, let's understand this with an example.

Take the dataset called names. It has 3 fields - **Name, Gender and Count** (number of people with that particular name).



Now let's partition it based on ingestion time (here let's take partitioning type as day).



Google Cloud BigQuery - BigQueryIndium - Google Sheets

How to Use Partitioned Tables in BigQuery

Loading CSV data from Cloud Storage

DISMISS ACTIVATE

Attention Customers from India: As of October 1st, due to regulatory changes by the Reserve Bank of India, we will not be able to charge your card for recurring payments if you have not set up an e-mandate or if the transaction is above 5000 INR. If your payments are being declined, Billing administrators will have to log into their Payments account and manually resubmit the payments. Read more.

DISMISS

Google Cloud BigQueryIndium

Search Products, resources, docs (/)

Explorer + ADD DATA

name_partitioning

SCHEMA DETAILS PREVIEW

Row	Name	Gender	Count
1	Emma	F	18688
2	Olivia	F	17921
3	Ava	F	14924
4	Isabella	F	14464
5	Sophia	F	13928
6	Charlotte	F	12940
7	Mia	F	12642
8	Amelia	F	12301
9	Harper	F	10582
10	Evelyn	F	10376
11	Aaliyah	F	9796
12	Emily	F	8656

name_partitioning created. GO TO TABLE

RESULTS HISTORY SAVED QUERIES

RESULTS per page: 50 1 – 50 of 32033

REFRESH

NOTE :- The tables can be differentiated as partitioned or not by looking at their symbols in front of their names.

Now, let's perform a query on the partitioned table **name_partitioning**.

Google Cloud BigQuery - BigQueryIndium - Google Sheets

How to Use Partitioned Tables in BigQuery

Loading CSV data from Cloud Storage

DISMISS ACTIVATE

Attention Customers from India: As of October 1st, due to regulatory changes by the Reserve Bank of India, we will not be able to charge your card for recurring payments if you have not set up an e-mandate or if the transaction is above 5000 INR. If your payments are being declined, Billing administrators will have to log into their Payments account and manually resubmit the payments. Read more.

DISMISS

Google Cloud BigQueryIndium

Search Products, resources, docs (/)

Explorer + ADD DATA

*Unsaved...y 2

RUN SAVE SHARE SCHEDULE MORE

This query will process 254.08 KB when run.

```
1 select _PARTITIONTIME as pt, _PARTITIONDATE as pd, name from `bigqueryindium-355006.partitions_clustering.name_partitioning`
```

Press Alt+F1 for accessibility options.

Query results

JOB INFORMATION RESULTS JSON EXECUTION DETAILS

Row	pt	pd	name
1	2022-07-20 00:00:00 UTC	2022-07-20	Emma
2	2022-07-20 00:00:00 UTC	2022-07-20	Olivia
3	2022-07-20 00:00:00 UTC	2022-07-20	Ava
4	2022-07-20 00:00:00 UTC	2022-07-20	Isabella
5	2022-07-20 00:00:00 UTC	2022-07-20	Sophia
6	2022-07-20 00:00:00 UTC	2022-07-20	Charlotte
7	2022-07-20 00:00:00 UTC	2022-07-20	Mia
8	2022-07-20 00:00:00 UTC	2022-07-20	Amelia
9	2022-07-20 00:00:00 UTC	2022-07-20	Harper
10	2022-07-20 00:00:00 UTC	2022-07-20	Evelyn

RESULTS per page: 50 1 – 50 of 32033

SAVE RESULTS EXPLORE DATA

PERSONAL HISTORY PROJECT HISTORY SAVED QUERIES

REFRESH

NOTE :- Whenever we query partition time and partition date we have to use aliases (in the above case, pt, pd).

The above dataset can also be partitioned by its count.

This screenshot shows the 'Create table' dialog in the Google Cloud BigQuery console. The 'Partition and cluster settings' section is open, specifically the 'Partitioning' tab. It shows a 'Count' partitioning rule applied to the 'Date' field. The 'Start' value is set to 10, and the 'End' value is set to 100. The 'Interval' value is set to 10. Below this, the 'Clustering order' section is visible. At the bottom, there are 'CREATE TABLE' and 'CANCEL' buttons.

Now, let's take another dataset to understand partitioning wrt date. It has the following schema.

This screenshot shows the 'Create table' dialog in the Google Cloud BigQuery console. The 'Schema' section is displayed, showing a list of fields: Name (Type STRING, Mode REQUIRED), Gender (Type STRING, Mode NULLABLE), Count (Type INTEGER, Mode NULLABLE), and Date (Type DATE, Mode NULLABLE). Below the schema, the 'Partition and cluster settings' section is shown, indicating 'No partitioning'. At the bottom, there are 'CREATE TABLE' and 'CANCEL' buttons.

Now, we also have the option of partition by date.

Its preview.

NOTE :- In the above preview row 1 does not throw an error even though it does not have a Date value as it considers it as part of NULL partition.

CLUSTERING

Clustering is another way of organising data which stores one next to the other all those rows that share similar values in the chosen clustering columns. This process increases the query efficiency and performance. Note that BigQuery supports this feature only on partitioned tables.

Clustering should be used under the following circumstances :-

We don't need strict cost guarantees before running the query.

We need more granularity than partitioning alone allows. To get clustering benefits in addition to partitioning benefits, we can use the same column for both partitioning and clustering.

Queries commonly use filters or aggregation against multiple particular columns.

The cardinality of the number of values in a column or group of columns is large.

Now, let's take up the same names dataset for understanding clustering.

The screenshot shows the 'Create table' dialog in the Google Cloud BigQuery interface. The 'Name' field is set to 'Gender'. The 'Field name *' section contains three fields: 'Gender' (Type: STRING, Mode: REQUIRED), 'Count' (Type: INTEGER, Mode: NULLABLE), and 'Name' (Type: STRING, Mode: NULLABLE). In the 'Partition and cluster settings' section, 'Partitioning' is set to 'Partition by ingestion time'. Under 'Partitioning type', 'By day' is selected. In the 'Clustering order' section, 'Gender' is listed. At the bottom, there are 'CREATE TABLE' and 'CANCEL' buttons.

NOTE :- We can give upto 4 fields to define clustering order.

The screenshot shows the Google Cloud BigQuery interface. The left sidebar displays pinned projects under 'bigqueryindium-355006'. The main panel shows the 'name_clustering' table details. The table is described as a partitioned table, specifically mentioning 'PARTITIONED BY DAY' and 'CLUSTERED BY Gender'. The schema includes columns for Name, Gender, and Count. The table has 32,033 rows.

Row	Name	Gender	Count
1	Clara	F	256
2	Daniela	F	256
3	Natalia	F	256
4	Chandler	F	256
5	Riley	F	256
6	Ainsley	F	768
7	Emma	F	18688
8	Nathalia	F	257
9	Winnie	F	257
10	Luz	F	257
11	Poppy	F	513
12	Stella	F	5121

Now, any query that is searched wrt **gender** will be very fast compared to a regular query.

Its preview.

The screenshot shows the Google Cloud BigQuery interface. The left sidebar displays pinned projects under 'bigqueryindium-355006'. The main panel shows the 'name_clustering' table preview. The table has 32,033 rows and 4 columns: Row, Name, Gender, and Count. The preview shows the first 12 rows.

Row	Name	Gender	Count
1	Clara	F	256
2	Daniela	F	256
3	Natalia	F	256
4	Chandler	F	256
5	Riley	F	256
6	Ainsley	F	768
7	Emma	F	18688
8	Nathalia	F	257
9	Winnie	F	257
10	Luz	F	257
11	Poppy	F	513
12	Stella	F	5121

