Optimizing Google BigQuery

A REAL-WORLD GUIDE



MATILLION

Google Cloud Platform

TABLE OF CONTENTS

4 **Introduction**

4 **About this book**

6 **Chapter 1: BigQuery Architecture**

9 **Chapter 2: External Data Load Optimization**

9 2.1 How to Load Google External Data into BigQuery

20 2.2 How to Load Other External Data from into BigQuery

34 **Chapter 3: Query Optimization**

34 3.1 Avoid SELECT *

35 3.2 LIMIT and WHERE clauses

36 3.3 WITH clauses

37 3.4 Approximate Aggregation

38 3.5 Efficient Preview Options

39 3.6 Early Filtering

40 3.7 JOIN clause optimization

42 3.8 Analytic Functions

44 3.9 ORDER BY clauses

44 3.10 Materialize Common Joins

46 3.11 Avoid OLTP Patterns

50 3.12 Denormalize

53 **Chapter 4: Cost Estimation and Management**

53 4.1 Data Storage Pricing

54 4.2 Partitioning

55 4.3 Monitor Query Costs

59 **Conclusion**

59 **About Matillion**

60 **Glossary**

Introduction

If you are reading this eBook, you are probably considering or have already selected Google BigQuery as your modern data warehouse in the cloud — way to go. You are ahead of the curve (and your competitors with their outdated on-premise databases)! Now what?

Whether you're a data warehouse developer, data architect or manager, business intelligence specialist, analytics professional, or tech-savvy marketer, you now need to make the most of that platform to get the most out of your data! With the growing masses of data being produced by sources as diverse as they are plentiful, a data-driven smart solution is needed. That's where BigQuery and this eBook come in handy!

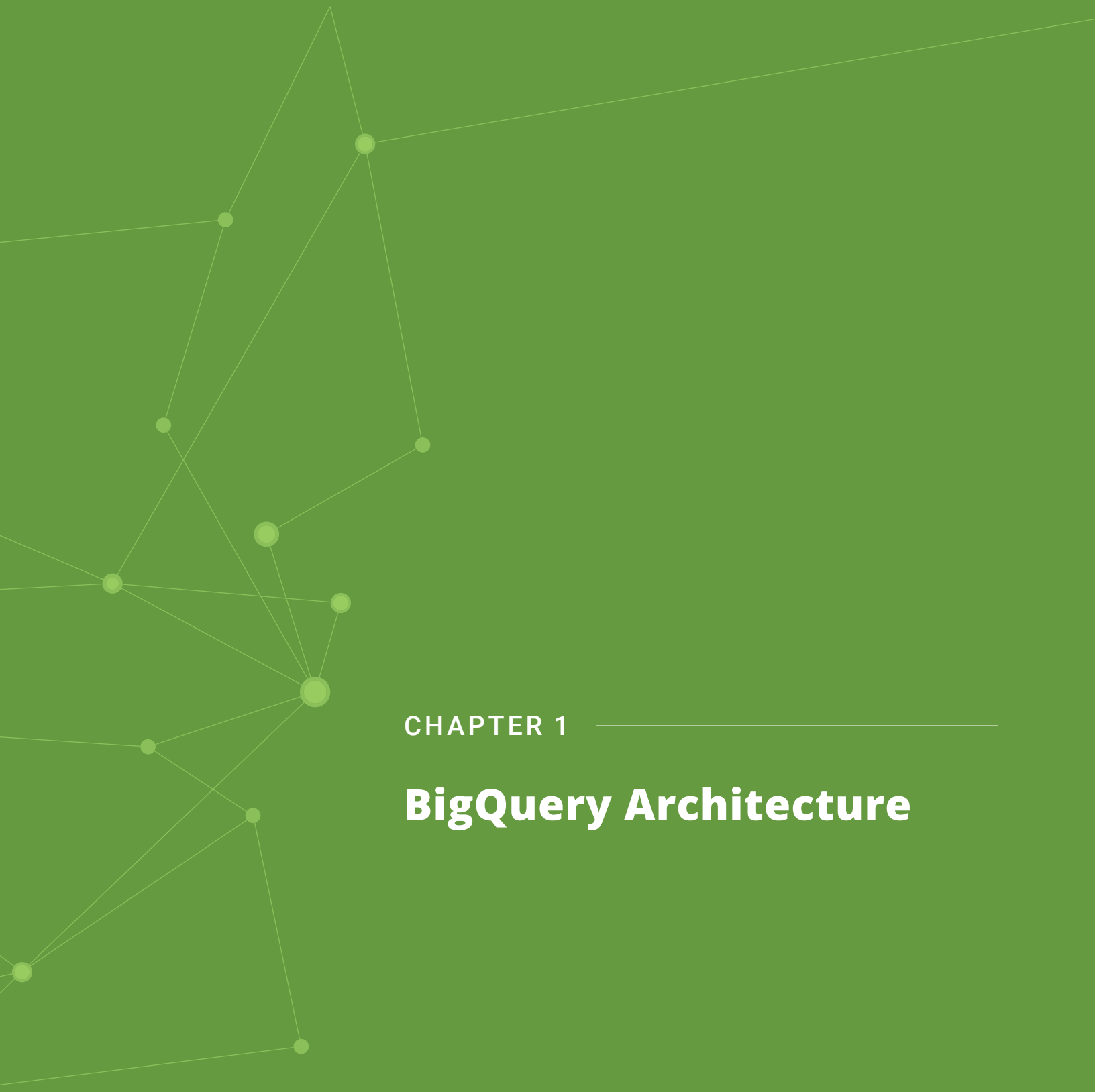
About this book

In **Chapter 1: BigQuery Architecture** we will look under the hood to understand BigQuery's technical workings. The primary differences between BigQuery and historical warehouses are scalability and power. **Google BigQuery** is a serverless, highly scalable, low cost enterprise data warehouse. BigQuery allows organizations to capture and analyze all their data in real-time using its powerful streaming ingestion capability so that your insights are always current. This means BigQuery is optimized for Extract-Load-Transform (ELT) workflows, as opposed to its predecessor Extract-Transformation-Load (ETL). We build on this foundation in subsequent chapters.

Chapter 2: External Data Load Optimization, presents the best practices for loading your data in the most optimal way. This is useful if you are just starting out and need to populate your BigQuery database with historical or external data from many different sources, or if you're looking to improve your data loading processes.

Chapter 3: Query Optimization explains how to make the most of BigQuery's power to query and transform data in a way that reduces (already low) costs and improves performance. This is a particularly useful chapter if you have previously used OLTP systems as there are some differences highlighted that will impact performance.

Finally, **Chapter 4: Cost Optimization** will look at costs. Cloud-based databases are renowned for being more cost efficient than on-premise warehouses that require a range of overheads, from hardware procurement to ongoing maintenance costs. Google BigQuery, a fully managed service, on the other hand, employs a pay-per-use pricing model. This is, in itself, cost saving but there are some steps you can take to forecast, limit and control affordable costs.



CHAPTER 1

BigQuery Architecture

CHAPTER 1

BigQuery Architecture

BigQuery is a "serverless" database, designed to be an ideal platform to host your enterprise data warehouse in the cloud.

Of course, it's not really serverless: there are plenty of virtual machines, networks and disks behind the scenes making it work, but this is all orchestrated and managed by Google on your behalf. You are left with the business focused task of gaining value and insights from your data without the headaches of hardware and software provisioning.

This means you are no longer burdened with laboriously extrapolating your data needs 1, 3, or 5 years down the line. So rather than paying up-front for dedicated hardware and software, you pay on-demand for data storage and query execution. This is a much more cost-efficient strategy for building, maintaining and growing a modern data warehouse, when compared to on-premise alternatives.

You'll always use Google BigQuery in the context of a 'Project'. Within every project you can create one or more 'Datasets' to store the actual tables and views. For example, within the **bigquery-public-data** project there are about 50 example Datasets covering a wide range of subject areas. Some of these Datasets will be used later in this document.

How does it work? When you use Google BigQuery, your data is stored on a Colossus filesystem: the same technology underpinning other main Google services. The data is physically organized by column rather than by row as it would be in a traditional or "OLTP" database. This is generally great for performance, and opens up some opportunities for performance and cost optimizations which will be discussed in later chapters.

All data is held encrypted at rest. You do have the option of managing encryption yourself using KMS. Please see **this document** for more information.

As a user, you can interact with BigQuery in a few ways:

- The Google BigQuery **web console**
- A **client-side API**, with bindings for many languages including Java, Python, C#, Go, Node.js, PHP and Ruby
- A **command line interface** (which itself uses the Python API)
- Industry standard interfaces such as **JDBC and ODBC**
- Third-party tools such as **Matillion**

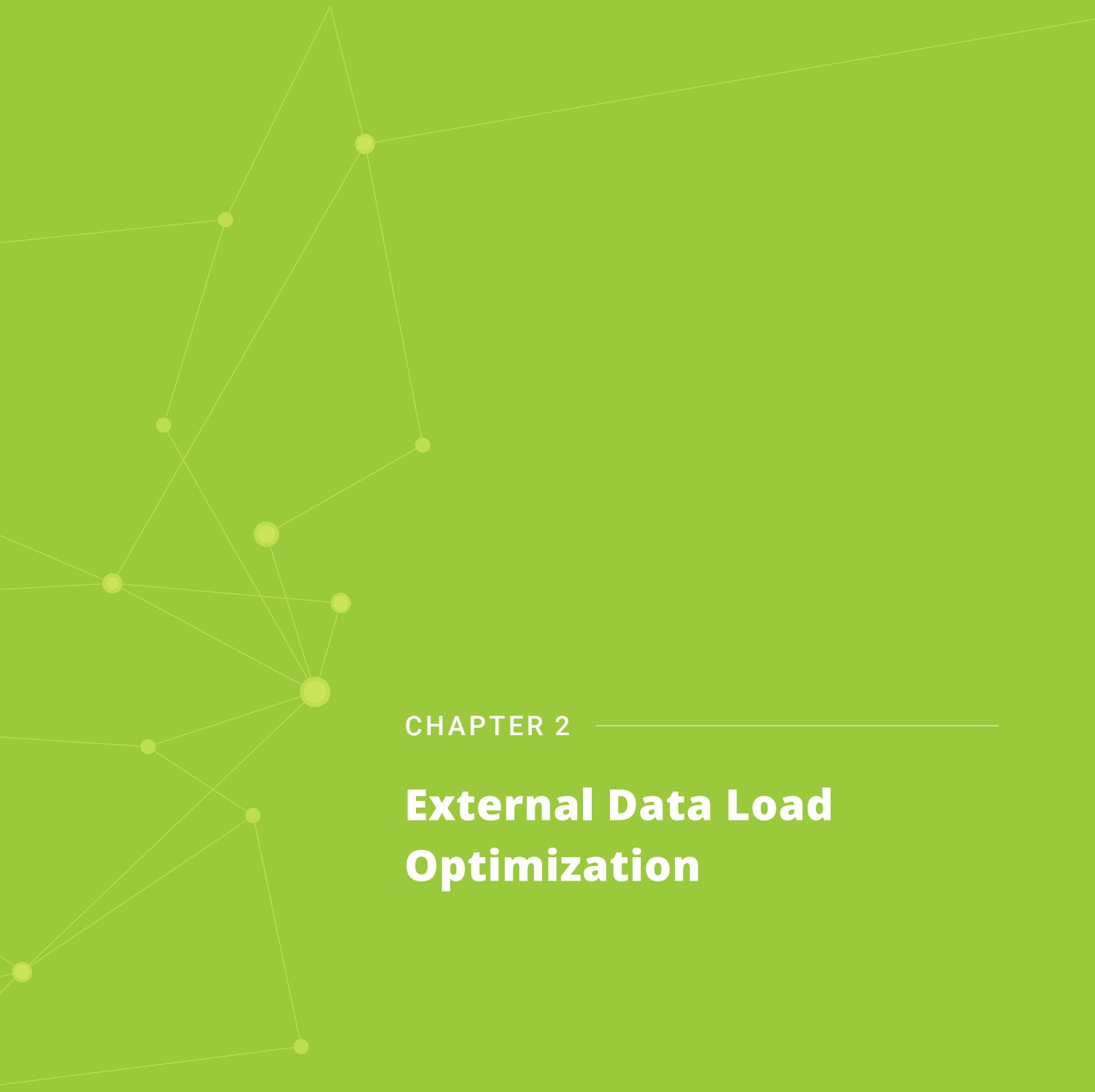
This makes your database easily accessible both in terms of data integration and to users with varied technical backgrounds.

Google BigQuery also supports SQL, and offers two modes of SQL execution: "legacy" (the original version) and "standard" (which has better compatibility with other tools). Google recommends that new work should use standard SQL. Legacy SQL is still supported, but **migrating from legacy SQL to standard SQL** is recommended. Whenever you execute a SQL statement, it is parsed and optimized internally by an execution engine called Dremel. You can influence the execution path — and therefore the speed and the cost — using techniques described in subsequent chapters.

You might want to query just a dozen bytes, or maybe a dozen gigabytes ... and you would not expect to have to wait 1000 million times longer for the second query to complete! So how does BigQuery handle the growing and diversified datasets you need to access? To achieve the vital aim of scalability, Google internally splits every SQL task between many "worker" compute nodes called "**slots**". This approach is known as Massively Parallel Processing or MPP. Often many thousands of slots will be called into action to work in parallel on your behalf, all done automatically and intelligently without any additional effort from you.

Some SQL tasks, especially joins and aggregations, require slots to exchange data at runtime. This is known as "shuffling" and is generally very fast, as it uses Google's specialized and dedicated network infrastructure. Nevertheless, it's another factor that you can sometimes influence to help get the most out of Google BigQuery while you're using it.

Now that you know a bit more about BigQuery and its "under the hood" workings we can start looking at its practical application and how you can make the most of this extremely powerful platform. Before you can think about Query Optimization and understanding costs, you'll need to get your data into BigQuery. Chapter 2 takes an in depth look at how you can best populate BigQuery with existing data from many different sources.



CHAPTER 2

External Data Load Optimization

CHAPTER 2

External Data Load Optimization

In this section we look at ways you can optimize the data load process in Google BigQuery for external data. This is especially useful if you are just getting started with Google BigQuery and you need to migrate multiple sources of data into BigQuery or if you have been using BigQuery and want to find out the best practices for bringing in new data in a way that increases performance and decreases cost.

In addition to initially populating your BigQuery database, there are numerous other reasons why you would want to load and keep your data in BigQuery. For example, store all data in one place for a single source of truth, compare with other data in BigQuery, analyze and aggregate data, share authorized data with colleagues without having to grant access to each individual source, and so on. This data will come from two primary sources, Google based external data (originating in Google Drive, Google Cloud Storage, Google Analytics, Google BigTable, etc.), or from another non-Google external data source. This Chapter will look at common data loading methods for each of these two sources.

2.1 How to Load Google External Data into BigQuery

Most users of BigQuery will hold data in Google sources that they will wish to review in BigQuery for the reasons discussed above. Generally, there are two options to getting your Google data into BigQuery. You can either link data directly from the source system, such as linking a Google Sheet in Google Drive, or physically copying data from the source and loading it into BigQuery. The advantage of linking is that updates made in the source will be visible immediately in the associated BigQuery table. The advantage of physically copying is that data can be snapshotted at a particular point in time. As a general rule of thumb it is recommended that data is physically copied into BigQuery to maximize Query Performance as linking can slow down your queries. That being said you should choose the method that best fits your use case and needs!

For the remainder of this section we will walk you through the options for loading your data from common Google sources into BigQuery using “out-of-box” Google functionality. We understand that there isn’t a one-size-fits-all solution for optimization, therefore, we will highlight the advantages of each to help you assess which is the optimal method to meet your business needs.

GOOGLE DRIVE (GOOGLE SHEETS)

Over **3 million businesses** are paying for and using GSuite, with another **800 million** non-paying users using Google Drive, a collaboration and sharing platform. In 2017, the platform exceeded **2 trillion** objects. Now that's what we call big data! With this high volume of data, you will most likely have a use case to bring these files, such as Google Sheets with user maintained data, into BigQuery for analysis and to enhance your existing database data.

In BigQuery it is possible to directly query external data which is held in Google Drive. There are two options for bringing in data from Google Drive into BigQuery, either linking or loading. We will walk you through both methods, using Google Sheets as an example.

Supported formats are:

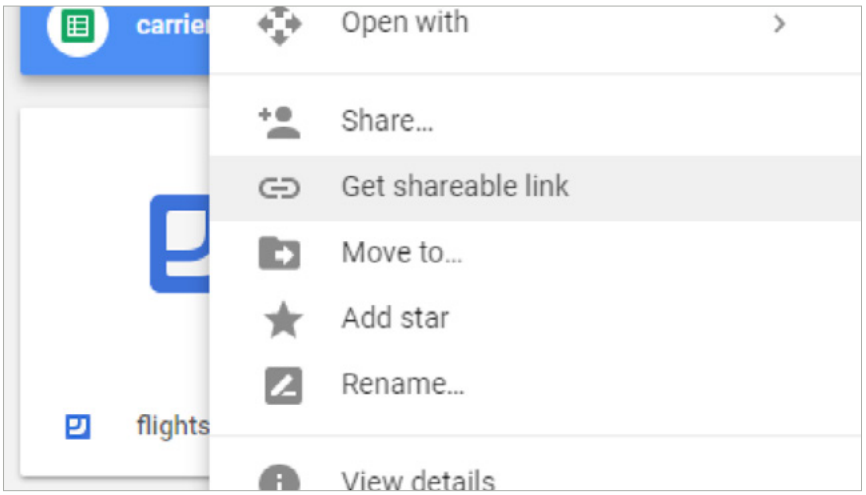
- CSV files
- JSON newline delimited files
- Avro files
- Google Sheets (first tab only)

Linking (Federated Data Source)

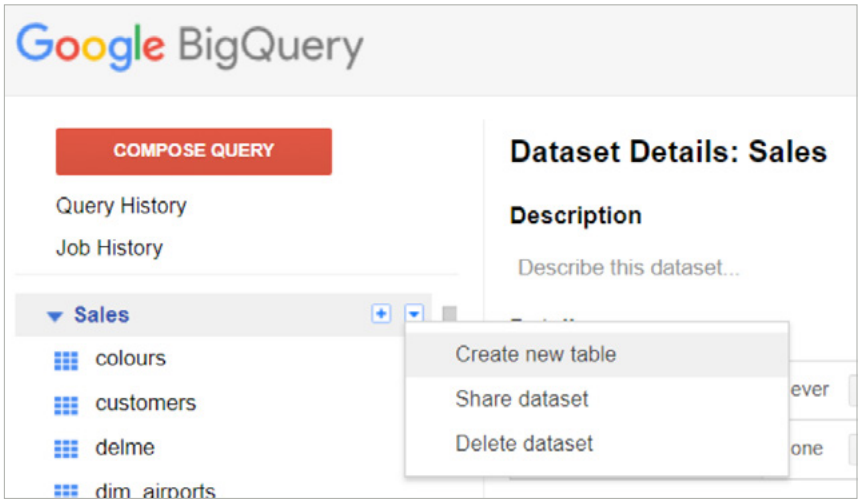
It is possible to set up a Google Sheets file which is held in Google Drive as a BigQuery Federated Data Source. This is done by creating a linked table in BigQuery which refers back to the data in the sheet.

This is done using the following steps:

1. Obtain the User Resource Identifier (URI), which is the unique identifier for the Google Sheets file. This is the shareable link accessed by right-clicking on the file in Google Drive:



2. Create a new table in BigQuery:

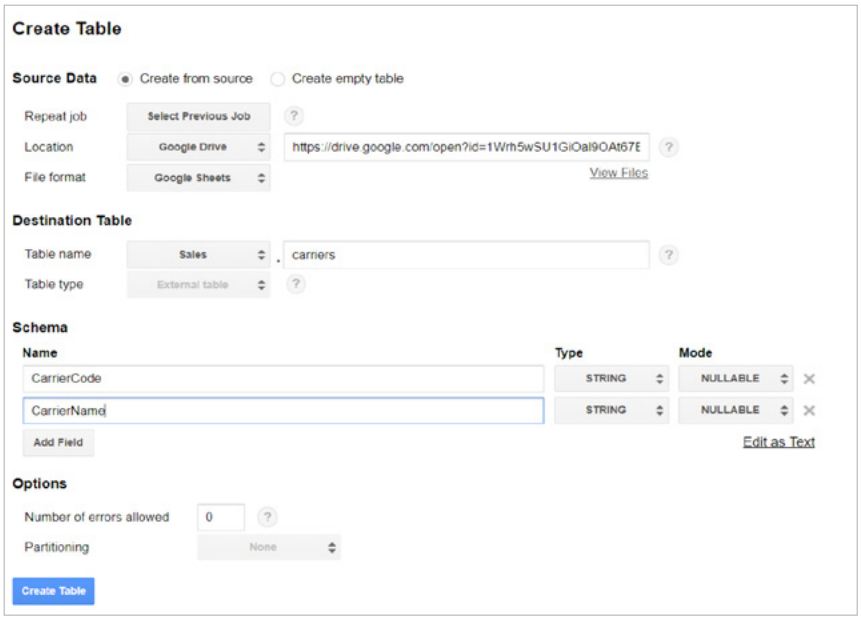


3. In the **Create Table** options, create the table from source with the **Location** as **Google Drive** and the link should be the shareable link from above

4. The **File Format** is Google Sheets

5. The **Schema** of the Table can be specified if required

6. Click **Create Table** button



The table will now be available in BigQuery to query as required. This table can be treated in the same way as standard BigQuery tables where the queries are run, and the data returned. It's as simple as that. The BIG advantage of maintaining the external data tables is that as soon as data is updated in Google Sheets, the

updates will be carried over into BigQuery (without manual effort) and appear in the relevant BigQuery queries.

However, if query performance is important, external tables such as the above should be avoided and instead external data should be copied into a BigQuery table. This is because, with linking, BigQuery has to look externally to process the query, which takes more computing resource and time than if that data was stored locally.

Linking a Google Drive Object (Google Sheet)

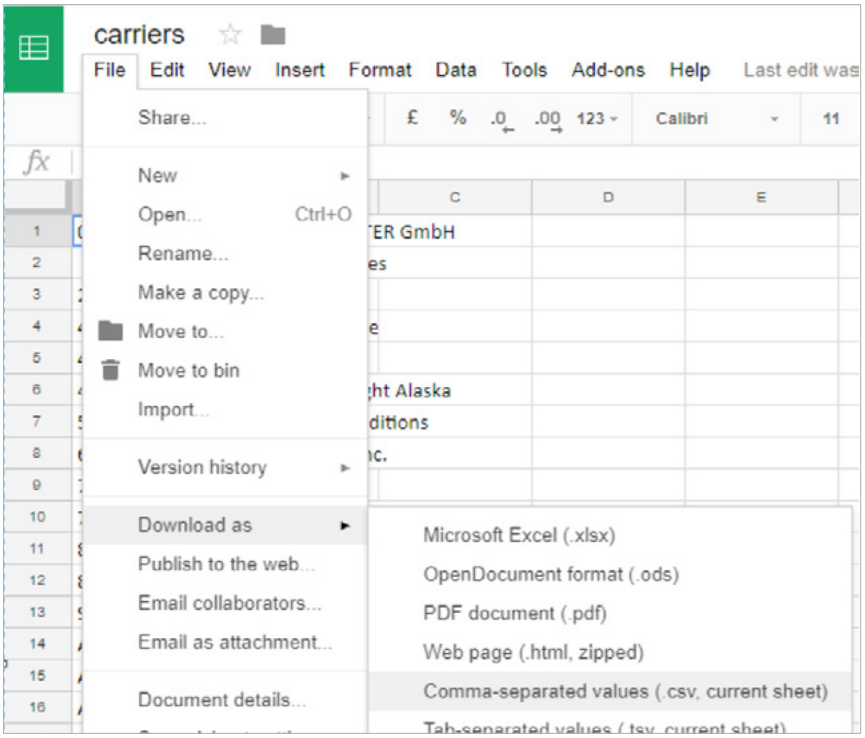
- ⊕ Source updated information is automatically sent to and received by BigQuery
- ⊖ Slower performance than the alternative, loading

Data Loading

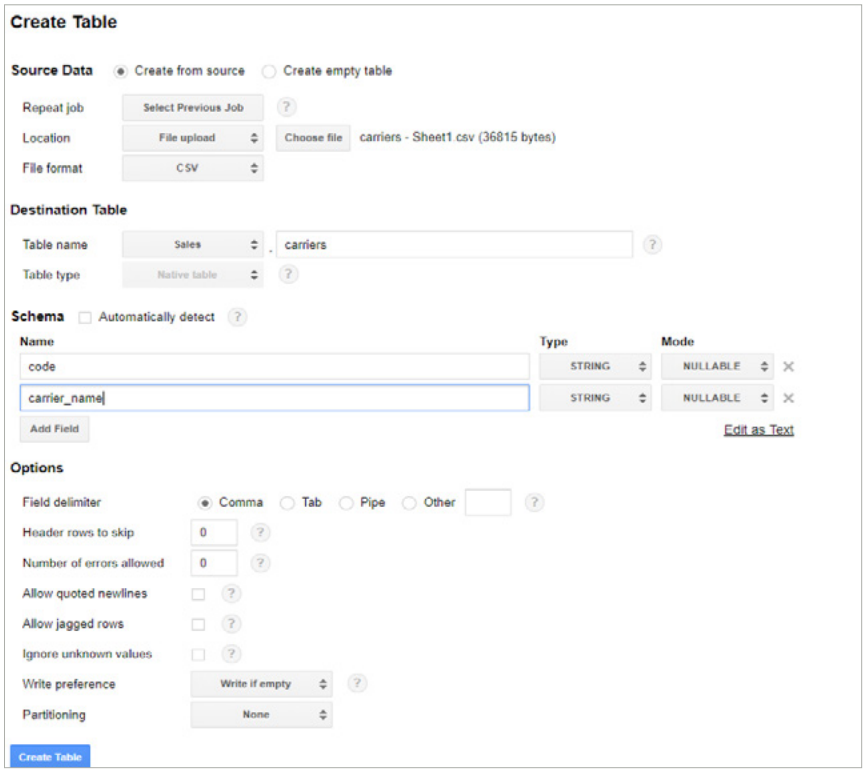
Loading data from the source into BigQuery can help address slow performance resulting from linking. This will create a new copy of the data in BigQuery which will not maintain it's link back to the original source. Therefore, if performance is most important to you, consider loading.

---

1. To copy data into a new BigQuery table, rather than using a shareable link, the file needs to be downloaded from Google Drive and then uploaded into BigQuery to create the new table. To download the file, this should be done directly from Google Drive by right clicking on the file and selecting **Download as**. You can see an example of this below where we download our "carriers" sheets as a CSV file:



2. Once downloaded we can import it into BigQuery using the console to create a new table:



3. Now the "carriers" table will be created with a copy of the data from the Google Sheet. Any updates made in Google Drive will NOT flow through to the table, however, query performance will be faster than the linked table

NOTE

This will only load the first tab of data from the Google Sheet. To copy other tabs, simply select the second tab in Google Drive and then click **File > Download As > Comma Separated Values**. This will download a new file with the contents of the second tab. Follow steps above.

Therefore, if you have a dataset that is fairly static without a lot of frequent changes being made, load and store in BigQuery to benefit from enhanced performance. However, if your dataset is constantly changing, you may benefit from linking reducing the effort of continuously uploading refreshed data.

Linking a Google Drive Object (Google Sheet)

⊕ Faster performance than the alternative, linking

⊖ Data updated at the source is not carried through to BigQuery

GOOGLE CLOUD STORAGE

Google Cloud Storage is a file storage web-based RESTful service that can help you maximize the scalability and flexibility of Google BigQuery. As with Google Drive, you can either link or copy data from Google Cloud Storage to BigQuery.

Supported formats are:

- CSV files
- JSON newline delimited files
- Avro files
- Datastore backup

Linking (Federated Data Source)

It is possible to create a BigQuery table which links directly to an external file in a Google Cloud Storage Bucket. This will create a Federated Data Source by using a similar method as discussed above.

1. In this example we looked at linking a JSON file using the Create Table tool discussed above

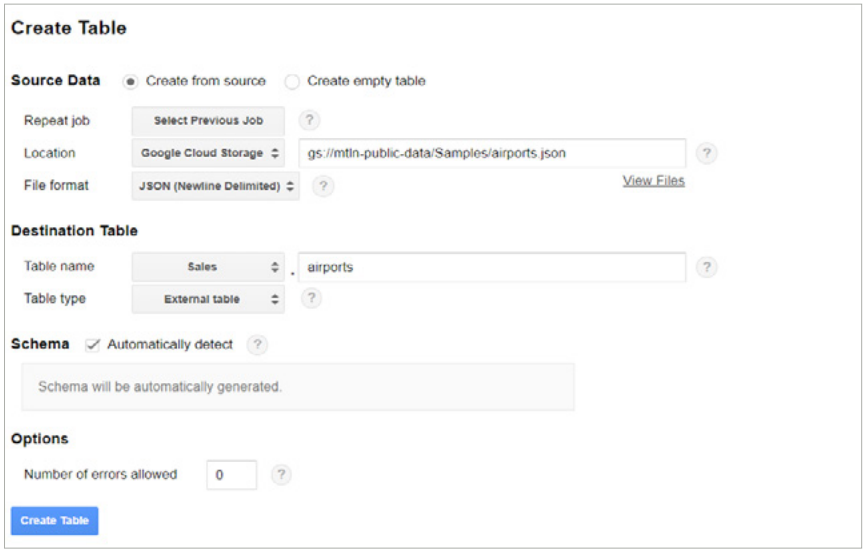
2. In the **Create Table** options, create the table from source with the Location as **Google Cloud Storage** and link to a JSON newline delimited file in a Cloud Storage bucket

3. The **File Format** is JSON

4. Fill in details regarding the **Destination Table**

5. The **Schema** of the Table can be specified if required

6. Click **Create Table** button



This will create a new table in BigQuery and the data can be viewed in the BigQuery interface and queried with SQL. Again, as with tables linked to Google Drive, if query performance is important then external tables should be avoided, and the content of the data files should be copied directly into a BigQuery table.

Linking a Google Cloud Storage Object (JSON)

- ⊕ Source updated information is automatically sent to and received by BigQuery
- ⊖ Slower performance than the alternative, loading

Data Loading

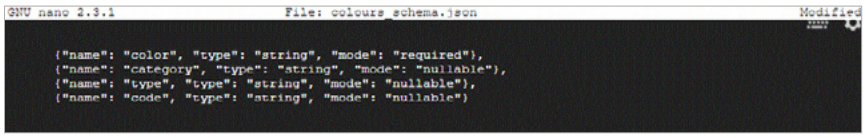
To load the contents of a file into BigQuery it is possible to do this via the console as described in the Google Sheets section above. Another option is to use the BigQuery **command line** to insert data via the API.

You will want to use the BigQuery API to copy data if you want to load many files into different BigQuery tables. As this is done programmatically it is much faster than other methods, thus improving performance you experience.

Here we will look at loading data from an example 'colors' file from our Cloud Storage Bucket to load into BigQuery as opposed to creating a linked table. The 'colors' file is a Newline Delimited JSON file with some details of different colors as shown below:

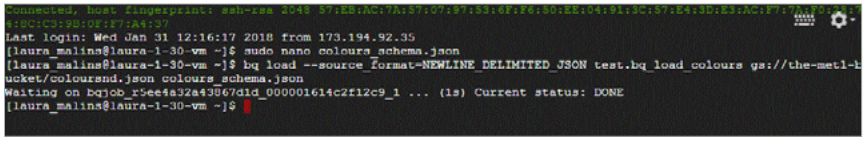
```
{"color":"black","category":"hue","type":"primary","code":"#000"}
{"color":"white","category":"value","code":"#FFF"}
{"color":"red","category":"hue","type":"primary","code":"#FF0"}
{"color":"blue","category":"hue","type":"primary","code":"#00F"}
{"color":"yellow","category":"hue","type":"primary","code":"#FF0"}
{"color":"green","category":"hue","type":"secondary","code":"#0F0"}
```

1. Create a schema file which details the columns of the data and their data types. We are loading the colors data with 4 columns, 3 of which are nullable



2. Use the BigQuery Load command (bq load) to copy the data into a new table. Specify the source format as JSON. If no source format is specified, CSV as the default will be used

3. The other parameters required by the command are the name of the new table in BigQuery to be created, *test.bq_load_colours*, the location of the file in the storage bucket and the schema file



4. The command line will show the progress of the BigQuery command and when all data has been loaded it will show as 'Done'

5. The data will now be available in the table to query:

Table Details: bq_load_colours

Schema | Details | Preview

Row	color	category	type	code
1	green	hue	secondary	#0F0
2	black	hue	primary	#000
3	yellow	hue	primary	#FF0
4	blue	hue	primary	#00F
5	red	hue	primary	#FF0
6	white	value	null	#FFF

In this case, any updates made in the file will not affect the table data.

There are tools that can help streamline the data loading process, including Matillion ETL for BigQuery, by mitigating the need to hand-code SQL and can auto-generate the schema files.



Find out more about **Matillion ETL for BigQuery** and our pre-build data loading components.

Loading a Google Cloud Storage Object (JSON):

- ⊕ Faster performance than the alternative, linking
- ⊕ API is faster when dealing with many files going to different BigQuery tables
- ⊖ Data updated at the source is not carried through to BigQuery
- ⊖ Need to manually write the program

GOOGLE ANALYTICS

Google Analytics data can be brought into BigQuery for further analysis by creating some required aggregations or by merging with other data, such as marketing data. As discussed Google Drive and Google Cloud Storage have two options for referencing external data, creating a link to the external data source or loading the external data into BigQuery, which can be leveraged based on user requirements. While it is possible to copy data from Google Analytics, we recommend linking data as best practice, outlined below.

Linking (Federated Data Source)

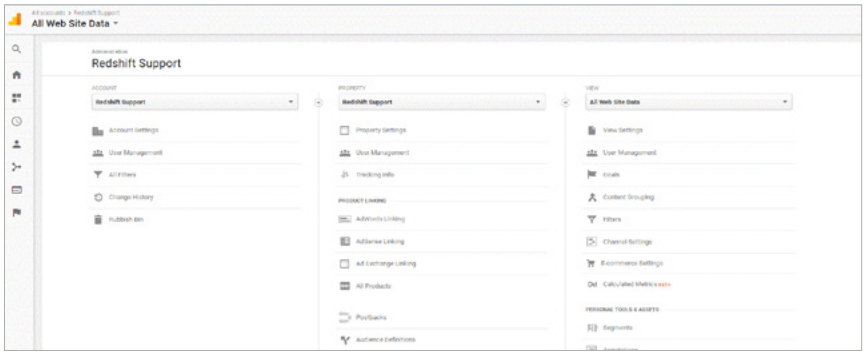
To create a Federated Data Source from Google Analytics data into BigQuery the recommended method would be to use Google Analytics 360. Google Analytics 360 has the necessary "BigQuery Export" functionality to link the data. To link the data using BigQuery Export follow the below steps:

1. Navigate to **https://analytics.google.com**

2. Click on **Admin** at the bottom left of the screen

3. Navigate to the property you want to link to

4. Click on **All Products** and then click **Link BigQuery**



5. Copy the **ProjectID** and **Project Number** from the relevant BigQuery project in the Google Cloud Platform Console

6. Select the **Google Analytics** view to Link and click **Save**

The Google Analytics data will now be available to query in BigQuery and will be updated daily with the previous day's data. Historical data will be available please see **here** for further details.

By linking data in this way users can use the Analytics data, for example, to review the effectiveness of traffic moving through their website. You could develop a deeper understanding by combining website traffic insights with sales data from

an Enterprise Resource Planning (ERP) system to review how that traffic coverts to sales. As previously mentioned, other options exist to copy Google Analytics data into BigQuery, but they involve doing a bulk export, which would have to be scheduled frequently making these workflows cumbersome and inefficient.

Linking Google Analytics data:

⊕ Best practice approach that is easy and efficient

⊖ Google Analytics 360 is required

GOOGLE BIGTABLE

Google Bigtable is a NoSQL Big Data database which is run as a service. It is good for applications that have frequent data ingestion as it is designed to handle low latency, high throughput data for large workloads. It is ideal as an operational or analytical database for applications like Internet of Things (IoT) or user analytics. Therefore, as with Google Analytics, linking is the most advantageous approach.

For the data you wish to obtain in Google Bigtable you need to retrieve:

- Your **ProjectID**
- Your Bigtable **InstanceID**
- **Name** of Bigtable table

These are used to form the Bigtable URI, which is required to be input into BigQuery.

NOTE: ACCESS CONTROL

The table in Bigtable must be a permanent external table and not a temporary table. At a minimum, read access to this table must be granted to the BigQuery user or service account querying the data.

URI table format

https://googleapis.com/bigtable/projects/**[PROJECT_ID]**/instances/**[INSTANCE_ID]**/tables/**[TABLE_NAME]**

Linking (Federated Data Source)

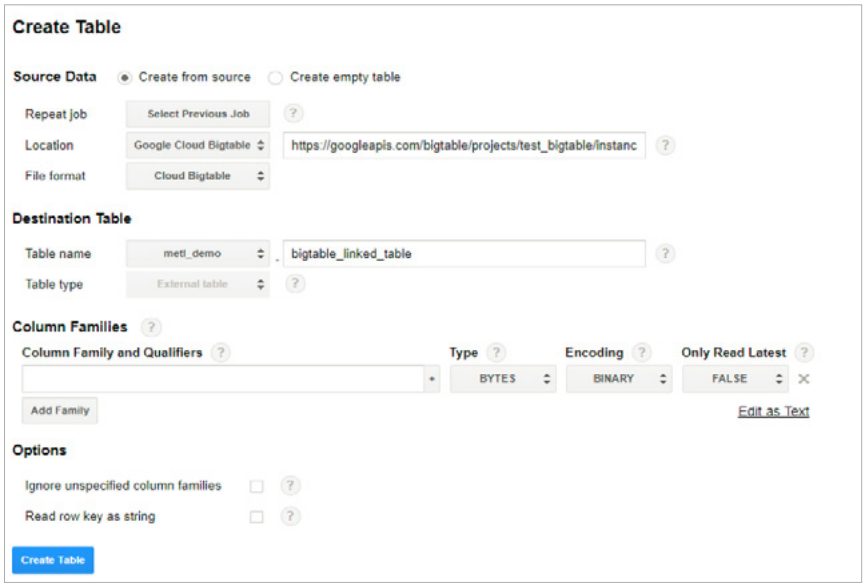
To create a Federated Data Source from Google Analytics data into BigQuery the recommended method would be to use Google Analytics 360. Google Analytics 360 has the necessary "BigQuery Export" functionality to link the data. To link the data using BigQuery Export follow the below steps:

1. Use the BigQuery web console in the **Create table** option
2. Select the **Location** as **Google Cloud Bigtable** and add the **URI** from above
3. The **Column Family** and **Qualifiers box** can be used to restrict the column families brought through into BigQuery and specify their types

Column Family

If this is left blank, ALL column families will be included in the linked table and they will all have a datatype of BYTE. This leads to inefficient storage and means you are unable to perform native functions you would expect. For example, if the column is a date, until it is converted to a datatype of 'date' no date functions will be available. Therefore, to make the most of your linked table we recommend including the Column Family details to correctly encode your columns.

4. Click **Create Table**



Once the table has been created in BigQuery it can be queried and live data will be returned from Google Bigtable. You may use this to bring operational data into BigQuery, so it can be aggregated, merged with other data and analyzed.

Linking to a Google Bigtable:

- ⊕ Best practice approach that is easy and efficient
- ⊖ Datatype will be inefficient if Columnar Family is not set, as this will default to BYTES

BIGQUERY DATA TRANSFER SERVICE

The BigQuery Data Transfer Service can be used to import data into BigQuery from other Google Data sources, including:

- Google AdWords
- DoubleClick Campaign Manager
- DoubleClick for Publishers
- YouTube - Channel Reports
- YouTube - Content Owner Reports

Further details are available **here**.

2.2 How to Load Other External Data into BigQuery

As well as loading data from Google sources, you may wish to bring other data external to Google into BigQuery. This could be data from a 3rd party provider API or from an on-premise system, such as an ERP or accounting system. This can be done by either streaming the data or batch loading data using a similar approach to loading the Google data discussed above.

BATCH LOADING

Batch loading is a particular type of loading. Similar to loading discussed above, batch loading takes a point-in-time copy of the source and carries it into a table in BigQuery. Querying copied data is significantly quicker than linked data as it is not dependent on the link as all data is stored locally in BigQuery. There are several options for batch loading your data.

We have found the most efficient way to batch load external data is to first copy it to a Google Cloud Storage bucket. Once the data is in Google Cloud Storage it can be loaded into BigQuery using the **loading method** discussed above.



With **Matillion ETL for BigQuery** you can also take advantage of components developed to streamline efforts, such as the built-in scheduler to ensure your BigQuery tables are all updated consistently at a convenient time interval.

Another way is to use tools specifically designed to help load external data, such as Matillion ETL for BigQuery. Matillion ETL, as with similar other tools, reduce the need for hand coded SQL and may offer a wider range of **integrations and connections**, thus reducing coding errors and allow you to easily access more data sources you want to bring into Google BigQuery.

Sign up for a demonstration of **Matillion ETL for Google BigQuery** and an opportunity to speak to a Solution Architect about your unique use case.

Another batch load option is to use Cloud Dataflow, which is discussed in the next section on streaming. Further details on Cloud DataFlow are available **here**.

STREAMING

Throughout the above sections we discussed linking data to create a live connection between the source data and the corresponding BigQuery table. Linking data enables you to have an up-to-date data source in BigQuery that is updated with every change made in the original source. The alternative to this is loading, which provides a snapshot of the data at the time it is copied.

There is, however, a third option, which is to live stream the data. This is a combination of the two benefits of the above because when new messages are generated in the source system they are immediately sent to BigQuery - you don't have to wait for a batch pull of data. Streaming your data, therefore, enables you to have up-to-date tables that you can query faster as the tables are still locally stored in BigQuery.

NOTE

Streaming data into BigQuery has a cost associated whereas loading data does not, therefore streaming data is only recommended for when there is a requirement to have high volume real time data in BigQuery. For further details please see **here**.

In this section we will discuss different methods for streaming data, starting with calling the BigQuery API to directly stream data into BigQuery. There are also other tools available in the Google Cloud Platform that can help ease the development and operational complexity of running a streaming application including Publish/Subscribe (Pub/Sub) and Cloud DataFlow. In some instances, using Pub/Sub as an intermediate may improve the efficiency of the API and reduce the amount of code to be written. An example of this is when data is coming from a third-party API. Google Pub/Sub can easily be set up to receive this data and the BigQuery API can be used to send this to BigQuery. Finally, for a simple no-coding approach Google Cloud DataFlow templates can be used to easily stream some data into BigQuery.

Streaming using the BigQuery API

We discussed a few ways you can interact with BigQuery in Chapter 1. Out of these options the most common are; the Google BigQuery **web console**, third-party tools such as **Matillion ETL** and a **command line interface** (which itself uses the Python API).

In most streaming cases we recommend using the BigQuery API to stream your data into BigQuery. This is because it gives you complete control through writeable scripts to define the data flow. Additionally, it is free, as opposed to alternative methods such as **Cloud DataFlow**.

BigQuery API

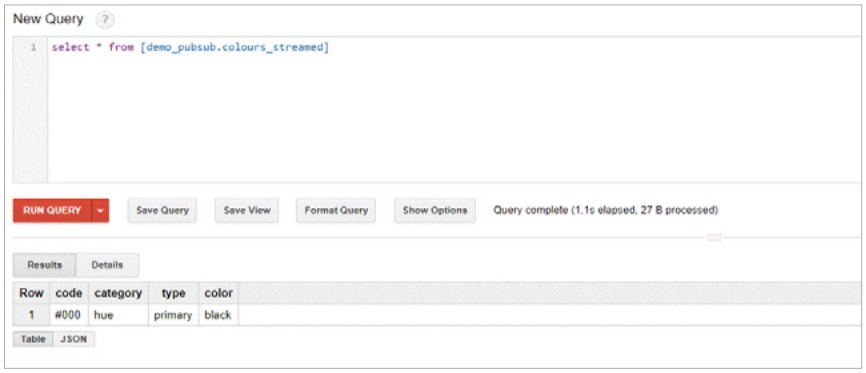
The API has a lot of other functionality that allows you to perform a range of tasks including creating, deleting, updating datasets, listing, querying and running jobs, and modifying tables. We recommend you familiarize yourself with this useful tool!

It also includes the option to stream data using the tabledata insertAll() or insert_data() command:

https://cloud.google.com/bigquery/docs/reference/rest/v2/tabledata/insertAll

In our example, we look at writing a simple Python script to stream data using the API, based on the example Python that is available on the bottom of the **BigQuery Streaming** support page.

1. Create a table in BigQuery as a target for the data. This table has been created with one row of data:



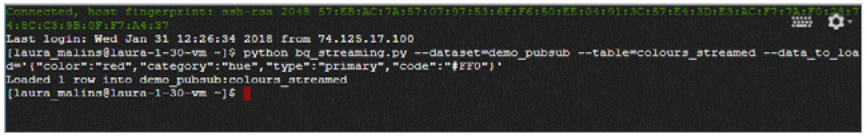
2. Using the example code from the **Google BigQuery Streaming API**, you can write a program which will take JSON data and load this into BigQuery. This could be used in conjunction with the output from, say an API call, to stream data directly into BigQuery

```
import argparse
import json
from pprint import pprint
from google.cloud import bigquery

def stream_data(dataset_name, table_name, json_data):
        bigquery_client = bigquery.Client()
        dataset = bigquery_client.dataset(dataset_name)
        table_ref = dataset.table(table_name)
        data = json.loads(json_data)

        # Reload the table to get the schema
        table = bigquery_client.get_table(table_ref)

        rows = [data]
        errors = bigquery_client.create_rows(table, rows)

        if not errors:
            print('Loaded 1 row into {}:{}'.format(dataset_name, table_name))
        else:
            print('Errors:')
            pprint(errors)


if __name__ == "__main__":
        parser = argparse.ArgumentParser()
        parser.add_argument('--dataset', help='Name of bq dataset', required=True, type=str)
        parser.add_argument('--table', help='Name of bq table', required=True, type=str)
        parser.add_argument('--data_to_load', help='Name of file to load', required=True, type=str)
        args = parser.parse_args()

        stream_data(args.dataset, args.table, args.data_to_load)
```

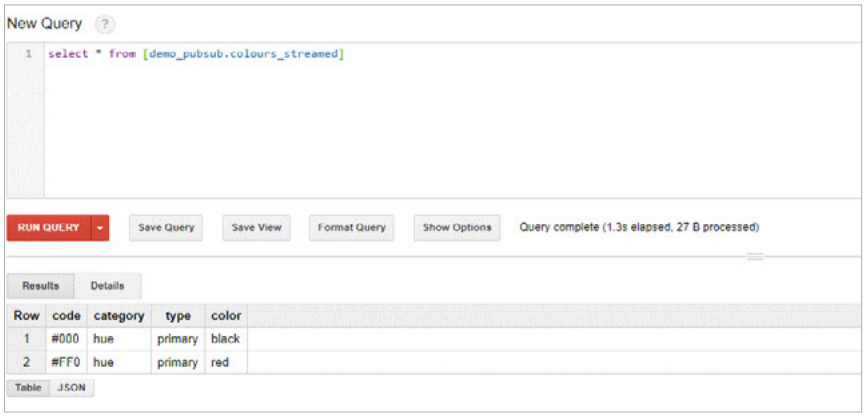
The Python code can be run with the three required arguments:

- **Dataset** — dataset in the BigQuery project to load to
- **Table** — name of the existing table to write data into
- **Data_to_load** — the JSON data to load into the table

3. Run the Python with the arguments:



4. The result in the command line indicates that the new row of data has been added and this can be viewed in the web console:



By using the BigQuery API for loading data, you have available in Python, you can get the same benefits as with streaming APIs for Publish/Subscribe, but without the data having to go via an intermediary. However, for the API to work well with the Python script, all data will need to be collated and formatted by the script, so this method isn't optimal if data is coming from a lot of different sources in different formats. The data format needs to be well formatted and consistent. If this is not the case, or writing the code is too involved or time consuming, you may wish to consider using Pub/Sub to help stream the data.

Streaming with BigQuery API:

- ⊕ Works well with Python which is free to use
- ⊖ Data does not have run through an intermediary, as with Pub/Sub, which can impact performance
- ⊖ Not optimal when streaming from a lot of different data sources/for mats

Publish/Subscribe into BigQuery

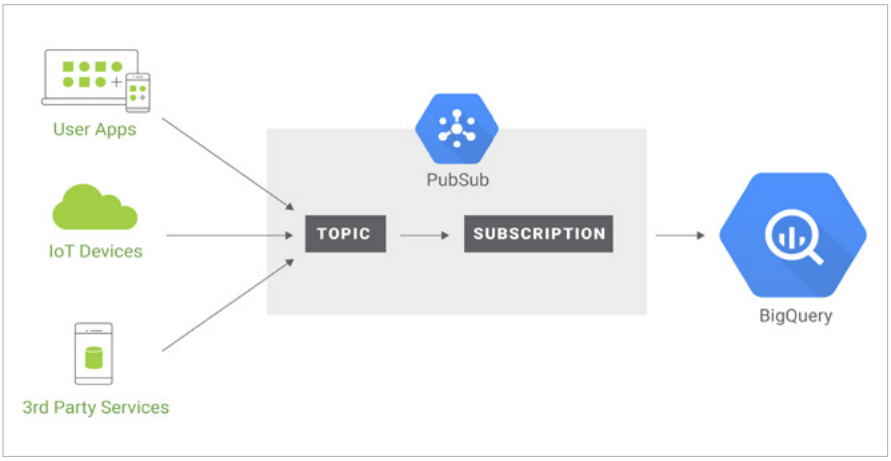
Google Publish/Subscribe (Pub/Sub) is a message-oriented middleware that sends and receives messages. It is therefore a tool that can help you stream data into BigQuery. Google Pub/Sub works by separating the messages from the sender and the receivers and allows independent programs to effectively communicate. It can be used to collate messages from many different sources in one place and stream the messages to different locations. You can use this to

monitor the progress of jobs running , push out updates to end users via email or for one application to notify another application that a particular job has run, and it is ready for a new update.

Google Pub/Sub consists of two items:

- A topic to store the messages in
- A subscriber to listen to the messages

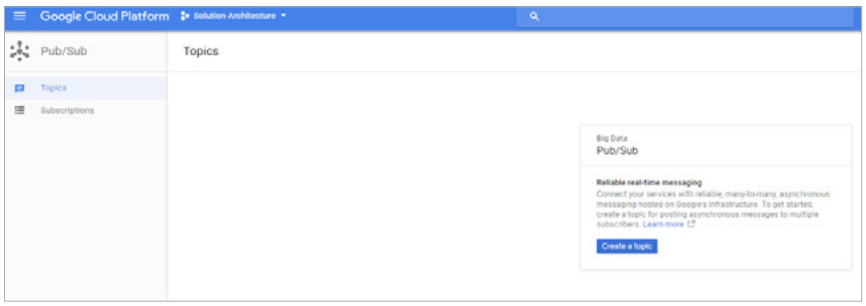
An external publisher can send messages to the topic and the subscriber can subscribe to the topic to receive the messages from it. This could be used to receive messages from external sources, such as a 3rd party service, end-user applications or IoT devices. These messages can then be collated in one place and processed accordingly.



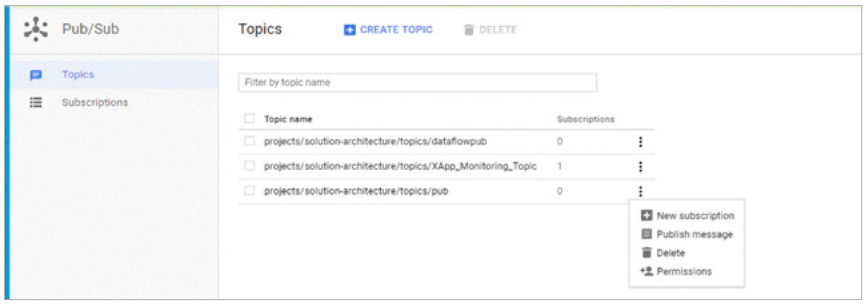
It is possible to load data from Pub/Sub into BigQuery, so all messages can be stored in a database to make analysis easy. For example, you may wish to collate IoT data from many devices and use this to evaluate how, when and for how long certain devices are used to understand trends and identify potential opportunities. In this example, we use Python with the BigQuery API to push manually entered messages from a Pub Topic to Google BigQuery.

---

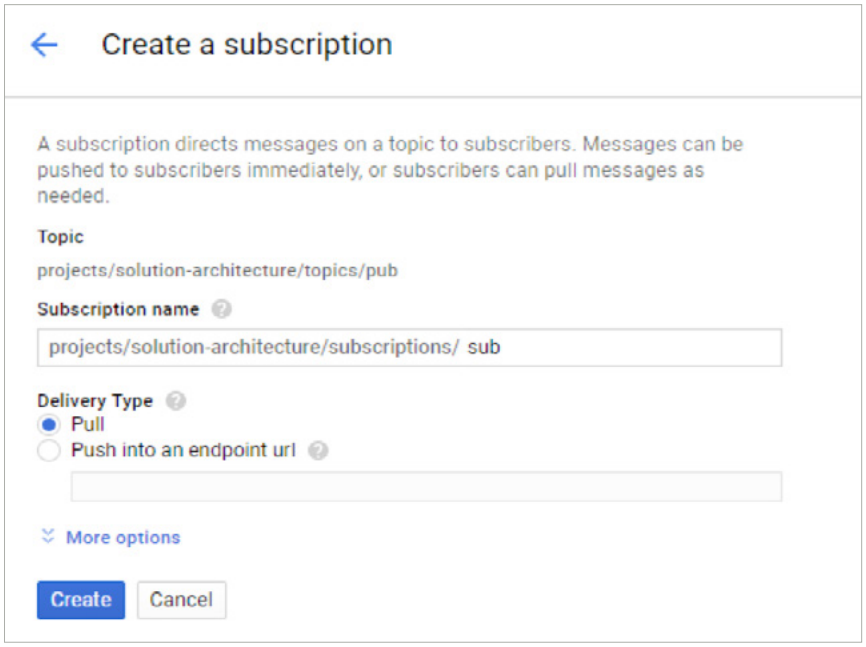
1. Create a new Pub **Topic** in the GCP Console:



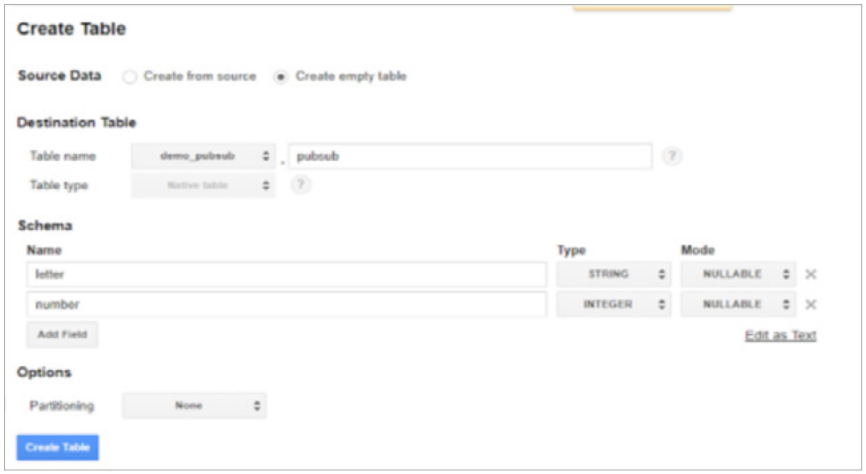
2. Add a **New Subscription** to the topic:



3. Create the Subscription as a **Pull**:



4. **Create empty table** in a BigQuery. This table will store the data from the messages. In this example, the messages will simply contain details of numbers and letters:



5. A simple Python script using the Google BigQuery API can be used to pull the messages from the subscriber and push them into a given table in BigQuery. This is done using the **google.cloud Python libraries**

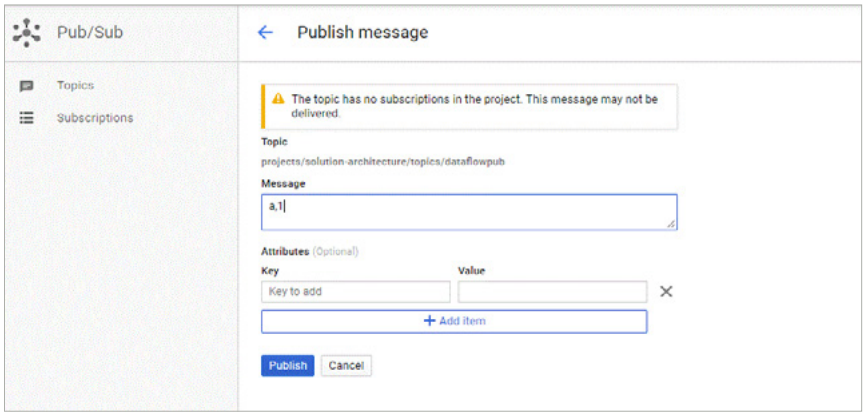
```
1   import argparse
2   from pprint import pprint
3
4   from google.cloud import pubsub
5   from google.cloud import bigquery
6
7   # ---------------------------------------------
8   # Set Vars
9   # ---------------------------------------------
10
11  topic_name = 'pub'
12  subscriber_name = 'sub'
13  dataset_name = 'demo_pubsub'
14  table_name = 'pubsub'
15
16
17  # ---------------------------------------------
18  # Streaming the data to BigQuery
19  # Batch Size = 10 messages
20  # pass through number of messages to parse
21  # ---------------------------------------------
```

NOTE

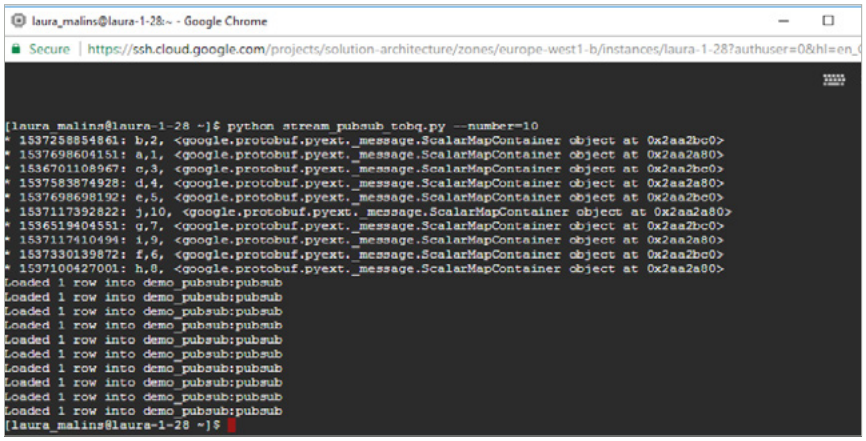
Lines 11 to 14 need to be edited with the **Relevant Topic, Subscriber, BigQuery Dataset and BigQuery Table**.

```
def stream_data_into_bigquery(number):
    pubsub_client = pubsub.Client()
    topic = pubsub_client.topic(topic_name)
    subscriber = topic.subscription(subscriber_name)

    bigquery_client = bigquery.Client()
    dataset = bigquery_client.dataset(dataset_name)
    table = dataset.table(table_name)

    #put messages into array
    batch = []
    i = 0
    while i < number:
        results = subscriber.pull(return_immediately=False)
        if results:
            for ack_id, message in results:
                print '* {}: {}, {}'.format(message.message_id, message.data, message.attributes)
                subscriber.acknowledge([ack_id for ack_id, message in results])
                batch.append(message.data)
                if len(batch) < 10: # batch size
                    i = i + 1
                else:
                    for j in xrange(0,len(batch)):
                        dict_ = {}
                        data = batch[j].split(',')
                        table.reload()
                        errors = table.insert_data([data])
                        if not errors:
                            print('Loaded 1 row into {}:{}'.format(dataset_name, table_name))
                        else:
                            print('Errors:')
                            pprint(errors)
                    del batch[:]
                    i = i + 1
```

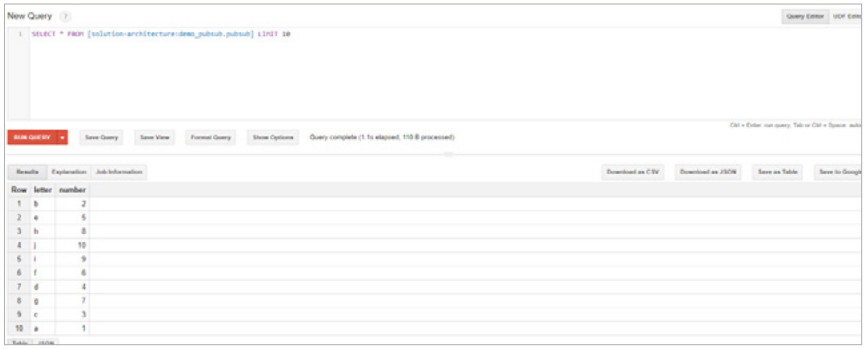
6.Messages can be manually added to the queue through the GCP console:



7. When 10 or more messages have been added, the Python script can be run to stream the messages to BigQuery. This should be done directly from the command line:



8. Here you can see the message content before they are loaded into BigQuery:



Now all data is available within one table in BigQuery, users can query this data and merge with additional external or internal data to review devices in their IoT to measure use or efficiency of devices. If devices are charged by hour, then the BigQuery data can be sent to the finance system to automate billing. This example works well with Pub/Sub rather than manually writing all the Python as the IoT devices can send their data directly to the Publisher, which can collate the messages in one place in a common format.

While you could use Cloud DataFlow, discussed in the next section, Python could also be used. The benefit of using Python is that it is free to run, however, you have the overhead of writing and setting it up as well and maintaining it. The Python also requires a server to run on which needs to be constantly available. If you wish to avoid writing Python all together and want a simple to set up managed service to stream the data, Google Cloud DataFlow is the option for you.

Publish/Subscribe (Pub/Sub):

- ⊕ Python has minimal associated costs (provided you have a service to run it)
- ⊕ Full control has you write the code
- ⊖ Data is run through an intermediary, which can impact performance
- ⊖ Must manually maintain code

Google Cloud DataFlow

Google Cloud DataFlow, another Google managed service, can also help with streaming data into BigQuery. It comes with some easy to configure templates which make setting up the data stream very simple. For example, you can easily create a DataFlow job to stream data from Pub/Sub to BigQuery. You may wish to do this to build a logging table for an application which is running. The application can publish messages up to a topic when it runs successfully or fails, and these messages can be picked up by the DataFlow and streamed to a BigQuery table.

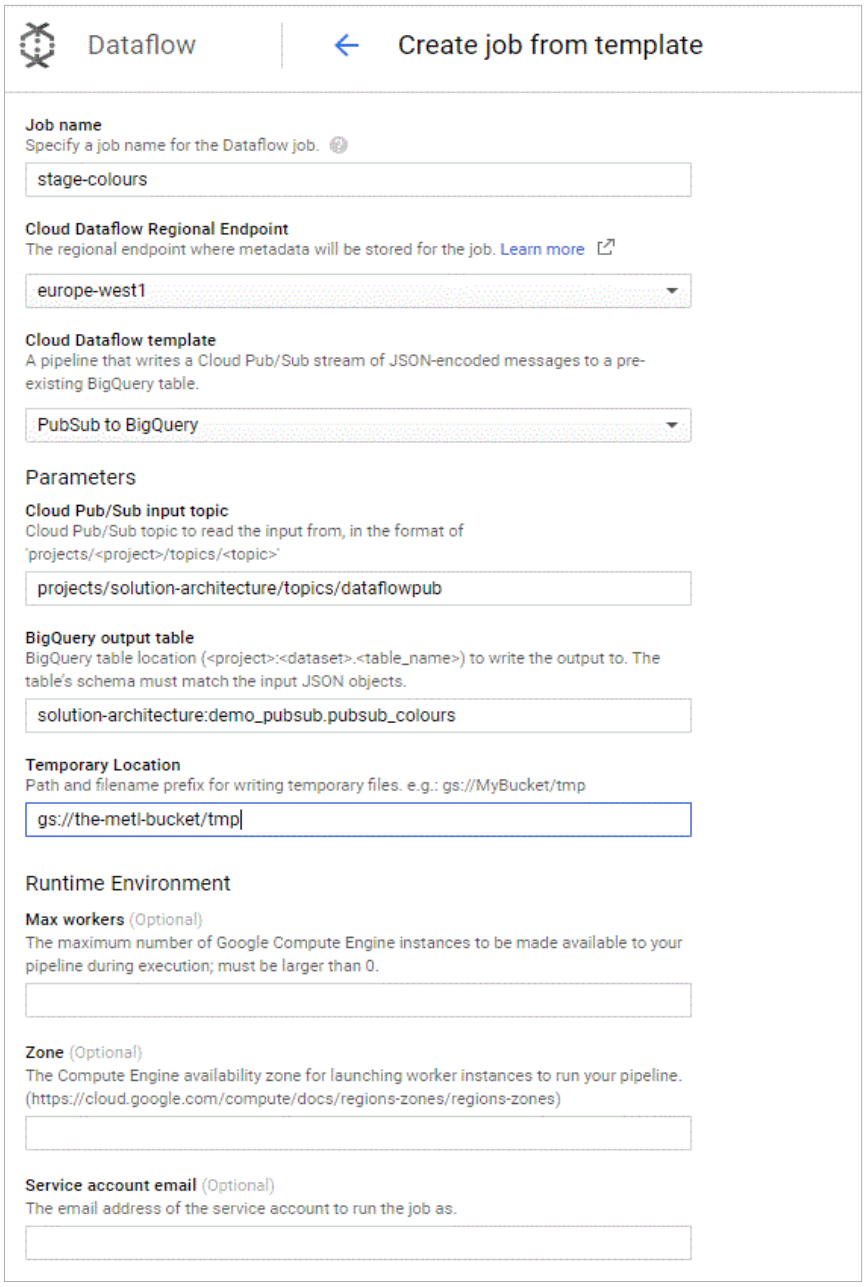
Google Cloud DataFlow can be run from the Google Cloud Console to either stream or batch load data from a variety of other sources, including data files stored in Cloud Storage Buckets, and DataStore into a variety of destinations including Pub/Sub, Cloud Storage Buckets and BigQuery.

Here we look at an example which takes messages from a topic and streams them to a BigQuery table. Again, we are using the 'colors' dataset where the messages are in a Newline delimited JSON format, which is a requirement for Cloud DataFlow.

1. First Create a blank table in BigQuery to hold the messages. We did this by uploading a simple JSON file with the contents of one message into the **Create New Table** option in BigQuery and allowing BigQuery to automatically detect the schema

2. Next navigate to **DataFlow** in the Console and select **Create Job from Template**

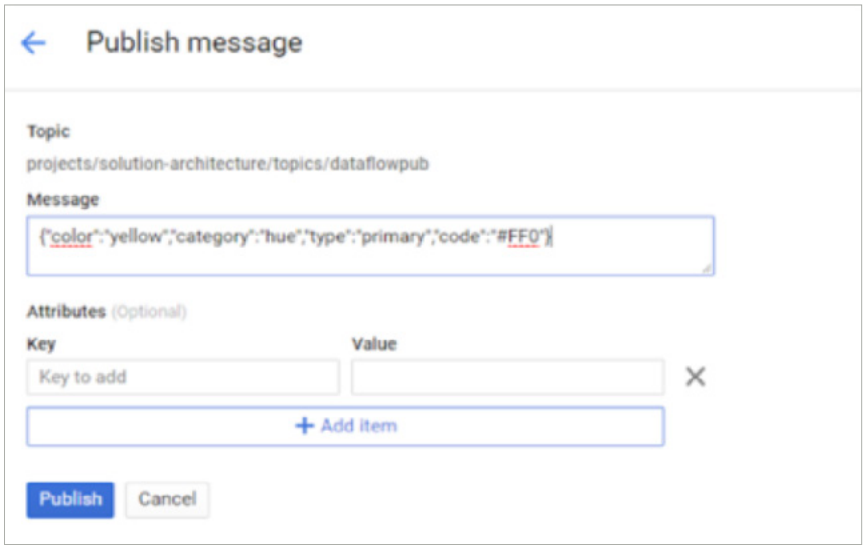
3. In the template give a **Job Name** and select a Regional **Endpoint**. Then fill in the topic details of what the message is on and the **BigQuery output table** the data is to be streamed to:



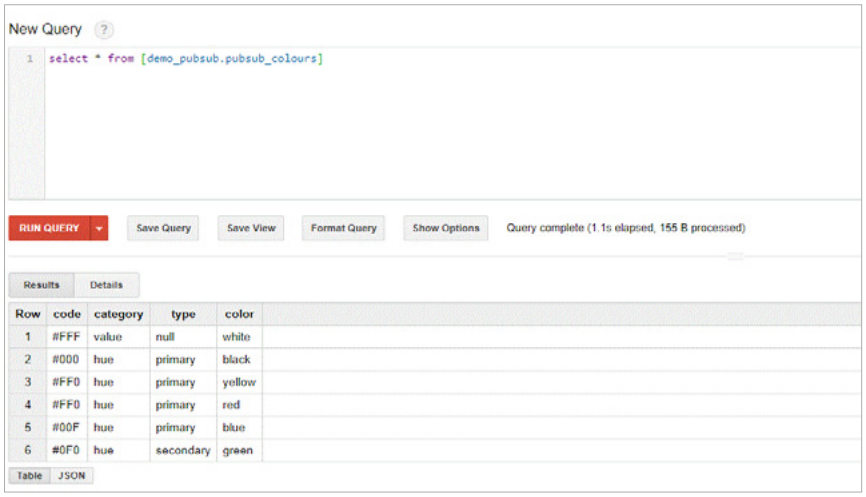
4. The Job will successfully **create** after hitting the Create button. You can monitor the job by clicking on it:



5. New messages can now be manually published to the topic in the required JSON format:



6. These messages will filter through to the BigQuery table



The DataFlow job will continue running and therefore, continue streaming data until it is manually stopped.

As we have seen, Cloud DataFlow is very easy to set up and is a great way for less technical BigQuery users to stream data from an endpoint into BigQuery without having to write any code. This solution is ideal for business analysts to take data from one or many end user systems which are able to publish data up to a topic. From there this data can be collated together in one table in BigQuery and then exposed to visualization tools for further analysis.

Google Cloud Storage

- ⊕ Code free approach to streaming data
- ⊕ Data does not have run through an intermediary, as with Pub/Sub, which can impact performance
- ⊖ Paid service
- ⊖ Data has to be in Newline delimited JSON format

We can't replace time that you've lost ... but we can give you more of it in the future.

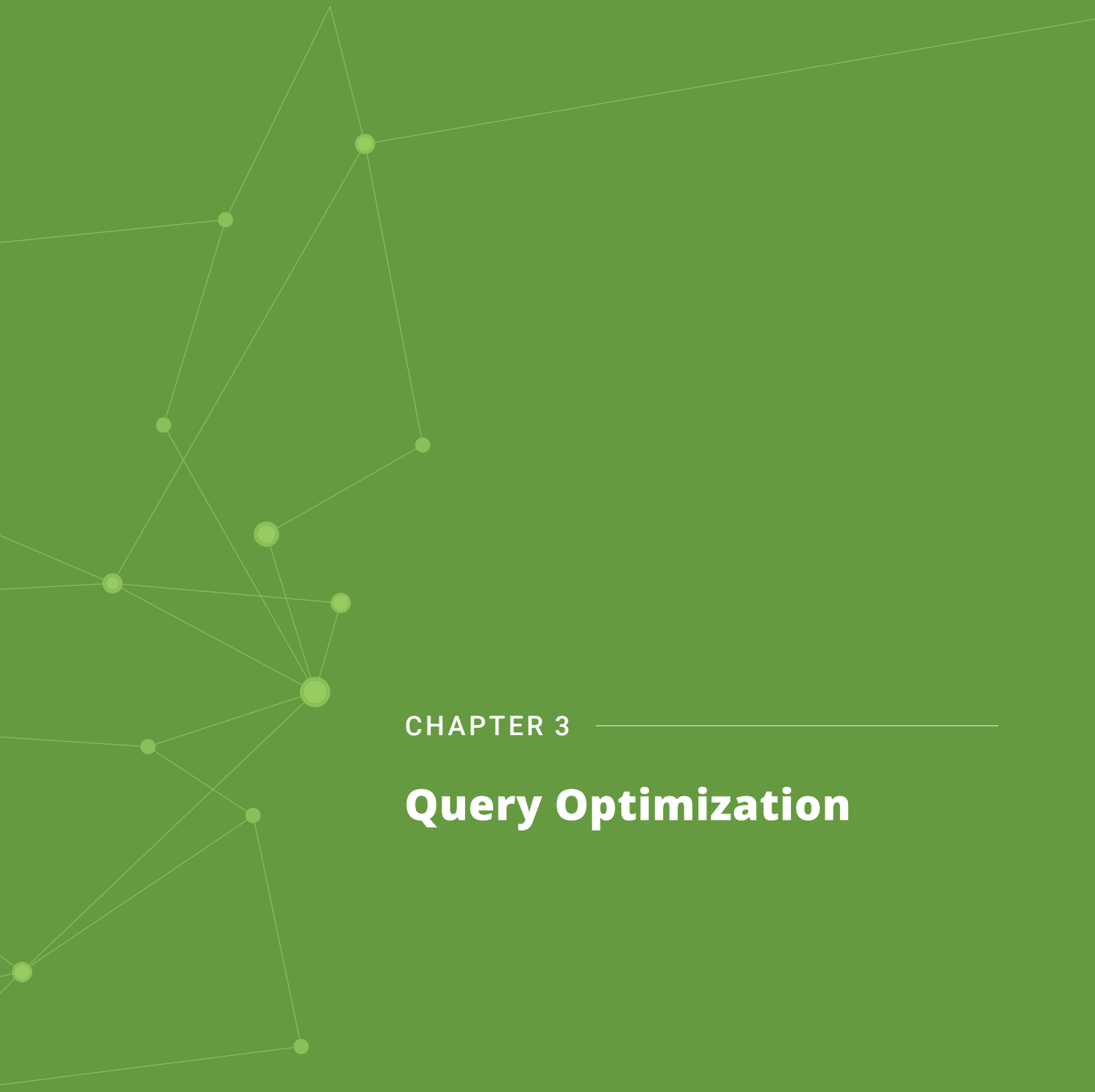
Matillion ETL for BigQuery installs in five minutes—and can cut months off of your projects.

- Optimizes your data warehouse
- Supports standard SQL
- Connects to the Google Cloud Platform

Find out more at
www.matillion.com/etl-for-bigquery

MATILLION

Google Cloud Platform



CHAPTER 3

Query Optimization

CHAPTER 3

Query Optimization

Once you have BigQuery populated with data you can start querying that data to get valuable insights. This section includes tips on how get the most of Google BigQuery when retrieving data. Google BigQuery is a managed service but nevertheless there are simple ways that you can optimize your queries to maximize performance and minimize cost.

Query Optimization can:

- maximize performance
- minimize cost

When queries are executed there is only one **pricing factor**: the number of bytes processed. You'll get the best performance, at the best price, if you minimize the amount of work Google BigQuery has to do in order to satisfy a query.

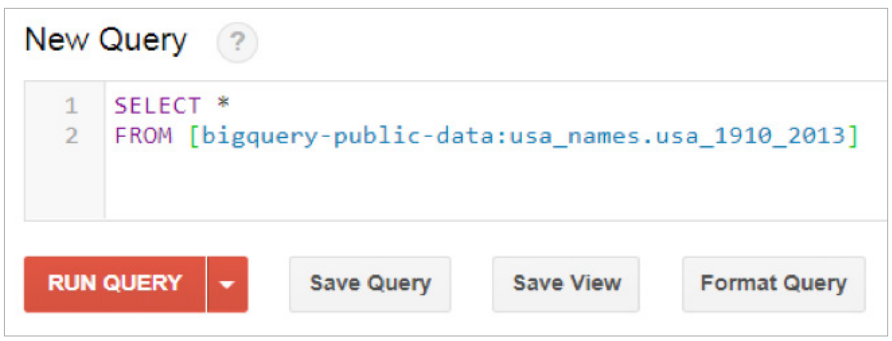
The following sub-sections of this chapter will provide guidance and recommendations for how you can run queries in an optimized way.

3.1 Avoid SELECT*

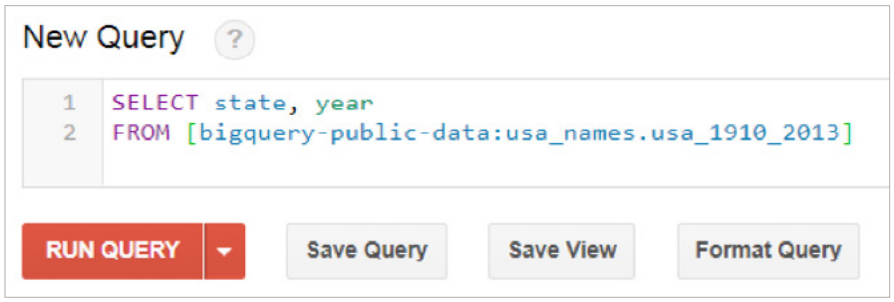
This is the simplest and most important technique, both for performance optimization and cost reduction.

BigQuery data is held in a *columnar* format. This means that the data for the different columns is held in physically different places. Unlike an OLTP database, the column values for every record are not stored together in a row-oriented format.

If you query using a SELECT *, BigQuery has to read ALL the storage volumes.



Whereas if you query only certain columns using a SELECT col1, col2, col3… then BigQuery only needs to retrieve data for the selected columns.



If your query is being used as the basis to materialize a new table later, there's a second bonus. The new table will only contain the necessary columns and will, therefore, be cheaper and faster to store and query in turn.

SELECT

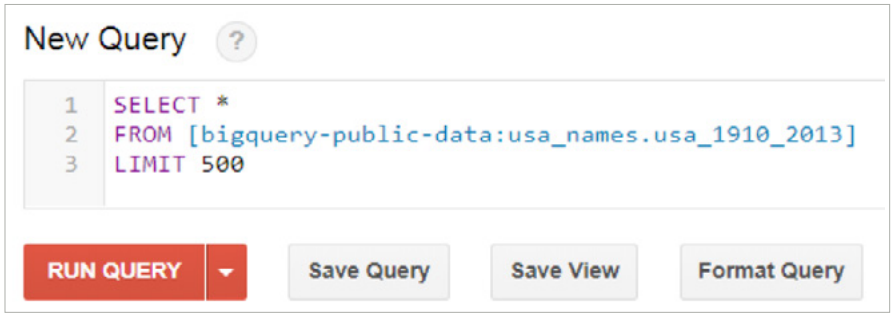
- ✗ SELECT*, which selects all columns
- ✓ SELECT specific columns to reduce the amount of data scanned

---

- ★ Cost
- ★ Performance

3.2 LIMIT and WHERE clauses

Note that the LIMIT clause *DOESN'T* affect query cost. LIMIT is a final step which restricts the amount of data shown, after the full query has been executed.



Similarly, a WHERE clause is applied *AFTER* data has been scanned. The cost of a query depends on the number of bytes read by the query *BEFORE* applying the WHERE clause.

This is an important consideration when designing data storage structures for ETL workflows.

If you find you are repeatedly querying a large base table, consider the following alternatives:

- Query it just once, using a superset of all the necessary WHERE clauses, and materialize an intermediate table. Then repeatedly query the **intermediate** table, using individual WHERE clauses.
- Use partitioning on the base table. See **Partitioning** for more details.

If you intend to perform ordering of a result set, LIMIT can be a useful way to reduce the amount of work required. A global ordering of all results can be time-consuming for BigQuery to perform.

LIMIT

- ✗ Doesn't limit the amount of data scanned
- ✓ Can be used to avoid having to order all results of a query

WHERE

- ✗ Isn't considered before the data is scanned
- ✓ Use an intermediary table based on WHERE clauses with you can query again and again
- ✓ Partition the data

- ★ Cost

3.3 WITH clauses

These are sometimes known as Common Table Expressions (CTEs). They can be useful syntactically, but are not materialized at runtime.

Similar to the previous tip, while it looks like the CTE will only need 1 scan, it will actually use 2 meaning your query will cost double and take twice as long as you might expect. If you're executing a complex CTE repeatedly, the solution is to use the CTE beforehand to create an intermediate table, and then use the intermediate table in the SQL rather than the complex CTE expression.

Example of executing a CTE repeatedly

```
#standardSQL
WITH t AS (SELECT homeTeamId AS Id, homeTeamName AS Name
        FROM `bigquery-public-data.baseball.schedules`
        GROUP BY homeTeamId, homeTeamName)
SELECT h.Name AS HomeTeamName, a.Name AS AwayTeamName,
COUNT(*)
FROM `bigquery-public-data.baseball.schedules` s
JOIN t h ON h.Id = s.homeTeamId
JOIN t a ON a.Id = s.awayTeamId
GROUP BY h.Name, a.Name
```

If you intend to perform ordering of a result set, LIMIT can be a useful way to reduce the amount of work required. A global ordering of all results can be time-consuming for BigQuery to perform.

WITH

- ✗ Not materialized at runtime
- ✓ Use an intermediary table rather than a CTE

- ★ Cost
- ★ Performance

3.4 Approximate Aggregation

You may be able to improve query performance when gathering some statistics by using an approximation function.

For example, during data profiling you might need to know the range of values occurring in a column. The *exact* method is to use COUNT(DISTINCT).

```
#standardSQL
SELECT COUNT(DISTINCT year)
FROM `bigquery-public-data.baseball.games_wide`
```

The equivalent *approximation* method, which would execute more quickly, is:

```
#standardSQL
SELECT APPROX_COUNT_DISTINCT(year)
FROM `bigquery-public-data.baseball.games_wide`
```

Other **approximation functions** are also available for calculating quantiles, and for finding the most frequent values.

For the largest datasets, BigQuery offers **HyperLogLog++** functions which use an efficient algorithm for estimating the number of distinct values.

The equivalent HyperLogLog++ approximation method that executes quickly on a very large dataset, is:

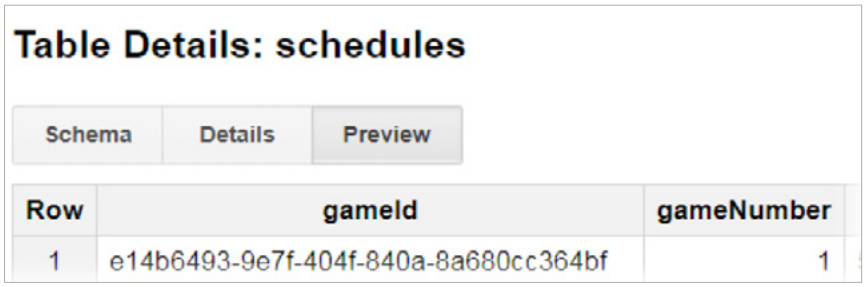
```
#standardSQL
SELECT
  HLL_COUNT.EXTRACT(year_hll) AS num_distinct
FROM (
  SELECT HLL_COUNT.INIT(year) AS year_hll
  FROM `bigquery-public-data.baseball.games_wide`)
```

3.5 Efficient Preview Options

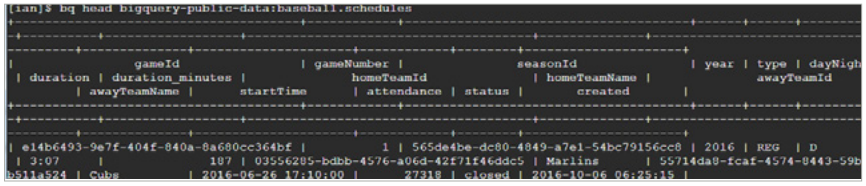
During your project's data exploration phase, Google recommends that you *DON'T* use SELECT * ... LIMIT to preview data (for the reasons stated above).

Instead, use one of the three following alternatives.

1. From the web console, press the **Preview** button on the Table Details page



2. From the CLI, use the **"bq head" command** and specify a limited number of rows



3. With the **API**, use the tabledata.list entry point, specifying which rows are needed. Please see **this document** for more details

Preview Options:

- ✗ Don't use SELECT * ... LIMIT
- ✓ Preview within the console, use the 2bq head2 command or use an API

- ★ Cost
- ★ Performance

3.6 Early Filtering

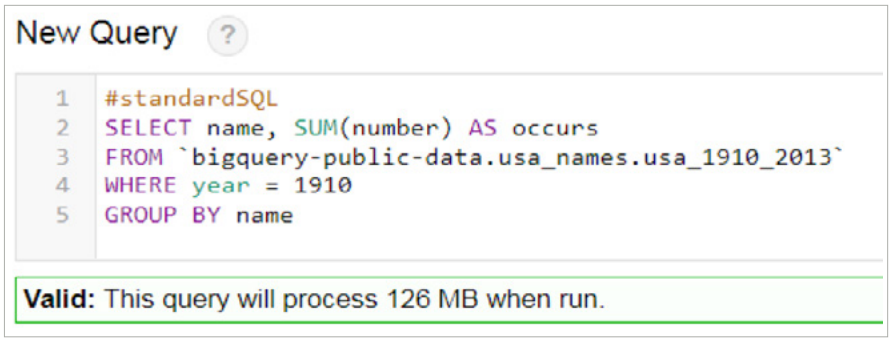
Filtering is fundamentally a performance-tuning recommendation aimed at helping Google BigQuery process queries as quickly as possible.

During query execution involving multiple slots, Google BigQuery often has to copy data between the slots in order to perform a join. This is known as **shuffling**. Filtering the tables in the join reduces the amount of data that needs to be shuffled.

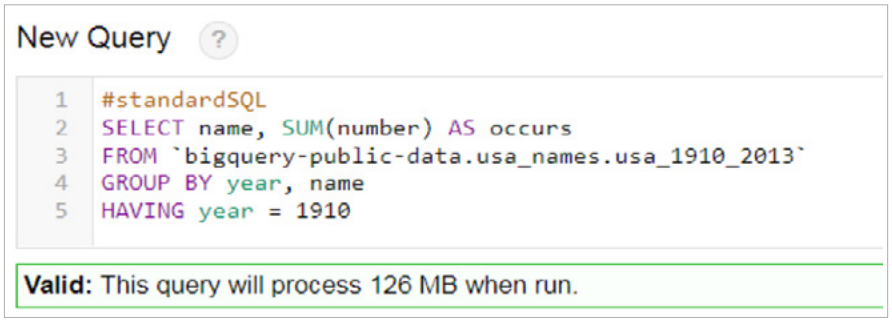
Shuffling requires network bandwidth and transient data storage. Slots do have a high but finite memory limit. If the amount of data shuffled exceeds this limit then queries will still execute, but disk caching will be required. This will adversely impact performance.

With WHERE, the filter is done before the aggregation, while HAVING it's done after the aggregation. This means that while the HAVING clause will achieve the same result, at the same cost, it will perform slower since it requires BigQuery to do more work. Therefore, using a WHERE clause is the optimal method.

Example using a WHERE clause



Same logic, expressed with a HAVING clause



Early Filtering

- Avoid HAVING clauses when able, instead use WHERE
- Reduce the amount of data being shuffled

- Performance

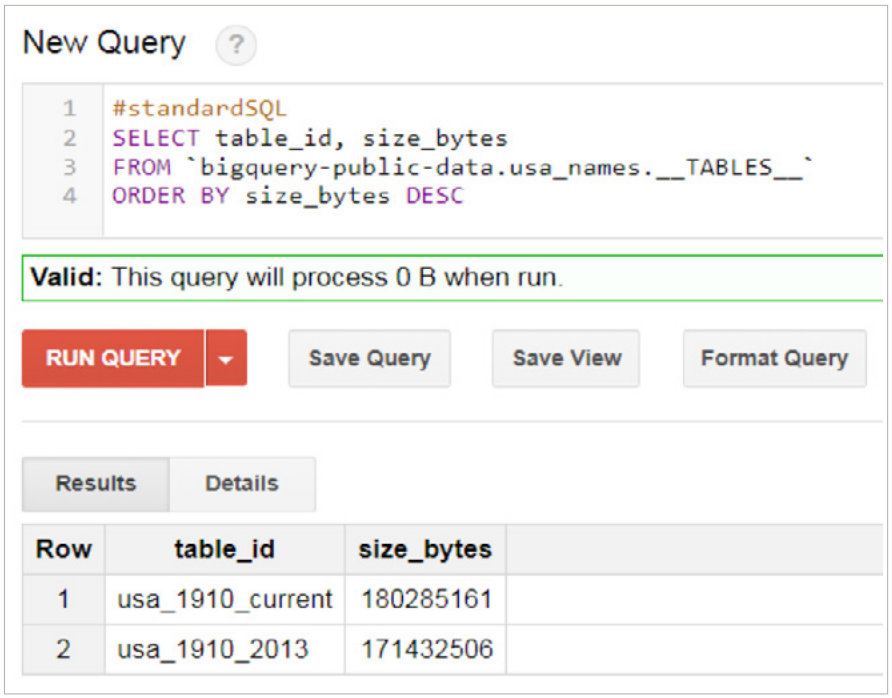
3.7 JOIN Clause Optimization

The ordering of tables in a WHERE clause is significant and can help the SQL query 'optimizer' (see **Optimize your join** patterns). Google recommends that you use the following ordering when performing joins:

- largest table first
- smallest table next
- the other tables, in descending size order

Performing joins in this order enables Google BigQuery to evaluate them in the most efficient way, by minimizing the amount of intermediate data required during execution.

To check the table sizes, query the built-in __TABLES__ object like this:



JOIN clause optimization

- ✓ Optimize your WHERE clauses by joining largest table first, smallest table next and then the other tables, in descending size order

- ★ Performance

DATE PARTITIONING

There's a great advantage if you have a date partitioned table. Simply make sure to use a WHERE clause predicated on the _PARTITIONTIME pseudocolumn, and Google BigQuery will only have to read data from the partitions specified by the date filter. This will reduce the data volume scanned and will also improve performance. See **Partitioning** for more details.

Date Partitioning

- ✓ Reduces the amount of data scanned

- ★ Cost
- ★ Performance

WILDCARD TABLES

It's possible to run a SELECT with a **wildcard** in the FROM clause, and this is often useful with manually sharded tables.

```
#standardSQL
SELECT _TABLE_SUFFIX AS yr,
    provider_state,
    ROUND(SUM(average_total_payments), 0) AS average_total_payments
FROM `bigquery-public-data.cms_medicare.inpatient_charges_*`
GROUP BY _TABLE_SUFFIX, provider_state
```

Whenever possible, express a filter in the wildcard rather than in the WHERE clause. In exactly the same way as date partitioning, this will both reduce the data volume scanned and improve performance.

When using a wildcard expression, you can take advantage of the _TABLE_SUFFIX pseudocolumn to achieve more sophisticated filtering. _TABLE_SUFFIX is available for every record and contains the string value matched by the wildcard part of the FROM clause.

For example, to restrict the above query to only consider 2014 and 2015, you could add:

```
WHERE _TABLE_SUFFIX = '2014' OR _TABLE_SUFFIX = '2015'
```

Wildcards

- ✗ When using wildcard tables, filter in the FROM clause in preference to using WHERE
- ✓ Use SELECT with a wildcard in a FROM clause with manually sharded table

---

- ★ Cost
- ★ Performance

3.8 Analytic Functions

Analytic function recommendations primarily concern cost control, although they should also result in performance improvements.

When calculating derived measures, it's common to require a value that relates to a group of records, rather than just the one at hand. A typical example is calculating the percentage represented by a single value. To calculate the percentage, you need to first calculate the *total* for that group of records. Only then can you find the proportion of each contributing record.

Without **analytic functions** you'd have to scan the data twice; (in other words using a self-join). First with an aggregate to find the group-level totals, and then after joining to the same data again, a second time to calculate the record-level percentages.

Without an analytic function (note the table is queried twice)

```
#standardSQL
WITH tot AS (SELECT year, SUM(homeFinalRuns) AS TotHomeFinalRuns
             FROM `bigquery-public-data.baseball.games_post_wide`
             GROUP BY year)
SELECT g.year, g.homeTeamName,
       SUM(homeFinalRuns) AS homeFinalRuns,
       SUM(homeFinalRuns) / tot.TotHomeFinalRuns AS RunFraction
FROM `bigquery-public-data.baseball.games_post_wide` g
JOIN tot ON tot.year = g.year
GROUP BY g.year, g.homeTeamName, tot.TotHomeFinalRuns
```

But with an analytic function SUM() in this case, only one scan is needed. Google BigQuery is able to calculate the group-level values at the same time.

Same logic, using an analytic function

```
#standardSQL
WITH tot AS (SELECT year, homeTeamName,
                    SUM(homeFinalRuns) AS TotHomeFinalRuns
             FROM `bigquery-public-data.baseball.games_post_wide`
             GROUP BY year, homeTeamName)
SELECT year, homeTeamName,
       TotHomeFinalRuns AS homeFinalRuns,
        TotHomeFinalRuns / SUM(TotHomeFinalRuns) OVER (PARTITION BY
year) AS RunFraction
FROM tot
```

Similar techniques can be used to calculate:

- Year-on-year comparison
- Valid-from and valid-to calculations in a slowly changing dimension

Analytic Functions

✓ Reduce duplicated scanning efforts

★ Cost

★ Performance

3.9 ORDER BY Clauses

The data sort operation is unusual in that it does not get parallelized: it is always performed on a single slot. It's advisable therefore to minimize the amount of work that this lone slot must do.

This means simply not using an ORDER BY if you don't need it. When you do need an ORDER BY clause you should perform it last, so as to avoid ordering intermediate results.

Individual slots have a finite amount of memory, and queries with an ORDER BY that return more than this amount of data will fail with a "Resources Exceeded" or "Response too large" error (depending if the target is a named or anonymous table). The obvious solution to this problem is to reduce the amount of data. This can be achieved easily by filtering the original data more extensively, or artificially restricting the output with a LIMIT clause.

NOTE

This advice does *not* apply to the ORDER BY within analytic functions, which were described separately earlier in the chapter.

ORDER BY

- ✗ Don't use ORDER BY unless it is necessary
- ✓ Perform ORDER BY last
- ✓ Use filters to reduce the amount of data
- ✓ Restrict data with LIMIT clauses

- ★ Cost
- ★ Performance

3.10 Materialize Common Joins

This recommendation applies if you find that you are repeatedly performing a join between two tables. For example, the public Major League Baseball dataset has a detailed table containing game commentary, and a reference table listing all the scheduled games.

Even if you have WHERE clauses on both tables, it's likely to be cheaper and faster, in the long run, to materialize this join into an intermediate table.

That means:

- Run the join once to create the new intermediate table
- Query the intermediate table repeatedly (no join is required)

There are two choices with respect to the granularity of the intermediate table.

1. At the detailed table level - Denormalize and repeat values from the parent table. This data will usually compress well, and so should not add too much cost when accessed.

2. At the parent table level - Use nested data structures to hold multivalued details. See Denormalization for more details.

Query at the detailed table level (341 records for the chosen game)

```
#standardSQL
SELECT p.homeTeamName, p.awayTeamName, d.gameId, d.seasonId,
d.createdAt, d.description
FROM `bigquery-public-data.baseball.games_wide` d
LEFT JOIN `bigquery-public-data.baseball.schedules` p
        ON p.gameId = d.gameId
        AND p.seasonId = d.seasonId
WHERE d.gameId = 'dc42dfe7-d6dd-4831-a9ad-c1dcfc8f62af'
AND d.seasonId = '565de4be-dc80-4849-a7e1-54bc79156cc8'
```

Query at the parent table level (one single record)

```
#standardSQL
WITH x AS (
SELECT p.homeTeamName, p.awayTeamName, d.gameId, d.seasonId,
d.createdAt, d.description
FROM `bigquery-public-data.baseball.games_wide` d
LEFT JOIN `bigquery-public-data.baseball.schedules` p
        ON p.gameId = d.gameId
        AND p.seasonId = d.seasonId
WHERE d.gameId = 'dc42dfe7-d6dd-4831-a9ad-c1dcfc8f62af'
AND d.seasonId = '565de4be-dc80-4849-a7e1-54bc79156cc8')
SELECT d.homeTeamName, d.awayTeamName, d.gameId, d.seasonId,
    ARRAY(SELECT AS STRUCT createdAt, description
        FROM x
        WHERE x.gameId = d.gameId
        AND x.seasonId = d.seasonId)
FROM (SELECT DISTINCT homeTeamName, awayTeamName,
        gameId, seasonId
    FROM x) AS d
```

Materializing and querying the parent-level statement would have the additional advantage that if the commentary was not required, the nested array could simply be omitted.

Common Jobs

- ✗ Avoid running the same join over and over again
- ✓ Use an intermediary table on common joins

- ★ Cost
- ★ Performance

3.11 Avoid OLTP Patterns

Google BigQuery's unique architecture leads to great batch mode and analytic SQL performance. But it can be easy to accidentally slip into design patterns from the OLTP world, which are usually not optimal. This section lists some common patterns to watch out for.

AVOID USER-DEFINED FUNCTIONS (UDFS)

Google BigQuery allows you to create JavaScript User-Defined Functions (UDFs) in both legacy SQL and standard SQL. This programming model gives enormous flexibility, since a UDF can change both the schema and the granularity of the data.

However, UDFs are subject to various limits especially around concurrency and memory. UDF execution requires the dynamic instantiation of server-side resources (V8 engine instances), which can result in poor performance.

Google recommends that wherever possible you avoid UDFs and instead use SQL or **Analytic Functions**.

Avoid User-Defined Functions

- ✗ User-Defined Functions (UDFs)
- ✓ SQL or analytic functions

- ★ Cost
- ★ Performance

AVOID HIGH-COMPUTE QUERIES

At the start of the chapter we stated that the only factor determining query cost is the number of bytes processed. While that's usually true, Google does reserve the right to prevent the execution of "High-Compute" queries which might only query a very small amount of data, but which nevertheless still require a large amount of compute infrastructure.

There's no hard and fast definition of what a "High-Compute" query is, but generally they involve either large numbers of joins, or else require the execution of complex UDFs (see above). You won't know that your query has been classified as "High-Compute" until it fails with a 'billingTierLimitExceeded' error. Such queries must run at a higher rate than the "$5 per TB" default, and need an account administrator to enable them.

Avoid High-Compute Queries

- ✗ UDFs
- ✓ Split a query involving many joins into multiple separate queries, each doing a subset of the joins and generating an intermediate table

---

- ★ Cost
- ★ Performance

AVOID SINGLE-ROW DATA MANIPULATION LANGUAGE (DML)

A side effect of Google BigQuery's columnar data storage model is that single-record updates are actually extremely inefficient. There is no single "record" object to update. Instead, a single record physically spans multiple columns, and each of them may need to be accessed and updated individually.

To work on multiple rows efficiently:

- DON'T: loop through them one by one, updating each record individually
- DO: express the same logic using batch SQL, for example with CASE statements

In fact, it's probably best to entirely avoid updates. Rather than updating a table, a better approach is usually to create a second table based on the first with SQL calculations applied. After successful creation of the second table you can optionally drop the first.

Avoid Single-Row DML

- ✗ Avoid updates all together
- ✓ Recreate tables and drop the previous table

- ★ Cost
- ★ Performance

AVOID SKEW

Skew can be a difficult problem to detect and solve. The root cause is multiple slots servicing a query, but some end up doing far more work than others.

The way to discover if skew is having an adverse effect is to check for large differences between the 'average' and the 'maximum' compute time ratios in the query execution plan. You'll need to check every stage of the plan, since skew can occur at almost any point.

A classic avoidable cause of skew is if you perform a join involving a nullable or low-cardinality column. During execution of the join, all rows with a certain column value will be shuffled into one slot. If there are 10 slots, but half the values are null, then one of the 10 slots will have to do fully half the total work on its own.

You could deskew the join by using an equality join on a calculated column like this:

```
CASE WHEN key IS NOT NULL THEN key
ELSE '#' || other_column
END
```

Prefixing a '#' prevents the join from working (exactly the same as a NULL value), but the records would be distributed much more evenly among the slots by the addition of a (basically random) second column.

Problems with skew really don't exist in OLTP systems, where the reverse is normally true. With an OLTP system it's better to be highly specific and have low-cardinality with joins, since they can then take best advantage of indexes.

Avoid Skew

- ✗ Avoid being highly specific
- ✗ Avoid joins involving a nullable or low-cardinality column
- ✓ Identify skew and use an equality join to rectify

- ★ Cost
- ★ Performance

PARTITION PRUNING

Regardless of the chosen partitioning method, partitioning can greatly improve performance and reduce cost by enabling BigQuery to fully service a query while only accessing data from a subset of the whole. The method of being very selective about which partitions are involved at runtime is known as 'Partition Pruning'.

To achieve partition pruning on sharded tables, you need to use legacy SQL, and can address individual partitions using a special "partition decorator" syntax. For example:

```
#legacySQL
SELECT col1, col2
FROM [your_proj:your_ds.partitioned_table$20160129]
```

To achieve partition pruning on a date-partitioned table, you just need to include a WHERE clause predicate on the _PARTITIONTIME pseudocolumn, like this:

```
#standardSQL
SELECT col1, col2
FROM `your_proj.your_ds.partitioned_table`
WHERE _PARTITIONTIME BETWEEN TIMESTAMP('2016-01-29')
                     AND TIMESTAMP('2016-01-30')
```

NOTE

There are still a few circumstances where a predicate on _PARTITIONTIME nevertheless still results in a scan of all partitions. Please see **this document** for more information.

HISTORICAL LOAD

Most data projects servicing a reporting layer have two load phases:

- **Initial** ("historical") load - one-off, higher volume
- **Repeated** ("incremental") loads - recurring, lower volume

To load historical data into a sharded table, you simply use "partition decorators" to name the target partition.

At the time of writing this eBook, BigQuery does not support DML statements over partitioned tables. So, to load data into a date-partitioned table you must begin with sharded tables, and then **convert them into a date-partitioned table**.

Partitioning

- ✗ DML not supported
- ✗ Partition Pruning may still scan all date partitioned data
- ✓ Partition using Sharing or date partitioning

- ★ Cost
- ★ Performance

3.12 Denormalize

This is a data modelling and design optimization which will help your queries to perform best, and at the lowest cost.

BigQuery has been designed to perform best when your data is denormalized. You should ideally design your target schema with nested and repeated fields rather than using a star or snowflake schema.

Denormalized data helps BigQuery to take the most advantage of its inbuilt parallelism, by ensuring that no data shuffling is needed. The additional storage needs associated with denormalization will be more than offset by the gains in runtime performance.

Google BigQuery has extra datatypes which enable you to extend the classic relational model to include nested structures. These are:

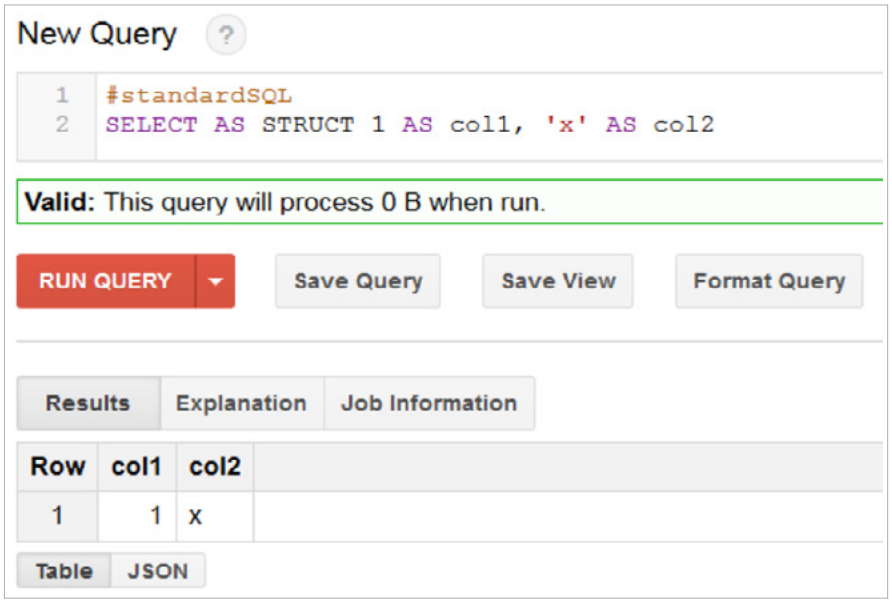
- RECORD (legacy SQL) / STRUCT (standard SQL)
- REPEATED (legacy SQL) / ARRAY (standard SQL)

The key principle here is to throw out the 3rd normal form modelling guideline of storing every instance of an entity in just one place. Instead the focus is on the fact that, while denormalized models require more storage, they can nevertheless respond faster because there is no need to join (and consequently shuffle) data while the queries are parallelized.

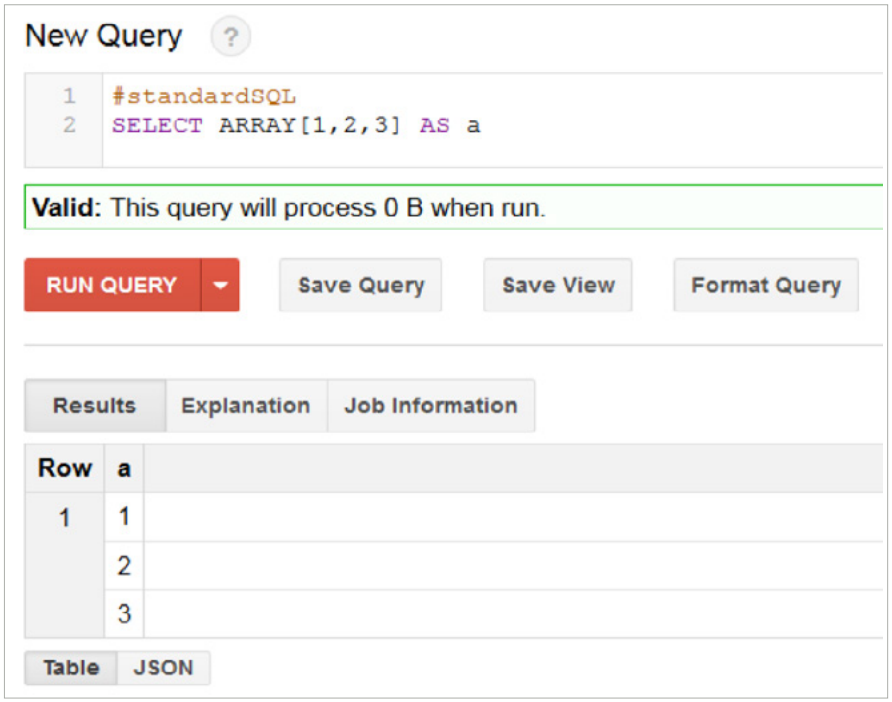
Incorporating nested and repeated columns in your data model enables you to have a denormalized structure which still includes relationships. A REPEATED RECORD is actually an inline 1:many relationship, which can be resolved by querying only one physical table.

The earlier **Materialize Common Joins** section showed how to materialize a result set at different granularity during a join. More generally, BigQuery offers methods to construct records and arrays using SQL.

For example, creating a STRUCT:



Creating an ARRAY:



Denormalization

- ✗ Star or snowflake schema, requiring joins
- ✗ Requires more storage
- ✓ Denormalized data mode with nested and repeated fields, easier to parallelize
- ✓ RECORD (legacy SQL) / STRUCT (standard SQL) or REPEATED (legacy SQL) / ARRAY (standard SQL)

- ★ Cost
- ★ Performance

CHAPTER 4

Cost Estimation and Management

CHAPTER 4

Cost Estimation and Management

This section describes additional tips and techniques to monitor and estimate costs while using Google BigQuery.

Monitoring estimated costs can help you avoid accidentally running a needlessly expensive query and being overcharged as a result.

There are various ways to apply custom quotas to the account, which apply a fixed spend limit while sacrificing availability.

4.1 Data Storage Pricing

Data exports and loads (excluding streaming inserts) are free, but there is a cost associated with simply keeping data inside Google BigQuery.

For ordinary tables in native BigQuery storage, data is charged by volume at $0.02 per GB, per month. This is prorated per MB, per second.

If you don't alter the data at all for 90 days, the cost will automatically half to $0.01 per GB, per month. Queries are permitted, but no inserts, updates or deletes. Exactly the same thing applies per partition for partitioned tables.

You can check the time of last modification using the following query:

NOTE

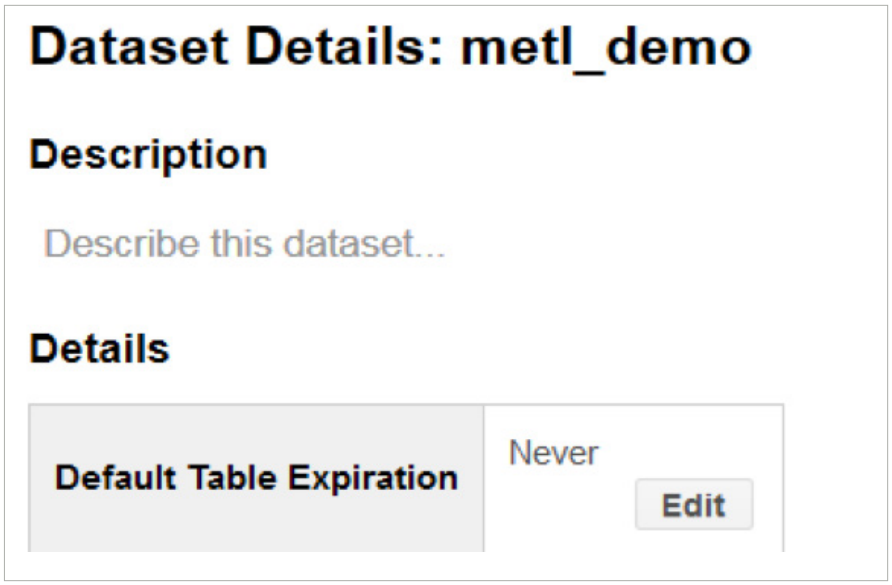
This is purely a cost saving and does not affect the performance of the table in any way.

```
#standardSQL
SELECT table_id,
    TIMESTAMP_MILLIS(last_modified_time) AS mod_time
FROM `bigquery-public-data.fec.__TABLES__`
```

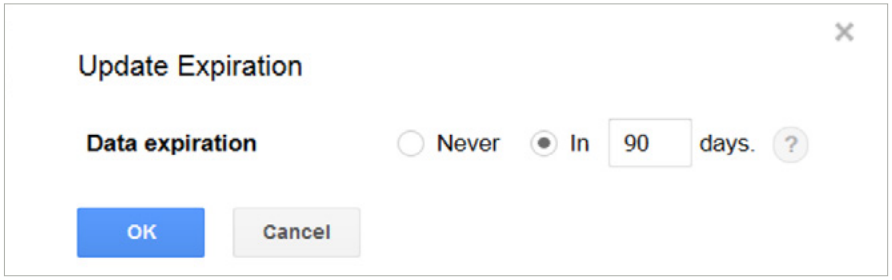
EXPIRATION

There is a small cost associated with storing data in Google BigQuery. You can help minimize this cost by switching on the option to automatically drop tables or partitions.

For ordinary tables, the default expiration is an optional property of the dataset:



Whenever you create a new table the dataset's default expiration is applied. Of course you can change this default if you wish. Similarly for partitioned tables, every individual partition can be given a partition expiration. You can set the expiration from the web user interface.



Expiration

- Set expirations for tables and partitions.
- Cost

4.2 Partitioning

Partitioning your data can increase your query efficiency and potentially reduce the cost of your queries. Partitioning your data essentially adds logical breakpoints, meaning you can query a subset of the whole database by querying a particular partition. Google offers two methods for partitioning in BigQuery:

- "Sharding" - where you manually manage multiple identical tables, each of which contains a subset of the data. It's best to enforce a naming standard which relates the table name to the contents

- Date partitioning - where Google automatically maintains the contents in separate "segments" according to a pseudocolumn called _PARTITIONTIME

If you have a large number of physically sharded tables, Google BigQuery has a non-trivial task of maintaining metadata (column names, permissions, etc.) for every table. This can impact query performance since all that metadata has to be scanned at runtime. While both methods are available, Google recommends using date partitioning where possible since it's more automated.

4.3 Monitor Query Costs

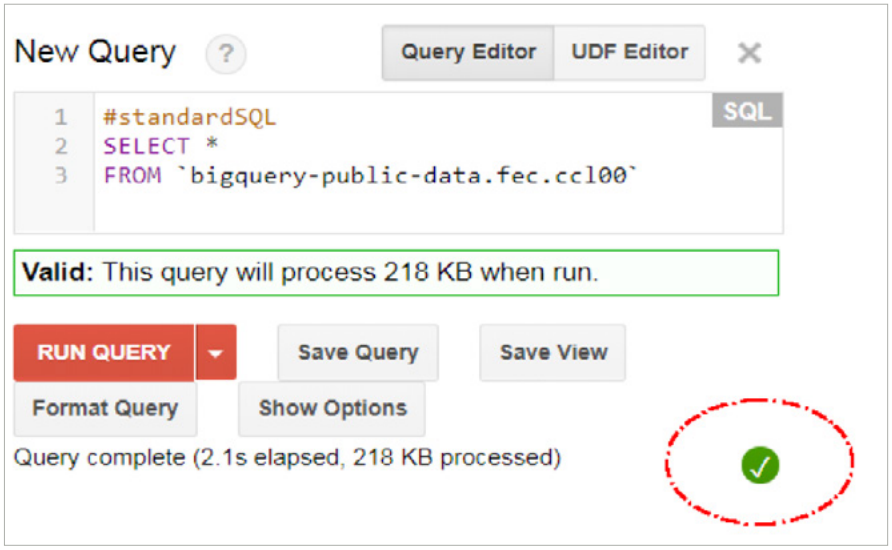
While we have discussed at length a number of ways you can help minimize your query costs, you won't know exactly how much these optimized queries will cost. Google BigQuery comes with some tools to help you forecast query costs and place limits to control them. This section provides you with a high-level overview of these functionalities.

ESTIMATE QUERY COSTS

To monitor potential costs, the BigQuery web console has a query validator which can be launched before a query is run:

NOTE

Press the green tick icon to open the validator. This will provide an estimate of how many bytes of data will be processed in running the query.



It is good practice to check the query validator before running any queries to see if any of the optimizations discussed in this document can be applied to reduce the volume of data processed and therefore reduce the costs.



Using the above example, the number of bytes processed has more than halved by only selecting the columns required. See **Avoid SELECT*** above.

You can easily query the __TABLES__ metadata to estimate the cost of a full query on every table in a dataset, at the flat rate of $5 per TB:

```
#standardSQL
SELECT table_id, size_bytes,
        ROUND(500 * size_bytes / (1024*1024*1024*1024), 2) AS cents_
to_query
FROM `bigquery-public-data.fec.__TABLES__`
```

PRICING CALCULATOR

The **Google Cloud Platform Pricing Calculator** can be used to estimate the monthly costs. It includes sections on storage of data, streaming inserts and query pricing.

EXPORT BILLING DATA

All of your billing data for your GCP project is available to export to BigQuery. A subset of this data will be your BigQuery billing data. You may wish to store all of this in BigQuery and use Data Studio to build charts and visualise the data. This will allow you to easily monitor costs. Further details are available **here**.

CUSTOM QUOTAS

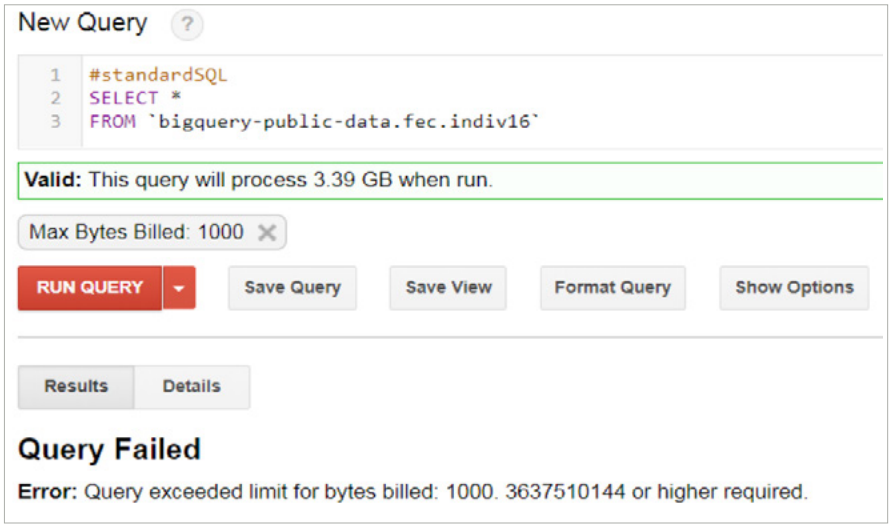
To control costs, BigQuery allows administrators to set limits on the amount of data which can be returned from a query per day. This can be done at either a project level or at a user level.

This quota can be used to control the cost of BigQuery queries. If a user tries to exceed either a project or user level quota they will simply receive an error message and no data will be returned. Quotas can be set using an **online request form**.

MAXIMUM BYTES BILLED LIMIT

The final option to control the costs, beyond the Custom Quotas discussed above, is to set the Maximum Bytes Billed Limit. This is an option applied at a query level to set a maximum on the number of bytes billed. If the query returns

more bytes than this, it will fail and return no data. This is set in the user interface with an example of 1000 bytes below:



This is a great way to prevent accidentally running unexpectedly large and costly queries.

Conclusion

We hope you enjoyed this eBook and that you have found some helpful hints and tips on how to make the most of Google BigQuery database. Implementing the best practices and optimizations described in this eBook should greatly enhance big data analytics performance and reduce your BigQuery costs. This way you can spend less resource on overhead and focus on what's really important - answering your organization's most pressing business questions.

About Matillion

Matillion is fundamentally changing data integration enabling our customers to innovate at the speed of business, with cloud-native data integration technology that solves individuals' and enterprises' top business challenges. Matillion ETL for BigQuery can be used with your modern BigQuery database to make data loading and transformation fast, easy, and affordable.

For more information about Matillion ETL for BigQuery visit **www.matillion.com/etl-for-bigquery/**.

By Ian Funnell, Head Solution Architect, Laura Malins, Solution Architect, and Dayna Shoemaker, Knowledge Engineer

Glossary

"Broadcast" join	A join method in which a smaller table is physically copied onto all compute slots where segments of a larger table are already located.
Common Table Expression (CTE)	The **WITH clause** in a SELECT statement. It acts as a named temporary table.
Copying	Copying See Data Loading.
Loading	The process of loading data involves taking a copy of the source data and physically loading it into Google BigQuery for storage.
Data Manipulation Language (DML)	This means issuing SQL which changes data
Federated Data Source	An external data source that you can query directly even though the data isn't stored in BigQuery. This eBook references Federated Data Sources as 'linking'.
Historical load	A once-only operation in which the bulk of historical data is loaded into a data warehouse representation
Incremental load	A repeated operation in which new data (e.g. from yesterday) is appended into its data warehouse representation
Linking	See Federated Data Source.
Partition Pruning	Deliberate restriction of a query to avoid scanning storage areas that could not possibly contain relevant data. This can greatly improve speed and reduces cost.
Sharding	A form of manually-managed partitioning, involving the creation of multiple similarly-named tables, each containing a subset of the data
Sharding	The replication of data between slots, for example which happens during a broadcast join. Data shuffling is an expensive but necessary part of servicing a query in any MPP database.
Skew	Undesirable situation in which the work performed by compute nodes is unbalanced. This leads to some finishing earlier than others, resulting in slower overall performance
Slot	A unit of capacity, corresponding to an MPP compute node. In BigQuery slots are managed automatically on your behalf.
UDF (User-Defined Functions)	In legacy SQL a UDF is a JavaScript function, provided by the database user, which accepts one record as input and returns any number of records as output. In standard SQL a UDF is a temporary JavaScript function which returns a BigQuery datatype (including ARRAY and STRUCT) in response to input parameters.
V8	Google's JavaScript execution engine

Fast.
Easy.
Affordable.

Google BigQuery is modern, infinite, and powerful;
our native ETL tool unlocks its full potential.

- **Push-down ELT**
- **Consumption-based pricing model**
- **Launches in ive minutes**
- **No commitments or upfront costs**



Try out Matillion ETL for Google BigQuery with a Free Test Drive, available through **Google Cloud Launcher**

MATILLION

Google Cloud Platform

