Part

I

BigQuery Fundamentals

In This Part

CHAPTER 1

The Story of Big Data at Google

Since its founding in 1998, Google has grown by multiple orders of magnitude in several different dimensions—how many queries it handles, the size of the search index, the amount of user data it stores, the number of services it provides, and the number of users who rely on those services. From a hardware perspective, the Google Search engine has gone from a server sitting under a desk in a lab at Stanford to hundreds of thousands of servers located in dozens of datacenters around the world.

The traditional approach to scaling (outside of Google) has been to scale the hardware up as the demands on it grow. Instead of running your database on a small blade server, run it on a Big Iron machine with 64 processors and a terabyte of RAM. Instead of relying on inexpensive disks, the traditional scaling path moves critical data to costly network-attached storage (NAS).

There are some problems with the scale-up approach, however:

- Scaled-up machines are expensive. If you need one that has twice the processing power, it might cost you five times as much.
- Scaled-up machines are single points of failure. You might need to get more than one expensive server in case of a catastrophic problem, and each one usually ends up being built with so many backup and redundant pieces that you're paying for a lot more hardware than you actually need.

- Scale up has limits. At some point, you lose the ability to add more processors or RAM; you've bought the most expensive and fastest machine that is made (or that you can afford), and it still might not be fast enough.
- Scale up doesn't protect you against software failures. If you have a Big Iron server that has a kernel bug, that machine will crash just as easily (and as hard) as your Windows laptop.

Google, from an early point in time, rejected scale-up architectures. It didn't, however, do this because it saw the limitations more clearly or because it was smarter than everyone else. It rejected scale-up because it was trying to save money. If the hardware vendor quotes you $1 million for the server you need, you could buy 200 $5,000 machines instead. Google engineers thought, "Surely there is a way we could put those 200 servers to work so that the next time we need to increase the size, we just need to buy a few more cheap machines, rather than upgrade to the $5 million server." Their solution was to scale out, rather than scale up.

Big Data Stack 1.0

Between 2000 and 2004, armed with a few principles, Google laid the foundation for its Big Data strategy:

- Anything can fail, at any time, so write your software expecting unreliable hardware. At most companies, when a database server crashes, it is a serious event. If a network switch dies, it will probably cause downtime. By running in an environment in which individual components fail often, you paradoxically end up with a much more stable system because your software is designed to handle those failures. You can quantify your risk beyond blindly quoting statistics, such as mean time between failures (MTBFs) or service-level agreements (SLAs).
- Use only commodity, off-the-shelf components. This has a number of advantages: You don't get locked into a particular vendor's feature set; you can always find replacements; and you don't experience big price discontinuities when you upgrade to the "bigger" version.
- The cost for twice the amount of capacity should not be considerably more than the cost for twice the amount of hardware. This means the software must be built to scale out, rather than up. However, this also imposes limits on the types of operations that you can do. For instance, if you scale out your database, it may be difficult to do a `JOIN` operation, since you'd need to join data together that lives on different machines.

- "A foolish consistency is the hobgoblin of little minds." If you abandon the "C" (consistency) in ACID database operations, it becomes much easier to parallelize operations. This has a cost, however; loss of consistency means that programmers have to handle cases in which reading data they just wrote might return a stale (inconsistent) copy. This means you need smart programmers.

These principles, along with a cost-saving necessity, inspired new computation architectures. Over a short period of time, Google produced three technologies that inspired the Big Data revolution:

- **Google File System (GFS):** A distributed, cluster-based filesystem. GFS assumes that any disk can fail, so data is stored in multiple locations, which means that data is still available even when a disk that it was stored on crashes.
- **MapReduce:** A computing paradigm that divides problems into easily parallelizable pieces and orchestrates running them across a cluster of machines.
- **Bigtable:** A forerunner of the NoSQL database, Bigtable enables structured storage to scale out to multiple servers. Bigtable is also replicated, so failure of any particular tablet server doesn't cause data loss.

What's more, Google published papers on these technologies, which enabled others to emulate them outside of Google. Doug Cutting and other open source contributors integrated the concepts into a tool called Hadoop. Although Hadoop is considered to be primarily a MapReduce implementation, it also incorporates GFS and BigTable clones, which are called HDFS and HBase, respectively.

Armed with these three technologies, Google replaced nearly all the off-the-shelf software usually used to run a business. It didn't need (with a couple of exceptions) a traditional SQL database; it didn't need an e-mail server because its Gmail service was built on top of these technologies.

Big Data Stack 2.0 (and Beyond)

The three technologies—GFS, MapReduce, and Bigtable—made it possible for Google to scale out its infrastructure. However, they didn't make it easy. Over the next few years, a number of problems emerged:

- MapReduce is hard. It can be difficult to set up and difficult to decompose your problem into Map and Reduce phases. If you need multiple MapReduce rounds (which is common for many real-world problems),

you face the issue of how to deal with state in between phases and how to deal with partial failures without having to restart the whole thing.

- MapReduce can be slow. If you want to ask questions of your data, you have to wait minutes or hours to get the answers. Moreover, you have to write custom C++ or Java code each time you want to change the question that you're asking.
- GFS, while improving durability of the data (since it is replicated multiple times) can suffer from reduced availability, since the metadata server is a single point of failure.
- Bigtable has problems in a multidatacenter environment. Most services run in multiple locations; Bigtable replication between datacenters is only eventually consistent (meaning that data that gets written out will show up everywhere, but not immediately). Individual services spend a lot of redundant effort babysitting the replication process.
- Programmers (even Google programmers) have a really difficult time dealing with eventual consistency. This same problem occurred when Intel engineers tried improving CPU performance by relaxing the memory model to be eventually consistent; it caused lots of subtle bugs because the hardware stopped working the way people's mental model of it operated.

Over the next several years, Google built a number of additional infrastructure components that refined the ideas from the 1.0 stack:

- **Colossus:** A distributed filesystem that works around many of the limitations in GFS. Unlike many of the other technologies used at Google, Colossus' architecture hasn't been publicly disclosed in research papers.
- **Megastore:** A geographically replicated, consistent NoSQL-type datastore. Megastore uses the Paxos algorithm to ensure consistent reads and writes. This means that if you write data in one datacenter, it is immediately available in all other datacenters.
- **Spanner:** A globally replicated datastore that can handle data locality constraints, like "This data is allowed to reside only in European datacenters." Spanner managed to solve the problem of global time ordering in a geographically distributed system by using atomic clocks to guarantee synchronization to within a known bound.
- **FlumeJava:** A system that allows you to write idiomatic Java code that runs over collections of Big Data. Flume operations get compiled and optimized to run as a series of MapReduce operations. This solves the

ease of setup, ease of writing, and ease of handling multiple MapReduce problems previously mentioned.

- **Dremel:** A distributed SQL query engine that can perform complex queries over data stored on Colossus, GFS, or elsewhere.

The version 2.0 stack, built piecemeal on top of the version 1.0 stack (Megastore is built on top of Bigtable, for instance), addresses many of the drawbacks of the previous version. For instance, Megastore allows services to write from any datacenter and know that other readers will read the most up-to-date version. Spanner, in many ways, is a successor to Megastore, which adds automatic planet-scale replication and data provenance protection.

On the data processing side, batch processing and interactive analyses were separated into two tools based on usage models: Flume and Dremel. Flume enables users to easily chain together MapReduces and provides a simpler programming model to perform batch operations over Big Data. Dremel, on the other hand, makes it easy to ask questions about Big Data because you can now run a SQL query over terabytes of data and get results back in a few seconds. Dremel is the query engine that powers BigQuery; Its architecture is discussed in detail in Chapter 9, "Understanding Query Execution."

An interesting consequence of the version 2.0 stack is that it explicitly rejects the notion that in order to use Big Data you need to solve your problems in fundamentally different ways than you're used to. While MapReduce required you to think about your computation in terms of Map and Reduce phases, FlumeJava allows you to write code that looks like you are operating over normal Java collections. Bigtable replication required abandoning consistent writes, but Megastore adds a consistent coordination layer on top. And while Bigtable had improved scalability by disallowing queries, Dremel retrofits a traditional SQL query interface onto Big Data structured storage.

There are still rough edges around many of the Big Data 2.0 technologies: things that you expect to be able to do but can't, things that are slow but seem like they should be fast, and cases where they hold onto awkward abstractions. However, as time goes on, the trend seems to be towards smoothing those rough edges and making operation over Big Data as seamless as over smaller data.

Open Source Stack

Many of the technologies at Google have been publicly described in research papers, which were picked up by the Open Source community and re-implemented as open source versions. When the open source Big Data options were in their

infancy, they more or less followed Google's lead. Hadoop was designed to be very similar to the architecture described in the MapReduce paper, and the Hadoop subprojects HDFS and HBase are close to GFS and BigTable.

However, as the value of scale-out systems began to increase (and as problems with traditional scale-up solutions became more apparent), the Open Source Big Data stack diverged significantly. A lot of effort has been put into making Hadoop faster; people use technologies such as Hive and Pig to query their data; and numerous NoSQL datastores have sprung up, such as CouchDB, MongoDB, Cassandra, and others.

On the interactive query front, there are a number of open source options:

- Cloudera's Impala is an open source parallel execution engine similar to Dremel. It allows you to query data inside HDFS and Hive without extracting it.
- Amazon.com's Redshift is a fork of PostgreSQL which has been modified to scale out across multiple machines. Unlike Impala, Redshift is a hosted service, so it is managed in the cloud by Amazon.com.
- Drill is an Apache incubator project that aims to be for Dremel what Hadoop was for MapReduce; Drill fills in the gaps of the Dremel paper to provide a similar open source version.
- Facebook's Presto is a distributed SQL query engine that is similar to Impala.

The days when Google held the clear advantage in innovation in the Big Data space are over. Now, we're in an exciting time of robust competition among different Big Data tools, technologies, and abstractions.

Google Cloud Platform

Google has released many of its internal infrastructure components to the public under the aegis of the Google Cloud Platform. Google's public cloud consists of a number of components, providing a complete Big Data ecosystem. It is likely that in the coming months and years there will be additional entries, so just because a tool or service isn't mentioned here doesn't mean that it doesn't exist. Chapter 2, "BigQuery Fundamentals," goes into more detail about the individual components, but this is a quick survey of the offerings. You can divide the cloud offerings into three portions: processing, storage, and analytics.

Cloud Processing

The cloud processing components enable you to run arbitrary computations over your data:

- **Google Compute Engine (GCE):** The base of Google's Cloud Platform, GCE is infrastructure-as-a-service, plain and simple. If you have software you just want to run in the cloud on a Linux virtual machine, GCE enables you to do so. GCE also can do live migration of your service so that when the datacenter it is running is turned down for maintenance, your service won't notice a hiccup.
- **AppEngine:** AppEngine is a higher-level service than GCE. You don't need to worry about OS images or networking configurations. You just write the code you actually want running in your service and deploy it; AppEngine handles the rest.

Cloud Storage

These cloud storage components enable you to store your own data in Google's cloud:

- **Google Cloud Storage (GCS):** GCS enables you to store arbitrary data in the cloud. It has two APIs: one that is compatible with Amazon.com's S3 and another REST API that is similar to other Google APIs.
- **DataStore:** A NoSQL key-value store. DataStore is usually used from AppEngine, but its REST API enables you to store and look up data from anywhere.
- **BigQuery (Storage API):** BigQuery enables you to store structured rows and columns of data. You can ingest data directly through the REST API, or you can import data from GCS.

Cloud Analytics

Google's cloud analytics services enable you to extract meaning from your data:

- **Cloud SQL:** A hosted MySQL instance in the cloud
- **Prediction API:** Enables you to train machine learning models and apply them to your data

- **Cloud Hadoop:** Packages Hadoop and makes it easy to run on Google Compute Engine
- **BigQuery:** Enables you to run SQL statements over your structured data

If you find that something is missing from Google's Cloud Platform, you always have the option of running your favorite open source software stack on Google Compute Engine. For example, the Google Cloud Hadoop package is one way of running Hadoop, but if you want to run a different version of Hadoop than is supported, you can always run Hadoop directly; Google's Hadoop package uses only publicly available interfaces.

Problem Statement

Before we go on to talk about BigQuery, here's a bit of background information about the problems that BigQuery was developed to solve.

What Is Big Data?

There are a lot of different definitions from experts about what it means to have Big Data; many of these definitions conceal a boast like, "Only a petabyte? I've forgotten how to count that low!" This book uses the term Big Data to mean more data than you can process sequentially in the amount of time you're willing to spend waiting for it. Put another way, Big Data just means more data than you can easily handle using traditional tools such as relational databases without spending a lot of money on specialized hardware.

This definition is deliberately fuzzy; to put some numbers behind it, we'll say a hundred million rows of structured data or a hundred gigabytes of unstructured data. You can fit data of that size on a commodity disk and even use MySQL on it. However, dealing with data that size isn't going to be pleasant. If you need to write a tool to clean the data, you're going to spend hours running it, and you need be careful about memory usage, and so on. And as the data size gets bigger, the amount of pain you'll experience doing simple things such as backing it up or changing the schema will get exponentially worse.

Why Big Data?

Many people are surprised at how easy it is to acquire Big Data; they assume that you need to be a giant company like Wal-Mart or IBM for Big Data to be relevant. However, Big Data is easy to accumulate. Following are some of the ways to get Big Data without being a Fortune 500 company:

- **Over time:** If you produce a million records a day, that might not be "Big Data." But in 3 years, you'll have a billion records; at some point you may find that you either need to throw out old data or figure out a new way to process the data that you have.
- **Viral scaling:** On the Internet, no one knows you're a small company. If your website becomes popular, you can get a million users overnight. If you track 10 actions from a million users a day, you're talking about a billion actions a quarter. Can you mine that data well enough to be able to improve your service and get to the 10 million user mark?
- **Projected growth:** Okay, maybe you have only small data now, but after you sign customer X, you'll instantly end up increasing by another 2 orders of magnitude. You need to plan for that growth now to make sure you can handle it.
- **Architectural limitations:** If you need to do intense computation over your data, the threshold for "Big Data" can get smaller. For example, if you need to run an unsupervised clustering algorithm over your data, you may find that even a few million data points become difficult to handle without sampling.

Why Do You Need New Ways to Process Big Data?

A typical hard disk can read on the order of 100 MB per second. If you want to ask questions of your data and your data is in the terabyte range, you either need thousands of disks or you are going to spend a lot of time waiting.

As anyone who has spent time tuning a relational database can attest, there is a lot of black magic involved in getting queries to run quickly on your-favorite-database. You may need to add indexes, stripe data across disks, put the transaction log on its own spindle, and so on. However, as your data grows, at some point it gets harder and harder to make your queries perform well. In addition, the more work you do, the more you end up specializing the schema for the type of questions you typically ask of your data.

What if you want to ask a question you've never asked before? If you are relying on a heavily tuned schema, or if you're running different queries than the database was tuned for, you may not get answers in a reasonable amount of time or without bogging down your production database. In these cases, your options are limited; you either need to run an extremely slow query (that may degrade performance for your entire database), or you could export the data and process it in an external system like Hadoop.

Often, to get queries to run quickly, people sample their data—they keep only 10 percent of user impressions, for example. But what happens if you want

to explore the data in a way that requires access to all the impressions? Maybe you want to compute the number of distinct users that visited your site—if you drop 90 percent of your data, you can't just multiply the remaining users by 10 to get the number of distinct users in the original dataset. This point is somewhat subtle, but if you drop 90 percent of your data, you might still have records representing 99 percent of your users, or you might have records representing only 5 percent of your users; you can't tell unless you use a more sophisticated way to filter your data.

How Can You Read a Terabyte in a Second?

If you want to ask interactive questions of your Big Data, you must process all your data within a few seconds. That means you need to read hundreds of gigabytes per second—and ideally more.

Following are three ways that you can achieve this type of data rate:

1. Skip a lot of the data. This is a good option if you know in advance the types of questions you're going to ask. You can pre-aggregate the data or create indexes on the columns that you need to access. However, if you want to ask different questions, or ask them in a different way, you may not be able to avoid reading everything.
2. Buy some *really* expensive hardware. For a few million dollars or so, you can get a machine onsite that will come with its own dedicated support person that can let you query over your terabytes of data.
3. Run in parallel. Instead of reading from one disk, read from thousands of disks. Instead of one database server, read from hundreds.

If you use custom hardware (solution #2) and you want it to go faster, you need to buy an even bigger data warehouse server (and hope you can sell the old one). And if you rely on skipping data (solution #1) to give you performance, the only way to go faster is to be smarter about what data you skip (which doesn't scale).

BigQuery, and most Big Data tools, take approach #3. Although it may sound expensive to have thousands of disks and servers, the advantage is that you get exactly what you pay for; that is, if you need to run twice as fast, you can buy twice as many disks. If you use BigQuery, you don't need to buy your own disks; you get a chance to buy small slices of time on a massive amount of hardware.

What about MapReduce?

A large proportion of the Big Data hype has been directed toward MapReduce and Hadoop, its Open Source incarnation. Hadoop is a fantastic tool that enables you to break up your analysis problem into pieces that run in parallel. The Hadoop File System (HDFS) can enable you to read in parallel from a lot of disks,

which allows you to perform operations over Big Data orders of magnitude more quickly than if you had to read that data sequentially.

However, Hadoop specifically and MapReduce in general have some architectural drawbacks that make them unsuited for interactive-style analyses. That is, if you want to ask questions of your data using MapReduce, you're probably going to want to get a cup of coffee (or go out to lunch) while you wait. Interactive analyses should give you answers before you get bored or forget why you were asking in the first place. Newer systems, such as Cloudera's Impala, allow interactive queries over your Hadoop data, but they do so by abandoning the MapReduce paradigm. Chapter 9 discusses the architecture in more detail and shows why MapReduce is better suited to batch workloads than interactive ones.

How Can You Ask Questions of Your Big Data and Quickly Get Answers?

Google BigQuery is a tool that enables you to run SQL queries over your Big Data. It fans out query requests to thousands of servers, reads from tens or hundreds of thousands of disks at once, and can return answers to complex questions within seconds. This book describes how BigQuery can achieve such good performance and how you can use it to run queries on your own data.

Summary

This chapter briefly documented the history of Google's Big Data systems and provided a survey of scale-out technologies, both at Google and elsewhere. It set the stage for BigQuery by describing an unfulfilled Big Data analytics niche. This chapter deliberately didn't mention BigQuery very much, however; Chapter 2 should answer all your questions about what BigQuery is and what it can do.