Google BigQuery: The Definitive Guide

Table of contents

Chapter 1: What Is Google BigQuery?

What Is Google BigQuery?

"Google BigQuery is a ***serverless****,* ***highly scalable data warehouse*** *that comes with a* ***built-in query engine****. The query engine is capable of running SQL queries on terabytes of data in a matter of seconds, and petabytes in only minutes."*

- Makes it possible to have data easily available to all departments (i.e., due to the possibility of using SQL queries)

Drawbacks of using RDBMS:

- OLTP databases are designed for data consistency, "complex" queries pose a tradeoff because they slow down performance -> not built for adhoc queries

Drawbacks of using MapReduce frameworks:

- Your organization will need to become expert in managing, monitoring, and provisioning Hadoop clusters.

BigQuery is the best of both worlds: Serverless (no infrastructure maintenance) and being able to use SQL

- BQ makes it possible to join data from different datasets

- BQ uses columnar storage format

- *"Because BigQuery separates compute and storage, it is possible to run BigQuery SQL queries against CSV (or JSON or Avro) files that are stored* *as-is on Google Cloud Storage; this capability is called **federated querying**."*

- BQ gives the option of creating visualizations directly in the UI

- Supports SQL:2011

- Your datasets can be shared using Google Cloud **Identity and Access Management (IAM)**

- Why was SQL the chosen language? One reason:

- Finally, and quite important for a choice of a cloud computation language, SQL is not "Turing complete" in a key way: it always terminates. This way nobody can write an infinite query and monopolize all compute resources.

What Makes BigQuery Possible?

- BigQuery is the first data warehouse to be a scale-out solution

Separation of Compute and Storage

- The size of data you store is independent of how many resources you can use to analyse it and vice verca

Storage and Networking Infrastructure

- Googles inhouse distributed file system and networking setup

Managed Storage

- BQ managaes storage for you (abstraction is always the table, not the file)
- BigQuery ensures that all the data held within a table has a consistent schema and enforces a proper migration path for historical data
- Data is validated at entry time, therefore you will never encounter a validation error at readtime
- DONWSIDE: It is more difficult to access the data directly -> therefore BQ provides a structured parallel API to read data.

Integration with Google Cloud Platform

StackDriver: monitoring and audit logs - > understand BQ usage for an organization
Cloud Dataproc: Use Apache Spark to read, process and write to BQ
Federated queries: Allows data to query data stored in Google Cloud Storage, Cloud SQL, BigTable (NoSQL), Spanner (distributed database) and Google Drive.

Google Cloud Data Loss Prevention: Manage sensitive data

AutoML: Provide ML models

Data Studio: Explore data with visualizations

Cloud Pub/Sub: Ingest streaming data

Cloud Scheduler and Cloud Functions: Scheduling or triggering of BigQuery queries

Chapter 2 Query Essentials

Query Essentials

- BigQuery supports a dialect of SQL that is compliant with SQL:2011.
- More data scanned -> higher costs
- In the past it was recommended to store everything in a denormalized form, but since certain updates, the performance will be roughly the same as with star data
- NOTE: Note that using a constrains only the amount of data displayed to you and not the amount of data the query engine needs to process.
- The clause operates on the columns in the clause; thus, it is not possible to reference aliases from the list in the clause -> therefore some transformations must be repeated in the where clause:

```
SELECT
  gender, tripduration / 60 AS minutes
FROM
  `bigquery-public-data`.new_york_citibike.citibike_trips
WHERE (tripduration / 60) < 10
LIMIT 5
```

- Use REPLACE to replace a column with a transformation:

```
SELECT
  * REPLACE(num_bikes_available + 5 AS num_bikes_available)
FROM
  `bigquery-public-data`.new_york_citibike.citibike_stations
```

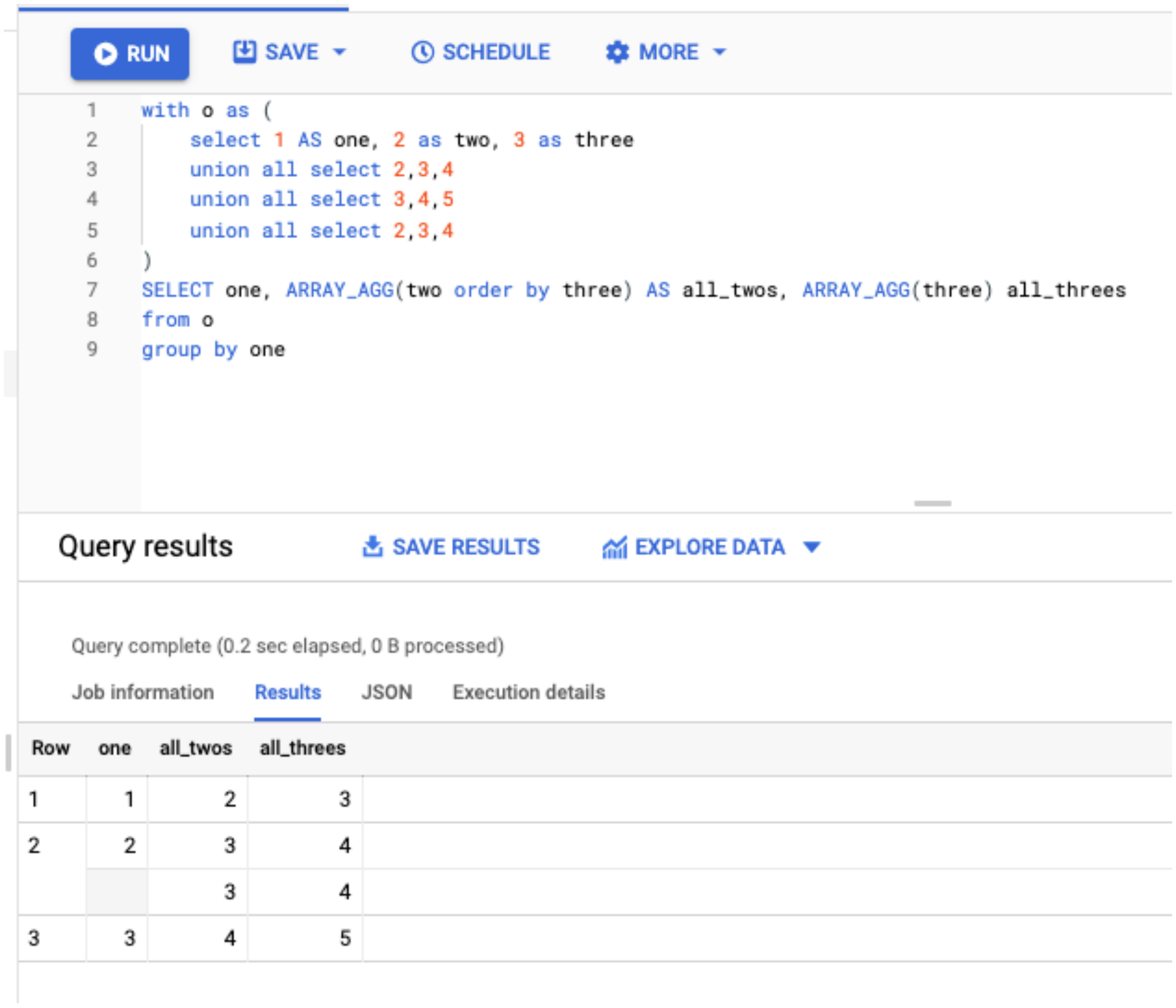
- Processing 0 bytes equates to 0 charges from BigQuery
- Create rows from UNION ALL SELECT statements:

```
WITH example AS (
  SELECT 'Sat' AS day, 1451 AS numrides, 1018 AS oneways
  UNION ALL SELECT 'Sun', 2376, 936
  UNION ALL SELECT 'Mon', 1476, 736
)

SELECT * from example
WHERE numrides < 2000
```

- Leading commas in queries makes it easy to comment out lines in a query and improves development speed

Arrays

  - An array is an ordered list of non-null elements
  - BigQuery can ingest hierarchical formats such as JSON, which can contain lists which would result in an array in BQ
- **ARRAY_AGG**:
  This can be used to keep all data in group by statements without aggregating. It will be perserved as its own column:



```sql
with o as (
    select 1 AS one, 2 as two, 3 as three
    union all select 2,3,4
    union all select 3,4,5
    union all select 2,3,4
)
SELECT one, ARRAY_AGG(two order by three) AS all_twos, ARRAY_AGG(three) all_threes
from o
group by one
```

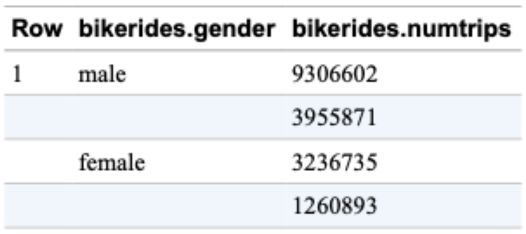
Query complete (0.2 sec elapsed, 0 B processed)

Row	one	all_twos	all_threes
1	1	2	3
2	2	3	4
		3	4
3	3	4	5

- Array of STRUCT: A is a group of fields in order. Each of the field should have a name assigned to them or they will get an arbitrary one, example:

```
SELECT
  [
     STRUCT('male' as gender, [9306602, 3955871] as numtrips)
     , STRUCT('female' as gender, [3236735, 1260893] as numtrips)
  ] AS bikerides
```

This results in the following:

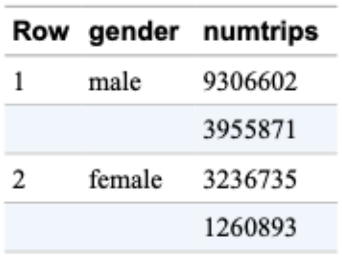


Row	bikerides.gender	bikerides.numtrips
1	male	9306602
		3955871
	female	3236735
		1260893

- When selecting from an array by index, the index starts from 0
- Fetch length of array with **ARRAY_LENGTH()**
- Fetch single item by selecting an
- **UNNEST()** returns the elements of an array as rows

```
SELECT * from UNNEST(
  [
     STRUCT('male' as gender, [9306602, 3955871] as numtrips)
     , STRUCT('female' as gender, [3236735, 1260893] as numtrips)
  ])
```

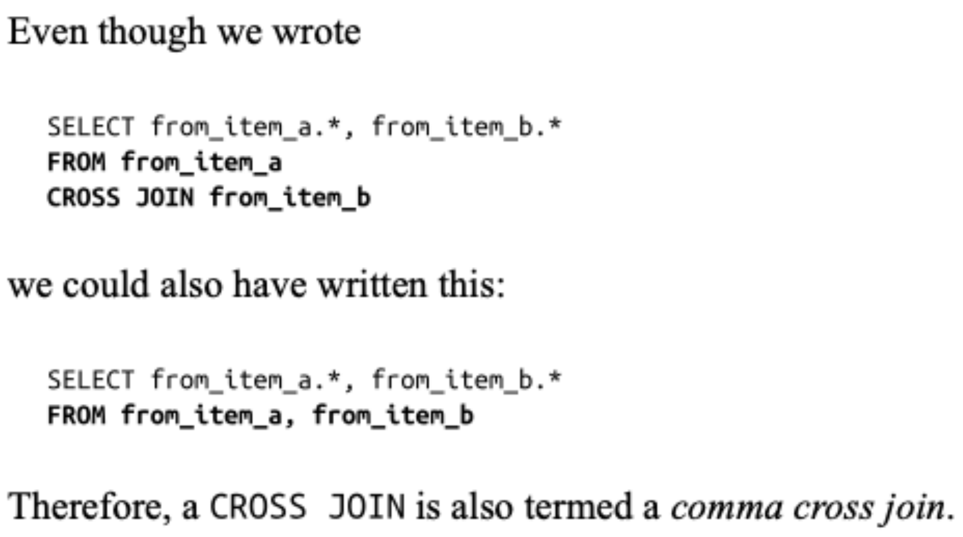
This yields the following:



Row	gender	numtrips
1	male	9306602
		3955871
2	female	3236735
		1260893

JOINS

- There is a special komma cross join syntax available in bigquery



Even though we wrote

```
SELECT from_item_a.*, from_item_b.*
FROM from_item_a
CROSS JOIN from_item_b
```

we could also have written this:

```
SELECT from_item_a.*, from_item_b.*
FROM from_item_a, from_item_b
```

Therefore, a `CROSS JOIN` is also termed a *comma cross join*.

Other non notable things discussed in this chapter:

- Aliasing with “AS”
- Filtering with WHERE
- Using „with“ for subqueries
- Aggregates (SUM, AVG, COUNT)
- HAVING keyword
- DISTINCT
- Inner joins

Chapter 3. Data Types, Functions, and Operators

Types of functions

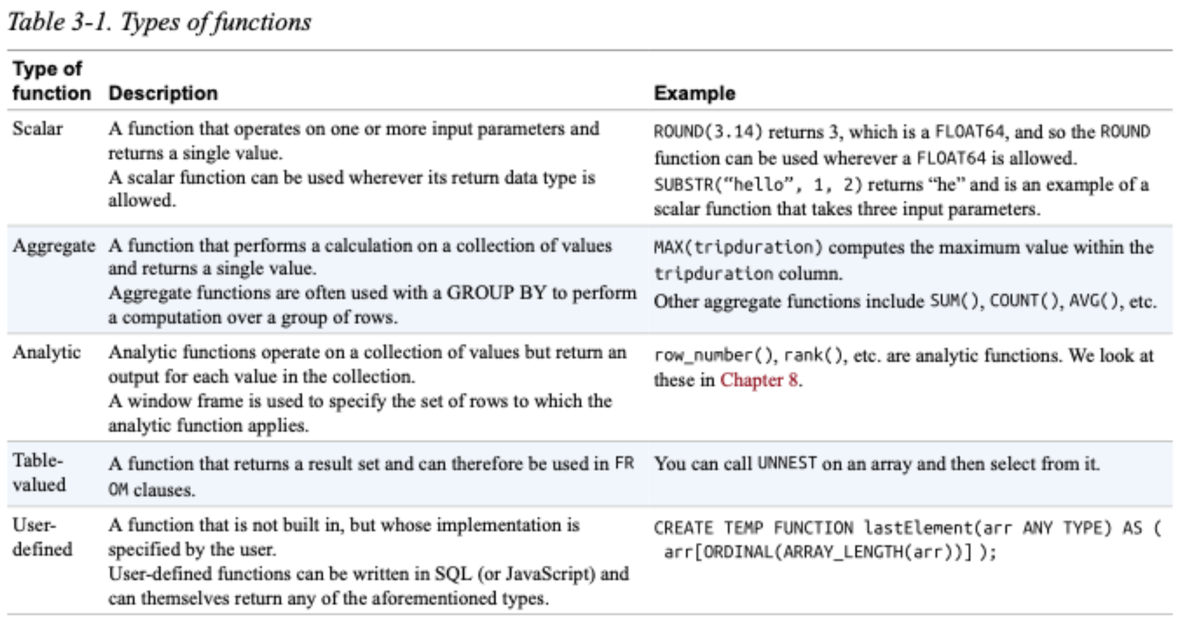
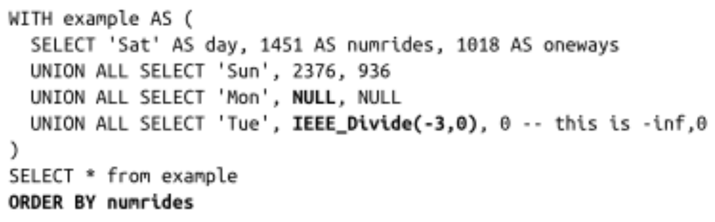


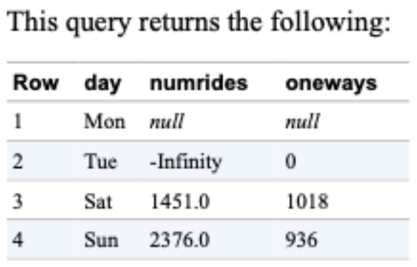
Table 3-1. Types of functions

Type of function	Description	Example
Scalar	A function that operates on one or more input parameters and returns a single value. A scalar function can be used wherever its return data type is allowed.	`ROUND(3.14)` returns 3, which is a `FLOAT64`, and so the `ROUND` function can be used wherever a `FLOAT64` is allowed. `SUBSTR("hello", 1, 2)` returns "he" and is an example of a scalar function that takes three input parameters.
Aggregate	A function that performs a calculation on a collection of values and returns a single value. Aggregate functions are often used with a GROUP BY to perform a computation over a group of rows.	`MAX(tripduration)` computes the maximum value within the `tripduration` column. Other aggregate functions include `SUM()`, `COUNT()`, `AVG()`, etc.
Analytic	Analytic functions operate on a collection of values but return an output for each value in the collection. A window frame is used to specify the set of rows to which the analytic function applies.	`row_number()`, `rank()`, etc. are analytic functions. We look at these in Chapter 8.
Table-valued	A function that returns a result set and can therefore be used in `FROM` clauses.	You can call `UNNEST` on an array and then select from it.
User-defined	A function that is not built in, but whose implementation is specified by the user. User-defined functions can be written in SQL (or JavaScript) and can themselves return any of the aforementioned types.	`CREATE TEMP FUNCTION lastElement(arr ANY TYPE) AS ( arr[ORDINAL(ARRAY_LENGTH(arr))] );`

- IEEE_Divide() function follows the standard set by the Institute of Electrical and Electronics Engineers (IEEE) and returns a special floating-point number called Not-a-Number ( ) when a division by zero is attempted
- SAFE Functions: You can make any scalar function return instead of raising an error by prefixing it with "SAFE."
- NULL is always smaller than -inf or NaN but also always results in False in comparisons:

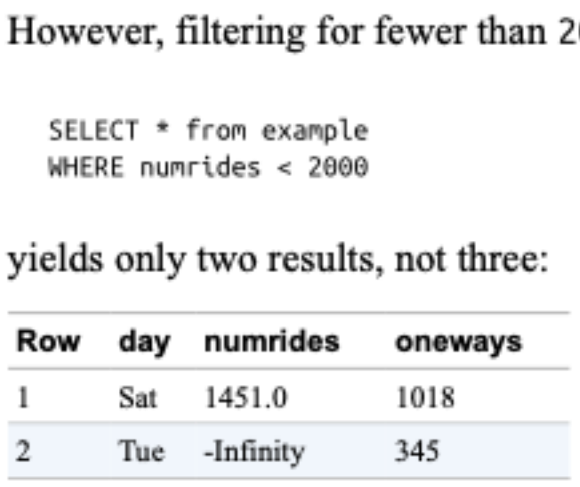


```
WITH example AS (
  SELECT 'Sat' AS day, 1451 AS numrides, 1018 AS oneways
  UNION ALL SELECT 'Sun', 2376, 936
  UNION ALL SELECT 'Mon', NULL, NULL
  UNION ALL SELECT 'Tue', IEEE_Divide(-3,0), 0 -- this is -inf,0
)
SELECT * from example
ORDER BY numrides
```



This query returns the following:

Row	day	numrides	oneways
1	Mon	*null*	*null*
2	Tue	-Infinity	0
3	Sat	1451.0	1018
4	Sun	2376.0	936

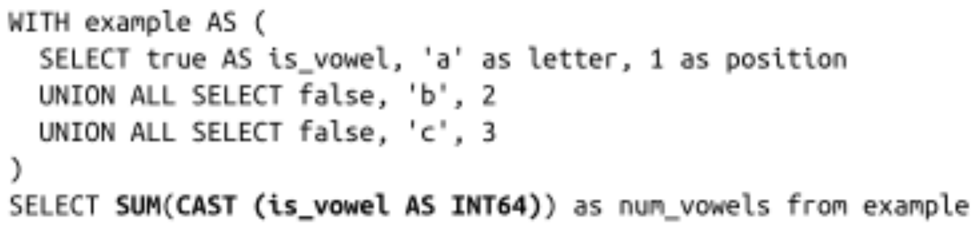

However, filtering for fewer than 2000 rides with

```
SELECT * from example
WHERE numrides < 2000
```

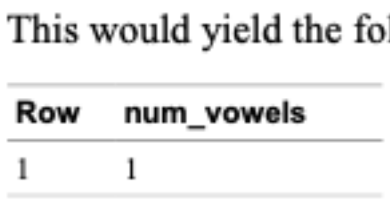
yields only two results, not three:

Row	day	numrides	oneways
1	Sat	1451.0	1018
2	Tue	-Infinity	345

- **Numeric** datatype offers a precise representation of floats, while int64 and float64 are designed to be fast and flexible. NUMERIC uses 16bytes of storage while the others use 64 bit
- You can't count booleans so transform them to ints before:

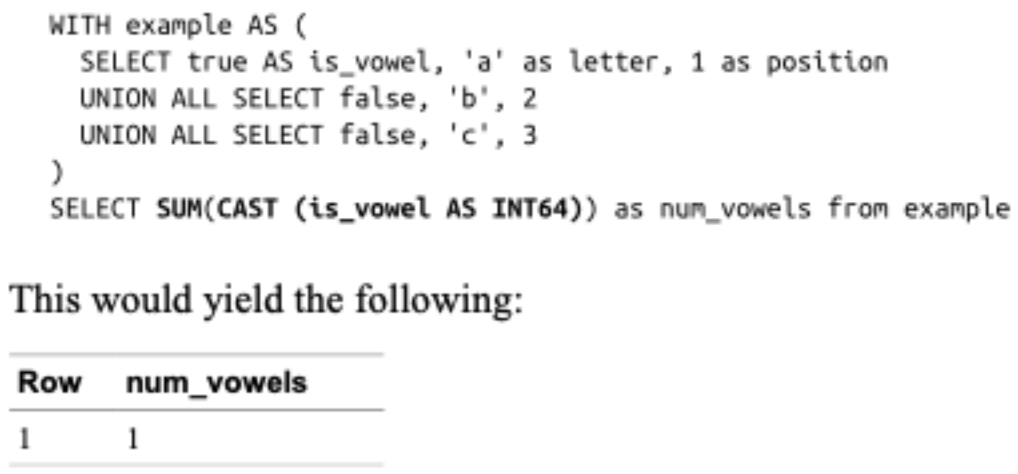


```
WITH example AS (
  SELECT true AS is_vowel, 'a' as letter, 1 as position
  UNION ALL SELECT false, 'b', 2
  UNION ALL SELECT false, 'c', 3
)
SELECT SUM(CAST (is_vowel AS INT64)) as num_vowels from example
```


This would yield the following:

Row	num_vowels
1	1

- String manipuation functions:



```
WITH example AS (
  SELECT true AS is_vowel, 'a' as letter, 1 as position
  UNION ALL SELECT false, 'b', 2
  UNION ALL SELECT false, 'c', 3
)
SELECT SUM(CAST (is_vowel AS INT64)) as num_vowels from example
```

This would yield the following:

Row	num_vowels
1	1

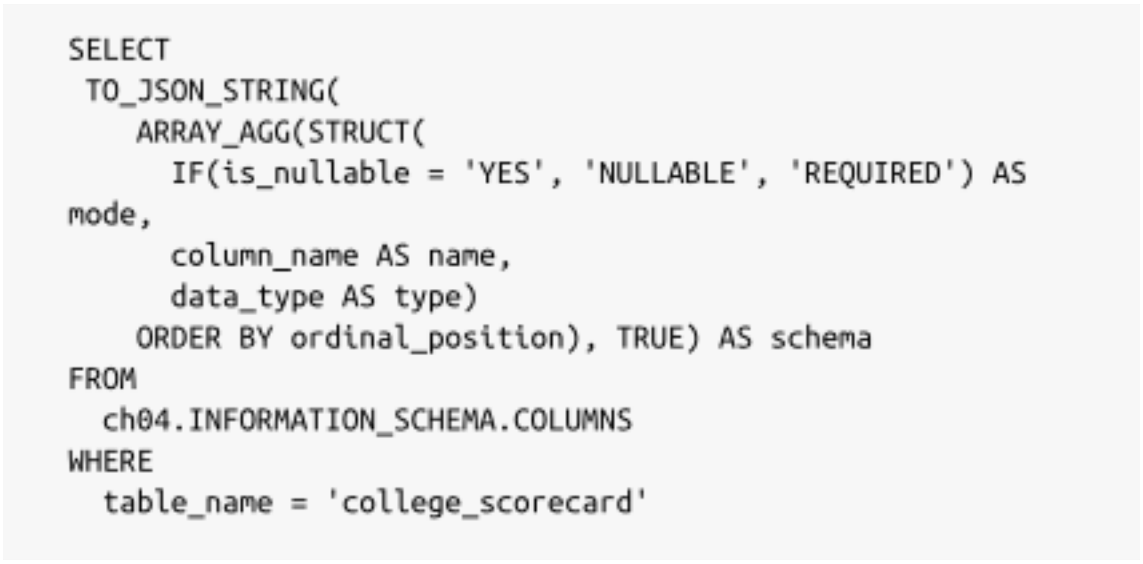
- DATETIME is a TIMESTAMP in a specific timezone and a DATETIME represents one exact point in time regardless of timezone

Chapter 4. Loading Data into BigQuery

- Load data from cloud shell: https://console.cloud.google.com/cloudshell

  Use **bq load** command

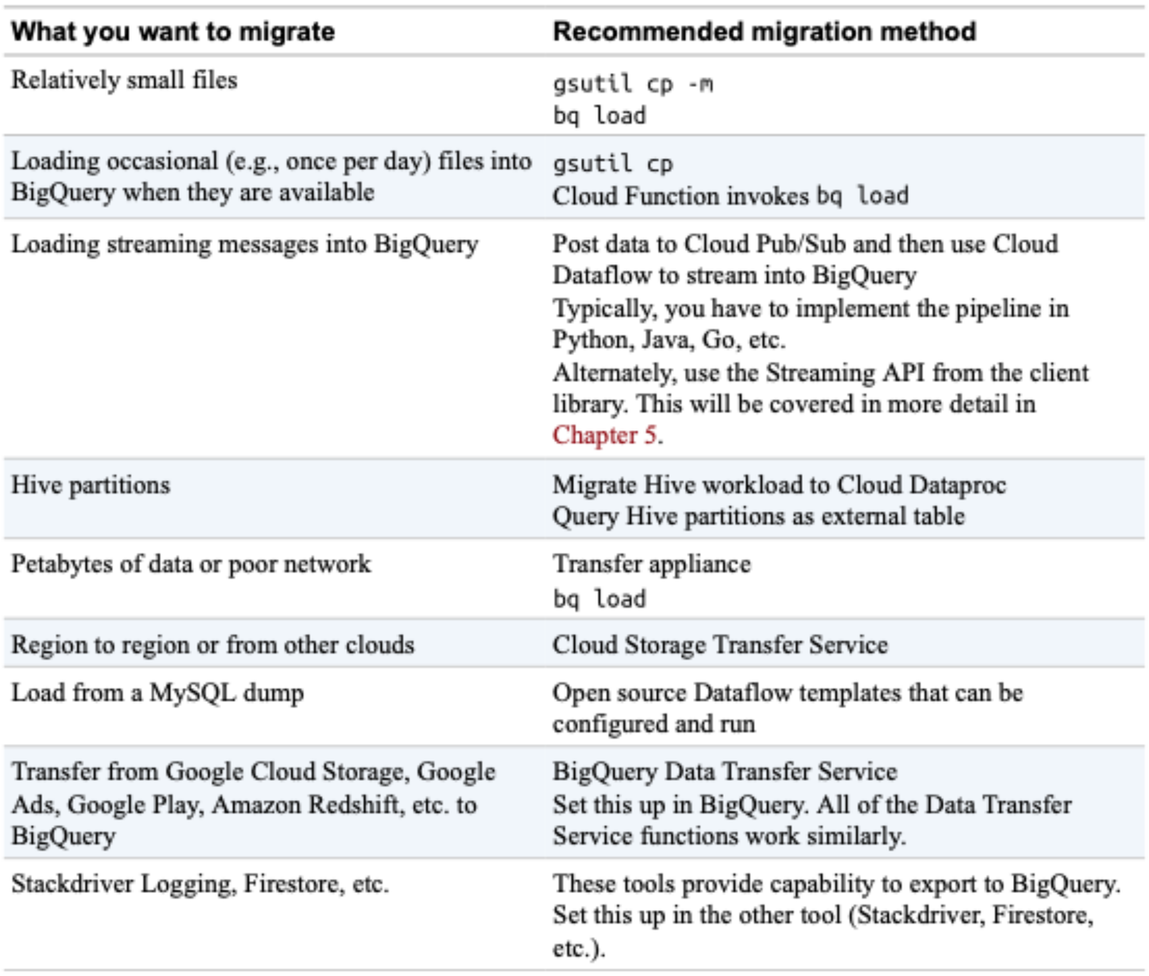
- Post upload options:
  - relax column to become nullable
  - expire table
  - add columns
- Best practice for loading data: For production workloads, insist on the data type for a column by specifying it at the time of load
- How to fetch the (JSON string) of of a table schema



```
SELECT
 TO_JSON_STRING(
    ARRAY_AGG(STRUCT(
      IF(is_nullable = 'YES', 'NULLABLE', 'REQUIRED') AS
mode,
      column_name AS name,
      data_type AS type)
    ORDER BY ordinal_position), TRUE) AS schema
FROM
  ch04.INFORMATION_SCHEMA.COLUMNS
WHERE
  table_name = 'college_scorecard'
```

- Book example of loading table with a modified schema definition
    - Load data into BG
    - get JSON schema definition as shown above
    - use this new schema to reload the data with the correct schema by providing it in the **bq load** command
- Loading Data Efficiently: The most efficient format is Avro
- BigQuery does not charge for loading data. Ingestion happens on a set of workers that is distinct from the cluster providing the slots used for querying
- **Federated queries** = query external data sources (Google Cloud Storage, Cloud Bigtable, Cloud SQL, and Google Drive)
- BigQuery can directly query data in Cloud Bigtable.
- Temporary tables can not be shared, permanent tables can be shared
- It is possible to map a Cloud Bigtable table to BigQuery and query it from there (once the queries are performed by the Bigtable cluster, performance will be limited by that cluster), therefore it is worth considering setting up an ELT pipeline to load data into BG and analyse it there
- It's possible to transfer data via recurring data loads from a variety of sources.
- Scheduled queries are built on top of the Data Transfer Service, so many of the features are similar
- BigQuery works best on datasets that are within the datacenter and behind the Google Cloud firewall

Migrating data to BigQuery from different sources



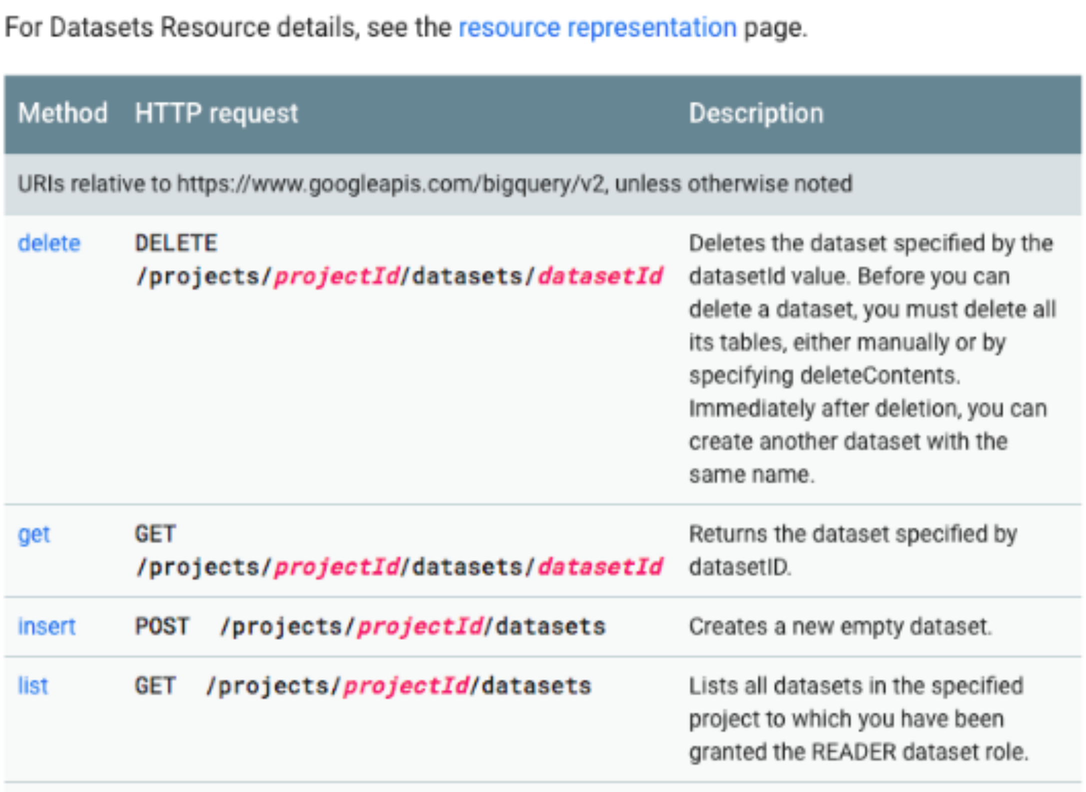
What you want to migrate	Recommended migration method

Relatively small files	`gsutil cp -m`
	`bq load`

Loading occasional (e.g., once per day) files into BigQuery when they are available	`gsutil cp` Cloud Function invokes `bq load`
Loading streaming messages into BigQuery	Post data to Cloud Pub/Sub and then use Cloud Dataflow to stream into BigQuery Typically, you have to implement the pipeline in Python, Java, Go, etc. Alternately, use the Streaming API from the client library. This will be covered in more detail in Chapter 5.
Hive partitions	Migrate Hive workload to Cloud Dataproc Query Hive partitions as external table
Petabytes of data or poor network	Transfer appliance `bq load`
Region to region or from other clouds	Cloud Storage Transfer Service
Load from a MySQL dump	Open source Dataflow templates that can be configured and run
Transfer from Google Cloud Storage, Google Ads, Google Play, Amazon Redshift, etc. to BigQuery	BigQuery Data Transfer Service Set this up in BigQuery. All of the Data Transfer Service functions work similarly.
Stackdriver Logging, Firestore, etc.	These tools provide capability to export to BigQuery. Set this up in the other tool (Stackdriver, Firestore, etc.).

Chapter 5. Developing with BigQuery

- The recommended approach for accessing BigQuery programmatically is to use the Google Cloud Client Library in your preferred programming language
- Like all Google Cloud services, exposes a traditional JSON/REST interface
  Example:


For Datasets Resource details, see the resource representation page.

Method	HTTP request	Description
URIs relative to https://www.googleapis.com/bigquery/v2, unless otherwise noted		
delete	DELETE /projects/*projectId*/datasets/*datasetId*	Deletes the dataset specified by the datasetId value. Before you can delete a dataset, you must delete all its tables, either manually or by specifying deleteContents. Immediately after deletion, you can create another dataset with the same name.
get	GET /projects/*projectId*/datasets/*datasetId*	Returns the dataset specified by datasetID.
insert	POST /projects/*projectId*/datasets	Creates a new empty dataset.
list	GET /projects/*projectId*/datasets	Lists all datasets in the specified project to which you have been granted the READER dataset role.

Figure 5-1. The BigQuery REST API specifies that issuing an HTTP DELETE request to the URL `/projects/<PROJECT>/datasets/<DATASET>` *will result in the dataset being deleted if it is empty*

Mainly this chapter was about how to access BQ:

Summary:

- A REST API that can be accessed from programs written in any language that can communicate with a web server
- A Google API client that uses autogenerated language bindings in many programming languages
- A custom-built BigQuery client library that provides a convenient way to access BigQuery from a number of popular programming languages

Chapter 6. Architecture of BigQuery

Lifecycle of a bigquery query

Step 1: HTTP POST

Bq cient sends post request to endpoint

- We use post because a query job is created
- An authorization token is required to authorize yourself via OAuth2
- It contains a JSON payload, which indicates we are running a query

Step 2: routing

The request is routed via a google front end service to the bigquery backend

This backend then routes the request to the region where the data lies

The router transforms the JSON HTTP request to Protocol Buffers (Protobufs), which is the platform- and language-neutral serialization format used for communication between virtually all Google services.

Step 3: Job Server

The BigQuery Job Server is responsible for keeping track of the state of a request.

The job server operates asynchronously.

The Job Server performs authorization to ensure that the caller is allowed to run a query that is billed to the enclosing project of the job

The Job Server is in charge of dispatching the request to the correct query server.

Step 4: Query engine

Queries are routed to a Query Master, which is responsible for overall query execution. The Query Master contacts the metadata server to establish where the physical data resides and how it is partitioned.

A slot is a thread of execution on a query worker shard; it generally represents half of a CPU core and about 1 GB of RAM. This amount is somewhat fuzzy, because slots can grow or shrink if they need more or fewer resources and as computers in a Google datacenter are upgraded

The scheduler decides how to farm out work among the query shards. A request for slots returns the addresses of the shards that will run the query. The Query Master then sends the query request to each of the Dremel shards in parallel

Step 5: Returning the query results

Results are split in two:

- The first page along with metadata is stored in Spanner, distributed relational db
- The remaining data is stored in colossus

BigQuery results are stored for 24 hours; they are functionally equivalent to a table and can be queried as if they were a table. Results are limited to 10 GB for normal SELECT queries

<u>**QUERY ENGINE DREMEL**</u>

Dremel Architecture

The query engine has three parts:

- Query Master: planning the query
    - Gets metadata / file locations
    - Create a query plan
- Scheduler: assigning slots
    - Slot = unit of work on a file / shuffle sink
    - Shuffle sink = temporary storage location for intermediate query results
- Worker Shard: executing query
    - Done on files in Colossus

Hash partitioning: When shuffling the values a hash function is used on them to determine the bucket they are sent to.

There are two types of joins in BigQuery; broadcast and hash.

Hash Join: The second type of common join is a hash join. This, in general, is much more computationally expensive. Hash joins work by hashing both sides of the join so that rows containing the same keys end up in the same bucket.

Broadcast join: Broadcast joins can be used when one of the tables is small: about 150 MB or less, as of this writing. Broadcast joins take the small table and send the entire table to every worker (in this case in the execution plan coalesce shows up: Coalesce is a dynamic stage that is added when BigQuery detects that one of the tables in a join is going to be small).

- join+ : this is a stage that joins and aggregates in one step
- EACH WITH EACH: Inner join
- EACH WITH ALL: Outer Join

- when loading data into BQ it is written to Capacitator files and stored in Colossus. This uses erasure encoding -> tays durable even if a large number of disks fail or are destroyed. Also the data is replicated to another availability zone within the same region

- BG uses erasure encoding (Erasure encoding stores mathematical functions of the data on other disks to trade off complexity for space)

- Replicated encoding: store multiple copies of the data (expensive)

Small cardinality = few distinct values

<u>Columnar storage</u>

- GB saves files as Capacitator files
- Storing entire rows is only good if you read entire rows at a time, but mostly this is not the case
- columnar storage is more easily compressable since a column often has lots of repreating values
- Other columnar storage formats: Parquet and Optimized Row Columnar (ORC)

- One of the key features of Capacitor is *dictionary encoding*. That is, for fields that have relatively small cardinality (few distinct values), it stores a dictionary in the file header -> this way you can circumvent scanning the entire table for matches, but you can just scan the dictionary and create a truth table from this (dictionary encoding makes filters efficient)

Storage set

- A storage set is an atomic unit of data, created in response to a load job, streaming extraction, or Data Manipulation Language (DML) query.
- The underlying physical storage for BigQuery is immutable. After a file is closed, it can never be changed again. Storage sets are likewise immutable; after they are committed, they are never changed again.

Metadata

Metadata has three layers in BQ:

- The outer layer is the dataset, which is a collection of tables, models, routines, and so on with a single set of access control permissions
- The next layer is the table, which contains the schema and key statistics
- The inner layer is the storage set, which contains data about how the data is physically stored

Time travel

- As of this writing, BigQuery supports time travel for seven days in the past, which means that you can read the state of the table at any point within that time window
- BigQuery keeps track of the timestamp at which storage set transitions happen

Storage optimization

The storage optimizer helps arrange data into the optimal shape for querying. It does this by periodically rewriting files. Files can be written first in a format that is fast to write (write-optimized storage) and later in a format that is fast to query (read-optimized storage). The storage system can be said to be generational, meaning that data is written into multiple generations, each one being older and more optimized.

Partitioning

Under the hood, a partition is essentially a lightweight table. Data for one partition is stored in a physically separate location from other partitions, and partitions have a full set of metadata

Clustering

Clustering is a feature that stores the data in semisorted format based on a key that is built up from columns in your data.

Reclustering

Periodically, in the background, BigQuery will recluster tables. BigQuery maintains a *clustering ratio*, which is the fraction of the data that is completely clustered.

PARTITIONING VERSUS CLUSTERING

Partitioning can be thought of as dividing your table into a lot of subtables based on data in a column. Clustering, on the other hand, is like sorting your tables on a particular set of columns. The differences can be subtle, but clustering works better when you have a large number of distinct values. For example, if you have a million customers and often do queries that look up a single customer, clustering by customer ID will make those lookups very fast. If partitioned by `customer_id`, lookups would be fast, but the amount of metadata needed to keep track of all of the partitions would mean that queries across all users would slow down.

Partitioning is often used in conjunction with clustering; you can partition by the low-cardinality field (e.g., event date) and cluster by the high-cardinality one (e.g., customer ID). This lets you operate over a date-range slice of the table as if it were itself a table, but it also lets you find records from a particular customer without having to scan all of the data in the partition.

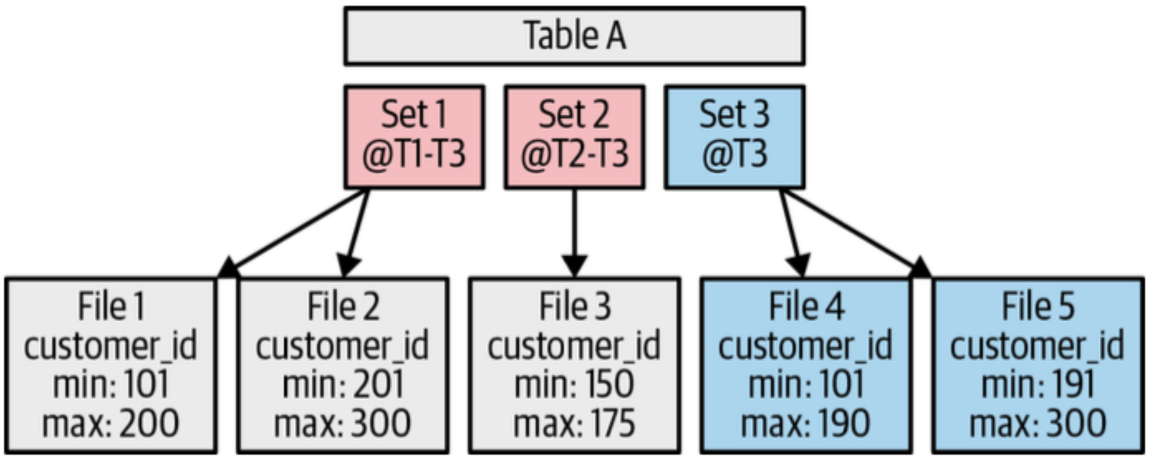


Table A
Set 1 @T1-T3
Set 2 @T2-T3
Set 3 @T3
File 1 customer_id min: 101 max: 200
File 2 customer_id min: 201 max: 300
File 3 customer_id min: 150 max: 175
File 4 customer_id min: 101 max: 190
File 5 customer_id min: 191 max: 300

Figure 6-34. Reclustering creates a new storage set

Chapter 7. Optimizing Performance and Cost

What to focus on:

- How much data is read from storage and how that data is organized
- How many stages your query requires and how parallelizable those stages are
- How much data is processed at each stage and how computationally expensive each stage is

Two types of pricing plans:

- **On-demand pricing**
  pay based on the amount of data processed
- **Flatrate**:
  your business gets a certain number of slots and you can run as much queries as you want without increase in cost
- For measuring queryspeed it is recommended to use “Workload Tester” (requires Gradle), you can install it in you cloudshell vm
- BQ logs can be viewed in the GCP web console in the Stackdriver Logging Section

Reading Query Plan Information

- Most steps start by reading in results of previous stages and end by writing intermediate resullts

Measure query speed and identify potential problems:

1. Measure the overall workload time using the BigQuery Workload Tester.
2. Examine the logs to ensure that the workload is not performing any unexpected operations.
3. Examine the query plan information of the queries that form the workload to identify bottlenecks or unnecessary stages.

Increasing Query Speed

- **Minimizing I/O**
  Minimize the data that is read (the columns that are used)
- **Caching the results of previous queries Performing**
  The BigQuery service automatically caches query results in a temporary table. But the query string must be the exact same, even a whitespace will make it miss the cache. Other conditions apply also.
  Use materialized views instead of running the same queries over and over.
- **Efficient joins**
  - If it is possible to avoid a join, or to reduce the amount of data being joined, do so
  - Avoid self-joins by using window functions for example
- **Avoiding overwhelming a worker**
- **Using approximate aggregation functions**
- **Reducing the number of expensive computations**
  This is something you need to be smart about, no hard rules apply here.
- **Accelerating queries with BI Engine**
  It will automatically store relevant pieces of data in memory (either actual columns from the table or derived results) and will use a specialized query processor tuned for working with mostly in-memory data.
- **Denormilization**
  One way to improve the read performance and avoid joins is to give up on storing data efficiently and instead add redundant copies of data. This way joins are avoided later because data is all in one table.
- **Avoiding Overwhelming a Worker**
  Some operations (e.g., ordering) need to be carried out on a single worker. Having to sort too much data can overwhelm a worker's memory and result in a "resources exceeded" error -> limit large sorts
  example: instead of ordering an entire data set by a date column we could only order by a subset of this data and order by the days in the date column. This way the sort is partitioned
- **Optimizing user-defined functions**

UDFs are supported but degrade performance so try to use as little data as possible with them

- **Using Approximate Aggregation Functions**
  for large tables this might have an advantage, we trade a certainty for performance. this should only be used if an approximate error of 1% can be tolerated.

USING “LIMIT”: Unless you are reading from a clustered table, applying a clause does not affect the amount of data you are billed for reading

Optimizing How Data Is Stored and Accessed

- **Minimizing Network Overhead**
  services that query the dataset should be located in the same region as the dataset to reduce network traffic
- **Compressed, partial responses**
  When invoking the REST API directly, you can minimize network overhead by accepting compressed, partial responses.
- **Batching multiple requests**
  When using the REST API, it is possible to batch multiple BigQuery API calls by using the content type and nesting HTTP requests in each of the parts
- **Choosing an Efficient Storage Format**
  - he fastest performance is obtained if you store data in such a way that queries need to do very little seeking or type conversion.
  - the fastest query performance will be obtained if you use native tables
  - use external sources for staging but set up a ETL process to peridically load data into BQ
  - If your use case is such that you need to query the data from Google Cloud Storage, store it in a compressed, columnar format
- **Setting up life cycle management on staging buckets**
  it is recommended to delete data after loading it into bigquery. If the data is heavily transformed though and we want to keep the raw data, we can set up life cycle management to move data to nearline storage or cold line storage to reduce costs.
- **Storing data as arrays of structs**

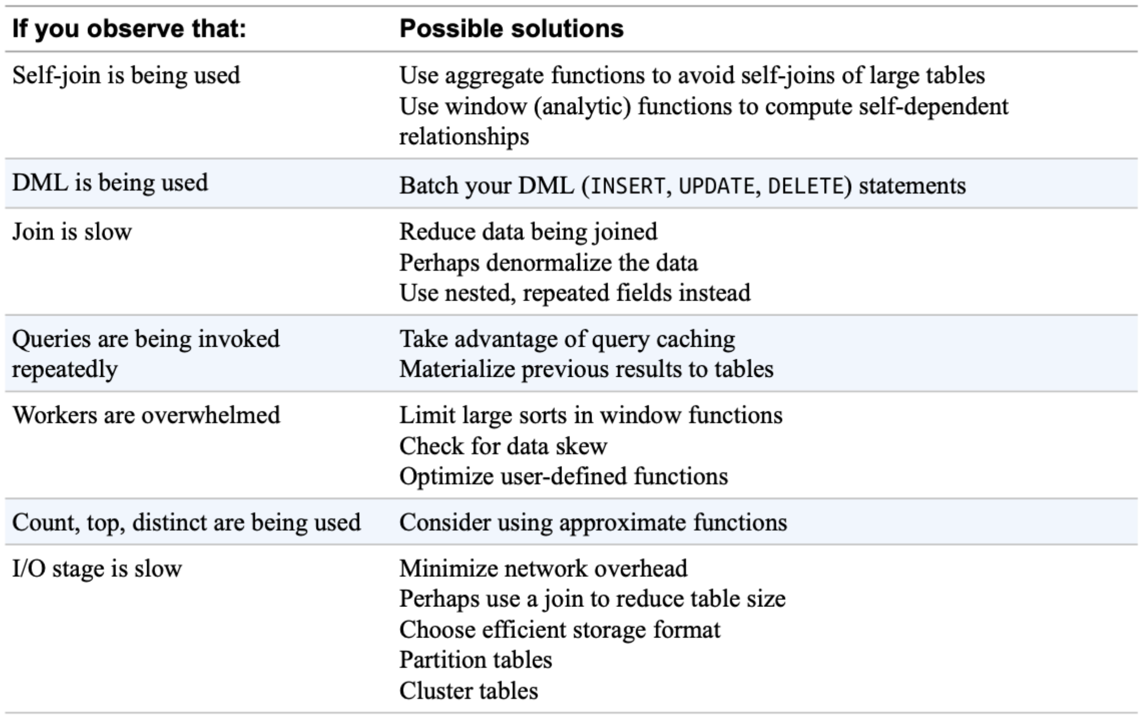
This has the potential to reduce row count dramatically. If we group data together by a certain attribute and save its attributes in an array, we only have one row for a certain item instead of multipole observations.

- **Storing data as geography types**
  ST_GeogFromText is expensive so save geographical data as geography types from the start
- **Partitioning Tables to Reduce Scan Size**
- **Clustering Tables Based on High-Cardinality Keys**
  clustering is like ordering the table internally with a certain criterion. this way scan size can be reduced
- **Clustering by the partitioning column**
  For example partition by data and cluster by hour
- **Reclustering**
  BQ reclusters table on its own, if the table receives a stream of data that breaks up the clustering. If you don’t want to wait for this, you can trigger this yourself using DML

Time-Insensitive Use Cases

- Batch Queries
  It is possible to submit queries that run as soon slots are available
- Instead of streaming data to BG you can use file loads. They don’t incur charges but are not as “up to date” as streams

Performance increase checklist

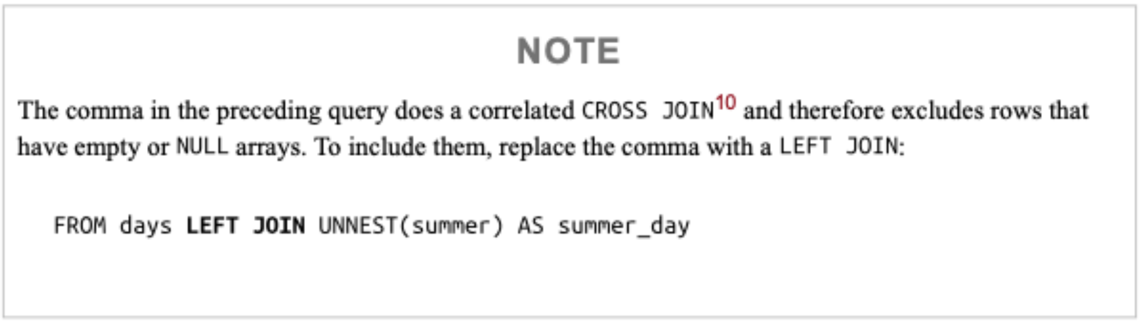


If you observe that:	Possible solutions
Self-join is being used	Use aggregate functions to avoid self-joins of large tables Use window (analytic) functions to compute self-dependent relationships
DML is being used	Batch your DML (`INSERT`, `UPDATE`, `DELETE`) statements
Join is slow	Reduce data being joined Perhaps denormalize the data Use nested, repeated fields instead
Queries are being invoked repeatedly	Take advantage of query caching Materialize previous results to tables
Workers are overwhelmed	Limit large sorts in window functions Check for data skew Optimize user-defined functions
Count, top, distinct are being used	Consider using approximate functions
I/O stage is slow	Minimize network overhead Perhaps use a join to reduce table size Choose efficient storage format Partition tables Cluster tables

Chapter 8. Advanced Queries

this just serves more as a reference to what topics can be found in the chapter rather than extracting knowledge

- Parameterized Queries
- **SQL User-Defined Functions / Persistent UDFs / Public UDFs**
- **Unnesting joins**

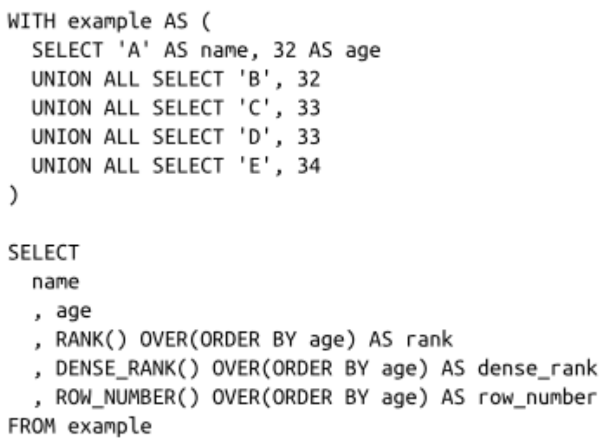


NOTE

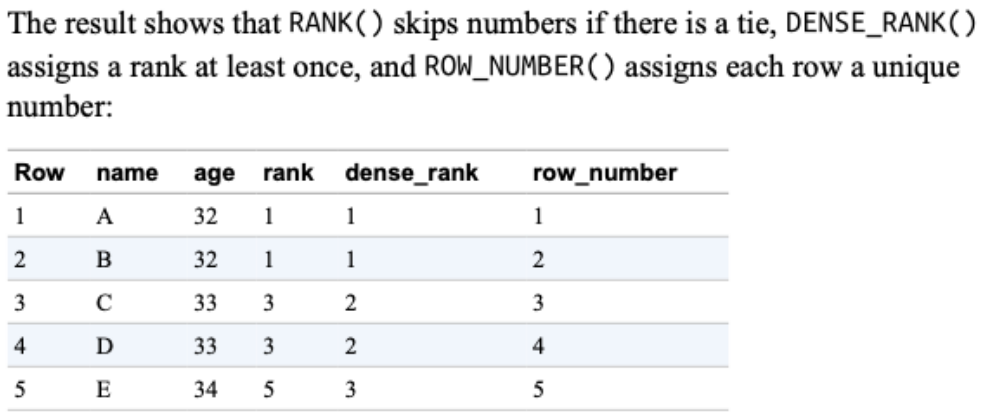
The comma in the preceding query does a correlated `CROSS JOIN`[10] and therefore excludes rows that have empty or `NULL` arrays. To include them, replace the comma with a `LEFT JOIN`:

```
FROM days LEFT JOIN UNNEST(summer) AS summer_day
```

- **Window Functions**



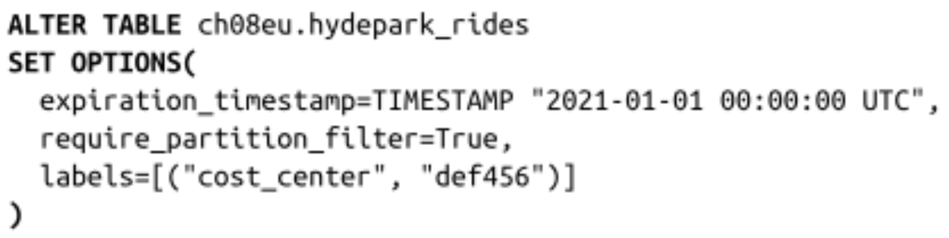
```
WITH example AS (
  SELECT 'A' AS name, 32 AS age
  UNION ALL SELECT 'B', 32
  UNION ALL SELECT 'C', 33
  UNION ALL SELECT 'D', 33
  UNION ALL SELECT 'E', 34
)

SELECT
  name
  , age
  , RANK() OVER(ORDER BY age) AS rank
  , DENSE_RANK() OVER(ORDER BY age) AS dense_rank
  , ROW_NUMBER() OVER(ORDER BY age) AS row_number
FROM example
```



The result shows that RANK() skips numbers if there is a tie, DENSE_RANK() assigns a rank at least once, and ROW_NUMBER() assigns each row a unique number:

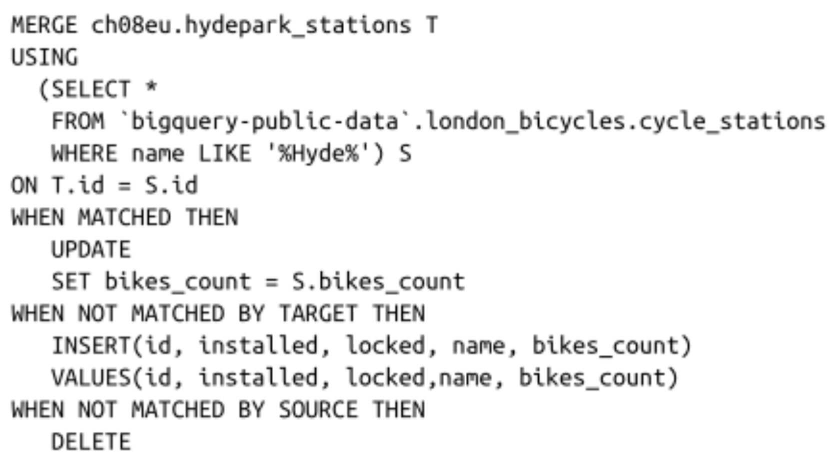
Row	name	age	rank	dense_rank	row_number
1	A	32	1	1	1
2	B	32	1	1	2
3	C	33	3	2	3
4	D	33	3	2	4
5	E	34	5	3	5

- You can **query metadata**
- You can create **Labels and tags**
- **Time travel**
  For up to seven days, you can query the historical state of a table. For example, to query a table as it existed six hours ago, you can use
- **DML**
  you can set options like expiration timestamp and require_partition with the set_options function when creating tables:

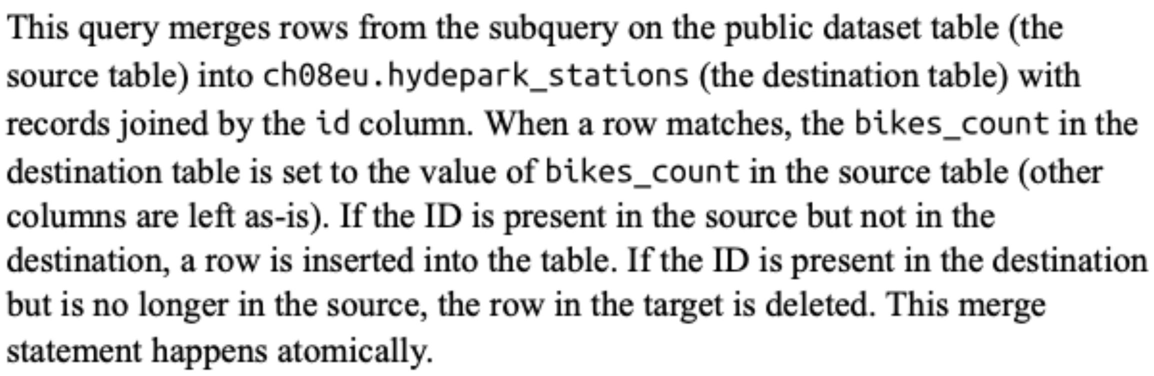


```
ALTER TABLE ch08eu.hydepark_rides
SET OPTIONS(
  expiration_timestamp=TIMESTAMP "2021-01-01 00:00:00 UTC",
  require_partition_filter=True,
  labels=[("cost_center", "def456")]
)
```

- **MERGE Statement**
  A statement is an atomic combination of INSERT, UPDATE, and DELETE operations that runs (and succeeds or fails) as a single statement
  Example:



```
MERGE ch08eu.hydepark_stations T
USING
  (SELECT *
   FROM `bigquery-public-data`.london_bicycles.cycle_stations
   WHERE name LIKE '%Hyde%') S
ON T.id = S.id
WHEN MATCHED THEN
   UPDATE
   SET bikes_count = S.bikes_count
WHEN NOT MATCHED BY TARGET THEN
   INSERT(id, installed, locked, name, bikes_count)
   VALUES(id, installed, locked,name, bikes_count)
WHEN NOT MATCHED BY SOURCE THEN
   DELETE
```



This query merges rows from the subquery on the public dataset table (the source table) into `ch08eu.hydepark_stations` (the destination table) with records joined by the `id` column. When a row matches, the `bikes_count` in the destination table is set to the value of `bikes_count` in the source table (other columns are left as-is). If the ID is present in the source but not in the destination, a row is inserted into the table. If the ID is present in the destination but is no longer in the source, the row in the target is deleted. This merge statement happens atomically.

- JavaScript UDFs
- Scripting
  We can write a BigQuery script consisting of multiple statements and send it to BigQuery in one request
- Temporary tables:
  How to: **CREATE TEMPORARY TABLES**
  Temporary tables exist for the lifetime of a script and are automatically cleaned up when the script completes.
- You can use loops and control flow statements in scripts
- Stored Procedures
  Like UDF, can be parameterized and invoked by the "CALL" keyword

  Other topics discussed:
- Geographical functions
- Statistical functions (Quantiles, Correlation)
- Hashing functions

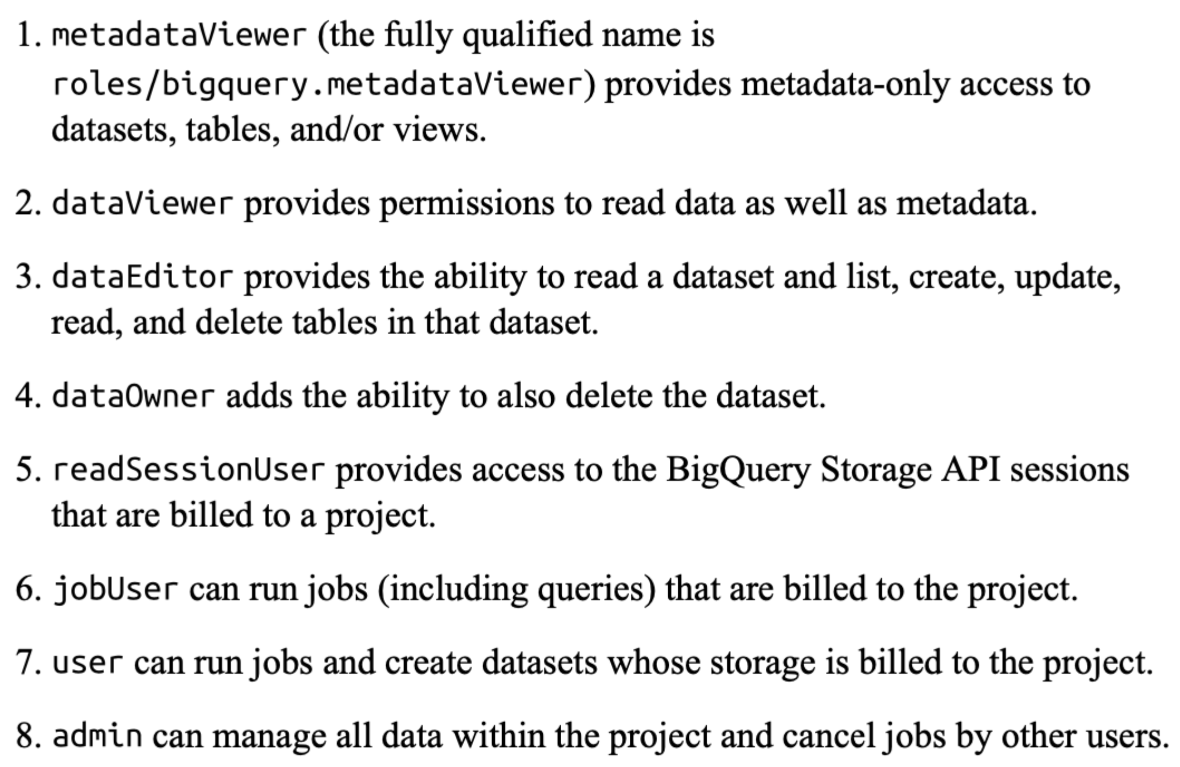
Chapter 10. Administering and Securing BigQuery

- In Google Cloud Platform (GCP), data is encrypted at rest and in transit, and the API-serving infrastructure is accessible only over encrypted channels
- IAM (Identiy and Access manangement) defines three things: Identity, role and resource
- You should prefer providing access to Google groups over providing access to individuals because it is easier to add members to and remove members from a Google group instead of updating multiple Cloud IAM policies to onboard or remove users.
- Providing access to ***allAuthenticatedUsers*** equates to making something public

Roles

- The role determines what access is allowed to the identity in question. A role consists of a set of permissions.

List of common roles:



1. `metadataViewer` (the fully qualified name is `roles/bigquery.metadataViewer`) provides metadata-only access to datasets, tables, and/or views.
2. `dataViewer` provides permissions to read data as well as metadata.
3. `dataEditor` provides the ability to read a dataset and list, create, update, read, and delete tables in that dataset.
4. `dataOwner` adds the ability to also delete the dataset.
5. `readSessionUser` provides access to the BigQuery Storage API sessions that are billed to a project.
6. `jobUser` can run jobs (including queries) that are billed to the project.
7. `user` can run jobs and create datasets whose storage is billed to the project.
8. `admin` can manage all data within the project and cancel jobs by other users.

- Primitve roles
  The primitive roles correspond to the “Owner”, “Publisher” and “Reader” roles that existed before the implementation of Cloud Identity and Access Management. According to GCP documentation, **it is recommended that you use the predefined Cloud IAM roles instead.**

Custom roles

- One reason to create a custom role is to subtract permissions from the predefined roles.

Creating roles:
You can create roles via the cloud console by specifying a yaml file for them, example:

```
title: "Data Supplier"
description: "Can create, but not delete tables"
stage: "ALPHA"
includedPermissions:
- bigquery.datasets.get
- bigquery.tables.list
- bigquery.tables.get
- bigquery.tables.getData
- bigquery.tables.export
- bigquery.datasets.create
- bigquery.tables.create
- bigquery.tables.updateData
```

Then you would run the following `gcloud` command to create the custom role:

```
PROJECT=$(gcloud config get-value project)
gcloud iam roles create dataSupplier --project $PROJECT \
       --file dataSupplier.yaml
```

Administering BigQuery

- It is possible to administer BigQuery from the BigQuery web UI, using the REST API, or using the command-line tool.

Recovering lost data:

- You can recover the state of a table up to 7 days back (this is the case when the contents of the table are messed up)
- Deleted tables (as opposed to deleted records within existing tables) can be recovered for up to two days only
- applying labels to datasets is a good way to later filter on these in datastudio visualizations