CHAPTER 10

Google BigQuery

Big data is a widely used term that refers broadly to huge (and growing) datasets that no traditional data-processing application is able to capture, store, curate, analyze, and visualize in real time. Simply put, the data's rate of growth is so high that the tools cannot process incoming data, and the gap between processed and unprocessed data keeps increasing forever.

Several characteristics define big data. Here are the top three characteristics:

- *Volume*: This is the most important characteristic and probably the one that makes data *big data*. On-premises data-storage facilities cannot cater to true big data, because sooner or later you run out of either storage capacity or real estate. Using the public cloud is the right choice from day one.
- *Variety*: This describes the different types of data, including many formats within each type that cannot be stored efficiently using traditional data-storage systems such as structured relational database systems. Data can also be semi-structured or unstructured. Examples of variety include images, audio, video, documents, emails, RFID tags, cell phone GPS signals, and so on.
- *Velocity*: As the name suggests, this characteristic states the speed at which data needs to be captured, processed, and accessed. For example, during development, a web app may be tested with 10 data sources; but after deployment, the number of input sources may be 100,000 or more. Hence, the system should be able to scale without pre-warming. Also, once the data is available, it is important to have a fast channel to process it and generate value before the window of opportunity closes. Think about algorithmic trading to make buy/sell/hold decisions about equity in real time.

Big data is a natural fit for Google, which has a mission to organize the world's information and make it universally accessible and useful. To tackle this internal corporate problem, Google has built an arsenal of big-data tools that it uses across its product line, from search to Gmail. In Google Cloud Platform, Google makes some of these tools available so that external developers can use and process their own big data; meanwhile, Google uses its big-data technologies as the foundation for its other products. This chapter looks at Google BigQuery, and Chapter 11 discusses Google Cloud Dataflow.

The chapter presents BigQuery via a practice-oriented approach, using a real-life use case that analyzes the trends of visitors to a web site. You achieve this by analyzing the web server access log files: specifically, Apache web server, the most popular web server on the Internet today. You follow the entire data analysis extract, transform, load (ETL) lifecycle to load and query data using BigQuery.

Building Blocks and Fundamentals

BigQuery is the product of a research technology called Dremel, developed at Google Research (http://research.google.com/pubs/pub36632.html). BigQuery is a scalable, interactive, ad hoc query system for analysis of read-only nested data that is petabytes in size. These salient features make BigQuery a useful and unique hosted big data analytics platform. Let us now understand these characteristics in more detail.

- *Scalable*: Every data query in BigQuery may potentially run on thousands of CPUs across thousands of nodes. BigQuery dynamically selects the number of nodes and CPUs based on the size of the data in the tables. Because BigQuery is a hosted product, it costs the same if *x* CPUs work for *y* amount of time or if one CPU works for *x.y* amount of time. Of source, the former scenario returns results quickly, thereby saving you a precious resource—time—and thus BigQuery adopts this approach.
- *Interactive*: The key characteristic of an interactive system is speed. Every BigQuery query gets a tiny time-slice of a huge BigQuery cluster. This architecture means that you get your answers fast and don't need to wait to bring up a cluster just to run your jobs. Another collateral benefit is that you don't need to feed it work constantly to make it worth your while.
- *Ad-hoc*: The key characteristic of an ad-hoc system is the ability to support queries in all the stored data. BigQuery achieves this by doing a full-table scan on each query, without requiring indexes or pre-aggregation. Many other data analysis tools require you to design your data ingestion according to the types of questions you want to ask. BigQuery eliminates this burden and allows you to ask any question you want, after the data is already gathered.

When you use BigQuery, that combines the previous three characteristics, what you achieve is that instead of having to devote days to writing MapReduce code, or waiting hours between queries in order to refine or drill down into your data, you can get answers in seconds and dig into your data in real time.

BigQuery achieves the above goals through a shared multi-tenant architecture and by using distributed storage, columnar data layout, multilevel execution trees and is capable of running aggregation queries over trillion-row tables in seconds. Let us now understand few of the important implementation details.

- *Tree architecture*: BigQuery uses a multilevel hierarchical serving tree and execution engines to handle all incoming queries. The tree architecture comprises a root server, intermediate servers, and leaf servers. In this execution model, each incoming query is rewritten to increase execution efficiency. For example, the root server extract table metadata, partitions the query space, and rewrites the query before handing it over to the intermediate servers. Eventually, the root server is responsible for querying the data from the storage layer or from the local disk. Individual results are aggregated at each layer and passed back to the root server before being returned to the client.
- *Columnar layout & Distributed Storage*: In a relational data storage system such as a MySQL database, records are stored sequentially using a row-based storage engine. However, it is common for a query to extract only certain columns and not the entire record. This means the storage engine has to skip unwanted fields in a record to extract the required fields, resulting in higher computational costs and the use of more clock time. For the larger datasets typically associated with big data, this difference is obvious even if the database engine uses multiple query engines and uses horizontal data slicing. BigQuery uses columnar storage and stores the various columns separately. This enables the system to return only the queried columns,

resulting in lower computational cost and less time required. BigQuery uses a distributed storage backend to store your data. Distributed storage also helps make column storage more effective because each column can be read from a different spindle.

- *Shared Multi-tenant Architecture*: BigQuery is a shared, multi-tenant system that allows each query to get a tiny slice of time on thousands of dedicated nodes. So instead of having to bring up your own cluster and keep it busy to get your money's worth, you can get a tiny time-slice of a huge number of workers, in order to get your answer fast. Of course, you pay for only the compute and storage resources that your query actually uses and not leasing the entire cluster.

BigQuery stores data using a hierarchy of containers called *projects*, *datasets*, and *tables*. The associated actions of loading data, running queries, and exporting information are called BigQuery *jobs*:

- *Projects*: Projects are top-level containers in Google Cloud Platform. In addition to holding computing resources and user data, projects store information such as billing data and authorized users. Each Google Cloud Platform project is referred to by three identifiers:
  - *Project number*: Auto-assigned by Google Cloud Platform for the project's lifetime. The user cannot influence the project number.
  - *Project ID*: A string of three English words delimited by two hyphens. The user can choose a unique string at project-creation time, but once chosen, the ID cannot be changed later. The project ID is also used by the `gcloud` command-line tool and APIs to refer to the project.
  - *Project name*: A friendly description of the project for developer reference, consisting of a phrase. This description can be changed any number of times after the project is created.
- *Datasets*: Datasets are BigQuery–specific data containers and are one level lower in the hierarchy than projects. As the name suggests, a datasets is a collection of data that helps you organize and cluster data into groups. However, datasets do not contain any data themselves; the data is stored in tables. A dataset is simply a group of tables; every table must live inside a dataset. A dataset is assigned to a single project. In addition, datasets let you control access to tables by using access control lists (ACLs).
- *Tables*: Tables contain the data in BigQuery, along with a corresponding table schema that describes field names, types, and whether certain fields are mandatory or optional. Tables are required in various data import scenarios: when data is loaded into a new table name in a dataset, when an existing table is copied into a new table name, and when running queries. BigQuery also supports *views*, which are virtual tables defined by a SQL query.
- *Jobs*: Jobs are actions that are constructed by developers and executed by BigQuery on their behalf. Jobs include actions to load data, query data, export data, and copy data. Because BigQuery is typically used with large datasets, jobs may take a long time to execute. Hence, all jobs are executed asynchronously by BigQuery and can be polled for their status. BigQuery saves a history of all jobs associated with a project, and this list is accessible via all the three access methods: web-based Google Developers Console, gcloud command line tool and BigQuery API.

BigQuery officially supports the following data types. Each field in a table should be one of these types:

- `STRING`: 64KB UTF-8 encoded string.
- `INTEGER`: 64-bit signed integer.
- `FLOAT`: Double-precision floating-point format.
- `BOOLEAN`: True or false (case insensitive), or 1 or 0.
- `TIMESTAMP`: One of two formats—Unix timestamps or calendar date/times. BigQuery stores `TIMESTAMP` data internally as a Unix timestamp with microsecond precision.

UNIX TIMESTAMPS

A Unix timestamp is either a positive or a negative decimal number. A positive number specifies the number of seconds since the epoch (1970-01-01 00:00:00 UTC), and a negative number specifies the number of seconds before the epoch. The timestamp preserves up to six decimal places (microsecond precision).

DATE/TIME STRINGS

A date/time string is in the format YYYY-MM-DD HH:MM:SS. The `UTC` and `Z` attributes are supported. You can supply a time zone offset in date/time strings, but BigQuery doesn't preserve the offset after converting the value to its internal format. If you need to preserve the original time-zone data, store the time zone offset in a separate column. Date/time strings must be quoted when using JSON format.

Importing Data

As you learned in the previous section, you need a dataset in a project in order to load data. Listing 10-1, Listing 10-2, and Listing 10-3 use the `gcloud bq` command-line tool to list current projects and datasets and create a new dataset.

Listing 10-1. `gcloud bq` Command to List All Projects

```
$ bq ls -p
      projectId               friendlyName
 ------------------- --------------------------
  cloud-platform-book   Google Cloud Platform Book
  cloud-3rdpartyweb      cloud-3rdpartyweb
  cloud-sotp             cloud-sotp
  new-cloud             New Cloud
  new-cloud-taiwan      New Taiwan
  mythical-sky-823      Cloud VPN Platform
```

Listing 10-2. `gcloud bq` Command to List All Datasets in a Single Project

```
$ bq ls cloud-platform-book:
  datasetId
 -----------
  logs
```

Listing 10-3. `gcloud bq` Command to Create a New Dataset

```
$ bq mk apache_logs
Dataset 'cloud-platform-book:apache_logs' successfully created.
$ bq ls
   datasetId
 -------------
  apache_logs
  logs
```

Let's begin the journey of analyzing Apache web server log files using BigQuery. First you need to load the data into BigQuery and follow the ETL process. You start by extracting the log files from the web server `logs` directory. Following this, you transform the data into a BigQuery–supported file format. Finally, you load the data using the `bq` BigQuery command-line tool.

In the ETL process, transformation is the challenging part; this requires you to understand both the source and target data formats. The data formats may depend on the versions of the software and whether any customizations have been made as part of the installation process. The test setup uses Apache web server version 2.2.22 running on Ubuntu Linux LTS version 12.04. If you use a different web server or different version of the Apache web server, then the log format may be different. However, it should contain all the basic information required in a web server access log.

Transform Apache Access Log Files

In an Apache web server configuration (version 2.2.22 specifically), the file format of the access log entries is controlled by `CustomLog` directives in the main server configuration file located at `/etc/apache2/apache2.conf`. The `CustomLog` directives are then referred to from virtual host definitions written as individual files in the `/etc/apache2/sites-available/` directory. Listing 10-4, Listing 10-5, and 11-6 show the relevant sections of the configuration files to help you understand the format of the access log.

Listing 10-4. `CustomLog` Directives in `/etc/apache2/apache2.conf`

```
# The following directives define some format nicknames for use with
# a CustomLog directive (see below).
# If you are behind a reverse proxy, you might want to change %h into %{X-Forwarded-For}i
#
LogFormat "%v:%p %h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\""
vhost_combined
LogFormat "%h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\"" combined
LogFormat "%h %l %u %t \"%r\" %>s %O" common
LogFormat "%{Referer}i -> %U" referer
LogFormat "%{User-agent}i" agent
```

Listing 10-5. Environment Variable Declaration in `/etc/apache2/envvars`

```
export APACHE_LOG_DIR=/var/log/apache2$SUFFIX
```

Listing 10-6. `AccessLog` declaration of `/etc/apache2/sites-available/default`

```
CustomLog ${APACHE_LOG_DIR}/access.log combined
```

This configuration writes log entries in a format known as the Combined Log Format (CLF), which is a superset of the Common Log Format. This is a standard format used by many different web servers and read by many log-analysis programs.

You can see that each access-log entry uses the format `"%h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\"" from the previous extracts`. The format is specified using a format string that looks like a C programming language–style `printf` format string and that can be configured. The following entry shows an example log-file entry produced in CLF:

```
127.0.0.1 - kris [01/Apr/2015:15:44:00 +0800] "GET /fire_dragon.jpg HTTP/1.1" 200 2025
 "http://www.example.com/start.html" (\"%{Referer}i\") "Mozilla/5.0 (Macintosh; Intel Mac
 OS X 10_10_2)
 AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.104 Safari/537.36"
```

The parts of this log entry are as follows:

- `127.0.0.1` (`%h`): The IP address of the client (remote host) that made the request to the server. The default configuration is to store the IP address and not do a reverse lookup of the hostname. The IP address reported here is not necessarily the address of the client machine. If a proxy server exists between the user and the server, this address is the address of the proxy, rather than the originating machine.
- `-` (`%l`): The RFC 1413 identity of the client, determined by `identd` on the client's machine. The hyphen in the output indicates that the requested piece of information is not available.
- `kris` (`%u`): The user ID of the person requesting the URL, as determined by HTTP authentication. This field is `-` if the URL is not password protected or if the user is not yet authenticated, just as in the previous field.
- `[01/Apr/2015:15:44:00 +0800]` (`%t`): The time at which the request was received. The format is [*day*/*month*/*year*:*hour*:*minute*:*second zone*]:
  - *Day*: 2 digits
  - *Month*: 3 letters
  - *Year*: 4 digits
  - *Hour*: 2 digits
  - *Minute*: 2 digits
  - *Second*: 2 digits
  - *Zone*: (`+' | `-'), 4 digits

- `"GET /fire_dragon.jpg HTTP/1.1" (\"%r\")`: The request line from the client, stored in double quotes. The request line contains a great deal of useful information. First, the method used by the client is `GET`. Second, the client requested the resource `/fire_dragon.jpg`. And third, the client used the protocol `HTTP/1.1`.
- `200 (%>s)`: The status code that the server sends back to the client. This information is very valuable, because it reveals whether the request resulted in a successful response (codes beginning with 2), a redirection (codes beginning with 3), an error caused by the client (codes beginning with 4), or an error in the server (codes beginning with 5). You can find the full list of possible status codes in the HTTP specification (RFC 2616, section 10) at `www.w3.org/Protocols/rfc2616/rfc2616.txt`.
- `2025 (%b)`: The size of the object returned to the client, not including the response headers. If no content was returned to the client, this value is `-`. Two alternate format strings exist for this field: `%B` logs `0` instead of `-` for no content; and `%O` logs total bytes sent, including headers, in which case it is never zero in value.
- `"http://www.example.com/start.html" (\"%{Referer}i\")`: The `Referer` (sic) HTTP request header. This field provides the web site from which the client reports having been referred. In this example, this should be the page that links to or includes the image file `/fire_dragon.jpg`. In cases where there is no referral, the value is simply `-`.
- `"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.104 Safari/537.36" (\"%{User-agent}i\")`: The `User-Agent` HTTP request header. This is the identifying information that the client browser reports about itself to the web site and content. The format differs across browsers and in the same browser on different operating systems.

In an Ubuntu Linux system, Apache web server log files are stored in the `/var/log/apache2/` directory. In its default configuration, Apache web server archives its log files every week. Hence, if you run a web server for more than a year, you are likely to see up to 52 log files. The output in Listing 10-7 shows the location and log files in the test setup.

Listing 10-7. Location of Apache Web Server Access Log Files in an Ubuntu Linux System

```
$ pwd
/var/log/apache2

$ ls -l access.log*
-rw-r----- 1 root adm  4602633 Apr  1 10:46 access.log
-rw-r----- 1 root adm 25051247 Mar 29 06:28 access.log.1
-rw-r----- 1 root adm  1470908 Jan 25 06:39 access.log.2.gz
<snip>
-rw-r----- 1 root adm  1470908 Jan 25 06:39 access.log.52.gz
```

The amount of data in each of these files is of trivial size in BigQuery scale, so we decided to combine the data from all 52 log files into a single file in chronological order. You can use several tools to achieve this outcome. In this case, we used the Linux command `cat`.

You now need to transform the data in CLF into either a comma-separated values (CSV) or JSON-encoded text file for import into BigQuery. This example uses CSV format because the data is not nested and does not require normalization. Given that you know the record format of the entries in the log file, all you need is a regular expression that is able to parse the access log file and write a CSV file.

In order to make it easy for you, we researched and found online two simple scripts written in the Perl and PHP programming languages. We used both scripts to transform the data with equivalent results. These scripts are hosted on the Github code-hosting platform, at the following URLs:

- **https://github.com/woonsan/accesslog2csv** (Perl script)
- **https://github.com/mboynes/Apache-Access-Log-to-CSV-Converter** (PHP script)

We made a few changes to the Perl script. Let's examine the key parts, highlighting the old and new methods and the rationale behind them. The updated script is available in the source repository accompanying this book.

The first change we made to the transformation script was to change the field delimiter in the output CSV file from a comma (`,`) to a `\t` (tab) character. This is required because the new user-agent strings of client browsers, especially on mobile devices, use commas, as shown in the earlier access log field description. Listing 10-8 and Listing 10-9 show the original and updated sections of the Perl script.

Listing 10-8. Original: Uses Commas as Delimiters

```
print STDOUT "\"Host\",\"Log Name\",\"Date Time\",\"Time Zone\",\"Method\",\"URL\",
\"Response Code\",\"Bytes Sent\",\"Referer\",\"User Agent\"\n";
```

Listing 10-9. New: Uses Tab Characters as Delimiters

```
print STDOUT "\"Host\"\t\"Log Name\"\t\"Date Time\"\t\"Time Zone\"\t\"Method\"\t\"URL\"\t
\"Response Code\"\t\"Bytes Sent\"\t\"Referer\"\t\"User Agent\"\n";
```

In BigQuery, the table schema accompanying the table lists the number of fields in each record. BigQuery is very strict about the number of fields because it uses a columnar storage backend. This is the primary reason to make sure the field delimiter is not present in the any of the fields, which will unintentionally increase the number of fields. Although the tab character is not expected to be present in the incoming HTTP request, you want to make sure it doesn't exist either. Toward this end, we added a new section in the Perl script to search and replace any tab characters, as shown in Listing 10-10.

Listing 10-10. Replacing Tabs with the String `TAB` in Field Values

```
$host =~ s/\t/TAB/g;
$logname =~ s/\t/TAB/g;
$year =~ s/\t/TAB/g;
$month =~ s/\t/TAB/g;
$day =~ s/\t/TAB/g;
$hour =~ s/\t/TAB/g;
$min =~ s/\t/TAB/g;
$sec =~ s/\t/TAB/g;
$tz =~ s/\t/TAB/g;
$method =~ s/\t/TAB/g;
$url =~ s/\t/TAB/g;
$code =~ s/\t/TAB/g;
$bytesd =~ s/\t/TAB/g;
$referer =~ s/\t/TAB/g;
$ua =~ s/\t/TAB/g;
```

Once the transformation to CSV file format is complete, you need to load the file into BigQuery before you can query the data. There are two options to load the data to BigQuery. First, you can load the data to Cloud Storage and then import it into BigQuery. Or, second, you can load the data directly to BigQuery. We recommend the first option—loading through Cloud Storage—because we found it to be more reliable and faster.

Loading data through Cloud Storage takes three steps: loading to a Cloud Storage bucket, making an empty BigQuery dataset, and transferring data from Cloud Storage to BigQuery. The console output in Listing 10-11, Listing 10-12, and Listing 10-13 shows these three steps using the command-line tools and the corresponding output. We used the Linux `time` system utility to measure the time taken for each step.

Loading Transformed Data to BigQuery via Cloud Storage

Listing 10-11. Step 1: Copying the Consolidated Apache Access Logs to Google Cloud Storage

```
$ time gsutil cp apache-access-log.csv gs://cloud-platform-book/
Copying file://apache-access-log.csv [Content-Type=text/csv]...
==> NOTE: <snip>
Uploading   gs://cloud-platform-book/apache-access-log.csv:       1.38 GiB/1.38 GiB

real    0m50.816s
user    0m8.071s
sys     0m5.819s
```

Listing 10-12. Step 2: Making a New BigQuery Dataset in the Current Project

```
$ time bq mk logs_gs
Dataset 'cloud-platform-book:logs_gs' successfully created.

real    0m3.830s
user    0m0.269s
sys     0m0.083s
```

Listing 10-13. Step 3: Loading the Data from Cloud Storage into BigQuery

```
$ time bq load --source:format=CSV logs_gs.access gs://cloud-platform-book/
apache-access-log.csv apache-access-logs-table-schema.txt
Waiting on bqjob_r4ffba7ac3eee8771_0000014c680a39c3_1 ... (42s) Current status: DONE

real    0m48.706s
user    0m0.371s
sys     0m0.095s
```

Step 3 uses a file-based approach to declare the table schema for the new table being created in BigQuery. This is the recommended approach for tables that contain a large number of columns. The file describes each field by providing its name, its type, and whether it is a recommended filed. The contents of this file are as follows:

```
[
  {"name": "IP", "type": "string", "mode": "required"},
  {"name": "Time", "type": "timestamp", "mode": "required"},
  {"name": "Request_Type", "type": "string", "mode": "required"},
  {"name": "Path", "type": "string", "mode": "required"},
  {"name": "Response", "type": "integer", "mode": "required"},
  {"name": "Referral_Domain", "type": "string", "mode": "required"},
  {"name": "Referral_Path", "type": "string", "mode": "required"},
  {"name": "User_Agent", "type": "string", "mode": "required"}
]
```

From this output, you can see that the process took 102 seconds of clock time to load a 1.38GB log file. Listing 10-14 shows the BigQuery command to load data from the console and the corresponding console output. You can see that the route for loading data through Cloud Storage is not only more reliable but also much faster. Note that we performed both approaches on the same day using the same system and network, one after another.

Listing 10-14. Directly loading Data into BigQuery

```
$ time bq load --source:format=CSV logs.access ./apache-access-log.csv
./apache-access-logs-table-schema.txt
Waiting on bqjob_r47465a3df1f3e23c_0000014c67ef3a35_1 ... (145s) Current status: DONE

real    147m31.160s
user    0m10.358s
sys     0m4.804s
```

In this command, the table schema is supplied in a text file. It is also possible to supply the table schema as part of the command line, as shown in Listing 10-15.

Listing 10-15. `gcloud bq` Command with the Table Schema on the Command Line

```
$ time bq load --source:format=CSV apache_logs.access_logs
gs://cloud-platform-book/apache-access-log.csv  Host:string,LogName:string,Time:timestamp,
TimeZone:string,Method:string,URL:string,
ResponseCode:integer,BytesSent:integer,Referer:string,UserAgent:string
Waiting on bqjob_r60bbcaa2cd7d8fc9_0000014c7fd0a597_1 ... (16s) Current status: DONE

real    0m42.363s
user    0m0.349s
sys     0m0.093s
```

The following bq command and corresponding console output show the table schema, number of rows, and total size of the data in the table:

```
$ bq show cloud-platform-book:logs_gs.apache_access_logs
Table cloud-platform-book:logs_gs.apache_access_logs

   Last modified             Schema            Total Rows   Total Bytes   Expiration
 ----------------- -------------------------- ------------ ------------- ------------
  30 Mar 12:07:45   |- Host: string            7028137      1112270720
                    |- LogName: string
                    |- Time: timestamp
                    |- TimeZone: string
                    |- Method: string
                    |- URL: string
                    |- ResponseCode: integer
                    |- BytesSent: integer
                    |- Referer: string
                    |- UserAgent: string
```

You are now ready to query the data!

Querying Data

In this section, you query the Apache web server access logs that you loaded to BigQuery in the previous section. You exclusively use the gcloud command-line tool to construct and execute the queries. The objective is to showcase the strengths and diversities of the SQL dialect supported by BigQuery, not to extract every possible statistics from the dataset. We ask a set of questions, discuss the answers, construct the query while explaining the syntax, execute the query, and finally interpret the results.

Here is Q1:

```
Legend: Objective = O, Query = Q, Response = R, Analysis = A

O1: List selected fields - Host, Time, URL, ResponseCode, BytesSent - of the first 5 records
in the data

Q1: SELECT Host, Time, URL, ResponseCode, BytesSent FROM [logs_gs.apache_access_logs] LIMIT 5

R1:
$ bq shell
Welcome to BigQuery! (Type help for more information.)
cloud-platform-book> SELECT Host, Time, URL, ResponseCode, BytesSent FROM
[logs_gs.apache_access_logs] LIMIT 5
```

```
Waiting on bqjob_r7af302e211713989_0000014c7e228796_1 ... (0s) Current status: DONE
+-----------+---------------------+-----+--------------+-----------+
|   Host    |        Time         | URL | ResponseCode | BytesSent |
+-----------+---------------------+-----+--------------+-----------+
| 127.0.0.1 | 2015-03-27 07:32:04 | *   |          200 |       126 |
| 127.0.0.1 | 2015-03-27 07:32:05 | *   |          200 |       126 |
| 127.0.0.1 | 2015-03-27 07:32:06 | *   |          200 |       126 |
| 127.0.0.1 | 2015-03-27 07:32:07 | *   |          200 |       126 |
| 127.0.0.1 | 2015-03-27 07:32:08 | *   |          200 |       126 |
+-----------+---------------------+-----+--------------+-----------+
cloud-platform-book> quit
Goodbye.
```

A1: We used the BigQuery shell to run an interactive query to meet this objective. The BigQuery shell is available as part of the Google Cloud SDK and can be easily started using the commands `bq shell`. Once inside the shell, you can type SQL queries just as you would in a regular SQL shell. When you are finished using the BigQuery shell, you can exit by using the `quit` command.

Let's move on to Q2:

```
Q2: What are the top 10 traffic-generating client IPs to our website?

Q2: SELECT Host, Count(Host) As Hits FROM [logs_gs.apache_access_logs]
Group by Host Order by Hits DESC LIMIT 10

R2:
Waiting on bqjob_r38c3621dab252a3c_0000014c7d08d446_1 ... (0s) Current status: DONE
+----------------+---------+
|      Host      |  Hits   |
+----------------+---------+
| 127.0.0.1      | 2127406 |
| 10.128.215.151 | 1609667 |
| 10.142.45.129  | 1400049 |
| 10.130.86.213  | 1094896 |
| 10.138.25.97   |  620975 |
| 204.246.166.52 |    9072 |
| 54.240.148.108 |    7847 |
| 54.240.148.78  |    6397 |
| 54.239.129.82  |    4636 |
| 204.246.166.16 |    4371 |
+----------------+---------+
```

A2: The `Host` field from the table schema contains the IP address. The query requests that BigQuery count the number of times each IP address appears in the entire dataset and label the total `Hits`. The output is then grouped by IP address so that each IP address appears only once, and the results are sorted by the value of the `Hits` column and in descending order. This query shows that temporary columns in results can be named and referred to later in the query.

PRIVATE IP ADDRESSES

In the Internet addressing architecture, a *private IP network* is a network that uses private IP addresses. This private IP address is defined by RFC 1918 for Internet Protocol Version 4 (IPv4) and RFC 4193 for Internet Protocol Version 6 (IPv6). The purpose of defining a private IP range is to enable local networking among IP-enabled systems whose traffic need not be routed globally over the public Internet. Public routers will drop such traffic. As such, private IP ranges are not allotted to organizations globally.

Private IP ranges are used extensively in home and office networks. A network address translation (NAT) router or a proxy server is required if the source traffic from a system with a private IP address must be routed over the public Internet.

The following are the private addresses in the IPv4 address range:

```
10.0.0.0 - 10.255.255.255: Class A IP range
172.16.0.0 - 172.31.255.255: Class B IP range
192.168.0.0 - 192.168.255.255: Class C IP range
```

From the Q2 results, you can see that the top five IP addresses that generate traffic to the web site are in the private IP address range. Because private IP addresses are not routable on the public Internet, this traffic is from private networks where the virtual machine (VM) is hosted. Let's query all the traffic from the private IP address range.

The `Host` field is defined as type `string` in the table schema. To accommodate all possible private IPs from the private IP range, you need to convert this into a numeric value so that you can apply range operators to filter them from the dataset. Fortunately, BigQuery offers a function called `PARSE_IP` that converts an IPv4 address into an unsigned number. The URL `https://cloud.google.com/bigquery/query-reference` defines `PARSE_IP` as follows:

> *PARSE_IP(readable_ip): Converts a string representing IPv4 address to unsigned integer value. For example, PARSE_IP('0.0.0.1') will return 1. If string is not a valid IPv4 address, PARSE_IP will return NULL.*

The `PARSE_IP` function breaks the IPv4 address into four octets and uses the base value of 256 along with a digit placeholder value to arrive at the unsigned integer number. Here is an example, to make this process easier to follow.

Let's convent 10.0.0.0 into an unsigned number:

```
= (first octet * 2563) + (second octet * 2562) + (third octet * 256) + (fourth octet)
= (first octet * 16777216) + (second octet * 65536) + (third octet * 256) + (fourth octet)
= (10 * 16777216) + (0 * 65536) + (0 * 256) + (0)
= 167772160
```

The following are the unsigned numbers of the border IP address in each of the private IP ranges:

```
10.0.0.0 - 10.255.255.255 : 167772160 - 184549375
172.16.0.0 - 172.31.255.255 : 2886729728 - 2887778303
192.168.0.0 - 192.168.255.255 : 3232235520 - 3232301055
```

Let's use this function along with logic OR operator to construct a BigQuery query and filter all the internal traffic from private IP addresses to the web site:

```
Q3: What are the top 10 traffic-generating private IP clients to our website?

$ bq query "Select Host, PARSE_IP(Host) as IPvalue, count(Host) as Hits
from [logs_gs.apache_access_logs] where PARSE_IP(Host) >= 167772160 AND
PARSE_IP(Host) <= 184549375 group by Host, IPvalue"
Waiting on bqjob_r517ed70cd5f77b7e_0000014c7f175436_1 ... (0s) Current status: DONE
+----------------+-----------+---------+
|      Host      |  IPvalue  |  Hits   |
+----------------+-----------+---------+
| 10.142.45.129  | 177089921 | 1400049 |
| 10.138.25.97   | 176822625 |  620975 |
| 10.130.86.213  | 176314069 | 1094896 |
| 10.128.215.151 | 176215959 | 1609667 |
+----------------+-----------+---------+
```

Here is the second part of Q3:

```
Q3-2: Find hosts with IP address between 172.16.0.0 - 172.31.255.255

$ bq query "Select Host, PARSE_IP(Host) as IPvalue, count(Host) as Hits from
 [logs_gs.apache_access_logs] where PARSE_IP(Host) >= 2886729728 AND
PARSE_IP(Host) <= 2887778303 group by Host, IPvalue"
Waiting on bqjob_r2f39df990fdbdb4_0000014c7f1a1cdb_1 ... (0s) Current status: DONE
```

The third part of Q3 is as follows:

```
Q3-3: Find hosts with IP address between 192.168.0.0 - 192.168.255.255

$ bq query "Select Host, PARSE_IP(Host) as IPvalue, count(Host) as Hits from
 [logs_gs.apache_access_logs] where PARSE_IP(Host) >= 3232235520 AND
PARSE_IP(Host) <= 3232301055 group by Host, IPvalue"
Waiting on bqjob_r64e5a0007cd9de80_0000014c7f1aeacf_1 ... (0s) Current status: DONE
```

The fourth part of Q3 is as follows:

```
Q3-4: Find hosts with IP address in all private IP address range

$ bq query "Select Host, PARSE_IP(Host) as IPvalue, count(Host) as Hits
from [logs_gs.apache_access_logs] where \
PARSE_IP(Host) >= 167772160 AND PARSE_IP(Host) <= 184549375 OR \
PARSE_IP(Host) >= 2886729728 AND PARSE_IP(Host) <= 2887778303 OR \
PARSE_IP(Host) >= 3232235520 AND PARSE_IP(Host) <= 3232301055 \
group by Host, IPvalue order by hits desc limit 10"
Waiting on bqjob_r75b709c83f874113_0000014c7f237be7_1 ... (2s) Current status: DONE
+----------------+-----------+---------+
|      Host      |  IPvalue  |  Hits   |
+----------------+-----------+---------+
| 10.128.215.151 | 176215959 | 1609667 |
| 10.142.45.129  | 177089921 | 1400049 |
| 10.130.86.213  | 176314069 | 1094896 |
| 10.138.25.97   | 176822625 |  620975 |
+----------------+-----------+---------+
```

And here is the last part of Q3:

```
Q3-5: Count the total traffic from private IP

$ bq query "Select count(Host) as Total from [logs_gs.apache_access_logs] where \
> PARSE_IP(Host) >= 167772160 AND PARSE_IP(Host) <= 184549375 OR \
> PARSE_IP(Host) >= 2886729728 AND PARSE_IP(Host) <= 2887778303 OR \
> PARSE_IP(Host) >= 3232235520 AND PARSE_IP(Host) <= 3232301055"
Waiting on bqjob_r7507383b330c328c_0000014c7f39bedd_1 ... (0s) Current status: DONE
+---------+
|  Total  |
+---------+
| 4725587 |
+---------+
```

In addition to the traffic from the private IP address range, which is presumably generated by infrastructure-monitoring services, there is also traffic from localhost or IP address 127.0.0.1. Human beings generated this traffic. Let's use a different query to discover the real traffic from public IP addresses only:

```
Q4. What is the top 10 public IP that are generating traffic to our website? What is the
total traffic to our website?
Q4.
# BigQuery query to discover top 10 public IP traffic generators to your web site

$ bq query "Select Host, count(Host) as Hits from [logs_gs.apache_access_logs] where \
Host != '127.0.0.1' AND (PARSE_IP(Host) < 167772160 OR PARSE_IP(Host) > 184549375) \
group by Host order by Hits DESC LIMIT 10"
```

```
Waiting on bqjob_r595bf414cb7bc939_0000014c7f648bed_1 ... (0s) Current status: DONE
+----------------+------+
|      Host      | Hits |
+----------------+------+
| 204.246.166.52 | 9072 |
| 54.240.148.108 | 7847 |
| 54.240.148.78  | 6397 |
| 54.239.129.82  | 4636 |
| 204.246.166.16 | 4371 |
| 54.240.148.85  | 3464 |
| 54.239.196.108 | 3448 |
| 54.251.52.153  | 2958 |
| 216.137.42.83  | 2909 |
| 54.240.158.56  | 2743 |
+----------------+------+

# BigQuery query to find out the total traffic to your web site

$ bq query "Select count(Host) as Total from [logs_gs.apache_access_logs] where \
> Host != '127.0.0.1' AND (PARSE_IP(Host) < 167772160 OR PARSE_IP(Host) > 184549375)"
Waiting on bqjob_r79f024b87cbba095_0000014c7f745729_1 ... (0s) Current status: DONE
+--------+
| Total  |
+--------+
| 175144 |
+--------+
```

A4: This query inverts the operators associated with the private IP range and includes the localhost IP address. You also compute the total traffic from public IP addresses using the `count` operation.

We hope the examples in this section have shown you the query variety and possibilities available when using BigQuery. See https://cloud.google.com/bigquery/query-reference for all the SQL functions you can use in BigQuery SQL queries.

Exporting Data and Creating Views

After you have imported and queried a massive dataset using BigQuery, you may need to export part of the data from BigQuery to be processed by another system, such as visualization, or to present it as part of a management report. You can do this easily with BigQuery, which supports data export to Google Cloud Storage.

The most common use case of the export feature is to export the results of a query. Let's export two pieces of useful information from the raw dataset: a list of unique IPs that have generated traffic to the web site, and a list of unique URLs in the web site that are accessed by client and their corresponding hit counts. You use this information in next section when you visualize it. You can use the `--destination_table` feature in BigQuery to export the results of queries and save the filtered result in two new tables, as shown in Listing 10-16 and Listing 10-17.

Listing 10-16. BigQuery Query to Save the List of Public IPs to a New Table

```
$ bq query --destination_table logs_gs.unique_ip_hits_table "Select Host, count(Host) as Hits
from [logs_gs.apache_access_logs] where \
> Host != '127.0.0.1' AND (PARSE_IP(Host) < 167772160 OR PARSE_IP(Host) > 184549375) \
> group by Host order by Hits DESC"
Waiting on bqjob_rb85945d06b195d6_0000014c8340f6bc_1 ... (0s) Current status: DONE
+-----------------+------+
|      Host       | Hits |
+-----------------+------+
| 204.246.166.52  | 9072 |
| 54.240.148.108  | 7847 |
| 54.240.148.78   | 6397 |
| 54.239.129.82   | 4636 |
| 204.246.166.16  | 4371 |
| 54.240.148.85   | 3464 |
| 54.239.196.108  | 3448 |
| 54.251.52.153   | 2958 |
| 216.137.42.83   | 2909 |
| 54.240.158.56   | 2743 |
| <snip>          |      |
+-----------------+------+
```

Listing 10-17. BiqQuery Query to Save the List of Unique URLs to a New Table

```
$ ls -l query.bash
-rwxr-xr-x  1 sptkrishnan  staff  305 Apr  4 18:26 query.bash

$ cat query.bash
#!/bin/bash
bq query --destination_table logs_gs.unique_urls "select REGEXP_REPLACE(URL,
r'(\?[a-zA-Z0-9-=._&/,+?();%:#*!@^~<>]*)','') as BASE_URL, count(URL) as Hits from
 logs_gs.apache_access_logs where NOT REGEXP_MATCH(URL,
r'\.[a-zA-Z0-9-=._&/,+?();%:#*!@^~<>]+') group by BASE_URL order by Hits DESC"

$ ./query.bash
Waiting on bqjob_r12c153c56cc766c3_0000014c83f78ac5_1 ... (1s) Current status: DONE
+-----------------------------------------------------+---------+
|                      BASE_URL                       |  Hits   |
+-----------------------------------------------------+---------+
| *                                                   | 2127408 |
| /                                                   |  153184 |
| <snip>                                              |  <snip> |
+-----------------------------------------------------+---------+
```

This query also uses BigQuery's regular-expressions capabilities. You now export this information from BigQuery to Cloud Storage and then to the local system, as shown in Listings 10-18 through 10-23.

Listing 10-18. BiqQuery Command to Copy a Unique IP Table as a CSV File to Cloud Storage

```
$ bq extract logs_gs.unique_ip_hits_table gs://cloud-platform-book/public-ip-hits.csv
Waiting on bqjob_r62f2820390d0cc7d_0000014c8424a54a_1 ... (42s) Current status: DONE
```

Listing 10-19. Cloud Storage Command to Copy the `public-ip-hits.csv` File to the Local System

```
$ gsutil cp gs://cloud-platform-book/public-ip-hits.csv public-ip-hits.csv
Copying gs://cloud-platform-book/public-ip-hits.csv...
Downloading file://public-ip-hits.csv:                          31.9 KiB/31.9 KiB
```

Listing 10-20. Listing the File Contents to Make Sure They Are the Same as the BigQuery Table

```
$ head -n 6 public-ip-hits.csv
Host,Hits
204.246.166.52,9072
54.240.148.108,7847
54.240.148.78,6397
54.239.129.82,4636
204.246.166.16,4371
```

Listing 10-21. BiqQuery Command to Copy a Unique URL Table as a CSV File to Cloud Storage

```
$ bq extract logs_gs.unique_urls gs://cloud-platform-book/unique_urls.csv
Waiting on bqjob_r195f68e14729f204_0000014c842bff76_1 ... (42s) Current status: DONE
```

Listing 10-22. Cloud Storage Command to Copy the `unique_urls.csv` File to the Local System

```
$ gsutil cp gs://cloud-platform-book/unique_urls.csv unique_urls.csv
Copying gs://cloud-platform-book/unique_urls.csv...
Downloading file://unique_urls.csv:                          120.9 KiB/120.9 KiB
```

Listing 10-23. Listing the File Contents to Make Sure They Are the Same as the BiqQuery Table

```
$ head -n 3 unique_urls.csv
BASE_URL,Hits
*,2127408
/,153184
```

In addition to exporting a table's contents as a CSV file, BigQuery can also export the contents into JSON and Apache AVRO format (http://avro.apache.org).

BigQuery also supports the creation of a SQL view, which is defined as a virtual table that stores the results of a query. Listing 10-24 gives the command template for defining a new view.

Listing 10-24. BigQuery `bq` Console Command Template to Create a View

```
$ bq mk --view="<query>" <dataset>.<view name>
```

Listing 10-25 and Listing 10-26 use this template to create a virtual table that stores all the unique public IPs and the number of hits from each of them.

Listing 10-25. Creating a BigQuery View that Extracts All Public IPs and Hits

```
$ bq mk --view "Select Host, count(Host) as Hits from [logs_gs.apache_access_logs] where \
Host != '127.0.0.1' AND (PARSE_IP(Host) < 167772160 OR PARSE_IP(Host) > 184549375) \
group by Host order by Hits DESC" logs_gs.unique_ip_hits_view
View 'cloud-platform-book:logs_gs.unique_ip_hits_view' successfully created.
```

Listing 10-26. Query BigQuery View to Find the Total Number of Public IPs.

```
$ bq query "select count(Host) as Total from logs_gs.unique_ip_hits_view"
Waiting on bqjob_r22c7d94650ebca23_0000014c82949925_1 ... (0s) Current status: DONE
+-------+
| Total |
+-------+
|  1964 |
+-------+
```

You can neither export the results from a view nor copy them into a real table. Hence, you should use views where you may want to rerun the underlying query transparently without retrying it again and again. This is useful when the original table has new data appended either through a load job or via data streaming.

Summary

This chapter introduced BigQuery, the big data–analysis tool from Google Cloud Platform. BigQuery, like other Google Cloud Platform products, does not charge up front for using or reserving resources. BigQuery makes it easy for individuals and organizations of any size to start using it while paying a low usage fee via a true pay-per-use model.

We guided you step by step through preparing data (using the ETL process), loading the data, running queries, and exporting filtered results. The exported results can be visualized using a variety of software such as Google Charts. We hope this example-based chapter has helped you to understand how to put BigQuery to work.