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 only 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 in February 2016. “I can start with 2.2Billion ‘things’ and compute/summarize down to 20K in < 1 min.” The scale and speed 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 toward agility and openness, plays a big part in increasing the pace of innovation. For example, Twitter recently reported in its blog that it was able to democratize data analysis with BigQuery by providing some of its most frequently used tables to Twitter employees from a variety of teams (Engineering, Finance, and Marketing were mentioned).

For Alpega Group, a global logistics software company, the increased innovation and agility offered by BigQuery were key. The company went from a situation in which real-time analytics was impossible to being able to provide fast, customer-facing analytics in near real time. Because Alpega Group does not need to maintain clusters and infrastructure, its small tech team is now free to work on software development and data capabilities. “That was a real eye opener for us,” says the company’s 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."

Imagine that you run a chain of equipment rental stores. 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:

1. Where the item was rented
2. When it was rented
3. Where the item was returned
4. When it was returned

Perhaps you record the transaction in a database every time a customer returns an item.[1]

From this dataset, you would like to find out how many "one-way" rentals occurred every month in the past 10 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

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, such as the following, 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 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 preceding query 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 this one are quite inefficient for an OLTP database to carry out. OLTP databases are tuned toward data consistency; the point is that you can read from the database even while data is simultaneously being written to it. This is achieved through careful locking to maintain data integrity. For the filtering on `station_name` to be efficient, you would need 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 hoc[2] 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.[3] This paradigm, known as *MapReduce*, became hugely influential and led to the development of Apache Hadoop.

Although the Hadoop ecosystem began with a library that was primarily built in Java, custom analysis on Hadoop clusters is now typically carried out using Apache Spark. 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:

1. Periodically export transactions to comma-separated values (CSV) text files in the Hadoop Distributed File System (HDFS).
2. For ad hoc analysis, write a Spark program that does the following:
    a. Loads up the data from the text files into a “DataFrame”
    b. Executes an SQL query, similar to the query in the previous section, except that the table name is replaced by the name of the DataFrame

    c. Exports the result set back to a text file

3. Run the Spark program on a Hadoop cluster.

Although 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 it usually requires that the compute nodes access data that is local to them. The HDFS must, therefore, be sharded across the compute nodes of the cluster. With both data sizes and analysis needs increasing dramatically but independently, it is often the case that clusters are underprovisioned or overprovisioned.[4] Thus, the need to execute Spark programs on a Hadoop cluster means that your organization will need to become expert in managing, monitoring, and provisioning Hadoop clusters. This might not be your core business.

BigQuery: A Serverless, Distributed SQL Engine

What if you could run SQL queries as in a Relational Database Management System (RDBMS) system, obtain efficient and distributed traversal through the entire dataset as in MapReduce, and not need 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 the need 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 and paste the following query in the window,[5] and then click the “Run query” button:

```
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 (UI) reported that the query involved processing 2.51 GB and gave us the result in about 2.7 seconds, as illustrated in Figure 1-1.

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

The equipment being rented out is bicycles, and so the preceding query 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 its 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 is 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 you[6] “view” access to this dataset. You didn’t need to start up a cluster or log in to one. Instead, you just submitted a query to the service and received 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 does 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 learn whether there are fewer bicycle rentals on rainy days?[7]

```
-- 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:[8]

```
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, and 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 format[9] that makes ETL an attractive methodology. The data pipeline, typically written in either Apache Beam or Apache Spark, extracts the necessary bits from the raw data (either streaming data or batch files), transforms what it has extracted to do any necessary cleanup or aggregation, and then loads it into BigQuery, as demonstrated in Figure 1-2.

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

Even though 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 (Extract and Load) 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 needs to stream data into BigQuery as it arrives, 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 (CI) and unit testing systems.

Besides the ETL and EL workflows, BigQuery makes it possible to do an Extract, Load, and Transform (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 in flux. For example, you might still be carrying out exploratory work to determine whether a particular timestamp needs to be corrected for the local time zone. 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 we’ve prepared a quick summary in Table 1-1.

Workflow	Architecture	When you’d use it
EL	Extract data from files on Google Cloud Storage. Load it into BigQuery’s native storage. You can trigger this from Cloud Composer, Cloud Functions, or scheduled queries.	Batch load of historical data. Scheduled periodic loads of lo
ETL	Extract data from Pub/Sub, Google Cloud Storage, Cloud Spanner, Cloud SQL, etc. Transform the data using Cloud Dataflow. Have Dataflow pipeline write to BigQuery	When the raw data needs to be transformed, or enriched befor BigQuery. When the data loading needs t the use case requires streaming When you want to integrate wi integration/continuous delivery perform unit testing on all com
ELT	Extract data from files in Google Cloud Storage. Store data in close-to-raw format in BigQuery. Transform the data on the fly using BigQuery views.	Experimental datasets where y of transformations are needed Any production dataset where expressed in SQL.

Table 1-1. Summary of workflows, sample architectures, and the scenarios i

The workflows in Table 1-1 are in the order that we usually recommend.

Powerful Analytics

The benefits of a warehouse derive from the kinds of analyses that you can do with the data held within it. The primary way you interact with BigQuery is via SQL, and because BigQuery is an 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. By clicking the “Explore in Data Studio” button in the BigQuery web UI, for example, we can quickly create a visualization of how our one-way bike rentals vary by month, as depicted in Figure 1-3.

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 the need 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.[10] BigQuery is, therefore, a powerful engine to carry out descriptive analytics.

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

Another BigQuery extension to standard SQL supports creating machine learning models and carrying out batch predictions. We cover the machine learning capability of BigQuery in detail in Chapter 9, 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. Even though 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 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 of Management

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. As of this writing, the cost of storage automatically drops to lower levels if a table is not updated for 90 days.[11]

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 talk about performance tuning, but performance tuning in BigQuery consists mainly of clear thinking and the appropriate choice of SQL functions. You will not need 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 need 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 need 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 multitenant 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 craft 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 need 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 pioneer. When you have "big data," Gray said, "you want to move the computation to the data, rather than move the data to the computation." Gray elaborates:

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.[12]

In the case of the data marketplace that we were building, users would not need 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 need to provide a download link, because users could work on their data without the need to move it around.[13]

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.

In what language should users write their computation 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. Second, 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.[14] 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 datacenter.

Next, we had to choose an 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 2006 by 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 (Figure 1-4) that could handle the Protocol Buffers (Protobufs) that are ubiquitous throughout Google.

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

Although 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 Protobuf that has hundreds of thousands of columns. If Andrey had used a typical record-oriented store, users would have needed 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 just a few of the thousands of fields in the log Protobufs, 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 toward the root.

By 2010, Google was scanning petabytes of data per 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 itself “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 (Atomicity, Consistency, Isolation, Durability) 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-premises data warehouses like Teradata or Vertica as well as from cloud data warehouses like Redshift and Microsoft Azure Data Warehouse. BigQuery is the first data warehouse to be a scale-out solution, so the only limit on speed and scale is 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 reside together on the same physical hardware. This colocation means that in order to add more storage, you might 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 need to figure out what data to throw out.

When you separate compute from storage as BigQuery does, it means that you never need to throw out data, unless you no longer want it. This might not sound like a big deal, but having access to full-fidelity data is immensely powerful. You might decide you want to calculate something in a different way, so you can go back to the raw data to requery it. You would not be able to do this if you had discarded the source data due to space constraints. You might decide that you want to dig into why some aggregate value exhibits strange behavior. You couldn’t do this if you had 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 cover slots in detail in Chapter 6). 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 multitenant 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 run only 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 filesystem. Most competitor systems need to cache data within compute nodes to get good performance. BigQuery, on the other hand, relies on two systems unique to Google, the Colossus File System and Jupiter networking, 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 for which a much larger logical switch would otherwise be needed. This topology of switches, along with a centralized software stack and custom hardware and software, allows one petabit of bisection bandwidth within a datacenter. That is equivalent to 100,000 servers communicating at 10 Gb/sec, and it means that BigQuery can work without the need to colocate 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 were in the same rack.

The fast networking fabric comes in handy in two ways: to read in data from a disk, and to shuffle between query stages. As discussed earlier, the separation of compute and storage in BigQuery enables any machine within the datacenter 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 6, 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 datacenters are connected through a backbone network called B4 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. 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 filesystem 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.[15] This means that the effective throughput is tens 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. Even though 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 difficult 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 become 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 of the storage, users don't need 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. You can run a query that updates or deletes rows in a table and leave it to BigQuery to determine the best way to modify the storage to reflect this information. BigQuery operations are ACID; that is, all queries will 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 query 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 (we cover these features in detail in Chapter 7) 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 because 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 more difficult 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 it also provides extra features, like projection, filtering, and dynamic rebalancing.

Integration with Google Cloud Platform

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 (GCP) 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 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 NoSQL database), Spanner (a distributed database), or Google Drive (which offers spreadsheets).
- Google Cloud Data Loss Prevention API helps you to manage sensitive data and provides the capability to redact or mask Personally Identifiable Information (PII) 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, and more in free-form text (such as those of customer reviews) held in some table column.
- AutoML Tables and AutoML Text can create high-performing custom machine learning models from data held in BigQuery tables.
- Cloud Catalog provides the ability to discover data held across your organization.
- You can use Cloud Pub/Sub to ingest streaming data and Cloud Dataflow to transform and load it into BigQuery. You can use Cloud Dataflow to carry out streaming queries as well. You can, of course, interactively query the streaming data within BigQuery itself.[16]

- 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 AI Platform provides the ability to train sophisticated machine learning programs from data held in BigQuery.
- Cloud Scheduler and Cloud Functions allow for scheduling or triggering of BigQuery queries as part of larger workflows.
- 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-premises 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, a data lake (queries over files), or a source of BI. The rest of this book paints 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 GCP, so it is possible to take a holistic view of your data security. BigQuery uses Google's 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 need 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. As of this writing, Google Cloud has more than two dozen datacenters 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.

Summary

BigQuery is a highly scalable data warehouse that provides fast SQL analytics over large datasets in a serverless way. Although users appreciate the scale and speed of BigQuery, company executives often appreciate the transformational benefits that come from being able to do ad hoc querying in a serverless way, opening up data-driven decision making to all parts of the company.

To ingest data into BigQuery, you can use an EL pipeline (commonly used for periodic loads of log files), an ETL pipeline (commonly used when data needs to be enriched or quality controlled), or an ELT pipeline (commonly used for exploratory work).

BigQuery is designed for data analytics (OLAP) workloads and provides full-featured support for SQL:2011. BigQuery can achieve its scale and speed because it is built on innovative engineering ideas such as the use of columnar storage, support for nested and repeated fields, and separation of compute and storage, about which Google went on to publish papers. BigQuery is part of the GCP ecosystem of big data analytics tools and integrates tightly with both the infrastructure pieces (such as security, monitoring, and logging) and the data processing and machine learning pieces (such as streaming, Cloud DLP, and AutoML) of the platform.

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

2 In this book, we 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.

3 Jeffrey Dean and Sanjay Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” OSDI ’04: Sixth Symposium on Operating Systems Design and Implementation, San Francisco, CA (2004), pp. 137–150. Available at *https://research.google.com/archive/mapreduce-osdi04.pdf*.

4 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 datacenters, 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.

5 For your copy and pasting convenience, you can find all of the code and query snippets in this book (including the query in the example) in the GitHub repository for this book.

6 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.

7 This code can be downloaded from the book’s GitHub repository.

8 Keep in mind that both authors live in Seattle, where it rains 150 days each year.

9 You can find more details on the columnar storage format in “How BigQuery Came About”.

10 For example, to compute conversion metrics based on the distance that a customer would need to travel to purchase a product.

11 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, as these are subject to change.

12 Jim Gray on eScience: A Transformed Scientific Method", from *The Fourth Paradigm: Data-Intensive Scientific Discovery*, ed. Tony Hey, Stewart Tansley, and Kristin Tolle (Microsoft, 2009), xiv. Available at *https://oreil.ly/M6zMN*.

13 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.

14 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.

15 To read more about Colossus, see *http://www.pdsw.org/pdsw-discs17/slides/PDSW-DISCS-Google-Keynote.pdf* and *https://www.wired.com/2012/07/google-colossus/*.

16 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.