

O'REILLY®

Google BigQuery

The Definitive Guide

Data Warehousing, Analytics, and Machine
Learning at Scale



Early
Release

RAW &
UNEDITED

Valliappa Lakshmanan
& Jordan Tigani

1. 1. What is Google BigQuery?
 - a. Data processing architectures
 - i. Relational Database Management System (RDBMS)
 - ii. MapReduce Framework
 - iii. BigQuery: A serverless, distributed SQL engine
 - b. Working with BigQuery
 - i. Deriving insights across datasets
 - ii. ETL, EL, and ELT
 - iii. Powerful Analytics
 - iv. Simplicity
 - c. How BigQuery came about
 - d. What makes BigQuery possible?
 - i. Separation of compute and storage
 - ii. Storage and Networking Infrastructure
 - iii. Managed storage
 - iv. Integration with GCP
 - v. Security and compliance
2. 2. Query Essentials
 - a. Simple queries
 - i. Retrieving rows with SELECT
 - ii. Aliasing column names with AS

- iii. Filtering with WHERE
- iv. Subqueries with WITH
- v. Sorting with ORDER BY

b. Aggregates

- i. Computing aggregates with GROUP BY
- ii. Counting records with COUNT
- iii. Filtering grouped items with HAVING
- iv. Finding unique values with DISTINCT

c. A brief primer on arrays and structs

- i. Creating ARRAYS using ARRAY_ AGG
- ii. ARRAY of STRUCT
- iii. TUPLE
- iv. Working with arrays
- v. UNNEST an array

d. Joining tables

- i. The JOIN, explained
- ii. Inner join
- iii. Cross Join
- iv. Outer join

e. Saving and sharing

- i. Query history and caching
- ii. Saved queries
- iii. Views vs. shared queries

f. Chapter Summary

3. 3. Data Types, Functions and Operators

- a. Numeric types and functions
- b. Working with BOOL
- c. String functions
- d. Working with TIMESTAMP
- e. Working with GIS functions
- f. Summary of data types in BigQuery

4. 4. Loading Data into BigQuery

- a. The basics
 - i. Loading from a local source
 - ii. Specifying a schema
 - iii. Copying into a new table
 - iv. Data management (DDL and DML)
 - v. Loading data efficiently
- b. Federated queries and external data sources
 - i. How to use federated queries
 - ii. When to use federated queries and external data sources
 - iii. Interactive exploration and querying of data in Google Sheets
 - iv. SQL queries on data in Cloud Bigtable
- c. Transfers and exports
 - i. Data Transfer Service

- ii. Export Stackdriver Logs
 - iii. Using Cloud Dataflow to read/write from BigQuery
 - d. Migrating on-premises data
 - i. Data migration methods
 - e. Summary
- 5. 5. Developing with BigQuery
 - a. Developing programmatically
 - i. Accessing BigQuery via REST API
 - ii. Google Cloud Client library
 - b. Accessing BigQuery from data science tools
 - i. Notebooks on Google Cloud
 - ii. pandas
 - iii. Working with BigQuery from R
 - iv. Dataflow
 - v. JDBC/ODBC drivers
 - vi. Incorporating BigQuery data into Google Slides (in GSuite)
 - c. Bash Scripting with BigQuery
 - i. Creating datasets and tables
 - ii. Executing queries
 - iii. BigQuery objects
 - d. Summary

Google BigQuery: The Definitive Guide

Data Warehousing, Analytics, and Machine Learning
at Scale

Valliappa Lakshmanan and Jordan Tigani



Google BigQuery: The Definitive Guide

by Valliappa Lakshmanan and Jordan Tigani

Copyright © 2020 Valliappa Lakshmanan and Jordan Tigani. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Jonathan Hassell

Production Editor: Kristen Brown

Copyeditor:

Proofreader:

Indexer:

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

January 2020: First Edition

Revision History for the Early Release

- 2019-03-04: First Release
- 2019-04-10: Second Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492044468> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc.

Google BigQuery: The Definitive Guide, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-04446-8

[FILL IN]

Chapter 1. What is Google BigQuery?

Data processing architectures

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 minutes. You get this performance without having to manage any infrastructure and without having to create or rebuild indexes.

BigQuery has legions of fans. Paul Lamere, a Spotify engineer, was thrilled that he could finally talk about how his team uses BigQuery to quickly analyze large datasets: “Google’s BigQuery is *da bomb*,” he tweeted. “I start with 2.2Billion ‘things’ and compute/summarize down to 20K in < 1 min.”¹ The scale and speed -- terabytes in seconds, petabytes in minutes -- are just two notable features of BigQuery. What is more transformative is not having to manage infrastructure because the simplicity inherent in serverless, ad-hoc querying can open up new ways of working.

Companies are increasingly embracing data-driven decision making and fostering an open culture where the data is not siloed within departments. BigQuery, by providing the technological means to enact a cultural shift towards agility and openness, plays a big part in increasing the pace of innovation.

For Alpega Group, a global logistics software company, the increased innovation and agility offered by BigQuery was key. They went from a situation where real-time analytics was impossible to being able to provide fast, customer-facing analytics in near real-time. Because they do not have to maintain clusters and infrastructure, their small tech team is now free to work on software development and data capabilities. “That was a real eye opener for us.”, says their lead architect Aart Verbeke, “In a conventional environment we would need to install, set up, deploy and host every individual building block. Here we simply connect to a surface and use it as required.”²

Imagine that you run a chain of stores that rents out equipment. You charge customers based on the length of the rental, so your records include the following details that will allow you to properly invoice the customer:

Where the item was rented

When it was rented

Where the item was returned

When it was returned

Perhaps you record the transaction in a database every time a customer returns an item.³

From this dataset, you would like to find out how many “one-way” rentals occurred every month in the past ten years. Perhaps you are thinking of imposing a surcharge for returning the item at a different store and you would like to find out what fraction of rentals would be affected. Let’s posit that wanting to know the answer to such questions is a frequent

occurrence -- it is important for you to be able to answer such ad-hoc questions because you tend to make data-driven decisions.

What kind of system architecture could you use? Let's run through some of the options.

Relational Database Management System (RDBMS)

When recording the transactions, you are probably recording them in a relational, online transaction processing (OLTP) database such as MySQL or PostgreSQL. One of the key benefits of such databases is that they support querying using Structured Query Language (SQL) -- your staff doesn't need to use high-level languages like Java or Python to answer questions that arise. Instead, it is possible to write a query that can be submitted to the database server:

```
SELECT
EXTRACT(YEAR FROM starttime) AS year,
EXTRACT(MONTH FROM starttime) AS month,
COUNT(starttime) AS number_one_way
FROM
mydb.return_transactions
WHERE
start_station_name != end_station_name
GROUP BY year, month
ORDER BY year ASC, month ASC
```

Ignore the details of the syntax for now -- we will cover SQL queries later in this book. Instead, let's focus on what this tells us about the benefits and drawbacks of an OLTP database.

First, notice that SQL goes beyond just being able to get the raw data in database columns -- the query above parses the timestamp and extracts the year and month from it. It also does aggregation (counting the number of rows), some filtering (finding rentals where the starting and ending locations are different), grouping (by year and month), and sorting. An important benefit of SQL is the ability to specify what we want and let the database software figure out an optimal way to execute the query.

Unfortunately, queries like the one above are quite inefficient for an OLTP database to carry out. OLTP databases are tuned towards data consistency -- the point is that you can read from the database even while data is being simultaneously written to it. This is achieved through careful locking to maintain data integrity. For the filtering on station_name to be efficient, you would have to create an index on the station name column. If the station name is indexed, then (and only then) does the database do special things to the storage to optimize searchability -- this is a tradeoff, slowing writing down a bit to improve the speed of reading. If the station name is not indexed, filtering on it will be quite slow. Even if the station name is an index, this particular query will be quite slow because of all the aggregating, grouping and ordering. OLTP databases are not built for this sort of ad-hoc⁴ query that requires traversal through the entire dataset.

MapReduce Framework

Because OLTP databases are a poor fit for ad-hoc queries and queries that require traversal of the entire dataset, special-purpose analyses that require such traversal might be coded in high-level languages like Java or Python. In 2003, Jeff Dean and Sanjay Ghemawat observed that they and their colleagues at Google were implementing hundreds of these special-purpose computations to process large amounts of raw data. Reacting to

this complexity, they designed an abstraction that allowed these computations to be expressed in terms of two steps -- a map function that processed a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merged all intermediate values associated with the same intermediate key.⁵ This paradigm, known as MapReduce, became hugely influential and led to the development of Apache Hadoop.

While the Hadoop ecosystem started out with a library that was primarily built in Java, custom analysis on Hadoop clusters is now typically carried out using Apache Spark.⁶ Spark programs can be written in Python or Scala, but among the capabilities of Spark is the ability to execute ad-hoc SQL queries on distributed datasets.

So, to find out the number of one-way rentals, you could set up the following data pipeline:

Periodically, export transactions to comma-separated text files in the Hadoop Distributed File System (HDFS)

For ad-hoc analysis, write a Spark program that:

Loads up the data from the text files into a “dataframe”

Executes a SQL query, similar to the query in the previous section, except that the table name is replaced by the name of the dataframe

Exports the result set back to a text file

Run the Spark program on a Hadoop cluster

While seemingly straightforward, this architecture imposes a couple of

hidden costs. Saving the data in HDFS requires that the cluster be large enough. One underappreciated fact about the MapReduce architecture is that the MapReduce architecture usually requires that the compute nodes access data that is local to them. The HDFS has to, therefore, be sharded across the compute nodes of the cluster. With data sizes and analysis needs both increasing dramatically but independently, it is often the case that clusters are under- or over-provisioned.⁷ Thus, the need to execute Spark programs on a Hadoop cluster means that your organization will have to become experts in managing, monitoring, and provisioning Hadoop clusters. This may not be your core business.

BigQuery: A serverless, distributed SQL engine

What if you could run SQL queries as in an RDBMS system, obtain efficient and distributed traversal through the entire dataset efficiently as in MapReduce, and not have to manage infrastructure? That's the third option, and it is what makes BigQuery so magical. BigQuery is serverless and you can run queries without having to manage infrastructure. It enables you to carry out analyses that process aggregations over the entire dataset in seconds to minutes.

Don't take our word for it, though. Try it out now. Navigate to <https://console.cloud.google.com/bigquery> (logging into Google Cloud Platform and selecting your project if necessary), copy-paste the following query in the window,⁸ and click on the button marked "Run query":

```
SELECT  
EXTRACT(YEAR FROM starttime) AS year,  
EXTRACT(MONTH FROM starttime) AS month,  
COUNT(starttime) AS number_one_way
```

```
FROM
`bigquery-public-data.new_york_citibike.citibike_trips`
WHERE
start_station_name != end_station_name
GROUP BY year, month
ORDER BY year ASC, month ASC
```

When we ran it, the BigQuery user interface reported that the query involved processing 2.5 GB and gave us the result in about 2.7 seconds:

Query editor

 HIDE EDITOR

```
1 SELECT
2   EXTRACT(YEAR FROM starttime) AS year,
3   EXTRACT(MONTH FROM starttime) AS month,
4   COUNT(starttime) AS number_one_way
5 FROM
6   `bigquery-public-data.new_york_citibike.citibike_trips`
7 WHERE
8   start_station_name != end_station_name
9 GROUP BY year, month
10 ORDER BY year ASC, month ASC
```

 Run query ▾

 Save query

 Save view

 More ▾

This query will process 2.51 GB when run.



Query results

 **SAVE RESULTS** ▾



Query complete (2.704 sec elapsed, 2.51 GB processed)

Job information **Results** JSON Execution details

Row	year	month	number_one_way
1	2013	7	815324
2	2013	8	970474
3	2013	9	1007799

Figure 1-1. Running a query to compute the number of one-way rentals in the BigQuery web user interface.

The equipment being rented out is bicycles, and so, the query above totals up one-way bicycle rentals in New York month-by-month over the extent of the dataset. The dataset itself is a public dataset (meaning that anyone can query the data held in it) released by New York City as part of their Open City initiative. From this query, we learn that in July 2013, there

were 815,324 one-way Citibike rentals in New York City.

Note a few things about this. One was that you were able to run a query against a dataset that was already present in BigQuery. All that the owner of the project hosting the data had to do was to give you⁹ “view” access to this dataset. You didn’t have to start up a cluster, or login to one. Instead, you just submitted a query to the service, and got back your results. The query itself was written in SQL:2011, making the syntax familiar to data analysts everywhere. Although we demonstrated on gigabytes of data, the service scales well even when it has to do aggregations on terabytes to petabytes of data. This scalability is possible because the service distributes the query processing among thousands of workers almost instantaneously.

Working with BigQuery

BigQuery is a data warehouse, implying a degree of centralization and ubiquity. The query we demonstrated in the previous section was applied to a single dataset. However, the benefits of BigQuery become even more apparent when we do joins of datasets from completely different sources or when we query against data that is stored outside BigQuery.

Deriving insights across datasets

The bicycle rental data comes from New York City. How about joining it against weather data from the US National Oceanic and Atmospheric Administration (NOAA), to find out whether there are fewer bicycle rentals on rainy days?¹⁰

-- Are there fewer bicycle rentals on rainy days?

```
WITH bicycle_rentals AS (
SELECT
COUNT(starttime) as num_trips,
EXTRACT(DATE from starttime) as trip_date
FROM `bigquery-public-data.new_york_citibike.citibike_trips`
GROUP BY trip_date
),
```

```
rainy_days AS
(
SELECT
date,
(MAX(prcp) > 5) AS rainy
FROM (
SELECT
wx.date AS date,
IF (wx.element = 'PRCP', wx.value/10, NULL) AS prcp
FROM
```

```
`bigquery-public-data.ghcn_d.ghcnd_2016` AS wx
```

```
WHERE
```

```
wx.id = 'USW00094728'
```

```
)
```

```
GROUP BY
```

```
date
```

```
)
```

```
SELECT
```

```
ROUND(AVG(bk.num_trips)) AS num_trips,
```

```
wx.rainy
```

```
FROM bicycle_rentals AS bk
```

```
JOIN rainy_days AS wx
ON wx.date = bk.trip_date
GROUP BY wx.rainy
```

Ignore the specific syntax of the query. Just notice that, in the bolded lines, we are joining the bicycle rental dataset with a weather dataset that comes from a completely different source. Running the query satisfyingly yields that, yes, New Yorkers are wimps -- they ride the bicycle nearly 20% fewer times when it rains:¹¹

```
Row num_trips rainy
1 39107.0 false
2 32052.0 true
```

What does being able to share and query across datasets mean in an enterprise context? Different parts of your company can store their datasets in BigQuery and quite easily share the data with other parts of the company and even with partner organizations. The serverless nature of BigQuery provides the technological means to break down departmental silos and streamline collaboration.

ETL, EL, and ELT

The traditional way to work with data warehouses is to start with an Extract-Transform-Load (ETL) process, wherein raw data is extracted from its source location, transformed, and then loaded into the data warehouse. Indeed, BigQuery has a native, highly efficient, columnar storage format¹² that makes ETL an attractive methodology. The data pipeline, typically written either in Apache Beam or Apache Spark, extracts the necessary bits from the raw data (either streaming data or batch files), transforms to do any necessary cleanup or aggregation, and

then loads it into BigQuery:

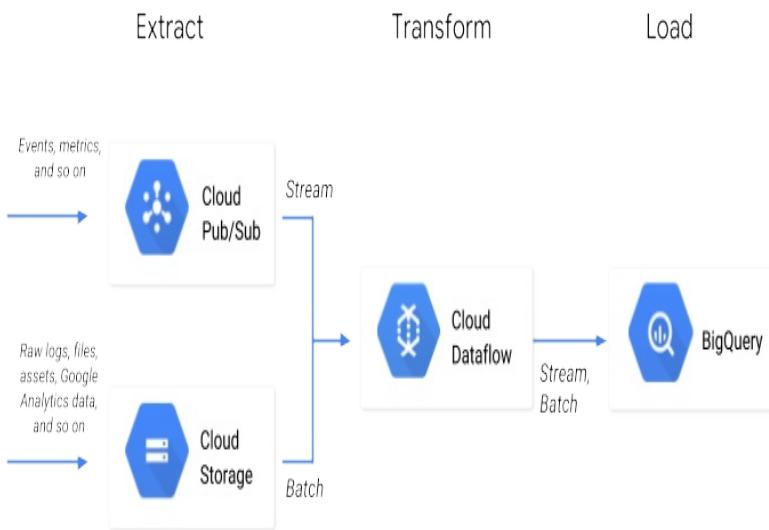


Figure 1-2. The reference architecture for ETL into BigQuery uses Apache Beam pipelines executed on Cloud Dataflow and can handle both streaming and batch data using the same code.

While building an ETL pipeline in Apache Beam or Apache Spark tends to be quite common, it is possible to implement an ETL pipeline purely

within BigQuery. 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. You can take advantage of federated queries to extract the data using SQL queries against data stored in Google Cloud Storage, transform the data within those SQL queries, and then materialize the results into a BigQuery native table.

If transformation is not necessary, BigQuery can directly ingest standard formats like CSV, JSON, or Avro into its native storage -- an EL workflow, if you will. The reason to end up with the data loaded into the data warehouse is that having the data in native storage provides the most efficient querying performance.

We strongly recommend that you design for an EL workflow if possible, and drop to an ETL workflow only if transformations are needed. If possible, do those transformations in SQL, and keep the entire ETL pipeline within BigQuery. If the transforms will be difficult to implement purely in SQL or if the pipeline has to stream data into BigQuery as it arrives, then build an Apache Beam pipeline and have it executed in a serverless fashion using Cloud Dataflow. Another advantage of implementing ETL pipelines in Beam/Dataflow is that, because this is programmatic code, such pipelines integrate better with continuous integration and unit testing systems.

Besides the ETL and EL workflows, BigQuery makes it possible to do an ELT workflow. The idea is to extract and load the raw data as-is and rely on BigQuery views to transform the data on the fly. An ELT workflow is particularly useful if the schema of the raw data is under flux. For example, you might still be carrying out exploratory work to determine

whether a particular timestamp needs to be corrected for the local timezone. The ELT workflow is useful in prototyping and allows an organization to start deriving insights from the data without having to make potentially irreversible decisions too early.

The alphabet soup can be confusing, so here's a quick summary.

Workflow	Architecture	When you'd do it
Extract - Load (EL)	<p>Extract data from files on Google Cloud Storage</p> <p>Load it into BigQuery's native storage</p> <p>You can trigger this from Cloud Composer, Cloud Functions, or scheduled queries.</p>	<p>Batch load of historical data</p> <p>Scheduled periodic loads of log files (e.g. once a day)</p>
Extract - Transform - Load (ETL)	<p>Extract data from Pub/Sub, Google Cloud Storage, Cloud Spanner, Cloud SQL, etc.</p> <p>Transform the data using Cloud Dataflow</p> <p>Have Dataflow pipeline write to BigQuery</p>	<p>When the raw data needs to be quality-controlled, transformed, or enriched before being loaded into BigQuery.</p> <p>When the data loading has to happen continuously, i.e. if the use case requires streaming.</p> <p>When you want to integrate with continuous integration / continuous delivery (CI/CD) systems and perform unit testing on all components.</p>
Extract - Load - Transform (ELT)	<p>Extract data from files in Google Cloud Storage</p> <p>Store data in close-to-raw format in BigQuery</p> <p>Transform the data on the fly using BigQuery views.</p>	<p>Experimental datasets where you are not yet sure what kinds of transformations are needed to make the data useable.</p> <p>Any production dataset where the transformation can be expressed in SQL</p>

The workflows in the table above are in the order that we usually recommend.

Powerful Analytics

The benefits of a warehouse derive from the kinds of analyses you can do with the data held in it. The primary way you interact with BigQuery is via SQL, and because BigQuery is a SQL engine, you can use a wide variety of Business Intelligence (BI) tools, such as Tableau, Looker, and Google Data Studio to create impactful analyses, visualizations, and reports on data held in BigQuery. Clicking on the “Explore in Data Studio” button in the BigQuery web user interface, for example, we can quickly create a visualization of how our one-way bike rentals vary by month:

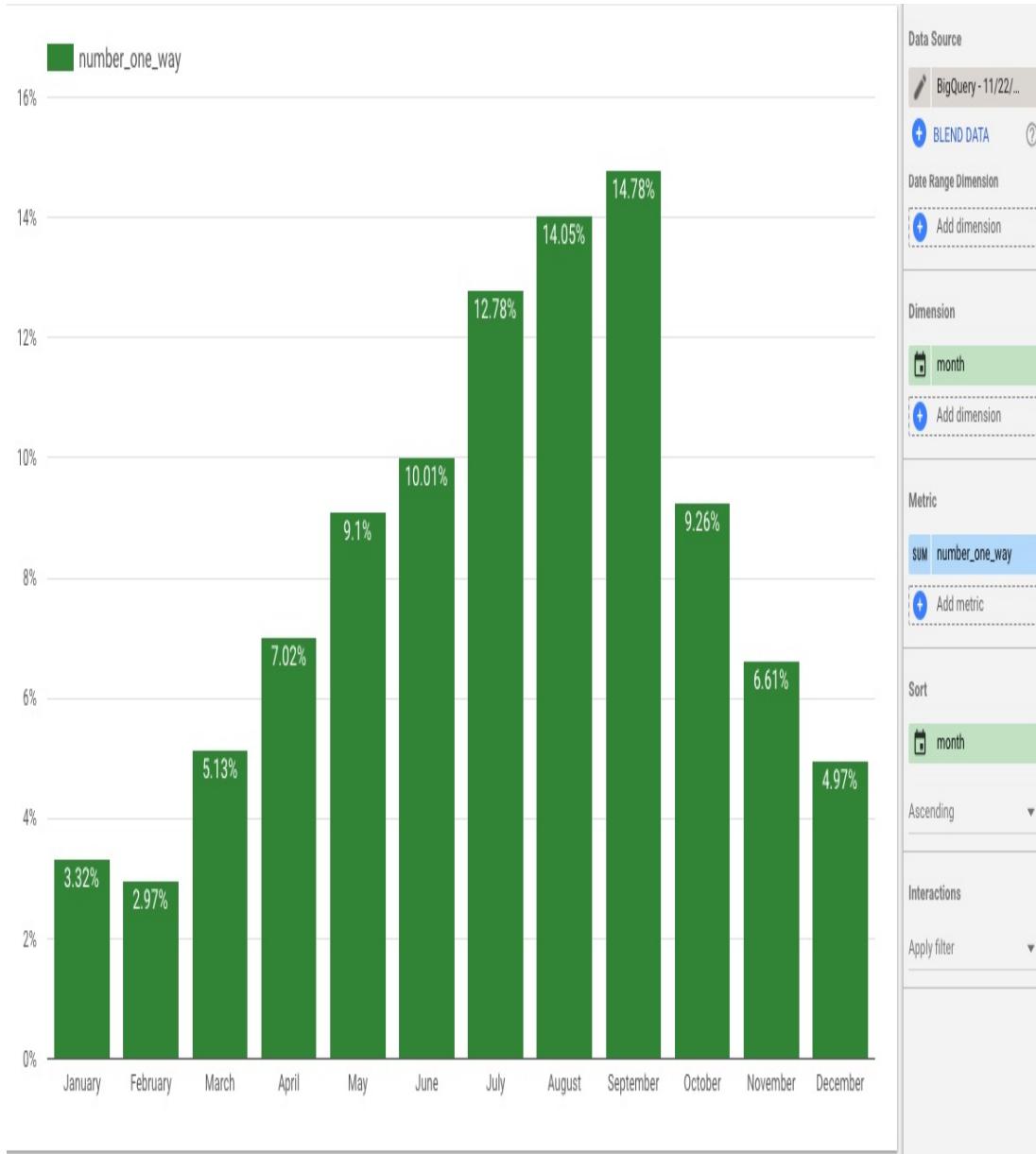


Figure 1-3. Visualization in Data Studio of how one-way rentals vary by month. Nearly 15% of all one-way bicycle rentals in New York happen in September.

BigQuery provides full-featured support for SQL:2011, including support for arrays and complex joins. The support for arrays in particular makes it possible to store hierarchical data (such as JSON records) in BigQuery without having to flatten the nested and repeated fields. Besides the support for SQL:2011, BigQuery has a few extensions that make it useful beyond the core set of data warehouse use cases. One of these extensions

is support for a wide range of spatial functions that enable location-aware queries including the ability to join two tables based on distance or overlap criteria.¹³ BigQuery is, therefore, a powerful engine to carry out descriptive analytics.

Another BigQuery extension to standard SQL supports creating machine learning models and carrying out batch predictions. We will cover the machine learning capability of BigQuery in detail in Chapter XX, but the gist is that you can train a BigQuery model and make predictions without ever having to export data out of BigQuery. The security and data locality advantages of being able to do this are enormous. BigQuery is, therefore, a data warehouse that supports not just descriptive analytics, but also predictive analytics.

A warehouse also implies being able to store different types of data. Indeed, BigQuery can store data of many types -- numeric and textual columns, for sure, but also geospatial data and hierarchical data. While you can store flattened data in BigQuery, you don't need to -- schemas can be rich and quite sophisticated. The combination of location-aware queries, hierarchical data, and machine learning combine to make BigQuery a powerful solution that goes beyond conventional data warehousing and business intelligence.

BigQuery supports the ingest both of batch data and of streaming data. You can stream data directly into BigQuery via a REST API. Often, users who want to transform the data, for example by adding time-windowed computations, use Apache Beam pipelines executed by the Cloud Dataflow service. Even as the data is streaming into BigQuery, you can query it. Having common querying infrastructure for both historical (batch) data and current (streaming) data is extremely powerful and

simplifies many workflows.

Simplicity

Part of the design consideration behind BigQuery is to encourage users to focus on insights rather than on infrastructure. When you ingest data into BigQuery, there is no need to think about different types of storage, or their relative speed and cost tradeoffs -- the storage is fully managed. At the time of writing, the cost of storage automatically drops to lower levels if a table is not updated for a long enough period of time.¹⁴

We have already talked about how indexing is not necessary -- your SQL queries can filter on any column in the dataset and BigQuery will take care of the necessary query planning and optimization. For the most part, we recommend that you write queries to be clear and readable and rely on BigQuery to choose a good optimization strategy. In this book, we will talk about performance tuning, but performance tuning in BigQuery consists mainly of clear thinking and the appropriate choice of SQL functions. You will not have to do database administration tasks like replication, defragmentation or disaster recovery -- the BigQuery service takes care of all that for you.

Queries are automatically scaled to thousands of machines and executed in parallel. You don't have to do anything special to enable this massive parallelization. The machines themselves are transparently provisioned to handle the different stages of your job -- you don't have to set up those machines in any way.

Not having to set up infrastructure leads to less hassle in terms of security. Data in BigQuery is automatically encrypted, both at rest and in transit. BigQuery takes care of the security considerations behind supporting

multi-tenant queries and providing isolation between jobs. Your datasets can be shared using Google Cloud Identity and Access Management (IAM), and it is possible to organize the datasets (and the tables and views within them) to meet different security needs, whether you need openness or auditability or confidentiality.

In other systems, provisioning infrastructure for reliability, elasticity, security, and performance often takes a lot of time to get right. Given that these database administration tasks are minimized with BigQuery, organizations using BigQuery find that it frees their analysts' time to focus on deriving insights from their data.

How BigQuery came about

In late 2010, the site director of the Google Seattle office pulled several engineers (one of whom is an author of this book) off their projects and gave them a mission: to build a data marketplace. We tried to figure out the best way to come up with a viable marketplace. The chief issue was data sizes because we didn't want to provide just a download link. A data marketplace is infeasible if people have to download terabytes of data in order to work with it. How would you build a data marketplace that didn't require users to start by downloading the datasets to their own machines?

Enter a principle popularized by Jim Gray, the database pioneer¹⁵. When you have “big data”, Jim Gray said, you want to move the computation to the data, rather than move the data to the computation.”

The other key issue is that as the datasets get larger, it is no longer possible to just

FTP or grep them. A petabyte of data is very hard to FTP! So at some point, you

need indices and you need parallel data access, and this is where databases can

help you. For data analysis, one possibility is to move the data to you, but the other

possibility is to move your query to the data. You can either move your questions

or the data. Often it turns out to be more efficient to move the questions than to

move the data.

^{”16}. In the case of the data marketplace that we were building, users would not have to download the datasets to their own machines if we made it possible for them to bring their computations to the data. We would not have to provide a download link since users could work on their data without having to move it around.¹⁷

We, the Googlers who were tasked with building a data marketplace, made the decision to defer that project and focus on building a compute engine and storage system in the cloud. After ensuring that users could do something with the data, we would go back and add data marketplace features.

What language should users write their computation in, when bringing computation to the data on the cloud? We chose SQL because of three key

characteristics. First, SQL is a versatile language that allows a large range of people, not just developers, to ask questions and solve problems with their data. This ease of use was extremely important to us. Secondly, SQL is “relationally complete”, meaning that any computation over the data can be done using SQL. SQL is not just easy and approachable. It is also very powerful. Finally, and quite important for a choice of a cloud computation language, SQL is not “Turing complete” in a key way: it always terminates.¹⁸ Because it always terminates, it is ok to host SQL computation without worrying that someone will write an infinite loop and monopolize all the compute power in a data center.

Next, we had to choose a SQL engine. Google had a number of internal SQL engines that could operate over data, including some that were very popular. The most advanced engine was called Dremel; it was used heavily at Google, and could process terabytes-worth of logs in seconds. Dremel was quickly winning people over from building custom MapReduce pipelines to ask questions of their data.

Dremel had been created in 2007 by an engineer, Andrey Gubarev, who was tired of waiting for MapReduces to finish. Column stores were becoming popular in the academic literature, and he quickly came up with a column storage format that could handle the Protocol Buffers that are ubiquitous throughout Google.

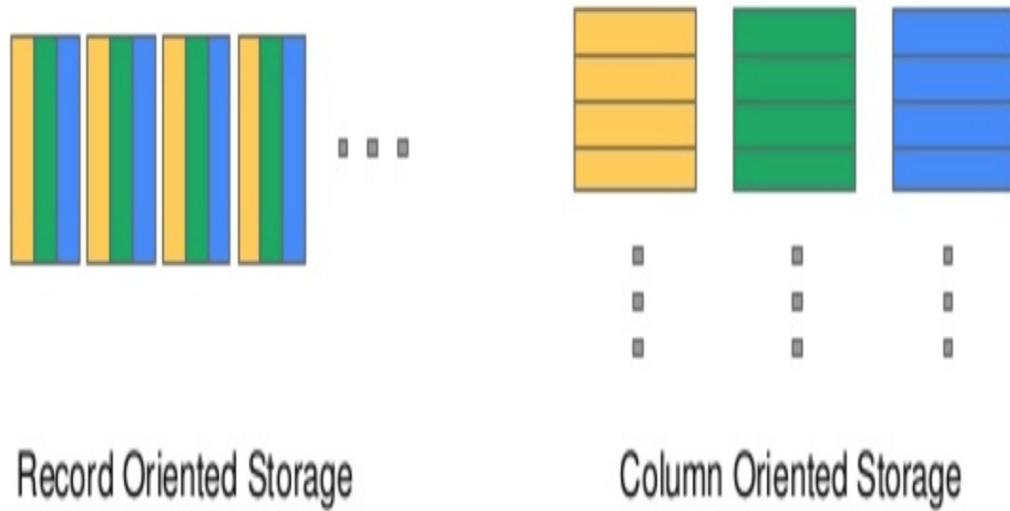


Figure 1-4. Column stores can reduce the amount of being read by queries that process all rows, but not all columns.

While column stores are great in general for analytics, they are particularly useful for logs analysis at Google because many teams operate over a type of Protocol Buffer that has hundreds of thousands of columns. If Andrey had used a typical record-oriented store, users would have had to read the files row-by-row, thus reading in a huge amount of data in the form of fields that they were going to discard anyway. By storing the data column-by-column, Andrey made it so that if a user needed a few of the thousands of fields in the log protos, they would need to read only a small fraction of the overall data size. This was one of the reasons why Dremel was able to process terabytes-worth of logs in seconds.

The other reason why Dremel was able to process data so fast was that its query engine used distributed computing. Dremel scaled to thousands of workers by structuring the computation as a tree, with the filters happening at the leaves and aggregation happening towards the root.

By 2010, Google was scanning petabytes of data a day using Dremel, and many people in the company used it in some form or another. It was the perfect tool for our nascent data marketplace team to pick up and use.

As the team productized Dremel, added a storage system, made it self-tuning, and exposed it to external users, the team realized that a Cloud version of Dremel was perhaps even more interesting than their original mission. The team renamed themselves “BigQuery”, following the naming convention for “Bigtable”, Google’s NoSQL database.

At Google, Dremel is used to query files that sit on Colossus, Google’s file store for storing data. BigQuery added a storage system that provided a table abstraction, not just a file abstraction. This storage system was key in making BigQuery simple to use and always fast, because it allowed key features like ACID transactions and automatic optimization, and it meant that users didn’t need to manage files.

Initially, BigQuery retained its Dremel roots and was focused on scanning logs. However, as more customers wanted to do data warehousing and more complex queries, BigQuery added improved support for joins and advanced SQL features like analytic functions. In 2016, Google launched support for standard SQL in BigQuery, which allowed users to run queries using standards-compliant SQL, rather than the awkward initial “DremelSQL” dialect.

BigQuery did not start out as a data warehouse, but it has evolved into one over the years. There are good things and bad things about this evolution. On the positive side, BigQuery was designed to solve problems people have with their data, even if they don’t fit nicely into data warehousing models. In this way, BigQuery is more than just a data warehouse. On the

downside, however, a few data warehousing features that people expect, like a Data Definition Language (DDL, e.g. CREATE statements) and a Data Manipulation Language (DML, e.g. INSERT statements), were missing until recently. That said, BigQuery has been focusing on a dual path: first, adding differentiated features that Google is in a unique position to provide, and second, becoming a great data warehouse in the cloud.

What makes BigQuery possible?

From an architectural perspective, BigQuery is fundamentally different from on-premise data warehouses like Teradata or Vertica, and from cloud data warehouses like RedShift and Azure Data Warehouse. BigQuery is the first data-warehouse to be a scale-out solution, so the only limits on speed and scale are the amount of hardware in the datacenter.

This section describes some of the components that go into making BigQuery successful and unique.

Separation of compute and storage

In many data warehouses, compute and storage live together on the same physical hardware. This co-location means that in order to add more storage, you might also need to add more compute power as well. Or to add more compute power, you'd also need to get additional storage capacity.

If everyone's data needs were similar, this wouldn't be a problem; there would be a consistent golden ratio of compute to storage that everyone would live by. But in practice, one or the other of the factors tends to be a limitation. Some data warehouses are limited by compute capacity, so they

slow down at peak times. Other data warehouses are limited by storage capacity, so maintainers have to figure out what data to throw out.

When you separate compute from storage as BigQuery does, it means you never have to throw out data unless you no longer want it. This may not sound like a big deal, but having access to full-fidelity data is immensely powerful. You may decide you want to calculate something in a different way, so you can go back to the raw data to re-query it. You cannot do this if you had gotten rid of the source data due to space constraints. You might decide that you want to dig into why some aggregate value exhibits strange behavior. You can't do this if you deleted the data that contributed to the aggregation.

Scaling compute is equally powerful. BigQuery resources are denominated in terms of “slots”, which are, roughly speaking, about half of a CPU core (we’ll cover slots in detail in Chapter X). BigQuery uses slots as an abstraction to indicate how many physical compute resources are available. Queries running too slow? Just add more slots. More people want to create reports? Add more slots. Want to cut back on your expenses? Decrease your slots.

Because BigQuery is a multi-tenant system that manages large pools of hardware resources, it is able to dole out the slots on a per-query or per-user basis. It is possible to reserve hardware for your project or organization, or you can run your queries in the shared on-demand pool. By sharing resources in this way, BigQuery can devote very large amounts of computing power to your queries. If you need more computing power than is available in the on-demand pool, you can purchase more via the BigQuery Reservation API.

Several BigQuery customers have reservations in the tens of thousands of slots, which means that if they only run one query at a time, those queries can consume tens of thousands of CPU cores at once. With some reasonable assumptions about numbers of CPU cycles per processed row, it is pretty easy to see that these instances can process billions or even trillions of rows per second.

In BigQuery, there are some customers that have petabytes of data but use a relatively small amount of it on a daily basis. Other customers store only a few gigabytes of data, but perform complex queries using thousands of CPUs. There isn't a one-size-fits-all approach that works for all use cases. Fortunately, the separation of compute and storage allows BigQuery to accommodate a wide range of customer needs.

Storage and Networking Infrastructure

BigQuery differs from other cloud data warehouses in that queries are served primarily from spinning disks in a distributed file system. Most competitor systems need to cache data within compute nodes in order to get good performance. BigQuery, on the other hand, relies on two systems unique to Google, the Colossus File System¹⁹ and Jupiter Networking²⁰, to ensure that data can be queried quickly no matter where it physically resides in the compute cluster.

Google's Jupiter networking fabric relies on a network configuration where smaller (and hence cheaper) switches are arranged to provide the capability that a much larger logical switch would otherwise be needed for. This topology of switches, along with a centralized software stack and custom hardware and software, allows 1 petabit of bisection bandwidth within a data center. That is the equivalent to 100k servers communicating

at 10 Gb/sec, and means that BigQuery can work without the need to co-locate the compute and storage. If the machines hosting the disks are at the other end of the datacenter from the machines running the computation, it will effectively run just as fast as if the two machines are in the same rack.

The fast networking fabric comes in handy in two places; to read in data from disk, and to shuffle between query stages. As discussed earlier, the separation of compute and storage in BigQuery allows any machine within the data center be able to ingest data from any storage disk. This requires, however, that the necessary input data to the queries be read over the network at very high speeds. The details of shuffle are described in Chapter X, but it suffices for now to understand that running complex distributed queries usually requires moving large amounts of data between machines at intermediate stages. Without a fast network connecting the machines doing the work, shuffle would become a bottleneck that slows down the queries significantly.

The networking infrastructure provides more than just speed -- it also allows for dynamic provisioning of bandwidth. Google data centers are connected through a backbone network called B4²¹ that is software-defined to allocate bandwidth in an elastic manner to different users, and to provide reliable quality of service for high-priority operations. This is crucial for implementing high-performing, concurrent queries.

Fast networking isn't enough, however, if the disk subsystem is slow or lacks enough scale. In order to support interactive queries, the data needs to be read from the disks fast enough so that they can saturate the network bandwidth available. Google's distributed file system is called Colossus and can coordinate hundreds of thousands of disks by constantly rebalancing old, cold data and distributing newly written data evenly

across disks.²² This means that the effective throughput is 10s of terabytes per second. By combining this effective throughput with efficient data formats and storage, BigQuery provides the ability to query petabyte-sized tables in minutes.

Managed storage

BigQuery's storage system is built on the idea that when you're dealing with structured storage, the appropriate abstraction is the table, not the file. Some other cloud-based and open source data processing systems expose the concept of the file to users, which puts users on the hook for managing file sizes and ensuring that the schema remains consistent. While creating files of an appropriate size for a static data store is possible, it is notoriously difficult to maintain optimal file sizes for data that is changing over time. Similarly, it is hard to maintain a consistent schema when you have a large number of files with self-describing schemas (e.g. Avro or Parquet) -- typically, every software update to systems producing those files results in changes to the schema. BigQuery ensures that all the data held within a table has a consistent schema, and enforces a proper migration path for historical data. By abstracting the underlying data formats and file sizes from the user, BigQuery can provide a seamless experience so that queries are always fast.

There is another advantage to BigQuery managing its own storage -- BigQuery can continue to get faster in a way that is transparent to the end user. For example, improvements in storage formats can be applied automatically to user data. Similarly, improvements in storage infrastructure become immediately available. Because BigQuery manages all the storage, users don't have to worry about backup or replication. Everything from upgrades and replication to backup and restoration are

handled transparently and automatically by the storage management system.

One key advantage of working with structured storage at the abstraction level of a table, rather than of a file, and of providing storage management to these tables transparently to the end-user is that tables allow BigQuery to support database-like features, such as DML (Data Manipulation Language). You can run a query that updates or deletes rows in a table and leave it to BigQuery to figure out the best way to modify the storage to reflect this information. BigQuery operations are ACID; that is, all queries will either commit completely or not at all. Rest assured that your queries will never see the intermediate state of another query, and queries started after another completes will never see old data. You do have the ability to fine-tune the storage by specifying directives that control how the data is stored, but these operate at the abstraction level of tables, not files. For example, it is possible to control how tables are partitioned and clustered (these features are covered in detail in Chapter XX) and thereby improve the performance and/or reduce the cost of queries against those tables.

Managed storage is strongly typed, which means that data is validated at entry to the system. Because BigQuery manages the storage, and allows users to interact with this storage only via its APIs, it can count on the underlying data not being modified outside of BigQuery. Thus, BigQuery can guarantee to not throw a validation error at read time about any of the data present in its managed storage. This guarantee also implies an authoritative schema, which is useful when figuring out how to query your tables. Besides improving query performance, the presence of an authoritative schema helps when trying to make sense of what data you have since a BigQuery schema contains not just type information, but also annotations and table descriptions about how the fields can be used.

One downside of managed storage is that it is harder to directly access and process the data using other frameworks. For example, had the data been available at the abstraction level of files, you might have been able to directly run a Hadoop job over a BigQuery dataset. BigQuery addresses this issue by providing a structured parallel API to read the data. This API lets you read at full speed from Spark or Hadoop jobs, but also provides extra features, like projection, filtering, and dynamic rebalancing.

Integration with GCP

Google Cloud follows the design principle called “separation of responsibility” wherein a small number of high-quality, highly focused products integrate tightly with each other. It is, therefore, important to consider the entire Google Cloud Platform when comparing BigQuery with other database products.

A number of different GCP products extend the usefulness of BigQuery or make it easier to understand how BigQuery is being used. We will talk about many of these related products in detail in this book, but it is worth being aware of the general separation of responsibilities:

StackDriver monitoring and audit logs provide ways to understand BigQuery usage in your organization.

Cloud Dataproc provides the ability to read, process, and write to BigQuery tables using Apache Spark programs.

Federated queries allow BigQuery to query data held in Google Cloud Storage, Cloud SQL (a relational database), Bigtable (a No-SQL database), Spanner (a distributed database), or Google Drive (which offers spreadsheets).

Google Cloud Data Loss Prevention API²³ helps you manage sensitive data and provides the capability to redact or mask personally identifiable information from your tables.

Other machine learning APIs extend what it is possible on data held in BigQuery -- for example, the Cloud Natural Language API can identify people, places, sentiment, etc. in free-form text (such as those of customer reviews) held in some table column.

Cloud Catalog provides the ability to discover data held across your organization.

Cloud Pub/Sub can be used to ingest streaming data and Cloud Dataflow to transform and load it into BigQuery. Cloud Dataflow can also be used to carry out streaming queries. You can, of course, interactively query the streaming data within BigQuery itself.²⁴

Data Studio provides charts and dashboards driven from data in BigQuery. Third-party tools such as Tableau and Looker also support BigQuery as a backend.

Cloud ML Engine provides the ability to train sophisticated machine learning programs from data held in BigQuery.

Cloud Composer allows for orchestration of BigQuery jobs along with tasks that need to be performed in Cloud Dataflow or other processing frameworks, whether on Google Cloud or on-prem in a hybrid cloud setup.

Taken together, BigQuery and the GCP ecosystem have features that span several other database products from other cloud vendors; you can use

them as an analytics warehouse, but also as an ELT system, as a data lake (queries over files), or a source of business intelligence. The rest of this book will paint a broad picture of how you can use BigQuery in all of its aspects.

Security and compliance

The integration with GCP goes beyond just interoperability with other products. Cross-cutting features provided by the platform provide consistent security and compliance.

The fastest hardware and most advanced software are of little use if you can't trust them with your data. BigQuery's security model is tightly integrated with the rest of Google's Cloud Platform, so it is possible to take a holistic view of your data security. BigQuery uses Google's Identity and Access Management (IAM) access control system to assign specific permissions to individual users or groups of users. BigQuery also ties in tightly with Google's Virtual Private Cloud (VPC) policy controls, which can protect against users who try to access data from outside your organization, or who try to export it to third parties. Both IAM and VPC controls are designed to work across Google cloud products, so you don't have to worry that certain products create a security hole.

BigQuery is available in every region where Google Cloud has a presence, enabling you to process the data in the location of your choosing. At the time of writing, Google Cloud has more than two dozen data centers around the world, and new ones are being opened at a fast rate. If you have business reasons for keeping data in Australia or Germany, it is possible to do so. Just create your dataset with the Australian or German region code, and all of your queries against the data will be done within

that region.

Some organizations have even stronger data location requirements that go beyond where data is stored and processed. Specifically, they want to ensure that their data cannot be copied or otherwise leave their physical region. GCP has physical region controls that apply across products; you can create a “VPC service controls” policy that disallows data movement outside of a selected region. If you have these controls enabled, users will not be able to copy data across regions or export to Google Cloud Storage buckets in another region.

1 <https://twitter.com/plamere/status/702168809445134336> on Feb 23, 2016

2 See <https://cloud.google.com/customers/alpega>

3 In reality, you’ll have to start the record keeping at the time customers borrow the equipment, so that you will know if customers abscond with the equipment. However, it’s rather early in this book to worry about that!

4 In this book, we’ll use “ad-hoc” query to refer to a query that is written without any attempt to prepare the database ahead of time by using features such as indexes.

5 MapReduce: Simplified Data Processing on Large Clusters, by Jeffery Dean and Sanjay Ghemawat, OSDI’04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA (2004), pp. 137-150. Available at <https://research.google.com/archive/mapreduce-osdi04.pdf>

6 See <http://spark.apache.org/>

7 On Google Cloud Platform, Cloud Dataproc (the managed Hadoop offering) addresses this conundrum in a different way. Because of the high bisectional bandwidth available within Google data centers, Cloud Dataproc clusters are able to be job-specific -- the data is stored on Google Cloud Storage and read over the wire on demand. This is possible only if bandwidths are high enough to approximate disk speeds. Don’t try this at home.

8 For your copy-paste convenience, all the code and query snippets in this book (including the query that follows) can be found in the GitHub repository for this book. See https://github.com/GoogleCloudPlatform/bigquery-oreilly-book/blob/master/01_intro/queries.txt

9 Not you specifically. This is a public dataset, and the owner of the dataset gave this

permission to all authenticated users. You can be less permissive with your data, sharing the dataset only with those within your domain or within your team.

- 10 See https://github.com/GoogleCloudPlatform/bigquery-oreilly-book/blob/master/01_intro/queries.txt
- 11 Keep in mind that both authors live in Seattle where it rains 150 days a year.
- 12 More details on the columnar storage format may be found in the following section.
- 13 For example, to compute conversion metrics based on the distance that a customer would have to travel to purchase a product.
- 14 We believe all mentions of price to be correct as of the writing of this book, but please do refer to the relevant policy and pricing sheets (<https://cloud.google.com/bigquery/pricing>) as these are subject to change.
- 15 See [https://en.wikipedia.org/wiki/Jim_Gray_\(computer_scientist\)](https://en.wikipedia.org/wiki/Jim_Gray_(computer_scientist))
- 16 See <http://itre.cis.upenn.edu/myl/JimGrayOnE-Science.pdf>
- 17 Today, BigQuery does provide the ability to export tables and results to Google Cloud Storage, so we did end up building the download link after all! But BigQuery is not just a download link -- most uses of BigQuery involve operating on the data in-place.
- 18 SQL does have a RECURSIVE keyword, but like many SQL engines, BigQuery does not support this. Instead, BigQuery offers better ways to deal with hierarchical data by supporting arrays and nesting.
- 19 See https://cloud.google.com/files/storage_architecture_and_challenges.pdf
- 20 See <https://cloudplatform.googleblog.com/2015/06/A-Look-Inside-Googles-Data-Center-Networks.html>
- 21 See <https://www.usenix.org/conference/atc15/technical-session/presentation/mandal>
- 22 See <http://www.pdsw.org/pdsw-discs17/slides/PDSW-DISCS-Google-Keynote.pdf> and <https://www.wired.com/2012/07/google-colossus/>
- 23 See <https://cloud.google.com/dlp>
- 24 The separation of responsibility here is that Cloud Dataflow is better for ongoing, routine processing while BigQuery is better for interactive, ad-hoc processing. Both Cloud Dataflow and BigQuery handle batch data as well as streaming data, and it is possible to run SQL queries within Cloud Dataflow.

Chapter 2. Query Essentials

BigQuery is first, and foremost, a data warehouse, by which we mean that it provides persistent storage for structured and semi-structured data (like JSON objects). The four basic CRUD operations are supported on this persistent storage:

Create, to insert new records, is implemented through load operations, by the SQL INSERT statement, and through a streaming insert API. You can also use SQL to create database objects like tables, views, and ML models as part of BigQuery's support of the Data Definition Language (DDL). We'll go into examples of each later.

Read, to retrieve records, is implemented by the SQL SELECT statement and by the bulk read API.

Update, to modify existing records, is implemented using the SQL UPDATE and MERGE statements as part of BigQuery's support of the Data Manipulation Language (DML). Note that, as we discussed in Chapter 1, BigQuery is an analytics tool and not meant to be used for frequent updates.

Delete, to remove existing records, implemented using SQL DELETE is also a DML operation.

BigQuery is a tool for data analysis and the majority of queries you can expect to write will be READ (#2 above) operations. Reading and analyzing your data is accomplished by the SELECT statement which is

the focus of this chapter. We will cover creating, updating, and deleting data in later chapters.

Simple queries

BigQuery supports a dialect of SQL that is compliant with SQL 2011¹. When the spec is ambiguous or otherwise lacking, BigQuery will follow the conventions set by existing SQL engines. There are other areas where there is no spec at all, such as with Machine Learning; in these cases, BigQuery defines its own syntax and semantics.

<NOTE>

For a long time, BigQuery supported only a limited subset of SQL with some Google enhancements. This was because BigQuery was based on an internal SQL query engine at Google (called Dremel) that was originally built to process log data held in protocol buffers.² Because it was not built as a general-purpose SQL engine, Dremel could use a dialect of SQL (now referred to as “legacy SQL”) that was well-suited to protocol buffers, which are used to hold hierarchical structures. For example, the legacy SQL dialect distinguished between records (the complete hierarchical structure pertaining to a log message) and rows (slices through the structure).³ For example, COUNT(*) in Dremel counts the number of non-NULL values in the most repeated field. While such features made certain types of queries much easier to write, Dremel took some getting used to because it was not standard SQL.

In this book, we will focus exclusively on standard SQL. The BigQuery user interface in the GCP web console defaults to standard SQL, and new features are not being backported to legacy SQL. However, some tools

and user interfaces still default to legacy SQL. If that is the case for any tool that you are using, preface the query with this comment on the first line: #standardsql:

```
#standardsql
```

```
SELECT DISTINCT gender  
FROM `bigquery-public-data`.new_york_citibike.citibike_trips
```

If the BigQuery service receives a query string whose first line consists of #standardsql, the query engine will treat what follows as standard SQL even if the client itself does not know about standard SQL.

```
</NOTE>
```

Retrieving rows with SELECT

The SELECT statement allows you to retrieve the values of specified columns from a table. For example, consider the [New York bicycle rentals dataset](#) -- it contains several columns relating to bicycle rentals, including the trip duration and the gender of the person renting the bicycle. We can pull out the values of these columns using the SELECT statement (lines starting with double dashes or # are comments):

```
-- simple select  
SELECT  
gender, tripduration  
FROM  
`bigquery-public-data`.new_york_citibike.citibike_trips  
LIMIT 5
```

The result looks something like this:

Row	gender	tripduration
1	male	371
2	male	1330
3	male	830
4	male	555
5	male	328

The result-set has two columns (gender and tripduration) in the order specified in the SELECT. There are 5 rows in the result-set because we limited it to 5 in the final line of the query. BigQuery distributes the task of fetching rows to multiple workers, each of which may read a different shard (or part) of the dataset, so if you run the query above, you might get a different set of 5 rows.

Note that using a LIMIT only limits the amount of data displayed to you and not the amount of data the query engine needs to process. You are typically charged based on the amount of data processed by your queries and this usually implies that the more columns your query reads, the higher your cost will be. The number of rows processed will usually be the total size of the table that you are reading, although there are ways to optimize this (covered in Chapter X). We'll cover performance and pricing considerations in later chapters.

The values are being retrieved from

`bigquery-public-data.new_york_citibike.citibike_trips`

Here, `bigquery-public-data` is the project id, `new_york_citibike` is the

dataset and citibike_trips is the table.

The project id indicates ownership of the persistent storage associated with the dataset and its tables. The owner of bigquery-public-data is paying the storage costs associated with the new_york dataset. The cost of the query is paid by the project within which the query is issued. If you run the query above, you pay the query costs. Datasets provide for identity and access management (IAM). The person who created the new_york_citibike dataset in BigQuery⁴ made it public, which is why we were able to list the tables in the dataset and query one of those tables. The citibike_trips table contains all the bicycle trips. The project, dataset, and table are separated by dots. The backtick is needed in this case because the hyphen (-) in the project name (bigquery-public-data) would otherwise be interpreted as subtraction. Most developers simply enclose the entire string within backticks:

```
-- simple select
SELECT
gender, tripduration
FROM
`bigquery-public-data.new_york_citibike.citibike_trips`
LIMIT 5
```

Although this is simpler, you lose the ability to use the table name (citibike_trips) as an alias. So, it is worth developing the habit of putting the backticks only around the project name and avoid using hyphens when you create your own datasets and tables.

For a long time, our recommendation was that tables in BigQuery be stored in denormalized form (i.e., a single table often contains all the data

you'd need without the need for joining multiple tables). However, with improvements in the service, this is no longer necessary. It is possible now to get good performance even with a star schema.

So to review, from `bigquery-public-data`.new_york_citibike.citibike_trips the three key components to know are:

BigQuery Object	Name	Description
Project	bigquery-public-data	Owner of the persistent storage associated with the dataset and its tables. The project also governs the use of all other GCP products as well.
Dataset	new_york_citibike	Datasets are top-level containers that are used to organize and control access to tables and views. A user can own multiple datasets.
Table / View	citibike_trips	A table or view must belong to a dataset, so you need to create at least one dataset before loading data into BigQuery. ^a

^a See <https://cloud.google.com/bigquery/docs/datasets-intro>

Distinguishing between each of these three will be critical later when we discuss data geographic location, access, and sharing of data.

Aliasing column names with AS

By default, the names of the columns in the result-set match those of the table from which the data are retrieved. It is possible to alias the column names using AS:

```
-- Aliasing column names
```

```
SELECT
```

```

gender, tripduration AS rental_duration
FROM
`bigquery-public-data`.new_york_citibike.citibike_trips
LIMIT 5

```

This now yields (your results may be a different set of five):

Row	gender	rental_duration
1	male	432
2	female	1186
3	male	799
4	female	238
5	male	668

Aliasing is useful when you are transforming data. For example, without the alias, a statement such as:

```

SELECT
gender, tripduration/60
FROM
`bigquery-public-data`.new_york_citibike.citibike_trips
LIMIT 5

```

would result in an automatically assigned column name for the second column in the result-set:

Row	gender	f0_
1	male	6.18333333333334
2	male	22.166666666666668
3	male	13.83333333333334
4	male	9.25
5	male	5.46666666666667

You can assign the second column a more descriptive name by adding an alias to your query:

```
SELECT  
gender, tripduration/60 AS duration_minutes  
FROM  
`bigquery-public-data`.new_york_citibike.citibike_trips  
LIMIT 5
```

which yields a result similar to this:

Row	gender	duration_minutes
1	male	6.18333333333334
2	male	22.166666666666668
3	male	13.83333333333334
4	male	9.25
5	male	5.466666666666667

Filtering with WHERE

To find rentals of less than 10 minutes, we could filter the results returned by the SELECT using a WHERE clause:

```
SELECT  
gender, tripduration  
FROM  
`bigquery-public-data`.new_york_citibike.citibike_trips  
WHERE tripduration < 600
```

LIMIT 5

As expected, the result-set now consists only of rows where the trip duration is less than 600 seconds:

Row	gender	tripduration
1	male	178
2	male	518
3	male	376
4	male	326
5	male	516

The WHERE clause can include boolean expressions. For example, to find trips rented by females and lasting between five and ten minutes, you could do:

SELECT

gender, tripduration

FROM

`bigquery-public-data`.new_york_citibike.citibike_trips

WHERE tripduration >= 300 AND tripduration < 600 AND gender = 'female'

LIMIT 5

The OR keyword also works, as does NOT. For example, to find non-female riders (i.e. male riders and those whose gender is unknown), the WHERE clause could be:

WHERE tripduration < 600 AND NOT gender = 'female'

It is also possible to use parentheses to control the order of evaluation. To find female riders who take short trips as well as all male riders, you could use:

```
WHERE (tripduration < 600 AND gender = 'female') OR gender = 'male'
```

The WHERE clause operates on the columns in the FROM clause, and so it is not possible to reference aliases from the SELECT list in the WHERE. To retain only riders shorter than 10 minutes, it is not possible to do:

```
SELECT  
    gender, tripduration/60 AS minutes  
FROM  
    `bigquery-public-data`.new_york_citibike.citibike_trips  
WHERE minutes < 10 -- CAN NOT REFERENCE ALIAS IN WHERE  
LIMIT 5
```

Instead, you will have to repeat the transformation in the WHERE also (we will see better alternatives later):

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

```
LIMIT 5
```

Subqueries with WITH

You can reduce the repetitiveness and retain the use of the alias by using a subquery:

```
SELECT * FROM (
```

```
SELECT
```

```
    gender, tripduration / 60 AS minutes
```

```
FROM
```

```
`bigquery-public-data`.new_york_citibike.citibike_trips
```

```
)
```

```
WHERE minutes < 10
```

```
LIMIT 5
```

The outer SELECT operates on the inner subquery that is enclosed within parentheses. Because the alias happens in the inner query, the outer query can use the alias in its WHERE clause.

Queries with parentheses can become quite hard to read. A better approach is to use a WITH clause to provide names to what would otherwise have been subqueries:

```
WITH all_trips AS (
```

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

In BigQuery, the WITH clause behaves like a named subquery and does not create temporary tables. We will refer to all_trips as a “from_item” -- it’s not a table, but you can select from it.

Sorting with ORDER BY

To control the order of rows in the result-set, use ORDER BY:

```
SELECT  
    gender, tripduration/60 AS minutes  
  
FROM  
    `bigquery-public-data`.new_york_citibike.citibike_trips
```

```
WHERE gender = 'female'
```

```
ORDER BY minutes DESC
```

```
LIMIT 5
```

By default, rows in results are not ordered. If an order column is specified, the default is ascending order. By asking for the rows to be listed in descending order and limiting to 5, we get the five longest trips by women in the dataset:

Row	gender	minutes
1	female	250348.9
2	female	226437.9333333332
3	female	207988.71666666667
4	female	159712.05
5	female	154239.0

Note that we are ordering by minutes, which is an alias -- because the ORDER BY is carried out after the SELECT, it is possible to use aliases in ORDER BY.

Aggregates

When we converted seconds to minutes by dividing by 60 (see the previous section), we operated on every row in the table and transformed it. It is also possible to apply a function to aggregate all the rows so that the result-set contains only one row.

Computing aggregates with GROUP BY

To find the average duration of trips by male riders, we could do:

```
SELECT
AVG(tripduration / 60) AS avg_trip_duration
FROM
`bigquery-public-data`.new_york_citibike.citibike_trips
WHERE
gender = 'male'
```

This yields:

Row	avg_trip_duration
1	13.415553172043886

indicating that the average bicycle trip taken by male riders in New York is about 13.4 minutes. Because the dataset is continuously updated, though, your result might be different.

How about female riders? While you could run the above query twice, once for male riders and the next for female ones, it seems wasteful to traverse through the dataset a second time, changing the WHERE clause. Instead, you can use a GROUP BY:

```
SELECT
gender, AVG(tripduration / 60) AS avg_trip_duration
FROM
`bigquery-public-data`.new_york_citibike.citibike_trips
WHERE
tripduration is not NULL
```

GROUP BY

gender

ORDER BY

avg_trip_duration

This yields:

Row	gender	avg_trip_duration
1	male	13.415553172043886
2	female	15.977472148805207
3	unknown	31.4395230232542

The aggregates have now been computed on each group separately. The SELECT expression can include the thing being grouped by (gender) and aggregates (AVG). Note that there are actually three genders in the dataset: male, female, and unknown.

Counting records with COUNT

To see how many rides went into the previous averages we can simply add a COUNT()

SELECT

gender,

COUNT(*) AS rides,

AVG(tripduration / 60) AS avg_trip_duration

FROM

```
`bigquery-public-data`.new_york_citibike.citibike_trips
```

WHERE

tripduration IS NOT NULL

GROUP BY

gender

ORDER BY

avg_trip_duration

Row	gender	rides	avg_trip_duration
1	male	35611787	13.415553172043888
2	female	11376412	15.97747214880521
3	unknown	6120522	31.439523023254207

Filtering grouped items with HAVING

It is possible to post-filter the grouped operations using the HAVING clause. So, we can find which genders take trips that, on average, last longer than 14 minutes using:

SELECT

gender, AVG(tripduration / 60) AS avg_trip_duration

FROM

```
`bigquery-public-data`.new_york_citibike.citibike_trips
```

WHERE tripduration IS NOT NULL

GROUP BY

gender

HAVING avg_trip_duration > 14

ORDER BY

avg_trip_duration

This yields:

Row	gender	avg_trip_duration
1	female	15.977472148805209
2	unknown	31.439523023254203

Note that while it is possible to filter the gender or tripduration with a WHERE clause, it is not possible to filter by average duration using a WHERE because the average duration is computed only after the items have been grouped (try it!).

Finding unique values with DISTINCT

What values of gender are present in the dataset? While we could use the GROUP BY, a simpler way to get a list of distinct values of a column is to use SELECT DISTINCT:

SELECT DISTINCT

gender

```
FROM  
`bigquery-public-data`.new_york_citibike.citibike_trips
```

This yields a result-set with just four rows:

Row	gender
1	male
2	female
3	unknown
4	

Four rows? What is the fourth row? Let's explore:

```
SELECT
```

```
bikeid,
```

```
tripduration,
```

```
gender
```

```
FROM
```

```
`bigquery-public-data`.new_york_citibike.citibike_trips
```

```
WHERE gender = “”
```

```
LIMIT 100
```

Row	bikeid	tripduration	gender
1	null	null	
2	null	null	
3	“”	“”	

3	null	null
...		

In this particular case, a blank gender value seems to indicate missing or poor quality data. We'll discuss missing data (NULL values) and how you can account for and transform them in Chapter 3, but briefly: if you want to filter for NULLs in a WHERE clause, use the IS NULL or IS NOT NULL operators since other comparison operators ($=$, \neq , $<$, $>$) applied to a NULL returns NULL and will, therefore, never match the WHERE condition.

Going back to our original query for DISTINCT genders, it's important to note that the DISTINCT modifies the entire SELECT, not just the gender column. To see what we mean, add a second column to the query's SELECT list:

```
SELECT DISTINCT  
    gender,  
    usertype  
FROM  
    `bigquery-public-data`.new_york_citibike.citibike_trips  
WHERE gender != "
```

This results in six rows, i.e., you get a row for every combination of unique gender and user type (subscriber or customer) that exists in the dataset:

Row	gender	usertype
1	male	Subscriber
2	unknown	Customer
3	female	Subscriber
4	female	Customer
5	male	Customer
6	unknown	Subscriber

A brief primer on arrays and structs

In this chapter, we will provide a brief primer on arrays so that we can illustrate many of the data types and functions in the next chapter on small, illustrative datasets. The combination of ARRAY (the square brackets in the query below) and UNNEST gives us a quick way to experiment with queries, functions, and data types.

For example, if we want to know how the SPLIT function of a string behaves, we could simply try it out:

```
SELECT
```

```
city, SPLIT(city, ' ') AS parts
```

```
FROM (
```

```
SELECT * from UNNEST([
```

```
'Seattle WA', 'New York', 'Singapore'
```

```
]) AS city
```

)

The result of the quick query above is:

Row	city	parts
1	Seattle WA	Seattle
		WA
2	New York	New
		York
3	Singapore	Singapore

This ability to hardcode an array of values in the SQL query itself will allow us to play with arrays and data types without having to find an appropriate dataset or wait for long queries to finish. Even better, this processes 0 bytes and, therefore, does not incur BigQuery charges⁵.

Another way to quickly experiment with a set of values employs UNION ALL to combine single row 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

which yields the two rows in the small inline dataset that have fewer than 2000 rides:

Row	day	numrides	oneways
1	Sat	1451	1018
2	Mon	1476	736

In the next chapter, we will use such inline datasets with hard-coded numbers to illustrate various aspects of the way different data types and functions behave.

The purpose of this section is to quickly introduce arrays and structs so that we can use them in illustrative examples. We will review these concepts in greater detail later in Chapter X, so feel free to quickly skim the remainder of this section for now.

Creating ARRAYS using ARRAY_AGG

Consider finding the number of trips by gender and year:

SELECT

gender

, EXTRACT(YEAR from starttime) AS year --

, COUNT(*) AS numtrips

FROM

`bigquery-public-data`.new_york_citibike.citibike_trips

WHERE gender != 'unknown' and starttime IS NOT NULL

GROUP BY gender, year

HAVING year > 2016

This returns:

Row	gender	year	numtrips
1	male	2017	9306602
2	male	2018	3955871
3	female	2018	1260893
4	female	2017	3236735

<TIP>

What's with the leading commas in the SELECT clause? Standard SQL (at least at the time of writing) does not support a trailing comma and so, moving the comma to the next line allows us to easily reorder or comment lines and still have a working query:

SELECT

gender

, EXTRACT(YEAR from starttime) AS year

-- comment out this line , COUNT(1) AS numtrips

FROM etc.

Trust us, the leading comma will become second nature after a while and

will greatly speed up your development.⁶

</TIP>

If we want to get a time-series of the number of trips associated with each gender over the years, i.e.,:

Row	gender	numtrips
1	male	9306602
		3955871
2	female	3236735
		1260893

we would need to create an array of the numbers of trips. You can represent that array in SQL using the ARRAY type and create such an array using ARRAY_AGG:

SELECT

gender

, ARRAY_AGG(numtrips order by year) AS numtrips

FROM (

SELECT

gender

, EXTRACT(YEAR from starttime) AS year

, COUNT(1) AS numtrips

```
FROM  
`bigquery-public-data`.new_york_citibike.citibike_trips  
  
WHERE gender != 'unknown' and starttime IS NOT NULL  
  
GROUP BY gender, year  
  
HAVING year > 2016  
  
)  
  
GROUP BY gender
```

Normally, when you group by gender, you will compute a single scalar value for the group, such as the AVG(numtrips) to find the average number of trips across all years. ARRAY_AGG allows you to collect the individual values and put them into an ordered list, or ARRAY.

The ARRAY type is not limited to the results of queries. Because BigQuery can ingest hierarchical formats such as JSON, it is possible that the input data contains JSON arrays. Perhaps:

```
[  
{  
  "gender": "male",  
  "numtrips": [  
    "9306602",
```

```
        "3955871"  
    ]  
},  
{  
    "gender": "female",  
    "numtrips": [  
        "3236735",  
        "1260893"  
    ]  
}  
]
```

Creating a table by ingesting such a JSON file will result in a table whose numtrips column is an ARRAY type. An array is an ordered list of non-null elements, e.g. ARRAY<INT64> is an array of integers.

<NOTE>

Technically, NULL elements in arrays are permissible as long as you don't try to persist them to a table. Thus, for example, the following will not work because you are trying to persist the array [1, NULL, 2] to the temporary table that holds the results:

```
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 ARRAY_AGG(IF(position = 2, NULL, position)) as positions
from example
```

However, this will work because the intermediate array with a null element is not being persisted:

```
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 ARRAY_LENGTH(ARRAY_AGG(IF(position = 2, NULL,
position))) from example
```

</NOTE>

ARRAY of STRUCT

A STRUCT is a group of fields in order. The fields can be named (if omitted, BigQuery will assign them names) but are recommended for readability:

```
SELECT
```

```
[
```

```
STRUCT('male' as gender, [9306602, 3955871] as numtrips)
```

```
, STRUCT('female' as gender, [3236735, 1260893] as numtrips)
```

```
] AS bikerides
```

This results in:

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

TUPLE

We could have left out the STRUCT keyword and the names of the fields, in which case we would have ended up with a tuple or anonymous struct. BigQuery will assign arbitrary names for unnamed columns and struct fields in the result of a query. Thus:

```
SELECT
```

```
[
```

('male', [9306602, 3955871])

, ('female', [3236735, 1260893])

]

yields:

Row	f0_._field_1	f0_._field_2
1	male	9306602
		3955871
	female	3236735
		1260893

Obviously, leaving out aliases for the field names makes subsequent queries unreadable and unmaintainable. Do not do this except for throw-away experimentation.

Working with arrays

Given an array, we can find the length of the array and retrieve individual items:

SELECT

ARRAY_LENGTH(bikerides) as num_items

, bikerides[OFFSET(0)].gender as first_gender

FROM

(SELECT

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

which yields:

Row	num_items	first_gender
1	2	male

Offsets are numbered starting at zero, which is why OFFSET(0) gives us the first item in the array⁷.

UNNEST an array

In the query:

```
SELECT
```

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

the SELECT returns exactly one row containing an array and so both genders are part of the same row (look at the Row column):

Row	f0_.gender	f0_.numtrips
1	male	9306602
		3955871
2	female	3236735
		1260893

UNNEST is a function that returns the elements of an array as rows, so we can UNNEST the result array to get a row corresponding to each item in the array:

```
SELECT * from UNNEST(
```

```
[
```

```
STRUCT('male' as gender, [9306602, 3955871] as numtrips)
```

```
, STRUCT('female' as gender, [3236735, 1260893] as numtrips)
```

```
])
```

This yields:

Row	gender	numtrips
1	male	9306602
		3955871
2	female	3236735
		1260893

Notice that UNNEST is actually a from_item -- you can SELECT from it. You can select just parts of the array as well. For example, we can get just the numtrips column using:

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

which yields:

Row	numtrips
1	9306602
	3955871
2	3236735
	1260893

Joining tables

Data warehouse schemas often rely on a primary “fact” table that contains events, and satellite “dimension” tables that contain extended, slowly-changing information. For example, a retail schema might have a “Sales” table as the fact table, and then “Products” and “Customers” tables as dimensions. When using this type of schema, the majority of queries will require a JOIN operation, such as to return the names of all the products purchased by a particular customer.

BigQuery supports all of the common JOIN types from relational algebra; inner joins, outer joins, cross joins, anti-joins, semi-joins, and anti semi-joins. While it can sometimes be faster to avoid a JOIN, BigQuery can

efficiently JOIN tables of almost any size. Chapter X discusses more about how to optimize JOIN performance, but for now, we'll just describe basic join operation.

The JOIN, explained

In Chapter 1, we looked at an example of a join across tables in two different datasets produced by two different organizations:

```
WITH bicycle_rentals AS (
  SELECT
    COUNT(starttime) as num_trips,
    EXTRACT(DATE from starttime) as trip_date
  FROM `bigquery-public-data`.new_york_citibike.citibike_trips
  GROUP BY trip_date
),
```

```
rainy_days AS
(
  SELECT
    date,
    (MAX(prcp) > 5) AS rainy
  FROM (
    SELECT
      wx.date AS date,
      IF (wx.element = 'PRCP', wx.value/10, NULL) AS prcp
    FROM
      `bigquery-public-data`.ghcn_d.ghcnd_2016 AS wx
    WHERE
      wx.id = 'USW00094728'
```

```
)  
GROUP BY  
date  
)
```

```
SELECT  
ROUND(AVG(bk.num_trips)) AS num_trips,  
wx.rainy  
FROM bicycle_rentals AS bk  
JOIN rainy_days AS wx  
ON wx.date = bk.trip_date  
GROUP BY wx.rainy
```

We asked you to ignore the syntax then, so let's parse it now.

The first WITH pulls out the number of trips by day from the citibike_trips table into a from_item called bicycle_rentals. This is not a table, but it is something that we can select from. Hence, we will refer to it as a “from_item”. The second from_item is called rainy_days and is created from the Global Historical Climate Network (GHCN) observation in each day. This from_item marks each day as being rainy or not depending on whether at least 5 mm of precipitation was observed at weather station ‘USW00094728’ which happens to be in New York.

So, now we have two from_items. Let's visualize them separately:

```
WITH bicycle_rentals AS (  
  
SELECT  
  
COUNT(starttime) as num_trips,
```

```
EXTRACT(DATE from starttime) as trip_date  
FROM `bigquery-public-data`.new_york_citibike.citibike_trips  
GROUP BY trip_date  
)
```

```
SELECT * from bicycle_rentals LIMIT 5
```

The bicycle_rentals from_item looks like this:

Row	num_trips	trip_date
1	31287	2013-09-16
2	22477	2015-12-30
3	37812	2017-09-02
4	54230	2017-11-15
5	25719	2013-11-07

Similarly, the rainy_days from_item looks like this:

Row	date	rainy
1	2016-10-11	false
2	2016-12-13	false
3	2016-09-28	false
4	2016-01-25	false
5	2016-05-24	false

We can now join both these from_items using the join condition that the trip_date in one is the same as the date in the second:

```
SELECT
```

```
bk.trip_date,  
bk.num_trips,  
wx.rainy  
FROM bicycle_rentals AS bk  
JOIN rainy_days AS wx  
ON wx.date = bk.trip_date  
LIMIT 5
```

This creates a table where columns from the two tables are joined by date:

Row	trip_date	num_trips	rainy
1	2016-07-13	55486	false
2	2016-04-25	42308	false
3	2016-09-27	61346	true
4	2016-07-15	48572	false
5	2016-05-20	52543	false

Given this, finding the average number of trips on rainy and non-rainy dates is straightforward.

What we have illustrated is called an inner join, and it is the type of join used if no join type is specified.

The way the JOIN works:

Create two from_items. These can be anything -- any two of a table, a subquery, an array, or a WITH statement that can be selected from.

Identify a join condition. The join condition does not need to be an

equality condition -- any boolean condition that uses the two from_items will do.

Select the columns you want. If identically named columns exist in both from_items, use aliases (bk, wx in the example query above) to clearly specify which from_item the column needs to come from.

If not using an INNER JOIN, specify a join type.

The only requirement for carrying out such a join is that all the datasets used to create the from_items are in the same BigQuery region (all BigQuery public datasets are in the US region).

Inner join

There are several types of joins. The inner join (INNER JOIN, or simply JOIN), to which the above example defaulted, creates a common set of rows to select from:

```
WITH from_item_a AS (
```

```
    SELECT 'Dalles' as city, 'OR' as state
```

```
    UNION ALL SELECT 'Tokyo', 'Tokyo'
```

```
    UNION ALL SELECT 'Mumbai', 'Maharashtra'
```

```
),
```

```
from_item_b AS (
```

```
    SELECT 'OR' as state, 'USA' as country
```

```
UNION ALL SELECT 'Tokyo', 'Japan'  
UNION ALL SELECT 'Maharashtra', 'India'  
)  
SELECT from_item_a.* , country  
FROM from_item_a  
JOIN from_item_b  
ON from_item_a.state = from_item_b.state
```

The first from_item has a list of cities and the second from_item identifies the country each of the states belongs to. Joining the two yields a dataset with three columns:

Row	city	state	country
1	Dalles	OR	USA
2	Tokyo	Tokyo	Japan
3	Mumbai	Maharashtra	India

Again, the join condition does not need to be an equality check. Any boolean condition will do, although it's best to use an equality condition if possible because BigQuery will return an error if the JOIN cannot be executed efficiently.

For example, we might have a business rule that shipping from one country to another involves a surcharge. To get a list of countries for which there will be a surcharge from a given location, we could have

specified:

```
SELECT from_item_a.*, country AS surcharge
```

```
FROM from_item_a
```

```
JOIN from_item_b
```

```
ON from_item_a.state != from_item_b.state
```

and obtained:

Row	city	state	surcharge
1	Dalles	OR	Japan
2	Dalles	OR	India
3	Tokyo	Tokyo	USA
4	Tokyo	Tokyo	India
5	Mumbai	Maharashtra	USA
6	Mumbai	Maharashtra	Japan

Notice that we get a row for each time that the join condition is met. Since there are two rows where the state doesn't match, we get two rows for each row in the original from_item_a. If the join condition is not met for some row, that row's data items will not make it to the output.

Cross Join

The cross join, or cartesian product, is a join with no join condition. All rows from both from_items are joined. This is the join that we would get if the join condition of an INNER JOIN always evaluated to true.

For example, suppose we organized a tournament and have a table of the

winners of each event in the tournament, and another table containing the gifts for each event. We can give each winner the gift corresponding to his/her event only, by doing an INNER JOIN:

WITH winners AS (

SELECT 'John' as person, '100m' as event

UNION ALL SELECT 'Hiroshi', '200m'

UNION ALL SELECT 'Sita', '400m'

),

gifts AS (

SELECT 'Google Home' as gift, '100m' as event

UNION ALL SELECT 'Google Hub', '200m'

UNION ALL SELECT 'Pixel3', '400m'

)

SELECT winners.*, gifts.gift

FROM winners

JOIN gifts

This returns:

person	event	gift
John	100m	Google Home
Hiroshi	200m	Google Hub
Sita	400m	Pixel3

row	person	event	gift
1	John	100m	Google Home
2	Hiroshi	200m	Google Hub
3	Sita	400m	Pixel3

On the other hand, if we wish to give each gift to each winner (i.e. each winner gets all three gifts), we could do a cross join:

WITH winners AS (

SELECT 'John' as person, '100m' as event

UNION ALL SELECT 'Hiroshi', '200m'

UNION ALL SELECT 'Sita', '400m'

),

gifts AS (

SELECT 'Google Home' as gift

UNION ALL SELECT 'Google Hub'

UNION ALL SELECT 'Pixel3'

)

SELECT person, gift

FROM winners

CROSS JOIN gifts

This yields a row for each potential combination:

Row	person	gift
1	John	Google Home
2	John	Google Hub
3	John	Pixel3
4	Hiroshi	Google Home
5	Hiroshi	Google Hub
6	Hiroshi	Pixel3
7	Sita	Google Home
8	Sita	Google Hub
9	Sita	Pixel3

While we wrote:

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

```
FROM from_item_a
```

```
CROSS JOIN from_item_b
```

we could also have written:

```
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.

Outer join

Let's say that we have winners in events for which there is no gift and gifts for events that didn't happen in our tournament:

WITH winners AS (

SELECT 'John' as person, '100m' as event

UNION ALL SELECT 'Hiroshi', '200m'

UNION ALL SELECT 'Sita', '400m'

UNION ALL SELECT 'Kwame', '50m'

),

gifts AS (

SELECT 'Google Home' as gift, '100m' as event

UNION ALL SELECT 'Google Hub', '200m'

UNION ALL SELECT 'Pixel3', '400m'

UNION ALL SELECT 'Google Mini', '5000m'

)

In an inner join (on the event column), the winner of the 50m dash doesn't get a gift and the gift for the 5000m event goes unclaimed. In a cross-join, as we noted, every winner gets every gift. Outer joins control what happens if the join condition is not met. The various types of joins and the resulting output are summarized below:

Syntax	What happens	Output												
<pre>SELECT person, gift FROM winners INNER JOIN gifts ON winners.event = gifts.event</pre>	Only rows that meet the join condition are retained	<table border="1"> <thead> <tr> <th>Row</th><th>person</th><th>gift</th></tr> </thead> <tbody> <tr> <td>1</td><td>John</td><td>Google Home</td></tr> <tr> <td>2</td><td>Hiroshi</td><td>Google Hub</td></tr> <tr> <td>3</td><td>Sita</td><td>Pixel3</td></tr> </tbody> </table>	Row	person	gift	1	John	Google Home	2	Hiroshi	Google Hub	3	Sita	Pixel3
Row	person	gift												
1	John	Google Home												
2	Hiroshi	Google Hub												
3	Sita	Pixel3												

Syntax	What happens	Output																		
<pre>SELECT person, gift FROM winners FULL OUTER JOIN gifts ON winners.event = gifts.event</pre>	All rows are retained even if the join condition is not met	<table border="1"> <thead> <tr> <th>Row</th><th>person</th><th>gift</th></tr> </thead> <tbody> <tr> <td>1</td><td>John</td><td>Google Home</td></tr> <tr> <td>2</td><td>Hiroshi</td><td>Google Hub</td></tr> <tr> <td>3</td><td>Sita</td><td>Pixel3</td></tr> <tr> <td>4</td><td>Kwame</td><td><i>null</i></td></tr> <tr> <td>5</td><td><i>null</i></td><td>Google Mini</td></tr> </tbody> </table>	Row	person	gift	1	John	Google Home	2	Hiroshi	Google Hub	3	Sita	Pixel3	4	Kwame	<i>null</i>	5	<i>null</i>	Google Mini
Row	person	gift																		
1	John	Google Home																		
2	Hiroshi	Google Hub																		
3	Sita	Pixel3																		
4	Kwame	<i>null</i>																		
5	<i>null</i>	Google Mini																		

```
SELECT person,  
       gift  
  FROM winners  
LEFT OUTER  
JOIN gifts  
    ON winners.event  
= gifts.event
```

All the winners are retained, but some gifts are discarded.

Row	person	gift
1	John	Google Home
2	Hiroshi	Google Hub
3	Sita	Pixel3
4	Kwame	<i>null</i>

```
SELECT person,  
       gift  
  FROM winners  
RIGHT OUTER  
JOIN gifts  
    ON winners.event  
= gifts.event
```

All the gifts are retained, but some winners aren't.

Row	person	gift
1	John	Google Home
2	Hiroshi	Google Hub
3	Sita	Pixel3
4	<i>null</i>	Google Mini

Saving and sharing

The BigQuery web user interface offers the ability to save and share queries. This is handy for collaboration, since you can send colleagues a

link to the query text that enables them to execute the query immediately. Note: Even if someone has your query they may not be able to execute it if they don't have access to your data. We'll discuss how to share and limit access to your datasets in Chapter X.

Query history and caching

It should be noted that BigQuery retains, for audit and caching purposes, a history of the queries that you submitted to the service (regardless of whether the queries succeeded or not):

The screenshot shows the BigQuery web interface with the "Query history" tab selected. On the left, there's a sidebar with "Saved queries", "Job history", and "Transfers" sections. Below that is a "Resources" section with a "+ ADD DATA" button and a search bar. The main area is titled "Query editor" and contains a query text area with the following code:

```

1 WITH winners AS (
2   SELECT 'John' as person, '100m' as event
3   UNION ALL SELECT 'Hiroshi', '200m'
4   UNION ALL SELECT 'Sita', '400m'
5   UNION ALL SELECT 'Kwame', '50m'
6 ),

```

Below the code, it says "Processing location: US". There are buttons for "Run query", "Save query", "Save view", and "More".

The main content area has a "Query history" header with a "REFRESH" button. It shows tabs for "Personal history" (which is selected) and "Project history". There are filters for "Sort by Date" and "Filter queries".

The "Today" section lists recent queries:

Time	Status	Query Text
1:48 AM	✓	WITH winners AS (SELECT 'John' as person, '100m' as ev
1:46 AM	✓	WITH winners AS (SELECT 'John' as person, '100m' as ev
1:45 AM	✓	WITH winners AS (SELECT 'John' as person, '100m' as ev
1:43 AM	✓	WITH winners AS (SELECT 'John' as person, '100m' as ev
1:43 AM	!	WITH winners AS (SELECT 'John' as person, '100m' as ev
1:39 AM	✓	WITH winners AS (SELECT 'John' as person, '100m' as ev

Figure 2-1. The history of queries submitted to the BigQuery service is available via the “Query history” tab in the web user interface.

This history includes all queries submitted by you to the service, not just those submitted via the web user interface. Clicking on any of the queries provides the text of the query and the ability to open the query in the

editor so that it can be modified and re-run. In addition, the historical information includes the amount of data processed by the query and the execution time. At the time of writing, the history is limited to 1000 queries and six months.

The actual results of the query are stored in a temporary table that expires after about 24 hours. If you are within that expiry window, you will also be able to browse the results of the query from the web user interface. Your personal history is available only to you. Administrators of the project to which your query was billed will also see your query text in the project's history.

This temporary table is also used as a cache if the exact same query text is submitted to the service and the query does not involve dynamic elements such as CURRENT_TIMESTAMP() or RAND(). Cached query results incur no charges, but note that the algorithm to determine whether a query is a duplicate simply does a string match -- even an extra whitespace can result in the query being re-executed.

Saved queries

You can save any query by loading it into the query editor, using the “Save query” button, and giving the query a name. BigQuery will provide a URL to the query text.

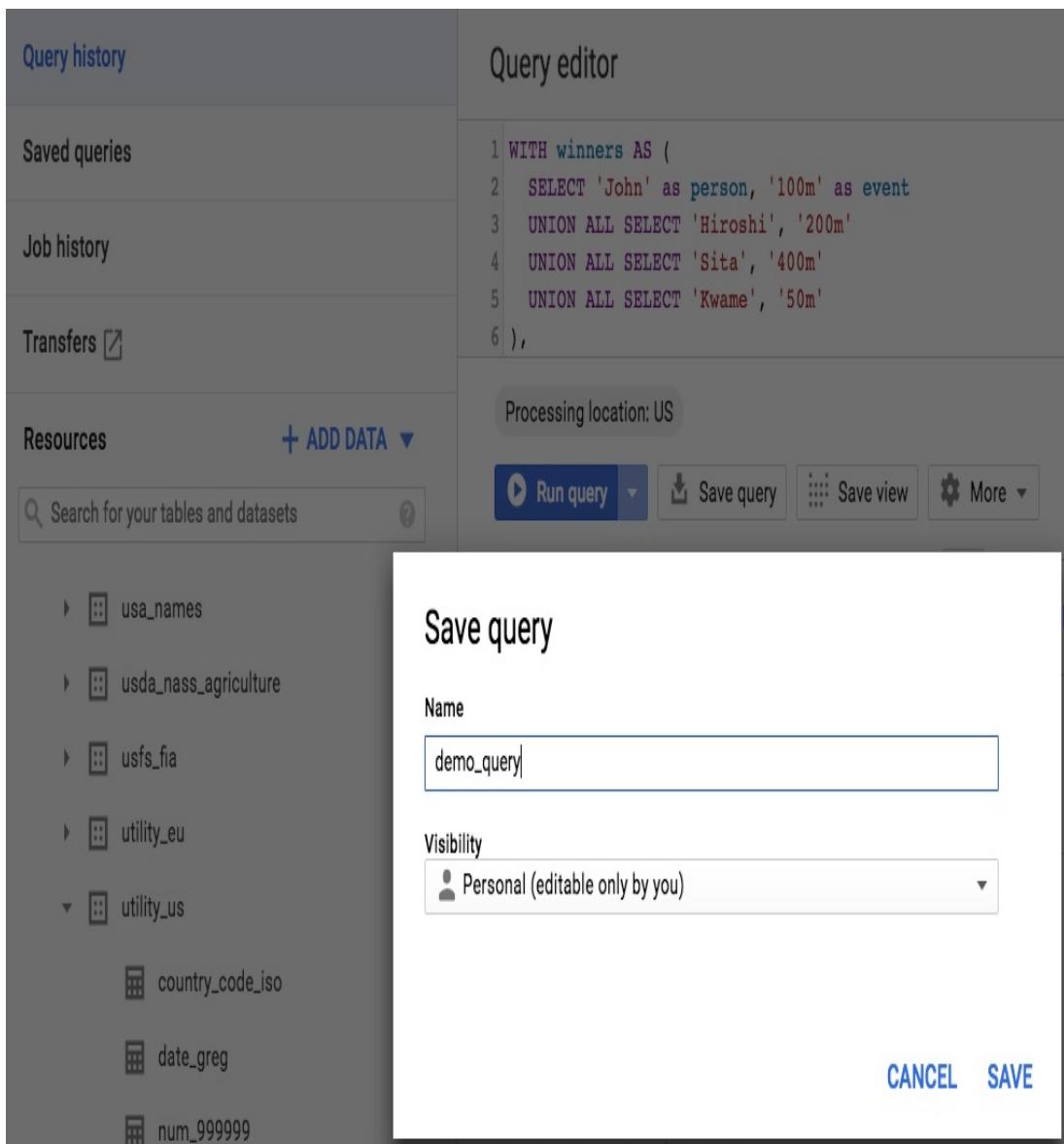


Figure 2-2. Save a query by clicking on the “Save query” button in the web user interface.

You can also choose to make the saved query shareable, in which case anyone who has the URL will get dropped into a page with the query text prepopulated.

When you share a query, all that you share is the text of the query, but not access to any data. Dataset permissions in order to execute the query have to be provided independently using the Identity and Access Management controls (these will be discussed in Chapter X). Also, unlike most

BigQuery features, the ability to save and share queries is available only from the web user interface. There is, at least at the time of writing, no REST API or client library available for this.

The list of saved queries is available from the user interface. You can turn off link sharing at any time to make the query text private again.

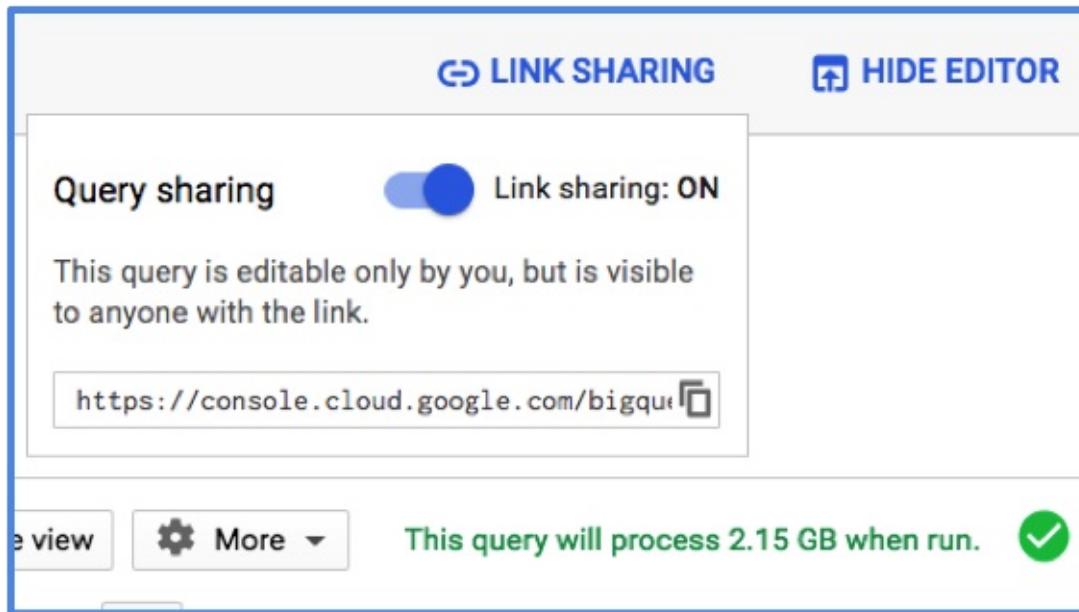


Figure 2-3. You can turn off link sharing at any time to make the query text private again.

Views vs. shared queries

One of the advantages of sharing a query link (as opposed to simply copying the text of the query into an email) is that you can continue to edit the query so that your collaborators always get the latest version of the query. This is useful when the envisioned use-case is that they might wish to examine the query text, modify it, and then run the query.

The query text does not have to be syntactically correct -- you can save and share incomplete query text or template queries that need to be completed by the end-user. These capabilities are helpful when

collaborating with colleagues.

If you expect the person to whom you are sending the query to subset or query the results of your query, it is better to save your query as a view and send them a link to the view. Another advantage of views over shared queries is that views are placed into datasets and offer fine-grained IAM controls. Views can also be materialized.

We will look at views in Chapter X.

Chapter Summary

BigQuery supports SQL 2011 -- selecting records (SELECT), aliasing column names (AS), filtering (WHERE), subqueries (parentheses and WITH), sorting (ORDER), aggregation (GROUP, AVG, COUNT, MIN, MAX, etc.), filtering grouped items (HAVING), filtering unique values (DISTINCT) and joining (INNER/CROSS/OUTER JOIN). There is also support for arrays (ARRAY_AGG, UNNEST) and structs (STRUCT). The history of queries (the text of the query, not the results) submitted to the service is available to the user who submitted the query, and to project administrators. It is possible to share query text through a link.

-
- 1 See <https://www.iso.org/standard/53681.html>
 - 2 This is a data format that is very popular within Google because it provides efficient storage in a programming-language-neutral way. It is now open-source -- see <https://developers.google.com/protocol-buffers/>
 - 3 For more details on Dremel, see <https://ai.google/research/pubs/pub36632>
 - 4 The “person” in this case is one of the members of the Google Cloud Platform public datasets team. See <https://cloud.google.com/public-datasets/> for what else is available.
 - 5 We believe all mentions of price to be correct as of the writing of this book, but please do refer to the relevant policy and pricing sheets (See <https://cloud.google.com/bigquery/pricing>) as these are subject to change.
 - 6 For an entertaining data-driven examination of the correlation between project success and the use of leading commas, see <https://hackernoon.com/winning-arguments-with-data-leading-with-commas-in-sql-672b3b81eac9/>
 - 7 You can also use ORDINAL(1) to work with 1-based indexing. We will look at arrays in more detail in Chapter X.

Chapter 3. Data Types, Functions and Operators

In the bike rental queries in the previous chapters, when we divided the trip-duration by 60, we were able to do so because trip-duration was a numeric type. Trying to divide the gender by 60 would not have worked because gender is a string. The functions and operations you have at your disposal may be restricted based on the type of data they are being applied to.

BigQuery supports several data types to store numeric, string, time, geographic, structured and semi-structured data:

INT64, the only integer type, which can represent numbers ranging from approximately 10-19 to 1019. For real-valued numbers, use FLOAT64 and for booleans, use BOOL.

NUMERIC, which offers 38 digits of precision and 9 decimal digits of scale and is suitable for exact calculations such as in finance.

STRING is a first-class type and represents variable-length sequences of Unicode characters. BYTES are variable-length sequences of characters (not Unicode).

TIMESTAMP, which represents an absolute point in time and DATETIME which represents a calendar date and time. DATE and TIME are also available separately.

GEOGRAPHY, to represent points, lines, and polygons on the surface of the earth.

STRUCT and ARRAY as described in the previous section.

Numeric types and functions

There is only one integer type (INT64) and only one floating point type (FLOAT64). Both of these types support the typical arithmetic operations (+, -, /, * for add, subtract, divide and multiply). Thus, we can find the fraction of rides that are one-way by simply dividing one column by the other:

WITH example AS (

```
SELECT 'Sat' AS day, 1451 AS numrides, 1018 AS oneways
```

```
UNION ALL SELECT 'Sun', 2376, 936
```

```
)
```

```
SELECT *, (oneways/numrides) AS frac_oneway from example
```

This yields:

Row	day	numrides	oneways	frac_oneway
1	Sat	1451	1018	0.7015851137146796
2	Sun	2376	936	0.3939393939393939

Besides the arithmetic operators, bitwise operations (<< and >> for shifting, & and | for bitwise AND and OR, etc.) are also supported on integer types.

To operate on data types, we can use functions. Functions perform operations on the values that are input to them. As with other programming languages, functions in SQL encapsulate reusable logic and abstract away the complexity of their implementation. There are several types of functions:

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

types.

MATHEMATICAL FUNCTIONS

If we wanted to round off the end-result of the query that computed the fraction of rides that were one-way, we would have used one of the many built-in mathematical functions¹ that work on integer and floating point types:

WITH example AS (

```
SELECT 'Sat' AS day, 1451 AS numrides, 1018 AS oneways
```

```
UNION ALL SELECT 'Sun', 2376, 936
```

```
)
```

```
SELECT *, ROUND(oneways/numrides, 2) AS frac_oneway from  
example
```

which yields:

Row	day	numrides	oneways	frac_oneway
1	Sat	1451	1018	0.7
2	Sun	2376	936	0.39

STANDARD-COMPLIANT FLOATING POINT DIVISION

The division operator fails if the denominator is zero or if the result overflows. Rather than protect the division by checking for zero values beforehand, it is better to use a special function for division whenever, as is the case above, the denominator could be zero. A better form of the query above is, therefore:

WITH example AS (

SELECT 'Sat' AS day, 1451 AS numrides, 1018 AS oneways

UNION ALL SELECT 'Sun', 2376, 936

UNION ALL SELECT 'Wed', 0, 0

)

SELECT

*, ROUND(IEEE_Divide(oneways, numrides), 2)

AS frac_oneway from example

The 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 (NaN) when a division by zero is attempted.

Also try the query above using the standard division operator and using SAFE_DIVIDE (discussed below).² Recall that, for your copy-pasting convenience, all the queries in this book are available in the GitHub repository corresponding to the book.³

SAFE FUNCTIONS

You can make any scalar function return NULL instead of raising an error by prefixing it with SAFE. For example, the following query will raise an error because the logarithm of a negative number is undefined:

```
SELECT LOG(10, -3), LOG(10, 3)
```

However, by prefixing the LOG with SAFE:

```
SELECT SAFE.LOG(10, -3), SAFE.LOG(10, 3)
```

you will get NULL for the result of LOG(10, -3):

Row	f0_	f1_
1	null	2.095903274289385

The SAFE prefix works for mathematical functions, string functions (for example, the SUBSTR function would normally raise an error if the starting index is negative, but return NULL if invoked as SAFE.SUBSTR), and time functions. It is, however, restricted to scalar functions and will not work for aggregate functions, analytic functions, or for user-defined functions.

COMPARISONS

Comparisons are carried out using operators. The operators <, <=, >, >= and != (or <>) are used to obtain the results of comparison. NULL, followed by NaN, is assumed to be smaller than valid numbers (including -inf) for the purposes of ordering. However, comparisons with NaN always return false and comparisons with NULL always return NULL. This can lead to seemingly paradoxical results:

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

returns:

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

because the WHERE clause returns only those rows for which the result is true and when NULL is compared to 2000, the result is NULL and not

true.

Note that the operators & and not | exist in BigQuery, but are used only for bitwise operations. The ! symbol as in != means NOT, but does not work stand-alone -- you can not, as in other languages, say !gender to compute the logical negative of gender. An alternate way to specify not-equals is to write <>, but be consistent on whether you use != or <>.

PRECISE DECIMAL CALCULATIONS WITH NUMERIC

INT64 and FLOAT64 are designed to be flexible and fast, but are limited by the fact they are stored in a base-2 (0s and 1s) form in a 64-bit area of computer memory while being used for calculations. This is a tradeoff well-worth making in most applications, but financial and accounting applications often require exact calculations for numbers represented in decimal (base-10).

The NUMERIC data type in BigQuery provides 38 digits to represent numbers, with 9 of those digits appearing after the decimal point. It uses 16 bytes for storage and can represent decimal fractions exactly, thus making it suitable for financial calculations.

For example, imagine you needed to compute the sum of three payments. You'd want the results to be exact. When using FLOAT64 values, however, the tiny differences between how the number is represented in memory and how the number is represented in decimals can add up:

WITH example AS (

```
SELECT 1.23 AS payment
```

UNION ALL SELECT 7.89

UNION ALL SELECT 12.43

)

SELECT

SUM(payment) AS total_paid,

AVG(payment) AS average_paid

FROM example

I got:

Row	total_paid	average_paid
1	21.549999999999997	7.18333333333334

In financial and accounting applications, these imprecisions can add up and make balancing the books tricky.

If we change the data type of payment to be NUMERIC:

WITH example AS (

SELECT NUMERIC '1.23' AS payment

UNION ALL SELECT NUMERIC '7.89'

UNION ALL SELECT NUMERIC '12.43'

)

SELECT

SUM(payment) AS total_paid,

AVG(payment) AS average_paid

FROM example

the problem goes away. The sum of the payments is now precise (the average can not be represented precisely even in NUMERIC because it is a repeating decimal):

Row	total_paid	average_paid
1	21.55	7.183333333

Note that NUMERIC types have to be directly ingested into BigQuery as strings (NUMERIC '1.23'). Otherwise, the floating point representation will obviate any of the precision gains to be had.

Working with BOOL

Boolean variables are those that can be either True or False. Because SQL is case-insensitive, TRUE, true, etc. also work.

LOGICAL OPERATIONS

Recall from the section on filtering within the WHERE clause that the WHERE clause can include boolean expressions that include AND, OR and NOT as well as parentheses to control the order of execution. We used this query to illustrate these options:

```
SELECT  
gender, tripduration  
FROM  
`bigquery-public-data`.new_york_citibike.citibike_trips  
WHERE (tripduration < 600 AND gender = 'female') OR gender = 'male'
```

You could use comparison operators with boolean variables as in:

```
WITH example AS (
```

```
SELECT NULL AS is_vowel, NULL as letter, -1 as position
```

```
UNION ALL SELECT true, 'a', 1
```

```
UNION ALL SELECT false, 'b', 2
```

```
UNION ALL SELECT false, 'c', 3
```

```
)
```

```
SELECT * from example WHERE is_vowel != false
```

This yields:

Row	is_vowel	letter	position
1	true	a	1

However, it is often simpler to use the IS operator when comparing against built-in constants. For example:

```
WITH example AS (
```

```
SELECT NULL AS is_vowel, NULL as letter, -1 as position
```

```
UNION ALL SELECT true, 'a', 1
```

```
UNION ALL SELECT false, 'b', 2
```

```
UNION ALL SELECT false, 'c', 3
```

```
)
```

```
SELECT * from example WHERE is_vowel IS NOT false
```

This yields:

Row	is_vowel	letter	position
1	null	null	-1
2	true	a	1

Note that the two queries yield different results. The comparators ($=$, \neq , $<$, etc.) return `NULL` for comparisons against `NULL` while the `IS` operator doesn't.

`<Tip>`

`NULL`s typically represent missing values or values that were not collected. They have no value and are not zero, empty strings, or blanks. If your dataset has `NULL`s, you must tread carefully since comparisons with `NULL` always return `NULL` and so the `WHERE` clause will filter out `NULL` values. Use the `IS` operator to check where a value is `NULL`.

`</tip>`

It is simpler and more readable to use boolean variables directly:

WITH example AS (

SELECT NULL AS is_vowel, NULL as letter, -1 as position

UNION ALL SELECT true, 'a', 1

UNION ALL SELECT false, 'b', 2

UNION ALL SELECT false, 'c', 3

)

SELECT * from example WHERE is_vowel

The result here is like is_vowel IS TRUE:

Row	is_vowel	letter	position
1	true	a	1

Of course, such readability depends on naming the boolean variables well!

CONDITIONAL EXPRESSIONS

It is not just in the WHERE clause that booleans are useful. It is possible to simplify many queries by using conditional expressions in the SELECT. For example, let's say we need to compute the sales prices of each item in our catalog based on the desired markup and tax rate corresponding to the item. If our catalog is missing values for some of the necessary information, we might want to impute a default markup or default tax rate. We can achieve this with the IF function:

```

WITH catalog AS (
    SELECT 30.0 AS costPrice, 0.15 AS markup, 0.1 AS taxRate
    UNION ALL SELECT NULL, 0.21, 0.15
    UNION ALL SELECT 30.0, NULL, 0.09
    UNION ALL SELECT 30.0, 0.30, NULL
    UNION ALL SELECT 30.0, NULL, NULL
)
SELECT
    *, ROUND(
        costPrice *
        IF(markup IS NULL, 1.05, 1+markup) *
        IF(taxRate IS NULL, 1.10, 1+taxRate)
        , 2) AS salesPrice
FROM catalog

```

This yields a valid salesPrice for all items except those for which we don't know the cost:

Row	costPrice	markup	taxRate	salesPrice
1	30.0	0.15	0.1	37.95

	costPrice	markup	taxRate	total
2	null	0.21	0.15	null
3	30.0	null	0.09	34.34
4	30.0	0.3	null	42.9
5	30.0	null	null	34.65

The way that the IF function works is that the first parameter is the condition to be evaluated. If the condition is true, then the second parameter is used, else the third parameter is used. Because it occurs in the SELECT, it is carried out row-by-row.

CLEANER NULL-HANDLING WITH COALESCE

What if we wish to do the imputation if a single value is missing, but not if more than one value is missing? In other words, if we have no tax rate, we are willing to impute a 10% taxrate, but not if we also don't know the markup on the item.

A convenient way to keep evaluating expressions until we get to a non-NULL value is to use COALESCE:

WITH catalog AS (

SELECT 30.0 AS costPrice, 0.15 AS markup, 0.1 AS taxRate

UNION ALL SELECT NULL, 0.21, 0.15

UNION ALL SELECT 30.0, NULL, 0.09

UNION ALL SELECT 30.0, 0.30, NULL

UNION ALL SELECT 30.0, NULL, NULL

)

SELECT

*, ROUND(COALESCE(
costPrice * (1+markup) * (1+taxrate),

costPrice * 1.05 * (1+taxrate),

costPrice * (1+markup) * 1.10,

NULL

),2) AS salesPrice

FROM catalog

This yields (only the last row is different from the previous computation):

Row	costPrice	markup	taxRate	salesPrice
1	30.0	0.15	0.1	37.95
2	null	0.21	0.15	null
3	30.0	null	0.09	34.34
4	30.0	0.3	null	42.9
5	30.0	null	null	null

The COALESCE short-circuits, i.e., later expressions are not evaluated once a non-null result is obtained. The final NULL in the COALESCE is not required, but makes the intent clearer.

BigQuery supports the IFNULL function as a simplification of

COALESCE when you have only two inputs. IFNULL(a, b) is the same as COALESCE(a, b) and yields b if a is NULL. In other words, IFNULL(a, b) is the same as IF(a IS NULL, b, a).

The very first query in this section on conditional expressions could have been simplified as:

```
SELECT
```

```
*, ROUND(
```

```
costPrice *
```

```
(1 + IFNULL(markup, 0.05)) *
```

```
(1 + IFNULL(taxrate, 0.10))
```

```
, 2) AS salesPrice
```

```
FROM catalog
```

CASTING AND COERCION

Consider this example dataset where the number of hours worked by an employee is stored as a string in order to accommodate reasons for a leave of absence (this is a bad schema design, but bear with us):

```
WITH example AS (
```

```
SELECT 'John' as employee, 'Paternity Leave' AS hours_worked
```

```
UNION ALL SELECT 'Janaki', '35'
```

```
UNION ALL SELECT 'Jian', 'Vacation'
```

```
UNION ALL SELECT 'Jose', '40'
```

```
)
```

and let's say that we want to find the total number of hours worked.

This won't work because the hours_worked is a string, not a numeric type:

```
WITH example AS (
```

```
SELECT 'John' as employee, 'Paternity Leave' AS hours_worked
```

```
UNION ALL SELECT 'Janaki', '35'
```

```
UNION ALL SELECT 'Jian', 'Vacation'
```

```
UNION ALL SELECT 'Jose', '40'
```

```
)
```

```
SELECT SUM(hours_worked) from example
```

We will have to cast it (i.e., explicitly convert it) to an INT64 before doing any aggregation. Explicit conversion is called casting, and requires the explicit use of the CAST() function. If casting fails, BigQuery raises an error. To have it return NULL instead, use SAFE_CAST. For example, the following raises an error:

```
SELECT CAST("true" AS bool), CAST("invalid" AS bool)
```

but:

```
SELECT CAST("true" AS bool), SAFE_CAST("invalid" AS bool)
```

returns:

Row	f0_	f1_
1	true	null

Implicit conversion is called coercion and happens automatically when a data type is used in a situation where another data type is required. For example, when we use an INT64 in a situation where a FLOAT64 is needed, the integer will be coerced into a floating point number. The only coercions done by BigQuery are to convert INT64 to FLOAT64 and NUMERIC, and NUMERIC to FLOAT64. Every other conversion is explicit and requires a CAST.

In our problem of the total number of hours worked, not all the hours_worked strings can be converted to integers, so we should use a SAFE_CAST:

WITH example AS (

```
SELECT 'John' as employee, 'Paternity Leave' AS hours_worked
```

```
UNION ALL SELECT 'Janaki', '35'
```

```
UNION ALL SELECT 'Jian', 'Vacation'
```

```
UNION ALL SELECT 'Jose', '40'
```

)

SELECT SUM(SAFE_CAST(hours_worked AS INT64)) from example

This yields:

Row	f0_
1	75

Had it simply been a schema problem, and all the rows contained numbers but were stored as strings, we could have used a simple cast:

WITH example AS (

SELECT 'John' as employee, '0' AS hours_worked

UNION ALL SELECT 'Janaki', '35'

UNION ALL SELECT 'Jian', '0'

UNION ALL SELECT 'Jose', '40'

)

SELECT SUM(CAST(hours_worked AS INT64)) from example

COUNTIF TO AVOID CASTING BOOLEANS

Consider this example dataset:

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 * from example
```

which yields:

Row	is_vowel	letter	position
1	true	a	1
2	false	b	2
3	false	c	3

and let's say that we want to find the total number of vowels.

While we might be tempted to simply do:

```
SELECT SUM(is_vowel) as num_vowels from example
```

This won't work (try it!) because SUM, AVG, etc. are not defined on booleans. We could cast the booleans to an INT64 before doing the aggregation:

```
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
```

which yields:

Row	num_vowels
1	1

However, you should try to avoid CASTing as much as possible. In this case, a cleaner approach is to use the IF statement on the booleans:

```
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(IF(is_vowel, 1, 0)) as num_vowels from example
```

An even cleaner approach is to use COUNTIF:

```
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 COUNTIF(is_vowel) as num_vowels from example
```

String functions

String manipulation is a common requirement for data wrangling, so BigQuery provides a library of built-in string functions.⁴ For example:

```
WITH example AS (
```

```
SELECT * from unnest([
```

```
'Seattle', 'New York', 'Singapore'
```

```
]) AS city
```

```
)
```

```
SELECT
```

```
city
```

```
, LENGTH(city) AS len
```

```
, LOWER(city) AS lower
```

```
, STRPOS(city, 'or') AS orpos
```

FROM example

compute the length of the string, lower-case the string and find the location of a substring in the “city” column, yielding:

Row	city	len	lower	orpos
1	Seattle	7	seattle	0
2	New York	8	new york	6
3	Singapore	9	singapore	7

The substring ‘or’ occurs in “New York” and in “Singapore”, but not in “Seattle”.

Two particularly useful functions for string manipulation are SUBSTR and CONCAT. SUBSTR extracts a substring and CONCAT concatenates the input values. The following query finds the position of the @ symbol in an email address, extracts the user name, and concatenates the city they live in:

WITH example AS (

```
SELECT 'armin@abc.com' AS email, 'Anapolis, MD' as city
```

```
UNION ALL SELECT 'boyan@bca.com', 'Boulder, CA'
```

```
UNION ALL SELECT 'carrie@cab.com', 'Chicago, IL'
```

)

```
SELECT
```

```
CONCAT(  
    SUBSTR(email, 1, STRPOS(email, '@') - 1), -- username  
    ' from ', city) AS callers
```

FROM example

which yields:

Row	callers
1	armin from Anapolis, MD
2	boyan from Boulder, CA
3	carrie from Chicago, IL

INTERNATIONALIZATION

Strings in BigQuery are Unicode, so avoid assumptions that rely on English. For example, the “upper” case is a no-op in Japanese and the default UTF-8 encoding that is carried out by the cast as bytes is insufficient for languages such as Tamil. Thus,

WITH example AS (

```
SELECT * from unnest([  
    'Seattle', 'New York', 'சிங்கப்பூர்', '東京'  
]) AS city  
)
```

```
SELECT
```

```
city
```

```
, UPPER(city) AS allcaps
```

```
, CAST(city AS BYTES) as bytes
```

```
FROM example
```

yields:

R _o w	city	allcaps	bytes
1	Seattle	SEATTLE	U2VhdHRsZQ==
2	New York	NEW YORK	TmV3IFlvcms=
3	சிங்கப்பூர்	சிங்கப்பூர்	4K6a4K6/4K6Z4K+N4K6V4K6q4K+N4K6q4K+C4K6w4 K+N
4	東京	東京	5p2x5Lqs

BigQuery supports three different ways to represent strings: as an array of Unicode characters, as an array of bytes, and as an array of Unicode code points (INT64):

```
WITH example AS (
```

```
SELECT * from unnest([
```

```
'Seattle', 'New York', 'சிங்கப்பூர்', '東京'
```

```
]) AS city
```

)

SELECT

city

, CHAR_LENGTH(city) as char_len

, TO_CODE_POINTS(city)[OFFSET(1)] as first_code_point

, ARRAY_LENGTH(TO_CODE_POINTS(city)) as num_code_points

, CAST (city AS BYTES) as bytes

, BYTE_LENGTH(city) as byte_len

FROM example

Note the difference between the results for CHAR_LENGTH and BYTE_LENGTH on the same strings, and how the number of code points is the same as the number of characters:

R o w	city	char _len	first_code _point	num_code _points	bytes	byte _len
1	Seattl e	7	101	7	U2VhdHRsZQ==	7
2	New York	8	101	8	TmV3IFlv cms=	8
3	சிங்க பூர்	11	3007	11	4K6a4K6/4K6Z4K+N4K6V4K6q4K+N4 K6q4K+C4K6w4K+N	33
4	東京	2	20140	2	5p2x5Lqs	6

Because of these differences, recognize which columns might contain text

in different languages, and be aware of language differences when using string manipulation functions.

PRINTING AND PARSING

You can simply CAST a string as an INT64 or FLOAT64 in order to parse it, but customizing the string representation will require the use of FORMAT:

SELECT

CAST(42 AS STRING)

, CAST('42' AS INT64)

, FORMAT('%03d', 42)

, FORMAT('%5.3f', 32.457842)

, FORMAT('%5.3f', 32.4)

, FORMAT('**%s**', 'H')

, FORMAT('%s-%03d', 'Agent', 7)

yields:

Row	f0_	f1_	f2_	f3_	f4_	f5_	f6_
1	42	42	042	32.458	32.400	**H**	Agent-007

FORMAT works similarly to C's printf, and accepts the same format specifiers.⁵ A few of the more useful specifiers are demonstrated above.

Although FORMAT also accepts dates and timestamps, it is better to use FORMAT_DATE and FORMAT_TIMESTAMP so that the display formats can be locale-aware.

STRING MANIPULATION FUNCTIONS

Manipulating strings is a common-enough need in ETL pipelines that these BigQuery convenience functions are worth having on speed dial:

SELECT

ENDS_WITH('Hello', 'o') -- true

, ENDS_WITH('Hello', 'h') -- false

, STARTS_WITH('Hello', 'h') -- false

, STRPOS('Hello', 'e') -- 2

, STRPOS('Hello', 'f') -- 0 for not-found

, SUBSTR('Hello', 2, 4) -- 1-based

, CONCAT('Hello', 'World')

The results of the above query are:

Row	f0_	f1_	f2_	f3_	f4_	f5_	f6_
1	true	false	false	2	0	ello	HelloWorld

Note how SUBSTR() behaves. The first parameter is the starting offset (i.e., it is one-based) and the second parameter is the desired number of

characters in the substring.

TRANSFORMATION FUNCTIONS

Another set of functions worth getting familiar with are those that allow you to manipulate the string:

SELECT

LPAD('Hello', 10, '*') -- left pad with *

, RPAD('Hello', 10, '*') -- right pad

, LPAD('Hello', 10) -- left pad with spaces

, LTRIM(' Hello ') -- trim whitespace on left

, RTRIM(' Hello ') -- trim whitespace on right

, TRIM (' Hello ') -- trim whitespace both ends

, TRIM ('***Hello***', '*') -- trim * both ends

, REVERSE('Hello') -- reverse the string

The result is:

Row	f0_	f1_	f2_	f3_	f4_	f5_	f6_	f7_
1	*****Hello	Hello*****	Hello	Hello	Hello	Hello	Hello	olleH

REGULAR EXPRESSIONS

Regular expressions provide much more powerful semantics than the

convenience functions. For example, STRPOS, etc. can find only specific characters while REGEXP_CONTAINS can be used for more powerful searches.

For example, to determine whether a column contains a US zipcode (the short form of which is a 5-digit number and the long form of which has an additional 4 numbers separated by either a hyphen or a space):

```
SELECT
```

```
column
```

```
, REGEXP_CONTAINS(column, r"\d{5}(?:[-\s]\d{4})?") has_zipcode
```

```
, REGEXP_CONTAINS(column, r'^\d{5}(?:[-\s]\d{4})?$') is_zipcode
```

```
, REGEXP_EXTRACT(column, r"\d{5}(?:[-\s]\d{4})?") the_zipcode
```

```
, REGEXP_EXTRACT_ALL(column, r"\d{5}(?:[-\s]\d{4})?")
```

```
all_zipcodes
```

```
, REGEXP_REPLACE(column, r"\d{5}(?:[-\s]\d{4})?", '*****') masked
```

```
FROM (
```

```
SELECT * from unnest([
```

```
'12345', '1234', '12345-9876',
```

```
'abc 12345 def', 'abcde-fghi',
```

```
'12345 ab 34567', '12345 9876'
```

] AS column

)

This yields:

Row	column	has_zipcod	is_zipcod	the_zipcod	all_zipcode	masked
w	e	e	e	s		
1	12345	true	true	12345	12345	*****
2	1234	false	false	null		1234
3	12345-9876	true	true	12345- 9876	12345-9876	*****
4	abc 12345 def	true	false	12345	12345	abc ***** def
5	abcde-fghi	false	false	null		abcde-fghi
6	12345 ab 34567	true	false	12345	12345	***** ab *****
					34567	
7	12345 9876	true	true	12345 9876	12345 9876	*****

A few things to note:

The regular expression \d{5} matches any string consisting of 5 decimal numbers.

The parentheses enclosing the second part of the expression looks for an optional (note the ? at the end of the parentheses) group (?:) of four decimal numbers (\d{4}) which is separated from the first five numbers by either a hyphen or by a space (\s)

The presence of \d, \s, etc. in the string could cause problems, so we prefix it with an r (for raw), which makes it a string literal.

The second expression illustrates how to find an exact match -- simply insist that the string in question has to start (^) and end (\$) with the specified string.

To extract the part of the string matched by the regular expression, use REGEXP_EXTRACT. This returns null if the expression is not matched and only the first match if there are multiple matches.

REGEXP_EXTRACT_ALL returns all the matches. If there is no match, it returns an empty array.

REGEXP_REPLACE replaces every match by the replacement string.

The regular expression support in BigQuery follows that of Google's open-source RE2 library.⁶ See <https://github.com/google/re2/wiki/Syntax> for the syntax accepted by this library. Regular expressions maybe cryptic, but they are a rich topic that is well-worth mastering.⁷

SUMMARY OF STRING FUNCTIONS

Because strings are so common in data analysis, it is worth learning the broad contours of what functions are available. You can always refer to the BigQuery documentation for the exact syntax. Briefly, string functions fall into these categories:

Categorie	Functions	Notes
Representation	CHAR_LENGTH, BYTE_LENGTH, TO_CODE_POINTS, CODE_POINTS_TO_STRING, SAFE_CONVERT_BYTES_TO_STRING, TO_HEX, TO_BASE32, TO_NATURAL_BYTES	Normalize allows, for example, different Unicode space characters to be made equivalent.

	<code>TO_BASE64, FROM_HEX, FROM_BASE32, FROM_BASE64, NORMALIZE</code>	
Printing and parsing	<code>FORMAT, REPEAT, SPLIT</code>	The syntax of FORMAT is similar to C's printf: format("%03d", 12) yields 012. For locale-aware conversions, use <code>FORMAT_DATE</code> etc.
Convenience	<code>ENDS_WITH, LENGTH, STARTS_WITH, STRPOS, SUBSTR, CONCAT</code>	The LENGTH function is equivalent to CHAR_LENGTH for Strings and to BYTE_LENGTH for Bytes
Transformations	<code>LPAD, LOWER, LTRIM, REPLACE, REVERSE, RPAD, RTRIM, TRIM, UPPER</code>	The default trim characters are Unicode whitespace, but it is possible to specify a different set of trim characters.
Regular expressions	<code>REGEXP_CONTAINS, REGEXP_EXTRACT, REGEXP_EXTRACT_ALL, REGEXP_REPLACE</code>	See https://github.com/google/re2/wiki/Syntax for the syntax accepted by BigQuery.

Working with TIMESTAMP

A timestamp represents an absolute point in time regardless of location. Thus, a timestamp of 2017-09-27 12:30:00.45 (Sep 9, 2017 at 12:30 UTC) represents the same time as 2017-09-27 13:30:00.45+1:00 (1:30pm at a timezone that is an hour behind):

```
SELECT t1, t2, TIMESTAMP_DIFF(t1, t2, MICROSECOND)
```

```
FROM (SELECT
```

```
    TIMESTAMP "2017-09-27 12:30:00.45" AS t1,
```

```
    TIMESTAMP "2017-09-27 13:30:00.45+1" AS t2
```

```
)
```

returns

Row	t1	t2	f0_
1	2017-09-27 12:30:00.450 UTC	2017-09-27 12:30:00.450 UTC	0

PARSING AND FORMATTING TIMESTAMPS

BigQuery is somewhat forgiving when it comes to parsing the timestamp. The date and time parts of this string representation can be separated either by a T or by a space in accordance with ISO 8601⁸. Similarly, the month, day, hour, etc. may or may not have leading zeros. However, best practice is to use the canonical representation shown in the previous paragraph. As that string representation would indicate, this timestamp can only represent four-digit years; years before the common era can not be represented using TIMESTAMP.

You can use PARSE_TIMESTAMP to parse a string that is not in the canonical format:

SELECT

fmt, input, zone

, PARSE_TIMESTAMP(fmt, input, zone) AS ts

FROM (

SELECT '%Y%m%d-%H%M%S' AS fmt, '20181118-220800' AS input,
'+0' as zone

UNION ALL SELECT '%c', 'Sat Nov 24 21:26:00 2018',
'America/Los_Angeles'

```
UNION ALL SELECT '%x %X', '11/18/18 22:08:00', 'UTC'
```

```
)
```

This yields:

Ro w	fmt	input	zone	ts
1	%Y%m%d% H%M%S	20181118-220800	+0	2018-11-18 22:08:00 UTC
2	%c	Sat Nov 24 21:26:00 2018	America/Los_An geles	2018-11-25 05:26:00 UTC
3	%x %X	11/18/18 22:08:00	UTC	2018-11-18 22:08:00 UTC

The first example uses format specifiers for the year, month, day, etc. to create a timestamp from the provided string. The second and third examples use pre-existing specifiers for commonly encountered date-time formats. For the full list of specifiers, please consult the documentation.⁹

Conversely, you can use FORMAT_TIMESTAMP to print out a timestamp in any desired format:

```
SELECT
```

```
ts, fmt
```

```
, FORMAT_TIMESTAMP(fmt, ts, '+6') AS ts_output
```

```
FROM (
```

```
SELECT CURRENT_TIMESTAMP() AS ts, '%Y%m%d-%H%M%S' AS  
fmt
```

```

UNION ALL SELECT CURRENT_TIMESTAMP() AS ts, '%c' AS fmt
UNION ALL SELECT CURRENT_TIMESTAMP() AS ts, '%x %X' AS
fmt
)

```

which results in:

Ro w	ts	fmt	ts_output
1	2018-11-25 05:42:13.939840 UTC	%Y%m%d-% %H%M%S	20181125-114213
2	2018-11-25 05:42:13.939840 UTC	%c	Sun Nov 25 11:42:13 2018
3	2018-11-25 05:42:13.939840 UTC	%x %X	11/25/18 11:42:13

The example above uses the function CURRENT_TIMESTAMP() to retrieve the system time at the time the query is executed. In both PARSE_TIMESTAMP and FORMAT_TIMESTAMP, the timezone is optional; if omitted, the timezone is assumed to be UTC.

EXTRACTING CALENDAR PARTS

Given a timestamp, it is possible to extract information about the Gregorian calendar corresponding to the timestamp. For example, we can extract information about Armistice Day¹⁰ using:

SELECT

ts

```

, FORMAT_TIMESTAMP('%c', ts) AS repr
, EXTRACT(DAYOFWEEK FROM ts) AS dayofweek
, EXTRACT(YEAR FROM ts) AS year
, EXTRACT(WEEK FROM ts) AS weekno
FROM (
SELECT PARSE_TIMESTAMP('%Y%m%d-%H%M%S', '19181111-
054500')11 AS ts
)

```

This results in:

Row	ts	repr	dayofweek	year	weekno
1	1918-11-11 05:45:00 UTC	Mon Nov 11 05:45:00 1918	2	1918	45

The week is assumed to start on Sunday, and days prior to the first Sunday of the year are assigned to week 0. This is not internationally safe. Hence, if in a country (such as Israel) where the week starts on Saturday, it is possible to specify a different day for the start of the week:

`EXTRACT(WEEK('SATURDAY') FROM ts)`

The number of seconds from the Unix epoch (Jan 1, 1970) is not available through EXTRACT. Instead, special functions exist to convert to/from the Unix epoch:

```

SELECT

UNIX_MILLIS(TIMESTAMP "2018-11-25 22:30:00 UTC")

, UNIX_MILLIS(TIMESTAMP "1918-11-11 22:30:00 UTC") --invalid

, TIMESTAMP_MILLIS(1543185000000)

```

This yields:

Row	f0_	f1_	f2_
1	1543185000000	-1613784600000	2018-11-25 22:30:00 UTC

Note that the second one overflows and yields a negative number, but no error is raised.

ARITHMETIC WITH TIMESTAMPS

It is possible to add or subtract time durations from timestamps. It is also possible to find the time difference between two timestamps. In all these functions, you need to specify the units that the durations are expressed in:

```

SELECT

EXTRACT(TIME FROM TIMESTAMP_ADD(t1, INTERVAL 1
HOUR)) AS plus_1h

, EXTRACT(TIME FROM TIMESTAMP_SUB(t1, INTERVAL 10
MINUTE)) AS minus_10min

, TIMESTAMP_DIFF(CURRENT_TIMESTAMP(),

```

```
TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 1  
MINUTE),
```

```
SECOND) AS plus_1min
```

```
, TIMESTAMP_DIFF(CURRENT_TIMESTAMP(),
```

```
TIMESTAMP_ADD(CURRENT_TIMESTAMP(), INTERVAL 1  
MINUTE),
```

```
SECOND) AS minus_1min
```

```
FROM (SELECT
```

```
TIMESTAMP “2017-09-27 12:30:00.45” AS t1
```

```
)
```

This returns the timestamps an hour from now, 10 minutes ago as well as the time difference in seconds corresponding to 1 minute from now and 1 minute earlier:

Row	plus_1h	minus_10min	plus_1min	minus_1min
1	13:30:00.450000	12:20:00.450000	60	-60

DATE, TIME, AND DATETIME

BigQuery has three other functions for representing time: DATE, TIME, and DATETIME. DATE is useful when you are tracking only the day in which something happens, and any more precision is unnecessary. TIME is useful to represent the time of day that things happen, and to perform mathematical operations with those times. With TIME, you can answer

questions like “What time will it be 8 hours from the starting time?” DATETIME is a TIMESTAMP rendered in a specific time zone, so it is useful when you have an unambiguous time zone in which an event occurred, and don’t need to do time zone conversions.

Counterparts to most of the TIMESTAMP functions are available for DATETIME. Thus, you can call DATETIME_ADD, DATETIME_SIB, DATETIME_DIFF as well as PARSE_DATETIME and FORMAT_DATETIME. You can also EXTRACT calendar parts from a DATETIME. The two types are quite interoperable: it is possible to extract a datetime from a timestamp and cast a datetime to a timestamp:

```
SELECT
```

```
EXTRACT(DATETIME FROM CURRENT_TIMESTAMP()) as dt
```

```
, CAST(CURRENT_DATETIME() AS TIMESTAMP) as ts
```

This yields:

Row	dt	ts
1	2018-11-25T07:03:15.055141	2018-11-25 07:03:15.055141 UTC

Note that the canonical representation of a datetime has the letter T separating the date part and the time part while the representation of a timestamp uses a space. The timestamp also explicitly includes the time zone whereas the time zone is implicit in the datetime. But for the most part, DATETIME and TIMESTAMP can be used interchangeably in BigQuery.

The DATE is just the date part of a DATETIME (or a TIMESTAMP,

interpreted in some time zone) and TIME is the time part. Because many real-world scenarios might happen on a certain date (i.e. at multiple times over that day), many database tables contain just a DATE. So there is some benefit to being able to directly parse and format dates. On the other hand, there is very little need for the TIME type other than as the “missing” part of a DATETIME.

For the most part, therefore, our advice is to just use TIMESTAMP and DATE. There is, however, one practical wrinkle to using TIMESTAMP. Timestamps in BigQuery are stored using eight bytes with microsecond resolution. This means that you can store years 0-9999, and any microsecond in between. In some other databases (e.g. MySQL), TIMESTAMP is stored using four bytes and DATETIME using eight bytes. In those systems, the range of a TIMESTAMP is within the limits of the Unix epoch time (years 1970 to 2038) which means that you cannot even store birthdays of 60-year-olds or expiry dates of 30-year-mortgages. So, while a TIMESTAMP might work in BigQuery, you may not be able to use the same schema in MySQL, and this might make moving queries and data between BigQuery and MySQL hard.

Working with GIS functions

We will look at Geography functions in much more detail in Chapter X, on advanced features. In this section, we will provide only a brief introduction.

The GEOGRAPHY type can be used to represent points, lines, and polygons on the surface of the earth (i.e. there is no height associated with them). Because Earth is a lumpy mass, points on the surface of the earth can only be represented on spherical and ellipsoidal approximations to the

surface. In BigQuery, the geographic positions of the points, and vertices of the lines and polygons are represented in the WGS84 ellipsoid.¹² Practically speaking, this is the same ellipsoid as used by the Global Positioning System (GPS), so you will be able to take the longitude and latitude positions reported by most sensors and directly use them in BigQuery.

The simplest geography is a point specified by its longitude and latitude. So, for example,

```
ST_GeogPoint(-122.33, 47.61)
```

represents a point at 47.61N and 122.33W, i.e. Seattle, Washington.

The BigQuery public datasets include a table that contains polygons corresponding to each of the US states and territories. We can therefore write a query to find out which state the geographic point is in:

```
SELECT
```

```
state_name
```

```
FROM `bigquery-public-data`.utility_us.us_states_area
```

```
WHERE
```

```
ST_Contains(
```

```
state_geom,
```

```
ST_GeogPoint(-122.33, 47.61))
```

As anticipated, this returns:

Row	state_name
1	Washington

The query uses the ST_Contains function to determine whether the state's geometry (stored as the state_geom column in the BigQuery dataset) contains the point we are interested in. The spatial functions that BigQuery supports follow the SQL MM 3 specification¹³, and are similar to what the PostGIS library provides for Postgres.¹⁴

Summary of data types in BigQuery

BigQuery supports the following data types:

D		
at		
a ty	Sample functions and operators supported	Notes
p		
e		
I		
N	Arithmetic operations (+, -, /, *)	
T	for add, subtract, divide and 6 multiply)	Approximately 10-19 to 1019
4		
N		
U		
M		
E	Arithmetic operations	38 digits of precision and 9 decimal digits of scale; this is suitable for financial calculations
R		
I		
C		
F		
L		
O		
A	Arithmetic operations Also: IEEE_DIVIDE	IEEE-754 behavior if one of the values is NaN or +/- inf
T		
6		
4		

B	Conditional statements MIN, MAX	
O		Is either True and False
O	However, SUM, AVG, etc. are	SQL is case-insensitive, so TRUE, true, etc. also
L	not supported (you'd have to cast the booleans to int64 first)	work.
S		
T	Use special String	
R	functions ^a such as CONCAT,	Strings are Unicode characters and are variable length.
I	LENGTH, etc. to operate on	
N	strings.	
G		
B		
Y		Variable length characters
T		Many String operations are also defined on BYTES.
E		
S		
T	CURRENT_TIMESTAMP() represents “now”.	
I		
M	You can extract month, year, E dayofweek etc. from a	Absolute point in time, to microsecond precision,
S	S timestamp.	represented in a subset of ISO 8601. This is the T recommended way to store times in BigQuery.
A		
M	Arithmetic on timestamps is P supported via special functions ^b ,	
P	not through arithmetic operators.	
D	CURRENT_DATE() represents the current date in the UTC time zone while	2018-3-14 (or 2018-03-14) is March 14, 2018
A	CURRENT_DATE("America/L os_Angeles") represents the current date in the Los Angeles time zone.	independent of timezone. ^d Because this represents T different 24-hour blocks in different time zones, use E TIMESTAMP to represent an absolute point in time. You can then construct a DATE from a TIMESTAMP relative to a particular time zone.
T		
E	Like TIMESTAMP, arithmetic on dates is supported via special functions. ^c	
D		
A		
T		
E		
T	As with DATE.	2018-03-14 3:14:57 or 2018-03-14T03:14:57.000000
I		is, like, DATE, independent of timezone. Most
M		applications will want to use TIMESTAMP.
E		

T	
I	As with DATETIME, except that the DATE part is absent.
M	Independent of a specific date or time zone, this ranges from 00:00:00 to 23:59:59.999999.
E	
G	
E	
O	
G	Topological functions on
R	geographies are supported via
A	special functions. ^e
P	
H	
Y	The simplest geography is a point specified by its longitude and latitude.
S	
T	
R	You can deference the fields by name.
U	
C	A collection of fields in order
T	The field name is optional, i.e. you could have either: STRUCT<INT64, STRING> or STRUCT<id INT64, name STRING>
A	
R	
R	
A	
Y	You can deference the items by offset, aggregate the items in the array, or unnest them to get the items one by one.
	Ordered list of non-null elements e.g. ARRAY<INT64>. Arrays of arrays are not allowed, but you can get around this by creating an Array of STRUCT where the struct itself contains an array i.e. ARRAY<STRUCT<ARRAY<INT64>>> (Arrays are covered in Chapter 2).
a	See https://cloud.google.com/bigquery/docs/reference/standard-sql/string_functions
b	See https://cloud.google.com/bigquery/docs/reference/standard-sql/timestamp_functions
c	See https://cloud.google.com/bigquery/docs/reference/standard-sql/date_functions
d	https://cloud.google.com/bigquery/docs/reference/standard-sql/data-types#timestamp-type
e	See https://cloud.google.com/bigquery/docs/reference/standard-sql/geography_functions

All data types, except for arrays and structs, can be used in ORDER BY and GROUP BY.

-
- 1 See https://cloud.google.com/bigquery/docs/reference/standard-sql/mathematical_functions
 - 2 The standard division operator raises a division-by-zero error. SAFE_Divide returns NULL for the entry where division by zero is attempted.
 - 3 See <https://www.github.com/GoogleCloudPlatform/bigquery-oreilly-book/>
 - 4 See https://cloud.google.com/bigquery/docs/reference/standard-sql/string_functions
 - 5 See <http://www.cplusplus.com/reference/cstdio/printf/>
 - 6 See <https://github.com/google/re2>
 - 7 Start with Mastering Regular Expressions by Jeffrey Friedl and published by O'Reilly Media. See <http://shop.oreilly.com/product/9781565922570.do>
 - 8 See <https://www.iso.org/iso-8601-date-and-time-format.html>
 - 9 See https://cloud.google.com/bigquery/docs/reference/standard-sql/timestamp_functions#supported_format_elements_for_timestamp
 - 10 According to https://en.wikipedia.org/wiki/Armistice_Day, the agreement was signed at 5.45am on Nov 11, 1918.
 - 11 In Winter 1918, unlike now, France was in the UTC timezone. See <https://www.timeanddate.com/time/zone/france/paris>
 - 12 See https://en.wikipedia.org/wiki/World_Geodetic_System
 - 13 See <http://doesen0.informatik.uni-leipzig.de/proceedings/paper/68.pdf> and <http://jtc1sc32.org/doc/N1101-1150/32N1107-WD13249-3--spatial.pdf>
 - 14 See <https://postgis.net/docs/manual-1.4/ch08.html>

Chapter 4. Loading Data into BigQuery

In the previous chapter, we wrote the query:

SELECT

state_name

FROM `bigquery-public-data`.utility_us.us_states_area

WHERE

ST_Contains(

state_geom,

ST_GeogPoint(-122.33, 47.61))

and learned that the city at the location (-122.33, 47.61) is in the American state of Washington. Where did the data for the state_name and state_geom come from?

Note the FROM clause in the query. The owners of the bigquery-public-data project had already loaded the state boundary information into a table called us_states_area in a dataset called utility_us. Because the team shared the utility_us dataset with all authenticated users of BigQuery (more restrictive permissions are available), we were able to query the

`us_states_area` table that is in that dataset.

But how did they get the data into BigQuery in the first place? In this chapter, we will look at various ways to load data into BigQuery, starting with the basics.

The basics

Data values like the boundaries of US states change rarely¹, and the changes are small enough that most applications can afford to ignore them. In data warehousing lingo, we call this a *slowly-changing dimension*. At the time of writing, the last change of US state boundaries happened on Jan 1, 2017 and affected 19 home owners and one gas station².

State boundary data is, therefore, the type of data that is often loaded just once. Analysts query the single table and ignore the fact that the data could change over time. For example, a retail firm may only care about which state a home is in currently, to ensure that the correct tax rate is applied to purchases from that home. So, when a change does happen, such as through a treaty between states or change in the path of a river channel, the owners of the dataset may decide to replace the table with more up-to-date data. The fact that queries could potentially return slightly different results after an update from what they did before the update is ignored.

Ignoring the impact of time on the correctness of the data may not always be possible. If the state boundary data are to be used by a land title firm that needs to track ownership of land parcels, or if an audit firm needs to validate the state tax paid on shipments made in different years, then it is important that there be a way to query the state boundaries as they existed

in years past. So, while the first part of this chapter covers how to do a one-time load, carefully consider whether you would be better off planning on periodically updating the data and allowing users of the data to know about the version of the data that they are querying.

Loading from a local source

The US government issues a “scorecard” for colleges to help consumers compare the cost and perceived value of higher education. Let’s load this data into BigQuery as an illustration. The raw data is available on catalog.data.gov³ -- for convenience, we have it also available in the GitHub repository of this book⁴. The comma-separated-values (CSV) file was downloaded from data.gov and compressed using an open-source software utility called gzip.

<tip>

Why did we compress the file? The raw, uncompressed file is about 136 MB while the gzipped file is only 18 MB. Since we are about to send the file over the wire to BigQuery, it makes sense to optimize the bandwidth being transferred. The BigQuery load command can handle gzipped files, but cannot load parts of a gzipped file in parallel. Loading would be much faster if we hand BigQuery a splittable file, either an uncompressed CSV file that is already on Cloud Storage (so that the network transfer overhead is minimized) or data in a format such as Avro where each block is internally compressed, but the file as a whole can be split across workers.

A splittable file can be loaded by different workers starting at different parts of the file, but this requires that the workers be able to “seek” to a predictable point in the middle of the file without having to read it from the beginning. Compressing the entire file using gzip doesn’t allow this,

but a block-by-block compression such as Avro does. Therefore, using a compressed, splittable format such as Avro is an unmitigated good. However, if you have CSV or JSON files that are splittable only when uncompressed, you should measure whether the faster network transfer is counterbalanced by the increased load time.

</tip>

From CloudShell, you can page through the gzipped file using zless⁵:

```
zless college_scorecard.csv.gz
```

The file contains a header line with the names of the columns. Each of the lines following the header contains one row of data.

In order to load the data into BigQuery, first create a dataset called ch04 to hold the data:

```
bq --location=US mk ch04
```

The bq command-line tool provides a convenient point of entry to interact with the BigQuery service on Google Cloud Platform, although everything you do with bq can be done using the REST API and most things can also be accomplished using the GCP web console. Here, we are asking it to make (mk) a dataset named ch04.

Datasets in BigQuery function like top-level folders that are used to organize and control access to tables, views, and machine learning models. The dataset is created in the current project⁶ and it is to this project that storage costs for tables in this dataset will be billed (queries are charged to the project of the querier).

We also specify that the dataset should be created in the US location (this is the default, so we could have omitted it). Location choices include multi-region locations (such as US, EU) and specific regions (e.g., us-east4, europe-west2, australia-southeast1, etc.⁷). Be careful when choosing a region for loading data -- at the time of writing, queries cannot join tables held in different regions. In this book, we will use the US multi-region location so that our queries can join against tables in the public datasets which are located in the US.

Then, from the directory containing your clone of the GitHub repository, load the data in the file as a table in BigQuery:

```
bq --location=US \
load \
--source_format=CSV --autodetect \
ch04.college_scorecard \
./college_scorecard.csv.gz
```

In this case, we are asking bq to load the dataset, telling the tool that the source format is CSV and that we would like the tool to auto-detect the schema (i.e., the data types of individual columns). We then specify that the table to be created is called college_scorecard in the dataset ch04 and that the data is to be loaded from college_scorecard.csv.gz in the current directory.

When we did this, though, we ran into an issue:

Could not parse ‘NULL’ as int for field HBCU (position 26) starting at location 11945910

This caused the load job to fail with:⁸

CSV table encountered too many errors, giving up. Rows: 591; errors: 1.

The problem is that based on most of the data in the CSV file, BigQuery’s schema autodetection expects that the 26th column (whose name is HBCU) should be an integer, but the 591st row of the file has the text NULL in that field -- this usually signifies that the college in question did not answer the survey question corresponding to this field.⁹

There are several ways we can fix this problem. For example, we could edit the data file itself if we knew what the value ought to be. Another fix could be to specify explicitly the schema for each column and change the column type of the HBCU column to be a string, so that NULL is an acceptable value. Alternatively, we could ask BigQuery to ignore a few bad records by specifying, for example, --max_bad_records=20. Finally, we could tell the BigQuery load program that this particular file uses the string “NULL” to mark nulls (the standard way in CSV is to use empty fields to represent nulls).

Let’s apply the last method because it seems to be the most appropriate¹⁰:

```
bq --location=US \
```

```
load --null_marker=NULL \
```

```
--source_format=CSV --autodetect \
```

```
ch04.college_scorecard \
```

```
./college_scorecard.csv.gz
```

You can find the full list of bq load options by typing: `bq load --help`. By default, `bq load` will append to a table. Here, we want to replace the existing table, so we should add `--replace`:

```
bq --location=US \
```

```
load --null_marker=NULL --replace \
```

```
--source_format=CSV --autodetect \
```

```
ch04.college_scorecard \
```

```
./college_scorecard.csv.gz
```

You can also specify `--replace=false` to append rows to an existing table.

<tip>

Loading data into BigQuery does not incur any charges, although you will be charged for storage once the data is loaded.¹¹ If you are on flat rate pricing, loading data into BigQuery uses computational resources that are separate from the slots that are paid for by the flat rate. Therefore, if you do not need near real-time data in your data warehouse, a frugal way to get data into BigQuery is to set up a scheduled Cloud Storage transfer (covered later in this chapter). If transformations are needed, you can use Cloud Composer or Cloud Functions to load data into BigQuery every day.

All that the bq command does is to invoke a REST API exposed by the BigQuery service. So, you can load the data in many other ways as well. Those methods invoke the same REST API. Client libraries in a number of languages including Java, Python and Node.js are available -- these provide convenient programmatic ways to upload the load. The use of client libraries will be discussed in Chapter X.

If you do need data in near real-time, then you should stream data into BigQuery.¹² Even though streaming incurs charges, you should prefer to use streaming over frequent loads if you need near real-time data. It is not a good idea to load data using a large number of small load jobs frequently (for example, to issue a load every minute). Tables that are loaded so frequently can end up with significant fragmentation and high metadata overhead, causing queries over them to be slow until BigQuery performs an optimization pass at some point in the future. Streaming, unlike frequent small loads, batches rows on the backend for a period of time before writing them to storage, thus limiting the fragmentation and keeping querying performant. Streamed data is available for querying immediately, whereas loads can take a while to complete. Moreover, if you rely on frequent small batch loads, any sort of throttling or backups in the systems that produce these files can result in unexpected delays in data being available.

</tip>

It is worth noting is that you can do one-time loads from the BigQuery web user interface. Click on your project and you will be presented with a button to create a dataset (ch04 in our case); click on the dataset and you will be presented with a button to create a table. You can then follow the prompts to upload the file as a BigQuery table. At the time of writing,

however, use of the web UI to load data from a local file is limited to data whose size is less than 10 MB and 16000 rows. Hence, it will not work for the college scorecard dataset unless we had staged it in GCS first.

Even if you did not (or cannot) use the web UI to load the data, it is a good idea to look at the created table using the web UI to ensure that details about the table as well as the autodetected schema are correct. It is also possible to edit some details about the table even after it has been created. For example, it is possible to specify that the table should automatically expire after a certain number of days¹³, add columns, or relax a required field to become nullable.

Regardless of how the table is loaded, it can be queried by anyone who is allowed to access the dataset in which the table is located. The default is to make a newly created dataset visible only to people with project-level view permissions. You can, however, share the dataset¹⁴ with specific individuals (identified by their Google account), a domain (e.g. xyz.com), or a Google group. Identify and access management to share datasets will be discussed in Chapter X. For now, though, anyone with view access to the project holding the dataset can query it:

SELECT

INSTNM

, ADM_RATE_ALL

, FIRST_GEN

, MD_FAMINC

, MD_EARN_WNE_P10

, SAT_AVG

FROM

ch04.college_scorecard

WHERE

SAFE_CAST(SAT_AVG AS FLOAT64) > 1300

AND SAFE_CAST(ADM_RATE_ALL AS FLOAT64) < 0.2

AND SAFE_CAST(FIRST_GEN AS FLOAT64) > 0.1

ORDER BY

CAST(MD_FAMINC AS FLOAT64) ASC

This query pulls out college name (INSTNM), admission rate, etc. for colleges whose average SAT score is more than 1300 and whose admission rate is less than 20%, which is a plausible definition of “elite” colleges. It also filters by colleges that admit first-generation college goers at a rate greater than 10% and ranks them in ascending order of median family income, thus finding elite colleges that admit culturally or economically disadvantaged students. The query also pulls the median earnings of students 10 years after entry:

R	INSTNM	ADM_RAT_E_ALL	FIRST_GEN	MD_FA_MINC	MD_EARN_WNE_P10	SAT_AVG
o	University of California-	0.16926878	0.34580	11000	64700	1400

1	Berkeley	30816	05249	51227	64/00	1422
2	Columbia University in the City of New York	0.06825366 802669	0.25049 05167	31310.5	83300	1496
3	University of California-Los Angeles	0.17992627 069775	0.38089 13934	32613.5	60700	1334
4	Harvard University	0.05404574 677902	0.25708 061	33066	89700	1506
5	Princeton University	0.06521516 568269	0.27739 72603	37036	74700	1493

Look, however, at the query itself. Notice how several of the WHERE clauses need a cast:

```
SAFE_CAST(ADM_RATE_ALL AS FLOAT64)
```

Had we not included the cast, we would have gotten an error:

No matching signature for operator > for argument types: STRING, INT64.

Had we simply cast as a float, it would have failed on a row where the value was a string (PrivacySuppressed) that can not be cast as a float:

Bad double value: PrivacySuppressed; while executing the filter ...

This is because the automatic schema detection did not identify the admission rate column as numeric. Instead, that column is being treated as a string because, in some of the rows, the value is suppressed for privacy reasons (e.g. if the number of applications is very small) and replaced by the text PrivacySuppressed. Indeed, even the median family income is a string (it happens to be always numeric for colleges that meet the criteria we outlined) and so we need to cast it before ordering¹⁵.

Specifying a schema

Inevitably, in real-world datasets, we will have to do some cleanup and transformations before loading the data into BigQuery. While we will look at how to build more sophisticated data processing pipelines to do this later in this chapter, a simple way is to use Unix tools to replace privacy-suppressed data with NULLs:

```
zless ./college_scorecard.csv.gz | \  
sed 's/PrivacySuppressed/NULL/g' | \  
gzip > /tmp/college_scorecard.csv.gz
```

Here, we are using a string editor (sed) to replace all occurrences of PrivacySuppressed by NULL, compressing the result and writing it to a temporary folder. Now, instead of loading the original file, we can load the cleaner file.

When presented with the cleaner file, BigQuery correctly identifies many more of the columns as integers or floats, but not the SAT_AVG or ADM_RATE -- those columns are still autodetected as strings. This is because the algorithm to autodetect the schema does not look at all the rows in the file, only a sample of them. Because a large fraction of rows have a null SAT_AVG (fewer than 20% of colleges report SAT scores), the algorithm was unable to infer the type of the field. The safe choice is to treat columns that the tool is not sure of as a string.

It is, therefore, best practice to not auto-detect the schema of files that you receive in production -- you will be at the mercy of whatever data happens to have been sampled. For production workloads, insist on the data type

for a column by specifying it at the time of load.

We can use the autodetect feature to avoid starting to write a schema from scratch. We can display the schema of the table as it currently exists:

```
bq show --format prettyjson --schema ch04.college_scorecard
```

We can also save the schema to a file using:

```
bq show --format prettyjson --schema ch04.college_scorecard >  
schema.json
```

<tip>

We haven't covered table metadata yet (we'll do so in Chapter X), but you can automate the creation of the schema using SQL itself. Here is a query to obtain the schema of all the tables in the dataset ch04:

```
SELECT
```

```
    table_name
```

```
    , column_name
```

```
    , ordinal_position
```

```
    , is_nullable
```

```
    , data_type
```

```
FROM
```

ch04.INFORMATION_SCHEMA.COLUMNS

You can then use the TO_JSON_STRING function to create the JSON of the schema in the necessary format, thus avoiding the need to drop to the command-line:

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’

This yields a JSON string of the form:

[

```
{  
  "mode": "NULLABLE",  
  "name": "INSTNM",  
  "type": "STRING"  
},  
  
{  
  "mode": "NULLABLE",  
  "name": "ADM_RATE_ALL",  
  "type": "FLOAT64"  
},  
  
...  
</tip>
```

Now, you can open the schema file in your favorite text editor (If you don't have a preference, use the pen icon in CloudShell to open up the default editor) and change the type of the columns we care about. Specifically, change the four columns in our WHERE clause (SAT_AVG, ADM_RATE_ALL, FIRST_GEN and MD_FAMINC) to be FLOAT64:

```
{
```

```
    "mode": "NULLABLE",  
  
    "name": "FIRST_GEN",  
  
    "type": "FLOAT64"  
  
},
```

In addition, also (for now) change the T4APPROVALDATE to be a STRING because it is in a non-standard date format¹⁶:

```
{  
  
    "mode": "NULLABLE",  
  
    "name": "T4APPROVALDATE",  
  
    "type": "STRING"  
  
},
```

With the schema updated, we can load the data with this schema, rather than with the auto-detect:

```
bq --location=US \  
  
load --null_marker=NULL --replace \  
  
--source_format=CSV \  
  
--schema=schema.json --skip_leading_rows=1 \  

```

```
ch04.college_scorecard \
```

```
./college_scorecard.csv.gz
```

Because we are supplying a schema, we have to tell BigQuery to ignore the first row of the CSV file (which contains the header information)

Once the table has been loaded, we can repeat the query of the previous section:

```
SELECT
```

```
INSTNM
```

```
, ADM_RATE_ALL
```

```
, FIRST_GEN
```

```
, MD_FAMINC
```

```
, MD_EARN_WNE_P10
```

```
, SAT_AVG
```

```
FROM
```

```
ch04.college_scorecard
```

```
WHERE
```

```
SAT_AVG > 1300
```

AND ADM_RATE_ALL < 0.2

AND FIRST_GEN > 0.1

ORDER BY

MD_FAMINC ASC

Notice that because SAT_AVG, ADM_RATE_ALL, etc. are no longer strings, our query is much cleaner since we no longer need to cast them to floating point numbers. The reason they are no longer strings, is that we made a decision on how to deal with the privacy suppressed data (to treat them as being unavailable) during the ETL process.

Copying into a new table

The table as loaded contains many columns that we do not need. It is possible to create a cleaner, more purposeful table from the original table using the CREATE TABLE statement and populating the new table with only the columns of interest:

CREATE OR REPLACE TABLE **ch04.college_scorecard_etl** AS

SELECT

INSTNM

, ADM_RATE_ALL

, FIRST_GEN

, MD_FAMINC

, SAT_AVG

, MD_EARN_WNE_P10

FROM ch04.college_scorecard

By using a robust ETL pipeline and making decisions early, downstream queries are cleaner and more concise. The tradeoff is that the ETL process involves extra work (figuring out the data types and specifying the schema) and might involve irrevocable decisions (e.g., there is no way to get back whether a field is unavailable because it was not collected, because it was suppressed due to privacy reasons, or because it was deleted). Later in this chapter, we discuss how an ELT pipeline in SQL can help us delay making irrevocable decisions.

Data management (DDL and DML)

Why cover data management in a chapter on loading data? Because, typically, loading data is only part of the task of managing data. If data is loaded by mistake, it might have to be deleted. Sometimes, data deletion has to happen because of regulations and compliance.

<warning>

Even though we normally want you to try all the commands and queries in this book, don't try the ones in this section since you will lose your data!

</warning>

The easiest way to delete a table (or view) as a whole is from the BigQuery UI. You can also carry out the delete from the bq command-line

tool:

```
bq rm ch04.college_scorecard
```

```
bq rm -r -f ch04
```

The first line removes a single table, whereas the second one removes recursively (-r) and without prompting (-f, for force) the dataset ch04 and all the tables it contains.

Deleting a table (or view) can also be accomplished using SQL:

```
DROP TABLE IF EXISTS ch04.college_scorecard_gcs
```

It is also possible to specify that a table needs to be expired at a certain time in the future. This can be achieved with the ALTER TABLE SET OPTIONS statement:

```
ALTER TABLE ch04.college_scorecard
```

```
SET OPTIONS (
```

```
    expiration_timestamp=TIMESTAMP_ADD(CURRENT_TIMESTAMP(),
```

```
    INTERVAL 7 DAY),
```

```
    description="College Scorecard expires seven days from now"
```

```
)
```

The DROP TABLE and ALTER TABLE statements above, like the CREATE TABLE statement, are examples of Data Definition Language

(DDL) statements.

It is possible to delete only specific rows from a table. For example:

```
DELETE FROM ch04.college_scorecard
```

```
WHERE SAT_AVG IS NULL
```

Similarly, it is also possible to INSERT rows into an existing table instead of replacing the entire table. For example, it is possible to insert more values into the college_scorecard table using:

```
INSERT ch04.college_scorecard
```

```
(INSTNM
```

```
, ADM_RATE_ALL
```

```
, FIRST_GEN
```

```
, MD_FAMINC
```

```
, SAT_AVG
```

```
, MD_EARN_WNE_P10
```

```
)
```

```
VALUES ('abc', 0.1, 0.3, 12345, 1234, 23456),
```

```
('def', 0.2, 0.2, 23451, 1232, 32456)
```

It is possible to use a subquery to extract values from one table and copy them into another:

```
INSERT ch04.college_scorecard  
  
SELECT *  
  
FROM ch04.college_scorecard_etl  
  
WHERE SAT_AVG IS NULL
```

The DELETE, INSERT and MERGE statements are examples of Data Manipulation Language (DML) statements.

<tip>

At the time of writing, BigQuery does not support a SQL COPY statement. To copy tables, use bq cp to copy one table to another:

```
bq cp ch04.college_scorecard someds.college_scorecard_copy
```

You are not billed for running a query but you will be billed for the storage of the new table. The bq cp command supports appending (specify -a or --append_table) and replacement (specify -noappend_table).

You can also use the idiomatic Standard SQL method of using either the CREATE TABLE AS SELECT or INSERT VALUES depending on whether the destination already exists. However, bq cp is faster (since it copies only the table metadata) and doesn't incur query costs.

</tip>

Loading data efficiently

While BigQuery can load data from CSV files, CSV files are inefficient and not very expressive (for example, there is no way to represent arrays and structs in CSV). If you have a choice, you should choose to export your data in a different format. What format should you choose?

The most efficient and expressive format is Avro¹⁷. Avro uses self-describing binary files that are broken into blocks and can be compressed block-by-block. Because of this, it is possible to parallelize the loading of data from Avro files and the export of data into Avro files. Because the blocks are compressed, the file sizes will also be smaller than the data size might indicate. In terms of expressiveness, the Avro format is hierarchical and can represent nested and repeated fields, something that BigQuery supports but CSV files don't have an easy way to store. Since Avro files are self-describing, you never need to specify a schema.

The one drawback to Avro files is that they are not human readable. If readability and expressiveness are important to you, use newline-delimited JSON files¹⁸ to store your data. JSON supports the ability to store hierarchical data, but requires that binary columns be base64-encoded. However, JSON files are larger than even the equivalent CSV files because the name of each field is repeated on every line.

Parquet files are a more recent addition to the set of file formats that BigQuery supports. The Parquet file format was inspired by Google's original Dremel ColumnIO format¹⁹ and, like Avro, Parquet is binary, block-oriented, compact, and capable of representing hierarchical data. However, while Avro files are stored row-by-row, Parquet files are stored column-by-column. Columnar files are optimized for reading a subset of the columns; loading data requires reading all columns, and so columnar

formats are somewhat less efficient at the loading of data. However, the columnar format makes Parquet a better choice than Avro for federated queries, a topic that we will discuss shortly. ORC files are another open source columnar file format, and are similar to Parquet files in performance and efficiency.

IMPACT OF COMPRESSION AND STAGING VIA GCS

For formats such as CSV and JSON that do not have internal compression, you should consider whether you should compress the files using gzip. Compressed files are faster to transmit and take up less space, but are slower to load into BigQuery. The slower your network, the more you should lean towards compressing the data.

If you are on a slow network or if you have many files or very large files, it is possible to setup a multithreaded upload of the data using gsutil cp. Once the data are all on Google Cloud Storage, then you can invoke bq load from the Cloud Storage location:

```
gsutil -m cp *.csv gs://BUCKET/some/location
```

```
bqload ... gs://BUCKET/some/location/*.csv
```

This experiment captures the various tradeoffs involved with compression and with staging the college scorecard data on Google Cloud Storage before invoking bq load. Your results will vary, of course, depending on your network and the actual data you are loading.²⁰ Therefore, you should carry out a similar measurement for your loading job and choose the method that provides you the best performance on the measures you care about.

Compress	Stage on Google	Cloud	Network time (if applicable)	Time to load into BigQuery	Total
----------	-----------------	-------	------------------------------	----------------------------	-------

Compressed file	Cloud Storage?	Storage size (separate)		BigQuery time	
Yes	No	None	N/A	105 sec	105 sec
No	No	None	N/A	255 sec	255 sec
Yes	Yes	16 MB	47 sec	42 sec	89 sec
No	Yes	76 MB	139 sec	28 sec	167 sec

Table 4-1. Tradeoffs involved with compression and staging the college scorecard data on Google Cloud Storage before invoking `bq load`.

Staging the file on Google Cloud Storage involves paying storage costs at least until the BigQuery load job finishes. However, storage costs are generally quite low and so, on this dataset and this network connection (see Table 4-1), the best option is to stage compressed data in Google Cloud Storage and load it from there. Even though it is faster to load uncompressed files into BigQuery, the network time to transfer the files dwarfs whatever benefits you'd get from a faster load.

At the time of writing, loading of compressed CSV and JSON files is limited to files less than 4 GB in size, since BigQuery has to uncompress the files on the fly on workers whose memory is finite. If you have larger datasets, split them across multiple CSV or JSON files. Splitting files yourself can allow for some degree of parallelism when doing the loads, but depending on how you size the files, this can lead to sub-optimal file sizes in the table until BigQuery decides to optimize the storage.

PRICE AND QUOTA

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. Hence, your queries (even on the same table into which you are ingesting data) are not slowed down by the fact that data are being ingested.

Data loads are atomic. Queries on a table will either reflect the presence of all the data that is loaded in through the bq load operation or reflect none of it. You will not get query results on a partial slice of the data.

The drawback of loading data using a “free” cluster is that load times can become unpredictable and bottlenecked by preexisting jobs. At the time of writing, load jobs are limited to 1,000 per table and 50,000 per project per day²¹. In the case of CSV and JSON files, cells and rows are limited to 100 MB while in Avro, blocks are limited to 16 MB. Each file has to be smaller than 5 TB. If you have a larger dataset, split it across multiple files, each of which is under 5 TB in size. However, a single load job can submit a maximum of 15 TB of data split across a maximum of 10 million files. The load job has to finish executing in under six hours or it will be cancelled.

Federated queries and external data sources

You can use BigQuery without first loading the data. It is possible to leave the data in-place, specify the structure of the data, and use BigQuery as just the query engine. In contrast to the queries thus far where BigQuery queried its own native storage, we will discuss the use of “federated queries” to query “external data sources” in this section and explain when you might want to use such queries.

Currently supported external data sources include Google Cloud Storage, Cloud Bigtable, Cloud SQL and Google Drive. You will notice that all

these sources are external to BigQuery but are, nevertheless, within the Google Cloud perimeter. This is necessary because, otherwise, the network overhead and security considerations would make the queries either slow or infeasible.

How to use federated queries

There are three steps to query data in an external data source:

Create a table definition using `bq mkdef`

Make a table using `bq mk` passing in the external table definition

Query the table as normal.

As with querying data in native storage, you can do this either in the web UI or using a programmatic interface. To use the web user interface, you will follow the steps to create a table, but make sure to specify that you want an external table, not a native one:

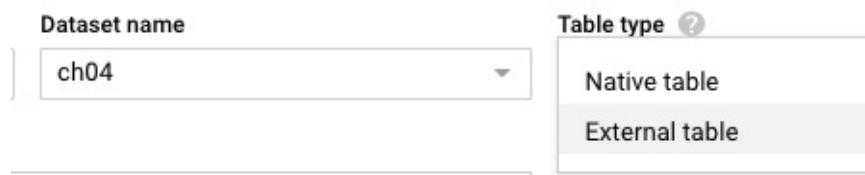


Figure 4-1. You can create an external table from the web UI by following the “Create Table” workflow, but specifying “External table” as the table type.

Using the command-line interface, create a table definition using `bq mkdef`. As with `bq load`, you have the option of using `--autodetect`:

```
bq mkdef --source_format=CSV \
```

```
--autodetect \
```

```
gs://bigquery-oreilly-book/college_scorecard.csv
```

This prints a table definition file to standard output. The normal course of action is to redirect this to a file, and use that table definition to make a table using bq mk:

```
bq mkdef --source_format=CSV \  
--autodetect \  
  
gs://bigquery-oreilly-book/college_scorecard.csv \  
  
> /tmp/mytable.json  
  
bq mk --external_table_definition=/tmp/mytable.json \  
  
ch04.college_scorecard
```

With these two steps, you can query the table college_scorecard as in the previous section except that the queries will happen on the CSV file stored in Google Cloud Storage -- the data is not ingested into BigQuery's native storage.

WILDCARDS

Many Big Data frameworks such as Apache Spark, Apache Beam, etc. shard their output across hundreds of files with names such as course_grades.csv-00095-of-00313. When loading such files, it can be convenient if we could avoid having to list each file individually.

Indeed, it is possible to use a wildcard in the path to bq mkdef (and bq load) so that you can match multiple files. For example:

```
bq mkdef --source_format=CSV \  
--autodetect \  
gs://bigquery-oreilly-book/college_* \  
> /tmp/mytable.json
```

will create a table that refers to all the files matched by the pattern.

TEMPORARY TABLE

It is also possible to condense the three steps above (mkdef, mk, and query) by passing in the table definition parameters along with a query, thus ensuring that the table definition will be used only for the duration of the query:

```
LOC="--location US"  
  
INPUT=gs://bigquery-oreilly-book/college_scorecard.csv  
  
SCHEMA=$(gsutil cat $INPUT | head -1 | awk -F, '{ORS=","}{for (i=1; i  
<= NF; i++){ print $i":STRING"; }}' | sed 's/,$/g'| cut -b 4- )  
  
bq $LOC query \  
--external_table_definition=cstable::${SCHEMA}@CSV=${INPUT} \  
'SELECT SUM(IF(SAT_AVG != "NULL", 1, 0))/COUNT(SAT_AVG)  
FROM cstable'
```

In the query above, the external table definition consists of the temporary

table name (cstable), two colons, the schema string, the @ symbol, the format (CSV), equals sign, and the GCS URL corresponding to the data file(s). If you already have a table definition file, you can specify it directly:

```
--external_table_definition=cstable::${DEF}
```

It is possible to specify a JSON schema file, as well as to directly query JSON, Avro, and other supported formats directly from Google Cloud Storage, Cloud Bigtable, and other supported data sources.

While undeniably convenient, federated queries leave much to be desired in terms of performance. Because CSV files are stored row-wise and the rows themselves are stored in some arbitrary order, much of the efficiency that we commonly associate with BigQuery is lost. It is also not possible for BigQuery to estimate how much data it is going to have to scan before running the query.

LOADING AND QUERYING PARQUET AND ORC

As previously mentioned, Parquet and ORC are columnar data formats. Therefore, federated querying of these formats will provide better query performance than if the data were stored in row-based formats such as CSV or JSON (queries will still be slower than BigQuery's native Capacitor storage, however).

Because Parquet and ORC are self-describing (i.e., the schema is implicit in the files themselves), it is possible to create table definitions without specifying a schema:

```
bq mkdef --source_format=PARQUET gs://bucket/dir/files* >
```

table_def.json

```
bq mk --external_table_definition=table_def.json <dataset>. <table>
```

As with querying external tables created from CSV files, querying this table works like querying any other table in BigQuery.

While Parquet and ORC files provide better query performance than row-based file formats, they are still subject to the limitations of external tables.

LOADING AND QUERYING HIVE PARTITIONS

Apache Hive²² allows reading, writing, and managing an Apache Hadoop based data warehouse using a familiar SQL-like query language. Cloud Dataproc, on Google Cloud, enables Hive software to work on distributed data stored in Hive partitions on Google Cloud Storage. A common public cloud migration pattern is for on-premises Hive workloads to be migrated to Cloud Dataproc, and newer workloads to be written using BigQuery's federated querying capability. This way, the current Hive workloads work as-is, while newer workloads can take advantage of the serverless, large-scale querying capability provided by BigQuery.

Hive partitions on Google Cloud Storage can be loaded by specifying a Hive partitioning mode to bq load:

```
bq load --source_format=ORC --autodetect \  
--hive_partitioning_mode=AUTO <dataset>. <table> <gcs_uri>
```

The GCS URI in the case of Hive tables needs to encode the table path prefix without including any partition keys in the wildcard. Thus, if the

partition key for a Hive table is a field named `datestamp`, the GCS URI should be of the form:

```
gs://some-bucket/some-dir/some-table/*
```

even if the files themselves all start with:

```
gs://some-bucket/some-dir/some-table/datestamp=
```

At the time of writing, the AUTO partitioning mode can detect the following types: STRING, INTEGER, DATE and TIMESTAMP. It is also possible to request that the partition keys be detected as strings (this can be helpful in exploratory work):

```
bq load --source_format=ORC --autodetect \  
--hive_partitioning_mode=STRINGS <dataset>. <table> <gcs_uri>
```

As with CSV files from Google Cloud Storage, federated querying of Hive partitions requires the creation of a table definition file, and the options closely mirror that of `load`:

```
bq mkdef --source_format=ORC --autodetect \  
--hive_partitioning_mode=AUTO <gcs_uri> > table_def.json
```

Once the table definition file is created, querying is the same whether the underlying external dataset consists of CSV files or Hive partitions.

In addition to ORC, as shown above, data in other formats are also supported. For example, to create a table definition of data stored in new-

line delimited JSON, you can use:

```
bq mkdef --source_format=NEWLINE_DELIMITED_JSON --autodetect  
--hive_partitioning_mode=STRINGS <gcs_uri> <schema> >  
table_def.json
```

Note that, in the command above, the partition keys are being auto-detected, but not the data types of the partition keys, since we explicitly specify that they ought to be treated as strings and not the data types of the other columns, since we pass in an explicit schema.

We started this section by saying that a common use case for querying Hive partitions is to support cloud migration efforts where significant Hive workloads already exist, but allow future workloads to be implemented using BigQuery. While Apache Hive allows full management (reading and writing) of the data, BigQuery's external tables are read-only. Moreover, while BigQuery can handle the data being modified (e.g. from Hive) while a federated query is running, it does not currently support concepts such as reading data at a specific point in time. Because external tables in BigQuery have these limitations, it is better over time to move the data to BigQuery's native storage and rewrite the Hive workloads in BigQuery. Once the data is in BigQuery's native storage, features such as Data Manipulation Language (DML), streaming, clustering, table copies, etc. all become possible.

When to use federated queries and external data sources

Querying external sources is slower than querying data that is natively in BigQuery and so federated queries are typically not recommended in the long-term for frequently accessed data. There are, however, situations

where federated queries can be advantageous:

Carry out exploratory work using federated queries to determine how best to transform the raw data before loading it into BigQuery. For example, evidence of actual analysis workloads could dictate the transformations present in production tables. You might also treat original, external data source as staging, and use federated queries to transform the data and write it to production tables.

Keep data in Google Sheets if the spreadsheet will be edited interactively and use federated queries exclusively if the results of those queries need to reflect the live data in that sheet.

Keep data in an external data source if ad-hoc SQL querying of the data is relatively infrequent. For example, you might keep the data in Cloud Bigtable if the predominant use of that data is for low-latency, high-volume streaming ingest and if most queries on the data can be accomplished using key prefixes.

For large, relatively stable, well-understood datasets that will be updated periodically and queried often, BigQuery native storage is a better choice. In the rest of this section, we will look at the implementation details of each of these situations, starting with exploratory work using federated queries.

EXPLORATORY WORK USING FEDERATED QUERIES

Autodetect is a convenience feature that works by sampling a few (on the order of hundreds) rows of the input files to determine the type of a column. It is not fool-proof unless you are using self-describing file formats, such as Avro, Parquet, or ORC. To ensure that your ETL pipeline

works properly, you should verify the value of every row to ensure that the data type for each column is correct. For example, it is possible that a column contains integers except for a handful of rows which have floats. Then, it's quite likely that the autodetect will detect the column as being an integer since the chance of selecting one of the rows containing the floating point value is rather low. You won't learn there is a problem until you issue a query that does a table scan of this column's values.

The best practice is to use self-describing file formats, in which case you don't have to worry about how BigQuery interprets the data. If you need to use CSV or JSON, it is recommended that you explicitly specify a schema. While it is possible to specify the schema in an accompanying JSON file, it is also possible to pass in the schema on the command line of `bq mkdef` by creating a string with this format:

`FIELD1:DATATYPE1,FIELD2:DATATYPE2,...`

If you are unsure of the quality of your data, you should specify everything as a STRING. Note this is the default data type, so the formatting command becomes just

`FIELD1,FIELD2,FIELD3,...`

Why treat everything as a string? Even if you believe that some of the fields are integers and others are floats, it is best to validate this assumption. Define everything as a string and learn what transformations you have to carry out as you query the data and discover errors.

We can extract the column names by looking at the first line of the CSV file and use it to create a schema string of the desired format:²³

```
INPUT=gs://bigquery-oreilly-book/college_scorecard.csv
```

```
SCHEMA=$(gsutil cat $INPUT | head -1 | cut -b 4- )
```

If we are going to specify the schema, we should ask the first row be skipped and that the tool allow empty lines in the file. We can do this by piping the table definition through sed, a line editor²⁴:

```
LOC="--location US"
```

```
OUTPUT=/tmp/college_scorecard_def.json
```

```
bq $LOC \
```

```
mkdef \
```

```
--source_format=CSV \
```

```
--noautodetect \
```

```
$INPUT \
```

```
$SCHEMA \
```

```
| sed 's/"skipLeadingRows": 0/"skipLeadingRows": 1/g' \
```

```
| sed 's/"allowJaggedRows": false/"allowJaggedRows": true/g' \
```

```
> $OUTPUT
```

We define that we are operating in the US location and that we wish to save the output (the table definition) to the /tmp folder.

At this point, we have a table that we can query. Note that: (a) this table is defined on an external data source, so we are able to start querying the data without having to wait for the data to be ingested and (b) all the columns are strings -- we have not made any irreversible changes to the raw data.

Let's start our data exploration by trying to do a cast:

```
SELECT
```

```
    MAX(CAST(SAT_AVG AS FLOAT64)) AS MAX_SAT_AVG
```

```
FROM
```

```
`ch04.college_scorecard_gcs`
```

The query fails with the error message:

```
Bad double value: NULL
```

indicating that we need to handle the non-standard way that missing data is encoded in the file. In most CSV files, missing data is encoded as an empty string, but in this one, it is encoded as the string NULL.

We could fix this problem by checking before we do the cast:

```
WITH etl_data AS (
```

```
    SELECT
```

```
        SAFE_CAST(SAT_AVG AS FLOAT64) AS SAT_AVG
```

```
FROM

`ch04.college_scorecard_gcs`


)

SELECT

MAX(SAT_AVG) AS MAX_SAT_AVG

FROM

etl_data
```

Notice that we have started a WITH clause containing all the ETL operations that have to be performed on the dataset. Indeed, as we go through exploring the dataset and culminate with the query of the previous section, we would have learned that we need a reusable function to clean up numeric data:

```
CREATE TEMP FUNCTION cleanup_numeric(x STRING) AS

(
  IF ( x != 'NULL' AND x != 'PrivacySuppressed',
    CAST(x as FLOAT64),
    NULL )
);
```

```
WITH etl_data AS (

    SELECT

        INSTNM

        , cleanup_numeric(ADM_RATE_ALL) AS ADM_RATE_ALL

        , cleanup_numeric(FIRST_GEN) AS FIRST_GEN

        , cleanup_numeric(MD_FAMINC) AS MD_FAMINC

        , cleanup_numeric(SAT_AVG) AS SAT_AVG

        , cleanup_numeric(MD_EARN_WNE_P10) AS MD_EARN_WNE_P10

    FROM

        `ch04.college_scorecard_gcs`


)

SELECT

    *

FROM

    etl_data

WHERE
```

```
SAT_AVG > 1300  
AND ADM_RATE_ALL < 0.2  
AND FIRST_GEN > 0.1  
ORDER BY  
MD_FAMINC ASC  
LIMIT 10
```

At this point, we can export the cleaned up data (note the SELECT *) into a new table (note the CREATE TABLE) for just the columns of interest by running the query:

```
CREATE TEMP FUNCTION cleanup_numeric(x STRING) AS  
(  
  IF ( x != 'NULL' AND x != 'PrivacySuppressed',  
    CAST(x as FLOAT64),  
    NULL )  
);
```

```
CREATE TABLE ch04.college_scorecard_etl  
OPTIONS(description="Cleaned up college scorecard data") AS
```

```
WITH etl_data AS (

SELECT

INSTNM

, cleanup_numeric(ADM_RATE_ALL) AS ADM_RATE_ALL

, cleanup_numeric(FIRST_GEN) AS FIRST_GEN

, cleanup_numeric(MD_FAMINC) AS MD_FAMINC

, cleanup_numeric(SAT_AVG) AS SAT_AVG

, cleanup_numeric(MD_EARN_WNE_P10) AS MD_EARN_WNE_P10

FROM

`ch04.college_scorecard_gcs`


)

SELECT * FROM etl_data
```

It is also possible to script this out by removing the CREATE TABLE statement from the query above, invoking bq query and passing in a --destination_table.

ELT IN SQL FOR EXPERIMENTATION

In many organizations, there are many more data analysts than there are engineers. So, the needs of the data analysis teams usually greatly outpace

what the data engineers can deliver. In such cases, it can be helpful if data analysts themselves can create an experimental dataset in BigQuery and get started with analysis tasks.

The organization can then use the evidence of actual analytics workloads to prioritize what data engineers focus on. For example, as a data engineer, you may not yet know what fields out of a log file you need to extract. So, you might set up an external data source as an experiment and allow data analysts to query the raw data on Google Cloud Storage directly. If the raw log files are in JSON format with each of the rows having a different structure because the logs come from different applications, the analysts could define the entire log message as a single BigQuery string column and use `JSON_EXTRACT` and string manipulation functions to pull out the necessary data. At the end of a month, you could analyze the BigQuery query logs for which fields they actually did access, and how they did such access, and then build a pipeline to routinely load those fields into BigQuery.

If the original data is in a relational database management system, it is possible to export the data periodically as a tab-separated file to Google Cloud Storage. For example, if you are using MySQL with a database named `somedb`, the relevant command would be

```
mysql somedb < select_data.sql | \
gsutil cp - gs://BUCKET/data_$(date -u "+%F-%T").tsv
```

The `select_data.sql` would contain a query to pull just the most recent records (here, those from the previous 10 days):

```
select * from my_table
```

```
where transaction_date >= DATE_SUB(CURDATE(), INTERVAL 10  
DAY)
```

Given these periodically exported files, it is straightforward for an analyst to get started querying the data using federated queries. Once the value of the dataset is proven, the data can be loaded routinely and/or in real-time through a data pipeline.

The reason that this is not always suitable for operationalization is that it doesn't handle the case of mutations to the database. If data that is more than 10 days old is updated, the tab-separated dumps will be out-of-sync. Realistically, dumps to tsv files work only for small datasets (on the order of a few gigabytes) where the original database fields themselves do not need to be transformed or corrected before they are used for analytics queries.

If you do want to operationalize synchronization from an operational database to BigQuery, there are a number of third party companies that partner with Google, each with a menu of connectors and transformation options.²⁵ These tools can do CDC (change data capture) to allow you to stream changes from a database to a BigQuery table.

Interactive exploration and querying of data in Google Sheets

Google Sheets is part of GSuite, a set of productivity and collaboration tools from Google Cloud. It provides the means of creating, viewing, editing, and publishing spreadsheets. A spreadsheet contains tabular values, some of which are data and some of which are the result of computations carried out on the values of other cells. Google Sheets brings

spreadsheets online -- multiple people can collaboratively edit a spreadsheet and you can access it from a variety of devices.

Loading Google Sheets data into BigQuery

Google Sheets is an external source, so loading and querying a Google Sheets spreadsheet is a federated query -- it works similarly to querying a CSV file from Google Cloud Storage. We create a table definition in BigQuery to point to the data in Google Sheets, and then we query that table as if it were a native BigQuery table.

Let's start by creating a Google Sheets spreadsheet that we can query. Open up a web browser and in the URL navigation bar, type <https://sheets.new> -- visiting this URL will open up a blank spreadsheet.

Type in the following data (or download the corresponding CSV file²⁶ from GitHub and do a File > Import of the data into Google Sheets):

Student	Home state	SAT score
Aarti	KS	1111
Billy	LA	1222
Cao	MT	1333
Dalia	NE	1444

Now, navigate to the BigQuery section of the GCP console, create a dataset (if necessary) and create a table, specifying that source of the table is on Drive, its URL, and that it is a Google Sheet. Ask for the schema to be auto-detected:

Create table

Source

Create table from:

Select Drive URI: ?

File format:

Drive

https://docs.google.com/spreadsheets/d/1Mac0a_CblRnD2BjwYG

Google Sh... ▾

Destination

Project name

cloud-training-demos

Dataset name

advdata

Table type ?

External table

Table name

students

Schema

Auto detect

Schema and input parameters

i Schema will be automatically generated.

Advanced options ▾

Create table

Cancel

Figure 4-2. The create table dialog allows you to specify that the external datasource is Google Sheets.

Once this is done, the spreadsheet can be queried like any other BigQuery table:

```
SELECT * from advdata.students
```

Try changing the spreadsheet and verify that the returned results reflect the current state of the table (the results of federated queries on external datasets are not cached).

While querying a spreadsheet using SQL like this is possible, it is unlikely that you'd want to do this, since it's usually more convenient to use the interactive filtering and sorting options built into Google Sheets. For example, we can click on the “Explore” button and type in the natural language query: “Average SAT score for students in KS” and get back:

← Answers X

average SAT score of students in KS X

For students!A1:C5 EDIT

QUESTION

average SAT score of students in KS

ANSWER

Average of SAT score by Student for Home state of KS

Student	AVERAGE of SAT scor
Aarti	1111
Grand Total	1111

INSERT PIVOT TABLE

Explore

Figure 4-3. Natural language query in Google Sheets

There are several broad use cases for the tie between Google Sheets and BigQuery:

Populating a spreadsheet with data from BigQuery

Exploring BigQuery tables using Sheets

Querying Sheets data using SQL

Let's look at these three cases.

Populating a Google Sheets spreadsheet with data from BigQuery

The BigQuery data connector in Google Sheets allows you to query BigQuery tables²⁷ and use the results to populate a spreadsheet. This can be extremely useful when sharing data with non-technical users. In most businesses, nearly all office workers know how to read/interpret spreadsheets. They don't need to have anything to do with BigQuery or SQL to be able to use Google Sheets and work with the data in the sheet.

From Google Sheets, click on Data > Data Connectors > BigQuery, select your project and write a query to populate the Sheet from the BigQuery table of college scorecard data:

SELECT

*

FROM

ch04.college_scorecard_etl

Exploring BigQuery tables using Sheets

One of the reasons that you might want to populate a Google Sheets spreadsheet with data from a BigQuery table is that Sheets is a familiar interface for business users creating charts, formulas and pivot tables. For example, from the College Scorecard data in Sheets, it is quite straightforward to create a formula to rank colleges by the increase in median income experienced by their graduates:

In a new column, enter the formula:

```
=ArrayFormula(IF(ISBLANK(D2:D), 0, F2:F/D2:D))
```

Note that the spreadsheet has now been populated with the ratio of the value in the F-column to the value in the D-column, i.e. by the increase in income

From the Data menu, create a filter on the newly created column and turn off blanks and zeros

Sort the sheet Z-A based on this column

Selecting the first few rows of the sheet, we can quickly create a chart to showcase the best colleges in terms of economic improvement of the student body:

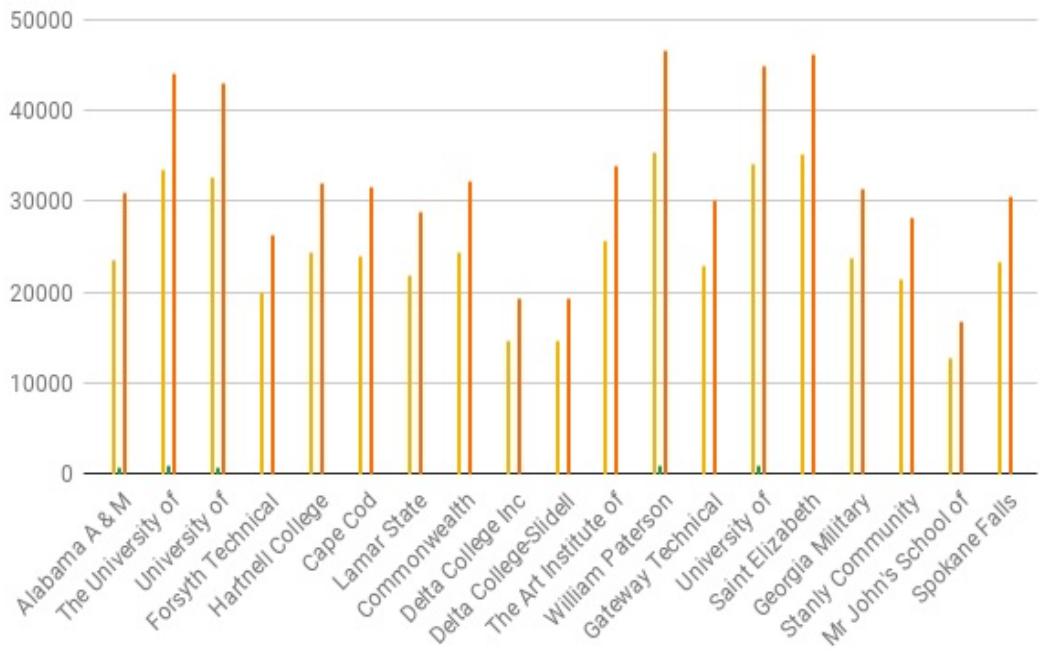
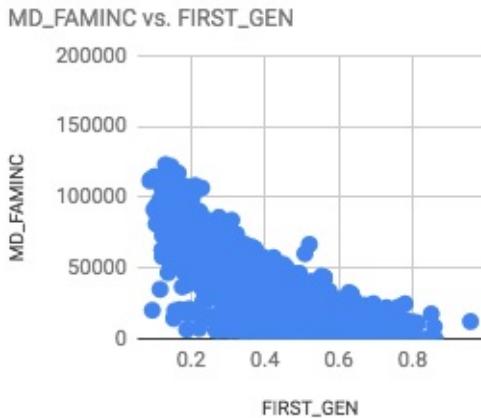


Figure 4-4. Chart that shows colleges that offer the greatest economic improvement to their graduates

In addition to interactively creating the charts you want, you can use the machine learning features of Google Sheets to further explore your data.

In Google Sheets, click on the Explore button and notice the charts that are automatically created through machine learning²⁸. For example, this automatically-generated insight captures a striking inequality:



For every increase of 0.1 in "FIRST_GEN", "MD_FAMINC" decreases by about 11400.

Figure 4-5. Sheets automatically generates the insight that colleges that serve first-generation college students also have poorer student bodies. For every 10% increase in first-generation college students, median family income decreases by \$11,400

The subsequent automatically-created chart puts the SAT_AVG in context:

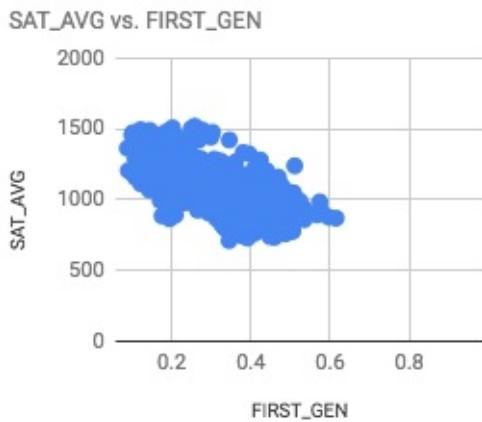


Figure 4-6. Colleges that serve first-generation college students tend to have lower SAT averages.

We can even ask for specific charts using natural language. Typing “histogram of sat_avg where first_gen more than 0.5” in the “Ask a question” box brings up the following answer:

QUESTION

histogram of `sat_avg` where `first_gen` more than 0.5

ANSWER

Distribution of SAT_AVG with FIRST_GEN > 0.5

Histogram ▾

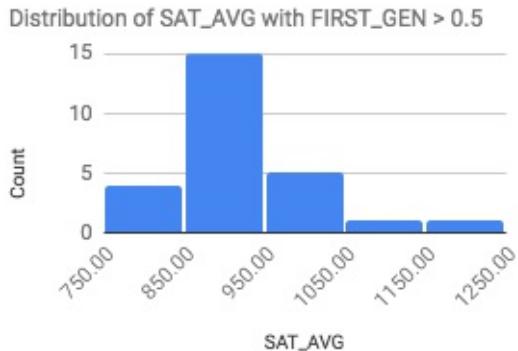


Figure 4-7. Getting the charts we want by simply asking for them in Google Sheets

Exploring BigQuery tables as a Data Sheet in Sheets

In the previous section, we loaded the entire BigQuery table into Google Sheets, but this was possible only because our college scorecard dataset was small enough. Loading the entire BigQuery table into Google Sheets is obviously not feasible for larger BigQuery tables.

Google Sheets does allow you to access, analyze, visualize, and share even large BigQuery datasets as a BigQuery Data Sheet. To try this out, start a new Google Sheets document and navigate via the menu to:

Data > Data Connectors > BigQuery Data Sheet

Choose your Cloud project (that should be billed) and navigate via the menu to the table you want to load into the Data Sheet:

`bigquery-public-data > usa_names > usa_1910_current > Connect`

This table contains nearly 6 million rows and is too large to load in its entirety. Instead, BigQuery acts as a cloud backend for the data shown in Sheets.

Unlike when loading the entire table into Sheets (as in the previous section), only the first 500 rows of a Data Sheet are loaded in the user interface. These 500 rows are best thought of as a preview of the full dataset. Another difference is in editing -- if the entire table is loaded, then Google Sheets holds a copy of the data, and so cells can be edited and the changed spreadsheet can be saved. On the other hand, if BigQuery is acting as a cloud back-end, then cells are not editable -- users can filter and pivot the BigQuery Data Sheet, but they can not edit the data. When users do filtering and pivoting, these actions happen on the entire BigQuery table, not just the preview that is shown in Sheets.

As an example of the kind of analysis that is possible, let's create a Pivot table by clicking on the Pivot table button. In the Pivot table editor, choose state as the Rows, year as the Columns. For Values, choose number and ask Sheets to summarize by COUNT_UNIQUE and show as Default (see snapshot below):



Unique names by state and year



Share

File Edit View Insert Format Data Tools Add-ons Help All changes saved i...



100%

\$

%

.0

.00

123

Arial

...

^



COUNTUNIQUE of number

	A	DA	DB	DC	DD
1	COUNTUNIQUE				
2	state	2013	2014	2015	2016
3	AK	43	42	42	36
4	AL	157	147	148	152
5	AR	112	114	111	101
6	AZ	197	200	195	191
7	CA	517	533	513	517
8	CO	165	164	169	161
9	CT	121	125	122	122
10	DC	52	50	51	54
11	DE	48	49	44	49
12	FL	323	333	331	341
13	GA	245	250	241	238
14	HI	53	51	51	50
15	IA	122	122	122	111
16	ID	75	75	72	73
17	IL	278	273	283	274
18	IN	193	203	202	197
19	KS	114	116	114	115
20	KY	158	154	152	149
21		160	155	153	150
22		183	183	192	187
23	MD	175	176	171	172
24	ME	60	57	60	59

Updated 11:56

Pivot table editor

usa_1910_current

Tr state

Order

Ascending

Sort by

state

 Show totals

Columns

Add

123 year

Order

Ascending

Sort by

year

 Show totals

Values

Add

123 number

Summarize by

COUNTUNIQ...

Show as

Default

+

≡

Sheet1



usa_1910_current

Pivot Table 1

Count: 54



Figure 4-8. Creating a Pivot table from a BigQuery Data Sheet

As you can see in the figure above, we get a table of the number of unique baby names in each state, broken down by year.

Joining Sheets data with a large dataset in BigQuery

Both BigQuery and Google Sheets are capable of storing and providing access to tabular data. However, BigQuery is primarily an analytics data warehouse while Google Sheets is primarily an interactive document. As we saw in the earlier sections, the familiarity of Sheets and the exploration and charting capabilities makes loading BigQuery data into Sheets very powerful.

However, there is a practical limitation on the size of BigQuery datasets that you can load into Sheets. For example, BigQuery holds information on Stack Overflow questions, answers, and users. Even with BigSheets, these petabyte-scale datasets are much too large to load directly into Google Sheets. However, it is still possible to write queries that join a small dataset in Sheets with such large datasets in BigQuery and proceed from there. Let's look at an example.

From the previous section, we have a spreadsheet with college scorecard data. Let's assume that we don't have the data already in BigQuery. We could create a table in BigQuery using the spreadsheet as a source, calling the resulting table college_scorecard_gs:

Create table

Source

Create table from:	Select Drive URI: https://docs.google.com/spreadsheets/d/1oEPjPY862GGfyxW41S	File format:
Drive		Google Sh...

Destination

Project name	Dataset name	Table type
cloud-training-demos	ch04	External table

Table name

Schema

Auto detect

Schema and input parameters

i Schema will be automatically generated.

Figure 4-9. Creating a Table in BigQuery using a Google Sheets spreadsheet as a source

Now, we can issue a query in BigQuery that joins this relatively small table (7700 rows) with a massive table consisting of Stack Overflow data (10 million rows) to find which colleges are most commonly listed in Stack Overflow users' profiles:

```
SELECT INSTNM, COUNT(display_name) AS numusers
```

```
FROM `bigquery-public-data`.stackoverflow.users ,
ch04.college_scorecard_gs
```

```
WHERE REGEXP_CONTAINS(about_me, INSTNM)
```

```
GROUP BY INSTNM
```

ORDER BY numusers DESC

LIMIT 5

This yields²⁹:

Row	INSTNM	numusers
1	Institute of Technology	2364
2	National University	332
3	Carnegie Mellon University	169
4	Stanford University	139
5	University of Mary	131

The first two entries are suspect³⁰, but it appears that Carnegie Mellon and Stanford are well-represented on Stack Overflow.

The result of this query is again small enough to load directly into Google Sheets and perform interactive filtering and charting. Thus, the SQL querying capability of Sheets data from BigQuery is particularly useful to join a small, human-editable dataset (in Google Sheets) with large, enterprise datasets (in BigQuery).

SQL queries on data in Cloud Bigtable

Cloud Bigtable is a fully managed NoSQL database service that scales up to petabytes of data. Cloud Bigtable is meant to be used in situations where some combination of low-latency (on the order of milliseconds), high-throughput (millions of operations per second), replication for high availability, and seamless scalability (from gigabytes to petabytes) is desired. Cloud Bigtable, therefore, finds heavy use in finance (trade reconciliation and analytics, payment fraud detection, etc.), Internet of

Things (IoT) applications (for centralized storage and processing of real-time sensor data), and advertising (real-time bidding, placement, and behavioral analysis). While Cloud Bigtable itself is available only on Google Cloud Platform, it supports the open-source Apache HBase API, enabling easy migration of workloads in a hybrid cloud environment.

NOSQL QUERIES BASED ON A ROW-KEY PREFIX

Cloud Bigtable provides high performance queries that look up rows or sets of rows that match a specific row-key, a row-key prefix, or a range of prefixes. Even though Cloud Bigtable requires an instance, consisting of one or more logical clusters, to be provisioned and available in your project, it uses that cluster only for compute (and not for storage) — the data itself are stored on Colossus and the nodes themselves need only to know about the location of row-ranges on Colossus. Because the data are not stored on the Cloud Bigtable nodes, it is possible to easily scale the Cloud Bigtable cluster up and down without expensive data migration.

In financial analysis, a common pattern is to store time-series data in Cloud Bigtable as they arrive in real-time and support low-latency queries on that data based on the row-key (e.g., all buy orders, if any, for GOOG stock in the last ten minutes). This allows dashboards that require recent data to provide automatic alerts and actions based on recent activity.

Cloud Bigtable also supports being able to obtain a range of data (e.g., all the buy orders for GOOG stock in any given day) quickly, a necessity for financial analytics and reporting. Prediction algorithms themselves have to be trained on historical data (e.g., the time-series of ask prices for GOOG over the past five years), and this is possible because machine learning frameworks like TensorFlow can read/write directly from Cloud Bigtable. These three workloads (real-time alerting, reporting, and ML training) can occur on the same data with the cluster potentially being scaled up-and-

down with workload spikes due to the separation of compute and storage.

All three workloads in the previous paragraph involve obtaining ask prices for Google stock — Cloud Bigtable will provide efficient retrieval of records if the row-key with which the time-series data are stored is of the form GOOG#buy#20190119-090356.0322234, i.e., the security name and the timestamp. Then, the queries of ask prices, whether over the previous ten minutes or the last 5 years, all involve requesting records that fall within a range of prefixes.

What if, though, we desire to perform ad hoc analytics over all the Cloud Bigtable data and our query is not of a form that will result in retrieving only a subset of records, i.e., our query does not filter based on the row-key prefix? Then, the NoSQL paradigm of Cloud Bigtable falls down, and it is better to resort to the ad hoc SQL querying capabilities offered by BigQuery, with the understanding that BigQuery results will be subject to higher latency.

AD HOC SQL QUERIES ON CLOUD BIGTABLE DATA

Just as BigQuery can directly query files in certain formats (CSV, Avro, etc.) in Google Cloud Storage by treating it as an external data source, BigQuery can directly query data in Cloud Bigtable. Just as with data in Cloud Storage, data in Cloud Bigtable can be queried using either a permanent table or a temporary table. A permanent table can be shared by sharing the dataset that it is part of; a temporary table is valid only for the duration of a query and so, cannot be shared.

A table in Cloud Bigtable is mapped to a table in BigQuery. In this section, we will use a time-series of point-of-sale data to illustrate — to follow along, run the script `setup_data.sh31` in the GitHub repository of

this book to create a Cloud Bigtable instance populated with some example data. Because the setup script creates a Cloud Bigtable instance with a cluster, remember to delete the instance once you are done.

We start by using the BigQuery user interface to create an external table in BigQuery to point to the data in Cloud Bigtable (see figure below). The location is a string of the form:

[https://googleapis.com/bigtable/projects/\[PROJECT_ID\]/instances/\[INSTANCE_ID\]/tables/\[TABLE_NAME\]](https://googleapis.com/bigtable/projects/[PROJECT_ID]/instances/[INSTANCE_ID]/tables/[TABLE_NAME]). The PROJECT_ID, INSTANCE_ID and TABLE_NAME refer to the project, instance and table in Cloud Bigtable.³²

Create Table

Source Data Create from source Create empty table

Repeat job [Select Previous Job](#) [?](#)

Location [Google Cloud Bigtable](#) <https://googleapis.com/bigtable/projects/cloud-training-d> [?](#)

File format [Cloud Bigtable](#) [?](#)

Destination Table

Table name ch04 . logs [?](#)

Table type External table [?](#)

Column Families [?](#)

Column Family and Qualifiers ?	Type ?	Encoding ?	Only Read Latest ?
sales	STRING	TEXT	TRUE
sales.itemid	STRING	TEXT	TRUE
sales.price	FLOAT	TEXT	TRUE
sales.qty	INTEGER	TEXT	TRUE

[Add Family](#) [Edit as Text](#)

Options

Ignore unspecified column families [?](#)

Read row key as string [?](#)

[Create Table](#)

Figure 4-10. Creating an external table in BigQuery to point to data in Cloud Bigtable. ³³

Data in Cloud Bigtable consists of records, each of which has a rowkey and data tied to the rowkey that is organized into column families, which are key-value pairs, where the key is the name of the column family and the value is a set of related columns.

Cloud Bigtable does not require every record to have every column family and every column allowed in a column family; in fact, the presence or absence of a specific column can itself be considered data. Therefore, BigQuery allows you to create a table that is tied to data in Cloud Bigtable without explicitly specifying any column names. If you do that, BigQuery exposes the values in a column family as an array of columns and each column as an array of values written at different time stamps.

In many cases, the column names are known beforehand, and if that is the case, it is better to supply the known columns in the table definition. In our case, we know the schema of each record in the logs-table of Cloud Bigtable:

A row-key which is the store id followed by the timestamp of each transaction

A column family named “sales” to capture sales transactions at the register

Within the sales column family, we capture:

The item id (a string)

The price at which the item was sold (a floating point number)

The number of items bought in this transaction (an integer)

Notice from the screenshot above that we have specified all this information in the column families section of the table definition.

Cloud Bigtable treats all data as simply byte strings, so the schema (string, float, integer) are meant more for BigQuery so that we can avoid having to

cast the values each time in our queries. Avoiding the cast is also the reason we ask for the row-key to be treated as a string. When the BigQuery table is created, each of the columns in Cloud Bigtable gets mapped to a column in BigQuery of the appropriate type:

sales. price	RECORD	NULLABLE	Describe this field...

With the BigQuery table in place, it is now possible to issue a good, old-fashioned SQL query to aggregate the total number of itemid 12345 that have been sold:

```
SELECT SUM(sales.qty.cell.value) AS num_sold
```

```
FROM ch04.logs
```

```
WHERE sales.itemid.cell.value = '12345'
```

IMPROVING PERFORMANCE

When we issue a federated query on data held in Google Cloud Storage, the work is carried out by BigQuery workers. On the other hand, when we

issue a federated query on data held in Cloud Bigtable, the work is carried out on the Cloud Bigtable cluster. The performance of the second query is, therefore, limited by the capacity of the Cloud Bigtable cluster and the load on it at the time that the query is being submitted.

As with any analytics query, the overall query speed also depends on the number of rows that need to be read and the size of the data being read. BigQuery does try to limit the amount of data that needs to be read by reading only the column families referenced in the query and Cloud Bigtable will split the data across nodes to take advantage of the distribution of row-key prefixes across the full dataset.

<warning>

If you have data that has a high update frequency or you need low-latency point lookups, Cloud Bigtable will provide the best performance for queries that can filter on a range of row-key prefixes. It can be tempting to think of BigQuery as providing a end-run around Cloud Bigtable performance by supporting ad-hoc point lookups of Cloud Bigtable data that aren't limited by row-keys. However, this pattern often gives disappointing performance, and you should benchmark it on your workload before deciding on a production architecture.

BigQuery stores data in a column-oriented order, which is optimized for table scans, while Cloud Bigtable stores data in a row-major order which is optimized for small reads and writes. Queries of external data stored in Cloud Bigtable do not provide the benefits of BigQuery's internal column-based storage and will be performant only if they read a subset of rows, not if they do a full table scan. Hence, you should be careful to ensure that your BigQuery federated queries filter on the Bigtable row key, otherwise

they will have to read the entire Cloud Bigtable every time.

</warning>

The knob you do have in your control is the number of nodes in your Cloud Bigtable cluster. If you are going to issue SQL queries against your Cloud Bigtable data routinely, monitor the Cloud Bigtable CPU usage and increase the number of Cloud Bigtable nodes if necessary.

Figure 4-11. Use a federated query to export selected to a BigQuery internal table and have your analytics workloads query the internal table.

As with federated queries over Google Cloud Storage, consider whether it is advantageous to set up an ELT pipeline when performing analytics over data in Cloud Bigtable — extract data from Cloud Bigtable using a federated query and load it into a BigQuery table for further analysis and transformations. This approach, illustrated in the diagram above, allows you to carry out your analytics workload in an environment where you are not at the mercy of the operational load on Cloud Bigtable. Analytics on an internal BigQuery table can be carried out on thousands of machines rather than a much smaller cluster. The analytics queries will, therefore, finish quicker in BigQuery (assuming that these analytics cannot be achieved using row-key prefixes) than if you use a federated queries on an external table. The drawback is, of course, that the extracted data is duplicated in both Cloud Bigtable and in BigQuery. Still, storage tends to be inexpensive, and the advantages of scale and speed might be enough compensation.

It is possible to schedule such data ingest into internal BigQuery tables to happen periodically. We will look at that in the next section.

<warning>

If you started a Cloud Bigtable instance to experiment with, delete it now so as to not run up charges.

</warning>

Transfers and exports

So far, we have looked at loading data on a one-off basis and avoiding the movement of data by using federated queries. In this section, we will look at turn-key services to transfer data into BigQuery from a variety of sources on a periodic basis.

Data Transfer Service

The BigQuery Data Transfer Service allows you to schedule recurring data loads from a variety of data sources into BigQuery. As with most BigQuery capabilities, you can access the BigQuery Data Transfer Service using the web UI, the command-line tool, and through a REST API. For repeatability, we will show you the command-line tool, but you can perform all of these steps programmatically using the REST API or interactively using the web UI.

Once you configure a data transfer, BigQuery will automatically load data on the schedule you specify. However, if there was a problem with the original data, you can also initiate data backfills to recover from any outages or gaps -- this is called refreshing, and it can be initiated from the web UI.

The Data Transfer Service supports loading data from a number of

Software as a Service (SaaS) applications such as Google Ads, Google Play, Amazon Redshift, and YouTube, as well as from Google Cloud Storage. We will look at how to set up routine ingest of files that show up in Cloud Storage, noting along the way any differences with data transfer of a SaaS dataset using YouTube channel reports as a running example.

DATA LOCALITY

As we discussed earlier in the chapter, BigQuery datasets are created in a specific region (such as `asia-northeast1`, which is Tokyo), or in a multi-region location (e.g., EU).³⁴ When you set up a data transfer service to a dataset, the transfer service will process and stage data in the same location as the target BigQuery dataset.

If your Cloud Storage bucket is in the same region as your BigQuery dataset, the data transfer does not incur charges. Transferring data between regions (e.g., from a Cloud Storage bucket in one region to a BigQuery dataset in a different region) will incur network charges, whether the transfer happens via loads, exports, or data transfers.

BigQuery Data Transfer Service needs to be enabled (you can do this from the BigQuery web UI) and you need to have been granted the `bigrquery.admin` role in order to create transfers and write data to the destination dataset.

SETTING UP DESTINATION TABLE

The data transfer service does not have the ability to create a new table, autodetect schema, etc. Instead, you need to provide a template table that has the desired schema. If you are writing all the data to a column-partitioned table, specify the partitioning column as a `TIMESTAMP` or `DATE` column when you create the destination table schema. Partitions

are covered in detail in Chapter X.

We will illustrate the process on the college scorecard dataset. We have it stored in the US multi-region, so you should create a dataset in the US multi-region if you want to try out the following steps.

In BigQuery, run the following query:

CREATE OR REPLACE TABLE

ch04.college_scorecard_dts

AS

SELECT * FROM ch04.college_scorecard_gcs

LIMIT 0

This is an example of a Data Definition Language (DDL) statement. It will save the result of the SELECT query (which will have no rows and not incur any charges) as a table named college_scorecard_dts in the ch04 dataset.

<sidebar>

Data definition language (DDL) statements allow you to create and modify BigQuery tables and views using standard SQL query syntax. For example the query:

CREATE TABLE

ch04.college_scorecard_valid_sat

AS

```
SELECT * FROM ch04.college_scorecard_gcs
```

```
WHERE LENGTH(SAT_AVG) > 0
```

will create a new table named ch04.college_scorecard_valid_sat and populate it with rows from ch04.college_scorecard_gcs where the SAT_AVG column is valid.

The CREATE TABLE DDL statement will return an error if the table already exists. Other options for the behavior when the table already exists include CREATE OR REPLACE (to replace the existing table) and CREATE IF NOT EXISTS (to leave the existing table as-is).

Instead of providing a SELECT statement, it is also possible to create an empty table with some desired schema:

```
CREATE TABLE ch04.payment_transactions
```

```
(
```

```
    PAYEE STRING OPTIONS(description="Id of payee"),
```

```
    AMOUNT NUMERIC OPTIONS(description="Amount paid")
```

```
)
```

By running the DDL query from the BigQuery command-line UI or invoking it using the REST API, it is possible to script out or programmatically create a table.

</sidebar>

CREATE TRANSFER JOB

On the command-line, issue the following command to set up a transfer job:

```
bq mk --transfer_config --data_source=google_cloud_storage \
--target_dataset=ch04 --display_name ch04_college_scorecard \
--params='{"data_path_template":"gs://bigquery-oreilly-
book/college_*.csv",
"destination_table_name_template":"college_scorecard_dts",
"file_format":"CSV", "max_bad_records":"10", "skip_leading_rows":"1",
"allow_jagged_rows":"true"}'
```

The above command specifies that the data source is to be Google Cloud Storage (if transferring from YouTube Channel, for example, the data source would be youtube_channel³⁵) and that the target dataset is ch04. The display name is used as a human readable name on various user interfaces to refer to the transfer job.

In the case of YouTube, the destination tables are automatically partitioned on the time of import and named appropriately. However, in the case of Cloud Storage, you will have to explicitly specify this in the destination table name. For example, specifying:

mytable_{run_time|"%Y%m%d"}

as the destination table name template indicates that the table name should

start with mytable and have the job runtime appended using the datetime formatting parameters specified.³⁶ A convenient shortcut is:

```
mytable_{run_date}
```

This simply uses the date in the format YYYYMMDD. It is also possible to supply a time offset. For example, to name the table based on the timestamp 45 minutes after the runtime, we could specify:

```
{run_time+45m"%Y%m%d"}_mytable_{run_time}"%H%M%S"}
```

This will yield a table name of the form 20180915_mytable_004500.

The parameters themselves are specific to the data source. In the case of transferring files from Google Cloud Storage, we should specify:

The input data path, with an optional wildcard

The destination table name template

The file format. The transfer service from Cloud Storage supports all the data formats that the federated querying capability supports (CSV, JSON, Avro, Parquet, etc.). In the case that the file format is CSV, we can specify CSV-specific options such as the number of header lines to skip.

The params for the YouTube channel data transfer include the page_id (in YouTube) and table_suffix (in BigQuery).

When you run the bq mk command above, you will get a URL as part of an OAuth2 workflow -- provide the necessary token by signing in via the browser, and the transfer job will be created.

You can also initiate a Data Transfer Service from the web UI. Initiate a transfer and choose the data source:

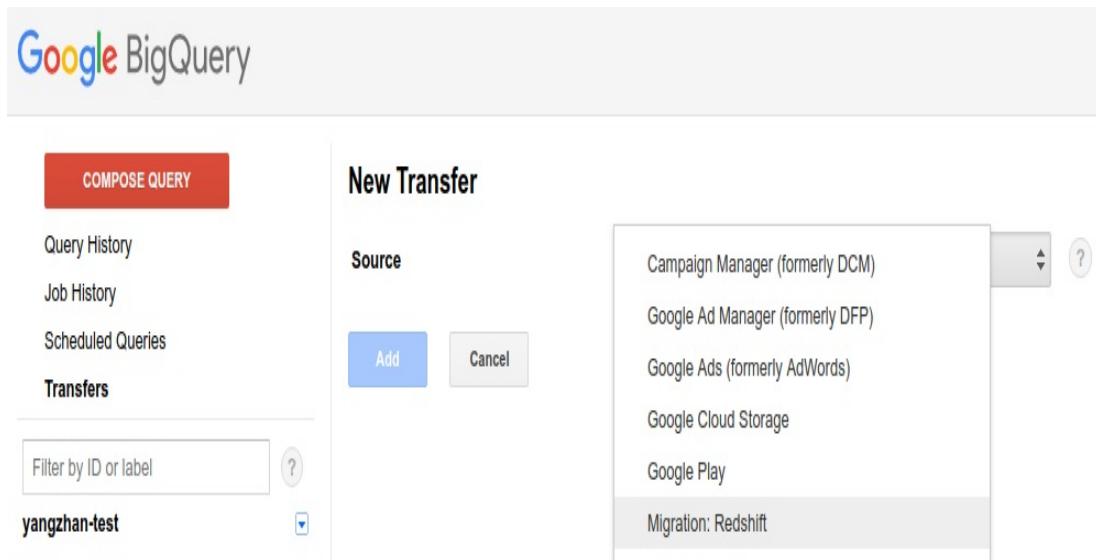


Figure 4-12. You can initiate a data transfer from the web UI as well

Note that we have not specified a schedule -- by default, the job will run every 24 hours, starting “now”. It is possible to edit the schedule of the transfer job from the BigQuery web UI:

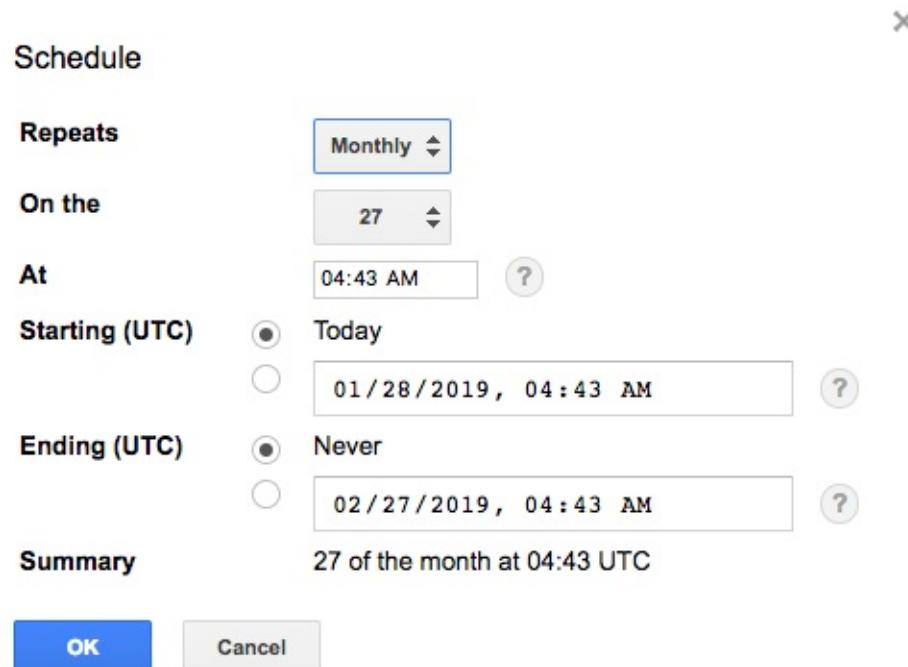


Figure 4-13. Editing the schedule of the transfer job from the web UI

The price of data transfer varies by the source. At the time of writing, data transfers from YouTube Channel costs \$5 per channel per month while data transfers from Cloud Storage incur no charge. However, because the transfer service uses load jobs to load Cloud Storage data into BigQuery, this is subject to the BigQuery limits on load jobs³⁷.

SCHEDULED QUERIES

BigQuery supports the scheduling of queries to run on a recurring basis and saving the results in BigQuery tables. In particular, you can use a federated query to extract data from an external data source, transform it, and load it into BigQuery. Since such scheduled queries can include Data Definition Language (DDL) and Data Manipulation Language (DML) statements, it is possible to build sophisticated workflows purely in SQL.

You can launch the dialog to set up a scheduled query by clicking on the “Schedule Query” button in the BigQuery UI³⁸:

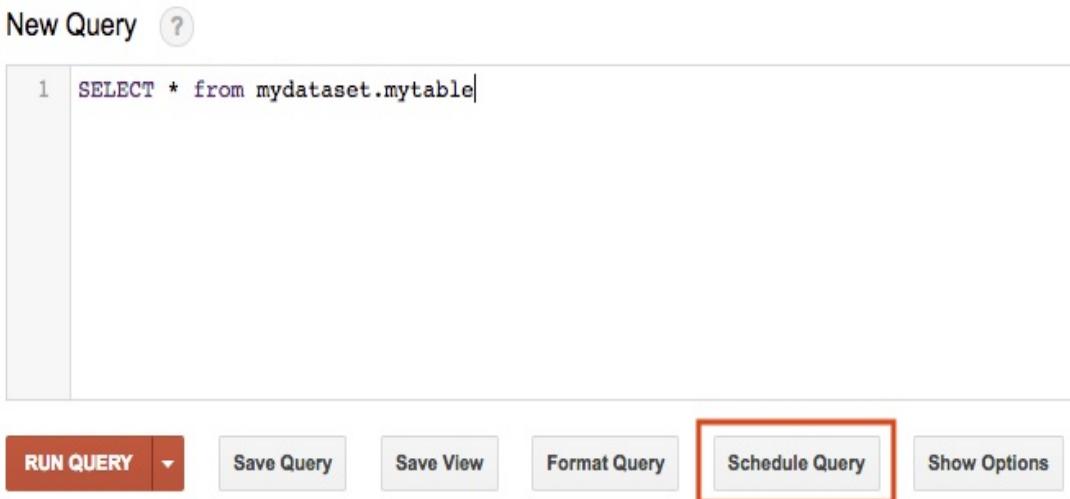


Figure 4-14. Schedule a query from the BigQuery user interface.

Scheduled queries are built on top of the Data Transfer Service, so many

of the features are similar. Thus, you can specify the destination table using the same parameter settings (e.g. run_date and run_time) as for the Data Transfer Service (see previous section).

CROSS-REGION DATASET COPY

BigQuery supports the scheduling of cross-region dataset copies via the Data Transfer Service. In the Data Transfer Service web user interface, choose “Cross Region Copy” as the Source. You will also have to specify as the source dataset the name of the dataset from which tables are to be copied into the destination dataset.

New Transfer

Source	Cross Region Copy	?
Display name	My Cross Region Copy	?
Schedule	every 24 hours	Reset Edit ?
Destination dataset	my_destination	?
Source dataset	my_source	?

Advanced

Add Cancel

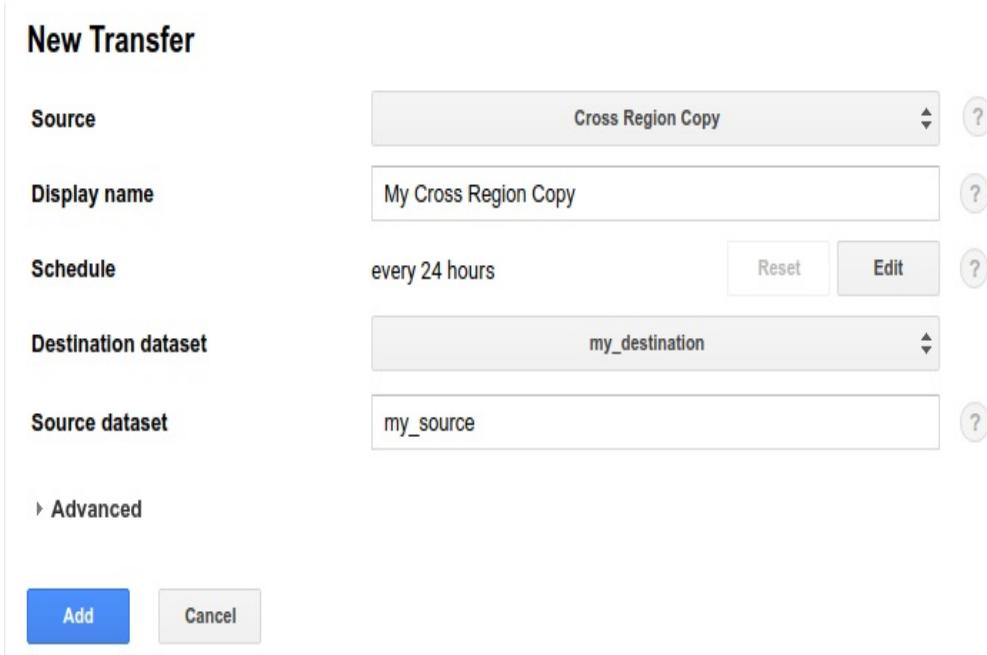


Figure 4-15. Initiate a scheduled cross-region dataset copy from the Data Transfer Service user interface by specifying that the source is a cross-region copy.

Because both the source and destination datasets are BigQuery datasets, the initiator needs to have permission to initiate data transfers, list tables in the source dataset, view the source dataset, and edit the destination dataset.

Export Stackdriver Logs

Log data from Google Cloud Platform VMs and services³⁹ can be stored, monitored and analyzed using Stackdriver Logging. Stackdriver Logging, thus, serves as a unified view of all the activity in your Google Cloud Platform account. It is helpful, therefore, to export Stackdriver and Firebase logs to BigQuery. This can be done using the command-line interface, using a REST API, or using the web UI.

The screenshot shows the Stackdriver Logs Viewer interface. At the top, there are three buttons: 'CREATE METRIC' (with a bar chart icon), 'CREATE EXPORT' (with an upward arrow icon), and a circular refresh icon. Below these are two dropdown menus: 'Filter by label or text search' and 'BigQuery'. A third dropdown menu, 'All logs', is shown to the right of the second one. A search bar is positioned above the log list. The main area displays a list of logs from the 'BigQuery' service. The logs are sorted by time, with the most recent at the top. Each log entry includes a timestamp, a severity level (indicated by an orange exclamation mark icon), the log type ('BigQuery'), and the log message (e.g., 'inser', 'list'). The log messages show BigQuery performing ingest jobs. The interface has a clean, modern design with a light gray background and white text on light blue buttons.

Time	Severity	Service	Message
2019-01-27 21:19:40.141 PST	!!	BigQuery	inser
2019-01-27 21:19:38.434 PST	!!	BigQuery	inser
2019-01-27 21:19:33.236 PST	!!	BigQuery	inser
2019-01-27 21:15:45.156 PST	!	BigQuery	list

Figure 4-16. To view logs from the BigQuery ingest jobs in the previous section, for example, you would go to the Stackdriver section of the GCP console.

To export logs, click on the Create Export button at the top of the Stackdriver Logs Viewer⁴⁰ and fill in the following information to export all the logs from the BigQuery service:

Select BigQuery and All Logs to view the logs from BigQuery. Do you see your recent activity?

A sink name, perhaps bq_logs

Specify the sink service: BigQuery because we want to export to

BigQuery

Specify the sink destination: ch04, the dataset we want to export to.

Let's look at the logs generated by running a query. Go to the BigQuery UI and try running a query:

SELECT

gender, AVG(tripduration / 60) AS avg_trip_duration

FROM

`bigquery-public-data`.new_york_citibike.citibike_trips

GROUP BY

gender

HAVING avg_trip_duration > 14

ORDER BY

avg_trip_duration

In the BigQuery UI, if you now do (change the date appropriately):

```
SELECT protopayload_auditlog.status.message FROM  
ch04.cloudaudit_googleapis_com_data_access_20190128
```

you will find a list of BigQuery log messages, including a message about reading the results of the above query. Depending on your date filter, you

should also see the logs corresponding to earlier operations that you carried out.

Note a few things about the export capability:

The schema and even the table name were set by Stackdriver. We simply specified the destination dataset.

The data were updated in near-real-time. This is an example of a streaming buffer -- a BigQuery table that is updated in real-time by Stackdriver (although the typical latency of BigQuery queries implies that the data you see are a few seconds old).

<warning>

To avoid running up charges for this streaming pipeline, go to the Stackdriver section of the console and delete the sink.

</warning>

Using Cloud Dataflow to read/write from BigQuery

As we've discussed, BigQuery supports federated querying from sources such as Google Sheets. Its data transfer service supports sources as Google Ads and YouTube. Products such as Stackdriver Logging and Firestore provide the ability to export their data to BigQuery.

What if you are using a product, such as MySQL, that does not provide an export capability and is not supported by the data transfer service? One option is to use Cloud Dataflow. Cloud Dataflow is a fully-managed service on Google Cloud Platform that simplifies the execution of data

pipelines that are built using the Open Source Apache Beam API by handling operational details such as performance, scaling, availability, security, and compliance so that users can focus on programming instead of managing server clusters. You can use Dataflow for transforming and enriching data both in streaming (real time) mode as well as in batch (historical) mode with the same reusable code across both streaming and batch pipelines.

USING A DATAFLOW TEMPLATE TO LOAD DIRECTLY FROM MYSQL

While you could write your own Cloud Dataflow pipelines (we'll do that in the next section), template Dataflow pipelines are available on GitHub for many common needs.⁴¹ Looking at the list of available templates, it appears that the Jdbc to BigQuery template might fit our requirements and allow us to transfer data from our MySQL database to BigQuery.

Visit the GCP console and navigate to the Cloud Dataflow section. Then, select “Create job from template”, choose “Jdbc to BigQuery”, and fill out the resulting form with information about the source database table in MySQL and the destination table in BigQuery:



Dataflow



Create job from template

Job name

Must be unique among running jobs. Use lowercase letters, numbers, and hyphens (-).

Cloud Dataflow template

A pipeline that reads from a Jdbc source and writes to a BigQuery table.

Required Parameters**Regional endpoint**

Choose where to deploy Cloud Dataflow workers and store metadata for the job.

Jdbc connection URL string

Url connection string to connect to the Jdbc source. E.g.
jdbc:mysql://some-host:3306/sampledb

Jdbc driver class name

Jdbc driver class name. E.g. com.mysql.jdbc.Driver

Jdbc source SQL query

Query to be executed on the source to extract the data. E.g. select * from sampledb.sample_table

BigQuery output table

BigQuery table location (<project>:<dataset>.<table_name>) to write the output to. The table's schema must match the source query schema.

GCS paths for Jdbc drivers

Comma separate GCS paths for Jdbc drivers. E.g. gs://<some-bucket>/driver_jar1.jar,gs://<some_bucket>/driver_jar2.jar

Temporary directory for BigQuery loading process

Example: gs://my-bucket/my-files/temp_dlr

Temporary Location

Path and filename prefix for writing temporary files. ex:
gs://MyBucket/tmp

Optional parameters

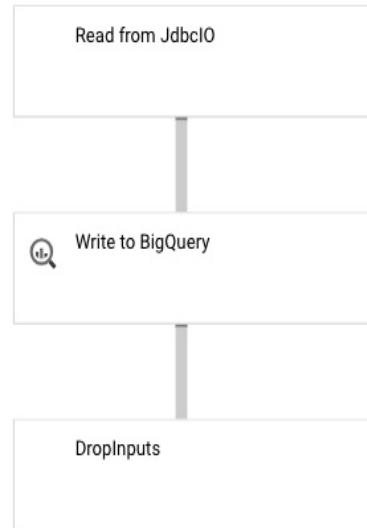


Figure 4-17. Create Dataflow job from template to transfer data from MySQL to BigQuery

When you click on the Run job button, a Dataflow job is launched. It will execute the JDBC query you specified, and write the resulting rows to BigQuery.

WRITING A DATAFLOW JOB

If you have a format for which there is no federated querying, no data transfer service, no export capability, and no prebuilt Dataflow template, then you can write your own Dataflow pipeline to load the data into BigQuery.

Even though both federated querying and a data transfer service exist for CSV files on Google Cloud Storage, we will use CSV files to demonstrate what this looks like. The code is written to the Apache Beam API and can be written in Python, Java, or Go. We'll show you Python.

The crux of the code is to extract the input data, transform it by extracting and cleaning up the desired fields, and load it into BigQuery:

```
INPATTERNS = 'gs://bigquery-oreilly-book/college_*.csv'
```

```
RUNNER = 'DataflowRunner'
```

```
with beam.Pipeline(RUNNER, options = opts) as p:
```

```
(p
```

```
| 'read' >> beam.io.ReadFromText(INPATTERNS, skip_header_lines=1)
```

```
| 'parse_csv' >> beam.FlatMap(parse_csv)
```

```
| 'pull_fields' >> beam.FlatMap(pull_fields)

| 'write_bq' >> beam.io.gcp.bigquery.WriteToBigQuery(bqtable,
bqdataset, schema=get_output_schema())

)
```

In the code above, we create a Beam pipeline specifying that it will be executed by Cloud Dataflow. Other options for the RUNNER include DirectRunner (executed on the local machine) and SparkRunner (executed by Apache Spark on a Hadoop cluster such as on Cloud Dataproc).

The first step of the pipeline is to read all the files that match the specified input patterns. These files can be on local disk, or on Google Cloud Storage. The data from the text files is streamed line-by-line to the next step of the pipeline where the parse_csv method is applied to each line:

```
def parse_csv(line):

try:

values = line.split(',')

rowdict = {}

for colname, value in zip(COLNAMES, values):

rowdict[colname] = value

yield rowdict

except:
```

```
logging.warn('Ignoring line ...')
```

The parse_csv method splits the line based on commas and converts the values into a dictionary where the key is the name of the column and the value is the value of the cell.

This dictionary is next sent to the method pull_fields which will extract the data of interest (the INSTNM column and a few numeric fields) and transform the data:

```
def pull_fields(rowdict):  
  
    result = {}  
  
    # required string fields  
  
    for col in 'INSTNM'.split(','):  
  
        if col in rowdict:  
  
            result[col] = rowdict[col]  
  
        else:  
  
            logging.info('Ignoring line missing {}', col)  
  
    return  
  
    # float fields  
  
    for col in
```

```
'ADM_RATE_ALL,FIRST_GEN,MD_FAMINC,SAT_AVG,MD_EARN  
_WNE_P10'.split(','):  
  
try:  
  
    result[col] = (float) (rowdict[col])  
  
except:  
  
    result[col] = None  
  
yield result
```

These dictionaries with the extracted fields are streamed into BigQuery row-by-row. The BigQuery sink (beam.io.gcp.bigquery.WriteToBigQuery) requires the name of the table, the name of the dataset and an output schema of the form:

INSTNM:string,ADM_RATE_ALL:FLOAT64,FIRST_GEN:FLOAT64...

.

The BigQuery table is created if needed, and rows are appended. Other options exist as well, for example to truncate the table (i.e., to replace it).

Running the Python program⁴² will launch a Dataflow job that will read the CSV file, parse it line-by-line, pull necessary fields, and write the transformed data to BigQuery.

While we demonstrated the Dataflow program on a batch pipeline (i.e. the input is not unbounded), essentially the same pipeline can be used to parse, transform and write out records received in a streaming mode (e.g.,

from Cloud Pub/Sub) as will be the case in many logging and IoT applications. The Dataflow approach thus provides a way to transform data on the fly and load it into BigQuery.

Note that Dataflow uses streaming inserts to load the data into BigQuery whether you are operating in batch mode or in streaming mode. Streaming inserts offer the advantage that the data shows up in a timely manner, into a streaming buffer, and can be queried even as the data is being written. The disadvantage is that unlike BigQuery load jobs, streaming inserts are not free. Recall that loading data into BigQuery may be free, but because of performance reasons, there are limits on how many load jobs you can do. Streaming inserts provide a way to avoid the limits and quotas placed on load jobs without sacrificing query performance.

USING THE STREAMING API DIRECTLY

While we presented Apache Beam on Cloud Dataflow as a way to extract, transform, and load data in BigQuery in streaming mode, it is not the only data processing framework that is capable of writing to BigQuery. If your team is more familiar with Apache Spark, writing the ETL pipeline in Spark and executing it on a Hadoop cluster (such as Cloud Dataproc on GCP) is a viable alternative to Dataflow. This is because client libraries exist for a variety of different languages, and BigQuery supports a streaming API.

We will cover the client library and streaming in greater detail in Chapter 5, but here is a snippet that illustrates how to load data using the Streaming API in Python once you have a client:

```
# create an array of tuples and insert as data becomes available
```

```
rows_to_insert = [  
    (u'U. Puerto Rico', 0.18,0.46,23000,1134,32000),  
    (u'Guam U.', 0.43,0.21,28000,1234,33000)  
]  
  
errors = client.insert_rows(table, rows_to_insert) # API request
```

As new data becomes available, the `insert_rows()` method on the BigQuery client is invoked. This method in turn invokes the REST API's `tabledata.insertAll` method. The data are held in a streaming buffer by BigQuery and available immediately for querying, although it can take up to 90 minutes for the data to become available for exporting.

Migrating on-premises data

In Chapter 1, we discussed that one of the key factors that makes BigQuery tick is the separation of compute and storage across a petabit-per-second bandwidth network. BigQuery works best on datasets that are within the data center and behind the Google Cloud firewall -- if BigQuery had to read its data from across the public internet or a slower network connection, it would not be as performant. Therefore, for BigQuery to work well, it is essential that the data be in the cloud.

BigQuery is a highly scalable analytics platform and is the recommended place to store structured data except those meant for real-time, transactional use. Since BigQuery is the place to store all structured data that will be used for data analytics, how do you migrate your on-premise data into BigQuery?

Data migration methods

If you have a good network with fast interconnect speeds to Google Cloud, then you could use `bq load` to load the data into BigQuery. As discussed in this chapter, it is preferable that the data being loaded are already present on Google Cloud Storage. You can use the command line tool `gsutil` to copy the data from on-premise to Cloud Storage.

When copying many files, especially large files, to Google Cloud Storage, use the `-m` option to enable multi-threading. Multi-threading will allow the `gsutil` tool to copy files in parallel:

```
gsutil -m cp /some/dir/myfiles*.csv gs://bucket/some/dir
```

Because it is likely that data continues to be collected, migrating data is often not a one-time process, but an ongoing one. One approach to handle this is to launch a Cloud Function to automatically invoke `bq load` whenever a file shows up on Cloud Storage.⁴³ As the frequency of file arrival increases (and as the size of those files grows smaller), you are better off using Cloud Pub/Sub⁴⁴ rather than Cloud Storage to store the incoming data as messages that will be processed by a Cloud Dataflow pipeline and streamed directly into BigQuery.

These three approaches -- `gsutil`, Cloud Functions, and Cloud Dataflow are shown in the first three rows of the table below and work when the network connection is quite good.

What you want to migrate	Recommended migration method
Relatively small files	<code>gsutil cp -m</code> <code>bq load</code>
Loading occasional (e.g. once a day) files	<code>gsutil cp</code>

into BigQuery when they are available	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 (DTS). Set this up in BigQuery. All the DTS functions work similarly.
Stackdriver Logging, Firestore, etc.	These tools provide capability to export to BigQuery. Set this up in the other tool (i.e., Stackdriver, Firestore, etc.).

Table 4-2. The recommended migration method for different situations.

While data migration using gsutil to stage the data on Cloud Storage, and then invoking bq load may be an easy effort if you only have a few small datasets, it is more difficult if you have many datasets or if your datasets are large. As data size increases, the incidence of errors also increases. Therefore, migrating large datasets requires attention to details like checksumming data at capture and ingest, working with firewalls so they don't block transfers or drop packets, avoiding exfiltration of sensitive data, and ensuring that your data is encrypted and protected against loss during and after migration.

Another issue with the gsutil method is that it is quite likely that your

business will probably not be able to dedicate bandwidth for data transfers since such dedicated bandwidth is often too expensive and will disrupt routine operations that convey data over the corporate network.

In cases where it is not possible to copy data to Google Cloud because of data size or network limitations, consider using the Transfer Appliance. This is a rackable, high-capacity storage server that is shipped to you -- fill it up and ship it back to Google Cloud or one of its authorized partners. The Transfer Appliance is best used for lots of data (hundreds of terabytes to petabytes), where your network situation won't meet transfer demands.

If your data is held not on-premise but in another public cloud (such as in an Amazon Web Services S3 bucket), you can use the Cloud Storage Transfer Service to migrate the data.

Common use cases include running an application on Amazon Web Services, but analyzing its log data in BigQuery. The Cloud Storage Transfer Service is also a great way to transfer large amounts of data between regions at Google.

The BigQuery Transfer Service automates loading data into BigQuery from Google properties like YouTube, Google Ads, etc. Other tools, such as Stackdriver Logging and Firestore, provide the capability to export to BigQuery.

While you can carry out data migration yourself, it is unlikely to be something that your IT department has much experience in since migration is often just a one-time task. It might be advantageous to use a Google Cloud authorized partner⁴⁵ to carry out the data migration.

Summary

The bq command-line tool provides a single point of entry to interact with the BigQuery service on Google Cloud Platform. Once your data is on Google Cloud Storage, you can do a one-time load of the data using the bq load utility. It supports schema autodetection, but can also use a specific schema that you supply. Depending on whether your load job is CPU-bound or I/O-bound, it might be advantageous to either compress the data or leave it uncompressed.

It is possible to leave the data in-place, specify the structure of the data, and use BigQuery as just the query engine. These are called external datasets, and queries over external datasets are called federated queries. Use federated queries for exploratory work, or where the primary use of the data is in the external format (e.g. low-latency queries in Cloud Bigtable or interactive work in Sheets). For large, relatively stable, well-understood datasets that will be updated periodically and queried often, BigQuery native storage is a better choice. Federated queries are also useful in an ELT workflow where the data is not yet well understood.

It is possible to set up a scheduled transfer of data from a variety of platforms into BigQuery. Other tools also support mechanisms to export their data into BigQuery. For routine loading of data, consider using Cloud Functions; for ongoing, streaming loads, use Cloud Dataflow. It is also possible to schedule queries (including federated queries) to run periodically, and have these queries load data into tables.

¹ Six to eight changes every decade. See
[https://en.wikipedia.org/wiki/Territorial_evolution_of_the_United_States#1946%E2%80%93_Present_\(Decolonization\)](https://en.wikipedia.org/wiki/Territorial_evolution_of_the_United_States#1946%E2%80%93_Present_(Decolonization))

² See: <https://abc7ny.com/news/border-of-north-and-south-carolina-shifted-on-january->

1st/1678605/ and <https://www.nytimes.com/2014/08/24/opinion/sunday/how-the-carolinas-fixed-their-blurred-lines.html>

- 3 See <https://catalog.data.gov/dataset/college-scorecard/resource/77d2e376-c5bb-46d7-a985-e214e009e36e>
- 4 See `04_load/college_scorecard.csv.gz` in <https://github.com/GoogleCloudPlatform/bigquery-oreilly-book/>.
- 5 Here are detailed steps:
Open CloudShell in your browser by visiting <https://console.cloud.google.com/cloudshell>
In the terminal window, type: `git clone https://github.com/GoogleCloudPlatform/bigquery-oreilly-book`
Navigate to the folder containing the college scorecard file: `cd bigquery-oreilly-book/04_load`
Now, type the command `zless college_scorecard.csv.gz` and use the spacebar to page through the data. Type the letter `q` to quit.
- 6 This is set through a dropdown box in the GCP web console, or when you last did a `gcloud init`. Typically, a project corresponds to a workload or to a small team.
- 7 See <https://cloud.google.com/bigquery/docs/locations> for an updated list.
- 8 The autodetect algorithm continues to handle more and more corner cases, and so this might not happen for you. In general, though, schema autodetection will never be perfect.
Regardless of the details of what aspect of the schema is not correctly captured, our larger point is this: use the autodetected schema as a starting point and build on top of it, as we do in this section.
- 9 It is possible for an integer column to be nullable, but the file is encoding null values in a non-standard way. BigQuery is interpreting the text “NULL” as a string, which is why the load fails.
- 10 The NULL string in the file represents a lack of data for that field and this is what a NULL value in our BigQuery table should mean as well.
- 11 As we reminded you in earlier chapters: we believe all mentions of price to be correct as of the writing of this book, but please do refer to the relevant policy and pricing sheets (<https://cloud.google.com/bigquery/pricing>) as these are subject to change.
- 12 See <https://cloud.google.com/bigquery/streaming-data-into-bigquery>
- 13 You can also do so using the `ALTER TABLE SET OPTIONS` statement, for example:

```
ALTER TABLE ch04.college_scorecard
SET OPTIONS (
  expiration_timestamp=TIMESTAMP_ADD(CURRENT_TIMESTAMP(), INTERVAL 7
  DAY),
  description="College Scorecard table that expires seven days from now"
)
```

See https://cloud.google.com/bigquery/docs/reference/standard-sql/data-definition-language#alter_table_set_options_statement for more details.

- 14 At the time of writing, this capability does not exist in the “new” UI, and so has to be accessed through the bq command-line tool.
- 15 Strings are sorted lexically. If stored as a string, “100” would be less than “20” for the same reason that “abc” comes before “de” when the two strings are sorted. When sorted numerically, 20 is less than 100 as one would expect.
- 16 The file contains D/M/YYYY while the standard format for a date is YYYY-MM-DD (which matches ISO-8601). While auto-detect can look at multiple rows and infer whether 12/11/1965 is the 12th of November or the 11th of December, we don’t want the schema-based BigQuery load making any such assumptions. The transformation pipeline that we build later in this chapter will convert the dates into the standard format. For now, let’s just treat it as a string.
- 17 See <https://avro.apache.org/>
- 18 Newline delimited JSON often goes by the name of jsonl, or “JSON lines format”.
- 19 See https://blog.twitter.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet.html. In Chapter 6, we’ll discuss Capacitor, BigQuery’s back-end storage format which is the successor to ColumnIO.
- 20 Try it out by running the load_*.sh scripts in the 04_load of the GitHub repository of this book.
- 21 See https://cloud.google.com/bigquery/quotas#load_jobs
- 22 See <https://hive.apache.org/>
- 23 This particular file includes a “byte order marker” (\u0eff) as its first character, so we remove the first few bytes using cut:
cut -b 4-
- 24 The complete script is called load_external_gcs.sh and is located in the GitHub repository for this book (<https://github.com/GoogleCloudPlatform/bigquery-oreilly-book>).
- 25 These partners include Alooma, Informatica, and Talend. Please see <https://cloud.google.com/bigquery/partners/> for a full, and current, list of BigQuery partners.
- 26 See https://github.com/GoogleCloudPlatform/bigquery-oreilly-book/blob/master/04_load/students.csv
- 27 At the time of writing, there are size restrictions on the BigQuery table.
- 28 Due to continuing changes and improvements in the products, the graphs you get might be different.

- 29 This query will be slow because we are doing a regular expression match and doing so 77 billion times.
- 30 Most likely, the rows include data from multiple colleges such as National University of Singapore, National University of Ireland, etc.
- 31 See https://github.com/GoogleCloudPlatform/bigquery-oreilly-book/tree/master/04_load/bigtable
- 32 If you followed along by running the setup_data.sh in the GitHub repository, the project_id will be your unique project ID, the instance_id will be bqbook-instance and the table_name will be logs-table.
- 33 At the time of writing, this capability was available only in the “old” UI at <https://bigquery.cloud.google.com/> and not in the “new” UI that is part of the GCP web console (<https://console.cloud.google.com/bigquery>).
- 34 See <https://cloud.google.com/bigquery/docs/locations> for BigQuery dataset locations and <https://cloud.google.com/storage/docs/bucket-locations> for Cloud Storage locations.
- 35 See <https://cloud.google.com/bigquery/docs/youtube-channel-transfer>
- 36 For a list of available formatting options, see the BigQuery docs for formatting datetime columns, see <https://cloud.google.com/bigquery/docs/reference/standard-sql/functions-and-operators#supported-format-elements-for-datetime>
- 37 See https://cloud.google.com/bigquery/quotas#load_jobs
- 38 At the time of writing, this is available only in the “classic UI”.
- 39 And also from VMs and services running in Amazon Web Services.
- 40 Go to <https://console.cloud.google.com/logs/>
- 41 See <https://github.com/GoogleCloudPlatform/DataflowTemplates>
- 42 See 04_load/dataflow.ipynb in the GitHub repository.
- 43 How to do so programmatically will be covered in Chapter 5.
- 44 A message bus service. See: <https://cloud.google.com/pubsub/>
- 45 See <https://cloud.google.com/bigquery/partners/>. As this book was being written, Google Cloud announced an intent to acquire Alooma, a provider of cloud migration services. See <https://cloud.google.com/blog/topics/inside-google-cloud/google-announces-intent-to-acquire-alooma-to-simplify-cloud-migration>.

Chapter 5. Developing with BigQuery

So far, we have mostly used the BigQuery web user interface and the `bq` command-line to interact with BigQuery. In this chapter, we will look at ways to programmatically interact with the service. This can be useful in order to script out or automate tasks that involve BigQuery. Programmatic access to BigQuery is also essential when developing applications, dashboards, scientific graphics, and machine learning models where BigQuery is only one of the tools being used.

We will start by looking at BigQuery client libraries that allow you to programmatically query and manipulate BigQuery tables and resources. Although you can programmatically access BigQuery using these low-level APIs, you want to be aware of customizations and higher-level abstractions available for particular environments (Jupyter notebooks and shell scripts). These customizations, which we will cover in the second half of this chapter, are easier to use, handle error conditions appropriately, and cut out a lot of boilerplate code.

Developing programmatically

The recommended approach for accessing BigQuery programmatically is to use the Google Cloud Client Library in your preferred programming language. The REST API is helpful in understanding what happens under the covers when you send a request to the BigQuery service, but the BigQuery client library is more practical. So, feel free to skim the section

on the REST API.

Accessing BigQuery via REST API

Sending a query to the BigQuery service can be accomplished by making a direct HTTP request to the server because BigQuery, like all Google Cloud services, exposes a traditional JSON/REST interface.¹ JSON/REST is an architectural style of designing distributed services where each request is stateless (i.e., the server does not maintain session state or context, and instead, each request contains all the necessary information) and both request and response objects are self-describing text format called JSON. Because HTTP is a stateless protocol, REST services are particularly well-suited to serving over the web. JSON maps directly to in-memory objects in languages like JavaScript and Python.

REST APIs provide the illusion that the objects referred to by the API are static files in a collection and provide CRUD (Create, Read, Update, Delete) operations that map to HTTP verbs. For example, to create a table in BigQuery you use POST, to inspect the table you use GET, to update it you use PATCH, and to delete it you use DELETE. There are some methods, like Query, that don't map exactly to CRUD operations, so these are often referred to as RPC (Remote Procedure Call)-style methods.

All BigQuery URIs start with the prefix

<https://www.googleapis.com/bigquery/v2>. Notice that it uses HTTPS, rather than HTTP, which stipulates that requests should be encrypted on the wire. The V2 part of the URI is the version number. While some Google APIs revise their version number frequently, BigQuery has adamantly stuck with v2 for several years, and is likely to do so for the foreseeable future.²

DATASET MANIPULATION

The REST interface involves issuing HTTP requests to specific URLs. The combination of the HTTP request method (GET, POST, PUT, PATCH, and DELETE) and URL specifies the operation to be performed. For example, to delete a dataset, the client would issue an HTTP DELETE request to the URL (inserting the id of the dataset and the project it is held in):

.../projects/<PROJECT>/datasets/<DATASET>

where the ... refers to <https://www.googleapis.com/bigquery/v2>. All BigQuery REST URLs are relative to this path.

<**note**>

When you type in a URL in a web browser's navigation toolbar, the browser issues an HTTP GET to that URL. To issue an HTTP DELETE, you need a client that gives you the option of specifying the HTTP method to invoke. One such client tool is curl, and we'll look at how to use it shortly.

</**note**>

We know this because the BigQuery REST API documentation³ specifies the HTTP request details :

Datasets

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 <code>/projects/<i>projectId</i>/datasets/<i>datasetId</i></code>	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 <code>/projects/<i>projectId</i>/datasets/<i>datasetId</i></code>	Returns the dataset specified by datasetID.
insert	POST <code>/projects/<i>projectId</i>/datasets</code>	Creates a new empty dataset.
list	GET <code>/projects/<i>projectId</i>/datasets</code>	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 issuing an HTTP DELETE request to the URL /projects/<PROJECT>/datasets/<DATASET> will result in the dataset being deleted if it is empty.

TABLE MANIPULATION

Deleting a table, similarly, involves issuing an HTTP DELETE to the URL:

`.../projects/<PROJECT>/datasets/<DATASET>/tables/<TABLE>`

Note that both these requests employ the HTTP DELETE method, and it is the URL path that differentiates them. Of course, not everyone who visits the URL will be able to delete the dataset or table. The request will

succeed only if the request includes an access token, and if the access token (covered shortly) represents appropriate authorization in the BigQuery or Cloud Platform scopes.⁴

As an example of a different HTTP method type, it is possible to list all the tables in a dataset by issuing an HTTP GET to:

```
.../projects/<PROJECT>/datasets/<DATASET>/tables
```

Listing the tables in a dataset requires only a read-only scope -- full access to BigQuery (such as to delete tables) is not necessary, although of course, the greater authority (e.g. BigQuery scope) also provides the lesser permissions.

We can try this using a Unix Shell:⁵

```
#!/bin/bash
```

```
PROJECT=$(gcloud config get-value project)
```

```
access_token=$(gcloud auth application-default print-access-token)
```

```
curl -H "Authorization: Bearer $access_token" \
```

```
-H "Content-Type: application/json" \
```

-X GET

```
"https://www.googleapis.com/bigquery/v2/projects/$PROJECT/datasets/c  
h04/tables"
```

The access token is a way to get application-default credentials -- these are

temporary credentials that are issued by virtue of being logged into the Cloud SDK. The access token is placed into the header of the HTTP request, and because we want to list the tables in the dataset ch04, we issue a GET request to the URL

`.../projects/$PROJECT/datasets/ch04/tables`

using the curl command.

<tip>

This chapter shows how to access BigQuery programmatically through a client API. However, consider whether you are better off using SQL queries to get this information. For example, creating and deleting tables can be accomplished via CREATE TABLE or DROP TABLE statements respectively. Using SQL might allow you to stay within the tool you use to explore and analyze data without having to integrate a programming or scripting tool into your workflow.

Listing the tables in the dataset corresponding the previous chapter (ch04) can be achieved in SQL by querying an INFORMATION_SCHEMA view:

`SELECT`

`table_name, creation_time`

`FROM`

`ch04.INFORMATION_SCHEMA.TABLES`

The use of INFORMATION_SCHEMA and SQL alternatives to the client API will be covered in Chapter 8. However, here is a quick table mapping between client API functions and SQL:

Client API Capability	SQL alternative
Creating tables (or views: just replace TABLE by VIEW)	CREATE TABLE CREATE TABLE IF NOT EXISTS CREATE OR REPLACE TABLE
Update table (or view)	ALTER TABLE SET OPTIONS ALTER TABLE IF EXISTS SET OPTIONS
Update table data	INSERT INTO DELETE FROM UPDATE MERGE
Delete table (or view)	DROP TABLE
Dataset metadata	Query: INFORMATION_SCHEMA.SCHEMATA INFORMATION_SCHEMA.SCHEMATA_OPTIONS
Table (or view) metadata	Query: INFORMATION_SCHEMA.TABLES INFORMATION_SCHEMA.TABLE_OPTIONS INFORMATION_SCHEMA.COLUMNS INFORMATION_SCHEMA.COLUMN_FIELD_PATHS

In many situations, using SQL might be a better alternative from a tooling and familiarity perspective.

</tip>

QUERYING

In some cases, issuing an HTTP GET request to a BigQuery URL is not enough. More information is required from the client. In such cases, the API requires that the client issue an HTTP POST and send along a JSON

request in the body of the request.

For example, to run a BigQuery SQL query and obtain the results, issue an HTTP POST request to:

.../projects/<PROJECT>/queries

and send in a JSON of the form:

```
{  
  "useLegacySql": false,  
  "query": "${QUERY_TEXT}"  
}
```

where QUERY_TEXT is a variable that holds the query to be performed:

```
read -d " QUERY_TEXT << EOF
```

```
SELECT
```

```
start_station_name
```

```
, AVG(duration) as duration
```

```
, COUNT(duration) as num_trips
```

```
FROM `bigquery-public-data`.london_bicycles.cycle_hire
```

```
GROUP BY start_station_name
```

```
ORDER BY num_trips DESC
```

```
LIMIT 5
```

```
EOF
```

We are using the heredoc⁶ syntax in bash to specify that the string EOF marks where our query begins and ends.

The curl request now is a POST that includes the request as its data:⁷

```
curl -H "Authorization: Bearer $access_token" \  
-H "Content-Type: application/json" \  
-X POST \  
-d "$request" \  
"https://www.googleapis.com/bigquery/v2/projects/$PROJECT/queries"
```

where \$request is a variable that holds the JSON payload (including the query text).

The response is a JSON message that contains the schema of the resultset and five rows, each of which is an array of values. In this case, the schema is:

```
"schema": {
```

```
  "fields": [
```

{

“name”: “start_station_name”,

“type”: “STRING”,

“mode”: “NULLABLE”

},

{

“name”: “duration”,

“type”: “FLOAT”,

“mode”: “NULLABLE”

},

{

“name”: “num_trips”,

“type”: “INTEGER”,

“mode”: “NULLABLE”

}

]

},

and the first row is:

{

“f”: [

{

“v”: “Belgrave Street , King’s Cross”

},

{

“v”: “1011.0766960393793”

},

{

“v”: “234458”

}

]

,

The f stands for fields, and the v for values. Each row is an array of fields, and each field has a value. This means that the highest number of trips was

at the station on Belgrave St, where the average duration of trips was 1011 sec and the total number of trips was 234458.

LIMITATIONS

In the case just considered, the query happens to finish within the default timeout period (it's possible to specify a longer timeout), but what if the query takes longer? Let's simulate this by artificially lowering the timeout and disabling the cache:⁸

```
{  
  "useLegacySql": false,  
  
  "timeoutMs": 0,  
  
  "useQueryCache": false,  
  
  "query": "${QUERY_TEXT}"  
}
```

Now, the response no longer contains the rows of the result set. Instead, we get a promissory note in the form of a job id:

```
{  
  "kind": "bigquery#queryResponse",  
  
  "jobReference": {  
  
    "projectId": "cloud-training-demos",  
  }
```

```
    "jobId": "job_gv0Kq8nWzXIkuBwoxsKMcTJIVbX4",
```

```
    "location": "EU"
```

```
},
```

```
    "jobComplete": false
```

```
}
```

We are expected to now get the status of the jobId using the REST API by sending a GET request to:

```
.../projects/<PROJECT>/jobs/<JOBID>
```

until the response has jobComplete set to true. At that point, we can obtain the query results by sending a GET request to:

```
.../projects/<PROJECT>/queries/<JOBID>
```

Sometimes, the query results are too large to be sent in a single HTTP response. Instead, the results are provided to us in chunks. Recall, however, that REST is a stateless protocol and the server does not maintain session context. Therefore, the results are actually stored in a temporary table that is maintained for 24 hours. This temporary table of results can be paged through by the client using a page token that serves as a bookmark for each call to get query results. In addition to all this complexity, add in the possibility of network failure and the necessity of retries, and it becomes clear that the REST API is quite difficult to program against. Therefore, even though the REST API is accessible from any language that is capable of making calls to web services, we typically

look for a higher level API.

Google Cloud Client library

The Google Cloud Client Library for BigQuery is the recommended option for accessing BigQuery programmatically. At the time of writing, a client library is available for seven programming languages: Go, Java, Node.js, Python, Ruby, PHP, and C++. Each client library provides a good developer experience by following the convention and typical programming style of the programming language.

You can install the BigQuery client library using pip (or easy_install):

```
pip install google-cloud-bigquery
```

To use the library, first instantiate a client (this takes care of the authentication⁹ that was accomplished using an access token when directly invoking the REST API):

```
from google.cloud import bigquery
```

```
bq = bigquery.Client(project=PROJECT)
```

The project passed into the Client is the globally unique name of the project that will be billed for operations carried out using the bq object.

<tip>

A Python notebook with all the code in this section may be found at https://github.com/GoogleCloudPlatform/bigquery-oreilly-book/blob/master/05_devel/bigquery_cloud_client.ipynb. Use the

notebook as a source of Python snippets to try out in your favorite Python environment.

The API documentation for the BigQuery client library is here:

<https://googleapis.github.io/google-cloud-python/latest/bigquery/reference.html>. Because it is impossible to cover the full API, we strongly suggest that you have the documentation open in a browser tab as you read through the following section. As you read the Python snippets, see how you could discover the Python methods to invoke.

</tip>

DATASET MANIPULATION

To view information about a dataset using the BigQuery client library, use the `get_dataset` method:

```
dsinfo = bq.get_dataset('bigquery-public-data.london_bicycles')
```

If the project name is omitted, the project passed into the Client at the time of construction is assumed, so that

```
dsinfo = bq.get_dataset('ch04')
```

will return an object with information about the dataset we created in the previous chapter.

Dataset information

Given the `dsinfo` object, it is possible to extract different attributes of the dataset. For example:

```
print(dsinfo.dataset_id)
```

```
print(dsinfo.created)
```

on the ch04 object yields:

```
ch04
```

```
2019-01-26 00:41:01.350000+00:00
```

```
while
```

```
print('{} created on {} in {}'.format(
```

```
dsinfo.dataset_id, dsinfo.created, dsinfo.location))
```

for the bigquery-public-data.london_bicycles dataset yields:

```
london_bicycles created on 2017-05-25 13:26:18.055000+00:00 in EU
```

It is also possible to examine the access controls on the dataset using the dsinfo object. For example, we could find which roles are granted READER access to the London bicycles dataset using:

for access in dsinfo.access_entries:

```
if access.role == 'READER':
```

```
    print(access)
```

This yields:

```
<AccessEntry: role=READER, specialGroup=allAuthenticatedUsers>
```

```
<AccessEntry: role=READER, domain=google.com>
```

```
<AccessEntry: role=READER, specialGroup=projectReaders>
```

It is because all authenticated users are granted access to the dataset (see the first line above) that we have been able to query the dataset in previous chapters.

Creating a dataset

To create a dataset named ch05 if it doesn't already exist:

```
dataset_id = "{}.ch05".format(PROJECT)
```

```
ds = bq.create_dataset(dataset_id, exists_ok=True)
```

By default, the dataset is created in the US. To create the dataset in another location, for example the EU, create a local Dataset object (we're calling it dsinfo), set its location attribute, and then invoke create_dataset on the client object using this Dataset (instead of the dataset_id as in the previous code snippet):

```
dataset_id = "{}.ch05eu".format(PROJECT)
```

```
dsinfo = bigquery.Dataset(dataset_id)
```

```
dsinfo.location = 'EU'
```

```
ds = bq.create_dataset(dsinfo, exists_ok=True)
```

Deleting a dataset

To delete a dataset named ch05 in the project passed into the Client, do:

```
bq.delete_dataset('ch05', not_found_ok=True)
```

To delete a dataset in a different project, qualify the dataset name by the project name:

```
bq.delete_dataset('{}.ch05'.format(PROJECT), not_found_ok=True)
```

Modifying attributes of a dataset

To modify information about a dataset, modify the dsinfo object locally by setting the description attribute, and then invoke update_dataset on the client object to update the BigQuery service:

```
dsinfo = bq.get_dataset("ch05")
```

```
print(dsinfo.description)
```

dsinfo.description = “Chapter 5 of BigQuery: The Definitive Guide”

```
dsinfo = bq.update_dataset(dsinfo, ['description'])
```

```
print(dsinfo.description)
```

The first print in the above snippet will print out None because the dataset ch05 was created without any description. After the update_dataset call, the dataset in BigQuery sports a new description:

None

Chapter 5 of BigQuery: The Definitive Guide

Changing tags, access controls, etc. of a dataset work similarly. For example, to give one of our colleagues access to the ch05 dataset, we could do:

```
dsinfo = bq.get_dataset("ch05")

entry = bigquery.AccessEntry(
    role="READER",
    entity_type="userByEmail",
    entity_id="xyz@google.com",
)

if entry not in dsinfo.access_entries:

    entries = list(dsinfo.access_entries)
    entries.append(entry)

    dsinfo.access_entries = entries

dsinfo = bq.update_dataset(dsinfo, ["access_entries"]) # API request

else:

    print('{} already has access'.format(entry.entity_id))
```

```
print(dsinfo.access_entries)
```

In the code above, we are creating an entry for a user, and if the user doesn't already have some sort of access to the dataset, we get the current set of access entries, append the new entry, and update the dataset with the new list.

TABLE MANAGEMENT

To list the tables in a dataset, invoke the `list_tables` method on the client object:

```
tables = bq.list_tables("bigquery-public-data.london_bicycles")
```

for table in tables:

```
    print(table.table_id)
```

This yields:

`cycle_hire`

`cycle_stations`

the two tables in the London bicycles dataset.

Obtaining table properties

In the code snippet above, we got the `table_id` from the `table` object. Besides the table id, other attributes of the table are available -- the number of rows, the descriptions, tags, the schema, etc.

<tip>

The number of rows in the table is part of the table metadata and can be obtained from the table object itself. Unlike a full-fledged query with a COUNT(*), getting the number of rows in this manner does not incur BigQuery charges:

```
table = bq.get_table(  
    "bigquery-public-data.london_bicycles.cycle_stations")  
  
print('{} rows in {}'.format(table.num_rows, table.table_id))
```

This yields:

787 rows in cycle_stations

</tip>

For example, we can search the schema for columns whose name contains a specific substring (count) using the table object:

```
table = bq.get_table(  
    "bigquery-public-data.london_bicycles.cycle_stations")
```

for field in table.schema:

if 'count' in field.name:

```
    print(field)
```

This yields:

```
SchemaField('bikes_count', 'INTEGER', 'NULLABLE', "", ())
```

```
SchemaField('docks_count', 'INTEGER', 'NULLABLE', "", ())
```

Of course, rather than hand-roll this sort of search, it would be better to use INFORMATION_SCHEMA (covered in Chapter 8) or use Data Catalog (covered in Chapter X).

Deleting a table

Deleting a table is similar to deleting a dataset, and if desired, we can ignore the error thrown if the table doesn't exist:

```
bq.delete_table('ch05.temp_table', not_found_ok=True)
```

<tip>

Use BigQuery's "time travel" capability to restore deleted tables. For up to 2 days, tables can be restored if accidentally deleted. For example, to restore the version of the table as it existed at a certain time within the past 7 days, make a copy of it specifying the timestamp:

```
bq --location=US cp ch05.temp_table@1418864998000 ch05.temp_table2
```

Here, 1418864998000 is the timestamp (number of seconds since epoch).

Note, however, that the snapshots are lost if a table bearing the same ID in the dataset was created after the deletion time or if the encapsulating dataset was also deleted. This has implications on your workflow -- you can save yourself a lot of grief if you minimize the chance of creating tables with the same name from different software applications. For example, you could use organizational boundaries or "owning" application

names in dataset names. You might also avoid creating tables from your applications -- instead, create them externally before your applications start.

</tip>

Creating an empty table

Creating an empty table is similar to creating a dataset, and if desired, we can ignore the exception thrown when the dataset already exists:

```
table_id = '{}.ch05.temp_table'.format(PROJECT)
```

```
table = bq.create_table(table_id, exists_ok=True)
```

Updating a table's schema

Of course, we don't usually want to create empty tables. We want to create an empty table with a schema and insert some rows into it. Because the schema is part of the attributes of the table, we can update the schema of the empty table similarly to the way we updated the access controls of the dataset. We get the table, modify the table object locally, and then update the table using the modified object specifying what aspects of the table object are being updated:

```
schema = [  
    bigquery.SchemaField("chapter", "INTEGER", mode="REQUIRED"),  
    bigquery.SchemaField("title", "STRING", mode="REQUIRED"),  
]
```

```
table_id = '{}.ch05.temp_table'.format(PROJECT)

table = bq.get_table(table_id)

print(table.etag)

table.schema = schema

table = bq.update_table(table, ["schema"])

print(table.schema)

print(table.etag)
```

To prevent race conditions, BigQuery tags the table with each update. So, when we get the table information using `get_table`, the table object includes an etag. When we upload a modified schema with `update_table`, this update succeeds only if our etag matches that of the server. The returned table object has the new etag. We can turn off this behavior and force an update by setting `table.etag` to be `None`.

When a table is empty, you can change the schema to anything you want. But when there is data in the table, any schema changes must be compatible with the existing data in the table. You can add new fields (as long as they are `NULLABLE`), and you can relax constraints from `REQUIRED` to `NULLABLE`.

After this code is run, we can check in the BigQuery web console that the newly created table has the right schema:

temp_table			
Schema	Details	Preview	
Field name	Type	Mode	Description
chapter	INTEGER	REQUIRED	
title	STRING	REQUIRED	

[Edit schema](#)

Figure 5-2. Schema of newly created table

Inserting rows into a table

Once we have a table with a schema, we can insert rows into the table using the client. The rows consist of Python tuples in the same order as defined in the schema:

```
rows = [
    (1, u'What is BigQuery?'),
    (2, u'Query essentials'),
]
errors = bq.insert_rows(table, rows)
```

The errors list will be empty if all rows were successfully inserted. If, however, we had passed in a non-integer value for the chapter field:

```
rows = [
    ('3', u'Operating on data types'),
```

```
(‘wont work’, u’This will fail’),  
(’4’, u’Loading data into BigQuery’),  
]  
  
errors = bq.insert_rows(table, rows)  
  
print(errors)
```

we will get an error whose reason is invalid on index=1 (the second row, since this is 0-based), location=chapter:

```
{'index': 1, 'errors': [{‘reason’: ‘invalid’, ‘debugInfo’: ”, ‘message’: ‘Cannot convert value to integer (bad value):wont work’, ‘location’: ‘chapter’}]}{}
```

Because BigQuery treats each user request as atomic, none of the 3 rows will be inserted. On the other rows, you will get an error whose reason is stopped:

```
{'index': 0, 'errors': [{‘reason’: ‘stopped’, ‘debugInfo’: ”, ‘message’: ”, ‘location’: ”}]}{}
```

In the BigQuery web console, the table details show that the inserted rows are in the streaming buffer, but the two inserted rows are not reflected in the table’s number of rows.

temp_table

QUERY TA

Schema Details Preview

Description 

None

Table info 

Table ID	cloud-training-demos:ch05.temp_table
Table size	0 B
Number of rows	0
Created	Mar 3, 2019, 10:46:40 AM
Table expiration	Never
Last modified	Mar 3, 2019, 10:48:23 AM
Data location	US

Streaming buffer statistics

Estimated size	41 B
Estimated rows	2
Earliest entry time	Mar 3, 2019, 10:48:00 AM

Figure 5-3. Newly inserted rows are in the streaming buffer and are not yet reflected in the number of rows shown in the table info.

<warning>

Because inserting rows into the table is a streaming operation, the table metadata is not updated immediately. For example, if we were to do:

```
rows = [
```

```
(1, u'What is BigQuery?'),
```

```
(2, u'Query essentials'),
```

```
]

print(table.table_id, table.num_rows)  
  
errors = bq.insert_rows(table, rows)  
  
print(errors)  
  
table = bq.get_table(table_id)  
  
print(table.table_id, table.num_rows) # DELAYED
```

The `table.num_rows` in the above code snippet will **not** show the updated row count. Moreover, streaming inserts, unlike load jobs, are not free.

</warning>

Queries on the table will reflect the two rows in the streaming buffer. For example, doing:

```
SELECT DISTINCT(chapter) FROM ch05.temp_table
```

shows that there are two chapters in the table:

Row	chapter
1	2
2	1

Creating an empty table with schema

Instead of creating a table and then updating this schema, a better idea is to provide the schema at the time of table creation:

```
schema = [  
    bigquery.SchemaField("chapter", "INTEGER", mode="REQUIRED"),  
    bigquery.SchemaField("title", "STRING", mode="REQUIRED"),  
]  
  
table_id = '{}.ch05.temp_table2'.format(PROJECT)  
  
table = bigquery.Table(table_id, schema)  
  
table = bq.create_table(table, exists_ok=True)  
  
print('{} created on {}'.format(table.table_id, table.created))  
  
print(table.schema)
```

The created table contains the desired schema:

```
temp_table2 created on 2019-03-03 19:30:18.324000+00:00
```

```
[SchemaField('chapter', 'INTEGER', 'REQUIRED', None, ()�,  
 SchemaField('title', 'STRING', 'REQUIRED', None, ())]
```

Loading a BigQuery table

The table created in the previous section is empty, and we'd use the technique above if we are going to do a streaming insert of rows into the table. What if, though, we already have the data in a file and we wish simply to create a table and initialize it with data from that file? In that case, load jobs are much more convenient. Unlike streaming inserts, loads

do not incur BigQuery charges.

The BigQuery Python client supports several methods of loading data: from a pandas dataframe, from a URI or from a local file. Let's look at these one by one.

Loading a pandas dataframe

pandas¹⁰ is an open source library that provides data structures and data analysis tools for the Python programming language. The BigQuery Python library supports directly loading data from an in-memory pandas dataframe. Since pandas dataframes can be constructed from in-memory data structures, and provides a wide variety of transformations, using pandas provides the most convenient way to load data from Python applications. For example, to create a DataFrame from an array of tuples, we can do:

```
import pandas as pd
```

```
data = [
```

```
(1, u'What is BigQuery?'),
```

```
(2, u'Query essentials'),
```

```
]
```

```
df = pd.DataFrame(data, columns=['chapter', 'title'])
```

Once we have created the DataFrame, we can load the data in the DataFrame into a BigQuery table using:¹¹

```
table_id = '{}.ch05.temp_table3'.format(PROJECT)

job = bq.load_table_from_dataframe(df, table_id)

job.result() # blocks and waits

print("Loaded {} rows into {}".format(job.output_rows,
                                       tblref.table_id))
```

Because load jobs can potentially be long-running, the `load_table_` functions return a job object that you can use either to poll using the `job.done()` method, or to block and wait using `job.result()`.

If the table already exists, the load job will append to the existing table as long as the data you are loading matches the existing schema. If the table doesn't exist, a new table is created with schema that is inferred from the pandas dataframe.¹² You can change this behavior by specifying a load configuration:

```
from google.cloud.bigquery.job \

import LoadJobConfig, WriteDisposition, CreateDisposition

load_config = LoadJobConfig(

    create_disposition=CreateDisposition.CREATE_IF_NEEDED,
    write_disposition=WriteDisposition.WRITE_TRUNCATE)

job = bq.load_table_from_dataframe(df, table_id,
```

job_config=load_config)

The combination of create and write disposition controls the behavior of the load operation:

CreateDisposition	WriteDisposition	Behavior
CREATE_NEVER	WRITE_APPEND	Append to existing table.
	WRITE_EMPTYSY	Appends to table, but only if it is currently empty. Otherwise, a duplicate error is thrown.
	WRITE_TRUNCATE	Clears out any existing rows in the table, i.e. overwrites the data in the table.
		Create new table based on schema of the input if necessary.
CREATE_IF_NEEDED	WRITE_APPEND	Append to existing or newly created table.
		This is the default behavior if job_config is not passed in.
	WRITE_EMPTYSY	Create new table based on schema of the input if necessary.
		Requires that, if the table already exists, it be empty. Otherwise, a duplicate error is thrown.
	WRITE_TRUNCATE	Create new table based on schema of the input if necessary.
		Clears out any existing rows in the table, i.e. overwrites the data in the table.

Loading from a URI

It is possible to load a BigQuery table directly from a file whose Google Cloud URI is known. In addition to Cloud Datastore backups and HTTP URLs referring to Cloud Bigtable, Google Cloud Storage wildcard patterns are also supported.¹³ We can, therefore, load the College Scorecard CSV file that we used in the previous chapter using:

```
job_config = bq.LoadJobConfig()
```

```
job_config.autodetect = True

job_config.source_format = bq.SourceFormat.CSV

job_config.null_marker = 'NULL'

uri = "gs://bigquery-oreilly-book/college_scorecard.csv"

table_id = '{}.ch05.college_scorecard_gcs'.format(PROJECT)

job = bq.load_table_from_uri(uri, table_id, job_config=job_config)
```

All the options that we considered in Chapter 4 (on loading data) can be set using the JobConfig flags. Here, we are using autodetect, specifying that the file format is CSV and that the file uses a non-standard null marker before loading the file from the URI specified.

While we can block for the job to finish as we did in the previous section, we can also poll the job every 0.1 seconds, and get the table details only after the load job is done:

```
while not job.done():

    print('.', end='', flush=True)

    time.sleep(0.1)

    print('Done')

table = bq.get_table(tblref)

print("Loaded {} rows into {}.".format(table.num_rows, table.table_id))
```

After a few dots to represent the wait state, we get back the number of rows loaded (7175).

Loading from a local file

To load from a local file, create a file object and use `load_table_from_file`:

with `gzip.open('..../04_load/college_scorecard.csv.gz')` as fp:

```
job = bq.load_table_from_file(fp, tblref, job_config=job_config)
```

In other respects, this is similar to loading from a URI.

Copying a table

You can copy a table from one dataset to another using the `copy_table` method:

```
source_tbl = 'bigquery-public-data.london_bicycles.cycle_stations'
```

```
dest_tbl = '{}.ch05eu.cycle_stations_copy'.format(PROJECT)
```

```
job = bq.copy_table(source_tbl, dest_tbl, location='EU')
```

```
job.result() # blocks and waits
```

```
dest_table = bq.get_table(dest_tbl)
```

```
print(dest_table.num_rows)
```

Note that we are copying the London cycle stations data to a dataset (ch05eu) that we created in the EU. Note also that we are making sure to copy tables only within the same region.

Extracting data from a table

We can export data from a table to a file in Google Cloud Storage using the extract_table method:

```
source_tbl = 'bigquery-public-data.london_bicycles.cycle_stations'
```

```
dest_uri = 'gs://{}tmp/exported/cycle_stations'.format(BUCKET)
```

```
config = bigquery.job.ExtractJobConfig()
```

```
destination_format =
```

```
bigquery.job.DestinationFormat.NEWLINE_DELIMITED_JSON)
```

```
job = bq.extract_table(source_tbl, dest_uri,
```

```
location='EU', job_config=config)
```

```
job.result() # blocks and waits
```

If the table is sufficiently large, the output will be sharded into multiple files. At the time of writing, extraction formats that are supported include CSV, Avro, and newline-delimited JSON. As with copying tables, make sure to export to a bucket in the same location as the dataset.¹⁴ Of course, once you export a table, Google Cloud Storage charges will start to accrue for the output files.

Browsing the rows of a table

In the BigQuery web user interface, you have the ability to preview a table without incurring querying charges. The same capability is available via the REST API as `tabledata.list`, and consequently through the Python API.

To list an arbitrary 5 rows from the cycle_stations table¹⁵, we could do:

```
table_id = 'bigquery-public-data.london_bicycles.cycle_stations'

table = bq.get_table(table_id)

rows = bq.list_rows(table,
                    start_index=0,
                    max_results=5)
```

Omitting the start_index and max_results allows us to get all the rows in the table:

```
rows = bq.list_rows(table)
```

Of course, the table has to be small enough to fit into memory. If that is not the case, we can paginate through the entire table, processing the table in chunks:

```
page_size = 10000

row_iter = bq.list_rows(table,
                        page_size=page_size)

for page in row_iter.pages:

    rows = list(page)

    # do something with rows ...
```

```
print(len(rows))
```

Instead of getting all the fields, we can select the id field and any columns whose name includes the substring count by:

```
fields = [field for field in table.schema
```

```
if 'count' in field.name or field.name == 'id']
```

```
rows = bq.list_rows(table,
```

```
start_index=300,
```

```
max_results=5,
```

```
selected_fields=fields)
```

We can then format the resulting rows to have a fixed width of 10 characters using:

```
fmt = '{!s:<10}' * len(rows.schema)
```

```
print(fmt.format(*[field.name for field in rows.schema]))
```

```
for row in rows:
```

```
    print(fmt.format(*row))
```

This yields:

```
id bikes_count docks_count
```

658 20 30

797 20 30

238 21 32

578 22 32

477 26 36

QUERYING

The major benefit of using the Google Cloud Client library comes when querying. Much of the complexity regarding pagination, retries, etc. is handled transparently.

The first step, of course, is to create a string containing the SQL to be executed by BigQuery:

```
query = """
```

```
SELECT
```

```
start_station_name
```

```
, AVG(duration) as duration
```

```
, COUNT(duration) as num_trips
```

```
FROM `bigquery-public-data`.london_bicycles.cycle_hire
```

```
GROUP BY start_station_name
```

```
ORDER BY num_trips DESC
```

```
LIMIT 10
```

```
““““
```

The above query finds the 10 busiest stations in London as measured by the total number of trips initiated at those stations, and reports the station name, the average duration of trips initiated at this station, and the total number of such trips.

Dry run

Before actually executing the query, it is possible to do a dry run to get an estimate of how much data will be processed by the query¹⁶:

```
config = bigquery.QueryJobConfig()  
  
config.dry_run = True  
  
job = bq.query(query, location='EU', job_config=config)  
  
print("This query will process {} bytes."  
     .format(job.total_bytes_processed))
```

When we ran the code above, we got:

This query will process 903989528 bytes.

Your result might be somewhat different since this table is refreshed with new data as it is made available.

<tip>

The dry run does not incur charges. Use dry runs to check that query syntax is correct both during development and in your testing harness. For example, dry runs can be used to identify undeclared parameters and to validate the schema of the query result without actually running it. If you are building an application that sends queries to BigQuery, you can use the dry run feature to provide billing caps. Performance and cost optimization will be covered more thoroughly in Chapter 7.

</tip>

Sometimes, it is impossible to compute the bytes processed ahead of time without actually running the query. In such cases, the dry run returns either zero or an upper bound estimate. This happens in two situations: when querying a federated table (where the data is stored outside BigQuery; see Chapter 4) and when querying a clustered table (see chapter X). In the case of federated tables, the dry-run will report 0 bytes, and in the case of clustered tables, BigQuery will attempt to figure out the worst case scenario and report that number. In either case, though, when actually performing the query, you'll only be billed for the data that actually needs to be read.

Executing the query

To execute the query, simply start to iterate over the job object. The job will be launched, and pages of results retrieved as you iterate over the job using the for loop:

```
job = bq.query(query, location='EU')
```

```
fmt = '{!s:<40} {:>10d} {:>10d}'
```

```
for row in job:
```

```
    fields = (row['start_station_name'],
```

```
              (int)(0.5 + row['duration']),
```

```
              row['num_trips'])
```

```
    print(fmt.format(*fields))
```

Given a row, it is possible to obtain the value for any of the columns in the resultset using the aliased name of the column in the SELECT (look at how the column num_trips appears in the result set).

The formatted result of the query is:

Belgrave Street , King's Cross 1011 234458

Hyde Park Corner, Hyde Park 2783 215629

Waterloo Station 3, Waterloo 866 201630

Black Lion Gate, Kensington Gardens 3588 161952

Albert Gate, Hyde Park 2359 155647

Waterloo Station 1, Waterloo 992 145910

Wormwood Street, Liverpool Street 976 119447

Hop Exchange, The Borough 1218 115135

Wellington Arch, Hyde Park 2276 110260

Triangle Car Park, Hyde Park 2233 108347

Creating a pandas dataframe

Earlier in this section, we saw how to load a BigQuery table from a pandas dataframe. It is also possible to execute a query and get the results back as a pandas dataframe, thus using BigQuery as a highly distributed and scalable intermediate step in a data science workflow:

```
query = """
```

```
SELECT
```

```
start_station_name
```

```
, AVG(duration) as duration
```

```
, COUNT(duration) as num_trips
```

```
FROM `bigquery-public-data`.london_bicycles.cycle_hire
```

```
GROUP BY start_station_name
```

```
"""
```

```
df = bq.query(query, location='EU').to_dataframe()
```

```
print(df.describe())
```

The code above uses the pandas describe() functionality to print out the distribution of the numeric columns in the resultset:

```
duration num_trips  
  
count 880.000000 880.000000  
  
mean 1348.351153 27692.273864  
  
std 434.057829 23733.621289  
  
min 0.000000 1.000000  
  
25% 1078.684974 13033.500000  
  
50% 1255.889223 23658.500000  
  
75% 1520.504055 35450.500000  
  
max 4836.380090 234458.000000
```

Thus, there are 880 stations in total, with an average of 27,692 trips starting at each station, although there is a station with only one trip and a station with 234,458 trips. The median station has supported 23,658 rides, and the majority of stations have had between 13,033 and 35,450 rides.

Parameterized queries

The queries do not need to be static strings. Instead, you can parameterize them, so that the query parameters are specified at the time the query job is created. Here is an example of a query that finds the total number of trips that were longer than a specific duration. The actual threshold,

`min_duration`, will be specified at the time the query is run:

```
query2 = """
```

```
SELECT
```

```
start_station_name
```

```
, COUNT(duration) as num_trips
```

```
FROM `bigquery-public-data`.london_bicycles.cycle_hire
```

```
WHERE duration >= @min_duration
```

```
GROUP BY start_station_name
```

```
ORDER BY num_trips DESC
```

```
LIMIT 10
```

```
"""
```

The `@` symbol identifies `min_duration` as a parameter to the query. A query can have any number of such named parameters.

<tip>

Creating a query by doing string formatting is an extremely bad practice.
String manipulation such as:

```
query2 = """
```

```
SELECT  
  
    start_station_name  
  
    , COUNT(duration) as num_trips  
  
FROM `bigquery-public-data`.london_bicycles.cycle_hire  
  
WHERE duration >= {}  
  
GROUP BY start_station_name  
  
ORDER BY num_trips DESC  
  
LIMIT 10  
  
""".format(min_duration)
```

can make your data warehouse subject to SQL injection attacks. We strongly suggest that you use parameterized queries, especially when constructing queries that include user input.

</tip>

When executing a query that has named parameters, you need to supply a job_config with those parameters:

```
config = bigquery.QueryJobConfig()  
  
config.query_parameters = [  
  
    bigquery.ScalarQueryParameter('min_duration', "INT64", 600)
```

```
]
```

```
job = bq.query(query2, location='EU', job_config=config)
```

Here, we are specifying that we wish to retrieve the number of trips over 600 seconds in duration.

As before, iterating over the job will allow us to retrieve the rows, and each row functions like a dictionary of column names to values:

```
fmt = '{!s:<40} {:>10d}'
```

```
for row in job:
```

```
    fields = (row['start_station_name'],
```

```
              row['num_trips'])
```

```
    print(fmt.format(*fields))
```

Running the code above yields:

Hyde Park Corner, Hyde Park 203592

Belgrave Street , King's Cross 168110

Waterloo Station 3, Waterloo 148809

Albert Gate, Hyde Park 145794

Black Lion Gate, Kensington Gardens 137930

Waterloo Station 1, Waterloo 106092

Wellington Arch, Hyde Park 102770

Triangle Car Park, Hyde Park 99368

Wormwood Street, Liverpool Street 82483

Palace Gate, Kensington Gardens 80342

In this section, we covered how to programmatically invoke BigQuery operations, whether they involve table or dataset manipulation, querying data, or streaming inserts. The programmatic APIs, especially the Google Cloud Client Library, are what you would use whenever you are building applications that need to access BigQuery.

However, in some specific instances, there are higher-level abstractions available. These are covered in the next section.

Accessing BigQuery from data science tools

Notebooks have revolutionized the way that data science is carried out. They are an instance of literate programming, a programming paradigm introduced by the computer science legend Donald Knuth, wherein computer code is intermixed with headings, text, plots, etc. Because of this, the notebook serves simultaneously as an executable program and as an interactive report.

Jupyter notebooks are the most popular of the notebook frameworks and work in a variety of languages including Python. In Jupyter, the notebook is a web-based interactive document in which code can be typed and

executed. The output of the code is embedded directly in the document.

Notebooks on Google Cloud

To create a notebook on Google Cloud Platform, launch a Deep Learning VM and get the URL to Jupyter. You can do this from the AI Factory section of the Google Cloud web console, or you can automate it using the gcloud command-line tool:¹⁷

```
#!/bin/bash

IMAGE=--image-family=tf-latest-cpu

INSTANCE_NAME=dlvm

MAIL=google-cloud-customer@gmail.com # CHANGE THIS

echo "Launching $INSTANCE_NAME"

gcloud compute instances create ${INSTANCE_NAME} \
    --machine-type=n1-standard-2 \
    --scopes=https://www.googleapis.com/auth/cloud-
    platform,https://www.googleapis.com/auth/userinfo.email \
    ${IMAGE} \
    --image-project=deeplearning-platform-release \
    --boot-disk-device-name=${INSTANCE_NAME} \
```

```
--metadata="proxy-user-mail=${MAIL}"  
  
echo "Looking for Jupyter URL on ${INSTANCE_NAME}"  
  
while true; do  
  
proxy=$(gcloud compute instances describe ${INSTANCE_NAME} 2>  
/dev/null | grep dot-datalab-vm)  
  
if [ -z "$proxy" ]  
  
then  
  
echo -n ".."  
  
sleep 1  
  
else  
  
echo "done!"  
  
echo "$proxy"  
  
break  
  
fi  
  
done
```

Access the URL (or from the GCP web console, navigate to the Notebook) and you will be in Jupyter. Click on the button to create a Python 3 notebook and you will be able to try out the snippets of code.

The Cloud AI Factory Notebook already has the Google Cloud Client library for BigQuery installed, but if you are on some other Jupyter environment, you can install the library and load the necessary extensions by running the following code in a code cell:

```
!pip install google-cloud-bigquery
```

```
%load_ext google.cloud.bigquery
```

In a Jupyter notebook, any line preceded by an exclamation (!) is run using the command-line shell while any line preceded by a percent sign (%) invokes an extension, also called a magic. So, in the code snippet above, the pip install is carried out on the command line while the extension named load_ext is used to load the BigQuery magics.

You can clone the repository corresponding to this book by clicking on the git icon (the last icon in the set of ribbons) and cloning

<https://github.com/GoogleCloudPlatform/bigquery-oreilly-book>:

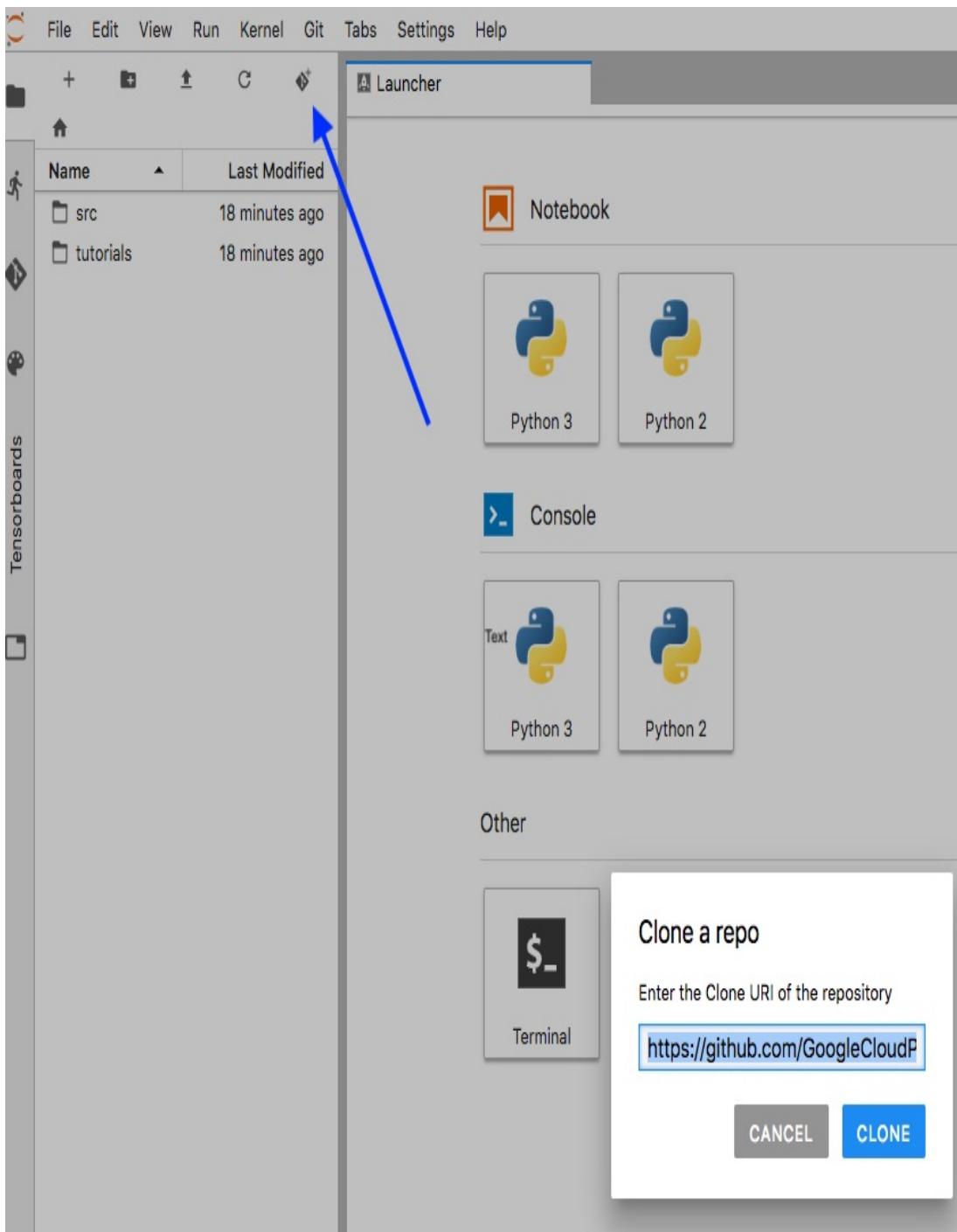


Figure 5-4. Click on the icon shown to clone a repository.

Browse to and open the 05_devel/magics.ipynb notebook to try out the code in this section of the book. Change the PROJECT variable in the notebook to reflect your project. Then, from the menu at the top, select Run > Run All Cells.

JUPYTER MAGICS

The BigQuery extensions for Jupyter make running queries within a notebook quite easy. For example, to run a query, you simply need to specify %%bigrquery at the top of the cell:

```
%%bigrquery --project $PROJECT
```

```
SELECT
```

```
    start_station_name
```

```
, AVG(duration) as duration
```

```
, COUNT(duration) as num_trips
```

```
FROM `bigquery-public-data`.london_bicycles.cycle_hire
```

```
GROUP BY start_station_name
```

```
ORDER BY num_trips DESC
```

```
LIMIT 5
```

Running a cell with the above code executes the query and displays a nicely formatted table with the five desired rows:

Run a query

```
[2]: %%bq --project $PROJECT
SELECT
    start_station_name
    , AVG(duration) AS duration
    , COUNT(duration) AS num_trips
FROM `bigquery-public-data`.london_bicycles.cycle_hire
GROUP BY start_station_name
ORDER BY num_trips DESC
LIMIT 5
```

	start_station_name	duration	num_trips
0	Belgrave Street , King's Cross	1011.076696	234458
1	Hyde Park Corner, Hyde Park	2782.730709	215629
2	Waterloo Station 3, Waterloo	866.376135	201630
3	Black Lion Gate, Kensington Gardens	3588.012004	161952
4	Albert Gate, Hyde Park	2359.413930	155647

Figure 5-5. To run a query in Jupyter, write a cell with the text of the query and on the first line of the cell, insert the %%bq magic. The result of the query, nicely formatted, is embedded into the document.

RUNNING A PARAMETERIZED QUERY

To run a parameterized query, specify --params in the magic. The parameters themselves are a Python variable that is typically defined elsewhere in the notebook:

Run a parameterized query

```
[3]: PARAMS = {"num_stations": 3}

[4]: %%bigquery --project $PROJECT --params $PARAMS
SELECT
    start_station_name
    , AVG(duration) as duration
    , COUNT(duration) as num_trips
FROM `bigquery-public-data`.london_bicycles.cycle_hire
GROUP BY start_station_name
ORDER BY num_trips DESC
LIMIT @num_stations
```

	start_station_name	duration	num_trips
0	Belgrave Street , King's Cross	1011.076696	234458
1	Hyde Park Corner, Hyde Park	2782.730709	215629
2	Waterloo Station 3, Waterloo	866.376135	201630

Figure 5-6. How to run a parameterized query in a notebook.

In the example above, the number of rows is a parameter that is specified as a Python variable and used in the SQL query.

SAVING QUERY RESULTS TO PANDAS

Saving the results of a query to pandas involves specifying the name of the variable (e.g. df) by which the pandas dataframe will be referenced:

```
%%bigquery df --project $PROJECT
```

```
SELECT
```

```
    start_station_name
```

```
        , AVG(duration) as duration
```

```
        , COUNT(duration) as num_trips
```

```
FROM `bigquery-public-data`.london_bicycles.cycle_hire
```

```
GROUP BY start_station_name
```

```
ORDER BY num_trips DESC
```

The variable df can be used like any other pandas dataframe. For example, we could ask for statistics of the numeric columns in df using:

```
df.describe()
```

We can also use the plotting commands available in pandas to draw a scatter plot of the average duration of trips and the number of trips across all the stations:

```
[7]: df.plot.scatter('duration', 'num_trips');
```

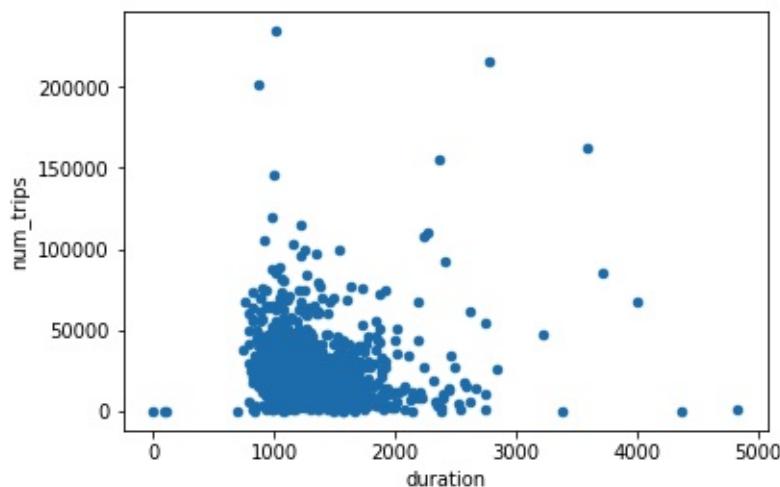


Figure 5-7. Plotting a pandas dataframe obtained using a BigQuery query.

pandas

We have introduced linkages between the Google Cloud Client library for BigQuery and pandas in several sections of this book. Because pandas is

the de-facto standard for data analysis in Python, it might be helpful to bring together all these capabilities and use them to illustrate a typical data science workflow.

Imagine that we are receiving anecdotes from our customer support team about bad bicycles at some stations. We'd like to send a crew out to do a spot check on a number of problematic stations. How do we choose which stations to spot check? We could rely on stations where we have received customer complaints, but we will tend to receive more complaints from busy stations simply because they have lots more customers.

We believe that if someone rents a bicycle for less than 10 minutes and returns the bicycle to the same station they rented it from, it is likely that the bicycle has a problem. Let's call this a bad trip (from the customer's view point, it is). We could have our crew do a spot check of stations where bad trips have occurred more frequently.

To find the fraction of bad trips, we can query BigQuery using the Jupyter magic and save the result into a pandas dataframe called `badtrips` using:

```
%%bigquery badtrips --project $PROJECT
```

```
WITH all_bad_trips AS (
```

```
SELECT
```

```
start_station_name
```

```
, COUNTIF(duration < 600 AND start_station_name = end_station_name)  
AS bad_trips
```

```

, COUNT(*) as num_trips

FROM `bigquery-public-data`.london_bicycles.cycle_hire

WHERE EXTRACT(YEAR FROM start_date) = 2015

GROUP BY start_station_name

HAVING num_trips > 10

)

SELECT *, bad_trips / num_trips AS fraction_bad FROM all_bad_trips

ORDER BY fraction_bad DESC

```

The WITH expression above counts the number of trips whose duration is less than 600 seconds and for which the starting and ending stations are the same. By grouping this by start_station_name, we get the total number of trips and bad trips at each station. The outer query computes the desired fraction and associates it with the station. This yields (only the first few rows are shown):

start_station_name	bad_trips	num_trips	fraction_bad
Contact Centre, Southbury House	20	48	0.416667
Monier Road, Newham	1	25	0.040000
Aberfeldy Street, Poplar	35	955	0.036649
Ormonde Gate, Chelsea	315	8932	0.035266
Thornfield House, Poplar	28	947	0.029567

...

It is clear that the top of the table consists of some oddball stations. Just 48 trips originated from the Southbury House station. Nevertheless, we can confirm this by using pandas to look at the statistics of the dataframe:

```
badtrips.describe()
```

This yields:

	bad_trips	num_trips	fraction_bad
count	823.000000	823.000000	823.000000
mean	75.074119	11869.755772	0.007636
std	70.512207	9906.268656	0.014739
min	0.000000	11.000000	0.000000
25%	41.000000	5903.000000	0.005002
50%	62.000000	9998.000000	0.006368
75%	91.500000	14852.500000	0.008383
max	967.000000	95740.000000	0.416667

Examining the results, we notice that fraction_bad ranges from 0 to 0.4 (look at the min and max), but is not clear how relevant this ratio is because the stations also vary quite dramatically. For example, the number of trips ranges from 11 to 95,740.

We can look at a scatter plot to see if there is any clear trend:

```
badtrips.plot.scatter('num_trips', 'fraction_bad');
```

This yields:

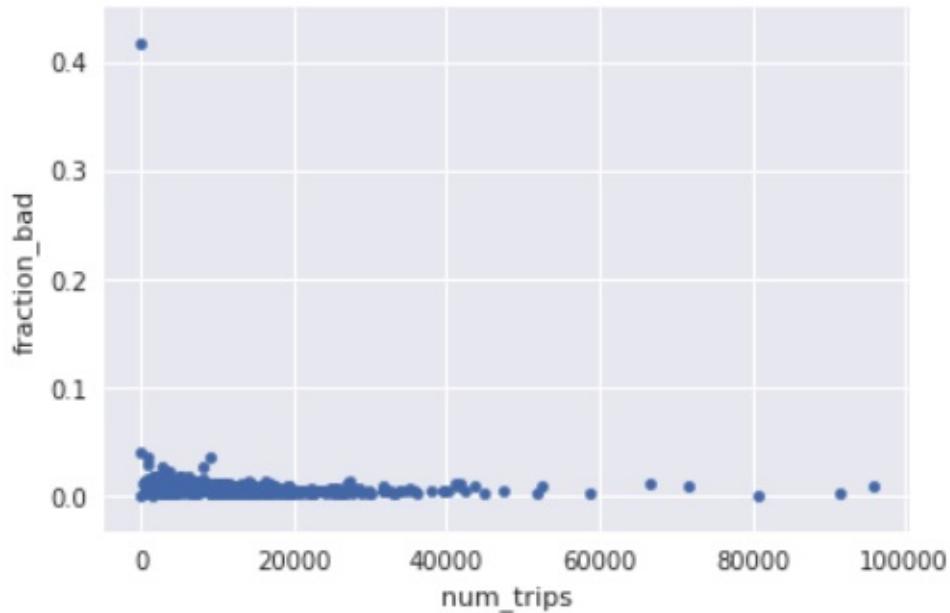


Figure 5-8. It seems that higher values of fraction_bad are associated with stations with low num_trips

It appears from the graph that higher values of fraction_bad are associated with stations with low num_trips but the trend is not clear because of the outlier 0.4 value. Let's zoom in a bit and add a line of best fit using the seaborn plotting package:

```
import seaborn as sns
```

```
ax = sns.regplot(badtrips['num_trips'],badtrips['fraction_bad']);  
  
ax.set_ylim(0, 0.05);
```

This yields a clear depiction of the trend between the fraction of bad trips and how busy the station is:

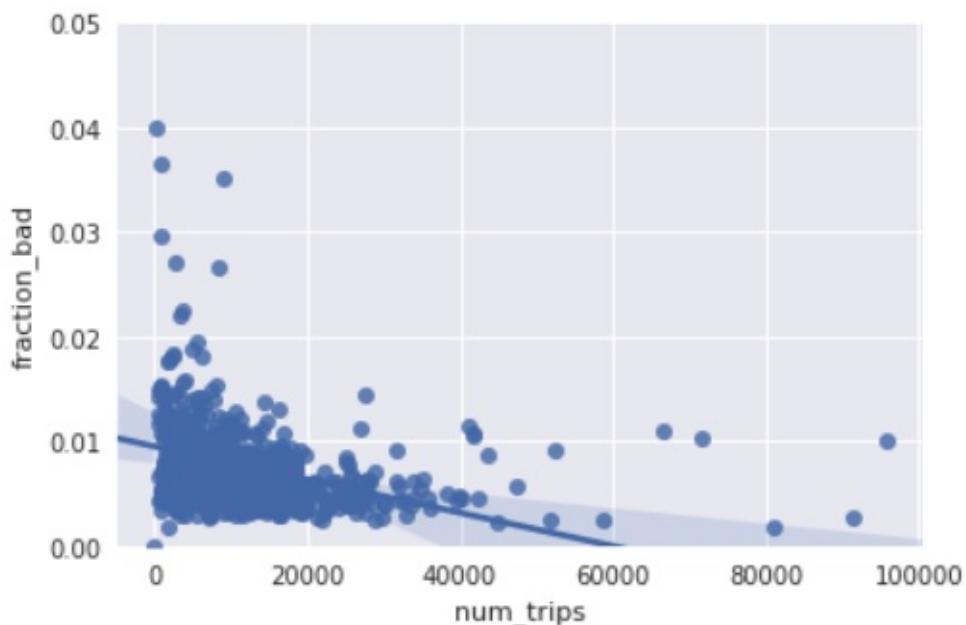


Figure 5-9. It is clear that higher values of fraction_bad are associated with stations with low num_trips

Because higher values of fraction_bad are associated with stations with low num_trips, we should not have our crew simply visit stations with high values of fraction_bad. So, how should we choose a set of stations to do a spot check on?

One approach could be to pick the 5 worst of the really busy stations, 5 of the next most busy, etc. We can do this by creating 4 different bands from the quantile of the station by num_trips, and within each band, finding the 5 worst stations. That is what this pandas snippet does:

```
stations_to_examine = []

for band in range(1,5):

    min_trips = badtrips['num_trips'].quantile(0.2*(band))

    max_trips = badtrips['num_trips'].quantile(0.2*(band+1))

    query = 'num_trips >= {} and num_trips < {}'.format(
        min_trips, max_trips)

    print(query) # band

    stations = badtrips.query(query)

    stations = stations.sort_values(
        by=['fraction_bad'], ascending=False)[:5]

    print(stations) # 5 worst

    stations_to_examine.append(stations)

print()
```

The first band consists of the 20th to 40th percentile of stations by busyness:

num_trips >= 4826.4 and num_trips < 8511.8

start_station_name bad_trips num_trips fraction_bad

6 River Street , Clerkenwell 221 8279 0.026694

9 Courland Grove, Wandsworth Road 105 5369 0.019557

10 Stanley Grove, Battersea 92 4882 0.018845

12 Southern Grove, Bow 112 6152 0.018205

18 Richmond Way, Shepherd's Bush 126 8149 0.015462

The last band consists of the 80th to 100th percentile of stations by busyness:

num_trips >= 16509.2 and num_trips < 95740.0

start_station_name bad_trips num_trips fraction_bad

25 Queen's Gate, Kensington Gardens 396 27457 0.014423

74 Speakers' Corner 2, Hyde Park 468 41107 0.011385

76 Cumberland Gate, Hyde Park 303 26981 0.011230

77 Albert Gate, Hyde Park 729 66547 0.010955

82 Triangle Car Park, Hyde Park 454 41675 0.010894

Notice that in the first band, it takes a fraction_bad of 0.015 to make the list while in the last band, a fraction_bad of 0.01 is sufficient. The smallness of these numbers might make you complacent, but this is a 50% difference.

We can then use pandas to concatenate the various bands and the BigQuery API to write these stations back to BigQuery:

```
stations_to_examine = pd.concat(stations_to_examine)

bq = bigquery.Client(project=PROJECT)

tblref = TableReference.from_string(
    '{}.ch05eu.bad_bikes'.format(PROJECT))

job = bq.load_table_from_dataframe(stations_to_examine, tblref)

job.result() # blocks and waits
```

We now have the stations to examine in a persistent storage, but we still need to get the data out to our crew. The best format for this is a map, and we can create it in Python if we know the latitude and longitude of our stations. We do, of course -- the locations of the stations is our favorite data warehouse and is only a JOIN away:

```
%%bigquery stations_to_examine --project $PROJECT
```

```
SELECT
```

```
start_station_name AS station_name
```

```
, num_trips
```

```
, fraction_bad
```

```
, latitude
```

, longitude

FROM ch05eu.bad_bikes AS bad

JOIN `bigquery-public-data`.london_bicycles.cycle_stations AS s

ON bad.start_station_name = s.name

This yields (not all rows are shown):

station_name	num_trips	fraction_bad	latitude	longitude
Ormonde Gate, Chelsea	8932	0.035266	51.487964	-0.161765
Stanley Grove, Battersea	4882	0.018845	51.470475	-0.152130
Courland Grove, Wandsworth Road	5369	0.019557	51.472918	-0.132103
Southern Grove, Bow	6152	0.018205	51.523538	-0.030556
...				

With the location information in hand, we can plot a map using the folium package:

```
import folium
```

```
map_pts = folium.Map(location=[51.5, -0.15], zoom_start=12)
```

```
for idx, row in stations_to_examine.iterrows():
```

```
    folium.Marker( location=[row['latitude'], row['longitude']],
```

```
        popup=row['station_name']).add_to(map_pts)
```

This yields a beautiful interactive map that our crew can use to check on the stations we have identified:

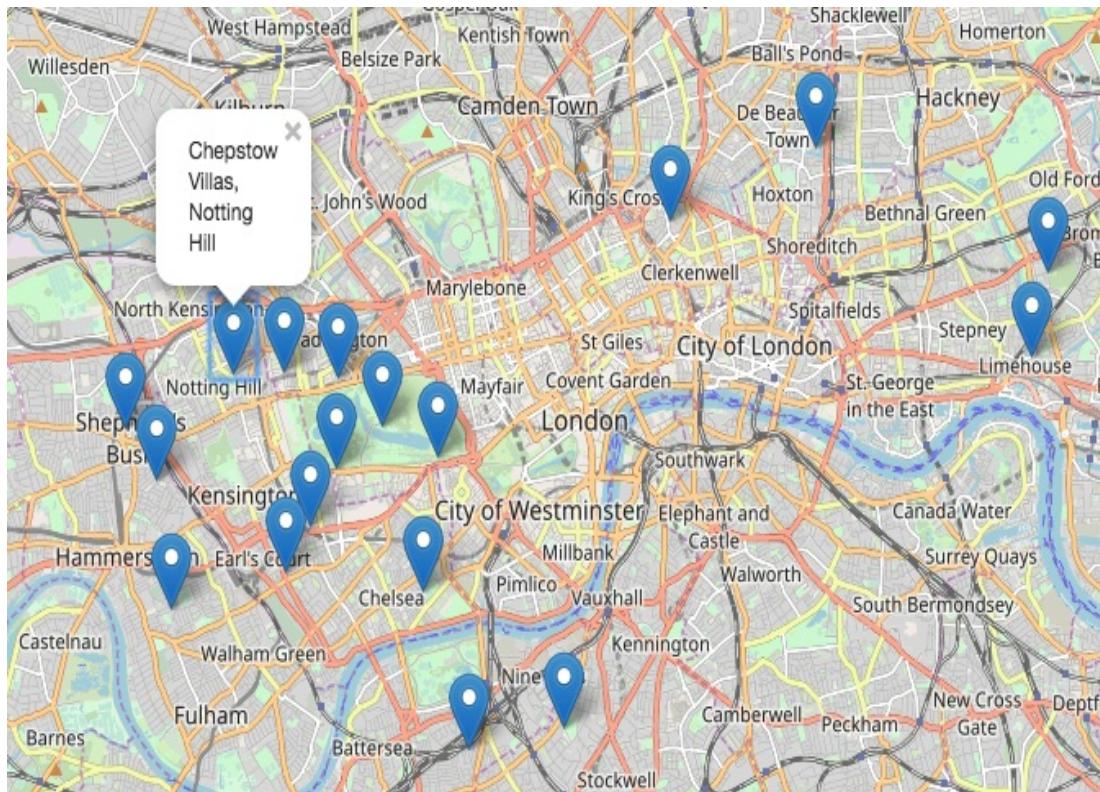


Figure 5-10. An interactive map of the stations that need to be checked.

We were able to seamlessly integrate BigQuery, pandas, and Jupyter to accomplish a data analysis task. We used BigQuery to compute aggregations over millions of bicycle rides, pandas to carry out statistical tasks, and Python packages such as folium to visualize the results interactively.

Working with BigQuery from R

While Python is one of the most popular languages for data science, it shares that perch with R, a long-standing programming language and software environment for statistics and graphics.

To use BigQuery from R, install the library `bigrquery` from CRAN:

```
install.packages("bigrquery", dependencies=TRUE)
```

Here's a simple example of querying the bicycle dataset from R:

```
billing <- 'cloud-training-demos' # your project name
```

```
sql <- "
```

```
SELECT
```

```
start_station_name
```

```
, AVG(duration) as duration
```

```
, COUNT(duration) as num_trips
```

```
FROM `bigquery-public-data`.london_bicycles.cycle_hire
```

```
GROUP BY start_station_name
```

```
ORDER BY num_trips DESC
```

```
LIMIT 5
```

```
"
```

```
tbl <- bq_project_query(billing, sql)
```

```
bq_table_download(tbl, max_results=100)
```

```
grid.tbl(tbl)
```

Use `bq_project_query` to create a BigQuery query and execute it using `bq_table_download`.

You can also use R from a Jupyter notebook. The conda environment for Jupyter¹⁸ has an R extension that can be loaded with:

```
!conda install rpy2
```

```
%load_ext rpy2.ipython
```

To carry out a linear regression to predict the number of docks at a station based on its location, we can first populate an R dataframe from BigQuery:

```
%%bigquery docks --project $PROJECT
```

```
SELECT
```

```
docks_count, latitude, longitude
```

```
FROM `bigquery-public-data`.london_bicycles.cycle_stations
```

```
WHERE bikes_count > 0
```

Then, Jupyter magics for R can be performed just like the Jupyter magics for Python. Thus, we can use the R magic to perform linear modeling (`lm`) on the `docks` dataframe:

```
%%R -i docks
```

```
mod <- lm(docks ~ latitude + longitude)
```

```
summary(mod)
```

Dataflow

We introduced Cloud Dataflow in Chapter 4 as a way to load data into BigQuery from MySQL. Cloud Dataflow is a managed service for executing pipelines written using Apache Beam. Dataflow is quite useful in data science since it provides a way to carry out transformations that would be hard to perform in SQL. At the time of writing, Beam pipelines can be written in Python, Java, and Go.

As an example of where this could be useful, consider the distribution of the length of bicycle rentals from an individual bicycle station:

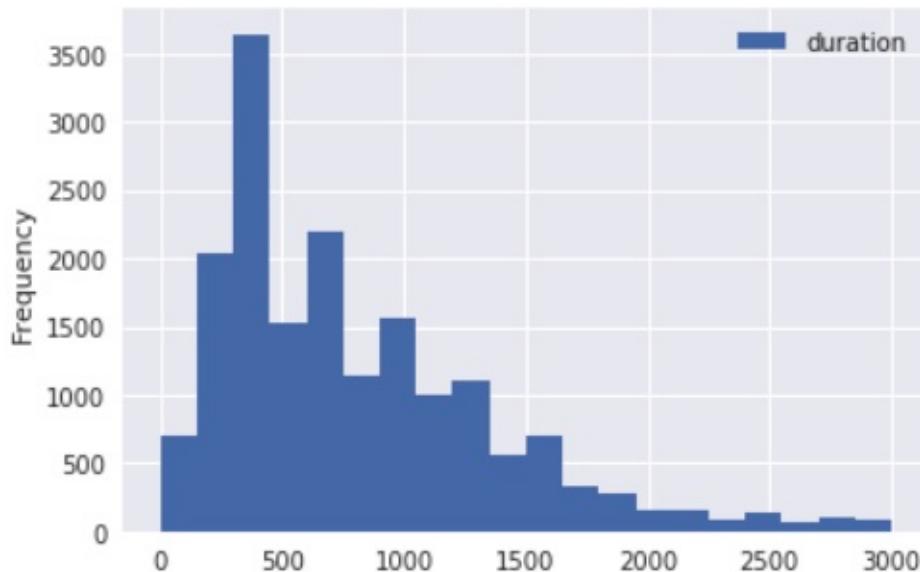


Figure 5-11. Distribution of the duration of bicycle rides from a single station

While the specific durations are available in the BigQuery table, it can be helpful to fit these values to a theoretical distribution so that we can carry out simulations and study the effect of pricing and availability changes

more readily. In Python, given an array of duration values, it is quite straightforward to compute the parameters of a Gamma distribution¹⁹ fit using the `scipy` package:

```
from scipy import stats  
  
ag,bg,cg = stats.gamma.fit(df['duration'])
```

Imagine that we want to go through all the stations and compute the parameters of the Gamma distribution fit to the duration of rentals from each of those stations. Because this is not convenient in SQL, but can be done quite easily in Python, we can write a Dataflow job to compute the Gamma fits in a distributed manner i.e. parallelize the computation of Gamma fits on a cluster of machines.

The pipeline starts with a query to pull the durations for each station, and then sends the resulting rows to the method `compute_fit`, and writes the resulting rows to BigQuery to the table `station_stats`:²⁰

```
opts = beam.pipeline.PipelineOptions(flags = [], **options)  
  
RUNNER = 'DataflowRunner'  
  
query = """  
  
SELECT start_station_id, ARRAY_AGG(duration) AS duration_array  
  
FROM `bigquery-public-data.london_bicycles.cycle_hire`  
  
GROUP BY start_station_id
```

```
'''
```

```
with beam.Pipeline(RUNNER, options = opts) as p:
```

```
(p
```

```
| 'read_bq' >> beam.io.Read(beam.io.BigQuerySource(query=query))
```

```
| 'compute_fit' >> beam.Map(compute_fit)
```

```
| 'write_bq' >> beam.io.gcp.bigquery.WriteToBigQuery(
```

```
'ch05eu.station_stats',
```

```
schema='station_id:string,ag:FLOAT64,bg:FLOAT64,cg:FLOAT64')
```

```
)
```

The compute_fit method is a Python function that takes in a dictionary corresponding to the input BigQuery row, and returns a dictionary corresponding to the desired output row:

```
def compute_fit(row):
```

```
    from scipy import stats
```

```
    result = {}
```

```
    result['station_id'] = row['start_station_id']
```

```
    durations = row['duration_array']
```

```
    ag, bg, cg = stats.gamma.fit(durations)
```

```
result['ag'] = ag
```

```
result['bg'] = bg
```

```
result['cg'] = cg
```

```
return result
```

The fit values are then written to a destination table.

After launching the Dataflow job, we can monitor it via the GCP console and see the job autoscaling to process the stations in parallel:

Autoscaling

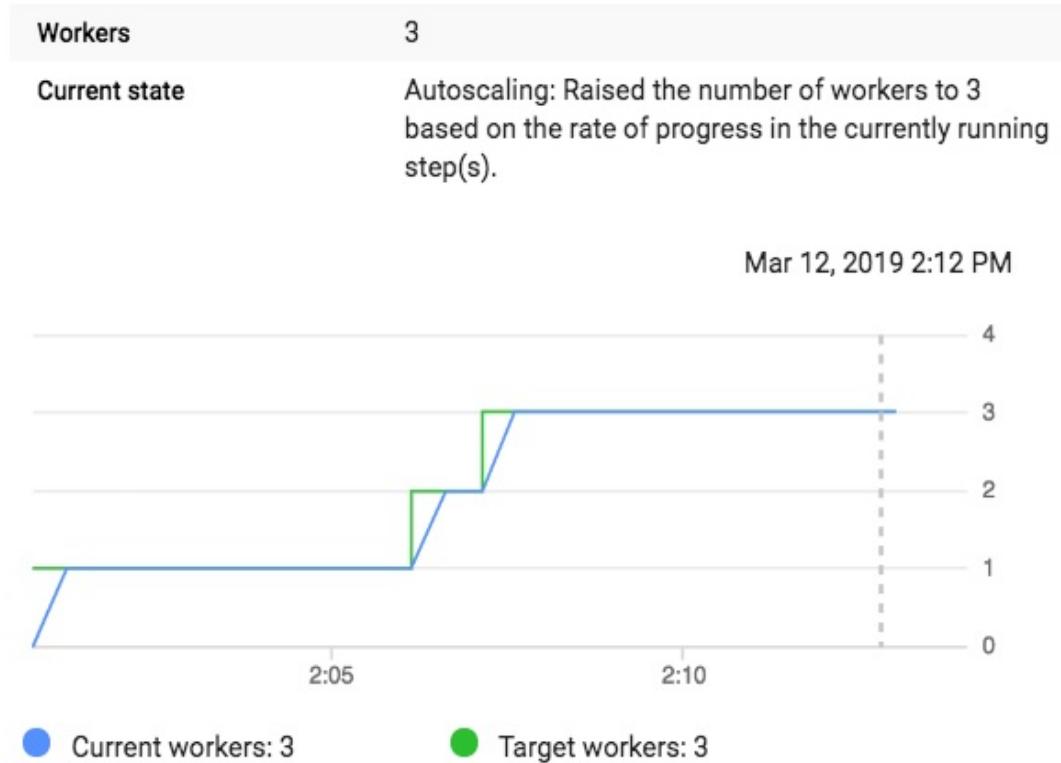


Figure 5-12. The Dataflow job is parallelized and run on a cluster whose size is autoscaled based on the rate of progress in each step

When the Dataflow job finishes, we can query the table and obtain statistics of the stations and plot the parameters of the Gamma distribution -- see the notebook²¹ on GitHub for the graphs.

JDBC/ODBC drivers

Since BigQuery is a warehouse for structured data, it can be convenient if database-agnostic APIs like JDBC and ODBC can be employed by a Java or .NET application to talk to a BigQuery database driver. This is not recommended for new applications -- use the client library instead. If, however, you have a legacy application that talks to a database today and has to be converted with minimal code changes to talk to BigQuery, the use of a JDBC/ODBC driver might be warranted.

<**warning**>

We strongly recommend the use of the client libraries (in the language of your choice) over the use of JDBC/ODBC drivers because the functionality exposed by the JDBC/ODBC driver is a subset of the full capabilities of BigQuery. Among the features missing are support for large scale ingestion (i.e. many of the loading techniques described in the previous chapter), large scale export (so data movement will be slow), and nested/repeated fields (preventing the use of many of the performance optimizations that will be covered in Chapter 7). Architecting new systems based on JDBC/ODBC drivers tends to lead to painful technical debt.

</**warning**>

Simba Technologies Inc is a Google partner that provides ODBC and JDBC drivers capable of executing BigQuery Standard SQL queries.²² To install the Java drivers, for example, you would download a zip file, unzip

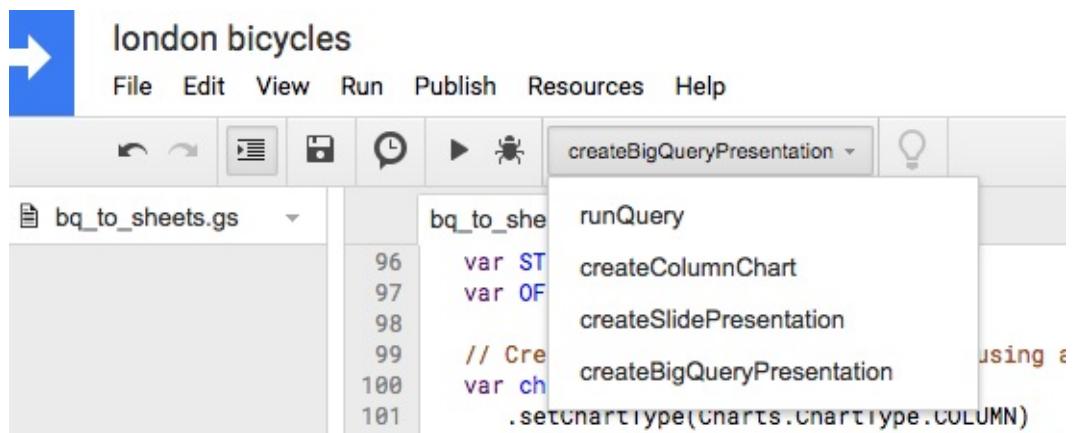
it, and place all the Java Archive (JAR) files in the zip folder in the classpath of your Java application. Using the Driver in a Java application typically involves modifying a configuration file that specifies the connection information. There are several options to configure the authentication and create a connection string that can be used within your Java application. Please see the Simba documentation²³ for details.

Incorporating BigQuery data into Google Slides (in GSuite)

It is possible to use Google Apps Script²⁴ to manage BigQuery projects, upload data, and execute queries. This is useful if you wish to automate the population of Google Documents, Google Sheets, or Google Slides with BigQuery data.

As an example, let's look at creating a pair of slides with analysis of the London bicycles data. Start by going to <https://script.google.com/create> to create a new script. Then, from the Resources menu, choose Advanced Google Services and flip on the bit for the BigQuery API (name the project if prompted).

The full Apps Script for this example is in the GitHub repository for this book²⁵, so copy the script and paste it into the text editor. Then, change the PROJECT_ID at the top of the script, choose the function createBigQueryPresentation and click on the run button:



The screenshot shows the Google Apps Script editor interface. The title bar says "london bicycles". The menu bar includes File, Edit, View, Run, Publish, Resources, and Help. Below the menu is a toolbar with icons for back, forward, search, and run. A dropdown menu titled "createBigQueryPresentation" is open, showing suggestions: "runQuery", "createColumnChart", "createSlidePresentation", "createBigQueryPresentation", and ".setChartType(charts.ChartType.COLUMN)". The main code editor window shows a script named "bq_to_sheets.gs" with the following code:

```
var ST  
var OF  
  
// Cre  
var ch  
  
.setChartType(charts.ChartType.COLUMN)
```

Figure 5-13. Use Google Apps Script to create a presentation from data in BigQuery

The resulting spreadsheet and slide deck will show up in Google Drive (you can also find their URLs by clicking on View > Logs). The slide deck will look something like this:

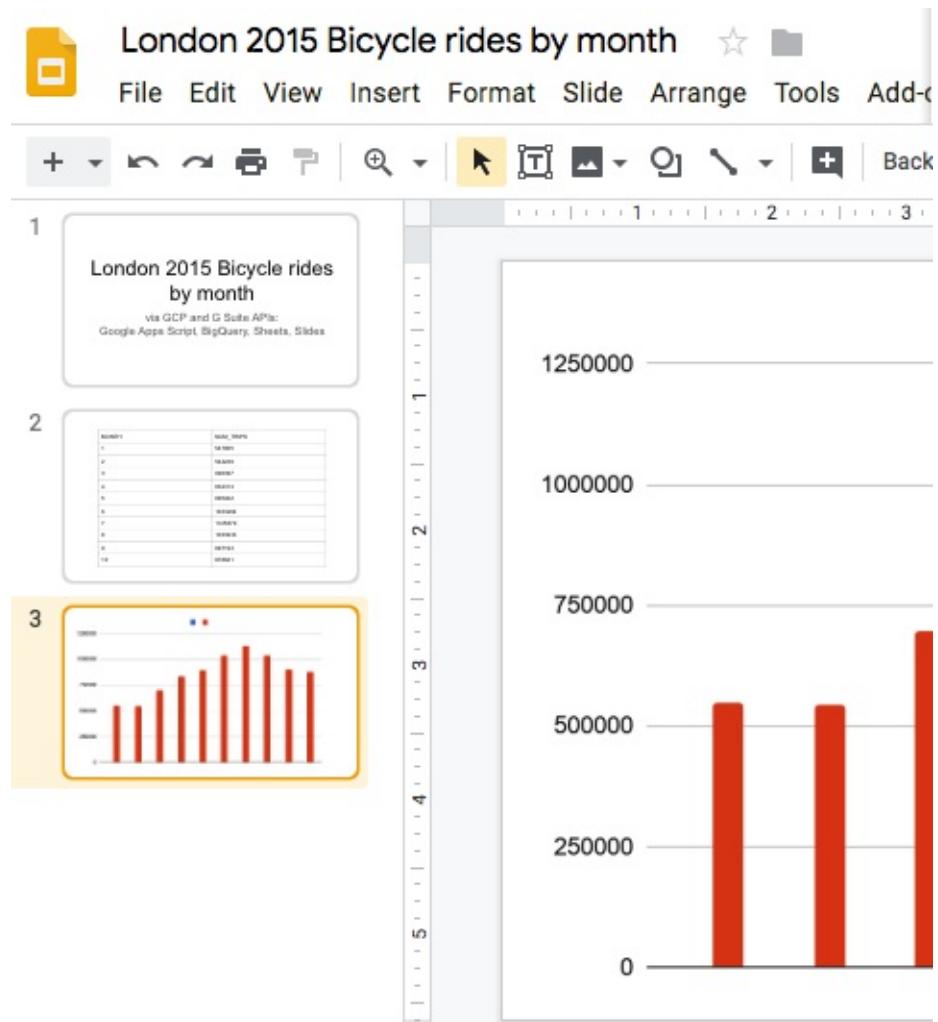


Figure 5-14. Slide deck created by the Google Apps Script

The function `createBigQueryPresentation` carried out the following code:

```
function createBigQueryPresentation() {
  var spreadsheet = runQuery();
  Logger.log('Results spreadsheet created: %s', spreadsheet.getUrl());
  var chart = createColumnChart(spreadsheet); // UPDATED
  var deck = createSlidePresentation(spreadsheet, chart); // NEW
```

```
Logger.log('Results slide deck created: %s', deck.getUrl()); // NEW  
}
```

Essentially, it calls three functions:

runQuery to run a query and store the results in a Google Sheets spreadsheet.

createColumnChart to create a chart from the data in the spreadsheet.

createSlidePresentation to create the output Google Slides slide deck.

The runQuery() method uses the Apps Scripts client library to invoke BigQuery and page through the results:

```
var queryResults = BigQuery.Jobs.query(request, PROJECT_ID);  
  
var rows = queryResults.rows;  
  
while (queryResults.pageToken) {  
  
  queryResults = BigQuery.Jobs.getQueryResults(PROJECT_ID, jobId, {  
  
    pageToken: queryResults.pageToken  
  
  });  
  
  rows = rows.concat(queryResults.rows);  
  
}  
}
```

Then, it creates a spreadsheet and adds these rows to the sheet. The other two functions employ Apps Scripts code to draw graphs, create a slide deck, and add both types of data to the slide deck.

Bash Scripting with BigQuery

The bq command-line tool that is provided as part of the Google Cloud Software Development Kit (SDK)²⁶ provides a convenient way to invoke BigQuery operations from the command-line. The SDK is installed by default on Google Cloud virtual machines and clusters. You can also download and install the SDK on your on-premises development and production environments.

You can use the bq tool to interact with the BigQuery service when writing bash scripts or by calling out to the shell from many programming languages without having to depend on the client library. Common uses of bq include creating and verifying the existence of datasets and tables, executing queries, loading data into tables, populating tables and views, and verifying the status of jobs. Let's look at these one-by-one.

Creating datasets and tables

To create a dataset, use bq mk and specify the location of the dataset (e.g., US, EU, etc.). It is also possible to specify non-default values for such things as the table expiration time. It is a best practice to provide a description for the dataset:

```
bq mk --location=US \
--default_table_expiration 3600 \
```

```
--description "Chapter 5 of BigQuery Book." \  
ch05
```

CHECKING IF DATASET EXISTS

The bq mk above fails if the dataset already exists. To create the dataset only if the dataset doesn't already exist, you have to list existing datasets using bq ls and check if that list contains a dataset with the name you desire:²⁷

```
#!/bin/bash
```

```
bq_safe_mk() {  
  
dataset=$1  
  
exists=$(bq ls --dataset | grep -w $dataset)  
  
if [ -n "$exists" ]; then  
  
echo "Not creating $dataset since it already exists"  
  
else  
  
echo "Creating $dataset"  
  
bq mk $dataset  
  
fi  
  
}
```

```
# this is how you call the function
```

```
bq_safe_mk ch05
```

CREATING A DATASET IN A DIFFERENT PROJECT

The dataset ch05 is created in the default project (specified when you logged into the VM or when you ran gcloud auth using the Cloud SDK). To create a dataset in a different project, qualify the dataset name with the name of the project the dataset should be created in:

```
bq mk --location=US \
```

```
--default_table_expiration 3600 \
```

```
--description "Chapter 5 of BigQuery Book." \
```

projectname:ch05

CREATING A TABLE

Creating a table is similar to creating a dataset except that we have to add --table to the bq mk command. The following creates a table named ch05.rentals_last_hour that expires in 3600 seconds and that has two columns named rental_id (a string) and duration (a float):

```
bq mk --table \
```

```
--expiration 3600 \
```

```
--description "One hour of data" \
```

```
--label persistence:volatile \
```

ch05.rentals_last_hour rental_id:STRING,duration:FLOAT

The label can be used to tag tables with characteristics; Data Catalog (covered in Chapter X) supports the ability to search for tables that have a specific label -- here persistence is the key and volatile is the label.

COMPLEX SCHEMA

For more complex schema that can not be easily expressed by a comma-separated string, specify a JSON file as explained in Chapter 4:

```
bq mk --table \  
  --expiration 3600 \  
  --description "One hour of data" \  
  --label persistence:volatile \  
  
ch05.rentals_last_hour schema.json
```

COPYING DATASETS

The most efficient way to copy datasets is through the command-line tool. For example, this copies a table from the ch04 dataset to the ch05 dataset:

```
bq cp ch04.old_table ch05.new_table
```

<tip>

Copying tables can take a while, but your script may not be able to proceed until the job is complete. An easy way to wait for a job to complete is to use bq wait:

```
bq wait --fail_on_error job_id
```

will wait forever until the job completes whereas:

```
bq wait --fail_on_error job_id 600
```

will wait a maximum of 600 seconds. If there is only one running job, you can omit the job_id.

</tip>

LOADING AND INSERTING DATA

We covered loading data into a destination table using bq load rather exhaustively in Chapter 4. Please see that chapter for details.

To insert rows into a table, write the rows as newline delimited JSON and use bq insert:

```
bq insert ch05.rentals_last_hour data.json
```

where the file data.json contains entries corresponding to the schema of the table being inserted into:

```
{"rental_id": "345ce4", "duration": 240}
```

EXTRACTING DATA

Data can be extracted from a BigQuery table to one or more files on Cloud Storage using bq extract:

```
bq extract --format=json ch05.bad_bikes gs://bad_bikes.json
```

Executing queries

To execute a query, use bq query and specify the query:

```
bq query \  
--use_legacy_sql=false \  
'SELECT MAX(duration) FROM `bigquery-public-  
data`.london_bicycles.cycle_hire'
```

The query string can also be provided via the standard input:

```
echo "SELECT MAX(duration) FROM `bigquery-public-  
data`.london_bicycles.cycle_hire" \  
| bq query --use_legacy_sql=false
```

Providing the query in a single string and escaping quotes, etc. can become quite cumbersome. For readability, use the ability of bash to read a multiline string into a variable:²⁸

```
#!/bin/bash  
  
read -d "QUERY_TEXT << EOF  
  
SELECT  
  
start_station_name  
  
, AVG(duration) as duration  
  
, COUNT(duration) as num_trips
```

```
FROM `bigquery-public-data`.london_bicycles.cycle_hire  
GROUP BY start_station_name  
ORDER BY num_trips DESC  
LIMIT 5  
EOF
```

```
bq query --project_id=some_project --use_legacy_sql=false  
$QUERY_TEXT
```

In the above code, we are reading into the variable QUERY_TEXT a multiline string that will be terminated by the word EOF. We can then pass that variable into bq query.

The code above also illustrates explicitly specifying the project that is to be billed for the query.

Remember to use --use_legacy_sql=false since the default dialect used by bq is not the Standard SQL that we cover in this book!

<tip>

SETTING FLAGS IN .BIGQUERYRC

If you tend to use the bq command-line interactively, it can be helpful to place common flags such as --location in \$BIGQUERYRC/.bigqueryrc or in \$HOME/.bigqueryrc if the environment variable \$BIGQUERYRC is not defined. Here is an example of a .bigqueryrc file:

```
--location=EU  
  
--project_id=some_project  
  
[mk]  
  
--expiration=3600  
  
[query]  
  
--use_legacy_sql=false
```

In the resource file above, all BigQuery commands will be invoked with --location=EU and billed to some_project while all bq mk will be invoked with --expiration=3600 and all bq query will be invoked with --use_legacy_sql=false. Explicitly specifying an --expiration on the command-line will override the value from the resource file.

If you do have a BigQuery resource file, be aware that any scripts you write or invoke will work differently on machines where you have this resource file installed (typically development machines) versus machines where you don't have the resource file (typically production machines). This can lead to a great deal of confusion. In our experience, any gains in productivity caused by having the resource file are canceled out by the increased debugging challenge when using the scripts on different machines. Your mileage may vary.

</tip>

PREVIEWING DATA

To preview a table, use bq head. Unlike a query of SELECT * followed by

`LIMIT`, this is deterministic and doesn't incur BigQuery charges.

To view the first 10 rows, we can do:

```
bq head -n 10 ch05.bad_bikes
```

To view the next 10 rows, we can do:

```
bq head -s 10 -n 10 ch05.bad_bikes
```

Note that the table is not actually not ordered, and so you should treat this as a way to read an arbitrary set of rows.

CREATING VIEWS

Views and materialized views can be created from queries using `bq mk`.

For example, this creates a view named `rental_duration` in the dataset `ch05`:

```
#!/bin/bash

read -d " QUERY_TEXT << EOF

SELECT

    start_station_name

    , duration/60 AS duration_minutes

FROM `bigquery-public-data`.london_bicycles.cycle_hire

EOF
```

```
bq mk --view=$QUERY_TEXT ch05.rental_duration
```

Views in BigQuery can be queried just like tables, but act like subqueries - - querying a view will bring the full text of the view into the calling query. Materialized views save the query results of the view into a table that is then queried. BigQuery takes care of ensuring that the materialized view is up-to-date. We will cover views and materialized views in more detail in Chapter X. To create a materialized view, replace --view in the above snippet by --materialized_view.

BigQuery objects

We looked at bq ls --dataset as a way to list the datasets in a project. There are other things you can list as well:

Command	What it lists
bq ls ch05	Tables in the dataset ch05
bq ls -p	All projects
bq ls -j some_project	All the jobs in the project
bq ls --dataset	All the datasets in the default project
bq ls --dataset some_project	All the datasets in the specified project
bq ls --models	List machine learning models
bq ls --transfer_run \ --filter='states:PENDING' \ --run_attempt='LATEST' \ projects/p/locations/l/transferConfigs/c	Transfer runs filtered to show only pending ones.
bq ls --reservation_grant \ --project_id=some_proj \ --location='us'	Reservation grants for slots in the specified project

SHOWING DETAILS

We can look at the details of a BigQuery object using bq show:

Command	Details of this object are shown
bq show ch05	The dataset ch05
bq show -j some_job_id	The specified job
bq show --schema ch05.bad_bikes	The schema of the table ch05.bad_bikes
bq show --view ch05.some_view bq show --materialized_view ch05.some_view	The view
bq show --model ch05.some_model	The model
bq show --transfer_run \ projects/p/locations/l/transferConfigs/c/runs/r	The transfer run

UPDATING

We can update the details of already created tables, datasets, etc. using bq update:

```
bq update --description "Bikes that need repair" ch05.bad_bikes
```

We can use bq update to update the query corresponding to a view or materialized view:

```
bq update \  
--view "SELECT ..."\n  
ch05.rental_duration
```

And even the size of a reservation (we will look at slots and reservations in Chapter X):

```
bq update --reservation --location=US \  
--project_id=some_project \
```

```
--reservation_size=2000000000
```

Summary

In this chapter, we looked at three different forms of BigQuery client libraries:

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 the Discovery Service to auto-generate language bindings in many programming languages.

A custom-built BigQuery client library that provides a convenient library for accessing BigQuery from a number of popular programming languages.

Of these, the recommended approach, provided one is available for your language of choice, is to use the BigQuery client library. If a BigQuery client library doesn't exist, use the Google API client, and only if you are working in an environment where even the API client is not available should you interact with the REST API directly.

There are a couple of higher level abstractions available that make programming against BigQuery easy in two commonly-used environments: Jupyter notebooks and shell scripts. We delved into the support for BigQuery from Jupyter and pandas and illustrated how the combination of these tools provides a powerful and extensible environment for sophisticated data science workflows. We also touched on integration with R and with GSuite and covered many of the capabilities

of the bq command line tool.

- 1 See <https://cloud.google.com/bigquery/docs/reference/rest/v2/>
- 2 Other APIs, especially non-REST APIs such as gRPC ones, will have a different API prefix.
- 3 See <https://cloud.google.com/bigquery/docs/reference/rest/v2/>
- 4 Specifically <https://www.googleapis.com/auth/bigquery> or <https://www.googleapis.com/auth/cloud-platform> have to be allowed.
- 5 This is the file 05_devel/rest_list.sh in the GitHub repository of this book -- <https://github.com/GoogleCloudPlatform/bigquery-oreilly-book>. You can run it anywhere where you have the Cloud SDK and curl installed (such as in CloudShell). Because we don't have a dataset yet, I'm using the dataset (ch04) you loaded in the previous chapter.
- 6 See <http://tldp.org/LDP/abs/html/here-docs.html>
- 7 See 05_devel/rest_query.sh in the GitHub repo.
- 8 See 04_devel/rest_query_async.sh in the GitHub.repo
- 9 See <https://cloud.google.com/docs/authentication/production>
- 10 See <https://pandas.pydata.org/>
- 11 This requires the pyarrow library. If you don't have it already, install it using pip install pyarrow.
- 12 Because pandas, by default, alphabetizes the column names, your BigQuery table will have a schema that is alphabetized and not in the order in which they appear in the tuples.
- 13 See <https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs#configuration.load.sourceUris> for the full set of support URIs.
- 14 To create a bucket in the EU region, use: **gsutil mb -l EU gs://some-bucket-name**.
- 15 Which 5 rows we will get is arbitrary because BigQuery does not guarantee ordering. The purpose of providing the start_index is so that we can get the “next page” of 5 rows by supplying start_index=5.
- 16 This is the same API used by the BigQuery web UI to show you the estimate.
- 17 This is the script 05_devel/launch_notebook.sh in the GitHub repository of this book.

- 18 At the time of writing, the PyTorch image for the Notebook Instance on GCP is built using conda.
- 19 See https://en.wikipedia.org/wiki/Gamma_distribution
- 20 See 05_devel/statfit.ipynb in the GitHub repository of this book.
- 21 See https://github.com/GoogleCloudPlatform/bigquery-oreilly-book/blob/master/05_devel/statfit.ipynb
- 22 See <https://cloud.google.com/bigquery/partners/simba-drivers/>
- 23 See <https://www.simba.com/drivers/bigquery-odbc-jdbc/>
- 24 See <https://developers.google.com/apps-script/advanced/bigquery> for more information on Apps Script.
- 25 See https://github.com/GoogleCloudPlatform/bigquery-oreilly-book/blob/master/05_devel/bq_to_slides.gs
- 26 See <https://cloud.google.com/sdk/>
- 27 This is, of course, subject to race conditions if someone else created the dataset in between your check and actual creation.
- 28 See 05_devel/bq_query.sh in the GitHub repository of this book.

About the Authors

Valliappa (Lak) Lakshmanan is a Tech Lead for Big Data and Machine Learning Professional Services on Google Cloud Platform. His mission is to democratize machine learning so that it can be done by anyone anywhere using Google's amazing infrastructure (i.e., without deep knowledge of statistics or programming or ownership of lots of hardware).

Jordan Tigani is engineering director for the core BigQuery team. He was one of the founding engineers on BigQuery, and helped grow it to be one of the most successful products in Google's Cloud Platform. He wrote the first book on BigQuery, and has also spoken widely on the subject. Jordan has twenty years of software development experience, ranging from Microsoft Research to Machine Learning startups.