



# Google Cloud Platform IN ACTION

JJ Geewax

MANNING



**MEAP Edition  
Manning Early Access Program  
Google Cloud Platform in Action  
Version 12**

Copyright 2018 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/google-cloud-platform-in-action>

**Licensed to Asif Qamar <[asif@asifqamar.com](mailto:asif@asifqamar.com)>**

# welcome

---

Thanks for buying the MEAP for *Google Cloud Platform in Action!* I'm really excited to be working on this book, so I hope that you get as much out of reading it as I get from writing it.

As we all know, technology moves incredibly quickly. It seems like just the other day I was on the phone with someone to set-up my first VPS, and now a new virtual machine is just a click away.

But this rapid rate of change means that most experts aren't really experts anymore, and that means there are plenty of mistakes to be made as we explore our way through this new world of "the cloud". This is exactly the motivation for this book.

First and foremost, the primary goal of Google Cloud Platform in Action is to help you navigate all of these new things. My hope is that if you understand the tools in your toolbox you'll be able to combine your existing expertise with all of these new cloud services, and build some amazing things.

Next, I try to move from the general concept of cloud services into the specific services offered by Google Cloud Platform -- as I've been using and working on these at Google for the past few years. My hope is that by better understanding the internals of these systems, you'll also pick up some of the best practices for using them (and avoid the pitfalls).

To help focus your journey through these topics I've split the book into a few different parts, starting with the basics of what cloud actually is, then going through the core of computing and storage, and finishing up with machine learning, big data analysis, and networking. As you might expect, the target audience of this book has some background in infrastructure (ie, you're not afraid of things like SSH) and an interest in scalable systems (ie, you might wonder how folks like SnapChat store all of their pictures!), but you don't need to be an expert in technical infrastructure to get something useful out of the book.

I'll aim to have an update for you at least once per month, and I hope to get lots of feedback from you so I can make the book as helpful as possible. Feedback from readers like you really is instrumental, so please don't be shy.

Thanks again, and enjoy!

— JJ Geewax

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/google-cloud-platform-in-action>

**Licensed to Asif Qamar <asif@asifqamar.com>**

# *brief contents*

---

## **PART 1: GETTING STARTED**

- 1 *What is "Cloud"?*
- 2 *Trying it out: Deploying WordPress on Google Cloud*
- 3 *The cloud data center*

## **PART 2: STORAGE**

- 4 *Cloud SQL: Managed relational storage*
- 5 *Cloud Datastore: Document storage*
- 6 *Cloud Spanner: Large scale SQL*
- 7 *Cloud Bigtable: Large scale structured data*
- 8 *Cloud Storage: Object storage*

## **PART 3: COMPUTING**

- 9 *Compute Engine: Virtual machines*
- 10 *Kubernetes Engine: Managed Kubernetes Clusters*
- 11 *App Engine: Fully managed applications*
- 12 *Cloud Functions: Serverless applications*
- 13 *Cloud DNS: Managed DNS hosting*

## **PART 4: MACHINE LEARNING**

- 14 *Cloud Vision: Image recognition*
- 15 *Cloud Natural Language: Text analysis*
- 16 *Cloud Speech: Audio to text conversion*
- 17 *Cloud Translation: Multi-language machine translation*
- 18 *Cloud Machine Learning Engine: Managed Machine Learning*

## **PART 5: DATA PROCESSING AND ANALYTICS**

- 19 *BigQuery: Highly scalable data warehouse*
- 20 *Cloud Dataflow: Large-scale data processing*
- 21 *Cloud Pub/Sub: Managed event publishing*

# Part 1

*Getting started*

This part of the book will help set the stage for the rest of our exploration of Google Cloud Platform.

In [chapter 1](#) we'll look at what "cloud" actually means and some of the principles that you should expect to bump into when using cloud services. Next, in [chapter 2](#), we'll take Google Cloud Platform for a test drive by setting up our own WordPress instance using Google Compute Engine. Finally, in [chapter 3](#), we'll explore how cloud data centers work and how you should think about "location" in the amorphous world of the cloud.

When you're finished with this part of the book, you'll be ready to dig much deeper into individual products and see how they all fit together to build much bigger things.

# 1

## *What is "Cloud"?*

### **This chapter covers:**

- Overview of "the cloud"
- When and when not to use cloud hosting
- Summary of what to expect from cloud hosting
- Explanation of cloud pricing principles
- What it means to build an application for the cloud
- A walk-through of Google Cloud Platform

### **1.1 What do we mean by "Cloud"?**

The term "Cloud" has been used in many different contexts over the past few years, typically ending with very different definitions, so it makes sense to define the term — at least in the context of this book.

Cloud is a collection of services built for developers that helps them to focus on their project rather than the infrastructure that powers it.

In more concrete terms, cloud services are things like Amazon's Elastic Compute Cloud (EC2) or Google's Compute Engine (GCE) which provide APIs to provision virtual servers, where customers pay per hour for the use of these servers.

In many ways, cloud is simply the next layer of abstraction in computer infrastructure, where computing, storage, analytics, networking, and more are all pushed higher up the computing stack. This takes the focus of the developer away from CPUs and RAM and towards APIs for higher level operations such as storing or querying for data. That is,

cloud services aim to solve your problem, not simply give you low-level tools for you to do so on your own. Further, cloud services are extremely flexible, most requiring no provisioning or long-term contracts. Due to this, relying on these services allows you to scale up and down with no advanced notice or provisioning, and ultimately only paying for the resources you use in a given month.

## **1.2 What is Google Cloud Platform?**

There are many cloud providers out there, including Google, Amazon, Microsoft, Rackspace, Digital Ocean, and more. In order to sustain so many different competitors in the space, each one of these companies must have their own take on how to best serve customers. It turns out that although each one provides many similar products, the implementation and details of how these products work tends to vary quite a bit.

Google Cloud Platform (often abbreviated as GCP) is a collection of products that allows the world to use some of Google's internal infrastructure. This collection includes many things that are common across all cloud providers, such as on-demand virtual machines via Google Compute Engine, or object storage for storing files via Google Cloud Storage, and also includes APIs to some of the more advanced Google-built technology like Bigtable, Datastore, or Kubernetes.

Although Google Cloud Platform is similar to other cloud providers, there are still quite a few key differentiating factors that are worth mentioning. First, Google is "home" to some amazing people which leads to some incredible new technologies created at Google and then shared with the world through research papers such as MapReduce (the paper that spawned Hadoop and changed how we handle "Big Data"), Bigtable (the paper that spawned HBase), and Spanner. With Google Cloud Platform, many of these technologies are no longer "only for Googlers".

Next, the scale at which Google operates means it sees many economic advantages which are passed on in the form of lower prices. Google owning immense physical infrastructure means that they buy and build custom hardware to support it, which means cheaper overall prices and often combined with improved performance. It's sort of like Costco letting you open up that 144-pack of potato chips and pay 1/144th the price for one bag.

## **1.3 Why Cloud?**

So why use cloud in the first place? First, cloud hosting offers a lot of flexibility, which is a great fit for situations where you don't know (or can't know) how much computing power you actually need. This means that you won't have to over-provision to handle situations where you might need a lot of computing power in the morning and almost nothing overnight.

Next, cloud hosting comes with the maintenance "baked in" for quite a few products. This means that cloud hosting has minimal "extra work" to host your systems compared to other options where you might need to manage your own databases, operating systems, and even your own hardware (in the case of a co-located hosting

provider). If you don't want to (or can't) manage these types of things, cloud hosting is a really great choice.

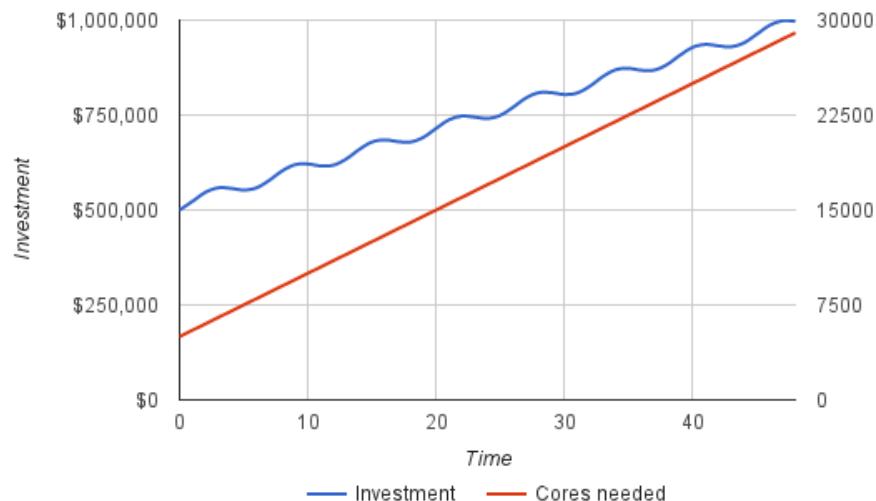
### 1.3.1 Why not Cloud?

Obviously this book is focused on using Google Cloud Platform, so there's an assumption that cloud hosting is a good option for your company, however it seems worthwhile to devote a few words to why you might **not** want to use cloud hosting. And yes, there are times that cloud is not the best choice despite the fact that it's often the cheapest of all the options.

Let's start with an extreme example: Google itself. Google's infrastructural footprint is exabytes of data, hundreds of thousands of CPUs, a relatively stable and growing overall workload. In addition, Google is a big target for attacks (e.g., denial of service attacks), government espionage, and has the budget and expertise to build gigantic infrastructural footprints. All of these things together make Google a bad candidate for cloud hosting.

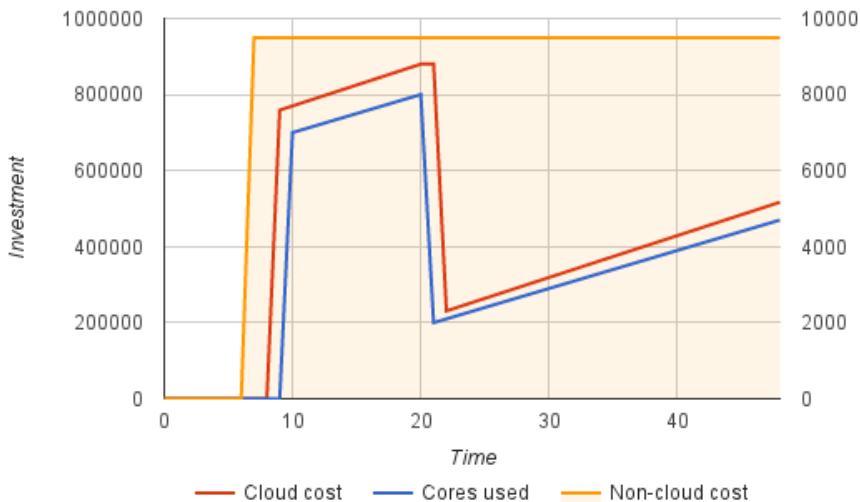
Here's an visual representation of a usage and cost pattern that would be a bad fit for cloud hosting. Notice how the growth of computing needs (the bottom line) steadily increases and the company is provisioning extra capacity regularly to stay ahead of its needs (the top, wavy line).

**Figure 1.1. Steady growth in resource consumption**



Compare this to a more typical company of the internet age, where growth is spiky and unpredictable, and tends to drop without much notice ahead of time. In this case, the company bought enough computing capacity (the top line) to handle a spike which was needed up front, but then when traffic fell (the bottom line), they were stuck with quite a bit of excess capacity.

**Figure 1.2. Unexpected pattern of resource consumption**



In short, if you have the expertise to run your own data centers (including the plans for disasters and other failures, and the recovery from those potential disasters), along with steady growing computing needs (measured in cores, storage, networking consumption, etc), cloud hosting might not be right for you. If you're anything like the typical company of today, where you don't know what you need today (and certainly don't know what you'll need several years from today) and don't really have the expertise in your company to build out huge data centers to achieve the same economies of scale that large cloud providers can offer, cloud hosting is likely to be a very good fit for you, offering the best bang for the buck available.

## 1.4 What to expect from cloud services

All of the discussion so far has been about cloud in the broader sense. Let's take a moment to look at some of the more specific things that you should expect from cloud services, particularly how cloud specifically differs from other hosting options.

### 1.4.1 Computing

You've already learned a little bit about how cloud computing is fundamentally different from virtual private, co-located, or on-premises hosting, primarily driven by the idea that you can hold much shorter leases on the physical those resources, and the ability to hold leases on virtual subdivisions of those resources, but this doesn't really give you a practical picture of what you might expect to see when running your application in the cloud. So let's take a look at what you can actually expect if you decide to take the plunge into the world of cloud computing.

The first thing you'll notice is that provisioning your machine will be fast. Compared to co-located or on-premises hosting, it should be significantly faster. In real terms, the typical expected time from "clicking the button" to "SSH'ing to the machine" will be

about a minute. If you're used to virtual private hosting, the provisioning time might be around the same, maybe slightly faster.

What's more interesting is what is missing in the process of turning on a cloud-hosted virtual machine. If you turn on a VM right now, you might notice that there's no mention of payment. Compare that to your typical VPS where you agree on a set price, and purchase the VPS for a full year, making monthly payments (with your first payment immediately, and maybe a discount for up-front payment). Google doesn't mention payment at this time for a very simple reason: they don't know how long you'll keep that machine running, so there's no way to know how much to charge you. In other words, they can only determine how much you owe either at the end of the month or when you turn off the VM.

**Table 1.1. Hosting choice comparison**

Hosting choice	Best if...	Kind of like...
Building your own data center	You have steady long-term needs at a very large scale	Purchasing a car
Using your own hardware in a co-location facility	You have steady long-term needs at a smaller scale	Leasing a car
Using virtual-private hosting	You have slowly changing needs	Renting a car
Using cloud hosting	You have rapidly changing (or unknown) needs	Taking an Uber

This type of pricing mechanism might correlate with transportation. On-premises hosting is sort of like buying your own car, co-located hosting is like leasing a car for a year, virtual-private hosting is like renting a car for a month, and cloud hosting is like taking an Uber where you need it.

## 1.4.2 Storage

Storage, although not the most glamorous part of computing, is incredibly necessary. Imagine if you weren't able to save your data when you were done working on it? Cloud's take on storage follows the same pattern you've seen so far with computing, abstracting away the management of your physical resources. This might seem unimpressive but the truth is that storing data is actually a complicated thing to do. For example, do you want your data to be edge-cached to speed up downloads for users on the internet? Are you optimizing for throughput or latency? Is it OK if the "time to first byte" is a few seconds? How available do you need the data to be? How many concurrent readers do you need to support?

The answers to these questions change what you build in pretty significant ways, so much so that you might end up building entirely different products if you were the one building a storage service. Ultimately, the abstraction provided by a storage service is giving you much more than simply "a thing that looks like a disk". It's providing the ability to configure your storage mechanisms for various levels of performance, durability, availability, and cost.

However, these systems come with a few trade-offs. First, the "failure" aspects of

storing data typically disappear. That is, you shouldn't ever get a notification or a phone call from someone saying that a hard drive failed and your data was lost. Next, with reduced-availability options, you might occasionally try to download your data and get an error saying to try again later, however you'll be paying much less for storage of that class than any other. Finally, for virtual disks in the cloud, you'll notice that you have lots of choices about how you want to store the data, both in capacity (measured in GB) and in performance (typically measured in IOPS). Once again, like computing in the cloud, storing data on virtual disks in the cloud will feel very familiar.

On the other hand, some of the custom database services, like Cloud Datastore, might feel a bit foreign. This is because these systems are in many ways completely unique to cloud hosting, relying on huge, shared, highly scalable systems built by and for Google. For example, Cloud Datastore is an adapted externalization of an internal storage system called Megastore, which until recently was the underlying storage system for many Google products including Gmail. These hosted storage systems sometimes come with a sense of "lock-in" given the need to integrate your own code with a proprietary API. This means that it'll become all the more important to keep a proper layer of abstraction between your code base and the storage layer. Even with the "lock-in", it still may make sense to rely on these hosted systems particularly due to the simple fact that all of the scaling is handled automatically.

### **1.4.3 Analytics (aka, Big Data)**

Analytics, although not something typically considered "infrastructure" is a quickly growing area of hosting—though you might often see this area called "Big Data". The trend so far seems to be that most companies are logging and storing almost everything, meaning the amount of data they have to analyze and use to draw new and interesting conclusions is growing faster and faster every day. This also means that new and interesting open-source projects are popping up to help make these enormous amounts of data more manageable, such as Apache Spark, HBase, and Hadoop.

As you might guess, many of these things are in use at the large companies that also offer cloud hosting, but what should you expect to see from cloud in the analytics and big data areas?

### **1.4.4 Networking**

Having lots of different pieces of infrastructure running is great, but without a way for those pieces to talk to each other your system isn't really a single system, it's more of a pile of lots of isolated systems. That's not really a big help to anyone. Traditionally, we tend to take networking for granted as just "something that should work". For example, when you sign up for virtual-private hosting and get access to your server, you tend to expect that it has a connection to the internet, and that it will be "fast enough".

In the world of cloud computing some of these assumptions remain unchanged, such as the assumption that your servers will have an internet connection, and that it will be "fast enough". The interesting parts come up when you start developing the need for

more advanced features, such as "faster than normal" network connections, advanced firewalling abilities (where you allow only certain IPs to talk to certain ports), load balancing (where requests come in and can be handled by any one of many machines), and SSL certificate management (where you want requests to be encrypted, but don't want to manage the certificates for each individual virtual machine).

In short, since networking on traditional hosting is typically hidden, most people won't really notice any differences for the simple reason that there's usually nothing to notice. For those of you who do have a deep background in networking, most of the things you can do with your typical computing stack (such as configure VPNs, set up firewalls with `iptables`, and balance requests across servers using HAProxy) are all still possible. Google Cloud's networking features will only act to simplify the common-cases, where instead of running a separate VM with HAProxy, you can rely on Google's Cloud Load Balancer to route requests.

#### **1.4.5 Pricing**

In the technology industry, it's been commonplace to find a single set of metrics, and latch onto those as the only factors in a decision-making process. While many times that actually turns out to be a good heuristic in making the decision, it turns out to take you further away from the market when estimating the total cost of infrastructure and comparing against the market price of the physical goods. In other words, comparing only the dollar cost of "buying the hardware" from a vendor versus a cloud hosting provider is going to favor the vendor, but it's not an apples-to-apples comparison. So how do we make everything into apples?

When trying to compare costs of hosting infrastructure, one great metric to use is called TCO or "total cost of ownership". This metric factors in not only the cost of purchasing the physical hardware, but also ancillary costs such as human labor (like hardware administrators or security guards), utility costs (electricity or cooling), and one of the most important pieces: support and on-call staff who make sure that any software services running stay that way—at all hours of the night. Finally, TCO also includes the cost of building redundancy for your systems so that (for example) data is never lost due to a failure of a single hard drive. This cost is actually more than just the cost of the extra drive, as you need to not only configure your system, but have the necessary knowledge to design the system for this configuration. In short, TCO is everything you actually pay for when buying hosting.

If you think more deeply about the situation, TCO for hosting will be very close to the cost of goods sold for a virtual-private hosting company. With cloud hosting providers, TCO is going to be much closer to what you pay, however due to the sheer scale of these cloud providers, and the need to build these tools and hire the ancillary labor anyway, they're able to reduce the TCO below traditional rates (and every reduction in TCO for a hosting company introduces more room for a larger profit margin).

### **1.5 Building an application for the cloud**

So far this chapter has been mainly discussion on what Cloud is and what it means for

developers looking to rely on it rather than traditional hosting options. Let's switch gears now and demonstrate how to actually deploy something meaningful using Google Cloud Platform.

### **1.5.1 What is a cloud application?**

In many ways, an application built "for the cloud" is like any other. The primary difference is in the assumptions made about the application's architecture. For example, in a traditional application, we tend to deploy things as binaries running on particular servers (e.g., running a MySQL database on one server and Apache with `mod_php` on another). Rather than thinking in terms of which servers handle which things, a typical cloud application will rely on hosted or managed services whenever possible, and in many cases rely on containers the way a traditional application would rely on servers. That is, instead of saying "this is the database server", you'd say "this is the database container, however I don't care which server it runs on". By operating this way, a cloud application will often be much more flexible, able to grow and shrink depending on the customer demand throughout the day.

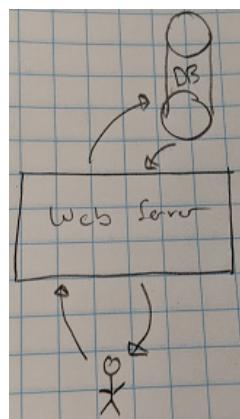
Let's take a moment to look at an example of a "cloud application" and how it might differ from the more traditional applications that you might already be familiar with.

### **1.5.2 Example: Serving photos**

If you've ever built a toy project that allows users to upload their photos (e.g., a Facebook clone that stores a profile photo), you're probably familiar with dealing with uploaded data and storing it. When you first started, you probably made the age old mistake of adding a `BINARY` or `VARBINARY` column to your database, calling it `profile_photo`, and shoving any uploaded data into that column.

If that's a bit too technical, try thinking about it from an architectural standpoint. That is, the old way of doing this was to store the image data in your relational database, and then whenever someone wanted to see the profile photo, you'd retrieve it from the database, and return it through your web server.

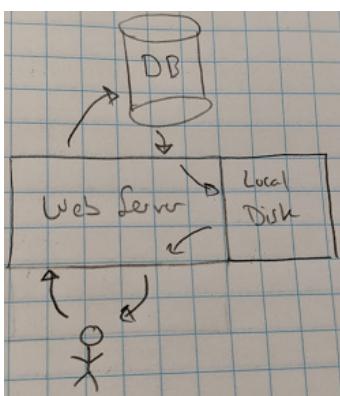
**Figure 1.3. Serving photos dynamically through your web server**



In case it wasn't clear, this is bad for a variety of reasons. First, storing binary data in your database is inefficient. There are reasons to store binary data in your database, but the best reason is for transactional support, which profile photos probably don't need. Secondly, and most importantly, by storing the binary data of a photo in your database, you're putting extra load on the database itself, but not using it for the things it's good at (like joining relational data together).

In short, if you don't need transactional semantics on your photo (which here, we don't), it makes more sense to put the photo somewhere on a disk, and then use the static serving capabilities of your web server to deliver those bytes. This leaves the database out of this completely, so it's free to do more important work.

**Figure 1.4. Serving photos statically through your web server**



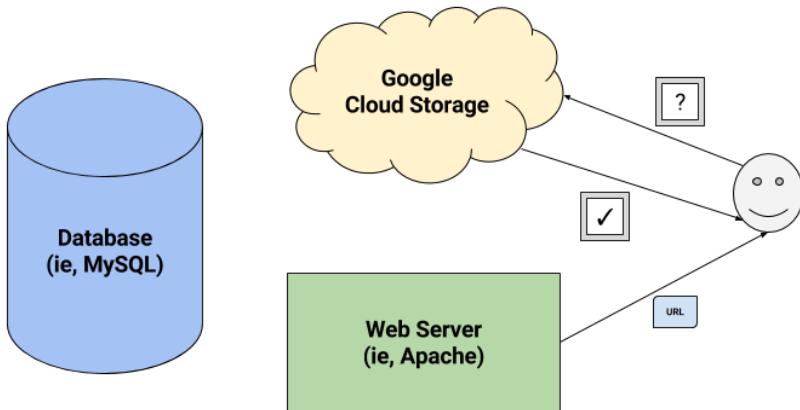
This is a huge improvement, and will probably perform quite well for most use cases, however you might be thinking that it really doesn't illustrate anything special about "the cloud", so let's take it a step further and consider geography for a moment. In your current deployment, you have a single web server living somewhere inside a data center, serving a photo it has stored locally on its disk. For simplicity, let's assume this server lives somewhere in the central United States. This means that if someone nearby (e.g., in New York) requests that photo, they'll get a relatively zippy response. But what if someone far away in Japan requests the photo? The only way to get it is to send a request from Japan to the US, and then the server needs to ship all the bytes from the US back to Japan.

This could be on the order of hundreds of milliseconds, which might not seem like a lot, but imagine you start requesting lots of photos on a single page. Those hundreds of milliseconds tend to start adding up. What can you do about this? Most of you might already know the answer is "edge caching" or relying on a "content distribution network". The idea of these services is that you give them copies of your data (in this case, the photos), and they store those copies in lots of different geographical locations. Then, instead of sending a URL to the image on your single server, you send a URL pointing to this content distribution provider, and they will return the photo using the closest available server to do it. So where does cloud come in?

Instead of optimizing your existing storage set up, the goal of cloud hosting is to provide managed services that solve the problem from start to finish. This means that instead of storing the photo locally and then optimizing that configuration by using a CDN, you'd use a managed storage service for the photo, which handles content distribution automatically. This is exactly what Google Cloud Storage does.

In this case, when someone uploads a photo to your server, you'd resize it and edit it however you want, and then forward the final image along to Google Cloud Storage using their API client to do the work up shipping the bytes securely. After that, all you'd do is refer to the photo using the Cloud Storage URL, and all of the problems from before are taken care of.

**Figure 1.5. Serving photos statically through Google Cloud Storage**



This is only one simple example, but the theme you should take away from this is that cloud is more than just a different way of managing computing resources. It's also about using managed or hosted services via simple APIs to do really complex things meaning you think less about the physical computers.

Since more complex examples are, naturally, more difficult to explain quickly, let's next introduce a few specific examples of companies or projects you might build or work on. We'll use these later to explore some of the interesting ways that cloud infrastructure attempts to solve the common problems found with these projects.

### 1.5.3 Example projects

In order to drive home some of the cool things about cloud infrastructure, let's explore a few concrete examples of projects you might work on.

#### To-do List

If you've ever researched a new web development framework, you've probably seen this example paraded around, specifically showcasing the speed at which you can do "something real". ("Look how easy it is to make a to-do list app with our framework!") To-do List is nothing more than an application that allows users to create lists, add

items to the lists, and mark them as complete.

Throughout this book, we'll rely on this example to illustrate how you might use Google Cloud for your personal or "hobbyist" projects, which quite often just involve storing and retrieving data, while serving either API or web requests to users. You'll notice that the focus of this example is building "something real", however it won't cover all of the edge cases (and there may be many), nor any of the more advanced or "enterprise-grade" features. In short, the To-do List is really going to be useful as a demonstration of "doing something real, but incredibly simple" with cloud infrastructure.

### **INSTASNAP**

InstaSnap is going to be our typical example of "the next big thing" in the start-up world. This application allows users to take photos or videos, share them on a "timeline" (akin to Instagram's or Facebook's timeline), and have them "self-destruct" (akin to SnapChat's expiration).

The wrench thrown in with InstaSnap is that while in the early days most of the focus was on building the application, the current focus is on scaling the application to handle hundreds of thousands of requests every single second. Additionally, all of these photos and videos, while small on their own, add up to enormous amounts of data, and celebrities have started using the system, meaning it's becoming more and more common for thousands of people to all request the same photos at the same time. We'll rely on this example to demonstrate how cloud infrastructure can be used to achieve stability even in the face of incredibly numbers of requests. We also may use this example when pointing out some of the more advanced features provided by cloud infrastructure.

### **E\*EXCHANGE**

E\*Exchange is our example of more "grown-up" application development, that tends to come with growing from a small or mid-sized company into a larger, more mature, more heavily capitalized company, which means audits, Sarbanes-Oxley, and all the other (potentially scary) requirements. To make things more complicated, E\*Exchange is an application for trading stocks in the US, and therefore will act as an example of applications operating in more highly regulated industries (such as finance).

E\*Exchange will come up whenever we explore several of the many enterprise-grade features of cloud infrastructure, as well as some of the concerns about using shared services, particularly with regard to security and access control. Hopefully these examples will be able to help you bridge the gap between cool features that seem fun (or boring features that seem useless) to real-life use-cases of these features, and how you can rely on cloud infrastructure to do some (or most) of the heavy lifting.

## **1.6 Getting started with Google Cloud Platform**

Now that you've learned a bit about cloud in general, and what Google Cloud Platform can do more specifically, let's begin exploring GCP.

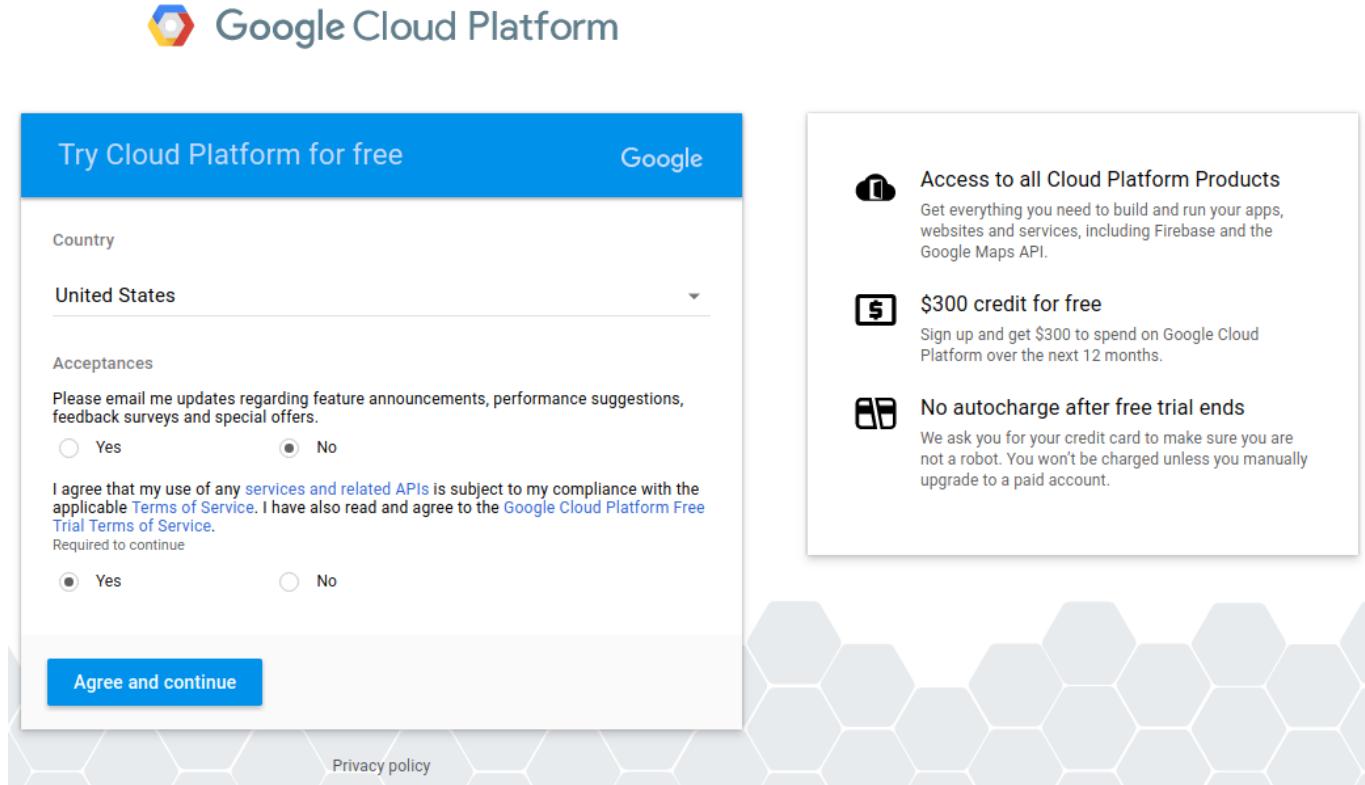
### 1.6.1 Signing up for GCP

Before you can start using any of Google's Cloud services, you first need to sign up for an account. If you already have a Google account (e.g., a Gmail account), you can use that to log in, but you'll still need to sign up specifically for a cloud account. Also, if you've already signed up for Google Cloud Platform, feel free to skip ahead. First, navigate to [cloud.google.com](https://cloud.google.com) and click on the button saying "Try it free!". This will take you through a typical Google sign-in process. If you don't have a Google account yet, just follow the sign-up process to create one.

**Figure 1.6. Google Cloud Platform**

If you're eligible for the free trial, you'll see a page prompting you to enter your billing information. The free trial gives you \$300 to spend on Google Cloud over a period of 12 months, which should be more than enough time to explore all the things in this book. Additionally, some of the products on Google Cloud Platform have an "always free" tier of usage that won't cost you anything. Either way, all the exercises in this book will remind you to turn off any resources after the exercise is finished.

**Figure 1.7. Google Cloud Platform Free Trial**



The screenshot shows the Google Cloud Platform Free Trial sign-up process. At the top, there's a blue header bar with the text "Try Cloud Platform for free" and the "Google" logo. Below this, a "Country" dropdown menu is set to "United States". Under "Acceptances", there's a checkbox for receiving updates and two radio buttons for "Yes" and "No". A note states: "I agree that my use of any services and related APIs is subject to my compliance with the applicable Terms of Service. I have also read and agree to the Google Cloud Platform Free Trial Terms of Service." Below this, another note says "Required to continue" with "Yes" checked. At the bottom left is a blue "Agree and continue" button, and at the bottom center is a link to "Privacy policy". To the right of the main form, there's a white box containing three bullet points: "Access to all Cloud Platform Products" (with a cloud icon), "\$300 credit for free" (with a dollar sign icon), and "No autocharge after free trial ends" (with a credit card icon). The background features a hexagonal pattern.

### 1.6.2 Exploring the console

After you've signed up, you'll be automatically taken to the Cloud Console, and a new project will be automatically created for you. You can think of a project like a container for your work, where the resources in a single project are isolated from those in all the other projects out there.

**Figure 1.8. Google Cloud Console**

The screenshot shows the Google Cloud Platform dashboard for the project "My First Project". The dashboard is divided into several sections:

- Project info:** Shows the project name ("My First Project"), project ID ("my-first-project-191916"), and project number ("675458577735"). A link to "Go to project settings" is also present.
- Compute Engine:** Shows a CPU usage chart from 10:30 to 11:15. The Y-axis represents CPU (%) from 0 to 1.0. The chart shows minimal activity, staying near zero.
- Google Cloud Platform status:** Shows "All services normal" and a link to "Go to Cloud status dashboard".
- Billing:** Shows estimated charges of \$0.00 for the billing period Jan 1 – 12, 2018. A link to "View detailed charges" is available.
- Error Reporting:** States "No sign of any errors. Have you set up Error Reporting?" and a link to "Learn how to set up Error Reporting".
- APIs:** Shows requests per second from 10:30 to 11:15. The Y-axis represents Requests (requests/sec) from 0 to 1.0. The chart shows minimal activity, staying near zero.
- Trace:** States "No trace data from the past 7 days" and a link to "Get started with Stackdriver Trace".
- Getting Started:** Includes links to "Enable APIs and get credentials like keys" and "Deploy a prebuilt solution".
- News:** Lists two articles: "Three ways to configure robust firewall rules" (2 hours ago) and "Why you should pick strong consistency, whenever possible" (22 hours ago).

On the left side of the page, you'll see lots of categories which correspond to all the different services that Google Cloud Platform offers (e.g., Compute, Networking, Big Data, and Storage) as well as other project-specific configuration sections (e.g., authentication, project permissions, and billing). Feel free to take a bit of time poking around in the console to familiarize yourself with where things live. We'll come back to all of these things later on as we explore each of these areas. Before we go any further, let's take a moment to look a bit closer at a concept that we sort of just threw out there: projects.

### 1.6.3 Understanding projects

When we first signed up for Google Cloud Platform, we learned that "a new project was created automatically", and that projects had something to do with isolation, but what does this actually mean? And what are projects anyway? Put simply, projects are

primarily a container for all the resources we create. For example, this means that if we create a new VM it will be "owned" by the parent project. Further, this ownership spills over into billing in that any charges incurred for resources are charged to the project. This means that the new VM we just mentioned will have its bill sent to the person responsible for billing on the parent project. (In our examples, this will be you!)

In addition to acting as the owner of resources, projects also act as a way of isolating things from one another, sort of like having a "workspace" for a specific purpose. This isolation applies primarily to security, to ensure that someone with access to one project doesn't have access to resources in another project unless specifically granted access. For example, if you create new service account credentials (which we'll do later on down the road) from inside one project, say `project-a`, those credentials will only have access to resources inside `project-a` unless you explicitly grant more access.

On the flip side, if you act as "yourself" (e.g., `you@gmail.com`) when running commands (which we'll try in the next section), those commands will have access to anything that you have access to inside the Cloud Console, which includes **all** of the projects you've created as well as ones that others have shared with you. This is one of the reasons why you'll see much of the code we write often explicitly specifies project IDs: you might have access to lots of different projects so we have to clarify which one we want to own the thing we're creating or which project should get the bill for usage charges. In general, if you imagine you're a freelancer building websites and want to keep the work you do for different clients separate from one another, you'd probably have one project for each of the websites you build, both for billing purposes (one bill per website) and to keep each website securely isolated from the others. This also makes it easy to grant access to each client if they wanted to take ownership over their website or edit something themselves.

Now that we've gotten that out of the way, let's get back into the swing of things and look at how to get started with the Google Cloud SDK.

#### **1.6.4 *Installing the SDK***

After you get comfortable with the Google Cloud Console, you'll want to install the Google Cloud SDK. The SDK is a suite of tools for building software that uses Google Cloud, as well as tools for managing your production resources. In general, anything you can do using the Cloud Console can be done with the Cloud SDK, `gcloud`. To install the SDK, just go to [cloud.google.com/sdk/](https://cloud.google.com/sdk/) and follow the instructions for your platform. For example, on a typical Linux distribution, you'd run:

##### **Listing 1.1. Install the SDK on Linux**

```
$ export CLOUD_SDK_REPO="cloud-sdk-$(lsb_release -c -s)"
$ echo "deb http://packages.cloud.google.com/apt $CLOUD_SDK_REPO main" | \
  sudo tee -a /etc/apt/sources.list.d/google-cloud-sdk.list
$ curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
$ sudo apt-get update && sudo apt-get install google-cloud-sdk
```

Feel free to install anything that looks interesting to you, because you can always add or remove components later on. For each exercise that we go through, we'll always start by reminding you that you may need to install extra components of the Cloud SDK. You also may be occasionally prompted to upgrade components as they become available. For example, here's what you'll see when it's time to upgrade:

#### **Listing 1.2. Update the components of the SDK**

```
Updates are available for some Cloud SDK components. To install
them, please run:
$ gcloud components update
```

As you can see, upgrading components is pretty simple: just run `gcloud components update` and the SDK will handle everything from there. After you have everything installed, you have to tell the SDK who you are by logging in. Google made this really easy by connecting your terminal and your browser. You can "login" to the Cloud SDK by typing:

#### **Listing 1.3. Login with your Google account**

```
$ gcloud auth login
Your browser has been opened to visit:

[A really long link is here]

Created new window in existing browser session.
```

You should see a normal Google login and authorization screen asking you to allow the Google Cloud SDK to have access to your cloud resources. This means that when you run future `gcloud` commands you can talk to Google Cloud Platform APIs as yourself. Once you click "Allow", the window should automatically close, and the prompt should update to look like Listing 1.4.

#### **Listing 1.4. What you should see after logging in successfully**

```
$ gcloud auth login
Your browser has been opened to visit:

[A really long link is here]

Created new window in existing browser session.
WARNING: `gcloud auth login` no longer writes application default credentials.
If you need to use ADC, see:
  gcloud auth application-default --help

You are now logged in as [your-email-here@gmail.com].
Your current project is [your-project-id-here]. You can change this setting by
running:
$ gcloud config set project PROJECT_ID
```

You're now authenticated and ready to use the Cloud SDK as yourself, however you may be wondering what the warning in that message is. This warning is basically saying that even though you're logged in and all the `gcloud` commands you run will be authenticated as you, code that you write may not be. We can make any code we write in the future automatically handle authentication by using "application default" credentials. We can get these using the `gcloud auth` subcommand once again, shown below.

#### **Listing 1.5. Setting up authentication for code to use**

```
$ gcloud auth application-default login
Your browser has been opened to visit:

[Another really long link is here]

Created new window in existing browser session.

Credentials saved to file:
[/home/jjg/.config/gcloud/application_default_credentials.json]

These credentials will be used by any library that requests
Application Default Credentials.
```

Now that we have dealt with all of the authentication pieces, let's look at how we can actually interact with Google Cloud Platform APIs.

## **1.7 *Interacting with GCP***

Now that you've signed up and played with the console, and your local environment is all set up, it might be a good idea to try a quick practice task in each of the different ways you can interact with GCP. Let's start by launching a virtual machine in the cloud, and then writing a script to terminate the virtual machine in JavaScript.

### **1.7.1 *In the browser: the Cloud Console***

Let's start by navigating to the Google Compute Engine area of the console. You can get to this by clicking on the Compute section, and once that expands you'll see a link underneath called "Compute Engine". The first time you click on this link, Google will "initialize" Compute Engine for you, which should take just a few seconds. Once that's complete, you should see a button that says "Create" which will bring you to a page that lets you configure your virtual machine.

**Figure 1.9. Google Cloud Console where you can create a new virtual machine**

The screenshot shows the Google Cloud Platform interface. At the top, there's a navigation bar with the 'Google Cloud Platform' logo, a dropdown for 'My First Project', a search bar, and several icons for notifications, help, and user profile.

The main area is titled 'Compute Engine' and 'VM instances'. On the left, a sidebar lists various Compute Engine services: VM instances (selected), Instance groups, Instance templates, Disks, Snapshots, Images, Committed use discounts, Metadata, Health checks, Zones, Operations, Quotas, and Settings. The 'VM instances' section contains a brief description of Compute Engine, mentioning virtual machines running on Google's infrastructure, and three buttons: 'Create', 'Import', and 'Take the quickstart'.

On the next page, there will be a form that lets you configure all the details of your "instance" so let's take a moment to look at what all of the options are.

**Figure 1.10. Form where you define your virtual machine**

[← Create an instance](#)

---

**Name** [?](#)  
learning-cloud-demo

**Zone** [?](#)  
us-east1-b

**Machine type**  
Customize to select cores, memory and GPUs.

1 vCPU    3.75 GB memory    [Customize](#)

**Container** [?](#)  
 Deploy a container image to this VM instance. [Learn more](#)

**Boot disk** [?](#)  
 New 10 GB standard persistent disk  
 Image  
 Debian GNU/Linux 9 (stretch)    [Change](#)

**Identity and API access** [?](#)

**Service account** [?](#)  
Compute Engine default service account

**Access scopes** [?](#)  
 Allow default access  
 Allow full access to all Cloud APIs  
 Set access for each API

**Firewall** [?](#)  
Add tags and firewall rules to allow specific network traffic from the Internet

Allow HTTP traffic  
 Allow HTTPS traffic

[▼ Management, disks, networking, SSH keys](#)

---

You will be billed for this instance. [Learn more](#)

[Create](#) [Cancel](#)

Equivalent [REST](#) or [command line](#)

First there is the "instance name". The name of your virtual machine will be unique inside your project, for example, if you try to create "instance-1" while you already have an instance with that same name, you'll get an error saying that name is already taken. You can name your machines anything you want, so let's name our instance

"learning-cloud-demo". Below that you'll see a field called "zone", which represents where the machine should live geographically. Google has data centers all over the place, so you can choose from quite a few options of where you want your instance to live. For now, let's put our instance in `us-central1-b` (which is actually in Iowa).

The next field you can play with is what's called the "machine type". Just like when you buy your own computer at home, you can choose how powerful you want your cloud instances as well. Google has lots of different sizing options, ranging from `f1-micro` (which is a very small, not very powerful machine) all the way up to `n1-highcpu-32` (which is a 32-core machine), or a `n1-highmem-32` (which is a 32-core machine with 208 GB of RAM). As you can see, there are quite a few options, but because we're just testing things out, let's leave the machine type as `n1-standard-1`, which is a single core machine with about 4 GB of RAM.

There are many many more knobs that you can turn to configure your machine, but for now, let's just launch this `n1-standard-1` machine to test things out. All you have to do to start the virtual machine is click "Create" and then wait a few seconds.

#### TESTING OUT YOUR INSTANCE

Once your machine is created, you should see a green checkmark in the list of instances in the console. But what can you do with this now? You might notice a link in the last column of the list of instances called "Connect", with a button that says "SSH" in the cell.

**Figure 1.11. The listing of your VM instances**

Name	Zone	Recommendation	Internal IP	External IP	Connect
learning-cloud-demo	us-east1-b		10.142.0.2	35.227.93.212	SSH

If you click on this button, a new window will pop up and after waiting a few seconds, you should see a terminal. This terminal is actually running on your new virtual machine, so feel free to play around — typing `top` or `cat /etc/issue` or anything else that you're curious about.

## 1.7.2 On the command line: gcloud

Now that you've tried out creating an instance in the console, you might be curious how the Cloud SDK comes into play. As mentioned before, anything that you can do in the Cloud Console can also be done using the `gcloud` command, so let's put that to the test by trying to look at the list of your instances, and then connecting to the instance just like you did with the SSH button. Let's start by listing the instances. To do this, just type `gcloud compute instances list`. You should see output that looks something like the following snippet.

### **Listing 1.6. The list of instances**

```
$ gcloud compute instances list
NAME          ZONE      MACHINE_TYPE  PREEMPTIBLE INTERNAL_IP  EXTERNAL_IP
STATUS
learning-cloud-demo us-central1-b n1-standard-1           10.240.0.2
104.154.94.41  RUNNING
```

Cool right? There's your instance that you created, just as it appeared in the console.

### **CONNECTING TO YOUR INSTANCE**

Now that you can see your instance, you probably are curious about how to connect to it like we did before with the SSH button. This is equally simple using the Cloud SDK, just type `gcloud compute ssh learning-cloud-demo`, choose the zone where we created the machine (`us-central1-b`) and you should be connected to your machine via SSH.

### **Listing 1.7. Connecting to your machine via SSH**

```
$ gcloud compute ssh learning-cloud-demo
For the following instances:
- [learning-cloud-demo]
choose a zone:
[1] asia-east1-c
[2] asia-east1-a
[3] asia-east1-b
[4] europe-west1-c
[5] europe-west1-d
[6] europe-west1-b
[7] us-central1-f
[8] us-central1-c
[9] us-central1-b
[10] us-central1-a
[11] us-east1-c
[12] us-east1-b
[13] us-east1-d
Please enter your numeric choice: 9

Updated [https://www.googleapis.com/compute/v1/projects/glass-arcade-111313].
Warning: Permanently added '104.154.94.41' (ECDSA) to the list of known hosts.
Warning: Permanently added '104.154.94.41' (ECDSA) to the list of known hosts.
Linux learning-cloud-demo 3.16.0-0.bpo.4-amd64 #1 SMP Debian 3.16.7-ckt11-
```

```
1+deb8u3~bpo70+1 (2015-08-08) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
jjg@learning-cloud-demo:~$
```

Under the hood, Google is using the credentials it obtained when you ran `gcloud auth login`, generating a new public/private key pair, securely putting the new public key onto the virtual machine, and then using the private key generated to connect to the machine. This means that you don't have to worry about key pairs when connecting. As long as you have access to your Google account, you can always access your virtual machines!

### **1.7.3 In your own code: google-cloud-\***

Now that we've created an instance inside the Cloud Console, then connected to that instance from the command line using the Cloud SDK, let's explore the last way you can interact with your resources: in your own code. What we'll do in this section is write a small Node.js script that connects and terminates your instance. This has the fun side-effect of turning off your machine so you don't waste any money during your free-trial! To start, if you don't have Node.js installed, you should do that by going to [nodejs.org](http://nodejs.org) and downloading the latest version. You can test that this all worked by running the `node` command with the `--version` flag:

#### **Listing 1.8. Check your Node.js version**

```
$ node --version
v7.7.1
```

After this, you should install the Google Cloud client library for Node.js. You can do this with the `npm` command:

#### **Listing 1.9. Install the @google-cloud/compute client library for Node**

```
$ sudo npm install --save @google-cloud/compute@0.7.1
```

After this, it's time to start writing some code that connects to your cloud resources. To start, let's try to list out the instances currently running. To do this, put the following code into a script called `script.js` and then run it using `node script.js`.

#### **Listing 1.10. Showing all VMs (script.js)**

```
const gce = require('@google-cloud/compute')({
  projectId: 'your-project-id' ①
});
const zone = gce.zone('us-central1-b');
```

```
console.log('Getting your VMs...');

zone.getVMs().then((data) => {
  data[0].forEach((vm) => {
    console.log('Found a VM called', vm.name);
  });
  console.log('Done.');
});
```

- ➊ Make sure to change this to your project ID!

If you run this script, the output should look something like the following output.

#### **Listing 1.11. Output of showing all VMs**

```
$ node script.js
Getting your VMs...
Found a VM called learning-cloud-demo
Done.
```

Now that we know how to list the VMs in a given zone, let's try turning off the VM using our script. To do this, update your code to look like this.

#### **Listing 1.12. Showing and stopping all VMs**

```
const gce = require('@google-cloud/compute')({
  projectId: 'your-project-id'
});
const zone = gce.zone('us-central1-b');

console.log('Getting your VMs...');

zone.getVMs().then((data) => {
  data[0].forEach((vm) => {
    console.log('Found a VM called', vm.name);
    console.log('Stopping', vm.name, '...');
    vm.stop((err, operation) => {
      operation.on('complete', (err) => {
        console.log('Stopped', vm.name);
      });
    });
  });
});
```

This script might take a bit longer to run, but when it's complete, the output should look something like the following.

#### **Listing 1.13. Output of stopping all VMs**

```
$ node script.js
Getting your VMs...
Found a VM called learning-cloud-demo
Stopping learning-cloud-demo ...
Stopped learning-cloud-demo
```

Now the virtual machine we started in the UI is in a "stopped" state, and can be restarted later when you want to use it again. Now that we've played with virtual machines and managing them with all of the tools available (the Cloud Console, the Cloud SDK, and your own code), let's keep the ball rolling by learning how to deploy a real application using Google Compute Engine.

## 1.8 **Summary**

- "Cloud" has become a buzz-word, but for this book it's a collection of services that abstract away computer infrastructure.
- Cloud is a good fit if you don't want to manage your own servers or data centers, and your needs change often or you don't know them.
- Cloud is a **bad** fit if your usage is very steady over very long periods of time.
- When in doubt, if you need tools for GCP, start at [cloud.google.com](https://cloud.google.com).

# 2

## *Trying it out:*

# *Deploying Wordpress on Google Cloud*

### **This chapter covers:**

- What is WordPress?
- Laying out the pieces of a WordPress deployment
- Turning on a SQL database to store your data
- Turning on a VM to run WordPress
- Turning everything off

## **2.1 Getting started with WordPress**

If you've ever explored hosting your own website or blog, chances are you've come across (or maybe even installed) WordPress. There's really not a lot of debate about WordPress's popularity, with millions of websites relying on it for their websites or blogs, but many public blogs are hosted by other companies such as HostGator, BlueHost, or WordPress's own hosted service, WordPress.com (not to be confused with the open source project WordPress.org).

To demonstrate the simplicity of Google Cloud, this chapter is going to walk you through deploying WordPress yourself using Google Compute Engine and Google Cloud SQL to host your infrastructure.

**NOTE**

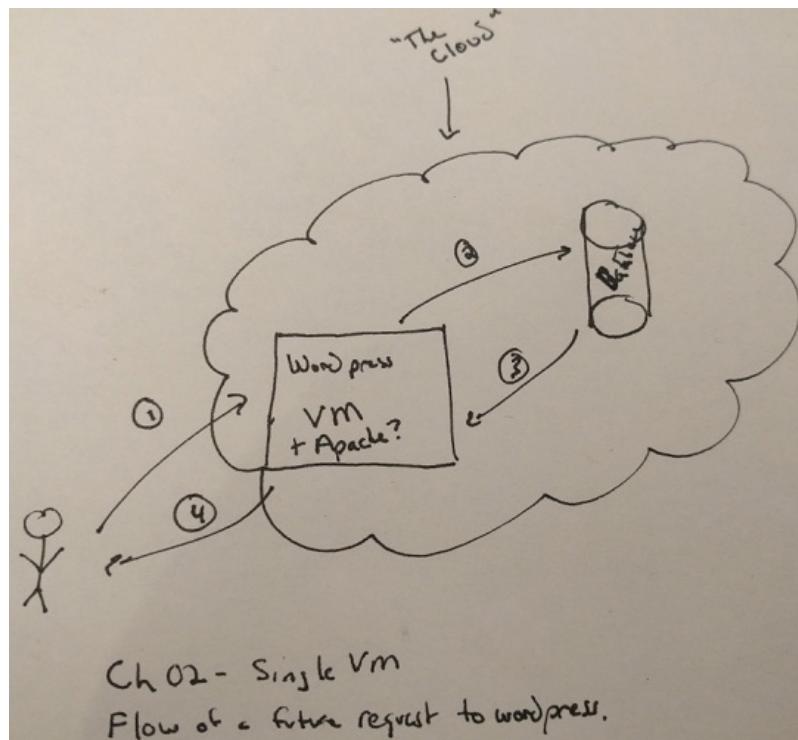
The pieces we'll turn on here will be part of the free-trial from Google, however if you leave them running past your free trial, your system will cost just around a few dollars per month.

First, let's put together an architectural plan for how we'll deploy WordPress using all the cool new tools you just learned about in the last chapter.

## 2.2 System layout overview

Before we get down to the technical pieces of turning on machines, let's start by looking at what we need to turn on. The way we'll do this is to look at the flow of an "ideal request" through our future system. In other words, we're going to imagine a person visiting our future blog, and look at where their request needs to go in order to give them a great experience. We'll start with a single machine, since that's the simplest possible configuration:

**Figure 2.1. Flow of a future request to a VM running WordPress**



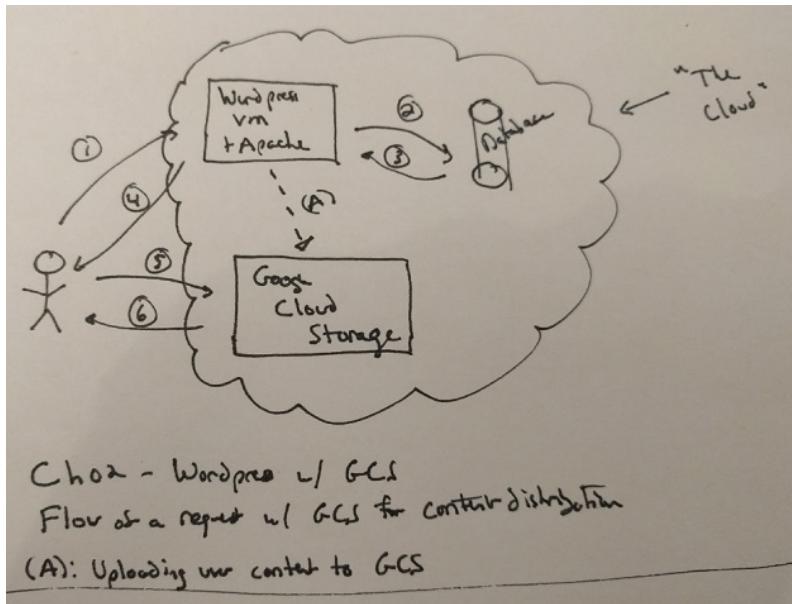
As you can see here, the flow is:

1. someone asks the WordPress server for a page
2. the WordPress server queries the database
3. the database sends back a result (e.g., the content of the page)
4. the WordPress server sends back a web page

Simple enough right? What happens as things get a bit more complex? Although we won't demonstrate this configuration here, you might recall in [chapter 1](#) where we discussed the idea of relying on cloud services for more complicated hosting problems like content distribution. (For example, if your servers are in the US, what's the experience going to be like for your readers in Asia?). To show an idea of how this might look, here's a flow diagram for a WordPress server using Google Cloud Storage

to handle static content (like images):

**Figure 2.2. Flow of a request involving Google Cloud Storage**



In this case, the flow is the same to start, however unlike before, when static content is requested it doesn't re-use the same flow. In this configuration, your WordPress server will modify references to static content so that rather than requesting it from the WordPress server, the browser will request it from Google Cloud Storage (steps 5 and 6).

This means that requests for images and other static content will be handled by Google Cloud Storage directly, which can do fancy things like distributing your content around the world and caching the data close to your readers. This means that your static content will be delivered quickly no matter how far they are from your WordPress server. Now that you have an idea of how the pieces will talk to each other, it's time to start exploring each piece individually and find out what exactly is happening under the hood.

## 2.3 **Digging into the database**

So far we've drawn this picture involving a database but we haven't really said much about what type of database we're talking about. There are tons of different databases available, but one of the most popular open-source databases is MySQL, which hopefully you've at least heard of. MySQL is great at storing relational data and has plenty of knobs to turn for when you need to start squeezing more and more performance out of it. For now, we're not all that concerned about performance, but it's nice to know that we'll have some wiggle room if things get bigger.

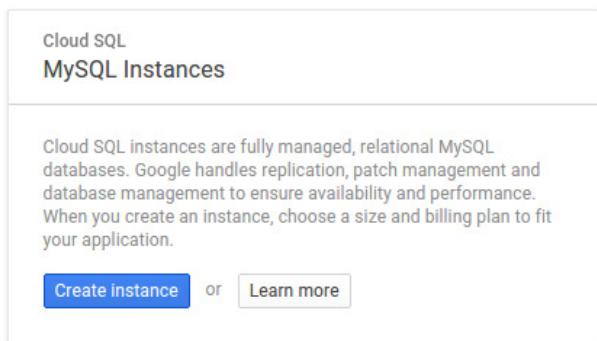
In the early days of cloud computing, the standard way to turn on a database like MySQL was to create a virtual machine, install the MySQL binary package, and then manage that virtual machine like any regular server. However as time went on, cloud providers started noticing that databases all seemed to follow this same pattern, so they started offering "managed database services" where you don't have to configure the virtual machine yourself, but instead turn on a "managed virtual machine" running a specific binary.

All of the major cloud hosting providers offer this sort of service, for example Amazon has its Relational Database Service (RDS), Azure has its SQL Database service, and Google has its Cloud SQL service. And since managing a database via Cloud SQL is quicker and easier than configuring and managing the underlying virtual machine and its software, we're going to use Cloud SQL for our database. This isn't always going to be the best choice (see Chapter X for much more detail about Cloud SQL), but for our WordPress deployment, which is pretty typical, Cloud SQL is a great fit as it looks almost identical to a MySQL server that you'd configure yourself, but is easier and faster to set up.

### 2.3.1 Turning on a Cloud SQL instance

The first step to turning on our database is to jump into the Cloud Console by going to the Cloud Console ([cloud.google.com/console](https://cloud.google.com/console)) and then clicking on "SQL" in the left-side navigation, underneath the "Storage" section. Once you get there, you'll see a nice blue button that says "Create instance":

**Figure 2.3. Prompt to create a new Cloud SQL instance**



After that, choose to create a "Second Generation" instance (see Chapter X for more detail on these) and then you'll be taken to a page where you can enter some information about your database. The first thing you should notice is that this page looks a little bit like the page you saw when creating a virtual machine. This is intentional because what's happening under the hood is you're actually creating a virtual machine which Google will manage for you, and kick off by installing and configuring MySQL for you. So, just like with a virtual machine, you need to name your database. For this exercise, let's name the database `wordpress-db` (also just like VMs, the name has to be unique inside your project, so you can only have one database

with this name at a time).

Next, we should probably choose the password we'll use to access MySQL. In this case you have an option of letting the Cloud Console automatically generate a new secure password, or you can choose your own. In this case, we'll choose `my-very-long-password!`. Finally, again just like a VM, you have to choose where (geographically) you want your database to live. For this example, we'll use `us-central1-c` as our zone.

**Figure 2.4. Form to create a new Cloud SQL instance**

The screenshot shows the 'Create a MySQL Second Generation instance' page. At the top, there's a navigation bar with a 'SQL' icon and a back arrow labeled 'Create a MySQL Second Generation instance'. On the left, there are input fields for 'Instance ID' (containing 'wordpress-db') and 'Root password' (containing 'my-very-long-password!'). Below these are checkboxes for 'No password' and 'Location' (set to 'us-central1'). On the right, there's a summary box showing 'Estimated monthly total' (\$51.89), 'Hourly rate' (\$0.071), and '~730 hours per month'. Below this are three price options: 'db-n1-standard-1 machine' (\$70.45 / month), '10 GB SSD, with backups' (\$2.58 / month), and 'Committed use discount' (-\$21.13 / month). At the bottom, there are 'Create' and 'Cancel' buttons.

To do any further configuration, we need to dig deeper by clicking the "Show configuration options" link towards the bottom of the page. For example, we might want to change the size of the VM instance for our database (by default, this uses a `db-n1-standard-1` type instance), or increase the size of the underlying disk (by default, Cloud SQL starts with a 10 GB SSD disk). It turns out that the options on this page can all be changed later on (in fact, the size of your disk will automatically increase as needed), so let's leave them as they are and create our instance. Once you've created your instance, you can use the `gcloud` command-line tool to show that it's all set with the `gcloud sql instances list` command, shown below.

#### **Listing 2.1. A list of running Cloud SQL instances**

```
$ gcloud sql instances list
```

NAME	REGION	TIER	ADDRESS	STATUS
wordpress-db	-	db-n1-standard-1	104.197.207.227	RUNNABLE

**QUIZ**

Can you think of a time that you might have a very large persistent disk that will be mostly empty?

Take a look at [chapter 9](#) if you're not sure.

### 2.3.2 Securing your Cloud SQL instance

Before we go any further, you should probably change a few settings on your SQL instance so that you (and hopefully only you) can connect to it. What we'll do for our testing phase is change the password on the instance, and then open it up to the world. Then, after we've tested it out, we'll change the network settings to only allow access from your Compute Engine VMs. First, let's change the password. You can do this from the command line with the `gcloud sql users set-password` command as shown below.

#### Listing 2.2. Setting the root password for a Cloud SQL instance

```
$ gcloud sql users set-password root "%" --password "my-changed-long-password-2!" \
--instance wordpress-db
Updating Cloud SQL user...done.
```

In this example, we reset the password for the root user across all hosts (MySQL's wild card character here is a percent sign). Now that the password is set, let's (temporarily) open the SQL instance to the outside world. To do this, go into the Cloud Console and navigate to your Cloud SQL instance. Towards the top you should see a few tabs, one of them is "Authorization". Under this tab, there's a bit button saying "Add network" which will let you control your "Authorized Networks". Click on that and add "the world" in CIDR notation (`0.0.0.0/0`, which means "all IPs possible") and click "Save".

**Figure 2.5. Configuring access to the Cloud SQL instance**

The screenshot shows the Google Cloud Platform interface for managing a Cloud SQL instance named "wordpress-db". The "AUTHORIZATION" tab is selected. Two notifications are displayed:

- A blue info icon notification: "You have not authorized any external networks to connect to your Cloud SQL instance. External applications can still connect to the instance through the Cloud SQL Proxy. [Learn more](#)".
- A yellow warning icon notification: "You have added 0.0.0.0/0 as an allowed network. This prefix will allow any IPv4 client to pass the network firewall and make login attempts to your instance, including clients you did not intend to allow. Clients still need valid credentials to successfully log in to your instance."

The "Authorized networks" section contains a "New network" dialog box. The dialog has the following fields:

- Name (Optional):** None
- Network:** Use CIDR notation. 0.0.0.0/0
- Buttons:** Done, Cancel

Below the dialog is a button labeled "+ Add network".

The "App Engine authorization" section states: "All apps in this project are authorized by default. To authorize apps in other projects, follow the steps below." It includes two items:

- Apps in this project: All authorized.
- Authorize apps in other projects

At the bottom are "Save" and "Discard changes" buttons.

**CAUTION** You'll notice a little warning about opening your database to any IP address. This is OK as we're just doing some testing, but **you should never leave this setting for your production environments**. You'll learn more about securing your SQL instance for your cluster later on.

Now it's time to test whether all of this worked!

### 2.3.2 Connecting to your Cloud SQL instance

If you don't have a MySQL client, the first thing to do is install one. On a Linux environment like Ubuntu you can install it by typing

#### Listing 2.3. Installing the MySQL client library on Ubuntu Linux

```
$ sudo apt-get install -y mysql-client
```

On Windows or Mac, you can download the package from MySQL's website: [dev.mysql.com/downloads/mysql/](http://dev.mysql.com/downloads/mysql/). After you have that installed, connecting to the database is pretty simple: just enter the IP address of your instance (you saw this before with `gcloud sql instances list`), use the username "root", and the password you set just a bit ago. For example, on a Linux computer, here's what you might do:

#### Listing 2.4. Connecting to a Cloud SQL instance using the MySQL client

```
$ mysql -h 104.197.207.227 -u root -p
Enter password: # <I typed my password here>
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 59
Server version: 5.7.14-google-log (Google)

Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

Great work! You now have a SQL database server living in the cloud! Next, let's run a few SQL commands to prepare your database for WordPress.

### 2.3.4 Configuring your Cloud SQL instance for WordPress

You have a MySQL database that you can successfully talk to, so let's get that database prepared for WordPress to start talking to it. Here's a basic outline of what we're going to do:

1. Create a database called `wordpress`
2. Create a user called `wordpress`
3. Give the `wordpress` user the appropriate permissions

The first thing you should do is get back to that MySQL command-line prompt. As you just learned, you can do this by running the `mysql` command. Next up is to create the database. Do this by running

#### **Listing 2.5. Creating the wordpress database in MySQL**

```
mysql> CREATE DATABASE wordpress;
Query OK, 1 row affected (0.10 sec)
```

Then we need to create a user account for WordPress to use for access to the database:

#### **Listing 2.6. Creating the wordpress user in MySQL**

```
mysql> CREATE USER wordpress IDENTIFIED BY 'very-long-wordpress-password';
Query OK, 0 rows affected (0.21 sec)
```

And then we need to give this new user the right level of access to do things to the database (like create tables, add rows, run queries, etc):

#### **Listing 2.7. Granting access to the wordpress user in MySQL**

```
mysql> GRANT ALL PRIVILEGES ON wordpress.* TO wordpress;
Query OK, 0 rows affected (0.20 sec)
```

And finally let's tell MySQL to reload the list of users and privileges. If we forget this command, MySQL would know about the changes when it restarts, but we don't want to restart our Cloud SQL instance just for this!

#### **Listing 2.8. Refreshing the database's access controls after changes**

```
mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.12 sec)
```

And that's all you have to do on the database! Next, let's make it do something real rather than just sit around as a toy.

**QUIZ**

How does your database get backed up?

Take a look at the chapter on Cloud SQL if you're not sure.

## **2.4 Deploying the WordPress VM**

Let's start by turning on the VM that will host our WordPress installation. As you learned previously, you can do this pretty easily in the Cloud Console, so let's just do that once more.

**Figure 2.6. Creating a new VM instance**

← Create an instance

Name ?  
wordpress

Zone ?  
us-central1-c

Machine type  
Customize to select cores, memory and GPUs.

\$26.27 per month estimated  
Effective hourly rate \$0.036 (730 hours per month)

Container ?  
 Deploy a container image to this VM instance. [Learn more](#)

Boot disk ?  
 New 50 GB standard persistent disk  
 Image  
 Ubuntu 16.04 LTS Change

Identity and API access ?  
 Service account ?

Access scopes ?  
 Allow default access  
 Allow full access to all Cloud APIs  
 Set access for each API

Firewall ?  
 Add tags and firewall rules to allow specific network traffic from the Internet  
 Allow HTTP traffic  
 Allow HTTPS traffic

Management, disks, networking, SSH keys

You will be billed for this instance. [Learn more](#)

Create Cancel

Equivalent [REST](#) or [command line](#)

Take note that the check boxes for allowing HTTP and HTTPS traffic are checked, because we want our WordPress server to be accessible to anyone through their browsers. Also make sure that the "Access scopes" section is left set allowing default

access. After that, you're ready to turn on your VM, so go ahead and click "Create".

**QUIZ**

- Where does your virtual machine physically exist?
- What will happen if the hardware running your virtual machine has a problem?

Take a look at [chapter 3](#) if you're not sure.

## 2.5 Configuring WordPress

The first thing to do now that your VM is up and running is to SSH into it. You can do this in the Cloud Console by clicking the SSH button, or use the Cloud SDK with the `gcloud compute ssh` command. For this walk through, we'll use the Cloud SDK to connect to our VM.

### **Listing 2.9. Using the Cloud SDK to SSH to the VM**

```
$ gcloud compute ssh --zone us-central1-c wordpress
Warning: Permanently added 'compute.6766322253788016173' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.13.0-1008-gcp x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

 Get cloud support with Ubuntu Advantage Cloud Guest:
      http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

jjg@wordpress:~$
```

Once you're connected, we need to install a few packages, namely Apache, MySQL Client, and PHP. We can do this using `apt-get`, shown below.

### **Listing 2.10. Installing Apache, MySQL, and PHP on a VM**

```
jjg@wordpress:~$ sudo apt-get update
jjg@wordpress:~$ sudo apt-get install apache2 mysql-client php7.0-mysql php7.0-libapache2-mod-php7.0 php7.0-mcrypt php7.0-gd
```

When prompted, confirm (by typing Y and hitting enter). Now that you have all the

prerequisites installed, it's time to actually install WordPress. Start by downloading the latest version from [wordpress.org/latest.tar.gz](http://wordpress.org/latest.tar.gz), and unzipping it into your home directory.

#### **Listing 2.11. Downloading and unzipping the latest version of WordPress**

```
jj@wordpress:~$ wget http://wordpress.org/latest.tar.gz
jj@wordpress:~$ tar xzvf latest.tar.gz
```

After you've done that, you'll need to set some configuration parameters, primarily where WordPress should store data, and how to authenticate. To do this, copy the sample configuration file to `wp-config.php`, and then edit the file to point to your Cloud SQL instance (in this example, I'm using vim, but you can use whichever text editor you're most comfortable with):

#### **Listing 2.12. Editing the default configuration of a WordPress installation**

```
jj@wordpress:~$ cd wordpress
jj@wordpress:~/wordpress$ cp wp-config-sample.php wp-config.php
jj@wordpress:~/wordpress$ vim wp-config.php
```

After editing `wp-config.php`, it should look something like this:

#### **Listing 2.13. WordPress configuration after making changes for our environment**

```
<?php
/**
 * The base configuration for WordPress
 *
 * The wp-config.php creation script uses this file during the
 * installation. You don't have to use the web site, you can
 * copy this file to "wp-config.php" and fill in the values.
 *
 * This file contains the following configurations:
 *
 * * MySQL settings
 * * Secret keys
 * * Database table prefix
 * * ABSPATH
 *
 * @link https://codex.wordpress.org/Editing_wp-config.php
 *
 * @package WordPress
 */

/** MySQL settings - You can get this info from your web host */
/** The name of the database for WordPress */
define('DB_NAME', 'wordpress');

/** MySQL database username */
define('DB_USER', 'wordpress');

/** MySQL database password */
define('DB_PASSWORD', 'very-long-wordpress-password');
```

```
/** MySQL hostname */
define('DB_HOST', '104.197.207.227');

/** Database Charset to use in creating database tables. */
define('DB_CHARSET', 'utf8');

/** The Database Collate type. Don't change this if in doubt. */
define('DB_COLLATE', ''');
```

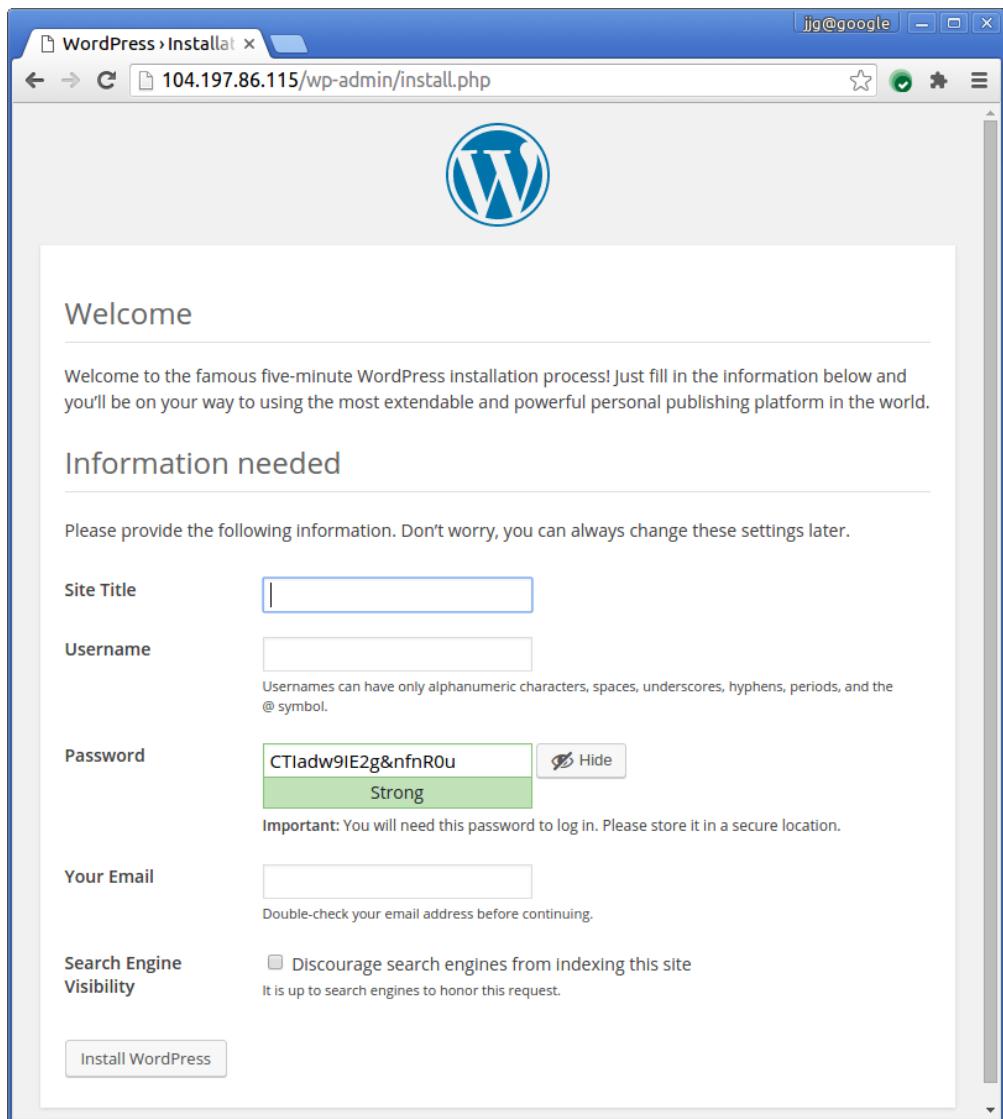
Once you have your configuration the way you want it (you should really only need to change the database settings), it's time to move all those files out of your home directory and into somewhere that Apache can serve them. (We also need to remove the Apache default page, `index.html`) The easiest way to do this is using `rm` and then `rsync`:

**Listing 2.14. Moving the updated configuration files to the WordPress installation directory**

```
jj@wordpress:~/wordpress$ sudo rm /var/www/html/index.html
jj@wordpress:~/wordpress$ sudo rsync -avP ~/wordpress/ /var/www/html/
```

Now all you have to do is navigate to the web server in your browser (e.g., [104.197.86.115](https://104.197.86.115) in this specific example), which should end up looking like this:

**Figure 2.7. Seeing that WordPress is successfully up and running**



From there, just following the prompts should take about 5 minutes and you'll have a working WordPress installation!

## 2.6 *Reviewing the system*

So what did we actually do here? Looking at this in order, we actually set up quite a few different pieces:

1. We turned on a Cloud SQL instance to store all of our data

2. We added a few users, and changed the security rules.
3. We turned on a Compute Engine virtual machine
4. We installed WordPress on that VM

Did we forget anything? Do you remember when we set the security rules on the Cloud SQL instance to accept connections from anywhere (`0.0.0.0/0`)? Now that we know where to accept requests from (our VM), we should fix that. If we don't the database is vulnerable to attacks from the whole world. However, if we lock down the database at the network level, even if someone discovers the password it's only useful if they are connecting from one of our known machines.

To do this, head back over to the Cloud Console and navigate to your Cloud SQL instance. Under the "Access Control" tab, edit the Authorized Network you had before, changing `0.0.0.0/0` to the IP address followed by `/32` (e.g., `104.197.86.115/32`) and rename the rule to read `us-central1-c/wordpress` so you don't forget what this rule is for. When you're done, the access control rules should look like this:

**Figure 2.8. Updating the access configuration for Cloud SQL**

#### Authorized Networks

Add IPv4 or IPv6 addresses below to authorize networks to connect to your instance. Networks will only be authorized via these addresses. If you add IPv4 networks, you must also use the checkbox above to assign an IPv4 address to your instance. Also, note that connections from Google Compute Engine only support IPv4.

The screenshot shows a configuration interface for 'Authorized Networks'. At the top, there is a 'Name (Optional)' field containing 'us-central1-c/wordpress'. Below it is a 'Network' section with a 'Use CIDR notation.' link and an input field containing '104.197.86.115/32'. At the bottom of the form is a blue button labeled '+ Add item'.

Remember that the IP of your VM instance is currently "ephemeral", meaning it could change out from under you. To avoid that, you'll need to reserve a static IP address, but we'll dig into that later on when we explore Compute Engine in more depth.

## 2.7 Turning it off

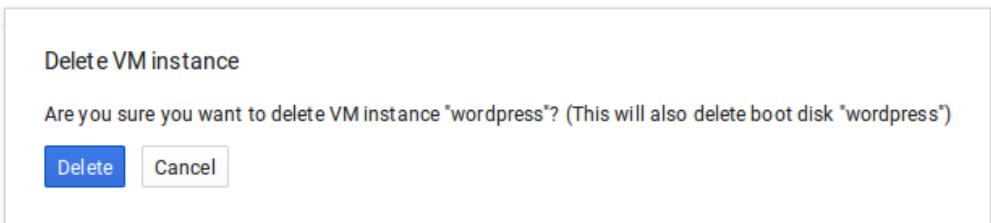
If you actually want to keep your WordPress instance up and running you can skip past this section. (Maybe you have always wanted to host your own blog, and the demo we picked happened to be perfect for you?) If not, let's go through the process of turning

off all those resources you created.

The first thing to turn off is the GCE virtual machine. You can do this using the Cloud Console in the Compute Engine section. When you click on your instance, you might notice that there are two options, "Stop" and "Delete". The difference between these is subtle but important. When you **delete** an instance, it's gone forever like it never existed in the first place. When you **stop** an instance, it's actually still there but in a "paused" state where you can pick up exactly where you left off.

So why wouldn't we just always stop instances rather than delete them? The catch with stopping is that you have to keep your persistent disks around, and those cost money. You won't be paying for CPU cycles on a stopped instance, but the disk that stores the operating system and all your configuration needs to stay around, and you'll be billed for your disks whether or not they're attached to a running virtual machine. In this case, if we're done with our WordPress installation, the right choice here is probably deleting rather than stopping. When you click delete, you should notice that the confirmation prompt reminds you that your disk (the boot disk) will also be deleted.

**Figure 2.9. Deleting the VM when we're finished**



After that, you can do the same thing to your Cloud SQL instance.

## 2.8 Summary

- Google Compute Engine allows you to turn on machines **really** fast: a few clicks and a few seconds of your time.
- When you choose the size of your persistent disk, don't forget that the size also determines the performance. It's OK (and expected) to have lots of empty space on a disk.
- Cloud SQL is "MySQL in a box" using GCE under the hood. It's a great fit if you don't need any special customization.
- You can connect to Cloud SQL databases using the normal MySQL client, so there's no need for any special software.
- It's a bad idea to open your production database to the world (`0.0.0.0/0`).

# The cloud data center



## **This chapter covers:**

- What is a data center?
- Where are the data centers?
- Security and privacy
- Typical problems with shared resources
- Communicating between data centers
- Regions, zones, and disaster isolation
- Automatic replication (computing paradigms)

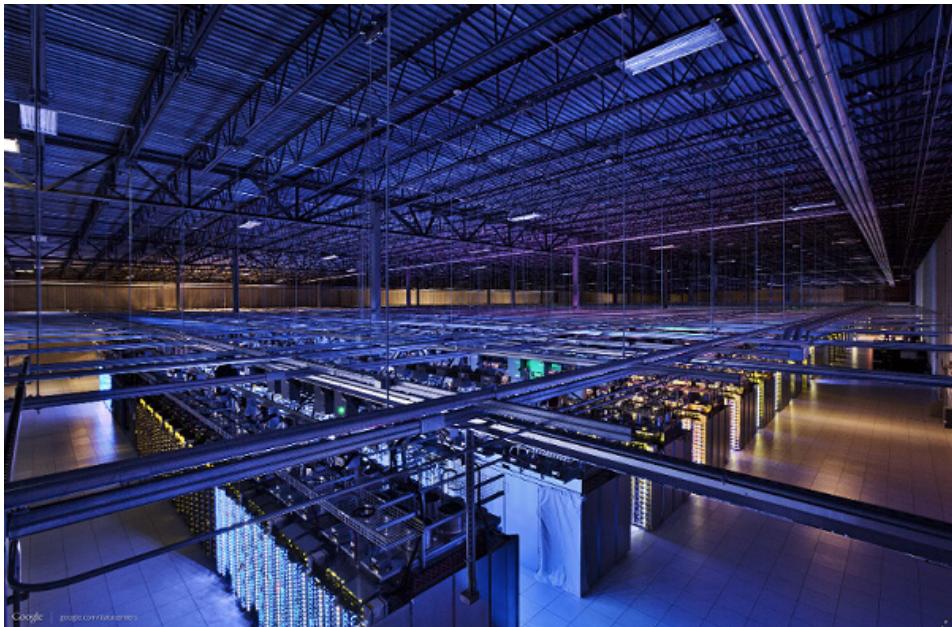
## **3.1 What are cloud data centers?**

If you've ever paid for web hosting before, it's very likely that the computer running as your web host was physically located in what is typically called "data center". And as you learned in [chapter 1](#), deploying "in the cloud" is very similar to traditional hosting, so as you'd expect, if you turn on a virtual machine in, or upload a file to "the cloud", your resources actually live inside a data center. But where are these "data centers"? Are they safe? Should you trust the employees who work in those data centers? Couldn't they steal your data? or the source code to your killer app?

All of these are valid questions, and the answers to them are pretty important — after all, if the data center was in somebody's basement you might not want to put your banking details on that server. The goal of this chapter is to explain how data centers have evolved over time, and highlight some of the details of Google Cloud Platform's data centers (after all, this is a book about GCP). Google's data centers are pretty

impressive (take a look at the photo), but this isn't a fashion show. Before you decide to run mission-critical stuff in a data center, you probably want to get an understanding of how it works.

**Figure 3.1. A Google data center**



Keep in mind though, many of the things you'll read about data centers here are industry wide standards, so many of the great things (such as very strict security to enter the premises) exist with other cloud providers as well (like Amazon Web Services or Microsoft Azure). I'll make sure to call out things that are Google-specific so that it's clear when there's something special to take note of. Let's start by laying out a map to understand Google Cloud's data centers.

## 3.2 Data center locations

You might be thinking that "location" in the world of "the cloud" seems a bit oxymoronic, right? Unfortunately, this is one of the side-effects of marketers pushing "the cloud" as some amorphous mystery, where all of your resources are multi-homed rather than living in a single place. As you'll read later, there are some services which do abstract away the idea of "location" so that your resources live in multiple places simultaneously, however for many services (such as Compute Engine) resources will actually live in a single place. This means that you'll likely want to choose one nearby to your customers.

To choose the right place, you first need to have a look at the choices, so let's get right to it. As of this writing, Google Cloud operates data centers in 15 different regions around the world, covering the US, Brazil, western Europe, India, Asia, and Australia.

**Figure 3.2. Cities where Google Cloud has data centers, and how many in each.**



This might not seem like a lot, however keep in mind that each city actually has many different data centers for you to choose from. By this measure, there are actually 13 different physical places where your resources can exist.

**Table 3.1. Zone overview for Google Cloud**

Region	City	Number of data centers
Eastern US	South Carolina, USA	3
Eastern US	North Virginia, USA	3
Central US	Iowa, USA	4
Western US	Oregon, USA	3
Canada	Montreal, Canada	3
South America	Sao Paulo, Brazil	3
Western Europe	London, UK	3
Western Europe	Belgium	3
Western Europe	Frankfurt, Germany	3
Western Europe	Netherlands	2
South Asia	Mumbai, India	3
South East Asia	Singapore	2
East Asia	Taiwan	3
North East Asia	Tokyo, Japan	3
Australia	Sydney, Australia	3
<b>Total</b>		<b>44</b>

How does this stack up to other cloud providers, as well as traditional hosting

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/google-cloud-platform-in-action>

**Licensed to Asif Qamar <asif@asifqamar.com>**

providers? Let's take a look:

**Table 3.2. Data center offerings by provider**

Provider	Data centers
Google Cloud	44 (across 15 cities)
Amazon Web Services	49 (across 18 cities)
Azure	36 (across 19 cities)
Digital Ocean	11 (across 7 cities)
Rackspace	6

Looking at these numbers, it seems that Google Cloud is performing pretty well compared to the other cloud service providers. That said, there are typically two reasons why you might choose a provider based on data center locations offered, both are focused on network latency:

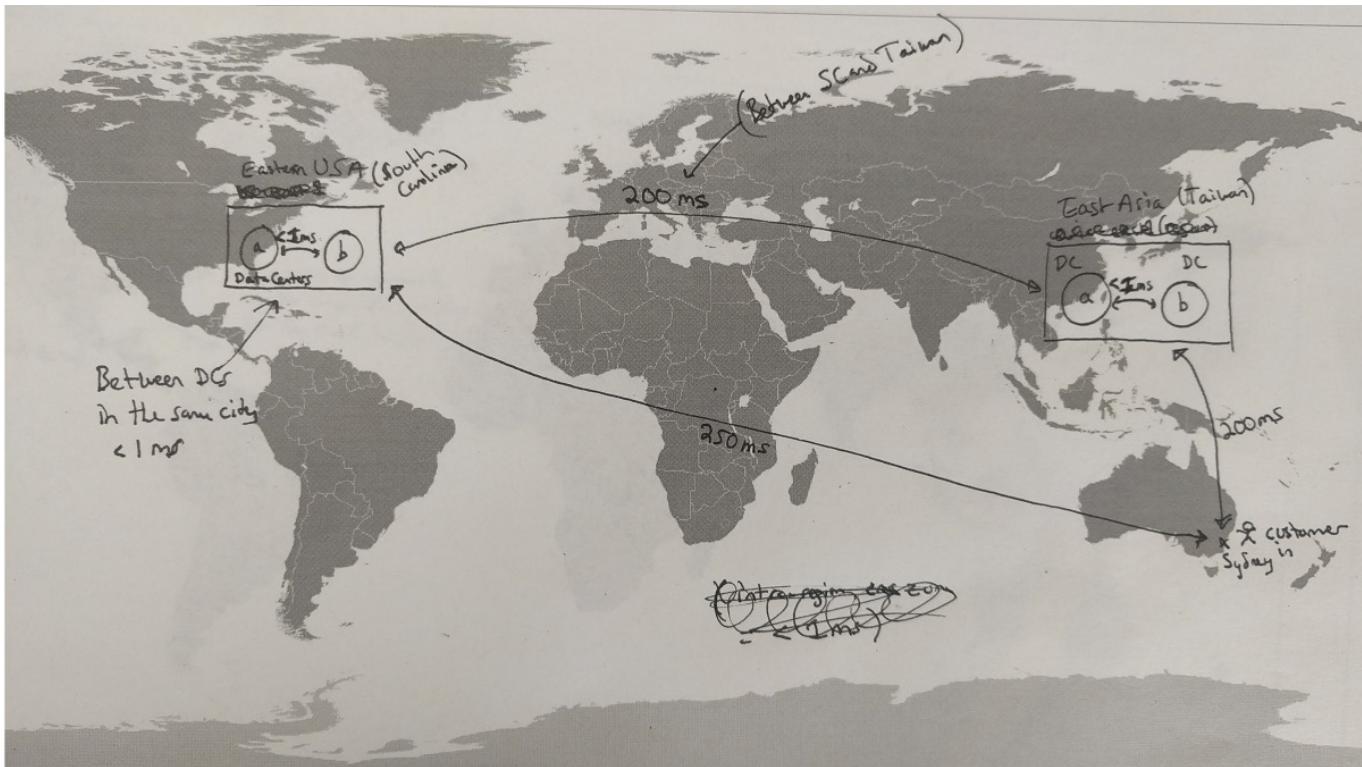
1. You need ultra-low latency between your servers and your customers. An example here is high-frequency trading where you typically need to host services only microseconds away from a stock exchange, because responding even one millisecond slower than your competitors means you still lose out on a trade.
2. You have customers that are really far away from the nearest data center. A common example is businesses in Australia where the nearest options for some services might still be very far away. This means that even something as simple as loading a web page from Australia could be frustratingly slow.

**NOTE**

There is a third reason based on legal concerns which is covered in the "Special cases" section.

If you're requirements are less strict, the location of data centers shouldn't make too much of a difference in choosing a cloud provider, but regardless it's important to understand your latency requirements and what geographical location means for that.

**Figure 3.3. Latencies between different cities and data centers**



Now that you know a bit about where Google Cloud's data centers are, and why location matters, let's briefly discuss the various levels of isolation. You'll need to know about these in order to design a system that will degrade gracefully in the event of some catastrophe.

### 3.3 Isolation levels and fault tolerance

Although we've so far talked about "cities" and "regions" and "data centers", we haven't yet been more specific in defining these different things. Let's start by talking about the types of places that resources can exist.

#### 3.3.1 Zones

A **Zone** is the smallest unit in which a resource can exist. Sometimes it's easiest to think of this as a single facility that holds lots of computers (like a single data center). This means, if you turn on two resources in the same Zone, you can think of that as the two resources living not only geographically nearby, but in the same physical building. There may be times where a single Zone is actually a bunch of buildings, but the point is that from a latency perspective (the ping time, for example) the two resources are very close by.

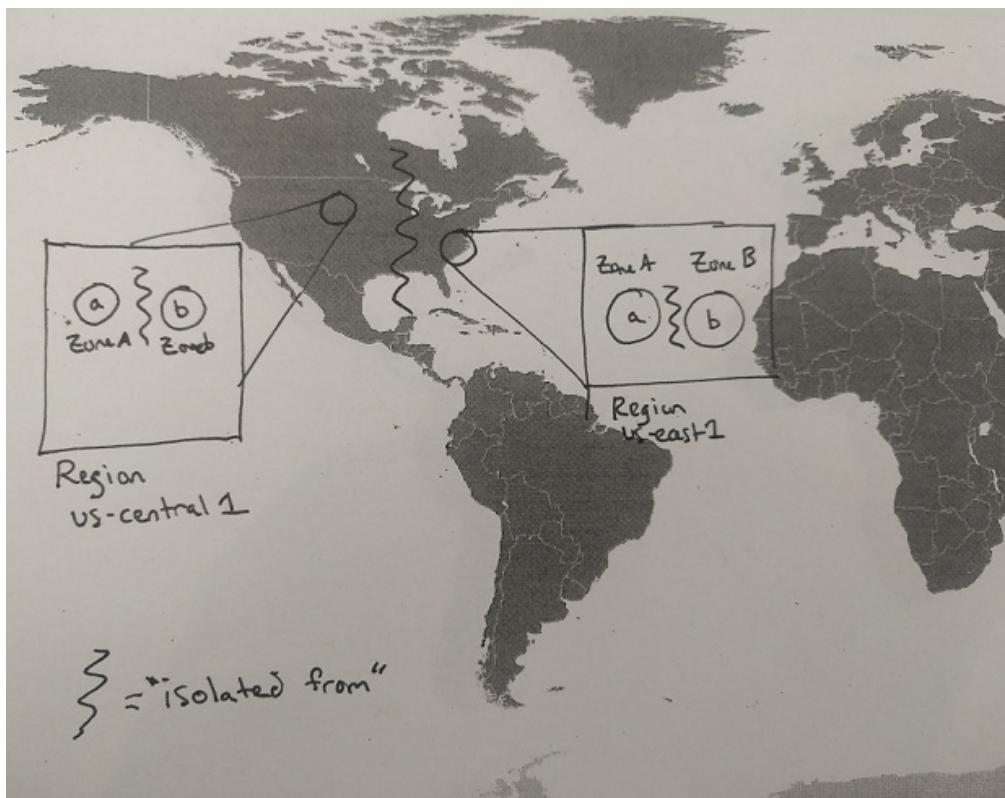
This also means that if there's some natural disaster (maybe a tornado comes through town), resources in this single Zone are likely to go offline together, as it's not likely the tornado will take down only half of a building, leaving the other half safe and untouched. More importantly, it means that if there was a malfunction (such as a power outage), it will likely affect the entire Zone. In the various APIs that take a zone (or location) as a parameter, you'll be expected to specify a Zone ID, which is a unique identifier to a particular facility and looks something like `us-east1-b`.

### 3.3.2 Regions

Moving up the stack, a collection of Zones is called a **Region**, and this corresponds loosely to a city as you saw in the table above, such as "Council Bluffs, Iowa, USA". If you turn on two resources in the same Region but different zones, say `us-east1-b` and `us-east1-c`, the resources will be somewhat close together (meaning the latency between them will be shorter than to a Zone in Asia), but they are guaranteed to **not** be in the same physical facility.

This means that while your two resources may be isolated from Zone-specific failures (like the power outage mentioned above) they may not be isolated from catastrophes (like the tornado). You might see Regions abbreviated by dropping the last letter on the Zone. For example, if the Zone is `us-central1-a`, the Region would be `us-central1`.

**Figure 3.4. Visual depiction of Regions and Zones**



### 3.3.3 Designing for fault tolerance

Now that you understand what we mean by Zones and Regions, we can talk more specifically about the different levels of isolation offered by Google Cloud. You might also hear these described as "control planes", borrowing the term from the networking world. When we refer to "isolation level" or the types of "control plane", we are really talking about what "thing" would have to go down to take your service down with it. In that sense, there are many different options:

#### **Zonal**

As we just mentioned in the example, a service that is "zonal" means that if the Zone it lives in goes down, it goes down also. This happens to be both the easiest type of service to build (all you need to do is turn on a single VM and you have a zonal service), as well as the least highly-available.

#### **Regional**

A regional service refers to something that is replicated throughout multiple Zones in a single Region. For example, if you have a MongoDB instance living in `us-east1-b`, and a hot-failover living in `us-east1-c`, you have a regional service. If one Zone goes

down, you automatically flip to the instance in the other Zone. However, if an earthquake swallows the entire city, both Zones will go down with the Region, taking your service with it. While this is unlikely, and regional services are much less likely to suffer outages, the fact that they are geographically co-located means you likely don't have enough redundancy for a mission-critical system.

#### ***Multi-regional***

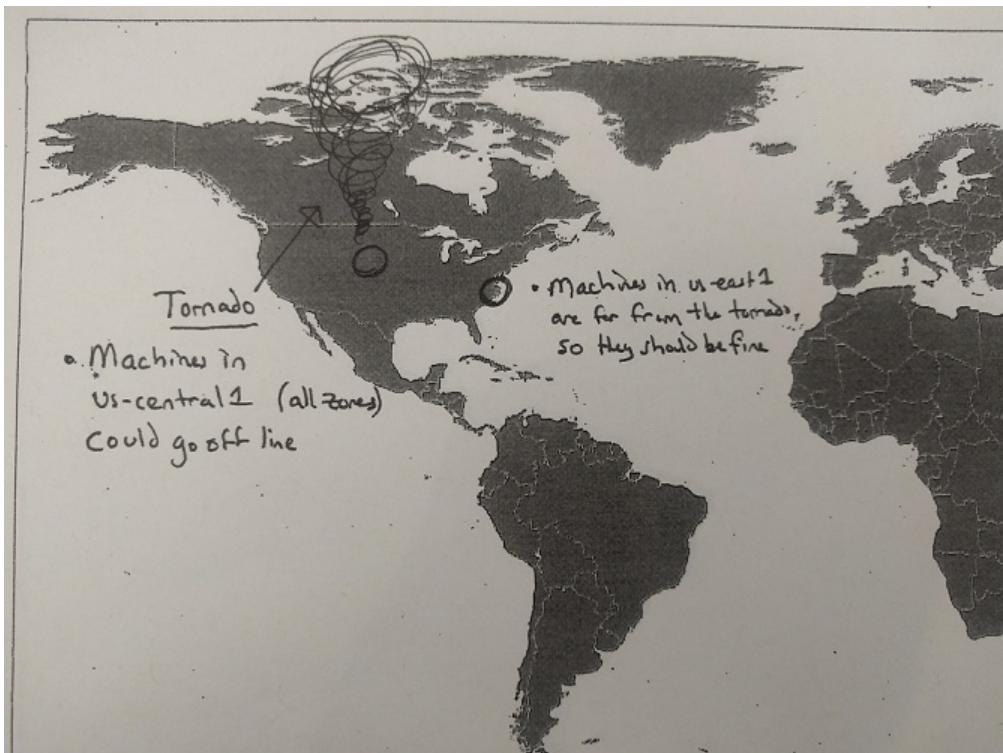
A multi-regional service is (as you might expect) just a composition of several different regional services. This means that if some sort of catastrophe occurs that takes down an entire Region, your service should still continue to run with minimal downtime.

#### ***Global***

A global service is a special case of a multi-regional service. With a global service, you typically have deployments in multiple regions, however these regions are spread across the world, crossing legal jurisdictions and network providers. Typically at this point you also want to use multiple cloud providers (for example, Amazon Web Services alongside Google Cloud) to mitigate against disasters spanning an entire company.

For most applications, regional or even zonal configurations will be secure enough. However as you become more mission-critical to your customers, you'll likely start to consider more fault-tolerant configurations, such as multi-regional or global.

**Figure 3.5. Disasters like tornadoes are likely to affect a single region at a time.**



The important thing when building your service isn't primarily configuring it in the most highly-available configuration, but knowing what your level of fault tolerance and isolation is at any point in time. If you're armed with that, you at least know, should any part of your system become absolutely critical, which pieces will need redundant deployments, and where those new resources should go. We'll talk much more about redundancy and high availability in the chapter on Compute Engine, so don't worry if you're still curious.

### 3.3.4 Automatic high availability

Over the years, certain common patterns have emerged where systems need to be "highly available". Based on these patterns, many cloud providers have designed richer systems that are automatically highly-available, meaning that you don't need to design the underlying system yourself. This means that instead of needing to build a multi-regional storage system yourself, you can rely on Google Cloud Storage, which provides the same level of fault isolation (among other things) for your basic storage needs.

There are several other examples of systems that follow this pattern such as Google Cloud Datastore, which is a multi-regional non-relational storage system, that stores your data in 5 different Zones, as well as Google App Engine, which offers two multi-

regional deployment options (one for the US and another for Europe) for your computing needs. This means that if you run an App Engine application, or save some data in Google Cloud Storage, or store records in Google Cloud Datastore, and an entire Region explodes, taking down all Zones with it, your application, data, and records will all be fine, and remain accessible to you and your customers. Pretty crazy right?

The downside with products like these is that you typically have to build things with a bit more structure. For example, when storing data on Google Cloud Datastore, you have to design your data model in a way that forces you to choose whether you want queries to always return the freshest data, or whether you want your system to be able to scale to large numbers of queries.

You can read more about this in the next few chapters, but it's important to know that while some services will require you to build your own highly-available systems, there are others that can do this for you, assuming you can manage under the restrictions they ask of you. Now that you understand fault tolerance, Regions, Zones, and all those other fun things, it's time to talk about a question that is simple yet important, and sometimes scary: is your stuff safe?

## **3.4 Safety concerns**

Over the past few years, personal and business privacy have become a mainstream topic of conversation, and for good reason. The many leaks of passwords, credit cards, and personal information mean that the online world has become far less trusting than it was in the past. This means that customers are warier of handing out things like credit card numbers or personal information due to (warranted) fears that the company holding that information will "get hacked" or a government organization will request access to the data under the latest laws to fight terrorism and increase national security. Put bluntly, putting your servers in someone else's data center typically means that you are giving up some control over your assets (such as data or source code) in exchange for other benefits (such as flexibility or lower costs). So what does this mean for you? A good way to really understand these trade-offs is to walk through them one at a time. Let's start with the security of your resources.

### **3.4.1 Security**

As you learned earlier, when you store data or turn on a computer using a cloud provider, although it's marketed as "living nowhere in particular" your resources do physically exist somewhere, in at least one place (and sometimes more). The biggest question for most people is... "where?"

If you store a photo on a hard drive in your home, you know exactly where the photo is (on your desk). Alternatively, if you upload a photo to a cloud service like Google Cloud Storage or Amazon's S3, the exact location of the data is a bit more complicated to determine, but you can at least pinpoint the region of the world of where it lives. Further, the photo is unlikely to live in only a single place. So what do you get for this trade? Is more ambiguity really worth it? When you use a cloud service to do

something like store your photos, you're not really paying for just the disk space (otherwise the fee would be a flat rate per byte rather than a recurring monthly fee), you're actually paying for quite a bit more.

To understand this in more detail, let's look at a real-life example of storing a photo on a local hard drive. By thinking about all the things that can go wrong, we can start to see how much work goes into preventing these issues, and why the solution for these results in some ambiguity about where things actually exist. After we go through all of these things, you should understand how exactly Google Cloud prevents them from happening, and have some more clarity on what it is that you get by using a cloud service over a hard drive at home.

When we talk about securing resources, our typical goals are typically three different things:

1. Privacy: Only authorized people should be able to access the resources.
2. Availability: The resources should never be "inaccessible" to authorized people.
3. Durability: The resources should never be corrupted or go missing.

Putting this into more specific terms with you and your photo, this might mean:

1. Privacy: No one besides you should be able to look at your photo.
2. Availability: You should never be told "not right now, try again later!" when you ask to look at your photo.
3. Durability: You should never come back and find your photo deleted or corrupted.

The goals seem simple enough, right? Let's look at how this breaks down with your hard drive at home "when real life happens" so to speak. The first thing that can go wrong is simple theft. For example, if someone breaks into your home and steals your hard drive, the photo you had stored on that drive is now gone. This breaks your goals for availability and durability right off the bat. If your photo wasn't encrypted at all, this also breaks the privacy goal, as the thief can now look at your photo when you don't want anyone else to be able to.

The next thing that can go wrong can be lumped into a large group called "unexpected disasters". This includes natural disaster such as earthquakes, fires, and floods, but in the case of storing data at home it also includes more common accidents such as power surges, hard drive failures, and kids spilling water on electronic equipment.

After that, you have to worry about more nuanced accidents, such as accidentally formatting the drive because you thought it was a different drive, or overwriting files that happened to have similar names. These issues are more complicated because the system is doing as it was told, you're just accidentally telling it to do the wrong thing. Finally, you have to worry about network security. If you expose your system on the internet and happen to use a weak password, it's possible that an intruder could gain access to your system and access your photo (even if you happened to encrypt the photo).

All of these types of accidents break the availability and durability goals from before, and some of them break the privacy goals. So how do cloud providers plan for these

problems? Couldn't you do this yourself? The typical way cloud providers deal with these problems comes down to a few different tactics:

#### **Secure facilities**

Any facility housing resources (like hard drives) should be a high-security area, limiting who can come and go, and what they can take with them. This is to prevent theft as well as sabotage.

#### **Encryption**

Anything stored on disks should be encrypted. This is to prevent theft compromising data privacy.

#### **Replication**

Data should be duplicated in many different places. This is to prevent a single failure resulting in lost data (durability) as well as a network outage limiting access to data (availability). This also means that a catastrophe (such as a fire) would only affect one of many copies of the data.

#### **Back-up**

Data should be backed up off-site and can be easily restored on request. This is to prevent a software bug accidentally overwriting all copies of the data. If this happens, you could just ask for "the old (correct) copy" and disregard the new (erroneous) copy.

As you might guess, to do this in your own home is not only challenging and expensive, but by definition you need more than one home! Not only would you need advanced security systems, you'd need full-time security guards, multiple network connections to each of your homes, systems that automatically duplicate data across multiple hard drives, key management systems for storing your encryption keys, and back-ups of data on rolling windows to different locations. I can comfortably say that this is definitely not something I'd want to do myself. Suddenly a few cents per Gigabyte per month doesn't sound all that bad.

### **3.4.2 Privacy**

Although we discussed encryption, we didn't really touch on privacy of your data — after all, Google Cloud Storage might keep your photo in an encrypted form, but when you ask for it back it arrives unencrypted. How can that be? The truth here is that while data is stored in encrypted form and transferred between data centers similarly, when you ask for your data, Google Cloud does have the encryption key and uses it when you ask for your photo. This also means that if Google were to receive a court order, they do have the technical ability to comply with it and decrypt your data without your consent.

To provide added security, many cloud services provide the ability to use your own encryption keys, meaning that the best Google can do is hand over encrypted data as they don't actually have the keys to decrypt it. If you're interested in more details about this topic, ou can learn more about this in the chapter on Google Cloud Storage.

### 3.4.3 Special cases

Sometimes you have special situations that require heightened levels of security or privacy. For example:

- Government agencies typically have very strict requirements
- Companies in the health care industry typically must comply with HIPAA regulations
- Companies dealing with personal data of German citizens typically must comply with the German BDSG.

For these cases, cloud providers have come up with a few different options:

- Amazon offers GovCloud to allow government agencies to use AWS.
- Google, Azure, and AWS will all sign BAAs to support HIPAA covered customers.
- Azure and Amazon offer data centers in Germany to comply with BDSG.

Each of these cases can be quite nuanced so if you're in one of these situations, you should know:

1. It is still possible to use cloud hosting
2. You may be slightly limited on which services you can use

You're probably best off involving legal counsel when making these kinds of very serious decisions about hosting providers. All that said, hopefully you're now relatively convinced that cloud data centers are safe enough for your typical needs, and open to exploring cloud data centers for your special needs. But we still haven't touched on the idea of sharing these data centers with all the other people out there. How does that work?

## 3.5 Resource isolation and performance

As you learned in [chapter 1](#), the big breakthrough that opened the door to cloud computing was the concept of virtualization, or breaking a single physical computer into smaller pieces, each one able to act like a computer of its own. What made cloud amazing was the simple fact that you could build a large cluster of physical computers, and then lease out a smaller virtual ones by the hour. This process would be profitable so long as the average cost to run the physical computers was covered by the smaller leases of the virtual ones.

This concept is fascinating, but it omits one important thing: do two virtual "half computers" run as fast as one physical "whole computer"? This leads to further questions such as whether two people, each using a virtual "half computer", can spill over into the resources of each other. In other words, could one person run a CPU-intensive workload and effectively "steal" some of the CPU cycles from the other person? What about network bandwidth? Or memory? Or disk access? This issue has come to be known as the "noisy neighbor problem" and is something everyone running inside a cloud data center should understand, even if just superficially.

**Figure 3.6. The noisy neighbor problem**



The short answer to this is:

You'll only get perfect resource isolation on "bare metal" (non virtualized) machines.

Luckily, many of the cloud providers today have known about this problem for quite a long time and therefore have spent years building solutions to it. Although, as mentioned above, there is likely no **perfect** solution, many of the preventative measures can be quite good to the point where fluctuations in performance might not even be noticeable.

In Google's case, all of the cloud services ultimately run on top of a system called "Borg", which as you can read in Wired Magazine, "is a way of efficiently parceling work across Google's vast fleet of ... servers". What this means is that because Google uses the same system internally for other services (such as Gmail and YouTube), resource isolation (or perhaps better phrased as "resource fairness") is a feature that has almost a decade of work behind it, and is constantly improving. More concretely, what this means for you is: if you purchase 1 vCPU worth of capacity on Google Compute Engine, you should get the same number of computing cycles, regardless of how much work other VMs are trying to do.

## 3.6 Summary

- Google Cloud has lots of data centers in lots of locations around the world for you to choose from.
- The speed of light is the limiting factor in latency between data centers, so consider that distance when choosing where to run your workloads.
- When designing for high-availability, always use multiple zones to avoid zone-level failures, and if possible multiple regions to avoid regional failures.
- Google's data centers are incredibly secure and services encrypt data before storing it.
- If you have special legal issues to consider (HIPAA, BDSG, etc), check with a lawyer before storing information in any cloud provider.

# Part 2

## Storage

Now that we have a better understanding of the fundamentals of "the cloud", it's time to start digging deeper into individual products. To kick things off, we'll begin by exploring the very diverse world of storing data.

Let's start by getting something out of the way: storing data tends to sound boring, like something that should "just be solved by now". In truth, when you get into the details, storing data is actually super complicated. As with anything deceptively complicated, this means that it can be really fascinating if you take the time to explore it properly.

In the following chapters we'll look at a variety of different storage systems and how they work in Google Cloud Platform. Some of these should be pretty familiar (e.g., [chapter 4](#)), while others were invented by Google and come with lots of new things to learn (e.g., [chapter 6](#)), but each of these options comes with a unique set of benefits and drawbacks. When you've finished this part of the book, you should have a great grasp on the various storage options available, and hopefully a clear choice of which is the best fit for your project.

# *Cloud SQL: Managed relational storage*

## **This chapter covers:**

- What is a relational database?
- What is Cloud SQL?
- Turning on a Cloud SQL instance
- Configuring a production-grade SQL instance
- Backing-up and restoring your data
- Deciding whether Cloud SQL is a good fit
- Choosing between Cloud SQL vs MySQL on a VM

## **4.1 What is a relational database?**

Relational databases, sometimes called SQL databases (pronounced like "sequel"), are one of the oldest forms of structured storage going back to the 1980s. The name "relational database" comes from the idea that these databases store related data and then allow us to combine this related data to ask complex questions, such as, "How old are this year's top 5 highest-paid employees?"

This ability makes relational databases great general-purpose storage systems, and as a result most cloud hosting providers offer some sort of "push-button" option to get a relational database up and running. In Google Cloud, this is called Cloud SQL, and if you went through the exercise in [chapter 2](#), you're already a little bit familiar with it.

In this chapter, we'll walk through Cloud SQL in much more detail and cover more real-life situations. Obviously entire books can be (and have been) written on various flavors of relational databases (such as MySQL or PostgreSQL) so if you decide to use

Cloud SQL in production, a book on MySQL is a great investment. The goal of this chapter is not to duplicate any information you'd find in books like those, but to highlight the things that Cloud SQL does differently as well as all the neat features that automate some of the administrative aspects of running your own relational database server.

## 4.2 What is Cloud SQL?

Cloud SQL is a VM that is hosted on Google Compute Engine, managed by Google, running a version of the MySQL binary. This means that you get a perfectly compatible MySQL server that you don't ever have to SSH into to tweak settings, as all of those can be changed in the Cloud Console, the Cloud SDK command-line tool, or the REST API. If you're familiar with Amazon's Relational Database Service (RDS), you can think of Cloud SQL as basically the same thing. And while Cloud SQL currently supports both MySQL and PostgreSQL, we'll only discuss MySQL here for now.

Since Cloud SQL is perfectly compatible with MySQL, this means that if you currently use MySQL anywhere in your current system, Cloud SQL is a viable option for you. It also means that "integrating" with Cloud SQL is nothing more than changing the hostname in your configuration to point at a Cloud SQL instance.

This also means that configuration and performance tuning will be **identical** for Cloud SQL and your own MySQL server so we won't get into those topics. Instead this chapter will explain how Cloud SQL automates some of the more tedious tasks like upgrading to a newer version of MySQL, recurring back-ups, and securing your Cloud SQL instance so that it only accepts connections from people you trust.

To kick things off, let's run through the process of turning on a Cloud SQL instance.

## 4.3 Interacting with Cloud SQL

As you learned in [chapter 1](#), there are many different ways to interact with Google Cloud (in the browser with the Cloud Console, on the command-line with the Cloud SDK, and from inside your own code using a client library for your language). This walk-through will use a combination of the Cloud Console and the Cloud SDK to turn on a Cloud SQL instance and talk to it from your local machine. More specifically, we're going to store our To-do list data in Cloud SQL and run a few example queries.

Start by jumping over to the SQL section of the Cloud Console in your browser ([cloud.google.com/console](https://cloud.google.com/console)). Once there, click on the button to create a new instance (which is analogous to a "server" in regular MySQL-speak).

When filling out the form, make sure to pick a region that is nearby, so that your queries won't be traveling across the world and back. In this example we'll create the instance in `us-east1`. Once you click "Create", Google will get to work setting up your Cloud SQL instance.

**Figure 4.1. Create a new Cloud SQL instance with our "non-requirements"**

The screenshot shows the 'Create a MySQL Second Generation instance' page. At the top, there's a 'SQL' tab and a back arrow. The main area has fields for 'Instance ID' (set to 'todo-list'), 'Root password' (set to 'very-long-root-password'), and 'Location' (set to 'us-east1'). On the right, there's a summary box showing 'Estimated monthly total: \$51.89' and 'Hourly rate: \$0.071'. Below the summary is a 'Details' section with a 'Show configuration options' link. At the bottom are 'Create' and 'Cancel' buttons.

SQL    [Create a MySQL Second Generation instance](#)

**Instance ID**  
Cannot be changed later. Use lowercase letters, numbers, and hyphens. Start with a letter.

**Root password**  
Set a password for the root user. [Learn more](#)  
 [Generate](#)

No password

**Location** ⓘ  
For better performance, keep your data close to the services that need it.

<b>Region</b>	<b>Zone</b>
us-east1	Any

[Show configuration options](#)

**Create** **Cancel**

Before talking to our database we need to make sure we have access. MySQL uses password authentication, so to grant additional access all we have to do is create new users. We can do this inside the Cloud Console by clicking on the Cloud SQL instance, choosing the "Users" tab.

**Figure 4.2. The Access Control section**

The screenshot shows the 'USERS' tab selected in the MySQL instance details page. The table contains the following data:

User name	Host name	⋮
root	% (any host)	⋮

Here you can create a new user or change the root user's password but keep track of the username and password that you end up creating. There's also a lot of other things we can do, but we'll get into these in more detail later on.

After you have a user created it's time to switch environments completely, from the browser over to the command-line. Open up a terminal and we'll start by seeing if we can see our Cloud SQL instance using the `instances list` command that lives in `gcloud sql`:

#### **Listing 4.1. Listing our Cloud SQL instances**

```
$ gcloud sql instances list
NAME      REGION    TIER          ADDRESS      STATUS
todo-list  us-east1  db-n1-standard-1  104.196.23.32  RUNNABLE
```

Now that we're sure our Cloud SQL instance is up and running (note the `STATUS` field showing us that it's `RUNNABLE`), let's try connecting to it using the MySQL command-line interface:

#### **Listing 4.2. Connecting to our Cloud SQL instance**

```
$ sudo apt-get install mysql-client
...
$ mysql -u user-here --password=password-here -h 104.196.23.32  ①
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 37
Server version: 5.6.25-google (Google)
```

```
Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights reserved.
```

```
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

```
mysql>
```

- 1 Make sure to substitute your username and password as well as the host IP of your instance.

Looks like everything worked! Notice that you're talking to a real MySQL binary, so any command you can run against MySQL in general will work on this server.

The first thing we have to do is create a database for our app, which we can do using the `CREATE DATABASE` command.

#### **Listing 4.3. Create the todo database**

```
mysql> CREATE DATABASE todo;
Query OK, 1 row affected (0.02 sec)
```

Now let's create a few tables for our to-do list. If you're not familiar with relational database schema design, don't worry — nothing here is super advanced.

First, we'll create a table to store our To-do lists, which will look something like this:

**Table 4.1. To-do lists table (todolists)**

ID (primary key)	Name
1	Groceries
2	Christmas shopping
3	Vacation plans

This translates into the following MySQL schema:

#### **Listing 4.4. Defining the todolists table**

```
CREATE TABLE `todolists` (
  `id` INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `name` VARCHAR(255) NOT NULL
) ENGINE = InnoDB;
```

Let's run that against our database, making sure we are running it against the database we created.

#### **Listing 4.5. Creating the todolists table in our database**

```
mysql> use todo;
Database changed
```

```
mysql> CREATE TABLE `todolists` (
->   `id` INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
->   `name` VARCHAR(255) NOT NULL
-> ) ENGINE = InnoDB;
Query OK, 0 rows affected (0.04 sec)
```

Let's create those example lists we mentioned above so we can see how things work.

#### **Listing 4.6. Adding some sample to-do lists**

```
mysql> INSERT INTO todolists (`name`) VALUES ("Groceries"),
->      ("Christmas shopping"),
->      ("Vacation plans");
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

And we can use a SELECT query to check if the lists are there.

#### **Listing 4.7. Looking up our to-do lists**

```
mysql> SELECT * FROM todolists;
+----+-----+
| id | name        |
+----+-----+
|  1 | Groceries   |
|  2 | Christmas shopping |
|  3 | Vacation plans |
+----+-----+
3 rows in set (0.02 sec)
```

Lastly, let's do this again, but this time for to-do items for each checklist. Our example data will look something like:

**Table 4.2. To-do items table (todoitems)**

ID (primary key)	To-do list ID (foreign key)	Name	Done?
1	1 (Groceries)	Milk	No
2	1 (Groceries)	Eggs	No
3	1 (Groceries)	Orange juice	Yes
4	1 (Groceries)	Egg salad	No

Which translates into the following MySQL schema:

#### **Listing 4.8. Creating the todoitems table**

```
> CREATE TABLE `todoitems` (
->   `id` INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
->   `todolist_id` INT(11) NOT NULL REFERENCES `todolists`.`id`,
->   `name` varchar(255) NOT NULL,
->   `done` BOOL NOT NULL DEFAULT '0'
-> ) ENGINE = InnoDB;
```

```
Query OK, 0 rows affected (0.03 sec)
```

And then we can add the example to-do items:

#### **Listing 4.9. Adding example items to the todoitems table**

```
mysql> INSERT INTO todoitems (`todolist_id`, `name`, `done`) VALUES
->     (1, "Milk", 0), (1, "Eggs", 0), (1, "Orange juice", 1),
->     (1, "Egg salad", 0);
Query OK, 4 rows affected (0.03 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

And now we can do things like ask for all the groceries that we still have to buy, that sound like "egg":

#### **Listing 4.10. Querying for groceries left to buy that sound like "egg"**

```
mysql> SELECT `todoitems`.`name` from `todoitems`, `todolists` WHERE
->   `todolists`.`name` = "Groceries" AND
->   `todoitems`.`todolist_id` = `todolists`.`id` AND
->   `todoitems`.`done` = 0 AND
->   SOUNDEX(`todoitems`.`name`) LIKE CONCAT(SUBSTRING(SOUNDEX("egg"), 1, 2),
"%");
+-----+
| name |
+-----+
| Eggs |
| Egg salad |
+-----+
2 rows in set (0.02 sec)
```

We'll continue to reference this example database throughout the chapter, but since you'll be paying for this Cloud SQL instance every hour it stays on, feel free to delete and recreate instances as you need.

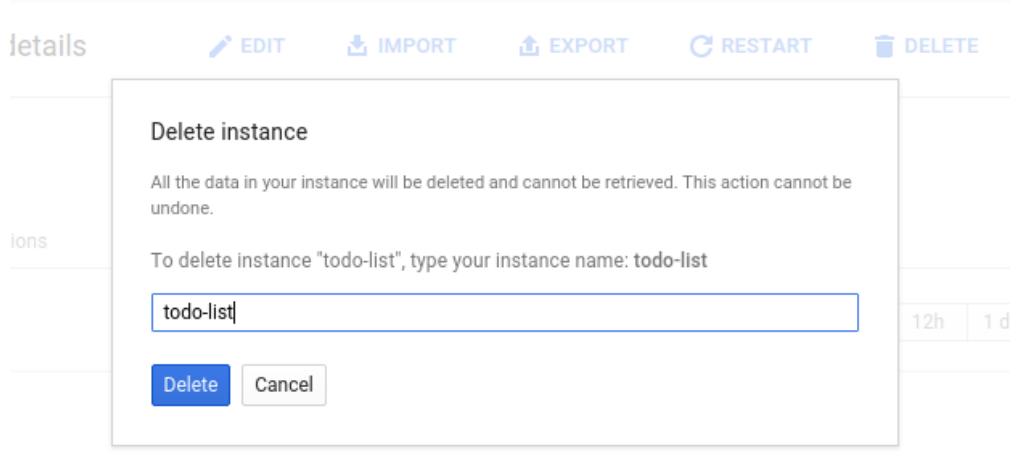
To delete a Cloud SQL instance, click on the "Delete" button in the Cloud Console.

**Figure 4.3. Deleting your Cloud SQL instance**

The screenshot shows the 'Instance details' section for a MySQL Second Generation master instance named 'wordpress-db'. Below the instance name, it says 'MySQL Second Generation master'. To the right of the instance name are several buttons: EDIT, IMPORT, EXPORT, RESTART, STOP, DELETE, and CLONE. The 'DELETE' button is highlighted with a red box.

After that, you'll need to confirm you're deleting the right database (we wouldn't want you deleting the wrong database accidentally!).

**Figure 4.4. Confirming the instance you meant to delete**



Now that you've seen how to work with Cloud SQL (and hopefully if you've used MySQL before, you're feeling right at home), let's look at some of the things we'll need to do to set up a Cloud SQL instance for real-life work.

## 4.4 Configuring Cloud SQL for production

Now that we've learned how to turn on a Cloud SQL instance, it's time to go through what it takes to run Cloud SQL in a production-grade environment.

Before we continue, it might be worthwhile to clarify that for the purposes of this chapter (and most of this book), when we say "production" we mean the environment that is safe for you to run a business. This means that we would have things like reliable back-ups, fail-over procedures, and proper security practices.

Now let's jump in by looking at one of the most obvious topics: access control.

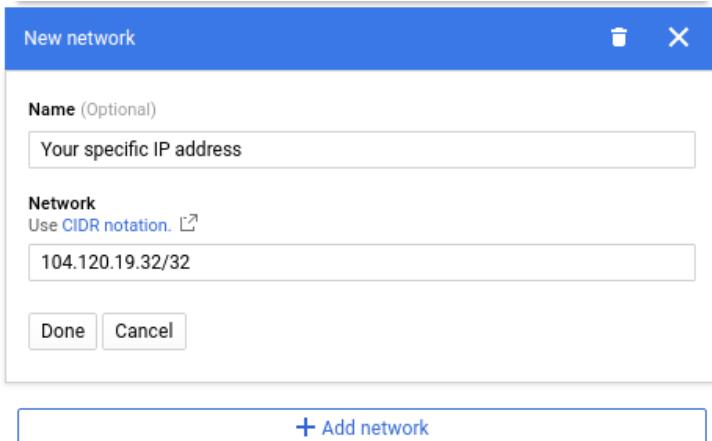
### 4.4.1 Access control

In some scenarios (e.g., kicking the tires on a new tool) it might make sense to temporarily ignore security. For example, we might allow open access to a Cloud SQL instance if it was just a toy that we intended to turn off later on (e.g., `0.0.0.0/0` in CIDR notation), but as things get more serious this is certainly not acceptable. This begs the question: what *is* acceptable? What IP addresses or sub-networks should we allow to connect to an instance?

If your system is spread out across many providers (e.g., maybe you have some VMs running in Amazon's EC2, some in Microsoft's Azure, and some in Google Compute Engine), the simplest thing to do is assign a static IP to these machines and then specifically limit access to just those in the "Authorization" section when looking at

Cloud SQL instance. For example, if you have a VM running using the IP address 104.120.19.32, you could allow access from that exact IP using CIDR notation, which would be 104.120.19.32/32 (the /32 here means "this must be an exact match"). These types of limits happen at the network level, which means that MySQL won't even hear about these requests coming in. This is a good thing in that unless you've allowed access to an IP, your database appears completely invisible.

**Figure 4.5. Setting access to a specific IP address**



If you have a relatively large system, adding lots and lots of IP addresses to the list of "who has access" could get really tedious. To deal with this, we can rely on the pattern of IP addresses and CIDR notation. Inside Compute Engine your VMs live on a "virtual network" that assigns IPs from a special subnet just for your project (for a more in-depth discussion on networking, see [chapter 9](#)). This means that by default all of your Compute Engine VMs on a single network will all have IP address following the same pattern, and we can grant access to *the pattern* rather than each individual IP address.

For example, the default network uses a special subnet for assigning internal IP addresses (10.240.0.0/16) which means that your machines will all have IPs matching this CIDR expression (e.g., 10.240.0.1). In other words, to limit access to just these machines, we can simply use 10.240.0.0/16.

The next type of security that often comes up is using an encrypted channel for your queries. Luckily, Cloud SQL makes it pretty easy to use SSL for your transport.

#### 4.4.2 Connecting over SSL

If you're new to this area, SSL ("Secure Sockets Layer") is nothing more than a standard way of sending data from point A to B over an untrusted wire. In other words, SSL provides a way to safely send sensitive information (like your credit card numbers) over a connection that someone could be listening in on.

This is really important because most of the time we think of SSL as just a thing for websites but if you securely send your credit card number to a web server, and the web server insecurely sends that credit number to a database, you still have a pretty big problem. So how do we make sure the connection to our databases is encrypted?

Whenever you're establishing a secure connection as a client, there are three things you need:

1. The server's CA certificate,
2. A client certificate, and
3. A client private key.

Once you have those, the MySQL client knows what to do with them to establish a secure connection, so there's not much else you'll need to do.

To get a hold of these, start off by viewing your instance in the Cloud Console, and jump into the "SSL" tab.

**Figure 4.6. Cloud SQL's SSL options**

The screenshot shows the 'Instance details' page for a MySQL Second Generation master instance named 'wordpress-db'. The 'SSL' tab is selected. A warning message states: 'Unsecured connections are allowed to connect to this instance.' It also notes that the server certificate expires on Feb 5, 2020, at 11:23:58 AM. Buttons for 'View Server CA Certificate' and 'Reset SSL Configuration' are visible. A section for 'Client Certificates' includes a 'Create a Client Certificate' button.

**SSL Connections**  
For security, it is recommended to always use SSL encryption when connecting to your instance. For more information, see [Configuring SSL](#).

**SSL Configuration**  
The server Certificate Authority (CA) certificate is required in SSL connections. Resetting the SSL configuration of the server revokes all client certificates and creates a new server CA certificate.

⚠️ Unsecured connections are allowed to connect to this instance.

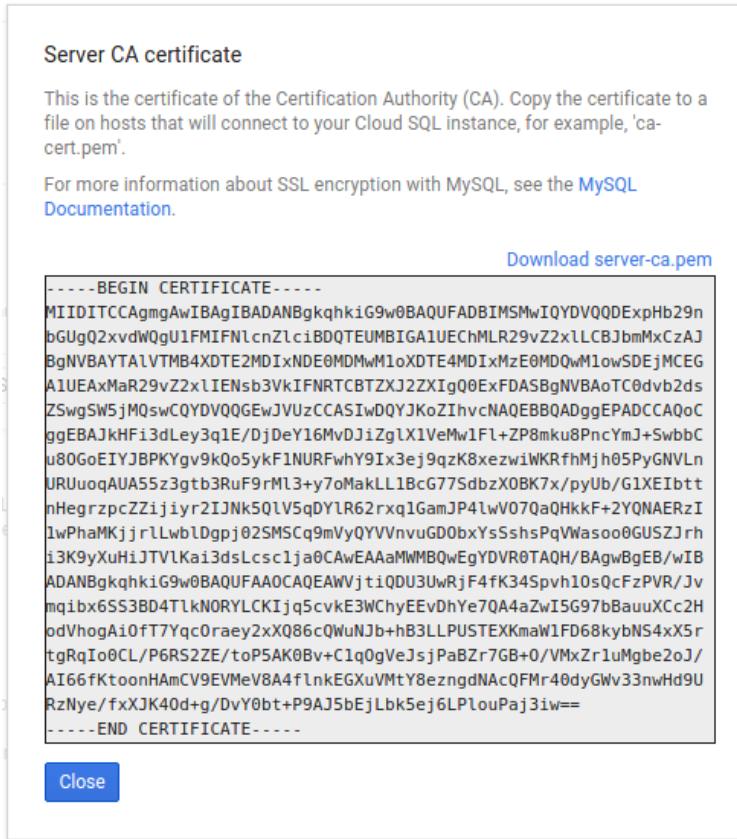
Your server certificate expires on Feb 5, 2020, 11:23:58 AM  
To issue a new server certificate, reset the SSL configuration.

[View Server CA Certificate](#) [Reset SSL Configuration](#)

**Create a Client Certificate**

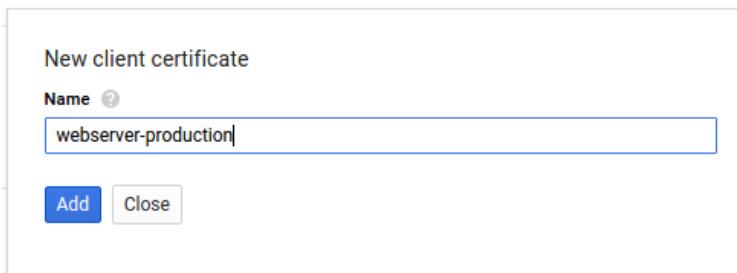
To get the server's CA certificate, click the aptly named button "View Server CA Certificate". You'll see a pop-up appear and you can either copy-and-paste the certificate, or download it as `server-ca.pem` using the link right above the text box.

**Figure 4.7. Cloud SQL's Server CA Certificate**



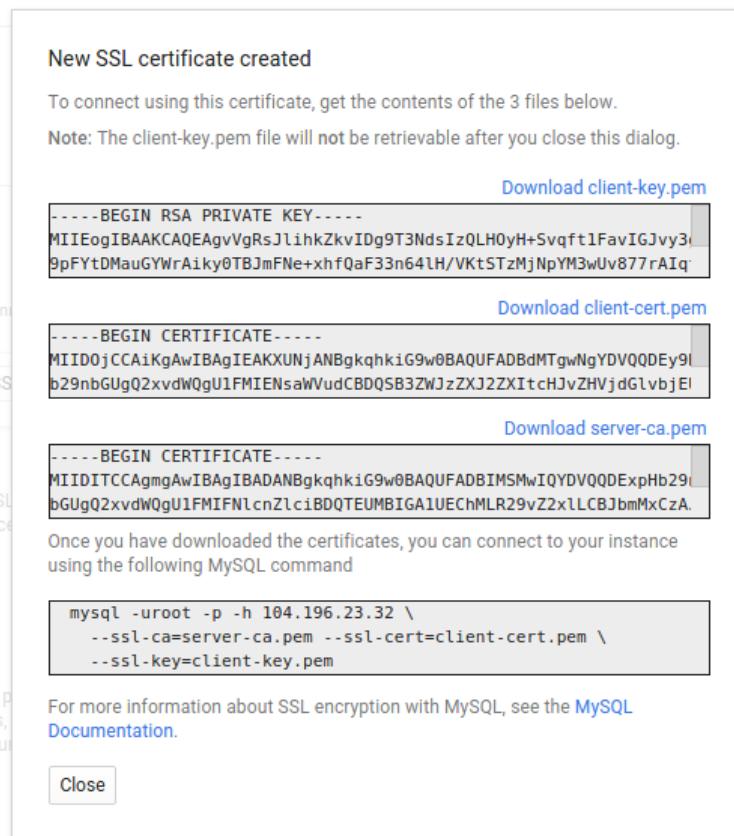
After that, we need to get the client certificate and private key. To do this, click the "Create a Client Certificate" button, and type in a name for your certificate. Typically you would name the certificate after the server that is using it to access your database. For example, if this certificate will be used on your production web servers to read and write to the database, you might call it "webserver-production".

**Figure 4.8. Creating a new client certificate**



Once you click "Add", you'll see a second pop-up showing the client certificate and private key. Just like before, you can either copy-and-paste or click the download links, but at the end of this you should now have both `client-key.pem` and `client-cert.pem`.

**Figure 4.9. Certificate created and ready to use**



### WARNING

While you can come back later to get `server-ca.pem` and `client-cert.pem` files if you happen to lose them, you **cannot** get the `client-key.pem` file if you lose it. If you do lose it, you'll just need to create a new certificate.

Once you have all of these, you can things out by running the command show on the pop-up.

### Listing 4.11. Connecting to Cloud SQL over SSL

```
$ mysql -u root --password=really-strong-root-password -h 104.196.23.32 \
--ssl-ca=server-ca.pem \
--ssl-cert=client-cert.pem \
--ssl-key=client-key.pem
```

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 646
Server version: 5.6.25-google (Google)

Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

To double check that your connection is actually encrypted, you can use MySQL's `SHOW STATUS` command.

#### **Listing 4.12. Verifying that a connection is secure**

```
mysql> SHOW STATUS LIKE 'Ssl_cipher';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Ssl_cipher    | DHE-RSA-AES256-SHA |
+-----+-----+
1 row in set (0.02 sec)
```

Notice that if you run this query over an insecure connection, the result is totally different.

#### **Listing 4.13. Insecure MySQL connections show that no cipher is in use**

```
mysql> SHOW STATUS LIKE 'Ssl_cipher';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Ssl_cipher    |      |
+-----+-----+
1 row in set (0.01 sec)
```

With these three files, you should be able to connect securely to your Cloud SQL instance from most client libraries, as the major ones know what to do with these. For example, if you use the `mysql` library for Node.js, you can simple pass in a `key`, `cert`, and `ca`.

#### **Listing 4.14. Connecting to MySQL from Node.js**

```
const fs = require('fs');
const mysql = require('mysql');

const connection = mysql.createConnection({
  host: '104.196.23.32',
  ssl: {
```

```

    ca: fs.readFileSync(__dirname + '/server-ca.pem'),
    cert: fs.readFileSync(__dirname + '/client-cert.pem'),
    key: fs.readFileSync(__dirname + '/client-key.pem')
  }
});
```

Now that we've gone through quite a bit about securing your Cloud SQL instance, let's talk a bit more in detail about the various configuration options and what they mean when you're trying to run a "production" database.

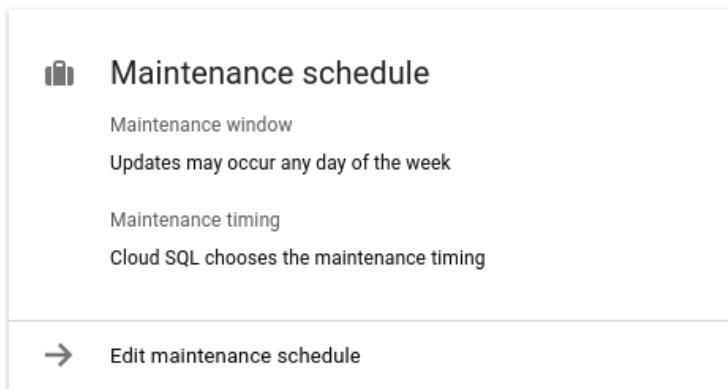
#### 4.4.3 Maintenance windows

One area we all tend to forget about during development time is the idea that our servers can live forever without any maintenance like security patches or upgrades to newer versions. Luckily this is one of the things that Cloud SQL does for you.

This also means that you might want to tell Google when it's OK to do things like system upgrades so that your customers don't notice the database disappearing or getting slower in the middle of the day. Cloud SQL lets you set a specific day of the week and time of the day (in 1-hour windows) that are "acceptable" windows for maintenance to be done. You need to set these because, obviously, Google doesn't know what your business is, and the maintenance window is probably very different for apps like E\*Exchange (where late at night on the weekends is a good time for maintenance) versus apps like InstaSnap (where slightly early morning on weekdays is a good time for maintenance).

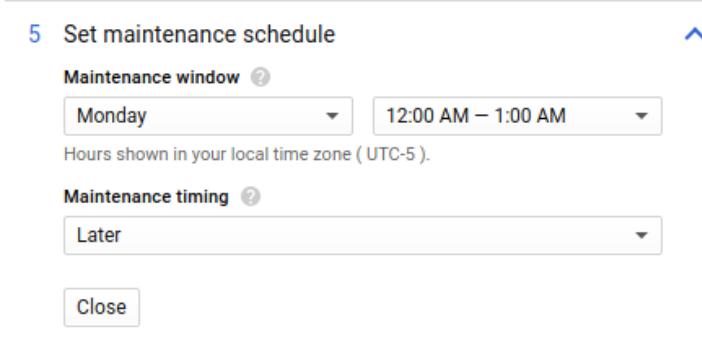
To set this window, jump over to the Cloud Console to your Cloud SQL instance's details page, and towards the bottom you'll see a "Maintenance schedule" section, with a link to edit the schedule.

**Figure 4.10. Cloud SQL instance details page with a maintenance schedule card**



On the editing page, you'll notice a section called Maintenance window, which may have been left as "Any window" (which tells Google that it's OK to perform maintenance on your Cloud SQL instance at any time on any day)—which is unlikely to be what you want!

**Figure 4.11. Choosing a maintenance window**



First, start by picking a day of the week. Typically, for working-hours business apps, the best days for maintenance are weekends, whereas for social or "just for fun" apps, the best days are weekdays early in the week (Mondays or Tuesdays).

After you pick a day, you can pick a single hour window that works for you. Keep in mind that this time is in your **local** timezone, not UTC, so if you are in New York (as I am), 8:00 AM means 8:00 AM Eastern time, which is either 12:00 or 13:00 UTC depending on the time of year (this difference is due to daylight savings time).

This works well if you are located near to your customers, but makes things a bit tricky if you're not in the same timezone. For example, if you were based in New York (GMT-5) but you were building E\*Exchange for customers in Tokyo (GMT+9), you should **add** 14 hours to the time, which could even change the day you pick (as 3:00 AM on Saturday in Tokyo, is actually 1:00 PM on Friday in New York).

The last option allows you to choose whether you want updates to arrive earlier or later in the release cycle. Earlier timing means that your instance will be upgraded as soon as the newest version is considered stable, whereas setting this to "later" will delay the upgrade for a while. In general, only choose "earlier" if you're dealing with test instances.

So this let's you configure when you want updates, but what about when you want to tweak MySQL's configuration parameters?

#### 4.4.4 Extra MySQL options

If you were managing your own VM and running MySQL, you'd have full control over all the configuration parameters by changing settings in the MySQL configuration file (`my.cnf`). In Cloud SQL, you obviously don't have access to the `my.cnf` file, but you can still change these parameters — just via an API (or via the Cloud Console) rather than a configuration file.

As discussed before, the tuning MySQL for maximum performance is an enormous topic, so if you're interested in getting the most from your Cloud SQL (or MySQL) database, you may want to pick up a copy of *High Performance MySQL* (a classic

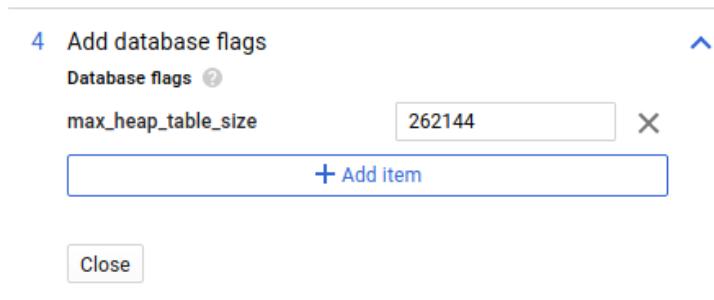
O'Reilly book on the topic). The purpose of this section is really just to clarify how you would set all of the parameters on Cloud SQL as you would on your own MySQL database.

As an example, let's say that you are creating really large in-memory temporary tables. By default, there's a limit to how big those in-memory tables can be which is 16 MB. If you end up going past that limit, MySQL automatically converts that in-memory table to an on-disk MyISAM table. If you knew you have more than enough memory (e.g., you're running with 104 GB of RAM) and you're often going past this limit, you may find that you get better performance by raising the limit from 16 MB up to something more in-line with your system, say 256 MB.

Typically, you'd do this by editing `my.cnf` on your MySQL server. To do this with Cloud SQL, you can use the Cloud Console.

Just click the "Edit" button again on the Cloud SQL instance details page, and choose "Add database flags" from the configuration options section. In this section, you can choose from a bunch of MySQL configuration flags, and set custom values for these options.

**Figure 4.12. Changing the `max_heap_table_size` for our Cloud SQL instance**



In our case, we want to change the `max_heap_table_size` to 256 MB (262144 KB). Once we've set the value, clicking "Save" will update the parameter.

You should be able to change almost any of the configuration options you'd see in `my.cnf` with a few exceptions related to where your data lives, SSL certificate locations, and other things like that which are carefully managed by Cloud SQL.

## 4.5 Scaling up (and down)

You read earlier that there's nothing wrong with starting out on a small VM type (maybe a single-core VM) and then moving to a larger, more powerful VM later on.

But how does that work? The answer is actually so simple that it might surprise you.

First, remember that there are two different things that go into determining the performance of your Cloud SQL instance:

1. Computing power (e.g., the VM instance type)
2. Disk performance (e.g., the size of the disk since size and performance are tied)

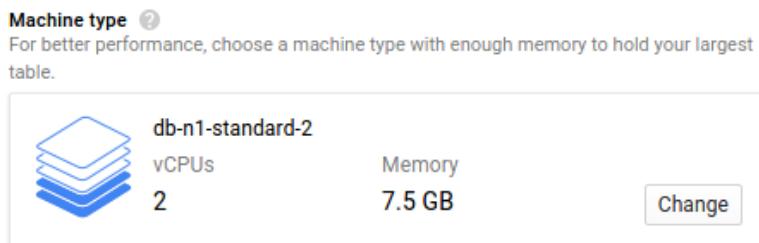
Let's start by discussing changing the amount of computing power behind your Cloud SQL instance.

#### 4.5.1 Compute power

Start back at the Cloud SQL instance details page, and click the "Edit" button at the top.

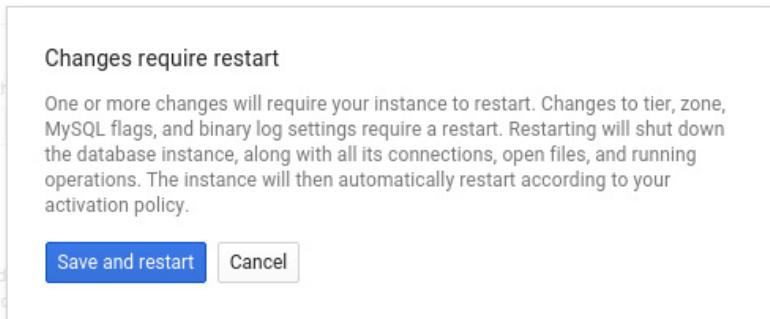
Once you're there, you'll notice that the same way you chose a VM type when you created your Cloud SQL instance, you can choose this again. This means that if you started with a single-core machine (`db-n1-standard-1`), you can just change the Machine type to a larger machine (e.g., `db-n1-standard-2`) and click save.

**Figure 4.13. Changing the machine type**



When you click "Save", you'll have to restart your database, so there is a little bit of downtime (typically just a few minutes), but that's all you have to do. When your database comes back up, it will be running on the larger (or smaller) machine type.

**Figure 4.14. Changing the machine type requires a restart**



Now that you have a bigger machine, what about disk performance? Or—even worse—what if you're running low on disk space?

#### 4.5.2 Storage

As we learned in [chapter 1](#), disk size and performance are tied together. That means

that a larger disk not only can store more bytes, but it provides more IOPS to access those bytes. For that reason, if you only plan to have 10 GB of data, but you plan to access it heavily you might want to actually allocate far more than 10 GB. You can read all about this in [chapter 9](#), but the key thing to remember here is that you may find yourself in a situation where you are running low on disk space, or your data, while not growing in size, is being accessed more frequently, and therefore needs more IOPS capacity.

In either situation, the answer is the same: make your disk bigger.

By default, disks used as part of Cloud SQL have "automatic growth" enabled. This means that as your disk gets full Cloud SQL will automatically increase the size available. If, however, you want to grow a mostly empty disk to increase the performance aspects, doing this is a pretty simple process which starts with the "Edit" button once again.

On the "Edit instance" page, under the "Configuration options", you should see a section called "Configure machine type and storage". Inside there, the "Storage capacity" section is free for you to change, so increasing the size (and performance) of your disk is as easy as changing the number in the text box to your target size.

**Figure 4.15. Changing the disk size**



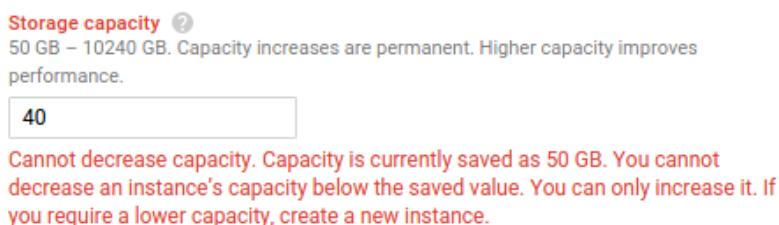
Note that this change doesn't require a restart of your database server, so your new disk space (and therefore disk performance) should be available almost instantaneously.

#### WARNING

Note that you can **increase** the size of your database, but **cannot** decrease it. If you try to make the available storage smaller, regardless of how much space you've actually used, you'll get an error saying that you can't do that.

Keep that in mind when you change your disk size, as going backwards is extra work.

**Figure 4.16. Disk size can only increase**



This explains how to scale your Cloud SQL instance up and down, but what about high

availability? Let's look at how you can use Cloud SQL to make sure your database stays running even in the face of accidents and other disasters.

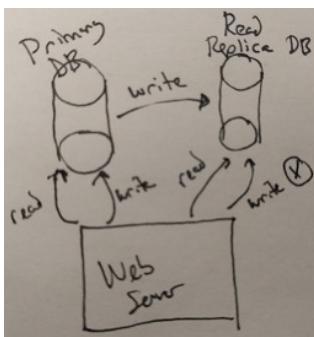
## 4.6 Replication

A fundamental component to designing highly available systems is removing any single points of failure, with the goal being that your system continues running without any service interruptions even as many parts of your system fail (and usually in new and novel ways every time). As you might have guessed, having a single database server is (by definition) a single point of failure, as a database crash (which can happen with no notice at all) would mean that your system is no longer functioning as intended.

The good news is that Cloud SQL makes it really easy to implement the most basic forms of replication. This is done by providing two different push-button replica types: read replicas, and failover replicas.

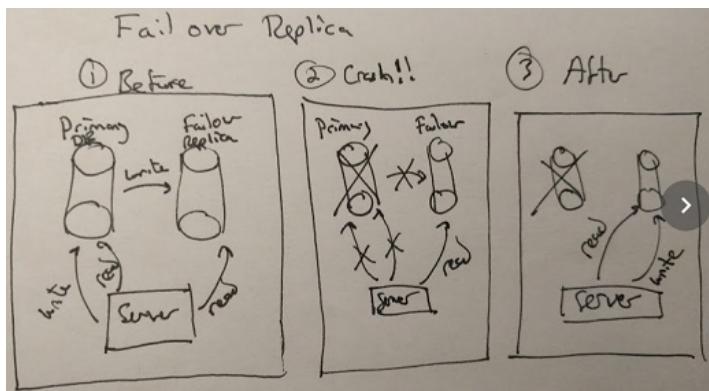
A read replica is a clone of your Cloud SQL instance that follows the "primary" or "master" instance, pulling in any changes made to the master. The read replica is strictly read-only, which means that any queries that modify data (such as `INSERT` or `UPDATE` queries) will be rejected by the read replica. As a result, read replicas are really useful when your application does a lot more reads than writes, because you can turn on a bunch of read replicas and route some of the read-only traffic to those instances. In effect, these instances allow you to scale horizontally (where you add more instances as a way of increasing capacity) rather than just vertically (where you make your machine bigger to increase capacity).

**Figure 4.17. Read replicas follow the primary database**



A failover replica is similar to a read replica except its primary job is to be ready as a replacement primary instance in case of some sort of disaster. In other words, you can think of a failover replica like an alternate in football, ready to replace a player if he or she is injured.

**Figure 4.18. Failover replicas step in when the primary database has a problem**



To create these, it's really no more than clicking in the Cloud Console. Let's start first by creating a failover replica.

Navigate over to the list of SQL instances, and you should notice a button that says "Add Failover".

**Figure 4.19. The list of SQL instances**

Instance ID	Type	IP address	Storage used	Storage type	Failover	Location
todo-list	Second Generation	104.196.23.32	1 GB of 50 GB (2.6%)	SSD	Add Failover	us-east1

When you click "Add Failover", you'll see a form that looks a lot like creating a new SQL instance — because it is — with a one extra option. Notice that you can choose a different zone within the same region. For example, with the current instance, the region is locked to `us-east1` but you can choose a different zone such as `us-east1-b`, or leave it as "Any" which is saying that you don't really care which zone the instance lives in.

**Figure 4.20. Create a failover replica**

[← Create failover replica](#)

**Instance ID**  
ID is permanent. Use lowercase letters, numbers, or hyphens. Start with a letter.

**Type** [?](#)  
Second Generation  
Failover replica of todo-list

**Database version**  
MySQL 5.6

**Location** [?](#)  
Choose a region and zone for your data. Or leave the zone as "Any" and Cloud SQL will choose the zone for you. For better performance, keep your data close to the services that need it.

Region	Zone
us-east1	Any

**Machine type** [?](#)  
For better performance, choose a machine type with enough memory to hold your largest table.

 db-n1-standard-2	vCPUs 2	Memory 7.5 GB	<a href="#">Change</a>
<b>Network throughput</b> <a href="#">?</a>		500 of max 15,000 MB/s	
			

**Storage type** [?](#)  
Choice is permanent.  
SSD

**Storage capacity** [?](#)  
Failover capacity is fixed at master capacity at the time you added the failover.  
Cannot edit.

50	GB		
<input type="checkbox"/> Enable automatic storage increase	Adds storage capacity whenever space is low. Up to 25 GB per increase. All increases are permanent. <a href="#">Learn more</a>		
<b>Disk throughput</b> <a href="#">?</a> <b>IOPS (16KB operations)</b> <a href="#">?</a>			
Read: 24 MB/s	Max: 240 MB/s	Read: 1,500 IOPS	Max: 4,000 IOPS
			
Write: 24 MB/s	Max: 151.5 MB/s	Write: 1,500 IOPS	Max: 5,000 IOPS
			

**Authorized networks**  
Add IPv4 addresses below to authorize networks to connect to your instance. Networks will only be authorized via these addresses.

Anyone (0.0.0.0/0)	<a href="#"></a>
<a href="#"> Add network</a>	

[▼ Show advanced options](#)

[Create](#) [Cancel](#)

The whole idea behind a failover replica is that you're preparing for some sort of catastrophe. This might be a simple database crash, but it could also be an outage of an entire zone. In other words by creating a failover replica in a different zone than the primary, you can be certain that even if one zone were to fail for whatever reason, your database would be able to continue working with little interruption.

In this example, we'll choose `us-east1-c` for our failover replica and click Create. Once that VM is created, you should see the replica underneath the primary instance in a hierarchical representation.

**Figure 4.21. The list of SQL instances with a failover**

Instance ID	Type	IP address	Storage used	Storage type	Failover	Location
todo-list	Second Generation	104.196.23.32	1 GB of 50 GB (2.6%)	SSD	Enabled	us-east1
todo-list-failover	Failover replica	104.196.218.206	1 GB of 50 GB (2.6%)	SSD	—	us-east1-c

To create a read replica, the process is very similar. In the list of instances choose "Create read replica" from the context menu as you can see in Figure 4.22.

**Figure 4.22. Cloud SQL instances context menu**

Instance ID	Type	IP address	Storage used	Storage type	Failover	Location
todo-list	Second Generation	104.196.23.32	1 GB of 50 GB (2.6%)	SSD	Enabled	us-east1
todo-list-failover	Failover replica	104.196.218.206	1 GB of 50 GB (2.5%)	SSD	—	us-east1-c

Create read replica
Create clone
  
Delete

At that point, you can continue just as you did with a failover replica, with one important addition: you can use a different instance type! This means that you can create a more powerful (or less powerful) read replica if you need. You can also provide it with a larger disk size if you suspect that you'll need more disk capacity.

over time. Then just click "Create" to turn on your read replica. Afterwards, your instance list should look something like Figure 4.23.

**Figure 4.23. Cloud SQL instances with replicas**

The screenshot shows the Google Cloud Platform interface for managing Cloud SQL instances. At the top, there are navigation icons and a search bar. Below that, a sidebar on the left has a 'SQL' tab selected. The main area is titled 'Instances' and features a 'CREATE INSTANCE' button. A table lists three instances:

Instance ID	Type	IP address	Storage used	Storage type	Failover	Location
todo-list	Second Generation	104.196.23.32	1 GB of 50 GB (2.6%)	SSD	Enabled	us-east1
todo-list-failover	Failover replica	104.196.195.137	—	SSD	—	us-east1
todo-list-replica	Second Generation replica	104.196.164.33	1 GB of 50 GB (2.4%)	SSD	—	us-east1

#### 4.6.1 Replica-specific operations

In addition to the typical operations you can do on a Cloud SQL instance (e.g., restart it, edit it, etc), there are a couple of operations that are only possible with read replicas: promoting and disabling replication.

Disabling replication does exactly what it says it does: pauses stream of data between the primary and the replica, effectively freezing the database as it is in the moment that replication was disabled. This can be handy if you're worried about a bug that might change your replica inadvertently or if you just want to freeze the data in a certain way for development.

If you choose to re-enable replication the replica will resume pulling data from the primary instance, and eventually come into sync with the primary.

"Promoting" an instance is Cloud SQL's way of allowing you to decouple a read replica from its primary instance. In effect, this allows you to take a read replica and then make it its own stand-alone instance, completely separate from the primary. This is useful in combination with disabling replication if you are worried about a bug that might corrupt your data. You can disable replication and then deploy the (potentially buggy) code. If there is a bug, you can promote the replica and delete the old primary, using the replica as the new primary. If there is no bug, you can simply re-enable replication and resume where you left off.

Now, let's look at something that might not seem important, but may become a life-or-death situation for your business: backups.

#### 4.7 Back-up and restore

When we talk about back-ups in the planning stages, most people's eyes gloss over, and then when disaster strikes suddenly the attitude changes entirely. Cloud SQL does

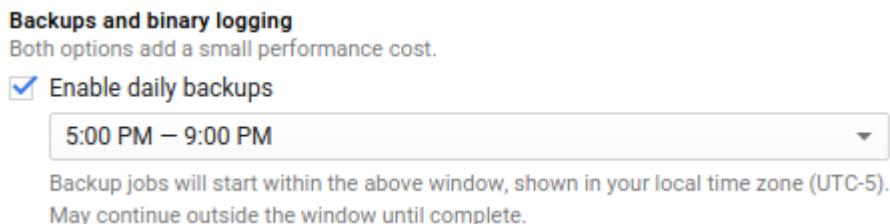
a pretty solid job of making back-ups simple so that you don't have to think about it until you need them. There are lots of different back-up methods, but let's start by looking the simplest: automatic daily back-ups.

#### 4.7.1 Automated daily back-ups

The simplest, quickest, and probably most useful back-up for Cloud SQL is the automatic one that's done daily at a time you specific when you create the Cloud SQL instance. While you can certainly disable this (for example, if you're just running a test database) it's probably a bad idea to turn it off for anything that stores data you care at all about.

To set this, all you have to do is choose a back-up window when creating your Cloud SQL instance (and you can always change this setting later on).

**Figure 4.24. Setting the automated back-up window**



When these are enabled, Cloud SQL will snapshot all of your data to disk every day, and keep a copy of that around for 7 days on a rolling window (so that you always have the last 7 days worth of back-ups). After that, you can see the list the available back-ups (either in the Cloud Console or using the command-line tool), and restore from any of them to recover your data as it exists in that snapshot.

The back-up itself is a disk-level snapshot, which begins with a special user (`cloudsqladmin`) sending a `FLUSH TABLES WITH READ LOCK` query to your instance. This command tells MySQL to write all data to disk and prevents writes to your database while that is happening. This means that if a back-up is in progress, any queries that write to your database (such as `UPDATE` and `INSERT` queries) will fail and need to be retried. This is also why it's so important to choose a back-up window that doesn't overlap with times where your users or customers are trying to modify data in your system.

Typically back-ups only take a few seconds, but if you've been writing a lot of data to your database it may take longer to persist everything to disk. Additionally, if there are long-running operations in progress when Cloud SQL tries to start the back-up job (such as data imports or exports), the job will fail, but it will be automatically retried throughout the back-up window.

Coming full circle, restoring back-ups is a simple single command, using the "due time" as the unique identifier for which back-up to restore from. Shown below is how you might restore your database to a previous back-up.

**Listing 4.15. Restoring from a back-up**

```
$ gcloud sql backups list --instance=todo-list --filter "status = SUCCESSFUL"
DUE_TIME           ERROR   STATUS
2016-01-15T16:19:00.094Z -      SUCCESSFUL
Listed 1 items.

$ gcloud sql instances restore-backup todo-list
--due-time=2016-01-15T16:19:00.094Z
Restoring Cloud SQL instance...done.
Restored [https://www.googleapis.com/sql/v1beta3/projects/your-project-id-
here/instances/todo-list].
```

**WARNING**

If your instance has replicas attached (e.g., read replicas or failover replicas), you must first delete those before restoring from a back-up.

This type of back-up is quick and easy, but what if you want more than one back-up per day? Or what if you want to keep back-ups longer than 7 days? Let's look at a more manual approach to back-ups.

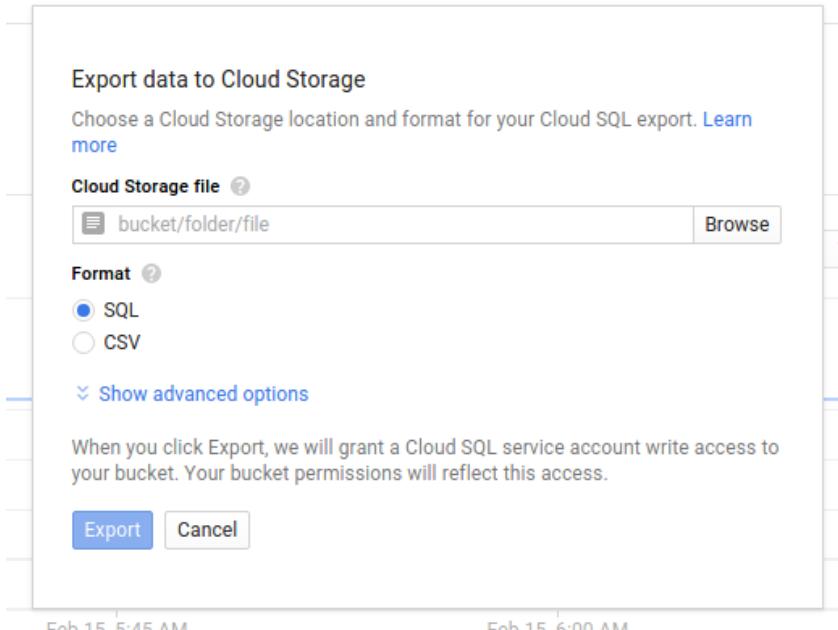
### **4.7.2 Manual data export to Cloud Storage**

In addition to the automated back-up systems, Cloud SQL also provides a managed import and export of your data that relies on Google Cloud Storage to store the back-up. This option is more manual, so if you want to automate and schedule data exports the script to do so is something you'd have to write yourself (however with the `gcloud` command-line tool it really wouldn't be all that difficult).

Under the hood, exporting your data is effectively telling Cloud SQL to run the `mysqldump` command against your database and put the output of that command into a bucket on Cloud Storage. This means that everything you've come to expect from `mysqldump` applies to this export process, including the convenient fact that exports are run with the `--single-transaction` flag (meaning that at least InnoDB tables won't be locked while the export runs).

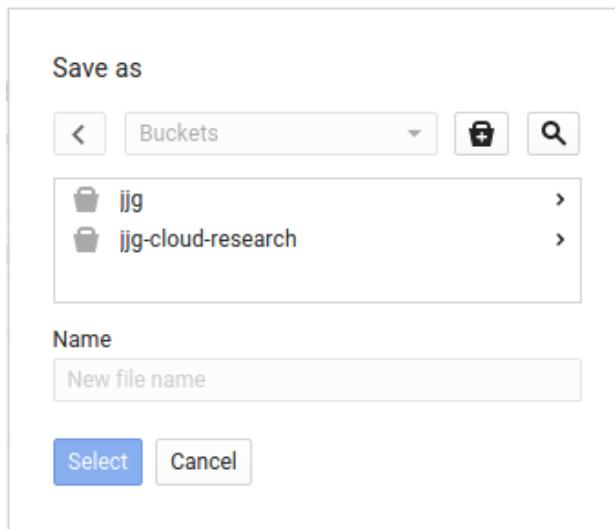
To get started, go to the instance details page for your Cloud SQL instance, and click the "Export" button at the top of the page. This will present you with a dialog box where you can set some options for the data export.

**Figure 4.25. Data export configuration**



On this page, the first field sets where you want to store the exported data. If you don't have any buckets yet in Cloud Storage that's OK — you can use this dialog to create a new one.

**Figure 4.26. Choosing a location for your export**



Click on the "Browse" button next to the text box, and at the top of the new dialog that

opens up, you should see a small icon that looks like a bucket with a plus-sign in the center. When you click this, you'll see a dialog where you can choose the name for your bucket, as well as the storage class and location. We go through the differences in all of the storage classes later on, but in general back-ups are a good fit for the "Nearline" storage class, as it's less expensive for infrequently accessed data.

**NOTE****Title**

You might also want to consider creating a read-replica and using that instance to export your data. By doing that, you avoid using your primary instance's CPU time while exporting data to Cloud Storage.

**Figure 4.27. Create a bucket**

The screenshot shows a 'Create a bucket' dialog box. At the top is the title 'Create a bucket'. Below it is a 'Name' field with a note: 'The bucket name must be unique across Cloud Storage.' A text input field is below this. Next is a 'Storage class' dropdown menu set to 'Nearline'. Then is a 'Location' dropdown menu set to 'United States'. A note about privacy follows: 'Privacy: Do not include sensitive information in the bucket name. Users cannot access your data without permission, but they can still try to access or create buckets to find out if the name exists.' At the bottom are two buttons: a blue 'Create' button and a white 'Cancel' button.

You'll want to choose a **globally** unique name (not just one unique to your project), so a good guideline is to use something like the name of your company combined with the purpose of the bucket. For example, InstaSnap might name their bucket `instasnapsql-exports`.

Once your bucket is created, double click on it in the list of buckets and type in a name for your data export. A good guideline here is to use the instance name combined with the date in a standard format. For example, InstaSnap's export from January 20, 2016 might be called `instasnapsql-2016-01-20.sql`. Also make sure that the file doesn't already exist as the export will abort if the target file already exists in your bucket.

Lastly, if you plan to use your data export as a complete back-up (that is, you intend to "revert" to the data stored exactly as it is in the export), make sure to choose the SQL

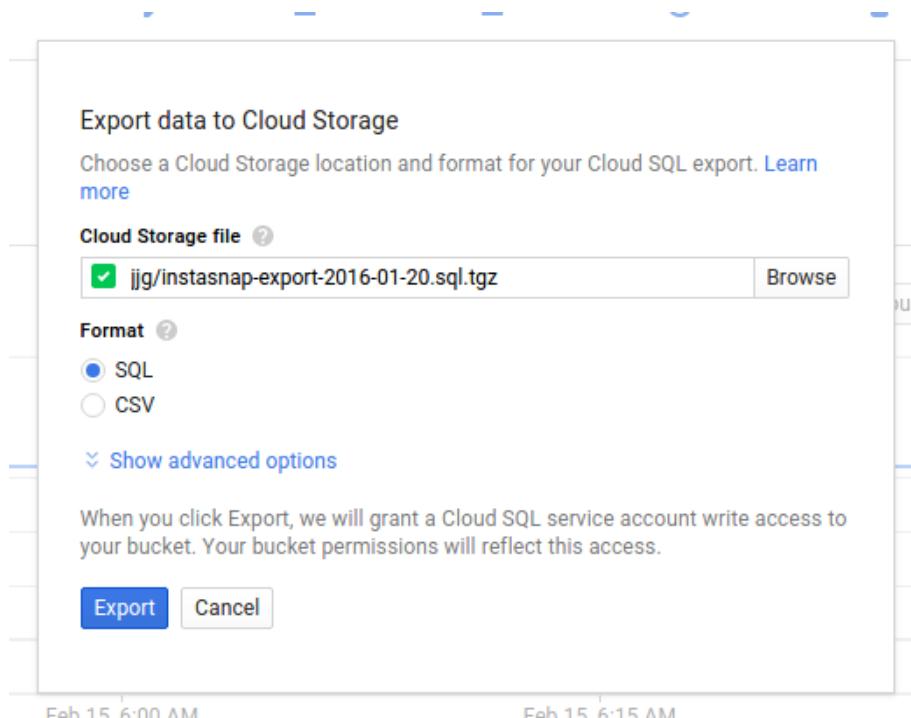
format (not CSV) which includes all of your table definitions along with your schema rather than just the data alone. With an export in SQL format, the output is effectively the SQL statements required to bring the database into the state that exists when the export is executed.

**TIP**

If you put `.tgz` at the end of your export file name, it will be automatically compressed using gzip.

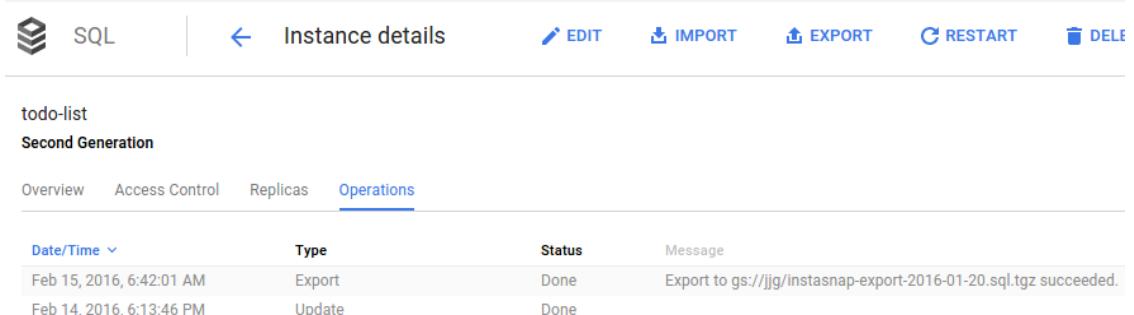
Once you click "Select", you'll be brought back to the export dialog, which should show your export path with a green check-mark next to it. Just click "Export" to start things off.

**Figure 4.28. Export data to Cloud Storage**



This could take a few minutes depending on how much data is in your Cloud SQL instance, but you can check on the status by clicking on the "Operations" tab on the instance details page. When the operation is complete you will see a row confirming that the export succeeded.

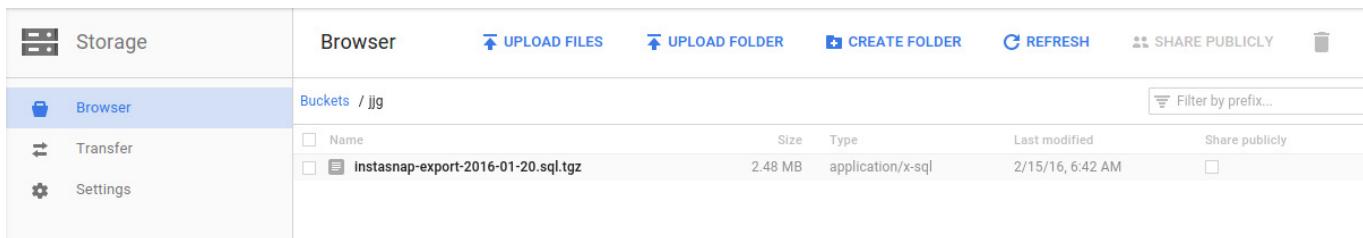
**Figure 4.29. Operations list with the successful export**



Date/Time	Type	Status	Message
Feb 15, 2016, 6:42:01 AM	Export	Done	Export to gs://jjg/instasnap-export-2016-01-20.sql.tgz succeeded.
Feb 14, 2016, 6:13:46 PM	Update	Done	

To confirm that your export actually worked, you can open your bucket in the Cloud Storage browser. If you browse to your bucket, you'll see the export available in the bucket, along with how big it is and other details.

**Figure 4.30. Your export will be visible in the Cloud Storage browser**

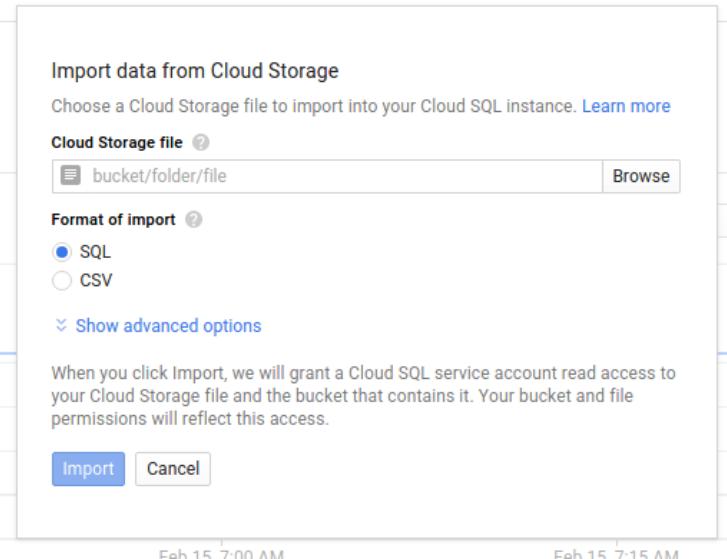


Storage	Browser	UPLOAD FILES	UPLOAD FOLDER	CREATE FOLDER	REFRESH	SHARE PUBLICLY	⋮
	Buckets / jjg					Filter by prefix...	
Browser							
Transfer							
Settings							

Name	Size	Type	Last modified	Share publicly
Instasnap-export-2016-01-20.sql.tgz	2.48 MB	application/x-sql	2/15/16, 6:42 AM	<input type="checkbox"/>

Now that you have an export on Cloud Storage, let's walk through how to restore it into your Cloud SQL instance. Start by clicking "Import" on the instance details page, and you should see a dialog that looks very similar to the one you used when creating the data export. From there, browse to the export file that you created beforehand, click "Import", and you're all done.

**Figure 4.31. The data import dialog**



What's neat about this is that you're not limited to importing data that was created using the export dialog. Instead, "importing" is nothing more than executing a set of SQL statements against your Cloud SQL instance and allowing you to use Cloud Storage as the source of the input. In other words, if you have a file full of SQL statements that happens to be large, you can upload that file to Cloud Storage and execute those by treating them as an "import".

At this point, we've seen quite a bit of detail about what Cloud SQL can do. Let's take a moment to step back and consider how much all of this is going to cost.

## 4.8 *Understanding pricing*

As you read in [chapter 1](#), there are two basic principles of pricing in Google Cloud for computing resources: computing time and storage. The prices for Cloud SQL follow these same principles, with a slight mark-up on CPU for managing the MySQL binary and configuration for you.

As of this writing, a "small" Cloud SQL instance would cost about 5 cents per hour, and the "top of the line" high-memory, 16-core, 104 GB memory machine would cost about 2 dollars per hour. For your data, the going price is the same as persistent SSD storage, which is 17 cents per Gigabyte per month. There is also the concept of "sustained use" discounts for computing resources, which is described in far more detail in [chapter 9](#), but the short version is running instances around the clock costs about 30% less than the sticker price.

To make this more clear, take a look at the following comparison. This doesn't include all of the different configurations for Cloud SQL instances, but covers a representative

spectrum of the more common options.

**Table 4.3. Different sizes of Cloud SQL instances and costs**

ID	CPU Cores	Memory	Cloud SQL	Compute Engine	Extra cost
g1-small	1	1.70 GB	\$0.0500	\$25.20	\$0.0350
n1-standard-1	1	3.75 GB	\$0.0965	\$48.67	\$0.0676
n1-standard-16	16	60.0 GB	\$1.5445	\$778.32	\$1.0810
n1-highmem-16	16	104 GB	\$2.0120	\$1,014.05	\$1.4084

You may also be wondering how these numbers compare to the "running your own VM" option discussed above. Let's start by looking at a comparison of these two options, focusing exclusively on the cost of computing power rather than storage since it's the same for both options. We'll also work using the assumption that we'll run our database for a full month to make the numbers a bit easier to relate to.

**Table 4.4. Cloud SQL vs Compute Engine monthly cost**

ID	CPU Cores	Memory	Cloud SQL	Compute Engine	Extra cost
g1-small	1	1.70 GB	\$25.20	\$13.68	\$11.52
n1-standard-1	1	3.75 GB	\$48.67	\$25.20	\$23.47
n1-standard-16	16	60.0 GB	\$778.32	\$403.20	\$375.12
n1-highmem-16	16	104 GB	\$1,104.05	\$506.88	\$597.17

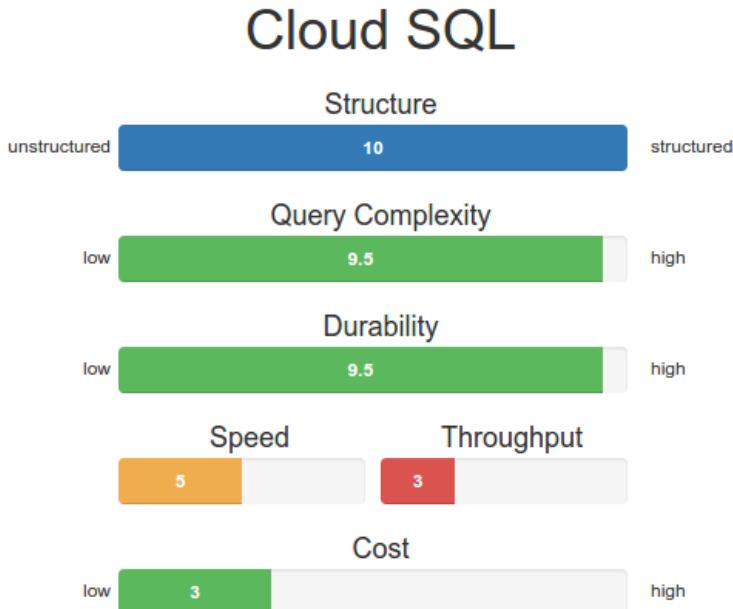
As you can see, since the cost of Cloud SQL is directly proportional to the hourly cost, as you scale up to larger and larger VM types, your absolute cost difference grows. While this might not mean much for the smaller-scale deployments (\$13 dollars vs \$11 dollars isn't a big deal), it starts to become a bigger deal as you add more and more machines. For example, if you were running 20 of the largest machine type in our table, you'd be paying \$12,000 in extra cost for your Cloud SQL instances **every month!** That's \$144,000 annually, which means you may be better off hiring someone to manage your databases and switching to Compute Engine VMs.

With this new knowledge about how much it costs to run using Cloud SQL, let's take a moment to explore when you should use Cloud SQL for your various projects.

## 4.9 When should I use Cloud SQL?

Before we decide whether Cloud SQL is a good fit, let's look at the score-card we mentioned before. Keep in mind that because Cloud SQL is essentially just MySQL, this score card is identical to the score card for running your own MySQL server on a virtual machine in a cloud service like Compute Engine or Amazon's EC2, or using Amazon's RDS mentioned above.

**Figure 4.32. Score card for Cloud SQL**



As you may have noticed, there are a few interesting things about this score card. Let's go through point by point to understand why the scores came out this way.

### 4.9.1 **Structure**

Most relational databases store highly structured data with a complete schema defined ahead of time that is strictly enforced. While this can sometimes be frustrating, especially with JSON-formatted data, it can often prevent data corruption errors that happen when different people make different assumptions about how types are cast from one to the other. This also means that your database can optimize your data a bit more as it has more information about both the data that exists currently and the data that will be added later.

As you can see, Cloud SQL scores highly on this metric, so if your data is, or can easily be, fit to a schema, Cloud SQL is definitely a good option.

### 4.9.2 **Query complexity**

As we mentioned initially, SQL is a very advanced language which provides some really impressive query capabilities. I think as far as query complexity, there are very few services that will come in ahead of SQL, which means that if you know you'll have really complex questions to ask of your data, SQL is probably a good fit. If, on the other hand, you really just want to look-up specific items by their IDs, change some data, and save the result back to the same ID, relational storage might be overkill, and

you may want to explore other storage options.

### 4.9.3 Durability

Durability is another area where relational databases really shine. If you're looking for something that **really** means it when it says, "I saved your data to disk", relational databases are a great choice. While you should still dig deeply on tuning MySQL for the level of durability you need, the general consensus is that relational storage systems (like MySQL) are capable of providing a very high level of durability. Further, since Cloud SQL runs on top of Compute Engine and stores all the data on Persistent Disk, you benefit from the higher levels of durability and availability offered by PD. For more detail on Persistent Disk, check out the section in [chapter 9](#).

Now let's start exploring the areas where relational storage tends to not be as great.

### 4.9.4 Speed (latency)

Generally the latency of a query over your data is a function of the amount of data that your database needs to analyze to come up with your answer. This means that while your database may start off being fast, as your overall data grows further your queries may get slower and slower. To make matters worse, assuming the query rate stays relatively even, as queries get backed up in your database future queries will pile up on top of each other, effectively making a very long line of people all asking for data and not getting answers.

Put simply, this means that if you plan to have hundreds of gigabytes of data, you may want to consider different storage strategies. If you simply aren't sure about how big your data will be, you can always start with Cloud SQL and migrate to something bigger when your query performance becomes unacceptable.

### 4.9.5 Throughput

Continuing on the topic of performance, relational storage provides strong locking and consistency guarantees (that is, the data is never "stale") but with these guarantees come things like pessimistic locking which means that the database tries to prevent lots of people from all writing at the same time, lowering the overall throughput for the database. This means that relational databases won't win the competition for the most queries handled in a 1-second period, particularly if those queries involve updating data or joining across many different tables.

Similarly to the discussion above, there's nothing wrong with starting on a relational system like Cloud SQL and migrating to a different system as your data and concurrency requirements increase beyond what's reasonably possible with something like MySQL.

### 4.9.6 Overall

Now that you understand what relational storage is good at (and not good at), let's look at our original examples and decide whether Cloud SQL would be a good fit.

### To-do list

If you recall, the to-do list application was intended as a good "starter-app" for learning new systems. Let's go through the various aspects of this application, and see how it lines up with Cloud SQL as a possible storage option.

**Table 4.5. To-do list application storage needs**

Aspect	Needs	Good fit?
Structure	Structure is fine, not necessary though	Sure
Query complexity	We don't have that many fancy queries	Definitely
Durability	High, we don't want to lose stuff	Definitely
Speed	Not a lot	Definitely
Throughput	Not a lot	Definitely

Based on this, it seems like Cloud SQL is a pretty good option for the to-do list.

What about something more complicated, like E\*Exchange?

### E\*Exchange

E\*Exchange was our online trading platform, where people could buy and sell stocks with a click of a button. Let's look through the list and see how Cloud SQL stacks up against the requirements for this application.

**Table 4.6. E\*Exchange storage needs**

Aspect	Needs	Good fit?
Structure	Yes, reject anything suspect, no mistakes	Definitely
Query complexity	Complex, we have fancy questions to answer	Definitely
Durability	High, we <b>cannot</b> lose stuff	Sure
Speed	Things should be pretty fast	Probably
Throughput	High, we may have lots of people using this	Maybe

Not quite as rosy of a picture for E\*Exchange, primarily owing to the performance metrics regarding latency (speed) and throughput. Cloud SQL can do a lot of querying, and can do so pretty fast, but the more data we accumulate, the slower queries tend to become. This can be addressed with read-slaves (as you'll see later), but that isn't a solution for the growing number of updates to the data, which would all still go through a single master MySQL server.

Additionally, this assumes that the only data being stored in here is customer data such as balances, bank account information, and portfolios. Trading data, which is likely to be much much larger than the customer data, would not be well suited for relational storage, but instead would fit better in some sort of data warehouse. We'll explore some options for this type of data in the section on Big Data and Analytics.

In short, this means that while Cloud SQL may be a good place to start while E\*Exchange has moderate amounts of data, if that data grows into tens to hundreds of

gigabytes, they may have to migrate to a different storage system or risk frustrating their customers with downtime or slow-loading pages.

### **InstaSnap**

InstaSnap was our super high-traffic application that caught on with celebrities all over the world — meaning lots of concurrent requests.

As we mentioned above, this alone is likely to disqualify something like Cloud SQL from the list of possibilities, but let's run through the score-card.

**Table 4.7. InstaSnap storage needs**

Aspect	Needs	Good fit?
Structure	Not really, structure is pretty flexible	Not really
Query complexity	Mostly look-ups, no highly complex questions	Not really
Durability	Medium, losing things is inconvenient	Sure
Speed	Queries must be very fast	Not really (with lots of data)
Throughput	Very high, Kim Kardashian uses this	Not really

This looks like Cloud SQL is a pretty bad fit for something of this scale, particularly when the most valuable features of a relational storage system like MySQL aren't even really necessary.

For a product like InstaSnap, the structure of the data isn't all that important, nor is the durability and transactional semantics. In a sense, using Cloud SQL would be sacrificing the high performance that we desperately need for transactions, high durability, and high consistency that we don't really care all that much about.

In short, Cloud SQL isn't a great fit for something like InstaSnap, so if your needs are similar to InstaSnap's, you should continue reading to learn about other storage options.

Let's assume, however, that Cloud SQL does fit your needs. If Cloud SQL is just a VM that runs MySQL, why not just turn on a VM on Compute Engine and install MySQL?

## **4.10 Why choose Cloud SQL over a VM?**

Cloud SQL was built with a very specific target audience in mind, that target is the group of people that "just want MySQL" and don't care all that much about customizing their instance all that much. In other words, if you were just planning to turn on a VM, install MySQL, and change the password, Cloud SQL was made for you.

As we discussed in [chapter 1](#), one of the primary motivations for shifting towards "the cloud" was to reduce your overall TCO (total cost of ownership). Cloud SQL does this not necessarily by reducing the cost of the hardware, but by reducing your maintenance and management costs. For example, if you were running your own VM running MySQL, you'd need to find the time to upgrade your operating system and MySQL

version for any new security patches that happen to come out (or accept the risk of your data being compromised, but we'll assume you'd never do that).

While this is a relatively small amount of work, it can be time consuming if you don't know your way around MySQL and fixing amateur mistakes could become costly. Further, with a self-managed MySQL deployment the cost of operation is tied to the price of an engineering-hour, rather than to the cost of the hardware.

In short, Cloud SQL's focus isn't to be a "better, faster" MySQL, it's to be a "simpler, lower overhead" MySQL. In this way, Cloud SQL is very similar to Amazon's RDS, and both are a great fit for the typical MySQL use-cases.

There will be times where you have more specific requirements for your database, and in those situations you may end up needing more flexibility than Cloud SQL can provide you. The most common scenario is requiring a different relational database such as PostgreSQL or Microsoft's SQL Server. Right now, Cloud SQL only handles MySQL, which means that if you need any other relational database flavor, Cloud SQL isn't a good fit. While MySQL is a very reasonable choice, other database systems have some impressive features (such as PostgreSQL 9.5's native JSON type support) and if you want or need these for whatever reason, the better fit is likely to be running your database on a VM and managing it yourself.

A slightly less common (but still possible) situation is the case where you need a particular version of MySQL for your system. As of this writing, Cloud SQL only offers MySQL version 5.6, so if you need to run against version 5.5 (or some other older version) then Cloud SQL won't work for you.

One other situation, which becomes more likely as your usage of MySQL becomes more complex and resource intensive, is when you need to use MySQL's advanced scalability features such as multi-master or circular replication. If you've not heard of these, that's OK, they aren't nearly as common as the much more standard master-slave replication option, which Cloud SQL does support and which you'll read about later.

In short, a good guideline for whether Cloud SQL is a good fit is pretty simple: Do you need anything really fancy? If not, give Cloud SQL a try.

If you find yourself needing "fancy" things later on (like circular replication or a special patched version of MySQL) you can easily migrate your data from Cloud SQL over to your own VMs running MySQL in exactly the configuration you want.

You may be thinking now, "This is all great, but how much will this cost me?" Let's dig into that — and it should be quick as the pricing scheme is pretty simple.

## 4.11 Summary

- Relational databases are great for storing data that "relates" to other data using foreign key references, such as a customer database.
- Cloud SQL is "MySQL in a box" which runs on top of Compute Engine.
- When choosing the size of your persistent disk, don't forget that the size is

directly related to performance. It's OK (and expected) to have lots of empty space.

- When you have enough Cloud SQL instances to justify a DBA, it might make sense to manage MySQL yourself on GCE instances.
- Always configure Cloud SQL to encrypt traffic using an SSL certificate to avoid eavesdropping on the internet.
- Don't worry if you chose too slow of a VM because you can always change the computing power later. You can also increase the storage space, but it's more work to decrease it if you overshoot.
- Use failover replicas if you want your system to be up even when a zone goes down.
- Enable daily backups if you want to be sure to never lose data.

# 5

## *Cloud Datastore: Document storage*

### **This chapter covers:**

- What is document storage?
- What is Cloud Datastore?
- How to interact with Cloud Datastore
- Deciding whether Cloud Datastore is a good fit
- Key distinctions between "hosted" and "managed" services

## **5.1 What is document storage?**

Document storage is a form of "non-relational" storage, which happens to be very different conceptually from the relational databases discussed in [chapter 4](#). With this type of storage, rather than thinking of "tables" containing "rows" and keeping all of your data in a rectangular grid, a document database thinks in terms of "collections" and "documents". These "documents" are arbitrary sets of key-value pairs and the only thing they must have in common is the "type" which matches up with the "collection". For example, in a document database you might have a "Employees" collection, and inside there you could have 2 documents:

### **Listing 5.1. Two example documents in the "Employees" collection**

```
{"id": 1, "name": "James Bond"}  
{"id": 2, "name": "Ian Fleming", "favoriteColor": "blue"}
```

Comparing this to a traditional table of similar data, you'll see that the grid format will look quite different from a document collection's jagged format.

**Table 5.1. Grid of employee records**

ID	Name	Favorite color
1	"James Bond"	null
2	"Ian Fleming"	"blue"

**Table 5.2. Jagged collection of employees**

Key	Data
1	{id: 1, name: "James Bond"}
2	{id: 2, name: "Ian Fleming", favoriteColor: "blue"}

This shouldn't look all that scary at first glance, however as you'll learn later there are going to be a few things you might assume about querying these documents that would surprise you. As an example, what would you expect the following query to return?

#### Listing 5.2. Give me everyone who's favorite color isn't blue

```
SELECT * FROM Employees WHERE favoriteColor != "blue"
```

You might be surprised to find out that in some document storage systems the answer to this query is an empty set. Even though James Bond's favorite color isn't "blue", he isn't returned in that query!

The reason for this will vary from system to system, however one reason for this is simply that a *missing* property isn't the same thing as a property with a null value, so the only documents considered are those that explicitly have a key called `favoriteColor`. So where did all of this come from?

Ultimately unusual behavior like this comes from the fact that these systems were designed with a focus on large-scale storage. This means that in order to make sure that all queries were consistently fast the designers had to trade away advanced features like joining related data and sometimes even having a globally consistent view of the world. As a result, these systems are perfect for things like "look ups" by a single key and simple scans through the data, but nowhere near as full-featured as a traditional SQL database.

## 5.2 What is Cloud Datastore?

Cloud Datastore, formerly called "the App Engine Datastore", originally came from a storage system Google built called "Megastore". It was first launched as the default way to store data in Google App Engine, and has since grown into a stand-alone storage system as part of Google Cloud Platform. As you might guess, it was designed to handle "large-scale data" and made many of the trade-offs that are common to other document storage systems.

Before we go into the key concepts you need to know when using Datastore, let's first look at some of these design decisions and trade-offs that went into Datastore.

### 5.2.1 Design goals for Cloud Datastore

There is one very obvious use case for a large-scale storage system that makes for a great example: Gmail. Think about if you were trying to build Gmail, and needed to store everyone's mailboxes. Let's look at all of the things that would go into how you'd design your storage system.

#### DATA LOCALITY

The first thing you'd notice is that while your "mail database" would need to store all e-mail for all accounts, you don't actually need to search across multiple accounts. In other words, you'd never run a search over Harry's **and** Sally's e-mails. This means that technically you could put everyone's e-mail on a completely different server and no one would notice a difference! In the world of storage, this concept of "where to put data" is called *data locality*. Datastore is designed in a way where you can choose which documents live near which other documents by putting them in the same *entity group*.

#### RESULT-SET QUERY SCALE

Another requirement with this database is that it'd be really frustrating if your inbox got slower as you received more e-mail. To deal with this, you'd probably want to "index" e-mails as they arrive so that when you want to search your inbox, the time it takes to run any query (e.g., searching for specific e-mails or listing the top 10) should be proportional only to the number of **matching** e-mails (**not** the total number of e-mails).

This idea of making queries as expensive as the number of results is sometimes referred to as *scaling with the size of the result set*. Datastore uses indexing to accomplish just this, so that if your query has 10 matches, it'll take the same amount of time regardless of whether you have 1 GB or 1 PB of e-mail data.

#### AUTOMATIC REPLICATION

Finally, we must worry about the fact that sometimes servers die, disks fail, and networks go down. To make sure that people can always access their e-mail we'd need to put e-mail data in lots of different places to make sure that it's always available. To make this happen, any data written should be automatically replicated to many different physical servers meaning that your e-mail is never on a single computer with a single hard-drive. Instead, each e-mail is distributed across lots of different places. Normally this is a very difficult thing to achieve if you start from traditional database software, however Google's underlying storage systems are actually well suited to this requirement and Cloud Datastore does just this.

Now that you understand some of the underlying design choices, let's explore a few of the key concepts and how you use them.

### 5.2.2 Concepts

You learned in bits and pieces that document storage is pretty different from relational

storage, however we didn't really dive into the specifics of Cloud Datastore's take on these differences. Let's look at the important pieces and discuss how they fit together.

## KEYS

The most primitive concept to learn first is the idea of a *key*, which is what Cloud Datastore uses to represent a unique identifier for anything that has been stored.

The closest thing to compare this to in the relational database world is the "unique ID" you often see as the first column in tables, however Datastore keys have 2 major differences from table IDs.

The first major difference is that since Datastore doesn't really have an identical concept of "tables", Datastore's keys actually contain both the "type" of the data as well as the unique identifier. To illustrate this with an example of storing "employees" in MySQL the typical pattern is to create a table called `employees` and have a column in that table called `id` which is a unique integer. Then you insert an employee and give it an ID of 1.

In Cloud Datastore, rather than creating a table and then inserting a row, it happens all in one step where you insert some data where the *key* is `Employee:1`. The "type" of the data here (`Employee`) is referred to as the *kind*.

The second major difference is that keys themselves can be *hierarchical*, which is a feature of the concept of *data locality* described above. This means that your keys can have "parent keys", which co-locates your data, effectively saying "put me nearby my parent". An example of a nested (or hierarchical) key would be `Employee:1:Employee:2` which is actually a pointer to employee #2.

If two keys have the same parent, they are in the same *entity group*. This means that parent keys are how you tell Datastore to put data near other data (give them the same parent!).

This gets tricky when you realize that there isn't always a great reason for nested keys of the same *kind*, but instead you might want to nest sub-entities inside one another. This is perfectly acceptable as keys can refer to multiple kinds in their "path" or the hierarchy, and the kind (type) of the data is actually the kind of the bottom-most piece. For example, you might want to store your employee records as children of the company they work for, which could be `Company:1:Employee:2`. The kind of this key is `Employee`, and the parent key is `Company:1` (whose kind is `Company`). This key refers to employee #2 and due to its parent (`Company:1`) it will be stored nearby all other employees of the same company (e.g., `Company:1:Employee:44` will be nearby).

Also note that although you've only seen numerical IDs in the examples, you can also specify keys as strings, such as

`Company:1:Employee:jbond` or `Company:apple.com:Employee:stevejobs`.

## ENTITIES

The primary storage concept in Cloud Datastore is an *entity*, which is Datastore's take

on a "document".

From a technical perspective, an entity is nothing more than a collection of properties and values combined with a unique identifier, called a *key*.

An entity can have properties of all the basic types, such as:

- booleans (true or false)
- strings ("James Bond")
- integers (14)
- floating-point numbers (3.4)
- dates or times (2013-05-14T00:01:00.234Z)
- binary data (0x0401)

#### **Listing 5.3. An example entity with just primitive types**

```
{
  "__key__": "Company:apple.com:Employee:jonyive",
  "name": "Jony Ive",
  "likesDesign": true,
  "pets": 3
}
```

In addition to the basic types, sometimes referred to as "primitives", Datastore also exposes some more advanced types such as:

- lists, which allow you to have a list of strings
- keys, which point to other entities
- embedded entities, which act as "sub-entities"

#### **Listing 5.4. An example entity with more advanced types**

```
{
  "__key__": "Company:apple.com:Employee:jonyive",
  "manager": "Company:apple.com:Employee:stevejobs", ①
  "groups": ["design", "executives"], ②
  "team": { ③
    "name": "Design Executives",
    "email": "design@apple.com"
  }
}
```

- ① The manager property is a key which points to another entity, which is as close to a "foreign key" as you can get.
- ② The groups property is a list of strings, but could easily be a list of integers, keys, etc.
- ③ The team property is an embedded entity, which itself could be structured just like any other entity stored in Datastore.

There are a few things that are unique about this configuration:

1. A reference to another Key is as close as you can get to the concept of foreign keys in relational databases.
2. There's no way to enforce that a reference is valid, so you have to keep these references up to date (e.g., if you delete the key, update the reference)
3. Lists of values are typically not supported in relational databases which typically use pivot tables to store a "has many" relationship. In Datastore, a list of primitives is the natural way to express this.
4. In relational databases you typically use a foreign key to store other structured data. In Datastore, if the structured data doesn't need its own reference, embedded entities are very useful. In some ways embedded entities are like anonymous functions in JavaScript.

Now that you understand entities and keys, what can you do with them?

### **OPERATIONS**

Operations in Cloud Datastore are pretty simple: they are the "things you can do" to an entity. The basic operations are:

- **get**: Retrieve an entity by its key
- **put**: Saves or updates an entity by its key
- **delete**: Deletes an entity by its key

Notice that it looks like all of these require the key for the entity, however if you omit ID portion of the key in a put operation, Datastore will actually generate one automatically for you.

Each of these operations would work almost identically to what you may have seen in a key-value store like Redis or Memcache, but what about querying the data you've added? This is where things get a little more complicated.

### **INDEXES AND QUERIES**

Now that you understand the fundamentals, we have to discuss the two concepts that pull it all together: *indexes* and *queries*.

In a typical database a query is nothing more than a SQL statement, such as `SELECT * FROM employees`. In Datastore this is also possible using GQL (a query language much like SQL), but there's also a more structured way of representing a query. What's interesting though is that which Datastore may look like it can "speak SQL", there are quite a few queries that can't be answered by Datastore. Further, relational databases tend to treat "indexes" as a way of *optimizing* a query, whereas Datastore uses indexes to make a query possible.

**Table 5.3. Queries and indexes, relational versus Datastore**

Feature	Relational	Datastore
Query	SQL, with joins	GQL, no joins, certain queries impossible
Index	Makes queries faster	Makes advanced query possible

So what is an index? And what type of queries go from "impossible" to "possible" with an index? It turns out that this is actually a bit more surprising than you'd think. Anytime you're filtering (e.g., using a `WHERE` clause) in your query, you're relying on an index, which is there to ensure that the query "scales with the result set".

Imagine if every time you needed to find all e-mails from Steve (`steve@apple.com`), you had to go through **all** of your e-mails, checking each one's `sender` property looking for "Steve". This clearly would work, but it means that the more e-mail you get, the longer this query takes to run which is obviously bad. The way we fix this problem is by creating an "index" which stays up-to-date whenever information changes, which we can scan through to find matching e-mails. An index is really nothing more than a specially ordered and maintained data set to make querying "always fast". For example, with our e-mail, an index over the `sender` field might look like the following table.

**Table 5.4. An index over the `from` field**

sender	Key
<code>eric@google.com</code>	<code>GmailAccount:me@gmail.com:Email:8495</code>
<code>steve@apple.com</code>	<code>GmailAccount:me@gmail.com:Email:2441</code>
<code>steve@apple.com</code>	<code>GmailAccount:me@gmail.com:Email:44043</code>
<code>tom@example.com</code>	<code>GmailAccount:me@gmail.com:Email:1036</code>

This index simply pulls out the `sender` field from e-mails and allows us to query over **all** e-mails with a certain `sender` value while also providing us with a guarantee of "when to stop querying". This means that the query for all e-mails from Steve (`SELECT * FROM Email WHERE sender = 'steve@apple.com'`) would rely on the index to find the first entry that matches, and then would continue scanning until it finds an entry that doesn't match (`tom@example.com`). As you can see, the more e-mails from Steve, the longer this query takes, however e-mails from other people (which do **not** match the query we're running) have no affect at all on how long this query takes to run.

This raises the obvious question: do I have to create an index to do a simple filtering query? Luckily no! Datastore was designed to automatically create an index for each property (called "simple indexes") so that those simple queries are possible by default. However if you want to do matching on multiple properties together you may need to create an index. For example, finding all e-mail from Steve where Eric is CC'd might be:

#### **Listing 5.5. Selecting filtered and sorted e-mails**

```
SELECT * FROM Emails WHERE sender = "steve@apple.com"
    AND cc = "eric@google.com"
```

To make sure this query scales with the result set (of matching e-mails), you'd need an index on both `sender` and `cc` that might look like the following.

**Table 5.5. An index over the sender and cc fields**

sender	cc	Key
eric@google.com	NULL	GmailAccount:me@gmail.com:Email:8495
steve@apple.com	eric@google.com	GmailAccount:me@gmail.com:Email:44043
steve@apple.com	jony@apple.com	GmailAccount:me@gmail.com:Email:9412
tom@example.com	NULL	GmailAccount:me@gmail.com:Email:1036

With this index you can do exactly as we described with the simpler query except this now involves two different properties. We call this a "composite index" and it's an example of an index you'll have to define yourself. Without an index like this, you won't be able to run the query at all, which is different from a relational database where this query would always run but might be slow without an index.

Now that you understand how indexes work and how they're used, you might be wondering what this means for performance of your queries as your data changes. For example, if you update an e-mail's properties wouldn't that mean all of the indexes that duplicated that data would need to be updated too? This is completely right, and opens the door to a much bigger question about the consistency of your data.

### 5.2.3 Consistency and replication

As you learned earlier, there are two key requirements of a distributed storage system for something like Gmail: to be "always available" and to "scale with the result set". This means that not only does data need to be replicated, but we need to create and maintain indexes for our queries, like the one described in [Listing 5.5](#).

Data replication, though complicated to implement, is somewhat of a "solved problem", with many different protocols around, each with their own trade-offs. One protocol which happens to be in use by Cloud Datastore uses something called a "two-phase commit".

In this method, we break the changes we want saved into two stages: a preparation phase and a commit phase. In the preparation phase a request is sent to a set of replicas, describing a change and asking the replicas to "get ready" to apply it. Once all replicas confirm that they have prepared the change, a second request is sent instructing all replicas to apply that change. In the case of Datastore, this second ("commit") phase is done asynchronously where some of those changes may hang around in the "prepared but not yet applied" state, leading to eventual consistency when running broad queries where the entity or the index entry may be out of date. Any "strongly consistent" query (e.g. get of an entity) will first push a replica to execute any pending commits of the resource and afterwards run the query, resulting in a "strongly consistent" result.

As you can see, maintaining entities and indexes in a distributed system is actually a much more complicated task as it means that the same save operation would also need to include the saves to any indexes that are affected by the change. (And remember that the indexes need to be replicated, so they need to be updated in multiple places as well.)

This means that Datastore would have two options:

1. Update the entity and the indexes everywhere **synchronously**, making the time it takes to confirm the operation take an unreasonably long time, particularly as you create more indexes.
2. Update the entity itself the indexes in the background, keeping request latency much lower since there's no need to wait for a confirmation from all replicas.

As you might guess, Datastore chose to update data asynchronously (option 2) to make sure that no matter how many indexes you add, the time it takes to save an entity is the same.

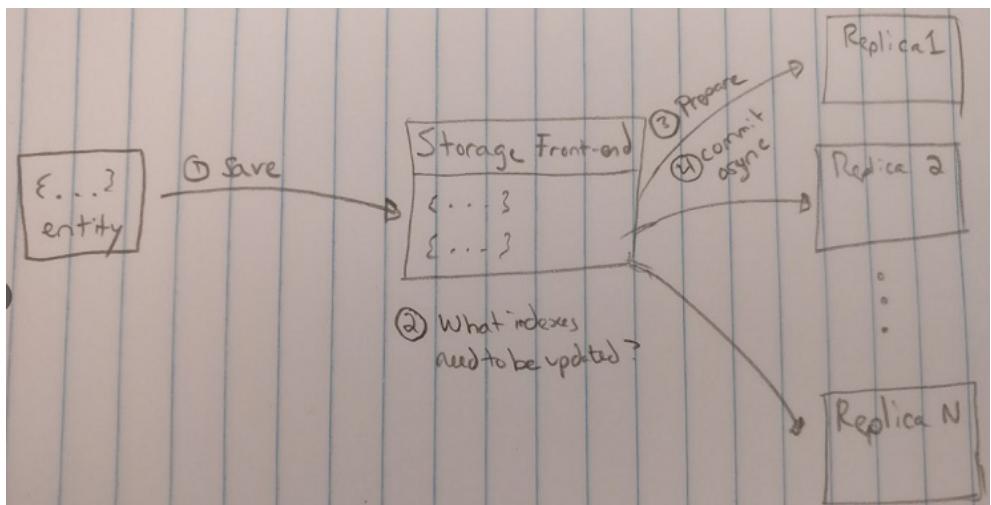
This means that when you use the `put` operation, under the hood Datastore is actually doing quite a bit of work:

1. Create or update the entity
2. Determine which indexes need to change as well
3. Tell the replicas to prepare for the change
4. Ask the replicas to apply the change when they can

And then later, whenever a strongly consistent query runs:

1. Ensure all pending changes to the affected entity group are applied
2. Execute the actual query

**Figure 5.1. Saving an entity in Cloud Datastore**

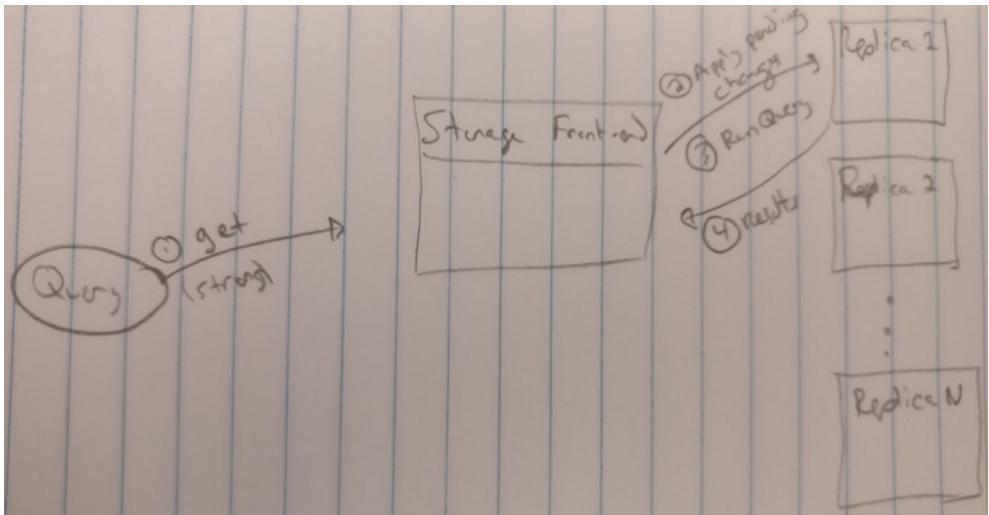


It also means that when you run a query, Datastore is using these indexes to make sure your query runs in time that is proportional to the number of matching results found. This means that a query actually does the following:

1. Send the query to Datastore
2. Search the indexes for matching keys

3. For each matching result, get the entity by its key in an eventually consistent way
4. Return the matching entities

**Figure 5.2. Querying for entities in Cloud Datastore**



At first glance this looks fantastic, however there is an unusual result hidden in the trade-off made to avoid the number of indexes from affecting the time it takes to save data. The key piece here is that the indexes are updated *in the background*. which means that there's no real guarantee on *when* the indexes will be updated.

This concept is called *eventual consistency*, which means that *eventually* your indexes will be up-to-date ("consistent") with the data you have stored in your entities. It also means that while the operations you learned about will always return "the truth", any queries you run are running over the indexes, which means that the results you get back may be slightly *behind the truth*.

For example, imagine that you've just added a new Employee entity to Cloud Datastore:

#### **Listing 5.6. Example Employee entity**

```
{
  "__key__": "Employee:1",
  "name": "James Bond",
  "favoriteColor": "blue"
}
```

Now you want to select all the employees with "blue" as their favorite color:

#### **Listing 5.7. GQL query for Employees with blue as their favorite color**

```
SELECT * FROM Employee WHERE favoriteColor = "blue"
```

If the indexes haven't been updated yet (they will *eventually*), then you won't get this employee back in the result. However, if you ask specifically for the entity, it'll be there:

#### **Listing 5.8. GQL query for Employee 1**

```
get(Key(Employee, 1))
```

In other words, your queries are eventually consistent specifically due to the fact that the indexes that Datastore uses to find those entities is updated in the background.

Note that this also applies when your entities are *modified*. For example, imagine that the indexes have reached a level of consistency and when you look for all employees with "blue" as their favorite color employee 1 is returned.

Now imagine that you change this employee's favorite color:

#### **Listing 5.9. Employee entity with a different favorite color**

```
{
  "__key__": "Employee:1",
  "name": "James Bond",
  "favoriteColor": "red"
}
```

If you run your query again, depending on which updates have been committed you may see different results, described in the table below.

**Table 5.6. Summary of the different possible results**

Entity updates	Index updated	Employee matches	Favorite color
Yes	Yes	No	Doesn't matter
No	Yes	No	Doesn't matter
Yes	No	Yes	red
No	No	Yes	blue

In short, the three possibilities are:

1. The employee won't be in the results.
2. The query still sees the employee as matching the query (`favoriteColor = blue`) so it ends up in the results.
3. The query still sees the employee as matching the query (`favoriteColor = blue`), but the entity doesn't actually match! (`favoriteColor = red`).

This must seem really strange for anyone working day-to-day with a SQL database. You may also be asking yourself, "How on earth can you build something with this?"

It's important to remember that systems like this were designed with things like Gmail in mind, which have different requirements than a typical SQL-backed web application. So how does this system benefit customers like Gmail? This brings us to

the next big topic: combining querying with data locality to get strong consistency.

#### 5.2.4 Consistency with data locality

We talked earlier about data locality as a tool for putting many pieces of data near each other (e.g., you group all of a single account's e-mails near to one another), but we didn't really clarify why that might matter.

Now that you understand the concept of eventual consistency (that your queries actually run over indexes rather than your data, and those indexes are eventually updated in the background), we can combine these two concepts together so that you can build real things where you're able to query without wondering whether the data is accurate.

Let's start with a hugely important fact: queries inside a single entity group are *strongly consistent* (not *eventually consistent*).

If you recall, an *entity group*, defined by keys sharing the same "parent" key, is how you tell Datastore to put entities near one another. This means that if you want to query over a bunch of entities that all have the same parent key, your query will be **strongly** consistent.

This is simply because by telling Datastore *where* you want to query over in terms of the "locality", Datastore now has a specific range of keys that it needs to make sure that any pending operations are fully committed prior to executing the query, resulting in strong consistency. This means that if you ask Datastore for all Apple employees who have blue as their favorite color, it knows exactly which keys could be in the result set, and before executing the query can first make sure there are no pending operations involving those keys. This means that the results will always be up to date.

Let's go back to the example before with Apple employees:

##### **Listing 5.10. Apple employee with favorite color "blue"**

```
{
  "_key_": "Company:apple.com:Employee:jonyive",
  "name": "Jony Ive",
  "favoriteColor": "blue"
}
```

Now let's change Jony's favorite color:

##### **Listing 5.11. Updating the favorite color to "red"**

```
{
  "_key_": "Company:apple.com:Employee:jonyive",
  "name": "Jony Ive",
  "favoriteColor": "red"
}
```

As you learned before, running a query across all employees may not accurately reflect

your data, however if you query over all Apple employees, you're guaranteed to get the latest data:

**Listing 5.12. Query for all Apple employees with favorite color "red"**

```
SELECT * FROM Employees WHERE favoriteColor = "blue" AND
__key__ HAS ANCESTOR Key(Company, 'apple.com')
```

Since this query is limited to a single entity group the results will always be consistent with the data, which is referred to as being "strongly consistent".

This begs the obvious question: "Why don't I just put everything in a single entity group? Won't I always have strong consistency then?"

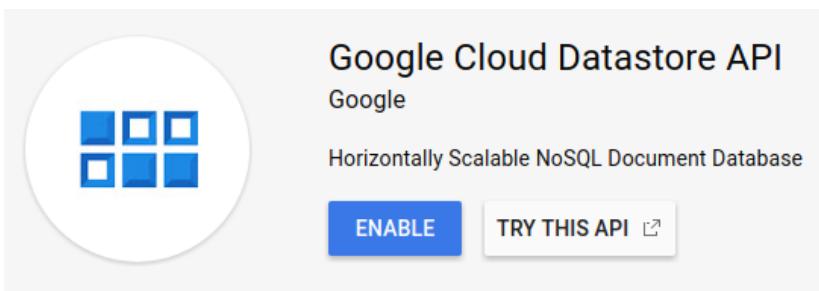
This is technically true, but that doesn't make it a good idea. The reason for this is that there are limitations on how many requests can be handled simultaneously by a single entity group (which is in the range of about 10 per second). This means that you'd basically be trading eventual consistency and getting pretty low throughput overall in return. If you really value strong consistency enough that you'd be willing to throw away the scalability of Datastore, you should probably be using a regular relational database instead.

Now that you have some idea of how Cloud Datastore works, let's kick the tires a bit to see what it's like to use it in your app.

### 5.3 Interacting with Cloud Datastore

Before you can use Cloud Datastore you may need to "enable" it in the Cloud Console. To do this, start by searching for "Cloud Datastore API" in the main search box, which should have only one result. Click on that to get to a page that should have a big button saying "Enable". (If you only see the ability to "Disable" the API, that means you're already set.)

**Figure 5.3. Enable the Cloud Datastore API**



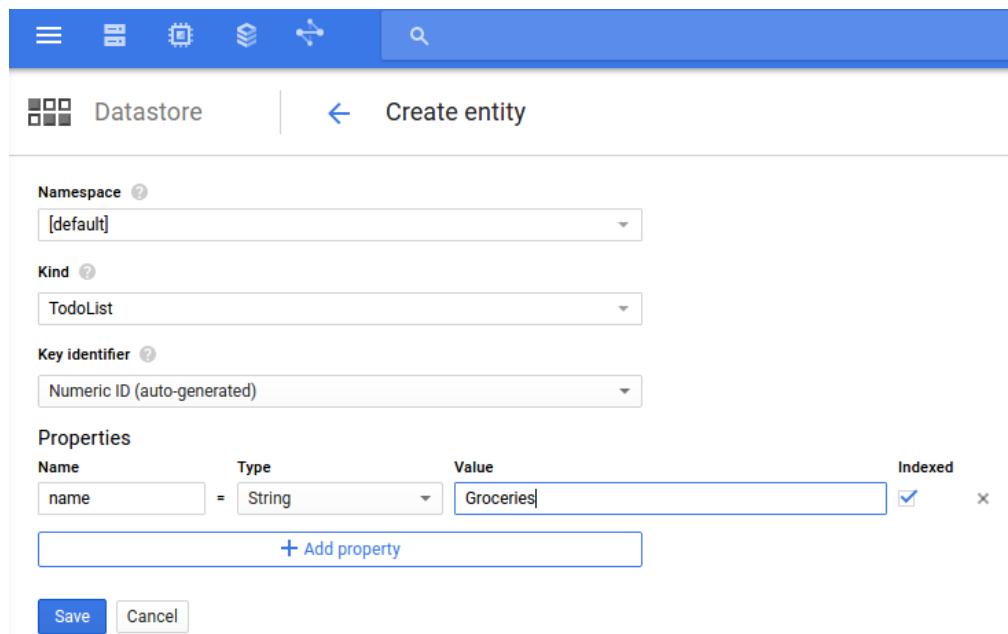
Once the API is enabled, jump to the Datastore UI from the left navigation and let's start by going back to our to-do list example and explore how this might look in Cloud Datastore.

Let's start by creating the "Todo List" entity. Notice that unlike a SQL database, we're

starting by creating some data rather than defining a schema. This is the nature of document-oriented databases and while it might seem strange at first, it's pretty typical for non-relational storage. You should see a big blue "Create Entity" button when you first visit the Datastore page, so let's start by clicking on that.

Next, let's leave our entity in the "[default]" namespace (we'll discuss namespaces a bit later), make it a "TodoList" kind, and we'll let Datastore automatically assign a numerical ID. After that, let's give our TodoList entity a "name". To do this, click the "Add property" button, set the name of the property to "name", leave the property type set to "String", and fill in the value of the property (in this case, the name of the list). In this example, the list is called "Groceries". Also note that since we may want to search based on this name we'll leave the property "indexed" (marked by the check box).

**Figure 5.4. Create the Groceries TodoList**



Then just click "Save" and you should see a newly created TodoList entity in your browser.

**Figure 5.5. Your TodoList entities**

The screenshot shows the Google Cloud Datastore interface. On the left is a sidebar with icons for Datastore, Dashboard, Entities (which is selected), Indexes, and Admin. The main area has a header with 'Entities', 'CREATE ENTITY', 'REFRESH', and 'DELETE' buttons. Below the header is a search bar with 'Query by kind' and 'Query by GQL'. A 'Kind' dropdown is set to 'TodoList' with a 'Filter entities' button next to it. A table below shows one entity with columns 'Name/ID' and 'name'. The entity row contains 'Id=5629499534213120' and 'Groceries'.

Let's take a moment now and look at how to interact with this entity in your own code. If you followed the tutorial in [chapter 1](#) you should already have all the right tools installed, but to get the library for Cloud Datastore, you'll need the `@google-cloud/datastore` package which you can install by running `$ npm install @google-cloud/datastore@0.4.0`. Let's look at how we can query for all of the lists in our Datastore instance.

Here's a quick Node.js script that asks Datastore for all of the `TodoList` entities and prints them to the screen.

**NOTE**

If you get an error saying "Not Authorized", make sure that you've run `gcloud auth application-default login` and have authenticated successfully.

**Listing 5.13. Query Cloud Datastore for all TodoList entities**

```
const datastore = require('@google-cloud/datastore')({
  projectId: 'your-project-id'
});

const query = datastore.createQuery('TodoList'); ①

datastore.runQuery(query)
  .on('error', console.error)
  .on('data', (entity) => {
    console.log('Found TodoList:\n', entity);
  })
  .on('end', () => {
    console.log('No more TodoLists');
});
```

- ① We can start by creating the Query object.
- ② Once that's created, we run the query and register listeners to handle data as it's found.

The output of this script should be something like this:

**Listing 5.14. Results of the found TodoList entities**

```
Found TodoList:
{ key:
  Key {
    namespace: undefined,
    id: 5629499534213120,
    kind: 'TodoList',
    path: [Getter] },
  data: { name: 'Groceries' } }
No more TodoLists
```

As you can see your grocery list is returned with the name you stored. Now let's try creating a TodoItem using the hierarchical key structure we described above. In this example, our grocery list items will have keys using the list as their parent.

**Listing 5.15. Creating a new TodoItem**

```
const datastore = require('@google-cloud/datastore')({
  projectId: 'your-project-id'
});

const entity = {
  key: datastore.key(['TodoList', 5629499534213120, 'TodoItem']), ①
  data: {
    name: 'Milk',
    completed: false
  }
};

datastore.save(entity, (err) => {
  if (err) {
    console.log('There was an error...', err);
  } else {
    console.log('Saved entity:', entity);
  }
});
```

① The number here is the ID that you got before when querying for TodoLists.

When you run this script you should see output that looks something like this.

**Listing 5.16. Output of saving the new TodoItem**

```
Saved entity: { key:
  Key {
    namespace: undefined,
    kind: 'TodoItem',
    parent:
      Key {
        namespace: undefined,
        id: 5629499534213120,
        kind: 'TodoList',
        path: [Getter] },
```

```
path: [Getter],
id: 5629499534213120 },
data: { name: 'Milk', completed: false } }
```

Take special notice of the key property which has a "parent" key pointing to your TodoList entity. Also note that the key has an automatically generated ID for you to reference later. Let's add a few more items to the grocery list with a script, but this time we'll save several of them in a single API call.

#### **Listing 5.17. Adding more items to the TodoList**

```
const itemNames = ['Eggs', 'Chips', 'Dip', 'Celery', 'Beer'];
const entities = itemNames.map((name) => {
  return {
    key: datastore.key(['TodoList', 5629499534213120, 'TodoItem']),
    data: {
      name: name,
      completed: false
    }
  };
});

datastore.save(entities, (err) => {
  if (err) {
    console.log('There was an error...', err);
  } else {
    entities.forEach((entity) => {
      console.log('Created entity', entity.data.name, 'as ID', entity.key.id);
    })
  }
});
```

When you run this, you should see that your entities were created and given IDs.

#### **Listing 5.18. Output of creating more TodoItems**

```
Created entity Eggs as ID 5707702298738688
Created entity Chips as ID 5144752345317376
Created entity Dip as ID 6270652252160000
Created entity Celery as ID 4863277368606720
Created entity Beer as ID 5989177275449344
```

Now let's go back to the Cloud Console and query for all of the items in our grocery list. As you might recall, this is done by querying for the items that are "descendants" of the TodoList entity (that is, they have this entity as an ancestor), and you express this in GQL as:

#### **Listing 5.19. GQL query to list items belonging to a single list**

```
SELECT * FROM TodoItem
WHERE __key__ HAS ANCESTOR Key(TodoList, 5629499534213120)
```

If you run this query using the GQL tool in the Cloud Console, you should see that all

of your grocery items are in your list.

**Figure 5.6. Viewing the items to buy at the grocery store.**

The screenshot shows a user interface for querying a database. At the top, there are buttons for 'CREATE ENTITY', 'REFRESH', and 'DELETE'. Below that, tabs for 'Query by kind' and 'Query by GQL' are visible, with 'Query by GQL' being selected. A code editor contains the following GQL query:

```
SELECT * FROM TodoItem
WHERE __key__ HAS ANCESTOR Key(TodoList, 5629499534213120)
```

To the right of the code editor, there is a dropdown menu labeled 'Number of columns to display' set to 50. Below the code editor are three buttons: 'Run query', 'Clear query', and 'GQL query help'. The results section displays a table with the following data:

<input type="checkbox"/> Name/ID	completed	name
<input type="checkbox"/> Id=4863277368606720	false	Celery
<input type="checkbox"/> Id=5144752345317376	false	Chips
<input type="checkbox"/> Id=5629499534213120	false	Milk
<input type="checkbox"/> Id=5707702298738688	false	Eggs
<input type="checkbox"/> Id=5989177275449344	false	Beer
<input type="checkbox"/> Id=6270652252160000	false	Dip

Let's check one of these items off the list, and then see if we can ask for only the uncompleted ones. Start by clicking on the item in the query results and then changing the `completed` field from `False` to `True`. Then just click "Save".

**Figure 5.7. Crossing "Beer" off the list**

The screenshot shows the 'Edit entity' screen for a TodoItem. At the top, there are buttons for 'Edit entity', 'REFRESH', and 'DELETE'. Below this, entity details are listed:

- Namespace: [default]
- Kind: Todoltem
- Key: TodoList id:5629499534213120 > TodoItem id:5989177275449344
- Key literal: Key(TodoList, 5629499534213120, TodoItem, 5989177275449344)
- URL-safe key: ahZzfmmdjcGlhLWRhdGFzdG9yZS10ZXN0cioLEghUb2RvTGlzdBiAgICAgICACgwL EghUb2RvSXRLbRiAgICAgOTRCgw

**Properties**

Name	Type	Value	Indexed
completed	Boolean	True	<input checked="" type="checkbox"/> <input type="button" value="x"/>
name	String	Beer	<input checked="" type="checkbox"/> <input type="button" value="x"/>

[+ Add property](#)

At the bottom are 'Save' and 'Cancel' buttons.

Now let's go back to code and see how we might query for all of the things we still need to buy at the grocery store. Notice that the query object has three important pieces which are listed below.

#### Listing 5.20. Querying for all uncompleted TodoItems in our list

```
const datastore = require('@google-cloud/datastore')({
  projectId: 'your-project-id'
});

const query = datastore.createQuery('TodoItem')
  .hasAncestor(datastore.key(['TodoList', 5629499534213120]))  
    1
  .filter('completed', false);  
    2
    3

datastore.runQuery(query)
  .on('error', console.error)
  .on('data', (entity) => {
    console.log('You still need to buy:', entity.data.name);
  });

```

- ➊ The Kind you're querying (TodoItem)
- ➋ The "parent" key (the TodoList entity)
- ➌ The filter for completed = false

When you run this, you should see that everything you added before is on the list except for the "Beer" item which you marked as completed.

**Listing 5.21. Output of querying for uncompleted TodoItems**

```
You still need to buy: Celery
You still need to buy: Chips
You still need to buy: Milk
You still need to buy: Eggs
You still need to buy: Dip
```

Now that we've explored a bit about how to interact with Cloud Datastore, let's look at how we might go about backing-up and restoring our data.

## **5.4 Back-up and restore**

Back-ups are one of those things that we tend to not really need until we **really** need them, particularly when we accidentally delete a bunch of data. Cloud Datastore back-ups are a bit unusual in that they're not really "back-ups" in the sense that we've gotten used to them. This is mainly because of Datastore's eventually consistent queries which we learned about before, which makes it pretty difficult to get the overall state of the data at a single point in time. Instead, asking for "all the data" tends to be more of a *smear* over the time that it takes the query to run.

What does this actually mean? First, the "back-up" ability of Datastore is really more of an "export" that is able to take a bunch of data from a regular Datastore query and ship it off to a Cloud Storage bucket. However, since a regular Datastore query is eventually consistent this means that the data exported to Cloud Storage could be equally inconsistent. For example, if we were to create a new entity every second, a "backup" of the data after 10 seconds could end up storing exactly the 10 entities, but could also store more than 10 entities. More confusingly, we might end up seeing fewer than 10!

Because of this, it's important to remember that exports are **not** a snapshot taken at a single point in time. and instead are more like a long-exposure photograph of your data. To minimize the effect of this long-exposure, it's possible to disable Datastore writes beforehand and then re-enable them once the export completes. With all that said, let's look at how we can actually export our data.

**NOTE**

As of this writing, this feature of Datastore is "Beta", meaning that the commands we'll run will start with `gcloud beta`.

First, we'll need a Cloud Storage bucket which is explained in [chapter 8](#). For now, just consider it a place that will hold our exported data, which we interact with using the `gsutil` command that comes with the Cloud SDK command-line tool.

**Listing 5.22. Creating a Cloud Storage bucket**

```
$ gsutil mb -l US gs://my-data-export
Creating gs://my-data-export/...
```

Once the bucket is created, we can disable writes to our Datastore instance using the

Cloud Console, using the "Admin" tab in the Datastore section.

**Figure 5.8. Disabling writes to Datastore using the Cloud Console**

Datastore	Admin
<a href="#"> Entities</a> <a href="#"> Dashboard</a> <a href="#"> Indexes</a> <a href="#"> Admin</a>	<b>Datastore Admin</b> Use Datastore Admin to back up, restore, copy, and delete entities in bulk. <a href="#">Learn more</a> <a href="#">Enable Datastore Admin</a>  <b>Datastore writes</b> Writes are currently enabled for this Datastore instance. Disabling writes will cause all Datastore writes to fail. <a href="#">Disable writes</a>

After that, we can trigger an export of our data into our bucket using the `datastore export` sub-command, shown below.

#### **Listing 5.23. Exporting data to Cloud Storage**

```
$ gcloud beta datastore export gs://my-data-export/export-1
Waiting for [projects/your-project-id-
here/operations/ASA1MTIwNzE4OTIJGnRsdWFmZWQHEmxhcnRuZWNzdS1zYm9qLW5pbWRhFAosEg] to
finish...done.
metadata:
  '@type':
type.googleapis.com/google.datastore.admin.v1beta1.ExportEntitiesMetadata
  common:
    operationType: EXPORT_ENTITIES
    startTime: '2018-01-16T14:26:02.626380Z'
    state: PROCESSING
    outputUrlPrefix: gs://my-data-export/export-1
name: projects/your-project-id-
here/operations/ASA1MTIwNzE4OTIJGnRsdWFmZWQHEmxhcnRuZWNzdS1zYm9qLW5pbWRhFAosEg
```

Once that completes we can verify that data arrived into our bucket again using the `gsutil` tool.

#### **Listing 5.24. Viewing the size of the export data**

```
$ gsutil du -sh gs://my-data-export/export-1
32.2 KiB  gs://my-data-export/export-1  ①
```

- ① We can see here that everything inside the `export-1` directory takes up about 32 kilobytes of space.

Now that we can see the export is complete, we can start talking about the other half of this puzzle: restoring.

Similar to how "backing up" is really "exporting", "restoring" is actually "importing", which raises a couple of topics worth mentioning. First, importing entities will use all the same IDs as before, and will therefore overwrite any entities that use that same ID. This means that if there are any accidental ID collisions, those entities will be overwritten. This should only be a problem if you choose your own IDs, but it's worth knowing. Second, since this is an "import" rather than a "restore", any entities that were created after the previous export (and are therefore unaffected by the import) will still remain. In other words, the import can edit and create entities, but will never delete any entities.

To run an import, we can do the same thing as above, remembering first to disable writes ahead of time. The only difference this time is that instead of pointing to a directory of where the data will live, we'll need to point to the metadata file that was created during the export. We can find this metadata file using the `gsutil` command once again, shown below.

#### **Listing 5.25. Listing the objects created by the export**

```
$ gsutil ls gs://my-data-export/export-1 ①
gs://my-data-export/export-1/export-1.overall_export_metadata ②
gs://my-data-export/export-1/all_namespaces/
```

- ① Here we use `gsutil` to list the objects that were created by the export job.
- ② Here we can see the export metadata created, which we'll reference during an import.

Now that we have the path to the metadata file for the export, we can trigger an import job using the `gcloud` command similar to before.

#### **Listing 5.26. Importing data from a previous export**

```
$ gcloud beta datastore import gs://my-data-export/export-1/export-
1.overall_export_metadata
Waiting for [projects/your-project-id-
here/operations/AiA4NjUwODEzOTIJGnRsdWFmZWQHEmxhcnRuZWNzdS1zYm9qLW5pbWRhFAosEg] to
finish...done.
metadata:
  '@type':
type.googleapis.com/google.datastore.admin.v1beta1.ImportEntitiesMetadata
  common:
    operationType: IMPORT_ENTITIES
    startTime: '2018-01-16T16:26:17.964040Z'
    state: PROCESSING
    inputUrl: gs://my-data-export/export-1/export-1.overall_export_metadata
  name: projects/your-project-id-
here/operations/AiA4NjUwODEzOTIJGnRsdWFmZWQHEmxhcnRuZWNzdS1zYm9qLW5pbWRhFAosEg
```

At this point, if we had made changes to any of the entities (or deleted any entities) those entities would be reverted to how they were at the time of the export. However, if we had created new entities, they would be left entirely alone because an import doesn't affect entities it hadn't seen before.

Now that we have a good grasp on using Cloud Datastore, let's look in more detail at how much all of this will cost us.

## 5.5 Understanding pricing

Cloud Datastore is priced based on two different things: the amount of data you store and the number of "operations" you perform on that data. Let's pick off the easy part first: storage.

### 5.5.1 Storage costs

Data stored in Cloud Datastore is measured in GB, costing \$0.18 per GB per month as of this writing. This might sound pretty straight forward, but it's actually a bit more complicated than it looks. In addition to just your actual data (the property values on your entities), the total storage size for billing purposes of a single "entity" includes the kind name (e.g., "Person"), the key name (or ID), all property names (e.g., "favoriteColor"), and 16 extra overhead bytes. Further, all properties have simple indexes created where each index entry includes the kind name, the key name, the property name, the property value, and 32 extra overhead bytes. Finally, don't forget that there are indexes for both ascending and descending order.

In short, long names (indexes, properties, and keys) tend to explode in size and mean that you'll have far more total data than just the actual data stored. For lots of detail about how the total storage size is computed, take a look at the only storage reference: [cloud.google.com/datastore/docs/concepts/storage-size](https://cloud.google.com/datastore/docs/concepts/storage-size). This is particularly important if you expect to have a lot of entities and indexes to query over those entities.

Now let's talk about the other pricing aspect, which in retrospect is much more straight forward: operations.

### 5.5.2 Per-operation costs

Operations, in short, are any requests that you send to Cloud Datastore, such as creating a new entity, or retrieving data. Cloud Datastore charges based on how many entities are involved in a given operation, at different rates for different types of operations. This means that some operations (such as updating or creating an entity) cost more than others (such deleting an entity). The price break-down is shown below.

**Table 5.7. Operation pricing breakdown**

Operation type	Cost per 100,000 entities
Read	\$0.06
Write	\$0.18
Delete	\$0.02

Unlike storage totals, there are far fewer "gotchas" in this type of pricing. For example, if you retrieve 100,000 of your entities your bill will be 6 cents. Similarly for updating and deleting those entities (totaling 18 and 2 cents respectively). The only thing to worry about is queries which involve retrieving each entity in the query. In other

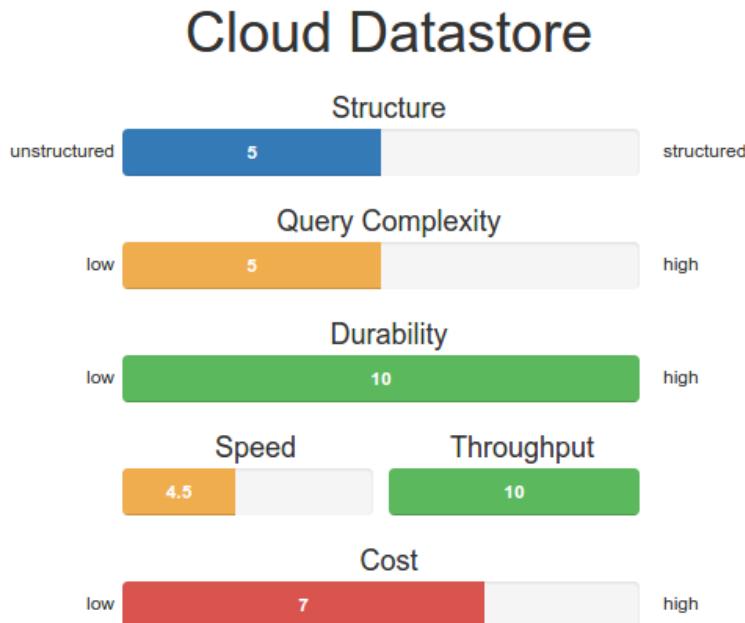
words, if you run a query selecting all of your entities, that will count as a "read" operation on each entity returned to you. If all you want is to look at the *key* of your entities, you can use a "keys-only" query, which is a free operation.

Now that we've got a grasp on how Datastore pricing works, it's time to think about when Cloud Datastore is a good fit for your projects.

## 5.6 When should I use Cloud Datastore?

Let's start with a scorecard to summarize some of the strong and weak points of Cloud Datastore. Notice that the two places where Datastore really shines are durability and throughput, and that cost is entering into the danger zone.

**Figure 5.9. Cloud Datastore scorecard**



### 5.6.1 Structure

As you learned, unlike relational databases Cloud Datastore excels at managing "semi-structured" data where attributes have types, however there is no single "schema" across all entities (or *documents*) of the same kind. You might choose to design your system such that entities of a single kind are homogeneous, but that's up to you to enforce in your application code.

Along with the document-style storage Datastore also allows you to express the locality of your data using hierarchical keys (where one key is prefixed with the key of its parent). This can be confusing but reflects the desire to segment data between units of isolation (e.g., a single user's e-mails). This aspect of Datastore, which enables

automatic replication of your data, is what allows Datastore to be so highly available as a storage system. While this provides many benefits, it also means that queries across all the data will be eventually consistent.

### **5.6.2 Query complexity**

As with any non-relational storage system, the typical relational aspects (e.g., the `JOIN` operator) are not supported. Cloud Datastore allows you to store keys which act as pointers to other stored entities, however provides no management for these values. Most noticeably this means that there is no referential integrity and no ability to cascade or limit changes involving referenced entities. In other words, when you delete an entity in Cloud Datastore, anywhere you pointed to that entity from elsewhere is effectively an invalid reference.

Further, certain queries require that you have indexes to enable those queries, which is somewhat different from a relational database where indexes are helpful but not necessary to run specific queries. Some of these limitations are the consequence of the structural requirements that went into designing Cloud Datastore, while other limitations enable consistent performance for all queries.

### **5.6.3 Durability**

Durability is where Cloud Datastore really starts to excel. Since Megastore was built on the premise that we can "never lose data", everything is automatically replicated and not considered "saved" until saved in several different places. While you have various levels of self-management for replication when using a relational database (even Cloud SQL requires that you configure your replicas), Datastore handles this entirely on its own, meaning that the only setting for durability is "as high as possible".

This fact combined with the indexes aspect discussed previously has an unfortunate side effect of global queries being "eventually consistent", which means that since your data needs to replicate to several places before being called "saved" there may be times where a query across all data may return stale results as it takes addition time for the indexes to be updated alongside the data.

### **5.6.4 Speed (latency)**

Compared to many in-memory storage systems (e.g., Redis) Cloud Datastore simply won't be as fast for the simple reason that even SSDs are slower than RAM. Compared to a relational database system like PostgreSQL or MySQL, Cloud Datastore will be in the same ballpark with one primary difference: as your SQL database gets larger or receives more requests at the same time, it will likely get slower. As you learned in this chapter Cloud Datastore's latency stays the same regardless of the level of concurrency, and the time a query takes to run scales with the size of the result set rather than the amount of data that needs to be sifted through.

The key thing to takeaway from this section is that Cloud Datastore certainly won't be blazing fast like in-memory NoSQL storage systems, however it will be on par with other relational databases and will remain consistent as you increase your levels of

concurrency as well as the size of your data.

### 5.6.5 *Throughput*

As you learned already, Cloud Datastore's throughput benefits from running on Google's infrastructure as a fully managed storage service which means that it can accommodate as much traffic as you care to throw at it. Because your data is automatically spread out across different groups (unless you specifically say not to do so), this means that the pessimistic locking that comes with relational database like MySQL doesn't apply, and instead you're able to scale up to many concurrent write operations.

This fact also means that should you ever grow so large that even Google has trouble supporting your traffic, it's simply a matter of adding more servers on Google's side to keep up. Compared to MySQL's throughput story, while reads can be dealt with using read-replicas, scaling up the number of concurrent write operations executing is quite a challenge. Cloud Datastore makes this something you don't have to worry about.

### 5.6.6 *Overall*

Now that you have an idea of where Cloud Datastore starts to do well, let's take our example applications and see whether Datastore is a good fit.

#### *To-do list*

As a "starter-app", your to-do list definitely won't need the high levels of throughput that Datastore can provide, however being a fully-managed offering it brings some interesting things to the table.

**Table 5.8. To-do list application storage needs**

Aspect	Needs	Good fit?
Structure	Structure is fine, not necessary though	Sure
Query complexity	We don't have that many fancy queries	Definitely
Durability	High, we don't want to lose stuff	Definitely
Speed	Not a lot	Definitely
Throughput	Not a lot	Sure

In short, Cloud Datastore is an acceptable fit, however it is a bit overkill on the scalability side. This is sort of like giving your grandmother a Lamborghini. It will get her to the grocery store just fine, but she probably won't be drag racing on her way there.

If this to-do list application could become something enormous, then Datastore is a very safe bet to go with as it means that "scaling to handle tons of traffic" is something you don't really need to worry about all that much.

#### *E\*Exchange*

E\*Exchange, the online trading platform, is a bit more complex compared to the to-do

list application. Specifically, the main difference is in the complexity of the queries that customers are likely to need.

**Table 5.9. E\*Exchange storage needs**

Aspect	Needs	Good fit?
Structure	Yes, reject anything suspect, no mistakes	Maybe
Query complexity	Complex, we have fancy questions to answer	No
Durability	High, we <b>cannot</b> lose stuff	Definitely
Speed	Things should be pretty fast	Probably
Throughput	High, we may have lots of people using this	Definitely

Looking at this table, Cloud Datastore is probably not the best fit for E\*Exchange if used on its own. For example, Cloud Datastore doesn't enforce strict schema requirements, however E\*Exchange wants clear validation of any data entering the system. To do this, you'd have to enforce that schema in your application rather than relying on the database. So while it's possible to do, it's not built into Datastore. Further, you learned that Datastore cannot do extremely complex queries, specifically things like joining two separate tables together. This means that, again, Datastore on its own is unlikely to be a good fit.

If E\*Exchange was hoping to benefit from Datastore's high durability, replication, and throughput abilities, it'd likely make the most sense to store the raw data in Datastore while using some sort of data warehouse or analytical storage engine for running the more complex queries. In other words, E\*Exchange would store each single trade as an entity, which would scale to extremely high throughput and always maintain high durability, while storing the analytical data in something like BigQuery (see Chapter X), or one of the many time-series databases such as HBase, InfluxDB, or OpenTSDB.

It's also important to mention that the fact that Datastore offers full ACID transaction semantics means that you never have to worry about multiple updates accidentally ending up in a half-committed state. For example, transferring shares would be an atomic transaction that decreases the seller's balance and increases the buyer's balance, and you don't have to worry that one of those changes will be committed while another is lost due to a failure of some sort.

### **InstaSnap**

InstaSnap, the very popular social media application, has a few requirements that seem to fit really well and only a couple that are a bit off.

**Table 5.10. InstaSnap storage needs**

Aspect	Needs	Good fit?
Structure	Not really, structure is pretty flexible	Definitely
Query complexity	Mostly look-ups, no highly complex questions	Definitely
Durability	Medium, losing things is inconvenient	Sure

Speed	Queries must be very fast	Maybe
Throughput	Very high, Kim Kardashian uses this	Definitely

The biggest issue for an app like InstaSnap is the single query latency, which needs to be extremely fast. This is yet another place where Datastore on its own isn't the best fit, however, if used in conjunction with some sort of in-memory cache like Memcache, this problem goes away entirely. Additionally, while InstaSnap's durability needs are not all that serious, the fact that Datastore provides higher levels than needed isn't such a big deal.

In short, InstaSnap is actually a pretty solid fit due to the relatively simple queries combined with the enormous throughput requirements. As a matter of fact, SnapChat (the real app) actually uses Datastore as one of its primary storage systems.

### 5.6.7 Other document storage systems

As a document storage system, Cloud Datastore is one of many options: from the other hosted services like Amazon's DynamoDB to the many open-source alternatives like MongoDB or Apache HBase (you'll learn more about HBase's parent system called Bigtable in [chapter 7](#)). This means that there are a few different aspects to choose from, each with their own benefits and drawbacks. In some cases, systems can act a bit like document-storage systems in certain configurations, even if they weren't designed for that.

The following is a table that attempts to summarize these different document storage systems and when you might want to choose one over another.

**Table 5.11. Brief comparison of document storage systems**

Name	Cost	Flexibility	Availability	Durability	Speed	Throughput
Cloud Datastore	High	Medium	High	High	Medium	High
MongoDB	Low	High	Medium	Medium	Medium	Medium
DynamoDB	High	Low	Medium	Medium	High	Medium
HBase	Medium	Low	Medium	High	High	High
Cloud Bigtable	Medium	Low	High	High	High	High

Notice that while it's possible to configure systems like HBase and MongoDB in "high-availability" configurations, when that happens, cost will go up significantly. You can read more on this in "[What's the difference between Bigtable and HBase?](#)", but now that you have a grasp on how Datastore stacks up let's take a look at pricing to see how much that overall cost actually is.

## 5.7 Summary

- Document storage keeps data organized as heterogeneous (jagged) documents rather than homogeneous rows in a table.
- Using document storage effectively may involve duplicating data for easy access (de-normalizing).

- Document storage is great for storing data that may grow to huge sizes and huge amounts of traffic, however it comes with the cost of not being able to do fancy queries (e.g., "joins" that we do in SQL).
- Cloud Datastore is a fully-managed storage system with automatic replication, result-set query scale, full transactional semantics, and automatic scaling.
- Cloud Datastore is a good fit if you need high scalability, and have relatively straight forward queries.
- Cloud Datastore charges for operations on entities, meaning the more data you interact with, the more you pay.



### This chapter covers:

- What is NewSQL?
- What is Spanner?
- Administrative interactions with Cloud Spanner
- Reading, writing, and querying data
- Interleaved tables, primary keys, and other advanced topics

## 6.1 What is NewSQL?

So far we've looked relational (SQL) databases and non-relational (NoSQL) databases, and learned about some of the trade-offs of each. On the one hand, SQL databases generally provide richer queries, strong consistency, and transactional semantics, but have trouble handling massive amounts of traffic. On the other, NoSQL databases tend to trade some or all of these in exchange for horizontal scalability, which allows the system to easily handle more traffic by simply adding more machines to the cluster. Obviously the choice you make between SQL or NoSQL will depend on your business needs, but this begs the question: wouldn't it be nice if you didn't have to make that choice at all? What if you could have rich querying, transactional semantics, strong consistency, **and** horizontal scalability? These types of systems are sometimes referred to as "NewSQL" databases.

NewSQL databases may deviate from the standard SQL we've become used to, but NewSQL databases look and act a lot like SQL databases but have the scaling properties of NoSQL databases. For example, a NewSQL database may require that data locality be expressed in the schema somehow, but you can still query your data

using familiar `SELECT * FROM ...` syntax. Let's explore a bit of Google's history in this area and what came out in an attempt to solve this problem.

## 6.2 What is Spanner?

For a long time many of Google's needs were no different than any other business, where data was structured and relational, and fit comfortably in MySQL. As the size of the data stored grew out of control, this obviously became a problem. The first step towards fixing this was to push the off-the-shelf databases beyond where they were designed to perform, sharding data and hiring lots of database administrators to fine-tune the system. This helped but didn't solve the problem, and the data just kept growing.

And unfortunately, using one of the in-house storage systems (like Megastore) wouldn't really work because the features needed were things like transactional or relational semantics as well as strong consistency, and those were exactly the features that were traded first when designing things like Megastore. At this point it was clear that there was a need for a system that combined the scalability of non-relational storage and added the features of a traditional MySQL database, leading to Spanner.

Spanner is a NewSQL database which offers many of the features of a relational database (like schemas and `JOIN` queries) with many of the scaling properties of a non-relational database (like being able to "throw more machines at the problem"). In the case of failures (or exceptionally large load) Spanner is capable of splitting and re-distributing data across more machines, even if those machines are in entirely separate data centers. This dynamic resizing and shuffling of data chunks means that the system is prepared for all types of disasters.

To top it all off, Spanner offers strongly consistent queries which means that you'll never have a stale version of the data. Following the pattern of Google Cloud Platform, Google has taken the Spanner database which so far has only been available to Google engineers, and made it available to anyone using Google Cloud Platform as a hosted storage system much like Cloud Datastore or Cloud Bigtable. Let's dive right into some of the concepts to see how you go about actually using Cloud Spanner.

## 6.3 Concepts

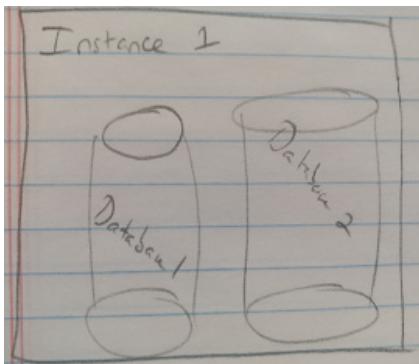
As with any storage system there are a few underlying concepts that you should understand before getting started. In this section we'll explore a few of those, starting with the infrastructural concept of an "instance", and then dive into the data-model concepts like tables and keys. Along the way, we'll touch on some of the more theoretical concepts like split points and transactions, which are relevant when digging into how best to use Spanner to get the best performance possible. Let's dive right in with instances.

### 6.3.1 Instances

In its most basic form, a Cloud Spanner instance acts as an infrastructural container

that holds a bunch of databases, but it also manages multiple discrete units of computing power which are ultimately responsible for serving your Spanner data. This means that there are two different aspects to Spanner instances: a data-oriented aspect and an infrastructural aspect. Let's start by exploring the data-oriented side of things.

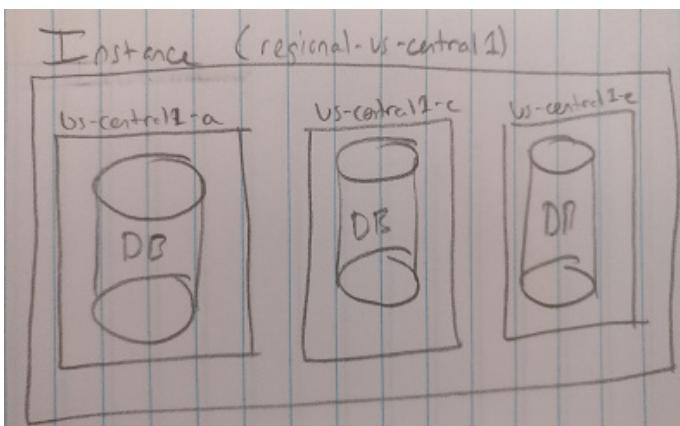
**Figure 6.1. At a high level, instances are containers for databases**



When we want to run a query and get back results, an instance acts as nothing more than a database container, sort of like a Cloud SQL instance. This means that when you run a query, you simply route it to the instance itself, and Spanner will do the heavy lifting. So what about the infrastructural side?

Unlike a single MySQL instance, Spanner instances are automatically replicated. This means that rather than choosing a specific zone where the instance will live, you choose a configuration which maps to some combination of different zones. For example, the `regional-us-central1` configuration represents some combination of zones inside the `us-central1` region. In other words, Spanner instances do have geographical "homes", however the location is much more general than the home of, say, a Compute Engine VM.

**Figure 6.2. Instance configurations determine the zones that data is replicated to**

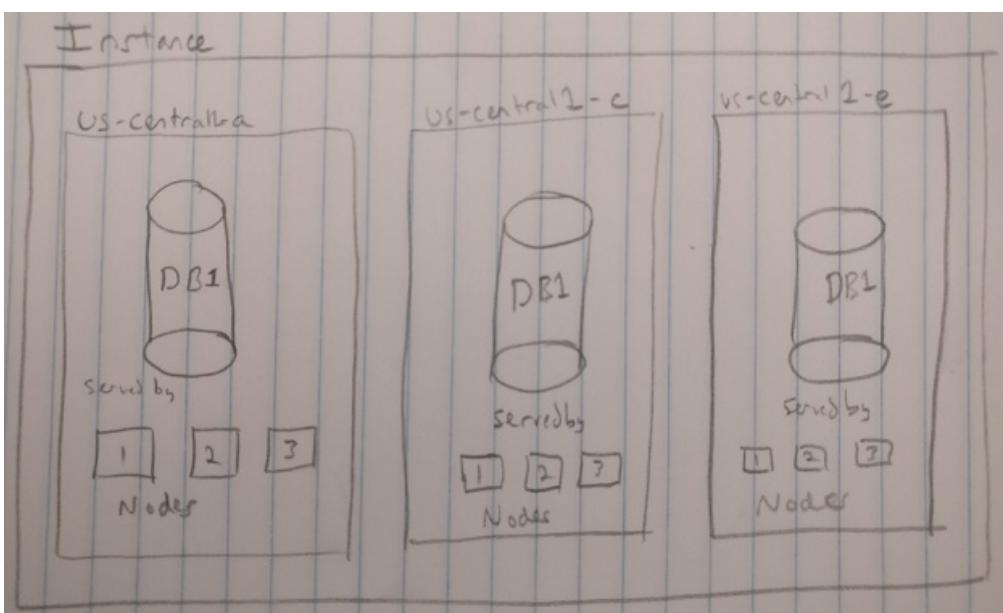


Now that you understand this dual nature of instances, let's look a bit more deeply at the physical component that makes up the computing power of an instance: a node.

### 6.3.2 Nodes

In addition to acting like containers of databases, and being replicated across multiple different zones, Spanner instances are made up of specific number of nodes that can be used to serve instance data. These nodes are the things that live in specific zones and are ultimately responsible for handling queries over the data. Since Spanner instances are fully replicated this means that you have identical replicas (that is, a collection of the same number of nodes) in each of the different zones. This ensures that if any zone has an outage of some sort, your data will continue serving without any problems.

**Figure 6.3. Instances have the same number of nodes in every replica.**



This type of replication means that if you have a 3-node instance in a regional configuration (replicated across 3 zones), you actually have 9 nodes in total since each replica is a copy of both the data and the serving capacity. While this might seem like overkill, recall that Spanner's guarantees are focused on rich querying, globally strong consistency, and high availability and performance. Notably missing from this is "low cost", as Spanner overcomes many of these issues by throwing more resources at the problem. Now that you understand instances and the replication configurations, let's move on a bit lower in the stack and explore how databases work.

### 6.3.3 Databases

Much like databases in MySQL, databases are primarily containers of tables. Typically a single database will act as a container of data for a single product which makes it

easy to do things like limit access permissions or atomically drop all data. Under the hood, databases are also the container that we'll use to make schema changes and query for data. Besides acting as the logical container of tables, there's not a whole lot about databases that we need to explore here, so let's dig a tiny bit deeper and scratch the surface of what Spanner tables are and how they work.

### 6.3.4 Tables

In most ways, Spanner tables are similar just like other relational databases, however there are some very important differences. Let's start by talking about what's the same, and then we'll explore the big differences later on in the chapter.

To start, tables have a schema which looks a lot like those of any other relational database. Tables have columns which have types (such as INT64) and modifiers (such as NOT NULL), and define the "shape" of your data. Just like a relational database, trying to add data that doesn't match the type defined in the schema will result in an error. Additionally, there are a few other constraints, such as a maximum cell size of 10 MiB, but in general Spanner tables shouldn't be very surprising. To demonstrate just how similar Spanner tables can be to other databases, let's look at an example and compare the two schema definitions. Below you'll see a table for storing employee records, which is valid for defining a table in MySQL.

#### **Listing 6.1. Storing employee IDs and names**

```
CREATE TABLE employees (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    start_date DATE
);
```

Below is an example of creating the same table in Cloud Spanner.

#### **Listing 6.2. Storing employee IDs and names in Spanner**

```
CREATE TABLE employees (
    employee_id INT64 NOT NULL,
    name STRING(100) NOT NULL,
    start_date DATE
) PRIMARY KEY (employee_id);
```

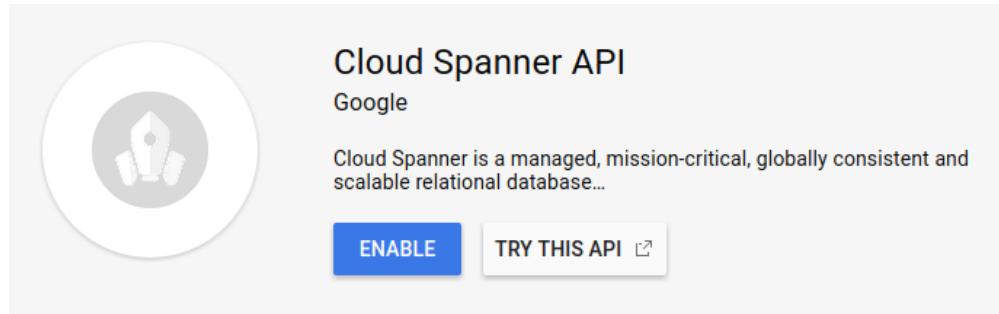
As you can see, these tables are almost identical, with some small differences in data type names and location of the primary key directive. Since we have enough background information to take Spanner for a test drive, so let's explore how to use Spanner, and then come back later to explore some of the more advanced topics.

## 6.4 Interacting with Cloud Spanner

Before we can store any data in Cloud Spanner, we first have to create some of the infrastructural resources. Let's start by doing that in the Cloud Console. As always, we have to start by enabling the Cloud Spanner API. To do that, in the Cloud Console type

in "Cloud Spanner API" in the main search box at the top of the page, which should show only one result. Click on that and you'll land on a page with a big "Enable" button. Once you click that, you should be good to go.

**Figure 6.4. Enable the Cloud Spanner API**



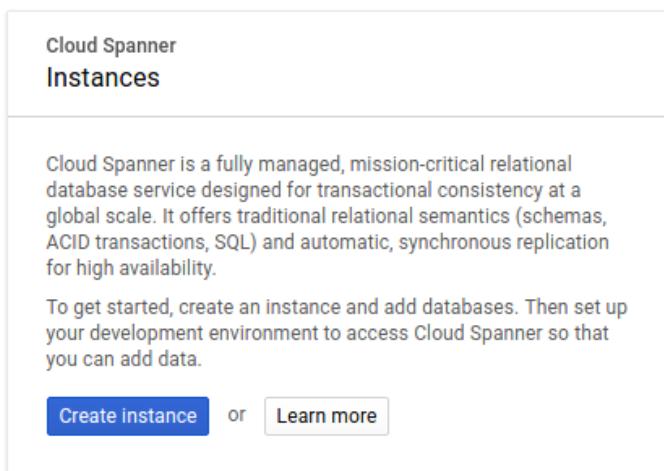
Once that's all done, head over to the Spanner interface by clicking on "Spanner" in the left-side navigation (in the "Storage" section).

#### 6.4.1 Creating an instance and database

**NOTE** While Cloud Spanner is very powerful, it can also be very expensive. This means that if you turn on an instance in this tutorial, don't forget to turn it off afterwards or you may get a bigger bill than you expected!

When you first start with Spanner, since you don't have any databases, you'll see a big prompt asking you to create a Spanner instance.

**Figure 6.5. The prompt you'll see on your first visit to the Spanner UI**



Since that's what we want to do, click on "Create instance" and you'll be brought to a

form where you can choose some of the details for your Spanner instance. For this example, let's call the instance "Test Instance". When you type that into the first field, you should notice that the field for the instance ID automatically fills in with a simplified version of the name. The first field is actually just the display name that you'll see in the UI, and the second field is the official ID of the instance that you'll need when addressing it in any API calls.

After that, we need to choose the configuration. As you learned earlier, Spanner configurations are sort of like Compute Engine zones, and have to do with the availability aspect. Just like with a VM, since we're going to be accessing the Spanner instance from your local machine, it's a good idea to choose a configuration that is geographically nearby to you. Additionally, when you're using your instance in production, you should generally have the VMs accessing Spanner in the same region as the instance itself. In other words, if you deploy your Spanner instance in the `us-central1` configuration, you'll want to put your VMs in `us-central1` zones (such as `us-central1-a`).

Lastly, for the purposes of our test unless you're looking to run a benchmark or performance test, leave the number of nodes set to 1. Under the hood, this is actually going to result in having 3 node replicas spread across 3 different zones (one node in each zone), which is plenty of capacity for our test.

**Figure 6.6. Creating a Spanner instance**

[← Create an instance](#)

---

**Instance name**  
For display purposes only.

**Instance ID**  
Unique identifier for instance. Permanent.

**Configuration**  
Determines where your data and nodes are located. Affects cost, performance, and replication. This choice is permanent. Select a configuration to view its details.

Regional  
 Multi-region

**Nodes**  
Add nodes to increase data throughput and queries per second (QPS). Affects billing.

[▼ Node guidance](#)

**Cost**  
Storage cost depends on GB stored per month. Nodes cost is an hourly charge for the number of nodes in your instance. [Learn more](#)

<b>Nodes cost</b> \$0.90 per hour	<b>Storage cost</b> \$0.30 per GB/month
--------------------------------------	--

[Create](#) [Cancel](#)

When you click "Create", the instance should show up and you should be brought to a page where you can view your new (but empty) instance.

**Figure 6.7. Viewing your newly created instance**

**Instance details**   [CREATE DATABASE](#)   [EDIT](#)   [DELETE](#)   [PERMISSIONS](#)

**Test Instance**

ID: test-instance Configuration: us-central1

Nodes	CPU utilization (mean)	Operations	Throughput	Total storage
1	0%	Read: 0/s Write: 0/s	Outbound: 0 B/s Inbound: 0 B/s	0 B

**Databases**  
No databases yet. Create a database to get started.

[Create database](#)   [Spanner documentation](#)

Now that we have our instance, the next thing we have to do is create a new database. To do that, just click the button on the page that says "Create database", and you should see a form where you can choose a database name and fill in a schema.

**Figure 6.8. Creating your first database**

[← Create a database](#)

Create a new database in this Spanner instance.

**✓ Name your database**

Enter a permanent name for your database.

**Name**

[Continue](#)

**2 Define your database schema**

Add tables and indexes to define your initial schema. You can add these anytime, but it's fastest to add them during database creation.

Edit as text

[+ Add table](#)   [+ Add index](#)

[Create](#)   [Cancel](#)

This is a two step process where you first choose a name for the database, and then you have a chance to create some tables that belong in your database. For now, let's leave the database completely empty (that is, don't add any tables), so we'll just call this `test-database` and then click "Create". After that you should land on a page where you can view your new (but empty) database.

**Figure 6.9. Viewing your newly created database**

The screenshot shows the 'Database details' page for a database named 'test-database'. At the top, there are navigation links: 'QUERY', '+ CREATE TABLE', 'DELETE', and 'PERMISSIONS'. Below the title, there are two tabs: 'Overview' (which is selected) and 'Monitor'. Under the 'Overview' tab, there are four performance metrics: CPU utilization (mean) at 1.49%, Operations (Read: 0/s, Write: 0/s), Throughput (Outbound: 0 B/s, Inbound: 0 B/s), and Total storage at 0 B. Below these metrics, there is a section titled 'Tables' with the sub-instruction 'No tables yet. Create a table to get started.' A blue 'Create table' button is visible. The overall interface is clean and modern, typical of Google's cloud services.

Now that you have an empty database, let's move onto the schema side of things and create a new table.

## 6.4.2 Creating a table

As you learned, Spanner tables are very similar to other relational databases. There are very important differences, however for now we're going to save those for later when we discuss the more advanced topics. To start, we're going to create a very simple table for storing employee information, which has just the two fields we used in our example before: a unique ID (primary key) for the employee, and the employee's name.

To get started, the first thing we need to do is click the "Create table" button when viewing our newly created database. When we do that, we'll see a form where we can create the table. The Cloud Console makes it easy to create a new table with a helpful schema-building tool, however since we're going to need to learn about the more advanced concepts later anyway, let's use the "Edit as text" option and paste in the schema for our `employees` table.

**Figure 6.10. Creating our employees table**

[← Create a table in test-database](#)

---

[Edit as text](#)

**DDL statements**  
Add Spanner Database Definition Language SQL statements below. Separate statements with a semicolon. [Learn more ↗](#)

```

1 CREATE TABLE employees (
2   employee_id INT64 NOT NULL,
3   name STRING(100) NOT NULL,
4   start_date DATE
5 ) PRIMARY KEY (employee_id);

```

[Create](#) [Cancel](#)

Once you click create, you'll be brought to a page where you can see the details of your table, such as the schema, any indexes (currently we have none) and a preview of the data (which will be empty as there's no data).

**Figure 6.11. Viewing our newly created table**

[Table details](#) [CREATE INDEX](#) [EDIT](#) [DELETE](#)

---

**employees**

[Schema](#) [Indexes](#) [Preview](#)

Column	Type	Nullable
employee_id	INT64	No
name	STRING(100)	No
start_date	DATE	Yes

[Show equivalent DDL](#)

Taking stock, we now have created an instance, a database belonging to the instance, and a table belonging to the database. But what good is an empty table? Let's move onto the interesting part of loading it up with some data.

### 6.4.3 Adding data

One of the key differences between Spanner and other relational databases is the way

to modify data. In a typical database, like MySQL, adding new data is done with an `INSERT` SQL query and updating existing data is done with an `UPDATE` SQL query, however Spanner doesn't support those two! This happens to be one of the places where Cloud Spanner shows its "NoSQL influences".

Instead of inserting data using the "query" interface, writing to Cloud Spanner is done via a separate API, in a way that's more similar to a non-relational key-value system, where you choose a primary key and then "set" some values for that key. To demonstrate this, let's use the `@google-cloud/spanner` Node.js package to add some employee data to our `employees` table in Spanner. We can install this using `npm`, by running `npm install @google-cloud/spanner@0.7.0`.

#### **Listing 6.3. Script to add some employees to our table**

```
const spanner = require('@google-cloud/spanner')({
  projectId: 'your-project-id'          ①
});

const instance = spanner.instance('test-instance'); ②
const database = instance.database('test-database');
const employees = database.table('employees');       ③

employees.insert([
  {employee_id: 1, name: 'Steve Jobs', start_date: '1976-04-01'},
  {employee_id: 2, name: 'Bill Gates', start_date: '1975-04-04'},
  {employee_id: 3, name: 'Larry Page', start_date: '1998-09-04'}
]).then((data) => {
  console.log('Saved data!', data);
});
```

- ① Remember to replace the project ID here with your own project ID.
- ② Create a pointer to the database that we created in the Cloud Console.
- ③ Create a pointer to the table that we created earlier.
- ④ Insert several rows of data, each row being its own JSON object.

If everything worked, you should see output confirming that the data was saved and the time stamp of the change being persisted.

#### **Listing 6.4. Output after saving new employees**

```
> Saved data! [ { commitTimestamp: { seconds: '1489763847', nanos: 466238000 } } ]
```

Now that we've seen how to get data into Spanner, let's look through how to get it out of Spanner.

#### **6.4.4 Querying data**

There are actually two different ways that we can do that, so let's take a look at each. First, we can use Spanner's Read API which is a way of asking questions about a single table. These questions can either be simply lookups of a specific key (or set of

keys) or a table scan with some filters applied. It just so happens that this method is probably the best fit to retrieve the three rows we just added.

Next, you can execute a SQL query on the **database** which allows you to query multiple tables using joins and other advanced filtering which you've come to know in other databases. In this case, we don't actually need to do anything complex so this would be overkill, but we'll demonstrate it anyway. Let's start by using the Read API, by calling `table.read()` in the Node.js client library to fetch one of the rows we added by the primary key.

#### **Listing 6.5. Using Spanner's Read API to retrieve a row by its key**

```
const spanner = require('@google-cloud/spanner')({
  projectId: 'your-project-id'
});

const database = spanner.instance('test-instance').database('test-database');
const employees = database.table('employees');
const query = {
  columns: ['employee_id', 'name', 'start_date'],
  keys: ['1']
};

employees.read(query).then((data) => {
  const rows = data[0];
  rows.forEach((row) => {
    console.log('Found row:');
    row.forEach((column) => {
      console.log(' - ' + column.name + ': ' + column.value);
    });
  });
});
```

After running this, you should see that the row we added was stored correctly.

#### **Listing 6.6. Output of the row successfully found**

```
Found row:
- employee_id: 1
- name: Steve Jobs
- start_date: Wed Mar 31 1976 19:00:00 GMT-0500 (EST)
```

But what if you wanted to get all of the rows in the database? Generally this is a bad idea, but since we're just trying to check that the three rows we added were stored successfully, we can do this by using a special `all` flag on the query, shown below.

#### **Listing 6.7. Retrieving all rows**

```
const spanner = require('@google-cloud/spanner')({
  projectId: 'your-project-id'
});

const database = spanner.instance('test-instance').database('test-database');
```

```

const employees = database.table('employees');
const query = {
  columns: ['employee_id', 'name', 'start_date'],
  keySet: {all: true}
};

employees.read(query).then((data) => {
  const rows = data[0];
  rows.forEach((row) => {
    console.log('Found row:');
    row.forEach((column) => {
      console.log(' - ' + column.name + ': ' + column.value);
    });
  });
});

```

After running this, you should see all of the data that we just added come back as the results, shown here.

#### **Listing 6.8. Output of all rows retrieved**

```

Found row:
- employee_id: 1
- name: Steve Jobs
- start_date: Wed Mar 31 1976 19:00:00 GMT-0500 (EST)
Found row:
- employee_id: 2
- name: Bill Gates
- start_date: Thu Apr 03 1975 20:00:00 GMT-0400 (EDT)
Found row:
- employee_id: 3
- name: Larry Page
- start_date: Thu Sep 03 1998 20:00:00 GMT-0400 (EDT)

```

Now that we've tried out the Read API, let's look at the more generic SQL querying API. The first notable difference when querying is that you send a query to a **database** rather than a specific table. This is because the query might involve other tables (e.g., if you JOIN two tables together). Additionally, instead of sending a structured object to represent the query, you send a string containing your SQL query.

Let's start by sending a simple query to retrieve all of the employees with a SQL query. As you might expect, the query itself is pretty straight forward, and identical to what it would be when querying something like MySQL.

#### **Listing 6.9. Executing a SQL query against Spanner**

```

const spanner = require('@google-cloud/spanner')({
  projectId: 'your-project-id'
});

const database = spanner.instance('test-instance').database('test-database');
const query = 'SELECT employee_id, name, start_date FROM employees'; ①

database.run(query).then((data) => {

```

```

const rows = data[0];
rows.forEach((row) => {
  console.log('Found row:');
  row.forEach((column) => {
    console.log(' - ' + column.name + ': ' + column.value);
  });
});
});

```

After running this, you should see the exact same output as the previous run, showing all of the employees and the columns involved.

#### **Listing 6.10. Output of all rows matching the SQL query**

```

Found row:
- employee_id: 1
- name: Steve Jobs
- start_date: Wed Mar 31 1976 19:00:00 GMT-0500 (EST)
Found row:
- employee_id: 2
- name: Bill Gates
- start_date: Thu Apr 03 1975 20:00:00 GMT-0400 (EDT)
Found row:
- employee_id: 3
- name: Larry Page
- start_date: Thu Sep 03 1998 20:00:00 GMT-0400 (EDT)

```

Now, let's try filtering this down to just Bill Gates. To do that, we need to add a WHERE clause in our SQL statement, but we'll also structure things bit more so that we can correctly inject parameters into the SQL query. This is generally good practice so that you avoid SQL injection attacks: any variable data you use in a query should always be properly escaped.

#### **Listing 6.11. Using parameter substitution on a SQL query**

```

const spanner = require('@google-cloud/spanner')({
  projectId: 'your-project-id'
});

const database = spanner.instance('test-instance').database('test-database');
const query = {
  sql: 'SELECT employee_id, name, start_date FROM employees WHERE employee_id = @id',
  params: {
    id: 2
  }
};

database.run(query).then((data) => {
  const rows = data[0];
  rows.forEach((row) => {
    console.log('Found row:');
    row.forEach((column) => {
      console.log(' - ' + column.name + ': ' + column.value);
    });
  });
});

```

```
});  
});
```

After running this, you should see only one row in the results, just including Bill Gates.

#### **Listing 6.12. Output of the row matching the SQL query**

```
Found row:  
- employee_id: 2  
- name: Bill Gates  
- start_date: Thu Apr 03 1975 20:00:00 GMT-0400 (EDT)
```

Now that you've seen how to run queries, let's look at what happens when you decide you want to store different information in your tables and have to change your schema.

#### **6.4.5 Altering database schema**

As your applications grow and evolve over time, you may find the need to change the structure of the data that you store. Just like any other relational database, Spanner supports schema alterations, however there are a few caveats to remember. Let's run through some of the things that are easy and obvious, and then we'll look into some of the more complicated changes.

First, the most basic change to a database is adding a new table. As you've seen already, this type of operation (`CREATE TABLE`) works just as you'd expect. Similarly, deleting entire tables (`DROP TABLE`) works as expected, though there is a limitation related to "child tables", which we explore later in the chapter.

Tables themselves can be modified in many of the ways you'd expect, though there are a few prerequisites to what types of changes are allowed. First, the new column cannot be a primary key. This should be obvious as you can only have one primary key, and it's required at the time when you create the table. Next, the new column cannot have a `NOT NULL` requirement. This is because you may have data already in the table and those existing rows clearly don't have a value for the new column, and therefore will need to be set to `NULL`.

Columns themselves can also be modified, with similar limitations involved when adding new columns. There are three different types of column alterations that are allowed:

1. You can change the type of a column from `STRING` to `BYTES` (or `BYTES` to `STRING`).
2. You can change the size of a `BYTES` or `STRING` column, so long as it's not a primary key column.
3. You can add or remove the `NOT NULL` requirement on a column.

In these situations, the limitations are related to data validation, and will fail in the case of a failure. For example, if you try to apply a `NOT NULL` limitation to a column that currently has rows where that column is set to `NULL`, the schema alteration will fail because the data won't fit with the altered column definition. Since all of the data must be checked against the new schema definition, these types of alterations can take quite

a long time, so in general it's not a great idea to do these often.

Let's take this for a spin, but this time we'll use the Cloud SDK's command-line tool (`gcloud`) to execute our queries and alter our schema. A simple (and quite common) task is to increase the length of a string column, so let's take our `employees` table and increase the length of the `name` column from 100 characters to "the maximum supported" which is denoted by a special value: `MAX` (with a maximum limit per column of 10 MiB). The query we need to run is shown below.

#### **Listing 6.13. SQL query to support longer employee names**

```
ALTER TABLE employees ALTER COLUMN name STRING(MAX) NOT NULL;
```

To run this, we'll use the `gcloud spanner` sub-command and request alterations using Spanner's DDL (data definition language).

#### **Listing 6.14. Using the Cloud SQL to execute the schema alteration**

```
$ gcloud spanner databases ddl update test-database \
--instance=test-instance \
--ddl="ALTER TABLE employees ALTER COLUMN name STRING(MAX) NOT NULL"
DDL updating...done.
```

Now if you go back the Cloud Console to look at our table, you'll see that the column has a new maximum length!

**Figure 6.12. The employees table after the alteration has been applied**

Column	Type	Nullable
employee_id	INT64	No
name	STRING(MAX)	No
start_date	DATE	Yes

Show equivalent DDL

Now that we've seen how we might change our minds about the schema, this covers the important basic things that you should know about Spanner. However none of the things we've described do anything more than demonstrate how Spanner is similar to a traditional relational database like MySQL. To understand where Spanner really shines, there's quite a bit more that we'll need to explore, so let's dive right into the advanced concepts that show the real power of Spanner.

## 6.5 Advanced concepts

While the basic concepts you've learned so far are enough to get you going with Cloud Spanner, to use it effectively and at the enormous scale for which it was designed, you'll need to understand quite a bit more about how Spanner blends a traditional relational database with a large-scale distributed storage system. Let's start by looking at the schema-level concept of interleaving tables with one another.

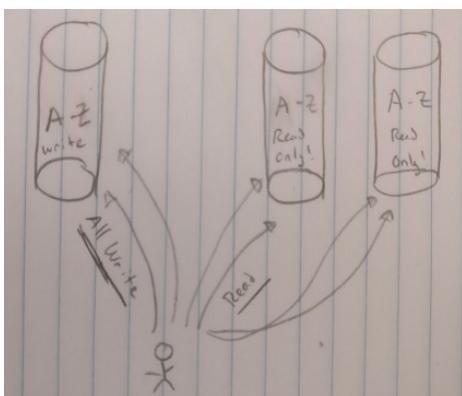
### 6.5.1 Interleaved tables

In a typical relational database, such as MySQL, the data itself is flat. That is, when you store a row, it tends to have a unique identifier and then some data, but the only hierarchical relationship is between the row and the table (that is, the row "belongs to" the table). In Cloud Spanner there happen to be additional relational aspects which is sometimes explained as relationships between tables themselves, with one table "belonging to" another. This might sound pretty weird at first, so to understand, we have to take a brief detour to explore one of the problems that comes up when databases experience heavy load.

When you have a large amount of data or a large number of requests for some data, sometimes a single server just can't handle it. One of the first steps to fixing is this is to create read replicas which duplicate data and act as alternative servers to query for the data. This solution is often the best one for systems that have heavy read-load (that is, lots of people asking for the data), and relatively light write-load (that is, modifications to the data). The reason for that is simply that read-replicas do what their name says: act as duplicate databases that you can read from. All changes to the data still need to be routed through the primary server, which means it's still the bottle-neck of your database.

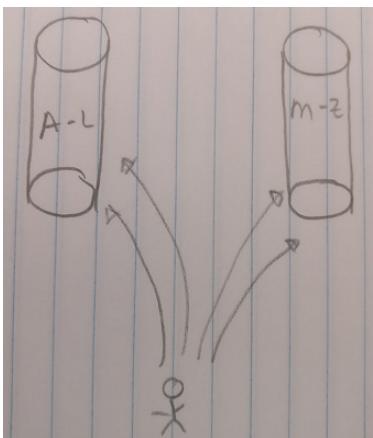
So what happens if you have a lot of modifications? Or if your databases is getting so large that it won't easily fit on a single server? Obviously in that case, read-replicas are unlikely to fix the problem for you, since it needs to duplicate **all** of the data.

**Figure 6.13. Using a read replica means one database is responsible for all writes**



In this situation, a common solution is to "shard" the data across multiple machines. That is, instead of creating many different machines each with a full copy of the data, (but only one capable of modifying that data), you instead chop the data up into distinct pieces and delegate responsibility for different chunks to different machines. For example, given an `employees` table that stores employee information, you might put data for employees with names in the range A through L on one server, and M through Z on another server. By doing this, you've doubled your capacity so long as someone doing the querying can figure out how to find the right data. To make this concrete, before this "sharding", a query for two employees (say, "Steve Jobs" and Mark Zuckerberg) would have been handled by a single machine. If the database was split as described above, these two queries would be handled by two different machines.

**Figure 6.14. Using data shards splits the read and write responsibility**



This sounds easy in that example, but that's because we were focused on a single table (`employees`). You also need to make sure that, for example, paycheck information, insurance enrollment, and other employee data in different tables is similarly chopped up. In addition, you'd want to make sure that all of the data is consistently split, particularly when you want to run a `JOIN` across those two tables. In other words, if you want to get an employees name and the sum of their last 10 paychecks, having the paycheck data on one machine and the employee data on another would mean that this query is incredibly difficult to run.

Even worse, what about when you need *even more* serving capacity? Doing this process again to split the range into 3 pieces (say, A through F, G through O, and P through Z) is a pretty big pain, and ideally you don't want to have to do this whenever your query load goes up or down. Even more perplexing is that this design assumes all users have around the same amount of traffic asking for their data. What if it turned out that two users (say, the Kardashians) are responsible for 80% of the traffic? In that case, it might make sense to give each of those their own server, and then segregate the rest of the data evenly as described above.

Wouldn't it be nice if your database could figure this out for you? That way, you wouldn't have to chop your data up manually, but it could instead dynamically split up and shift around to make sure that your resources were being used optimally. This is what Spanner does with interleaved tables.

Splitting up the data is easy for Spanner to do, as a matter of fact this has been a feature of Bigtable for quite some time. What's unique is the idea of being able to provide hints to Spanner of where it should do the splitting, so that it doesn't do crazy things like put an employee's paycheck and insurance information on two separate machines.

Interleaving tables is the way that you tell Spanner which data should live "nearby" other data (and should move around together), even if that data is split across multiple tables. In our example above, the `employees` table might be what's called a "parent table", and the others (storing paycheck or insurance information) would be *interleaved* within the `employees` table as "child tables". Note also that the `employees` table has no more parents, so it's considered to be a "root table".

Let's look at this a bit more concretely to see how it actually works by using some demonstration tables. In a traditional layout, storing employees and their paycheck amounts would just be two separate tables, with a foreign key pointing from the `paychecks` table to the `employees` table (in this case, the "User ID" column).

**Table 6.1. Typical structure to store users and their posts**

Employees		Paychecks			
ID	Name	ID	User ID	Date	Amount
1	Tom	1	3	2016-06-09	\$3,400.00
2	Nicole	2	1	2016-06-09	\$2,200.00

As you just learned, if you went to shard these tables by ID, it's possible that the paycheck information for a user (say Nicole) would end up on one machine but the employee record would end up elsewhere. Obviously this is an issue.

In Spanner, you can fix this by "interleaving" the two tables together. Where you want to convey that an employee and their corresponding paychecks should be located near each other and move around together, your data would look somewhat different, shown below.

**Table 6.2. Employees interleaved with paychecks**

ID	Name	Date	Amount
Employees(1)	Tom		
Employees(2)	Nicole		
Paychecks(2, 2)		2016-06-09	\$2,200.00
Employees(3)	Kristen		
Paychecks(3, 1)		2016-06-09	\$3,400.00

An equivalent representation with the IDs separated apart would look something like the following table.

**Table 6.3. Alternative key-style of employees interleaved with paychecks**

Employee ID	Paycheck ID	Name	Date	Amount
1		Tom		
2		Nicole		
2	2		2016-06-09	\$2,200.00
3		Kristen		
3	1		2016-06-09	\$3,400.00

As you can see in this case, related data is put together, even though this means that data from two different tables aren't separated. This layout also means that the ID fields become condensed, so let's look in more detail at what those keys are.

### 6.5.2 Primary keys

While not a requirement in a typical relational database, it's generally a good practice to give each row what's called a "primary key". Often this key is numeric (though that isn't required either) but the value has a uniqueness constraint, meaning that duplicate values are not permitted, so that the primary key can be used for indexing and addressing a single row. In Spanner, the primary key is required, but rather than being a single field, it can be composed of multiple fields as you saw in the example of the interleaved tables above.

To clarify this a bit more, let's look at the same example above (employees and paychecks) but instead of relying on an example table let's take a peek at the underlying SQL-style query that defines the schema and see what each piece does.

**Listing 6.15. Example schema for the employees and paychecks tables**

```

CREATE TABLE employees (
    employee_id INT64      NOT NULL,          ①
    name        STRING(1024) NOT NULL,
    start_date  DATE        NOT NULL
) PRIMARY KEY(employee_id);                  ②

CREATE TABLE paychecks (
    employee_id   INT64 NOT NULL,
    paycheck_id   INT64 NOT NULL,
    effective_date DATE NOT NULL,
    amount_cents  INT64 NOT NULL
) PRIMARY KEY(employee_id, paycheck_id),     ③
    INTERLEAVE IN PARENT employees ON DELETE CASCADE; ④
                                                ⑤

```

- ① Here we define the ID for each employee. We call it `employee_id` (rather than just `id`) for clarity in the future.
- ② Here we define that the `employee_id` field is the primary key for this table. This means that it will be required to be unique, and used to identify a given row.
- ③
- ④
- ⑤

- ③ In the paychecks table, we keep track of the employee's ID as well as the ID of the paycheck, similar to how we had the fields defined in a typical relational database.
- ④ Unlike in a typical relational database, rather than defining a foreign key relationship (pointing from `employee_id` in `paychecks` to `employee_id` in `employees`), we make the relationship a part of the compound primary key.
- ⑤ To clarify that the `paychecks` table should be kept near to the `employees` table, we use the `INTERLEAVE IN PARENT` statement, and specify that if an employee is deleted, the `paychecks` should also be deleted.

As you can see in this example, there are two tables: `employees` and `paychecks`. The each employee has an ID and a name, whereas each paycheck has am ID, a pointer to the employee (the employee's ID), a date, and an amount. This should feel very familiar, but there are two very important things to notice:

1. Primary keys can be defined as a combination of two IDs (e.g., `employee_id` and `paycheck_id`).
2. When interleaving tables, the parent's primary key must be the start of the child's primary key (e.g., `paychecks` primary key must start with the `employee_id` field) or you'll see an error.

Now that we've gone through that, recall the idea of sharding data into chunks and splitting it across servers. We said that by interleaving tables the related data would be kept together, but we didn't really dive into how that works. Let's take a moment to walk through how data is divided up using something called "split points", as this can have some pretty important performance implications.

### 6.5.3 Split points

Just as the name suggests, split points are the exact positions at which data in a table might be split into separate chunks and potentially handed off to another machine to cope with request load or data size. So far we've said where we **don't** want any data to be split up, and explained that in our schema by interleaving the paycheck data with the employee data. In other words, by using a compound primary key in the `paychecks` table, we've said that all paychecks of each employee should be kept alongside the record for the "parent" employee.

Notice however that we haven't clarified how exactly we say that data **can** be split up. That is, we've never said which employees can be separated and handed off. Funnily enough, it turns out that Spanner makes a big assumption: if you didn't say that things must stay together, they can and may be split. These points that you haven't specifically prohibited, which lie between two rows in a "root table", are called "split points".

To make this more clear, let's look at our example table of employees and paychecks again and see where the split points are. Recall that a "root table" is a table without a parent, which in this case is our `employees` table. This means that there are split points between every two different primary keys belonging to the root table. In other words, there are split points before every unique employee ID.

**Figure 6.15. Split points between every unique employee ID**

Employee-id	Paycheck-id	Name	Date	Amount
1		Toran		
2		Niede		
3	2		JUN 9th	2200
3	1	Kristen	JUN 9th	3400
3	1			

Notice that this means all the records with the same employee ID at the start of the primary key will be kept together, but each chunk of records can be shifted around as necessary. For example, it's possible that employees 1, 2, and 3 all could be on different machines, but paycheck 2 will be on the same machine as employee 2, and paycheck 1 will be on the same machine as employee 3.

**NOTE**

If you read [chapter 5](#), you should notice some similarities. In this case, Datastore has the same concept, but talks about *entity groups* as the indivisible chunks of data, whereas Spanner talks about the points **between** the chunks and calls them *split points*.

This leads us to one final topic on this tricky business of interleaving tables, split points, and primary keys: choosing a good primary key.

#### 6.5.4 Choosing primary keys

You may be asking yourself, "Choosing a primary key? Why not just use numbers?" And you're not crazy. Choosing primary keys is not something you typically do in a relational database. For example, MySQL offers a way to specify that fields should be "automatically incremented", so that if you omit the field, it will be substituted by the highest value incremented by one. But this happens to be an area in which Spanner is quite different.

So far, you've heard that chunks of data will be kept together, but there's actually a deeper reason for that: sequential data is kept together as well because Spanner actually keeps all of the data in the database sorted lexicographically by primary key. While data will only be divided exactly on split points between these chunks (e.g., between different employees) this means that employees 10 and 11 will be next to each other (unless Spanner has decided to divide them up at the split point between the two).

This might seem like "no big deal", but it's actually very powerful as it means you can distribute your writes evenly across the key space (and therefore across your Spanner infrastructure) by choosing keys that are evenly distributed. Sadly this also means that you can effectively cripple yourself if you choose keys that all happen to hit a single Spanner node. To make this more clear, let's look at a classic example of a terrible

primary key to use: timestamps.

#### **Listing 6.16. Example schema using a timestamp**

```
CREATE TABLE events (
    event_time TIMESTAMP NOT NULL,
    event_type STRING(64) NOT NULL
) PRIMARY KEY(event_time);
```

Let's imagine now that we had millions of sensors somewhere all broadcasting events so that the total request rate was one write every microsecond (that's 60,000 writes per second). Spanner should be able to handle that, right? Not so fast. Let's imagine what happens when Spanner tries to deal with this scenario.

First, lots of traffic is coming to a single node since each event is only 1 microsecond away from the previous one. To deal with this overload, Spanner will pick a split point (in this case, between any two events since this is a "root table") and chop the data in half. One half of the data will have IDs as times *before* the split point and the other half *after* the split point. Now more traffic comes in. Can you guess which side will be responsible for the new rows?

All the new rows are guaranteed to have IDs as times *after* the split point, because time continues to count upwards! So this means that we're right back where we started with a single node handling all of the traffic. If you do this same process again, you'll notice that it continues to **not** fix the problem! This problem, which happens quite often, is called "hot-spotting" as you've created a "hot spot" that is the focus of all the requests.

The moral of this story is that when writing new data, you should choose keys that are evenly distributed, and never choose keys that are counting or incrementing (such as A, B, C, D, or 1, 2, 3). This also means that keys with the same prefix and counting increments are just as bad as the counting piece alone (e.g., `sensor1-<timestamp>` is just as bad as using a time stamp). So instead of using counting numbers of employees, you might want to choose a unique random number or a reversed fixed-size counter. There are libraries that can help with this, for example, Groupon's `locality-uuid` package (see [github.com/groupon/locality-uuid.java](https://github.com/groupon/locality-uuid.java)).

Now that you understand all of these concepts of data locality, choosing primary keys, split points, and interleaving tables, let's explore how and why you might want to use indexes on your tables.

#### **6.5.5 Secondary indexes**

For many of us, indexes are "those things you add later when your database gets slow", and while that description is somewhat accurate (and often practical), indexes are actually a very important performance tool for a database. Let's take a moment to review this concept of how indexes work and then we'll dig into how Spanner uses them to speed up queries.

In their most basic form, indexes act as a way of telling your database to maintain some alternative ordering of data in addition to the data already stored in the database.

That is, instead of just storing the list of employees sorted by their primary keys, you might want the database to store a list of employees sorted by their name as well.

The reason for this is pretty simple: if you have data sorted by a column that you intend to filter on (e.g., `WHERE name = "Joe Gagliardi"`) the search for on that column can be done much more quickly. This is because searching an ordered list is much faster than searching an unordered list for a variety of reasons.

To see what I mean, imagine I asked you to find everyone in the phone book with the name "Richard Feynman" (Feynman, Richard). Easy right? This is because the phone book's "primary key" is (`last name, first name`). Imagine instead that you had to find everyone in the phone book with the first name "Richard" and a phone number ending in 5691. As you might guess, this query might take you a while because the phone book doesn't have an index for those fields. To do this query, you'd actually have to "scan through" all of the records in the phone book, which as you can imaging might take a while. So why wouldn't you just index everything? Wouldn't that make all of your queries fast?

It turns out that while indexes can make queries over your data run more quickly, those indexes also need to be updated and maintained. In other words, searching for a specific person by name might be faster thanks to the index on the employee names, however whenever you update an employee's name (or create a new employee), you need to update the row in the table along with the data in each index that references the name column. If you don't do this the data will get "out of sync" and strange things might happen, such as a query returning a matching row, but retrieving that supposedly matching row ends up not matching after all.

This means that if you added an index on employee names to make those look-ups and filters faster, updating a name now involves writes to two different resources (the table itself and the index you created). In short, this is just a trade-off: in exchange for slightly more work being done at write-time, you get much less work needing to be done at read-time.

To make things more complicated, indexes also take up extra space, and while the size at first may be no big deal, as you add more and more data the total space consumed can become pretty significant. To see this, just imagine for a second how large the phone book would be if it had both the regular data (by last name) as well as the index from our example above (first name and phone number). You might not have to store all the pictures in the index, but it would certainly have exactly the same number of entries as those that are in the phone book.

So how do you decide when to add an index? This is actually pretty complicated, (there are entire books on the subject) but in general you should start by looking at the queries you need to run against the database. In other words, while the shape of your data will influence the schema of your tables, it's the queries you run that will influence the indexes you need. If you understand the queries you're running, you can see exactly what types of indexes you need and add them as needed (or remove them when they become unnecessary). It's probably best to walk through this using more

clear examples, so let's look at Spanner's take on secondary indexes, and expand our example from earlier with employees and paychecks.

Luckily Spanner's idea of secondary indexes is pretty close to other common relational databases: without them Spanner queries will run just fine but may be slower than usual, and with them writes have extra work to do but queries should get faster. That said, they have a couple of key differences that stem from the concept of interleaved tables that we explored previously. Let's start by looking at some of the similarities.

In our current database schema (with a `paychecks` table interleaved in an `employees` table), it seems like we'll want to do look-ups and searches by an employee's name, however running this query will involve a full table scan (looking at every row to be sure that we've found all matches). To see this, you can run a query that does a name lookup from the Cloud Console and take a look at the "Explanation" tab to see that the query starts off with a table scan.

**Figure 6.16. Finding employees by name without an index results in a table scan**

Query database: test-database

```
1 select employee_id from employees where name = "Larry Page"
```

Run query Clear query SQL query help Run query: Ctrl + Enter

Results table Explanation

Total elapsed time	CPU time	Rows returned	Rows scanned
16.24 msecs	14.19 msecs	0	0

Operator reference | Guided tour

Operator	Rows returned	Executions	Latency
■ Distributed union	0	1	0 ms
↑ Local distributed union	0	1	0 ms
↑ Serialize Result	0	1	0 ms
↑ Filter ▾	0	1	0 ms
⌚ Table Scan: employees ▾	0	1	0 ms

Let's make this faster by creating an index on the `name` column, using a DDL statement that looks something like the following.

**Listing 6.17. Schema alteration to add an index to the employees table**

```
CREATE INDEX employees_by_name ON employees (name)
```

You can use the `gcloud` command like before to create the index.

**Listing 6.18. Create the index at the command line**

```
$ gcloud spanner databases ddl update test-database \
--instance=test-instance \
--ddl="CREATE INDEX employees_by_name ON employees (name)"
DDL updating...done.
```

And once the index is created, you should see it in the Cloud Console.

**Figure 6.17. The newly created index on employee names**

`employees_by_name`

Table indexed: `employees`

Columns indexed

Column	Sort order
<code>name</code>	Ascending

[Show equivalent DDL](#)

The fun part, however, comes from re-running that same query again to find a specific employee by name. As you can see in the explanation below, the results now rely on our newly created index rather than scanning through the entire table.

**Figure 6.18. Spanner uses the new index to execute the query**

Query database: test-database

```
1 | select employee_id from employees where name = "Larry Page"
```

**Run query** **Clear query** **SQL query help** Run query: Ctrl + Enter

**Results table** **Explanation**

Total elapsed time	CPU time	Rows returned	Rows scanned
3.16 msecs	1.46 msecs	1	1

[Operator reference](#) | [Guided tour](#)

Operator	Rows returned	Executions	Latency
▪ Distributed union	1	1	0 ms
↑ Local distributed union	1	1	0 ms
↑ Serialize Result	1	1	0 ms
↑ Index Scan: employees_by_name ▾	1	1	0 ms

Something strange happens when you alter this query to ask for more than just the employee ID. If you run a query for `SELECT * FROM employees WHERE name = "Larry Page"`, the explanation says that we're back to using the table scan. What happened? Why didn't it use the index that we have?

Our index was very specific about exactly what data is being stored, in this case, the primary key (as that's always stored) and the name. This means that if all you want is the primary key and the name (which is all our first query asked for), then the index is sufficient. If you ask for data that isn't there in the index, using the index itself won't really be any faster because after we've found the right primary keys that match our query, we still have to go back to the original table again to get the other data (in this case, the `start_date`).

Let's imagine that we often run a query that asks for the `start_date` of an employee where we filter based on a name. In other words, `SELECT name, start_date FROM employees WHERE name = "Larry Page"`. To make that query fast, we actually have to pay a storage penalty. This means that in order to rely on an index to handle the lookup, we also need to ask the index to store the `start_date` field, even though we don't want to filter on it. Spanner can easily do this, simply by adding a

simple STORING clause at the end of the DDL when creating the index.

#### **Listing 6.19. Creating an index which stores additional information**

```
CREATE INDEX employees_by_name ON employees (name) STORING (start_date)
```

Once you add this index, running a query like the one above will use the newly created index. In contrast, a query filtering on a specific ID (such as `SELECT name, start_date FROM employees WHERE employee_id = 1`) will still rely on a table scan, however that's the fastest kind of scan as it's a primary key lookup.

**Figure 6.19. Spanner now can rely on just the index for the entire query**

#### **Query database: test-database**

```
1 | SELECT name, start_date FROM employees WHERE name = "Larry Page"
```

**Run query**

**Clear query**

**SQL query help**

Run query: Ctrl + Enter

**Results table**

**Explanation**

Total elapsed time	CPU time	Rows returned	Rows scanned
1.69 msecs	0.74 msecs	1	1

[Operator reference](#) | [Guided tour](#)

Operator	Rows returned	Executions	Latency
■ Distributed union	1	1	0 ms
↑ Local distributed union	1	1	0 ms
↑ Serialize Result	1	1	0 ms
↑ Index Scan: employees_by_name_with_start_date ▾	1	1	0 ms

Now that you have your feet wet creating and modifying indexes, let's look at how this relates to the previous topics of interleaved tables. Just as you can interleave one table (the child table) into another (the parent table), indexes can similarly be interleaved with a table. By doing this, you end up with a "local index" that will be applied within each row of the parent table. This is a bit tricky to follow, so let's look

at some examples where we want to see paycheck amounts.

If we want to look at the paychecks sorted by amount, the query would be across **all** employees, so this query would be what we'll call "global".

#### **Listing 6.20. Querying for paychecks across all employees**

```
SELECT amount_cents FROM paychecks ORDER BY amount_cents
```

If we wanted the same information but only for a specific employee, the query is only across the paychecks belonging to a single employee. Since the `paychecks` table is interleaved into the `employees` table, we can think of this query as "local" as it's only scanning over a subset of rows, whittled down by our employee criteria, which we've already designated as rows we want to "keep nearby one another".

#### **Listing 6.21. Querying for paychecks of a single employee**

```
SELECT amount_cents FROM paychecks WHERE employee_id = 1 ORDER BY amount_cents
```

As you might guess, if you were to look at the explanation of both of these queries you'd see that they both involve a table scan over the `paychecks` table. So what indexes would make these faster?

For our first "global" query, having an index across the `paychecks` table on the `amount_cents` column would do the trick. But for the second one we want to take advantage of the fact that paycheck entries are interleaved in employee entries. To do this, you can interleave the index in the parent table and get a "local index" that will work when you happen to be looking within rows in a child table that are filtered by a row in a parent table.

In this case, the two indexes would look quite similar, the difference being an additional row in the index (`employee_id`), and the fact that the index itself would be interleaved with employee records, just like the paycheck records themselves.

#### **Listing 6.22. Create two indexes, one global and one local**

```
CREATE INDEX paychecks_by_amount ON paychecks(amount_cents);

CREATE INDEX paychecks_per_employee_by_amount_interleaved
    ON paychecks(employee_id, amount_cents),
    INTERLEAVE IN employees;
```

If you were to re-run the same query, the explanation would say that this time the query relied on our interleaved index.

You may be wondering why exactly you'd care about interleaving the index in the `employees` table. In other words, why not just create the index on those fields and leave out that `INTERLEAVE IN` part? Technically that is a valid index, however it loses out on the benefits of co-locating related rows near to each other. In other words, updates to a paycheck record may be handled by one server, and the corresponding (required)

update to the index may be handled by another server. By interleaving the index with the table in the same way that paycheck records are interleaved, we guarantee that the two records will be "kept together" and therefore keep updates to these two close by one another, which improves overall performance.

As you can see, indexes are incredibly powerful, however they are a double-edged sword. On the one hand they can make your queries much faster by virtue of having your data in exactly the format you need. On the other hand you must be willing to pay the cost of having to update them as your data changes and store additional data as needed to avoid further table scans.

Further, figuring out what indexes are most useful can be tricky, and there are entire books devoted to figuring out how best to index your data. The good news is that when you run queries against Spanner, it will automatically pick the one that it thinks will be the fastest unless you specifically force it to use an index. You can do this with the `force_index` option on the statement, for example, `SELECT amount_cents FROM paychecks@{force_index=paychecks_by_amount}`, however generally it's best to allow Spanner to choose the best way of running queries. Now that we've gone through at least the basics of indexing in Spanner, let's take a moment to explore something equally important: transactional semantics.

## 6.5.6 **Transactions**

If you've worked with a database (or any storage system really), you should be familiar with the idea of a "transaction", and the acronym that tends to go define the term: ACID. Databases that support ACID transactional semantics are said to have: atomicity (either all the changes happen or none of them do), consistency (when the transaction finishes, everyone gets the same results) isolation (when you read a chunk of data, you're sure that it didn't change out from under you), and durability (when the transaction finishes, the changes are truly saved and not lost if a server crashes), and these semantics allow you to focus on your application and not on the fact that multiple people might be reading and writing to your database at the same time.

Without support for transactions, all sorts of problems can occur, from the simple (such as seeing a duplicate entry in a query) to the horrifying (you deposit money in a bank account and your account isn't credited). Being a full-featured database, Spanner supports ACID transactional semantics, even going so far as supporting distributed transactions (although at a performance cost). Spanner supports two types of transactions: read-only and read-write. As you might guess, read-only transactions are not allowed to write, which makes them much simpler to understand, so we'll start there.

### **READ-ONLY TRANSACTIONS**

Read-only transactions, in short, offer the ability to make several reads of data in your Spanner database all at a specific point in time. That is, you never have to worry about getting a "smear" of the data spread across multiple times. For example, imagine that you need to run one query, do some processing on that data, and then query again

based on the output of that processing. By the time that you run the second query, it's possible that the underlying data has changed (e.g., some rows have been updated or deleted) and your queries might not make sense anymore! With read-only transactions, you can be sure that the data hasn't changed out from under you because you're always reading data at a specific point in time.

A read-only transaction doesn't hold any locks on your data, and therefore doesn't block any other changes that might be happening (such as someone deleting all the data or adding more data). To demonstrate how this works, let's look at a quick sample querying our employee data.

### **Listing 6.23. Querying data from inside and outside a transaction**

```
const spanner = require('@google-cloud/spanner')({
  projectId: 'your-project-id'
});
const instance = spanner.instance('test-instance');
const database = instance.database('test-database', {max: 2}); ①

const printRowCounts = (database, txn) => {
  const query = 'SELECT * FROM employees';
  return Promise.all([database.run(query), txn.run(query)]).then((results) => {
    const inside = results[1][0], outside = results[0][0];
    console.log('Inside transaction row count:', inside.length);
    console.log('Outside transaction row count:', outside.length);
  });
}

database.runTransaction({readOnly: true}, (err, txn) => { ③
  printRowCounts(database, txn).then(() => { ④
    const table = database.table('employees');
    return table.insert({
      employee_id: 40,
      name: 'Steve Ross',
      start_date: '1996-01-23'
    });
  }).then(() => { ⑤
    console.log('--- Added a new row! ---');
  }).then(() => { ⑥
    printRowCounts(database, txn);
  });
});
```

- ➊ Since the client uses a session pool to send manage concurrent requests, we want to make sure that we're using more than a single session (in this case, we'll use two).
- ➋ This is a helper function that gets the row counts from two connections: one from the transaction provided, the other from the database outside of the transaction.
- ➌ Start by creating a read-only transaction.
- ➍ Count all the rows from both inside and outside the transaction.
- ➎ From outside of the transaction, create a new employee in our table.
- ➏ Count all the rows again from both inside and outside the transaction.

In this script, we're trying to demonstrate how our transaction maintains an isolated view of the world, despite new data showing up from other people accessing (and writing to) the database. To be more specific, the "inside" counts should always remain the same ("inside" being the row count as seen by queries run from the `txn` object), regardless of what's happening "outside". Queries from "outside" the transaction, however, **should** see the newly added row when running the query. To see that this works, run the script above and you should see output that looks like the following.

#### **Listing 6.24. Running our read-only transactional test**

```
$ node transaction-example.js
Inside transaction row count: 3
Outside transaction row count: 3
--- Added a new row! ---
Inside transaction row count: 3
Outside transaction row count: 4
```

As you can see, our "inside" counts always stayed the same (at 3), while the "outside" counts (that is, the query run from outside our transaction) see the new row after it's committed. This really important because it demonstrates that read-only transactions act as containers for reads at a point frozen in time. Additionally, since a read-only transaction holds no locks on any of the data, you can create as many as you want and everything should work as expected. Because of these properties, sometimes it makes sense to think of a read-only transaction as an additional "filter" on your data:

#### **Listing 6.25. Example of the implicit restriction of queries run at a specific time**

```
SELECT <columns> FROM <table> WHERE <your conditions> AND \
  run_query_frozen_at_time = <time when you started your transaction>
```

Luckily, this concept of "freezing time" is pretty easy to understand and has almost none of those pesky "what if" awkward scenarios. However, read-write transactions are somewhat more complicated, so let's take a look at how they work and some of those weird cases.

#### **READ-WRITE TRANSACTIONS**

As the name suggests, read-write transactions are transactions that both read and modify data stored in Spanner. These transactions tend to be the very important ones that prevent you from doing things like "losing" an ATM deposit by operating on data that changed out from under you, so it's important to understand how they work and how to use them correctly.

Let's imagine we found a mistake in Employee 40's paycheck it's \$100 less than it should be. To make this change using Spanner's API, we need to do two things:

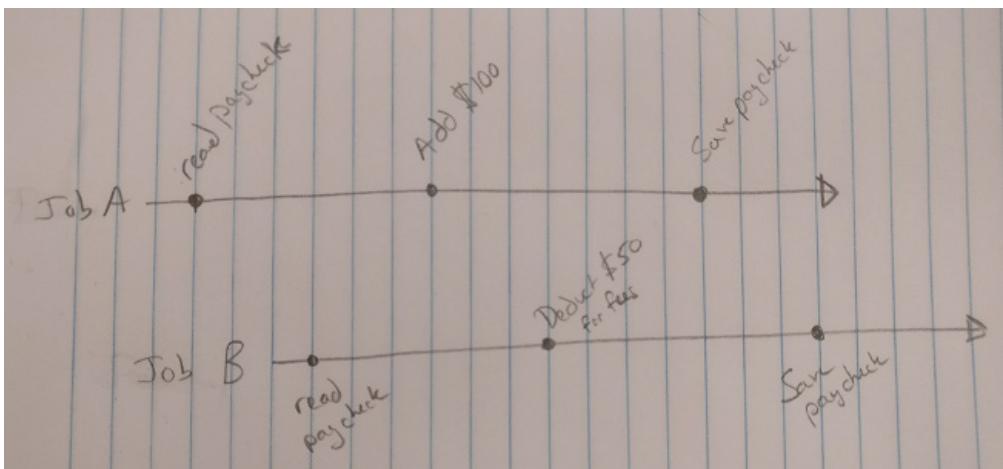
1. Read the amount of the paycheck
2. Update the amount of the paycheck to `amount + $100`

This might seem pretty boring, but in a distributed system where you may have lots of

people all doing things at once (some of them potentially conflicting with what you want to do) this task actually can become quite difficult. To see this, let's imagine that two jobs are running at once to update paychecks. One job is fixing an error where all paychecks were \$100 less than expected, and another is fixing an error where a \$50 fee wasn't taken out. If you run these jobs serially (that is, one after another) then everything should work just fine. Also, if you "combine" these jobs (that is, turn them into one job that adds \$50) things will also work out just fine. But those options aren't always available, so for this example, let's imagine them running side by side.

The problems begin to arise when both jobs happen to operate on the same paycheck at almost the same time. In those scenarios, it's possible that one job will "clobber" the work of the other, resulting in either **only** a \$100 paycheck increase or **only** a \$50 paycheck decrease rather than both (which in this situation is always a \$50 increase).

**Figure 6.20. Example of the "fee deducting job" clobbering the "\$100 increase job"**



To fix this, we need a way of "locking" certain areas of the data as a way of telling other jobs "don't mess with this, I'm using it". This is where Spanner's read-write transactions come in to save the day. Read-write transactions provide a way of locking exactly what you need, even when there is a very close overlap of data. This means that in the exact time line described above, Job A's write would complete and when Job B tries to save the changes, it will see a failure and be instructed to retry.

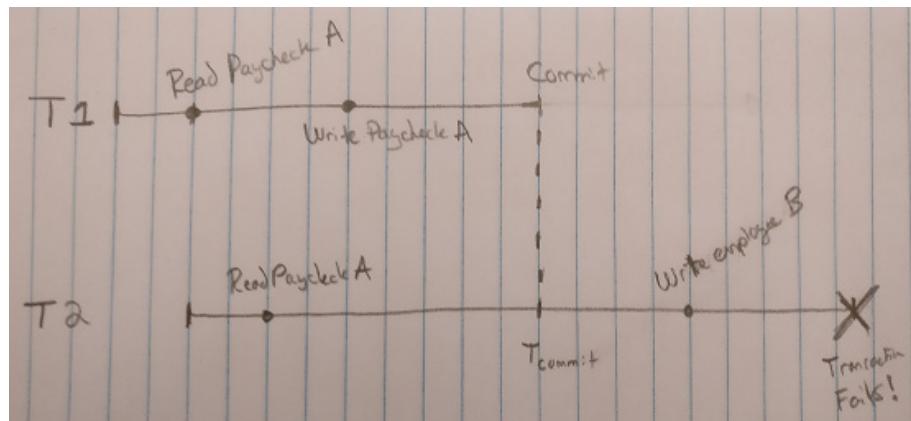
Read-write transactions also guarantee atomicity, which means that all the writes done inside the transaction either all happen at the same time, or all don't happen at all. For example, if you wanted to transfer \$5 from one paycheck to another, there are two operations: deduct \$5 from paycheck A, and add \$5 to paycheck B. If those two don't happen atomically, it means that one part of the process could happen and be saved without its corresponding partner, which would result in either disappearing money (\$5 deducted but not transferred) or free money (\$5 added but not deducted).

Additionally, reads inside a read-write transaction see all data that has been committed

before the transaction itself commits. That is, if someone else modifies a paycheck after your transaction "starts", everything will work as expected so long as you read the data after that other transaction commits. To see this, let's look at two examples of overlapping transactions, one failing, one succeeding.

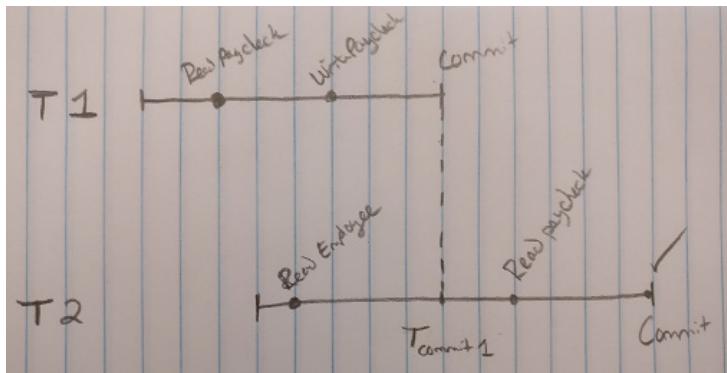
Transactions guarantee that all reads will happen at a single point in time (as we explained in the section read-only transactions), but it also guarantees that a transaction will fail if any of the data read during the transaction became stale during the life of the transaction. In this case, this means that if you read some data at the start of a transaction, and another transaction commits a change to that same data, the transaction will fail no matter what, regardless of what data you end up writing. In the example below, Transaction 2 is attempting to write the record of Employee B based on a read of Paycheck A. Between the read and the write, Paycheck A is modified by Transaction 1, meaning that Paycheck A's data is out of date, and as a result the transaction must fail.

**Figure 6.21. Transactions fail if any of the data read becomes "stale"**



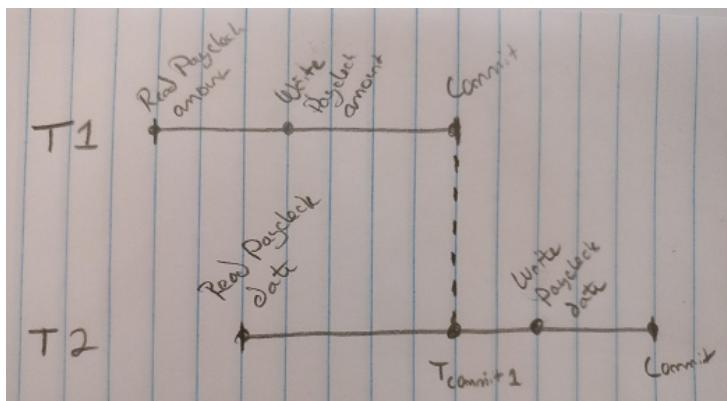
On the other hand, transactions are smart enough to ensure that if just reading **any** data won't force your transaction to fail. In other words, if you were to read some data at the start of your transaction, then another transaction modifies some unrelated data, and then you read the data that was just modified, your transaction can still commit successfully.

**Figure 6.22. Reading data after it's been changed doesn't cause transaction failures**



To make things even better, locking on data is done on a cell level (that is, a row and a column), which means that transactions modifying different parts of the same row won't conflict with one another. For example, if you read and update only the date of Paycheck A in one transaction, and then read and update only the amount of Paycheck A in another transaction, even if the two overlap, they will be able to succeed.

**Figure 6.23. Example of cell-level locking avoid conflicts**



To see this in action, let's write some code that illustrates successful cell-level locking. And then some more that demonstrates a failure.

#### **Listing 6.26 Demonstration of non-overlapping read-write transactions touching the same row**

```
const spanner = require('@google-cloud/spanner')({
  projectId: 'your-project-id'
});
const instance = spanner.instance('test-instance');
const database = instance.database('test-database', {max: 5});
const table = database.table('employees');
```

```

Promise.all([database.runTransaction(), database.runTransaction()]).then( ①
  (txns) => {
    const txn1 = txns[0][0], txn2 = txns[1][0]; ②

    const printCommittedEmployeeData = () => { ③
      const allQuery = {keys: ['1'], columns: ['name', 'start_date']};
      return table.read(allQuery).then((results) => {
        console.log('table:', results[0]);
      });
    }

    const printNameFromTransaction1 = () => { ④
      const nameQuery = {keys: ['1'], columns: ['name']};
      return txn1.read('employees', nameQuery).then((results) => {
        console.log('txn1:', results[0][0]);
      });
    }

    const printStartDateFromTransaction2 = () => { ⑤
      const startDateQuery = {keys: ['1'], columns: ['start_date']};
      return txn2.read('employees', startDateQuery).then((results) => {
        console.log('txn2:', results[0][0]);
      });
    }

    const changeNameFromTransaction1 = () => { ⑥
      txn1.update('employees', {
        employee_id: '1',
        name: 'Steve Jobs (updated)'
      });
      return txn1.commit().then((results) => {
        console.log('txn1:', results);
      });
    }

    const changeStartDateFromTransaction2 = () => { ⑦
      txn2.update('employees', {
        employee_id: '1',
        start_date: '1976-04-02'
      });
      return txn2.commit().then((results) => {
        console.log('txn2:', results);
      });
    }

    printCommittedEmployeeData() ⑧
      .then(printNameFromTransaction1)
      .then(printStartDateFromTransaction2)
      .then(changeNameFromTransaction1)
      .then(changeStartDateFromTransaction2)
      .then(printCommittedEmployeeData)
      .catch((error) => {
        console.log('Error!', error.message);
      });
  }
);

```

- ① Start by creating two transactions, both read-write.

- ② The results of the `Promise.all()` call are the two transaction objects.
- ③ This helper function prints out the data for Employee 1 that is committed in Spanner (that is, it doesn't include any uncommitted data).
- ④ This helper function reads **only** the name of the employee through the first transaction (`txn1`).
- ⑤ This helper function reads **only** the start date of the employee through the second transaction (`txn2`).
- ⑥ This helper function changes **only** the name of the employee and commits the first transaction (`txn1`).
- ⑦ This helper function changes **only** the start date of the employee and commits the second transaction (`txn2`).
- ⑧ This is the control flow which ensures that these different functions are executed in order, so we can be sure of the overlap described.

As you learned above, despite these two transactions modifying the exact same row, the locking is at the **cell** level, which means that these two transactions do not overlap one another at all. This is specifically because there was no overlap in the cells read or modified. To see that this works as expected, if you run the script above you should see output looking something like the following.

#### **Listing 6.27. Output from two non-conflicting read-write transactions**

```
$ node run-transactions.js
table: [ [ { name: 'name', value: 'Steve Jobs' },
  { name: 'start_date', value: 1976-04-01T00:00:00.000Z } ] ]
 txn1: [ { name: 'name', value: 'Steve Jobs' } ]
 txn2: [ { name: 'start_date', value: 1976-04-01T00:00:00.000Z } ]
 txn1: [ { commitTimestamp: { seconds: '1490101784', nanos: 765552000 } } ]
 txn2: [ { commitTimestamp: { seconds: '1490101784', nanos: 817660000 } } ]
table: [ [ { name: 'name', value: 'Steve Jobs (updated)' },
  { name: 'start_date', value: 1976-04-02T00:00:00.000Z } ] ]
```

This is pretty neat, but what about if the second transaction also **read** the name value. This would mean that our control flow would look something like:

#### **Listing 6.28. Looking at the name causes the transaction to fail**

```
const printNameAndStartDateFromTransaction2 = () => {
  const startDateQuery = {keys: ['1'], columns: ['name', 'start_date']}; ①
  return txn2.read('employees', startDateQuery).then((results) => {
    console.log('txn2:', results[0][0]);
  });
}

/* ... */

printCommittedEmployeeData()
  .then(printNameFromTransaction1)
  .then(printNameAndStartDateFromTransaction2) ②
  .then(changeNameFromTransaction1)
  .then(changeStartDateFromTransaction2)
  .then(printCommittedEmployeeData);
```

- ➊ This helper function is almost identical to `printStartDateFromTransaction2` however it also includes the name column.
- ➋ Instead of printing just the start date, we'll also print the name value.

As you might guess, since we've read an outdated version of the name value from the second transaction, after the first transaction commits, the second will fail! This is because we can't be sure that the second transaction didn't make any bad decisions based on stale data. The error result is shown below.

#### **Listing 6.29. Error result due to reading stale data**

```
table: [ [ { name: 'name', value: 'Steve Jobs (updated)' },
          { name: 'start_date', value: 1976-04-02T00:00:00.000Z } ] ]
txn1: [ { name: 'name', value: 'Steve Jobs (updated)' } ]
txn2: [ { name: 'name', value: 'Steve Jobs (updated)' },
          { name: 'start_date', value: 1976-04-02T00:00:00.000Z } ]
txn1: [ { commitTimestamp: { seconds: '1490116055', nanos: 805223000 } } ]
Error! Transaction was aborted. It was wounded by a higher priority transaction due
to conflict on key [1], column name in table employees.
```

Obviously transactional semantics and concurrency are both very complicated, so there is far more information that we can't go into in this chapter, however Spanner's online documentation is pretty detailed and worth a read. Further, the general guideline when it comes to transactions is to be specific about the data you want from Spanner, and put critical pieces that must be atomic inside transactions, and Spanner is capable of doing "the right thing" to make sure that your queries execute both safely (that is, correctly) and optimally (that is, as fast and at the highest levels of concurrency possible).

Now let's move on from these more advanced topics and take a quick look at how much all of this will cost you.

## **6.6 Understanding pricing**

Cloud Spanner pricing has three different components: computing power, data storage, and network cost. That said, network cost is not typical in most Spanner configurations. Let's start by looking at the computing power.

Similar to Cloud SQL, Cloud Spanner is billed by total number of nodes created and priced on an hourly basis, with variations in price depending on the location (e.g., Asia tends to be more expensive than the central US). Unlike Cloud SQL, the replication that happens under the hood is baked into the overall hourly price.

Spanner currently runs at \$0.90 USD per node per hour (for a US-based instance), with a recommendation of a three-node instance for anything that needs "production-level availability". Under the hood, all configurations are currently replicated across three different zones, meaning that in total a "3-node instance" is actually 9 total nodes (3 node replicas each in 3 different zones). To put this in perspective, the total monthly computing power cost for a 3-node Cloud Spanner instance in the central US works out to around \$2,000 USD per month.

In addition to computing power, the data stored in Spanner is charged at a rate of \$0.30 USD per month. Unlike Compute Engine's persistent disks, this storage space is measured based on how much data you actually have rather than a specific block of data that you've provisioned. This rate means that a Spanner database holding 1 TB of data would cost around \$300 USD per month in total.

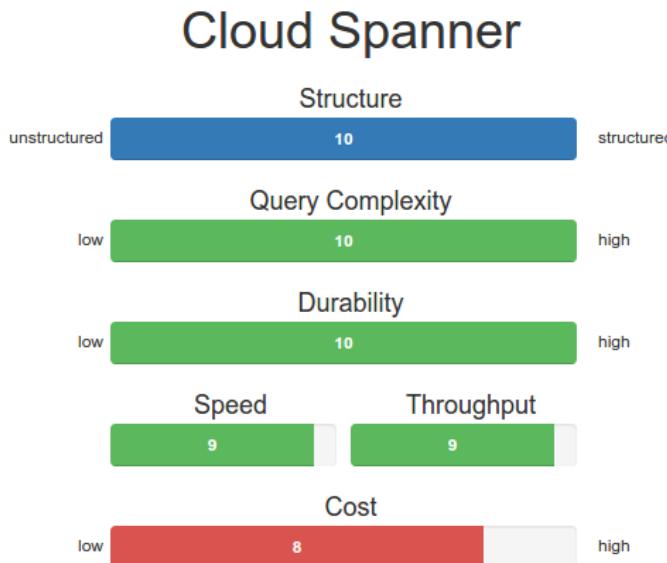
Lastly, any data sent from Spanner to the outside world (that is, to machines outside of Google's network) or between separate regions (that is, from a Spanner instance in `asia-east1` to a Compute Engine VM in `us-central1-c`) are charged at global network rates which varies by location but is currently \$0.01 USD per GB in the US.

Generally, when you're using Spanner the queries you run will be sending data to and from Compute Engine or App Engine instances in the same region, meaning that the network cost is complete free (since those don't leave the Google Cloud network). This cost can become meaningful if you try to run an "export" of your data outside of Google Cloud or send lots of queries across multiple regions. Now that you understand how billing works, let's look through what factors make Spanner a good or bad fit for your projects.

## 6.7 When should I use Cloud Spanner?

To figure out whether Cloud Spanner is a good fit, let's start by looking at the score card which summarizes the various criteria that we might care about.

**Figure 6.24. Cloud Spanner scorecard**



### 6.7.1 Structure

Spanner is a full-featured SQL-style database, which means that you define columns

which have specific types and data will be rejected if it doesn't fit in properly. This also includes NOT NULL modifiers which makes certain columns required, so on the scale of how structured we'd consider Spanner, as high up as possible (alongside Cloud SQL or any other SQL database).

Interestingly enough, Spanner also imposes additional structure which is not possible with traditional SQL databases with the ability to interleave a child table into a parent table. In a sense, if this scale could go any higher, Spanner would be right there at the top.

### **6.7.2 Query complexity**

Just as Spanner topped the charts on overall structure (due to being as structured as a regular SQL database and adding additional structural components with table interleaving), it also tops the charts when it comes to query complexity. Not only can you do single key lookups and specify which columns you're interested in, you can do arbitrarily complex SQL statements involving joins across tables, fancy groupings, and advanced filtering of rows. This level of query complexity is generally not available in other databases that are focused on providing high performance and availability such as Cloud Bigtable, making this a pretty powerful feature.

### **6.7.3 Durability**

Just like Cloud Datastore, Cloud Spanner is replicated across multiple different zones, which ensures that data, once persisted, doesn't go anywhere. This is made explicit with the transactional semantics such that not only is there no need to worry about data loss, it's also clear exactly when a transaction has been committed which means that you always have a consistent view of the world about what data is committed and what isn't.

### **6.7.4 Speed (latency)**

When it comes to overall query latency, Spanner's key lookups are extremely fast. For other more complex queries, obviously there will be some additional latency, but in general most queries to Spanner should complete within a few milliseconds. What's really impressive is that Spanner latency can be kept consistently fast even as request load increases so long as the number of nodes is turned on to handle the load. Should the latency increase, the simple fix to this is to turn on more nodes which will split the work up and keep queries fast.

### **6.7.5 Throughput**

Unlike object storage systems or file systems, Spanner's benefit is not in how many bytes it can ship over the wire in a given amount of time (throughput) but how quickly it can respond to a give query (latency). Further, the data stored in Spanner is generally large in overall size, but smaller on a per-query basis. This means that while overall system throughput may be sufficiently large, the per-query throughput is not typically measured (think about how often you thought about how many MB per second could be sent out of your MySQL instance). That said, Spanner as a whole is capable of large

overall throughput, and scores highly on the scale, in line with other systems like Cloud Bigtable.

### 6.7.6 Cost

With the overall cost being around \$650 USD per node per month, and the general guideline being to have at least 3 nodes for any production traffic, Spanner comes in at the high end of the price range, costing almost \$2,000 USD per month in the minimum suggested configuration. This is certainly much more than a single small VM running a database (either unmanaged through GCE or managed using Cloud SQL) which costs around \$50 USD per month.

The primary difference lies in the fact that the number of nodes is actually triple what is shown due to Spanner's full replication across three different zones. Looking at that in numbers, it means that the true cost for a single node in a single zone is \$0.30 USD per hour, which comes to about \$200 USD per node per month. This puts you in the same overall price range as a 4-core Cloud SQL machine (`db-n1-standard-4`), so if you were deploying a 9-node cluster of these machines you're monthly costs would come to \$1,750 USD per month which is right in the same range as Cloud Spanner. In other words, when you might use a large SQL cluster to handle your database traffic, Spanner will cost around the same amount and handle all of the management and replication for you automatically. In short, while Spanner does rank highly on the overall cost scale, this is primarily because you're getting so much computing power which is masked by replication under the hood.

### 6.7.7 Overall

Now that you can see how Cloud Spanner works and where it shines, let's look through our sample applications (the to-do list, InstaSnap and E\*Exchange) and see how they each stack up.

#### To-do list

As you learned before, the to-do-list application is certainly not in need of either of the performance characteristics of Cloud Spanner (neither the super low latency or the high throughput). The structure offered will come in handy, but it seems like the other aspects all end up being overkill.

**Table 6.4. To-do list application storage needs**

Aspect	Needs	Good fit?
Structure	Structure is fine, not necessary though	Overkill
Query complexity	We don't have that many fancy queries	Overkill
Durability	High, we don't want to lose stuff	Definitely
Speed	Not a lot	Overkill
Throughput	Not a lot	Overkill

Overall, Cloud Spanner is an acceptable fit if you don't care at all about your bank

account. For all of the performance related aspects as well as the querying abilities, using Spanner for this project is a bit like swatting a fly with a sledge hammer. That said, if the to-do list application could become something enormous, where everyone in the world were using it, then Cloud Spanner starts becoming a much better fit as a traditional SQL database may start falling over after the first billion users.

### **E\*Exchange**

E\*Exchange, the online trading platform, is a bit more complex compared to the to-do list, specifically when it comes to the queries that need to be run, and the transactional semantics that are needed to ensure that concurrent users don't clobber one another. This leads to Cloud Spanner being a slightly better fit.

**Table 6.5. E\*Exchange storage needs**

Aspect	Needs	Good fit?
Structure	Yes, reject anything suspect, no mistakes	Definitely
Query complexity	Complex, we have fancy questions to answer	Definitely
Durability	High, we <b>cannot</b> lose stuff	Definitely
Speed	Things should be pretty fast	Definitely
Throughput	High, we may have lots of people using this	Definitely

Looking through this, it looks like E\*Exchange is a pretty great fit, offering the advanced querying and transactional semantics that you need for the project, but also keeping queries fast (low latency) even under heavy load (high throughput).

### **InstaSnap**

InstaSnap, the very popular social media photo sharing application, has a few requirements that seem to fit really well and only a couple that are a bit off.

**Table 6.6. InstaSnap storage needs**

Aspect	Needs	Good fit?
Structure	Not really, structure is pretty flexible	Overkill
Query complexity	Mostly look-ups, no highly complex questions	Overkill
Durability	Medium, losing things is inconvenient	Overkill
Speed	Queries must be very fast	Definitely
Throughput	Very high, Kim Kardashian uses this	Definitely

As you can see, some of the querying features are overkill for InstaSnap as it's more key-value oriented, however the ability to remain fast as more and more people start using the app makes Spanner "less overkill" than it was for other simple apps (such as our to-do list).

The primary concern for InstaSnap is about single query latency as request load goes through the roof (that is, when a famous person posts a photo and the whole world wants to see it at the same time), and in this scenario Cloud Spanner does very well,

and will do even better with a cache, like Memcache, around to help out.

## 6.8 Summary

- Spanner is a relational database (like MySQL) with the scaling abilities of a non-relational database (like Cassandra or MongoDB).
- Spanner has the ability to automatically split data into chunks based on hints you provide, which allows it to evenly spread request load across many different servers, keeping query latency low even under heavy load.
- Spanner is always deployed in a replicated regional configuration, with multiple complete replicas in several zones.
- Spanner is generally a good fit when you need the features of a SQL database, but the scalability of a non-relational system.



# *Cloud Bigtable: Large scale structured data*

**This chapter covers:**

- What is Bigtable? What went into its design?
- Data modeling and infrastructural key concepts
- How to create Bigtable instances and clusters
- How to interact with your Bigtable data
- When is Bigtable a good fit?
- What's the difference between Bigtable and HBase?

## 7.1 **What is large-scale data storage?**

Over the years, the amount of data stored has been growing and continues to grow considerably. One reason for this is the fact that businesses have become more interested in the history of changes made to data over time than just a snapshot at a single point. And storing every change to a given value takes up much more space than a single instance of a value. This is exacerbated by the fact that the cost of storing a single byte has dropped significantly, making it far cheaper to keep data around "in case you need it later". Following this practice has led to engineering projects focused on discovering more uses for all of this "just in case" data such as machine learning, pattern recognition, and prediction engines.

These new uses have lead to a need for storage systems that can provide very fast access to extremely large datasets, while also maintaining the ability to update these datasets continuously. One of these systems is Google's Bigtable which was first announced in 2006, and has been re-implemented as the open-source project Apache

HBase. Based on the success of HBase in the wild, Google launched Cloud Bigtable as a managed cloud service to address the growing need for these large-scale storage systems. So what is Bigtable? Let's explore that next and dig into some of the technical details that went into building Bigtable.

## 7.2 What is Bigtable?

Bigtable began as the storage system for the web search index at Google and has since become one of the main technologies backing many of the other storage systems at Google, such as Megastore and Cloud Datastore. It was built to solve a very specific, but very complex problem: how do you store and continuously update petabytes of data, with incredibly high throughput, low latency, and high availability?

The obvious question is: why can't we just toss all of this into MySQL? Obviously MySQL falls over pretty quickly in attacking this problem, so Google came up with an interesting way of using a globally sorted key-value map, which automatically re-balances data based on service utilization to reach the performance and scale requirements needed. Let's now look more closely at the design goals (and non-goals) that went into building Cloud Bigtable, and how they affect whether or not you should use Bigtable in your own applications.

### 7.2.1 Design goals

Since the primary driving use-case for Bigtable was the web search index, let's look specifically at those requirements. Since Google's web search index is one of those things that must be "always on, always fast", it should come as no surprise that many of the requirements are related to both performance and scale, which come at the cost of sacrificing many of the "nice to have" features common in modern databases.

#### LARGE AMOUNTS OF (REPLICATED) DATA

The first thing to consider is that the search index will obviously be enormous, with overall sizes measured in petabytes, which means that it's far too large for any single server to manage. This is also a benefit in that one of the hidden requirements for the index would be that it is distributed across many different servers, each one being in some sense "commodity hardware" (aka, cheap). This problem is further exacerbated by the need to ensure that the data itself was stored in more than one place — after all, hard drives and servers fail from time to time and you wouldn't want a chunk of the data to disappear (even temporarily) due to occasional hardware failure.

#### LOW LATENCY, HIGH THROUGHPUT

Regardless of the size of the data to be stored, the search index will clearly be seeing a ton of traffic, potentially millions of queries every second. If the search index starts falling over as more and more requests come in at the same time, folks will take their searches elsewhere, which is bad.

Additionally, each search request needs to return a result really really fast, measured in milliseconds, which includes sending the search results back over the wire. When you

include all of the other things that need to happen to hit that overall deadline, this leaves relatively little time to "query the database", likely only a few milliseconds. This very likely means that anything more than "get the data at this address" will exceed the time deadline.

### RAPIDLY CHANGING DATA

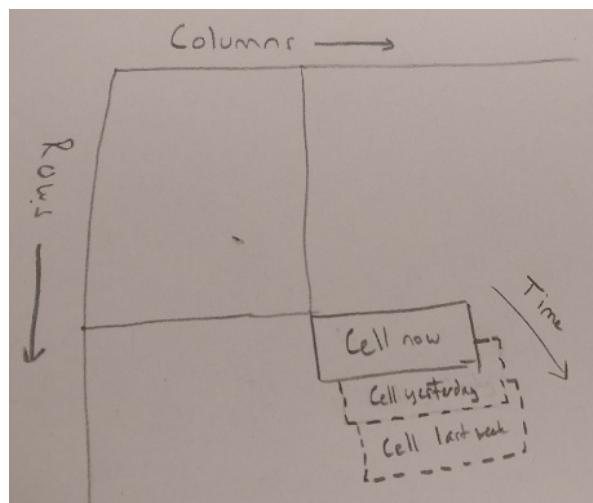
The next thing to think about is that new web pages are added all the time, and the search index will be updated by a web crawler all the time. This means that regardless of the number of queries asking to find web pages (number of reads per second), the system will need to handle lots of updates at the same time (number of writes per second).

While these writes likely have a less extreme latency requirement (e.g., they can take longer than a user-facing search request), if these updates take too long they will start to pile up and the index will slip out of date, which would be very bad. This means that while a single write request can take longer to finish the total number of write operations that can be done in a given period of time needs to be a large number.

### HISTORY OF DATA CHANGES

Since the data we're storing will be rapidly changing over time, we want a way to easily see the data as it was at a particular point in time. This could be done manually by the client by constructing keys with timestamps to signify which version of the data we're referring to, however letting the storage system keep track of change history will keep our clients thin and simple. In some ways, you can think of this as a "third dimension" to data, where typically databases have a row and column position (2 dimensions), to see history of the data in that row we'll need a third (time) dimension.

**Figure 7.1. Time as a third dimension in Bigtable**



With this ability, we'll be able to ask for "the latest" value in a row, as well as all the

values that this row has had over time if we're interested in that data.

### **STRONG CONSISTENCY**

Next up is the need for "strong consistency", which basically means that there's no way for someone querying the index to ever see "stale data". In other words, any updates either happen everywhere or don't happen.

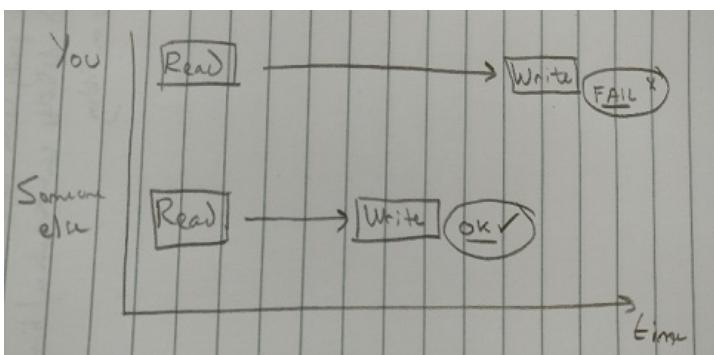
If the system didn't have this property (and was, say, eventually consistent instead) it'd be possible for someone to search for the same thing in two browser windows and see different results, which is definitely not good.

### **ROW-LEVEL TRANSACTIONS**

In addition to always presenting a consistent view of the world, this system would need to allow atomic "read-modify-write" sequences or else risk two updates clobbering one another. This means that the system must expose a way to return an error in the case that someone else has changed the data of a row while you're attempting to work on it.

To make this more concrete, the following diagram shows what we want to happen when two competing writes overlap during a transaction on a single row.

**Figure 7.2. What should happen if two clients write over-top of one another**



While this is definitely a requirement for a single row, it's unlikely that we'll have multiple rows in the search index that would require an atomic update across those rows. This means that while this system would need to provide atomic writes for a single row to avoid write contention, it would not need general transactional semantics across multiple rows.

### **SUBSET SELECTION**

Finally it's important to remember that we don't always want to request **all** of the data stored for a given set of results, so it'd be nice if the system had a way of asking only for a specific set of properties. This could be a specific set of column families, columns, or timestamps which would allow us to ask for things like "only the two most recent values".

By having the ability to limit which pieces the storage system should return to us, it

follows that we're able to store more data in one chunk and request only small bits of that large chunk.

### 7.2.2 Design non-goals

It's important to remember that in exchange for all of these "features" there are quite a few things that are **not** necessarily required. In other words, they'd be lovely to have, but we can do without them if it makes the other aspects possible.

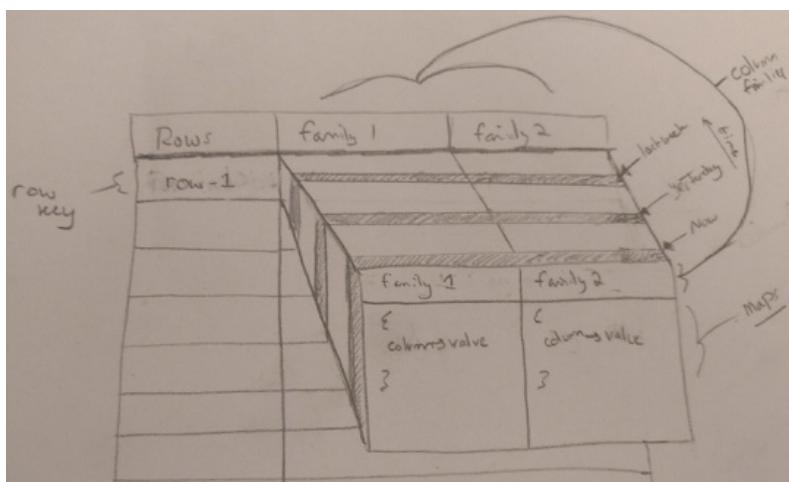
In the case of Bigtable, it turns out that the enormous scale of the datasets combined with the throughput and latency requirements means that we'd need to drop most of the "nice to have" features such as secondary indexes (e.g., the ability to run queries like `SELECT * FROM users WHERE name = "Jim"`), multi-row transactional semantics, and many of the other things we've all come to expect from "databases".

### 7.2.3 Design overview

What came out of all of these requirements was a pretty unique storage system, which did things quite differently from most of the non-relational systems that existed at the time (2006).

As the name suggests, Bigtable is a very large "table" of data with some important differences from the tables we've all come to know. While in many ways it can act like a traditional table, the storage model of Bigtable is much more like a jagged key-value map than a grid. As a matter of fact, the authors of the research paper describing Bigtable called it "a sparse, distributed, persistent, multi-dimensional sorted map" (the key word at this point being "map"). Put visually, this design looks something like Figure 7.3

**Figure 7.3. Bigtable design overview**



In short, this means that Bigtable is less like a relational database and a bit more like a big key-value store, which distributes data across lots of servers, while keeping all the

keys in that map sorted. Thanks to that global sorting, Bigtable allows you to do both key lookups (as you would in any key-value store) as well as scans over key ranges and key prefixes.

Lastly, hidden in this list of features is the idea that the map is "multi-dimensional". In this case, the extra dimension attached to all data stored in Bigtable is a timestamp which effectively allows you to "go back in time" and view data as it was at a previous point. It is this unique set of features that makes Bigtable so powerful.

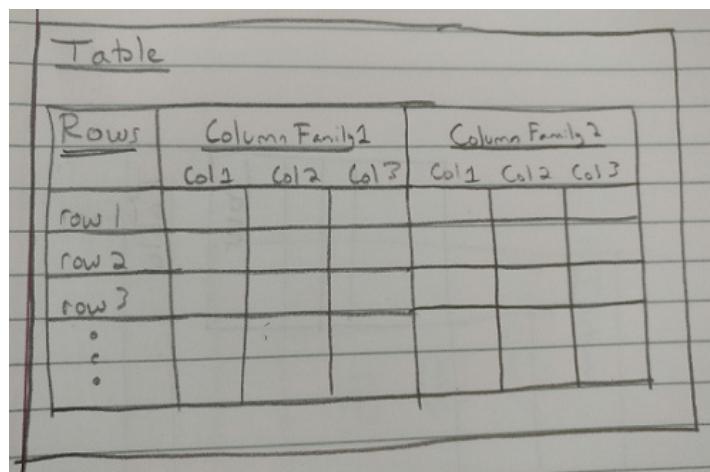
## 7.3 Concepts

While Bigtable is incredibly powerful, it will require you to think a bit differently about the structure and access patterns of your data, similar in some ways to our previous discussion of Cloud Datastore. This means that, generally, you'll have to think ahead of time about what types of questions you'll want to ask about your data, as the ability to answer different questions will be determined in many ways by the way in which you structure that data.

### 7.3.1 Data model concepts

Let's kick things off by looking at the concepts you'll need to understand, starting with the data model concepts show in the following diagram: tables, rows, column families, and column qualifiers.

**Figure 7.4. Data model concept hierarchy**



It might be important to note that while the data model concepts apply most specifically to Cloud Bigtable, they're also going to be relevant if you happen to use HBase which was designed following the publication of the Bigtable paper in 2006. The second section, on the other hand, which is focused on the infrastructural details of Cloud Bigtable as a managed service, will apply almost exclusively to Cloud Bigtable and very little to HBase.

## Row keys

While Bigtable may look a bit like a relational database at a glance, data stored is actually much more like a key-value store (as we saw with Cloud Datastore), where the "key" used to find content is called the "row key". This key can be anything you want, but as you'll read later, choosing the format of this key should be done carefully.

If you're familiar with relational databases, you've likely seen the term **PRIMARY KEY** somewhere in SQL to denote that a specific column is both unique and used to identify a given row. In Bigtable the row key is used for the same purpose, and you can think of it as the "address" of a given chunk of data. It's important to note that while Bigtable allows you to quickly find data using a row key, it does **not** allow you to find data using any secondary indexes (since they don't exist). This means that even though in a relational system you're able to do look-ups based on other columns (e.g., `SELECT * FROM users WHERE name = 'Jim'`) you cannot do this kind of lookup in Bigtable.

### Row key sorting

As mentioned above, choosing how to structure and format row keys is important for a few different reasons:

1. Row keys are always unique which means if you have collisions you'll overwrite data.
2. Row keys are lexicographically sorted across the entire table which means that high traffic to lots of keys with the same prefix could result in very serious performance problems.
3. Row key prefixes and ranges can be used in queries to make the query more efficient which means that poorly structured keys will require inefficient full-table scans of your data.

These reasons mean that some of the more common key formats are not a good fit for Bigtable. For example, in MySQL or PostgreSQL primary keys typically take the format of a sequence of numbers (1, 2, 3, 4, ...). Using an incrementing sequence for your row key means that you may have collisions, that you are very likely to encounter performance problems, and that you are precluded from doing queries by key-range as it's not super useful to say "Please give me Users 1 through 20".

The most subtle issue in choosing a row key is choosing a format that is both useful to you as the application developer and efficient for Bigtable to store, so take your time in choosing a key and make sure that whatever you choose fits the criteria above. To simplify this for you a bit, let's walk through some examples of row key formats that would be a particularly good fit for Bigtable.

### String IDs (hashes)

If your identifier is an opaque ID (such as `Person #52`), using a hash of that value (e.g., `'person_' + crc32(52)`). The hash ensures that the row key is both a fixed length and evenly distributed (lexicographically) throughout the key space.

The underlying assumption here is that while writes to the entire system (e.g., all

`person_rows`) may need to handle extremely heavy load, writes to a single person's row (e.g., `person_2b3f81c9`) are much more likely to be a tiny fraction of the overall load. Since all the rows will be evenly distributed, Bigtable can optimize where each row lives and evenly distribute the load across lots of different machines. This is discussed in more detail later.

### **Timestamps**

It's often the case that you'll need to retrieve data based on a point in time, which makes timestamps an obvious choice. It's important that you **do not use a timestamp** as the key itself (or the start of the key) as this will ensure that all write traffic will always be concentrated in a specific area of the key space, which would force all traffic to be handled by a small number of machines (or even a single machine).

A good rule of thumb is to prefix time-series data with another key that is useful for querying. For example, if you are storing stock price information over time consider prefixing the row key with the hash of the stock ticker symbol (e.g., `stock_c318f29c#1478519731` which is `'stock_' + crc32(GOOG) + '#' + NOW()`).

### **Combined values**

Sometimes rows contain information relating two different concepts, for example, a "tag" of a person in a post involves the person tagged and the person doing the tagging. In these cases, you can actually combine the two keys into a single key to simplify looking up all messages between two people.

This would mean that if Alice tagged Bob in a post, you might use a key that looks like `post_6ef2e5a06af0517f` which is `'post_' + crc32(alice) + crc32(bob)`. You can then store the post content in the row data.

### **Hierarchical structured content**

The last format, similar to Java package naming formats, is to use a "reverse hierarchy" prefix format. The most common example of this is "reverse domain name" such as `com.manning.gcpia(gcpia.manning.com backwards)`, which makes row key ranges very convenient in that you can ask for everything belonging to `manning.com` (prefixed with `com.manning.`) or everything belonging to `gcpia.manning.com` (prefixed with `com.manning.gcpia.`).

This works well with anything hierarchical in that the reversed hierarchy allows filtering by providing a longer (more specific) prefix, however the assumption is that specific rows would follow the guidelines above using hashed final values to ensure relatively even key distribution.

Note that non-reversed hierarchical representations does **not** put related rows next to one another (e.g., `gcpia.manning.com` is not lexicographically next to `forum.manning.com`), so it's not a good idea to use the non-reversed format.

Now that you have a better grasp on what row keys are, let's look at the data that they

point to and how that is structured.

### COLUMNS AND COLUMN FAMILIES

In many key-value stores, the data that's pointed at by a particular key is a completely unstructured piece of data. It could be a bunch of bytes representing an image, or it could be JSON document storing a user profile, but the storage system typically has no understanding (nor need to understand) what the value actually is. With Bigtable, even though you don't define secondary indexes, it does allow you to define certain aspects resembling a "schema", which makes it easy to specify which bits of data to retrieve. This "schema" is really more like extra criteria to define the structure of a key-value map that will ultimately hold your data.

In the world of Bigtable the keys of this map are called "column qualifiers" (sometimes shortened to "columns") which are often dynamic pieces of data, where each of these belongs to a single "family" which is a grouping that holds these "column qualifiers" and act much more like a static column in a relational database. This may seem very strange, and at first it is, so don't be worried. It's a unique combination of static and dynamic data, and oddly enough, it means that, unlike normalized SQL databases, "column qualifiers" can be anything you want and can be thought of as "data"—something you'd never do in a relational database. This type of structure also means that when you visualize data in Bigtable as an actual table, most of the cells will be empty. This is what we mean above when we say a "sparse ... map".

### VISUALIZING YOUR BIGTABLE DATA

To make this more concrete, let's imagine a to-do list where we're storing items that have been completed. If you don't recall, the to-do list application was a simple tracker of items that each user wants to complete, along with tracking when each item was completed and any notes recorded at that time. To store this same data in Bigtable, we'd have a "completed" column family (this is the static part) where each individual column qualifier corresponds to the ID of the item (these are the dynamic "keys" of each row). In addition, we may add another column qualifier to store optional "notes" that we may want to write down when completing the item. This would look something like Table 7.1.

**Table 7.1. Visualizing the to-do list**

Row key	completed			
	item-1	item-1-notes	item-2	item-2-notes
237121cd (user-3)			true	"Right on time!"
4d4aa3c4 (user-1)	true		true	"2 days late!"
946ce0c9 (user-2)				

While this looks similar to any other relational database table at first, when you look more closely it starts to seem pretty strange and inefficient. Why aren't the completed items stored in a table with a user ID, an item ID, and a completed field? And then why are notes stored on that particular row? And why are there so many empty spaces? Isn't

that inefficient?

First, notice the row key is acting as our "address" or the lookup key of where to find the data. This means that while it looks like a "relational table", our data is probably better thought of as a key-value store. (For example, there is no way to query for all users with item-1 completed). Second, each row stores only the data present in that row, which means that there is no penalty for those empty spaces you see. This is what we mean by referring to Bigtable as a sparsely populated table. Finally, the column qualifiers (e.g., item-1) can be thought of as a dynamic value adding further detail to the "completed" column family. That is, the static part (similar to the column name in a relational database) is the "completed" column family. Inside that column family is an arbitrary set of dynamic data, but we happen to visualize that as somewhat of a "sub-table" with more columns and values for those columns.

To put this differently you could also visualize this as a key-value store where the keys are further maps of data, a bit like how we first saw Cloud Datastore.

**Table 7.2. Visualizing the to-do list as maps**

User	Data
237121cd (user-3)	{item-2: true, item-2-notes: "Right on time"}
4d4aa3c4 (user-1)	{item-1: true, item-2: true, item-2-notes: "2 days late!"}
946ce0c9 (user-2)	{}

However the "completed" column family is really a static key in the map, so we could look at this differently by adding some hierarchy.

**Table 7.3. Visualizing the to-do list as maps with hierarchy**

User	Data
237121cd (user-3)	{item-2: {completed: true, notes: "Right on time"}}}
4d4aa3c4 (user-1)	{item-1: {completed: true}, item-2: {completed: true, notes: "2 days late!"}}
946ce0c9 (user-2)	{}

**Table 7.4. Visualizing the to-do list as maps with a different hierarchy**

User	Data
237121cd (user-3)	{completed: {item-2: true, item-2-notes: "Right on time"}}
4d4aa3c4 (user-1)	{completed: {item-1: true, item-2: true, item-2-notes: "2 days late!"}}
946ce0c9 (user-2)	{}

These are all simply different ways of looking at the same data, but for the rest of the chapter we'll use the format you see in Table 7.1 which is similar to what you'll see in documentation for Bigtable as well as HBase.

Notice how in the final visualization all of the data is grouped first with a key called completed. Even though we only have that single column family, you could imagine further column families in the table, perhaps a followers column family

which shows who is watching the items you complete. With that, it becomes a bit more clear why column families are useful, in that you're able to ask specifically for the chunks of data you want. In this case, if you wanted to see what items were completed by a user, you'd ask for only the `completed` column family which would return just the data in the `completed` key as you see above. If you wanted to see followers, you could ask for the `followers` column family, which would give you just that subset of data. In other words, column families are really helpful groupings that, in some ways, can be thought of as the keys pointing to maps of arbitrary maps of more data.

Now that you understand what columns and column families are, let's explore the different layouts of tables which, as you might guess, give you the ability to query your data in different ways.

#### TALL VERSUS WIDE TABLES

So far our example above used what we'll call a "wide" table, which is a table having relatively few rows but lots of column families and qualifiers. In the example, each row stores the data about a particular user, which contains quite a few potential column qualifiers (one for each item that user completes). This allows us to ask very useful questions such as "What items has user-020b8668completed?" However, what if we wanted to know whether a user has completed a particular item? This is where a "tall" table would come into play.

As you might guess from the definition of a "wide" table, a "tall" table is one with relatively few column families and column qualifiers, but quite a few rows, each one corresponding somehow to a particular data point. In this case the row key would be a combination of values as we discussed in the previous section, which is easy to calculate given the data you're interested in. For example, using a "tall" table to store whether a given item was completed, we might use a combined hash of the user and the item, which might look like `4d4aa3c4931ea52a` (`crc32(user-1) + crc32(item-1)`). In the column data we might store the notes in a similarly named "completed" column family. This would make our tall table look something like Table 7.5.

**Table 7.5. Tall table version of the to-do list**

Row key	completed	
	item-id	notes
<code>4d4aa3c4931ea52a</code> (user-1, item-1)	item-1	
<code>4d4aa3c44a38e627</code> (user-1, item-2)	item-2	"2 days late!"
<code>b516476c4a38e627</code> (user-3, item-2)	item-2	"Right on time"

There are quite a few differences with this table style compared to what we've discussed so far. The first and most obvious difference is that rather than growing out wider as more items are completed will actually grow longer (or taller) instead. The shape itself is interesting, but this leads to more differences. In addition to looking different on paper, this tall version of our table allows you to query very quickly to ask the very simple question "Did user-1 completed item-1?" In this case, that is a matter

of computing the proper row key (`crc32(user-1) + crc32(item-1)`) and checking if the row exists.

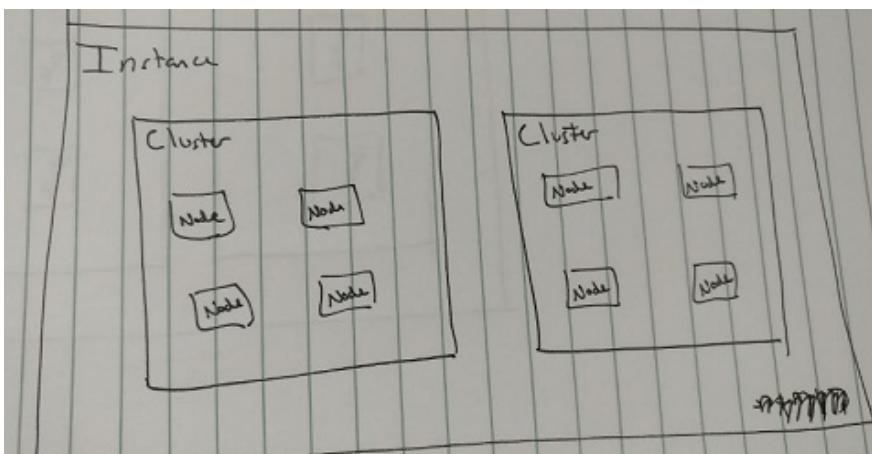
Finally, this table allows you to ask the question "What items did `user-1` complete?" however answers it in a different way. In the first example (using a wide table), looking up the complete items was a single "get" for a given user, but in this new example, to find out the list of items completed for a user you would execute a scan based on a row prefix, in this case `crc32(user-1)`. This prefix would return all rows starting with that value, which you can then iterate through to find all of the items that were completed (one per row).

While these two tables do ultimately allow you to ask similar questions, it would appear that the tall version allows you to be a bit more specific at the cost of more single entry lookups to get bulk information. This means that if you're going to be asking bulk-style questions (e.g., "What did user X do?") a wide table may be a better fit and if you intend to ask more specific questions ("Did user X do thing Y?") a tall table is likely to be a better fit. Now that we've gone deeper into the data modeling concepts, let's switch back to the infrastructural world and see how exactly you turn on and use Cloud Bigtable.

### 7.3.2 Infrastructure concepts

As we discussed before, Cloud Bigtable acts as a managed service, which means that you don't have to manage individual virtual machines like you would if you were running your own HBase cluster. However, with the automated management there are some new concepts that you'll need to understand. Unfortunately, Bigtable is one of the services that is a bit more confusing, particularly due to how replication is handled. It's also a bit confusing because Bigtable itself has a concept of a "tablet" which isn't directly exposed via the Cloud Bigtable API. To try to keep things as simple as possible, let's start by first looking at the hierarchy of concepts that you can actually manage yourself: instances, clusters, and nodes.

**Figure 7.5. Hierarchy of instances, clusters, and nodes**



As you can see, the basic structure here is that an *instance* is the top-most concept which can contain many *clusters*, and each cluster contains several *nodes* (with a minimum of 3).

### **INSTANCES**

Think of an instance as the primary resource you refer to when thinking about your Bigtable deployment, similarly to how you'd think of "the database server" when deploying a MySQL cluster (with a primary and a read-slave). This means that when you write data "to Bigtable", you'd refer to writing it to a specific "Bigtable instance".

Unlike a MySQL cluster where you always write data to the primary, in Bigtable you simply send your data to "the instance" which ensures that those changes are propagated to all the other clusters. While you can address specific clusters directly if needed, this shouldn't be necessary as Bigtable should route your queries to the cluster that is closest, and therefore should be reliably fast. Instances are globally scoped, meaning that they will remain addressable regardless of whether a particular zone is experiencing an outage.

### **CLUSTERS**

Before we go into too much detail about clusters, let's start with an important caveat: while the diagram shows multiple clusters per instance, as of today this is currently not yet possible. Specifically, you're limited to a single cluster per instance. That said, Bigtable will almost certainly support replication with multiple clusters per instance in the future. Given that impending launch of the feature, let's look at how clusters function with the assumption that you'll soon be able to maintain many of them inside a single instance.

Clusters, unfortunately (or fortunately, depending on who you ask), are pretty boring. They are simply a grouping for a bunch of nodes, each of which is responsible for handling some subset of queries sent to a Bigtable instance. A cluster has a unique name, a location (zone), and some performance settings such as the type of disk storage to use as well as the number of nodes to run. This means that clusters themselves have an hourly computing cost, as well as a monthly storage cost to reflect the amount of data stored in that particular cluster. Since each cluster holds a copy of your data, more clusters would imply higher availability of your data with the obvious trade-off of higher costs. As you'd expect, your hourly computing cost goes up as you add more nodes, with the benefit that you'll never hit a bottleneck of "too many nodes" as has been known to happen with other systems such as HBase.

### **NODES**

Nodes are even more boring than clusters for one very important reason: from our perspective, they are basically invisible. While we talk about nodes as discrete individual entities, in reality you'll never actually experience them that way except for seeing them on your bill. The reason for this is that, while you can think of a cluster as grouping together multiple nodes, the nodes themselves are hidden from you in the API. This means that you're only able to communicate with a particular cluster which

is responsible for routing your request to a particular node.

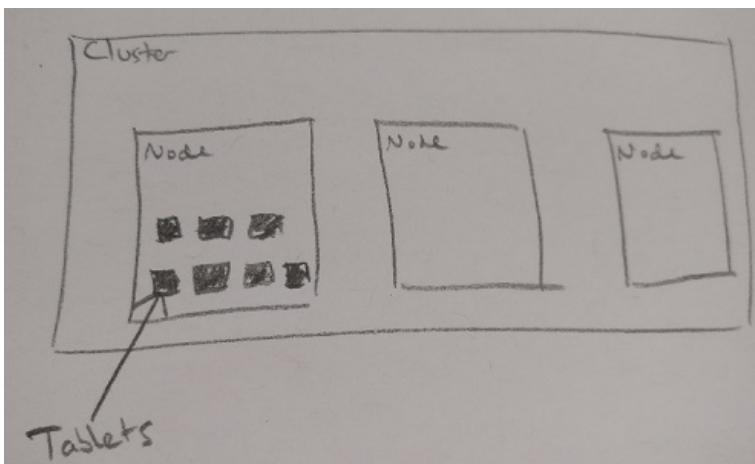
This allows the cluster to ensure that requests are spread evenly across the nodes, and also allows the cluster to re-balance data to maintain this even distribution. If nodes themselves were addressable the cluster wouldn't be able to move data around as freely, which could lead to a case where a single node held all the "hot data", driving down performance during busy times. In fact, this leads us to the Bigtable concept of a "tablet" which we haven't yet discussed, but is important to understand when you're concerned about performance.

### TABLETS

Tablets are a way of referencing "chunks" of data that live on a particular node. The cool thing about tablets is that they can be split, combined, and moved around to other nodes in order to keep access to data spread evenly across the available capacity. As with nodes, you'll never address tablets directly, so you won't see these in the API, but you can influence how data is written to tablets through the choice of your keys. For example, writing lots of data very quickly over a long period of time to keys with two distinct prefixes (e.g., `machine_` and `sensor_`) will typically lead to the data being on two distinct tablets (e.g., `machine_` prefixed data wouldn't be on the same tablet as `sensor_` prefixed data). Let's take a quick look at the progression of data as you add more (and query more) over time.

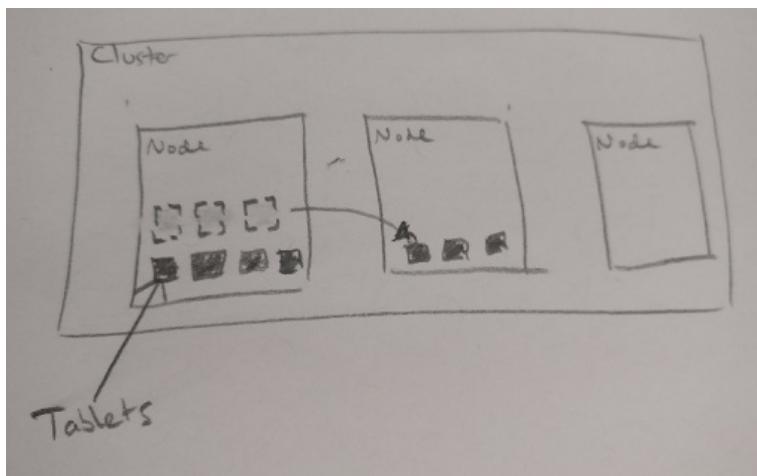
When you first start writing data, your Bigtable cluster will likely put most of the data on a single node, shown below.

**Figure 7.6. When starting, Bigtable might put data on a single node**



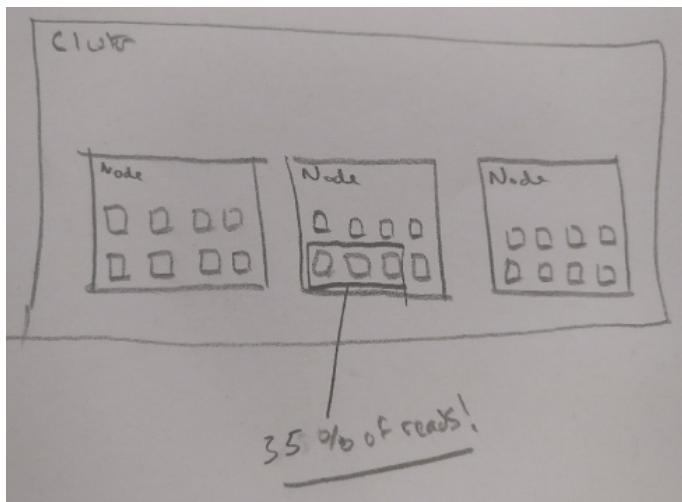
As more tablets accumulate on a single node, the cluster may relocate some of those tablets onto another node to redistribute the data in a more balanced fashion, shown here.

**Figure 7.7. Bigtable will redistribute tablets to spread data more evenly across nodes**



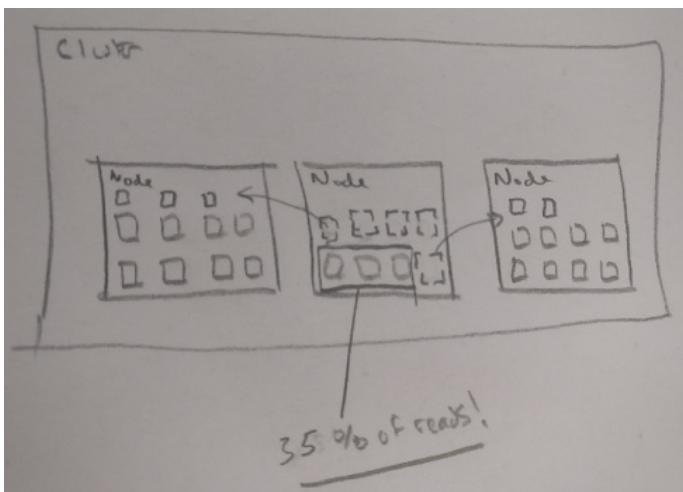
As more and more data is written over time, it's possible that there will be some tablets that are more frequently accessed than others. In this example, three tablets are responsible for 35% of all the read queries on the entire system.

**Figure 7.8. Sometimes a few tablets are responsible for a high percentage of traffic**



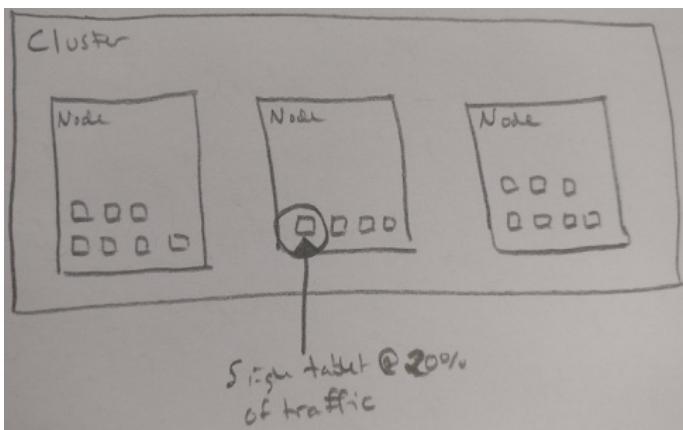
In scenarios like these, where a few "hot tablets" are co-located on a single node, Bigtable will re-balance the cluster by shifting some of the less frequently accessed tablets to other nodes that have more capacity in order to ensure that each of the three nodes sees about one third of the total traffic, shown here.

**Figure 7.9. Bigtable shifts data away from "hot tablets"**



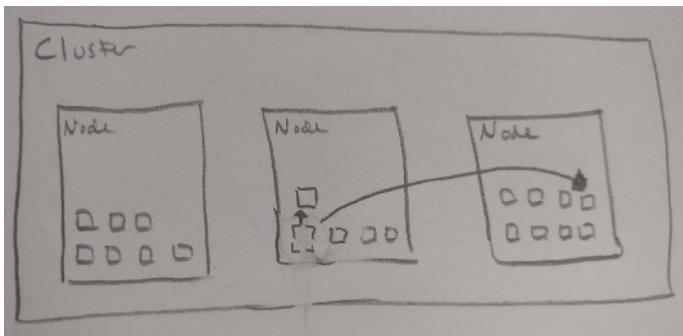
It's also possible that a **single** tablet could become "too hot" (that is, it's being written to or read from far too frequently). In scenarios like these, moving the tablet as it is to another node doesn't really fix the problem. Instead, Bigtable may *split* this tablet in half and then re-balance the tablets as we saw above, shifting one of the halves to another node.

**Figure 7.10. Sometimes a single tablet is responsible for a high percentage of traffic**



For example, in the scenario above, a single tablet represents 20% of the traffic to the cluster. To reduce the load on the single node, moving the table won't have much effect. As a result, Bigtable may split this tablet and move it elsewhere. In short, this allows the traffic of what once was a single tablet to be served by two different nodes.

**Figure 7.11. Bigtable splits tablets and shifts them to other nodes**



If you're a bit confused by the inner workings on Bigtable, that's OK. While it's great to understand nodes, tablets, and re-balancing of data, Bigtable itself is a very complex storage system and understanding every nuance is going to be incredibly difficult. In general, the most important thing you can do when using Bigtable is to choose row keys very carefully so that they don't concentrate traffic in a single spot. If you do that, Bigtable should "do the right thing" and perform well with your dataset. By now you should have a pretty decent understanding of how all the parts fit together, so let's take a minute to walk through how to manage your own Bigtable instance.

## 7.4 *Interacting with Cloud Bigtable*

As you saw before, Bigtable has relatively simple hierarchy involving instances, clusters, and nodes, and the data model for each of these is pretty simple as well, involving tables, rows, column families, and column qualifiers. But so far we've only talked about these. Let's take a look at how to actually create these different resources, using a combination of the Cloud Console and some command-line tools, starting with creating an instance.

### 7.4.1 *Creating a Bigtable Instance*

Before you can do anything with Cloud Bigtable, you first have to create a new instance. As we discussed before, currently you're limited to a single cluster per instance, which can be a bit off-putting when you're expecting a true hierarchy. For now let's try not to worry about that and go ahead with creating our instance.

Start by navigating into the Bigtable section of the Cloud Console using the left-side navigation. Then you can click on the "Create instance" button on the top which will bring you to a form with quite a few fields to fill out. It's important to start by filling in the first field (instance name) right away, and you'll notice that when you do that and click elsewhere on the form the next two fields (instance ID and cluster ID) will complete themselves automatically as you can see in Figure 7.12. The reason for this is simply that a cluster ID can be automatically computed from the instance name, and since there is currently a limit of one cluster per instance, this is currently not a very useful field (though it will be eventually).

**Figure 7.12. Bigtable instance identifiers**

A Cloud Bigtable instance is a container for your cluster. Choose the instance and cluster properties below.

### Instance properties

**Instance name**  
For display purposes only.

**Instance ID**  
ID is permanent. Use lowercase letters, numbers, or hyphens.

**Cluster properties**

**Cluster ID**  
ID is permanent. Use lowercase letters, numbers, or hyphens.

Next you'll need to choose a zone. As we discussed before, a cluster is a zonal resource which means that its availability is subject to that of the zone. This is where your data will live, and is therefore permanent for the cluster, so you should aim to choose a zone that is nearby to any VMs that need to read or write Bigtable data. In other words, this zone should be the same as where all of your VMs live.

**Figure 7.13. Bigtable zone setting**

**Zone**  
Choice is permanent. Determines where cluster data is stored. To reduce latency and increase throughput, store your data near the services that need it.

The next two pieces of information are specifically about the performance of your Bigtable instance. In this case you have two knobs you can turn to provide more or less capacity. The first is computing throughput where you set how many "nodes" you want to keep running. You can always change this number later, so don't worry too much about it, but the minimum you can choose is 3. As you might expect, the number of nodes you have will increase the read, write, and scan capacity of your instance in a mostly linear way. If you're just getting started, a good first choice is to leave this set

to 3, and expand your instance with more nodes later if you need the extra capacity.

The next piece related to performance is the type of disk to use to store your data. As you know from before, a solid-state disk is going to have much better performance in both latency and throughput, which makes a very big difference when in your Bigtable instance. For this reason, unless you specifically know that the SSD storage type is overkill for your use case and you know that standard disk (HDD) is acceptable, you should plan to leave this set to SSD.

**Figure 7.14. Bigtable performance characteristics**

**Nodes (3 – 30)** Add nodes to increase data throughput and queries per second (QPS). Contact us to request more than 30 nodes.

**Storage type**

- SSD (Recommended)** Most popular choice. Lower latency. Higher QPS and data throughput.
- HDD** May be preferable for very large data sets (>10 TB). Much cheaper per GB. Best for infrequently read data that can tolerate up to 200ms latencies.

**Performance** The storage type and the number of nodes in your cluster determine performance.

Reads	Writes	Scans
30,000 QPS @ 6ms	30,000 QPS @ 6ms	660 MB/s

For comparison, the following table shows the performance differences of the different storage types for Cloud Bigtable.

**Table 7.6. Storage type comparison for Cloud Bigtable**

Attribute	SSD (recommended)	HDD	Comparison
Read throughput	10,000 QPS / node	500 QPS / node	20x better with SSD
Read latency	6 ms	200 ms	33x better with SSD
Write throughput	10,000 QPS / node	10,000 QPS / node	Same
Write latency	6 ms	50 ms	33x better with SSD
Scan throughput	220 MB/s	180 MB/s	1.2x better with SSD

## 7.4.2 Creating your schema

As we learned, your schema will have long-lasting effects, so it's something we should try to get right ahead of time. Unlike a traditional relational database, updating the

schema later on isn't quite as simple as running an `ALTER TABLE` statement. Instead, it involves updating every single row you have stored to fit your new schema, similarly to how you would with any other key-value storage system. While this is certainly possible by adding code complexity (e.g., adding code that understands multiple schema versions, and when saving, rewrites data in the newest version), it's still worthwhile to invest time up-front to push any schema changes off into the distant future.

Although it's a terrible example of how to use Bigtable, let's use the to-do list project from before to test what it's like interacting with Bigtable. To refresh your memory, we'll be using the "tall" table format, where our row key is a combination of hashes for the user and the item with a single Column Family called `completed`. In this case, the column qualifiers themselves will be known in advance which is not always the case (as you saw in the "wide" table format). This means our table will look a bit like the following table.

**Table 7.7. Tall table version of the to-do list**

<b>Row key</b>	<b>completed</b>	
	<b>item-id</b>	<b>notes</b>
4d4aa3c4931ea52a (user-1, item-1)	item-1	
4d4aa3c44a38e627 (user-1, item-2)	item-2	"2 days late!"
b516476c4a38e627 (user-3, item-2)	item-2	"Right on time"

Before we can write some code to interact with Cloud Bigtable, we'll need to install the `@google-cloud/bigtable` client using `npm`, by running `npm install @google-cloud/bigtable@0.9.1`. Once the client is installed, we can test it out by listing out the instances and clusters, which looks like Listing 7.1.

#### **Listing 7.1. Listing instances and clusters**

```
const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

console.log('Listing your instances and clusters');

bigtable.getInstances().then((data) => {          ①
  const instances = data[0];
  for(let i in instances) {
    let instance = instances[i];
    console.log(`- Instance`, instance.id);
    instance.getClusters().then((data) => {        ②
      const clusters = data[0];
      const cluster = clusters[0];
      console.log(`  - Cluster`, cluster.id);
    });
  }
});
```

- ➊ Use `.getInstances()` to iterate through the list of available instances.
- ➋ Use `.getClusters()` to iterate through the list of clusters in an instance.

When you run this after creating an instance, you should see something like the following.

#### **Listing 7.2. List of the instances and clusters**

```
Listing your instances and clusters:
- Instance projects/your-project-id/instances/test-instance
  - Cluster projects/your-project-id/instances/test-instance/clusters/test-
instance-cluster
```

In this case, it appears exactly as we expect, with one instance and one cluster belonging to that instance. Now let's look at how to actually create our table and schema.

#### **Listing 7.3. Creating a table**

```
const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

const instance = bigtable.instance('test-instance'); ①

instance.createTable('todo', {
  families: ['completed']
}).then((data) => { ②
  const table = data[0];
  console.log('Created table', table.id);
}); ③
```

- ➊ This time we simply construct an `instance` object using its ID rather than trying to look it up via the API as we did before.
- ➋ Notice that you create a table with `instance.createTable()` rather than `cluster.createTable()`. As noted before, the instance is the owner of tables.
- ➌ Column families are just defined as a list of strings.

After running this, you should see output looking something like this:

#### **Listing 7.4. Confirmation that a table was created**

```
Created table projects/your-project-id/instances/test-instance/tables/todo
```

That's it! You've now created a table called `todo` with a single column family called `completed`. But a schema alone isn't all that useful, so let's look now at how we can manage the data that goes in your table.

### 7.4.3 Managing your data

As with any storage system there are two sides to managing data: one going in (writing), the other going out (reading or querying). Let's start by adding some new rows to our todo table, and then we'll see how we can query to get these rows back. If you recall from before a single row in our "tall" table will have a row key that is a concatenated hash of the user ID and item ID, and then our columns would be statically defined as `item-id` and `notes`.

**NOTE**

We'll use `CRC32` as the "hash" for this exercise, which means you'll need to install the library by running `npm install fast-crc32c`.

The code to add a row completing an item would look something like the following.

#### Listing 7.5. Inserting data into Bigtable

```
const crc = require('fast-crc32c');
const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

const instance = bigtable.instance('test-instance');
const table = instance.table('todo'); ①

const userId = 'user-84';
const itemId = 'item-24';
const notes = 'This was a few days later than expected';

const userHash = crc.calculate(userId).toString(16); ③
const itemHash = crc.calculate(itemId).toString(16);
const key = userHash + itemHash;

const entries = [ ④
  {
    key: key,
    data: {
      completed: {
        'item-id': itemId,
        'notes': notes
      }
    }
  }
];

table.insert(entries, (err, insertErrors) => {
  console.log('Added rows:', entries);
});
```

- ① As we constructed the `instance` from its ID, this time we do the same with the `table` that we've created, using the ID to create a `table` reference.
- ② This is the actual data we plan to store, put into variables so it's easy to read.
- ③ We determine the row key by hashing both values and then just concatenating them together. In this case, the row key is `c4ae6082` combined with `8900c74c`.

- ④ The list of entries has to be in a particular format, specifically with the row key in a field called key, and the data in a field called data.

After you run this code, you should see a confirmation that the data was added:

#### **Listing 7.6. Confirmation that data was added**

```
Added rows: [ { key: 'c4ae60828900c74c',
  data: { completed: [Object] },
  method: 'insert' } ]
```

Now that we have some data added let's look at how we'd retrieve this row, starting with a single key lookup. This works by constructing the row key as we did before, and simply looking up the row with that key.

#### **Listing 7.7. Retrieving data by key**

```
const crc = require('fast-crc32c');
const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

const instance = bigtable.instance('test-instance');
const table = instance.table('todo');

const userId = 'user-84';
const itemId = 'item-24';

const userHash = crc.calculate(userId).toString(16);           ①
const itemHash = crc.calculate(itemId).toString(16);
const key = userHash + itemHash;

const row = table.row(key);  ②
row.get().then((data) => { ③
  const row = data[0];
  console.log('Found row', row.id, row.data.completed); ④
});
```

- ① As before, we compute the row key using a hash of the user and item IDs.
- ② Just like we did with `instance` and `table`, we use the row key to create a row reference.
- ③ Finally, we use the `.get()` method on our `row` object to attempt to retrieve the row from Bigtable.
- ④ Obviously there is more data in the `row` object, but to make it easy to read we're printing just the `completed` column family to the console.

If you run this code (and you created the row previously), you should see output that looks something like the following. Note that the timestamps will be different.

#### **Listing 7.8. Viewing the data of a single row**

```
Found row c4ae60828900c74c { 'item-id':
  [ { value: 'item-24',
```

```

    labels: [],
    timestamp: '1479145189752000',
    size: 0 } ],
notes:
[ { value: 'This was a few days later than expected',
  labels: [],
  timestamp: '1479145189752000',
  size: 0 } ] }

```

Now that you understand how to retrieve a single row, let's look at a more powerful type of query that shows off the benefits of using this tall table. Let's try adding some more data and then iterating over the items completed by a particular user.

### **Listing 7.9. Inserting a bunch of rows**

```

const crc = require('fast-crc32c');
const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

const instance = bigtable.instance('test-instance');
const table = instance.table('todo');

const getRowEntry = (userId, itemId, notes) => { ❶
  const userHash = crc.calculate(userId).toString(16);
  const itemHash = crc.calculate(itemId).toString(16);
  const key = userHash + itemHash;
  return {
    key: key,
    data: {
      completed: {
        'item-id': itemId,
        'notes': notes
      }
    }
  }
};

const rows = [ ❷
  ['user-1', 'item-1', undefined],
  ['user-1', 'item-2', 'Late!'],
  ['user-1', 'item-3', undefined],
  ['user-1', 'item-5', undefined],
  ['user-2', 'item-2', 'Partially complete'],
  ['user-2', 'item-5', undefined],
  ['user-84', 'item-5', 'On time'],
  ['user-84', 'item-20', 'Done 2 days early!'],
  ['user-84', 'item-21', 'Done but needs review'],
];
const entries = rows.map((row) => {
  return getRowEntry.apply(null, row); ❸
});
table.insert(entries, console.log); ❹

```

- ➊ We'll use a helper function called `getRowEntry` to take a few pieces of information and return an object that is in the format that `table.insert` expects.
- ➋ To make the data easier to read, we'll write it as an array of rows, sort of like a CSV file in the format of `[userId, itemId, notes]`.
- ➌ Take the CSV-style data and get back properly formatted row entries.
- ➍ Add the data to Bigtable!

Running this snippet should give you a `null`, `[]` in your console (meaning, no errors to speak of) which is good. It means that you've added the entries and had no errors with any of the rows. Now let's figure out which items were completed by a particular user. To do this, we'll rely on the fact that we chose our row keys to be in the format of `crc(userId) + crc(itemId)` combined with the ability of Bigtable to very easily scan across rows with fixed start and end points. In other words, we'll start with any key "greater than" (or lexicographically "after") `crc(userId)`, and stop with the next key (`crc(userId) + 1`).

#### **Listing 7.10. Scanning rows for user-2**

```
const crc = require('fast-crc32c');
const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

const instance = bigtable.instance('test-instance');
const table = instance.table('todo');

const userId = 'user-2'; ①
const userHash = crc.calculate(userId).toString(16);

table.createReadStream({ ②
  start: userHash, ③
  end: (parseInt(userHash, 16)+1).toString(16) ④
}).on('data', (row) => {
  console.log('Found row', row.id, row.data.completed);
}).on('end', () => {
  console.log('End of results.');
});
```

- ➊ We're going to scan over all the rows pertaining to `user-2`.
- ➋ Here we use the `createReadStream` method which allows us to use Javascript's event emitter-style `.on()` handlers.
- ➌ As noted before, we "start" at the `userHash` key.
- ➍ Since we know `userHash` to be a string representing a hexadecimal number, we know that incrementing the number by 1 is where we should stop scanning.

When you run this short snippet, if you added the data listed previously, you should see the two items completed by `user-2`. which shows the item completed as well as any notes that were stored.

**Listing 7.11. Viewing the rows for user-2**

```

Found row 79c375855dc6587 {
  'item-id':
    [ { value: 'item-5',
        labels: [],
        timestamp: '1479145268897000',
        size: 0 } ],
  notes: []
}
Found row 79c37588116016c {
  'item-id':
    [ { value: 'item-2',
        labels: [],
        timestamp: '1479145268897000',
        size: 0 } ],
  notes:
    [ { value: 'Partially complete',
        labels: [],
        timestamp: '1479145268897000',
        size: 0 } ]
}
End of results.

```

Now that you understand how to read and write data to and from Bigtable, let's talk briefly about how to manage imports and exports.

#### **7.4.4 Importing and exporting data**

As with any storage system it's particularly important to have a back-up strategy for a variety of reasons. The obvious one is in the case of data corruption or physical drive failures, but this shouldn't be a concern with managed services on Google Cloud Platform. However, there are many other cases not related to this, one of the most common being invalid deployments that write corrupt or incorrect data, in other words, protecting yourself from yourself. To deal with these potential issues, Bigtable offers the ability to both export data and re-import data using Hadoop sequence files as the format.

Hadoop, as you may remember, is Apache's open-source version of Google's MapReduce, and is commonly used alongside HBase, Apache's open-source version of Google's Bigtable. Thanks to the similarity of these systems, Bigtable can rely on the Hadoop file format, which makes it easy for you to export and import data not just to Cloud Bigtable, but also to HBase if you happen to use that elsewhere.

**NOTE**

**Importing and exporting data in Bigtable is currently done by using Google Cloud Dataproc, a managed Hadoop service, however you do not need to know anything about Hadoop or Dataproc in order to import or export data.**

Unlike the other import and export operations we've gone through so far, Bigtable has a unique problem: it's a ton of data. Since Bigtable can (and often does) store Petabytes worth of data, asking a single machine to copy all of that data somewhere is not exactly going to be a "fast" process. Therefore, to import or export quickly we'll rely on the magic of distributed systems and actually turn on many machines under the

hood to make this happen.

Obviously managing machines is a bit of a distraction when all you want to do is export some data from Bigtable, but luckily there is a managed service to handle the hard work for you called Google Cloud Dataproc, and all you'll need to do is run a single command to do the work. Also, since we're dealing with potentially enormous amounts of data, it's probably best to put that data in Google Cloud Storage. So how does this all fit together? The general process of importing and exporting relies on Google Cloud Dataproc to do the actual work, looks something like this:

1. Download the import/export package from GitHub
2. Compile the package (using Maven)
3. Turn on a Dataproc cluster
4. Submit the import/export job to your cluster
5. Turn off the Dataproc cluster

Let's start by going through the preparation work we'll need for both imports and exports.

**NOTE**

If you don't have Java set up on your machine, you can always use the Google Cloud Shell, which is available in the Cloud Console in the top right hand, next to the search box, and comes with all the tools pre-installed and configured.

The first thing we need to do is download the import/export package from GitHub, and jump into the Dataproc example, shown below.

**Listing 7.12. Download the code for the import/export package from GitHub**

```
$ git clone https://github.com/GoogleCloudPlatform/cloud-bigtable-examples.git
$ cd cloud-bigtable-examples/java/dataproc-wordcount
```

Next we need to compile the package. To do this, we'll use Maven (`mvn`) which is a popular build manager for Java. (If you're using Ubuntu, you can install Maven by running `apt-get install maven`.) When compiling we'll need to pass in both the project ID and the instance ID that we'll be talking to. Note that the format for passing in data via the command line is to use `-D` with no space following it, which might look strange to non-Java developers.

**Listing 7.13. Build the import/export package**

```
$ mvn clean package -Dbigtable.projectID=your-project-id \ ❶
-Dbigtable.instanceID=your-bigtable-instance-id
```

❶ Make sure to substitute in your project ID and Bigtable instance ID when running this command.

After the build command finishes, you'll be left with a Jar file located in the `target/` directory which is what will do the heavy lifting to import and export data. Let's look first at how we'll export the data that we added to our `todo` table.

The first thing we need to do is decide where to put this data. The easiest, and recommended, choice is to use a Google Cloud Storage bucket, so let's create one. And, since our Bigtable cluster is in the `us-central1-c` zone, let's make sure our bucket lives in the same region. You can do this with the `gsutil` command:

#### **Listing 7.14. Create a new bucket in the same location as our Bigtable instance**

```
$ gsutil mb -l us-central1 gs://my-export-bucket
Creating gs://my-export-bucket/...
```

Now we can create a Dataproc cluster in the same zone as our Bigtable instance, and deploy our "export" operation to the cluster. This is no more than running two commands:

#### **Listing 7.15. Create a Dataproc cluster and submit an export job to it**

```
$ gcloud dataproc clusters create my-export-cluster --zone us-central1-c \
--single-node ①

$ gcloud dataproc jobs submit hadoop --cluster my-export-cluster \
--jar target/wordcount-mapreduce-0-SNAPSHOT-jar-with-dependencies.jar \
-- \
②
export-table todo gs://my-export-bucket/todo-export-2016-11-01
```

- ① If you're just testing, it might save some money to use a single-node Dataproc cluster. If you are actually exporting a lot of data, leave this flag off.
- ② Make sure that the `--` is separated from the `export-table`. The double dashes by themselves tell `gcloud` to forward these flags to the Java code.

After running these two commands (which might take a little while, don't worry), your data should be available as Hadoop sequence files in the bucket created before. You can verify this by "listing" the contents of the bucket using `gsutil`:

#### **Listing 7.16. List the contents of our bucket to see exported data**

```
$ gsutil ls gs://my-export-bucket/todo-export-2016-11-01/
gs://my-export-bucket/todo-export-2016-11-01/
gs://my-export-bucket/todo-export-2016-11-01/_SUCCESS
gs://my-export-bucket/todo-export-2016-11-01/part-m-00000
```

Now let's look at how we might re-import the same sequence files into a table. To do this, we can use the exact same Dataproc cluster and Jar that we built, but we'll be make a few slight tweaks to the parameters. We also need to make sure there is a table ready to accept the imported data, which we can do quickly using code as we learned before.

#### **Listing 7.17. Use Node.js to create the table to hold imported data**

```
const bigtable = require('@google-cloud/bigtable')({
```

```

    projectId: 'your-project-id'
});

const instance = bigtable.instance('test-instance');
instance.createTable('todo-imported', {
  ①
    families: ['completed']
  ②
});

```

- ① In this example, we're calling the new table `todo-imported`.
- ② Note that we must specify the same column families again. Skipping this will lead to errors during the import process.

Once we have the table set up, all we need to do is submit a job to Dataproc to load the data from our bucket and import it into our newly created table.

#### **Listing 7.18. Submit the import job to Cloud Dataproc**

```
$ gcloud dataproc jobs submit hadoop --cluster my-export-cluster \
--class com.google.cloud.bigtable.mapreduce.Driver \
--jar target/wordcount-mapreduce-0-SNAPSHOT-jar-with-dependencies.jar -- \
import-table todo-imported gs://my-export-bucket/todo-export-2016-11-01
```

Note that we've changed `export-table` to `import-table`, and the table name changed from `todo` to `todo-imported`. Also, while the value for the data location is the same, this time that data is being used as source data rather than as a destination of exported data.

And that's it! At this point you should have a pretty strong understanding of both the theory underlying Bigtable as well as the operational aspects of actually using it. Now let's take a moment to look at how much all of these things will cost.

## **7.5 Understanding pricing**

Similar to Cloud SQL (see [chapter 4](#)), Cloud Bigtable splits pricing into a couple of different areas: compute costs (hourly rate for running nodes), storage costs (monthly rate for GB stored), and network costs (per GB rate for data sent outside the same region). Additionally, these costs vary depending on the location in which Bigtable is running. This makes the pricing model pretty straight forward, however there are a few things worth mentioning.

First, as we learned before, the minimum size of an instance is 3 nodes. This means that the minimum hourly rate for any production instance is technically three times the per-node hourly rate. Next, storage can be either on solid-state drives (SSDs) or standard hard disks (HDDs), and each of these have different prices. In other words, your choice of how to store data will have an effect on your monthly per-GB cost. Finally, networking costs are only charged for outbound (egress) traffic, and even then only when the traffic is leaving the region where the instance lives. This means that if you send data only from a Bigtable instance to a Compute Engine instance in the same zone (or even different zones in the same region) that traffic is entirely free. And to make things a bit easier for US-based instances, if you happen to send traffic between

to different regions both inside the US, traffic is billed at a discounted rate of \$0.01 per GB sent. To sum this all up, the following table shows an overview of the costs broken down by the different locations where you can run Bigtable instances.

**Table 7.8. Bigtable pricing for some locations**

Location	Compute (per Node-hour)	HDD (per GB-month)	SSD (per GB-month)
Iowa (US)	\$0.65 (\$1.95 minimum)	\$0.026	\$0.17
Singapore	\$0.72 (\$2.16 minimum)	\$0.029	\$0.19
Taiwan	\$0.65 (\$1.95 minimum)	\$0.026	\$0.17

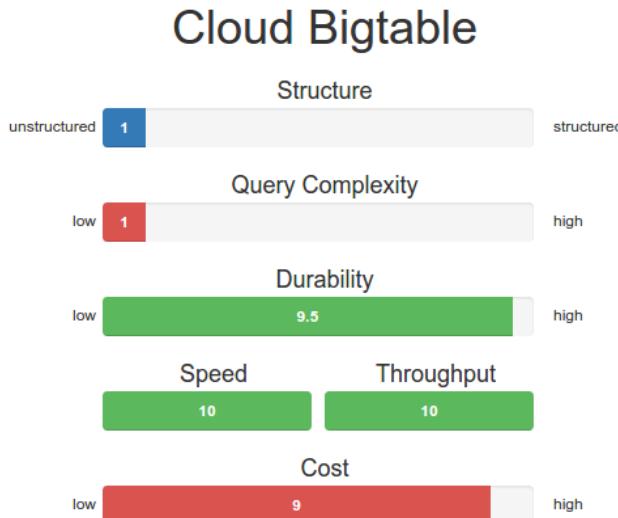
To make this a bit more concrete, let's take a look at an example Bigtable instance running in Iowa starting with 3 nodes and about 100 GB of data on SSDs, and after that we'll look at growing that to 10 nodes with 10 TB of data. To start, our 3 nodes in Iowa will cost \$0.65 per Node-hour (\$1.95 per hour), meaning the total monthly cost is about \$1,400 per month for the nodes ( $\$0.065 * 3 \text{ nodes} * 24 \text{ hours per day} * 30 \text{ days per month}$ ). On top of that we have 100 GB of data stored on SSDs, adding an additional \$17 ( $\$0.17 * 100 \text{ GB per month}$ ). In this case, the storage cost is basically rounding error on top of the compute cost, so we can round this off to around \$1,400 per month for this cluster.

If we were to expand this to 10 nodes and 10 TB of data, our numbers jump up a bit as you'd expect. The compute cost is now \$4,680 per month ( $\$0.65 * 10 \text{ nodes} * 24 \text{ hours per day} * 30 \text{ days per month}$ ), and the storage cost jumps to \$1,700 per month ( $\$0.17 * 10,000 \text{ GB per month}$ ). This brings our grand total for this (pretty large) Bigtable instance to about \$6,400 per month. Note that we're assuming all traffic is staying inside the same region (e.g., we're interacting with Bigtable using Compute Engine instances nearby), so there's no egress network cost for serving data around the world.

At this point you should have a good grasp about how much Bigtable costs, but you may still be wondering "Why would I use Bigtable over something else?" or more specifically "When is it a good fit for my project?" Let's spend a bit more time going through the benefits and drawbacks of Bigtable which may help inform your decision about whether to use it in your project.

## 7.6 When should I use Cloud Bigtable?

To get an overview, let's look at the scorecard for Bigtable and go through the attributes point by point.

**Figure 7.15. Scorecard for Cloud Bigtable**

### 7.6.1 Structure

As you learned throughout the chapter, Bigtable is very loosely structured when compared to the other storage systems we've seen such as Cloud Datastore or Cloud SQL. While it does require specific column family names, the column qualifiers can be dynamic and created on the fly, meaning the column qualifiers can themselves store data.

In many ways, the "structured" aspect of Bigtable applies more to the concepts than it does to the data. Inside that conceptual framework the column qualifiers and the values can be basically anything you want them to be. This freedom, however, means that you lose out on many of the more advanced features that you might be used to in other storage systems, which we'll look at next.

### 7.6.2 Query complexity

If a strict key-value storage system (e.g., Memcache) was an example of a system that offered the minimal query complexity possible, Bigtable should be considered just a hair above that. As you saw earlier, Bigtable can mimic the key-value querying by constructing a row key and asking for the data with that row key, but it allows you to do something critical that services like Memcache don't: scan the key space.

In most key-value systems, you can request a given key but have no way of asking for "all keys matching a specific prefix" (or even "all keys"). In Bigtable you're able to specify a range of keys to return, making it important to choose row keys that serve this purpose. In some ways, this is a bit like being able to choose one *and only one* index for your data. This means that many things you're used to with relational databases are simply **not** possible, such as:

- Querying based on data inside a row (`SELECT * FROM employees WHERE name = 'Jimmy' AND age > 20`)
- Computing new values based on data (`SELECT AVERAGE(age) FROM employees`)
- Joining sets of data together in a query (`SELECT * FROM employees, employers WHERE employees.employer_id = employer.id`)

### 7.6.3 Durability

Since all Bigtable data is stored on persistent disk the chances of losing any data stored is extraordinarily low. However, like any storage system, in addition to worrying about the underlying storage system (in other words, the physical disks) you have to consider the software system's persistence model.

In Bigtable's case, the system is built to shard data across multiple machines (and multiple *tablets*) so that the load is spread evenly across the system. Also, Bigtable's row-level atomicity means that when writing a row, the write will either persist or fail. This means that when using Bigtable, "losing data" is really not something to worry about.

### 7.6.4 Speed (latency)

One of the main reasons to use Bigtable is its performance. The whole reason you're not able to run fancy complex queries or operate atomically on more than a single row means that things like reading a single row will be incredibly fast (typically below 10 ms, even with thousands of writes per second). While some in-memory storage systems are capable of this, few of them can maintain this level of speed without sacrificing durability or concurrency (e.g., throughput). That said, the system is able to keep this latency low because it automatically moves your data around, so choosing a row key is very important and may have adverse effects on performance if done poorly.

### 7.6.5 Throughput

As we hinted to previously, throughput on Bigtable is best-in-class for storage systems. The same aspects of data redistribution that help to keep latency low also help keep throughput high. By using SSD disks, random reads and writes are extremely fast, and allow many of these reads and writes to happen concurrently. By combining the high performance of the low-level storage with the even load balancing across tablets, Bigtable clusters as a whole can handle extraordinarily large levels of throughput, with measurements starting in the tens of thousands of requests per second.

Further, adding more capacity to the cluster is as simple as adding more nodes. Since Bigtable will shift data around to nodes that are under-utilized, adding more nodes is the same as having empty nodes with no traffic to them. As you'd expect, Bigtable will notice these empty and idle nodes, shift tablets to them based on the traffic to those tablets, and at the end you have a larger cluster with traffic evenly balanced across each node, improving your overall throughput.

### 7.6.6 Cost

As we've discussed throughout the chapter Bigtable's primary benefit above all else is its performance. Unlike some of the other storage systems we've discussed so far, Cloud Bigtable has no free tier, and has a minimum cluster size of 3 nodes which translates to about \$1,400 USD per month as a minimum. This is quite a change from the \$30 USD per month minimum for Cloud SQL.

In short, this high initial and on-going cost for Cloud Bigtable means that you should use it when you absolutely need it due to the scale you expect to see. If you can make due with something else (e.g., MySQL) it's probably going to be a better fit.

### 7.6.7 Overall

As you might notice, most of the value from Bigtable comes from performance with both speed and throughput topping the charts. Aside from the performance, Bigtable acts much like any other key-value store, with almost no structure (you have a row key which points to mostly unstructured data) and very little supported query complexity (you ask for a row key, or sequence of row keys and get back subsets of data). If you find yourself still wondering why you'd want to use Cloud Bigtable, don't worry because you're not alone. Bigtable is incredibly powerful but the lack of common features (e.g., secondary indexes) tends to be a big drawback for most projects. So why might we want to use Bigtable?

First and foremost, Bigtable should always be on the list of options whenever you have a very large data set. In this case, "very large" typically means terabytes or more. If you only have data that counts up to the gigabyte range (which is typical for a database storing user information), you're probably better off with something else.

Second, Bigtable is great for usage sustained over a long period of time. In this case, "a long period of time" is measured in hours or days rather than seconds or minutes. If you only use Bigtable to store and query data infrequently, you're probably better off with some other analytical storage system.

Third, Bigtable is likely to be a good fit if you need extraordinarily high levels of throughput. In this case "extraordinarily large" means tens to hundreds of thousands of queries every second. If you only need a few queries per second, you have many options and may want to start with another system.

Finally, if you need very basic access to your data in the form of lookups and simple scans across keys, then Bigtable may be a good fit. If you need more than this (like secondary indexes), you're probably better off using a relational database. To make this more concrete, let's look briefly at our example applications and see whether Cloud Bigtable might be a good fit.

#### **To-do list**

As we mentioned already, the to-do list application, which stores history of items to complete, along with when someone finished the items definitely won't need the levels of performance offered by Bigtable and is primarily application-focused data rather

than analytical data. This means that even though we used it as our example, it really is **not** a very good fit for Cloud Bigtable.

**Table 7.9. To-do list application storage needs**

Aspect	Needs	Good fit?
Structure	Structure is fine, not necessary though	Sure
Query complexity	We don't have that many fancy queries	Not really
Durability	High, we don't want to lose stuff	Definitely
Speed	Not a lot	Overkill
Throughput	Not a lot	Overkill

In short, Cloud Bigtable is acceptable on a few of the storage needs, not a great fit when it comes to the queries we'd want to run, and completely overkill for our performance requirements. This means that while you certainly could use Bigtable to store to-do list data, it is going to be way more expensive than you need, and you'll likely be very frustrated as your to-do list application grows in complexity far more than it does in scale, and you realize that you need to run more advanced queries over a relatively small amount of data.

#### **E\*Exchange**

As we saw before, E\*Exchange, the online trading platform that allows people to trade stocks and bonds online, will require far more complicated queries for customer data, which is one aspect that Bigtable is particularly bad at.

**Table 7.10. E\*Exchange storage needs**

Aspect	Needs	Good fit?
Structure	Yes, reject anything suspect, no mistakes	Not really
Query complexity	Complex, we have fancy questions to answer	Definitely not
Durability	High, we <b>cannot</b> lose stuff	Definitely
Speed	Things should be pretty fast	Probably overkill
Throughput	High, we may have lots of people using this	Probably overkill

Looking at this table, while Bigtable happens to fit the durability requirements, the performance requirements are yet again overkill. Additionally, the query complexity needed by an online trading platform are very difficult to handle with a storage system like Bigtable. Finally, the need for data validation and structure at the storage layer is really not what Bigtable is designed for, so these features simply aren't available. This means that Bigtable is really not a great fit for the trading platform's business level data. But what about the stock trading data?

We didn't discuss this before, but what if E\*Exchange wanted to store historical stock trading data? This data will have lots of small events, including the stock symbol, the time, the trade amount, and the price paid. And there are millions (or more) of these every day, even if only counting larger orders that are filled. Would this aspect of

E\*Exchange be a good fit for Cloud Bigtable?

**Table 7.11. E\*Exchange stock trading storage needs**

Aspect	Needs	Good fit?
Structure	Not really	Definitely
Query complexity	Simple lookups and scans	Definitely
Durability	Medium, a few items can be lost	Definitely
Speed	Things should be pretty fast	Probably overkill
Throughput	High, we have tons of traffic	Definitely

Looking at this table it seems like the stock trading data might be an excellent fit for Cloud Bigtable, even though single row latency might be overkill.

### **InstaSnap**

InstaSnap, the very popular social media application that let's people post images, follow, and like others images, has a few requirements that seem to fit really well and only a couple that are a bit off.

**Table 7.12. InstaSnap storage needs**

Aspect	Needs	Good fit?
Structure	Not really, structure is pretty flexible	Definitely
Query complexity	Mostly look-ups, no highly complex questions	Definitely
Durability	Medium, losing things is inconvenient	Sure
Speed	Queries must be very fast	Definitely
Throughput	Very high, Kim Kardashian uses this	Definitely

As we saw when evaluating InstaSnap before, the biggest issue is the single query latency, which needs to be extremely fast and Bigtable happens to excel at. The performance requirements are certainly met by Bigtable, and the fact that most of the queries are simple lookups or scans means that Bigtable's query complexity limitations should not be a cause for concern. In short, while InstaSnap could potentially run using something providing more complex queries (such as Cloud Datastore), as the service grows larger and larger, something like Cloud Bigtable is likely to be the better overall fit.

## **7.7 What's the difference between Bigtable and HBase?**

If you're familiar with HBase, there are a few things you should know about how Cloud Bigtable is different. First, a few of the advanced features aren't available with Bigtable. An example here is co-processors where HBase allows you to deploy some Java code to be run on the server with your HBase instance. There are many reasons for this, but an obvious simple one is that Bigtable is written in C, so it'd be tricky to connect HBase co-processors (written in Java) to the Bigtable service (written in C).

Second, due to an underlying design difference between Bigtable and HBase, Bigtable

(currently) is able to scale more easily to a larger number of nodes, and as a result is able to handle more overall throughput for a given instance. The reason for this is that HBase's design requires a "master node" to handle fail-overs and other administrative operations, which means that as you add more and more nodes (in the thousands) to handle more and more requests, the master node will become a performance bottleneck. Cloud Bigtable, while similar to HBase in many respects, does not have this same design limitation and will scale to arbitrarily large cluster sizes without introducing this same performance bottleneck.

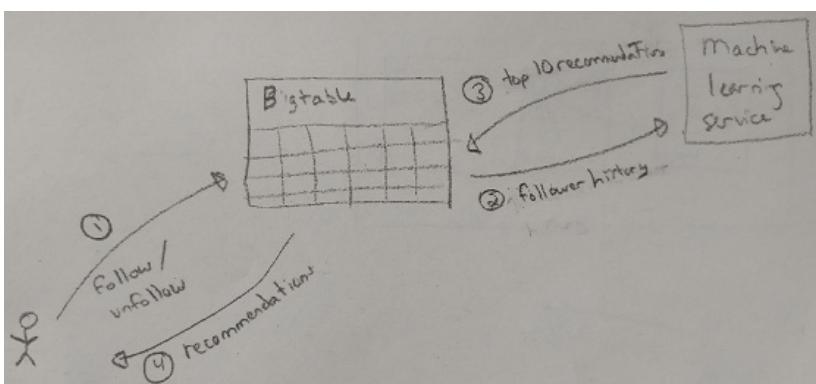
Lastly, there are the typical "cloud-like" benefits, in particular the automatic upgrade of binaries (that is, you don't have to upgrade Bigtable like you do on HBase nodes), as well as easy and stable resizing of your cluster (that is, you can change how much serving capacity you have with zero downtime), and the obvious "pay for what you use" principle applies to data storage.

## 7.8 Case study: InstaSnap recommendations

As you may recall, InstaSnap was our sample application that allowed users to post images and share them with their followers, and had some potentially large scaling requirements (after all, some celebrities might use InstaSnap). To demonstrate one way InstaSnap could use Bigtable, let's imagine that we want to build a recommendation system for InstaSnap.

Unlike some other examples, the code for querying a table for data is not all that complicated, however choosing the right table schema can be pretty complicated. As a result, rather than zooming in on code samples, let's focus on designing a system that can make recommendations and the tables that will store this data. To kick things off, let's look at the various components and how they talk to each other.

**Figure 7.16. Overview of InstaSnap's recommendation pipeline**



First, we need to store an overview for each user which tracks who they follow as well as who follows them. Based on these signals, we could construct some sort of machine learning service that would use that information to notice various overlapping patterns, and ultimately come up with a list of recommendations. To make this more concrete,

imagine that you just followed George Clooney on InstaSnap. The machine learning service could notice that the majority of people who follow George Clooney also happen to follow Leonardo DiCaprio. Based on this information, it seems likely that Leo might be good a suggestion.

To keep things relatively simple and focused on Bigtable, we're going to assume that this machine-learning service is somewhat magical, and uses the "A follows B" data to come up with recommendations. Given that, let's look at how we might design a schema so that InstaSnap can store all the necessary data in Bigtable.

### **7.8.1 Querying needs**

Before we start, we need to figure out how we'll want to query our data. This machine learning service will have quite a few different "questions" it needs to ask in order to get the data it needs about a given user, such as:

1. Who does user X follow?
2. Who follows user X?

In short, it seems like we need to be able to provide lists of followers, both of a particular user and by a particular user. If we provide the machine learning service with these answers on-demand, it can probably use that information to come up with some recommendations. Additionally, the recommendation results need to be stored back in Bigtable, and InstaSnap will need to ask, "Who's recommended based on just following user X?" Now that we have an idea of the questions we'd want to ask, let's look at some possible schemas and decide which fits best.

### **7.8.2 Tables**

Based on the questions we need answered, it looks like there should be a total of 3 tables:

1. User's followers and followees (`users`)
  - Who does user X follow?
  - Who follows user X?
2. Recommendation results (`recommendations`)
  - If I just followed user X, who else is recommended?

Let's start by looking at the `users` table which will let us figure out who a user follows as well as who follows that user.

### **7.8.2 Users table**

When it comes to storing "followers", we could use either a "tall" table or a "wide" table. Let's take a look at the differences, starting with a tall table. As you learned earlier, a tall table has lots of rows to represent data and accomplishes this by adding information to the row key. Then, to get lists of related information that spans many rows you use a prefix scan over the rows. The table below shows how we might store

some rows representing one user following another user.

**Table 7.13. Followers represented as a tall table**

<b>Row key</b>	<b>Follows (column family)</b>
	<b>Username</b>
14ccc4ac79c3758	user-2
79c3758f5f7b45b	user-3
f5f7b45b14ccc4ac	user-1
f5f7b45b79c3758	user-2

Recall that we generate the row key by hashing both the follower and the followee, and concatenating the results. For example, user-1 following user-2 would have a row key of  $\text{crc32c}(\text{user-1}) + \text{crc32c}(\text{user-2})$  which turns out to be 14ccc4ac79c3758. As expected, this table structure makes it very easy to ask the question "Does user-1 follower user-2?" All you have to do is compute the hashes and retrieve the row. If the row exists, then the answer is "yes".

It's also easy to request all the people that a user follows using a prefix scan. All you have to do is compute the hash of the user you're interested in, and use that value as the prefix. For example, finding the users that user-1 follows would be a prefix scan of  $\text{crc32c}(\text{user-1})$  which comes out to 14ccc4ac. Finally, it's pretty easy to add and remove followers as all you have to do is add and remove rows corresponding to the mappings.

But what about finding all the followers of a given user? How do we this? It turns out that with this type of tall table, finding everyone following a given user can't be done with a simple table scan. You can do a prefix scan which asks "who does the prefix follow?" however there is no way to do a "suffix scan" which asks "who follows the suffix?" If you think about it, you'll also realize that even if a suffix scan existed, the row keys are in lexicographical order, so the idea of scanning based on a suffix runs against what Bigtable was designed to do. Since we are stuck with this so far, let's check a few other options to be able to answer this question.

One option that would work with a tall table is to store two different rows for the bi-directional relationship. In other words, we store one row saying "A follows B" and another row saying "B is followed by A", using a special token in between the  $\text{crc32c}$  hashes of A and B to denote "follows" or "is followed by". This might look a bit like Table 7.14 .

**Table 7.14. Followers represented as a tall table**

<b>Row key</b>	<b>Follows (column family)</b>
	<b>Username</b>
14ccc4ac > 79c3758	user-2
14ccc4ac < f5f7b45b	user-3
79c3758 > f5f7b45b	user-3

79c3758 < 14ccc4ac	user-1
79c3758 < f5f7b45b	user-3
f5f7b45b > 14ccc4ac	user-1
f5f7b45b > 79c3758	user-2
f5f7b45b < 79c3758	user-2

In this table you can see that we've constructed a strange-looking, but completely valid, row key that stores rows for both "A follows B" ( $\text{crc32c}(a) > \text{crc32c}(b)$ ), and "B is followed by A" ( $\text{crc32c}(b) < \text{crc32c}(a)$ ). This means that if we want to ask "who does A follow?" we do a prefix scan on  $\text{crc32c}(a) >$ , and if we want to ask "who follows A?" we do a slightly different prefix scan on  $\text{crc32c}(a) <$ . The value stored in the row is always the *unknown* side of the query, which in this case is the user on the right side of the arrow. This is because while we know the value that we hashed to run the prefix scan, we can't go backwards from a hash to the actual user.

While this table schema will certainly work, it really isn't very space efficient as it's technically storing twice the number rows to convey the same information. In other words, the row  $\text{crc32c}(a) > \text{crc32c}(b)$  (A follows B) conveys the same information as  $\text{crc32c}(b) < \text{crc32c}(a)$  (B is followed by A), just in two different ways. Since none of the tall table schemas look like a perfect fit, let's look at a the wide-format table to see if this comes out any better.

In this case, a wide table might store a row key for each user and then a column family to store other users being followed. Inside that column family, each user being followed gets its own column with a placeholder value. This might look a bit like Table 7.15.

**Table 7.15. Followers represented as a wide table**

Row key	Follows (column family)		
	user-1	user-2	user-3
user-1		1	
user-2			1
user-3	1	1	

This table structure makes it very easy to ask "Who does A follow?" by simply asking for the row for the user and the "Follows" column family. All the keys in the returned map will be the people that A follows. Likewise, it's easy to ask "Does A follow B?" as you'd simply ask for the row for the user, and a specific column inside the "Follows" column family, as it will have the flag value set for the target user (in this case, B).

But what about finding everyone followed by a single user? ("Which users follow A?") It looks like this schema is going to run into the exact same problems as before where going one direction ("Who does A follow?") is fast and easy, and the other direction ("Who follows A?") is very tricky. So let's see if we can tweak this schema to handle both directions. To do this, we could add a second column family which represents the

inverse relationship ("B is followed by A") and store followers in that map as well. Then you'd simply ask for that column family to answer the other side of the question ("Who follows A?"). This would make our new schema look a bit like Table 7.16.

**Table 7.16. Bi-directional followers represented as a wide table**

Row key	Follows (column family)			Followed by (column family)		
	user-1	user-2	user-3	user-1	user-2	user-3
user-1		1				1
user-2			1	1		1
user-3	1	1			1	

The "Follows" column family (the left side of the table) helps answer the question "Who does A follow?" by storing a sparse map with flag values set. For example, "Who does user-1 follow?" would return {"user-2": 1}. The "Followed by" column family (the right side of the table) answers the question "Who follows A?" by storing the same style of sparse map. For example, "Who follows user-2?" would return {"user-1": 1, "user-3": 1}.

So what are the downsides of this schema? It turns out that if we use this wide table, we'll need to update two rows for every "follow" and "unfollow" action. For example, if user-3 wants to "unfollow" user-2, we need to do two actions:

1. Update row user-3 and delete the column user-2 from the "follows" column family.
2. Update row user-2 and delete the column user-3 from the "followed by" column family.

As you might recall, this presents a bit of an issue due to the fact that Bigtable doesn't support the ability to change multiple rows in a single transaction. But how big of a problem is this?

The failure condition of only one of the two actions happening (but not both), would be a bit strange, but not critical. If we ended up in this "bad state", it would mean that depending on the question we ask, we might get different results. For example, this would mean that "Does A follow B?" might say "Yep!", but asking "Is A followed by B?" might say "Nope!" One easy fix for this is always ask this question the same way (e.g., always ask "Does A follow B?" and never ask "Is B followed by A?").

Our next problem would concern listing followers in both directions. If we have a failure and end up in this "bad state", then looking at the list of people that A follows might show B in that list, but looking at the list of people followed by B might not show A in that list. In the grand scheme of things, this seems like it'd be a tough consistency issue to spot, so much so that you'd have to go specifically looking for this problem, and even then it'd be tough for a human to notice. Given that, this really doesn't seem like a big deal.

Overall, it seems like the wide table is probably going to be easier to manage, so let's

next look at how data recommendation data might be stored in Bigtable.

### 7.8.3 Recommendations table

The recommendations table is the one that brings everything together. In short, it's the table that stores the output of our machine learning job, so that we can come up with a set of recommendations when someone on InstaSnap follows someone new and it turns out to be pretty simple.

At the time when we're presenting some recommendations of who else to follow, we've just had a "follow" event, meaning our question would be phrased as "Given I've just followed User X, who else should I follow?" This means that our queries are user-based, which makes for an easy row key (the same as we had with our Users table)!

The column family would simply be called "recommendations", with a column for each user that is recommended, with a score set as the value rather than a simple flag. An example of how this might look is shown in Table 7.17.

**Table 7.17. Recommendations table example**

Row key	Recommendations (column family)		
	user-1	user-2	user-3
user-1		0.5	
user-2			0.4
user-3	0.4	0.6	

Using this table design, you might ask "I just followed user-1. Who else should I follow?" This translates to asking for the user-1 row of the recommendations table. The results would come to be `{"user-2": 0.5}`, and the InstaSnap application would show that as a suggested recommendation, and the application could sort through the list of users by their values, prioritizing the more highly recommended users over others. Further, to keep this table clean, the machine-learning job would basically overwrite stale data every time the recommendation job runs.

### 7.8.4 Processing data

Since it's unlikely that running a deep learning algorithm isn't exactly a "quick" operation, we should probably design our system so that the learning happens periodically and then requests for suggestions would be pulled from cached results of the previous run. Luckily we can use Bigtable as the middle-man in this process. At a high level, referring to Figure 7.16, getting recommendations would fall in two different steps:

1. Every so often, the machine learning job will kick off and come up with a set of "follow recommendations."
2. Whenever a user follows someone new, show them a set of recommendations related to that action ("You might also be interested in ...").

As you saw in the diagram above, we can use Bigtable as an intermediary where it reads follower data from Bigtable as designed above, computes recommendations, and then stores the results of that computation back in Bigtable. Then, when we need to show recommendations to a user, it's a simple read from those results in Bigtable. Let's look at each step and see what the code might look like when interacting with Bigtable as an intermediary.

First, the machine learning job will need to retrieve lists of followers. This can be on a single row basis (e.g. `getFollowers('user-1')`) or as a full table scan if the job is re-processing the recommendations. Let's start with a simple way of grabbing the followers for a given user.

#### **Listing 7.19. Getting followers of a single user**

```
const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

const instance = bigtable.instance('test-instance');
const table = instance.table('users'); ①

const getFollowers = (userId) => {
  const row = table.row(userId); ②
  return row.get(['followed-by']).then((data) => {
    return Object.keys(row.data); ③
  });
}
```

- ① Start by pointing to the users table.
- ② The row is nothing more than the user ID, so we can jump right to it.
- ③ We ask specifically for the column family storing the followers, and return the keys (remember the values are just placeholders).

Next we need to provide a way to scan through the table. Since we're doing a **full** table scan, we really should partition the search space so that multiple VMs can all pull data out of Bigtable. We can use the `sampleRowKeys()` method to give us the "borders" of tablets to help us decide where to split the data.

#### **Listing 7.20. Finding the split points and returning them as key range filters**

```
const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

const instance = bigtable.instance('test-instance');
const table = instance.table('users');

const getKeyRanges = () => {
  return table.sampleRowKeys().then((data) => { ①
    const ranges = [];
    const currentRange = {start: null, end: null};
```

```

    for (let splitPoint in data[0]) {
      currentRange.start = currentRange.end;
      currentRange.end = splitPoint.key;           ②
      ranges.push(currentRange);                 ③
    }
    return ranges;
}
}

```

- ➊ First, use the `sampleRowKeys` method to find the split points (the "borders") that you can use to split up how you consume the rows in the table.
- ➋ Take the end of the previous range and make it the start of the next, and make the new split point the end of the current range.
- ➌ Add the range to the list of results.

As you can see, this method will ask Bigtable for the split points (or the "borders") to use when splitting up the work of asking for all of the data in the table, and return it as a list of ranges. After this, it's just a matter of using the `createReadStream` method to scan between those ranges.

#### **Listing 7.21. Scanning the table in chunks**

```

const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

const instance = bigtable.instance('test-instance');
const table = instance.table('users');

getKeyRanges().then((ranges) => {           ①
  for (let range in ranges) {
    runOnWorkerMachine(() => {               ②
      table.createReadStream({                ③
        start: range.start, end: range.end
      }).on('data', (row) => {
        addRowToMachineLearningModel(row);   ④
      });
    });
  }
});

```

- ➊ Start by fetching the split points for the table.
- ➋ Here we use the fictitious `runOnWorkerMachine` which would take the method provided and forward it to a separate worker (perhaps by broadcasting a message to perform the work).
- ➌ When creating the read stream, use the start and end keys of the range as provided from the `getKeyRanges` method.
- ➍ Finally, we perform some magic that adds the new row to the model to be used when making recommendations with machine learning.

In this case, despite the need to use a couple of "fake" methods (`runOnWorkerMachine` and `addRowToMachineLearningModel`), you can see how we

would scan through the table using multiple consumers of data.

## 7.9 Summary

- Bigtable is a large-scale data storage system, originally built for Google's web search index.
- It was designed to handle large amounts of replicated rapidly changing data, and can be queried very quickly (low latency) with high concurrency (high throughput), while maintaining strong consistency throughout.
- Cloud Bigtable is a fully-managed version of Google's Bigtable, exposing almost all of the features available in Google's original version.
- Bigtable is likely a good fit if you have a large amount of data and primarily access it using key lookups or key scans, but not a great fit if you need secondary indexes or relational queries.



# *Cloud Storage: Object storage*

**This chapter covers:**

- What is object storage?
- What is Cloud Storage?
- Interacting with Cloud Storage
- Access control and lifecycle configuration
- Deciding whether Cloud Storage is a good fit

## **8.1 What is object storage?**

If you've ever built an application that involves storing an image (such as a user's profile photo), you've run into the problem of deciding where to put that photo. Chances are that to keep making progress on your project, you went with the easiest place: right in your database or on your local file system. This will work for a little while, but if your website becomes very popular, the disk that holds all of these images and videos might get overwhelmed. This is the exact problem that object storage services aim to solve.

In addition to storing data correctly, a primary design goal of these systems is to abstract away the complexity of the underlying disks and data centers, and instead provide a simple API for uploading and retrieving files, a bit like key-value storage for large values with automatic replication and caching around the world.

Of all the cloud services that exist today, object storage tends to be one of the most common and most standardized. For example, Google Cloud Storage and Amazon S3 have the same concepts and are actually capable of speaking the same XML API.

While there are many similarities across object storage systems, they each tend to have slight differences either in the pricing model, replication strategy, or storage class.

Google Cloud Storage is the default object storage system on GCP, so let's look at the key concepts that you need to understand to store your data.

## 8.2 Concepts

### 8.2.1 Buckets and objects

Cloud Storage, like many other object storage systems (such as Amazon's S3) uses two key concepts: buckets and objects. You can think of a **bucket** as a "container" that will store your data. The bucket has a globally unique name, rather than just unique to your project, as well as a few other options you can set such as the geographical location and the "storage class" (both discussed later on). In many ways, you can think of buckets as "disks" in the sense that you can choose what type of disk you want (e.g., SSD, regular disk, replicated disk across the US, etc) and where you want that disk to live (e.g., Europe or the US).

The big difference is that this "disk" is extraordinarily large in that there's no limit to how many bytes can end up in a bucket; the only limit is that each file in the bucket can be up to 5 Terabytes. Additionally, this "disk" doesn't have the same failure semantics as a typical physical disk. This is because the bucket itself is replicated and spread across many physical disks in order to maintain high levels of durability and availability.

**Objects** are the "files" that you put inside a bucket which have a unique name inside the bucket, and like you may be used to on typical file systems, slashes (/) are treated specially so that you can browse "directories" like you do on any traditional Linux system. There are some other advanced features (e.g., storing the "generation" of an object) which we'll discuss later, but objects themselves are pretty straight forward: named chunks of bytes that you can retrieve on demand.

### 8.2.2 Locations

Just like VMs that you turned on in Compute Engine, buckets can have locations as well, however rather than always choosing a specific zone (e.g., us-central1-a), buckets exist either at the regional level (e.g., us-central1), or even spread across multiple regions (e.g., "United States" or "Asia"). This is because VMs can only exist in a single place but data can be copied and live in multiple places simultaneously. So why might you choose these different locations for your data? It turns out that it depends on what you need.

If you need your data to be always available, even if lightning strikes all of the data centers in the us-central1 region, you probably want to create a multi-regional bucket (e.g., set the location to "United States"). A multi-regional bucket is by definition replicated across several regions, which means that even a complete outage of all data centers in a single region can't stop your data from being available.

If you're concerned about latency between your VMs and your data on GCS you might want to choose a specific region (e.g., `us-east1`) for your data. The reason for this is that if you make a request from your VM in `us-east1-a` to a bucket located in "United States", that request could end up going to either `us-east1` or `us-central1`, which means that the data may end up taking the long way to you. If you're unsure where you'll put your VMs (or if you'll even have any VMs accessing your data at all) you might want a multi-regional bucket which will make sure that data is always closest to where you or your customers are.

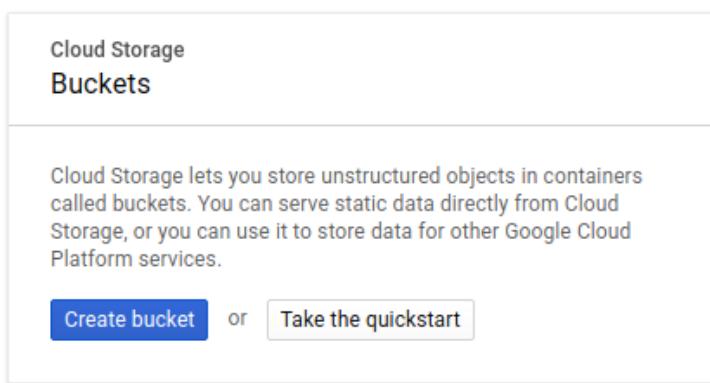
Finally, as you'll learn later on in the section on pricing, if you make a mistake and put a bucket far away from your VMs, you'll end up paying a premium for reading your data due to cross-region network transfer fees. This can range from being very obvious (e.g., a bucket in "Asia" and your VMs in `us-central1-a`) to the much more subtle (e.g., a bucket in `us-central1` and your VMs in `us-east1-b`), so it's important to be careful or you may accidentally put your data far away from where you need it.

### **8.3 Storing data in Cloud Storage**

As always, there are many ways to get started with Cloud Storage, so we'll walk through a few different ways starting with the Cloud Console, then moving onto the command line with the Cloud SDK (`gsutil`), and then using your own code in Node.js (`@google-cloud/storage`).

Before you can start storing data you first have to create a bucket. Since bucket names need to be globally unique, you won't be able to use the same bucket name used here so feel free to append your name to the bucket to keep it unique. Start by heading over to the Cloud Console, and choosing "Storage" from the left navigation. You should see a prompt to create a bucket that looks like this:

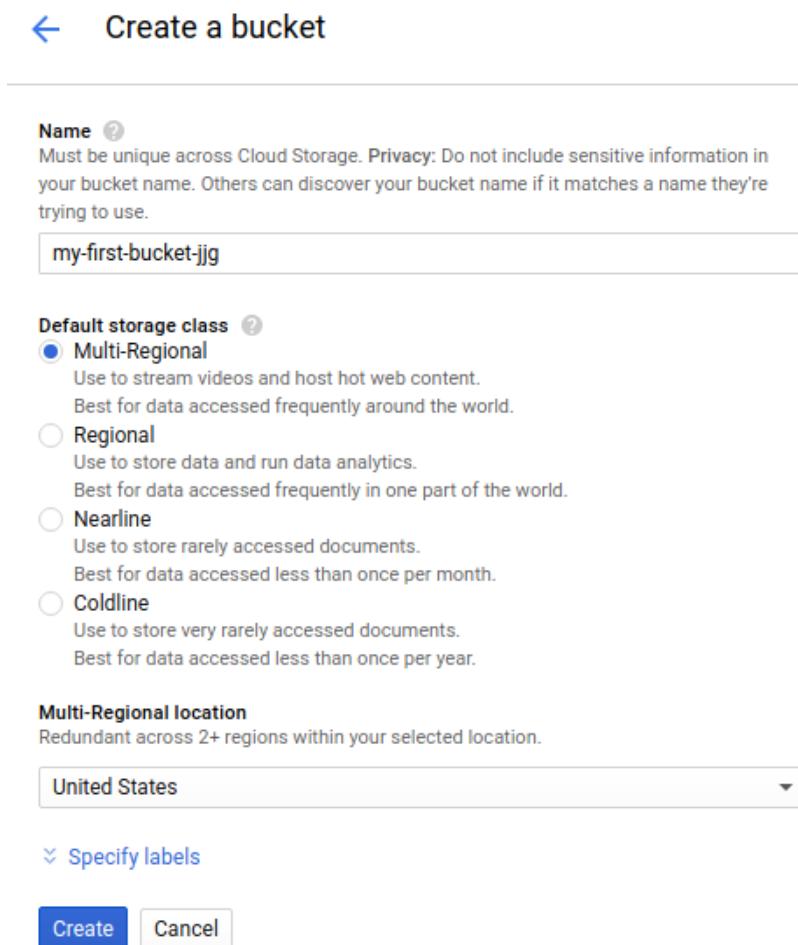
**Figure 8.1. Your first visit to the Cloud Storage UI**



When you click there, you'll see a field for the name of the bucket, as well as drop-down selectors for the storage class and location. For now, leave the drop-downs as they are (we'll discuss more about these later) and just enter in a unique name for your

bucket. Here we're using `my-first-bucket-jjg`.

**Figure 8.2. Create your first bucket**



Once that's finished, let's try exploring Cloud Storage with the command line.

**NOTE**

Cloud Storage currently has its own separate command-line tool called `gsutil`. Even though it's under a different command, it's still installed and updated with the Cloud SDK. If you don't see the command on your machine, try running `gcloud components install gsutil`.

First, try listing out the buckets available to you with `gsutil ls`. (And don't forget to make sure you're authenticated with `gcloud auth login`.)

**Listing 8.1. Listing your buckets with gsutil**

```
$ gsutil ls
gs://my-first-bucket-jjg/
```

Now let's try uploading a simple text file with `gsutil`. If you have a file laying around that you want to upload feel free to use that. If you don't, just create a small text file for this example.

**Listing 8.2. Uploading your first file**

```
$ echo "This is my first file!" > my_first_file.txt
$ cat my_first_file.txt
This is my first file!

$ gsutil cp my_first_file.txt gs://my-first-bucket-jjg/
Copying file://my_first_file.txt [Content-Type=text/plain]...
Uploading   gs://my-first-bucket-jjg/my_first_file.txt:           23 B/23 B
```

Now let's look in our bucket back in the Cloud Console to see if it worked!

**Figure 8.3. Checking that our file was uploaded**

The screenshot shows the Google Cloud Storage interface for a bucket named 'my-first-bucket-jjg'. At the top, there are navigation links for 'Browser', 'UPLOAD FILES', 'UPLOAD FOLDER', 'CREATE FOLDER', 'REFRESH', 'SHARE PUBLICLY', and a trash can icon. Below this is a search bar with the placeholder 'Filter by prefix...'. The main table lists one object:

Name	Size	Type	Last modified	Share publicly
my_first_file.txt	23 B	text/plain	5/22/16, 1:06 PM	<input type="checkbox"/>

As you can see, the file (called an object in this context) made its way into your newly created bucket. Now let's try accessing Cloud Storage from your own code. To do this, we'll need the `@google-cloud/storage` package which we can install by running `npm install @google-cloud/storage@0.2.0`. Once that's ready, we can test the waters by listing out the contents of a bucket, shown below.

**Listing 8.3. Listing the contents inside a bucket**

```
const storage = require('@google-cloud/storage')({
  projectId: 'your-project-id'
});
const bucket = storage.bucket('my-first-bucket-jjg');
bucket.getFiles()
  .on('data', (file) => {
    console.log('Found a file called', file.name);
  })
  .on('end', () => {
    console.log('No more files!');
  });
});
```

Make sure to plug in your bucket name and your project ID before you run the script.

Afterwards, you should see output that looks something like this:

#### **Listing 8.4. Output of listing files in a bucket**

```
Found a file called my_first_file.txt
No more files!
```

And what about uploading files? Let's try uploading a new file. First, create `my_second_file.txt` by adding some text to a new file (e.g., `echo "This is my second file!" > my_second_file.txt`) and then we'll write a script that uploads the file.

#### **Listing 8.5. Script to upload a file to Cloud Storage**

```
const storage = require('@google-cloud/storage')({
  projectId: 'your-project-id'
});
const bucket = storage.bucket('my-first-bucket-jjg');
bucket.upload('my_second_file.txt', (err, file) => {
  if (err) {
    console.log('Whoops! There was an error:', err);
  } else {
    console.log('Uploaded your file to', file.name);
  }
});
```

If we run this script, we should see a message saying the file was uploaded. After this, if you re-run the script to list files, you should see the new file listed in the results.

#### **Listing 8.6. Running the script to upload a file**

```
Uploaded your file to my_second_file.txt
```

Now that you understand how to interact with Cloud Storage, let's jump back to some of the topics we skipped over before such as the "class" of storage for your buckets.

## **8.4 Choosing the right storage class**

Just like there are different types of hard drives (e.g., SSD or magnetic), Cloud Storage offers a few different types of buckets that you can configure in Cloud Storage. These different storage classes come with different performance characteristics (that is, both latency and availability) as well as different prices. The reason is simply that different use-cases require different features, so Cloud Storage offers a few different choices that are likely to best match the situation you're in.

Let's start by running through the most common one: Multi-regional storage.

### **8.4.1 Multi-regional storage**

Multi-regional storage is the most commonly used option and the one that is likely to fit the needs of most applications. The flip side is that it's also the most expensive of

the options available because it replicates data across several different regions inside the chosen location. (The current location options are "US", "EU", and "Asia".)

If you don't know exactly where you'll be requesting your data from, multi-regional storage provides the best latency available due to Google's ability to cache data at the nearest edge to the requester. In addition to this, since the data is replicated across several different regions this means that it can offer the highest availability.

Due to all of this, multi-regional storage is likely the best choice for frequently served content to lots of different destinations, such as website content, streaming videos to users, or mobile application data. Generally, if your users are going to waiting on this data (and you want them to get it quickly) you probably want to use multi-regional storage.

### **8.4.2 Regional storage**

In many ways, the regional storage class is like a slimmed-down version of the multi-regional storage class. This is because instead of replicating data across a bunch of different regions inside an area (e.g., "US"), this class replicates the data across a few different zones inside a single region (e.g., "Iowa"). Because this storage class doesn't spread data as far apart, this means it offers slightly lower availability, and latency to destinations far away from the region chosen (e.g., sending data from the Iowa region to Belgium) might be slightly higher.

In exchange for this, data stored in the regional storage class costs about 20% less per GB stored, making it attractive if you happen to know where your data will be needed in the future.

### **8.4.3 Nearline storage**

Nearline storage attempts to closely match the data archival use-case by making a few key trade-offs that you shouldn't even notice if you're using the data as intended. For example, Nearline storage offers slightly lower availability as well as higher latency to the first byte. In other words, Nearline focuses on the scenario where you don't really need your data all that often, and when you do, you can wait a bit for the download to start.

In exchange for these differences, data stored in the Nearline storage class has a slightly different pricing model. This model is explored in much more detail in "[Understanding pricing](#)", but the key difference is that in addition to the other pricing components you'll learn about, per-operation cost is slightly higher (e.g., overhead of running a "get"), data retrieval is not free like it is with regional or multi-regional storage, and there's a 30-day minimum cliff for data in this class. On the other hand, the cost for data in this class is around 60% less per GB stored, which means it's a great deal when it matches your system's needs.

On the other hand, if you need to make frequent changes to your data or even retrieve the data on more than a monthly basis, this storage class will end up being much more expensive than the other options. This means that it's typically a poor choice for

anything that is "customer-facing" (such as downloads on a website).

#### 8.4.4 Coldline storage

Coldline storage is targeted at the extreme end of the data-archival spectrum. By this we mean the data that is used primarily in the case of a serious disaster. For example, we might need to restore our database back-ups on a monthly basis for one reason or another, making that data a great fit for Nearline. On the other hand, if there is a security breach of some sort and we're calling in the FBI to investigate, they might want all transaction logs for the past year. That data would be a much better fit for the Coldline storage class because we probably aren't calling the FBI on a monthly basis but we do still want the data around in case.

Outside of this, Coldline is very similar to Nearline in that it has similar per-operation costs as well as data retrieval costs, however instead of a 30-day minimum storage duration Coldline storage has a 90-day minimum. In exchange for all of this, Coldline is about 30% cheaper than Nearline on a per-GB basis, making it about 70% cheaper than multi-regional storage. This means that if you happen to fit the mold for Coldline storage, using this class can save you quite a bit of money.

In general, Coldline is a great choice for scenarios that seem to fit into Nearline, but taken to an extreme. In other words, you'd want to use Coldline in scenarios where you have data that you rarely need (e.g., once per year) but want to make sure that it is there when you do end up needing it. In exchange for not needing the data very often, you get a much lower price overall to store it.

#### 8.4.5 Summary

**Table 8.1. Overview of storage classes**

	Multi-regional	Regional	Nearline	Coldline
<b>Cost per GB</b>	\$0.026	\$0.02	\$0.01	\$0.007
<b>SLA</b>	99.95%	99.9%	99.0%	99.0%
<b>Data retrieval costs</b>	No	No	Yes	Yes
<b>Per-operation costs</b>	Normal	Normal	Higher	Higher
<b>Minimum duration</b>	None	None	30 days	90 days
<b>Typical use-case</b>	Website data	Analytical data	Archival	Disaster archival

Generally, since the cost difference for small amounts of data (10s of GB) your safest bet is to use multi-regional storage whenever you're unsure how often you'll need to access your data or how quickly you'll need it. As you start storing more data, you should take a look at your access patterns, keeping an eye specifically for whether you're accessing data in one single place, or very infrequently. If you see all data being accessed from a single zone (or region), it's worth looking at regional storage simply for the cost savings. Additionally, if you find you're not accessing certain data very often, it may be a good idea to investigate using Nearline (or even Coldline if the access is *really* infrequent).

Regardless of this, all of your data is replicated and saved across Google's data centers, so you shouldn't worry about **losing** your data. The storage classes are specifically about the performance (how long it takes Google to start sending you the file after you request it) and availability as well as the overall price per GB, but never about the *durability*. Your data is always the same amount of "safe", with a 99.999999999% durability guarantee (that's 11 total 9s in case you didn't want to count).

Now that you understand some of the fundamentals, let's dig a bit deeper into the more advanced concepts. These might not seem important at first but as you begin using Cloud Storage in more real-life scenarios these features will become far more interesting.

## 8.5 Access control

We've talked about Cloud Storage being a safe place to put all of your data, but haven't actually explained much about how to control who is able to access or modify the data once it's stored.

### 8.5.1 Limiting access with ACLs

So far we've discussed interacting with your data while authorized as a service account (the thing in `key.json` in your code examples) or as yourself (you start by typing `gcloud auth login`). But how does it work when you want to allow others to access your data? How do you restrict who can do what?

Before we get into more detail it might be worthwhile to say that **by default** everything you create is locked down to be accessible by only those people who have access to your project. This means that if you are working alone in your project, all of your data is restricted to "just you". When you add someone else, just as they get access to other parts of your project they also will have access to your data in Cloud Storage. For example, if you add someone as another "owner" of the project they will be able to control your Cloud Storage data (buckets and objects) just like you can, so be careful about who you add to your project! Now that that's out the way, let's dive into some of the specific things you should understand in order to control who can access your data.

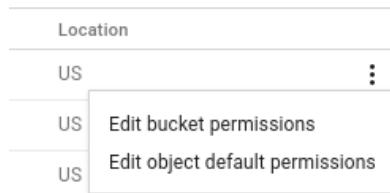
Cloud Storage allows fine-grained access control of your buckets and objects through a security mechanism called "Access Control Lists" or ACLs. These lists do exactly what you expect by letting you say which accounts can do which operations (e.g., "read" or "write").

These operations are conveyed by three groups, but these mean different things for buckets and objects.

**Table 8.2. Description of roles for Cloud Storage**

Role	Meaning (buckets)	Meaning (objects)
Readers	Bucket readers can list the objects in a bucket	Object readers can download the contents of an object
Writers	Bucket writers can list, create, overwrite, and delete objects from a bucket	(This doesn't apply as you can't have object writers.)
Owners	Bucket owners can do everything readers and writers can do, as well as update metadata such as ACLs.	Object owners can do everything readers can do as well as update metadata such as ACLs.

As you might expect, you control access to your objects by assigning these roles to different actors (e.g., a particular user). Let's start by looking at the ACL for your bucket in the Cloud Console. You can do this by clicking the "..." button on the far right in your list of buckets and choosing "Edit bucket permissions".

**Figure 8.4. Choose from the menu**

Once you choose "Edit bucket permissions" you should see something like this:

**Figure 8.5. Edit bucket permissions**

The screenshot shows the "Edit bucket permissions" dialog for the "jjg-personal" bucket. The dialog has the following sections:

- jjg-personal permissions**: A heading with a note: "These permissions only affect the bucket itself and do not apply to existing objects in the bucket."
- Share your bucket with this URL**: A text input field containing the URL <https://console.cloud.google.com/storage/browser/jjg-personal>.
- ENTITY NAME ACCESS**: A table listing three entries:
 

ENTITY	NAME	ACCESS
Project	editors-648816988819	Owner
Project	viewers-648816988819	Reader
Project	owners-648816988819	Owner
- + Add item**: A button to add a new permission entry.
- Save** and **Cancel**: Buttons at the bottom of the dialog.

As you can see, the default access on the bucket is based on the project with project editors and owners having Owner access and project viewers having Reader access. Adding access to a specific person is as easy as entering their e-mail address and choosing the access level. For example, here's what it looks like to grant "Reader" access to "[your-email@gmail.com](mailto:your-email@gmail.com)".

**Figure 8.6. Granting Reader access**

ENTITY	NAME	ACCESS
Project	editors-648816988819	Owner
Project	viewers-648816988819	Reader
Project	owners-648816988819	Owner
User	your-email@gmail.com	Reader

**Add item**

**Save** **Cancel**

Adding access to a specific user means that they will need to log in with Google's traditional sign-in, which means that they'll need to have a Google account.

In addition to adding access to individuals Cloud Storage also allows you to control access based on a few other things:

- User `allUsers`, as you might expect, is referring to "anyone". In other words, if you give Reader access to the `allUsers` User entity, the resource will be readable by anyone who asks for it.
- User `allAuthenticatedUsers` is similar to `allUsers`, but refers to anyone who is logged in with their Google account.
- Groups (e.g., `mygroup@googlegroups.com`) refer to all members of a specific Google Group. This allows you to grant access once and then control further access based on group membership.
- Domains (e.g., `mydomain.com`) refer to a Google Apps managed domain name. If you use Google Apps, this is a quick way to limit access to only those who are registered as users in your domain.

As we hinted before, in addition to setting permissions on your bucket you can also set these similar permissions on your individual objects, but this might raise questions about how the two lists interact. For example, what happens if you are an "Owner" for the bucket, but the object is only readable by a single person (not you)?

The answer is actually quite simple: each of the permissions conveys specific activities that are allowed so there is no "hierarchy" of permissions that trickle down. For example, imagine that you have Owner access to a bucket but only have Reader access to an object. In this scenario, while you can manipulate any data inside the bucket, you cannot update the metadata for the object itself. In other words, if you wanted to change the metadata, you'd have to re-create the object so that you have the requisite permissions.

### DEFAULT OBJECT ACLS

In addition to permissions on both buckets and objects Cloud Storage allows you to decide up-front what ACLs should be set on **newly created** objects in the form of a bucket's "default object ACLs". This follows the exact same pattern as a single object ACL (that is, you can have various Readers and Owners), but the ACL is defined at the bucket level, and then applied to all objects when they are created. For example, if you define your default object ACL to have `allUsers` as a Reader, all objects that you upload will be publicly readable as you create them.

It's important to note that default object ACLs are a template applied when you **create** an object and this does **not** modify existing objects in any way.

### PRE-DEFINED ACLS

As you might expect there are a few common scenarios that entail quite a bit of clicking (or typing) to get configured. To make this easier Cloud Storage has a few "pre-defined" ACLs that you can set using the `gsutil` command-line tool. This means that when you want to do common things like make an object "publicly readable" or "private" or "private to the project", you can do this with far fewer keystrokes. Let's try uploading a file to Cloud Storage and making it publicly readable.

#### Listing 8.7. Set a pre-defined ACL

```
$ gsutil mb gs://my-public-bucket          ①
Creating gs://my-public-bucket/...

$ echo "This should be public" > public.txt
$ gsutil cp public.txt gs://my-public-bucket ②
Copying file:/public.txt [Content-Type=text/plain]...
Uploading   gs://my-public-bucket/public.txt:                                23 B/23 B
```

- ① Start by creating a new bucket.
- ② Then we create new file and upload it to the bucket.

After that, we should look at the default ACL file. To get the ACL that GCS created by default, just run `gsutil acl get gs://my-public-bucket/public.txt`, you should

see something like the following.

#### **Listing 8.8. Stored ACL**

```
[
  {
    "entity": "project-owners-243576136738",
    "projectTeam": {
      "projectNumber": "243576136738",
      "team": "owners" ①
    },
    "role": "OWNER"
  },
  {
    "entity": "project-editors-243576136738",
    "projectTeam": {
      "projectNumber": "243576136738",
      "team": "editors" ①
    },
    "role": "OWNER"
  },
  {
    "entity": "project-viewers-243576136738",
    "projectTeam": {
      "projectNumber": "243576136738",
      "team": "viewers" ①
    },
    "role": "READER"
  },
  {
    "entity": "user-
00b4903a978dcf75fbff509edb5b5658a3c6972b0ef52feca6618b156ced45d8",
    "entityId": "00b4903a978dcf75fbff509edb5b5658a3c6972b0ef52feca6618b156ced45d8",
    "role": "OWNER"
  }
]
```

① Notice how by default the ACL has owners, editors, and viewers pre-set.

Now let's try getting that file over the public internet (e.g., not through `gsutil`), and then update the ACL to be public after that fails.

#### **Listing 8.9. Inspect and update the ACL**

```
$ curl https://my-public-bucket.storage.googleapis.com/public.txt
<?xml version='1.0' encoding='UTF-
8'?><Error><Code>AccessDenied</Code><Message>Access
denied.</Message><Details>Anonymous users does not have storage.objects.get access
to object my-public-bucket/public.txt.</Details></Error>

$ gsutil acl set public-read gs://my-public-bucket/public.txt
Setting ACL on gs://my-public-bucket/public.txt...

$ curl https://my-public-bucket.storage.googleapis.com/public.txt
This should be public!
```

As you can see in this example, if we look at the ACL that was created by default, it shows the project roles as well as the owner ID. When we try to access the object through curl it is rejected with an XML "Access Denied" error as expected. Then we can set the pre-defined ACL (`public-read`) with a single command, and after that the object is visible to the world. This isn't limited to just `public-read`. The following table shows more of the pre-defined ACLs in order of the likelihood that you'll use them.

**Table 8.3. Pre-defined ACL definitions**

Name	Meaning
<code>private</code>	Removes any permissions besides the single owner (creator).
<code>project-private</code>	The default for new objects which gives access based on roles in your project.
<code>public-read</code>	Gives anyone (even anonymous users) reader access.
<code>public-read-write</code>	Gives everyone (even anonymous users) reader and writer access.
<code>authenticated-read</code>	Gives anyone logged in with their Google account reader access.
<code>bucket-owner-read</code>	Used only for objects (not buckets), gives the creator owner access and the bucket owners read access.
<code>bucket-owner-full-control</code>	Gives object and bucket owners the owner permission.

It's important to point out that by using a pre-defined ACL, you are actually **replacing** the existing ACL. This means that if you have a long list of users who have special access and you apply any of the pre-defined ACLs the list you had previously will be overwritten. In other words you should be careful when applying pre-defined ACLs, particularly if you've spent a long time curating ACLs in the past. You should also try to use Group and Domain entities often rather than specific User entities as group membership won't be lost by setting a pre-defined ACL.

#### ACL BEST PRACTICES

Now that you understand quite a bit about ACLs, it seems useful to spend a bit of time describing a few "best practices" of how to manage ACLs and choose the right permissions for your buckets and objects. Keep in mind that this is a list of guidelines and not rules, so you should feel comfortable deviating from this list if you have a good reason.

##### ***When in doubt, give out the minimum access possible.***

This is a general security guideline and is just as relevant to controlling access to your data on Cloud Storage. If someone only needs permission to read the data of an object, give them Reader permission only. If you give out more than this, don't be surprised when someone borrowing their laptop accidentally removes a bunch of ACLs from the object.

In general, it's best to remember that you can always grant more access if someone should need it. You can't always undo things that a malicious or absent-minded user did.

***The Owner permission is powerful, so be careful with it.***

Owners can change ACLs and metadata, which means that unless you trust someone to grant further access appropriately, you shouldn't give them the owner permission.

Following on the principle above, when in doubt give the writer permission instead. There is no undo with your data, so you should trust that any new owners will "do the right thing", but that they also are careful enough to make sure that no one else can "do the wrong thing" either accidentally or purposefully.

***Allowing access to the public is a big deal, so do it sparingly.***

It's been said before that once something is on the internet it's there forever. This is just as true about your data once you expose it to the world. When using the allUsers or allAuthenticatedUsers (and therefore the public-read or authenticated-read) tokens, recognize that this is the same as publishing your content to the world.

We'll also discuss a concern about this when we dive into pricing later in this chapter.

***Default ACLs happen automatically, so choose sensible defaults.***

It's easy to miss when an overly open default ACL is set precisely because you don't notice until you look at the newly created object's ACL. It's also easy to break the rule about giving out the minimum access when you have a relatively "loose" ACL as default. In general, it's best to use one of the more strict pre-defined ACLs as your object default, such as project-private or bucket-owner-full-control if you're on a small team, and private or bucket-owner-read if you're on a larger team.

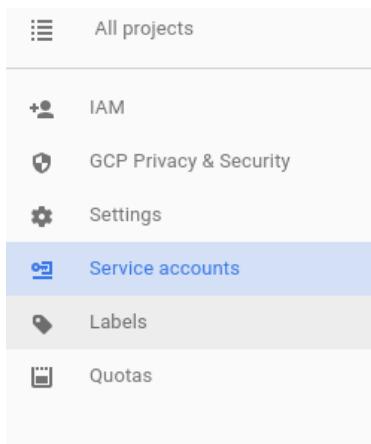
Now that you understand how to control access in the general sense, let's look at how to handle those "one-off" scenarios where you want to grant access to a single operation.

## 8.5.2 Signed URLs

It turns out that sometimes you don't want to add someone to the ACL forever, but would rather give someone access for a fixed amount of time. Further, you're not so much concerned about authenticating the user with their Google account, but have authenticated them with your own login system and simply want to say "this person has access to view this data". Luckily, Cloud Storage provides a pretty simple way to do with signed URLs.

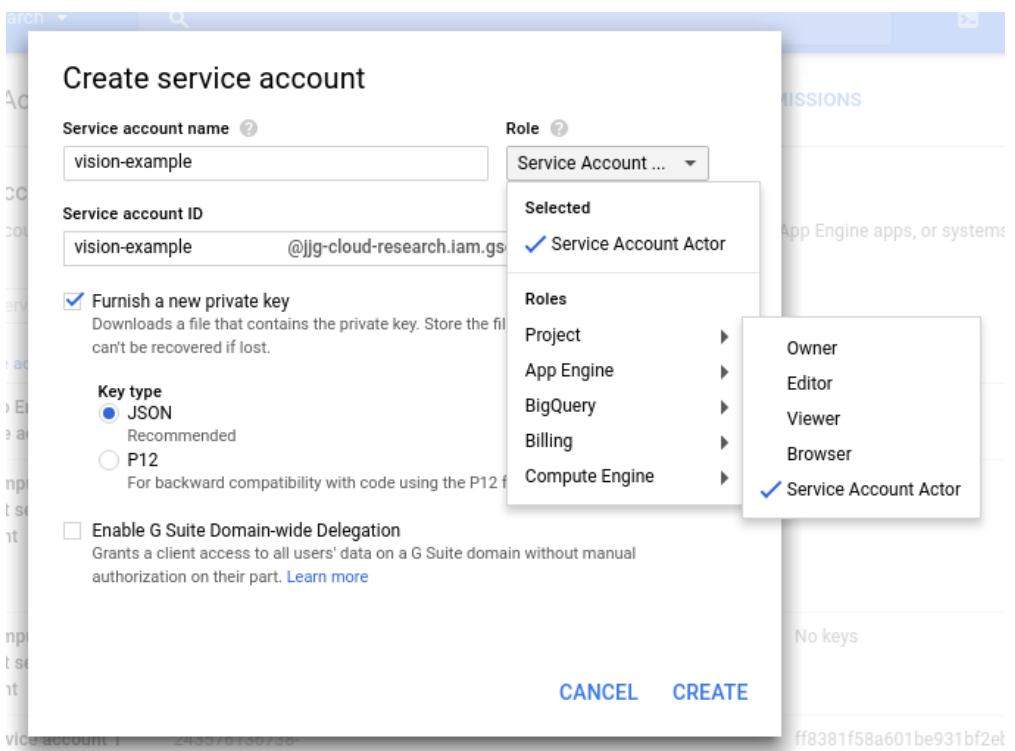
The way this works is by taking an "intent" to do an operation (e.g., download a file) and signing that intent with a credential that has access to actually do the operation. This allows you, someone with no access at all, to present this one-time pass as your credential to do exactly what the pass says you can do. Let's run through a simple example like creating a signed URL to download a text file from GCS. To start, you'll need a private key, so jump over to the IAM & Admin section, and choose "Service accounts" from the left side navigation.

**Figure 8.7. Choose "Service accounts" from the left side navigation.**



Then create a new service account, making sure to have Google generate a new private key in JSON format. In this case, we'll use the name "gcs-signer" as the name for this account.

**Figure 8.8. Create a new service account**



When you create the service account you should notice that it added the account to the list but also kicked off a download of a JSON file. Don't lose this file as it's the only copy of the private key for your account (Google doesn't keep a copy of the private key around for security reasons). Now let's quickly upload a file that we're sure is "private".

#### **Listing 8.10. Uploading a file which is private by default**

```
$ echo "This is private." > private.txt
$ gsutil cp private.txt gs://my-example-bucket/
Copying file://private.txt [Content-Type=text/plain]...
Uploading   gs://my-example-bucket/private.txt:           17 B/17 B

$ curl https://my-example-bucket.storage.googleapis.com/private.txt
<?xml version='1.0' encoding='UTF-
8'?><Error><Code>AccessDenied</Code><Message>Access
denied.</Message><Details>Anonymous users does not have storage.objects.get access
to object my-example-bucket/private.txt.</Details></Error>
```

And finally we should make sure the service account we created has access to the file. (Remember, the service account can only sign for things that it is able to do, so if it doesn't have access to the file, its signature is worthless.) We can grant access to our new service account by using `gsutil acl ch` ("ch" standing for "change").

#### **Listing 8.11. Grant access to a service account**

```
$ gsutil acl ch -u gcs-signer@your-project-id.iam.gserviceaccount.com:R gs://my-
example-bucket/private.txt
Updated ACL on gs://my-example-bucket/private.txt
```

Notice that the ACL we changes was of the form `-u service-account-email:R`. Service accounts are treated like users, so we use the `-u` flag, then we use the e-mail address based on the name of the service account, and finally we use `:R` to mean "add Reader privileges". Now that we have the right permissions, we just have to provide the right parameters to `gsutil` to build a signed URL.

**Table 8.4. Parameters for signing a URL with gsutil**

Parameter	Flag	Meaning	Example
Method	<code>-m</code>	The HTTP method for your request	GET
Duration	<code>-d</code>	How long until the signature expires	1h (one hour)
Content type	<code>-d</code>	The content type of the data involved (used only when uploading)	image/png

In our example we want to download (GET) a file called `private.txt`, and let's assume that the signature should expire in 30 minutes (30m). This means the parameters to `gsutil` would be:

**Listing 8.12. gsutil command to sign a URL**

```
$ gsutil signurl -m GET -d 30m key.json gs://my-example-bucket/private.txt
URL      HTTP Method      Expiration      Signed URL
gs://my-example-bucket/private.txt      GET      2016-06-21 07:07:35
https://storage.googleapis.com/my-example-bucket/private.txt?GoogleAccessId=gcs-
signer@your-project-
id.iam.gserviceaccount.com&Expires=1466507255&Signature=ZBufnbBAQ0z1oS8ethq%2B519C7
YmVHvbNM%2F%2B43z9XDcsTgpWoCbAMj2ZhugI%2FZWE665mxD%2BJL%2BJzVsY7BAD7qFWTok0vDn5a0s
q%2Be78nCJmgE01DTERQpnXSvb0ht0yV1Fr8p3StKU0ST1wKoNIcehfRXWD45fEMMFmchPhkI8M8ASwaI%
2FVNZOXp5HxtZvZac047NTC1B5k9uKBL1MEg65RAbBTt5huHRG06XkYgnyKDY87rs18HSEL4dMauUZpaYC4
ZPb%2FSBpWAMOneaXpTH1h4cKXXNl1rQ03MUf5w3sKKJBsUWB10xoAsf3HpdnnrFjW5sUZUQu1RRTqHyztc4
Q%3D%3D
```

It's a bit tough to read but the last piece of output there is a URL that will allow you to read the file `private.txt` from any computer for the next 30 minutes. After that, it expires and you'll go back to getting the Access Denied errors we saw before. To test this out, we can try getting the file with and without the signed piece.

**Listing 8.13. Retrieving our file**

```
$ curl -S https://storage.googleapis.com/my-example-bucket/private.txt
<?xml version='1.0' encoding='UTF-
8'?><Error><Code>AccessDenied</Code><Message>Access
denied.</Message><Details>Anonymous users does not have storage.objects.get access
to object my-example-bucket/private.txt.</Details></Error>

$ curl -S "https://storage.googleapis.com/my-example-
bucket/private.txt?GoogleAccessId=gcs-signer@your-project-
id.iam.gserviceaccount.com&Expires=1466507255&Signature=ZBufnbBAQ0z1oS8ethq%2B519C7
YmVHvbNM%2F%2B43z9XDcsTgpWoCbAMj2ZhugI%2FZWE665mxD%2BJL%2BJzVsY7BAD7qFWTok0vDn5a0s
q%2Be78nCJmgE01DTERQpnXSvb0ht0yV1Fr8p3StKU0ST1wKoNIcehfRXWD45fEMMFmchPhkI8M8ASwaI%
2FVNZOXp5HxtZvZac047NTC1B5k9uKBL1MEg65RAbBTt5huHRG06XkYgnyKDY87rs18HSEL4dMauUZpaYC4
ZPb%2FSBpWAMOneaXpTH1h4cKXXNl1rQ03MUf5w3sKKJBsUWB10xoAsf3HpdnnrFjW5sUZUQu1RRTqHyztc4
Q%3D%3D"
This is private.
```

Note that we added quotes around the URL (since there are extra parameters that would be interpreted by the command line).

You might be thinking that this is great when you happen to be sitting at your computer, but isn't the more common scenario where you have content in your app that you want to share temporarily with users? For example, you might want to serve photos, but you don't want them always available to the public to discourage things like hot-linking. Luckily this is pretty easy to do in code, so let's look at a short example snippet in Node.js.

The basic premise is the same, but we'll do it in JavaScript rather than on the command line with `gsutil`.

**Listing 8.14. Sign a URL to grant specific access**

```
const storage = require('@google-cloud/storage')({
  projectId: 'your-project-id'
  keyFilename: 'key.json'
});
const bucket = storage.bucket('my-example-bucket');
const file = bucket.file('private.txt');

file.getSignedUrl({
  action: 'read', // This is equivalent to HTTP GET.
  expires: new Date().valueOf() + 30*60000, // This says "30 minutes from now"
}, (err, url) => {
  console.log('Got a signed URL:', url);
});
```

When you run this you should see something like the following:

**Listing 8.15. Running our script to sign a URL**

```
Got a signed URL: https://storage.googleapis.com/my-example-
bucket/private.txt?GoogleAccessId=gcs-signer@your-project-
id.iam.gserviceaccount.com&Expires=1466508154&Signature=LW0AqC4E31I7c1JgMhuljeJ8WC0
1qnazEeqE%2B2ikSPmzThauAqtFxo2WYfL%2F5MnbBF%2FUDj1gsESjwB2Ar%2F5EoRDFY209GRE50Iu0
hAoWk3KbqQ4sIUrxmSF%2BZymU1Nou1BEEPXaHgeQNICY1snkjF7pQpEU9fkjTcwxFtBcYx7n3irIW27IY
Jx4JQ8146bfFweiHei%2B7fvZkO81fP5XY%2BM2kCovfewSb8KclPZ8501tW9g8Xmo%2Fvf3rZpwF27rgV4
UPDwz247Fn7UAm17T%2B%2FmEeANY1RoQtb8I1hnH110ta36iWKOV1GQ%2FYh7F2JsDhJxZTwXkIR51zSR
8nD2Q%3D%3D
```

This means that if you wanted to render this new value as the image `src` attribute, you could do that instead of using a `console.log` statement.

Now that you understand how to change the access restrictions on data, let's also look at how to keep track of who's accessing your data.

### **8.5.3 Logging access to your data**

If you're managing any data that is somewhat sensitive (for example, perhaps you're storing employee records) you probably want to keep track of when this data is accessed. Cloud Storage makes this pretty simple by allowing you to set that a specific bucket should have its access logged. To do this, you use the Cloud Storage API to specify a logging configuration which just says where the logs should end up (the `logBucket`) and whether Cloud Storage should put a prefix on the beginning of the log files (the `logObjectPrefix`).

Let's try interacting with our logging configuration using the `gsutil` command-line tool.

**Listing 8.16. Interacting with the logging configuration using gsutil**

```
$ gsutil logging get gs://my-example-bucket
gs://my-example-bucket/ has no logging configuration.
```

①

```
$ gsutil mb -l US -c multiRegional gs://my-example-bucket-logs ②
Creating gs://my-example-bucket-logs/...

$ gsutil acl ch -g cloud-storage-analytics@google.com:W \
  gs://my-example-bucket-logs ③

$ gsutil logging set on -b gs://my-example-bucket-logs \
  -o example-prefix gs://my-example-bucket
Enabling logging on gs://my-example-bucket/...

$ gsutil logging get gs://my-example-bucket
{
  "logBucket": "my-example-bucket-logs",
  "logObjectPrefix": "example-prefix"
} ⑤
```

- ① We start by checking the logging configuration for a bucket (my-example-bucket), and then configure logging on it.
- ② To do this, we create a bucket that will hold all of the logs (my-example-bucket-logs)
- ③ After that, we grant access to the "logger" account (cloud-storage-analytics@google.com) that will be responsible for putting the logs into that bucket.
- ④ Finally, we configure the logging details, telling Cloud Storage to place all access logs into the newly created bucket.
- ⑤ To check that it worked, we can use the gsutil logging get command to show the configuration we saved and make sure it's all accurate.

Once you have your configuration set, Cloud Storage will store all access logs in the logging bucket every hour that there is activity to report. The log files themselves will be named based on your prefix, a timestamp of the hour being reported, and a unique ID (e.g., 1702e6). For example, a file from our logging configuration might look like example-prefix\_storage\_2016\_06\_18\_07\_00\_00\_1702e6\_v0. Inside each of the log files, you'll see lines of comma-separated fields (you've probably seen .csv files before), with the following schema.

**Table 8.5. Schema of access log files**

Field (type)	Description
time_micros (int)	The time that the request was completed, in microseconds since the Unix epoch.
c_ip (string)	The IP address from which the request was made.
c_ip_type (integer)	The type of IP in the c_ip field (1 for IPv4, and 2 for IPv6)
c_ip_region (string)	Reserved for future use.
cs_method (string)	The HTTP method of this request.
cs_uri (string)	The URI of the request.
sc_status (integer)	The HTTP status code the server sent in response.
cs_bytes (integer)	The number of bytes sent in the request.
sc_bytes (integer)	The number of bytes sent in the response.
time_taken_micros (integer)	The time it took to serve the request in microseconds.

<code>cs_host (string)</code>	The host in the original request.
<code>cs_referer (string)</code>	The HTTP referrer for the request.
<code>cs_user_agent (string)</code>	The User-Agent of the request. For requests made by lifecycle management, the value is GCS Lifecycle Management.
<code>s_request_id (string)</code>	The request identifier.
<code>cs_operation (string)</code>	The Google Cloud Storage operation.
<code>cs_bucket (string)</code>	The bucket specified in the request. If this is a list buckets request, this can be null.
<code>cs_object (string)</code>	The object specified in this request. This can be null.

Note that each of the fields in the access log entry will be prefixed by a `c`, `s`, `cs`, or `sc`. These prefixes are explained in the following table.

**Table 8.6. Access log field prefix explanation**

Prefix	Stands for...	Meaning
<code>c</code>	client	Information about the client making a request.
<code>s</code>	server	Information about the server receiving the request.
<code>cs</code>	client to server	Information sent from the client to the server.
<code>sc</code>	server to client	Information sent from the server to the client.

Although uncommon, it's possible that the log entries will have duplicates, so you should use the `s_request_id` field as a unique identifier if you ever need to be completely confident that an entry is not a duplicate.

Now that you have a grasp of access control, let's move on to another slightly more advanced topic: versioning.

## 8.6 Object versions

Just like version control (like Git, Subversion, or Mercurial), Cloud Storage has the ability to turn on versioning, where you can have objects with multiple revisions over time. Also, when versioning is enabled you can revert back to an older version just like you can with files in a Git repository.

The biggest change when object versioning is enabled is simply that overwriting data doesn't truly overwrite the original data. Instead the previous version of the object will be archived and the new version marked as the "active" version. This means that if you upload a 10 MB file called "data.csv" into a bucket with versioning enabled, and then re-upload the revised 11 MB file of the same name, you'll end up with the original 10 MB file archived in addition to the new file, which means you're storing a total of 21 MB (not just 11 MB).

In addition to version of objects, Cloud Storage also supports different versions of the metadata on the objects. In the same way that an object could be archived and a new "generation" is added in its place, when metadata (such as ACLs) are changed on a versioned object, the metadata gets a new "metageneration" to keep track of its changes. This means that in any version-enabled bucket, every object will have a

generation (tracking the object version) along with a metageneration (tracking the metadata version). As you might imagine this feature becomes really useful when you have object data (or metadata) that changes over time, but you want to have easy access to the latest version. Let's explore how to get this set up and then demonstrate how you can do some of these common tasks that we just mentioned.

As you just learned object versioning is a feature that's enabled on a bucket, so the first thing you need to do to get started is enable the feature.

#### **Listing 8.17. Enable object versioning**

```
$ gsutil versioning set on gs://my-versioned-bucket
Enabling versioning for gs://my-versioned-bucket/...
```

Now let's check that versioning is enabled and then try uploading a new file.

#### **Listing 8.18. Check versioning is enabled and upload a text file**

```
$ gsutil versioning get gs://my-versioned-bucket
gs://my-versioned-bucket: Enabled

$ echo "This is the first version!" > file.txt
$ gsutil cp file.txt gs://my-versioned-bucket/
Copying file://file.txt [Content-Type=text/plain]...
Uploading   gs://my-versioned-bucket/file.txt:                                27 B/27 B
```

Now let's look more closely at the file by using the `ls -la` command. The `-l` flag shows the "long" listing which includes some extra information about the file, and the `-a` flag shows non-current (e.g., archived) objects along with extra metadata about the object such as the generation and meta-generation.

#### **Listing 8.19. Listing objects with -la flags**

```
$ gsutil ls -la gs://my-versioned-bucket
27 2016-06-21T13:29:38Z gs://my-versioned-bucket/file.txt#1466515778205000
metageneration=1
TOTAL: 1 objects, 27 bytes (27 B)
```

As you can see, the metageneration (or the version of the metadata) is pretty obvious (`metageneration=1`). The generation (or version) of the object isn't as obvious, but it's that long number after the # in the file name, in this example `1466515778205000`. As we learned before, when versioning is enabled on a bucket new files of the same name archive the old version before replacing the file, so let's try that and then look again at what ends up in the bucket.

#### **Listing 8.20. Upload a new version of the file**

```
$ echo "This is the second version." > file.txt
$ gsutil cp file.txt gs://my-versioned-bucket/
Copying file://file.txt [Content-Type=text/plain]...
```

```
Uploading gs://my-versioned-bucket/file.txt: 28 B/28 B

$ gsutil ls -l gs://my-versioned-bucket
28 2016-06-21T13:39:11Z gs://my-versioned-bucket/file.txt
TOTAL: 1 objects, 28 bytes (28 B)

$ gsutil ls -la gs://my-versioned-bucket
27 2016-06-21T13:29:38Z gs://my-versioned-bucket/file.txt#1466515778205000
metageneration=1
28 2016-06-21T13:39:11Z gs://my-versioned-bucket/file.txt#1466516351939000
metageneration=1
TOTAL: 2 objects, 55 bytes (55 B)
```

Notice how when listing objects without the `-a` flag we only see the latest generation, but when listing with it you can see all generations. In addition to that the total data stored in the first operation appears to be 28 bytes, however when listing everything (with the `-a`) flag the total data stored is 55 bytes. Finally, when we look at the latest version it should appear to be the more recent file we uploaded.

#### **Listing 8.21. Inspecting the latest version**

```
$ gsutil cat gs://my-versioned-bucket/file.txt
This is the second version.
```

However, if you want to look at the previous version you can do this by referring to the specific generation you want to see. Let's try looking at the previous version of our file.

#### **Listing 8.22. Inspecting a different generation**

```
$ gsutil cat gs://my-versioned-bucket/file.txt#1466515778205000
This is the first version!
```

As you can see, versioned objects are just like any other but have a special "tag" on the end referring to the exact generation. This means that you can treat them as "hidden" objects, but still objects, so you can delete prior versions just like any other object.

#### **Listing 8.23. Deleting a prior generation**

```
$ gsutil rm gs://my-versioned-bucket/file.txt#1466515778205000
Removing gs://my-versioned-bucket/file.txt#1466515778205000...

$ gsutil ls -la gs://my-versioned-bucket
28 2016-06-21T13:39:11Z gs://my-versioned-bucket/file.txt#1466516351939000
metageneration=1
TOTAL: 1 objects, 28 bytes (28 B)
```

There is a bit of surprising behavior though when deleting objects from versioned buckets in that deleting the file itself doesn't delete other generations! For example, if we were to delete our file (`file.txt`) then "getting" the file would return a 404, however the exact generation of the file would still exist, and we could read that file

still by its specific version. Let's demonstrate this by continuing our example.

#### **Listing 8.24. Deleting the current file**

```
$ gsutil ls -la gs://my-versioned-bucket/
28 2016-06-21T13:54:26Z gs://my-versioned-bucket/file.txt#1466517266796000
metageneration=1
TOTAL: 1 objects, 28 bytes (28 B)

$ gsutil rm gs://my-versioned-bucket/file.txt
Removing gs://my-versioned-bucket/file.txt...
```

At this point, we've deleted the latest version of the file so we expect it to be gone. Let's look at the different views to see what actually happened.

#### **Listing 8.25. Listing the contents of our versioned bucket after deleting**

```
$ gsutil ls -l gs://my-versioned-bucket/
$ gsutil ls -la gs://my-versioned-bucket/
28 2016-06-21T13:54:26Z gs://my-versioned-bucket/file.txt#1466517266796000
metageneration=1
TOTAL: 1 objects, 28 bytes (28 B)

$ gsutil cat gs://my-versioned-bucket/file.txt
CommandException: No URLs matched: gs://my-versioned-bucket/file.txt

$ gsutil cat gs://my-versioned-bucket/file.txt#1466517266796000
This is the second version.
```

Notice that while the file appears to be gone, a prior version still exists and is actually readable if referred to by its exact generation ID! This allows you to actually restore your previous versions if needed, which you can do by "copying" the previous generation into place. Let's look at how to restore the second version of our file.

#### **Listing 8.26. Restoring a version**

```
$ gsutil cp gs://my-versioned-bucket/file.txt#1466517266796000 gs://my-versioned-
bucket/file.txt
Copying gs://my-versioned-bucket/file.txt#1466517266796000 [Content-
Type=text/plain]...
Copying      gs://my-versioned-bucket/file.txt:                                28 B/28 B

$ gsutil cat gs://my-versioned-bucket/file.txt
This is the second version.
```

You might expect that you just "brought the old version back to life", but let's look at the directory listing to see if that's true.

#### **Listing 8.27. Listing versions after a restoration**

```
$ gsutil ls -la gs://my-versioned-bucket
28 2016-06-21T13:54:26Z gs://my-versioned-bucket/file.txt#1466517266796000
```

```
metageneration=1
28 2016-06-21T13:59:39Z gs://my-versioned-bucket/file.txt#1466517579727000
metageneration=1
TOTAL: 2 objects, 56 bytes (56 B)
```

It turns out that you actually created a new version by restoring the old one, so technically you now have two files with the same content in your bucket! Luckily if you actually want to remove the file along with all of its previous versions you can do this by passing the `-a` flag to the `gsutil rm` command.

#### **Listing 8.28. Remove the file and all versions**

```
$ gsutil rm -a gs://my-versioned-bucket/file.txt
Removing gs://my-versioned-bucket/file.txt#1466517266796000...
Removing gs://my-versioned-bucket/file.txt#1466517579727000...
$ gsutil ls -la gs://my-versioned-bucket/
```

As you can see, by using the `-a` flag you can get rid of all the previous versions of an object in one swoop.

To summarize quickly, specific object generations can be treated as individual objects in the sense that you can operate on them like any other object. They have special features in the sense that they are automatically archived when you overwrite (or delete) the object, but as far as usage goes archived versions shouldn't scare you any more than hidden files on your computer (and coincidentally you use the same commands to view those files on most systems).

You might be wondering now about how to keep your bucket from growing out of control. For example, it's easy to decide you're done with a file (and all of its versions), but how do you decide when you're done with a version? How old is too old? And isn't it obnoxious to have to continuously clean up old versions of objects in your bucket? Let's look at how we deal with this problem next.

## **8.7 Object lifecycles**

As you add more objects to your buckets you might notice that it's easy for you to accumulate a bunch of less-than-useful data in the form of old or out of date objects. This problem can be compounded when you have versioning enabled on your bucket because old versions will build up based on changes and they won't be as noticeable if you happen to be browsing your buckets for files that can be deleted.

To deal with this accumulation problem Cloud Storage allows you to define a way for you to conditionally delete data automatically so that you don't have to remember to clean up your bucket every so often. You'll hear this concept referred to elsewhere as "lifecycle management" as it's a definition of when an object is at the "end of life" and therefore should be deleted.

There are a few different conditions that you can set up to define when objects should be automatically deleted in your bucket:

### **Per-object age (Age)**

This is equivalent to fixing a number of days to live (sometimes referred to as a TTL). When you have an age condition, you're effectively saying delete this object N days after its creation date.

### **Fixed date cut-off (CreatedBefore) ...**

When setting a lifecycle configuration, you can specify that any objects with a creation date before the configured one will be deleted. This is an easy way to throw away any created before a fixed date.

### **Version history (NumberOfNewVersions)**

If you have versioning enabled on your bucket, this condition allows you to delete any objects that are the Nth oldest (or older) version of a given object. This is sort of like saying "I only need the last 5 revisions, remove anything older than that". Note that this isn't related at all to timing, but the volatility (number of changes) to the object.

### **Latest version (IsLive)**

This allows you to delete only the archived (or non-archived) versions, effectively allowing you to discard all version history if you want to make a fresh start. To apply a configuration, we have to assemble these conditions into a JSON file as a collection of "rules". Then we apply the configuration to the bucket. Inside each rule, all of the conditions are AND-ed together, and if they all match then the object is deleted.

Let's look at a simple example lifecycle configuration where we just want to delete any object older than 30 days.

#### **Listing 8.29. Delete objects older than 30 days**

```
{
  "rule": [
    {
      "action": {"type": "Delete"},
      "condition": {"age": 30}
    }
  ]
}
```

Imagine we like that rule, but also want to delete objects older than 30 days, as well as any objects that have more than 3 newer versions. To do this, we use two different rules which are applied separately.

#### **Listing 8.30. Delete things older than 30 days, and anything with at least 3 newer versions.**

```
{
  "rule": [
    {
      "action": {"type": "Delete"},
```

```

        "condition": {"age": 30}
    },
{
    "action": {"type": "Delete"},
    "condition": {
        "isLive": false,
        "numNewerVersions": 3
    }
}
]
}

```

Note that inside a single rule, the conditions are AND-ed in the sense that both of the conditions must be met, however each individual rule is applied separately, which effectively means the rules are OR-ed in the sense that if any rule matches the file will be deleted.

Now that you understand the format of a lifecycle configuration policy, let's try setting these rules on our buckets. For the purpose of demonstration, let's choose an policy that's easy to test, such as "delete anything that has at least 1 newer version".

#### **Listing 8.31. Delete anything with at least 1 newer version**

```
{
    "rule": [
        {
            "action": {"type": "Delete"},
            "condition": {
                "isLive": false,
                "numNewerVersions": 1
            }
        }
    ]
}
```

Start by saving this demonstration policy to a file called `lifecycle.json`. Then let's apply this policy to our versioned bucket from before.

#### **Listing 8.32. Interacting with the lifecycle configuration using gsutil**

```
$ gsutil lifecycle get gs://my-versioned-bucket
gs://my-versioned-bucket/ has no lifecycle configuration.

$ gsutil lifecycle set lifecycle.json gs://my-versioned-bucket
Setting lifecycle configuration on gs://my-versioned-bucket/...

$ gsutil lifecycle get gs://my-versioned-bucket
{"rule": [{"action": {"type": "Delete"}, "condition": {"isLive": false, "numNewerVersions": 1}}]}
```

If you try uploading some files you might notice that they aren't immediately deleted according to the configuration you just set up. This might seem strange but keep in mind that the "clean up" happens on a regular interval, not immediately.

That said, while the object might not be deleted immediately, you aren't billed for storing objects that satisfy the lifecycle configuration but haven't been deleted yet. However, if you access a file that isn't yet deleted you will be billed for those operations and bandwidth. Put simply: once an object **should** be deleted, you are no longer charged for storage, but are charged for any other operations.

Now that you understand how to keep your data tidy, let's take a look at how you might connect Cloud Storage to your app in an event-driven way.

## 8.8 Change notifications

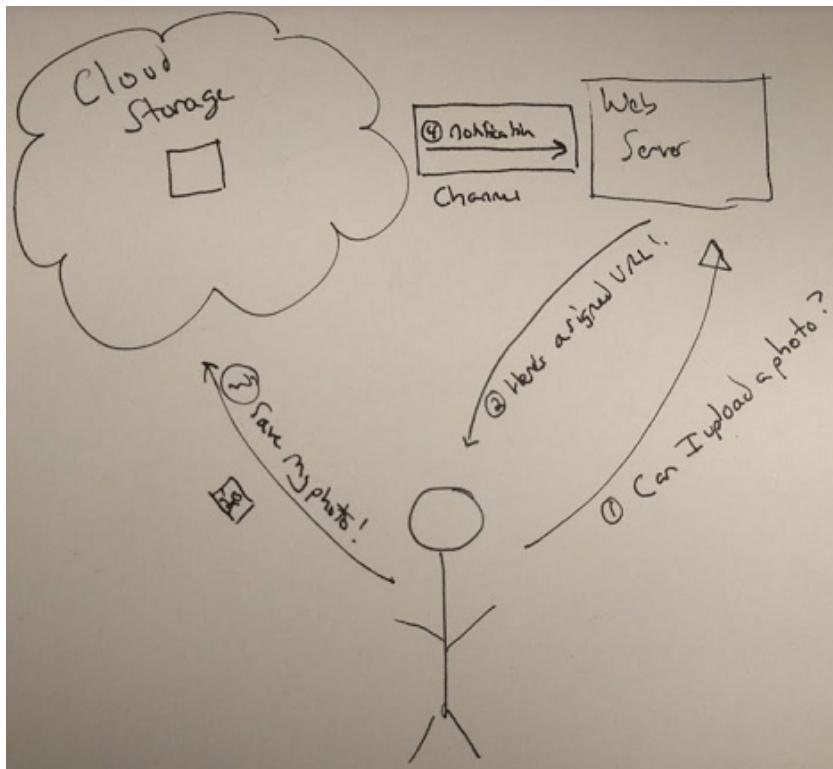
So far all of the interaction with Cloud Storage has been "pull", in other words the interaction was initiated by us contacting Cloud Storage, either uploading or downloading data. Wouldn't it be nice if we could use some of these features like access control policies and signed URLs to allow users to upload or update files and have Cloud Storage notify you when things happen? This is absolutely possible by setting up change notifications.

If you couldn't guess from the name, change notifications allow you to set a URL that will receive a notification whenever objects are created, updated, or deleted, and then you can do whatever other processing you might need based on the notification.

A common scenario for this is to have a bucket acting a bit like an "inbox" that accepts new files and then processes those files into a known location. For example, you might have a bucket called `incoming-photos` and whenever an image is uploaded you process the image into a bunch of different sizes as thumbnails and store those for use later on. This lends itself nicely to using signed URLs for allowing one-time passes to upload files into the incoming bucket.

This process works by setting up a notification "channel" which acts as the conduit between an event happening in your bucket and a notification being sent to your servers. This process would look something like the following:

1. A user sends a request to your web server for a signed URL effectively asking "Can I upload a file?"
2. The server should respond with a signed URL granting the user access to put a file into the bucket.
3. The user then uploads their image into the bucket
4. When that file is saved, a notification channel sends a request to let the server know that a new file has arrived.

**Figure 8.9. Common object notification flow**

Setting up a notification channel is actually pretty easy to do with the `gsutil` command-line tool. All you need to do is use the `watchbucket` sub-command and provide three pieces of information:

1. The URL that should be notified
2. The bucket that you want to watch
3. The ID for the channel that you're creating, which should be unique for the bucket.

Based on those things, setting up a watch command should look like:

#### **Listing 8.33. Configuring notifications using gsutil**

```
$ gsutil notification watchbucket -i channel-id https://mydomain.com/new-image
gs://my-bucket
```

In this example, the channel ID is `channel-id` and the bucket is `my-bucket`, which effectively says to send a request to the specified URL ([mydomain.com/new-image](https://mydomain.com/new-image)) whenever any changes happen inside `my-bucket`. You can also specify a channel token which acts as a unique "password" of sorts so you can be sure that any requests sent are actually from Google and not from somewhere else.

Once a channel is set up, you'll start to receive POST requests from Cloud Storage for the various events that occur in your bucket. These requests will have a variety of parameters that arrive in the form of HTTP headers.

**Table 8.7. Parameters in a notification request**

Header name	Meaning (example)
X-Goog-Channel-Id	The channel ID of the notification (e.g., channel-id)
X-Goog-Channel-Token	The token of the notification (e.g., my-secret-channel-token)
X-Goog-Resource-Id	The ID of the resource being modified (e.g., my-bucket/file.txt)
X-Goog-Resource-State	The "event" prompting this notification (e.g., sync, exists, not_exists)
X-Goog-Resource-Url	The URL corresponding to the resource ID (e.g., <a href="https://www.googleapis.com/storage/v1/b/BucketName/o/file.txt">www.googleapis.com/storage/v1/b/BucketName/o/file.txt</a> )

Corresponding to the X-Goog-Resource-State header, each state corresponds to a different "event", effectively saying what happened to trigger the event. There are only three distinct states, corresponding to four different events.

#### **Sync (sync)**

A sync event is the first event you'll receive, which happens when the notification channel is created. This serves to let you know that the channel is "open", so you can use it to initialize anything on the server side.

#### **Object deletions (not\_exists)**

Whenever an object is deleted, you'll get a request with a state saying not\_exists. It's less likely that you'll need this event but it's available nonetheless.

#### **Object creations and updates (exists)**

When an object is either created or updated you'll get an event with the resource state set to exists. Along with the headers you get in every request you'll also get the object metadata in the body of the request.

### **8.8.1 URL restrictions**

Unfortunately, when you try to run the command to watch a bucket with a custom URL (Listing 8.33) you'll find out that there are a few 'gotchas' about which URLs are allowed. Let's look briefly at why this is and how we can go about resolving any issues.

#### **SECURITY**

First, notice that in the example the URL starts with https and not http. It turns out that Google wants to make sure that no one is able to spy on changes happening in buckets, and so the notification URL must be at an encrypted endpoint. This means that no matter what URL you put in there, if it starts with http it will be rejected as invalid.

While this will certainly be frustrating when you're testing things out, thanks to the

wonderful people over at Let's Encrypt, setting up SSL certificates that "just work" is surprisingly easy. Take a look at [letsencrypt.org/getting-started](https://letsencrypt.org/getting-started) for a summary of how to get SSL set up for your system, which should take just a few minutes.

#### **WHITELISTED DOMAINS**

In addition to requiring that your notification endpoint is "secure", you also need to prove that you own a domain before using it as an endpoint for object change notifications. This is specifically to prevent you from using Google Cloud to make requests of servers you don't own (either intentionally or accidentally). For example, what would stop you from setting up loads of endpoints all pointing at [your-competitor.com/dos-attack](https://your-competitor.com/dos-attack)?

Regardless of your intentions, the point remains that we'll need to prove that we're authorized to send traffic to the domain before Cloud Storage will start sending notifications there, which means we have to "whitelist" it. You can whitelist a domain in a few different ways, but the easiest by far is to use Google Domains for managing your domain name. You can do this by registering or transferring a domain into [domains.google.com](https://domains.google.com).

If that isn't an option (which for many, it won't be), you can also prove ownership through Google Webmaster Central by setting a DNS record or special HTML meta-tag. To get started with this, visit [google.com/webmasters/tools](https://google.com/webmasters/tools) which will guide you through the process. Once your Google account is registered as an "owner" of the domain name, Cloud Storage will consider your domain to be whitelisted and your notification URL can use the domain name in question.

## **8.9 Common use-cases**

Now that you understand the building blocks common to object storage, let's spend a bit of time exploring some of the common use cases, specifically how you can put these building blocks together to do real life things such as hosting profile pictures, websites, or archiving your data in case of a disaster.

### **8.9.1 Hosting user content**

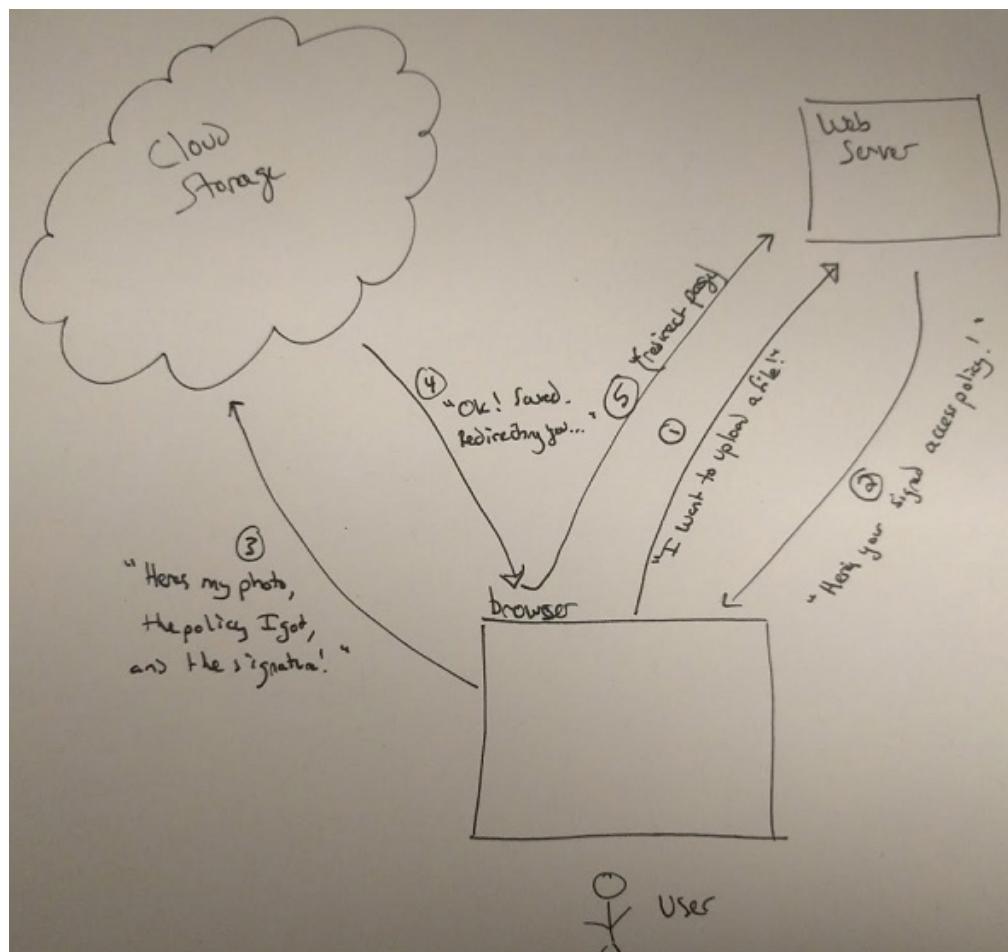
One of the most common scenarios is safely storing user content, such as profile photos, uploaded videos, or voice recordings. Using the concept of signed URLs, described in ["Signed URLs"](#), you can set up a pretty simple system for processing user-uploaded content, such as the photos stored in InstaSnap.

As you learned in the section about signing URLs, while you want to accept user-created content, you don't want to give anyone in the world general access to your Cloud Storage buckets as that could lead to some scary things (e.g., people deleting data or looking at data they shouldn't). However, it is wasteful of resources for users to first send their content to your server and asking your server to forward it along to Cloud Storage. Ideally, you'd just allow customers to send their content directly into your bucket — with a few limitations of course.

To accomplish this, Cloud Storage provides a way to create policy tokens which are kind of like "permission slips" that children get in school to attend outside functions. The way this works is you generate a "policy" document saying what a user can upload, and then digitally sign that policy and send the signature back to the user. For example, a policy might convey something like "this person can upload up a png image up to 5 MB in size".

Then the user uploads their content to Cloud Storage, and also passes along the policy and the signature of the policy. Cloud Storage then just checks that the signature is valid, and that the operation you're trying to do is covered by the policy. If it is, the operation completes. Let's look at this from a flow-diagram perspective.

**Figure 8.10. Uploading content using a policy signature**



The steps are:

1. The user should make a request to your web server asking "Can I upload?" (This

- could be when a user navigates to an "upload" page.)
2. The server should generate a policy and send it back (along with the signature).
  3. The user should send the content (e.g., an image) along with the policy and signature to Cloud Storage using a standard HTML <form>.
  4. Cloud Storage should accept and save the image and then redirect the user to another web page.

### **8.9.2 Data archival**

As you've heard a few times, specifically when we were discussing Nearline and Coldline storage, Cloud Storage can be a very cost-effective way to archive your data. Whether it's access logs, processed data, or old movies you've converted from DVDs, Cloud Storage just cares about making sure your data stays safe.

Given that archived data is much less frequently accessed, the Nearline and Coldline storage classes happen to be ideal options. This is primarily due to the fact that you won't often need to download this data, and therefore your bill at the end of the month will be much lower than if you'd chosen multi-regional storage. Let's look briefly at how you might use Cloud Storage to archive your logs.

Logs are usually plain text files that a running process (such as a web server) append to over time, and cycle to a new file name every so often (sometimes based on the size of the file, sometimes based on timestamps). With Cloud Storage your goal is to get those files off of your machine's persistent disk and into a Cloud Storage bucket. Typically, your logging system will package up your logs into a gzipped format when it makes the cut, so all you really need to do is set up a schedule task to upload the right files to your bucket.

For example, you can use the `gsutil` command's `rsync` functionality as part of your system's crontab to synchronize your MySQL logs to Cloud Storage every day at 3 AM. We're ignoring timezones for the purposes of this conversation.

#### **Listing 8.34. crontab entry to synchronize your logs with Cloud Storage**

```
0 3 * * * gsutil -m rsync /var/log/mysql gs://my-log-archive/mysql
```

This command will synchronize your local log files into a Google Cloud Storage bucket, which avoids re-uploading data that you've already saved and copies any newly created (or modified) files all in a single command! Now let's move on to see how pricing works for Cloud Storage.

## **8.10 Understanding pricing**

We've spent a whole lot of time understanding what Cloud Storage is, the features that it comes with, and how you put those features together to do real things. But how do you pay for it? And how much does it actually cost? Let's spend a bit of time walking through the different ways things cost money, and then we'll take a few common examples and look at how much each of these cost.

Cloud Storage pricing is broken down into several different components:

1. Amount of data stored
2. Amount of data transferred (also known as network traffic)
3. Number of operations executed (e.g., number of GET operations)

In addition to these, the Nearline and Coldline storage classes have two extra components which we'll discuss in more detail later on:

1. Amount of data retrieved (in addition to served)
2. 30-day (or 90-day) minimum storage

### **8.10.1 Amount of data stored**

Data storage is the simplest and most obvious component of your Cloud Storage bill, and should remind you of other storage providers like DropBox. Every month, Cloud Storage charges you based on the amount of data you keep around in your bucket measured in Gigabytes per month prorated based on how long the object was stored. This means that if you store an object for 15 out of 30 days, your bill for a single 2 GB object will be  $2 \text{ (GB)} * 0.026 \text{ (USD)} * 15/30 \text{ (months)}$  which is 31 cents. And if you store it for only 1 hour (1/24th of one day) out of a 31 day month, your data storage cost will be  $2 \text{ (GB)} * 0.026 \text{ (USD)} * (1/24) \text{ days} / 31 \text{ (days in the month)}$  which is effectively zero (0.000069892 USD). The data storage component gets even cheaper if you change to different storage classes such as Nearline or Coldline that we learned about before.

First, let's look at prices for the multi-region locations, which currently are the US, the EU, and Asia. These three locations allow multi-region, Nearline, and Coldline storage classes split across multiple regions inside the location.

**Table 8.8. Pricing by storage class in multi-region locations per GB stored**

Class	Price per GB per month
Multi-regional	2.6 cents (\$0.026)
Nearline	1 cent (\$0.01)
Coldline	0.7 cents (\$0.007)

For single region locations, only regional, Nearline, and Coldline storage classes are supported. As you might guess, the prices for these vary from one location to the next, shown below for a few common locations.

**Table 8.9. Pricing by storage class (and location) per GB stored**

Location	Regional	Nearline	Coldline
Oregon (US)	\$0.02	\$0.01	\$0.007
South Carolina (US)	\$0.02	\$0.01	\$0.007
London (UK)	\$0.023	\$0.016	\$0.013
Mumbai (India)	\$0.023	\$0.016	\$0.013

Singapore	\$0.02	\$0.01	\$0.007
Sydney (Australia)	\$0.023	\$0.016	\$0.013
Taiwan	\$0.02	\$0.01	\$0.007

As discussed before, these costs are strictly for the amount of data that you store in Cloud Storage. Any redundancy offered to provide high levels of durability are baked right into the regular price.

This storage cost might not seem like very much, but when you look at the cost for larger and larger amounts of data the cost differences can actually start to be material. Let's look at a quick summary of storing increasing amounts of data for one month in the different storage classes.

**Table 8.10. Monthly storage cost for different classes**

Class	10 GB	100 GB	1 TB	10 TB	100 TB	1 PB
Multi-regional	\$0.26	\$2.60	\$26.00	\$260.00	\$2,600.00	\$26,000.00
Regional (Iowa)	\$0.20	\$2.00	\$20.00	\$200.00	\$2,000.00	\$20,000.00
Nearline	\$0.10	\$1.00	\$10.00	\$100.00	\$1,000.00	\$10,000.00
Coldline	\$0.07	\$0.70	\$7.00	\$70.00	\$700.00	\$7,000.00

Notice that if you're storing large amounts of data (e.g., a petabyte), using a different storage class such as Nearline can be significantly cheaper than multi-regional for the data storage component of your bill.

**NOTE**

**Metadata is data too!**

In addition to storing your data, any metadata you store on your object will be counted as though it were part of the object itself.

This means that if you store an extra 64 characters in metadata, you should expect an extra 64 bytes of storage to appear on your bill.

But your data doesn't just sit still, it needs to be sent around the internet, so let's look at how much that costs.

### 8.10.2 Amount of data transferred

In addition to paying for data storage, you also will be charged for sending that data to customers or to yourself. This cost is sometimes called "network egress" which just refers to the amount of data that is being sent **out** of Google's network. For example, if you download a 1 MB file from your Cloud Storage bucket onto your office desktop, you'll be charged for egress network traffic at Google's normal rates.

Because networking is dependent on geography (that is, different places in the world have different amounts of network cable) so network costs will vary depending on where you are in the world. In Google's case, mainland China and Australia are the two regions in the world that currently cost more than everywhere else.

Additionally, as you send more data in a given month beyond a terabyte, you'll get a

reduced rate in the ballpark of 5% to 10%. In the following table you can see how the prices stack up, however it's most likely that an average user would fall into the first column (serving up to 1 TB of data per month), and if based in the US and targeting US-based customers, the last row will be the most common. This means that in the average US-focused case, network charges will come to 12 cents per Gigabyte served.

**Table 8.11. Egress network prices per GB**

Region	First TB / mo	Next 9 TB / mo	Beyond 10 TB / mo
China (not Hong Kong)	\$0.23	\$0.22	\$0.20
Australia	\$0.19	\$0.18	\$0.15
Anywhere else (e.g., the US)	\$0.12	\$0.11	\$0.08

To put this into context this means that if you download a 1 MB file from your Cloud Storage bucket to your office desktop in New York City, you'll be charged  $0.001 \text{ (GB)} * 0.12 \text{ (USD)}$  or \$0.00012 to download the file. If you download that same file 1,000 times, your total cost will come to  $1 \text{ (GB)} * 0.12 \text{ (USD)}$  or \$0.12. If you happen to go on vacation to Australia and do the same thing, your bill becomes  $1 \text{ (GB)} * 0.19 \text{ (USD)}$  or \$0.19. There is one big exception to this component of your bill: "in-network" traffic.

In Google Cloud, network traffic that stays inside the same **region** is free of charge. This means that if you create a bucket in the US and then transfer data from that bucket to your Compute Engine instance in the same region, you won't be charged anything for that network traffic. On the flip side, if you have data stored in a bucket in Asia and download it to a Compute Engine instance in `us-central1-a`, you'll actually be paying for that network traffic whereas downloading it to an instance in `asia-east1-c` would be free.

### 8.10.3 Number of operations executed

Lastly, in addition to charges that depend on the amount of data you're storing or sending over the internet, Cloud Storage charges for a certain subset of operations you might perform on your buckets or objects. Of the non-free operations, there are two different "classes": a "cheap" class (e.g., getting a single object) costing 1 cent for every 10,000 and an "expensive" class (e.g., updating an object's metadata) which costs 10 cents for every 10,000. A good way to think of whether an operation is one of the cheap ones or one of the expensive ones is to look at whether it modifies any data in Cloud Storage. If it is writing data, it's likely one of the expensive operations, though there are exceptions.

**Table 8.12. Types of operations**

Type	Cheap operations (\$0.01 per 10k)	Expensive operations (\$0.10 per 10k)
Read	<ul style="list-style-type: none"> <li>• *.get</li> <li>• *AccessControls.list</li> </ul>	<ul style="list-style-type: none"> <li>• buckets.list</li> <li>• objects.list</li> </ul>
Write	Any notifications sent to your callback URL	<ul style="list-style-type: none"> <li>• *.insert</li> <li>• *.patch</li> <li>• *.update</li> <li>• objects.compose</li> <li>• objects.copy</li> <li>• objects.rewrite</li> <li>• objects.watchAll</li> <li>• *AccessControls.delete</li> </ul>

Notice that a few operations are missing from these lists, which is because they are free. The free operations are (as you might expect) focused on deleting:

- channels.stop
- buckets.delete
- objects.delete

With that, let's move on and look in more detail at how Nearline and Coldline pricing works.

#### 8.10.4 Nearline and Coldline pricing

As mentioned in “[Nearline storage](#)” and “[Coldline storage](#)”, data in the Nearline and Coldline storage classes have a significantly cheaper data storage cost, however there can be drawbacks if the data is frequently accessed. In addition to the storage, network, and operations cost that you've learned about so far, Nearline and Coldline also include an extra cost for "data retrieval" which is currently \$0.01 USD per GB retrieved on Nearline and \$0.05 USD per GB for Coldline. This is sort of like an internal networking cost that applies no matter where your destination is, which means that even downloading inside the same region from Cloud Storage to a Compute Engine instance will cost \$0.01 or \$0.05 per GB retrieved.

This might seem strange but keep in mind that Nearline and Coldline were designed primarily for archival, so in exchange for making it much cheaper to store your data safely, these classes add back that per-GB amount only if you retrieve your data. To put this in a more quantitative context, storing your 1 GB in multi-regional storage (\$0.026 per month) is effectively the same cost as storing 1 GB in Nearline (\$0.01 per month) and accessing that 1 GB exactly 1.6 times every month (e.g., retrieving 1.6 GB throughout the month, costing \$0.016). This means that your "break-even" point for storage will depend on whether you retrieve 1.6 times the amount of data stored.

To really drive this point home, imagine that you have 10,000 user-uploaded images totaling 1 GB and need to decide where to store these images. Let's also imagine that you happen to be archiving these images and therefore plan to download all of these

images only once per year. Let's further make the assumption that the "download" will be to a Compute Engine instance in the same region, which lets us ignore network egress costs.

**Table 8.13. Pricing comparison (yearly access)**

Class	Storage	Retrieval	Total
Nearline	\$1.20 (= 10 GB * \$0.01 per GB per month * 12 months)	\$0.10 (= 10 GB * 1 download per year * \$0.01 per GB downloaded)	\$1.30
Coldline	\$0.84 (= 10 GB * \$0.007 per GB per month * 12 months)	\$0.50 (= 10 GB * 1 download per year * \$0.05 per GB downloaded)	\$1.34
Multi-regional	\$3.12 (= 10 GB * \$0.026 per GB per month * 12 months)	\$0.00 (= 10 GB * 1 download per year * \$0.00 per GB downloaded)	\$3.12

This means that if you only download your data once per year, you're going to pay less if you use Nearline to store your data.

If you happen to access your data frequently (e.g., each image is downloaded at least once per week), this changes the pricing dynamic quite a bit.

**Table 8.14. Pricing comparison (weekly access)**

Class	Storage	Retrieval	Total
Nearline	\$1.20 (= 10 GB * \$0.01 per GB per month * 12 months)	\$5.20 (= 10 GB * 52 downloads per year * \$0.01 per GB downloaded)	\$6.40
Coldline	\$0.84 (= 10 GB * \$0.007 per GB per month * 12 months)	\$26.00 (= 10 GB * 52 downloads per year * \$0.05 per GB downloaded)	\$26.84
Multi-regional	\$3.12 (= 10 GB * \$0.026 per GB per month * 12 months)	\$0.00 (= 10 GB * 52 downloads per year * \$0.00 per GB downloaded)	\$3.12

In this scenario, you'll end up paying around twice as much to use Nearline with the cost driven almost exclusively by the data retrieval cost.

If you happen to never need to access the data, we'll see how Coldline really shines, shown below.

**Table 8.15. Pricing comparison (no access)**

Class	Storage	Retrieval	Total
Nearline	\$1.20 (= 10 GB * \$0.01 per GB per month * 12 months)	\$0.00 (= 10 GB * 0 downloads per year * \$0.01 per GB downloaded)	\$1.20
Coldline	\$0.84 (= 10 GB * \$0.007 per GB per month * 12 months)	\$0.00 (= 10 GB * 0 downloads per year * \$0.05 per GB downloaded)	\$0.84
Multi-regional	\$3.12 (= 10 GB * \$0.026 per GB per month * 12 months)	\$0.00 (= 10 GB * 0 downloads per year * \$0.00 per GB downloaded)	\$3.12

Hopefully this gives you some insight into when Nearline or Coldline storage may be good choices for your system. When in doubt, if you're saving stuff "for a rainy day", Nearline might be better, depending on how often it rains. If you're using the data in

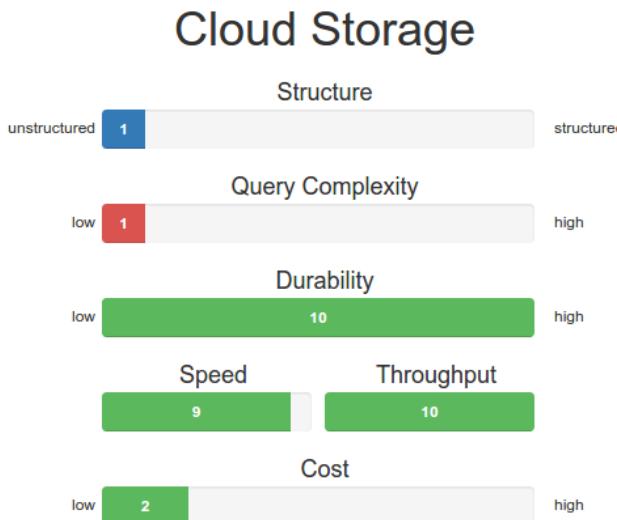
your application and serving it to users, multi-regional (or regional) storage is probably a better fit.

## 8.11 When should I use Cloud Storage?

Unlike our other storage systems, Cloud Storage is complementary to your system in more than one way. In a sense, using object storage is a bit more like a check box than one of the multiple choice options.

As a result, this section will summarize Cloud Storage briefly using the same scorecard as the other services, however this will focus more on how Cloud Storage complements your other storage systems rather than whether Cloud Storage is a good fit at all.

**Figure 8.11. Cloud Storage scorecard**



### 8.11.1 Structure

Cloud Storage is by definition an **unstructured** storage system and is therefore meant to be used purely as a key-value storage system with no ability to run any other queries besides "give me the object at this key".

While you can technically query Cloud Storage for a list of objects based on a prefix, that querying ability should be treated more as an administrative function and not something to be used as a feature in your application.

### 8.11.2 Query complexity

Due to the complete lack of structure and the pure key-value nature of Cloud Storage, there is simply no ability to run queries of any complexity at all. That said, you shouldn't be using Cloud Storage if you need to query your data!

### **8.11.3 Durability**

Durability happens to be an aspect where Cloud Storage really shines, offering a 99.99999999% durability guarantee (that's 11 total 9s). Even with the cheaper options (Nearline or Coldline), your data is always automatically replicated in several different places by default. This means that your data is basically as safe as you can possibly make it, on par with Cloud Datastore or Persistent Disks in Compute Engine.

Under the hood this is done using erasure coding, which is a form of error correction, that chops data up into lots of pieces and stores that data redundantly on lots of different disks spread out across lots of different failure domains (considering both network failure and power failure). This means, for example, that even if two different disks fail with your data on them your data is still safe and hasn't been lost.

### **8.11.4 Speed (latency)**

This is one area where Cloud Storage allows you to choose what type of latency you expect for your application. By default, multi-regional storage is sufficiently fast to bring the latency (measured as "time to the first byte") into the milliseconds. If you happen to be less interested in first-byte latency and more interested in saving money, you can choose either Nearline or Coldline storage if you are dealing more with archival or infrequently accessed data.

In other words, if you need the speed it's there. If you don't, you can save some money.

### **8.11.5 Throughput**

This is the area where Cloud Storage really shines. Since Cloud Storage is optimized for throughput, you effectively can treat it as a "never ending resource" for throughput. Under the hood it is obviously not an infinite resource for throughput — after all, there's only so much network cable in the world — but Google automatically manages capacity on a global scale to make sure that you never get stuck in need of a faster download.

### **8.11.6 Overall**

As mentioned before, instead of focusing on the typical storage needs of each application and seeing how this service stacks up, this section will focus on the ways that each application can use Cloud Storage and see how good of a fit it is.

#### **To-do list**

The to-do list will probably not have all that much use for Cloud Storage specifically because most of the data stored is textual rather than binary. However, if your to-do list wants to support image uploads, Cloud Storage is a great place to put that data.

**Table 8.16. To-do list use for storage classes**

Storage class	Use case
Multi-regional	Storing customer image uploads (e.g., profile pictures)
Regional	Storing larger customer attachments (e.g., Excel files)
Nearline	Archiving database back-ups
Coldline	Archiving request logs

Given the to-do list is serving customer data, you'll most likely want to use the multi-regional storage class for your bucket. If you're really trying to pinch pennies, regional could technically work, however we don't really know where users will be, meaning some of them that happen to be far away from the data may see worse overall performance.

#### **E\*Exchange**

E\*Exchange is much less likely to need attachments, but might still need to store trading history in an archive, or tax documents as PDFs. In these cases, the best choice will likely be multi-regional for user-facing downloads, Nearline for trading reports, and Coldline for trade logs in case there is annual audit by the SEC. If the exchange happens to run some analysis over trading data, since we know where the computation will happen, regional storage may be a good choice here.

**Table 8.17. E\*Exchange storage needs**

Storage class	Use case
Multi-regional	Customer tax documents in PDF form
Regional	Data analysis jobs
Nearline	Customer trading reports
Coldline	System-wide audit logs

#### **InstaSnap**

InstaSnap is as user-facing an app you can find, which also happens to be focused mostly on customer uploaded images, with image latency being pretty important. Because of this, multi-regional storage with the lowest latency and highest availability is likely the right choice. While you also might want to archive database back-ups using Nearline and user access logs using Coldline.

**Table 8.18. InstaSnap storage needs**

Storage class	Use case
Multi-regional	Storing customer uploaded images
Regional	No obvious use case
Nearline	Weekly database back-ups
Coldline	Archive user-access logs

## 8.12 Summary

- Google Cloud Storage is an object storage system, meaning it allows you to store arbitrary chunks of bytes ("objects") without worry about disk drives, replication, etc.
- Cloud Storage has several different storage classes available, each with its own trade-offs (e.g., lower cost for lower availability).
- Although Cloud Storage is mainly about storing chunks of data, it also provides extra features like automatic deletion for old data ("lifecycle management"), storing multiple versions of data, advanced access control (using ACLs), and notification of changes to objects and buckets.
- Unlike other storage systems we've learned about, Cloud Storage complements the others and as a result, is typically used *in addition to* those rather than *instead of* them.

# Part 3 Computing

Now that we've gone through lots of ways to store data, it's time to think about the various "computing" options we can use to interact with that data.

Similar to storage systems, there are quite a few different computing options available, each with their own benefits and drawbacks. Additionally, each of these options allows you to express the computational work to be done using different layers of abstraction, from the lowest level (working with a virtual machine) all the way up to a single JavaScript function running "in the cloud".

In this part of the book, we'll look at the various computing environments and dig down into how they all work. Some of these might feel very familiar if you've worked with any sort of server before (e.g., [chapter 9](#) which just hands you a virtual server) while others might seem pretty foreign (e.g., [chapter 11](#) which is a full-featured hosting environment), but it's important to understand the differences in order to make an informed decision when it comes time to build your next project.

And finally, as an added bonus, in [chapter 13](#) we'll explore how you can use Cloud DNS to give human-readable names to all the computing resources you end up creating over time.



# Compute Engine: Virtual machines

## **This chapter covers:**

- What are virtual machines (VMs)?
- How to use persistent storage with virtual machines
- The different types of persistent storage
- How auto-scaling works
- How to spread traffic across multiple machines with a load balancer
- Compute Engine's pricing structure

## **9.1 What are virtual machines?**

As we've learned, virtual machines are simply chopped up pieces of a single physical system that are then shared between several people. This is not really a new idea (even 10 years ago this was how "virtual private servers" were sold), but the idea has certainly gotten more advanced with Cloud hosting platforms like Google Cloud Platform or Amazon Web Services. For example, it's now possible to decouple the virtual machine from the physical machine, which means that machines can be taken offline for maintenance while the virtual machine is "live migrated" elsewhere, all without any downtime or significant changes to performance.

Advances like these enable even more neat features like automatic scaling, where the hosting provider can automatically provision more or fewer virtual machines based on incoming traffic or CPU usage, but these features can sometimes be tricky to understand and configure. As a result, the goal of this chapter is to get you comfortable with virtual machines, explain some of the interesting performance characteristics

(particularly those that might seem counter intuitive), and walk you through the more advanced features (like automatic scaling).

Keep in mind, Compute Engine is an enormous system with almost 40 different API resources, meaning it'd be possible to write an entire book on *just* Compute Engine. Since that would be too long, we'll instead focus on the most common and useful things you can do with Compute Engine, and dive deep into the details on a few of those where necessary. Let's dive right in by creating a virtual machine on Google Compute Engine (GCE).

## 9.2 Launching your first (or second) VM

Since we already learned how to launch a VM from the Cloud Console (see [chapter 2](#)), let's try launching one from the command line using `gcloud`.

**NOTE** If you don't have `gcloud` installed yet, check out [cloud.google.com/sdk](https://cloud.google.com/sdk) for instructions on how to get set up.

The first thing you'll need to do is authenticate using `gcloud auth login`. Then you should make sure you have your project set as default by using `gcloud config set project your-project-id-here`. After that, you can create a new instance in the `us-central1-a` zone, and connect to it using the `gcloud compute ssh` command, both shown below.

### Listing 9.1. Creating and SSH'ing to a new instance

```
$ gcloud compute instances create test-instance-1 --zone us-central1-a ①
Created [https://www.googleapis.com/compute/v1/projects/your-project-id-
here/zones/us-central1-a/instances/test-instance-1].
NAME          ZONE          MACHINE_TYPE  PREEMPTIBLE  INTERNAL_IP
EXTERNAL_IP    STATUS
test-instance-1  us-central1-a  n1-standard-1           10.240.0.13
104.197.64.184  RUNNING

$ gcloud compute ssh --zone us-central1-a test-instance-1 ②
Warning: Permanently added 'compute.1446186297696272700' (ECDSA) to the list of
known hosts.

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
jjg@test-instance-1:~$
```

- ① First, create the new instance.
- ② Then, connect to the instance over SSH.

If this all seems a bit too easy, that's kind of the point. The goal of cloud computing generally is to simplify physical infrastructure so that you can focus on building your

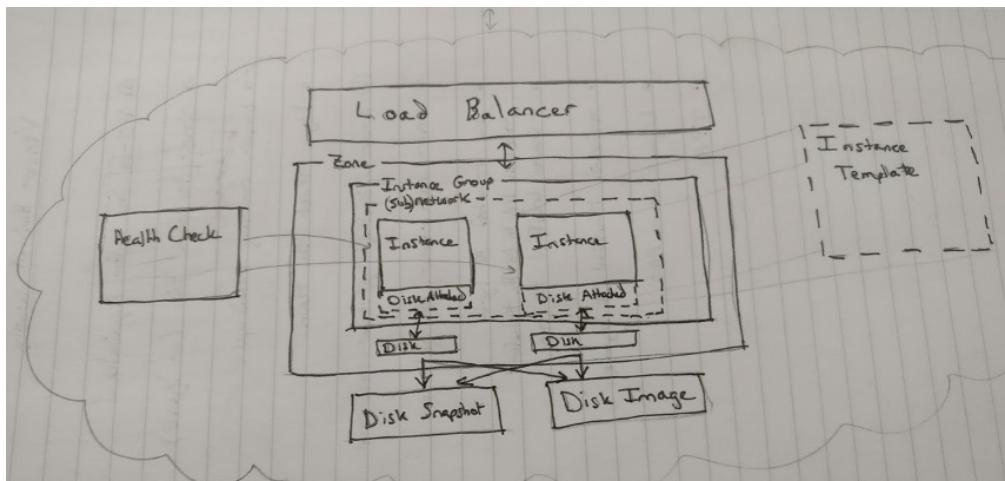
software rather than deal with the hardware that it runs on. That said, there's far more to Compute Engine than turning on VMs.

Let's take a look at an overview of all of the different components in a fully auto-scaling system built using Compute Engine, capable of expanding or contracting (e.g., creating VMs or destroying VMs) based on the load on the system at any given time.

**NOTE**

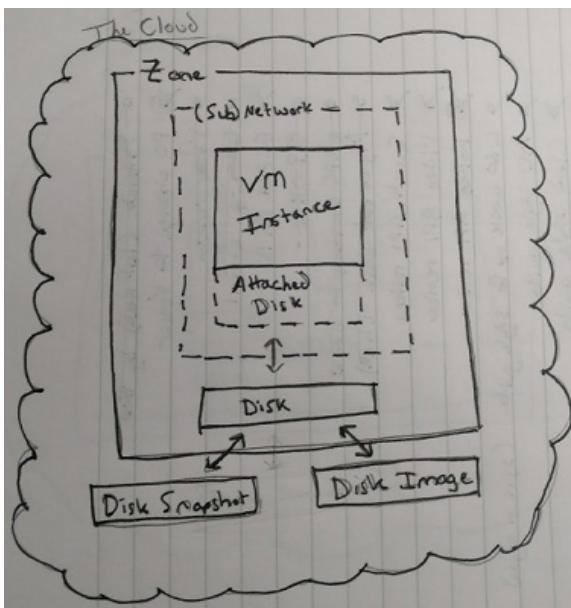
Don't worry! This is supposed to look scary at first! We'll walk through all of these pieces and at the end of the chapter you should understand all of them!

**Figure 9.1. A complete overview of Compute Engine**



As you can see there is quite a lot going on in this diagram. Why on earth do you need so much stuff? The short version is: **you don't!** If all you need is a simple VM that you can SSH into and run a server or two, you've learned all you need to learn. The longer version is that you may eventually want to do more advanced things like customize your virtual machines or balance server requests across a set of many machines. Google Compute Engine gives you ways to do all of these things, but they're a bit more complicated than typing a single `gcloud` command, so there are a few concepts you need to understand first. Let's move on to the next phase of customizing your deployment by starting with a simplified version of the scary diagram which is much easier to digest, shown below.

**Figure 9.2. A simpler overview of Compute Engine**



### 9.3 Block storage with Persistent Disks

As we discussed briefly in [part 1](#), a persistent disk is sort of a bit like an external hard drive. You can get a disk in varying sizes (e.g., 100 GB or 1 TB) and plug it into any computer nearby to see the data stored on them. This might sound like a basic *must-have* thing, but originally this storage was entirely *ephemeral*. This meant that whenever you restarted a machine, all of the data stored on the local disk would be completely gone, which could be anywhere from frustrating to dangerous.

To fix the issue cloud hosting providers came up with a storage service that looked and acted like a regular disk, but was replicated and highly available, in Google's case this is called Persistent Disk. Let's look a bit further into the details of how these disks work.

#### 9.3.1 Disks as resources

So far we've only dealt with disks as part of creating virtual machines, but this doesn't have to be the case. Disks themselves can be created and managed separately from VMs, and they can even be attached and detached from instances while they're running! In other words, we've only ever looked at disks when they were "attached" to a VM, but disks can be in many other states. Let's go through the life cycle of a disk and all of the things you can do with them.

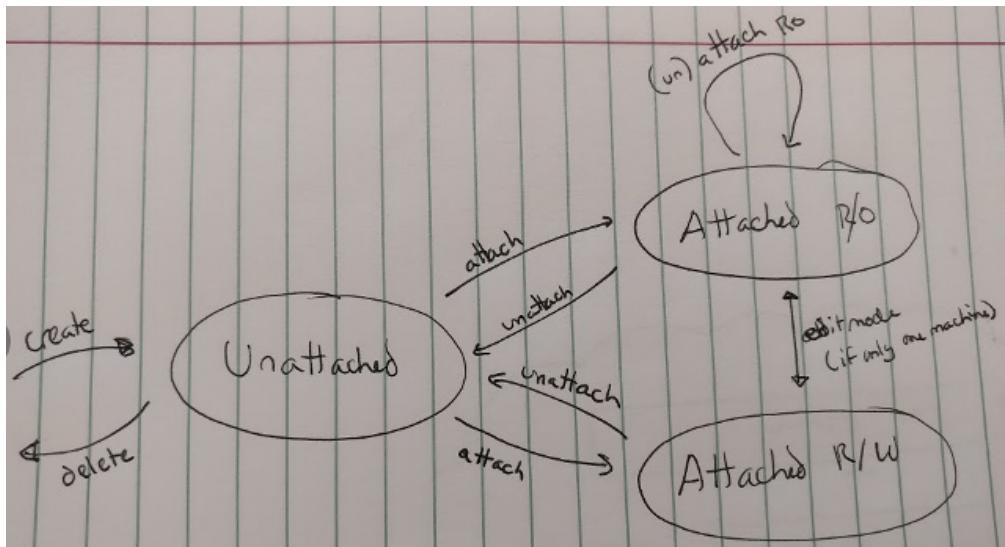
At any point in time, a persistent disk can be in one of three different states:

1. Unattached, where you've created the disk, but it's not "mounted" on any VMs.
2. Attached in "read-only" mode, where the VM can only read from the disk.

3. Attached in "read-write" mode, where the VM can both read **and** write to the disk.

So far we've only really talked about disks in the "Attached (read-write)" state, since that has been the default when creating a VM, so let's explore how all these states work and how we transition between them.

**Figure 9.3. Disk states and transitions**



Let's start by looking at Figure 9.3 which shows how you can transition between the various disk states. As you can see, the default value when creating a disk in Compute Engine is actually the "Unattached" state, which means the disk exists but is not in use by any VMs. You might think of this disk as "archived" somewhere ready for use sometime later.

You can attach disks to a VM in multiple modes (read-only and read-write) which are pretty self-explanatory, however it's worth noting that the "read-write" mode is exclusive, while "read-only" mode is not. This means that you can attach a single disk in read-only mode to as many VMs as you want, but if a disk is attached in read-write mode to a VM, it can't be attached to any other VMs, *regardless of the mode*.

Now that you have a grasp of the disk states and the rules for attaching them, let's explore how you actually go about doing this.

### 9.3.2 Attaching and detaching disks

To make these ideas a bit more concrete let's actually create a disk and take it through the different states. To do this, we're going to assume you have two VMs that exist already, called `instance-1` and `instance-2`. The details of the VMs aren't really important, so don't get hung up on those:

**Listing 9.2. The instances we have before starting**

\$ gcloud compute instances list						
NAME	ZONE	MACHINE_TYPE	PREEMPTIBLE	INTERNAL_IP	EXTERNAL_IP	STATUS
instance-1	us-central1-a	g1-small		10.240.0.3	104.197.251.111	RUNNING
instance-2	us-central1-a	g1-small		10.240.0.4	104.197.46.97	RUNNING

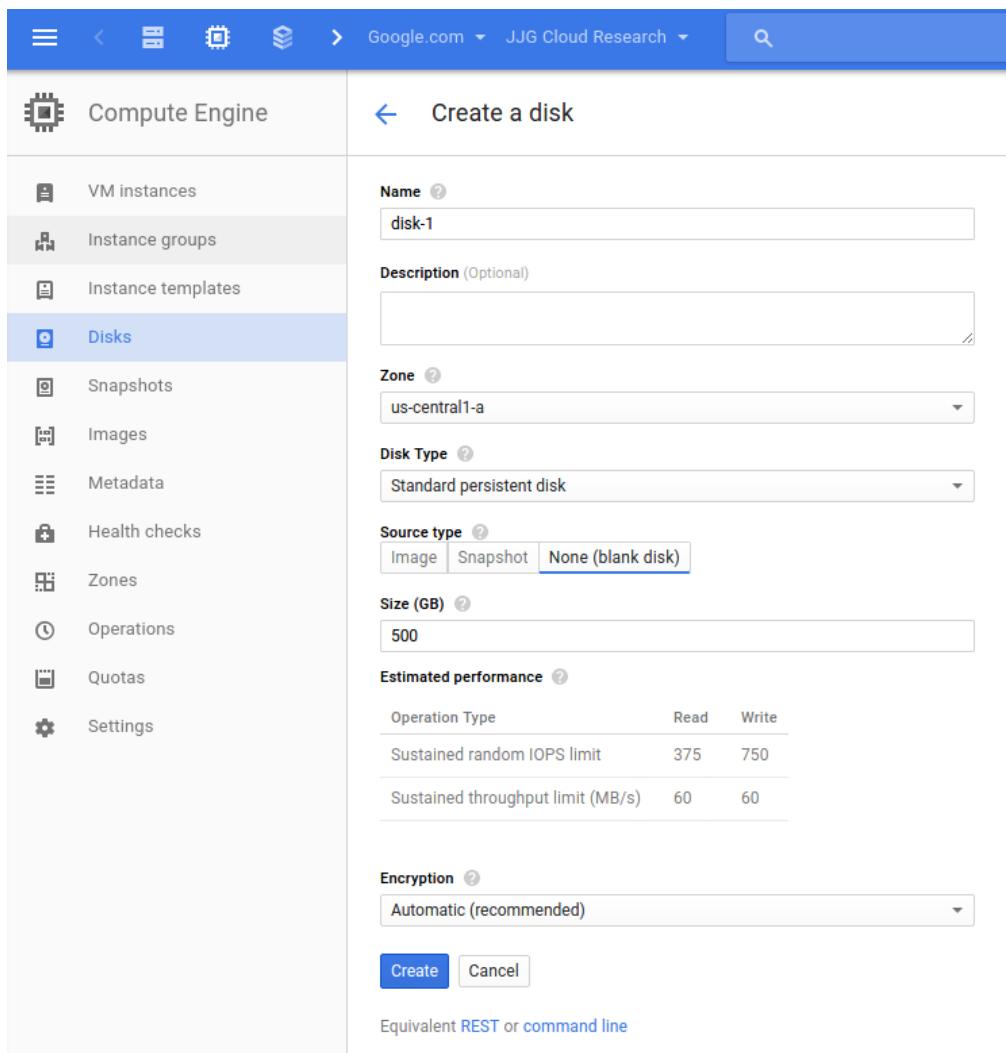
In the Cloud Console, jump into the Compute Engine section and choose Disks in the left side navigation. Then click "Create Disk" which will bring to a page that should feel pretty familiar. The first thing that should jump out at you is that you need to choose a name for your disk, which, just like a VM instance, must be unique. Also just like a VM, disks live inside specific zones, which means that the uniqueness of the name is specific to a single zone.

**TIP**

While you can have two disks with the same name in different zones, it's generally not a good idea as it's pretty easy to mix them up.

When it comes to choosing a location, remember that in order to be attached to an instance a disk must live in the same zone as that instance (otherwise there would be the risk of latency spikes when accessing data). Next, we'll need to choose a disk "type" which is focused mainly on performance, with "standard" disks acting a lot like traditional hard drives and SSD disks acting like solid-state drives. The right choice will depend on your access patterns, where SSDs have much faster random operations and traditional drives are adequate for large sequential operations. After that, leave the "Source type" as "None (blank disk)" to create an empty disk resource and choose any size you want for the disk (for example, 500 GB).

As we discussed in [chapter 2](#), disk size and performance are directly related such that larger disks can handle more "operations per second" (IOPS). This means that it's typical that applications which don't store a lot of data but have heavy access patterns (lots of reads and writes) will provision a larger disk not for the storage capacity but for the performance characteristics. You'll also notice that as you enter a size (in GB) you can see the estimated performance just below the field. Finally there's a field to choose what type of encryption to use. For now, let's leave this as is, and we'll discuss disk encryption later on.

**Figure 9.4. Creating your disk**

Now you can use the command-line to look at your disks just like you did with looking at the running instances.

**Listing 9.3. Listing your disks**

```
$ gcloud compute disks list
NAME      ZONE      SIZE_GB  TYPE      STATUS
disk-1    us-central1-a  500    pd-standard  READY
instance-1  us-central1-a  10    pd-standard  READY
instance-2  us-central1-a  10    pd-standard  READY
```

If you're wondering why there are 3 disks listed instead of just one, remember that

when you create an instance Compute Engine has to create a disk also, and it comes with a preset value of 10 GB for the total storage size. The disks that we're seeing here (`instance-1` and `instance-2`) are just the disks that were automatically created when we turned on our instances.

Now that you have a newly created disk, what can you do with it? Using the state diagram shown in Figure 9.3, this disk is in the "unattached" state. Let's move it through the other states by first attaching it as a "read-only" disk to `instance-1` and then also to `instance-2`. To start, let's go back to the Cloud Console and look at `instance-1`. If you scroll down the page a bit you'll see a section called "Additional disks" which has the very informative statement "None" listed there.

**Figure 9.5. No additional disks**

Boot disk and local disks			
Name	Size (GB)	Type	Mode
instance-1	10	Standard persistent disk	Boot, read/write
<input checked="" type="checkbox"/> Delete boot disk when instance is deleted			
<b>Additional disks</b>			
None			
<b>Network</b>			
default			
<b>Firewalls</b>			

To attach our disk to this instance, choose "Edit" from the top of the page, and then click "+ Add item" under that "Additional disks" heading. You can choose your new disk (`disk-1`) from the list, but be certain you choose to attach it in "Read only" mode! Then scroll to the bottom and click "Save" and you should see `disk-1` is attached to your instance!

**Figure 9.6. Attach an additional disk**

Additional disks (Optional)

Name	Mode	When deleting instance
disk-1	Read only	Keep disk

**+ Add item**

Now your disk is in the "attached read-only" state, which means that it can continue to be attached to other VMs, but if you were to try to write to this disk from `instance-1` the operation would fail with an error. This comes in handy when you have information on a persistent disk that you don't intend to modify from a VM, as it prevents disasters if you accidentally run a script or type `rm -rf` in the wrong place.

Let's now attach that same disk to `instance-2`, again in read-only mode, and this time let's do this with the `attach-disk` sub-command. Before we do that though, let's try attaching our disk to `instance-2` in read-write mode, to see that it throws an error.

#### **Listing 9.4. Attaching our disk again**

```
$ gcloud compute instances attach-disk instance-2 --zone us-central1-a \
--disk disk-1 --mode rw ①
ERROR: (gcloud.compute.instances.attach-disk) Some requests did not succeed:
- The disk resource 'disk-1' is already being used by 'instance-1'

$ gcloud compute instances attach-disk instance-2 --zone us-central1-a \
--disk disk-1 --mode ro ②
Updated [https://www.googleapis.com/compute/v1/projects/your-project-id-
here/zones/us-central1-a/instances/instance-2].
```

- ① Attaching the disk in read-write mode fails because it's already attached elsewhere.
- ② Attaching the disk in read-only mode succeeds as expected.

After this, if you go back to the Cloud Console and look at your list of disks, you'll see that `disk-1` is "in use" by both `instance-1` and `instance-2`. So now that disks are attached, how do we start saving data on them?

### **9.3.3 Using your disks**

So far we've looked at creating and managing disks, but haven't actually read or written any data from them. It turns out that under the hood when you attach a disk to an instance, it's kind of like plugging your external hard drive into the VM. Just like any brand-new drive, before we can do anything else we first have to mount the disk device and then format it. In Ubuntu this can be done with the `mount` command as well as calling the `mkfs.ext4` shortcut to format the disk with the `ext4` file system.

First, we have to get our disk into "read-write" mode, which we can do by detaching the disk from both instances and then reattaching the disk to `instance-1` to in read-write mode, shown below.

#### **Listing 9.5. Attaching the disk in read-write mode**

```
$ gcloud compute instances detach-disk instance-1 --zone us-central1-a \
--disk disk-1 ①
$ gcloud compute instances detach-disk instance-2 --zone us-central1-a \
--disk disk-1
$ gcloud compute instances attach-disk instance-1 --zone us-central1-a \
--disk disk-1 --mode rw ②
```

- ① First, detach the disk from both instances.
- ② Next, we reattach the disk in read-write mode.

Once you have `disk-1` attached to `instance-1` **only**, and in "read-write" mode, let's SSH into `instance-1` and look at the disks, which are conveniently located

in `/dev/disk/by-id/`. Shown below, we can see that `disk-1` has a friendly alias as `/dev/disk/by-id/google-disk-1` which you can use to point at the Linux device.

#### **Listing 9.6. Listing attached disks in Ubuntu**

```
jjg@instance-1:~$ ls -l /dev/disk/by-id
total 0
lrwxrwxrwx 1 root root 9 Sep  5 19:48 google-disk-1 -> ../../sdb
lrwxrwxrwx 1 root root 9 Aug 31 11:36 google-instance-1 -> ../../sda
lrwxrwxrwx 1 root root 10 Aug 31 11:36 google-instance-1-part1 -> ../../sda1
lrwxrwxrwx 1 root root 9 Sep  5 19:48 scsi-0Google_PersistentDisk_disk-1 ->
../../sdb
lrwxrwxrwx 1 root root 9 Aug 31 11:36 scsi-0Google_PersistentDisk_instance-1 ->
../../sda
lrwxrwxrwx 1 root root 10 Aug 31 11:36 scsi-0Google_PersistentDisk_instance-1-part1
-> ../../sda1
```

**WARNING**

Don't run this against a disk that has data on it as it **deletes** all data on the disk!

Let's start by formatting the disk using its device ID (`/dev/disk/by-id/google-disk-1`). In the example below we'll use some "extended" options (passed in via the `-E` flag) that are recommended by the disk team at Google. Once the disk is formatted, we'll mount it just like any other hard drive.

#### **Listing 9.7. Formatting and mounting the disk**

```
jjg@instance-1:~$ sudo mkfs.ext4 -F -E
lazy_itable_init=0,lazy_journal_init=0,discard \
/dev/disk/by-id/google-disk-1
mke2fs 1.42.12 (29-Aug-2014)
Discarding device blocks: done
Creating filesystem with 144179200 4k blocks and 36044800 inodes
Filesystem UUID: 37d0454e-e53f-49ab-98fe-dbed97a9d2c4
Superblock backups stored on blocks:
      32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208,
      409600, 7962624, 11239424, 20480000, 23887872, 71663616, 78675968,
      102400000

Allocating group tables: done
Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done

jjg@instance-1:~$ sudo mkdir -p /mnt/disks/disk-1
jjg@instance-1:~$ sudo mount -o discard,defaults /dev/disk/by-id/google-disk-1
/mnt/disks/disk-1
```

At this point the disk is ready, but it's still owned by `root` which could be irritating, so it may be worthwhile to change the owner to yourself (shown below). This means we can start writing data to the disk just like we would on a regular desktop.

**Listing 9.8. Setting disk ownership and writing data**

```

jjg@instance-1:~$ cd /mnt/disks/disk-1
jjg@instance-1:/mnt/disks/disk-1$ sudo mkdir workspace
jjg@instance-1:/mnt/disks/disk-1$ sudo chown jjg:jjg workspace/

jjg@instance-1:/mnt/disks/disk-1$ cd workspace
jjg@instance-1:/mnt/disks/disk-1/workspace$ echo "This is a test" > test.txt
jjg@instance-1:/mnt/disks/disk-1/workspace$ ls -l
total 4
-rw-r--r-- 1 jjg jjg 15 Sep 12 12:43 test.txt
jjg@instance-1:/mnt/disks/disk-1/workspace$ cat test.txt
This is a test

```

Now that we've seen how to interact with this disk, let's look at a common problem: running out of space.

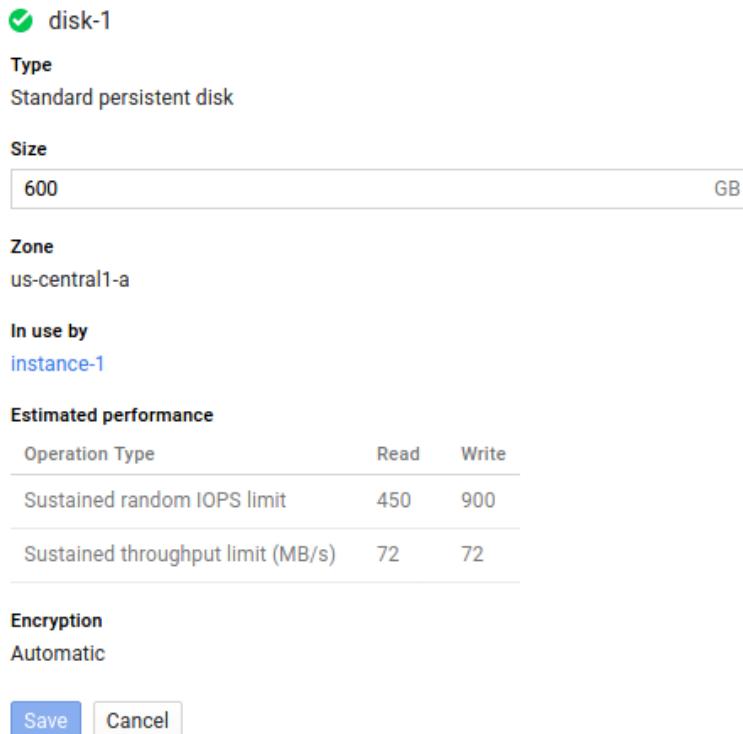
### **9.3.4 Resizing disks**

There are a variety of reasons we might want to resize a disk. In addition to running out of space, you might recall how the size of the disk is directly correlated to the "speed" of the disk. In other words, the bigger the disk, the faster it is. So how do we resize the disk itself?

First, you have to increase the size of the virtual disk in the Cloud Console by clicking "Edit" on the disk, and then typing in the new size.

**WARNING**

Keep in mind that you can always make a disk larger by increasing the size, but you can't make a disk **smaller**. This means that you should be particularly careful when increasing the size of your disk, since it will cost more money and be time consuming to undo.

**Figure 9.7. Resizing the disk in the Cloud Console**

Once you've done that, you have to resize your file system (in the example above this was the ext4 file system) to fill up the newly allotted space. To do this, you can use the `resize2fs` command on an unmounted disk.

**Listing 9.9. Resizing your file system**

```
jjg@instance-1:~$ sudo umount /mnt/disks/disk-1/ ❶
jjg@instance-1:~$ sudo e2fsck -f /dev/disk/by-id/google-disk-1
e2fsck 1.42.12 (29-Aug-2014)
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
/dev/disk/by-id/google-disk-1: 13/36044800 files (0.0% non-contiguous),
2312826/144179200 blocks
jjg@instance-1:~$ sudo resize2fs /dev/disk/by-id/google-disk-1
resize2fs 1.42.12 (29-Aug-2014)
Resizing the filesystem on /dev/disk/by-id/google-disk-1 to 157286400 (4k) blocks.
The filesystem on /dev/disk/by-id/google-disk-1 is now 157286400 (4k) blocks long.
```

- ❶ If you get an error about the target being busy, make sure you're not doing anything with the disk, and then wait a bit. This could be a background process preventing it from being unmounted.

Now we can re-mount the disk, and we should see that the disk has "expanded" to fill the available space on the virtual device:

#### **Listing 9.10. Remounting the disk after it's been "expanded"**

```
jjg@instance-1:~$ sudo mount -o discard,defaults /dev/disk/by-id/google-disk-1
/mnt/disks/disk-1
jjg@instance-1:~$ df -h | grep disk-1
/dev/sdb      591G   70M  561G  1% /mnt/disks/disk-1
```

Now that you've seen how to manage disks, let's explore some of the aspects of disk that are unique to virtualized devices like Google's Persistent Disks, starting with the concept of "disk snapshots".

### **9.3.5 Snapshots**

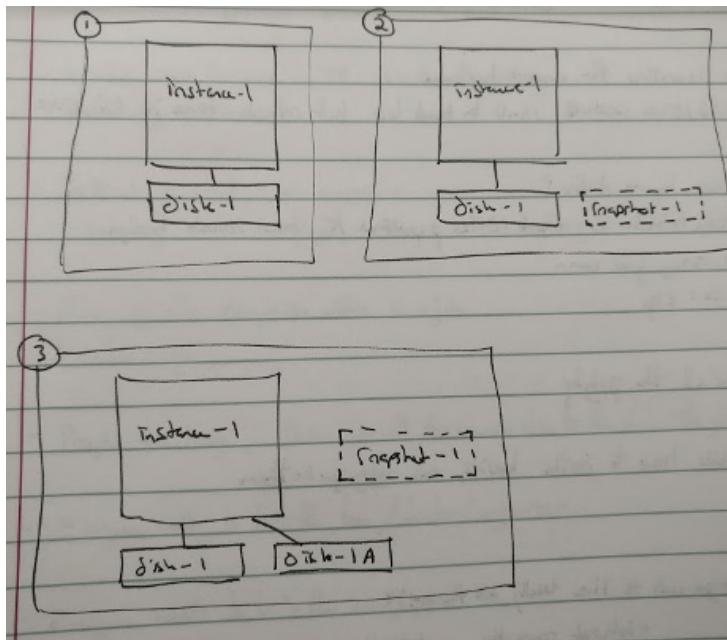
Have you ever wanted to "freeze" your computer at a point in time, and be able to jump right to that checkpoint? Maybe because you accidentally deleted a file? While snapshots weren't intended solely for those "oops" moments, they can act as those "checkpoints" of the data on your disk, allowing you to jump around in time by restoring a snapshot to a disk instance. And because snapshots act like checkpoints rather than copies of your disk, And since snapshots are like checkpoints that point at a specific version of a disk, they end up costing you much less than a full back-up.

This works because snapshots use **differential storage**, effectively storing only what's changed from one snapshot to the next. For example, this means that if you create a snapshot, change one block of data, and then take another snapshot, the second snapshot will only store the difference (or *delta*) between those two (in this case, only the one block) rather than an entire copy.

Storage savings aside, snapshots act mostly like regular disks in that you can create and delete them at any time, but unlike a regular disk, you can't read or write from it directly. Instead, once you have a snapshot of a disk, you can create a new disk based on the content from the snapshot. This comes in very handy when you want to snapshot your database's disk, or any of your servers so that you can restore to that point in time later on.

To see how this works, let's do a quick experiment using `disk-1` from above that walks through the lifecycle of a snapshot in the following steps:

1. Snapshot `disk-1` for later.
2. Change some data on the mounted copy of `disk-1`
3. Create a new disk instance based on our snapshot and mount it to our VM

**Figure 9.8. Visualizing our experiment**

By doing this, we'll effectively have two "versions" of `disk-1` attached to the VM, one of the disks will be the "old" version reflecting the snapshot from step 1, and the other the "current" version with the data we modified in step 2. Let's start by taking a snapshot of `disk-1`. Start by looking at the list of disks and clicking on `disk-1`. Then click "Create Snapshot" at the top of the page, which should bring you to a form to create a new snapshot from your disk.

**Figure 9.9. Creating a new snapshot**

[←](#) Create a snapshot

---

Name ?

Description (Optional)

Source disk ?

Encryption ?

Integrate volume shadow copy service ?  
 Enable VSS

[Create](#) [Cancel](#)

Equivalent [REST](#) or [command line](#)

Click "Create" and wait a few seconds, and you'll be sent over to a page listing out the snapshots in your project.

**Figure 9.10. A list of your snapshots**

Snapshots		<a href="#"> CREATE SNAPSHOT</a>	<a href="#"> REFRESH</a>	<a href="#"> DELETE</a>
<hr/>				
<input type="checkbox"/>	Name ^	Source disk	Creation time	Disk size
<input checked="checked" type="checkbox"/>	disk-1	disk-1	Sep 12, 2016, 5:44:21 PM	600 GB
				Snapshot size
				3.25 MB

Now that we have a new snapshot, let's change some data on our current disk-1.

**Listing 9.11. Change some of the data on disk-1**

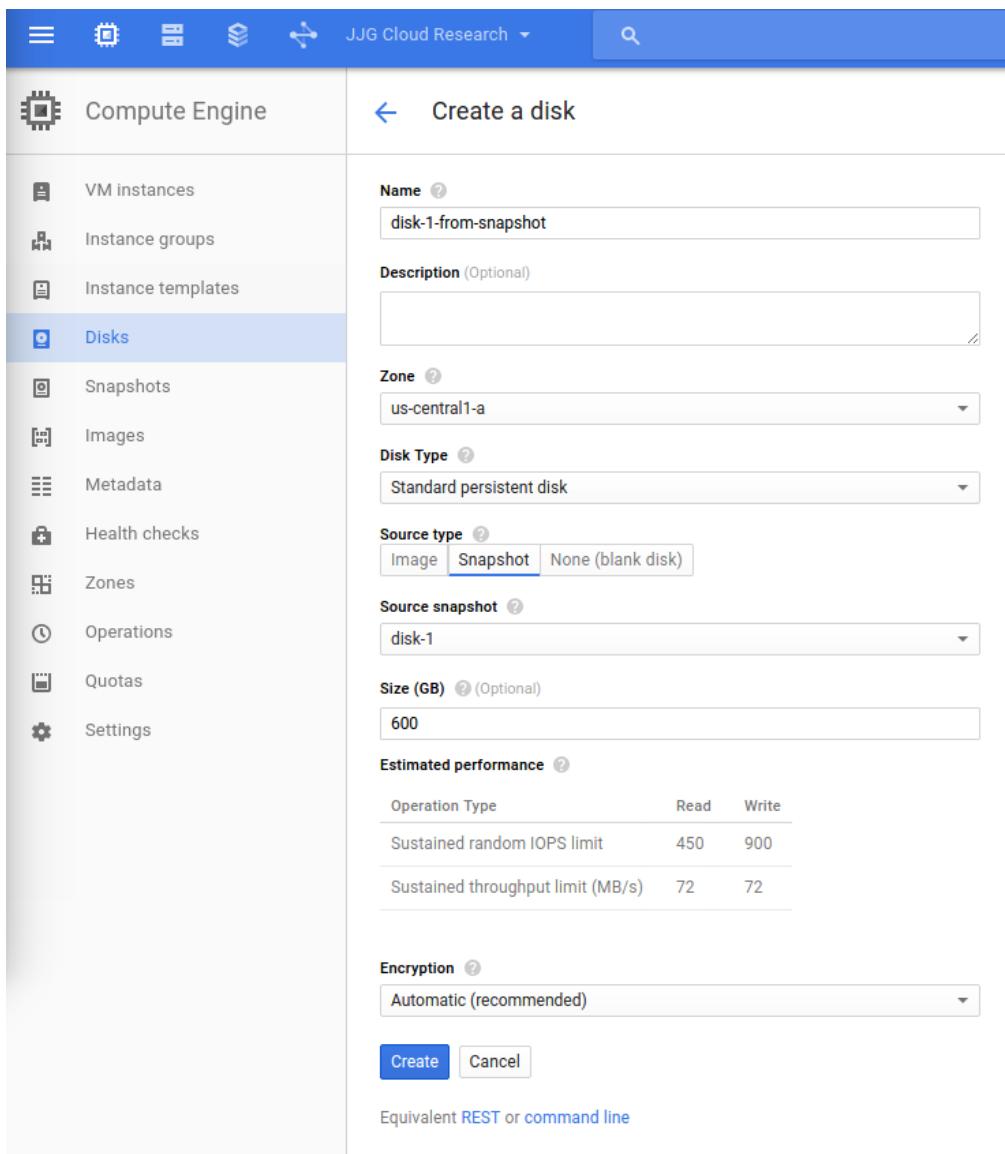
```
jjg@instance-1:~$ cd /mnt/disks/disk-1/workspace/
```

```
jjg@instance-1:/mnt/disks/disk-1/workspace$ echo "This is changed after the
snapshot!" > test.txt
jjg@instance-1:/mnt/disks/disk-1/workspace$ cat test.txt
This is changed after the snapshot!
```

Now imagine you forgot what `test.txt` used to have written in it and want to go back in time. To do this, let's create a disk instance from our snapshot and then mount it to our machine like any other disk. Start by navigating to the list of disks and choosing "Create Disk". Instead of creating a "blank disk" like we did before, this time we're going to choose "Snapshot" as the source type, and then choose `disk-1` as our source snapshot. The rest of the fields should look similar to the other times you've created a disk.

**WARNING**

**Don't forget to choose us-central1-a for the zone! Otherwise you won't be able to mount your disk to your VM!**

**Figure 9.11. Creating a disk instance from a snapshot**

When you click "Create", you'll be brought to the list of disks, and you should see your newly created disk (in our case, `disk-1-from-snapshot`) in the list.

**Figure 9.12. List of disks**

The screenshot shows a table of disks. The columns are: Name, Type, Size, Zone, and In use by. There are five rows:

Name	Type	Size	Zone	In use by
disk-1	Standard persistent disk	600 GB	us-central1-a	
disk-1-from-snapshot	Standard persistent disk	600 GB	us-central1-a	
instance-1	Standard persistent disk	50 GB	us-central1-c	wordpress
instance-2	SSD persistent disk	10 GB	us-central1-c	

Now let's just attach that disk to your VM (this time, we'll do it from the command line), and then we mount the disk on the remote machine.

#### **Listing 9.12. Attaching and mounting your disk to instance-1**

```
$ gcloud compute instances attach-disk instance-1 --zone us-central1-a --disk disk-1-from-snapshot
Updated [https://www.googleapis.com/compute/v1/projects/your-project-id-here/zones/us-central1-a/instances/instance-1].
$ gcloud compute ssh --zone us-central1-a instance-1
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Mon Sep  5 19:46:06 2016 from 104.132.34.72
jjg@instance-1:~$ sudo mkdir -p /mnt/disks/disk-1-from-snapshot
jjg@instance-1:~$ sudo mount -o discard,defaults /dev/disk/by-id/google-disk-1-from-snapshot /mnt/disks/disk-1-from-snapshot
```

Now we have both disks mounted to the same machine, where `disk-1-from-snapshot` holds the data we had **before** we modified it, and `disk-1` holds the data from **afterwards**. To see the difference, let's just print out the contents of our `test.txt` file for each disk.

#### **Listing 9.13. Content of test.txt**

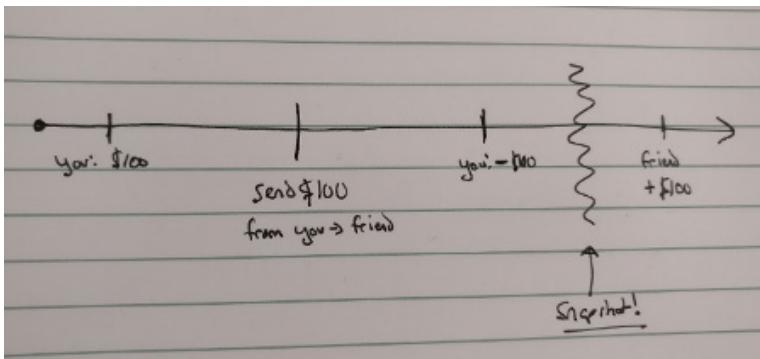
```
jjg@instance-1:~$ cat /mnt/disks/disk-1-from-snapshot/workspace/test.txt
This is a test
jjg@instance-1:~$ cat /mnt/disks/disk-1/workspace/test.txt
```

This is changed after the snapshot!

### SNAPSHOT CONSISTENCY

But what happens if you're writing to your disk, and you take a snapshot right in between two important disk operations? Using the analogy of a bank transfer, what happens if you take the snapshot right between the bank deducting \$100 from your account and crediting \$100 to your friend's account?

**Figure 9.13. Bad timing for a snapshot**



This issue is fundamentally very "low level" in that the problem arises due to the fact that computers tend to "cache" things in memory instead of always writing the data to your hard drive. This means that to avoid these types of problems where you take a snapshot at what we'll call "a bad time" you have to tell your virtual machine to "flush" any data that is stored in memory but not yet on the disk. But this only gets you so far: anything running on the machine might continue storing data in memory but not flushing it to the disk.

The end result is that in order to avoid a potentially disastrous snapshot, you should shut down your applications that are writing data (e.g., stop your MySQL server binary or using MySQL's `FLUSH` command), flush your disk buffers (using the `sync` command), freeze the disk (using `fsfreeze`) and only then take the snapshot.

### Listing 9.14. Stop servers, flush buffers, then snapshot

```
jjg@instance-1:~$ # Stop your applications!
jjg@instance-1:~$ sudo sync
jjg@instance-1:~$ sudo fsfreeze -f /mnt/disks/disk-1
jjg@instance-1:~$ # Create your snapshot!
jjg@instance-1:~$ sudo fsfreeze -u /mnt/disks/disk-1
```

In the section while your disk is "frozen" (after calling `fsfreeze`), any attempts to write to the disk will wait until the disk is "unfrozen". This means that if you don't halt your applications (e.g., your MySQL server) they'll hang until your unfreeze the file system.

Now that you understand how to use snapshots to protect your data over time, let's take

a brief detour to talk about disk "images" and why you might want to use them.

### 9.3.6 Images

Images are similar to snapshots in that they can both be used as the "source" of content when you create a new disk. The primary difference is that images are meant as "starting templates" for your disk whereas snapshots are meant as a form of back-up to pinpoint your disk's content at a particular point in time. As a matter of fact, every time we create a new VM from a base operating system we're using an image under the hood. The primary difference is that an image doesn't rely on differential storage, which may mean it's more expensive to keep around.

In general, this means that images are good for the starting point of your VMs, and even though you can create custom images, the curated list provided by Google should cover the common scenarios. Because of this, we're not going to dig into the details of creating custom images, but you can find a tutorial of how to do this in the Compute Engine documentation. Now that we've covered that, let's switch gears from the mechanics of storing data and start looking at disk performance.

### 9.3.7 Performance

As we briefly discussed in earlier chapters, persistent disks are designed to abstract away the details of managing physical disks (e.g., things like RAID arrays). We also learned that sometimes it makes sense to create a disk that is larger than needed due to the performance requirements. While this can feel counter-intuitive (and wasteful), rest assured that it's common and expected to create a disk that is larger than you actually need in order to get the performance that you need. That said, there are several different classes of persistent disk (Standard, SSD, and Local SSD), each with slightly different performance characteristics. Let's take a moment to look through each of these and explore when you might want to use the different classes.

**Table 9.1. Disk performance summary**

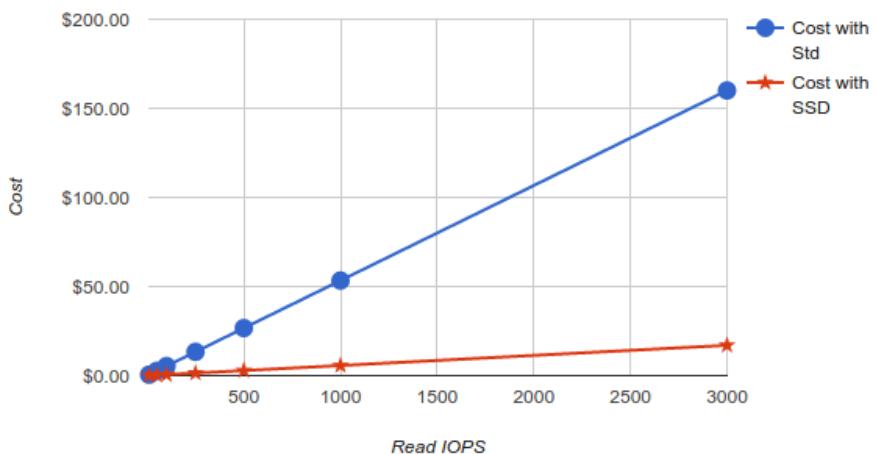
Type	Random (IOPS) per GB		Throughput (MB/s) per GB		Cost per GB
	Read (max)	Write (max)	Read (max)	Write (max)	
Standard	0.75 (3k max)	1.5 (15k max)	0.12 (180 max)	0.12 (120 max)	\$0.04
SSD	30 (25k max)	30 (25k max)	0.48 (240 max)	0.48 (240 max)	\$0.17
Local SSD	267 (400k max)	187 (280k max)	1.0 (1.5k max)	0.75 (1k max)	\$0.218

There are a few interesting things hiding in this table that we should look at. To start, it's clear that local SSD disks provide the most performance by far. Don't get too excited though, because these are **local** disks, meaning they aren't replicated and should be considered ephemeral rather than persistent. Put differently, if your machine goes away, so does all the data on local disks. Since Local SSDs are so different from the others, let's instead focus on the two truly persistent types: Standard and SSD.

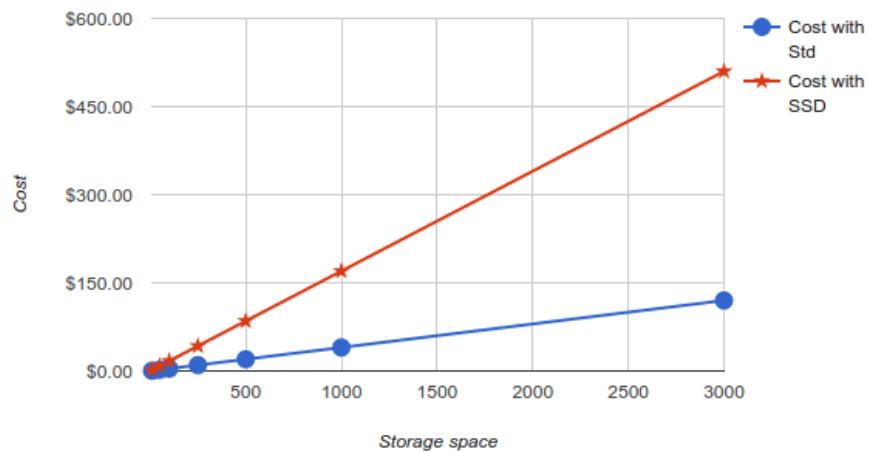
SSD and Standard disks have two very different performance profiles, which can be summarized pretty quickly. Standard disks are great if you need lots of space and don't

need super high performance, whereas SSDs are great for the super fast reads and writes. To put this in perspective, the following graphs show a comparison between SSD and Standard disks against disk size and read operation capacity.

**Figure 9.14. Graph of the cost by Read IOPS**



**Figure 9.15. Graph of the cost by Gigabyte stored**



In the first graph, you can see the comparative cost to achieve a given level of read IOPS. Notice how the cost for additional IOPS with a Standard disk (circles) is far more than an SSD (stars). In the second graph, you can see the comparative cost to store a given amount of data (GB). Notice how the trend line is almost exactly reversed, this time with the SSD (stars) each additional gigabyte of storage costs far more than with a standard disk (circles). Ultimately, when it comes time to create a disk that has the performance level you need, you have to look at both your throughput needs (e.g., how many GB/s do you need to read and write?) as well as your random

access needs (e.g., how many operations per second do you need?) and decide what disk type and size fits best.

Now that you understand disk performance a bit better, it might be a good idea to jump back to that drop-down box we ignored talking about disk encryption and briefly explain what that is and how it works.

### 9.3.8 Encryption

As you might guess, storing data in the cloud brings different risks than storing data locally on your home computer. Instead of worrying about someone breaking into your house, you have to worry about unauthorized access to your data "in the cloud". It also means that you don't have to worry about fires at your house, and instead worry only about fires in a Google data center so it's really a double-edged sword.

When we say "unauthorized access" to your data, we tend to imagine a hacker in a foreign country trying to steal some private data. A less-commonly imagined scenario is a Google employee copying your data, which could be equally bad. To deal with this, Google encrypts the data stored on your disks so that even if someone were to copy the bytes directly they're useless without the encryption keys. By default, Google comes up with their own random encryption key for your disk and stores that in a secure place with access logged, but if you're worried that Google's storage (and access logging) for the keys encrypting your disks isn't trustworthy enough, you can elect to keep these keys for yourself and give Google the key only when you need to decrypt the disk (such as when you first attach it to a VM).

To make this more concrete, let's quickly walk through the process of creating an encrypted disk where we manage the keys ourselves. Let's start the process by getting a random key to use. To do this, we'll use `/dev/urandom` in Linux combined with the `tr` command to put a random chunk of bytes into a file called `key.bin`. To see what these bytes look like, you can use the `hexdump` command.

#### **Listing 9.15. Get a random chunk of binary data**

```
$ head -c 32 /dev/urandom | tr '\n' = > key.bin
$ hexdump key.bin
00000000 2a65 92b2 aa00 414b f946 29d9 c906 bf60
00000010 7069 d92f 80c8 4ad1 b341 0b7c 4d4f f9d6
00000020
```

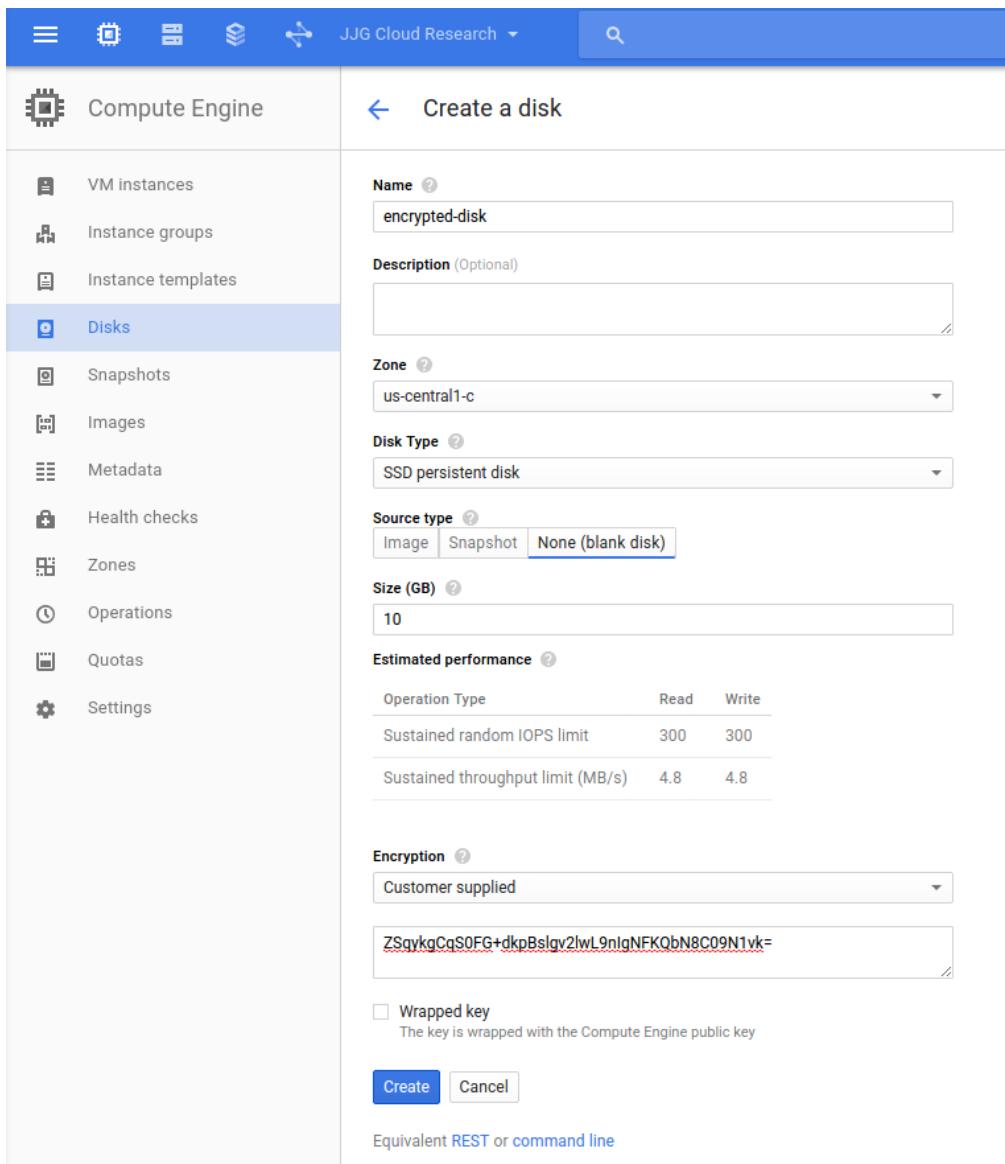
At this point, you can decide either to use RSA encryption to "wrap" your key, or simply leave the key as is. In the world of cryptography, "wrapping a key" is basically encrypting it with a public encryption key so that it can only be decrypted by the corresponding private key. In this case, it's the way that you ensure your secret is only able to be decrypted by Google Cloud Platform systems. As with most situations in security, storing things in "plain text" is typically bad so it's recommended that you wrap your keys, but for the purposes of this example we'll leave our key alone. You can read more about how to wrap your keys in the Google Compute Engine documentation. This means that all we have left to do is put the key in `base64` format,

which you can do with the `base64` command in Linux.

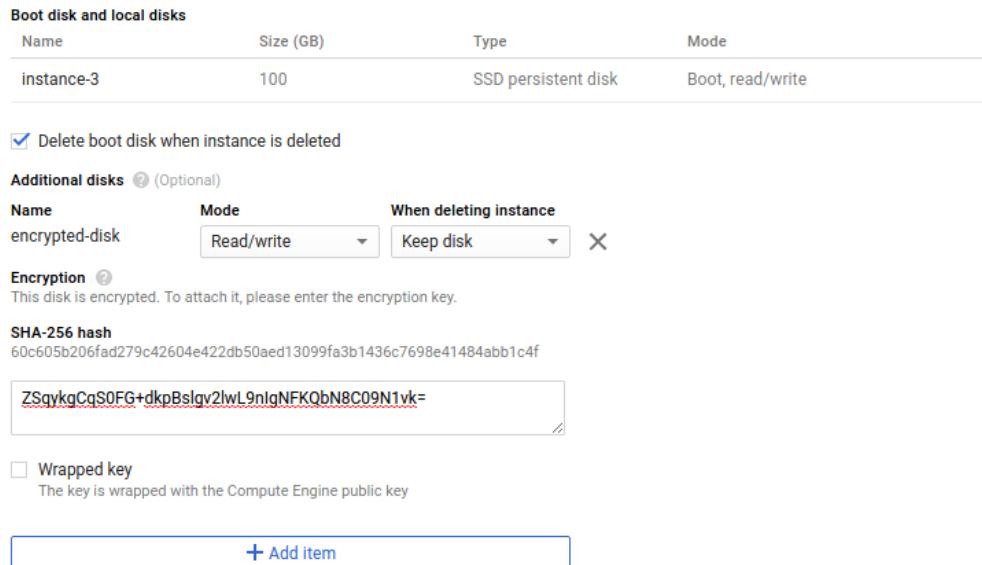
#### **Listing 9.16. Encode your binary key to base-64**

```
$ base64 key.bin  
ZSqykgCqS0FG+dkpBslgv2lwL9nIgNFKQbN8C09N1vk=
```

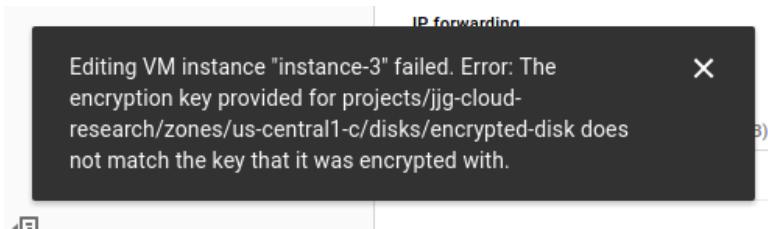
At this point, you create a disk like you usually would, but choose "Customer supplied" from the Encryption drop-down, and leave the box "Wrapped key" unchecked (since we're leaving the key as plain text) as you can see in Figure 9.16. To finish up, just click Create and your disk should appear.

**Figure 9.16. Create an encrypted disk**

Now that you have a disk that is encrypted with our keys, let's look next at how you attach a disk encrypted with your own key. To start, navigate to an existing instance from before and choose Edit. Just as before, in the section where you attach new disks, choose the newly created "encrypted disk" from the drop-down. You should notice something new this time which is a section saying that the disk is encrypted and you'll need to provide the key from before!

**Figure 9.17. Attach an encrypted disk**

Once you've pasted in the encryption key just scroll down and click "Save". If you use the wrong key, you'll see an error like Figure 9.18 . Google figures this out based on the hash of the key which is stored along with the disk (in other words, we can be sure the key provided is the right one without storing the actual key).

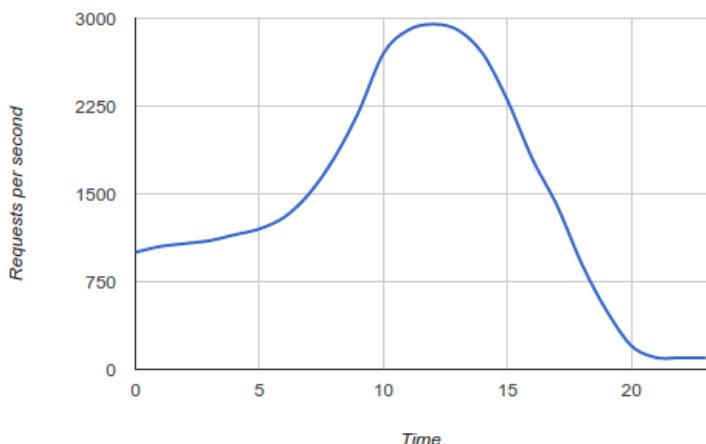
**Figure 9.18. Invalid encryption key**

As you can see, dealing with encrypted disks is quite similar to dealing with an unencrypted disk, with the main difference being that you have to provide some extra information (the key) when you attach an encrypted disk to an instance. Once the disk is mounted to the instance it will act just like a regular disk that we've talked about before, so everything you learned previously still applies. Now that you understand all about disks, let's talk a bit more about the computing features that we brought up earlier which will show you why cloud computing is a league above traditional VPS hosting.

## 9.4 Instance groups and dynamic resources

Given that you have a firm grip on how disks and instances interact, it's time to explore one of the more unique aspects of cloud computing: auto-scaling. By auto-scaling we mean the ability to expand or contract the number of VMs running to handle requests based on how much traffic is being sent to them (which, for example, may show up as CPU usage). To make this a bit more clear, let's use a concrete example of a system that experiences request load that varies over the course of the day as shown in Figure 9.19.

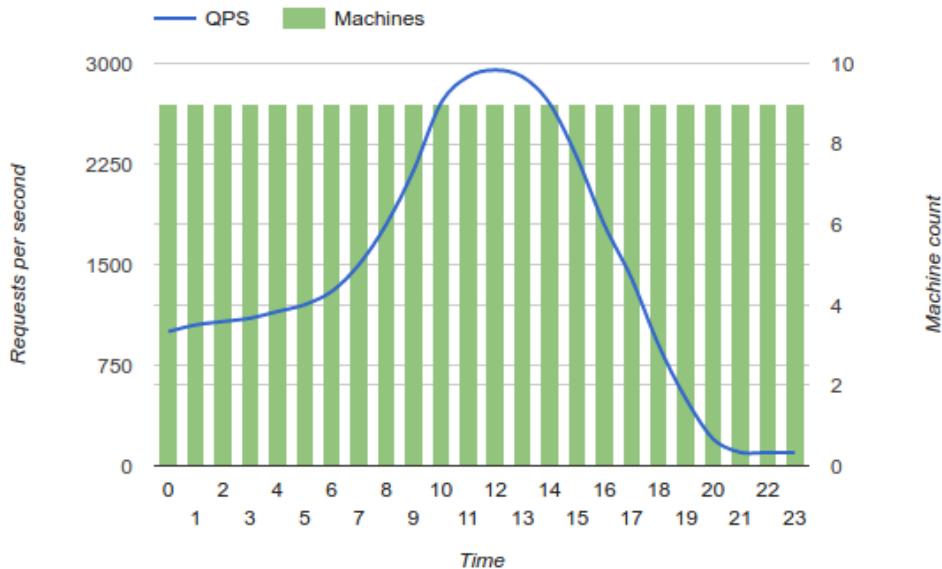
**Figure 9.19. Requests per second through the day**



As you can see, at the start of the day the system sees around 1,000 requests every second, growing quickly until about noon and only slows as it approaches 3,000 requests per second. Then it steadily falls off down to about 100 requests per second.

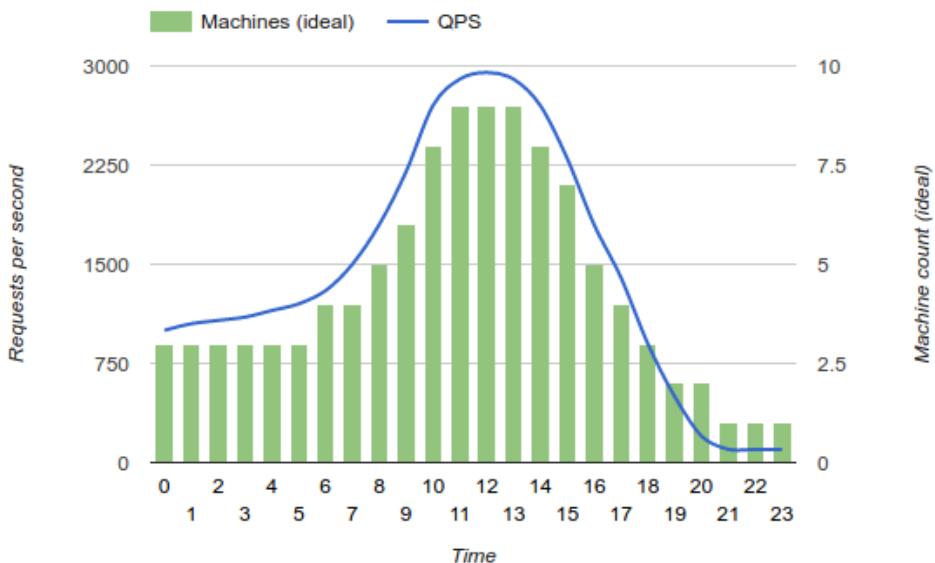
In a perfect world this system would have exactly the right amount of capacity available to handle the number of requests needed. This means that if you need 3 machines at the start of the day, you'd need somewhere around 3 times that amount toward the middle of the day, and no more than one at the end. Currently, and unfortunately, all you've seen so far with Compute Engine is the very wasteful version of this graph where you turn on the exact number of machines that you need to handle the worst part of the day. This means that those machines would be sitting idle for around half the day as you can see in [Figure 9.20](#).

**Figure 9.20. Machines provisioned for the worst part of the day**



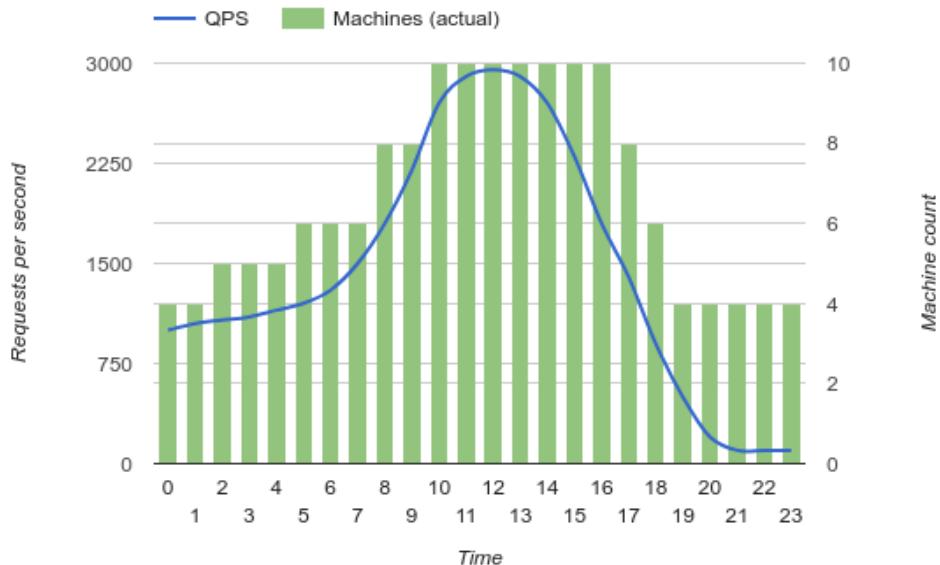
To make real life a bit more like the perfect world where your system grows and shrinks to meet your demands, Compute Engine has a concept of auto-scaling. To put this visually, the number of machines added to this graph would ideally look something like [Figure 9.21](#).

**Figure 9.21. Machines required to handle requests**



Since computing power is reactive, we might not be able to get as close to the line you see above, however it could be possible to get much closer than the block shape from having a specific number of machines always on regardless of the traffic sent to them.

**Figure 9.22. Machines that might actually handle requests with auto-scaling**



So how would this actually work? The main idea here is based on "templates", where you effectively teach Compute Engine how to turn on instances configured to your liking. Once GCE knows how to turn on *your* instances, it can monitor the overall CPU usage of the currently running instances, and decide whether to turn on more, turn off some currently running, or do nothing. To put all of this together, there are two new concepts we need to understand: Instance Groups and Instance Templates.

Instance Templates are sort of like the "recipe" for your instances, and contain all of the information needed to turn on a new VM instance that looks exactly how you want it to. Higher up the chain, an Instance Group acts as a container of these managed instances and, given a template and some configuration parameters, is the thing that decides whether to turn on more, turn off some, or leave things alone.

Let's dive right in, first by creating an instance template. To do this, navigate to the Compute Engine section of the Cloud Console, and choose Instance Templates from the left-hand navigation. From there, click on the "Create Instance Template" button at the top where you'll see a screen that looks very similar to the one you saw when creating a single Compute Engine instance. Start by naming your instance (e.g., `first-template`) and then click the "Create" button at the bottom.

**Figure 9.23. Creating your first instance template**

← Create an instance template

Describe a VM instance once and then use that template to create groups of identical instances [Learn more](#)

**Name** [?](#)  
first-template

**Machine type**

1 vCPU 3.75 GB memory [Customize](#)

**Containers**  
 Deploy your container images on Compute Engine VMs

**Boot disk** [?](#)

New 10 GB standard persistent disk  
Image  
Debian GNU/Linux 8 (jessie) [Change](#)

**Identity and API access** [?](#)

**Service account** [?](#)  
Compute Engine default service account

**Access scopes** [?](#)  
 Allow default access  
 Allow full access to all Cloud APIs  
 Set access for each API

**Firewall** [?](#)  
Add tags and firewall rules to allow specific network traffic from the Internet

Allow HTTP traffic  
 Allow HTTPS traffic

[Management, disk, networking, SSH keys](#)

[Create](#) [Cancel](#)

Equivalent REST or command line

When that completes, you'll see a list of templates with your newly created one in that list. If you click on the template, you'll be brought to a details page that should feel pretty familiar. To actually use this "recipe" of an instance template, click the button at the top that says "Create Instance Group". This page is where we decide how the template will be applied to create actual instances.

Start by naming the group itself, and then let's set the group to be a single-zone group

in us-central1-c. (Note that you can choose a "regional" configuration by selecting "Multi-zone" and choosing the region to host the instances.) Leave the group type as a "managed instance group" and make sure that the instance template is set to the one we just created. Finally, let's change the number of instances from 1 to 3 (leaving the "Autoscaling" setting to "Off"), and then click the "Create" button at the bottom of the page.

**Figure 9.24. Creating your first instance group**

[←](#) Create a new instance group

Use an instance group when configuring a load-balancing backend service or to group VM instances. [Learn more](#)

**Name** [?](#)

**Description** (Optional)

**Location**  
 Multi-zone groups span multiple zones which assures higher availability [Learn more](#)  
 Single-zone  
 Multi-zone

**Zone** [?](#)

**Specify port name mapping** (Optional)

**Group type**  
 **Managed instance group**  
 Managed instance group contains identical instances, created from an instance template, and supports autoscaling, autohealing, rolling updating, load balancing and more. VM instances are stateless and disks are deleted on VM deletion or recreation.  
[Learn more](#)  
 **Unmanaged instance group**  
 Unmanaged instance group is best for load balancing dissimilar instances, which you can add and remove arbitrarily. Autoscaling, autohealing, and rolling updating are not supported. [Learn more](#)

**Instance template** [?](#)

**Autoscaling** [?](#)

**Number of instances**

**Autohealing**  
 VMs in the group are recreated as needed. You can use a health check to recreate a VM if the health check finds the VM unresponsive. If you do not select a health check, VMs are recreated only when stopped. [Learn more](#)

**Health check**

**Initial delay** [?](#)  
 seconds

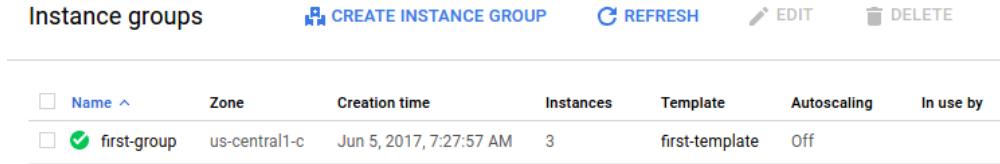
[▼ Advanced creation options](#)

[Create](#) [Cancel](#)

Equivalent [REST](#) or [command line](#)

It will take a few seconds to finish, but you should eventually see your instance group fully deployed.

**Figure 9.25. Listing your instance groups**



The screenshot shows a table with the following data:

<input type="checkbox"/> Name ^	Zone	Creation time	Instances	Template	Autoscaling	In use by
<input checked="" type="checkbox"/> first-group	us-central1-c	Jun 5, 2017, 7:27:57 AM	3	first-template	Off	

If you click on the instance group, you'll see a list of the 3 instances that were created using the template from before. Now that we have an instance group, let's explore what we can do with it, starting with growing and shrinking our group.

### 9.4.1 Growing and shrinking

The cool part about instance groups is that you can easily change the size of the group, and Compute Engine will do all the heavy lifting. This makes growing and shrinking the group pretty simple and straightforward. Since we have an instance group that we defined with a specific number of instances, we can easily shrink our instance group by simply "deleting" some of the instances.

For example, if we wanted to shrink down to a single instance, all we'd have to do is check the boxes next to two of the three instances, and then from the top right corner choose the context menu (it looks like three dots). From that menu, we just click "Delete instance" and we should see loading icons next to the two instances we selected. After a minute or so, we should see the instances disappear and the group is now made up of a single instance.

**Figure 9.26. Deleting two instances from the group**

Name	Template	Internal IP	External IP	Connect
<input checked="" type="checkbox"/> first-group-7kqf	first-template	10.240.0.6	104.197.226.237	SSH ▾
<input checked="" type="checkbox"/> first-group-s8zm	first-template	10.240.0.7	130.211.177.173	SSH ▾
<input type="checkbox"/> first-group-zvk2	first-template	10.240.0.5	104.198.232.50	SSH ▾

To grow the instance again, we simply need to click the pencil icon to reach the "edit" form, and change the number of instances back to 3. After a few seconds we should see our new instances being created again!

What's happening here is that since the instance group had Autoscaling set to "Off", by deleting the instances we're effectively telling the instance group that we want the number of instances to be smaller (in this case, we want a total of 1 instance). As a result, the instance group updates itself to reflect our intent.

When we then go change the number of instances back to 3, we're telling the group that we want exactly 3 instances once again. As a result, it creates two new instances to satisfy our new requirement.

Let's make things even more interesting by looking at how to "upgrade" our instance group.

### 9.4.2 Rolling updates

Sometimes we might have a new software package that we want to deploy across a bunch of machines, but we want to do it in stages rather than all at once. In other words, we might want to upgrade, say, half of the instances, while leaving the other half alone in case the newest version runs into any problems. Instance groups can do this using something called Rolling Updates, which we'll explore next.

To see how this works, let's start by creating a new instance template that turns on a simple Apache web server. Go back to the page to create an instance template, and do the same thing we did before, but with two key changes:

1. Choose an Ubuntu 17.04 boot disk (instead of Debian 8)
2. Make sure to check the "Allow HTTP traffic" box

Finally, under the "Management" tab you'll see a section called "Startup script". In that box, simply install Apache using the `apt-get` command for Ubuntu.

#### **Listing 9.17. Startup script to install Apache**

```
#!/bin/bash
sudo apt-get install -y apache2
```

**Figure 9.27. Creating our new instance template**

[← Create an instance template](#)

Describe a VM instance once and then use that template to create groups of identical instances [Learn more](#)

**Name** [?](#)

**Machine type**

1 vCPU	3.75 GB memory	<a href="#">Customize</a>
--------	----------------	---------------------------

**Container** [?](#)  
 Deploy a container image to this VM instance. [Learn more](#)

**Boot disk** [?](#)

 New 10 GB standard persistent disk	<a href="#">Change</a>
Image	
Ubuntu 16.04 LTS	

**Identity and API access** [?](#)

**Service account** [?](#)

**Access scopes** [?](#)  
 Allow default access  
 Allow full access to all Cloud APIs  
 Set access for each API

**Firewall** [?](#)  
 Add tags and firewall rules to allow specific network traffic from the Internet

Allow HTTP traffic  
 Allow HTTPS traffic

[Management](#) [Disks](#) [Networking](#) [SSH keys](#)

**Description** (Optional)

**Labels** [?](#) (Optional)  
 [+ Add label](#)

**Automation**

**Startup script** (Optional)  
 You can choose to specify a startup script that will run when your instance boots up or restarts. Startup scripts can be used to install software and updates, and to ensure that services are running within the virtual machine. [Learn more](#)

```
#!/bin/bash
sudo apt-get install -y apache2
```

**Metadata** (Optional)  
 You can set custom metadata for an instance or project outside of the server-defined

Once you've created the new template, let's use a "rolling update" to phase it into use. Go back to the instance group we created already, and choose "Rolling update" from the top of the page. Once there, let's migrate our 2 of our instances to use our new Apache-enabled template.

First, click "Add item" to create a new row, and then choose the new Apache template (called `apache-template` in the image) from the target template. Next, set the "target size" to 2 instances. Before clicking "Update", let's look at a few of the other parameters here. First, there is a choice between when you want the update to happen. Compute Engine can either start it immediately (called "Proactive") or whenever there's an outage of a machine (e.g., if it becomes "unhealthy"). For this example, we'll use the proactive mode. When handling an upgrade, old instances are only turned off after the replacement is ready. This means that you'll go above your maximum instance count during a rolling upgrade. To avoid turning on twice the number of instances you have, you can choose to set how big this "surge" over the limit will be. For our example, we'll use 1 instance.

Next, you can control how much work should happen concurrently. In other words, you can limit how many instances (or what percentage of instances) can be down at a given time. For our example, let's only upgrade 1 at a time, but if you have a larger cluster you may want to do this in a larger group (e.g., 10 at a time).

Finally, you can configure how long to wait between updates. That is, you may have some extra configuration happening on your instance and it may take a few minutes to be truly considered "healthy". If you have that scenario, you may want to set this to a safe amount of time that it takes your newly created instance to boot up and become active in the cluster. For our Apache example let's use 30 seconds just to make sure Apache installed and is running.

**Figure 9.28. Configuring our rolling update**

[← Update first-group](#)

A rolling update is a gradual update of the instances in the instance group to the target configuration of templates. [Learn more](#)

**Current configuration**

first-template : 100% (3 instances)

**New configuration**  
Select 1-2 template(s), using the target size to specify percentage or number of instances per template

Template	Target size	X
first-template	Currently 1 out of 3 instances	X
apache-template	2	instance(s) X
<a href="#">+ Add item</a>		

**Update mode**

Proactive  
Starts the update immediately

Opportunistic  
Only updates when new instances are created or the instance group is resized. [Learn more](#)

**Maximum surge**  
Maximum number (or percentage) of temporary instances to add while updating. [Learn more](#)

1 instance(s) ▾

**Maximum unavailable**  
Maximum number (or percentage) of instances that can be offline at the same time while updating. [Learn more](#)

1 instance(s) ▾ out of 3 instances

**Minimum wait time**  
Time to wait between consecutive instance updates. [Learn more](#)

30 s

You are deploying template "first-template" to 1 instance and template "apache-template" to 2 instances in instance group "first-group".  
1 instance will be taken offline at a time and 1 instance will be temporarily added to support the update.  
The managed instance group will wait 30 seconds after an updated instance is healthy before considering the instance as ready and proceeding to the next instance.

[Update](#) [Cancel](#)

After you click update, you should see a progression of your instances turning off and turning on, and the end result will be that 2 of the instances use the Apache template, and one of them was left alone.

**Figure 9.29. After the rolling update has completed**

<input type="checkbox"/> Name ^	Template	Internal IP	External IP	Connect
<input type="checkbox"/> <span style="color: green;">✓</span> first-group-nsjg	apache-template	10.240.0.6	130.211.151.73 ↗	SSH ▾
<input type="checkbox"/> <span style="color: green;">✓</span> first-group-qjch	apache-template	10.240.0.9	104.154.247.150 ↗	SSH ▾
<input type="checkbox"/> <span style="color: green;">✓</span> first-group-zvk2	first-template	10.240.0.5	104.198.232.50	SSH ▾

To do a complete update, just do the same process again while setting the end-state of 100% of instances using the Apache template. Now that you've seen a rolling update, let's explore how auto-scaling works.

### 9.4.3 Autoscaling

Auto-scaling builds on the principles we saw with a rolling update, but looks at various measures of "health" to decide when to replace an instance, or grow and shrink the cluster as a whole. In other words, if a single instance becomes unresponsive, the instance group can mark it as "dead" and replace it with a new one. Additionally, if instances become overloaded (e.g., the CPU usage goes above a certain percentage) the instance group can increase the size of the pool to accommodate the unexpected load on the system. Conversely, if all instances are far under the CPU limit for a set amount of time, the instance group can retire some of the instances to remove unnecessary cost.

The underlying idea behind this isn't all that complicated, but configuring the instance groups and templates to do this kind of thing is a bit trickier. Currently, the instance group we have is configured to always be 3 instances. Let's start by changing this to scale between 1 and 10 instances whenever CPU usage goes above a certain threshold.

Start by looking at the instance group we created and click on the edit button (which looks like a pencil icon). On that page, you'll notice a drop-down called "autoscaling" which we had previously set to "off". When we flip this to "on" we're able to choose how exactly we want to scale the group, but to start, let's use CPU usage.

Leave the "Autoscale based on" option to "CPU usage", but let's change the target CPU usage to 50%. Next, make sure the minimum and maximum instances are set to 1 and 10 respectively. Finally, let's look at the cool-down period setting. The cool-down period is the minimum amount of time that the group should wait before deciding that an instance is "above the threshold". In our configuration this basically means that if you have a spike of CPU usage that lasts less than the cool-down period, it won't trigger a new machine being created. In a real-life scenario it probably makes sense to leave this to at least 60 seconds, but for this example let's make the cool-down period a bit shorter so we can see changes to the group more quickly.

**Figure 9.30. Configuring autoscaling**

**Edit first-group**

**Zone**  
us-central1-c

**Specify port name mapping (Optional)**

**Instance template** apache-template

**⚠️** The template applies only to newly created VMs. After saving these settings, use [this gcloud command](#) to recreate existing VMs using the new template

**Autoscaling** On

**Autoscale based on** For best results read [Configuring autoscaling instance groups](#)

CPU usage

**Target CPU usage** Scaling dynamically creates or deletes VMs to meet the group target. [Learn more](#)  
50 %

**Minimum number of instances** 1

**Maximum number of instances** 10

**Cool-down period** 15 seconds

**Autohealing**  
VMs in the group are recreated as needed. You can use a health check to recreate a VM if the health check finds the VM unresponsive. If you do not select a health check, VMs are recreated only when stopped. [Learn more](#)

**Health check** No health check

**Initial delay** 300 seconds

**Save** **Cancel**

After you click save and wait a few seconds, you should notice that two of your instances have disappeared! The instance group noticed that CPU usage was low and as

a result removed two of them that weren't needed anymore.

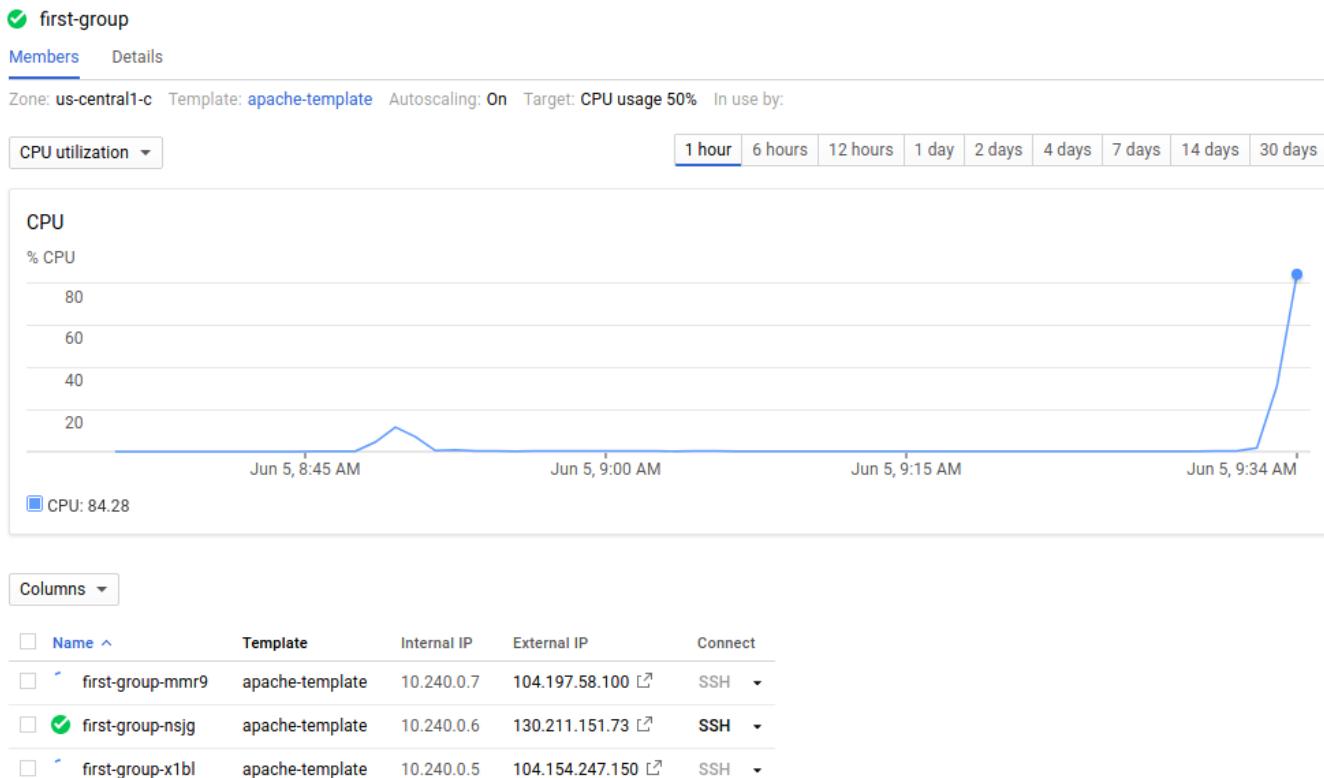
Let's try making things go the other way now by making the remaining instance very very busy so that the group sees the CPU usage jump. SSH into the remaining instance and run the following command which uses the `stress` library to force the CPU to do some extra work.

#### Listing 9.18. Use the stress library to increase the CPU usage

```
$ sudo apt-get install stress
$ stress -c 1
```

While that's running keep an eye on the CPU graph in the Cloud Console. You should notice it start to increase rapidly, and as it gets higher and stays that way for a few seconds, you should see at least one completely new VM get created!

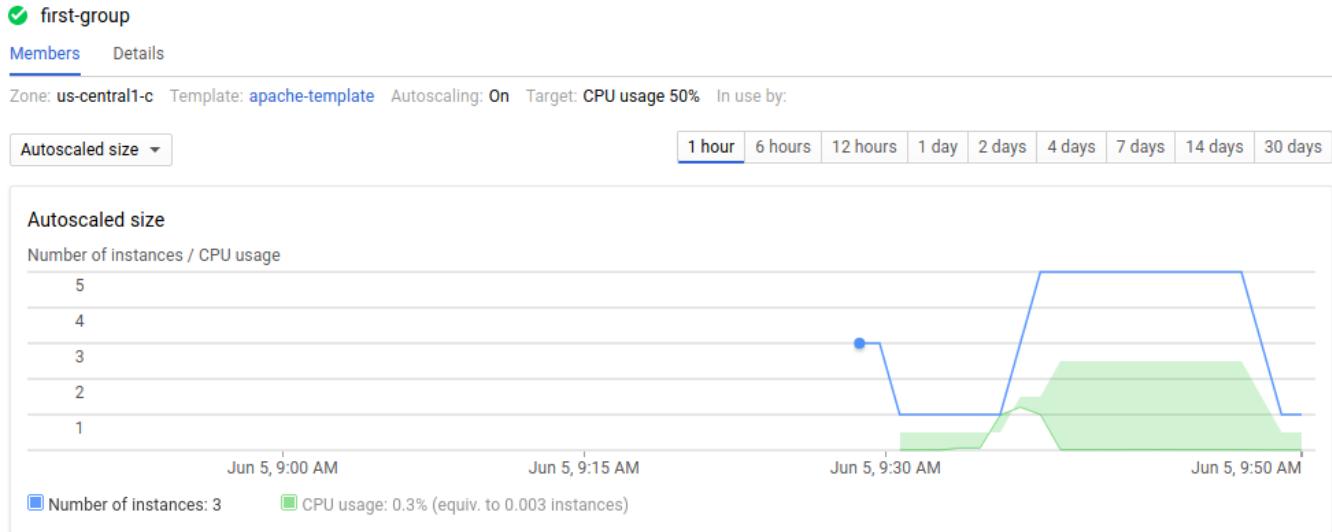
**Figure 9.31. CPU usage makes new machines appear**



If you leave the stress tool running, even more instances will be created to handle the (apparent) large load on that one instance. To close things up, hit **CTRL+C** to kill the stress tool, and watch that in just a few minutes the group should shrink back down to a

single instance. If you look at the "autoscaled size" chart, you can see how your cluster has grown and shrank based on the CPU usage.

**Figure 9.32. Autoscaled size graph**



As you can see, once you teach Compute Engine how to turn on instances configured the way you need to run your application, it can handle the rest based on how busy the VMs become. In addition to allowing you to automatically scale your compute capacity, using these templates also opens the door to a new form of computing that can significantly reduce your costs, using preemptible VMs. Let's take a look at how that works and when you should consider it as a viable option for your workloads.

## 9.5 *Ephemeral computing with preemptible VMs*

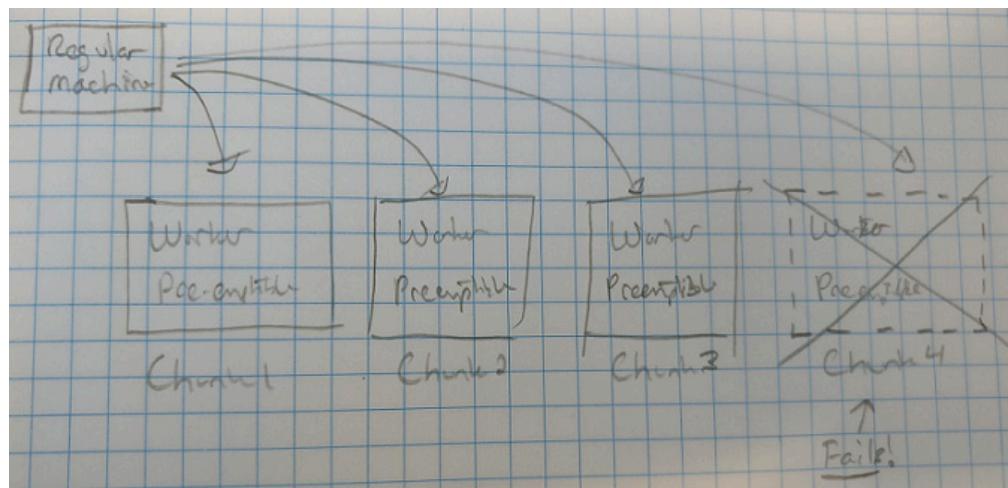
So far, all of the machines we've talked about have been virtual, but also relatively persistent. That is, once you create the VM it continues to run until you tell it to stop. But Compute Engine also has another type of machine that's "ephemeral", meaning it might disappear at any moment and never lives for more than 24 hours. In short, Google gives you a discount on the regular per-hour price in exchange for being able to reclaim the machine at any time and resell it if someone else shows up willing to pay the full price.

### 9.5.1 *Why use preemptible machines?*

So far, the most common use for preemptible machines is large batch workloads where you have lots of worker machines that process a small piece of an overall job. The reason is pretty simple: preemptible machines are cheap, and if one of those machines is terminated without notice, the job might complete a bit slower but it won't be canceled all together.

In other words, imagine you have a job that you want to split into 4 chunks. You could use a regular VM to orchestrate the process, and then 4 preemptible VMs to do the actual work. By doing this, any one of the chunks could be canceled at any point, but you could just re-try that chunk of work again on another preemptible VM. By doing this, you can use cheaper computing capacity to work on the job and in exchange for costing less, the workers may get killed and need replacement.

**Figure 9.33. Four workers, one getting killed**

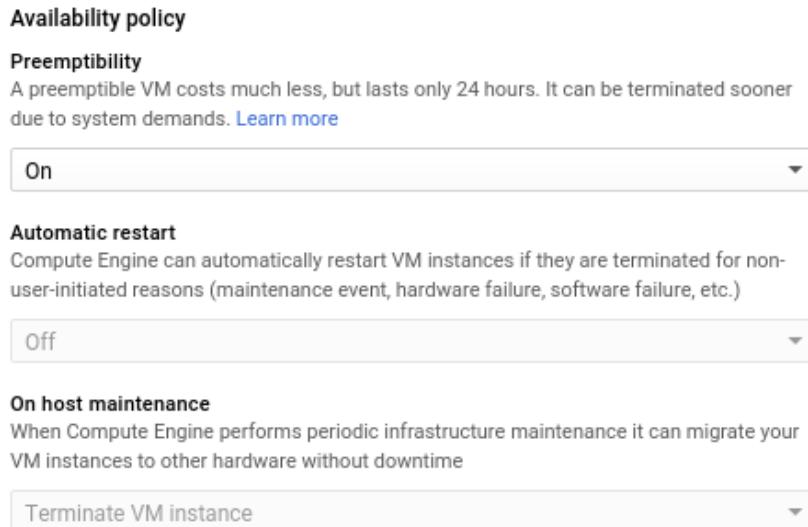


Now that we've gone through the theory behind why preemptible machines exist and what they're good for, the next obvious question is how it all actually works. To understand that, let's explore the two new things we have think about: turning on machines as "preemptible", and handling requests for machines to be terminated.

### 9.5.2 Turning on preemptible VMs

Creating a preemptible VM is quite simple: when you're creating your VM (or your instance template), in the "Advanced" section (the link that says "Management, disks, networking, SSH keys"), you'll notice under "Availability policy" that you can set "Preemptibility. Changing this from "Off" to "On" means that your VMs will be preemptible.

**Figure 9.34. Drop-down to create a preemptible VM**



As you'd guess, it comes with the side-effect that both automatic restarts and live-migration during host maintenance are both disabled. This shouldn't be a problem because when designing for preemptible machines we're already expecting that the machine can disappear at any given moment. Now that you understand how to create a preemptible machine, let's jump to the end of its life and see how to handle the inevitable terminations from GCE.

### 9.5.3 Handling terminations

Since preemptible machines can be terminated at any point in time (and will definitely be terminated within 24 hours), understanding how to gracefully handle these terminations is critically important. Luckily, Compute Engine doesn't sneak up on you with these, but instead gives you a reasonable notification window to let you know that your VM is going to be terminated. This means that you can listen for that notification, and "finish up" any pending work.

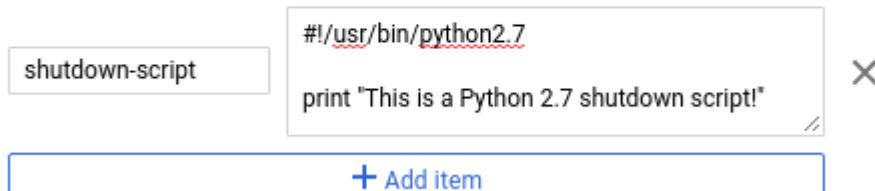
The easiest way to listen for this notification is by setting a shutdown script, sort of the opposite of what we did with our instance templates' startup scripts. Once triggered, Compute Engine gives the VM 30 seconds to finish up, and then sends a firm termination signal (the equivalent of pressing your machine's power button) and switches the machine to terminated. This basically means that your shutdown script has 30 seconds to do its work. After that 30 seconds, "the plug gets pulled".

To set a shutdown script, you can use the metadata section of the instance (or instance template), with a key called `shutdown-script`.

**Figure 9.35. Setting a shutdown script when creating a VM**

**Metadata (Optional)**

You can set custom metadata for an instance or project outside of the server-defined metadata. This is useful for passing in arbitrary values to your project or instance that can be queried by your code on the instance. [Learn more](#)

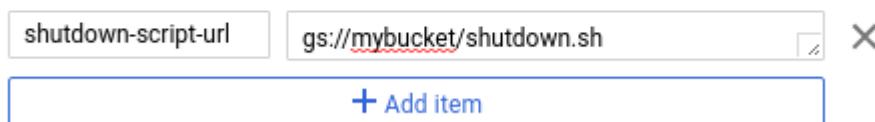


If you have a shutdown script stored remotely on Cloud Storage, you can link to that using metadata for the key `shutdown-script-url`, and a URL starting with `gs://`.

**Figure 9.36. Setting a shutdown script URL when creating a VM**

**Metadata (Optional)**

You can set custom metadata for an instance or project outside of the server-defined metadata. This is useful for passing in arbitrary values to your project or instance that can be queried by your code on the instance. [Learn more](#)



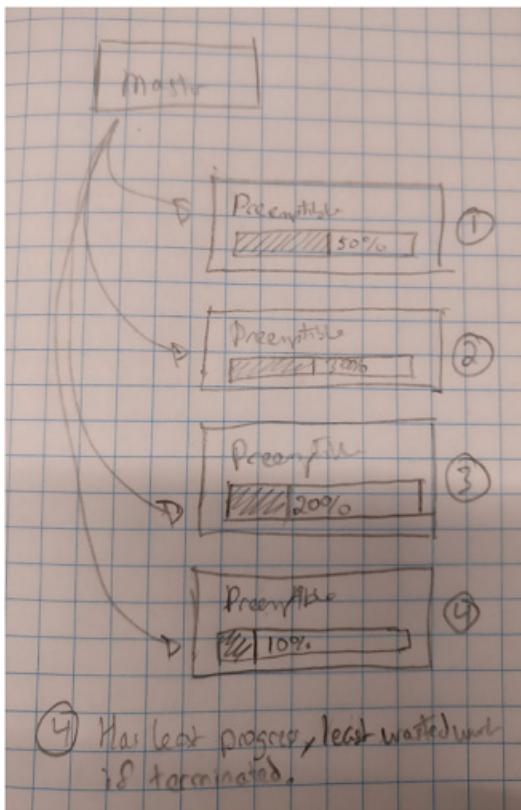
Since termination of a preemptible machine triggers a regular shutdown script, this means that you can test your scripts by simply "stopping" the machine. So how often will this termination actually happen? Let's look at how GCE decides which VMs to terminate first.

#### 9.5.4 Preemption selection

It turns out that when selecting a VM to terminate, GCE chooses the "youngest" one, which might seem counter intuitive. To see why they do this, let's think about what the best case is for choosing to terminate a VM.

Imagine that we have a job where each VM needs to download a large (5 GB) file. If we boot our VMs in order, and they get right to work downloading the file, at any given time, the first VM will have more download progress than the second, third, and fourth. Now imagine that Google needs to terminate one of these VMs. Which one is most convenient to terminate? Which VM causes the least amount of wasted work if it has to be terminated and start over? Obviously this is the one that started last and has downloaded the least amount of data.

**Figure 9.37. Selecting which machine to terminate**

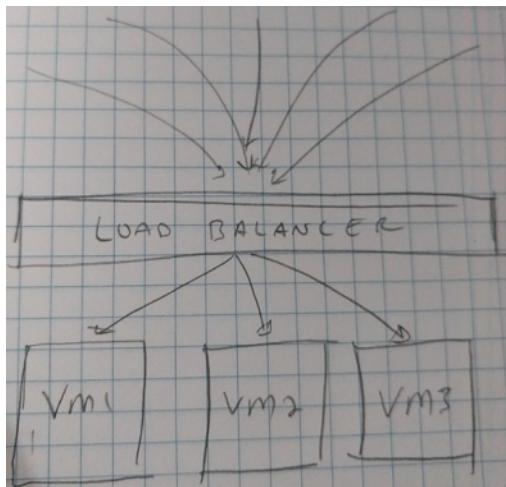


Since GCE will terminate the youngest machine, one concern is "thrashing" where machines continually get terminated and never make any progress. While this is definitely possible, GCE will distribute terminations evenly at a global level. This means that if Compute Engine wants to reclaim 100 VMs, it will take those 100 from lots of different customers rather than a single one. Due to this, you should only see repeated terminations during extreme circumstances. For these rare cases where thrashing does happen remember that GCE doesn't charge for VMs that are forcibly terminated within their first 10 minutes. With that, let's explore how to balance requests across multiple machines using a load balancer.

## 9.6 Load balancing

Load balancing is a relatively old topic, so if you've run any sort of sizable web application you're probably at least familiar with the concept. The underlying principle is that sometimes the overall traffic that you need to handle across your entire system is far too much for a single machine. As a result, instead of making your machines bigger and bigger to handle the traffic, you use more machines and rely on a load balancer to split (or "balance") the traffic (or "load") across all the available resources.

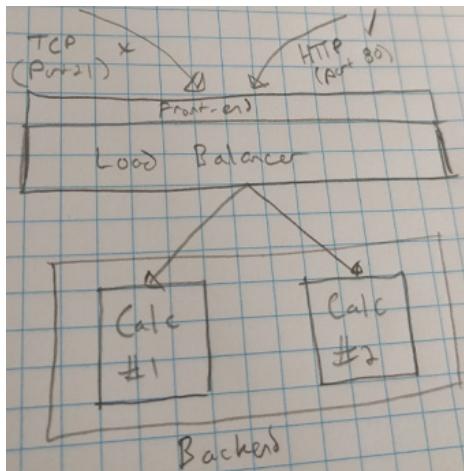
**Figure 9.38. Load balancers spread traffic across all of the available resources**



Traditionally, if you needed a load balancer (and you weren't handling many millions of hits per second), you'd turn on a VM and install some load balancing software, which could be anything from HAProxy to Squid, or even nginx. But since this is such a common practice, Google Cloud Platform offers a fully-managed load balancer that does all of the things that traditional software load balancers can do.

Since load balancers take incoming requests at the front and spread them across some set of machines at the back, this means that any load balancer will have both "frontend" and "backend" configurations. These configurations basically decide how the load balancer will listen for new requests, and where it will send those requests as they come in. And these configurations can range anywhere from super simple to extremely complex. Let's look at a simple one to understand better how this all works.

**Figure 9.39. A calculator application using a load balancer**



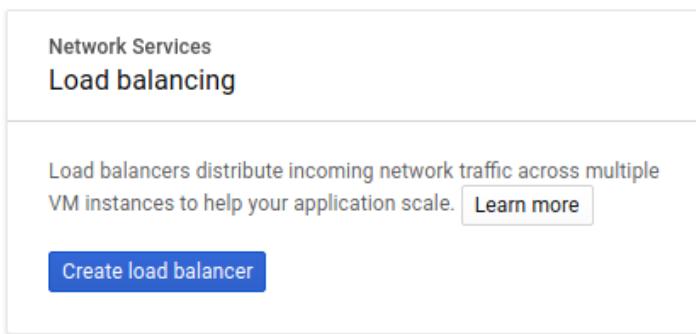
Imagine that you have a calculator web application where users can type a calculation into a box and the application prints out the correct answer (that is,  $2 + 2$  would return a page saying "4"). This calculator isn't all that useful but the important thing to note is that it doesn't need to store anything, so we say it's "stateless". Now let's imagine that for some reason, the calculator has gotten so popular that the single VM handling calculations is overwhelmed by the traffic.

In this scenario, you'd create a second VM running the exact same software as the first and then you'd create a load balancer to spread the traffic evenly across the two machines. The load balancer front-end would listen for HTTP traffic and forward all requests to the back-end, which is composed of the two VMs that handle calculations. Once the calculating is done, the VMs respond with the result and the load balancer forwards those responses back over the connection used to make the request.

It's important to note that the VMs don't have any idea there is a load balancer involved at all, since from their perspective requests come in and responses go out, so the load balancer looks just like any other client making requests. And the most convenient part of all of this is that as traffic grows even more we can simply turn on another "clone" and configure the load balancer back-end to also send requests to the new VM. If we wanted to automate this even further, we could configure the load balancer to use an auto-scaling instance group for its back-end, which would mean that you wouldn't have to worry about adding new capacity and registering it with the load balancer.

Let's run through the process of setting up a load balancer and splitting traffic across several different VMs. To do this, we'll re-purpose the VM instance group we learned about in "[Instance groups and dynamic resources](#)" as the backend of our load balancer. To create the load balancer, choose "Networking services" from the left-side navigation in the Cloud Console, and choose "Load balancing" after that. On that page, you should see a prompt to create a new load balancer.

**Figure 9.40. Prompt to create a new load balancer**



When you click to create a new load balancer, you'll see a few choices of the different types of load balancing you can do. For this exercise, we'll use HTTP(S) load balancing since we're trying to balance HTTP requests across our web application.

Click on "Start configuration" and you'll see a new page that has a place to choose a name (let's use `first-load-balancer`) and 3 steps towards configuring the new load balancer: "Backend", "Host and path rules", and "Frontend". Since we want to configure the load balancer to take HTTP requests and send them to our instance group as a back-end, let's start by setting our back-end configuration.

### 9.6.1 Back-end configuration

The first thing we need to do is create what's called a "back-end service". This service represents a collection of back-ends (typical VMs running in Compute Engine), which currently just refer to instance groups. To do this, click on the drop-down that says "Create or select backend services & backend buckets", and choose "Backend services", and then "Create a backend service", which open a new form where we can configure our new service.

**NOTE**

You may be wondering why we have this extra level of indirection, and why you have to create a service just to contain a single instance group. The simple answer is that even though you only have one instance group now, you may want to later add more groups behind the load balancer.

Back-end services allow the load balancer to always point at one thing, so that you can add and remove back-ends (*instance groups*) to and from the service.

Let's continue our naming pattern and call this `first-backend-service`, and then choose our `first-group` as the back-end. You'll notice there are quite a few extra options that should feel somewhat similar to the auto-scaling configuration of the instance group. While these are definitely similar, they actually have very different purposes.

In an instance group, these things like target CPU usage were used to tell Compute Engine when it should turn on more instances. In a load balancer, these things describe when the system should consider the back-end to be "over capacity". In other words, if the back-end as a whole goes over these limits, the load balancer will consider it to be "unhealthy", and stop sending requests to it. As a result, we should choose these targets pretty carefully to make sure we don't unnecessarily start returning errors that say the system is over capacity when it really isn't.

**Figure 9.41. Creating a new backend service**

Create backend service

Name [?](#)  
first-backend-service

Protocol: HTTP Named port: http Timeout: 30 seconds [Edit](#)

Backends

New backend [Delete](#) [X](#)

Instance group [?](#)  
first-group (us-central1-c)

Port numbers [?](#)  
80

Balancing mode [?](#)  
 Utilization  
 Rate

Maximum CPU utilization [?](#)  
80 %

Maximum RPS (Optional) [?](#)  
Max total RPS. Leave blank for unlimited RPS per instance

Capacity [?](#)  
100 %

[Less](#)

[Done](#) [Cancel](#)

[+ Add backend](#)

Health check [?](#)  
tcp-80 (TCP)  
port: 80, timeout: 5s, check interval: 5s, unhealthy threshold: 2 attempts

Session affinity [?](#) [Affinity cookie TTL](#) [?](#)  
None 0 seconds

[Advanced configurations](#)

Cloud CDN [?](#)  
 Enable Cloud CDN

Let's leave these settings in the default values for now. Next, you'll notice that we need to set a "health check" before we can actually save our backend service! Let's dig into what these are and how they work.

#### **CREATING A HEALTH CHECK**

Now that we've almost finished configuring our back-end service, we'll need to create something called a "Health check" (which we can do directly from the drop-down menu on the "New backend" form). Health checks are similar to the measurements that are taken to decide whether you need more VMs in an instance group, but are less about the metrics of the virtual machine (such as CPU usage) and instead focus on asking the application itself if everything is OK. In other words, a health check is more like the nurse asking you if you feel OK, whereas the other checks about things like CPU utilization are like the nurse checking your temperature.

These health checks can be a simple static response page to show that the web server is up and running, or something more advanced like a test of whether the database connection is working properly. Let's create a simple TCP check to see that port 80 is indeed open, and name the health check `tcp-80`. We'll leave all of the other settings the same as well (such as how long to wait between checks, how long to wait before timing out, and more).

**Figure 9.42. Creating a new health check**

Name

Description (Optional)

Protocol  Port

Proxy protocol

Request (Optional)

Response (Optional)

### Health criteria

Define how health is determined: how often to check, how long to wait for a response, and how many successful or failed attempts are decisive

Check interval <input type="text" value="5"/> seconds	Timeout <input type="text" value="5"/> seconds
Healthy threshold <input type="text" value="2"/> consecutive successes	Unhealthy threshold <input type="text" value="2"/> consecutive failures

Once you create the health check, your back-end service is ready, so click "Create" to see the summary of your back-end service and we can move on to the front-end configuration.

## 9.6.2 Front-end configuration

When configuring our front-end, the first thing we'll look at is the host and path rules, and since we want all requests to be handled by all VMs there's really no need to do any special configuration here. On the other hand, if you had a situation where certain more complicated calculations had to be handled by a specific back-end the "Host and path rules" section would be where you route specific URLs to that back-end and all others to the regular back-ends. Since there's nothing for us to do here, we can continue on to front-end configuration, where we define what types of requests the load balancer should listen for and route to our back-end service.

In our example case, we want to handle normal HTTP traffic, which is the default setting. The only potential issue is that the IP address for our load balancer may change from time to time (that's what it means where it says the IP address is "Ephemeral"). If we were setting this service up to have a domain name like `mycalculator.com`, we'd create a new static IP address (by choosing "Create IP address" from the drop down) and then that IP address would always be the same, so we could add it to a DNS entry. For now (and because this is just a demonstration), we'll stick with an ephemeral IP for our load balancer.

**Figure 9.43. Our simple front-end configuration**

## Frontend configuration

Specify an IP address, port and protocol. This IP address is the frontend IP for your clients requests. For SSL, a certificate must also be assigned.

The screenshot shows a configuration dialog box titled "New Frontend IP and port". It includes fields for "Name (Optional)" (containing "lowercase, no spaces"), "Add a description", "Protocol" (set to "HTTP"), "IP version" (set to "IPv4") and "IP address" (set to "Ephemeral"), "Port" (set to "80"), and "Done" and "Cancel" buttons. Below the dialog is a button labeled "+ Add Frontend IP and port".

As you can see, it's possible to create multiple front-end configurations if you wanted to listen on multiple ports or multiple different protocols. For this demonstration we'll stick to just boring old HTTP on port 80. Click "Done" and then you can jump to the final step where we can review all of the configurations we have set up to verify that they're correct.

### 9.6.3 Reviewing the configuration

As you can see, we have an instance group called `first-group`, which has no host- or path-specific rules, and exposed on an ephemeral IP address on port 80, and a health check called `tcp-80` which will be used to figure out which of the instances in the group are able to handle requests.

**Figure 9.44. Summary of our load balancer configuration**

#### Review and finalize

##### Backend

###### Backend services

###### 1. first-backend-service

Endpoint protocol: **HTTP** Named port: **http** Timeout: **30 seconds** Health check: **tcp-80** Session affinity: **None**

Cloud CDN: **disabled**

Advanced configurations

Instance group	Zone	Autoscaling	Balancing mode	Capacity
first-group	us-central1-c	Off	Max CPU: 80%	100%

##### Host and path rules

Hosts ^	Paths	Backend
All unmatched (default)	All unmatched (default)	first-backend-service

##### Frontend

Protocol ^	IP:Port	Certificate
HTTP	EPHEMERAL:80	—

Clicking "Create" will start the process of creating the load balancer, the health checks, assigning an ephemeral IP, and getting ready for your load balancer to start receiving requests.

**Figure 9.45. Our newly created load balancer**

**Load balancing**    **CREATE LOAD BALANCER**    **REFRESH**

**Load balancers**    **Backends**    **Frontends**

Load balancer	Protocol
<b>first-load-balancer</b>	

**Details**    Monitoring    Caching

**Frontend**

Protocol	IP:Port	Certificate
HTTP	35.186.226.157:80	—

**Host and path rules**

Hosts	Paths	Backend
All unmatched (default)	All unmatched (default)	first-backend-service

**Backend**

**Backend services**

1. **first-backend-service**

Endpoint protocol: **HTTP**    Named port: **http**    Timeout: **30 seconds**    Health check: **tcp-80**    Session affinity: **None**

Cloud CDN: **disabled**

Advanced configurations

Instance group	Zone	Healthy	Autoscaling	Balancing mode	Capacity
first-group	us-central1-c	0 / 0	Target CPU usage 50%	Max CPU: 80%	100%

After a couple minutes (while the health checks determine that the backend is ready), visiting the address of your load balancer in a browser should show you the Apache 2 default page. In other words, seeing this page shows us that the request was routed to one of your VMs in the instance group!

Now imagine tons of people sending requests to the load balancer. As the number of requests goes up and the CPU usage of a given VM increases, the auto-scaling instance group will turn on more VMs just as you learned earlier. The big difference is that since you're routing all requests through your load balancer, as more requests come in, they'll automatically be balanced across all of the VMs that were turned on by the instance group! This means that you now have a truly auto-scaling system that will handle all the requests you could possibly throw at it, and it will grow and shrink automatically, and distribute the request load automatically!

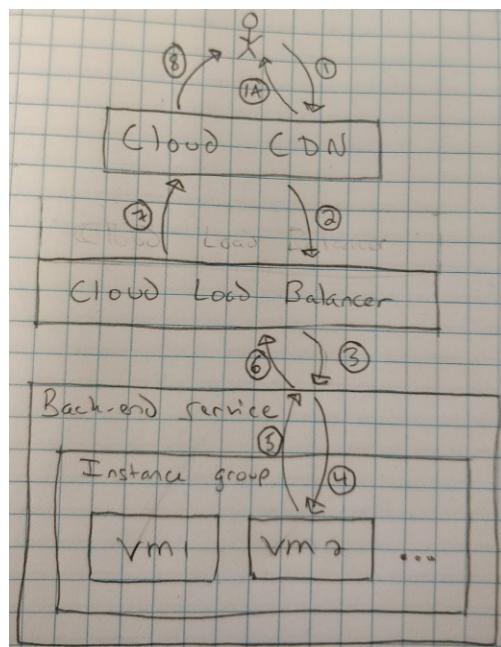
Now that we have the functionality working, it might make sense to look at ways to optimize this configuration. One common "low-hanging fruit" that we can pick off is figuring out how to avoid duplicating work by caching results whenever possible, so let's see how we might use that principle to make our system more efficient.

## 9.7 Cloud CDN

In any application it's pretty likely that there will be lots of identical requests to the servers running the application. And sometimes identical requests also yield identical responses. Though this is most common with static content like images this can apply to dynamic requests as well to avoid duplicating effort. It turns out that Google Cloud Platform has something called Google Cloud CDN that can automatically cache responses from back-end services, and is designed to work with Compute Engine and the load balancer that we just created.

Cloud CDN effectively sits between the load balancer and the various people making requests to the service, and attempts to "short-circuit" the request. This means that if a request can be handled by Cloud CDN, the rest of your system (that is, the VMs that actually serve data) will never even see the request, as shown below in (1A).

**Figure 9.46. The flow of a request with Cloud CDN enabled**

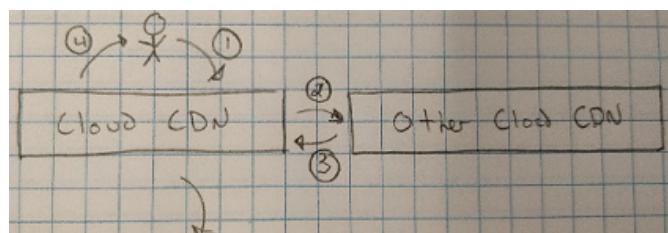


As you can see, a request starts from a user (1) and if it can be handled by Cloud CDN, the response is immediately returned (1A). The load balancer, back-end service, instance group, and VM instances never even see the request. If the request is not handled by Cloud CDN, it follows the traditional request path of visiting the load

balancer (2), then the back-end service (3) which routes the request to an individual VM (4), and then the response flows back over the same path to the load balancer (5, 6). After the response is returned by the load balancer, instead of going directly to the end-user as we saw previously, the response flows through Cloud CDN (7) and from there back to the end-user (8). This allows Cloud CDN to inspect the response and determine if it can be cached so that a future identical request can be handled via the short-circuit route (1 and 1A).

In addition to this, if a request appears that it *could be* cached but the given Cloud CDN end-point doesn't have a response to send back, it has the ability to ask other Cloud CDN end-points if they've handled this request before and have a response. If the response is available somewhere else in Cloud CDN, the local instance can return that value as well as storing it locally for the future, shown below.

**Figure 9.47. Cloud CDN can look to other caches for a response**

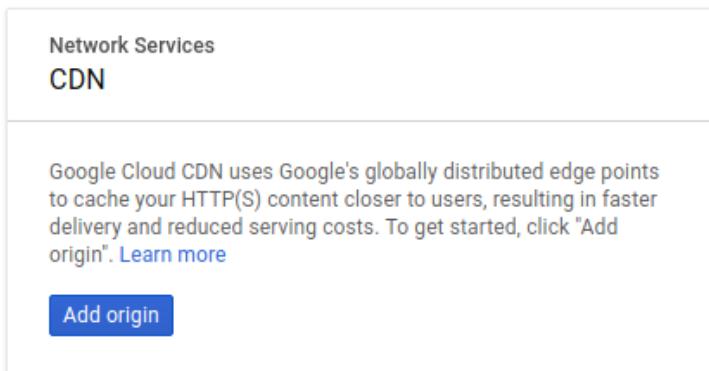


As you can see, if a request can be cached but happened to be cached elsewhere, the request will flow to the nearest Cloud CDN end-point (1) and over to another Cloud CDN (2) which has a response. The response will then flow back to the original endpoint (3) where it is stored locally and ultimately returned back to the user (4). Let's look at how we can enable Cloud CDN for the load balancer we created previously.

### 9.7.1 Enabling Cloud CDN

In the left-side navigation of the Cloud Console, choose "Cloud CDN" from the "Network Services" section. Once there, you'll see a form prompting you to add a new origin for Cloud CDN.

**Figure 9.48. Prompt to add an origin to Cloud CDN**



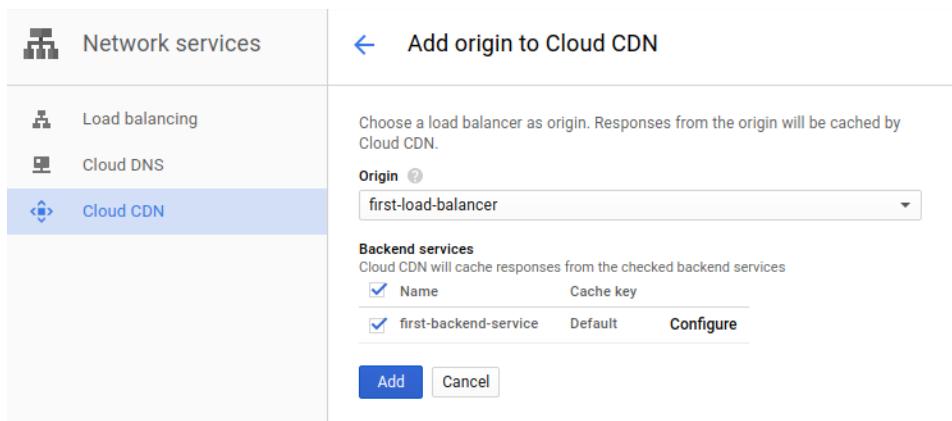
Choosing the load balancer we created previously will populate a list of back-end services that can be cached by Cloud CDN.

**Figure 9.49. Choosing the correct load balancer to cache using Cloud CDN**

The screenshot shows the 'Add origin to Cloud CDN' dialog. On the left, a sidebar lists 'Network services' options: Load balancing, Cloud DNS, and Cloud CDN, with 'Cloud CDN' selected. The main area displays instructions to choose a load balancer as the origin for caching. A dropdown menu titled 'Select origin' is open, showing a 'Filter' input field and a list containing 'Create a load balancer...' and 'Load balancer first-load-balancer'.

Here we can also add some extra configuration to the specific back-end by clicking "Configure" (not shown). This allows you to customize the way in which pages are cached, by saying, for example, that pages served over HTTP should be cached separately from pages served over HTTPS. By clicking "Add", Cloud CDN is enabled on the load balancer for the selected back-end services.

**Figure 9.50. Ensuring the right back-ends are selected to be cached**



At this point we can see in the list that our load balancer (`first-load-balancer`) is being cached by Cloud CDN and a specific set of back-ends.

**Figure 9.51. Listing load balancers actively cached by Cloud CDN**

Network services	Cloud CDN	+ ADD ORIGIN	REFRESH
<ul style="list-style-type: none"> <li><span>Load balancing</span></li> <li><span>Cloud DNS</span></li> <li><span>Cloud CDN</span></li> </ul>	<p>Origin name ▾</p> <p><input checked="" type="checkbox"/> first-load-balancer</p> <p>Backends</p> <p>first-backend-service (Enabled)</p> <p>Cache hit ratio n/a</p> <p>⋮</p>		

We can also see that Cloud CDN is enabled by looking at the details of our load balancer, and noting the "Cloud CDN: **enabled**" annotation listed under the back-end service, shown below.

**Figure 9.52. Viewing the load balancer shows "Cloud CDN: enabled"**

The screenshot shows the Google Cloud Load Balancing interface. At the top, there are buttons for 'CREATE LOAD BALANCER' and 'REFRESH'. Below this, a navigation bar has 'Load balancers' selected, with 'Backends' and 'Frontends' as other options. The main area is titled 'Load balancer' and 'Protocol'. A specific load balancer named 'first-load-balancer' is selected, indicated by a green checkmark icon. The 'Details' tab is active, showing a 'Frontend' section with one entry: 'HTTP' protocol on port '35.186.226.157:80'. Below this is a 'Host and path rules' section, which is currently empty. The 'Backend' section shows a single service named 'first-backend-service'. Underneath it, detailed configuration for this service is shown, including the endpoint protocol as 'HTTP', named port as 'http', timeout as '30 seconds', health check as 'tcp-80', session affinity as 'None', and 'Cloud CDN: enabled'. There is also a link for 'Advanced configurations'. The 'Capacity' table lists the instance group 'first-group' with one healthy instance in the 'us-central1-c' zone.

Instance group ^	Zone	Healthy	Autoscaling	Balancing mode	Capacity
first-group	us-central1-c	0 / 1	Target CPU usage 50%	Max CPU: 80%	100%

Let's take a moment now to look at the details of how Cloud CDN decides what pages can and cannot be cached.

## 9.7.2 Cache control

By default, Cloud CDN will attempt to cache all pages that are "allowed". This definition follows mostly from IETF standards (such as RFC-7234) meaning that the rules are what you'd expect if you're familiar with HTTP caching generally. For example, the follow all must be true in order for a response to a request to be considered "cache-able" by Cloud CDN:

1. Cloud CDN must be enabled
2. The request uses the `GET` HTTP method
3. The response code was "successful" (e.g., `200`, `203`, `300`, etc)

4. The response has a defined content length or transfer encoding (specified in the standard HTTP headers)

In addition to these rules, the response also must explicitly state its caching preferences using the `Cache-Control` header (e.g., set it to `public`) **and** it must explicitly state an expiration using either a `Cache-Control: max-age` header or an `Expires` header.

Even further, Cloud CDN will actively not cache certain responses if they break some other rules, such as:

1. The response has a `Set-Cookie` header
2. The response size is greater than 10 MB
3. The request or response has a cache control header indicating it should not be cached (e.g., set to `no-store`).

In addition, as we noted above, you can also configure whether you want to distinguish between URLs based on the scheme (e.g., HTTP vs HTTPS), query string (e.g., stuff after the `?` in a URL), and more, to get fine grained control over how different responses are cached. That said, the moral of the story here is that Cloud CDN will follow the rules that most browsers and load balancing proxy servers follow with regard to caching, but will do the actual caching work in a location much closer to the end-user than your VMs will typically be.

Finally, there may be times when we have cached something and need to forcibly "uncache" it. That is, we want the request to be sent to the actual back-end service rather than be handled by the cache. This is a common scenario when, for example, we deploy new static files such as an updated `style.css` file, and don't want to wait the content to expire from the cache.

To do this, we can use the Cloud Console by clicking on the "Cache invalidation" tab. Here we can enter a pattern to match against (such as `/styles/*.css`) and all matching cache keys will be evicted. This means that on a subsequent request for these files they will be fetched first from the back-end service and then cached as usual.

**Figure 9.53. Invalidating a particular cached URL**

The screenshot shows the Cloud CDN Cache Invalidation interface. At the top, there's a navigation bar with a left arrow, the text "Origin details", and buttons for "EDIT" and "REMOVE". Below this is a section titled "first-load-balancer" with a green checkmark icon. A horizontal menu bar includes "Settings", "Monitoring", and "Cache invalidation", with "Cache invalidation" being the active tab. Underneath, a sub-header reads "Path pattern to invalidate" with a question mark icon. A note states: "Cached objects matching the path pattern will be invalidated in all Cloud CDN caches". Below this is a text input field containing the placeholder "For example: /path.jpg or www.example.com/path/\*" and a blue "Invalidate" button.

At this point you should have a pretty good grasp on most of the things that Compute Engine can do. This means it's time to look at how much all of this costs to use. And as you might guess, pricing can be a bit complicated given all of the various ways you can use Compute Engine.

**NOTE**

Before we get into looking at the how much everything costs, now is a great time to turn off any resources you created while reading this chapter so that you don't end up getting charged unnecessarily!

## 9.8 ***Understanding pricing***

While the basic features of Compute Engine have pretty straightforward prices, some of the more advanced features can get complicated, and even more complicated when you consider a very important discount available for sustained use. Let's start by looking at the simple parts, and then we'll move into the more complicated aspects of Compute Engine pricing.

There are three pricing aspects to consider with Compute Engine:

1. Computing capacity using CPUs and memory
2. Storage using persistent disks
3. Network traffic leaving Google Cloud

Let's start by looking at computing costs.

### 9.8.1 ***Computing capacity***

The most common way of using Compute Engine is with a predefined instance type, such as `n1-standard-1` which we used in [chapter 1](#). By turning on an instance of a particular predefined type, you are charged a specific amount every hour for the use of the computing capacity. That capacity is a set amount of CPU time, which is measured in vCPUs (a virtual CPU measurement), and memory, which is measured in GB. Each predefined type has a specific number of vCPUs, a specific amount of memory, and a fixed hourly cost. Table 9.2 shows a brief summary of common instance types and how much they cost on an hourly and monthly basis in the `us-central1` region. As expected, more compute power and memory means more cost.

**Table 9.2. Cost and details for some common instance types**

Instance type	vCPUs	Memory	Hourly cost	Monthly cost
<code>n1-standard-1</code>	1	3.75 GB	\$0.0475	about \$25
<code>n1-standard-2</code>	2	7.5 GB	\$0.0950	about \$50
<code>n1-standard-8</code>	8	30 GB	\$0.3800	about \$200
<code>n1-standard-16</code>	16	60 GB	\$0.7600	about \$400
<code>n1-standard-64</code>	64	240 GB	\$3.0400	about \$1,500

For the times where one of these instance types doesn't quite fit your needs, typically this is where you need a lot of memory but little CPU (or vice-versa), there are other

predefined machine types which have a pricing structure similar to the table above.

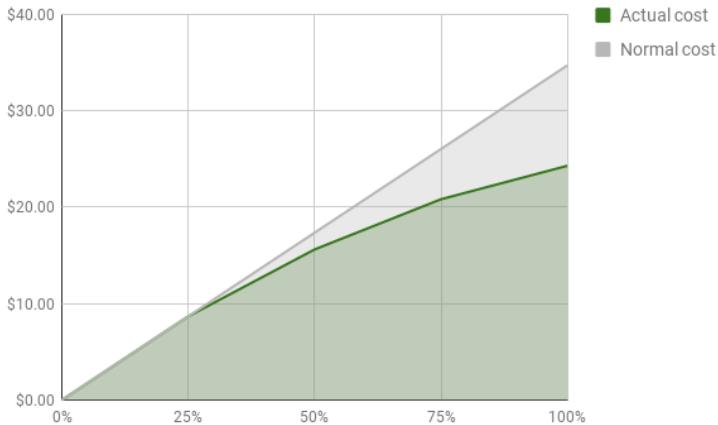
For the truly unusual scenarios where no predefined types fit you can design your own custom machine profile. For example, imagine that you want to store a huge amount of data in memory, but don't need the machine to do anything else than act as a cache. In that case, you might want to have a lot of memory but not a lot of CPU. Of the predefined types, your choices are pretty limited (either too little memory or too much CPU). To handle situations like these, you can customize machine types to the right size with a specific number of vCPUs and memory, where each CPU costs about \$0.033 per hour and each GB of memory costs \$0.0045 per hour.

If you've been doing the math along the way, you may notice that these numbers don't quite line up as you'd expect. For example, we said that the one month of our n1-standard-8 instance costing \$0.38 per hour. The problem comes out when we look at the cost per month, which is listed as "about \$200", but \$0.38 per hour \* 24 hours per day \* ~30 days per month is about \$270, not \$200! It turns out that Compute Engine gives you a discount when you use VMs for a sustained period of time.

## **9.8.2 Sustained use discounts**

Sustained use discounts, are a bit like getting a discount for buying in bulk. What makes these discounts particularly cool though is that you don't have to commit to buying anything. Sustained use discounts work by looking back over the past month and figuring out how many VM-hours you used, and computing the overall hourly price based on that, with a "bulk discount" if you happened to use VMs for a long period of time. Think of it a bit like paying less on your electricity bill as you consume more throughout the month. As you use more electricity, the per-unit cost drops until you're paying wholesale prices.

Sustained use discounts have 3 steps, with a maximum net discount of 30% for the month. The way it works is by applying a new base rate for the second, third, and fourth quarters of each month. In other words, after a VM has been running for 25% of the month the following 25% of the month is billed at 20% off the regular rate. The next 25% is billed at 40% off, and the final 25% is billed at 60% off. When you put this all together, running 100% of the month means that you end up paying 30% less than you would've without the discount, which you can see in the following chart.

**Figure 9.54. Sustained use discount visualized**

In this chart the top line is the "normal cost" which follows a straight line. The "actual cost" follows the bottom line where the slope of the curve decreases over time. At the end of the month, the "actual cost" line ends up being about 30% lower. This example is pretty straight forward, but what if you have two VMs that you run for half of the month each? Or what if you have a custom machine type? What about when you have auto-scaling turned on? It turns out that this all gets pretty complicated, so rather than trying to enumerate all of these examples, it might be better to communicate the underlying principle used when doing these calculations.

First, Compute Engine tries to infer a consistent amount of usage even if you reconfigure your machines frequently. In other words, it looks at the actual number of VM instances that were running, and tries to combine them into a more dense configuration to figure out the minimum number of simultaneously running VM instances. Using this condensed graph of "inferred instances", Compute Engine will try to calculate the maximum discount possible given the configuration. If you're terrified of that, you're not alone, so let's make it more clear with a picture.

**Figure 9.55. Inferred instances and discount computation**

Actual use				Inferred Instances				Discount computation			
Week 1	Week 2	Week 3	Week 4	Week 1	Week 2	Week 3	Week 4	Week 1	Week 2	Week 3	Week 4
VM 1				VM 1		VM 3		VM 1		VM 3	
	VM 2				VM 2		VM 4	VM 2		VM 4	
		VM 3				VM 5		VM 5			
			VM 4								
			VM 5								

In the example above, you can see that we had 5 instances running throughout the month, with some different overlaps (e.g., VM 4 was running at the same time as VM 3 and VM 5). First, Compute Engine condenses or "flattens" the images to get the minimum number of "slots" you'd need in order to handle this usage scenario. In our

example, rather than turning on VM 3 in week 3 as we did, we could've just recommissioned VM 1 to do the same work. This means that we can flatten these two together and treat them as a single run of "sustained use" even though we actually turned on machine off and another one on.

Once we have the inferred instances, we slide everything to the start of the month to figure out how to apply the discounted rate for each segment as you saw in Figure 9.55. We use this final condensed and shifted graph to compute how much of a discount can be applied.

In other words, the more time that multiple VMs are running concurrently, the less condensation we can do when calculating a discount. In short this means that not all VM hours cost the same. Running a single VM for a full month (~730 hours) might cost the same as running 730 machines for one hour each, but only if all of those machines run in order (turn one off and turn another one at the same time). If you run 730 machines all for the exact same hour, you won't see any discounts at all, so the overall cost will be 30% more expensive. For example, Figure 9.56 shows us running more instances at the same time (only VM 2 and VM 4 don't overlap at all), which means we can't condense as much and therefore a smaller discount is applied.

**Figure 9.56. Less condensation possible when there's more overlap**

Actual use				Inferred Instances				Discount computation			
Week 1	Week 2	Week 3	Week 4	Week 1	Week 2	Week 3	Week 4	Week 1	Week 2	Week 3	Week 4
VM 1				VM 1				VM 1			
VM 2				VM 2		VM 4		VM 2		VM 4	
	VM 3				VM 3			VM 3			
		VM 4				VM 5		VM 5			
		VM 5									

Finally, before we move onto storage costs, it's important to remember that all of the cost numbers so far have been based on resources based in the United States. Compute Engine offers lots of different regions in which you can run VMs, and it turns out that the prices in the various regions aren't all the same. The reason behind this is nothing more than variable costs to Google (in the form of electricity, property, etc), but also tends to relate to available capacity. As a result, the costs for the different resources (such as predefined machine types, custom machine type vCPU and memory, as well as extended memory) varies quite a bit from one region to the next. For example, the following table shows a few different prices per vCPU and GB of memory in different regions.

**Table 9.3. Prices per vCPU based on location**

Resource	Iowa	Sydney	London	Oregon
vCPU	\$0.033174	\$0.04488	\$0.040692	\$0.033174
GB memory	\$0.004446	\$0.00601	\$0.005453	\$0.004446

To put that in perspective, this means that a VM in London might cost around 25% more than the same VM in Iowa. (For example, a n1-standard-16 costs about \$388 per

month in Iowa but about \$500 per month in London.) In general it'll be cheapest to your VMs in US-based regions (like Iowa), and you should typically only run VMs in other regions if you have a meaningful reason for doing so, such as needing low latency to your customers in Australia, or having concerns about data living outside of the EU.

### 9.8.3 Preemptible prices

In addition to regular list prices for VMs and sustained use discounts, preemptible VMs have special price reductions in exchange for the restrictions on these instances. As always, these prices vary from location to location, but the structure remains the same with per-hour prices for the use of the instance. Shown below are some example prices for a few different instance types for a few popular locations.

**Table 9.4. Preemptible instance hourly prices for a few locations**

Instance type	Iowa	Sydney	London	Oregon
n1-standard-1	\$0.01	\$0.01349	\$0.01230	\$0.01
n1-standard-2	\$0.02	\$0.02698	\$0.02460	\$0.02
n1-standard-4	\$0.04	\$0.05397	\$0.04920	\$0.04
n1-standard-8	\$0.08	\$0.10793	\$0.09840	\$0.08
n1-standard-16	\$0.16	\$0.21586	\$0.19680	\$0.16
n1-standard-32	\$0.32	\$0.43172	\$0.39360	\$0.32
n1-standard-64	\$0.64	\$0.86344	\$0.78720	\$0.64

As you can see, these prices do indeed vary by location, however they are around 4-5x cheaper than the list hourly prices. This means that if you happen to be cost conscious, it might make sense to see if you can find a way to make preemptible instances work for your project.

Now that we've gone through the hard part, let's finish up by looking at the easier aspects of Compute Engine pricing: storage and networking.

### 9.8.4 Storage

Compared to VM pricing, storage pricing is a piece of cake. As you learned earlier, there are a few different classes of persistent disk storage that you can use with your VM instances, each with different performance capabilities. Each of these different classes has a different cost (with SSD disks costing more than standard storage), and the rates tend to differ depending on the region, just like VM prices. The following table shows some of the different rates per GB per month of disk storage for the same four regions we discussed before (Iowa, Sydney, London, and Oregon).

**Table 9.5. Data storage rates based on location and disk type**

Disk type	Iowa	Sydney	London	Oregon
Standard	\$0.040	\$0.054	\$0.048	\$0.040
SSD	\$0.170	\$0.230	\$0.204	\$0.170

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/google-cloud-platform-in-action>

**Licensed to Asif Qamar <asif@asifqamar.com>**

To put that in perspective, this means that a solid-state persistent disk in London might cost around 20% more than the same disk in Iowa. (For example, a 1 TB SSD costs about \$170 per month in Iowa but about \$200 per month in London.) In general, just like with VMs, it'll be cheapest to keep your data in US-based regions and you should typically keep persistent disks in other regions only if you have a good geographical reason to do so. Additionally, as you can see, the price of SSDs far outstrips the cost of a standard disk, meaning you should use SSDs only if you have a strong performance need to do so.

For example, if you have a 1 TB SSD disk in Sydney that you want to back-up, you might want to consider uploading the important data somewhere else (like a Cloud Storage bucket), as it will cost you about \$200 to have it as a disk (or \$35 if you save it as a snapshot only), but only about \$15 if you store it in Cloud Storage or even as low as \$10 if you use Nearline storage described in "[Nearline storage](#)". Now that we've gone through how much it costs to store data on persistent disks, let's look at the final piece of the puzzle: networking costs.

### **9.8.5 Network traffic**

Typically, when you build something using Compute Engine you don't intend for it to live entirely in a vacuum with no communication with the outside world. On the contrary, most of the VMs you create will be sending data back to customers, like images or videos or other web pages. While the incoming data is always free, unfortunately, sending this data around the world isn't free. Just like VM prices, network cables around the world have varying costs, which means that the outgoing (or *egress*) networking costs vary depending on where they exit from Google's network. This means that sending data out of places like Iowa will cost less than out of places like Sydney. Additionally, sending data from one Google zone to another isn't free either, since it's effectively a "fast pipe" across long distances on Google-owned network cables.

To understand costs for networking, we need to look at both where the traffic comes from, as well as where the traffic is going. This is because Google uses its own network infrastructure to make sure that your data gets to its destination as quickly as possible, which, as you'd expect, costs more to go a further distance. For example, getting a packet from Iowa to New York City is far less costly than getting a packet from Iowa to Australia.

That said, it turns out that the networking prices for traffic to Australia or mainland China is the same regardless of the source, but it's possible that this could change down the line. For all other locations (basically everywhere except those two) the cost varies depending on where the VM is sending the data from.

Additionally, just like when buying bulk from Costco gets you a discount, traffic prices go down as you send more data. For Compute Engine, there are three different prices for each GB of traffic: one for the first TB (which should be most of us), another price for the next 9 TB (more than 1 TB, up to 10 TB), and then a final bulk price for all data after the first 10 TB. The table below lists some example prices from the same

four regions as before (Iowa, Sydney, London, and Oregon) when sending data to most locations.

**Table 9.6. Network prices for most locations**

Price group	Iowa	Sydney	London	Oregon
First TB	\$0.12	\$0.19	\$0.12	\$0.12
Next 9 TB	\$0.11	\$0.18	\$0.11	\$0.11
Above 10 TB	\$0.08	\$0.15	\$0.08	\$0.08

For sending data to those two special places (Australia and mainland China), the prices are currently the same, regardless of the origin.

**Table 9.7. Network prices from anywhere to special places (mainland China and Australia)**

Price group	To mainland China	to Australia
First TB	\$0.23	\$0.19
Next 9 TB	\$0.22	\$0.18
Above 10 TB	\$0.20	\$0.15

To put this in perspective, let's imagine that we have a video file that we want to serve on our web site which is about 50 MB in size. Let's assume that we get 10,000 different people watching the video, so that's 10,000 hits of 50 MB each for a total of 500 GB of data all together. Typically this would be enough to figure out the cost, but as we learned earlier we need to consider where these hits are coming from since the destination of our 50 MB video costs more for certain places than it does for others. We also need to know where the VM serving the video is running, since the source of the data matters as well!

Let's imagine that the video is on a VM in Iowa (so we'll use the Iowa egress cost table above), and that we have 10% of the hits from Australia, 10% from mainland China, and the other 80% of hits are coming from elsewhere in the world, such as New York, Hong Kong (not part of mainland China), and London. Our cost calculations can be broken down as shown in the table below.

**Table 9.8. Breakdown of cost calculations based on location**

Location	GB served	Cost per GB	Total cost
China	50 GB	\$0.23	\$11.50
Australia	50 GB	\$0.19	\$9.50
Elsewhere	400 GB	\$0.12	\$48.00
Total			\$69.00

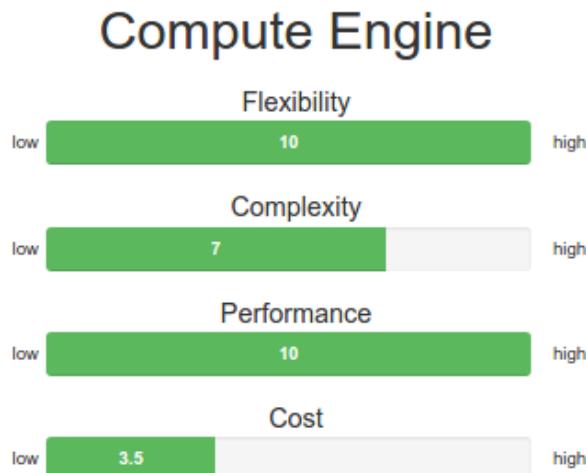
As you can see, most of our data transfer costs of the 500 GB that we sent around the world came from all of the other locations, but the special destinations held a disproportionate amount of the total cost. In this case, mainland China destinations were 10% of the traffic, but resulted in about 16% of the total cost! Obviously there's

not a whole lot you can do to limit who downloads what, short of restricting access to certain regions by IP address, however it's worth keeping in mind that network costs to special destinations can add quite a bit if you happen to handle a lot of traffic from those places.

## 9.9 When should I use Compute Engine?

To figure out whether Compute Engine is a good fit, let's start by looking at the score card which summarizes the various computing aspects that we might care about.

**Figure 9.57. Compute Engine scorecard**



### 9.9.1 Flexibility

The first thing to note with Compute Engine is that it is as flexible as you can get in a cloud computing environment. This is simply because Compute Engine focuses specifically on providing general purpose infrastructure for you to build on top of.

While you do have access to fancier things, like auto-scaling Instance Groups and load balancing, those are extras that you can use if they fit your needs. If, for example, you found that you needed some special load balancing feature, you could opt to not use a hosted Load Balancer and instead turn on your own VM to run the load balancing software. That said, there are limits to Compute Engine, but those limits tend to be standard across all cloud hosting providers. For example, it's currently not possible to bring your own hardware into a cloud data center, which means that you won't be able to run your own hardware-based load balancer (such as one of the products from F5).

### 9.9.2 Complexity

As you can see from the size of this chapter, Compute Engine is far from simple. That said, if all you want is a virtual machine that runs your software you could've stopped reading this chapter a long while ago. On the other hand, if you want to use the other

more powerful features of Compute Engine, things can get much more difficult very quickly. For example, in order to take advantage of Compute Engine's auto-scaling capability you first have to understand how instance templates work, then load balancers, and finally health checks since without these it's not possible to get the full benefit of an auto-scaled system. Put more simply, since you can get going relatively quickly with Compute Engine, the *overall difficulty* is somewhat lower than other computing systems which require you to learn everything before being even trivially useful.

### 9.9.3 Performance

When it comes to performance, Compute Engine scores particularly well. Being as close to bare metal as you'll get in Google Cloud means that you have as few abstraction layers between your code and the physical CPU doing the work. In other computing systems (for example, App Engine Standard or Heroku), there are simply more layers of abstraction between the physical CPU and your code which means they'll be slightly less efficient, and therefore have slightly worse performance.

This is not to say that other managed computing platforms aren't useful or are materially inefficient, however the nature of their design (that is, being higher up the stack) means that there's simply more work to be done, so CPU cycles that would be spent on your code are spent instead on other things.

### 9.9.4 Cost

Finally, Compute Engine is relatively low on the cost scale, given that you're only paying for raw virtual machines and disks. Additionally, computing resources are discounted as you use them throughout the month, meaning that you can get large discounts on resources without having to reserve them ahead of time. Notice in particular that Compute Engine's rates are hourly, meaning your costs should be much easier to estimate compared to a fully managed service like Cloud Datastore that depends on how many requests you make to the service.

### 9.9.5 Overall

Now that you can see how Compute Engine works, let's look at how we might use it for each of our sample applications (the to-do list, InstaSnap and E\*Exchange), to see how they stack up. It's important to note that Compute Engine **will** work for each of the examples, so we're going to look also at whether you might want to use "just the basics" or some of the more advanced features.

#### To-do list

Our to-do list application, being a small toy and unlikely to see tons of traffic, is unlike to need any of the advanced features of Compute Engine like auto-scaling or preemptible VMs. This is simply because the traffic patterns we expect are nothing more than going from zero (no one using the app) to a moderate amount (a few people using it at peak hours). This means that if you use Compute Engine you're buying into a guaranteed price, and have to learn and configure quite a bit in order to use the

automatic scaling features. It also means that there's no way to "lay dormant" for the time where there is no traffic to the to-do list application.

As a result, while you could use Compute Engine, it's likely you'll only turn on a single VM and leave that running around the clock. For this type of hobby project that won't have a lot of traffic in total, nor a lot of volatility in the traffic patterns, a fully-managed system like App Engine (which you'll learn about later) might be a better fit.

**Table 9.9. To-do list application computing needs**

Aspect	Needs	Good fit?
Flexibility	Not all that much	Overkill
Complexity	Simpler is better	Not so good
Performance	Low to moderate	Slightly overkill during non-peak time
Cost	Lower is better	Not ideal, but not awful either

Overall, Compute Engine is an acceptable fit if you only use the basic aspects of the platform, however it's likely to cost more than necessary and unlikely that the application will make use of the more advanced features available.

#### **E\*Exchange**

E\*Exchange, the online trading platform, has more complex features, as well as much more flexibility in what makes a good fit for running the computing resources.

**Table 9.10. E\*Exchange computing needs**

Aspect	Needs	Good fit?
Flexibility	Quite a bit	Definitely
Complexity	Fine to invest in learning	OK
Performance	Moderate	Definitely
Cost	Nothing extravagant	Definitely

First, an application like this needs a quite a bit more flexibility than the to-do list. For example, rather than just handling requests to look at and modify to-do items, we may need to run background computation jobs to collect statistics and e-mail them to users as reports. This is a fine fit for Compute Engine which is able to handle general computing needs like this.

When it comes to complexity, if we're building something as complex as this trading application, we probably have the time to invest in learning about the system's more complex features. This means it's not necessarily a bad fit to have a complex system to understand.

Next, our performance needs are not extraordinarily large, but they aren't tiny either. In other words, it seems plausible that we may want to use some of the larger instance types so that viewing pages in a browser feels quick and snappy. Similarly, our budget for an application like this is not necessarily enormous, but we do have a reasonable

budget to spend on computing resources. As a result, since Compute Engine's prices are pretty reasonable, it's unlikely that the bill will come to anything extravagant for our application serving web pages and running reports.

All of this means that E\*Exchange is actually a pretty good fit to use Compute Engine, offering the right balance of flexibility with a relatively low cost and solid performance.

### **InstaSnap**

InstaSnap, the very popular social media photo sharing application, has a few requirements that seem to fit really well and a few others that are a bit off.

**Table 9.11. InstaSnap computing needs**

Aspect	Needs	Good fit?
Flexibility	A lot	Definitely
Complexity	Eager to use advanced features	Mostly
Performance	High	Definitely
Cost	No real budget	Definitely

As you can imagine, for InstaSnap we need a lot of flexibility and a lot of performance, which is a great fit for Compute Engine. We're also willing to pay for the best stuff, and all the venture capital funding we get means that we have a pretty big budget, making Compute Engine fit here also.

We also are really interested in the bleeding-edge of cool features such as auto-scaling and managed load balancing, making Compute Engine a good fit here also. That said, you'll learn later that while Compute Engine's advanced features are great, there are other systems that offer even more advanced orchestration, such as Kubernetes Engine. This means that it's a reasonably good fit, but there may be better options.

## **9.10 Summary**

- Virtual machines are virtualized computing resources, a bit like slices of a physical computer somewhere.
- Compute Engine offers virtual machines for rent priced by the hour (billable by the second) as well as persistent replicated disks to store data for the machines.
- Compute Engine can automatically turn on and off machines based on a template allowing you to automatically scale your system up and down.
- With highly scalable workloads where workers can turn on and off quickly and easily, preemptible VMs can reduce costs significantly with the caveat that machines can live no longer than 24 hours and may die at any time.
- Compute Engine is best if you want fine-grained control of your computing resources and want to be as close to the physical infrastructure as possible.

# 10

## *Kubernetes Engine: Managed Kubernetes Clusters*

### ***This chapter covers:***

- What is a container? and Docker? and Kubernetes?
- How does Kubernetes Engine work?
- Setting up a managed Kubernetes cluster using Kubernetes Engine
- Upgrading cluster nodes
- Resizing a cluster
- When Kubernetes Engine is a good fit

### **10.1 What are containers?**

A common problem in software deployment is the final packaging of all your hard work into something that is easy to work with in a production setting. A container is an infrastructural tool aimed at solving this problem by making it easy to package up your application, its configuration, and any dependencies into a standard format. By relying on containers, it becomes really easy to share and replicate a computing environment across many different platforms, and also happens to act as a unit of isolation which means that you don't have to worry about competing for limited computing resources as each container is isolated from the others.

If all of this sounds intimidating, don't worry: containers are pretty confusing when you're just learning about them. Let's walk through each piece, one step at a time, starting with configuration.

### 10.1.1 Configuration

If you've ever gone to deploy your application and realized you had a lot more dependencies than you thought, you're not alone. Because of this, one of the benefits of cloud computing (easily created fresh-slate virtual machines) can actually be a bit of a pain! Being engineers, we've invented lots of ways of dealing with this over the years (for example, using a shell script that runs when a machine boots) but configuration remains a frustrating problem. Containers happen to solve this problem by making it easy to set up a clean system, describe how you want it to look, and then keep a snapshot of that once it looks exactly right. Later you can boot a container and have it look exactly as you had described.

You may be thinking about the Persistent Disks Snapshots we learned about in [chapter 9](#) and wondering why we shouldn't just use those to manage our configuration. While that's totally reasonable, it suffers from one big problem: those snapshots only work on Google! This problem brings us to the next issue: standardization.

### 10.1.2 Standardization

A long time ago (pre-1900s), if you wanted to send a table and some chairs across the ocean from England to the US, you had to take everything to a ship and figure out how to fit it inside, sort of like playing a real-life game of Tetris. In other words, it was like packing your stuff into a moving van, just bigger — and shared with everyone else who was putting their stuff in there too.

**Figure 10.1. Shipping before containers**



Eventually the shipping industry decided that this way of packing things was silly and started exploring the idea of containerization. That is, instead of packing things like puzzle pieces, have people solve the puzzle themselves using big metal boxes called

"containers" before they even get to a boat. This way the boat only ever deals with these standard-sized containers, and never has to play Tetris ever again. In addition to reducing the time it took to load boats, standardizing on a specific type of box with specific dimensions meant that the shipping industry could build boats that are good at holding containers, tools that are good at loading and unloading containers, and charge prices based on the number of containers. All of this made shipping things easier, more efficient, and cheaper.

**Figure 10.2. Shipping after containers**



Software containers do for your code exactly what big metal boxes did for shipping. They act as a standard format representing your software and its environment, and offer tools to run and manage those environments that run on every platform. This means that if a system understands containers you can be sure that when you deploy your code there, it will "just work". More concretely, you can focus specifically on getting your code into a container, effectively playing Tetris up front instead of when you're trying to deploy to production. There's one last piece here that needs mentioning, which comes as a by-product of using containers: isolation.

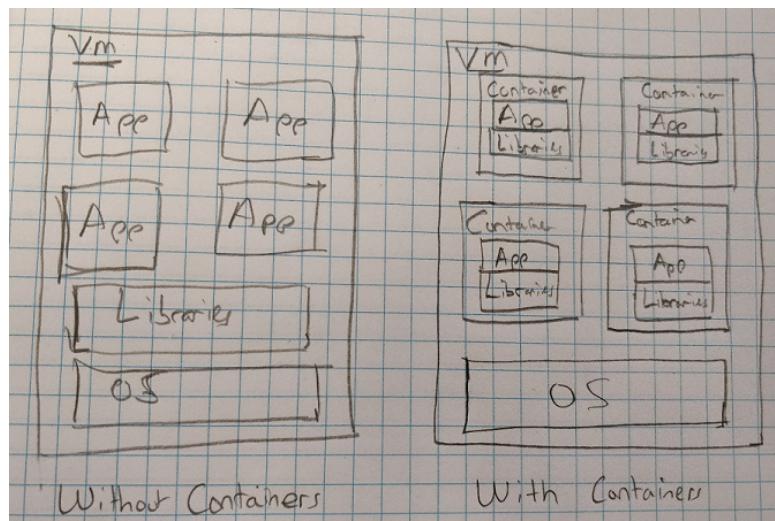
### 10.1.3 Isolation

One thing you might have noticed in the two pictures above is that shipping stuff before containers looks a bit risky, where things might get crushed by other, heavier things. Luckily, inside a container you only worry about your own stuff, and this same concept applies to containers for your code. For example, you might want to take large machine and chop it into two pieces: one for a web server and another for a database. Without a container, if the database were to get tons of SQL queries, the web server would have far fewer CPU cycles to handle web requests. Using two separate containers, however, means that this problem goes away. Just as physical containers have walls to prevent the piano from crushing your stuff, software containers run in a virtual environment with similar walls where you decide exactly how to allocate the underlying resources.

Further, while applications running on the same virtual machine may share the same libraries and operating system, this might not always be the case. When some

applications running on the same system requires different versions of shared libraries reconciling these problems can become quite complicated. By "containerizing" the application, shared libraries aren't really shared anymore, meaning your dependencies are isolated to a single application.

**Figure 10.3. With containers vs without containers**



Now that you understand these benefits (such as configuration, standardization, and isolation), let's jump up a layer in the stack and think about the ship that will hold all of these containers, and the captain who will be steering the ship.

## 10.2 What is Docker?

There are many different systems that are capable of running virtualized environments, but one has taken the lead over the past few years: Docker. Docker is a tool for actually running containers and acts a bit like the ship we described above that carries all of the containers from one place to another. At a fundamental level, Docker handles the lower-level virtualization, which means that it takes the definitions of container images and executes the environment and code defined by the container.

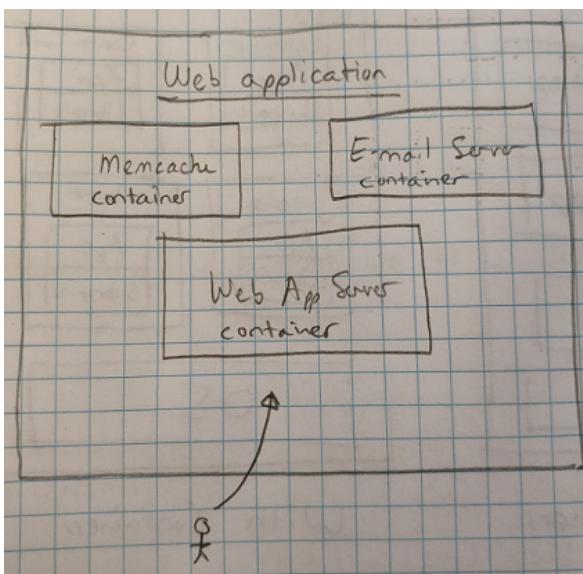
In addition to being the most common base system for running containers, Docker has also become the standard for how we define container images, using a format called a Dockerfile. Dockerfiles let you define a container using a bunch of commands which do anything from running a simple shell command (e.g. `RUN echo "Hello World!"`) all the way to more complex things like exposing a single port outside the container (`EXPOSE 8080`) or inheriting from another predefined container (`FROM node:8`). We'll refer back to the Dockerfile format throughout this chapter, and while you should understand what they're trying to accomplish, don't worry if you don't feel comfortable writing one from scratch. If you get deeper into containers, there are entire books on Docker which will help you learn how to write a Dockerfile of your own.

**NOTE** If you want to follow along with the code and deployment in this chapter, you should install the Docker runtime on your local machine, which is available at [docker.com/community-edition](https://docker.com/community-edition) for most platforms.

## 10.3 What is Kubernetes?

It turns out that if you start using containers, it becomes pretty natural to split things up based on what they're responsible for. For example, if you were creating a traditional web application, you might have a container that handles web requests (e.g. a web app server that handles browser-based requests), another container that handles caching frequently accessed data (e.g. running a service like Memcache), and another container that handles more complex work like generating fancy reports, shrinking pictures down to thumbnail sizes, or sending e-mails to your users.

**Figure 10.4. Overview of a web-application as containers**



Managing where all of these containers run and how they talk to one another turns out to be pretty tricky. For example, you might want all of the web-app servers to have Memcache running on the same physical (or virtual) machine, so that you can talk to Memcache over `localhost` rather than a public IP. As a result, it should be no surprise that there are a bunch of systems that try to fix this problem, one of which is Kubernetes.

Kubernetes is a system that manages your containers and allows you to break things into chunks that make sense for your application regardless of the underlying hardware that runs the code. It also allows you to express more complex relationships, like the fact that you want any VMs that handle web requests to have Memcache on the same machine. Further, since it's open-source, using it doesn't tie you to a single hosting

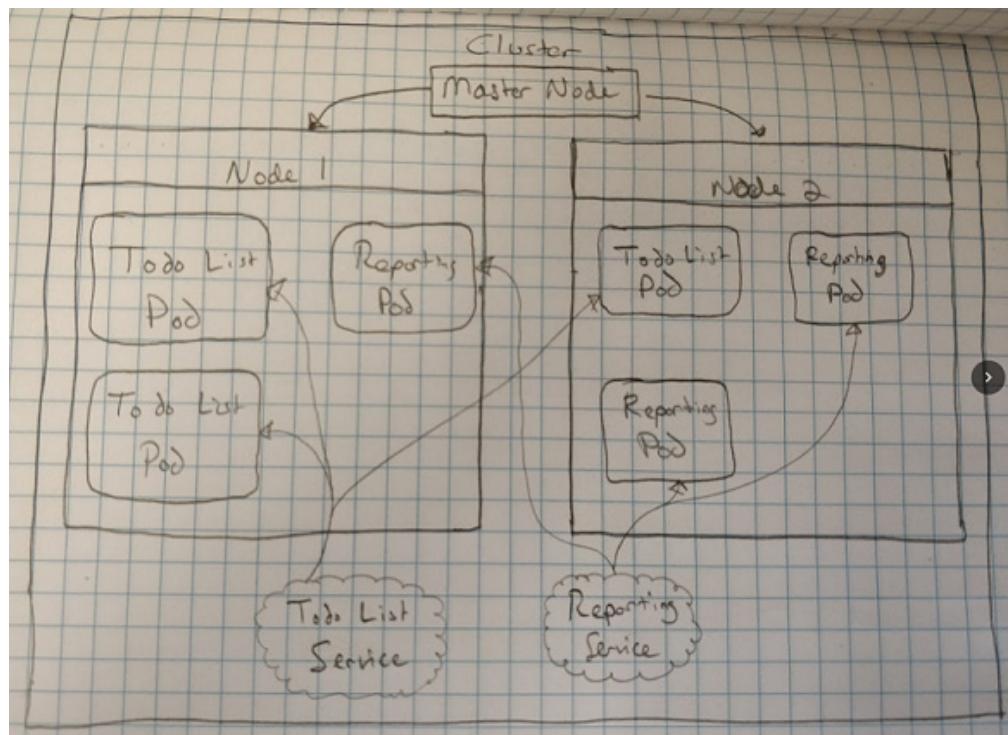
provider. This means that you can run it on any cloud provider or skip out on the cloud entirely by using your own hardware. To do this all of this, Kubernetes builds on the concept of a container as a fundamental unit and introduces several new concepts which you can use to represent your application and the underlying infrastructure, which we'll explore in the next section.

**NOTE**

Kubernetes itself is an enormous platform that has been evolving for several years and becoming more and more complex as time goes on, meaning it's simply too large to fit everything into a single chapter. As a result, we're going to focus on demonstrating how you can use Kubernetes, so if you're interested in Kubernetes, you might want to check out *Kubernetes in Action* as well.

Since there's so much to cover about Kubernetes, let's start by looking at a big scary diagram showing most of the core concepts in Kubernetes. and then zoom in on the four key concepts: clusters, nodes, pods, and services.

**Figure 10.5. Kubernetes cluster overview**



### 10.3.1 Clusters

At the top of the diagram, you'll see the concept of a *cluster*, which is the thing that everything else that we're going to talk about lives inside of. Clusters tend to line up with a single application, which means when we're talking about the deployment for all of the pieces of an application, we'd say that they all run as part of "its Kubernetes

cluster". For example, we'd refer to the production deployment of our to-do list application as our "to-do list Kubernetes cluster".

### **10.3.2 Nodes**

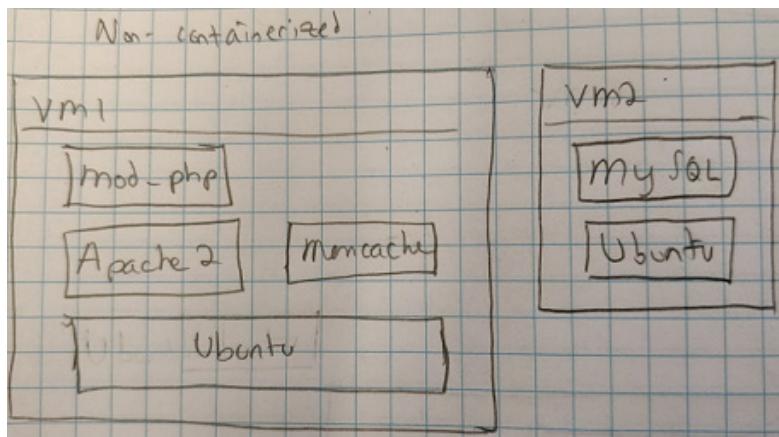
Nodes live "inside" a cluster, and correspond to a single machine (e.g. a VM in GCE) capable of running your code. In this example cluster, there are two different nodes (called Node 1 and Node 2), that are running some aspects of the to-do list application. Each cluster will usually contain several nodes, which are collectively responsible for handling the overall work needed to run your application.

It's important to stress the "collective" aspect of nodes since a single node is not necessarily tied to a single purpose. In other words, it's totally possible that a given node will be responsible for many different tasks at once, and that those tasks might change over time. For example, in our diagram, we have both Node 1 and Node 2 responsible for a mix of responsibilities but this might not be the case later on as work shuffles around across the available nodes.

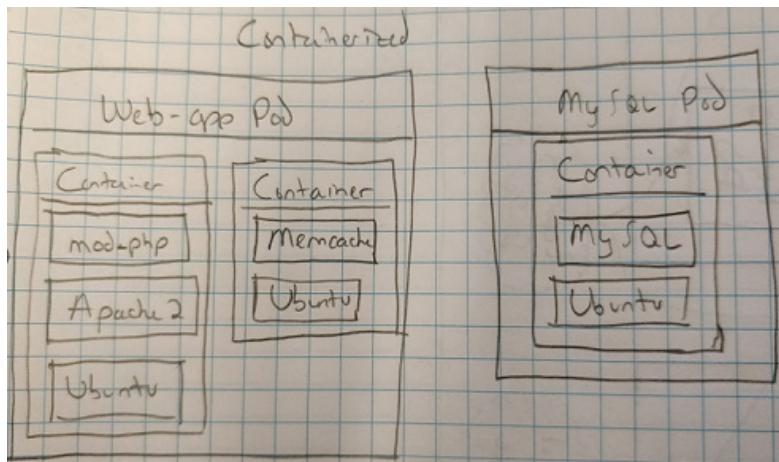
### **10.3.3 Pods**

*Pods* are groups of containers that act as single discrete units of functionality to be run by any given node. This means that the containers that make up a pod will all be kept together on a single node, and will also share the same IP address and port space. This means, for example, that containers on the same pod can communicate via `localhost`, and they can't both bind to the same port (e.g. if Apache is running on port 80, Memcache can't also bind to that same port). The concept of a pod can be a bit confusing, so to clarify, let's look at a more concrete example and compare the traditional version with the Kubernetes-style version.

A LAMP-stack is a common deployment style that consists of running Linux (as the operating system), Apache (to serve web requests), MySQL (to store data), and PHP (to do the actual work of your application). If you were running such a system in a traditional environment, you might have a server running MySQL to store data, another running Apache with `mod_php` (to process PHP code), and maybe one more running Memcache to cache values (either on the same machine as the Apache server or a separate one).

**Figure 10.6. Non-containerized version of a LAMP stack**

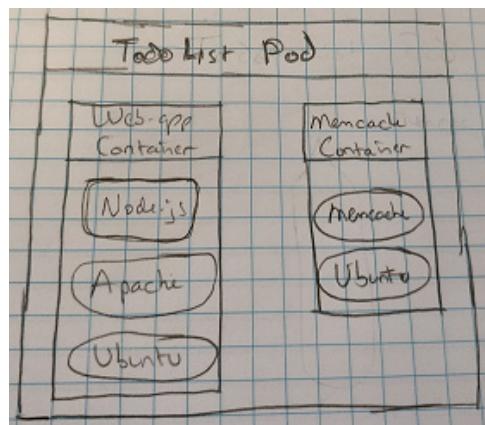
If you were to think of this in terms of containers and pods, you might rearrange this a bit, but the important thing to note is that you think in terms of containers and pods, leaving VMs (and nodes) out of the picture entirely. You might have one pod responsible for serving the web-app (which would be running Apache and Memcache, each in their own container), and another pod responsible for storing data (with a container running MySQL).

**Figure 10.7. Containerized version of a LAMP stack**

These pods might be running on a single VM (or node) or be split across two different VMs (or nodes), but it turns out that you don't really need to care about where a pod is running. So long as each pod has enough computing resources and memory it should be irrelevant. The idea of using pods is that you can focus on what should be grouped together rather than how it should be laid out on specific hardware (whether that's virtual or physical).

Looking at this from the perspective of our to-do list application, we had two different pods: the to-do list web-app pod and the report-generation pod. An example of the to-do list pod is shown below, which is similar to the LAMP-stack described above, with two containers: one for web requests and another for caching data.

**Figure 10.8. To-do list pod**



While this ability to arrange different functionality across lots of different physical machines is really neat, you may be worried about things "getting lost". For example, how do you know where to send web requests for your to-do list application if it might live on a bunch of different nodes?

#### 10.3.4 Services

A *service* is the abstract concept that we use to keep track of where the various pods happen to be running. For example, since the to-do list web-app service could be running on either (or both) of the two nodes, we want a way of finding out where to go if we want to make a request to the web-app. This makes a service a bit like an entry in a phone book, providing a layer of abstraction between a name of someone and the specific place where you can contact them. Further, since things can jump around from one node to another, this phone book needs to be updated quite often. By relying on a service for the various pieces of your application (e.g., in our to-do list, we have the pod that handles web requests), we never worry about where the pod happens to be running as the service can always help route you to the right place.

At this point you should have some idea about some of the core concepts of Kubernetes, but only in an abstract sense. That is, you should understand that a service is a way to help route you to the right pod, and that a pod is a group of containers with a particular purpose, but we've said nothing at all about how to actually create a cluster or a pod or a service. That's OK! We're going to do some of this later on, but we've finally reached the point where we can explain what exactly Kubernetes Engine is! All our talk about containers and Kubernetes and pods has finally paid off!

## 10.4 What is Kubernetes Engine?

Kubernetes is an open-source system, which means that if you want to create clusters and pods and have requests routed to the right nodes, you have to actually install, run, and manage the Kubernetes system yourself. To minimize this burden, we can use Kubernetes Engine which is a hosted and managed deployment of Kubernetes that runs on Google Cloud Platform (using Compute Engine instances under the hood).

You still use all of the same tools that you would if you were running Kubernetes yourself, however the administrative operations (such as creating a cluster and the nodes inside of it) can be done using the Kubernetes Engine API system for you to run pods on. To see how all of this works, let's define a simple Kubernetes application, and then see how we can deploy it to Kubernetes Engine.

## 10.5 Interacting with Kubernetes Engine

### 10.5.1 Defining your application

Let's start by defining a simple Hello World Node.js application using Express.js. You should be familiar with Express, but if you're not it's nothing more than a Node.js web framework. A simple application might look something like the following, saved as `index.js`.

#### **Listing 10.1. Simple Hello World Express application**

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello world!');
});

app.listen(8080, '0.0.0.0', () => {
  console.log('Hello world app is listening on port 8080.');
});
```

This web application will listen for requests on port 8080, and always reply with the text "Hello world!". We also need to make sure we have our Node.js dependencies configured properly, which we can do with a `package.json` file.

#### **Listing 10.2. package.json for our application**

```
{
  "name": "hellonode",
  "main": "index.js",
  "dependencies": {
    "express": "~4"
  }
}
```

So how would we go about "containerizing" this application? To do this we create a

Dockerfile which will look sort of like a start-up script for a VM, but a bit strange. Don't worry though, you're not supposed to be able to write this from scratch. Instead, let's look at each piece and see what it does.

#### **Listing 10.3. An example docker file**

```
FROM node:8          1
WORKDIR /usr/src/app 2
COPY package.json .   3
RUN npm install       4
COPY . .             5
EXPOSE 8080          6
CMD ["node", "index.js"] 7
```

- ① This is the base image (node:8) which is provided by Node.js itself, and gives us a base operating system that comes with Node v8 pre-installed and ready to go.
- ② This is the equivalent of cd to move into a current working directory, but it also makes sure the directory exists before moving into it.
- ③ The COPY command does exactly as you'd expect, moving something from the current directory on your machine, to the specified directory on the Docker image.
- ④ The RUN command tells docker to execute a given command on the Docker image. In this case, it installs all of our dependencies (e.g. express) so that they'll be present when we want to run our application.
- ⑤ We use COPY again to move the rest of the files over to the image.
- ⑥ EXPOSE is the same as opening up a port for the rest of the world to have access. In this case, our application will use port 8080, so we want to be sure that it's available.
- ⑦ The CMD statement is the default command that will run. In this case, we want to start a Node.js process running our service (which is in index.js).

Now that you've written a Dockerfile, it might make sense to test it out locally before trying to deploy it to the cloud, so let's take a look at how to do that.

#### **10.5.2 Running your container locally**

Before you can actually run a container on your own machine, you'll first need to install the Docker runtime. Docker Community Edition is free and you can install it for almost every platform out there. For example, there is a .deb package file for Ubuntu available on [docker.com/community-edition](https://docker.com/community-edition).

As we learned before, Docker is nothing more than a tool that understands how to run containers that are defined using the Dockerfile format above. This means that once you have Docker running on your machine, you can tell it to run your Dockerfile and you should see your little web-app running. To test whether you have Docker set up correctly, run `docker run hello-world`, which tells Docker to go find a container image called "hello-world". Docker knows how to go find publicly available images, so it will download the `hello-world` image automatically and then run it. The output of running this should look something like the following.

**Listing 10.4. Running the hello-world Docker image**

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally ①
latest: Pulling from library/hello-world ②
b04784fba78d: Pull complete
Digest: sha256:f3b3b28a45160805bb16542c9531888519430e9e6d6ffc09d72261b0d26ff74f
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
# ... More information here ...
```

- ① Docker realized that this image isn't available locally.
- ② As a result, Docker goes looking for the image from Dockerhub (a place that hosts images).

To run our image, we have to take our Dockerfile which describes a container and build it into an actual container image. This is a bit like compiling source code into a runnable binary when you're writing code in a compile language like C++ or Java. Make sure the contents of your Dockerfile is in a file called `Dockerfile`, and then we'll use `docker build` to create our image and tag it as "hello-node".

**Listing 10.5. Building a container image using Docker**

```
$ docker build --tag hello-node .
Sending build context to Docker daemon 1.345MB
Step 1/7 : FROM node:8
Step 2/7 : WORKDIR /usr/src/app
Step 3/7 : COPY package.json .
Step 4/7 : RUN npm install
Step 5/7 : COPY . .
Step 6/7 : EXPOSE 8080
Step 7/7 : CMD node index.js
Successfully built 358ca555bbf4
Successfully tagged hello-node:latest
```

You'll see a whole lot happening under the hood, and it will line up one-to-one with the commands we defined in the Dockerfile. First it will go looking for the publicly available base container that has Node v8 installed, then set up our work directory, all the way through to running the `index.js` file that defines our web application. Note that this is simply **building** the container, not running it, which means that the container itself is in a state that is "ready to run", but isn't actually running at the moment. If we want to test out that things worked as expected, we can use the `docker run` command with some special flags.

**Listing 10.6. Running a Docker image locally**

```
$ docker run -d -p 8080:8080 hello-node
485c84d0f25f882107257896c2d97172e1d8e0e3cb32cf38a36aee6b5b86a469 ①
②
```

- ➊ The `-d` flag tells Docker to run this container image in the background, and the `-p 8080:8080` tells Docker to take anything on your machine that tries to talk to port 8080 and forward it onto your container's port 8080.
- ➋ The result of running your container image is a unique ID that you can use to address that particular image (after all, you might have lots of the same image running at the same time).

To check that your image is actually running, you can use the `docker ps` command, and you should see the `hello-node` image in the list.

#### **Listing 10.7. Viewing running Docker containers locally**

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
485c84d0f25f	hello-node	"node index.js"	About a minute ago	Up About a minute	0.0.0.0:8080->8080/tcp	dreamy_spence

As you can see in this snippet, the container has only been running for about a minute, and the container's port 8080 is bound to the machine's port 8080. Also note that the container ID has been shortened to just the first few letters of the unique ID. You can shorten this even further so long as the ID doesn't match more than one container, so for this exercise, we'll refer to this container as 485c. Let's check the output of our container so far, after all we told Node to print to the console when it started listening for requests.

#### **Listing 10.8. Printing out the logs of a running Docker container**

```
$ docker logs 485c
Hello world app is listening on port 8080.
```

As you can see, the output here is exactly what we'd expect. So let's try connecting to the container's Node.js server using `curl`.

#### **Listing 10.9. Testing that the server is running in the container**

```
$ curl localhost:8080
Hello world!
```

Like magic we have a Node.js process running and serving HTTP requests from inside a container being run by the Docker service on our machine. If we wanted to stop this container, we could use the `docker stop` command.

#### **Listing 10.10. Stopping the container using Docker**

```
$ docker stop 485c
485c
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
485c84d0f25f	hello-node	"node index.js"	1 minute ago	Up 1 minute	0.0.0.0:8080->8080/tcp	dreamy_spence

Here once we stop the docker container, it no longer appears in the list of running containers shown using `docker ps`. So now that we have an idea of what it feels like to

run our simple application as a container using Docker, let's look at how we could switch from using our local Docker instance to a full-fledged Kubernetes cluster (which itself uses Docker under the hood), starting with how we package up our containerized application and deploy it to your private container registry.

### 10.5.3 Deploying to your container registry

At this point, we've build and run a container locally, but if we want to deploy it we'll need it to exist on Google Cloud. In other words, just like we need to upload our source code somewhere in order to run it on Compute Engine, we need to upload our container itself in order to run it on Kubernetes Engine. To do this, Google offers a private per-project container registry which basically acts as storage for all of your various containers.

To get started, we first need to tag our image in a special format. In the case of Google's container registry, the tag format is `gcr.io/your-project-id/your-app` (which can come with different versions on the end, like `:v1` or `:v2`). In this case, this means that we need to tag our container image as `gcr.io/your-project-id/hello-node:v1`. To do this, we'll use the `docker tag` command. As you recall, the image we created was called `hello-node`, and you can always double check the list of images using the `docker images` command, shown below.

#### **Listing 10.11. Listing of available Docker images**

\$ docker images				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-node	latest	96001025c6a9	12 days ago	671.8 MB

#### **Listing 10.12. Re-tagging our hello-node Docker image**

```
$ docker tag hello-node gcr.io/project-id/hello-node:v1
```

Once the image is re-tagged, you should see an extra image show up in the list of available Docker images. Also notice that the `:v1` part of our naming shows up under the special "TAG" heading below, making it easy to see when you have the same container of multiple versions.

#### **Listing 10.13. Listing of available Docker images after our re-tag**

\$ docker images				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
gcr.io/project-id/hello-node	v1	96001025c6a9	12 days ago	671.8 MB
hello-node	latest	96001025c6a9	12 days ago	671.8 MB

**NOTE**

You could always build your container with this name from the start, but we'd already built the container beforehand.

Now all that's left is to upload the container image to our container registry, which we can do with the `gcloud` command-line tool.

**Listing 10.14. Upload our container to the container registry**

```
$ gcloud docker -- push gcr.io/project-id/hello-node:v1
The push refers to a repository [gcr.io/project-id/hello-node]
b3c1e166568b: Pushed
7da58ae04482: Pushed
2398c5e9fe90: Pushed
e677efb47ea8: Pushed
aaccc8d23649: Pushed
348e32b251ef: Pushed
e6695624484e: Pushed
da59b99bbd3b: Pushed
5616a6292c16: Pushed
f3ed6cb59ab0: Pushed
654f45ecb7e3: Pushed
2c40c66f7667: Pushed
v1: digest: sha256:65237913e562b938051b007b8cbc20799987d9d6c7af56461884217ea047665a
size: 2840
```

You can verify that this worked by going into the Cloud Console, and choosing "Container Registry" from the left-side navigation. Once there, you should see your "hello-node" container in the listing, and clicking on it should show the beginning of the hash and the "v1" tag that we applied.

**Figure 10.9. Container Registry listing of our hello-node container**

The screenshot shows the Google Cloud Container Registry interface. On the left, there's a sidebar with icons for Container Registry, Build triggers, and Build history. The main area has a header with back, refresh, show pull command, and delete buttons. Below the header, it says 'gcr.io / project-id / hello-node'. There's a search bar labeled 'Filter by name or tag'. A table lists the container details:

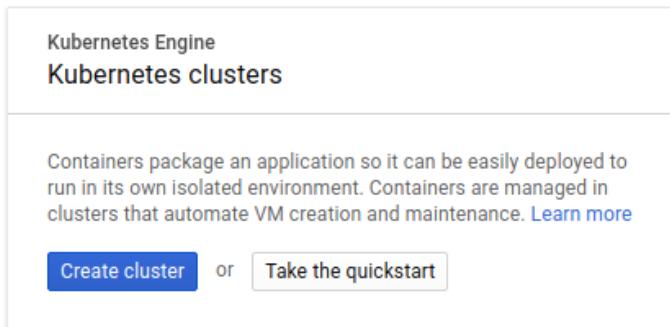
Name	Tags	Virtual size	Uploaded
65237913e562	v1	250.2 MB	5 minutes ago

Now that our container is uploaded to Google Cloud, we can go about getting our Kubernetes Engine cluster ready.

#### 10.5.4 Setting up your Kubernetes Engine cluster

Just like we needed to install Docker on a local machine in order to run a container, we'll need to set up a Kubernetes cluster if we want to deploy our containers to Kubernetes Engine. Luckily, this is a lot easier than it might sound, and you can do it from the Cloud Console just like turning on a Compute Engine VM. To start, choose Kubernetes Engine from the left-side navigation of the Cloud Console. Once there, you'll see a prompt to create a new Kubernetes cluster, and when you click on this you'll see a page that should look very familiar to that of creating a new Compute Engine VM.

**Figure 10.10. Prompt to create a new Kubernetes Engine cluster**



Since we're just trying to kick the tires of Kubernetes Engine, let's leave everything set to the defaults. That is, we'll use the us-central1-a zone, a single vCPU per machine, and a size of 3 VMs for the whole cluster (remember, we can always change these things later). The only thing we should do is pick a name for our cluster (we'll use `first-cluster`) in this example. Once you've verified that the form shows what you expect, click the Create button and then wait a few seconds while the VMs are created and Kubernetes is configured on your cluster of machines.

Once the cluster is created and marked as running, you can verify that it's working properly by listing your VMs. Remember that a Kubernetes Engine cluster relies on Compute Engine VMs under the hood, so we can look at them just like any other VM running.

#### **Listing 10.15. Listing of GKE-related instances in us-central1-a**

NAME	ZONE	MACHINE_TYPE	PREEMPTIBLE	INTERNAL_IP	EXTERNAL_IP	STATUS
gke-first-cluster-default-pool-e1076aa6-c773	us-central1-a	n1-standard-1		10.128.0.4	130.211.169.131	RUNNING
gke-first-cluster-default-pool-e1076aa6-mdcd	us-central1-a	n1-standard-1		10.128.0.5	35.193.144.47	RUNNING
gke-first-cluster-default-pool-e1076aa6-xhxp	us-central1-a	n1-standard-1		10.128.0.3	35.192.153.28	RUNNING

Now that we have a cluster running, and we can see that there are 3 VMs running that make up the cluster, let's dig into how to actually interact with the cluster.

#### **10.5.5 Deploying your application**

We have our container deployed, and our cluster created, so the next thing we need to do is find a way to communicate and deploy things to this cluster. After all, we have a bunch of machines running doing basically nothing! Since this cluster is made up of machines running Kubernetes under the hood, this means we can use the existing tools for talking to Kubernetes to talk to our Kubernetes Engine cluster. In this case, the tool we'll use to talk to our cluster is called `kubectl`.

**NOTE**

Keep in mind that some of the operations we'll run using `kubectl` will always return quickly, but they're likely doing some background work under the hood. As a result, you may have to wait a little bit before moving onto the next step.

To make this process easy in case you're not super familiar with Kubernetes (which we're not), Google Cloud offers a fast installation of `kubectl` using the `gcloud` command-line tool. This means that installing `kubectl` is nothing more than running a simple `gcloud` command, shown below.

**Listing 10.16. Install kubectl using gcloud CLI**

```
$ gcloud components install kubectl
```

**NOTE**

If you've installed `gcloud` using a package manager (like `apt-get` for Ubuntu), you might see a recommendation from `gcloud` saying to use the same package manager to install `kubectl` (e.g., `apt-get install kubectl`).

Once you have `kubectl` installed, we need to be sure that it's properly authenticated to talk to our cluster. We can do this using another `gcloud` command that fetches the right credentials and ensures that `kubectl` has them available.

**Listing 10.17. Fetching credentials for use with kubectl**

```
$ gcloud container clusters get-credentials --zone us-central1-a first-cluster
Fetching cluster endpoint and auth data.
kubeconfig entry generated for first-cluster.
```

Once `kubectl` is set up, this means that we can use it to deploy a new application using our container image under the hood. We can do this by running `kubectl run`, and then verify that it was deployed to a pod using `kubectl get pods`.

**Listing 10.18. Deploying the application to our cluster using kubectl**

```
$ kubectl run hello-node --image=gcr.io/your-project-id-here/hello-node:v1 --port
8080
deployment "hello-node" created
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
hello-node-1884625109-sjq76  1/1     Running   0          55s
```

We're now almost done, with one final step before we can check whether things are working as expected. Remember that `EXPOSE 8080` command in our Dockerfile? We have to do something similar with our cluster to make sure that the ports we need to handle requests are properly exposed. To do this, we can use the `kubectl expose` command, and under the hood, Kubernetes Engine will configure a load balancer just like we learned about in [chapter 9](#).

**Listing 10.19. Expose the port using a load balancer**

```
$ kubectl expose deployment hello-node --type=LoadBalancer --port 8080
service "hello-node" exposed
```

Once this is done, you should see a load balancer appear in the Cloud Console that points to your 3 VM instances that make up the cluster.

**Figure 10.11. Automatically created load balancer in the Cloud Console**

The screenshot shows the Google Cloud Platform Load Balancer interface. At the top, there's a green checkmark icon and the identifier 'ac262639293db11e781a142010a80014'. To the right are a trash can icon and an 'Edit' button. Below this, the 'Frontend' section is visible, showing a single TCP rule with IP:Port '104.154.231.30:8080'. The 'Backend' section shows a load balancer named 'ac262639293db11e781a142010a80014' in the 'us-central1' region with 'None' session affinity. It states: 'This load balancer has no health check, so traffic will be sent to all instances regardless of their status'. The 'Instances' table lists three VM instances: 'gke-first-cluster-default-pool-e1076aa6-xhxp', 'gke-first-cluster-default-pool-e1076aa6-mdcd', and 'gke-first-cluster-default-pool-e1076aa6-c773', all associated with the IP 104.154.231.30. Each instance has a red exclamation mark icon next to it.

At this point, you may be remembering that pods are the way we keep containers together to serve a common purpose, and not something that you'd talk to individually. And you're right! If we want to actually talk to our application, we have to use the proper abstraction for this, which you should recall is known as a "service".

We can look at the available services (in this case, our application) using `kubectl get services`.

**Listing 10.20. Listing of available services**

```
$ kubectl get service
NAME      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
hello-node  10.23.245.188  104.154.231.30  8080:31201/TCP  1m
kubernetes  10.23.240.1    <none>          443/TCP      10m
```

Notice at this point that we have a generalized Kubernetes service (which handles administration), as well as the service for our application. Additionally, our application has an external IP address that we can use to see if everything worked by making a simple request to the service.

**Listing 10.21. Verifying our application is running**

```
$ curl 104.154.231.30:8080
Hello world!
```

And sure enough, everything worked exactly as expected! We now have a containerized application running using one pod and one service inside Kubernetes, managed by Kubernetes Engine! This alone is pretty cool, but the real magic happens when we need to handle more traffic, which we can do by replicating the application.

### **10.5.6 Replicating your application**

Recall that using Compute Engine, you could easily turn on new VMs by changing the size of the cluster, but in order to get your application running on those machines you needed to either set them to run automatically when the VM turned on, or manually connect to the machine and start the application. So what about doing this with Kubernetes? At this point in our deployment, we have a 3-node Kubernetes cluster, with two services (one for Kubernetes itself, and one for our application), and our application is running in a single pod. Let's look at how we might change that, but first let's benchmark how well our cluster can handle requests in the current configuration.

You can use any benchmarking tool you want, but for this illustration let's try using Apache Bench (`ab`). If you don't have this tool installed, you can install it on Ubuntu by running `sudo apt-get install apache2-utils`. To test this, we'll send 50,000 requests, 1,000 at a time, to our application, and see how well the cluster does with handling these requests.

**Listing 10.22. Running Apache Bench against our application**

```
$ ab -c 1000 -n 50000 -qSd http://104.154.231.30:8080/
This is ApacheBench, Version 2.3 <$Revision: 1604373 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 104.154.231.30 (be patient).....done

# ...

Concurrency Level:      1000
Requests per second:   2980.93 [#/sec] (mean) ①
Time per request:      335.465 [ms] (mean) ②
# ...
```

- ① Here we can see that we handled about 3,000 requests per second.
- ② Additionally, we can tell that most requests were completed in 300 milliseconds.

What if we could scale our application up to take advantage of more of our cluster? It turns out that this can be done in one single command: `kubectl scale`. Let's scale our application to run on 10 pods at the same time.

**Listing 10.23. Scale up to three replicas of our application**

```
$ kubectl scale deployment hello-node --replicas=10
deployment "hello-node" scaled
```

Immediately after we run this command, looking at the pods available will show that we're going from one available up to ten different pods.

**Listing 10.24. Listing available pods**

```
$ kubectl get pods
NAME          READY   STATUS        RESTARTS   AGE
hello-node-1884625109-8ltzb  1/1    ContainerCreating  0          3m
hello-node-1884625109-czn7q  1/1    ContainerCreating  0          3m
hello-node-1884625109-dzs1d  1/1    ContainerCreating  0          3m
hello-node-1884625109-gw6rz  1/1    ContainerCreating  0          3m
hello-node-1884625109-kvh9v  1/1    ContainerCreating  0          3m
hello-node-1884625109-ng2bh  1/1    ContainerCreating  0          3m
hello-node-1884625109-q4wm2  1/1    ContainerCreating  0          3m
hello-node-1884625109-r5msp  1/1    ContainerCreating  0          3m
hello-node-1884625109-sjq76  1/1    Running         0          1h
hello-node-1884625109-tc2lr  1/1    ContainerCreating  0          3m
```

And after a few minutes, these pods should come up and be available as well.

**Listing 10.25. Listing pods after they are running**

```
$ kubectl get pods
NAME          READY   STATUS      RESTARTS   AGE
hello-node-1884625109-8ltzb  1/1    Running     0          3m
hello-node-1884625109-czn7q  1/1    Running     0          3m
hello-node-1884625109-dzs1d  1/1    Running     0          3m
hello-node-1884625109-gw6rz  1/1    Running     0          3m
hello-node-1884625109-kvh9v  1/1    Running     0          3m
hello-node-1884625109-ng2bh  1/1    Running     0          3m
hello-node-1884625109-q4wm2  1/1    Running     0          3m
hello-node-1884625109-r5msp  1/1    Running     0          3m
hello-node-1884625109-sjq76  1/1    Running     0          1h
hello-node-1884625109-tc2lr  1/1    Running     0          3m
```

At this point, we have ten pods running across our three nodes, so let's try our benchmark a second time and see if the performance is any better.

**Listing 10.26. Benchmarking the service using Apache Bench (ab)**

```
$ ab -c 1000 -n 50000 -qSd http://104.154.231.30:8080/
This is ApacheBench, Version 2.3 <$Revision: 1604373 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 104.154.231.30 (be patient).....done

# ...
```

```
Concurrency Level:      1000
Requests per second:   5131.86 [#/sec] (mean)  ①
Time per request:     194.861 [ms] (mean)  ②
# ...
```

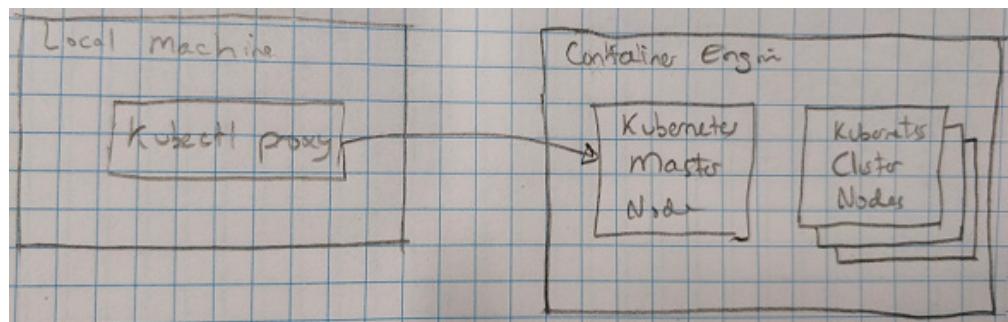
- ① Our newly scaled-up cluster handles about 5,000 requests per second.
- ② Additionally, the time per request has dropped to around 200 milliseconds.

At this point you may be wondering if there's a UI for all of this information. In other words, is there a UI to look at pods in the same way there's a UI to look at GCE instances? It turns out there is, but it's part of Kubernetes itself and not Kubernetes Engine.

### 10.5.7 Using the Kubernetes UI

Kubernetes comes with a built-in UI and because Kubernetes Engine is just a managed Kubernetes cluster, this means that you can view the Kubernetes UI for your Kubernetes Engine cluster the same way you would any other Kubernetes deployment. To do this, we can use the `kubectl` command line tool to effectively open up a tunnel between our local machine and the Kubernetes master, so that you can talk to, say, <localhost:8001> and the request would be securely routed to the Kubernetes master (and not a server on your local machine).

**Figure 10.12. Proxying local requests to the Kubernetes master**

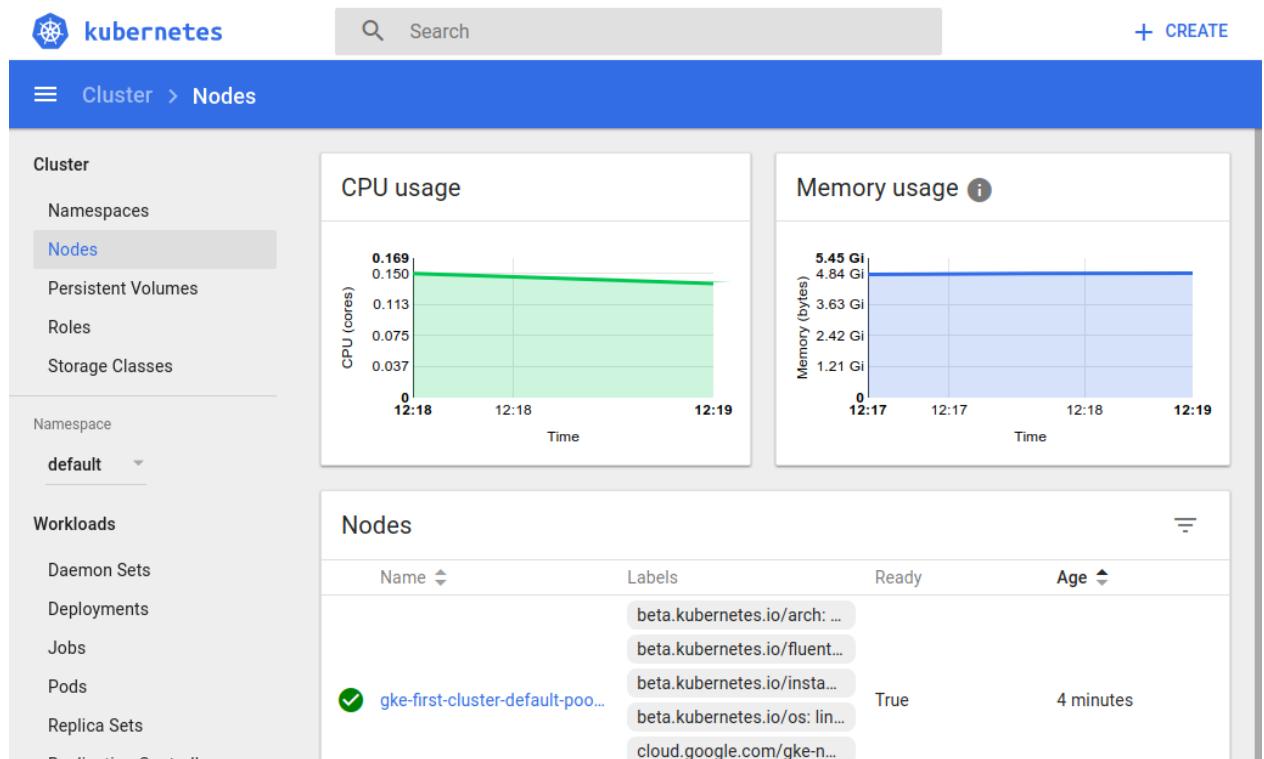


**Listing 10.27. Using kubectl to open a secure connection to the Kubernetes master**

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

Once the proxy is running, connecting to <localhost:8001/ui/> will show the full Kubernetes UI, which provides lots of helpful management features for your cluster.

**Figure 10.13. Kubernetes UI using kubectl proxy**



At this point, you've seen how Kubernetes works at a very simplified level, but the part that's important to remember is that you didn't have to configure or install Kubernetes at all on your cluster because Kubernetes Engine did it all for you. As we mentioned before, Kubernetes is a huge system so this chapter is not about teaching you everything there is to know about Kubernetes. For example, you can see how to access the Kubernetes UI but we're not going into any detail at all about what you can do using the UI. Instead the goal of this chapter is to show you how Kubernetes "just works" when you rely on Kubernetes Engine to handle all of the administrative work.

If at this point you're interested in doing more advanced things with Kubernetes, such as deploying a more advanced cluster made up of lots of pods and databases, now is the time to pick up a book about Kubernetes, because Kubernetes Engine is nothing more than a managed Kubernetes deployment under the hood. That said, there are quite a few things that are specific to Kubernetes Engine and not general across Kubernetes itself, so let's look briefly at how you can manage the underlying Kubernetes cluster using Kubernetes Engine and the Google Cloud tool chain.

## 10.6 Maintaining your cluster

Like any software, new versions come out and sometimes it makes sense to upgrade.

For example, if Apache releases new bug fixes or security patches it makes quite a bit of sense to upgrade to the latest version. The same goes for Kubernetes but remember that because we're using Kubernetes Engine, instead of deploying and managing our own Kubernetes cluster, we need a way of managing that Kubernetes cluster via Kubernetes Engine. As you might guess, this is pretty easy. Let's start with upgrading the Kubernetes version.

In your Kubernetes cluster, there are two distinct pieces: the master node which is entirely hidden and managed by Kubernetes Engine (that is, you don't see that node listed in the list of nodes), and your cluster nodes which are transparently managed by Kubernetes Engine, which means these are the ones we see when listing the active nodes in the cluster. If Kubernetes has a new version available, you'll have the ability to upgrade the master node, all the cluster nodes, or both. Although the upgrade process is similar for both types of nodes, there are some important differences and different things to worry about for each, so we'll look at these separately, starting with the Kubernetes master node.

### **10.6.1 Upgrading the Kubernetes master node**

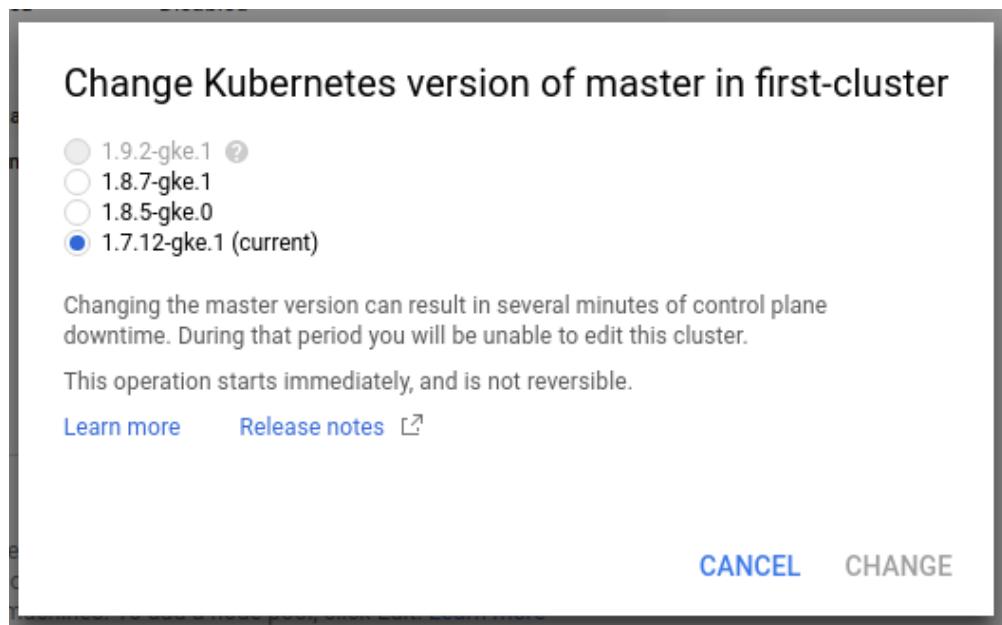
By default, as part of being managed by Google, master nodes are automatically upgraded to the latest supported version after it's released, but if you want to jump to the latest supported version of Kubernetes, you can choose to manually upgrade your cluster's master node ahead of schedule. When there is an update available for Kubernetes, your Kubernetes Engine cluster will show a link next to the version number that you can click on to change the version. For example, the following is the link shown when an upgrade is available for your master node.

**Figure 10.14. Upgrade Kubernetes on Kubernetes Engine**

Master version	1.7.12-gke.1	Upgrade available
Endpoint	104.198.26.195	Show credentials
Client certificate	Enabled	
Kubernetes alpha features	Disabled	
Total size	3	
Master zone	us-central1-a	

When you click on this "Upgrade" link, you'll see a prompt that allows you to choose a new version of Kubernetes. As the prompt notes, there are a few things to keep in mind while changing the version of Kubernetes.

**Figure 10.15. Prompt to upgrade Kubernetes master node**



First, upgrading from an older version to a new version on the Kubernetes Engine cluster's master node is a one-way operation. In other words, if you decide later that you don't like the new version of Kubernetes (maybe there's a bug no one noticed or an assumption that doesn't hold anymore) you can't use this same process to go back to the previous version. Instead, you'd have to create a new cluster with the old Kubernetes version, and re-deploy your containers to that other cluster. As a result, in order to avoid downtime, it's usually a good idea to try out a separate cluster with the new version to see if everything works as you'd expect. After you've tested out the newer version and found that it works as you expected, it should be safe to upgrade your existing cluster.

Next, changing the Kubernetes version requires that the Kubernetes control plane (the service that `kubectl` talks to in order to scale or deploy new pods) be stopped, upgraded, and restarted. This means that while the upgrade operation is running, you'll be unable to edit your cluster at all, and all `kubectl` calls that try to talk to your cluster won't work. This means that if you suddenly receive a spike of traffic in the middle of the upgrade, you won't be able to run `kubectl scale`, which could result in downtime for some of your customers.

Finally, don't forget that this is an optional step. If you wait around for a bit, your Kubernetes master node will automatically upgrade to the latest version without you noticing. That said, this is not the case for your cluster nodes, so let's look at those in more detail.

## 10.6.2 Upgrading cluster nodes

Unlike the master node, cluster nodes aren't hidden away in the shadows and instead are visible to us as regular Compute Engine VMs similar to managed instance groups. Also unlike the master node, the version of Kubernetes that is running on these managed VMs is not automatically upgraded every so often. This means that it is up to us to decide when to make this change. We can change the version of Kubernetes on our cluster's nodes by looking in the Cloud Console next to the "Node version" section of our cluster, and clicking on the "Change" link.

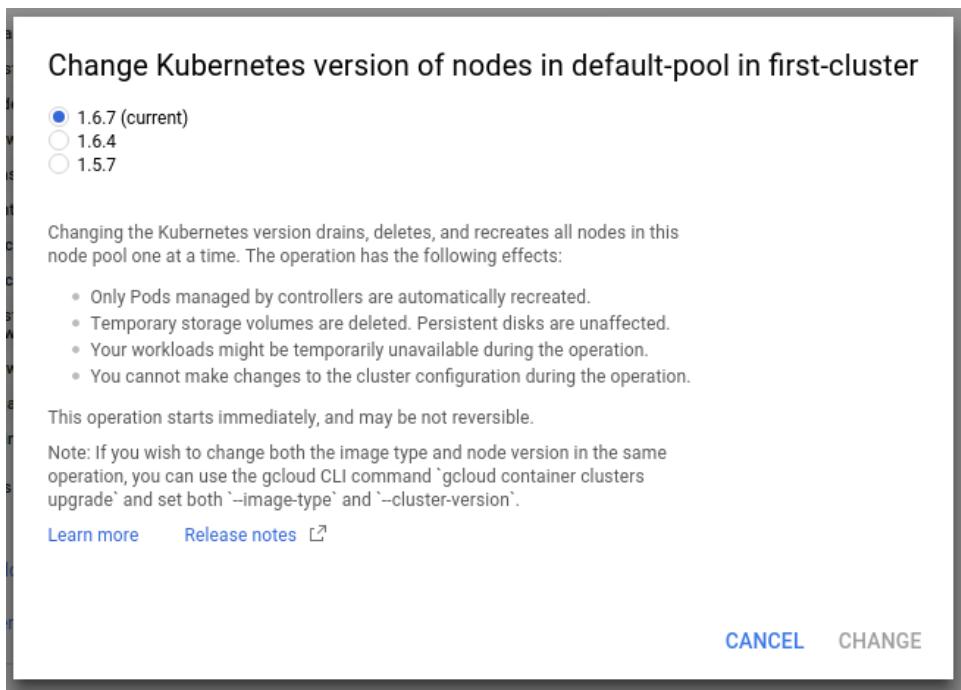
**Figure 10.16. Change the version of cluster nodes**

### Node Pools

Node pools are separate instance groups running Kubernetes in a cluster. You may add node pools in different zones for higher availability, or add node pools of different type machines. To add a node pool, click Edit. [Learn more](#)

Name	default-pool
Size	3
Node version	1.6.7
Node image	<a href="#">Container-Optimized OS (cos)</a>
Machine type	<a href="#">n1-standard-1 (1 vCPU, 3.75 GB memory)</a>

You may be wondering why we're talking about "changing" the node version rather than "upgrading". This is primarily because, unlike the master node version, this operation is sometimes reversible (though not always). That is, you can downgrade to 1.5.7 and then decide to upgrade back to 1.6.4. When you click on the link, you'll see a prompt that allows you to choose the target version, and explains quite a bit about what's happening under the hood.

**Figure 10.17. Prompt to change the version of cluster nodes**

First, since there is always **at least** one cluster node (unlike the master node, which is always a single instance), changing the Kubernetes version on the cluster nodes is done by applying a "rolling update" to your cluster, similarly to updates were applied to a single machine at a time. In order to do this, Kubernetes Engine will first make the node as "unscheduleable" (which means that no new pods will be scheduled on the node) and then "drain" any pods on the node (which means it will terminate them and, if needed, put them on another node). This means that the fewer nodes you have, the more likely it is you'll experience some form of down-time. For example, if you have a single-node cluster, then your service will be unavailable for the duration of the downtime (100% of your nodes will be down at some point). On the other hand, if you have a 10-node cluster, you'll be down by 10% capacity at most (1 of the 10 nodes at a single instance).

Second, notice that the choices available in this prompt are not the same as those in the prompt for upgrading the master node. This is simply because the cluster nodes must be compatible with the master node, which basically means "not too far behind" the master node (and never ahead of the master node). This means that if you have a master node at version 1.6.7, you can use version 1.6.4 on your cluster nodes, but if you are using a later version, this same cluster node version might be "too far behind". As a result, it's a good idea to upgrade your cluster nodes every three months or so.

Next, unlike the master node which is hidden from our view, it's possible that we've come to expect that any data stored on the cluster nodes will be there forever. In truth,

unless you explicitly set up persistent storage for your Kubernetes instance, the data you have stored will be lost when you perform an upgrade. This is because the boot disks for each cluster node are deleted and new ones created for the new nodes. Any other non-boot disks (and non-local disks) will be persisted. You can read more about connecting Google Cloud persistent storage in the Kubernetes documentation (or the Kubernetes book you're reading after this chapter); just look for a section on storage volumes and the `gcePersistentDisk` volume type.

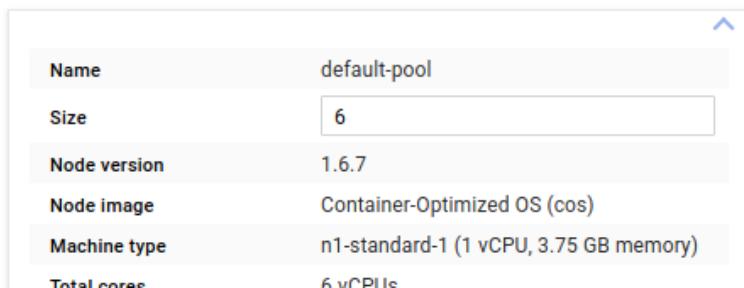
Finally, similarly to upgrading the master node, while the version change is in progress on cluster nodes the cluster itself won't be editable. This means that in addition to the downtime you might experience due to nodes being drained of their pods, the control-plane operations will also be unavailable for the duration of the version change.

### 10.6.3 Resizing your cluster

Just like scaling up the number of pods using `kubectl scale` was really easy, changing the number of nodes in your cluster is just as easy. In the Cloud Console, if you click "Edit" on your cluster, you'll see a field called "size" which we originally set to "3" when we created the cluster.

Changing this number will effectively scale the number of nodes available in your cluster, and you can either set the size to a larger number which will add more nodes to provide more capacity or set it to a smaller number which will shrink the size of your cluster. If you shrink the cluster, similarly to a version change on the cluster nodes, a node will first be marked as "unschedulable", then drained of all pods, and then shut down. As an example, the following shows what it's like to change our cluster from 3 nodes to 6.

**Figure 10.18. Resize the cluster to 6 nodes**



You can also do this using the `gcloud` command-line tool. For example, the following snippet resizes the cluster from 6 nodes back to 3.

**Listing 10.28. Resizing the cluster back to 3 nodes**

```
$ gcloud container clusters resize first-cluster --zone us-central1-a --size=3
Pool [default-pool] for [first-cluster] will be resized to 3.

Do you want to continue (Y/n)? Y
```

```
Resizing first-cluster...done.
Updated [https://container.googleapis.com/v1/projects/your-project-id-
here/zones/us-central1-a/clusters/first-cluster].
```

At this point, you may want to spin down your Kubernetes cluster. You can do this by deleting the cluster either in the Cloud Console or using the command-line tool, but make sure to move any data you might need to a safe place before deleting the cluster. With that, it's time to move on to looking at how pricing works.

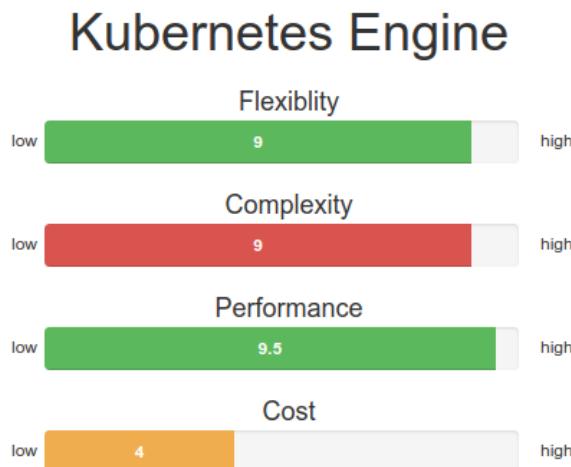
## 10.7 Understanding pricing

Similar to some of the other services on Google Cloud Platform, Kubernetes Engine relies on Compute Engine to provide the infrastructure for the Kubernetes cluster. As a result, the cost of the cluster itself is based primarily on the cluster nodes. This means that you can refer back to [chapter 9](#) for information on how much each node costs. In addition to the cost of the nodes themselves, we can't forget about the Kubernetes master node, which is entirely hidden from us by Kubernetes Engine. Since we don't have control over this node explicitly, there is no charge for this overhead.

## 10.8 When should I use Kubernetes Engine?

At this point you may be wondering specifically how Kubernetes Engine stacks up against other computing environments, primarily Compute Engine. Let's use the standard score card for computing to see how Kubernetes Engine stacks up against the others.

**Figure 10.19. Kubernetes Engine score card**



### 10.8.1 Flexibility

Similar to Compute Engine, Kubernetes Engine is quite flexible, however it's not quite

the same as having a general purpose set of VMs that run whatever you want. For example, you're required to specify your environment using container images (in other words, with Dockerfiles), rather than custom start-up scripts or GCE disk images. While this is technically a limitation that reduces flexibility, it really isn't a bad thing to formalize how you define your application in terms of a container. In other words, even though Kubernetes Engine is ever so slightly more restrictive, this might actually be a good thing.

There are other limitations as well, such as the requirement that you cluster nodes Kubernetes version must be compatible with the version of your master node, or that when you upgrade your nodes using Kubernetes Engine the boot disk data is lost. Again, these things, while technically restrictions, shouldn't be considered deal-breakers at all. This means that for most scenarios Kubernetes Engine really isn't any less flexible than Compute Engine and at the same time provides quite a few benefits with the ability to scale up and down both nodes and pods. As a result, if you look past the requirement that you define your application using containers there are no other major restrictions between Kubernetes Engine and Compute Engine. That said, the big difference comes next in terms of complexity.

### **10.8.2 Complexity**

As we've seen, computing environments can be complicated and Kubernetes Engine (which relies on Kubernetes under the hood) is no different. In this case though, the complications are really initial learning costs in order to benefit from the overall simplification. In other words, while a car is a lot more complex than a bicycle, but once you learn how to drive the car the benefits become clear.

Since we only scratched the surface of what Kubernetes is capable of, the complexity of the system as a whole really wasn't demonstrated as much as it could've been, but it was still far more complicated than "turn on a VM". Putting this into realistic context, if you wanted to deploy a simple to-do list application with a single node that would never really need to grow beyond that node, Kubernetes Engine is likely to be overkill. If, on the other hand, you wanted to deploy a large cluster of API servers to handle huge spikes of traffic, it's probably worth the effort to understand Kubernetes and then maybe rely on Kubernetes Engine to manage your Kubernetes cluster.

### **10.8.3 Performance**

Unlike using raw VMs like Compute Engine, Kubernetes has a few layers of abstraction between application code and the actual hardware executing that code. As a result, the overall performance can't be as good as a plain old VM and certainly not as good as a non-virtualized system. In other words, the performance of Kubernetes Engine won't be as efficient as something like Compute Engine, however efficiency and performance are not the same thing.

This means that while you might need more nodes in your cluster to get the same performance as using non-virtualized hardware, you can more easily change the overall performance capacity of your system with Kubernetes Engine than you can with

Compute Engine or non-virtualized machines. As a result, if you know your performance requirements exactly, and you're sure that they'll stay exactly the same over time, using Kubernetes Engine would be providing you with flexibility that you don't need at all. On the other hand, if you're unsure of how much power you need and want the ability to change your mind whenever you want, Kubernetes Engine makes that easy and simple, with a slight cost on overall efficiency.

Further, since this efficiency difference is so slight it really should only come into question when you have an enormous deployment of hundreds of machines (where the slight differences add up to become meaningful differences). If your system is relatively small, the efficiency differences shouldn't even be noticeable.

#### **10.8.4 Cost**

For small clusters, Kubernetes Engine is no more costly than the raw Compute Engine VMs that are needed to power the underlying Kubernetes cluster. On the other hand, larger clusters incur an additional hourly cost for the master node, which makes Kubernetes Engine rank slightly higher on the cost scale.

That said, while this might be a meaningful amount for a cluster that is just barely over the limit (say, 6 or 7 nodes), as your cluster grows this number will become less and less material when compared to the overall cost of the cluster. This means that generally, you should consider the overall cost of a Kubernetes Engine cluster to be mostly the same as a cluster of Compute Engine VMs of the same size.

#### **10.8.5 Overall**

At this point, you may be wondering how to choose between Computer Engine and Kubernetes Engine given that they're both very flexible, perform similarly, and are priced similarly, but using Kubernetes Engine means that you have to learn and understand Kubernetes which itself is pretty complex. While this is all true, the distinguishing factor tends to rely on how large your overall system will be and how much you want your deployment configuration to be represented as code. In other words, the benefits of using Kubernetes Engine over other computing platforms isn't about the cost or the infrastructure, but about the benefits of Kubernetes itself as a way of keeping your deployment procedure clear and well documented.

As a result, the general rule of thumb is to use Kubernetes Engine when you have a large system that you (and your team) will need to maintain over a long period of time. On the other hand, if you have a need for a few VMs to do some computation and plan to turn them down after a little while, relying on Compute Engine might be easier. To make this more concrete, let's walk through the three example applications that we've discussed and see which makes more sense.

##### **To-do list**

The sample to-do list application is a simple application that keeps track of to-do lists and whether they're done or not. As a result, it's unlikely that this application will need to "scale up" due to extreme amounts of load. As a result, Kubernetes Engine is

probably a bit overkill for the needs.

**Table 10.1. To-do list application computing needs**

Aspect	Needs	Good fit?
Flexibility	Not all that much	Overkill
Complexity	Simpler is better	Not so good
Performance	Low to moderate	Slightly overkill during non-peak time
Cost	Lower is better	Not ideal, but not awful either

Overall, the to-do list application, while it can run on Kubernetes, is probably not going to make use of all the features, and will require a bit more learning than is necessary for this application. As a result, something simpler like a single Compute Engine VM might be a better choice.

#### **E\*Exchange**

E\*Exchange, the online stock trading platform, has many more complex features, and each of these may be divided into many different categories. For example, there may be an API server that handles requests to the main storage layer, a separate service to handle sending e-mails to customers, another that handles a web-based user interface, and still another that might handle caching the latest stock market data. This is quite a few distinct pieces, which might get you thinking about each as a set of containers, with some that might be grouped together into a pod.

**Table 10.2. E\*Exchange computing needs**

Aspect	Needs	Good fit?
Flexibility	Quite a bit	Definitely
Complexity	Fine to invest in learning	Definitely if it makes things easy
Performance	Moderate	Definitely
Cost	Nothing extravagant	Definitely

Since E\*Exchange was a reasonable fit for Compute Engine, it is likely to be a good fit for Kubernetes Engine. It also turns out that the benefits of investing in learning Kubernetes and deploying the services using Kubernetes Engine might end up saving quite a bit of time and simplifying the overall deployment process for the application. Unlike the to-do list application there are quite a few distinct pieces of this application, each with their own unique needs and requirements. Using multiple different pods for these means that you can keep them all in a single cluster and scale them up or down as needed. Overall, Kubernetes Engine is probably a great fit for the E\*Exchange application.

#### **InstaSnap**

InstaSnap, the very popular social media photo sharing application, lies somewhere in the middle of the two previous examples in terms of overall system complexity. It probably doesn't have as many distinct systems as E\*Exchange, but definitely has

more than the simple to-do list service. For example, there might be an API server that handles requests from the mobile app, a service for the web-based UI, and perhaps a background service that handles processing videos and photos into different sizes and formats. That said, the biggest concern for InstaSnap is performance and scalability. In other words, we may need the ability to increase the resources available to any of the various services if there is a spike in demand (which happens often). This requirement makes InstaSnap a great fit for Kubernetes Engine because we can easily resize the cluster as a whole as well as the number of pod replicas running in the cluster.

**Table 10.3. InstaSnap computing needs**

Aspect	Needs	Good fit?
Flexibility	A lot	Definitely
Complexity	Eager to use advanced features	Definitely
Performance	High	Mostly
Cost	No real budget	Definitely

As we can see here, even though we don't have as many distinct services, Kubernetes Engine is still a great fit, particularly when it comes to using the advanced scalability features. Even though the performance itself is slightly lower based on more abstraction happening under the hood, this requirement really doesn't have much of an effect on the choice of a computing platform because we can always add more machines if we need more capacity (which is OK due to the "No real budget" requirement for cost).

## 10.9 Summary

- Containers are an infrastructural tool that make it easy to package up code along with all dependencies down to the operating system.
- Docker is the most common way of defining a container, using a format called a Dockerfile.
- Kubernetes is an open-source system for orchestrating containers, helping them act as a cohesive application.
- Kubernetes Engine is a hosted and fully-managed deployment of Kubernetes, minimizing the overhead of running your own Kubernetes cluster.
- You can manage your Kubernetes Engine cluster just like any other Kubernetes cluster, using `kubectl`.

# 11

## *App Engine: Fully managed applications*

**This chapter covers:**

- What is App Engine?
- Building an application using App Engine Standard and App Engine Flex
- Managing how your applications scale up and down
- Using App Engine Standard's managed services
- When is App Engine a good fit?

### 11.1 What is App Engine?

As we've learned, there are many different types of computing platforms, with a large variety in terms of complexity, flexibility, and performance. Where Compute Engine was an example of "low-level infrastructure" (just a VM), App Engine is a fully-managed cloud computing environment that aims to consolidate all of the work needed when deploying and running your applications. This means that in addition to being able to run some code like you would on a VM, App Engine offers several other additional services that come in handy when building applications.

For example, if we had a to-do list application where we store lists of work we need to finish, it's not unusual that we'd need to store some data, send e-mails, or schedule a background job every day (like recomputing your to-do list completion rate). Typically we'd need to do all of this ourselves by turning on a database, signing up for an e-mail sending service, running a queuing system like RabbitMQ, and relying on Linux's cron service to coordinate it all. App Engine offers a suite of hosted services to do this that you don't have to manage yourself.

App Engine itself is made up of two separate environments which have some pretty important differences. While one of these environments is built using open-source tools like Docker containers, the other is built using more proprietary technology that allows Google to do interesting things when automatically scaling your app, but imposes quite a few limitations on what you can actually do with your code. This means that even though these are both under the "App Engine umbrella" they are pushing against the boundaries of what you could consider a single product. As a result, we'll look at these environments together in a single chapter, but there will be a few places where we'll hit a fork in the road and it makes sense to split the two environments apart.

The App Engine Standard Environment, released in early 2008, offers a fully-managed computing environment complete with storage, caching, computing, scheduling, and more, but is limited to a few specific programming languages. In this type of environment, your application tends to be specifically tailored to App Engine, but comes with the benefit of living in an "always auto-scaling" environment. This means that sudden spikes of traffic sent to your application are handled gracefully and periods where your application is inactive don't cost any money at all.

App Engine Flexible Environment (often just called App Engine Flex) aims at providing a fully-managed environment with fewer restrictions and somewhat more portability, trading some scalability in exchange. App Engine Flex is based on Docker containers, you're not limited to any specific versions of programming languages, and you can still take advantage of many the other benefits of App Engine such as the hosted cron service.

If you're confused about which environment is right for you, keep reading on. We'll first explore some of the organizational concepts, then dig into the details, and finally look at how to choose whether App Engine is right for you. If it turns out that App Engine is a great fit, we'll explore how to choose which of the two environments fits best.

### **11.1.1 Concepts**

Since App Engine is a hosted environment, the API-layer has a few more organizational concepts that we'll need to understand in order to use it as a computing platform. In other words, App Engine wants to understand more about your application so it uses a few different organizational pieces: applications, services, versions, and instances.

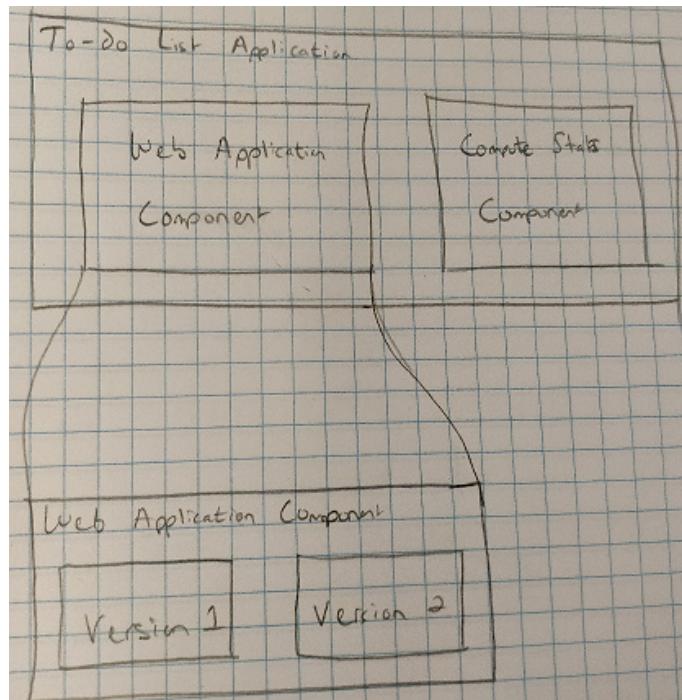
**NOTE**

Keep in mind that even though there are different environments for App Engine, the concepts across both environments are the same (or very similar). You won't have to re-learn these concepts to switch between environments.

In addition to looking at your application in terms of its components, App Engine also keeps track of the different versions of those components. For example, in our to-do list application we might break the system into separate components: one component representing the web-application itself and another responsible for recomputing statistics every day at midnight. After that, revising components from time to time,

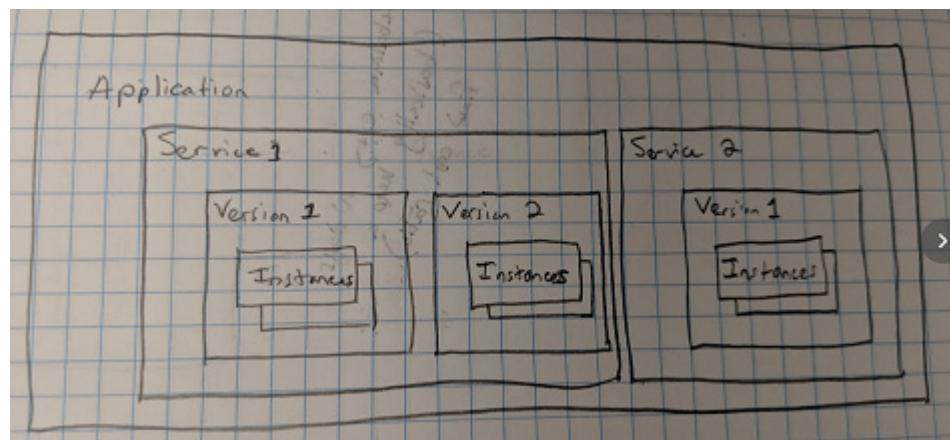
(e.g., fixing a bug in the web-application) might bring about a new "version" of that component.

**Figure 11.1. An overview of a to-do list application's components and versions**



All of these things are used and understood by App Engine, which we'll explore in more detail in this section.

**Figure 11.2. An overview of App Engine concepts**



Let's start at the top by looking at the idea of an App Engine "application".

### APPLICATIONS

The basic starting place of using App Engine to host your work is the top-level application. Each of your projects is limited to a single application, with the idea that each project should be limited to a single purpose. Just like a project acts as a container for your Compute Engine VMs, the application itself acts as a container as well for your code which may be spread across multiple services (which we'll discuss in the next section).

The application also has lots of different settings, some of which are easily configurable and simple to change, while others are permanent and locked to the application once set. For example, you can always re-configure a new SSL certificate for your application, but once you've chosen the region for your application, that can't be changed.

**NOTE**

**The location of your application also impacts how much it costs to run, which we explore towards the end of the chapter.**

To see how this actually works, click on the App Engine section in the left-side navigation of the Cloud Console. If you haven't configured App Engine before, you'll be asked to choose a language (e.g., Python) and then you'll land on a page where you choose the location of your application. This particular setting controls where the physical resources for your application will live, and is an example of a setting which, once chosen, cannot be changed.

**Figure 11.3. Choosing a location for an App Engine application**

The screenshot shows the 'Select a location' step of the 'Your first app with Python' wizard. At the top, there are two tabs: 'App Engine' and 'Your first app with Python'. Below the tabs, there are two buttons: '1 Select a location' (highlighted in blue) and '2 Deploy'. The main content area is titled 'In which region would you like to serve your app?'. It includes a note: 'Your app will be served from the selected region. Anyone can use your app, but users closer to the selected region will have lower latency. You can't change the region for this project later.' Below the note is a world map with several regions marked with blue location pins. The regions labeled are: NORTH AMERICA, EUROPE, ASIA, OCEANIA, AUSTRALIA, SOUTHERN OCEAN, and SOUTH AMERICA. The map also shows the Atlantic Ocean, Indian Ocean, and Pacific Ocean. At the bottom left, there is a dropdown menu labeled 'Select a region' with 'us-central' selected. At the bottom right, there are links for 'Map data ©2017' and 'Terms of Use'.

Outside of that, the more interesting pieces are those contained by an App Engine application, so let's move onto looking at App Engine "services".

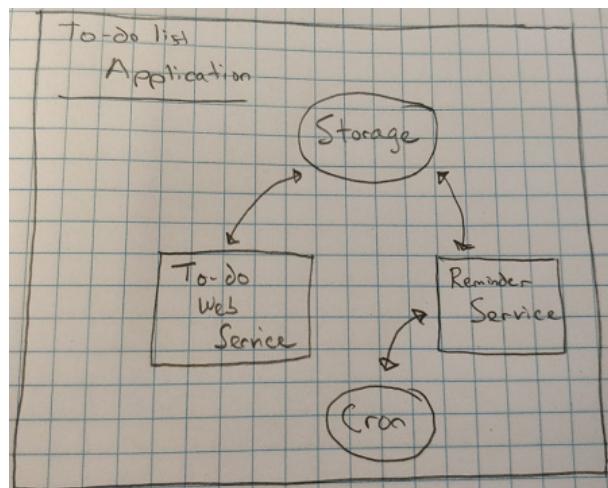
## SERVICES

Services on App Engine are intended to act as a way to split your application into smaller, more manageable pieces. Similar to micro-services, App Engine services act as independent pieces of computing, but typically share access to the various App Engine services. For example, you can access the same cron service mentioned above from any of the various services you might have as part of your application.

For example, imagine that you are building a web-application that tracks your to-do

list. At first it might just be simple text, but as you grow you may want to add a feature that sends e-mail reminders to finish something on your list. In that case, rather than trying to add the "e-mail reminder" feature to the main application, you might define it as a separate service inside your application. In other words, since its job is completely isolated from the main job of storing to-do items, it can live as a separate service and avoid cluttering the main application.

**Figure 11.4. To-do list application with two services**



The service itself consists of your source code files and extra configuration, such as which runtime to use (for App Engine Standard), and unlike applications (which has a one-to-one relationship with your project) services can be created (deployed) as well as deleted. The first set of source code that you deploy on App Engine will create a new service and this will be registered as the default service for your application, meaning that when you make a request for your application without specifying the service, that request is routed to this new default service.

In addition to all of this, services act as yet another container for different revisions of your application. This means that in our example above, you could deploy new versions of your reminder service to your application without any need to touch the web-application service. As you might guess, having this ability to isolate changes between related systems can be very useful particularly with large applications built by large teams, each team owning their own distinct piece of the application.

Let's take a moment to look at how versions work.

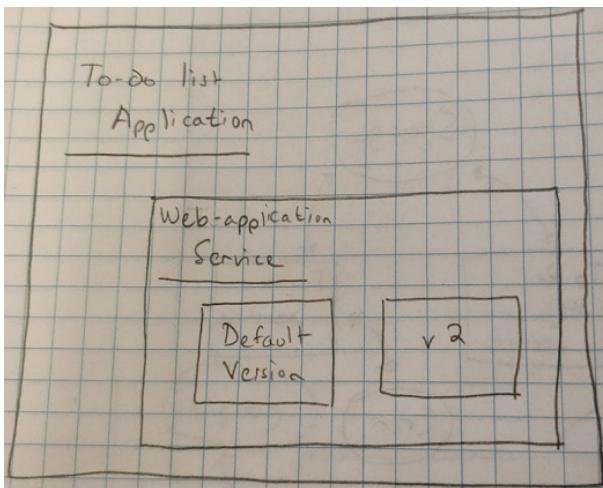
#### VERSIONS

Versions themselves are a lot like point-in-time snapshots of a service. In other words, if your service is a bunch of code inside a single directory, a version of your service corresponds to the code in that directory at the exact time that you decided to deploy it to App Engine. A neat side-effect of this is that you can have multiple versions of the

same service all running at the same time.

Just as deploying your first App Engine service, becomes the "default" service for your entire application, the code that you deploy in that first service becomes the "default" version of that service. Further, just as you can address an individual service of your application, you can address an individual version of any given service as well. For example, you can see your web-application by navigating to `webapp.my-list.appspot.com` (or explicitly, `default.webapp.my-list.appspot.com`), you can view a "newly deployed" version (let's say we called that version v2) by navigating to `v2.webapp.my-list.appspot.com`.

**Figure 11.5. Deploying a new version of the web-application service**

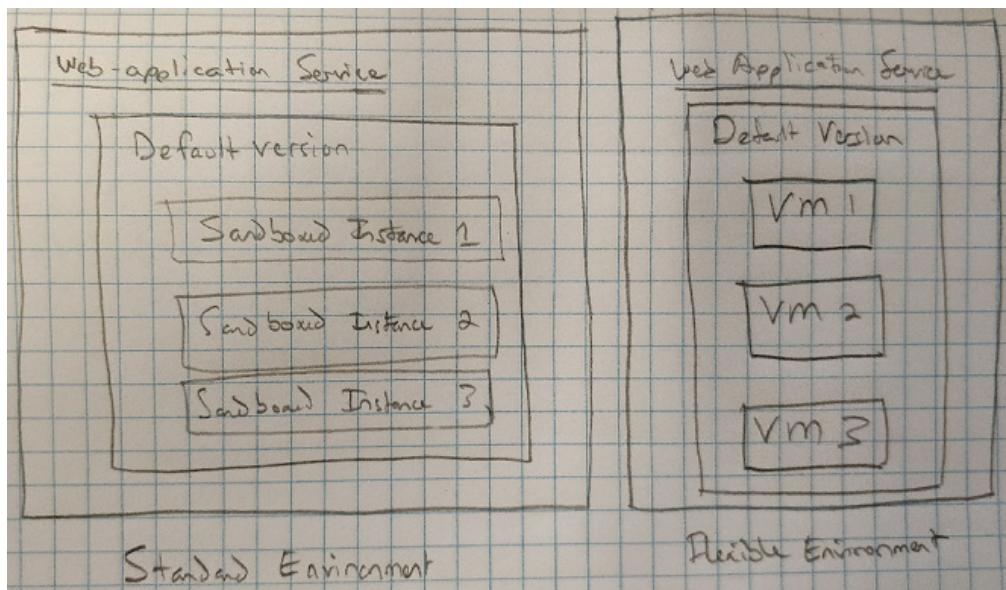


Now that you understand the organizational concepts, let's take a moment to look at an infrastructural one: instances.

### INSTANCES

So far we've looked at the organizational concepts in App Engine, but we haven't really seen how App Engine goes about running your code. After what we've learned so far it should come as no surprise that App Engine uses the concept of an "instance" to mean a chunk of computing capacity for your application. Unlike the concepts above, there are a couple of slight differences in the instances depending on whether you're using the Standard or Flexible environment, which are worth exploring in a bit more detail.

**Figure 11.6. App Engine instances for Standard vs Flexible environments**



In App Engine Standard, these instances represent an abstract chunk of CPU and memory available to run your application inside a special sandbox. Further, these instances scale up and down automatically based on how many requests are being sent to your application, and since they are lightweight sandbox environments, your application can scale from zero to thousands of instances very quickly. You can choose the "type" of App Engine instance to use for your application from a list of available types that have varying costs and amounts of CPU and memory.

Since App Engine Flex is built on-top of Compute Engine and Docker containers, it simply uses Compute Engine instances to run your code which comes with a couple important caveats. First, since Compute Engine VMs take a bit of time to turn on, Flex applications must always have at least a single VM instance running at any given time. Because of this, Flex applications end up costing money around the clock. Second, this additional start-up time also means that if you see a huge spike of traffic to your application, it might take a bit to scale up to handle the traffic. During this time, existing instances could become overloaded which would lead to timeouts for incoming requests.

It's also important to remember that App Engine instances are specific to a single version of your service, which means that a single instance only handles requests for the specific version of the service that was sent the request. As a result, if you host lots of versions concurrently those versions will spawn instances as necessary to service the traffic, and if running inside App Engine Flex this means that each version will have at least one VM running at all times.

Now that we have a grasp on the different concepts in App Engine, let's actually get

down to business and look at how to actually use App Engine.

## 11.2 Interacting with App Engine

At this point you should have a decent understanding of the underlying organizational concepts that App Engine has (such as services or versions), but that's not all that helpful until we actually do something with those. To dig into this, let's create a simple "Hello world" application for App Engine, deploy it, and verify that it works. Let's start by building our application for App Engine Standard first.

### 11.2.1 Building an application in App Engine Standard

As we discussed previously, App Engine Standard is a fully-managed environment where you code runs inside a special sandbox rather than a full virtual machine like it would on Compute Engine. As a result, this means that we have to build our "Hello world" application using one of the approved languages. Of the languages available (PHP, Java, Python, and Go), Python seems as good a choice as any (it's easy to read and pretty powerful), so just for this section, we're going to switch to using Python to build our application.

**NOTE**

Don't worry if you aren't familiar with Python. Any Python code that isn't super obvious will be annotated to explain what it does.

One thing to keep in mind, however, is that white-space (e.g., spaces and tabs) are important in Python. This means that if you find yourself with syntax errors in your Python code, it very well could be that you used a tab when you meant to use four spaces, so be careful!

Before we get into building our application code itself, we first need to make sure we have the right tools installed, since we'll need them to deploy our code to App Engine.

#### INSTALLING PYTHON EXTENSIONS

In order to develop locally using App Engine (and specifically using App Engine Standard's Python runtime), we'll need to install the Python extensions for App Engine, which you can do using the `gcloud components` subcommand. This package contains the local development server, various emulators, and other resources and libraries needed to build a Python App Engine application.

#### Listing 11.1. Installing the App Engine Python extensions

```
$ gcloud components install app-engine-python
```

**TIP**

If you installed the Cloud SDK using a system-level package manager (like `apt-get`), you'll get an error message saying to use that same package manager and the command to run to install the Python extensions.

Now that we've got everything installed, we can get to the real work of building our application.

## CREATING AN APPLICATION

Since we're just testing out App Engine, let's start by focusing on nothing more than a "Hello, world!" application that sends a static response back whenever it receives a request. We'll start our Python app by using the `webapp2` framework which is compatible with App Engine.

**NOTE**

You can use other libraries and frameworks as well (such as Django or Flask), but `webapp2` is the easiest to use with App Engine.

The following snippet shows a simple `webapp2` application that defines a single request handler and connects it to the root URL (/). In this case, whenever you send a GET HTTP request to this handler, it simply sends back "Hello, world!"

### Listing 11.2. Defining our simple web application

```
import webapp2

class HelloWorld(webapp2.RequestHandler):
    def get(self):
        self.response.write('Hello from App Engine!');

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
])
```

Let's put this code into a file called `main.py` and then we'll move over to defining the configuration of our App Engine application. The way we tell App Engine how to configure an application is with an `app.yaml` file. YAML (Yet Another Markup Language) is just an easily readable syntax for some structured data, which looks a lot like Markdown, and the `app.yaml` name is a special file that App Engine looks for when you deploy your application. It contains a whole bunch of settings about the runtime involved, handlers for URL mappings, and more. The `app.yaml` file that we'll use is shown below.

### Listing 11.3. Defining app.yaml

```
runtime: python27
api_version: 1
threadsafe: true
handlers:
- url: /.*
  script: main.app
```

- ➊ Here we tell App Engine to run our code inside the Python 2.7 sandbox.
- ➋ This tells App Engine which version of the API we're consuming. Currently, there is only one version for Python 2.7, so this should be set to 1.

- ③ The `threadsafe` parameter tells App Engine that you've written your code to be thread safe and that App Engine can safely spawn multiple copies of your application without worrying about those threads stepping on each other.
- ④ This section holds the "handlers" that map URL patterns to a given script.
- ⑤ This is a regular expression that is matched against requests. If a request URL matches, the script will be used to handle the request.
- ⑥ Here we point to the `main.py` file, but tell App Engine to treat it as a WSGI application (basically says to look at the `app` variable in `main.py`)

At this point we have everything we need to actually test out our application, so let's try running it locally and making sure it works as we want.

#### TESTING THE APPLICATION LOCALLY

To run our application we'll use the App Engine development server, which was installed as `dev_appserver.py` when we installed the Python App Engine extensions. Just navigate to the directory that contains your `app.yaml` file (and the `main.py` file), and run `dev_appserver.py` pointing to the current directory (`.`). You should see some debug output that says where the application itself is available (usually on `localhost` on port `8080`). Once the development server is running with our application, we can test that it did the right thing by connecting to [localhost:8080/](http://localhost:8080/).

#### **Listing 11.4. Verifying the application is running**

```
$ curl http://localhost:8080/
Hello from App Engine!
```

So far so good! Now that we've built and tested our application, it's time to see whether we can deploy it to App Engine itself. After all, running the application locally isn't going to work when we have lots of incoming traffic.

#### DEPLOYING TO APP ENGINE STANDARD

Deploying the application to App Engine is pretty easy since we have all the right tools installed. To do this, we can use the `gcloud app deploy` sub-command, confirm the place we're deploying to, and then just wait for the deployment to complete, shown below.

#### **Listing 11.5. Deploying the application to App Engine**

```
$ gcloud app deploy
Initializing App Engine resources...done.
Services to deploy: ①
descriptor:      [/home/jjg/projects/appenginehello/app.yaml]
source:          [/home/jjg/projects/appenginehello]
target project:  [your-project-id-here] ②
target service:  [default] ③
target version: [20171001t160741] ④
target url:     [https://your-project-id-here.appspot.com] ⑤
```

```
Do you want to continue (Y/n)? Y 6
Beginning deployment of service [default]...
Some files were skipped. Pass `--verbosity=info` to see which ones.
You may also view the gcloud log file, found at
[/home/jjg/.config/gcloud/logs/2017.10.01/16.07.33.825864.log].
Uploading 2 files to Google Cloud Storage
File upload done.
Updating service [default]...done.
Waiting for operation [apps/your-project-id-here/operations/1fad9f55-35bb-45e2-8b17-3cc8cc5b1228] to complete...done.
Updating service [default]...done.
Deployed service [default] to [https://your-project-id-here.appspot.com]

You can stream logs from the command line by running:
$ gcloud app logs tail -s default

To view your application in the web browser run:
$ gcloud app browse
```

- ➊ Here the deployment tool verifies the configuration about what we're planning to deploy.
- ➋ As you learned, the project ID and the application ID are the same (since they have a one-to-one relationship).
- ➌ If no service name is set (which is the case here), App Engine uses the "default" service.
- ➍ Similarly to services, if no version name is specified, App Engine generates a default version number based on the date.
- ➎ After deploying your application, it will be available at a URL in the appspot.com domain.
- ➏ Here the deployment tool asks you to confirm the deployment parameters to avoid accidentally deploying the wrong code or to the wrong service.

Once this is completed, you can verify that everything worked either using the curl command or through your browser by sending a GET request to the target URL noted above.

#### **Listing 11.6. Verifying the application works in App Engine**

```
$ curl http://your-project-id-here.appspot.com
Hello from App Engine!
```

You can also verify that SSL works with your application by connecting using https:// as the scheme instead of plain http://.

#### **Listing 11.7. Verifying that SSL works as expected**

```
$ curl https://your-project-id-here.appspot.com
Hello from App Engine!
```

Lastly, since the service name is officially "default", we can address this service

directly at `default.your-project-id-here.appspot.com`, shown below.

#### **Listing 11.8. Addressing the default service directly**

```
$ curl http://default.your-project-id-here.appspot.com
Hello from App Engine!
```

You can check inside the App Engine section of the Cloud Console and see how many requests have been sent, how many instances you have turned on currently, and more. An example of what this might look like is shown below.

**Figure 11.7. App Engine overview in the Cloud Console**

The screenshot shows the Google Cloud Platform App Engine Dashboard. On the left, there's a sidebar with various icons and links: Services, Versions, Instances, Task queues, Security scans, Firewall rules, Quotas, Blobstore, Memcache, Search, and Settings. The main area has a title "Dashboard". At the top, it says "Service default" and "Version 20171001t160741 (100%)". Below that is a "Summary" dropdown menu with options like "Summary", "Logs", "Metrics", "Logs & Metrics", and "Logs & Metrics (Experimental)". There are also time range filters: "1 hour", "6 hours", "12 hours", "1 day", "2 days", "4 days", "7 days", "14 days", and "30 days". To the right, it shows the URL `jig-personal.appspot.com` and the region "us-central". A summary chart titled "Summary" shows "Count/sec" over time from 4:50 to 5:45. It has a sharp peak around 5:25. A note says "Total requests: 0.25/s". Below the chart is a timestamp "Oct 2, 2017 5:24 AM". Under the chart, there's a "Debug | Source" link. Further down, there's an "Instance summary" section with a table:

App Engine Release	Total Instances	Average QPS	Average Latency	Average Memory
1.9.54	1	0	0 ms	13.8 MB

Below this is a "Billing status" section with a note: "Enabled (Daily spending limit: Unlimited) Change" and "Quotas reset every 24 hours. Next reset: 22 hrs". It shows a table of resources and their costs:

Resource	Usage	Billable	Price	Cost
Frontend Instance Hours	0.37 Instance Hours	0.00	\$0.05 / Hour	\$0.00
Outgoing Bandwidth	0.000002 GB	0.00	\$0.12 / GB	\$0.00
Cloud Storage Class A Operations	0	0	\$5.00 / Million Ops	\$0.00
Cloud Storage Network (Egress) - Americas and EMEA	0.000002 GB	0.00	\$0.12 / GB	\$0.00
Estimated cost for the last 2 hours				\$0.00*

So how do we create new services then? Let's take a moment and explore how to deploy code to a service other than the default which we've already done.

## DEPLOYING ANOTHER SERVICE

In some ways, you can think of a new service like a new chunk of code. As a result, we'll need to make sure we have a safe place to put this code. Commonly, the easiest way to do this is to separate code by directory, where the directory name matches up with the service name.

To show how this works, let's make two new directories called `default` and `service2`, and move the `app.yaml` and `main.py` files into each directory. This effectively rearranges our code so that we have two copies of both the code and the configuration in each directory.

To see this more clearly, here's how it should look when we're done.

### **Listing 11.9. Layout of multiple services**

```
$ tree
.
└── default
    ├── app.yaml
    └── main.py
└── service2
    ├── app.yaml
    └── main.py

2 directories, 4 files
```

Now let's do a few things to define a second service (and clarify that the current service happens to be the default):

1. Update both `app.yaml` files to **explicitly** pick a service name. This means that `default` will actually be called `default`.
2. Update `service2/main.py` to print something else.
3. Re-deploy both services over again.

After updating the `app.yaml` files of each, they should look like the following.

### **Listing 11.10. Updated default/app.yaml**

```
runtime: python27
api_version: 1
threadsafe: true
service: default ①

handlers:
- url: /.*
  script: main.app
```

- ① Here we explicitly state that the service involved is the "default" one. This has no real effect in this case, but it clarifies what this `app.yaml` file controls.

**Listing 11.11. Updated service2/app.yaml**

```
runtime: python27
api_version: 1
threadsafe: true
service: service2 ①

handlers:
- url: /.*
  script: main.app
```

- ① Here we choose a new service name. It can be basically any ID-style string that you want.

As you can see here, we've made it explicit that each `app.yaml` file controls a different service. We've also made sure the service name matches the directory name, meaning it's easy to keep track of all of this.

Next, we can update `service2/main.py`, changing the output so that we know it came from this other service. Doing this might make your application look like the following.

**Listing 11.12. "Hello, world!" application in Python**

```
import webapp2

class HelloWorld(webapp2.RequestHandler):
    def get(self):
        self.response.write('Hello from service 2!'); ①

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
])
```

- ① Here we make it clear that "Service 2" is responding.

Finally, we can deploy our new service simply by running `gcloud app deploy` and pointing at the `service2` directory instead of the `default` directory.

**Listing 11.13. Deploying the new service to App Engine**

```
$ gcloud app deploy service2 ①
Services to deploy:

descriptor:      [/home/jjg/projects/appenginehello/service2/app.yaml] ②
source:          [/home/jjg/projects/appenginehello/service2]
target project:  [your-project-id-here]
target service:  [service2] ③
target version:  [20171002t053446]
target url:      [https://service2-dot-your-project-id-here.appspot.com] ④

# ... More information here ...
```

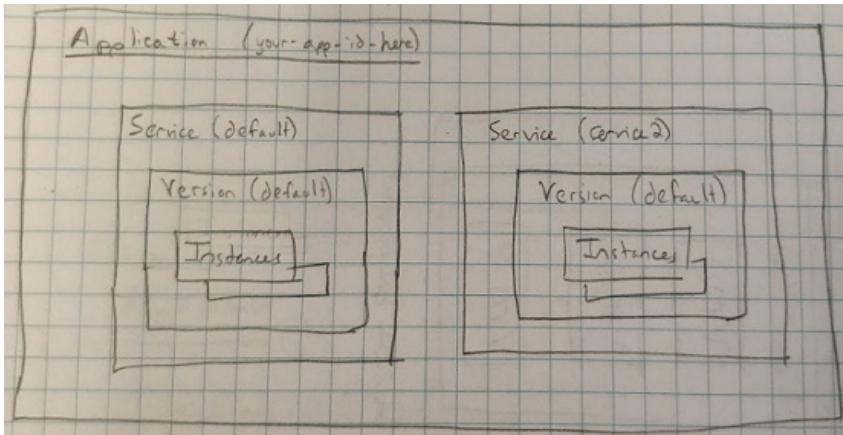
- ① Here we point the `gcloud` deployment tool to our `service2` directory.

- ② As a result, the deployment tool looks for the copied app.yaml file, which states a different service name.
- ③ The tool figures out that the service name should be service2 as we defined.
- ④ Since the service name isn't default, we get a separate URL where we can access our code.

And just like before, our new application service should be live!

At this point, our system conceptually looks a bit like the following diagram.

**Figure 11.8. Organizational layout of our application so far**



We can verify that the deployment worked by navigating to the URL again in a browser or we can use the command-line, shown below.

#### **Listing 11.14. Verifying that we can access the newly deployed service**

```
$ curl https://service2-dot-your-project-id-here.appspot.com
Hello from service 2!
```

If this URL looks strange to you, that's not unusual. The syntax of <service>-dot-<application> is definitely new. The reason this exists is due to how SSL certificates work. App Engine ensures that \*.appspot.com is secured, but doesn't allow additional "dots" nested deeper in the DNS hierarchy. This means that when accessing your app over HTTP (not HTTPS), you can technically make a call to <service>. <application>.appspot.com, but if you were to try that with HTTPS, you'd run into trouble, shown here.

#### **Listing 11.15. Accessing our service with . separated DNS name**

```
$ curl http://service2.your-project-id-here.appspot.com
Hello from service 2!
$ curl https://service2.your-project-id-here.appspot.com ❶
# Error
```

- ➊ Here the result is an error code due to the SSL certificate not covering the domain specified.

Now that we've seen how to deploy a brand new service, let's look at a slightly less adventurous change by deploying a new version of an existing service.

### DEPLOYING A NEW VERSION

While you may only create new services every so often, any new updates to your application will probably result in a new version, and updates happen far more often. So how do we actually update versions? Where are the versions stored?

To start, let's confirm how our application is laid out. We can inspect our current application either in the Cloud Console or from the command-line, shown below.

#### **Listing 11.16. Listing App Engine deployed services and versions**

```
$ gcloud app services list
SERVICE  NUM_VERSIONS
default   1
service2  1

$ gcloud app versions list
SERVICE  VERSION          TRAFFIC_SPLIT  LAST_DEPLOYED           SERVING_STATUS
default  20171001t160741  1.00        2017-10-01T16:08:18-04:00  SERVING
service2  20171002t053446  1.00        2017-10-02T05:34:56-04:00  SERVING
```

As you can see, we currently have two services, each with a single default version specified. So let's imagine that we want to update default service but this "update" should be create a new version and not overwrite the currently deployed version of the service.

To do this, we have two options. The first is to rely on App Engine's default version naming scheme which is based on the date and time when the deployment happened. That is, when you deploy your code, App Engine actually creates a new version automatically for you and never overwrites the currently deployed version, which is especially helpful when you accidentally deploy the wrong code! The other is to use a special flag (-v) when deploying our code and the result will be a new version named as we specified.

This means that if we want to update our default version, we can make our code changes and deploy it like we did before. In this example, we'll update the code to say "Hello from version 2!" Once the deployment completes, we can verify that everything worked as expected by trying to access the URL as before.

#### **Listing 11.17. Verifying that the new version was successfully deployed**

```
$ curl https://your-project-id-here.appspot.com
Hello from version 2!
```

This might look like we've accidentally blasted out the previous version, but if we inspect the list of versions again we'll see that the previous version is still there and

serving traffic.

#### **Listing 11.18. Listing available versions for the default service**

```
$ gcloud app versions list --service=default
SERVICE  VERSION      TRAFFIC_SPLIT  LAST_DEPLOYED          SERVING_STATUS
default  20171001t160741  0.00        2017-10-01T16:08:18-04:00  SERVING
default  20171002t072939  1.00        2017-10-02T07:29:46-04:00  SERVING
```

Notice that the "traffic split" between the two versions has shifted and all traffic is pointing to the later version, with zero traffic being routed to the previous version. We'll discuss this in more detail later on. So if the version is still there, how can we talk to it? It turns out that just like we can access a specific service directly, we can access the previous version by addressing it directly as well in the format of <version>. <service>.your-project-id-here.appspot.com (or using -dot-separators for HTTPS), shown below.

#### **Listing 11.19. Verifying that we can access the previous version**

```
$ curl http://20171001t160741.default.your-project-id-here.appspot.com
Hello from App Engine!
$ curl https://20171001t160741-dot-default-dot-your-project-id-here.appspot.com
Hello from App Engine!
```

If you're worried about a new version "going live" right away, this is completely reasonable. It turns out that there is a way to tell App Engine that you want the new version deployed, but you don't want to immediately route all traffic to the new version. Let's update the code again to change the message and deploy another version, without it becoming the live version immediately. To do this we'll set the `promote_by_default` flag to "false".

#### **Listing 11.20. Deploying a new version without promoting it immediately**

```
$ gcloud config set app/promote_by_default false
Updated property [app/promote_by_default].

$ gcloud app deploy default
Services to deploy:

descriptor:      [/home/jjg/projects/appenginehello/default/app.yaml]
source:          [/home/jjg/projects/appenginehello/default]
target project:  [your-project-id-here]
target service:  [default]
target version:  [20171002t074125]
target url:      [https://20171002t074125-dot-your-project-id-here.appspot.com]

(add --promote if you also want to make this service available from
[https://your-project-id-here.appspot.com])

# ... More information here ...
```

At this point, the new service version should be deployed but not actually live and serving requests. We can look at the list of services to verify this, or check by making a request to the target URL as we did before.

#### **Listing 11.21. Listing available versions for the default service**

```
$ gcloud app versions list --service=default
SERVICE  VERSION      TRAFFIC_SPLIT  LAST_DEPLOYED      SERVING_STATUS
default   20171001t160741  0.00        2017-10-01T16:08:18-04:00  SERVING
default   20171002t072939  1.00        2017-10-02T07:29:46-04:00  SERVING
default   20171002t074125  0.00        2017-10-02T07:41:31-04:00  SERVING

$ curl http://your-app-id-here.appspot.com/
Hello from version 2!
```

We can also verify that the new version was deployed correctly by accessing it the same way as we did before.

#### **Listing 11.22. Verifying that the newly deployed service is available**

```
$ curl http://20171002t074125.default.your-project-id-here.appspot.com
Hello from version 3 which is not live yet!
```

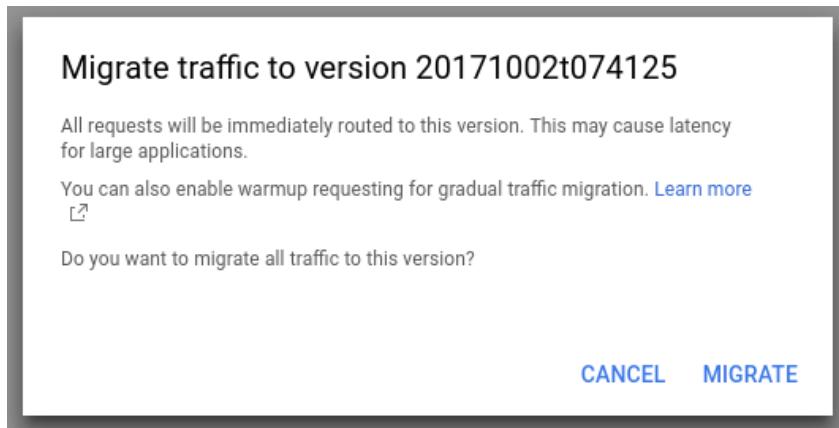
Once we see that it works the way we expect, we can safely promote the new version by migrating all traffic to the new version using the Cloud Console. To do this, we simply browse to the list of versions, then check the version we want to migrate traffic to, and click "Migrate traffic" at the top of the page.

**Figure 11.9. Check the box for the version and click "Migrate traffic"**

Version	Status	Traffic Allocation	Instances	Runtime	Environment	Size	Deployed	Diagnose	Tools	View
<input checked="" type="checkbox"/> 20171002t074125	Serving	<div style="width: 0%;">0%</div>	0	python27	Standard	358 B	Oct 2, 2017, 7:41:31 AM by jjg@google.com		<a href="#">Tools</a>	<a href="#">View</a>
<input type="checkbox"/> 20171002t072939	Serving	<div style="width: 100%;">100%</div>	1	python27	Standard	336 B	Oct 2, 2017, 7:29:46 AM by jjg@google.com		<a href="#">Tools</a>	<a href="#">View</a>

When you click the button, you'll see a pop-up where you can confirm that you want to route all new traffic to the selected version.

**Figure 11.10. Confirm that we want to migrate traffic to the new version**



When this is complete, we'll see that 100% of traffic is being sent to our new version.

**Listing 11.23. Listing the available versions for the default service**

```
$ gcloud app versions list --service=default
SERVICE VERSION      TRAFFIC_SPLIT LAST_DEPLOYED          SERVING_STATUS
default  20171001t160741 0.00      2017-10-01T16:08:18-04:00  SERVING
default  20171002t072939 0.00      2017-10-02T07:29:46-04:00  SERVING
default  20171002t074125 1.00      2017-10-02T07:41:31-04:00  SERVING

$ curl https://your-project-id-here.appspot.com
Hello from version 3 which is not live yet! ①
```

① Obviously it's live now!

Now that we've seen how deployment works on App Engine Standard Environment, let's take a bit of a detour and look at how things work in the Flexible Environment.

## 11.2.2 On App Engine Flex

As we discussed previously, whereas App Engine Standard is limited to some of the popular programming languages and runs inside a sandbox environment, App Engine Flex is based on Docker containers, which means that you can use any programming language you want. This means that we get to switch back to Node.js when building our "Hello, world!" application. Let's get started!

### CREATING AN APPLICATION

Similarly to the example we used when building an application for App Engine Standard, let's start by building another "Hello, world!" application using Express (a popular web-development framework for Node.js).

**NOTE**

You can use any web framework you want. Express just happens to be very popular and well documented, so we'll use that for our example.

First, let's create a new directory to hold the code for this new application called `default-flex` alongside the other directories we have already. After that, we should initialize the application using `npm` (or `yarn`) and add `express` as a dependency, shown below.

**Listing 11.24. Initialize the package for our Node.js application with npm**

```
$ mkdir default-flex
$ cd default-flex
$ npm init
# ...
Wrote to /home/jjg/projects/appenginehello/default-flex/package.json:

{
  "name": "appengineflexhello",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}

$ npm install express
# ...
```

Now that we've defined our package and set the dependencies we need, let's write a simple script that uses Express to handle HTTP requests. This script, which we'll put inside `app.js`, will take any request sent to `/` and send "Hello, world!" as a response.

**Listing 11.25. Defining a simple "Hello, world!" application in Node.js**

```
'use strict';

const express = require('express'); ①
const app = express();

app.get('/', (req, res) => {
  res.status(200).send('Hello from App Engine Flex!').end(); ②
});

const PORT = process.env.PORT || 8080; ③
app.listen(PORT, () => {
  console.log(`App listening on port ${PORT}`);
  console.log('Press Ctrl+C to quit.');
});
```

① For this example, we'll use the popular Express framework for Node.js.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/google-cloud-platform-in-action>

Licensed to Asif Qamar <[asif@asifqamar.com](mailto:asif@asifqamar.com)>

- ② This directive simply sets the response code to 200 OK and returns a short "Hello World" message.
- ③ We'll try to read a port number from the environment, but default to 8080 if it's not set.

Once we done this, we can test that our code here works by running it and then trying to connect to it using `curl` or our browser, shown here.

#### **Listing 11.26. Verifying that our "Hello, world!" application works as expected**

```
$ node app.js
App listening on port 8080
Press Ctrl+C to quit.

$ curl http://localhost:8080 ①
Hello from App Engine Flex!
```

- ① Obviously this is executed from a separate terminal on the same machine.

Now that we're sure the code works, we can get back to work on deploying it to App Engine. Just like before, we'll need to also define a bit of configuration that explains to App Engine how to run our code. Just like with App Engine standard, we'll put the configuration options in a file called `app.yaml`. The main difference here is that since a Flex-based application is based on Docker, the configuration we put in `app.yaml` is far less involved.

#### **Listing 11.27. Defining configuration in app.yaml for App Engine Flex**

```
runtime: nodejs
env: flex ①
service: default ②
```

- ① Here we see a new parameter called `env` which is used to explain to App Engine that we intend to run our application outside of the standard environment.
- ② For this example we'll deploy the Flex service on top of the Standard service.

As you can see this configuration file is far simpler than the one we used when building an application to App Engine Standard. This is because App Engine Flex just needs to know how to take your code and put it into a container. Next, instead of setting up routing information, we need to tell App Engine how to start our server. It turns out that App Engine Flex's `nodejs` runtime will always try to run `npm start` as the "initialization command", so we can set up a hook for this in our `package.json` file that executes `node app.js`.

**NOTE**

This format we're demonstrating is a feature of `npm` rather than App Engine. All App Engine does is call `npm start` when it turns on the container.

**Listing 11.28. Add a start script to package.json**

```
{
  "name": "appengineflexhello",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "dependencies": {
    "express": "^4.16.1"
  },
  "scripts": {
    "start": "node app.js"
  }
}
```

That should be all we need. The next step here is to actually deploy this application to App Engine.

**DEPLOYING TO APP ENGINE FLEX**

As you might guess, deploying an application to App Engine Flex is very similar to App Engine Standard. The main difference you'll notice at first is that it takes a bit longer to complete the deployment. This is primarily because as part of the deployment process App Engine actually builds a Docker container from your application code, uploads it to Google Cloud, provisions a Compute Engine VM instance, and starts the container on that instances. Since this is quite a bit more to do than using App Engine Standard the deployment process for Flex takes a bit longer, but the process itself from our perspective is basically the same and can be done with the `gcloud` command-line tool.

**TIP**

If you followed along with all of the code examples in the previous section, you might want to undo the change we made to the `promote_by_default` configuration setting by running `gcloud config set app/promote_by_default true`. Otherwise, when you attempt to visit your newly deployed application, it will still be served by the previous version we deployed.

**Listing 11.29. Deploying our application to App Engine Flex**

```
$ gcloud app deploy default-flex
Services to deploy:

descriptor:      [/home/jjg/projects/appenginehello/default-flex/app.yaml]
source:          [/home/jjg/projects/appenginehello/default-flex]
target project:  [your-project-id-here]
target service:  [default]
target version:  [20171002t104910]
target url:      [https://your-project-id-here.appspot.com]

(add --promote if you also want to make this service available from
[https://your-project-id-here.appspot.com])

# ... More information here ...
```

Now that our application has been deployed we can test that it works in the same way that we did with App Engine Standard.

#### **Listing 11.30. Verifying that the deployment worked as expected**

```
$ curl https://your-project-id-here.appspot.com/
Hello from App Engine Flex!
```

What might surprise you at this point is that we actually deployed a new version of the "default" service, which uses a completely different runtime. This means that we actually have two versions running right now, one using the Standard environment and the other using the Flexible environment! We can see this by listing the versions of the default service again.

#### **Listing 11.31. Listing the various versions**

SERVICE	VERSION	TRAFFIC_SPLIT	LAST_DEPLOYED	SERVING_STATUS
default	20171001t160741	0.00	2017-10-01T16:08:18-04:00	SERVING
default	20171002t072939	0.00	2017-10-02T07:29:46-04:00	SERVING
default	20171002t074125	0.00	2017-10-02T07:41:31-04:00	SERVING ①
default	20171002t104910	1.00	2017-10-02T10:51:00-04:00	SERVING ②

- ① This was the previously live default version on App Engine Standard.
- ② Our new version runs on App Engine Flex alongside the other versions.

This means that we can access our previous service from App Engine Standard by addressing it directly as we did before.

#### **Listing 11.32. Verifying that our previous App Engine Standard application still works**

```
$ curl http://20171002t074125.default.your-project-id-here.appspot.com/
Hello from version 3 which is not live yet!
```

As you might expect, this means that deploying other services and versions is identical to how we just learned using App Engine Standard.

To demonstrate this, let's deploy one last new service named `service3`, putting all of our code side-by-side. To start, we'll just copy and paste the code for `default-flex` into `service3`.

#### **Listing 11.33. Directory tree for our new code layout.**

```
$ tree -L 2 .
.
└── default
    ├── app.yaml
    └── main.py
└── default-flex
    └── app.js
```

```

    └── app.yaml
    └── node_modules
    └── package.json
  └── service2
    └── app.yaml
    └── main.py
  └── service3
    └── app.js
    └── app.yaml
    └── node_modules
    └── package.json

```

At this point, we'll have to make two changes. The first is to the `app.yaml` file where we change the `service` name to "service3". The second is to update our `app.js` code so that it says "Hello from service 3!" The updated contents for both files is shown below.

#### **Listing 11.34. Updated app.yaml for the new service**

```

runtime: nodejs
env: flex
service: service3 ①

```

- ① We want this to deploy as a new service, so we call it "service3".

#### **Listing 11.35. Updated app.js for the new service**

```

'use strict';

const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.status(200).send('Hello from service 3!').end(); ①
});

const PORT = process.env.PORT || 8080;
app.listen(PORT, () => {
  console.log(`App listening on port ${PORT}`);
  console.log('Press Ctrl+C to quit.');
});

```

- ① Here we update the response in our application to state that it's coming from the new service.

Once we've done that, we can move into the parent directory and deploy the new service with `gcloud app deploy service3` (since we named the directory `service3`).

#### **Listing 11.36. Deploying the new service to App Engine**

```

gcloud app deploy service3
Services to deploy:

descriptor:      [/home/jjg/projects/appenginehello/service3/app.yaml]
source:          [/home/jjg/projects/appenginehello/service3]
target project:  [your-project-id-here]

```

```
target service: [service3]
target version: [20171003t062949]
target url: [https://service3-dot-your-project-id-here.appspot.com]

# ... More information here ...
```

And now we can test that everything works by using `curl` or your browser to talk to all of the various services we've deployed.

#### **Listing 11.37. Verifying that all services are available**

```
$ curl http://service3.your-app-id-here.appspot.com
Hello from service 3!

$ curl http://service2.your-app-id-here.appspot.com
Hello from service 2!

$ curl http://your-app-id-here.appspot.com
Hello from App Engine Flex!
```

Now that we've deployed multiple services on App Engine Flex, let's try digging even deeper into deploying services with custom runtime environments.

#### **DEPLOYING CUSTOM IMAGES**

So far we've always relied on the built-in run-times (in this case, `nodejs`), but since App Engine Flex is based on Docker containers (which we discussed in [chapter 10](#)), this means that technically we can use any container we want! To see how this works, let's try building an entirely different type of "Hello, world!" application relying on a typical Apache web server. To do this, the first thing we should do is create another directory named after our service. In this case, let's call it `custom1` and let's put it right next to the other directories for the other services in our application.

Once we've created the directory, we'll need to define a Dockerfile that defines our application. Since we're just trying to demonstrate how this works, we'll keep it very simple and stick with using Apache to serve a static file.

#### **NOTE**

**Don't worry if you don't understand all of this completely. If you're interested in using custom runtime environments like this, you should probably read up on Dockerfile syntax, but it's not really a requirement to use App Engine.**

#### **Listing 11.38. A Dockerfile to run our application**

```
FROM ubuntu:16.04                                1

RUN apt-get update && apt-get install -y apache2    2

# Set Apache to listen on port 8080 instead of 80 (what App Engine expects) 3
RUN sed -i 's/Listen 80/Listen 8080/' /etc/apache2/ports.conf
RUN sed -i 's/:80/:8080/' /etc/apache2/sites-enabled/000-default.conf

# Add our content                                4
```

```
COPY hello.html /var/www/html/index.html
RUN chmod a+r /var/www/html/index.html

EXPOSE 8080 5

CMD ["apachectl", "-D", "FOREGROUND"] 6
```

- ➊ This says that the base image (and operating system) will be a plain version of Ubuntu 16.04.
- ➋ Here we update packages and install Apache.
- ➌ These replacements simply update Apache to listen on port 8080 instead of 80 (since App Engine expects HTTP traffic on the container to be on 8080).
- ➍ Here we copy the `hello.html` file (we'll write that in a bit) over the the Apache static content directory (and make sure it's readable by the world).
- ➎ This tells Docker to expose port 8080 to the outside.
- ➏ Finally we start the Apache service in the foreground.

Now that we have a simple Dockerfile that serves some content, the next thing we'll need to do is update our `app.yaml` file to rely on this custom runtime. To do that, we'll just replace `nodejs` in our previous definition to `custom`. This tells App Engine to look for a Dockerfile and use that instead of the built-in `nodejs` Dockerfile that we used previously.

#### **Listing 11.39. The updated app.yaml file**

```
runtime: custom
env: flex
service: custom1
```

The last thing we'll need to do is define the static file we want to serve, which is pretty easy. Just write some simple HTML that says "Hello from Apache!".

#### **Listing 11.40. A simple "Hello, world" HTML file**

```
<html>
  <body>
    <h1>Hello from Apache!</h1>
  </body>
</html>
```

At this point our directory should look something like the following. Notice that our new service has no actual code in it and is made up entirely by the Dockerfile, some static content, and the App Engine configuration. While this is valid, it's unlikely that a real App Engine application would look be this simple.

#### **Listing 11.41. Directory structure for our services**

```
$ tree . -L 2
.
└── custom1 ❶
    └── app.yaml
```

```

    └── Dockerfile
        └── hello.html
    └── default
        └── app.yaml
        └── main.py
    └── default-flex
        └── app.js
        └── app.yaml
        └── node_modules
        └── package.json
    └── service2
        └── app.yaml
        └── main.py
    └── service3
        └── app.js
        └── app.yaml
        └── node_modules
        └── package.json

```

- ➊ Notice that we're treating this custom runtime environment just like any of our other services.

We can test that our code works using Docker. First, we build the container image using `docker build custom1`, and then we'll start the container and verify that Apache is doing what we expect.

#### **Listing 11.42. Building and testing the new Docker container.**

```

$ docker build custom1          ①
Sending build context to Docker daemon 4.608 kB
Step 1 : FROM ubuntu:16.04
16.04: Pulling from library/ubuntu

# ... Lots of work here ...

Successfully built 431cf4c10b5b

$ docker run -d -p 8080:8080 431c ②
e149d89e7f619f0368b0e205d9a06b6d773d43d4b74b61063d251e0df3d49f66

$ docker ps                      ③
CONTAINER ID   IMAGE      COMMAND           CREATED          STATUS          PORTS          NAMES
e149d89e7f61   431c      "apachectl -D FOREGRO"   17 minutes ago   Up 17 minutes   0.0.0.0:8080->8080/tcp   jovial_sinoussi

$ curl localhost:8080            ④
<html>
  <body>
    <h1>Hello from Apache!</h1>
  </body>
</html>

```

- ➊ First we build the container which will package everything up into a single Docker image.  
➋ Next we run the container using the ID of the image, being sure to link our local machine's port 8080 to the container's port 8080.  
➌ We can verify that the container is running using `docker ps`.

- ④ Finally, connecting to it over HTTP on port 8080 shows that it serves the correct HTML content that we wrote above.

Now that we've checked our application actually works we can deploy it to App Engine using the same `deploy` command as we have before.

#### **Listing 11.43. Deploying the custom runtime container to App Engine**

```
$ gcloud app deploy custom1
Services to deploy:

descriptor:      [/home/jjg/projects/appenginehello/custom1/app.yaml]
source:          [/home/jjg/projects/appenginehello/custom1]
target project:  [your-project-id-here]
target service:  [custom1]
target version:  [20171003t123015]
target url:      [https://custom1-dot-your-project-id-here.appspot.com]

# ... More information here ...
```

At this point you can verify that the deployment worked by making the same request again to `custom1.your-project-id-here.appspot.com` and seeing that the result is the HTML file we wrote previously.

The moral of the story here is that anything that fits into a Docker container can be run and scaled on App Engine Flex. This means that basically anything you can run on your local machine can also run on App Engine just like we saw with Compute Engine and Kubernetes Engine.

**WARNING**

If you deployed an application using App Engine Flex, this means there are compute resources running under the hood regardless of whether anyone is sending requests to your application.

If you don't want to be billed for these resources, make sure to "stop" any running Flex versions. You can do this in the App Engine section of the Cloud Console by choosing "Versions" in the left-side navigation, checking the boxes for running versions, and clicking the "Stop" button at the top of the page.

Now that we've seen all the many different ways that we can build applications for App Engine, there's one topic that we've sort of glossed over: scaling. That is, how exactly do we control how many instances are running at a given time. Let's take a moment to run through that on both App Engine Standard and App Engine Flex.

### **11.3 Scaling your application**

So far, we've sort of assumed that all the scaling on App Engine was taken care of for us. And while that is mostly true there are still lots of ways you can configure both the scaling style and the underlying instances that run your code.

As we discussed previously, there are a few differences between App Engine Standard and App Engine Flex when it comes to scaling and instance configuration, so we'll have to look at each of these individually. Keep in mind that you can only choose a

single type of scaling for any given service, so it's important to understand how they all work and to choose the one that best suits what your application needs. Let's look in more detail about how you can control how scaling works on both of the App Engine environments.

### 11.3.1 Scaling on App Engine Standard

So far, all of our services have been deployed without any mention about how to scale them, which in essence means we rely on "the default options" for scaling our application. It turns out, however, that App Engine Standard has quite a few different scaling options that we can fine tune so that they fit our needs. Let's start by looking at the default option that also happens to be one of the main features of App Engine: automatic scaling.

#### AUTOMATIC SCALING

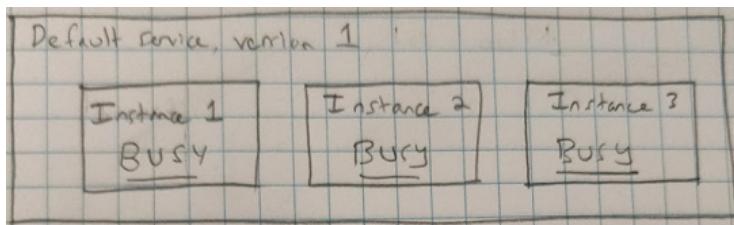
The default scaling option is "automatic" which means that App Engine will decide when to turn on new instances for you based on a few metrics such as the number of concurrent requests, the longest a given request should wait around in a queue to be handled, and the number of instances that can be idle at any given time. Even though these all have default settings, we can change them in `app.yaml`. Let's start by looking at the simplest of the settings: idle instances.

#### *Idle instances*

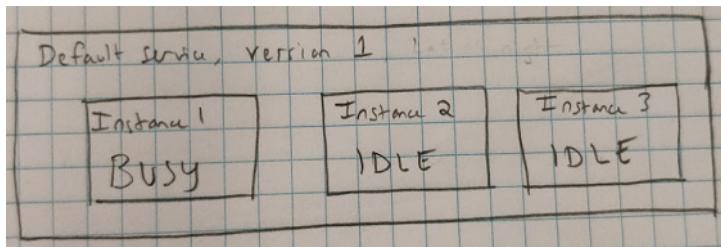
Since App Engine Standard instances are really just "chunks of CPU and memory" rather than a full virtual machine (and each of these instances costs money, which we'll look at later), App Engine Standard provides a way for you to decide the minimum and maximum number of instances that can sit idle waiting for requests before being "turned off".

In a way, this setting is a bit like choosing how many "buffer instances" to keep around that aren't actively in use. For example, let's imagine that we deploy a service that gets enough traffic to keep 3 instances busy, followed by a period where only 1 instance is busy.

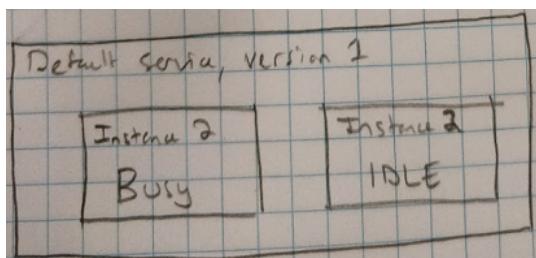
**Figure 11.11. At first, all three instances are busy**



If we set the minimum idle instances to 2 (and maximum to 3) then we'll end up with 1 busy instance and 2 idle instances that sit around doing nothing except waiting for more requests.

**Figure 11.12. The two idle instances will be kept around**

On the other hand, if we set the minimum and maximum idle instances both to 1, then App Engine will turn off instances until there is exactly one sitting idle waiting for requests. In this case, this would mean that we'd have 1 busy instance and 1 idle.

**Figure 11.13. One idle instance is kept around, the other is terminated**

In this example scenario, we'd update our `app.yaml` file with a category called `automatic_scaling`, and fill in the `min_idle_instances` and `max_idle_instances` settings. Below you can see what the first configuration we described above might look like in real life.

#### **Listing 11.44. Updated app.yaml with scaling based on idle instances**

```
runtime: python27
api_version: 1
service: default
automatic_scaling:
  min_idle_instances: 2
  max_idle_instances: 3
```

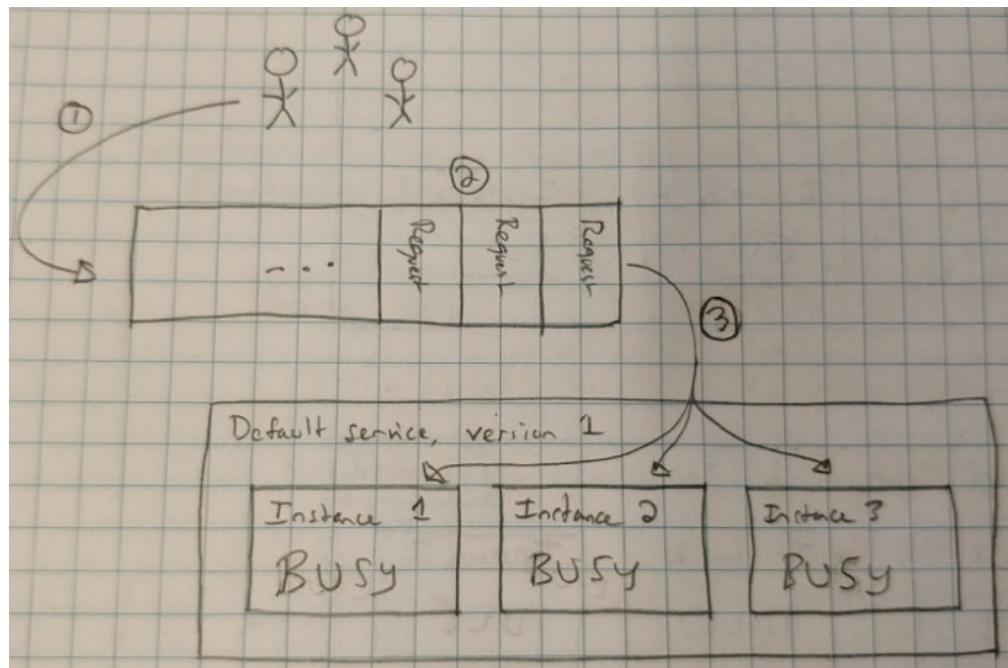
Next, let's look at pending latency.

#### **Pending latency**

When you make a request to App Engine, the request is handled by Google's front-end servers and ultimately routed to a specific instance running your service. It turns out, however, that App Engine actually keeps a queue of requests in case some of them aren't ready to be handled yet. In other words, if there's no instance available to respond to a given request immediately as it arrives, it can sit in a queue for a bit until an instance becomes available. Shown below is an example of how we might think of

the flow of a request through your App Engine service.

**Figure 11.14. Requests queue up before being routed to an instance**



The flow begins with (1) lots of people making requests to the service. Requests are immediately queued up to be processed (2) in a standard work queue-style service, and ultimately handled by an individual instance (3) to do the work required by the request. Hidden in this flow is an important metric to keep in mind: how long a request sits in that queue. Since App Engine has the ability to turn on more instances, if a request is sitting in a queue for too long it seems like a good idea to do that since it will help get through the queue of work more quickly. It turns out that App Engine let's us choose the minimum and/or maximum amount of time that any given request should spend sitting in this queue, called "pending latency".

The maximum pending latency means that if requests are spending more than this amount of time in the queue, we should turn on more instances. For example, we might set this to 10 seconds and App Engine will keep an eye on this metric, turning on new instances whenever the typical request spends more than 10 seconds in the queue. In general, a lower maximum pending latency means that App Engine will start instances more frequently. The minimum pending latency is the way we set a lower bound when telling App Engine when it's OK to turn on more instances. When you set a minimum pending latency it tells App Engine never to turn on a new instance if requests aren't waiting **at least** a certain amount of time. For example, we might set this to 5 seconds to ensure that we don't turn on new instances to handle requests that are just waiting a few seconds.

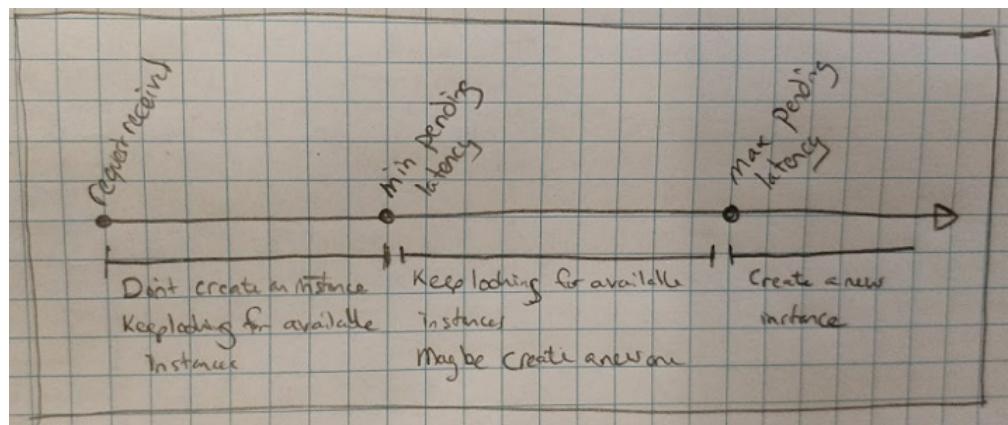
In a sense, these settings are a bit like setting how stretchy a spring is. Lower values for both minimum and maximum mean the spring is super stretchy (App Engine will expand capacity quickly to handle requests) and higher values mean that the spring is much more stiff (App Engine will tolerate requests sitting in the queue for a while). In the example we described above, with a minimum of 5 seconds and a maximum of 10 seconds of pending latency in the queue, the configuration would look the following.

#### **Listing 11.45. Updated app.yaml with scaling defined based on pending latency**

```
runtime: python27
api_version: 1
service: default
automatic_scaling:
  min_pending_latency: 5s
  max_pending_latency: 10s
```

It can sometimes be confusing how these two settings interact, so the following diagram shows the different cut-off points. As you can see, up until the minimum pending latency mark, App Engine will keep looking for an available instance, and will never create a new one due to this request. After that minimum pending latency point has passed, App Engine will keep looking for an available instance, but considers itself free to create a new one if it makes sense. If the request is still sitting in the queue by the maximum latency time, App Engine (if allowed by other parameters) will create a new instance to handle the request.

**Figure 11.15. Time line of what actions are possible based on minimum and maximum pending latency**



In other words, spending more than the maximum amount of time in the queue will trigger that more instances be turned on, acting sort of as a gas pedal that spurs more scaling. On the other hand, the minimum latency time acts more like a brake on scaling that prevents App Engine from turning on instances before they're really needed. Let's now move on to the final metric we can control as part of automatic scaling has to do with the concurrency level of a given instance.

### **Concurrent requests**

Since instances can handle more than one request at a time, the number of requests happening at once ("level of concurrency") happens to be another metric to use when auto-scaling. App Engine allows you to set a concurrency level as a way to trigger that more instances should be turned on or off, meaning you can set a target of how many requests an instance can handle at the same time before it's considered "busy. Obviously a higher value here will try to send more requests to a single instance which could overload the instance, but a super low value here will leave the instances under utilized.

By default, App Engine will aim to handle 8 requests concurrently on your instances, and while you can crank this all the way up to 80, it's worth testing and monitoring your instances to tune this number. Just like the other settings, changing the concurrent request parameter is done with a setting inside your `app.yaml` file.

#### **Listing 11.46. Updated app.yaml file with scaling based on concurrent requests**

```
runtime: python27
api_version: 1
service: default
automatic_scaling:
  max_concurrent_requests: 10
```

Now that we've covered the most common scaling option, let's quickly look through the other simpler scaling configurations.

### **BASIC SCALING**

Basic scaling is another option provided by App Engine Standard which is sort of like a slimmed down version of automatic scaling. Basic scaling has only two options to control which are the maximum number of instances and how long to keep an idle instance around before turning it off. As you might guess, the maximum instances is a limit on the number of instances running at any given time, which is helpful if you're worried about being surprised by a large bill at the end of the month.

The next setting has to do with the "termination policy" for the instances handling requests to your service. As we learned before, it's likely that as traffic to your application fluctuates instances that were created to handle spikes of traffic might go idle during a lull. When you use basic scaling you're able to specify how long those instances should sit idle before being turned off. By default, instances will sit idle for no longer than 5 minutes, but you're free to make that longer or shorter depending on your needs. For example, to set basic scaling with no more than 10 instances, and a maximum idle time of 3 minutes, we can update our `app.yaml` file to look like the following.

#### **Listing 11.47. Updated app.yaml file with basic scaling configured**

```
runtime: python27
api_version: 1
```

```
service: default
basic_scaling:
  max_instances: 10
  idle_timeout: 3m
```

As you can see, "basic scaling" really does mean basic as there are very few options for this type of scaling and none of them are particularly complicated. Let's look at an even **more** simple form of scaling: manual.

#### **MANUAL SCALING**

Manual scaling is a bit of a misnomer as it is almost like no scaling at all. In this configuration you specifically tell App Engine exactly how many instances you want running at a given time. When you do this, all requests are routed to this pool of machines which may become overwhelmed to the point where requests time out. As a result, this type of scaling should be for things where you have a strict budget, but don't care all that much that your application is "always available" to your customers.

If you've decided that you want manual scaling, you simply choose the number of instances you want and update your `app.yaml` file. The example below shows what it would look like if you wanted to always have exactly 10 instances running at all times for your service.

#### **Listing 11.48. Updated app.yaml file showing manual scaling**

```
runtime: python27
api_version: 1
service: default
manual_scaling:
  instances: 10
```

Now that we've covered how to scale your App Engine Standard services, let's look at how scaling works when using App Engine Flexible Environment.

### **11.3.2 Scaling on App Engine Flex**

Since App Engine Flex is based on Docker containers and Compute Engine VMs, the scaling configurations should feel very familiar to the way that we looked at auto-scaling Compute Engine instances with instance groups and instance templates. Similar to App Engine Standard, Flex has two scaling options: automatic and manual. The only difference is that Flex is lacking the "basic" scaling option. Let's start by looking at the more common scaling method: automatic.

#### **AUTOMATIC SCALING**

App Engine Flex is capable of scaling your services up and down just like we did previously when we looked at automatic scaling of Compute Engine instances, and it turns out that the parameters you can configure for App Engine Flex services are pretty similar to Compute Engine's instance groups. Since all of the options are pretty straight forward, we'll run through them quickly and then show a demonstration of how they work together.

First we can control the number of VM instances that can be running at any given time. As we learned, there must be at least 1 instance running at all times but it's recommended to have a minimum of 2 instances to keep latency low in the face of traffic spikes (which happens to be the default). You can also set a maximum number of instances to avoid your application scaling out of control. By default Flex services are limited to 20 instances, but this can be increased or decreased.

Next we can control how App Engine decides whether additional instances are needed, which is done by looking at the CPU utilization and comparing it to a target. If the CPU usage is above the target we'll get more instances and if it's below the target, some instances will be turned off. By default App Engine Flex services will aim for a 50% utilization (0.5) across the fleet.

Aiming for a target is great, but turning instances on and off isn't an immediate action, which could cause some problems. In other words, turning on a new instance might take a few seconds, so turning off an instance immediately after the utilization is low might not make a lot of sense. Luckily, App Engine Flex has a way to control how aggressively to terminate instances when the overall utilization comes in below the target amount called the "cool down period". This setting controls how long to hold off after the utilization drops before terminating instances (by default, 2 minutes). As you'd expect, a higher value here means you'll typically have excess capacity, while a lower value may lead to periods where requests queue up waiting for available capacity.

Now that we've gone through all of the automatic scaling settings for App Engine Flex, let's look at a sample configuration where we want to have somewhere between 3 and 8 instances with a CPU utilization target of 70% and a 5 minute cool down period.

#### **Listing 11.49. Updated app.yaml showing a configuration of automatic scaling for Flex**

```
runtime: nodejs
env: flex
service: default
automatic_scaling:
  min_num_instances: 3
  max_num_instances: 8
  cool_down_period_sec: 300 # 5 minutes * 60 = 300
  cpu_utilization:
    target_utilization: 0.7
```

This brings us to the next option for scaling, which is quite similar to App Engine Standard.

#### **MANUAL SCALING**

Just like App Engine Standard's manual scaling options, App Engine Flex has an option to decide up-front exactly how many VM instances to run for your service. And it turns out that the syntax is identical, with an example of 4 VM instances shown below.

**Listing 11.50. Updated app.yaml file showing manual scaling**

```
runtime: nodejs
env: flex
service: default
manual_scaling:
  instances: 4
```

Now that we've covered all of the different scaling options, it's time to look at what exactly we're scaling.

### **11.3.3 Choosing instance configurations**

So far we've talked about the number of instances and how to scale them, and we've also thought of instances as chunks of CPU and memory (either sandboxes or virtual machines), but we haven't looked at the details of these instances themselves. Let's explore what these instances actually are and how to choose instance configurations that suit your application, starting with App Engine Standard.

#### **APP ENGINE STANDARD INSTANCE CLASSES**

Since App Engine Standard involves running your code in a special sandbox environment, we'll need a way of configuring the computing power of that environment. To do this, we'll use a setting called `instance_class` in our `app.yaml` file. You can view the full list of instance class options in the App Engine documentation, but a few common options are listed below.

**Table 11.1. Resources for various App Engine Instance classes**

Name	Memory	CPU
F1	128 MB	600 MHz
F2	256 MB	1.2 GHz
F4	512 MB	2.4 GHz
F4_1G	1024 MB	2.4 GHz

By default, automatically scaled services use F1 instances, so if we wanted to increase the instance class from the default F1 up to the F2 type, we could update our configuration, shown below.

**Listing 11.51. Updated app.yaml file configuring a different instance class**

```
runtime: python27
api_version: 1
service: default
instance_class: F2 ①
```

① Here we change the instance class to be our desired F2 type.

In general, the best way to choose an instance type is using experimentation (similar to how you'd choose the scaling parameters such as minimum / maximum pending

latency). After changing instance classes you can look at performance characteristics using benchmarking tools to see what fits best.

Don't forget to adjust your concurrent requests scaling parameter at the same time as changing the instance class. Typically larger classes can handle more concurrent requests (and vice-versa for small classes). This would mean that when you make the change above to use F2 instances, you might also want to double the limit of concurrent requests for your service, show below.

#### **Listing 11.52. Adjusting instance class and concurrent request limits together**

```
runtime: python27
api_version: 1
service: default
instance_class: F2 ①
automatic_scaling:
  max_concurrent_requests: 16 ②
```

- ① First we change the instance class to be our desired F2 type as before.
- ② Then to ensure we're making good use of the newly added resources, we double the default limit of concurrent requests per instance from 8 to 16.

Another general rule of thumb about choosing an instance class is to keep in mind that a more resources typically doesn't reduce the overall latency of requests (due to the typical pattern involving lots of I/O), but instead allows a single instance to handle more of them at the same time. This means that if you're hoping to make a single request faster, instance class isn't guaranteed to fix that for you.

Now let's switch gears and dig into how App Engine Flex let's you define instances.

#### **APP ENGINE FLEX INSTANCES**

Since App Engine Flex is based on Docker containers and Compute Engine instances we actually get quite a bit more freedom when choosing virtual hardware. Although Compute Engine has specific instance types with the ability to customize those to suit your projects, App Engine Flex sticks to the idea of declaring the resources you need and allowing App Engine itself to provision machines that match those needs.

In other words, instead of saying "I want this machine type" we instead say "I need at least 2 CPUs and at least 4 GB of RAM". App Engine takes that and provisions a VM for your service that has **at least** those resources (it may have more than that). If we wanted to configure our service in the way we just described (2 CPUs, 4 GB of RAM), we'd simply update our `app.yaml` file to express this using a `resources` heading.

#### **Listing 11.53 Updated app.yaml file configuring the Compute Engine instance memory and CPU**

```
runtime: nodejs
env: flex
service: default
```

```
resources:
  cpu: 2 ①
  memory_gb: 4.0
```

- ① As you can see, we just set the desired configuration.

By default (that is, if you leave these fields out entirely) you'll get a single core VM with 0.6 GB of RAM which should be enough for relatively simple web applications, but should you find that your service is handling lots of memory intensive work or computational work that can be easily parallelized and split across more cores, adding more memory or more CPU is likely a good idea.

Just like with Compute Engine, memory and CPU are related and limited so that they don't stray too far from one another. For these instances, RAM in GB can be anywhere from 90% to 650% of the number of CPUs, but there is some overhead memory (about 0.4 GB) that is used by App Engine on your instance. This means that for the 2 CPUs requested above, our VMs are limited to anywhere from 1.8 GB to 13 GB of RAM, so we can only request between 1.4 GB ( $1.8 - 0.4$ ) and 12.6 GB ( $13 - 0.4$ ) in this configuration.

In addition to setting the CPU and memory targets, we can also choose the size of our boot disk and attach other temporary file system disks. (If your Docker image is larger than the default limit of 10 GB, you'll need to increase this size to fit your image.)

**WARNING**

Even though App Engine Flex instances have a boot disk, this disk should be considered **temporary** as it will disappear anytime an instance is turned down.

Since different disk sizes have different performance characteristics, it might make sense to increase the size of your boot disk if your Docker image has lots of local data that you want to load up quickly. Below you can see how we might increase the size of the boot disk to 20 GB from the default of 10 GB.

**Listing 11.54. Updating app.yaml to increase the size of instance boot disks**

```
runtime: nodejs
env: flex
service: default
resources:
  disk_size_gb: 20 ①
```

- ① In this situation, we double the boot disk size to 20 GB which will not only store more data, but will also provide higher performance.

At this point, we've explored in depth the computing environment provided by App Engine (both Standard and Flexible environments), and all of the infrastructural considerations to keep in mind when building applications on App Engine. What we haven't done is looked in detail at how you might actually write your services to make use of all the hosted services on App Engine. Let's spend some time exploring a few of App Engine's managed services and how we might use them to build out an

application.

## 11.4 Using App Engine Standard's managed services

If you were building an application that stores data, you'd need to build the application itself and then make sure you have a database server running as well that can hold the persistent data. App Engine aims to help make building applications easier by providing services (like storing data) that "just work" so that you don't have to worry about the surrounding infrastructure.

As you might guess, there are a lot of different services and lots of ways to use them. If you're interested in digging into the details of each and every service, you may want to explore a book on Google App Engine to supplement this chapter. For now we'll focus on just a few of the important services, covering briefly how you can use them. We'll also be limiting the discussion here to App Engine Standard since App Engine Flex is really just Compute Engine VMs. Let's get started by looking at the most common thing an application needs to do: store data.

### 11.4.1 Storing data with Cloud Datastore

As you learned in [part 2](#), there are many different ways to go about storing data, and Google Cloud Platform has many different services available to help you with this. This is good news because it turns out that you can access these services from inside App Engine. Instead of learning how each of the different storage systems work (since there are whole chapters about each), we'll focus instead on how you might connect to the different services from inside your App Engine application.

As we learned in [chapter 5](#), Cloud Datastore is a non-relational storage system that stores documents and provides ways to query those documents. To make life easy, Cloud Datastore comes pre-baked into App Engine with APIs directly built into the runtime. In the case of Python, App Engine Standard provides a Datastore API package called `ndb` which acts as an ORM (object-relational mapping). We won't even scratch the surface of what `ndb` is capable of doing, but the following code shows how you might define a `TodoList` model and interact with entities in Datastore. It starts by defining a "model" which is the type of an entity, then creating a new to-do list, querying the available lists, and then deleting the list we created.

#### **Listing 11.55. Example interaction with Datastore from App Engine Standard's `ndb` library**

```
from google.appengine.ext import ndb
1

class TodoList(ndb.Model):
    name = ndb.StringProperty()
2
    completed = ndb.BooleanProperty()
3

# Create a new TodoList
my_list = TodoList(name='Groceries', completed=False)
4
key = my_list.put()
```

```
# Find TodoLists by name
lists = TodoList.query(name='Groceries')      5

# Delete the TodoList by ID
my_list.delete()                            6
```

- ➊ First we import the `ndb` library to use it (similar to `require` in Node.js).
- ➋ Next we define the model itself which is a bit like setting up a table in a typical relational database (defining the name of the entity type and the fields that we intend to set on entities).
- ➌ `ndb` allows you to set many different property types such as strings, lists, booleans, and more.
- ➍ To create a new entity, we simply create an instance of a model and use the `put()` method to persist it to Datastore.
- ➎ We can query Datastore for matching entities using the `query()` method.
- ➏ Finally, we can delete the entity by calling `.delete()` on it.

In the code above there are a couple of interesting things that are worth mentioning. First, we didn't talk about authentication at all. This is because authentication happens automatically due to the fact that your code is running inside a managed sandbox environment. As a result, you don't have to set which URL to send API requests or specify which project you're interacting with, or provide any private keys to gain access. By virtue of running inside App Engine you are guaranteed secure and easy access to your instance of Cloud Datastore. Next, we didn't have to define any special dependencies in order to use the `ndb` package in our application. This is because the sandbox environment that your code runs in is automatically provided with the code needed to access `ndb`.

If you're interested in using Cloud Datastore from inside App Engine Standard, you should definitely read more about the various libraries available in the language you intend to work in. App Engine has different libraries for Java, Python, and Go, each of which has a different API to interact with your data in Datastore. Let's move on and look at how we might cache data temporarily using Memcache.

### **11.4.2 Caching ephemeral data**

In addition to storing data permanently, it's pretty common that applications will want to store temporary data as well. For example, a query might be particularly complex and put quite a bit of strain on the database, or a calculation might take a while compute and you might want to keep it around rather than do the computation again. For these types of problems a cache is typically a great answer, and it turns out that App Engine Standard provides a hosted Memcache service that you can use with no extra setup at all.

**NOTE**

If you're not familiar with Memcache, it's pretty simple. Memcache offers an incredibly simple way to store data temporarily, always using a unique key. In a way, think of it like a big shared Node.js JSON object store that you manipulate by calling `value = get(key)`, `set(key, value)`, etc.

App Engine's Memcache service acts like a true Memcache service which means that the API you use to communicate with it should feel very familiar if you've ever used Memcache yourself. Shown below is some code that writes, reads, and then deletes a key from App Engine's Memcache service.

**Listing 11.56. Example interaction with App Engine Standard's Memcache service**

```
from google.appengine.api import memcache ①
memcache.set('my-key', 'my-value')          ②
memcache.get('my-key')                     ③
memcache.delete('my-key')                  ④
```

- ① We start by importing the App Engine `memcache` library.
- ② Setting keys is done using the `set(key, value)` method.
- ③ We can retrieve keys using `get(key)`.
- ④ Finally we can remove the key by using `delete(key)`.

Even though the API to talk to App Engine's Memcache service is the same as a regular Memcache instance running on a VM, it isn't a true Memcache binary running in that same way and is instead a large shared service that acts like Memcache. As a result, there are a few things to keep in mind.

First, your Memcache instance will be limited to about 10,000 operations per second. This means that if your application gets a lot of traffic, you may need to think about using your own Memcache cluster of VMs inside Compute Engine. Additionally, you may find that certain keys in Memcache receive more traffic than others. For example, if you have a single key that you use as a way to count the number of visitors to your site, this makes it difficult for App Engine to distribute that work which will result in degraded performance.

**TIP**

For more information on distributing access to keys, take a look at [chapter 7](#) which addresses this problem head on.

Next, we have to address the various limits. The largest key you can use to store your data is 250 bytes, and the largest value you can store is 1 MB. If you try to store more than this, the service will reject the request. Additionally, since Memcache supports "batch" or "bulk" operations where you set multiple keys at once, the most data you can send in one of those requests (that is, size of the keys combined with the size of the values) is 32 MB.

Finally, we must consider the shared nature (by default) of the Memcache service and how that affects the lifetime of your keys. In other words, since the Memcache service

is shared by everyone (though it is isolated so only you have access to your data), App Engine will attempt to retain keys and values as long as possible, but makes no guarantees about how long a key will exist.

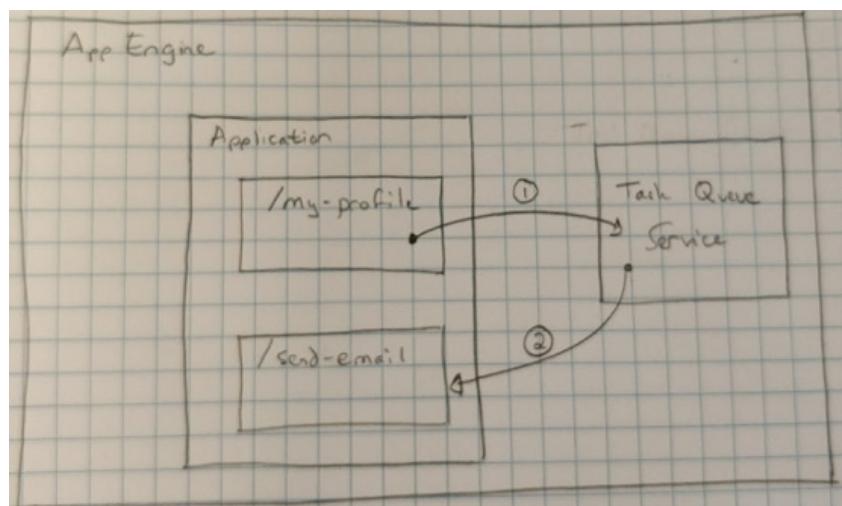
This means that you could write a key and come back for it a few minutes later, only to find that it's been removed. Following the precedent of traditional caching systems, Memcache will evict keys on a "least recently used" (LRU) basis, meaning that a rarely accessed key is far more likely to be evicted ahead of a frequently accessed key. Let's now dive into a more complex style of hosted services, where you can defer work for later using App Engine Task Queues.

### 11.4.3 Deferring tasks

In many applications, you may find that your code has some work to do that need not be done right away but instead could be delayed. This "work" might be sending an e-mail or re-calculating some difficult result, but typically it's something that takes a while and can be done in the background. To handle this, we may end up building our own system to handle "work to be done later" (e.g., storing the work in a database and having a worker process handle it) or use a third-party system, but App Engine comes with a system built-in that makes it easy to push work off until later, called Task Queues.

To see how this works, let's imagine we have a web application with a "profile page" that stores a user's e-mail address. If they want to change that e-mail address we might want to send a confirmation e-mail to the new address in order to prove that they actually control the e-mail they provided. In this case, "sending an e-mail" might take a while so we wouldn't want to sit around waiting on the e-mail to be sent. Instead, we want to schedule the work to be done and once it's confirmed as "scheduled" we can send a response telling the user that they should get an e-mail soon.

**Figure 11.16. An application that uses Task Queues to schedule future work**



First, the code that updates the e-mail in the `/my-profile` URL makes a request to the Task Queues service (1) that says "Make sure to call the `/send-email` URL with some parameters". At some point in the future, the Task Queues service will make a request to that URL as we scheduled and our code picks up the baton, doing the actual e-mail sending work. In Python code, this might look something like the following snippet.

**Listing 11.57. An example application that uses Task Queues to schedule work for later**

```
import webapp2
from google.appengine.api import taskqueue

class MyProfileHandler(webapp2.RequestHandler):
    def get(self):
        self.response.write('This is your profile.') ②

    def post(self): ③
        task = taskqueue.add(④
            url='/send-email',
            params={'email': self.request.get('email')})

class SendEmailHandler(webapp2.RequestHandler):
    def post(self):
        some_email_library.send_email(email=self.request.get('email')) ⑤

app = webapp2.WSGIApplication([
    ('/my-profile', MyProfileHandler),
    ('/send-email', SendEmailHandler),
]) ⑥
```

- ① We first need to import the task queue libraries for Python.
- ② Here we might render a full HTML page where users can change their e-mail.
- ③ When someone makes a POST request to the `/my-profile` URL, it's handled here instead of in the previous method.
- ④ We can use the `taskqueue.add()` method to schedule a future execution, in this case to make a POST request to `/send-email` with some request parameters.
- ⑤ The Task Queues service will make the request as we scheduled, so it's our job to define what happens at that point. In this case we would send an e-mail to the desired recipient.

The Task Queues service is incredibly powerful and has far more features than we can go through in a single chapter, for example, you can schedule requests to be handled by other services, limit the rate of requests that are processed at a given time, and even use a simpler code syntax for Python that allows you to defer a single function that doesn't necessarily have a URL mapping defined in your application.

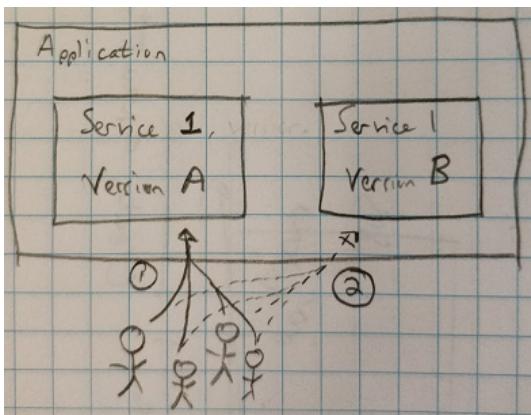
All of these things and more can be found in a book on App Engine itself or in the Google Cloud Platform documentation, so if you're particularly interested in this feature you should definitely explore it further in those other resources. Let's look at one more feature of App Engine that is particularly unique as well as useful: traffic splitting.

#### 11.4.4 Splitting traffic

As we saw when deploying new versions of services, it's possible to trigger a deployment without making the new version "live" yet, which allows you to run multiple versions side by side and then do hot switch-overs between versions. Switching over immediately is great, but what if we wanted to slowly test out new versions, shifting traffic from one version to another over the course of the day?

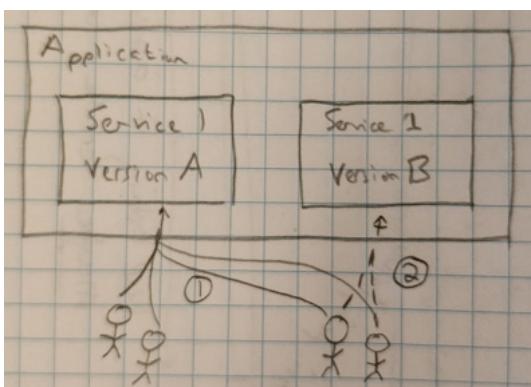
For example, shown below you can see a hard cut-over from version A to version B, where 100% of the traffic is originally sent towards version A (1) and then immediately jumps over to version B (2).

**Figure 11.17. A hard cut-over of all traffic from version A (1) to version B (2)**



With traffic splitting, you can control what percentage of traffic goes to which versions, meaning that you could go from a state where 100% of traffic is sent to version A (1) and transition to a state where 50% remains on version A and 50% is migrated to version B (2).

**Figure 11.18. A soft transition of 50% of traffic to version B (2)**



There are lots of reasons why you might want to use traffic splitting. For example, you

may want to do "A/B testing" where you show different versions to different groups and decide which one to make "official" after feedback from these users. Or it may be a more technical reason, where a new version rewrites some data into a new format so you want to slowly expand the number of people using it to avoid bombarding your database with updates. We'll talk about A/B testing here because it most clearly illustrates the functionality offered by App Engine's traffic splitting.

**NOTE**

**It may be helpful to set the `promote_by_default` flag back to `false` to avoid the automatic hard cut-over during a typical deployment.**

To demonstrate this, let's deploy two versions of a service (called `trafficsplit`) using the `--version` flag to name them `version-a` and `version-b`. Let's start by deploying `version A`, which is just our "Hello, world" application tweaked so that it says "Hello from version A!" After that, let's deploy a second version, called `version-b` which is just slightly modified to say "Hello from version B!" (instead of "version A!"). Once you're done deploying, you should be able to access both versions by their names, with `version-a` being the default, demonstrated below.

**Listing 11.58. After both versions are deployed**

```
$ curl http://trafficsplit.your-project-id-here.appspot.com
Hello from version A!

$ curl http://version-a.trafficsplit.your-project-id-here.appspot.com
Hello from version A!

$ curl http://version-b.trafficsplit.your-project-id-here.appspot.com
Hello from version B!
```

At this point we have one version that is the default (`version-a`) and another that is deployed by not yet the default (`version-b`). If we click on the "Versions" heading in the left-side navigation, and choose `trafficsplit` from the service dropdown, we can see the current "split" (or allocation) of traffic is 100% to `version-a` and 0% to `version-b`.

**Figure 11.19. Available versions**

The screenshot shows the Google Cloud Platform App Engine interface. On the left, a sidebar lists various services: Dashboard, Services, Versions (which is selected and highlighted in blue), Instances, Task queues, Security scans, Firewall rules, and Quotas. The main content area is titled "Versions" and shows a table of deployed versions for the service "trafficsplit". The table includes columns for Version, Status, Traffic Allocation, Instances, Runtime, Environment, Size, Deployed date, and Tools/View. Two rows are present: "version-b" (Serving, 0% traffic, 0 instances, nodejs runtime, Flexible environment, 0 B size, deployed on Oct 6, 2017, 9:36:58 AM by jjg@google.com) and "version-a" (Serving, 100% traffic, 3 instances, nodejs runtime, Flexible environment, 0 B size, deployed on Oct 6, 2017, 9:26:37 AM by jjg@google.com).

If we wanted to split 50% of the traffic currently going to `version-a`, we could do this by clicking on the "Split traffic" icon (which looks like a road sign forking into two arrows), which brings us to a form where we can configure how to split the traffic.

**Figure 11.20. The form where we can choose how to split traffic between versions**

The screenshot shows the "Split traffic" configuration form. At the top, there's a note: "You can split incoming traffic to different versions of your app. Traffic splitting is useful for slowly rolling out new versions or A/B testing different designs and features [Learn more](#)". Below this, there are two sections: "Split traffic by" (with options for IP address, Cookie, and Random, where Random is selected) and "Traffic allocation". The "Traffic allocation" section contains two entries: "version-a" (50%) and "version-b" (50%). There's also a "Save" button at the bottom.

First, for the purposes of this demonstration we'll use the "random" strategy when deciding which requests go to which versions. Generally, the cookie-based strategy is best for user-facing services since it means that the same user won't see a mix of versions but instead will stick with a single version per session. After that, we'll add `version-b` to the traffic allocation list and route 50% of the traffic to that version. Once that's done, just click "Save", at which point viewing the same list of versions for

our service should now show that half of the traffic is heading towards `version-a`, and the other half towards `version-b`.

**Figure 11.21. The list of versions with traffic split evenly between each**

The screenshot shows the Google Cloud Platform App Engine interface. On the left, there's a sidebar with options like Dashboard, Services, Versions (which is selected), Instances, Task queues, Security scans, Firewall rules, and Quotas. The main area is titled 'Versions' and shows a table of deployed versions for a service named 'trafficsplit'. The table has columns for Version, Status, Traffic Allocation, Instances, Runtime, Environment, Size, Deployed (with a timestamp and user info), Diagnose, and View. Two rows are visible: 'version-b' and 'version-a', both marked as 'Serving' with 50% traffic allocation across 2 instances, running on nodejs in a Flexible environment.

Version	Status	Traffic Allocation	Instances	Runtime	Environment	Size	Deployed	Diagnose	View
version-b	Serving	50%	2	nodejs	Flexible	0 B	Oct 6, 2017, 9:36:58 AM by jjg@google.com	Tools	View
version-a	Serving	50%	2	nodejs	Flexible	0 B	Oct 6, 2017, 9:26:37 AM by jjg@google.com	Tools	View

To check whether this worked, we can make a few requests to the default URL for our service, and see how we flip-flop between answers from the various versions, shown below.

#### **Listing 11.59. Requests are handled by different versions randomly as we configured**

```
$ curl trafficsplit.your-project-id-here.appspot.com
Hello from version A!
$ curl trafficsplit.your-project-id-here.appspot.com
Hello from version B!
$ curl trafficsplit.your-project-id-here.appspot.com
Hello from version B!
$ curl trafficsplit.your-project-id-here.appspot.com
Hello from version A!
$ curl trafficsplit.your-project-id-here.appspot.com
Hello from version B!
```

As mentioned before, there are many many more things that App Engine is capable of, which could fill an entire book so if you're interested in learning about all of the different features of App Engine, it's definitely worth picking up a book focusing exclusively on the topic. Unfortunately there simply isn't enough space to talk about everything, and so it's now time to switch gears and look at how much it costs to run your applications on App Engine.

## **11.5 Understanding pricing**

Since App Engine has many different services, each with their own pricing schemes, instead of going through every single service and looking at how much it costs we'll look at how much the computational aspects of App Engine are priced and look at

costs for a few of the services that we discussed in this chapter, starting with computing costs.

Since App Engine Flex is built on top of Compute Engine instances, the cost is identical to Compute Engine, which were discussed in depth in [“Understanding pricing”](#). App Engine Standard, on the other hand, uses a sandbox with different instance types, but still follows the same principle: App Engine Standard instances are priced on a per-hour basis, which vary depending on the location of your application. For example, the F4 instance in Iowa (`us-central1`) costs \$0.20 per hour, but in Sydney (`australia-southeast1`) that same instance will cost \$0.27 (35% more). The table below shows prices for the various instance types in Iowa.

**Table 11.2. Cost for various App Engine Instance types**

Instance type	Cost
F1	\$0.05 per hour
F2	\$0.10 per hour
F4	\$0.20 per hour
F4_1G	\$0.30 per hour

In addition to the cost for computing resources, App Engine also charges for outgoing network traffic just like the other computing environments we've seen. For App Engine Flex, the cost is again equivalent to the cost for Compute Engine network traffic. For App Engine Standard, there is a flat rate per GB which varies by location from \$0.12 per GB in Iowa (`us-central1`) to \$0.156 per GB in Tokyo (`asia-northeast1`).

Finally, many of the other API services offered (e.g., Task Queues or Memcache) do not charge for API calls but might charge for data stored in the API. For example, in the case of Task Queues the cost is \$0.03 per GB of data stored, but for shared Memcache there is no charge for data cached. To see more about this, it's worth looking through the pricing details which are available at [cloud.google.com/appengine/pricing](#). Now that we've covered how much everything costs, let's zoom out and look at the big picture of when to use App Engine and if so, which environment is the best fit.

## 11.6 When should I use App Engine?

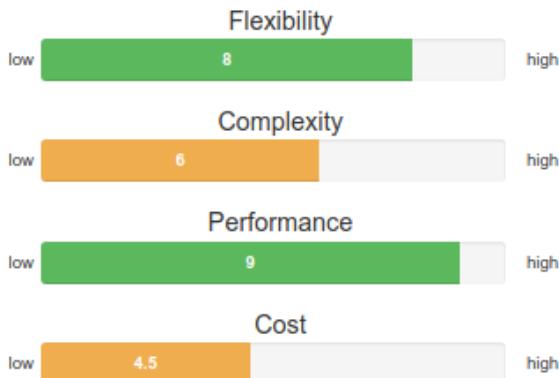
To figure out whether or not App Engine is a good fit, let's start by looking at the score card which gives us a broad overview of the different characteristics of App Engine. However, since App Engine's different environments are almost like entirely separate computing platforms, it seems worthwhile to have a separate score card for the different options. Let's start by looking at App Engine Standard.

**Figure 11.22. Scorecard for App Engine's two environments**

## App Engine Standard



## App Engine Flex



### 11.6.1 Flexibility

The first thing to notice about App Engine is that while App Engine Flex offers similar levels of flexibility (that is, what code you can run) to Compute Engine or Kubernetes Engine, App Engine Standard is far more limited. This is due to the fact that App Engine Standard relies on a sandbox runtime to execute code and therefore is limited to specific programming languages. That said, after looking in more detail at the flexibility of each environment regarding the control and management of underlying resources, it turns out that there is much less of a discrepancy. For example, we have different ways to configure how scaling works (e.g., manual or automatic scaling), for both environments, and are able to choose specific instance configurations for both environments. Overall, App Engine Standard is slightly less flexible with regard to

instance configuration whereas App Engine Flex allows you to control almost all details about the resources that will be running your code.

### **11.6.2 Complexity**

When it comes to the complexity of the two environments, we see a discrepancy, despite the overall moderate score in this area. With App Engine Standard, there is quite a lot to learn, specifically with regard to the runtime environments and the limitations that come with them. For example, when we were building a "Hello, world!" application Python we relied on the `webapp2` framework which happens to work quite well with App Engine. If we wanted to use a different Python web framework (such as Django or Flask), we'd have to do a bit of work to ensure that everything runs correctly, rather than it running "right out of the box".

App Engine Flex on the other hand is similar in overall complexity to something like Compute Engine, though slightly scaled down since you don't really need to understand all of the scaling details like instance groups. It's also slightly less complex than Kubernetes Engine as there's no need to learn and understand all the details of Kubernetes. In short, App Engine Flex has a relatively shallow learning curve, whereas App Engine Standard has a much steeper one.

### **11.6.3 Performance**

Since App Engine Flex relies on Compute Engine VMs, the overall performance of your services should be about as good as you'll get on a cloud computing platform. In other words, in App Engine Flex, there's really only a small bit of overhead consuming any of the CPU time on the instances running your code. On the other hand, since App Engine Standard executes your code in a sandbox environment, we see a very different performance profile. This poor showing is primarily due to the runtime itself having extra work to do in order to ensure that code executes safely, which means that executing intense computational work may not be the best fit for App Engine Standard.

### **11.6.4 Cost**

The first thing to get out of the way is that App Engine Flex has pricing that is almost identical to Compute Engine, making it quite reasonable. Since you're paying for Compute Engine instances, the rates themselves are the same, however the scaling is (by default) controlled by App Engine, meaning that you may over-provision leading to a higher overall cost. App Engine Standard has a similar pricing model, though overall it seems to be a bit more expensive.

For example, in Iowa (`us-central1`) we saw that a `F1` instance costs \$0.05 per hour, however in that same region an `n1-standard-1` Compute Engine instance costs slightly less (at \$0.0475 per hour). Additionally, the Compute Engine instance has 3.75 GB of memory available where the App Engine `F1` instance has only 128 MB, and the 1 vCPU in GCE is equivalent to a 2.0+ GHz CPU whereas the `F1` instance is roughly equivalent to a 600 MHz CPU (though this is in a sandbox, not running a full operating system). Overall, this is a very difficult comparison to make, though generally it seems

that the overall performance you get for a Compute Engine instance will outperform an equivalently sized App Engine Standard instance.

Lastly, App Engine Standard has both a permanent free tier as well as the ability to "scale down to zero" (costing no money at all when an application isn't in use), whereas neither Compute Engine, Kubernetes Engine, nor App Engine Flex have this ability. This means that for toy or hobby applications that don't see a lot of steady traffic, this feature alone makes App Engine Standard a clear winner.

### **11.6.5 Overall**

Now that we've seen the various ways that App Engine stacks up, let's look at our example applications and see whether App Engine might be a good choice.

#### **To-do list**

The first example application we discussed was a to-do list service where people could create lists of things to do and add items to those lists, crossing them off as things were completed. Since this application is pretty unlikely to see a lot of traffic (and also happens to be a common getting-started toy project), App Engine Standard might actually be a great fit simply from the perspective of cost. Let's look at how App Engine Standard stacks up first.

**Table 11.3. To-do list application computing needs**

Aspect	Needs	Good fit for Standard?	Good fit for Flex?
Flexibility	Not all that much	Definitely	Overkill
Complexity	Simpler is better	Mostly	Mostly
Performance	Low to moderate	Definitely	Overkill
Cost	Lower is better	Perfect	Not ideal

Overall, App Engine Standard is a pretty good fit particularly in the cost category due to the ability to "scale down to zero". App Engine Flex on the other hand is a bit overkill in a few areas and not quite a perfect fit when it comes to the cost goal.

#### **E\*Exchange**

E\*Exchange, an application that provides an online stock trading platform, has more complex features, may require the ability to run custom code in a variety of languages, and wants to ensure pretty efficient use of computing resources to avoid overpaying for computing power. Additionally, this application is representing a real business which is quite different from a toy project like a to-do list. Shown below is how the computing needs of E\*Exchange pan out for both App Engine Flex and Standard.

**Table 11.4. E\*Exchange computing needs**

Aspect	Needs	Good fit for Standard?	Good fit for Flex?
Flexibility	Quite a bit	Not so good	Definitely
Complexity	Fine to invest in learning	Mostly	Mostly

Performance	Moderate	Not so good	Definitely
Cost	Nothing extravagant	Acceptable	Definitely

As you can see, the limitations of App Engine Standard outweigh the benefits of the free tier and ability to "scale to zero" (since this application is very unlikely to ever be with any traffic at all). Even though the cost of App Engine Standard is acceptable, it seems that App Engine Flex is a much better fit than the others. App Engine Flex provides the needed flexibility, performance, and cost, with a pretty reasonable fit when it comes to the learning curve of getting up to speed on App Engine Flex. Overall, while App Engine Standard doesn't quite fit, App Engine Flex is a fine choice when deciding where to run the E\*Exchange application.

### **InstaSnap**

InstaSnap, the very popular social media photo sharing application, is a bit of a hybrid in its computing needs, with some demands (like performance and scalability) being quite extreme and others (like cost) being quite lax. As a result, finding a fit for InstaSnap is a bit more like looking at what **doesn't** fit as a way to rule out an option, which in this case is quite obvious.

**Table 11.5. InstaSnap computing needs**

Aspect	Needs	Good fit for Standard?	Good fit for Flex?
Flexibility	A lot	Not at all	Mostly
Complexity	Eager to use advanced features	Not really	Mostly
Performance	High	Not at all	Definitely
Cost	No real budget	Definitely	Definitely

As shown in the table above, InstaSnap's demands for performance and flexibility (given that it wants to try everything under the sun) rules out App Engine Standard right away. Compare that to App Engine Flex and we see quite a different story. All of the performance and flexibility needs are there given that App Engine Flex is based on Docker containers and Compute Engine instances, while the learning curve is certainly not a deterrent to adopting App Engine Flex.

**NOTE**

It's interesting to note here that SnapChat actually began on App Engine Standard and continues to run quite a bit of computing infrastructure there as of this writing. That said, App Engine Flex is a far better choice and had it existed when SnapChat was founded it's likely they would have chosen to start there (or Kubernetes Engine).

The desire to use bleeding edge features, however, makes something like Kubernetes and Kubernetes Engine a slightly better fit for this project than App Engine Flex. The reason is nothing more than the fact that Kubernetes is open-source which means that it's easy to customize scaling options, adopt or write plug-ins, and extend the scaling platform itself whereas with App Engine Flex you're limited to the settings exposed to your `app.yaml` file.

## 11.7 Summary

- App Engine is a fully-managed cloud computing environment aimed at simplifying the overhead needed for all applications (such as setting up a cache service).
- App Engine has two very different environments: Standard which is the more restricted environment, and Flex which is less restrictive and container-based.
- App Engine Standard supports a specific set of language run-times, whereas App Engine Flex supports anything that can be expressed in a Docker container.
- The fundamental concept of App Engine is the application, which can contain lots of services. Each service can then contain several versions that may run concurrently.
- Underneath each running version of an application's services are virtualized computing resources.
- The main draw of App Engine is automatic scalability, which can be configured to meet the needs of most modern applications.
- App Engine Standard comes with a specific set of managed services which are accessed via client libraries provided to the run-time directly (e.g., the `google.appengine.api.memcache` API for Python).
- App Engine pricing is based on the hourly consumption of the underlying compute resources. In the case of App Engine Flex the prices are identical to Compute Engine instance pricing.

# 12

## *Cloud Functions: Serverless applications*

### **This chapter covers:**

- What are micro-services?
- What is Google Cloud Functions?
- Creating, deploying, and triggering functions
- Updating and deleting functions
- Managing function dependencies
- How pricing works for Google Cloud Functions

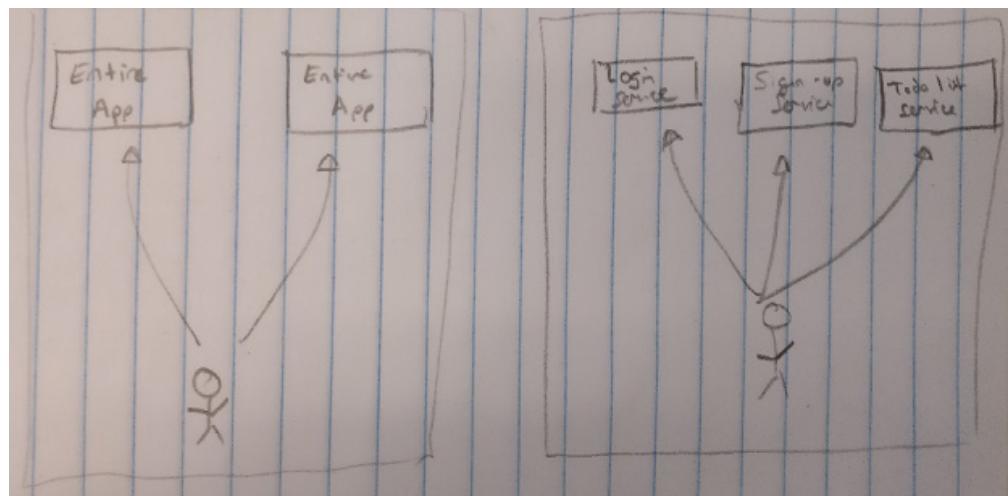
### **12.1 What are micro-services?**

A "micro-service architecture" is a way of building and assembling an application that keeps each concrete piece of the application as its own loosely-coupled part (called a micro-service). This might not seem all that interesting, but it's important to note that each micro-service can stand on its own, whereas a traditional application has many parts that are intertwined with one another, incapable of running on their own.

For example, when creating a typical application, you'd start a project and then start adding "controllers" that handle the different parts of the application. That is, when building a to-do list application you might start by adding the ability to sign up and login, and then add more functionality such as creating to-do lists, then creating items on those lists, searching through all the lists for matching items, and more. In short, this big application would be a single code-base, running on a single server somewhere, where each server was capable of doing all of those actions because it's just different functionality added to a single application.

Micro-services take a hatchet to this design, chopping up each bit of functionality into its own loosely coupled piece, responsible for a single standalone feature. In the case of the to-do list example, this might mean you'd have a micro-service responsible for signing up, another for logging in, and others for searching, adding items, creating lists, and more. In a sense, you can think of this as a very fine-grained service-oriented architecture (commonly known as SOA).

**Figure 12.1. Micro-service architecture compared to a traditional application**



So why would you want to have this type of architecture? Where is the benefit over a typical "monolithic" application? One of the biggest benefits is hiding in the fact that each service is only loosely coupled to any other services.

Since each micro-service can run on its own, this means that development (particularly testing) is very narrow and constrained, which means that it's easier for new team members to get up to speed. Also, having each piece isolated from the others means that deployment is much more straight forward. Further, since each piece must fulfill a contract (e.g., the login service must set a cookie or return a secure login token), the implementation under the hood doesn't really matter so long as that contract is fulfilled by the service. This means that what would be major changes in a monolithic application (e.g., rewriting a piece in a different language) is pretty simple: just rewrite the micro-service and make sure it upholds the same contractual obligations.

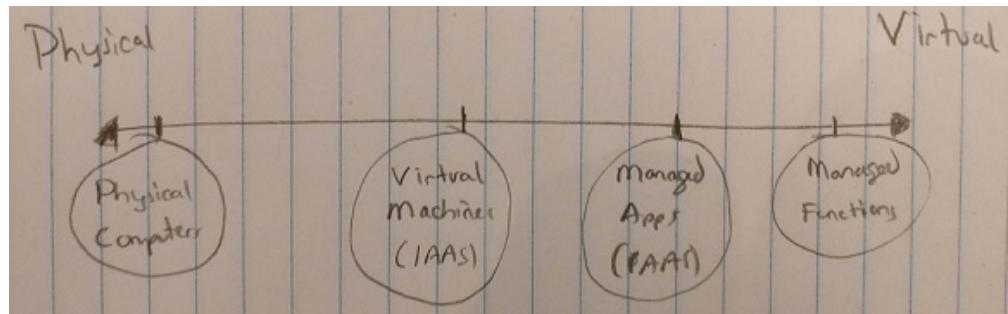
There are many more benefits to using a micro-service architecture (and entire books written on the subject), so let's jump ahead and look now at how Google Cloud Platform makes it easy to design, build, deploy, and run micro-services on GCP.

## 12.2 What is Google Cloud Functions?

As you learned in [chapter 9](#), the first step towards enabling cloud computing has been the abstraction of physical infrastructure in favor of virtual infrastructure. In other words, you used to worry about installing and running a physical computer but now

you're able to turn on a virtual computer in a few seconds. This pattern of abstracting away more and more has continued, and Cloud Functions takes that concept to the far end of the spectrum. This also happens to fit really well with micro-service architectures, as the goal there is to design lots of stand-alone pieces, each responsible for a single part of an application.

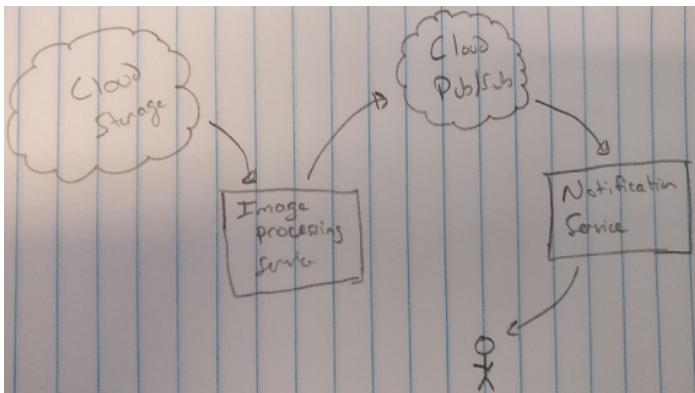
**Figure 12.2. The spectrum of computing from physical to virtual**



With Cloud Functions, instead of thinking about virtual servers (like Compute Engine), or containers (like Kubernetes Engine), or even "applications" (like App Engine), you think only about single functions which run in an entirely "serverless" environment. This means that instead of building and deploying an application to a server and worrying about how much disk space you need, you write only short narrowly-scoped functions and these functions are run for you on demand. These single functions can be considered the micro-services that we discussed above.

While the idea of a single function on its own isn't all that exciting, the glue that brings these functions together is what makes them special. In the typical flow of an application, most requests are triggered by users making requests, usually over HTTP (e.g., a user somewhere logs into your app). In the world of Cloud Functions, other types of events from lots of different cloud services can trigger requests (in addition to regular HTTP requests). For example, a function can be triggered by someone uploading a file to Cloud Storage or a message being published via Cloud Pub/Sub.

**Figure 12.3. Using other cloud services' events as glue between micro-services**

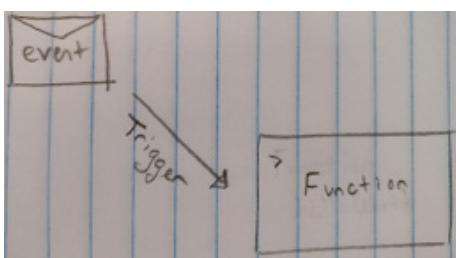


All of these events can be monitored by different triggers which can then run different functions, and it's this unique ability to compose different pieces together that makes Cloud Functions so interesting. Put more concretely, Cloud Functions allow you to "hook" small pieces of code to different events, and have that code run whenever those events happen. For example, you could hook a function up so that it runs whenever a customer uploads a file into a Cloud Storage bucket, and that function might run automatically tag the image with labels from the Cloud Vision API. Now that we've gone through what micro-services are and what makes Cloud Functions unique, let's dig into the underlying building blocks needed to actually do something with Cloud Functions.

### 12.2.1 Concepts

Cloud Functions is the overarching name for a category of concepts, one of them being an actual "Function". But a function all by itself isn't all that useful without the ability to connect it to other things, which leads us to a few other concepts: events and triggers. These all work together to form a pipeline that you can use to build interesting applications. We'll get into lots of detail in just a moment, but before doing that, let's look at how these different parts fit together, starting from the bottom up (and shown in the diagram below).

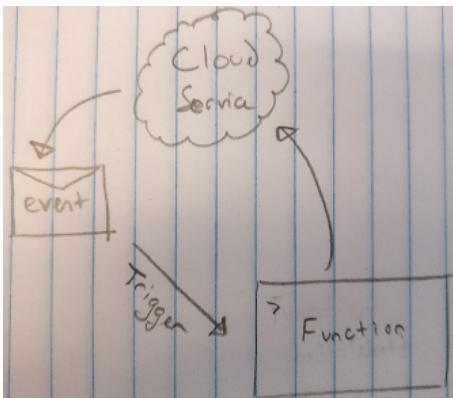
**Figure 12.4. Overview of different concepts**



**Events** are things that can happen (e.g., a Cloud Storage Object is

created). **Functions** are chunks of code that run in response to events. **Triggers** are ways of coupling a function to some events. In other words, creating a trigger is like saying, "Make sure to run Function X whenever a new GCS Object is created." Additionally, since functions can call into other cloud services, they could cause further events in which other triggers cause more functions to run. This is how you could connect multiple micro-services together to build complex applications out of lots of simple pieces.

**Figure 12.5. Building complex applications out of simple concepts**



## EVENTS

As you learned already, an event corresponds to something happening, which may end up causing a function to run. The most common "event" that you're very likely familiar with is an HTTP request, but these events can also come from other places such as Google Cloud Storage or Google Cloud Pub/Sub. Every event will come with a few different attributes which you can use when building your function. This includes the basic details of an event (such as a unique ID, the type of the event, and a timestamp of when the event occurred), as well as the resource targeted by the event and payload of data specific to the event. While the data attached to an event will depend on the type of the event, common data types are used to minimize the learning curve. For example, HTTP events will have an Express Request object in the data field, whereas events from Cloud Storage will have the affected Cloud Storage Object in the data field.

Even though events from different sources share quite a bit in common, they actually fall into two categories. Events based on HTTP requests are "synchronous" events (meaning that the requester is waiting for a response), while those coming from other services such as Cloud Pub/Sub are "asynchronous" (that is, they run in the background). This is an important distinction because the code you write to list for synchronous events will be slightly different from that for asynchronous events. As you can see, events are the basic building blocks used to pass along information about things happening, a bit like the body of a notification. To understand how you can act on this information, let's look at functions and how you actually write them.

## FUNCTIONS

As you learned above, the idea of a micro-service architecture is to split different responsibilities of an application into separate services which run on their own. In the world of Cloud Functions, the function itself is the equivalent of a single micro-service in an application. This means that a function should be responsible for a single thing and have no problem standing on its own.

So what makes up a function? At its core, a function is an arbitrary chunk of code that can run on-demand. It also comes with extra configuration that tells the Cloud Functions runtime how to execute the function. This includes things like how long to run before "timing out" and returning an error (defaulting to 1 minute, but configurable up to 9 minutes), and the amount of memory to allocate for a given request (defaulting to 256 MB).

All that said, the key part of any Cloud Function is the code that you're able to write. Google Cloud Functions let's you write these functions in JavaScript, but depending on whether you're dealing with a synchronous event (like an HTTP request) or an asynchronous event (like a Pub/Sub message) the structure of the function can be slightly different. To start, let's look at synchronous events. Functions written to handle synchronous events use a "request and response" syntax, similar to request handlers in Express. For example, a function body that echoes back what was sent would look like the following.

### **Listing 12.1. A Cloud Function that echoes back the request if it was plain-text**

```
exports.echoText = (req, res) => { ①
  if (req.get('content-type') !== 'text/plain') { ②
    res.status(400).send('I only know how to echo text!');
  } else {
    res.status(200).send(req.body); ③
  }
};
```

- ① This function is named `echoText` and mapped to the same name when exported.
- ② Here we can read the request header for content type and show an error for non plain-text requests.
- ③ If the request was plain-text, we can echo the body back in the response.

If you're at all familiar with web development in JavaScript, this function should not be a surprise at all. If you're not, the idea of this is that you get both a request and a response as arguments to the function. You can read from the request and send data back to the user by calling functions (like `.send()`) on the response. When the function completes, the response will be closed and the request considered completed.

So how about the other class of functions? How do you write code for asynchronous events to handle things like a new message arriving from Cloud Pub/Sub? Functions written to handle asynchronous events like this are called "background functions", and instead of getting the request and response as arguments, they just get the event along with a callback which signals the completion of the function. For example, let's look at

a function that logs some information based on an incoming Pub/Sub message.

### **Listing 12.2. A Cloud Function that logs a message from Cloud Pub/Sub**

```
exports.logPubSubMessage = (event, callback) => { ①
  const msg = event.data; ②
  console.log('Got message ID', msg.messageId); ③
  callback(); ④
};
```

- ① Background functions are provided with an event and a callback rather than a request and a response.
- ② The Pub/Sub message itself is stored in the event data.
- ③ Just like a regular message, the event ID is attached and accessible.
- ④ Call the callback to signal that the function has completed its work.

As you can see in this function, the event is passed in as an argument which we can read from and do things with, and when we're done we simply call the `callback` provided. The obvious question though is, "How did the Pub/Sub message get routed to the function?" or "How did an HTTP request get routed to the first function?" This is a great question, and brings us to the concept of triggers which allow you to decide which events are routed to which functions.

#### **TRIGGERS**

Triggers, for lack of a better analogy, are like the glue in Google Cloud Functions. They are the way you specify which events (and which types of events) should be routed to a given function. Currently, this is done on the basis of the provider. In other words, you specify that you're interested in events from a given service (such as Cloud Pub/Sub), as well as some filter to narrow down which resource you want events from (such as a specific Pub/Sub topic).

This is pretty simple and something you do when you deploy your function, but this brings us to the next question, "How do you get your functions 'up there in the cloud'?" To see how this works, let's explore building, deploying, and triggering a function from start to finish.

## **12.3 Interacting with Cloud Functions**

Working with Cloud Functions involves a few steps. First, we have to write the function itself in JavaScript. After that, we have to deploy it to Google Cloud Functions, and in the process we'll define what exactly triggers it (e.g., HTTP requests, Pub/Sub messages, or Cloud Storage notifications). Then we'll need to verify that everything works by making some test calls and then some live calls. Let's start by writing a function that responds to HTTP requests by echoing back the information sent, and adding some extra information.

### **12.3.1 Creating a function**

The first step towards working with Cloud Functions is for us to write our function.

Since this will be a synchronous function (rather than a background function), we'll write it in the "request and response" style as you saw above. Let's start by creating a new directory called echo and then in that directory, a new file called `index.js`. Then just put the following code in that file.

#### **Listing 12.3. A function that echoes some information back to the requester**

```
exports.echo = (req, res) => {
  let responseContent = {
    from: 'Cloud Functions'
  };

  let contentType = req.get('content-type');

  if (contentType == 'text/plain') {
    responseContent.echo = req.body;
  } else if (contentType == 'application/json') {
    responseContent.echo = req.body.data;
  } else {
    responseContent.echo = JSON.stringify(req.body);
  }

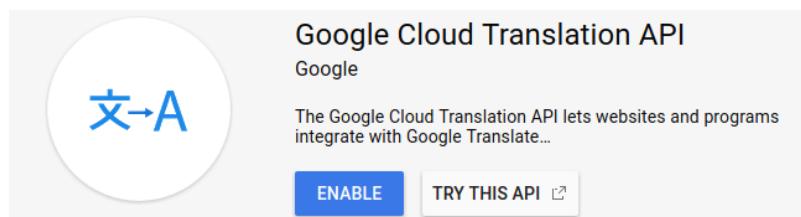
  res.status(200).send(responseContent);
};
```

As you can see, this request specifically accepts text requests and responds with a JSON object with the text provided along with some extra data saying that this came from Cloud Functions. At this point we now have a function on our local file system, but we now have to get it "in the cloud". To see how to do that, let's move along and look at how to deploy our function.

#### **12.3.2 Deploying a function**

Taking a function that we wrote locally and deploying it is the one step of the process where we'll need to do a little bit of set-up. More specifically, we'll need a Cloud Storage bucket which is where the actual content of our functions will live. Additionally, if you haven't already, you'll need to enable the Cloud Functions API in your project. Let's start with enabling the Cloud Functions API. To do this, navigate to the Cloud Console and type in "Cloud Functions API" in the search box at the top of the page. Click on the first (and only) result, and then on the next page, click on the "Enable" button and you should be all set.

**Figure 12.6. Enable the Cloud Functions API**



Next, we need to create our bucket. To do this, you can use the Cloud Console, or use the command-line tool (`gcloud`), but for this example we'll use the Cloud Console. Start by navigating to the Cloud Console and choose Storage from the left-side navigation. You should see a list of buckets you already have, and to create a new one, just click the "Create bucket" button at the top of the page. In this example, we'll leave the bucket as multi-regional in the US (take a look at [chapter 8](#) for more details on these options).

**Figure 12.7. Create a new bucket for your cloud function**

The screenshot shows the 'Create a bucket' dialog. At the top, there's a back arrow and the title 'Create a bucket'. Below that is a 'Name' field containing 'my-cloud-functions'. Under 'Default storage class', 'Multi-Regional' is selected. It describes this as best for streaming video and hosting web content. There are also options for 'Regional', 'Nearline', and 'Coldline'. A 'Multi-Regional location' dropdown is set to 'United States'. Below that is a 'Specify labels' section with a dropdown menu. At the bottom are 'Create' and 'Cancel' buttons.

**Create a bucket**

**Name** [?](#)  
Must be unique across Cloud Storage. Privacy: Do not include sensitive information in your bucket name. Others can discover your bucket name if it matches a name they're trying to use.

my-cloud-functions

**Default storage class** [?](#)

Multi-Regional  
Use to stream videos and host hot web content.  
Best for data accessed frequently around the world.

Regional  
Use to store data and run data analytics.  
Best for data accessed frequently in one part of the world.

Nearline  
Use to store rarely accessed documents.  
Best for data accessed less than once per month.

Coldline  
Use to store very rarely accessed documents.  
Best for data accessed less than once per year.

**Multi-Regional location**  
Redundant across 2+ regions within your selected location.

United States

Specify labels

**Create** **Cancel**

Once you have a bucket to hold your functions, we'll use the `gcloud` tool to deploy our function from the parent directory.

**Listing 12.4. Command to deploy our new function**

```
$ tree ①
.
└── echo
    └── index.js

1 directory, 1 file
$ gcloud beta functions deploy echo --source=./echo/ \
--trigger-http --stage-bucket=my-cloud-functions ②
```

- ① Your directory tree should show the echo directory with your index.js file living inside.
- ② Make sure to change the bucket name to match your bucket name!

This command tells Cloud Functions to create a new function handle called echo, from the file that we noted in echo/index.js and from the function that we exported (which happened to be called echo). This also says to trigger the function from HTTP requests, and to put the function itself into our staging bucket.

After running this function, you should see the following output.

**Listing 12.5. Output of deploying a function**

```
$ gcloud beta functions deploy echo --source=./echo/ \
--trigger-http --stage-bucket=my-cloud-functions
Copying file:///tmp/tmp4tZGmF/fun.zip [Content-Type=application/zip]...
/ [1 files][ 247.0 B/ 247.0 B]
Operation completed over 1 objects/247.0 B.
Deploying function (may take a while - up to 2 minutes)...done.
availableMemoryMb: 256
entryPoint: echo
httpsTrigger:
  url: https://us-central1-your-project-id-here.cloudfunctions.net/echo
latestOperation:
operations/ampnLNsb3VkJlc2VhcmNoL3VzLWNlbnRyYWwxL2VjaG8vaVFZMTM5bk9jcUk
name: projects/your-project-id-here/locations/us-central1/functions/echo
serviceAccount: your-project-id-here@appspot.gserviceaccount.com
sourceArchiveUrl: gs://my-cloud-functions/us-central1-echo-mozfapskkzki.zip
status: READY
timeout: 60s
updateTime: '2017-05-22T19:26:32Z'
```

As you can see, gcloud starts by bundling up the functions you have locally and uploading them to your Cloud Storage bucket. Once the functions are safely in the bucket, it tells the Cloud Functions system about the function mappings, and in this case, creates a new URL that you can use to trigger your function ([us-central1-your-project-id-here.cloudfunctions.net/echo](https://us-central1-your-project-id-here.cloudfunctions.net/echo)). How about we take our new function out for a spin?

### **12.3.3 Triggering a function**

With our newly deployed Cloud Function since it's triggered via HTTP it comes with a friendly URL to trigger the function. Let's try that out using curl in the command-line.

**Listing 12.6. Checking that the function works using curl**

```
$ curl -d '{"data": "This will be echoed!"}' \
-H "Content-Type: application/json" \
"https://us-central1-your-project-id-here.cloudfunctions.net/echo"
{"from":"Cloud Functions","echo":"This will be echoed!"}
```

As you can see, the function was run and returned what we expected! But what about functions triggered by something besides HTTP? It turns out that this would be a real pain if you had to actually do the thing that would trigger the event (e.g., actually create an object in Cloud Storage). To deal with this, the `gcloud` tool has a `call` function that triggers a function and allows you to pass in the relevant information. This method simply executes the function and passes in the data that would have been sent by the trigger, so you can think of it a bit like an argument override. To see how this works, below is how we would execute the same thing using `gcloud`.

**Listing 12.7. Calling the function using gcloud**

```
$ gcloud beta functions call echo --data '{"data": "This will be echoed!"}'
executionId: 707s1yel116c
result: {"from":"Cloud Functions","echo":"This will be echoed!"}
```

Hopefully by now you have a decent enough grasp of how to write, deploy, and call a Cloud Function. To take this to the next step, let's look at a few common, but advanced, things that you'll need to know in order to build more complicated (and full-featured) applications with your functions, starting with updating an existing function.

## **12.4 Advanced concepts**

While the section happens to be called "advanced" concepts, most of these are actually pretty basic ideas, but are a bit hazy in this new runtime environment of Google Cloud Functions, and as a result, they become a bit more advanced. Let's start with something easy that you'll definitely need to do when building your functions: update an existing one.

### **12.4.1 Updating functions**

It may come as a surprise to learn that updating a function is the same thing as deploying it yet again. For example, let's tweak our `echo` function from before by adding a second parameter in the response content, just to show that we made a change. In other words, inside our `echo` function, let's start off `responseContent` with an extra field.

**Listing 12.8. Adding a new parameter to our response content**

```
let responseContent = {
  from: 'Cloud Functions',
  version: 1
```

```
};
```

If we were to re-deploy this function and then call it again, we should see the modified response.

#### **Listing 12.9. Re-deploying the echo function**

```
$ gcloud beta functions deploy echo --source=./echo/ \
--trigger-http --stage-bucket=my-cloud-functions
Copying file:///tmp/tmpgFmeR6/fun.zip [Content-Type=application/zip]...
/ [1 files][ 337.0 B/ 337.0 B]
Operation completed over 1 objects/337.0 B.
Deploying function (may take a while - up to 2 minutes)...done.
availableMemoryMb: 256
entryPoint: echo
httpsTrigger:
  url: https://us-central1-your-project-id-here.cloudfunctions.net/echo
latestOperation:
operations/ampnLWNsb3VkJc2VhcmNoL3VzLWNlbnRyYWwxL2VjaG8vUDB2SUM2dzhDeG8
name: projects/your-project-id-here/locations/us-central1/functions/echo
serviceAccount: your-project-id-here@appspot.gserviceaccount.com
sourceArchiveUrl: gs://my-cloud-functions/us-central1-echo-afkbzeghcgu.zip
status: READY
timeout: 60s
updateTime: '2017-05-22T22:17:27Z'

$ gcloud beta functions call echo --data='{"data": "Test!"}'
executionId: nwluxpwmef91
result: '{"from": "Cloud Functions", "version": 1, "echo": "Test!"}' ①
```

① As you can see, the new parameter (`version`) is returned after re-deploying.

Note also that if you were to list the items in your Cloud Functions bucket (in the example, this is `my-cloud-functions`), you'd see the previously deployed functions. This means that you have a safe back-up for your deployments in case you ever accidentally deploy the wrong one. Now that you've seen how to update (re-deploy) functions, let's look at deleting old or out-of-date functions.

#### **12.4.2 Deleting functions**

There will come a time when every function has served its purpose and is ready to be retired. In other words, you may find yourself needing to delete a function you'd previously deployed. This is easily done using the `gcloud` tool, shown below deleting the `echo` function that we built previously.

#### **Listing 12.10. Deleting our echo function**

```
$ gcloud beta functions delete echo
Resource
[projects/your-project-id-here/locations/us-central1/functions/echo]
will be deleted.

Do you want to continue (Y/n)? y
```

```
Waiting for operation to finish...done.
Deleted [projects/your-project-id-here/locations/us-central1/functions/echo].
```

Keep in mind that this doesn't delete the source code locally, nor does it delete the bundled up source code that was uploaded to your Cloud Storage bucket. Instead, you can think of this as de-registering the function so that it will no longer be served and removing all of the metadata such as the timeout, memory limit, and trigger configuration (in the case of our echo function this is just the HTTP endpoint).

That wraps up the things you might want to do to interact with your functions, so let's take a step back and look more closely at more advanced ways you can build your function. For starters, let's look at how to deal with dependencies on other Node.js packages.

### 12.4.3 Using dependencies

Very rarely is every line of code in your application written by you and your team. More commonly, you end up depending on one of the plethora of packages available on via the Node Package Manager (or NPM). As a result, it would be kind of annoying if you had to download and re-deploy duplicates of these packages in order to run your function. So let's see how Cloud Functions deals with these types of dependencies.

Let's imagine that our echo function wanted to include the Moment JavaScript library so that we can properly format dates, times, and durations. When developing our typical application, we'd simply use npm to do this and maintain the packaging details by running `npm install --save moment`. But what do we do with Cloud Functions? It turns out that we can use those same tools to make sure our dependencies are handled properly. To see this in action, let's start by "initializing" our package (using `npm init`) and then installing Moment inside the echo directory that we created previously.

#### **Listing 12.11. Initializing our package and installing Moment**

```
~/ $ cd echo
~/echo $ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (echo)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
```

```

author:
license: (ISC)
About to write to /home/jjg/echo/package.json:

{
  "name": "echo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}

Is this ok? (yes)

~/echo $ npm install --save moment
echo@1.0.0 /home/jjg/echo
└─ moment@2.18.1

npm WARN echo@1.0.0 No description
npm WARN echo@1.0.0 No repository field.

```

At this point, if you were to look at the file created, called `package.json`, you should see a dependency for the `moment` package.

#### **Listing 12.12. The newly created package.json file**

```
{
  "name": "echo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "moment": "^2.18.1"
  }
}
```

Now that our package is ready, let's modify our `echo` function to also say how much time has passed since Christmas in 2016.

#### **Listing 12.13. Using the dependency on Moment.js**

```

const moment = require('moment'); ①
exports.echo = (req, res) => {
  let now = moment(); ②
  let christmas2016 = moment('2016-12-25');

```

```

let responseContent = {
  from: 'Cloud Functions',
  christmas2016: moment.duration(christmas2016 - now).humanize(true) ③
};

let contentType = req.get('content-type');

if (contentType == 'text/plain') {
  responseContent.echo = req.body;
} else if (contentType == 'application/json') {
  responseContent.echo = req.body.data;
} else {
  responseContent.echo = JSON.stringify(req.body);
}

res.status(200).send(responseContent);
};

```

- ① Start by requiring the dependency as you always would.
- ② Use the library just as you would in a typical application.
- ③ Calculate the humanized difference between now and Christmas 2016.

Once this is done, let's re-deploy the function with the new code and dependencies.

#### **Listing 12.14. Re-deploying our function with the new dependency**

```

$ gcloud beta functions deploy echo --source=./echo/ \
--trigger-http --stage-bucket=my-cloud-functions

# ... Lots of information here ...

$ gcloud beta functions call echo --data='{"data": "Echo!"}'
executionId: r92y6w489inj
result: '{"from": "Cloud Functions", "christmas2016": "5 months ago", "echo": "Echo!"}'

```

As you can see, the new code we have successfully uses the Moment package to say that (as of this writing), Christmas 2016 was 5 months ago! Now that you can see that using other libraries works just as you'd expect, let's look at how you might call into other Cloud APIs, such as Cloud Spanner to store data.

#### **12.4.4 Calling other Cloud APIs**

Very rarely are applications completely stateless (that is, they have no need to store any data). As you can probably imagine then, it might make sense to allow your functions to read and write data from somewhere! To see how this works, let's look at how you can access a Cloud Spanner instance from your function.

First, if you haven't read [chapter 6](#) now is a great time to do that. If you're not interested in the particulars of Spanner but just want to follow along with an example showing how to talk to another Cloud API, that's fine too. In order to demonstrate reading and writing data from Spanner, we should start by creating an instance, a database, and then a table. For the first two, take a look at the chapter on Cloud

Spanner. However, for the table, let's create a simple logs table that has a unique ID (`log_id`) and a place to put some data (`log_data`), both as `STRING` types for simplicity.

The next thing we'll need to do is install (and add to our dependencies) a library to generate UUID values (`uuid`), and the Google Cloud Spanner Client for Node.js (`@google-cloud/spanner`). We can install these easily using `npm`.

#### **Listing 12.15. Install (and add dependencies for) the Spanner client library and UUID**

```
$ npm install --save uuid @google-cloud/spanner
```

Once those are installed, we'll need to update our code quite a bit, making two key changes. First, whenever we echo something, we'll log the content to Cloud Spanner by creating a new row in the logstable. Second, in each echo response, we'll return a count of how many entries exist in the logs table.

**NOTE**

**It's generally a bad idea to run a full count over your entire Spanner table, so this isn't recommended for something living in production.**

The code below does just this, while still pinning to the `echo` function.

#### **Listing 12.16. Our new Spanner-integrated echo function**

```
const uuid4 = require('uuid/v4');          ①
const Spanner = require('@google-cloud/spanner');

const spanner = Spanner();                 ②

const getDatabase = () => {               ③
  const instance = spanner.instance('my-instance');
  return instance.database('my-db');
};

const createLogEntry = (data) => {         ④
  const table = getDatabase().table('logs');
  let row = {log_id: uuid4(), log_data: data}; ⑤
  return table.insert(row);
};

const countLogEntries = () => {           ⑥
  const database = getDatabase();
  return database.run('SELECT COUNT(*) AS count FROM logs').then((data) => {
    let rows = data[0];
    return rows[0].toJSON().count.value;
  });
};

const getBodyAsString = (req) => {         ⑦
  let contentType = req.get('content-type');
  if (contentType == 'text/plain') {
    return req.body;
  } else if (contentType == 'application/json') {
    return req.body.data;
  }
};
```

```

    } else {
      return JSON.stringify(req.body);
    }
};

exports.echo = (req, res) => {
  let body = getBodyAsString(req);
  return Promise.all([
    createLogEntry('Echoing: ' + body),
    countLogEntries()
  ]).then((data) => {
    res.status(200).send({ echo: body, logRowCount: data[1] });
  });
};

```

- ➊ Start by importing our two new dependencies.
- ➋ This call creates a new Spanner client.
- ➌ getDatabase returns a handle to the Cloud Spanner database. Make sure to update these IDs to the IDs for your instance and database.
- ➍ createLogEntry is the function that actually logs the request data to a new row in the logs table.
- ➎ Here we use the UUID library to generate a new ID for the row.
- ➏ countLogEntries executes a query against our database to count the number of rows in the logs table.
- ➐ getBodyAsString is a helper function of the logic we used to have in our old echo function, to retrieve what should be echoed back.
- ➑ Since these two promises are independent (one adds a new row, another counts the number of rows), we can run them in parallel and return when the results are ready for both.

When you deploy and call this new function, you'll see that the logRowCount returned will continue to increase just as planned!

#### **Listing 12.17. Call the newly deployed function which displays the row count**

```

$ gcloud beta functions call echo --data '{"data": "This will be echoed!"}'
executionId: o571oa83hdvs
result: '{"echo":"This will be echoed!","logRowCount":"1"}'

$ gcloud beta functions call echo --data '{"data": "This will be echoed!"}'
executionId: o571yr41okz0
result: '{"echo":"This will be echoed!","logRowCount":"2"}'

```

If you go to the Cloud Spanner UI in the Cloud Console, you'll also see that the preview for your table will show the log entries created by these calls. Now that you've seen that your functions can talk to other Cloud APIs, it's time to change tracks a bit. If you're wondering whether this deployment process of relying on a Cloud Storage bucket for staging your code is a bit tedious, you're not alone. Let's look at another way to manage the code behind your functions.

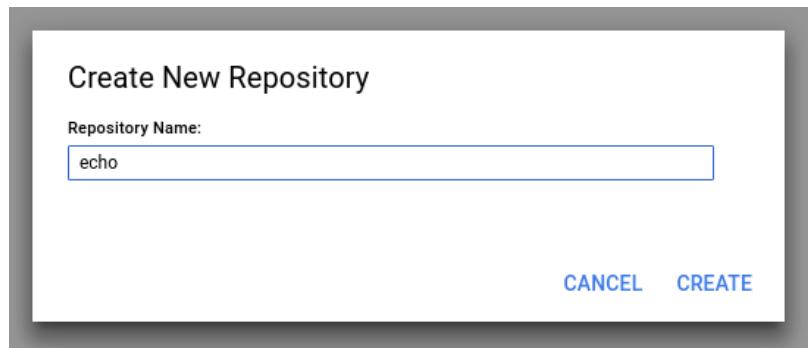
#### **12.4.5 Using a Google Source Repository**

As you learned, deploying a function that you've declared locally involves using the

Cloud SDK (`gcloud`) to package up your code files, upload them to a staging bucket on Cloud Storage, and then deploy from there. If you were hoping for a better way to manage and deploy your code, you're in luck.

Cloud Source Repositories are nothing more than a hosted code repository, a bit like a slimmed-down version of what's offered by GitHub, BitBucket, or GitLab. They also happen to be a place where you can store the code for your Cloud Functions. To see how these work, let's migrate our `echo` function from a local file into a hosted source repository, and then re-deploy from there. The first thing we need to do is create a new repository. You can do this from the Cloud Console by choosing "Source Repositories" from the left-side navigation (it's towards the bottom under the Tools section). Once there, you'll see a list of existing repositories (which should include a default repository), and clicking on the "Create Repository" button at the top will prompt you for a name. For this exercise, let's call this repository "echo".

**Figure 12.8. Create a new source repository**



Once the new repository is created, you'll see a few different ways to configure the empty repository, including the full URL that points to the newly created repository (something like [source.developers.google.com/projects/your-project-id-here/repos/echo](https://source.developers.google.com/projects/your-project-id-here/repos/echo)). There are helpers for common providers (e.g., mirroring a repository from GitHub), but to get started, let's clone our newly created (and empty) repository into the directory with our function and its dependencies. This process involves a couple of steps. First, we have to initialize our directory as a new git repository. After that, we need to configure some helpers to make sure authentication is handled by the Cloud SDK. Finally, we add a new remote endpoint to our git repository. Once we have that set up, we can push to the remote just like any other git repository.

#### **Listing 12.18. Initializing a new source repository with our code**

```
$ git init 1
Initialized empty Git repository in /home/jjg/echo/.git/

$ git remote add google \
  https://source.developers.google.com/projects/your-project-id-here/repos/echo 2

$ git config credential.helper gcloud.sh 3
```

```
$ git add index.js package.json          ④
$ git commit -m "Initial commit of echo package"
[master (root-commit) a68a490] Initial commit of echo package
 2 files changed, 60 insertions(+)
create mode 100644 index.js
create mode 100644 package.json

$ git push --all google                ⑤
Counting objects: 4, done.
Delta compression using up to 12 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 967 bytes | 0 bytes/s, done.
Total 4 (delta 0), reused 0 (delta 0)
remote: Approximate storage used: 57.1KiB/8.0GiB (this repository 967.0B)
To https://source.developers.google.com/projects/your-project-id-here/repos/echo
 * [new branch]      master -> master
```

- ① Start by initializing the current directory as a local git repository.
- ② Add the new source repository's URL as a git remote location.
- ③ Using git's configuration, tell it to use the Cloud SDK for authentication when interacting with the remote repository.
- ④ Add and commit our files to the git repository.
- ⑤ Finally, push all of our local changes to the new google remote that we created.

After that, if you go back to the Cloud Console and refresh the view of your source repository you should see all of the files you pushed listed there.

**Figure 12.9. Our newly pushed source repository**

## Source Code

The screenshot shows the Google Cloud Source Repository interface. At the top, there are dropdown menus for the repository name ('echo') and branch ('master'). Below that is a search bar with a dropdown arrow. The main area displays a list of files:

Name	Latest Commit	Author	Date (UTC-4)
index.js	Initial commit of echo package	JJ Geewax	8:22 AM
package.json	Initial commit of echo package	JJ Geewax	8:22 AM

At this point, the code for our function is officially stored on a Cloud Source Repository, which means that if we wanted to re-deploy the function we could use this repository as the source for the function. To do that, you can use the Cloud SDK (`gcloud`) once again but with slightly different parameters.

### **Listing 12.19. Deploying from the source repository**

```
$ gcloud beta functions deploy echo \
```

```
> --source=https://source.developers.google.com/projects/your-project-id-
here/repos/echo \ ①
> --trigger-http
Deploying function (may take a while - up to 2 minutes)...done.
availableMemoryMb: 256
entryPoint: echo
httpsTrigger:
  url: https://us-central1-your-project-id-here.cloudfunctions.net/echo
latestOperation:
operations/ampnLWNsb3VkJlc2VhcmNoL3VzLWNlbnRyYWwxL2VjaG8vendQSGFSVFR2Um8
name: projects/your-project-id-here/locations/us-central1/functions/echo
serviceAccount: your-project-id-here@appspot.gserviceaccount.com
sourceRepository:
  branch: master
  deployedRevision: a68a490928b8505f3be1b813388690506c677787
  repositoryUrl: https://source.developers.google.com/projects/your-project-id-
here/repos/echo
  sourcePath: /
status: READY
timeout: 60s
updateTime: '2017-05-23T12:30:44Z'

$ gcloud beta functions call echo --data '{"data": "This will be echoed!"}'
executionId: hp34ltbpibrk
result: '{"echo":"This will be echoed!","logRowCount":"5"}'
```

- ① Make sure you substitute your own project ID in this URL.

And that's it! You've just re-deployed from your source repository instead of your local file system! Now that you've seen what Cloud Functions is capable of, let's take a step back and look at how much all of this actually costs.

## 12.5 Understanding pricing

Following on the tradition of using Google Cloud Platform, Cloud Functions only charges you for what you actually use, and in this case it's actually incredibly granular. Unlike some of the other products, there are several different aspects that go into calculating the bill for your function, so let's go through them each, one at a time, and then we'll look at the perpetual free-tier where you will find that most hobbyist projects can run for free.

The first aspect also happens to be the most straight forward, which is the number of "invocations" (e.g., requests) sent to your function. This is measured in "millions of requests" and currently billed at \$0.40 USD per million, meaning each request costs \$0.0000004 just to run. The next aspect is common across all of Google Cloud Platform: networking cost. Across GCP, all inbound traffic, which in this case is the data sent to your function, is free of charge, however outbound traffic costs \$0.12 USD per GB. That is, any data generated by your function and sent back to requesters will be billed at this rate.

For the next two aspects of billing, compute time and memory time, it makes sense to combine them together to make things look a bit more like Compute Engine (for more

on GCE, see [chapter 9](#)). You may remember that when we deployed our function, there was an extra parameter that controls how much memory is given to the function for each request. What's left out of this is that the amount of memory you specify happens to also determine the amount of CPU capacity provided to your function. This means that you effectively have 5 different computing profiles to choose from, each with a different overall cost.

**Table 12.1. Cost of 1m requests, 100ms per request**

Memory	CPU	Price of 1m requests, 100ms each
128 MB	200 MHz	\$0.232
256 MB	400 MHz	\$0.463
512 MB	800 MHz	\$0.925
1024 MB	1.4 GHz	\$1.65
2048 MB	2.4 GHz	\$2.90

This is all based on a simple pricing formula, which looks specifically at the amount of memory and CPU capacity consumed in a given second.

#### **Listing 12.20. Formula for calculating the cost of 1m requests**

```
seconds consumed * ($0.0000100 * GHz configured + $0.0000025 * GB configured)
```

You can use this formula to calculate the cost of the smallest configuration (128 MB and 200 MHz):  $1,000,000 * 0.1s (0.2 \text{ GHz} * 0.0000100 + 0.0000025 * 0.128 \text{ GB}) = \$ 0.232$ . Hopefully you can see now why it's a bit easier to think in terms of configurations like a Compute Engine instance, and look at the overall cost for 1 million requests, each taking 100ms.

If things weren't confusing and complicated enough, to cap all of this off Cloud Functions comes with a perpetual free-tier, which means that some chunk of the resources you use are completely free. With Cloud Functions, this means that the following numbers represent "free-tier usage" and won't count towards your bill:

- Requests: the first 2 million requests per month
- Compute: 200,000 GHz-seconds per month
- Memory: 400,000 GB-seconds per month
- Network: 5GB of egress traffic per month

## **12.6 Summary**

- Micro-services allow you to build applications in separate stand-alone pieces of functionality.
- Cloud Functions are one way to deploy and run micro-services on Google Cloud Platform.
- There are two types of function handlers: synchronous and asynchronous (or background), where synchronous functions respond to HTTP requests.

- Functions register triggers which then pass along events from another service such as Cloud Pub/Sub.
- Cloud Functions allows you to write your function code in JavaScript, and manage dependencies just like you would for a typical Node.js application.

# 13

## *Cloud DNS: Managed DNS hosting*

**This chapter covers:**

- An overview and history of the Domain Name System (DNS)
- How the Cloud DNS API works
- How Cloud DNS pricing is calculated
- An example of assigning DNS names to VMs at start-up

### 13.1 What is DNS?

DNS is a hierarchical distributed storage system that keeps track of the mapping of Internet names (like `www.google.com`) to numerical addresses. In essence, DNS is the Internet's phone book, which as you can imagine is pretty large and rapidly changing. The system works by storing a set of "resource records", which are the actual mappings from names to numbers, and splits these records across a hierarchy of "zones". These zones act as a way to delegate responsibility for owning and updating subsets of records. This means that if you own the "zone" for `yourdomain.com`, you can easily control the records that might live inside that zone (such as, `www.yourdomain.com` or `mail.yourdomain.com`).

Resource records come in many flavors, sometimes pointing to specific numeric addresses (such as A or AAAA records), sometimes storing arbitrary data (such as TXT records), and other times storing aliases for other information (such as CNAME records). For example, an A record might say that `www.google.com` maps to 207.237.69.117, whereas a CNAME record might say that `storage.googleapis.com` maps to `storage.l.googleapis.com`. These records are like the entries in the phone book, directing people to the right place without them needing to memorize a long number.

Zones are specific collections of related records. These act as a way to delegate responsibility and ownership over certain groups of records from someone higher up the food chain to someone lower. In a sense, this is a bit like each company in the yellow pages being responsible for what shows up inside their individual box in the phone book. The main phone book is still the overall coordinator of all of the records and controls the overall layout of the book, but responsibility for certain areas (such as a box advertising for a local plumber) can be delegated to that company itself to fill that box with whatever content it wants.

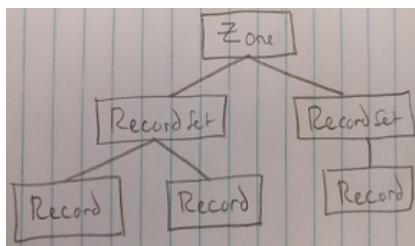
Since DNS is a distributed system and only expected to be eventually consistent (that is, data might be stale from time to time), anyone can set up a server to act as a cache of DNS records. It may not surprise you to learn that Google already does this with public-facing DNS servers at 8.8.8.8 and 8.8.4.4. Further, anyone is able to turn on their own DNS server (using a piece of software called BIND) and tell a registrar of domain names that the records for that domain name are stored on that particular server. However, as you might guess, running your own DNS server is a bit of a pain and happens to fall in the category of problems that Cloud services are best at fixing, which brings us to Google Cloud DNS.

## **13.2 What is Cloud DNS?**

Google Cloud DNS is a managed service that acts as a DNS server, able to answer DNS queries just like other servers such as BIND. One simple reason for using this service is to manage your own DNS entries, without running your own BIND server. Another more interesting reason for this service is to expose an API which makes it possible to manage DNS entries automatically. For example, with an API for managing DNS entries, you can configure virtual machines to automatically register a new DNS entry at boot time, giving you friendly names such as `server1.mydomain.com`. This is important because BIND, while battle tested over the years and proven to be quite reliable, is somewhat inconvenient to run and maintain and doesn't have a modern API to make changes to DNS records. Instead, updating records involves modifications to files on the machine running the BIND service followed by reloading the contents into the processes memory.

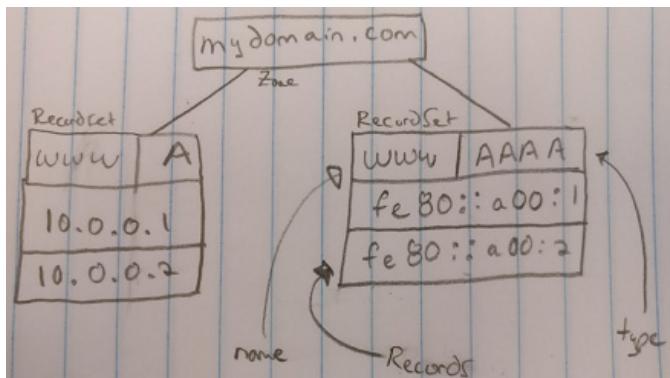
So how does Cloud DNS work? To start, just like the DNS system, Google Cloud DNS offers the same resources as BIND: zones (called "Managed Zones") and records (called "Resource Record Sets"). Each record set holds DNS entries just like they do in a true DNS server like BIND.

**Figure 13.1. Hierarchy of Cloud DNS concepts**

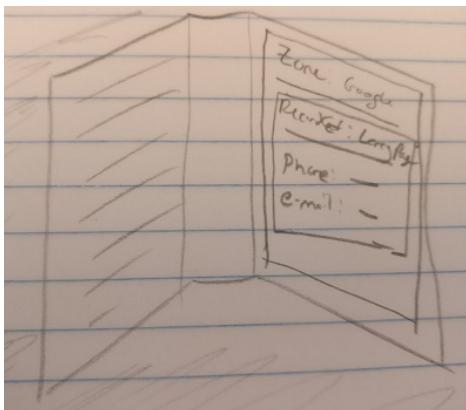


As you can see in the image above, each zone contains a collection of record sets, and each record set has a collection of records. These records are where the actual useful data is stored, whereas the other resources are focused on categorization of this data.

**Figure 13.2. Example hierarchy of DNS records**



While a zone is defined by nothing more than a name (e.g., `mydomain.com`), a record set stores both a name (e.g., `www.mydomain.com`), a "type" (such as A or CNAME), and a "time to live" (abbreviated as `ttl`) which instructs clients how long these records should be cached. This means that we have the ability to store multiple records for a single given sub-domain and type. For example, this structure allows you to store several different IP addresses for `www.mydomain.com` by setting multiple records in a record set with type A. This is similar to having multiple phone numbers listed for your business in the phone book.

**Figure 13.3. DNS records as a phone book**

Using the phone book analogy once again, a zone is like the section delegated to a company that was described before (e.g., Google, Inc), a record set is equivalent to a single person working at the company (e.g., Larry Page), and each record is a different contact method for the person (e.g., 2 phone numbers, an e-mail address, and a physical address).

### **13.2.1 Example DNS entries**

Let's look briefly at an example domain, `mydomain.com`, containing some sample records. In this example we have a name server (NS) record which is responsible for delegating ownership to other servers, a few "logical" (A or AAAA) records which point to IP addresses of a server, and a "canonical name" (CNAME) record which acts as an alias of sorts for the domain entry. As you can see in the table the domain has three distinct sub-domains: `ns1`, `docs`, and `www`, each entry with at least one record.

**Table 13.1. DNS entries by record set**

Zone	Sub-domain	Record set	Record
<code>mydomain.com</code>	<code>ns1</code>	A	<code>10.0.0.1</code>
	<code>www</code>	A	<code>10.0.0.1</code>
			<code>10.0.0.2</code>
	<code>docs</code>	CNAME	

In a regular DNS server like BIND, you manage these as "zone files", which are basically text files stating out in a special format the exact DNS records. The example below shows an equivalent BIND zone file to express these records.

#### **Listing 13.1. Example BIND zone file**

```
$TTL 86400 ; 24 hours could have been written as 24h or 1d
$ORIGIN mydomain.com.
@ 1D IN SOA ns1.mydomain.com. hostmaster.mydomain.com. (
    2002022401 ; serial
```

```

            3H ; refresh
            15 ; retry
            1w ; expire
            3h ; nxdomain ttl
        )
IN  NS      ns1.mydomain.com. ; in the domain

ns1  IN  A      10.0.0.1
www  IN  A      10.0.0.1
www  IN  A      10.0.0.2
docs  IN  CNAME  ghs.google.com.

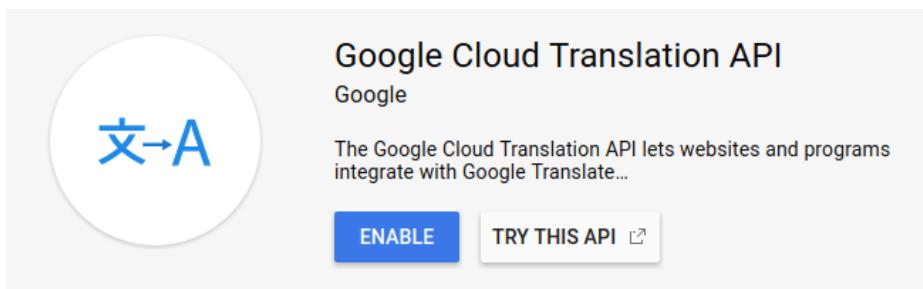
```

As you can see, exposing an API to update these remotely and then reloading the DNS server is a non-trivial amount of work. This is even more difficult if you want it to be always available, so having a service that does this for you would save quite a bit of time. Cloud DNS does exactly this, exposing zones and record sets as resources that you can create and manage. Let's look next at how this works.

### 13.3 Interacting with Cloud DNS

Since Cloud DNS is just an API that is ultimately equivalent to updating a BIND zone file and restarting the BIND server, let's go through an example that creates the example configuration described above. To begin, we have to enable the Cloud DNS API so let's head over to the Cloud Console. Type in "Cloud DNS API" in the search box at the top and you should see just one result in the list. After clicking on that you should land on a page with a big "Enable" button, shown below. After you click that, you should be good to go.

**Figure 13.4. Enable the Cloud DNS API**

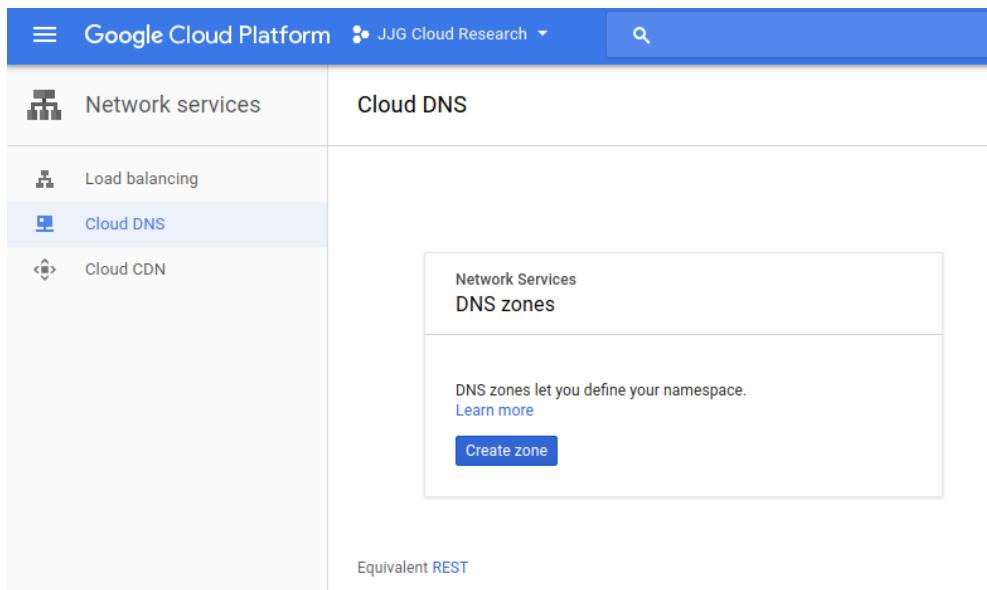


Now that the API is enabled, so let's continue using the UI to work with Cloud.

#### 13.3.1 Using the Cloud Console

Let's start our exploration of Cloud DNS by creating a zone. To do this, in the left-side navigation click on the entry called "Network services" under the Networking section. Once you do that, you should see a "Cloud DNS" item in the list on the left side of the page which will take you to the UI for Cloud DNS. This page will allow you to manage your zones and records for Cloud DNS. To start, let's create the zone for `mydomain.com`.

**Figure 13.5. Managing Cloud DNS entries from the UI**



You be wondering at this point how we're going to control the DNS records for a domain that we clearly don't own (since `mydomain.com` is taken). Remember the concept of delegation that we described above? In order for any records to be official (and discovered by anyone asking for the records of `mydomain.com`), a higher level authority needs to direct them to your records. This is done at the domain registrar level where you can set which name server to use for a domain that you currently own.

Since we definitely don't own `mydomain.com`, what we're doing now is basically like writing up an advertisement for the plumber in the yellow pages, but instead of sending it to the phone book to publish we'll just glue it into the phone book, meaning it'll only ever be seen by us. This means that we can do all of the work to set up DNS entries, and if you happen to own a domain, you can update your registrar to delegate its DNS records to Google Cloud DNS and then they will become official. Let's continue now that we understand that we're not creating anything official and instead are making a page for the phone book that will never be published.

Clicking "Create zone" will bring you to a form where you have to enter three different values: a unique ID for the zone, the domain name, and an optional description.

**Figure 13.6. Form to create a new zone**

[← Create a DNS zone](#)

---

A DNS zone is a container of DNS records for the same DNS name suffix. In Cloud DNS, all records in a managed zone are hosted on the same set of Google-operated authoritative name servers. [Learn more](#)

**Zone name** ?

**DNS name** ?

**DNSSEC** ?

**Description (Optional)**

**Create** **Cancel**

Equivalent [REST](#) or [command line](#)

You may be wondering why DNS asks for two different names. After all, what's the difference between a DNS name and a "Zone name"? Surprisingly, these two serve very different purposes. The zone name is a unique ID inside Google Cloud which is similar to a Compute Engine instance ID or a Cloud Bigtable instance ID. The DNS name is specific to the domain name system and refers to the sub-group of records that this zone will act as a delegate for. In our example, the DNS name will be `mydomain.com` which says that this zone will be responsible for every sub-domain of `mydomain.com` (such as `www.mydomain.com` or `anything.else.mydomain.com`). To create the example zone we described, let's use `mydomain-dot-com` as the zone name and `mydomain.com` as the DNS name as shown below.

**Figure 13.7. Creating our example zone**

← Create a DNS zone

---

A DNS zone is a container of DNS records for the same DNS name suffix. In Cloud DNS, all records in a managed zone are hosted on the same set of Google-operated authoritative name servers. [Learn more](#)

**Zone name** [?](#)  
mydomain-dot-com

**DNS name** [?](#)  
mydomain.com

**DNSSEC** [?](#)  
Off

**Description (Optional)**

**Create** **Cancel**

Equivalent [REST](#) or [command line](#)

Once you click "Create", you'll be brought to a screen that lets you manage the records for the zone. You may be surprised to see some record sets already in the list! Don't worry though — these records are the default (and necessary) NS records which simply state that there are no further delegations of zones. In other words, these records say that anything inside `mydomain.com` should be handled by Google Cloud DNS name servers (e.g., `ns-cloud-b1.googledomains.com`).

Let's continue by adding a demo record through the UI (one that wasn't in our list). First, click the "Add record set" button at the top of the page. This will expose a form to fill out with the DNS name of the record set (e.g., `demo.mydomain.com`), as well as a list of records (e.g., an A record of `192.168.0.1`), shown below. Note that to add more records to the set simply click "Add item" and a new box will appear to enter a new record.

**Figure 13.8. Add demo.mydomain.com A records**

mydomain-dot-com

mydomain.com.

Resource Record Sets

<input type="checkbox"/> DNS name ^	Type	TTL (seconds)	Data
mydomain.com.	NS	21600	ns-cloud-b1.googledomains.com. ns-cloud-b2.googledomains.com. ns-cloud-b3.googledomains.com. ns-cloud-b4.googledomains.com.
mydomain.com.	SOA	21600	ns-cloud-b1.googledomains.com. c

Add record set

DNS Name ①

Resource Record Type ②

A

TTL ③

5 minutes

IPv4 Address ④

192.168.0.1

192.168.0.2

+ Add item

Create Cancel

- ① Start by choosing a sub-domain to create a record set for.
- ② Next, choose the record type (in this case, a A record).
- ③ Finally, enter the IP addresses to store for this record.

Once you click create you will see the records added to the list, To check whether this worked we can actually make a regular DNS query for `demo.mydomain.com`, however we need to specify during the lookup that we are only interested in "our version" of this DNS record. In other words, we need to ask Google Cloud DNS directly rather than the global network. This is equivalent to pulling out our version of the plumber's phone book page from our file cabinet rather than looking it up in the real yellow pages.

To do this, we can use the Linux terminal utility called `dig`, aimed at a specific DNS server.

### Listing 13.2. Asking Google Cloud DNS for the records we just added

```
$ dig demo.mydomain.com @ns-cloud-b1.googledomains.com ①
# ... More information here ...

;; QUESTION SECTION:
;demo.mydomain.com.          IN      A
;; ANSWER SECTION:
demo.mydomain.com.    300    IN      A      192.168.0.1
demo.mydomain.com.    300    IN      A      192.168.0.2
```

- ① Make sure to use the right DNS server here. In the example above, this happens to be ns-cloud-b1.googledomains.com, but it could be something else for your project (e.g., ns-cloud-a1.googledomains.com).

As you can see, our two entries (192.168.0.1 and 192.168.0.2) are both there in the "answers section".

Note that if you were to ask globally for this entry (without the special @ns-cloud-b1.googledomains.com part of the command), you would see no answers resulting from the query.

### Listing 13.3. No results when asking for these entries in the global system

```
$ dig demo.mydomain.com
# ... More information here ...

;; QUESTION SECTION:
;demo.mydomain.com.          IN      A
;; AUTHORITY SECTION:
mydomain.com.        1799    IN      SOA      ns1.mydomain.com.
hostmaster.mydomain.com. 1335787408 16384 2048 1048576 2560
```

In order to make this "global" and get results for `dig demo.mydomain.com`, you'd need to own the domain name and update the DNS servers for the domain to be those shown in the NS section (e.g., ns-cloud-b1.googledomains.com). Now that we've tested this works, let's move on to accessing this API from inside Node.js so we can really benefit from the purpose of Cloud DNS.

#### **13.3.2 Using the Node.js client**

Before we get started writing some code to talk to Cloud DNS, you'll first need to install the Cloud DNS client library, which you can do by running `npm install @google-cloud/dns@0.6.1`. Next we call explore how exactly the Cloud DNS API works under the hood. Unlike some other APIs, the way we actually update records on DNS entries is using the concept of a "mutation" (called a **change** in Cloud DNS). The purpose behind this is to ensure that we can apply modifications in a transactional way.

Without this, it's possible that when applying two related or dependent changes (e.g., a new CNAME mapping along with the A record with an IP address), someone may end up seeing an inconsistent view of the world, which can be quite problematic. This means we'll create a few records and then use `zone.createChange` to apply changes to a zone, shown below.

#### **Listing 13.4. Adding new records to our zone**

```
const dns = require('@google-cloud/dns')({
  projectId: 'your-project-id'
});
const zone = dns.zone('mydomain-dot-com'); ①

const addRecords = [ ②
  zone.record('a', { ③
    name: 'www.mydomain.com.',
    data: '10.0.0.1',
    ttl: 86400
  }),
  zone.record('cname', {
    name: 'docs.mydomain.com.',
    data: 'ghs.google.com.',
    ttl: 86400
  })
];

zone.createChange({add: addRecords}).then((data) => { ④
  const change = data[0];
  console.log('Change created at', change.metadata.startTime,
    'as Change ID', change.metadata.id);
  console.log('Change status is currently', change.metadata.status);
});
```

- ① Start by creating a Zone object using the unique name (not DNS name) from before in the Cloud Console.
- ② Here we create a list of the records we're going to add.
- ③ We use the `zone.record` method to create a Cloud DNS Record, which contains the DNS name and the data. This also includes the TTL (time-to-live) which controls how this value should be cached by clients like web browsers.
- ④ Here we use the `zone.createChange` method to apply a mutation that adds our records defined above.

If you run this snippet, you should see output looking something like this.

#### **Listing 13.5. Output from creating changes to our zone**

```
> Change created at 2017-02-15T10:57:26.139Z as Change ID 6
Change status is currently pending
```

The change being in the "pending" state means that Cloud DNS is applying the mutation to the DNS zone, and usually completes in a few seconds. We can check on

whether these new records have been applied in the UI by simply refreshing the page, which should show our new records in the list.

**Figure 13.9. Newly added records in the Cloud DNS UI**

DNS name ^	Type	TTL (seconds)	Data	
mydomain.com.	NS	21600	ns-cloud-a1.googledomains.com. ns-cloud-a2.googledomains.com. ns-cloud-a3.googledomains.com. ns-cloud-a4.googledomains.com.	✎
mydomain.com.	SOA	21600	ns-cloud-a1.googledomains.com. cloud-dns-hostmaster.google.com. 1 21600 3600 259200 300	✎
demo.mydomain.com.	A	86400	192.168.0.1 192.168.0.2	✎
docs.mydomain.com.	CNAME	86400	ghs.google.com.	✎
www.mydomain.com.	A	86400	10.0.0.1	✎
www.mydomain.com.	AAAA	86400	fe80::a00:1	✎

### 13.3.3 Using the gCloud command-line

In addition to using the UI or the client library, we can also interact with our DNS records using the `gcloud` command-line tool, which has a `gcloud dns` sub-command. For example, let's look at the newly updated list of our DNS records for the `mydomain-dot-com` zone. Note that as mentioned before, when referring to a specific managed zone you use the Google Cloud unique name that we chose (`mydomain-dot-com`) and not the DNS name for the zone (`mydomain.com`).

#### Listing 13.6. Listing records for mydomain.com with gcloud

```
$ gcloud dns record-sets list --zone mydomain-dot-com
NAME          TYPE    TTL     DATA
mydomain.com.  NS      21600   ns-cloud-b1.googledomains.com.,ns-cloud-
b2.googledomains.com.,ns-cloud-b3.googledomains.com.,ns-cloud-b4.googledomains.com.
mydomain.com.  SOA     21600   ns-cloud-b1.googledomains.com. cloud-dns-
hostmaster.google.com. 1 21600 3600 259200 300
demo.mydomain.com. A      300    192.168.0.1,192.168.0.2
docs.mydomain.com. CNAME  86400   ghs.google.com.
www.mydomain.com. A      86400   10.0.0.1
```

This can be incredibly handy if you happen to have an existing BIND server that you want to move over to Cloud DNS, using the `gcloud dns` sub-command's `import` functionality.

#### IMPORTING BIND ZONE FILES

Let's say you have a BIND-style zone file with your existing DNS records for `mydomain.com`, an example of which is shown below. Notice that we've changed a few of the addresses involved, but the record names are all the same (`ns1`, `www`,

and docs).

#### **Listing 13.7. BIND zone file for mydomain.com (master.mydomain.com file)**

```
$TTL 86400 ; 24 hours could have been written as 24h or 1d
$ORIGIN mydomain.com.
@ 1D IN SOA ns1.mydomain.com. hostmaster.mydomain.com. (
    2002022401 ; serial
    3H ; refresh
    15 ; retry
    1w ; expire
    3h ; nxdomain ttl
)
IN NS      ns1.mydomain.com. ; in the domain

ns1  IN  A      10.0.0.91
www  IN  A      10.0.0.91
www  IN  A      10.0.0.92
docs IN  CNAME  new.ghs.google.com.
```

We can use the `import` command with a special flag to replace all of our DNS records in the managed zone with the ones in our zone file. To start, let's double check the current records.

#### **Listing 13.8. Listing current DNS records for mydomain-dot-com**

```
$ gcloud dns record-sets list --zone mydomain-dot-com
NAME          TYPE   TTL    DATA
mydomain.com.  NS     21600  ns-cloud-b1.googledomains.com.,ns-cloud-
                b2.googledomains.com.,ns-cloud-b3.googledomains.com.,ns-cloud-b4.googledomains.com.
mydomain.com.  SOA    21600  ns-cloud-b1.googledomains.com. cloud-dns-
hostmaster.google.com. 1 21600 3600 259200 300
demo.mydomain.com. A     300    192.168.0.1,192.168.0.2
docs.mydomain.com. CNAME  86400   ghs.google.com.
www.mydomain.com. A     86400   10.0.0.1
```

Now we can replace the records with the ones in our file above.

#### **Listing 13.9. Importing records from a zone file with gcloud**

```
$ gcloud dns record-sets import master.mydomain.com --zone mydomain-dot-com
> --delete-all-existing --replace-origin-ns --zone-file-format
Imported record-sets from [master.mydomain.com] into managed-zone [mydomain-dot-
com].
Created [https://www.googleapis.com/dns/v1/projects/your-project-id-
here/managedZones/mydomain-dot-com/changes/8].
ID  START_TIME           STATUS
8   2017-02-15T14:08:18.032Z pending
```

Just as before we can check on the status of this either by looking in the UI or using the `gcloud` command to "describe" the change, shown below.

**Listing 13.10. Viewing the status of our DNS change**

```
$ gcloud dns record-sets changes describe 8 --zone mydomain-dot-com | grep status
status: done
```

Since this says that our change has been applied, we can now look at our updated records with the `record-sets list` directive.

**Listing 13.11. Listing all record sets with gcloud**

```
$ gcloud dns record-sets list --zone mydomain-dot-com
NAME          TYPE    TTL     DATA
mydomain.com.  NS      86400   ns1.mydomain.com.
mydomain.com.  SOA     86400   ns-cloud-b1.googledomains.com.
hostmaster.mydomain.com. 2002022401 10800 15 604800 10800
docs.mydomain.com. CNAME   86400   new.ghs.google.com.
ns1.mydomain.com. A       86400   10.0.0.91
www.mydomain.com. A       86400   10.0.0.91,10.0.0.92
```

Notice that the `10.0.0.1` entries have changed to `10.0.0.91` as described in our zone file. Now that you've seen how to interact with Cloud DNS, let's look at what this will actually cost.

## **13.4 Understanding pricing**

As with most things in Google Cloud, Cloud DNS only charges for the resources and capacity that you use. In this case, the two factors to look at are the number of managed zones and the number of DNS queries handled.

While there is a tiered pricing table, at most you'll end up paying 20 cents per managed zone per month, and 40 cents per million queries per month. As you have more zones and more queries, the per-unit prices go down pretty dramatically. For example, while your first billion queries will be billed at 40 cents per million, after that queries are billed at 20 cents per million. Further, while your first 25 managed zones cost 20 cents each, after that the per-unit price drops to 10 cents, and then 3 cents for every zone more than 100,000. To make this more concrete, let's look at two examples: personal DNS hosting and a start-up businesses DNS hosting.

### **13.4.1 Personal DNS hosting**

In a typical personal configuration, you typically see no more than 10 different domains being managed. In addition, it'd be surprising if these 10 websites each got more than 1 million monthly unique visitors. This brings our total to 10 zones and 10 million DNS queries per month.

**NOTE**

The **unique** part is important because it's likely that other DNS servers will cache the results, meaning a DNS query usually only happens on the first visit. This will also depend on the TTL values in your DNS records.

**Table 13.2. Personal DNS pricing summary**

Resource	Count	Unit cost	Cost
1 managed zone	10	\$0.20	\$2.00
1 million DNS queries	10	\$0.40	\$4.00
<b>Total</b>			<b>\$6.00 per month</b>

So what about a more "professional" situation, such as a start-up that needs DNS records for various VMs and other services.

### 13.4.2 Start-up business DNS hosting

In a typical start-up, it's not uncommon to have 20 different domains floating around to cover issues like separating user-provided content from the main service domain, vanity domain redirects, etc. In addition, it's likely that the traffic to the various domains may have quite a few different unique users, and may have shorter TTL values to allow modifications to propagate more quickly, resulting in more overall DNS queries. In this situation it's possible to have upwards of 50 million monthly DNS queries to handle. Let's estimate that this will come out to 20 zones and 50 million DNS queries per month.

**Table 13.3. Personal DNS pricing summary**

Resource	Count	Unit cost	Cost
1 managed zone	20	\$0.20	\$4.00
1 million DNS queries	50	\$0.40	\$20.00
<b>Total</b>			<b>\$24.00 per month</b>

As you can see, this should end up being "rounding error" in most businesses, and the all-in cost of running a DNS server of your own is likely to be far higher than the cost of managing zones using Cloud DNS. Now that we've gone through pricing, let's take a look at an example of how we might set up our VMs to register themselves with our DNS provider when they first boot up, so that when we turn on VMs we can access them using a custom domain name.

## 13.5 Case study: Giving machines DNS names at boot

If you're not familiar with Google Compute Engine yet, now may be a good time to head back and look at [chapter 2](#) or [chapter 9](#) which walk you through how Compute Engine works. Regardless, you should be able to follow along with this example without needing to understand the details of Compute Engine. In many Cloud computing environments, when a new virtual machine comes to life it's given some public-facing name so that you can access it from wherever you are (after all, the

computer in front of you isn't in the same data center). Sometimes this is just a public-facing IP address (e.g., 104.14.10.29), and other times it's a special DNS name (e.g., ec2-174-32-55-23.compute-1.amazonaws.com), but the point is that you have some public-facing way to talk to the machine.

However, both of those examples are not all that pretty, and definitely difficult to remember. Wouldn't it be nice if we could make sure that as new servers came up, we could talk to them with a name that was part of our domain (e.g., mydomain.com). For example, it'd be nice if new web servers would automatically turn on and register themselves as something like web7-uc1a.mydomain.com. As you learned throughout the chapter, this is a great use of Cloud DNS which exposes an API to interact with DNS records. To do this, we'll need a few different pieces of metadata about our machine:

1. The instance name (e.g., instance-4)
2. The Compute Engine zone (e.g., us-central1-a)
3. The public-facing IP address (e.g., 104.197.171.58)

To do this we'll rely on Compute Engine's metadata service, described in [chapter 9](#). Let's write a helper function that will return an object with all of this metadata.

#### **Listing 13.12. Defining the helper methods to get instance information**

```
const request = require('request');

const metadataUrl = 'http://metadata.google.internal/computeMetadata/v1/';
const metadataHeader = {'Metadata-Flavor': 'Google'};

const getMetadata = (path) => {
  const options = {
    url: metadataUrl + path,
    headers: metadataHeader
  };
  return new Promise((resolve, reject) => {
    request(options, (err, resp, body) => {
      resolve(body) ? err === null : reject(err);
    });
  });
};

const getInstanceName = () => {
  return getMetadata('instance/name');
};

const getInstanceZone = () => {
  return getMetadata('instance/zone').then((data) => {
    const parts = data.split('/');
    return parts[parts.length-1];
  });
};

const getInstanceIp = () => {
  const path = 'instance/network-interfaces/0/access-configs/0/external-ip';
  return getMetadata(path);
};
```

```
const getInstanceDetails = () => {
  const promises = [getInstanceName(), getInstanceZone(), getInstanceIp()];
  return Promise.all(promises).then((data) => {
    return {
      name: data[0],
      zone: data[1],
      ip: data[2]
    };
  });
};
```

If we try running our helper method (`getInstanceDetails()`) from a running GCE instance, we should see output looking something like the following.

#### **Listing 13.13. Output from our helper method**

```
> getInstanceDetails().then(console.log);
Promise { <pending> }
> { name: 'instance-4',
  zone: 'us-central1-f',
  ip: '104.197.171.58' }
```

Now let's write a quick start-up script that uses this metadata to automatically register a friendly domain name.

#### **Listing 13.14. Start-up script to register with DNS**

```
const dns = require('@google-cloud/dns')({
  projectId: 'your-project-id'
});
const zone = dns.zone('mydomain-dot-com');

getInstanceDetails().then((details) => {
  return zone.record('a', {
    name: [details.name, details.zone].join('-') + '.mydomain.com.',
    data: details.ip,
    ttl: 86400
  });
}).then((record) =>{
  return zone.createChange({add: record});
}).then((data) => {
  const change = data[0];
  console.log('Change created at', change.metadata.startTime,
             'as Change ID', change.metadata.id);
  console.log('Change status is currently', change.metadata.status);
});
```

After running this, you should see output showing that the change is being applied.

#### **Listing 13.15. Output of running our start-up script**

```
Change created at 2017-02-17T11:38:04.829Z as Change ID 13
Change status is currently pending
```

We can then check that the change was applied using the `getRecords()` method.

#### **Listing 13.16. Listing all DNS records for our zone**

```
> zone.getRecords().then(console.log)
Promise { <pending> }
> [ [ Record {
    zone_: [Object],
    type: 'NS',
    metadata: [Object],
    kind: 'dns#resourceRecordSet',
    name: 'mydomain.com.',
    ttl: 86400,
    data: [Object] },
  Record {
    zone_: [Object],
    type: 'SOA',
    metadata: [Object],
    kind: 'dns#resourceRecordSet',
    name: 'mydomain.com.',
    ttl: 86400,
    data: [Object] },
  Record {
    zone_: [Object],
    type: 'CNAME',
    metadata: [Object],
    kind: 'dns#resourceRecordSet',
    name: 'docs.mydomain.com.',
    ttl: 86400,
    data: [Object] },
  Record {
    zone_: [Object],
    type: 'A',
    metadata: [Object],
    kind: 'dns#resourceRecordSet',
    name: 'instance-4-us-central1-f.mydomain.com.',
    ttl: 86400,
    data: [Object] },
  Record {
    zone_: [Object],
    type: 'A',
    metadata: [Object],
    kind: 'dns#resourceRecordSet',
    name: 'ns1.mydomain.com.',
    ttl: 86400,
    data: [Object] },
  Record {
    zone_: [Object],
    type: 'A',
    metadata: [Object],
    kind: 'dns#resourceRecordSet',
    name: 'www.mydomain.com.',
    ttl: 86400,
    data: [Object] } ] ]
```

- ➊ Here you can see that our record was applied properly!

Finally, we should verify that this worked from the perspective of a DNS consumer. To do that, we can use the `dig` command just like before, specifically checking for our record. Note that you can do this from any computer (and it might be best to test this from outside your GCE VM as the goal is to be able to find your VM easily from the outside world).

#### **Listing 13.17. Viewing our newly created (non-authoritative) DNS record**

```
$ dig instance-4-us-central1-f.mydomain.com @ns-cloud-b1.googledomains.com

; <>> DiG 9.9.5-9+deb8u9-Debian <>> instance-4-us-central1-f.mydomain.com @ns-
cloud-b1.googledomains.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 60458
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
;; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;instance-4-us-central1-f.mydomain.com. IN A

;; ANSWER SECTION:
instance-4-us-central1-f.mydomain.com. 86400 IN A 104.197.171.58

;; Query time: 33 msec
;; SERVER: 216.239.32.107#53(216.239.32.107)
;; WHEN: Fri Feb 17 11:42:36 UTC 2017
;; MSG SIZE rcvd: 82
```

As we discussed before, these records won't be authoritative until the registrar that controls which name servers are in control actually point to Cloud DNS, so to make this work for real you'll have to update your domain settings. After you do that, you won't need the `@ns-cloud-b1.googledomains.com` part and everything should automatically work. Once that's done, you can use this as a start-up script for your VMs and they will register themselves in Cloud DNS once the boot process is completed.

## **13.6 Summary**

- DNS is a hierarchical storage system for tracking pointers of human-readable names to computer-understandable addresses.
- Cloud DNS is a hosted, highly-available set of DNS servers with an API that we can program against.
- Cloud DNS charges prices based on the number of "zones" (domain names) as well as the number of DNS look-up requests.

# Part 1 Machine learning

One of the most exciting areas of research today is the world of machine learning and artificial intelligence, so it should be no surprise that Google has invested quite a lot to make sure that ML "just works" on Google Cloud Platform.

In this section, we'll dig into the high level APIs available to cover some of the more traditional machine learning problems (e.g., identifying things in photographs or translating text between languages) and then we'll finish up by looking at generalized machine learning using TensorFlow and Cloud Machine Learning Engine to build your own ML models in the cloud.

# 14

## *Cloud Vision: Image recognition*

### **This chapter covers:**

- An overview of image recognition
- The different types of recognition supported by Cloud Vision
- How Cloud Vision pricing is calculated
- An example evaluating whether profile images are acceptable

## **14.1 What is image recognition?**

Image recognition to humans is one of those things that's easy to understand, but difficult to define. We can ask toddlers "What's this picture of?" and get an answer, but asking "Explain to me what it means to recognize an image?" will probably get a blank stare. To move into a slightly more philosophical area, you might say that we know what it means to "understand an image," but tough to explain clearly what exactly constitutes that understanding.

This is one of the many reasons why it's difficult to get a computer to "recognize" an image. Things that are difficult to define clearly are typically difficult to express as code, and "understanding an image" tends to fall in that category. As with many definition problems, we get around this by choosing a very specific definition and sticking to that. In the case of Cloud Vision, we're going to look at "image recognition" as being able to slap a bunch of annotations on a given image where each annotation covers a visual area and provides some structured context about the region.

**Figure 14.1. Vision as annotations**



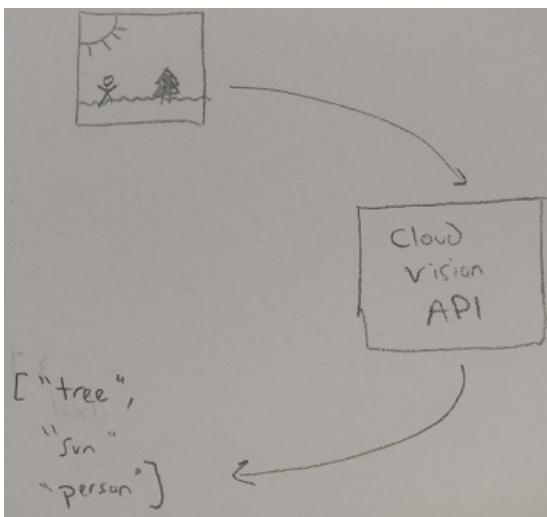
For example, [Figure 14.1](#) shows how a human might label an image, adding several annotations to different areas of the image. Notice that the annotations are not limited to *things* like "dog", but can be other attributes such as colors like "green". Often, the complexity is subtle and can be very frustrating. For example, it's easy for us to recognize a mirror, but since a mirror shows itself by duplicating whatever else is in the picture, in order to recognize a mirror we need to understand that it's not two dogs in the picture, but one dog and a mirror.

This difficulty isn't limited to conceptual understanding. You might recall a big argument on the internet over the color of a dress, with a pretty even split between white and gold or blue and black. This is a case where millions of people couldn't decide on the color of a dress by looking at a picture. All of this should go to show two things: image recognition is super complicated and therefore somewhat amazing, and image recognition is not at all an exact science. The first should make you glad that someone else is solving this problem and the second should encourage you to build some "fudge factor" into your code, taking the results of a particular annotation as a *suggestion* rather than absolute fact. With that said, let's look at how you can actually use the Cloud Vision API to start "recognizing" (or "annotating") images.

## 14.2 Annotating images

The general flow for annotating images is a simple request-response pattern, where you send an image along with the desired annotations you're interested in to the Cloud Vision API, and the API will send back a response containing all of those annotations. Unlike some of the other APIs we've explored so far, this one is entirely stateless, which means that you don't have to create anything before using it. Instead you can just send your image and get back some details about it.

**Figure 14.2. Request-response flow for Cloud Vision**



Since there's no state to maintain, all you'll have to do when sending your image is specify which annotation types you're interested in, and the result will be limited to just those. That said, there may be extra knobs you can turn for each type of annotation, but we'll explore these one at a time. Since we've already given a few examples of label annotations, let's start there.

### 14.2.1 Label annotations

In short, labels are nothing more than a quick textual description of a concept that Cloud Vision recognized in the image. As you learned, these labels aren't limited to the *physical* things found in an image and can be many other concepts. Additionally, it's important to remember that image recognition is not an exercise leading to absolute facts. What looks like a tree to you may look like a telephone pole to the algorithm, so in general it's best to treat the results as suggestions to be validated later by a human. Let's start by looking at some code that asks the Cloud Vision API to put label annotations on our image. To make this work, we'll first need to set up a service account and download the credentials.

**NOTE**

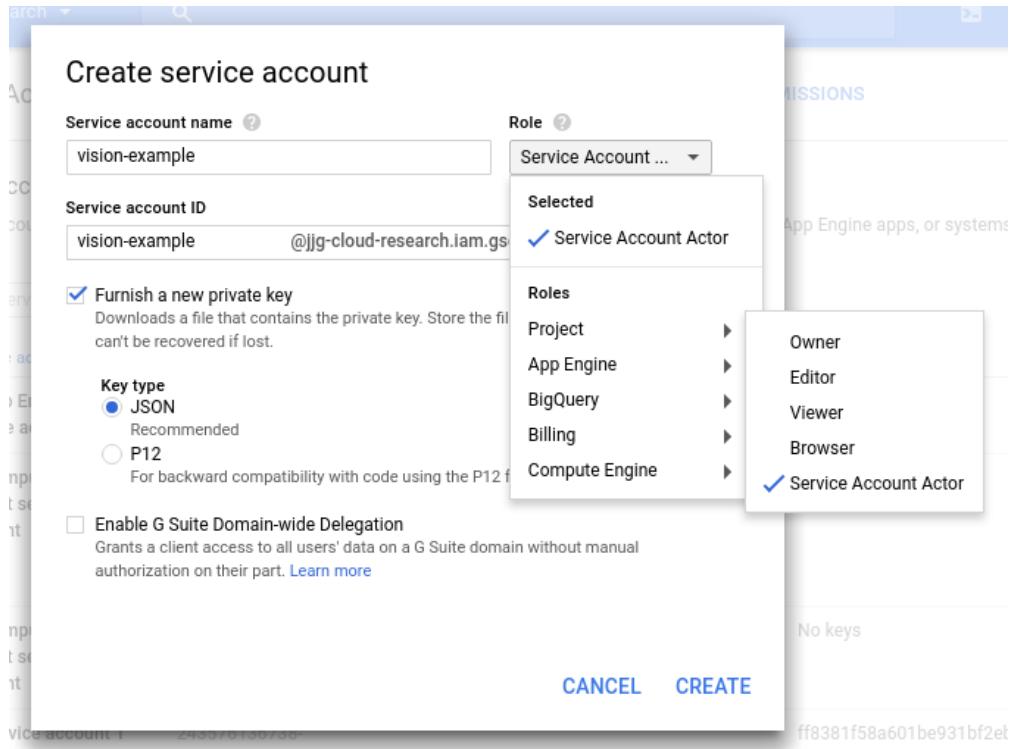
If you skip this part and instead try to use the credentials you got by running `gcloud auth login`, you'll see an error about the API not being enabled.

This is a tricky problem with the scope of the OAuth 2.0 credentials, where the request won't pass along your project and instead uses a shared project. For now, all you really need to know is that you should use a service account.

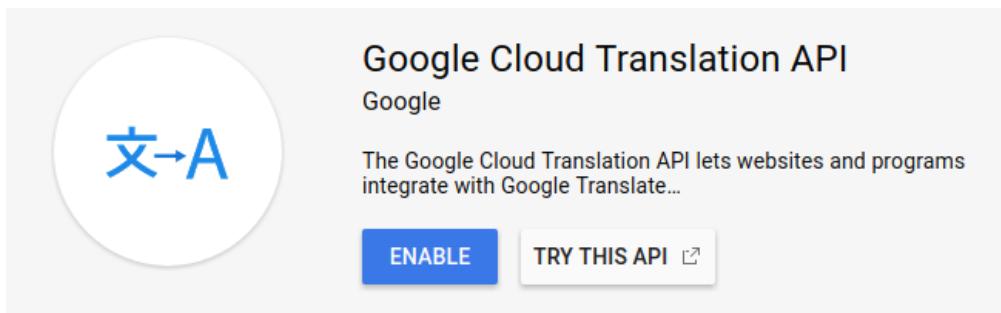
To get a service account, in the Cloud Console choose "IAM & Admin" from the left side navigation, and then choose "Service Accounts". Click "Create Service Account", and fill in some details as you can see in Figure 14.3. Make sure to choose "Service Account Actor" as the role which limits what this particular service account can do.

Also, don't forget to acquire a new private key (by checking the box). Once you click "Create", you'll see that a download automatically starts with a .json file. It's this file that we'll refer to as `key.json` in the following examples.

**Figure 14.3. Create a new service account**



Once you have your key file, we just need to install the client library. To do this, we can just run `npm install @google-cloud/vision@0.11.5` which will pull down a specific version of the Node.js client library. Next, we'll need to make sure to enable the Cloud Vision API using the Cloud Console. To do this, in the search bar at the top of the Cloud Console, simply type in "Cloud Vision API" and there should be a single result. Click on that result and on the next page just click "Enable" and you should be good to go.

**Figure 14.4. Enable the Cloud Vision API**

Now that that's done, we can move onto actually recognizing an image. In this example, we'll use the dog image from above, saved as `dog.jpg`, to see what labels the Cloud Vision API comes up with.

**Listing 14.1. Recognizing entities in an image of a dog**

```
const vision = require('@google-cloud/vision')({
  projectId: 'your-project-id',          ①
  keyFilename: 'key.json'                 ②
});

vision.detectLabels('dog.jpg').then((data) => { ③
  console.log('labels: ', data[0].join(', '));
});
```

- ① Make sure to specify your project ID here.
- ② In this case, you'll need to point to the service account key file that you downloaded before.
- ③ Use the `detectLabels` method to get label annotations on the image.

If you run this, you should see the following output:

**Listing 14.2. Cloud Vision sees a dog as we'd expect**

```
> labels:  dog, mammal, vertebrate, setter, dog like mammal
```

Obviously it seems like those labels go from specific to vague, so if you want more than one, you can go down the list. But what if you want to only use labels that have a certain "confidence level"? In other words, what if you wanted to ask Cloud Vision, "Only show me labels that you're 75% confident in"? In these situations, you can turn on "verbose mode" which will show you lots of other details about the image and the annotations. Let's look at the output of a "verbose" label detection.

**Listing 14.3. Enable verbose mode to get more information about labels detected**

```
const vision = require('@google-cloud/vision')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
```

```

});  

vision.detectLabels('dog.jpg', {verbose: true}).then((data) => { ①
  const labels = data[0];
  labels.forEach((label) => {
    console.log(label); ②
  });
});
}

```

- ① Notice the {verbose: true} modifier in the detectLabels call.
- ② Go through each label (which is an object) and print it out.

When you run this code, you should see something more detailed than just the label value.

#### **Listing 14.4. Verbose output includes a score for each label**

```

> { desc: 'dog', mid: '/m/0bt9lr', score: 96.969336 }
{ desc: 'mammal', mid: '/m/04rky', score: 92.070323 }
{ desc: 'vertebrate', mid: '/m/09686', score: 89.664793 }
{ desc: 'setter', mid: '/m/039nnd', score: 69.060057 }
{ desc: 'dog like mammal', mid: '/m/01z5f', score: 68.510407 }

```

They are the same label values, but they also include two extra fields: `mid` and `score`. The `mid` value is an opaque ID for the label which you should store if you intended to save these. The `score` is a confidence level for each label, giving you some indication of how confident the Vision API is in each label being "accurate". In our example of looking for only things with 75% confidence or above, our code to do this might look something like this.

#### **Listing 14.5. Only show labels with a score of 75% or higher**

```

const vision = require('@google-cloud/vision')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});  

vision.detectLabels('dog.jpg', {verbose: true}).then((data) => {
  const labels = data[0]
    .filter((label) => { return label.score > 75; }) ①
    .map((label) => { return label.desc; }); ②
  console.log('Accurate labels:', labels.join(', '));
});

```

- ① First, use JavaScript's `.filter()` method to remove any labels with low scores.
- ② Next, keep around only the description rather than the whole object.

After running this, we should see only the labels with confidence greater than 75%, which turns out to be dog, mammal, and vertebrate.

**Listing 14.6. Output of only high scoring labels**

```
> Accurate labels: dog, mammal, vertebrate
```

Now that you understand labels, let's take a step further into image recognition and look at detecting faces in images.

### **14.2.2 Faces**

Detecting faces, in many ways, is a lot like detecting labels, however rather than just getting "what's in this picture", you'll get details about faces in the image as well as specifics about where each face is as well as where facial feature is (e.g., the left eye is at this position). Further, you're also able to discover details about the emotions of the face in the picture which includes things like happiness, anger, and surprise. As well as other facial attributes such as whether or not the person is wearing a hat, whether the image is blurred, and the tilting of the image.

As with the other image recognition aspects, many of these things are expressed as scores, confidences, and likelihoods. This is because, as we mentioned earlier, even we don't know for sure whether someone is sad in an image (perhaps they're just pensive). The API will express how similar a facial expression is to others where it was trained that those were sad. Let's start with a simple test to detect whether or not an image has a face. For example, you may be curious if the dog image from before counts as a "face" or whether the Vision API only considers humans.

**Listing 14.7. Detecting whether or not a face is in an image of a dog**

```
const vision = require('@google-cloud/vision')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

vision.detectFaces('dog.jpg').then((data) => { ①
  const faces = data[0];
  if (faces.length) {
    console.log("Yes! There's a face!");
  } else {
    console.log("Nope! There's no face in that image.");
  }
});
```

And when you run this little snippet, you'll see that the dog's face doesn't count.

**Listing 14.8. The dog doesn't appear to have a face**

```
> Nope! There's no face in that image.
```

Well that was boring. Let's try looking at an actual face and all the various annotations that come back on the image. Here's a picture that I think looks like it has a face and seems pretty happy to me.

**Figure 14.5. A happy kid (kid.jpg)**



Here we'll look at the image of what we think is a happy kid, and check whether the Cloud Vision API agrees with our opinion.

#### **Listing 14.9. Detecting a face and aspects about that face**

```
const vision = require('@google-cloud/vision')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

vision.detectFaces('kid.jpg').then((data) => {
  const faces = data[0];
  faces.forEach((face) => { ①
    console.log('How sure are we that there is a face?', face.confidence + '%');
    console.log('Does the face look happy?', face.joy ? 'Yes' : 'No');
    console.log('Does the face look angry?', face.anger ? 'Yes' : 'No');
  });
});
```

① Here we use the joy and anger attributes of the face to see

And when you run this little snippet, you'll see that the dog's face doesn't count.

#### **Listing 14.10. Confidence of a face in the image, along with whether it is happy or angry**

```
> How sure are we that there is a face? 99.97406%
Does the face look happy? Yes
Does the face look angry? No
```

But wait — those look like absolute certainty for the emotions. I thought we said that there'd be only likelihoods and not certainties. In this case, it happens that the `@google-cloud/vision` client library for Node.js is making some assumptions for you, basically saying "If the likelihood is `LIKELY` or `VERY_LIKELY`, then just use `true`." If you want to be more specific and only take the **highest** confidence level, you can look specifically at the API response to see the details. Here's an example where we

only want to say with certainty that the kid is happy if it's **very** likely.

#### **Listing 14.11. Enforce more strictness about whether a face is happy or angry**

```
const vision = require('@google-cloud/vision')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

vision.detectFaces('kid.jpg').then((data) => {
  const rawFaces = data[1]['responses'][0].faceAnnotations; ①
  const faces = data[0];

  faces.forEach((face, i) => {
    const rawFace = rawFaces[i]; ②
    console.log('How sure are we that there is a face?', face.confidence + '%');
    console.log('Are we certain the face looks happy?', 
      rawFace.joyLikelihood == 'VERY_LIKELY' ? 'Yes' : 'Not really'); ③
    console.log('Are we certain the face looks angry?', 
      rawFace.angerLikelihood == 'VERY_LIKELY' ? 'Yes' : 'Not really');
  });
});
```

- ① We can grab the `faceAnnotations` part of the response in our `data` attribute.
- ② The faces should be in the same order, so `face 1` is face annotation 1.
- ③ We need to look specifically at the `joyLikelihood` attribute and check that its value is `VERY_LIKELY` (and not just `LIKELY`).

After running this, it turns out that the likelihood of the face being joyful turns out to be `VERY_LIKELY`, so the API is very confident that this is a happy kid (which I happen to agree with).

#### **Listing 14.12. Show whether a face is happy or angry**

```
> How sure are we that there is a face? 99.97406005859375%
Are we certain the face looks happy? Yes
Are we certain the face looks angry? Not really
```

Let's move onto a somewhat more boring aspect of computer vision: recognizing text in an image.

### **14.2.3 Text recognition**

Text recognition (sometimes called OCR for "optical character recognition") first became popular when desktop image scanners came on the scene. People would scan documents and be left with an image of the document, which was better than nothing, however they wanted to be able to edit that document in a word processor. To make that happen, many companies found a way to recognize the actual words and convert the document from an image to text that you could treat like any other electronic document. While you might not use the Cloud Vision API to recognize a scanned document, it can be very helpful when you're shopping at the store and want to

recognize the text on the label. Let's try doing just this to get an idea of how image recognition works. Figure 14.6 shows a picture of a bottle of wine made by "Brooklyn Cowboy Winery".

**Figure 14.6. The label from a bottle of wine made by Brooklyn Cowboy Winery**



Let's see what the Cloud Vision API detects when we ask it to detect the text.

#### **Listing 14.13. Detecting text from an image**

```
const vision = require('@google-cloud/vision')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

vision.detectText('wine.jpg').then((data) => {
  const textAnnotations = data[0];
  console.log('The label says:', textAnnotations[0].replace(/\n/g, ' ')); ②
});
```

- ① Use the `detectText()` method to find text in our image.
- ② Replace all newlines in the text with spaces to make it easy to print.

If you run this code, you should see friendly output that says:

#### **Listing 14.14. Output from detected wine**

```
> The label says: BROOKLYN COWBOY WINERY
```

As with all the other types of image recognition, the Cloud Vision API will do its best to find text in an image and turn it into actual text, however it isn't perfect as there always seems to be some subjective aspect to putting text together to be useful.

Greeting cards are a great example, so let's see what happens with a particularly interesting card.

**Figure 14.7. Thank you card**



As humans it's pretty easy for us to understand what's written on this card ("Thank you so much" from "Evelyn and Sebastian") but for a computer, there are a few difficult aspects to this card. First, the text is in a long-hand font with lots of flourishes and overlaps. Second, the "so" is in a bit of a weird position, sitting about a half-line down below the "Thank you" and the "much". In cases like this, even if a computer can recognize the text in the fancy font, the order of the words is something that takes more than recognizing text and is really about understanding that "thank so you much" isn't quite right, and the artist must have intended to have three distinct lines of text: "thank you", "so", and "much". Let's look at the raw output from the Cloud Vision API when trying to understand this image.

#### **Listing 14.15. Looking at raw response from the Vision API**

```
const vision = require('@google-cloud/vision')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

vision.detectText('card.png', {verbose: true}).then((data) => { ①
  const textAnnotations = data[0];
  textAnnotations.forEach((item) => {
    console.log(item);
  });
});
```

① We turn on "verbose mode" here to include the bounding box coordinates you see below.

In this case, it turns out that the Vision API can only understand the "Evelyn & Sebastian" text at the bottom, and doesn't find anything else in the image.

#### **Listing 14.16. Details about text detected including bounding boxes**

```
> { desc: 'EVELYN & SEBASTIAN\n',
  bounds:
```

```
[ { x: 323, y: 357 },
  { x: 590, y: 357 },
  { x: 590, y: 379 },
  { x: 323, y: 379 } ] }

{ desc: 'EVELYN',
  bounds:
    [ { x: 323, y: 357 },
      { x: 418, y: 357 },
      { x: 418, y: 379 },
      { x: 323, y: 379 } ] }

{ desc: '&',
  bounds:
    [ { x: 427, y: 357 },
      { x: 440, y: 357 },
      { x: 440, y: 379 },
      { x: 427, y: 379 } ] }

{ desc: 'SEBASTIAN',
  bounds:
    [ { x: 453, y: 357 },
      { x: 590, y: 357 },
      { x: 590, y: 379 },
      { x: 453, y: 379 } ] }
```

Hopefully what you've learned from these two examples is that understanding images is complicated and computers aren't quite to the point where they perform better than humans. That said, if you have well defined areas of text (and not text that appears more artistic than informational) the Cloud Vision API can do a good job of turning that into usable text content. Let's dig into another area of image recognition that is a bit unique by trying to recognize some popular logos.

#### 14.2.4 Logo recognition

As you have certainly noticed in the world, logos often tend to be combinations of text and art which can be tricky to find. Sometimes a detecting text will come up with the right answer (e.g., if you tried to run text detection on the Google logo, you'd likely come up with the right answer), but other times it might not work so well. This could be for a variety of reasons such as the logo looking a bit like the "thank you" card we saw above, or from the logo not including text at all (e.g., Starbucks' or Apple's logos). Regardless of the difficulty of detecting a logo, you may one day find yourself in the unenviable position of needing to "take down" images that contain copyrighted or trademarked material (and logos fall in this area of covered intellectual property).

This is where logo detection in the Cloud Vision API comes in. Given an image, it can often find and identify popular logos independent of whether they contain the name of the company in the image. Let's go through a couple of quick examples, starting with the easiest one:

**Figure 14.8. FedEx's logo**



We can detect this similar to how we detected labels and text.

#### **Listing 14.17. Script to detect a logo in an image**

```
const vision = require('@google-cloud/vision')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

vision.detectLogos('logo.png').then((data) => {
  const logos = data[0];
  console.log('Found the following logos:', logos.join(', '));
});
```

① We turn on "verbose mode" here to include the bounding box coordinates you see below.

In this case, we find the right logo as expected:

#### **Listing 14.18. Output of the logos detected**

```
> Found the following logos: FedEx
```

Let's run the same code again with a more complicated logo:

**Figure 14.9. Tostitos' logo**



Running the same code again on this logo figures out which it was!

#### **Listing 14.19. Output of the logos detected with a different source image**

```
> Found the following logos: Tostitos
```

But what about a logo with no text and just an image?

**Figure 14.10. Starbucks' logo**



If we run the same code yet again, we get the expected output:

**Listing 14.20. Output of a logo including no text**

> Found the following logos: Starbucks

Finally, let's look at an image containing many logos:

**Figure 14.11. Pizza Hut and KFC next to each other**



In this case, running our logo detector will come out with two results:

**Listing 14.21. Output of detecting multiple logos**

> Found the following logos: Pizza Hut, KFC

Let's run through one more type of detection that may come in particularly useful when handling user-provided content: sometimes called "safe search", the opposite commonly known online as "NSFW" meaning "not safe for work".

### 14.2.5 Safe-for-work detection

As far as the "fuzziness" of image detection goes, this area tends to be the most fuzzy in that no workplace has the exact same guidelines or culture, so even if we were able to come up with an absolute number quantifying "how inappropriate" an image is, each workplace would need to make their own decisions about whether or not something is appropriate for their particular workplace.

However we're not even lucky enough to have that capability. Even the Supreme Court of the US wasn't quite able to quantify pornography, famously falling back on a definition of "I know it when I see it". If Supreme Court justices can't even define what constitutes pornographic material, it seems a bit unreasonable to expect a computer to be able to define it. That said, a fuzzy number is better than no number at all, so here we'll look at the Cloud Vision API and some of the things it can discover. And hopefully you'll be comfortable relying on this fuzziness as it is the same vision algorithm that filters out "unsafe" images when you do a Google search for images.

**NOTE**

As you might guess we won't be using pornographic or violent demonstration images, and instead will point out the lack of these attributes in images.

Before we begin, let's look at a few of the different "safe" attributes that the Cloud Vision API can detect. The obvious one we just mentioned was pornography, known by the API as "adult" content. This likelihood is simply whether or not the image likely contains any type of adult material, with the most common type being nudity or pornography.

Related but somewhat different is whether the image represents "medical" content (such as a photo of surgery or a rash). While there is often overlap between medical images and "adult" images, there are many images that are "adult" content and not medical. This attribute can be helpful when you're trying to enforce rules in scenarios like medical schools or research facilities. Similar again to adult content is an aspect of whether an image depicts any form of violence. Just like adult content, violence tends to be something subjective that might differ depending on who is looking at it (for example, showing a picture of tanks rolling into Paris might be considered to be violent).

The final aspect of "safe search" is called **spoof detection**. As you might guess, this detect whether an image appears to have been altered somehow, particularly if the alterations lead to the image looking offensive. This might include things like putting devil horns onto photos of celebrities, or other similar alterations. Now that we've walked through the different categories of "safety detection", let's look at the image of the dog again but this time we'll investigate whether we should consider it "safe for work". Obviously we should, but let's see if Cloud Vision agrees.

**Listing 14.22. Script to detect attributes about whether something is "safe for work"**

```
const vision = require('@google-cloud/vision')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

vision.detectSafeSearch('dog.jpg').then((data) => {
  const safeAttributes = data[0];
  console.log(safeAttributes);
});
```

As you might guess this image isn't violent or pornographic as we can see in the result:

**Listing 14.23. Raw output of safety categories**

```
> { adult: false, spoof: false, medical: false, violence: false }
```

As we learned before though, these `true` and `false` values are likelihoods where `LIKELY` and `VERY_LIKELY` become `true` and anything else becomes `false`. To get more detail, we need to use the `verbose` mode that we saw before.

**Listing 14.24. Requesting verbose output from the Vision API**

```
const vision = require('@google-cloud/vision')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

vision.detectSafeSearch('dog.jpg', {verbose: true}).then((data) => {
  const safeAttributes = data[0];
  console.log(safeAttributes);
});
```

As you might expect, the output of this detection with more detail shows that all of these types of content (spoof, adult, medical, and violence) are all very unlikely.

**Listing 14.25. Likelihoods of the various safety categories**

```
> { adult: 'VERY_UNLIKELY',
  spoof: 'VERY_UNLIKELY',
  medical: 'VERY_UNLIKELY',
  violence: 'VERY_UNLIKELY' }
```

So far we've looked at what each detection does, but what if you want to detect multiple things at once? Let's explore how we can combine multiple types of detection into a single API call.

### **14.2.6 Combining multiple detection types**

It turns out that the Cloud Vision API was actually designed to allow multiple types of detection in a single API call, and what we've been doing when we call `detectText`,

for example, is specifically asking for only a single aspect to be analyzed. Let's look at how we can use the generic `detect` method to pick up multiple things at once. The following picture is of a protest outside McDonald's where employees are asking for higher wages. Let's see what is detected when we ask the Cloud Vision API to look for logos, as well as violence and other generic labels.

**Figure 14.12. McDonald's protest**



**Listing 14.26. Requesting multiple annotations in the same request**

```
const vision = require('@google-cloud/vision')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

vision.detect('protest.png', ['logos', 'safeSearch', 'labels']).then((data) => {
  const results = data[0];
  console.log('Does this image have logos?', results.logos.join(', '));
  console.log('Are there any labels for the image?', results.labels.join(', '));
  console.log('Does this image show violence?',
    results.safeSearch.violence ? 'Yes' : 'No');
});
```

It turns our that while there are some labels and logos in the image, the crowd doesn't seem to trigger a violence categorization.

**Listing 14.27. Output showing the logos detected and labels detected**

```
> Does this image have logos? McDonald's
Are there any labels for the image? crowd
Does this image show violence? No
```

Now we've looked at quite a few details about image recognition, but we haven't said how all these examples will affect your bill at the end of the month. Let's take a moment to look at the pricing for the Cloud Vision API so that you can feel comfortable using it in your projects.

### **14.3 Understanding pricing**

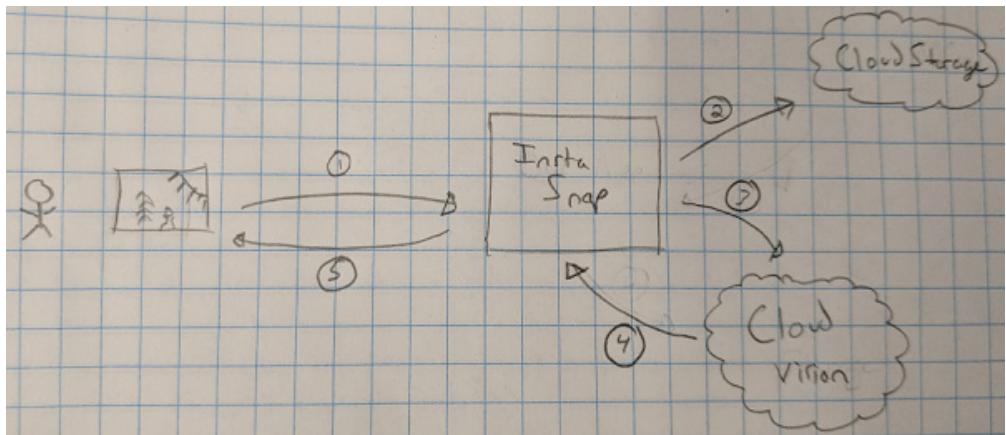
As with most of the APIs you've read about so far, Cloud Vision follows a "pay-as-you-go" pricing model where you are charged a set amount for each API request you make. What's not made clear in your code though is that it's not each API request that costs money, but each "type of detection". For example, if you make a request like we did above in the protest image where we asked for logos, safe-search, and labels, that would cost the exact same as making one request for each of those features. This means that the only benefit from running multiple detections at once is in latency, not price.

The good news is that there is a free-tier for the Cloud Vision API where the first 1,000 requests per month are absolutely free. This means that the examples we just went through should cost you absolutely nothing. After those free requests are used up, the price is \$1.50 for every chunk of 1,000 requests (about \$.0015 per request). And remember that a request is defined as asking for one feature (which means asking for logos and labels on one image is two requests!). Eventually as you do more and more work using the Cloud Vision API, you'll qualify for bulk pricing discounts which you can look up if you're interested. But enough about money. Let's look at how you might use this API in the InstaSnap application.

### **14.4 Case study: Enforcing valid profile photos**

As you might recall, InstaSnap was a cool application that allows you to upload images and share them with your friends. We've talked about where we might store the images (it seemed like Google Cloud Storage was the best fit), but what if we want to make sure that a profile photo actually has a person in it? Or at least show a warning if not? Let's look at how we might do this using the Cloud Vision API. After reading this far you should be familiar with the detection type that we'll need here: faces. The "flow" of how this might work in our application is shown below.

**Figure 14.13. Flow of enforcing valid profile photos**



As you can see here, the idea is that a user would start by uploading a potential profile photo to our InstaSnap application (1). Once received, it would be saved to Cloud Storage (2); after all, if someone wants their profile picture to be of their cat, that's fine, we just want to warn them about it. After the image is saved, we'll send it to the Cloud Vision API (3) to check whether it has any faces in it. We'll then use the response content to flag whether there were faces or not (4), and then pass that flag back to the user (5) along with any other information.

We've already learned how to upload data to Cloud Storage (see [chapter 8](#)), so let's focus first on writing the function that decides on the warning, and then on how it might plug into the existing application. The following function uses a few lines of code to take in a given image and return a boolean value about whether or not a face is in the image. Note that this function assumes you've already constructed a vision client to be shared by your application.

**Listing 14.28. A helper function to decide whether an image has a face in it**

```
const imageHasFace = (imageUrl) => {
  return vision.detectFaces(imageUrl).then( (data) => {
    const faces = data[0];
    return (faces.length == 0);
  });
}
```

- ➊ We'll use this `imageHasFace` method later on to decide whether to show a warning or not.

Once we have this helper method, we can look at how to plug it into our request handler that is called when users upload new profile photos. Note that the follow code snippet is a piece of a larger system so it leaves some methods undefined (such as `uploadToCloudStorage`).

**Listing 14.29. Adding the verification step into the flow**

```

const handleIncomingProfilePhoto = (req, res) => {    1
  const apiResponse = {};
  const url = req.user.username + '-profile-' + req.files.photo.name;
  return uploadToCloudStorage(url, req.files.photo)   2
    .then( () => {
      apiResponse.url = url;
      return imageHasFace(url);                      3
    })
    .then( (hasFace) => {
      apiResponse.hasFace = hasFace;                  4
    })
    .then( () => {
      res.send(apiResponse);                         5
    });
}
  
```

- ➊ We'll start by defining the request handler for an incoming profile photo. This method follows the standard request/response style used by libraries like Express.
- ➋ We'll use a generic object to store the API response as we build it up throughout the flow of promises.
- ➌ To kick things off, first upload the photo itself to our Cloud Storage bucket. This method is defined elsewhere but is easy to write if you want.
- ➍ After the image is stored in our bucket, we use our helper function which returns a promise for whether or not the image has a face in it.
- ➎ Based on the response from our helper function, we set a flag in our API response object which says whether or not the image has a face. In the application, we can use this field to decide whether to show a warning to the user about their profile photo.

And now we can see how an ordinary photo-uploading handler can turn into a more advanced one capable of showing warnings when the photo uploaded doesn't contain a face!

## 14.5 Summary

- Image recognition is the ability to take a chunk of visual content (like a photo) and annotate it with information (such as textual labels).
- Cloud Vision is a hosted image recognition service that can add lots of different annotations to photos, including recognizing faces and logos, detecting whether content is "safe", finding dominant colors, and labeling things that appear in the photo.
- Since Cloud Vision uses machine learning, it is always improving. This means that over time the same image may produce different (likely more accurate) annotations.

# 15

## *Cloud Natural Language: Text analysis*

**This chapter covers:**

- An overview of natural language processing
- How the Cloud Natural Language API works
- The different types of analysis supported by Cloud Natural Language
- How Cloud Natural Language pricing is calculated
- An example to suggest hash tags

### **15.1 What is natural language processing?**

Natural language processing is the act of taking text content as input and deriving some structured meaning or understanding from it as output. For example, we might take the sentence "I'm going to the mall" and derive {action: "going", target: "mall"}. It turns out that this is much more difficult than it looks, which we can see by looking at the following ambiguous sentence:

Joe drives his Broncos to work.

There's obviously some ambiguity here in what exactly is being "driven". Currently "driving" something tends to point towards a vehicle. Driving tends to refer to steering a vehicle, but about 100 years ago it probably meant directing horses. In the US, Denver has a sports team with the same name, so this could refer to a team that Joe coaches (e.g., "Joe drives his Broncos to victory"). Looking at the term Bronco on Wikipedia we can see it has a long list of potential meanings: 22 different sports teams, 4 vehicles, and quite a few others (including the default which is the horse).

In truth, this sentence is ambiguous and we cannot say with certainty whether it means

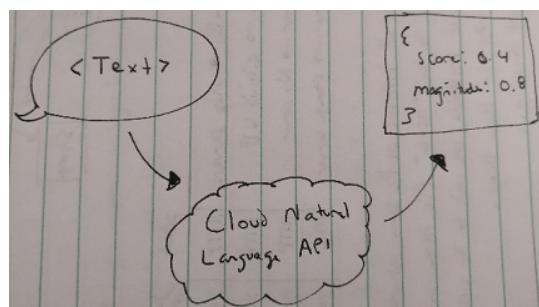
that Joe forces his Bronco horses to his work place or whether he gets in one of the many Ford Bronco cars he owns and uses one of them to transport himself to work, or something else completely. The point is that without more context we cannot accurately determine the meaning of a sentence and therefore it's unreasonable to expect a computer to do so.

Because of this, natural language processing is very complex and still an active area of research. The Cloud Natural Language API attempts to simplify this down so that you can use machine learning to process text content without keeping up with all the research papers. Like any machine learning API the results are "best guesses" which means that you should treat the output as suggestions that may morph over time rather than absolute unquestionable facts. Let's explore some of what the Cloud NL API can actually do, and see how you might use it in real life, starting with looking at sentiment.

## 15.2 How does the Natural Language API work?

Similar to Google Cloud's other machine learning APIs, the Natural Language API is a stateless API where you send it some input (in this case the input is text), and the API returns some set of annotations about the text.

**Figure 15.1. Natural Language API flow overview**



As of this writing, the NL API is able to annotate 3 different features of input text: syntax, entities, and sentiment. Let's look briefly at each of these to get an idea of what they all mean.

### Syntax

Much like diagramming sentences in grade school, the NL API can take a document and parse it into sentences, finding "tokens" along the way. These tokens would have a part of speech, canonical form of the token, and more.

### Entities

Since the NL API can parse the syntax of a sentence, once it does that it can also look at each token individually and do a look-up in Google's knowledge graph to associate the two together. This means that if you write a sentence about a famous person (e.g., Barack Obama), you're able to both find that a sentence is about Barack Obama, and

have a pointer to a specific entity in the knowledge graph. Further, using the concept of salience (or "prominence"), you'll be able to see whether the sentence is focused on Barack Obama or whether he's just mentioned in passing.

### **Sentiment**

Perhaps the most interesting aspect of the NL API is the ability to look at a chunk of input text and understand the emotional content involved. In other words, recognizing that a given sentence expresses positive or negative emotion, and in what quantity. With this, you're able to look at a given sentence and get an idea of the emotion the author was attempting to express.

As with all machine learning APIs, these values should be treated as somewhat "fuzzy" in that even our human brains can't necessarily come up with perfectly correct answers — sometimes because there is none! But having a hint in the right direction is still better than knowing nothing about your text. Let's dive right in and explore how some of these analyses work, starting with sentiment.

## **15.3 Sentiment analysis**

One particularly interesting aspect of "understanding" is recognizing the sentiment or emotion of what was said. As humans, we can generally tell whether a given sentence is happy or sad, but asking a computer to do this is still relatively new. For example, the sentence "I like this car." is something most of us would consider to be positive, and the sentence "This car is ugly" would likely be considered to be "negative". But what about those odd cases that are both positive and negative?

Consider the input "This car is really pretty. It also gets terrible gas mileage." These two taken together lie somewhere in the middle of positive and negative as it notes a good thing about the car as well as a bad thing. However it's not quite the same as a truly neutral sentence such as "This is a car." So how do we distinguish a truly neutral and unemotional input from a highly emotional input that happens to be neutral because the positive emotions cancel out the negative?

To do this, we need to keep track of both the sentiment itself as well as the magnitude of the overall sentiment that went into coming up with the final sentiment result. To see this, take a look at the following sentences where the overall sentiment may end up being neutral even though the emotional magnitude is high.

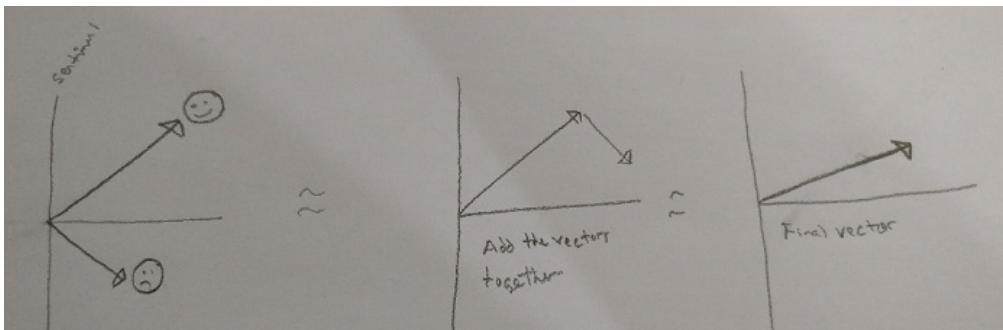
**Table 15.1. Comparing sentences with similar sentiment and different magnitudes**

<b>Sentence</b>	<b>Sentiment</b>	<b>Magnitude</b>
"This car is really pretty."	Positive	High
"This car is ugly."	Negative	High
"This car is pretty. It also gets terrible gas mileage."	Neutral	High
"This is a car."	Neutral	Low

Putting this in a more technical way, you can consider an expression of the overall

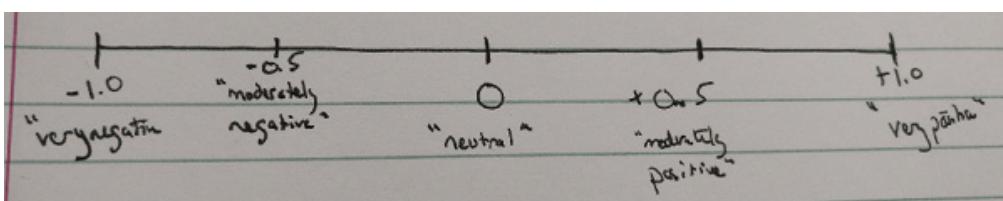
sentiment as a vector which conveys both a rating of the positivity (or negativity) and a magnitude which expresses how strongly that sentiment is expressed. Then, to come up with the overall sentiment and magnitude, simply add the two vectors together to get a final vector. It should become clear that the "magnitude" dimension of the vector will be a sum of both, even if the sentiment dimensions happen to cancel each other out and return a mostly neutral overall sentiment.

**Figure 15.2. Combining multiple sentiment vectors into a final vector**



In cases where the score is significant (e.g., not close to neutral), the magnitude isn't really helpful. However in those cases where the positive and negative cancel each other out, the magnitude can help distinguish between a truly unemotional input and one where the emotions positivity and negativity just happen to neutralize one another. When you send a chunk of text to the Natural Language API, you'll get back both a score and a magnitude, which together represent these two aspects of the sentiment. The score will be a number between -1 and 1 (negative numbers represent negative sentiment) which means that a "neutral" statement would have a score close to zero.

**Figure 15.3. Sentiment scale from -1.0 to +1.0**



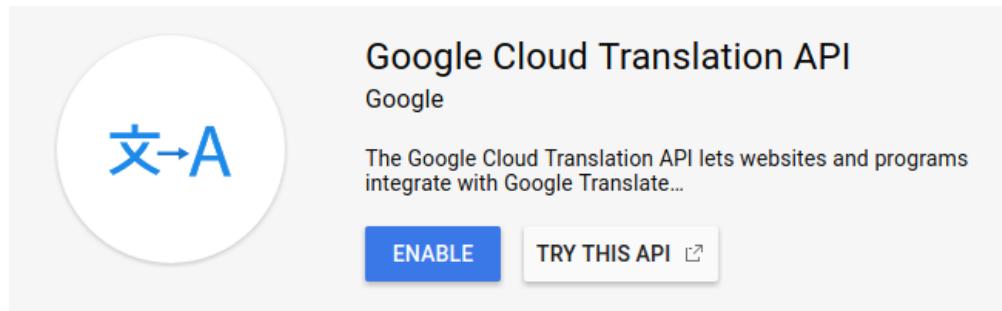
In cases where that score is close to zero, the magnitude value will represent how much emotion actually went into it. The magnitude will be a number greater than zero, with zero meaning that the statement was truly neutral, and a larger number representing more emotion. For a single sentence, the score and magnitude will be equivalent since sentences are the smallest unit analyzed. This, oddly, means that a sentence containing both positive and negative emotion will have different results than two sentences with equivalent information. To see how this works, let's try writing some code that analyzes the sentiment of a few simple sentences.

**NOTE**

As you may have read previously, you'll need to have a service account and credentials in order to use this API.

To start, make sure to enable the Natural Language API using the Cloud Console. You can do this by searching for "Cloud Natural Language API" in the main search box at the top of the page. That query should come up with just one result, and if you click on it you should land on a page with a big "Enable" button. Then you just have to click that and you're ready to go.

**Figure 15.4. Enable the Natural Language API**



Once you have that working, you'll need to install the client library for Node.js. To do this, just run `npm install @google-cloud/language@0.8.0` and then we can start writing some code!

#### **Listing 15.1. Detecting sentiment for a sample sentence**

```
const language = require('@google-cloud/language')({
  projectId: 'your-project-id', ①
  keyFilename: 'key.json'        ②
});

language.detectSentiment('This car is really pretty.')
  .then((result) => {
    console.log('Score:', result[0]);
  });

```

- ① Don't forget that the project ID must match the credentials in your service account.
- ② Remember to use the service account key file for credentials or your code won't work!

If you run this code with the proper credentials, you should see output saying something like the following:

#### **Listing 15.2. Detected sentiment score**

```
> Score: 0.5
```

It should be no surprise that the overall sentiment of that sentence was moderately positive. Remember, 0.5 is effectively 75% of the way (not half way!) between "totally

negative" (1.0) and "totally positive" (-1.0). It's always worth mentioning that it's completely normal if you get a slightly different value for a score. With all machine learning APIs the algorithms and underlying systems that generate the outputs are constantly learning and improving, so the specific results here may vary over time. Let's try looking at one of those sentences that was overall neutral.

#### **Listing 15.3. Detecting sentiment for a sample neutral sentence**

```
const language = require('@google-cloud/language')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

const content = 'This car is nice. It also gets terrible gas mileage!';
language.detectSentiment(content).then((result) => {
  console.log('Score:', result[0]);
});
```

When you run this, you should see exactly what we predicted: a score of zero. So how do we tell the difference between content that is "neutral" overall, but highly emotional and something truly neutral? Let's try comparing two inputs while increasing the verbosity of the request.

#### **Listing 15.4. Representing the difference between neutral and non-sentimental sentences**

```
const language = require('@google-cloud/language')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

const inputs = [
  'This car is nice. It also gets terrible gas mileage!', ❶
  'This is a car.' ❷
];

inputs.forEach((content) => {
  language.detectSentiment(content, {verbose: true}) ❸
    .then((result) => {
      const data = result[0];
      console.log([
        'Results for "' + content + '":',
        '  Score: ' + data.score,
        '  Magnitude: ' + data.magnitude
      ].join('\n'));
    });
});
```

- ❶ This input is emotional but should overall be close to neutral.
- ❷ This sentence is unemotional and should be overall close to neutral.
- ❸ Make sure to request "verbose" output which includes the magnitude in addition to the score.

When you run this, you should see something like:

#### **Listing 15.5. Output showing detected sentiment differences and magnitudes**

```
Results for "This is a car.":
Score: 0.20000000298023224
Magntiude: 0.20000000298023224
Results for "This car is nice. It also gets terrible gas mileage!":
Score: 0
Magntiude: 1.2999999523162842
```

As you can see, it turns out that the "neutral" sentence actually had quite a bit of emotion. Additionally, it seems that what we thought to be a neutral statement ("This is a car.") is rated slightly positive overall, which helps to show how judging the sentiment of content is a bit of a fuzzy process without a clear and universal answer. Now that you understand how to analyze text for emotion, let's take a detour over towards another area of analysis and look at how to recognize key entities in a given input.

## **15.4 Entity recognition**

Entity recognition is a way of taking a piece of input text and finding if there are any special entities. These can be people, places, organizations, works of art, or anything else you'd consider a "proper noun". This works by parsing the sentence for tokens and comparing those tokens against the entities that Google has stored in its knowledge graph. This allows the API to recognize things *in context* rather than with a plain old text-matching search.

It also means that the API is able to distinguish between terms that could be "special" depending on their use (such as "blackberry" the fruit versus "Blackberry" the phone). Overall, it should be clear that if you're interested in doing things like suggesting tags or metadata about textual input, you can use entity detection to try to come up with which entities are present in your input. To see this in action, consider the following sentence:

Barack Obama prefers an iPhone over a Blackberry when vacationing in Hawaii.

Let's take this sentence and try to identify all of the "entities" that were mentioned.

#### **Listing 15.6. Recognizing entities in a sample sentence**

```
const language = require('@google-cloud/language')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

const content = 'Barack Obama prefers an iPhone over a Blackberry when ' +
  'vacationing in Hawaii.';

language.detectEntities(content).then((result) => {
  console.log(result[0]);
});
```

If you run this, the output should look something like the following.

#### **Listing 15.7. Detected entities broken down by category**

```
> { people: [ 'Barack Obama' ],
  goods: [ 'iPhone' ],
  organizations: [ 'Blackberry' ],
  places: [ 'Hawaii' ] }
```

As you can see, the Natural Language API detected 4 distinct "entities": Barack Obama, iPhone, Blackberry, and Hawaii. This can be helpful if you're trying to discover whether there are famous people in a given sentence, or specific set of places. But were all of these equally important in the sentence? It seems to me that "Barack Obama" was far more prominent in the sentence than "Hawaii".

It turns out that the Natural Language API can distinguish between differing levels of "importance" or "prominence". This means that it attempts to rank things according to how important they are in the sentence so that, for example, you could only consider the "most important" entity in the sentence (or just the top 3). To see this extra data all you have to do is use the "verbose" mode when detecting entities as shown here.

#### **Listing 15.8. Detecting entities with verbosity turned on**

```
const language = require('@google-cloud/language')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

const content = 'Barack Obama prefers an iPhone over a Blackberry when ' +
  'vacationing in Hawaii.';
const options = {verbose: true}; ①

language.detectEntities(content, options).then((result) => {
  console.log(result[0]);
});
```

① Use `{verbose: true}` to get more context on the annotation results.

When you run this, rather than just seeing the names of the entities, you'll see the entity raw content which includes the entity category (type), some extra metadata (including a unique ID for the entity), and, most importantly, the salience which is a score between 0 and 1 of how important the given entity is in the input (higher salience meaning "more important").

#### **Listing 15.9. Additional data provided with `{verbose: true}`**

```
> { people:
  [ { name: 'Barack Obama',
    type: 'PERSON',
    metadata: [Object],
    salience: 0.5521853566169739,
```

```

        mentions: [Object] } ],
goods:
[ { name: 'iPhone',
  type: 'CONSUMER_GOOD',
  metadata: [Object],
  salience: 0.1787826418876648,
  mentions: [Object] },
organizations:
[ { name: 'Blackberry',
  type: 'ORGANIZATION',
  metadata: [Object],
  salience: 0.15308542549610138,
  mentions: [Object] },
places:
[ { name: 'Hawaii',
  type: 'LOCATION',
  metadata: [Object],
  salience: 0.11594659835100174,
  mentions: [Object] } ] }

```

So what if you want to specifically get the "most salient" entity in a given sentence? And what effect does the phrasing have on salience? Consider the following two sentences:

1. "Barack Obama prefers an iPhone over a Blackberry when in Hawaii."
2. "When in Hawaii an iPhone, not a Blackberry, is Barack Obama's preferred device."

Let's look at these two and ask the API to decide which entity is deemed most important:

#### **Listing 15.10. Comparing two similar sentences with different phrasing**

```

const language = require('@google-cloud/language')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

const inputs = [
  'Barack Obama prefers an iPhone over a Blackberry when in Hawaii.',
  'When in Hawaii an iPhone, not a Blackberry, is Barack Obama\'s preferred
device.',
];
const options = {verbose: true};

inputs.forEach((content) => {
  language.detectEntities(content, options).then((result) => {
    const entities = result[1].entities;
    entities.sort((a, b) => {
      return -(a.salience - b.salience); ❶
    });
    console.log(
      'For the sentence "' + content + '"',
      '\n  The most important entity is:', entities[0].name,
      '(' + entities[0].salience + ')');
  });
});

```

```
});
```

- ① Sort entities by decreasing salience (largest salience first).

After running this snippet, you can see just how different the values turn out to be given different phrasing of similar sentences. Compare this to the basic way of recognizing a specific set of strings where you only get an indicator of what appears, rather than how important it is to the sentence.

#### **Listing 15.11. Different importance of entities depending on phrasing**

```
> For the sentence "Barack Obama prefers an iPhone over a Blackberry when in Hawaii."
The most important entity is: Barack Obama (0.5521853566169739)
For the sentence "When in Hawaii an iPhone, not a Blackberry, is Barack Obama's preferred device."
The most important entity is: Hawaii (0.440546065556892395)
```

Let's take it up a notch and see what happens when we look at inputs that are in languages besides English!

Hugo Chavez era un dictador de Venezuela.

It turns out that the Natural Language API does actually support more than just English, currently including both Spanish (es) and Japanese (jp). Let's run an entity analysis on our sample Spanish sentence which translates to "Hugo Chavez was a dictator of Venezuela."

#### **Listing 15.12. Detecting entities in Spanish**

```
const language = require('@google-cloud/language')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

language.detectEntities('Hugo Chavez era de Venezuela.', {
  verbose: true, ①
  language: 'es' ②
}).then((result) => {
  console.log(result[0]);
});
```

- ① We turn on verbose mode to see the salience rankings.
- ② Here we use the BCP-47 language code for Spanish (es), however, if you leave this empty the API will try to guess which language you're using.

When you run this, you should see something like the following:

#### **Listing 15.13. Salience and entities detected in Spanish**

```
> { people:
```

```
[ { name: 'Hugo Chavez',
  type: 'PERSON',
  metadata: [Object],
  salience: 0.7915874123573303,
  mentions: [Object] } ],
places:
[ { name: 'Venezuela',
  type: 'LOCATION',
  metadata: [Object],
  salience: 0.20841257274150848,
  mentions: [Object] } ] }
```

As you can see, the results are what you'd expect where the API recognizes Hugo Chavez and Venezuela. Now let's move onto the final area of textual analysis provided by the Natural Language API: syntax.

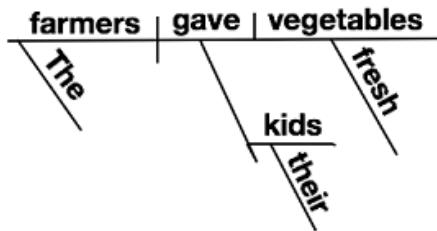
## 15.5 Syntax analysis

You may recall elementary school English classes where your teacher asked you to diagram a sentence to point out the various parts of speech such as the phrases, verbs, nouns, participles, adverbs, and more. In a sense, diagrams like that are really dependency graphs which allow you to see the "core" of the sentence, and push modifiers and other non-essential information to the side. For example, let's take the following sentence:

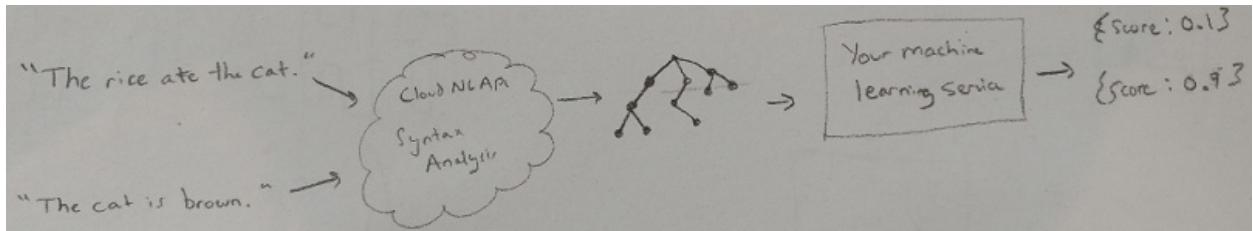
The farmers gave their kids fresh vegetables.

Diagramming this the way our teachers showed us might look something like this:

**Figure 15.5. Diagram of a sample sentence**



Similarly, the Natural Language API can provide a dependency graph given the same sentence as input. Unlike the other detection features of this API, building a syntax tree is offered to make it easier to build your own machine learning algorithms on natural language inputs. For example, let's say you wanted to build a system that detected whether a sentence "makes sense" or not. You could use the syntax tree from this API as the first step in processing your input data. Then, based on that syntax tree, you could build a model that returned a "sense score" for the given input, as shown below.

**Figure 15.6. Pipeline for an example sense-detection service**

In short, you probably wouldn't use this API directly in your applications, but it could be really useful for lower-level processing of data, to build models that you'd then use directly. This API works by first parsing the input for sentences, then tokenizing the sentence and recognizing the part of speech of each word, and building a tree of how all the words fit together in the sentence. Using our example sentence once again, let's look at how the API understands input and tokenizes it into a tree.

#### **Listing 15.14. Detecting syntax for a sample sentence**

```
const language = require('@google-cloud/language')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

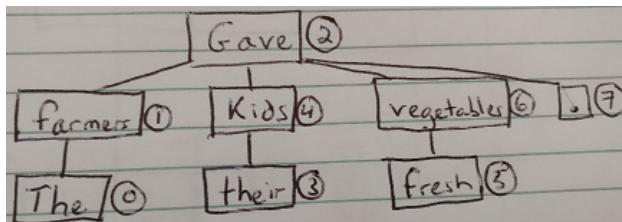
const content = 'The farmers gave their kids fresh vegetables.';
language.detectSyntax(content).then((result) => {
  const tokens = result[0];
  tokens.forEach((token, index) => {
    const parentIndex = token.dependencyEdge.headTokenIndex;
    console.log(index, token.text, parentIndex);
  });
});
```

Running this will give you a table of the dependency graph, which should look like Table 15.2.

**Table 15.2. Dependency graph of the tokens**

Index	Text	Parent
0	'The'	1 ('farmers')
1	'farmers'	2 ('gave')
2	'gave'	2 ('gave')
3	'their'	4 ('kids')
4	'kids'	2 ('gave')
5	'fresh'	6 ('vegetables')
6	'vegetables'	2 ('gave')
7	''	2 ('gave')

You could use these numbers to build a dependency tree that looks something like this:

**Figure 15.7. Dependency graph represented as a tree**

Now that you understand the different types of textual analysis that the Natural Language API can handle, let's look at how much this will end up costing.

## 15.6 Understanding pricing

As with most Cloud APIs, the Cloud Natural Language API charges based on the actual usage, in this case the amount of text sent for analysis, with different rates for the different types of analysis. To simplify the unit of billing, the NL API measures the amount of text in chunks of 1,000 characters, meaning that all of our examples so far would be billed as a single unit, however if you send a very long document for entity recognition, it'd be billed as the number of 1,000 character chunks needed to fit the entire document (in other words, `Math.ceil(document.length / 1000.0)`).

This type of billing is easiest when we assume that most requests only involve documents with fewer than 1,000 characters, in which case the billing is the same as "per request". Next, it turns out that different types of analysis cost different amounts, with entity recognition leading the pack at \$0.001 USD each. However, as you make more and more requests in a given month, the per-unit price drops (in this case, by half) as shown in Table 15.3. Additionally, the first 5,000 requests per month of each type are free of charge.

**Table 15.3. Pricing table for Cloud Natural Language API**

Feature	Cost per unit			
	First 5,000	Up to 1 million	Up to 5 million	Up to 20 million
Entity recognition	Free!	\$0.001	\$0.0005	\$0.00025
Sentiment analysis	Free!	\$0.001	\$0.0005	\$0.00025
Syntax analysis	Free!	\$0.0005	\$0.00025	\$0.000125

Multiplying these by 1,000 makes for much more manageable numbers, coming to \$1 USD per thousand requests for most entity recognition and sentiment analysis operations. Also note that when you combine two types of analysis (e.g. a single request for sentiment and entities), the cost is the combination (e.g. \$0.002) for that request. To show this in a quick example, let's say that every month you're running entity analysis over 1,000 long-form documents (e.g. about 2,500 characters), and sentiment analysis over 2,000 short tweet-like snippets every day. The cost breakdown is summarized below.

**Table 15.4. Pricing example for Cloud Natural Language API**

<b>Item</b>	<b>Quantity</b>	<b>1k character "chunks"</b>	<b>Cost per unit</b>	<b>Total per month</b>
Entity detection (long-form)	1,000	3,000	\$0.001	\$3.00
Sentiment analysis	60,000	60,000	\$0.001	\$60.00
<b>Total</b>				<b>\$63.00</b>

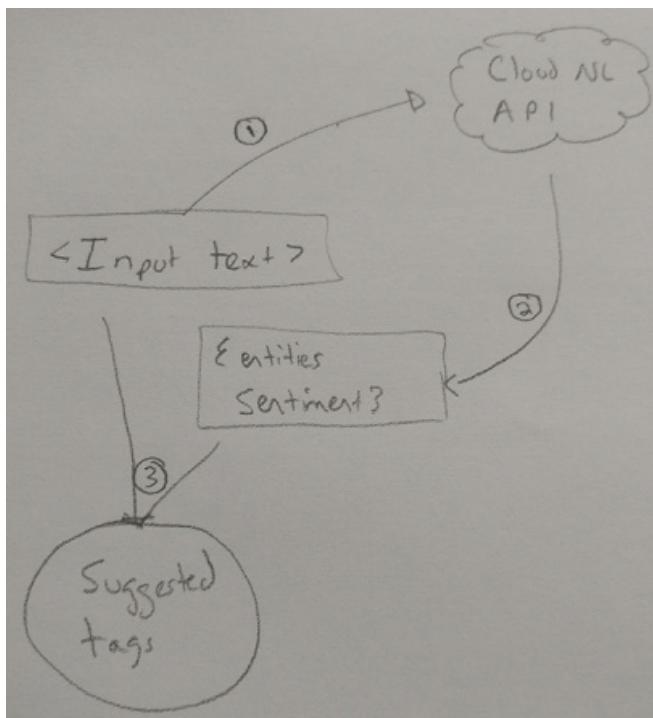
Note specifically that the long-form documents ballooned into 3x the number of "chunks" because they're about 2,500 characters (which needs 3 chunks), and that our sentiment analysis requests were defined as 2,000 *daily* rather than *monthly*, resulting in a 30x multiplier. Now that you've seen the cost structure and all the different types of analysis offered by the Natural Language API, let's try putting a couple of them together into something that might provide some value to users: hash-tagging suggestions.

## 15.7 Case study: Suggesting InstaSnap hash-tags

As you may recall, our sample application called InstaSnap is an app that allows people to post pictures and captions, and share them with their friends. Considering that, along with the fact that the NL API is able to take some textual input and come up with both a sentiment analysis as well as the entities in the input, what if we were able to take a single post's caption and come up with some tags that are likely to be relevant? How would this work?

First, we'd take a post's caption as input text, and send it to the Natural Language API. Next, the Natural Language API would send back both sentiment and any detected entities. After that, we'd have to coerce some of the results into a format that's useful in this scenario, for example, #0.8 isn't a great tag, but #happy is. Finally, we'd display a list of suggested tags to the user.

**Figure 15.8. Overflow flow of the tagging suggestion process**



Let's start by looking at the code to request both sentiment and entities in a single API call.

**Listing 15.15. Detecting sentiment and entities in a single API call**

```

const language = require('@google-cloud/language')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

const caption = 'SpaceX lands on Mars! Fantastic!';
const document = language.document(caption);
const options = {entities: true, sentiment: true, verbose: true};
document.annotate(options).then((data) => {
  const result = data[0];
  console.log('Sentiment was', result.sentiment);
  console.log('Entities found were', result.entities);
});
  
```

- ① Here we're assuming Elon Musk has finally managed to land on Mars (and uses InstaSnap).
- ② To handle multiple annotations at once, create a "document" and operate on that.

If you run this snippet, you should see output that looks pretty familiar.

**Listing 15.16. Detecting sentiment and entities at once**

```
> Sentiment was { score: 0.400000059604645, magnitude: 0.800000011920929 }
Entities found were { organizations:
  [ { name: 'SpaceX',
      type: 'ORGANIZATION',
      metadata: [Object],
      salience: 0.7309288382530212,
      mentions: [Object] } ],
  places:
  [ { name: 'Mars',
      type: 'LOCATION',
      metadata: [Object],
      salience: 0.26907116174697876,
      mentions: [Object] } ] }
```

Now let's see what we can do to apply some tags, starting with entities first. For most entities, we can just toss a # character in front of the place and call it a day. In this case, "SpaceX" would become #SpaceX, and "Mars" would become #Mars. Seems like a good start. We can also dress it up and add suffixes for organizations, places, and people. For example, "SpaceX" could become #SpaceX4Life(adding "4Life"), and "Mars" could become #MarsIsHome (adding "IsHome"). These might also change depending on the sentiment, so maybe we have some suffixes that are "positive" and some "negative".

What about for the sentiment? Let's come up with some "happy" and "sad" tags, and use those when the sentiment passes certain thresholds. Then we can make a `getSuggestedTags` method that does all the hard work.

**Listing 15.17. Our method for getting the suggested tags**

```
const getSuggestedTags = (sentiment, entities) => {
  const suggestedTags = [];

  const entitySuffixes = {  
    organizations: { positive: ['4Life', 'Forever'], negative: ['Sucks'] },  
    people: { positive: ['IsMyHero'], negative: ['Sad'] },  
    places: { positive: ['IsHome'], negative: ['IsHell'] },  
  };  
  
  const sentimentTags = {  
    positive: ['#Yay', '#CantWait', '#Excited'],  
    negative: ['#Sucks', '#Fail', '#Ugh'],  
    mixed: ['#Meh', '#Conflicted'],  
  };  
  
  // Start by grabbing any sentiment tags.  
  let emotion;  
  if (sentiment.score > 0.1) {  
    emotion = 'positive';  
  } else if (sentiment.score < -0.1) {  
    emotion = 'negative';  
  } else if (sentiment.magnitude > 0.1) {  
    emotion = 'mixed';  
  }
```

```

    } else {
        emotion = 'neutral';
    }

    // Add a random tag to the list of suggestions.
    let choices = sentimentTags[emotion];
    if (choices) {
        suggestedTags.push(choices[Math.floor(Math.random() * choices.length)]);
    }

    // Now run through all the entities and attach some suffixes.
    for (let category in entities) {
        let suffixes;
        try {
            suffixes = entitySuffixes[category][emotion]; ⑤
        } catch (e) {
            suffixes = [];
        }

        if (suffixes.length) {
            entities[category].forEach((entity) => {
                let suffix = suffixes[Math.floor(Math.random() * suffixes.length)];
                suggestedTags.push('#' + entity.name + suffix);
            });
        }
    }

    // Return all of the suggested tags.
    return suggestedTags;
};


```

- ① Come up with a list of possible suffixes for each category of entity.
- ② Store a list of emotional tags for each category (positive, negative, mixed, or neutral).
- ③ Use the sentiment analysis results to choose a tag from the category.
- ④ Don't forget to check the magnitude to distinguish between "mixed" and "neutral".
- ⑤ Use a try/catch block in case you don't happen to have tag choices for each particular combination.

And now that we have that method, our code to evaluate it and come up with some suggested tags should look pretty simple.

#### **Listing 15.18. Detecting sentiment and entities in a single API call**

```

const language = require('@google-cloud/language')({
    projectId: 'your-project-id',
    keyFilename: 'key.json'
});

const caption = 'SpaceX lands on Mars! Fantastic!';
const document = language.document(caption);
const options = {entities: true, sentiment: true, verbose: true};

document.annotate(options).then((data) => {
    const sentiment = data[0].sentiment;
    const entities = data[0].entities;
    const suggestedTags = getSuggestedTags(sentiment, entities); ①
}

```

```

        console.log('The suggested tags are', suggestedTags);
        console.log('The suggested caption is',
                    ''' + caption + ' ' + suggestedTags.join(' ') + ''');
    });
}

```

- ➊ Here we use the helper function to retrieve suggested tags given the detected sentiment and entities.

When you run this code your results might be different from those here due to the random selection of the options, but given this sample caption, a given output might look something like this:

#### **Listing 15.19. Output from suggesting tags on a caption**

```

> The suggested tags are [ '#Yay', '#SpaceX4Life', '#MarsIsHome' ]
The suggested caption is "SpaceX lands on Mars! Fantastic! #Yay #SpaceX4Life
#MarsIsHome"

```

## **15.8 Summary**

- The Natural Language API is a very powerful textual analysis service.
- If you need to discover details about text in a scalable way, the Natural Language API is likely a good fit for you.
- The API can analyze text for "entities" (people, places, organizations), syntax (tokenizing and diagramming sentences), and sentiment (understanding the emotional content of text).
- As with all machine learning today, the results from this API should be treated as suggestions rather than absolute fact (after all, it can be tough for people to decide on whether a given sentence is happy or sad).

# 16

## *Cloud Speech: Audio to text conversion*

**This chapter covers:**

- An overview of speech recognition
- How the Cloud Speech API works
- How Cloud Speech pricing is calculated
- An example of generating automated captions from audio content

### 16.1 What is speech recognition?

When we talk about speech recognition, we generally mean taking an audio stream (e.g., an MP3 file of a book on tape) and turning it into text (e.g., back into the actual written book). This process sounds pretty straightforward, however as you may know language is a particularly tricky human construct. For example, there is a psychological phenomenon called the McGurk effect where what we *see* changes what we *hear*. In one classic example, the sound "ba" can be perceived as "fa" so long as we see someone's mouth forming an "f" sound. This, as you might expect, means that an audio track alone is not always enough to completely understand what was said.

This might seem weird given that we've survived with phone calls all these years. It turns out that there is a difference between "hearing" and "listening". When you "hear" something, you're taking sounds and turning them into words. When you "listen", you're taking sounds and combining them with your context and understanding, which means you can fill in the blanks when some sounds are ambiguous. For example, if you heard someone say "I drove the -ar back", even if you missed the first consonant of that "ar" sound, you can use the context of "drove" to guess that this word was "car".

This phenomenon leads to some pretty interesting (and funny) scenarios, particularly when the listener decides to "take a guess" at what was said. For example, Ken Robinson spoke at a TED conference about how kids sometimes guess at things when they hear the words but don't quite understand the meaning. In his example, some children were putting on a play about the nativity for Christmas, and the wise men went out of order when presenting the gifts. The order in the script was gold, frankincense, and then myrrh, however the children said, "I bring you gold." "I bring you myrrh." and finally the last child said, "Frank sent this." The words all sounded the same, and the last child tried to "take a guess" based on the context.

**Figure 16.1. Understanding based on context ("Who is Justin Bieber?")**



So what does all this mean for you? In general, it means that you should treat the results from a given audio file to be helpful suggestions that are usually right, but not guaranteed. To put this in context, you probably wouldn't want to use a machine learning algorithm for court transcripts just yet, but they may help stenographers improve their efficiency by using the output as a baseline to build from. At this point, let's look at how the Cloud Speech API works and how you can use it in your own projects.

## 16.2 Simple speech recognition

Similar to the Cloud Vision API, the Cloud Speech API has textual content as an output but requires a more complex input — in this case an audio stream. Therefore,

the simplest way of recognizing the textual content in an audio file is to take the audio file (e.g., a .wav file) and send it to the Cloud Speech API for processing. The output of the audio will be what was said in the audio file. There are a few things we've left out though.

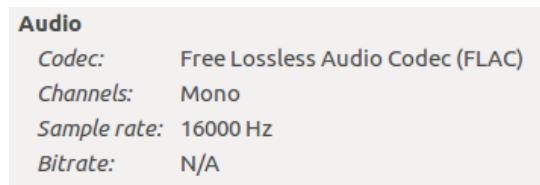
First, we'll need to tell the Cloud Speech API about the format of the audio, as there are many different formats out there, each with their own compression algorithms. Next, even given the audio format, you also need to know a bit about the sample rate of the file. This is an important aspect of digital signal processing and tells the audio processor the clock time covered by each data point (higher sample rates are closer to the raw analog audio). In other words, to make sure the API "hears" the audio at the right speed, it's important to know the sample rate.

**TIP**

**While you probably don't know the sample rate of a given recording, the software that did the recording likely added a metadata tag to the file saying what the sample rate was. You can usually find this by looking at the "properties" of the file in your file explorer.**

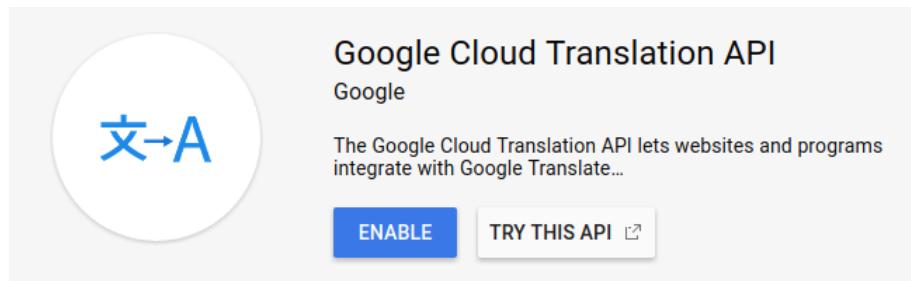
Finally, if you know the language spoken in the audio it's helpful to tell the API what that is. This tells the API which lingual model to use when recognizing content in the audio file. Let's dig into some of the code to actually do this, using a pre-made recording of an audio file stored on Google Cloud Storage to start. The audio format properties of this file are shown below.

**Figure 16.2. Audio format properties of the pre-made recording**



Before we get going we'll need to enable the API in the Cloud Console. To do this, in the main search box at the top of the page type in "Cloud Speech API". This should only have one result and the page it brings you to should have a big "Enable" button. Just click this and you should be all set.

**Figure 16.3. Enabling the Cloud Speech API**



Now that the API is enabled and ready to go, we'll need to install the client library. To do this, just run `npm install @google-cloud/speech@0.8.0` and you should be ready to go. Now that that client library is all set up, let's write some code that recognizes the text in this file.

#### **Listing 16.1. Recognizing text from an audio file**

```
const speech = require('@google-cloud/speech')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

const audioFilePath = 'gs://cloud-samples-tests/speech/brooklyn.flac'; ①
const config = { ②
  encoding: 'FLAC',
  sampleRate: 16000
};
speech.recognize(audioFilePath, config).then((response) => {
  const result = response[0];
  console.log('This audio file says: "' + result + '"');
});
```

- ① Notice that this file lives on Google Cloud Storage rather than as a local audio file.
- ② In this API, the configuration is required as you need to tell the API about the audio format and sample rate (in this case, FLAC and 16,000).

When you run this code snippet, you should see some interesting output, shown below.

#### **Listing 16.2. Output from recognizing the audio file**

```
> This audio file says: "how old is the Brooklyn Bridge"
```

One important thing to notice about this recognition is how long it took. The reason is pretty simple: the Cloud Speech API needs to "listen" to the entire audio file, so the recognition process is directly correlated to the length of the audio. This means that extraordinarily long audio files (e.g., more than a few seconds) shouldn't be processed in the way we just did. Another important thing to notice is that there's no concept of "confidence" in this result. In other words, how sure is the Cloud Speech API that the audio says that exact phrase? To get that type of information, we can use the `verbose` flag.

#### **Listing 16.3. Recognizing text from an audio file with verbosity turned on**

```
const speech = require('@google-cloud/speech')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
};

const audioFilePath = 'gs://cloud-samples-tests/speech/brooklyn.flac';
const config = {
  encoding: 'FLAC',
```

```

        sampleRate: 16000,
        verbose: true ①
    };
    speech.recognize(audioFilePath, config).then((response) => {
        const result = response[0][0];
        console.log('This audio file says: "' + result.transcript + '',
                   '(with ' + Math.round(result.confidence) + '% confidence)');
    });
}

```

- ① Here we set the verbose option to true.

When you run this, you should see the output looking something like the following.

#### **Listing 16.4. Output of recognition with confidence value**

```
> This audio file says: "how old is the Brooklyn Bridge" (with 98% confidence)
```

So how do we deal with audio files that are longer? Or what about audio that isn't complete yet, but instead is a running stream? Let's look at how the Cloud Speech API deals with continuous recognition.

### **16.3 Continuous speech recognition**

It turns out that sometimes you can't take an entire audio file and send it as one chunk to the API for recognition. The most common case of this is simply a large audio file, which is too big to treat as one big blob, so instead you have to break it up into many smaller chunks. But this is also true when you're trying to recognize streams that are live (as in, not prerecorded), since these streams keep going until you decide to turn it off. To handle this, the Speech API allows asynchronous recognition which will accept chunks of data, recognizing them along the way, and return a final result once the audio stream is completed. Let's look at how to do that with our same file, but treated as chunks.

#### **Listing 16.5. Recognizing with a stream**

```

const fs = require('fs');
const speech = require('@google-cloud/speech')({
    projectId: 'your-project-id',
    keyFilename: 'key.json'
});

const audioFilePath = 'gs://cloud-samples-tests/speech/brooklyn.flac';
const config = {
    encoding: 'FLAC',
    sampleRate: 16000,
    verbose: true
};

speech.startRecognition(audioFilePath, config).then((result) => { ①
    const operation = result[0];
    operation.on('complete', (results) => { ②
        console.log('This audio file says: "' + results[0].transcript + ''', ③
    });
}

```

```

        '(with ' + Math.round(results[0].confidence) + '% confidence)');
    });
});

```

As you can see, this example looks very similar to the previous examples, however there are some very important differences. Let's go through these one at a time.

- ➊ Instead of demanding that the Speech API recognize some text immediately, we "start recognizing", which kicks off a streaming version of recognition.
- ➋ The result of this `startRecognition` method is a "long-running operation", which will emit events as the recognition process continues.
- ➌ When the operation completes, it returns the recognized transcript as the result.

If you run this code, you should see the exact same result as before:

#### **Listing 16.6. The same output, recognized as a stream**

```
This audio file says: "how old is the Brooklyn Bridge" (with 98% confidence)
```

Now that you've seen how recognition works, let's dig a bit deeper into some of the customization possible when trying to recognize different audio streams.

## **16.4 Hinting with custom words and phrases**

Since language is an ever-evolving aspect of communication, it's important to recognize that new words will be invented all the time. This means that sometimes the Cloud Speech API might not be "in the know" about all the cool new words or slang phrases, and may end up guessing wrong. This is particularly true as we invent new interesting names for companies (e.g., Google was a misspelling of "Googol"), so to help the Speech API better recognize what was said, you're actually able to pass along some suggestions of "valid phrases" that can be added to the APIs ranking system for each request. To demonstrate how this works, let's see if we can throw in a new suggestion that might make the Speech API misspell "Brooklyn Bridge". In the example below, we update our `config` with some additional context and then re-run the script.

#### **Listing 16.7. Speech recognition with suggested phrases**

```

const config = {
  encoding: 'FLAC',
  sampleRate: 16000,
  verbose: true,
  speechContext: { phrases: [ ➊
    "the Brooklynne Bridge"
  ] }
};

```

- ➊ Here we suggest the phrase "the Brooklynne Bridge" as a "valid phrase" for the Speech API to use when recognizing.

If we were to run this script, you'd see that the Speech API does indeed use the alternate spelling provided.

**NOTE**

As with all of the ML APIs we've learned about so far, results will vary over time as the underlying systems "learn" more and get better. If the output of the code is not exactly what you see, don't worry! It just means that the API has improved since this writing.

**Listing 16.8. Output of running our code showing the phrase took precedence**

```
> This audio file says: "how old is the brooklynne bridge" (with 90% confidence)
```

Notice, however, that the confidence is somewhat lower than before. This is because there are two relatively high-scoring results that would've come back: "Brooklyn Bridge" and (our suggestion) "brooklynne bridge". These two competing possibilities make the Speech API less confident in its choice, although it is still pretty confident (90%).

In addition to custom words and phrases, the Speech API provides a profanity filter to avoid accidentally displaying potentially offensive language. By setting the `profanityFilter` property to `true` in the configuration, any recognized profanity will be "starred out" except for the first letter (e.g., "s\*\*\*"). Now that you have a grasp of some of the advanced customizations, let's talk briefly about how much this will cost you.

## 16.5 Understanding pricing

Following the pattern of the rest of Google Cloud Platform, the Cloud Speech API will only charge you for what you use. In this case, the measurement factor is the length of the audio files that you send to the Speech API to be recognized, measured in minutes. The first 60 minutes per month are part of the free-tier, where you won't be charged at all, and beyond that it costs 2.4 cents per minute of audio sent.

Since there's an initial overhead cost involved, the Cloud Speech API currently rounds audio inputs up to the nearest 15-second increment and bills based on that (so the actual amount is 0.6 cents per 15-seconds). This means that a 5 second audio file is billed as 1/4 minute (\$0.006 USD), and a 46 second audio field is billed as a full minute (\$0.024 USD). Finally, let's move on to a possible use of the Cloud Speech API: generating hash tag suggestions for InstaSnap videos.

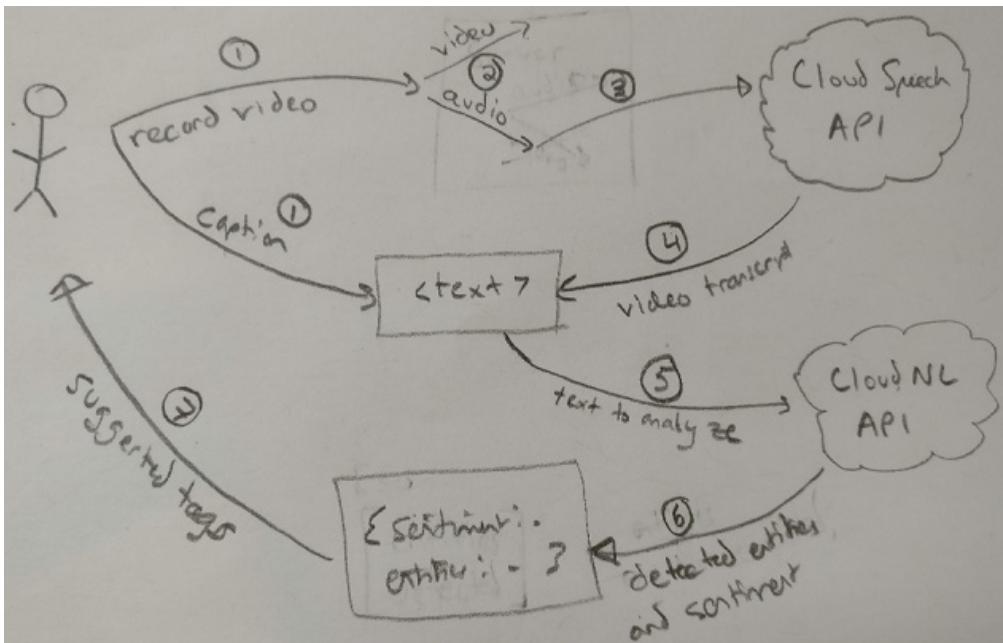
## 16.6 Case study: InstaSnap video captions

As you may recall, InstaSnap is our example application where users can post photos and captions, and share those with other users. Let's imagine that InstaSnap has added the ability to record videos and share those as well.

In the last chapter on the Cloud Natural Language API, we saw how we could generate some suggested hash tags based on the caption of a photo. Wouldn't it be neat if we could suggest tags based on what's being said in the video? From a high level, we'll

still rely on the Cloud Natural Language API to recognize any entities being discussed (if you aren't familiar with this, check out the chapter on the Cloud Natural Language API), but we'll need to pull out the audio portion of the video and figure out what's being said, and come back with suggested tags. The following diagram shows the flow of each step, starting at a recorded video and ending at suggested tags.

**Figure 16.4. Overview of our hash tag suggestion system**



So what does each step actually do?

- ① First, the user records and uploads a video (and types in a caption)
- ② Here, our servers would need to separate the audio track from the video track (and presumably format into a normal audio format).
- ③ Next, we need to send the audio content to the Cloud Speech API for recognition.
- ④ The Speech API should return a transcript as a response, which we then combine with the caption that was set in step 1.
- ⑤ We then take all of the text (caption and video transcript), and send these to the Cloud Natural Language API.
- ⑥ The Cloud NL API will recognize entities and detect sentiment from the text, which we can process to come up with a list of suggested tags.
- ⑦ Finally, we send the suggested tags back to the user.

If you read the chapter on natural language processing, steps 5, 6, and 7 should look very familiar. They're actually the exact same ones! So let's focus on the earlier (1 through 4) steps that are specific to recognizing the audio content and turning it into text. Let's start by writing a function that will take a video buffer as input, and return a

JavaScript promise for the transcript of the video. We'll call this function `getTranscript`.

### **Listing 16.9. Defining a new `getTranscript` function**

```
const Q = require('q');          ①
const speech = require('@google-cloud/speech')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

const getTranscript = (videoBuffer) => {
  const deferred = Q.defer();    ②

  extractAudio(videoBuffer).then((audioBuffer, audioConfig) => { ③
    const config = {
      encoding: audioConfig.encoding, // e.g., 'FLAC'
      sampleRate: audioConfig.sampleRate, // e.g., 16000
      verbose: true
    };
    return speech.startRecognition(audioBuffer, config);
  }).then((result) => {
    const operation = result[0];
    operation.on('complete', (results) => {
      const result = results[0];
      const transcript = result.confidence > 50 ? result.transcript : null;
      deferred.resolve(transcript);
    });
    operation.on('error', (err) => { ④
      deferred.reject(err);
    });
  }).catch((err) => {           ④
    deferred.reject(err);
  });

  return deferred.promise;      ⑤
};
```

- ① Here we're relying on an open-source promise library called Q. You can install it with `npm install q`.
- ② We use `Q.defer()` to create a deferred object which we can resolve or reject in other callbacks.
- ③ We're assuming that there is a preexisting function called `extractAudio` which returns a promise for both the audio content as a buffer and some configuration data about the audio stream (such as the encoding and sample rate).
- ④ If there are any errors, we simply "reject" the deferred object, which will trigger a failed promise.
- ⑤ Here we return the promise from the deferred object, which will be resolved if everything works and rejected if there's a failure.

Now we have a way to grab the audio and recognize it as text, so we can use that along with the code from [chapter 15](#) to do the rest of the work. To make things easier, let's generalize the functionality from [chapter 15](#) and write a quick method that will take any given content and return a JavaScript promise for the sentiment and entities of that content. We'll call this method `getSentimentAndEntities`. (See [chapter 15](#) for more

background if this is new to you.)

#### **Listing 16.10. Defining a getSentimentAndEntities function**

```
const Q = require('q');
const language = require('@google-cloud/language')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

const getSentimentAndEntities = (content) => {
  const document = language.document(content); ①
  const config = {entities: true, sentiment:true, verbose: true};
  return document.annotate(config).then( ②
    return new Q(data[0]); // { sentiment: {...}, entities: [...] } ③
  );
};
```

- ① We start by creating a NL document.
- ② Then we annotate the document with sentiment and entities found.
- ③ Finally, we return a promise whose value will have properties for both the sentiment and the entities found in the text provided.

Now we have all the tools we need to put our code together. To wrap everything up, let's build the final handler function that accepts a video that has properties for the video buffer and caption) and prints out some suggested tags. The function that comes up with the suggested tags (`getSuggestedTags`) is the same one that we wrote in [chapter 15](#).

#### **Listing 16.11. Defining a getSuggestedTags function**

```
const Q = require('q');
const authConfig = {
  projectId: 'your-project-id',
  keyFilename: 'key.json'
};
const language = require('@google-cloud/language')(authConfig);
const speech = require('@google-cloud/speech')(authConfig);

const handleVideo = (video) => {
  Q.allSettled([
    getTranscript(video.buffer).then((transcript) => { ①
      return getSentimentAndEntities(video.transcript);
    }),
    getSentimentAndEntities(video.caption) ③
  ]).then((results) => {
    let suggestedTags = [];
    results.forEach((result) => { ④
      if (result.state === 'fulfilled') {
        const sentiment = result.value.sentiment;
        const entities = result.value.entities;
        suggestedTags = suggestedTags.concat(getSuggestedTags(sentiment, ⑤
          entities));
      }
    });
  });
};
```

```

    }
  });
  console.log('The suggested tags are', suggestedTags);
  console.log('The suggested caption is',
    ''' + caption + ' ' + suggestedTags.join(' ') + "'");
});
}

```

- ➊ We're relying on Q's `allSettled` method which waits until all promises have either succeeded or failed. This means that we should end up with lots of results, some in a `fulfilled` which means we can use those results.
- ➋ Here we create a promise to return the sentiment and entities based on the audio content in the uploaded video.
- ➌ Next we create a promise to return the sentiment and entities from the caption set when uploading the video.
- ➍ After all the results are settled (via `Q.allSettled`), we iterate over each and use only those that resolved successfully.
- ➎ Based on the sentiment and entities from the text, we use the function we built in [chapter 15](#) to come up with a list of suggested tags, and add them to the list.

And that's all there is to it! You now have a pipeline that takes an uploaded video and returns some suggested tags based on both the caption set by the user **and** the audio content in the recorded video. Additionally, since we did each suggestion separately, if the caption was happy and the audio sounded sad, you might have a mixture of happy tags ("#yay") and sad ones ("#fail").

## 16.7 Summary

- Speech recognition takes a stream of audio and converts it into text, which can be deceptively complicated due to things like the McGurk Effect.
- Cloud Speech is a hosted API that can perform speech recognition on audio files or streams.
- If an audio stream contains lots of jargon or industry-specific terminology, you can provide custom words as hints for Cloud Speech to rely on.
- Cloud Speech charges based on the number of minutes of audio sent to be processed, rounding to 15-second increments.

# 17

## *Cloud Translation: Multi-language machine translation*

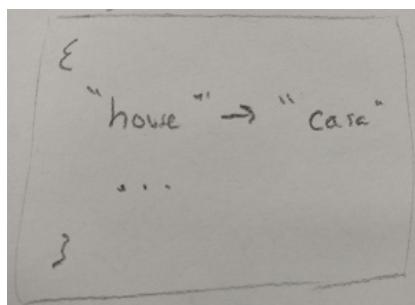
### **This chapter covers:**

- An overview of machine translation
- How the Cloud Translation API works
- How Cloud Translation pricing is calculated
- An example of translating image captions

### **17.1 What is machine translation?**

If you've ever tried to learn a foreign language, you'll probably recall that it starts out easy with vocabulary problems where you memorize the foreign equivalent of a word you know. In a sense, this is just memorizing a simple map going from language A to language B (e.g. `houseInSpanish = spanish['house']`).

**Figure 17.1. Mapping of English to Spanish words**



While this is definitely challenging for us humans, computers are good at memorizing stuff, so this wouldn't be a hard problem to solve. It turns out that this memorization problem is nowhere near as challenging as a *true* understanding of the language where you take one "conceptual representation" in one language and translate it to another, phrasing things in a way that actually "sounds like" the new language. And this is the exact problem machine translation aims to solve.

Unfortunately, things are even worse. It turns out that us humans are pretty strange in how we developed language. Much like how cities tend to grow from a small city center and expand, it's believed that languages started with simple words and grew from there, evolving over hundreds and hundreds of years into the languages you know today. For example, if you were to hope in a time machine back to the middle ages, it's unlikely that you'd understand anyone at all!

**NOTE**

Obviously there are exceptions such as the Esperanto language which was designed fully rather than evolving (much like Amsterdam was designed all the way to the suburbs rather than expanding from a single planned city center), however this appears to be the exception rather than the rule.

All of these issues make this problem particularly difficult. There are the languages with extraordinarily high levels of complexity, where you use different words depending on who you're talking to (e.g. Japanese), slang expressions that are so ubiquitous that they're used everywhere, new expressions that are on their way to being ubiquitous, words that don't have an exact translation in another language, different dialects of the same language (e.g., English in Britain versus America) and finally, as with anything involving us humans, ambiguity of the overall meaning — which has nothing to do with computers!

As you can see, what started as a simple mapping from words in one language to another, has suddenly entered into a world where it's not even clear to us as humans what the right answer might be. Let's look at a specific example of some of the strangeness of language.

As an English speaker, think about prepositions (*about, before, on, beside*) and when you might use them. Is there a difference between being *on* an airplane versus *in* an airplane? Is one more correct than the other? Do they convey different things?

Obviously this is open to interpretation, but to me, being "on an airplane" implies that the airplane is in motion and I'm really talking about being "on an airplane *trip*" or "on an airplane *flight*". Being "in an airplane" conveys the containment *inside* of the airplane, where I might say it this way when someone asks why my cell phone reception is so bad. But the point would be that I'm stationary while *inside* this airplane.

What's interesting is that the distinction is so subtle, that if said with a perfect American accent, we probably wouldn't consciously notice the difference but it might "sound a little off". And we're talking about a single letter difference in two prepositions that might translate to the same word in other languages (e.g. in Spanish,

as *en*).

The simple fact that there is an entire StackExchange community to answer questions about grammar, usage, and other aspects of the English language hopefully goes to demonstrate that even today we haven't quite "figured out" all the aspects of language, let alone how to seamlessly go from one to another.

Now that you have a grasp for the extent of the problem we're trying to solve and how complex it is, let's talk about how machine translation works and how the Cloud Translation API works under the hood.

## 17.2 How does the Translation API work?

So if this problem is so impossible, how does Google Translate work? Is it any different from the Cloud Translation API?

Let's start by looking at this question of resolving such a complicated problem of understanding vocabularies and grammatical rules. One way to attack this problem would be to try to "teach" the computer all of the different word pairs (e.g., `EnglishToSpanish('home') == 'casa'`) and grammatical words ("English uses Subject Verb Object (SVO) structure"). However, as we discussed above, not only is language extraordinarily complex, with exceptions for almost every rule, but it is constantly evolving meaning that you'd be chasing a moving target. This means that this method, while it might work with enough effort, certainly isn't going to be a very "scalable" way of solving the problem (though it is certainly better than hiring humans to translate text for you).

Another way (and the way that Google Translate actually does it for many languages) uses something called "Statistical Machine Translation" (abbreviated as SMT). Fundamentally, SMT relies on the same concept as the Rosetta stone, which was a stone that had the same thing written on it in both Greek and Egyptian, which meant that if one language was understood (which Greek was), the other language could be "figured out" (hieroglyphics were still somewhat of a mystery). In the case of SMT, rather than Greek and Egyptian on a single stone, the algorithm relies on millions and millions of documents that have equivalents in at least one language pair (e.g., English and Spanish).

In short, SMT looks at these millions of documents that have translations in several languages (created by human translators) and identifies common patterns across the two documents, ones that occur between the two (the translation and the original text) that are very unlikely to be "just pure luck". The assumption is that if you see these patterns often, it's very likely that you've found a match between a phrase in the original text and the equivalent phrase in the translated text. As you might guess, the larger the overlap, the closer you get towards a true human translation, given the "training data" was translated by a human.

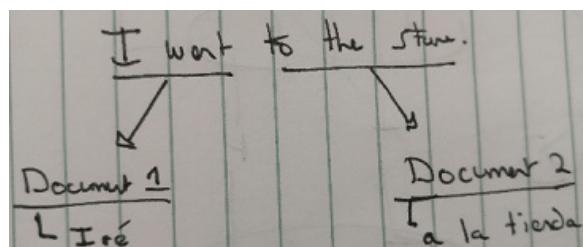
To make this more concrete, imagine a trivial example where you have lots of books in both English and Spanish. You see the word "house" over and over in the English translation and the word "casa" in the Spanish appears similarly frequent. As you see

more and more of this pattern (where you have exactly the same occurrences of "house" in English and "casa" in Spanish), it becomes very likely that when someone wants to know the Spanish equivalent of the word "house", you can guess that the "most correct" answer is "house". As you continue to train your system on these new inputs and it identifies more and more patterns, it's possible that you'll get closer and closer to a true human translation.

As you might guess, there's an obvious drawback to this: sentences are translated piece by piece rather than as a whole. If you happen to ask for a translation of an exact sentence that the SMT system has already seen before, obviously you'll get an exact (human) translation, however it's unlikely that you'll have that exact same input, and far more likely that your translation will be made up multiple translations covering several phrases in the sentence. Sometimes this works out fine, but often it means that the translation might come across as "choppy" due to drawing translations of phrases from different places. It also means that if you're translating a word that hasn't been seen before in any of the training documents, you're sort of "out of luck" so to speak.

For example, translating "I went to the store" from English to Spanish comes across pretty reasonably. Chances are the entire sentence was in a document somewhere, but even if not, "I went" and "to the store" are likely in those documents, and combining them is pretty natural.

**Figure 17.2. Translating based on multiple documents**



But what about a more complex sentence?

Probleme kann man niemals mit derselben Denkweise lösen, durch die sie entstanden sind.

Translating this sentence from German to English comes out as:

No problem can be solved from the same consciousness that they have arisen.

I don't know about you, but that sentence feels a bit unnatural to me, and is likely the result of pulling several phrases from several places, rather than looking at the sentence as a whole.

This led Google to focus on some of the newer areas of research, including the same technology underlying the Natural Language API, and the Vision API: neural networks.

While this machine learning is still an area of active research, and you could write an

entire book on neural networks and applied machine learning, I won't go into too many of the specifics except to say that Google's Neural Machine Translation (GNMT) system relies on a neural network, uses custom Google-designed and built hardware to keep things fast, has a "coverage penalty" to avoid the neural network from "forgetting" to translate some parts of the sentence, and many more technical optimizations to minimize the overall cost of training and storing the neural network handling translation.

What this means is that you end up with "smoother" translations. For example, that same sentence in German above becomes much more readable:

Problems can never be solved with the same way of thinking that caused them.

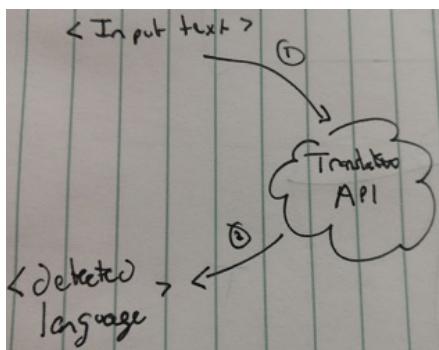
As of this writing, Google Translate and the Cloud Translation API both use neural networks for translating common languages (between English and French, German, Spanish, Portuguese, Chinese, Japanese, Korean, and Turkish — a total of 8 language pairs), and rely on SMT ("the old way") for the other language pairs.

Now that you understand a bit of what's happening under the hood, let's get down to the real business of seeing what this API can do and using it with some code, starting with something easy: language detection.

### 17.3 Language detection

The simplest application of the Translation API is looking at some input text, and figuring out what language it is. While some other APIs require you to start by storing information, the Cloud Translation API is completely stateless, meaning that you store nothing and send all the information required in a single request.

**Figure 17.3. Language detection overview**

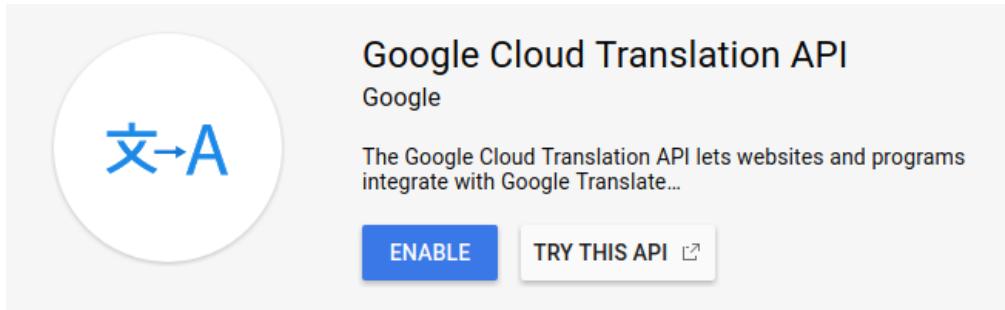


As you might guess, sometimes this is easy (in the example of the German sentence above) and sometimes this isn't quite as easy (particularly with languages that are very similar, or sentences that are very short). For example, "No." is a sentence in English, but it's also a sentence (with the same meaning) in Spanish! This means that in general, very short sentences should be avoided.

Let's start by looking at a few examples and detecting the language of each.

The first thing to do is enable the Translation API as you may recall from several other APIs. To do this, type "Cloud Translation API" into the main search box at the top of the page. This query should come up with just one result, and clicking on that should bring you to a page with a big "Enable" button. Once you click that, the API will be enabled and the code samples should work as expected.

**Figure 17.4. Enable button for the Cloud Translation API.**



Before we write any code though, we'll first need to install the client library. We can do this using `npm` by running `npm install @google-cloud/translate@1.0.0`. Once that's done, we can dive right in with some language detection samples.

#### **Listing 17.1. Detecting the language of input text**

```
const translate = require('@google-cloud/translate')({
  projectId: 'your-project-id',
  keyFilename: 'key.json' ①
});

const inputs = [
  ('Probleme kann man niemals mit derselben Denkweise lösen, ' +
   'durch die sie entstanden sind.'),
  'Yo soy Americano.'
];

translate.detect(inputs).then((response) => {
  const results = response[0];
  results.forEach((result) => {
    console.log('Sentence: "' + result.input + '"',
               '\n- Language:', result.language,
               '\n- Confidence:', result.confidence);
  });
});
```

When you run this, you should see something that looks like the following:

#### **Listing 17.2. Detected languages with a confidence level**

```
> Sentence: "Probleme kann man niemals mit derselben Denkweise lösen, durch die sie
entstanden sind."
- Language: de
```

```

- Confidence: 0.832740843296051
Sentence: "Yo soy Americano."
- Language: es
- Confidence: 0.26813173294067383

```

There are few important things to notice here.

First, and most importantly for our purposes, the detections were accurate. That is, the German sentence was identified as de (the language code for German), and likewise for the Spanish sentence. Clearly this algorithm does a few things right!

Second, in addition to the result of the detected language, there's a "confidence" level associated with the result. Just like many of the other machine learning APIs you've read about, this confidence expresses numerically (in this case, from 0 to 1) how confident the algorithm is that the result is correct. This gives you some indication of how much you should trust the result, with higher scores being more trustworthy.

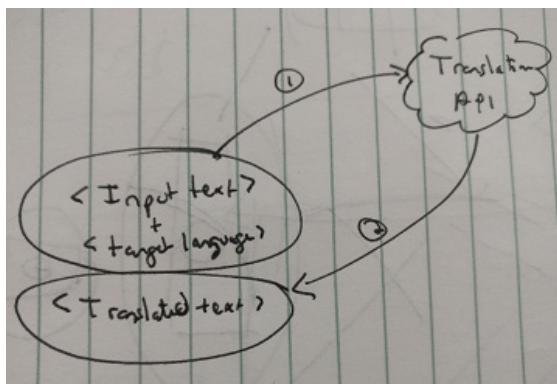
Finally, notice that the confidence score for the German sentence is *much* higher than that of the Spanish sentence. This could be for quite a few reasons, but one of them we've mentioned already: length. The longer the sentence, the more input the algorithm has to work with, which leads to a more confident result. In a short sentence that means "I'm American", it's hard to be very confident in the detected result. Spanish clearly scored the highest, but with only three words, it's difficult to say with the same confidence as the longer sentence.

If you try running this code yourself and get confidence numbers that are different from the ones you see above, don't worry! The reason for this is that the underlying machine learning algorithms are changing and improving over time. This means that the results you get one day may be slightly different later on, so make sure that you treat these numbers with a bit of flexibility.

Now that you've seen how you can detect the language of some content, let's get into the real work: translating text.

## 17.4 Text translation

Similar to detecting the language of text, translating text involves the same process. Given some input text and a target output language, the API will return the translated text (if it can). Similar to the language detection functionality, translating text is stateless as well, where you send everything necessary to translate your inputs in the initial request.

**Figure 17.5. Translating text overview**

Notice that you only specify the language you want along with the input text, but you **don't** specify the language of the input text! Technically, you can tell the Translation API the source language, however if you leave it blank (which many often do), it automatically detects the language (for free) as part of the translation.

Given that, let's take those same examples from before and try to translate them all to English (en).

### **Listing 17.3. Translating from multiple languages to English**

```

const translate = require('@google-cloud/translate')({
  projectId: 'your-project-id',
  keyFilename: 'key.json' ①
});

const inputs = [
  ('Probleme kann man niemals mit derselben Denkweise lösen, ' +
   'durch die sie entstanden sind.'),
  'Yo soy Americano.'
];

translate.translate(inputs, {to: 'en'}).then((response) => {
  const results = response[0];
  results.forEach((result) => {
    console.log(result);
  });
});
  
```

When you run this, you'll see a simple bit of output with just the sentences translated:

### **Listing 17.4. English translations of the input text**

```

> No problem can be solved from the same consciousness that they have arisen.  
I am American.
  
```

Notice that there are a few things missing that you saw previously when detecting the language.

First, there's no "confidence" or "score" associated with the translation. This means that, unfortunately, you aren't able to express how confident you are that the translated text is accurate. This is due to the fact that it's actually pretty hard to decide what the "right" answer is. While you can say with some level of confidence that a given chunk of text is in a specific language (because it presumably was written by someone intending a language), the meaning in another language might vary depending on who is doing the translating. Thanks to this ambiguity, a confidence rating wouldn't be all that useful.

You might also notice that the "source language" is not coming back as a result. In this case, if you want that result, you can look at the raw API response and it will show you the "detected language". The following snippet shows how you can get that if needed:

#### **Listing 17.5. Detecting source language when translating**

```
const translate = require('@google-cloud/translate')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
}); ①

const inputs = [
  ('Probleme kann man niemals mit derselben Denkweise lösen, ' +
   'durch die sie entstanden sind.'),
  'Yo soy Americano.'
];

translate.translate(inputs, {to: 'en'}).then((response) => {
  const results = response[1].data.translations; ②
  results.forEach((result, i) => {
    console.log('Sentence', i+1,
               'was detected as', result.detectedSourceLanguage); ③
  });
});
```

When you run this example, you should see output that shows the detected languages.

#### **Listing 17.6. Showing the detected languages during translation**

```
> Sentence 1 was detected to be de
Sentence 2 was detected to be es
```

As you can see, the core features of the Translation API are quite straightforward: take some text, get back a detected language, take some text and a target, get back a translation to the target.

Now let's look briefly at the pricing considerations to take into account.

## **17.5 Understanding pricing**

As with an Cloud API, you tend to pay for only what you use. In the Translation API, this is no different.

When you are translating or detecting languages, you're charged based on the number of "characters" you send to the API (at a rate of \$20 USD per million). The question then becomes, "What is a character?"

In the case of the Translation API, billing is focused on character as a "business concept" rather than a technical one. In other words, this means that even if a given character is multiple bytes (such as a Japanese character), you're only charged for that one "character". If you're familiar with the underlying technology of character encoding, the definition of a "character" here is a "code point" for your given encoding.

Another open question is, "What about whitespace?" The simple answer is that since whitespace characters are necessary to understand the breaks between words, they are charged for like any other character (or "code point"). So for the purposes of billing, "Hello world" is treated as 11 characters due to the space between the two words rather than the 10 non-whitespace characters.

Now let's move onto some more real-world stuff, looking at a specific example of how you might integrate the Translation API into an application.

## **17.6 Case study: Translating InstaSnap captions**

As you may recall, InstaSnap is our sample application that allows users to post photos and captions to share with the rest of the world. But as it turns out, not everyone is from the US! In particular, there are several celebrities that are famous worldwide, and those fans want to know what the celebrities are saying in their captions! Let's see if we can use the Translation API to fix this.

Breaking the problem down a bit more, it seems that what we really want is to detect if the language of a given caption isn't the same as the language of the user. If it isn't, we may want to translate it.

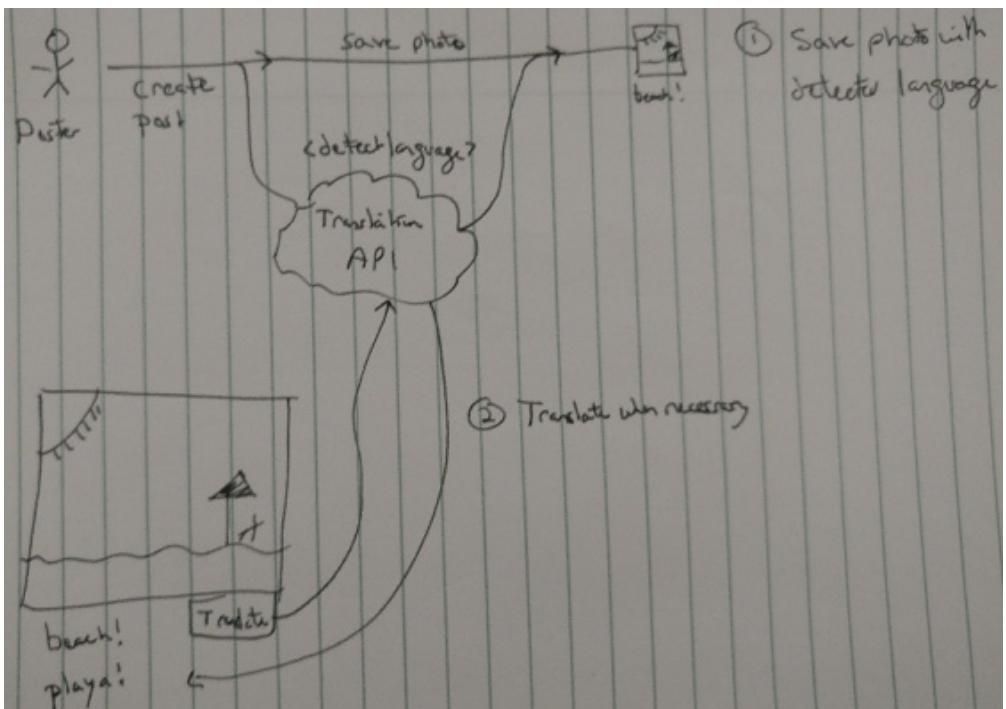
This means that the simple solution to this is to automatically attempt to translate the text of the language to the language of the user. This is a solution we might want to call "automatic translation at view-time".

The problem with this solution is that it sounds like it'll get expensive. For starters, it's unlikely that every one of the captions needs translating! Beyond that, it's unlikely that even if the caption needed translating, the user might not be interested in that content.

As a result, we should change our design a little bit. Instead of trying to translate everything, we could detect the language of text when the caption is created and store that on the post. We can also assume that we know the "primary language" of every user since they chose one when they signed up for InstaSnap.

If we have the detected language at "post-time", we can compare it to the viewer's language, and if they are different display a button saying "Translate to English" (localized for whatever the viewer's primary language is). Then when the viewer clicks the button, we can request a translation to the viewer's primary language.

**Figure 17.6. Overview of the flow when posting and viewing on InstaSnap**



Let's start by writing some code at the upload time to store the detected language. We would simply call this method after the photo was uploaded.

#### **Listing 17.7. Detecting and saving the language of a caption**

```
const translate = require('@google-cloud/translate')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

const detectLanguageOfPhoto = (photo) => {
  translate.detect(inputs).then((response) => {
    const result = response[0][0];
    if (result.confidence > 0.1) {
      photo.detectedLanguage = result.language;
      photo.save();
    }
  });
};
```

- ➊ Given a "saved photo", detect the language and save the result.
- ➋ If the confidence is pretty poor, don't do anything.

Next, we can write a function to decide whether or not to display the "translate" button.

**Listing 17.8. Determining whether to display a translate button**

```
const shouldDisplayTranslateButton = (photo, viewer) => {
  if (!photo.detectedLanguage || !viewer.language) { ①
    return false;
  } else {
    return (photo.detectedLanguage != viewer.language); ②
  }
}
```

- ① If the detected language is empty, we can't do any translating. Similarly, without a target language to translate to, we can't do any translating.
- ② If the two languages are different, this evaluates to true.

Finally, at view time, we can write a function that will do the actual translating work.

**NOTE**

This code won't run as it uses several "fake" components. It's here simply to demonstrate how you would wire everything together.

**Listing 17.9. Run-time code to handle optional translation of captions**

```
const translate = require('@google-cloud/translate')({
  projectId: 'your-project-id',
  keyFilename: 'key.json' ①
});

const photoUiComponent = getCurrentPhotoUiComponent();
const photo = getCurrentPhoto();
const viewer = getCurrentUser();
const translateButton = new TranslateUiButton({ ②
  visible: shouldDisplayTranslateButton(photo, viewer),
  onClick: () => {
    photoUiComponent.setCaption('Translating...'); ③
    translate.translate(photo.caption, {to: viewer.language})
      .then((response) => {
        photoUiComponent.setCaption(response[0][0]); ④
      })
      .catch((err) => {
        photoUiComponent.setCaption(photo.caption); ⑤
      });
  }
});
```

- ① We're using a "fake" concept of a Translate button that we can operate on.
- ② We say whether the button is visible by using our previously written function.
- ③ Before we make the API request, we set the caption to "Translating..." to show that we're doing some work under the hood.
- ④ If we get a result, we set the photo caption to the translation result.
- ⑤ If there are any errors, we just reset the photo caption as it was.

## 17.7 Summary

- Machine translation is the ways computers try to translate from one language to another, with as much accuracy as possible.
- Until recently, most translation was done using mappings between languages, but the quality of the translations can sometimes seem a bit "mechanical".
- Cloud Translation is a hosted API that can translate text between languages, using neural machine translation, a specialized form of translation that uses neural networks instead of direct mappings between languages.
- Cloud Translation charges prices based on the number of characters sent to be translated, where a "character" is defined as a "code point".

## 17.8 Online appendices

- Google's Neural Machine Translation System ([arxiv.org/pdf/1609.08144v2.pdf](https://arxiv.org/pdf/1609.08144v2.pdf))

# 18

## *Cloud Machine Learning Engine: Managed Machine Learning*

### **This chapter covers:**

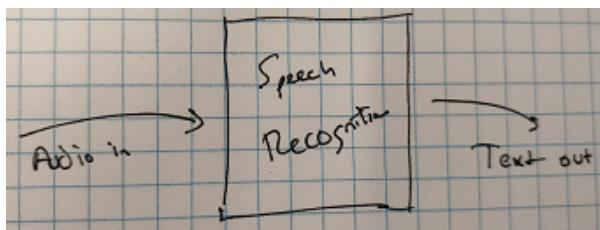
- What is machine learning?
- What are neural networks?
- What is TensorFlow?
- What is Cloud ML Engine?
- How is Cloud ML Engine different from other ML APIs?
- Creating a deploying your own ML model

### **18.1 What is machine learning?**

Although we've explored various machine learning APIs already, we've so far focused only the real-world applications and not talked at all about how they work under the hood. In this chapter, we're going to look inside and move beyond these pre-programmed ML problems.

Before we go any further, it's important to note that machine learning and artificial intelligence are enormous topics with quite a lot of ongoing research, and this chapter is in no way comprehensive to the topic. While we'll try to take a tour through some of the core concepts of ML and demonstrate how to write a very simple bit of ML code, we'll gloss over the majority of the mathematical theory and most of the actual calculations. If you're passionate about machine learning, you should absolutely explore other books that provide far more information about the fundamentals of machine learning. So with that out of the way, let's get to exploring what exactly is going on inside these ML APIs.

**Figure 18.1. Machine learning (speech recognition) as a black-box system**



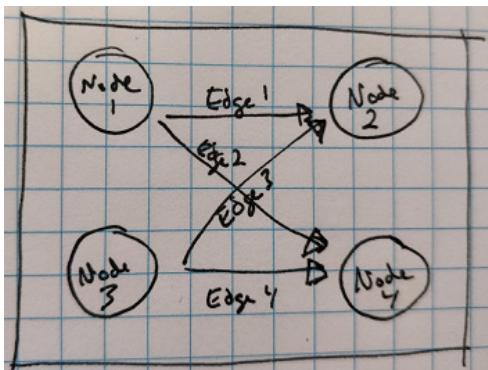
While there are many nuances to the different types of ML, we generally define machine learning as the idea that a system that can be trained with some data and then make predictions based on that training. The key distinguishing factor here is that this is very different from how we typically build software. In general if we want a computer to do something for us, a programmer translates that goal into explicit instructions or "rules" for the computer to follow. Machine learning is the idea of the computer figuring out the rules on its own rather than by having someone teach them explicitly.

For example, if you wanted the computer to know how to double a value you'd take that goal ("multiply by two") and write the program `console.log(input * 2)`. Using machine learning, you'd instead show the system a bunch of inputs and desired outputs (such as  $2 \rightarrow 4$  and  $40 \rightarrow 80$ ), and using those examples the system would be responsible for figuring out the rules on its own. Once it's done that, it can make predictions about what  $5 * 2$  is without having seen that particular example before by assuming 5 is the input and making a prediction about  $5 \rightarrow ?$ .

There are lots of different methods that we can build systems capable of "learning", but the one that has gotten the most interest recently is actually modeled after the human brain. Let's take a quick look at this method and how it works at a fundamental level.

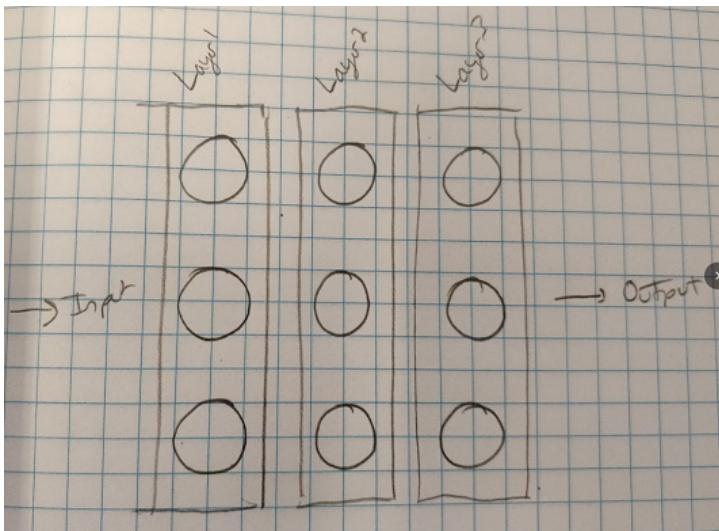
### 18.1.1 What are neural networks?

One of the fundamental components in modern machine learning systems is called a "neural network". These networks are the pieces that do all of the heavy lifting of both "learning" and "predicting", and can vary in complexity from super simple (like the one drawn below) to extremely complex (like your brain).

**Figure 18.2. Neural network as a directed graph**

A neural network is a directed graph containing a bunch of nodes (the circles) connected to one another along edges (the lines with arrows), where each line has a certain weight. The "directed" part means that things flow in a single direction, indicated by the way the arrow is pointing. These weights determine how much of an input signal is transmitted into an output signal, or in other words, how much the value of one node affects the value of another node that it's connected to.

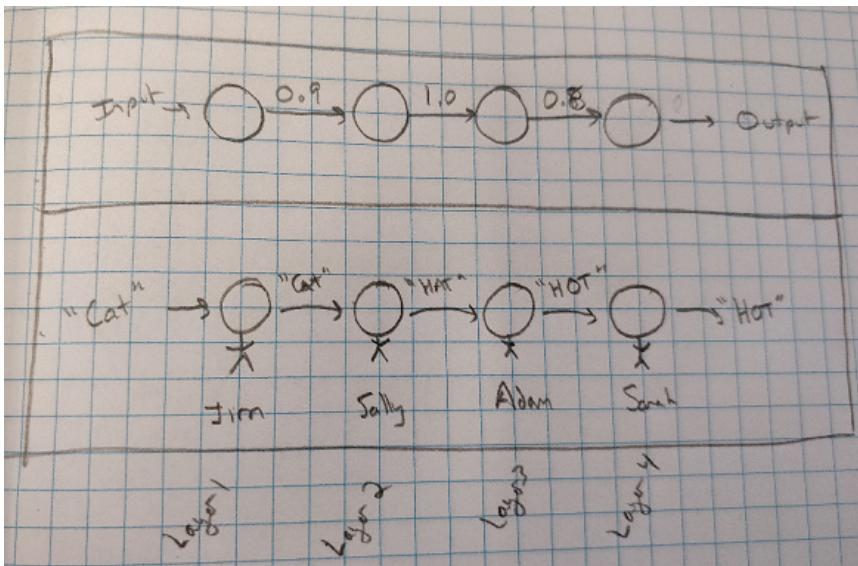
The nodes themselves are organized into layers, with the first layer accepting a set of input values and the last layer representing a set of output values. The neural network works by taking these input values and using the weights to pass those values along from one layer to another until they come out on the other side.

**Figure 18.3. The layers of a neural network**

If you've ever played the game "telephone" where everyone whispers a word down a long line, you're familiar with how easily an input can be manipulated bit by bit and

end up completely different. The game of telephone is sort of like a neural network with lots of layers, each consisting of a single node, where each node represents a person in the chain. The weights on the edges between each node are sort of like how well the next person can understand the previous person's whispers.

**Figure 18.4. A game of "telephone" like a neural network's transformations**



You can "train" a neural network by taking an input, sending it into the network to get an output, and then adjusting the weights based on how far off the actual output was from the expected output. Using our analogy of the game of telephone, this is a bit like seeing that an input of "Cat" yielded an output of "Hot", and suggesting that Adam (the one who took "Hat" and said "Hot") be more sensitive to his vowel sounds. If you make lots and lots of these adjustments for lots and lots of example data points, lots of times over and over, it turns out that the network can actually get pretty good at making predictions for data that it hasn't seen before.

In addition to varying the weights between nodes throughout training, you can also adjust values that are external to the training data entirely. These adjustments, called *hyperparameters*, are used to tune the system for a specific problem in order to get the best predictive results. We won't get into much more detail about hyperparameters, but you should certainly know that they exist and that they typically come from heuristics as well as trial and error.

This is by no means a complete course on neural networks, and neural networks aren't even the only way to build machine learning systems, but as long as you understand the fundamental point (something takes input, looks at output, and makes adjustments), you're in good shape to follow along with the rest of this chapter.

Since understanding the concepts doesn't really help us do anything, we need to learn

how to actually do real things with these machine learning systems. That is, how we do take this concept of a self-adjusting system and do something like figure out whether a cat is in an image? To do this, there are many libraries out there that make dealing with neural networks and other machine learning concepts much easier than the diagrams shown above. One of these which we'll discuss for its use with Cloud ML Engine is called TensorFlow.

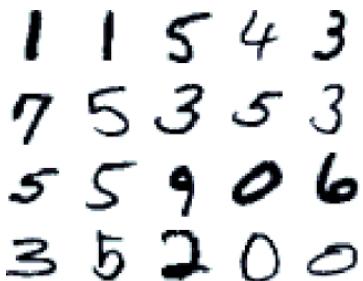
### **18.1.2 What is TensorFlow?**

TensorFlow is a machine learning development framework that makes it a bit easier to express machine learning concepts (and the underlying math) in code rather than in scary mathematical equations. It provides abstractions to keep track of the different variables, utilities like matrix multiplication, neural network optimization algorithms, and various different estimators and optimizers that give you control over how all of those adjustments are applied to the system itself during the learning period.

In short, TensorFlow acts as a way of bringing all the fancy math of neural networks and other machine learning algorithms into code (in this case, Python code). For the purposes of this chapter, we're not going to get into the details of how to do complex machine learning with TensorFlow (since there are entire books devoted to this), but in order to move forward we need to be a bit familiar with TensorFlow, so let's look at a simple TensorFlow script that can actually make some predictions. Since we're not trying to teach you how to write your own TensorFlow scripts, don't be scared if you don't follow exactly what's happening here. The point is just to give you a feel for what TensorFlow looks like so it doesn't paralyze you with confusion.

To demonstrate how TensorFlow works, we'll use a sample data set called MNIST which is a collection of images represented numbers written by hand. Each image is a square of 28 pixels, and each data point has the image itself as well as the number which is represented in the image. These images are typically used as a beginner problem in machine learning since there are both hand-written numbers to use for training and a separate set to use when testing how well the model does using data it hasn't seen before. All of the images look something like those in the grid below.

**Figure 18.5. MNIST sample hand-written numbers**



Since TensorFlow makes it really easy to pull in these sample images, we'll use them to build a model that can take a similar image and predicting what number is written in the image. In a way, we're building a super slimmed-down version of Cloud Vision's

text recognition API which we learned about in [chapter 14](#). Our script will train on the sample "training" data and then use the "evaluation" data to test out how effective our model is at identifying a number from an image that wasn't used during the training.

### **Listing 18.1. Example TensorFlow script that recognizes hand-written numbers**

```
import tensorflow as tf ①

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data/', one_hot=True) ②

# Learning model info
x = tf.placeholder(tf.float32, [None, 28*28]) ③
weights = tf.Variable(tf.zeros([28*28, 10]))
bias = tf.Variable(tf.zeros([10]))
y = tf.nn.softmax(tf.matmul(x, weights) + bias)

# Cross entropy ("How far off from right we are")
y_ = tf.placeholder(tf.float32, [None, 10]) ④
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))

# Training
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy) ⑤
sess = tf.InteractiveSession()
tf.global_variables_initializer().run()

for _ in xrange(1000): ⑥
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})

# Evaluation
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1)) ⑦
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
result = sess.run(accuracy,
                  feed_dict={x: mnist.test.images, y_: mnist.test.labels})

print('Simple model accuracy on test data:', result) ⑧
```

- ① We start by importing the TensorFlow library, which is installed by running `pip install tensorflow`.
- ② TensorFlow comes with some example data sets, which we import and load into memory here.
- ③ We define the structure of our inputs, weights, and biases, and then our model ( $y$ ) which is a bit like  $y = mx + b$  in algebra.
- ④ Next we need to measure how far the predicted output is from the "correct" output, which we call "cross entropy".
- ⑤ Now that everything is defined, we have tell TensorFlow to "train" the model by making adjustments that try to **minimize** the cross entropy we defined.
- ⑥ To actually execute the training, we run through 1,000 iterations where at each step we input a new image and adjust based on being told what the correct answer was (this data is in `mnist.train`)
- ⑦ We evaluate the model by inputting data from `mnist.test` and looking at how accurate the predictions are.
- ⑧ Finally, we print out the accuracy to see how we did.

If you're intimidated by this script, even with the annotations, don't worry: you're not alone. TensorFlow can be quite complicated and this example doesn't even use a deep neural network! If you were to run this script though, you'd see that it's pretty accurate (over 90%)!

### **Listing 18.2. Output of our machine learning system**

```
$ python mnist.py
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes. ①
Extracting MNIST_data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
('Simple model accuracy on test data:', 0.90679997) ②
```

- ① TensorFlow will automatically download all of the training data for you.
- ② Here we can see the output is just about 91%.

This script, as short as it is, has managed to recognize hand-written numbers with a 90% accuracy, which is pretty cool since we didn't actually explicitly teach it to recognize anything. Instead, we told it how to handle our input training data (which happened to be an image of a number and the actual number), then gave it the "correct answer" (since all of the data is labeled) and it figured out how to make the predictions based on that. So what happens if we increase the number of iterations from 1,000 to 10,000? If you make that change and run the script again, the output will look something like the following.

### **Listing 18.3. Output of higher accuracy with more training iterations**

```
python mnist.py
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
('Simple model accuracy on test data:', 0.92510003) ①
```

- ① By increasing the amount of training we see accuracy rise to above 92%.

There are three things that are important to notice:

1. Since we already downloaded the MNIST data set, we don't download it again.
2. The accuracy went up by a couple of points (to 92%) by running more training iterations.
3. It took longer to run this script!

If we change the number of iterations even further (say, to 100,000) we might get a slightly higher number (in my case it went up to 93%) but at the cost of the script

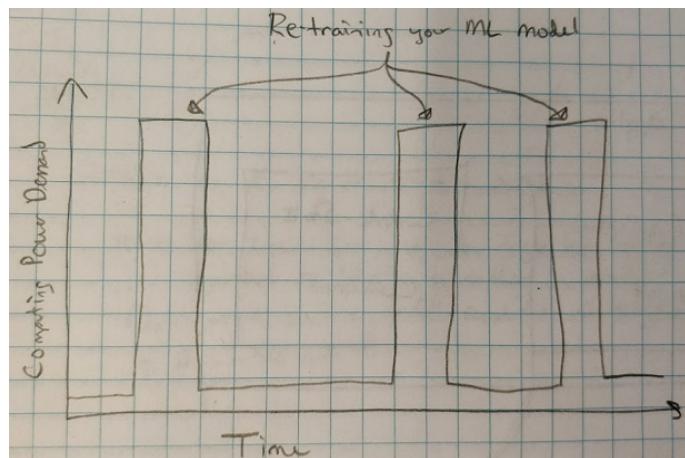
taking **much** longer to execute. This brings to a problem: how are we supposed to provide adequate training of our ML models that will be far more complex than this example, if it takes so long to run the computation? This is exactly the problem that Cloud Machine Learning Engine was built to solve. Let's look in more detail at what it is and how it works.

## 18.2 What is Cloud ML Engine?

As we've now seen, training machine learning models can start out being pretty quick but since it's such a computationally intensive process, doing more iterations or using a more complex machine learning model could end up taking quite a bit of time to compute. Further, while our example was based on data that doesn't really change (that is, handwritten numbers typically don't change that often), it's not unusual that a machine learning model you build would be based on your own data, and that data will probably be customized to individual users and change over time as users do new things. This means that as the data evolves, your machine learning model should evolve as well, which would require that you re-train your model to get the most up-to-date predictions.

If you were to run do this yourself with your computer, the demand for resources would probably end up looking something like the following graph, where every so often you need a lot of power to re-train the model and then the rest of the time you don't need all that much. If you have a feeling that cloud infrastructure is a good fit for this type of workload (remember that cloud resources are great for handling your spikes in demand), you're right!

**Figure 18.6. Spikes of demand for resources to retrain a machine learning model**



Cloud Machine Learning Engine (which we'll abbreviate to ML Engine) helps with this problem by acting as a hosted service for your machine learning models that can provide infrastructure to handle storage, training, and prediction. This means that in addition to offering computing power for training models, ML Engine can also store

and host trained models so that you can send your inputs to ML Engine and request that a particular model be used to calculate the predicted outputs.

Put in terms of our hand-written numbers example from before, this means that you can send Cloud ML Engine something like our TensorFlow script which you can use to train the model and once that model is trained you can send inputs into the model and get a prediction for what number was written. In a sense, ML Engine allows you to turn your custom models into something much more similar to the other hosted machine-learning APIs like the Vision API that we learned about in [chapter 14](#). Before we get into the details of how to use Cloud ML Engine, let's switch gears briefly to understand the core pieces of the system.

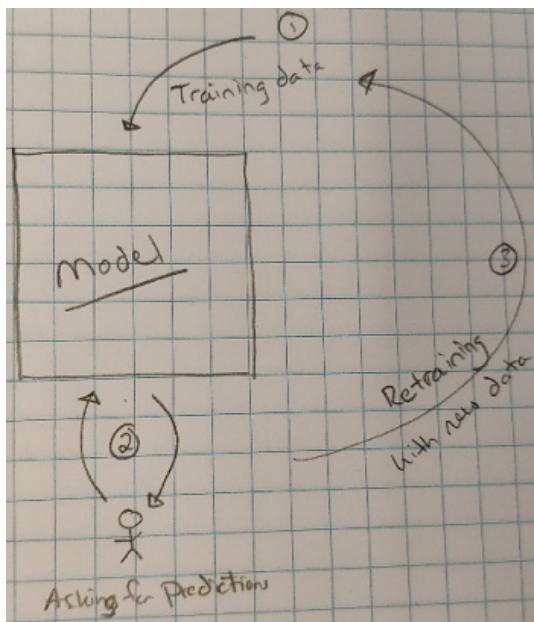
### **18.2.1 Concepts**

Like many of the hosted services in Google Cloud Platform, Cloud ML Engine has some core concepts that allow you to organize your project's machine learning pieces so that they're easy to use and manage. In some ways, Cloud ML Engine is a bit like App Engine in that you can run arbitrary machine learning code but you can also organize those into separate pieces, with different versions as things evolve over time. Let's dig into these different ways of organizing our work, starting with a word we've used quite a bit but never really defined: models.

#### **MODELS**

A machine learning model is sort of like a black box container that conforms to a specific interface that offers two primary functions: train and predict. How these functions are implemented are what distinguishes one model from another, but the key point here is that a model should be able to conceptually accomplish these two things.

For example, if you look back at our example script that recognizes hand-written numbers, the script itself actually does both of these. It starts by training the model based on a chunk of labeled images, and then attempts to get predictions from some images it hasn't seen before. Since the test data is also labeled we were able to test how accurate the model was, but this won't always be the case. After all, the idea behind using machine learning is to find the answers that we don't already know. As a result, the life-cycle of a model will usually follow this same pattern of (1) ingesting training data, then (2) handling requests to make predictions, and, potentially, (3) starting over with even more new training data.

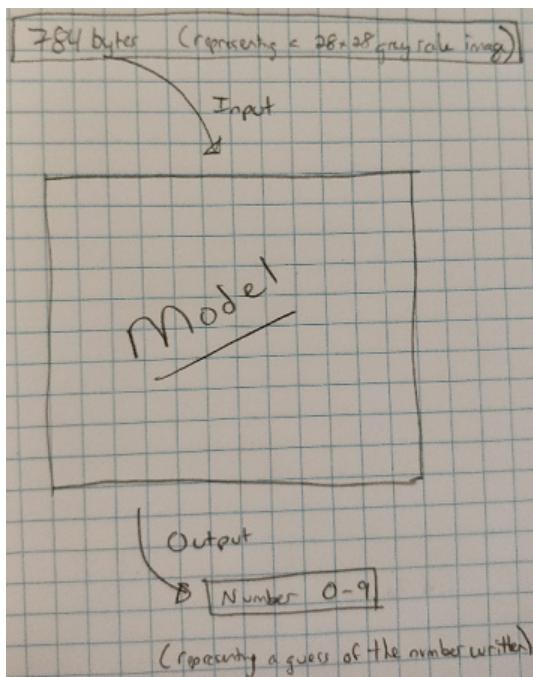
**Figure 18.7. Life-cycle of a model**

In addition to conforming to this interface where these two functions (train and predict) must exist, it's also important to note that the format of the data they understand will differ from one model to another. In other words, models are designed to ingest data of a specific format meaning that if we were to send data of other formats to the model (either for training or predicting purposes) the results would be undefined. For example, in the script from before that recognizes hand-written numbers the model is designed to understand input data in the form of a gray scale bitmap image of a hand-written number. If we were to send it data in any other format (such as a color image, a JPEG-formatted image, or anything else), any results that come out would simply be meaningless.

Additionally, the situations would differ depending on whether you're in the training or predicting stage. If invalid data (such as an unknown image format) was the input during a prediction request we'd likely see a bad guess for the number drawn or an error. On the other hand if we were to use this invalid data during the training process, we'd likely reduce the overall accuracy as the model would be training itself on data that doesn't make much sense.

Coming back to the example of recognizing hand-written numbers, the model in our TensorFlow script was designed to handle a 28 by 28-pixel gray scale bitmap image (784 bytes of data) as input, and return a value of 0 through 9 (its guess of what number was written) as output. In other words, the contract for the "model" we built before could be thought of as the black-box shown below.

**Figure 18.8. Machine learning model that recognizes hand-written images**

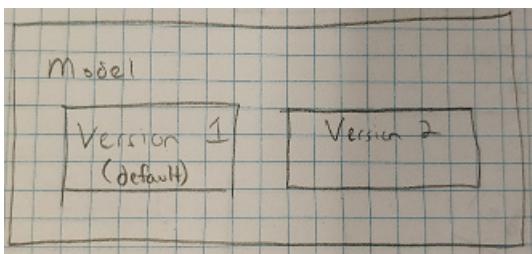


The important thing to note about our definition of a model is that what's inside the box is not as important as the contract fulfilled by the box (both the functions and the format of the data). In other words, if the inputs or outputs change the model itself is different, whereas if the model's internal functionality fulfills the contract but uses different technology under the hood, the model may have different accuracy levels but it still conceptually does the same job. So how would we distinguish between two models that fulfill the same contract but do so in different ways (maybe different training data, maybe different design)?

#### VERSIONS

Just like a Node.js package, App Engine service, or shared Microsoft Word document, Cloud ML models can have different versions as the inner-workings of a model evolves over time. Cloud ML exposes this concept explicitly so that you can compare different versions against one another for things like cost or accuracy. Under the hood, the thing we interact with is actually a version, however because a model has a "default version", we can interact with the model itself which implicitly means we're interacting with the default version.

**Figure 18.9. Models have many versions, and one default version**



Having the ability to create many versions of a model allows you to try lots of different things when building your model and test which of the configurations results in the best predictions for your use case. In the example above we might tweak lots of different parameters and see which of the versions is best at predicting the number written in an image. Then we might rely on the version that had the highest accuracy and delete the others. It's important to remember that a model is defined by the contract it fulfills, which means that all versions of a given model should accept the same inputs and produce the same outputs. This means that if we were to change the "contract" of the model (that is, change the input or output formats) we'd actually be creating an entirely different model rather than a new version of a model.

Also keep in mind that a specific version of a model is defined both by the code written as well as the data used to train the model. This means that you could take the exact same model code (similar to our TensorFlow script above), train it using two different sets of data, and end up with two different versions of a model that might produce different predictions based on the same input data.

Finally, we've talked about training a model using some data and then making predictions, but we've not really talked about where all of this data lives (both the training data and the data that defines the model version itself). So where is it? Under the hood Cloud ML Engine actually uses Google Cloud Storage to keep track of all of the data files that represent the model, as well as using Cloud Storage as a staging ground where you can put data to be used to train the model. You can read more about Cloud Storage in [chapter 8](#), and we'll come back to this a bit later on, but for now it's sufficient to understand that a model version represents a specific instance of a model that we interact with by training it and using it to make predictions. So how do we actually interact with these models? This is where jobs come into the picture.

## JOBS

As we learned before, the two key distinguishing features of a model are the ability to be "trained" and then the ability to make predictions based on that training. We also learned that sometimes the amount of data involved in things like training can be exceptionally large, which presents a bit of a problem since we wouldn't want to use an API call that has to upload 5 TB of training data. To deal with this, we rely on a "job" which is a way of requesting work be done asynchronously. Once we start one of these jobs we can come back and check on the progress later and then decide what to do

when the job completes.

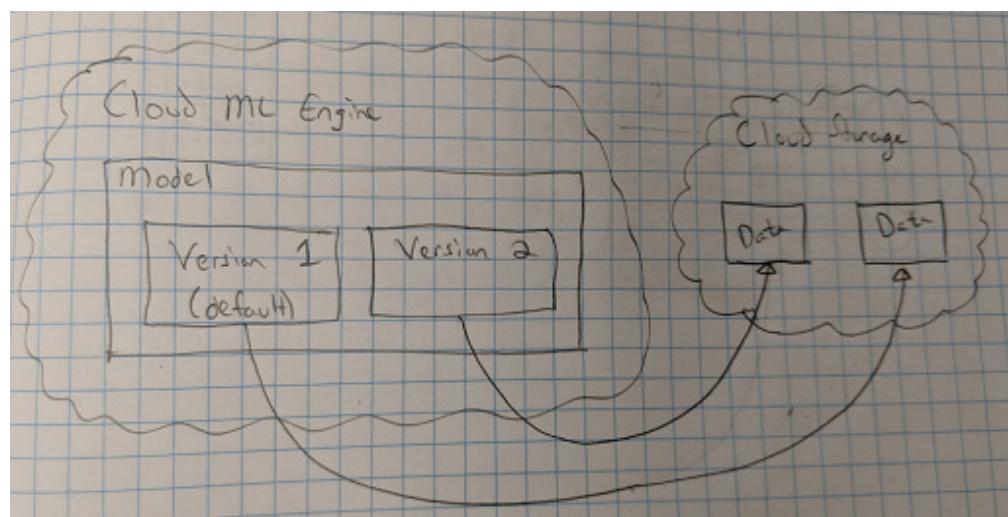
A job itself is made up primarily of some form of input (either training input or prediction input) that results in an output of the results, and it will run for as long as necessary to complete to work. In addition, the work that the job actually does can be run on a variety of different configurations which we specify when submitting the job to ML Engine. For example, if your ML model code can take advantage of GPUs you can choose a configuration with GPU hardware attached. Further, you can control the level of parallelization of the work when submitting the job by specifying a custom number of worker servers.

In short, a job is the tool we'll use to interact with our models whether it's training them to make predictions, actually making those predictions, or retraining them over time as we have new data. To get a better grasp of what jobs look like let's look at the high level architecture of how all of these pieces (jobs, models, and versions) all fit together.

### 18.2.2 Putting it all together

Now that we've looked at all of the concepts that ML Engine uses, we need to understand how they all get stitched together to do something actually useful. We've already learned that ML Engine stores data in Cloud Storage, but what does that actually look like? Whenever you have a model and versions of that model (remember, every model has a default version), the underlying data for that model lives in Google Cloud Storage. This means under the hood, models are sort of like pointers over to other data that lives in Cloud Storage, shown below.

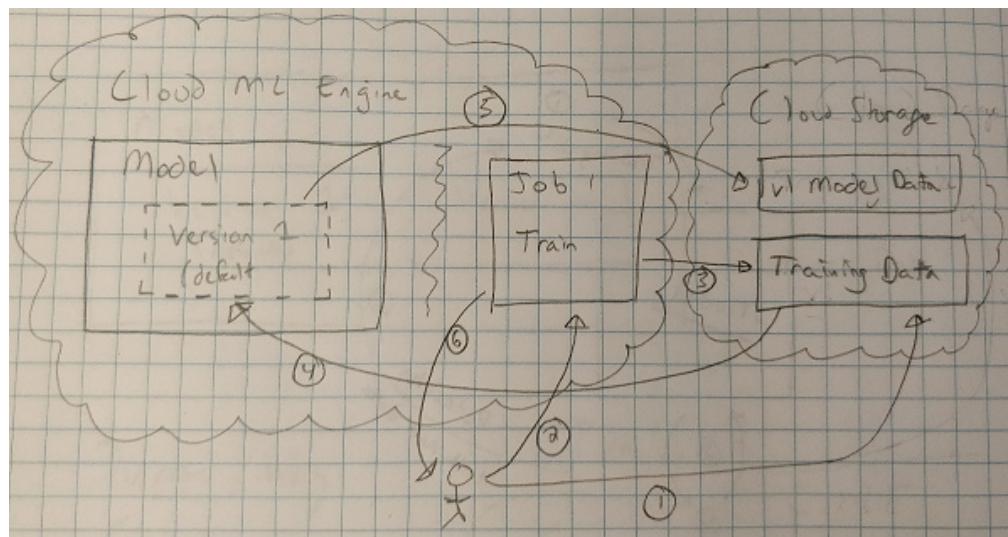
**Figure 18.10. Model data is stored in Cloud Storage**



But how did the model data get there in the first place? As we learned before we interact with ML Engine using jobs, so to get model data stored in Cloud Storage, we'd

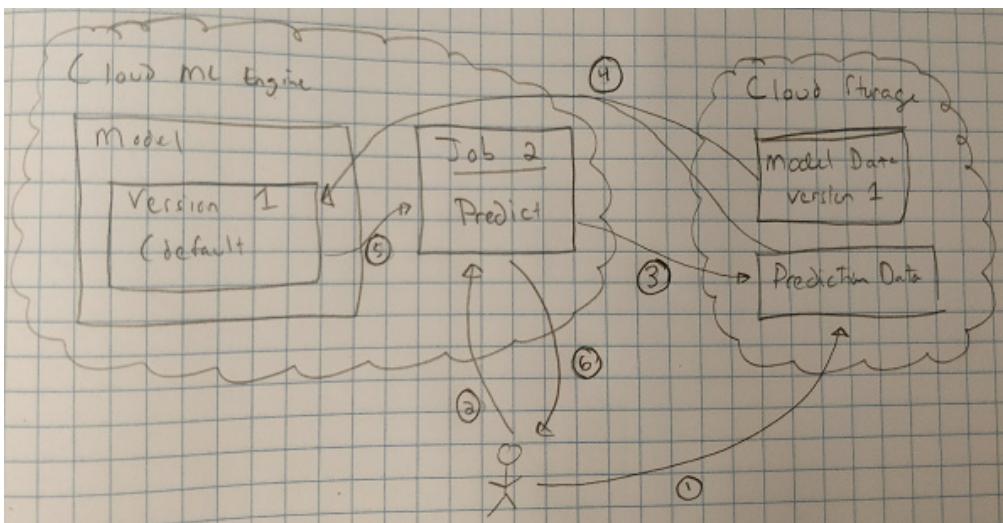
use a "training" job. When we create the job we'd tell ML Engine to look for the training data somewhere in Cloud Storage and then ask it to put the output job somewhere in Cloud Storage when it completes. This process of starting a training job would look like the following.

**Figure 18.11. Flow of training a model**



First (1), we'd upload the training data to Cloud Storage so that it's always available (that is, you don't have to worry about your computer crashing in the middle of your training job). Next (2), we create a job in ML Engine asking for that data to be used to train a version of our model (in this example, version 1). That job (3) would take the training data from Cloud Storage and use it to train the new model version by running it through the model using the TensorFlow script we'd write (4). Once the training is done (5), the mode would store its output back on Cloud Storage so that we can use it for predicting, and the job would complete (6) and let us know that everything worked. When this is all done, we'd end up with a trained model version in Cloud ML Engine with all the data needed being stored in Cloud Storage. Once a model has been trained and is ready to make predictions, we can run a prediction job in a similar manner.

**Figure 18.12. Flow of getting predictions based on a model**



Just like before, we'd start by uploading the data we want to make predictions on to Cloud Storage (1) so that it's always available. After that, we'd create a new prediction job on ML Engine (2) specifying where our data is and which model to use to make the prediction. That job would collect the prediction data (3) and then get to work running both it and the model version data on ML Engine (4). When a prediction is ready, it's sent to the job (5) and ultimately returned back to us (6) with all the details of what happened.

As you can see, the process to generate predictions using custom models is quite a bit more work than what we've been used to with the other ML APIs like Cloud Vision or Cloud Natural Language. In addition to designing and training your own model, the prediction process is a bit more hands-on as well, requiring that Cloud ML Engine and Cloud Storage work together to generate and return a prediction. This means that if you happen to have a problem that can be easily solved using the pre-built machine learning APIs, it's probably a far better idea to use those. However, if you have a machine learning problem that requires custom work, ML Engine aims to minimize the actual management work you'll need to do to train and interact with models. Now that we've seen the flow of things when training a model and using it for predictions, let's take a look at what this actually looks like under the hood.

### 18.3 Interacting with Cloud ML Engine

To demonstrate the different work flows we learned above (training a model and then making predictions using a model), it's probably best to run through an example with real data and real predictions. Unfortunately, however, designing an ML model and gathering all of the data involved is a pretty complicated task. To get around this, we're going to have to be a bit vague about the details of what's in the ML model (and all the data) from a technical perspective and instead focus on what the model intends to do

and how we can interact with it.

This means that we're going to gloss over the internals of the model and the data involved, and simply highlight the points that are important so that it makes conceptual sense. If you're interested in building models of your own (and dealing with your own data), there are plenty of great books about machine learning out there, and some others about TensorFlow which are definitely worth reading together with this chapter. With that said, let's look at a common example using real-life data that we can use to train a model and then make predictions based on that model.

### 18.3.1 Overview of US Census data

If you're unfamiliar with the US Census, it's simply a country-wide survey that's done every 10 years which asks general questions about the population such as ages, number of family members, and other basic data. In fact, this survey is how the US measures the overall population of the country. It turns out that this data is also available to the public, and we can use some of it to make some interesting predictions. The Census dataset itself is obviously pretty huge, so we'll look at a subset which includes basic personal information including education and employment details.

**NOTE**

All US Census data we'll use is anonymous, so we're never looking at an individual person

So what does this data actually look like? A given row in our data set will contain things like an individual's age, employment situation (e.g., private employer, government employer, etc), level of education, marital status, race, income category (e.g. less than or more than \$50,000 annual income), and more. For example, some simplified rows are shown below.

**Table 18.1. Example rows from the US Census data**

Age	Employment	Education	Marital status	Race	Gender	Income
39	State-gov	Bachelors	Never-married	White	Male	≤50K
50	Self-emp	Bachelors	Married	White	Male	>50K
38	Private	HS-grad	Divorced	White	Male	≤50K
53	Private	11th	Married	Black	Male	≤50K

So what we can do is use all of the other data in the row as a way to train a model that can make predictions about income category based on the other information. In other words, we'll train a model that is able to predict whether a person makes more than \$50,000 in a year based on their age, employment status, marital status, etc. This means we could provide data that looks like the following table and use our ML model to "fill in the blank" so to speak.

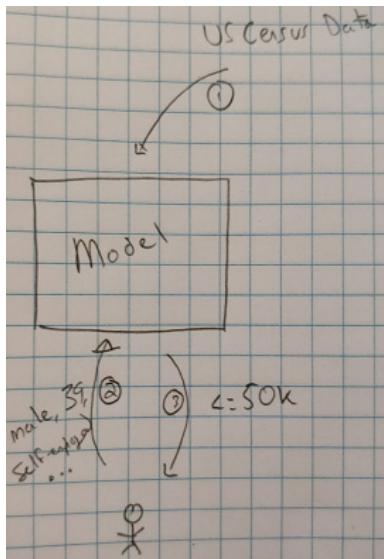
**Table 18.2. Example rows with Income missing**

Age	Employment	Education	Marital status	Race	Gender	Income
40	Private	Bachelors	Married	Black	Male	?
37	Self-emp	HS-grad	Divorced	White	Male	?

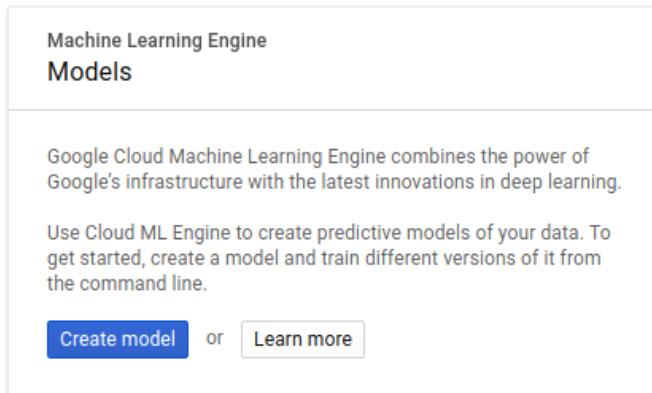
So how do we do get Cloud ML Engine to fill in these question marks with a guess of what should be there? Let's start by creating a model.

### 18.3.2 Creating a model

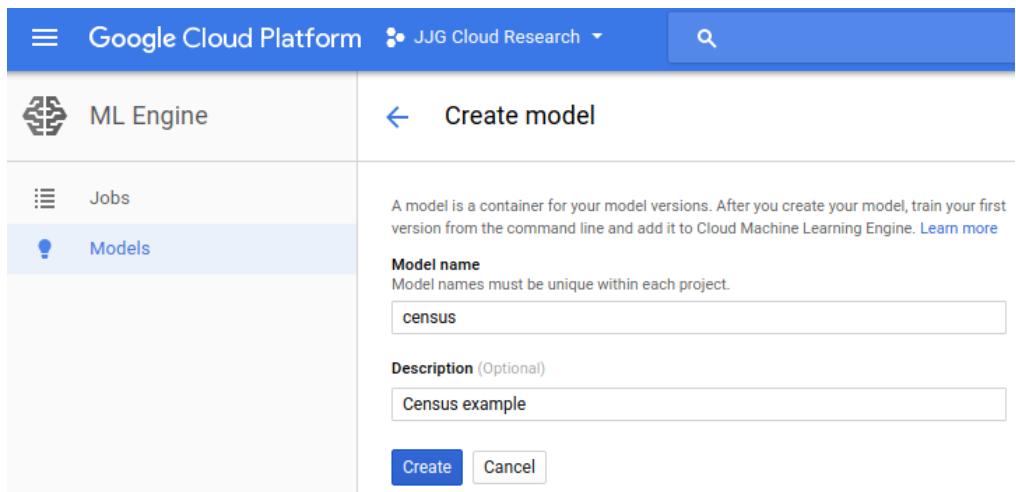
As we learned previously, a model acts as a container of a prediction function that fulfills a specific contract. In our case, when we want to make a prediction our model's contract accepts rows of US Census data (with the income category field missing) as input, and returns the predicted income category as output.

**Figure 18.13. Overview of the model flow**

Looking at this visually, the process (1) starts by using complete Census data to train our model to predict the income category field based on the rest of the row. After we finish training the model we can then send it rows with the income category missing (2) and it will send back predictions of the income category for that row (3). Since the model is just a container, we can create it using the Cloud Console. Choose ML Engine from the left-side navigation (it's towards the bottom under "Big Data") and you'll see a screen where you can create a new model.

**Figure 18.14. Prompt to create a new model**

After you click "Create model" you'll see a short form where you can name and describe the model. For our model we'll use the name `census` which is how we'll uniquely identify the model from now on.

**Figure 18.15. Creating our Census model**

Once you create the model, you can click into it and see that there are currently no versions. We can also see this by using the `gcloud` command-line tool to list out all models and versions for a given model, shown below.

**Listing 18.4. Listing models and versions on the command-line**

```
$ gcloud ml-engine models list
NAME      DEFAULT_VERSION_NAME
census
```

```
$ gcloud ml-engine versions list --model=census
Listed 0 items.
```

As you can see, the model exists but there are no versions (and no default version). This means that we effectively have a model that has no code defining it and hasn't been trained at all, so the next step is to train the model with some data. Before we can do that though, we first need to get Cloud Storage set up with all the right code and data that we'll use for training.

### 18.3.3 Setting up Cloud Storage

Now that our census model exists, we have to train it in order to make some predictions. To do that, we'll need a bunch of US Census data to use for training purposes, and we'll need to make sure that the data lives in the right place in Google Cloud Storage. We can download the some example data from the US Census data set using the gsutil tool as the example data itself is available in a public Cloud Storage bucket for exactly this purpose. If you're not familiar with Cloud Storage, take a look at [chapter 8](#) first.

#### **Listing 18.5. Downloading the US Census data set from Cloud Storage**

```
$ mkdir data
$ gsutil -m cp gs://cloudml-public/census/data/* data/ ①
Copying gs://cloudml-public/census/data/adult.data.csv...
Copying gs://cloudml-public/census/data/adult.test.csv...
/ [2/2 files][ 5.7 MiB  5.7 MiB] 100% Done
Operation completed over 2 objects/5.7 MiB. ②
```

- ① We'll put our data in a directory called data.
- ② This command copies all of the files from a public bucket into the data directory.

Notice that this data is pretty small to start (only about 6 MB). Even though this is pretty tiny, it should still be able to help us make some reasonably accurate predictions. Also keep in mind that there are two different data sets: a "data" and "test". The first (`adult.data.csv`) is the data we'll use to train our model, and the second (`adult.test.csv`) is the data we can use to evaluate our model.

Think of the first as data that we'll use for learning, sort of like example problems that you work through with a teacher in school. The second data set is more like the final exam at the end of the course where you figure out just how well you did. It wouldn't really make sense to give you the exact same problems that you'd already done in class, so these are some new ones that we haven't seen before. The next thing we have to do is create a new bucket in Cloud Storage that will hold our copy of this data. In addition, this bucket will also hold the data representing the model after it's trained, as well as any data we want to send via a prediction job later on, but for now we'll use it just for storing the US Census data.

**NOTE**

You may be wondering why we don't just rely on the public Cloud Storage bucket to host the training data. In this example, we want to be sure that the data in question doesn't change out from under us, and the safest way to do that is to keep our own copy available in a bucket that we own and control.

We'll also need to make sure the bucket is located in a single region rather than distributed across the world. This is to avoid cross-region data transfer costs which could be pretty large if you have a lot of data and are sending it from a multi-regional bucket to your ML Engine jobs. In other words, if you had a lot of data stored in a bucket in Asia then a training job in the US would involve sending all of that data across the world and back again with the final result. Even though our example is only dealing with a few megabytes, keeping the data nearby to the resources that will actually do the training means you won't waste any money needlessly sending data all over the place.

For this example, we'll use the `us-central1` region to act as the home for our bucket, as well as the home for resources we'll use for training later on. We can create this bucket using the `gsutil` command again, relying on the `-l` flag to indicate that we want our bucket to live in that specific location.

**Listing 18.6. Creating a new bucket in us-central1**

```
$ gsutil mb -l us-central1 gs://your-ml-bucket-name-here
Creating gs://your-ml-bucket-name-here/...
```

Once we have both the data we need and the bucket to hold our data, we can actually get to uploading the data using `gsutil` again.

**Listing 18.7. Uploading a copy of the data to our newly created bucket**

```
$ gsutil -m cp -R data gs://your-ml-bucket-name-here/data
Copying file://data/adult.data.csv [Content-Type=text/csv]...
Copying file://data/adult.test.csv [Content-Type=text/csv]...
- [2/2 files][ 5.7 MiB  5.7 MiB] 100% Done
Operation completed over 2 objects/5.7 MiB.

$ gsutil ls gs://your-ml-bucket-name-here/data
gs://your-ml-bucket-name-here/data/adult.data.csv
gs://your-ml-bucket-name-here/data/adult.test.csv
```

Finally, all the data is stored in our bucket, which is located in the `us-central1` region, and we can start looking at how to define and train our model.

### **18.3.4 Training our model**

Now that all the data is in the right place, it's time to start thinking about the code for our model and then the job we'll use to train our model using that code and the data we previously uploaded. Let's start by downloading some of the code.

**WARNING**

As we discussed early on in this chapter, the actual TensorFlow code involved here would take quite a while to explain and builds on concepts that are better left to a book on TensorFlow. As a result, you're not expected to understand the code and we won't reproduce it here. Instead, we'll treat the code itself as a black-box and focus on what it can do using Cloud ML Engine.

The example code that will train our model is located on GitHub in the `@GoogleCloudPlatform/cloudml-samples` repository. This means we can clone the repository using git, or, if you're not familiar with Git at all, you can download it as a zip file from [github.com/GoogleCloudPlatform/cloudml-samples](https://github.com/GoogleCloudPlatform/cloudml-samples). The example code we're interested in is located in the `census` directory.

**Listing 18.8. Cloning the Git repository containing the census model code**

```
$ git clone https://github.com/GoogleCloudPlatform/cloudml-samples
Cloning into 'cloudml-samples'...
remote: Counting objects: 1065, done.
remote: Compressing objects: 100% (70/70), done.
remote: Total 1065 (delta 45), reused 59 (delta 19), pack-reused 967
Receiving objects: 100% (1065/1065), 431.81 KiB | 11.07 MiB/s, done.
Resolving deltas: 100% (560/560), done.

$ cd cloudml-samples/census/tensorflowcore/
```

Once we have the same code, we'll need to submit a new training job. As we learned earlier jobs represent the way that we schedule some work to be done that might take a while due to lots of data or computationally intense machine learning code. That said, given the size and complexity of what we're trying to do, the training job itself shouldn't take all that long. On the other hand, the command we'll need to run to start the training job is actually pretty complicated so we'll walk through it piece by piece.

**Listing 18.9. Command to submit a new training job**

```
$ gcloud ml-engine jobs submit training census1 \
  --stream-logs \
  --runtime-version 1.2 \
  --job-dir gs://your-ml-bucket-name-here/census \
  --module-name trainer.task \
  --package-path trainer/ \
  --region us-central1 \
  ...
  --train-files gs://your-ml-bucket-name-here/data/adult.data.csv \
  --eval-files gs://your-ml-bucket-name-here/data/adult.test.csv \
  --train-steps 10000 \
  --eval-steps 500
```

- ➊ We start by submitting a new training job for our census model.
- ➋ This is instructing Cloud ML Engine to use TensorFlow version 1.2.
- ➌ When we run our job, we instruct Cloud ML Engine to put the various output data (that is, the trained model data) in a specific place in Cloud Storage.

- ④ These two lines are where we tell Cloud ML where the code for our TensorFlow model is located and how to execute the training. You can explore this code if you're interested by looking in this directory at the two Python files.
- ⑤ Since we created the bucket to hold our data in us-central1, we'll also instruct Cloud ML Engine to run the training workload on resources located in the same region.
- ⑥ This line might seem innocuous, but it's super important. It says that the following parameters should be passed along to our TensorFlow script rather than be consumed by the gcloud command.
- ⑦ Here we point to the data to use for training and evaluation.
- ⑧ Finally we specify how many times to iterate at improving our accuracy for predictions. Since we have a lot of compute power available we can use a large number here.

After running this, you should see quite a bit of output explaining the progress of training, but the whole process shouldn't take all that long (a couple of minutes generally).

**NOTE**

If you get an error about ML Engine not being able to read from the GCS path, the error should also include a service account name that is trying to access the data (e.g., service-12345678989@cloud-ml.google.com.iam.gserviceaccount.com).

You can use grant read-only access to this service account in the Cloud Console by editing the bucket permissions and making the service account listed an "object viewer" and an "object creator".

To see the output, we can use gsutil again because we instructed our job to put all of the output data into our Cloud Storage bucket.

#### **Listing 18.10. Listing the output of the training job**

```
$ gsutil ls gs://your-ml-bucket-name-here/census
gs://your-ml-bucket-name-here/census/
gs://your-ml-bucket-name-here/census/checkpoint
gs://your-ml-bucket-name-here/census/events.out.tfevents.1509708579.master-
88f54a3b38-0-tlmnd
gs://your-ml-bucket-name-here/census/graph.pbtxt
gs://your-ml-bucket-name-here/census/modelckpt-4300.data-00000-of-00003
...
gs://your-ml-bucket-name-here/census/eval/
gs://your-ml-bucket-name-here/census/export/
gs://your-ml-bucket-name-here/census/packages/

$ gsutil ls gs://your-ml-bucket-name-here/census/export
gs://your-ml-bucket-name-here/census/export/
gs://your-ml-bucket-name-here/census/export/saved_model.pb ❶
gs://your-ml-bucket-name-here/census/export/variables/
```

- ❶ This file (saved\_model.pb) is the important one as it contains the actual model that we can import and use for predictions.

To finish everything off, we need to create a new model version based on the output of our training job. Since the output is located in census/export/saved\_model.pb, we can

do this using the Cloud Console by creating a new version and point it to that specific file. To do this, navigate to the Cloud ML Engine section in the Cloud Console and click on your model. Inside that page you'll see some text saying that the model currently has no versions yet, along with a link to create one.

**Figure 18.16. The census model without any versions yet**

The screenshot shows the Google Cloud Platform interface for the ML Engine. In the top navigation bar, 'Google Cloud Platform' and 'JJG Cloud Research' are visible. On the left, there's a sidebar with 'ML Engine', 'Jobs', and 'Models'. The 'Models' tab is selected. The main content area is titled 'Model details' for the 'census' model. It shows a 'Census example' and a 'Versions' section with the message: 'This model has no versions yet. Create at least one version to start using your model. [Create a version](#)'. There are 'CREATE VERSION' and 'DELETE' buttons at the top right.

Clicking on the link will show a form where we can name the version and choose where the data for the model version actually lives. Since this is our first version of the census model, we'll use v1 as the name for the version and then as we saw before when listing the output from the training job, the model itself is located in the /census/export/ directory of our storage bucket.

**Figure 18.17. Creating a new version from our training output data**

The screenshot shows the 'Create version' dialog box. The title is 'Create version'. The 'Name' field is filled with 'v1'. The 'Description (Optional)' field is empty. The 'Source' field contains the path 'your-ml-bucket-name-here/census/export/'. At the bottom are 'Create' and 'Cancel' buttons.

Once we set that, we can click "Create" and the model version will be loaded up, and automatically set as the default version for our census model, which you can see by looking at the model details page, shown below.

**Figure 18.18. The census model with v1 as the default version**

The screenshot shows the Google Cloud Platform interface for the ML Engine. In the top navigation bar, 'Google Cloud Platform' and 'JJG Cloud Research' are visible. On the left sidebar, 'ML Engine' is selected, followed by 'Jobs' and 'Models'. The main content area is titled 'Model details' for 'census'. It shows a 'Census example' and a table of 'Versions'. The first version, 'v1 (default)', is listed with a creation time of 'Nov 3, 2017, 8:29:12 AM'. There are buttons for '+ CREATE VERSION' and 'DELETE'.

Name	Creation time	Last use time ^
v1 (default)	Nov 3, 2017, 8:29:12 AM	<a href="#">Set as default</a> <a href="#">Delete</a>

So now that we finally have a trained model, let's look at how we can use it to make predictions, and see how well it actually does at making them.

### 18.3.5 Making predictions

As we learned before, once a model is trained we can use it to make some predictions. In this case, we trained a model on US Census data targeting the "income category" field so that later on we can send it rows of data with that field missing and it will make a guess about what value should go in that field. In other words, we send it details about a person and then ask it to predict whether that person is likely to earn more than or less than \$50k per year.

The way we actually do this depends on the number of predictions that we want to make at once. For example, if we want to make a prediction on a single row we can do this directly by sending the row directly to the model, however if we have lots and lots of rows that we want predictions for it's better to do this using a prediction job and putting the input and output data on Cloud Storage sort of how we did with training. Let's start by looking at a single row and then move onto multiple rows using prediction jobs. To start, we'll need an incomplete row of data, but missing the income category. In the GitHub repository that has some example data which we can use as a demonstration. Inside `census/test.json` you'll see a row of data representing a 25-year old person. Below, you can see a summary of a few of the fields.

**Table 18.3. A summary of the row in test.json**

Age	Employment	Education	Marital status	Race	Gender
25	Private	11th grade	Never-married	Black	Male

If we were to run this data through our predictor, we'll get back some predictions as well as a confidence level, shown below.

**Listing 18.11. Making a prediction directly (without a prediction job)**

```
$ gcloud ml-engine predict \
--model census --version v1 \
--json-instances test.json
CONFIDENCE PREDICTIONS
0.78945 <=50K
```

- ➊ Here we request using the census model that we created.
- ➋ In this case we specify a path to the JSON data in a local file.
- ➌ Our model returns an output made up of a prediction of an income category along with a confidence level.

As we can see, the test data provided predicts that the person in question likely earns less than \$50,000 dollars per year, but the confidence of that prediction is not quite perfect. If you're interested, in playing with this, you can always try tweaking some of the fields of the JSON file and looking at what happens. For example, if we were to change the age of this same person to 20 years old (instead of 25), the confidence level goes up that the person is earning less than \$50k annually, shown below.

**Listing 18.12. Making another prediction based on modified data**

```
$ gcloud ml-engine predict --model census --version v1 --json-instances
./test2.json
CONFIDENCE PREDICTIONS
0.825162 <=50K
```

So what if we had a lot of instances that we wanted predictions for? As we discussed, this is what jobs are primarily made for: dealing with large amounts of work to be done in the background.

This process works similarly to how we saw when training our model. We'll first upload the data we want to use get predictions for to Cloud Storage, then submit a prediction job asking ML Engine to pull that data and place the output predictions into another location on Cloud Storage. To do this, we can use the same file (`test.json`) again, but let's modify it to add a few more similar rows. In this example, let's just reproduce the same rows and increase the age by 5 years for each row. This would mean if we go from 25 up to 65, we'll have 10 different rows that we want to make predictions for. First, we'll upload the file to Cloud Storage, shown below.

**Listing 18.13. Copying the modified data to Cloud Storage**

```
$ gsutil cp data.json gs://your-ml-bucket-name-here/data.json
Copying file://data.json [Content-Type=application/json]...
/ [1 files][ 3.1 KiB/ 3.1 KiB]
Operation completed over 1 objects/3.1 KiB.
```

Now we can submit a prediction job pointing to the uploaded data and ask the output to be placed in a different location.

**Listing 18.14. Submitting a new prediction job for the modified data on Cloud Storage**

```
$ gcloud ml-engine jobs submit prediction prediction1 \
--model census --version v1 \
--data-format TEXT \
--region us-central1 \
--input-paths gs://your-ml-bucket-name-here/data.json \
--output-path gs://your-ml-bucket-name-here/prediction1-output
```

Once the job completes we can look at the output which will live on Cloud Storage in the prediction1-output directory of our bucket.

**Listing 18.15. Viewing the output of the prediction job**

```
$ gsutil ls gs://your-ml-bucket-name-here/prediction1-output ①
gs://your-ml-bucket-name-here/prediction1-output/prediction.errors_stats-00000-of-
00001
gs://your-ml-bucket-name-here/prediction1-output/prediction.results-00000-of-00001

$ gsutil cat gs://your-ml-bucket-name-here/prediction1-output/prediction.results-
00000-of-00001 ②
{"confidence": 0.8251623511314392, "predictions": " <=50K"}
{"confidence": 0.7894495725631714, "predictions": " <=50K"}
{"confidence": 0.749710738658905, "predictions": " <=50K"}
{"confidence": 0.7241880893707275, "predictions": " <=50K"}
{"confidence": 0.7074624300003052, "predictions": " <=50K"}
{"confidence": 0.7138040065765381, "predictions": " <=50K"}
{"confidence": 0.7246076464653015, "predictions": " <=50K"}
{"confidence": 0.7297274470329285, "predictions": " <=50K"}
 {"confidence": 0.7511150240898132, "predictions": " <=50K"}
 {"confidence": 0.784980833530426, "predictions": " <=50K"}
```

- ① We can see the files that resulted by listing the contents of the directory.
- ② We can print the output of the result using the cat sub-command of gsutil.

As we can see in with the predictions, increasing the age while holding everything else the same doesn't change the prediction itself but it does tend to decrease the confidence. That is, our model is more confident about its predictions for a younger person than for an older person. Now that we've seen how to make predictions (both directly and using a prediction job), we really should take a step back and look at what actually is happening under the hood when interacting with our models.

### **18.3.6 Configuring our underlying resources**

In the jobs that we've run so far, we sort of glossed over the whole idea that there were some computers somewhere actually doing the computational work. For example, when we submitted our training job we never really discussed anything about the VMs that pulled down the data and the CPU cycles consumed when we ran that data through the ML model itself. To understand this we need to look at the concept of scale tiers, machine types, and ML training units, all of which are related to the actual computing (and memory) resources in use during training. Let's start by looking at the basics of scale tiers.

## SCALE TIER

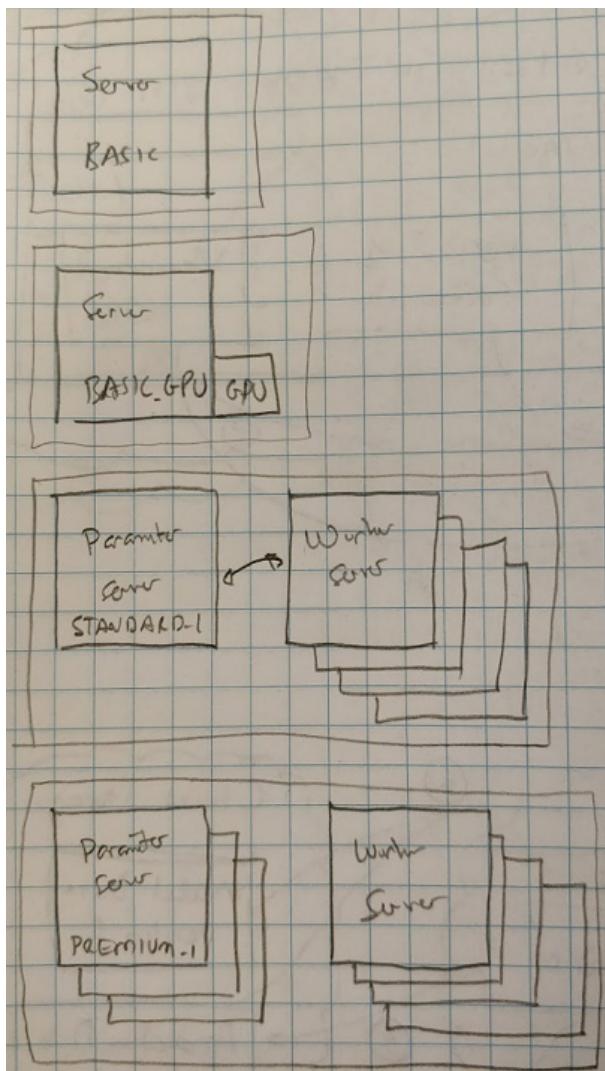
When creating a training job on ML Engine, you have the option to specify something called a "scale tier" which is basically a predefined configuration of compute resources that are likely to do a pretty good job at handling your training workload. In other words, the default scale tiers are a "pretty good guess" for the typical work done in a machine learning job.

These different configurations have a few different pieces that result in very different performance profiles. First, there is the concept of a "worker server" which is like a VM that actually does the computation needed to train a model. Next, if there are multiple workers, we need to make sure that the model being computed stays synchronized between the various worker servers. This is the job of a "parameter server" which we won't say much more about except for noting the fact that these servers are responsible for coordinating the efforts of the various worker servers. Finally, these servers can have many different hardware configurations by virtue of simple things like different CPUs or amounts of memory, or by attaching different pieces of computational hardware like GPUs that can speed up various mathematical operations.

We'll get into the details of these in a moment, but first we need to look at the various preset scale tiers available, which are:

1. **BASIC**, which is a single worker server that trains a model,
2. **BASIC\_GPU**, which is a single worker server that comes with a GPU attached,
3. **STANDARD\_1**, which uses lots of worker servers but has a single parameter server,
4. **PREMIUM\_1**, which uses lots of workers and lots of parameter servers to coordinate the shared model state.

To get a better idea of this, the following image shows how these different preset scale tiers look.

**Figure 18.19. Various scale tiers**

Setting a specific scale tier is easy: just use the `--scale-tier` flag when submitting your training job. By default (that is, if you don't set a scale tier or any other configuration), ML Engine will use the `BASIC` scale tier. For instance, in the example from before we didn't specify a tier and therefore ran using this basic tier. This tier is generally good for kicking the tires and testing out ML Engine, but not very good if you have a lot of data or a particularly complex model. If we wanted to configure this explicitly, the command would look something like the following.

**Listing 18.16. Running a training job using the BASIC scale tier**

```
$ gcloud ml-engine jobs submit training censusbasic1 \
--stream-logs \
--runtime-version 1.2 \
--job-dir gs://your-ml-bucket-name-here/censusbasic1 \
--module-name trainer.task \
--package-path trainer/ \
--region us-central1 \
--scale-tier BASIC ①
-- \
--train-files gs://your-ml-bucket-name-here/data/adult.data.csv \
--eval-files gs://your-ml-bucket-name-here/data/adult.test.csv \
--train-steps 10000 \
--eval-steps 1000
```

- ① We can specify a tier explicitly when submitting a training job.

Similarly to the **BASIC** tier, the **BASIC\_GPU** tier is also good for testing things out when you can take advantage of hardware acceleration as the single server will have an NVIDIA Tesla K80 GPU attached.

The next two tiers (**STANDARD\_1** and **PREMIUM\_1**) are the only ones that are recommended for real production workloads as they are distributed models that can handle things like large amounts of data. These two both have lots of worker servers that will do the computational work to train your model, but there is one key difference. When there are multiple worker servers each of these may be busy performing lots of calculations but all of these servers still have to work together or risk losing out on the benefits of having lots of workers in the first place. To do this, workers rely on a parameter server to be the central authority for the cluster of worker servers, which means that there could result in a bottleneck where a parameter server is overwhelmed by all the workers. In the **STANDARD\_1** tier, there is only a single parameter server which could become a single point of failure for a training job with a great number of workers. On the other hand, in the **PREMIUM\_1** tier the system spins up lots of parameter servers which avoids this bottleneck.

You may be wondering why there isn't a **STANDARD\_GPU** or **PREMIUM\_GPU** tier, or how you control the specific number of servers, or whether you can control how much CPU or memory is available, which is completely reasonable. To do this, we have to dig into the concept of a "machine type" on ML Engine which is somewhat different from the "instance type" on Compute Engine.

#### **MACHINE TYPE**

If the preset scale tiers offered by ML Engine aren't a great fit (which, if you need access to GPUs, this will likely be the case), ML Engine provides the ability to customize the hardware configuration to the specifics of your jobs. The following table shows all of the different machine types that we can use, but before we look at that, there are a few things to note.

First, notice that there are really only two choices for configuring parameter

servers: `standard` and `large_model`. This is because a parameter server really can't benefit from more CPU or hardware acceleration, but may end up needing a lot of memory if the model itself is particularly large. That leads to the obvious difference between these two machine types: memory, with the `large_model` machine type having four times as much memory as the `standard` machine type.

Next, unlike Compute Engine's instance types, ML Engine's machine types don't specify the exact amount of CPU or memory available to the machine. Instead, we have some reference amount of capacity and larger machine types having rough multiples of that reference amount. This means that instead of saying the specific amount of memory available from one machine type to the next, we can think of the next step up being roughly twice as much of a resource. This means, for example, that the `complex_model_m` (medium) machine type is about twice as much CPU and memory as the `complex_model_s` (small) machine type.

**Table 18.4. Summary of the different machine types**

Machine type	Best for	CPU	Memory	GPUs
<code>standard</code>	All servers	1x	4x	None
<code>standard_gpu</code>	Worker servers	1x	4x	1x K80
<code>standard_p100</code>	Worker servers	1x	4x	1x P100
<code>large_model</code>	Parameter servers	2x	16x	None
<code>complex_model_s</code>	Worker servers	2x	2x	None
<code>complex_model_m</code>	Worker servers	4x	4x	None
<code>complex_model_m_gpu</code>	Worker servers	4x	4x	4x K80
<code>complex_model_m_p100</code>	Worker servers	4x	4x	4x P100
<code>complex_model_l</code>	Worker servers	8x	8x	None
<code>complex_model_l_gpu</code>	Worker servers	8x	8x	8x K80

So if we want to use these machine types in our training jobs how do we do this? Instead of passing all of this information about our underlying resources in the form of command-line arguments, we can put it into a configuration file and pass that along instead. The configuration can be in either JSON or YAML format, and should look something like the following.

#### Listing 18.17. Job configuration file

```
trainingInput:
  scaleTier: CUSTOM      ①
  masterType: standard   ②
  workerType: standard_gpu
  parameterServerType: large_model
  workerCount: 10         ③
  parameterServerCount: 2
```

- ① Here we clarify that we want a custom scale tier rather than one of the presets.

- ② We can set the types of the various servers (master, workers, and parameter servers) to anything we want. Here we've used standard for the master, standard\_gpu for the worker, and large\_model for the parameter server.
- ③ In addition to controlling the type of the machine, we can also control how many of each we deploy. The master always is a single server, but we can add more workers and more parameter servers as well. In this example, we use 10 workers and 2 parameter servers.

If we save this information to a file (say, `job.yaml`), we can then submit a new training job where we leave everything else as before except we don't specify a scale tier and instead refer to the configuration file.

#### **Listing 18.18. Submitting a new training job using a configuration file**

```
$ gcloud ml-engine jobs submit training censuscustom1 \
  --stream-logs \
  --runtime-version 1.2 \
  --job-dir gs://your-ml-bucket-name-here/customcensus1 \
  --module-name trainer.task \
  --package-path trainer/ \
  --region us-central1 \
  --config job.yaml \ ①
  \
  --train-files gs://your-ml-bucket-name-here/data/adult.data.csv \
  --eval-files gs://your-ml-bucket-name-here/data/adult.test.csv \
  --train-steps 10000 \
  --eval-steps 1000
```

- ① Instead of setting a scale tier, we point the command-line tool at the configuration file that we created earlier.

So now that we've seen how to change the underlying resources for our training jobs, let's look at how this works when making predictions.

#### **PREDICTION NODES**

In the case of prediction, the work is much more uniform and as a result there is really only one type of server in use: workers. And unlike training jobs, a prediction job doesn't offer a way to modify the type of machine involved which means we only ever think terms of "how many" rather than "of what type". That said, the count of servers (or "prediction nodes" as they're known) is a limit rather than the fixed count that we saw with training jobs. This is because there is an element of automatic scaling based on the amount of work submitted in the job. For example, a small job of a few predictions as we tried before might not benefit from more than one node running at a time, however a large job of millions of predictions will likely complete more quickly if there are lots of workers.

As a result, ML Engine will continue to turn on new workers until either it reaches the limit set or workers run out of work to do. This allows ML Engine to optimize for the fastest completion time of any prediction job within some reasonable limitations. We can easily control this limit by setting the `--max-worker-count` flag. For example,

the following shows how we could modify our previous prediction job to use no more than 2 workers.

#### **Listing 18.19. Specifying a limit on the number of workers in a prediction job**

```
$ gcloud ml-engine jobs submit prediction prediction2workers \
--model census --version v1 \
--data-format TEXT \
--region us-central1 \
--max-worker-count 2 \ ❶
--input-paths gs://your-ml-bucket-name-here/data.json \
--output-path gs://your-ml-bucket-name-here/prediction2workers-output
```

- ❶ Here we can set the maximum number of workers to use to 2.

This leaves the open question of how many nodes are used during an on-line prediction request made directly rather than as part of a batch job. So how does this work? In this case, automatic scaling comes into play once again, where ML Engine will keep a certain number of workers up and running in order to minimize latency on incoming prediction requests. This means that as more prediction requests arrive, ML Engine will turn on more workers to ensure that the prediction operations complete quickly.

The number of workers running to handle on-line prediction requests is scaled entirely automatically which means that there's nothing for us to do besides send requests as we need them. With that said, it's important to remember that on-line prediction should not be used as a replacement for batch prediction. In other words, on-line prediction is great for kicking the tires and sending a steady stream of prediction requests that may fluctuate a bit but won't ever spike to extreme levels with little warning. Now that we've gone through all of the details about underlying resources we have to ask the inevitable question: how much does all of this cost?

## **18.4 Understanding pricing**

Since ML Engine has two distinct operations that it supports (predicting and training), there are actually two different pricing schemes for each of these. Since training is the more complicated of the two, let's start by looking at how much it costs and then we'll move onto the cost of making predictions from ML Engine models.

### **18.4.1 Training costs**

Similar to Compute Engine, ML Engine pricing is based on an hourly compute unit cost, however there are quite a few important differences. First, the table of machine types never really specified exactly how much compute power was available for each of the different types and instead focused on how larger types are "roughly double the size" others. Second, we never really clarified what types of machines were in use when using one of the preset scale tiers. So how does all of this work?

At the end of the day, all of the pricing for ML Engine boils down to "ML training units" consumed which have a price per hour of use. Luckily this price can be chopped

up into 1-minute increments to pay for only what you consume, but just like Compute Engine, there's a 10-minute minimum to deal with overhead. This means that if you were to consume 5 minutes worth of work, you'd pay the 10-minute minimum, but if you were to use 15 minutes, you'd only pay for exactly 15 minutes. So how do we figure out the hourly rate? Let's start by looking at the rate (in ML training units) for the various scale tiers.

#### **SCALE TIER-BASED PRICING**

As we just learned, computing time is measured in ML training units which themselves have an hourly cost. This means that each of the different scale tiers costs a certain number of ML training units per hour. Additionally, these costs vary depending on the geographical location, where US-based locations cost a bit less than their equivalents in Europe or Asia. The table below shows a summary of the different scale tiers, the number of ML training units for each, and the overall hourly cost for the different locations.

**Table 18.5. Costs for various scale tiers**

Scale tier	ML training units	US cost	Europe / Asia cost
BASIC	1	\$0.49 per hour	\$0.54 per hour
BASIC_GPU	3	\$1.47 per hour	\$1.62 per hour
STANDARD_1	10	\$4.90 per hour	\$5.40 per hour
PREMIUM_1	75	\$36.75 per hour	\$40.50 per hour

As you can see, the "basic" tiers (`BASIC` and `BASIC_GPU`) are very light on resources, which is why they are much cheaper than others like `PREMIUM` which is an order of magnitude more power (and cost).

In our example training job from before, where we ended up using the default `BASIC` scale tier, we ended up paying \$0.49 per hour since our job was run in the `us-central1` region. This means that, assuming we fell under the 10 minute minimum charge, that simple job cost about 8 cents ( $10 \text{ minutes} / (60 \text{ minutes per hour}) * \$0.49 \text{ per hour} = \$0.08167$ ). So what about the custom deployments that we just learned about? Let's look in more detail at the pricing for customized resources.

#### **MACHINE TYPE-BASED PRICING**

Just like scale tiers, each machine type comes with a cost defined in ML training units, which then follows the same pricing rules that we learned about before. The table below shows an overview of a few machine types, number of ML training units for that type, and the overall hourly cost.

**Table 18.6. Costs for various machine types**

Machine type	ML training units	US cost	Europe / Asia cost
standard	1	\$0.49 per hour	\$0.54 per hour
standard_gpu	3	\$1.47 per hour	\$1.62 per hour
complex_model_m	3	\$1.47 per hour	\$1.62 per hour
complex_model_m_gpu	12	\$5.88 per hour	\$6.48 per hour

To see how this works in practice, let's look at our example from before and calculate how much it would cost on an hourly basis. Recall that in our previous example configuration file we customized the types of all the different machines and set specific numbers of servers. The following table shows the totals of each.

**Table 18.7. Summary of ML training units with a custom configuration**

Role	Machine type	Number	ML training units
Master	standard	1	1
Worker	standard_gpu	10	30
Parameter server	large_model	2	6
<b>Total</b>			37

This means that our example configuration consumes a total of 37 ML training units, which at US-based prices would be \$18.13 per hour. Assuming our job completed quickly (under the 10 minute minimum), the job itself would have cost just about three dollars ( $10 \text{ minutes} / (60 \text{ minutes per hour}) * \$18.13 \text{ per hour} = \$3.02167$ ).

Ultimately, calculating this out each time is going to be frustrating. Luckily we can jump right to the end by looking at the job itself either in the command-line or the Cloud Console, where we can see the number of ML training units consumed in a given job. Shown below is the Cloud Console for our example training job.

**Figure 18.20. Looking at the details of a training job in the Cloud Console**

ML Engine	Job details
 <a href="#">Jobs</a>  <a href="#">Models</a>	<a href="#">←</a> <a href="#">Job details</a> <b>census1</b> <span style="color: green;">✓ Succeeded (2 min 2 sec)</span> Creation time: Nov 3, 2017, 7:28:23 AM Start time: Nov 3, 2017, 7:29:54 AM End time: Nov 3, 2017, 7:30:25 AM Logs: <a href="#">View logs</a> Consumed ML units: 1.67

We can also see the same information in the command-line instead of the browser by using the `describe` subcommand to request the details of a job. Shown below is the same information about the job in the command-line.

#### **Listing 18.20. Viewing the details of a training job using the command-line**

```
$ gcloud ml-engine jobs describe census1
# ... More information here ...
trainingInput:
  # ...
  region: us-central1
  runtimeVersion: '1.2'
  scaleTier: BASIC
trainingOutput:
  consumedMLUnits: 1.67 ①
```

① Here we can see that this consumed 1.67 ML training units.

#### **18.4.2 Prediction costs**

As we learned before, predicting consumes resources just like training, however the prediction work is done entirely by prediction nodes. Although these nodes act just like the others, we don't have the ability to customize them, and have much less control over how many of them are running at any given time. As a result, and just like the costs for training, predicting is also based primarily on an hourly cost for each prediction node running. Currently, nodes in US-based locations cost \$0.40 per hour, and Europe- or Asia-based nodes cost \$0.44 per hour. This means that if you end up consuming 5 minutes worth of 10 prediction nodes' resources, the cost would come out to about \$0.33 ( $\$0.40 \text{ per hour} * 5 \text{ minutes} / 60 \text{ minutes per hour} * 10 \text{ nodes}$ ).

Unlike training jobs, there is also a flat rate of \$0.10 per 1,000 predictions (\$0.11 for non-US locations) in addition to the hourly-based costs. Further, this cost per prediction applies the same to individual on-line predictions as well as each individual prediction in a batch job. This means that if, following our example above, was a 5-minute prediction job that covered 10,000 data points, the per-prediction fee would be \$1.00 ( $\$0.10 \text{ per chunk of 1,000 predictions} * 10 \text{ chunks}$ ). This would bring our overall cost for the prediction job to a grand total of about \$1.33.

At this point you should have a pretty good grasp on how the bill is calculated, however this still leaves the question of figuring out exactly how many prediction node hours were consumed. Luckily you can view this in the details for each job just as we did with training jobs. Shown below is the view of our prediction job in the command-line where we can clearly see how many prediction node hours were consumed as well as how many predictions were performed.

#### **Listing 18.21. Viewing the details of a prediction job.**

```
$ gcloud ml-engine jobs describe prediction1
# ... More information here ...
predictionOutput:
```

```

nodeHours: 0.24          ①
outputPath: gs://your-ml-bucket-name-here/prediction1-output
predictionCount: '10'    ②
startTime: '2017-11-03T14:15:41Z'
state: SUCCEEDED

```

- ① Here we can see that we consumed 0.24 prediction node hours.
- ② In this case, those node hours went towards making 10 predictions.

Based on this, this job cost \$0.0001 for the predictions themselves ( $10 \text{ predictions} / 10,000 \text{ predictions per chunk} * \$0.10 \text{ per chunk}$ ) and \$0.096 for the node hours consumed ( $0.24 \text{ node hours} * \$0.40 \text{ per hour}$ ) meaning a grand total of \$0.0961 which is rounds to about 10 cents.

## 18.5 Summary

- Machine learning is the overarching concept that we can train computers to perform a task using example data rather than explicitly programming them.
- Neural networks are one method of "training" computers to perform tasks.
- TensorFlow is an open-source framework that makes it easy to express high-level machine learning concepts (such as neural networks) in Python code.
- Cloud Machine Learning Engine (ML Engine) is a hosted service for training and serving machine learning models built with TensorFlow.
- You can configure the underlying virtual hardware to be used in ML Engine, using either predefined tiers or more specific parameters (e.g. machine types).
- ML Engine charges based on hourly resource consumption (similar to other computing-focused services like Compute Engine) for both training and prediction jobs.

# Part 5

## *Data processing and analytics*

Large-scale data processing has become pretty important ever since "big data" became a buzz word of most enterprises. And as you might guess, processing and analyzing loads and loads of data (measured in Terabytes, Petabytes, or more) is a pretty complicated job. In this section we'll explore some of the tools available on Google Cloud Platform that were designed to simplify this work.

We'll start by looking at BigQuery which allows you to query immense amounts of data very quickly, and then move onto Cloud Dataflow where you can take your Apache Beam data processing pipelines and execute them on Google's infrastructure. Lastly, we'll look at how you may want to communicate across lots of systems using Cloud Pub/Sub as the glue in your various data processing jobs.

# 19

## *BigQuery: Highly scalable data warehouse*

### **This chapter covers:**

- What is BigQuery?
- How does BigQuery work under the hood?
- Bulk loading and streaming data into BigQuery
- Querying data
- How pricing works

## **19.1 What is BigQuery?**

If you deal with a lot of data, you probably remember the frustration of sitting around for a few minutes (or hours, or days) waiting for a query to finish running. At some point you may have looked at MapReduce (e.g., Hadoop) to speed up some of the larger jobs and then been frustrated again when every little change meant you had to change your code, re-compile, re-deploy, and run the job again. Which leads us to BigQuery: a relational-style cloud database that is capable of querying enormous amounts of data in seconds rather than hours.

Since BigQuery uses with SQL instead of Java or C++ code, exploring large data sets is both easy and fast, meaning you can run a query, tweak it a bit if it's not quite what you wanted, and run the query again. That said, it's important to remember the "analytical" nature of BigQuery because even though BigQuery is capable of running traditional OLTP-style queries (e.g., `UPDATE table SET name = 'Jimmy' WHERE id = 1`), it's most powerful when used as an analytical tool for scanning, filtering, and aggregating lots and lots of rows into some meaningful summary data.

### **19.1.1 Why BigQuery?**

Now that you understand what BigQuery is and what it's used for, you may be confused about why you might use BigQuery instead of some of the other systems out there. For example, why can't we just use MySQL to explore our data? As a matter of fact, you can use MySQL for most cases, but as you have to scan over more and more data, MySQL will start falling over. When that happens, it makes sense to start exploring other options.

First we might try to tune MySQL's performance-related parameters so that certain queries might run faster. Then we might try to turn on read-replicas so that we aren't running super difficult queries on the same database that handles user-facing requests. Next we might look at using a data warehouse system like Netezza, but the price tag of these things can be pretty big (usually millions of dollars), which could be more than we're willing to pay. So what then?

This is exactly where BigQuery can come in to save the day. We'll explore the pricing model for BigQuery later on, but keeping true to the promise of cloud infrastructure, BigQuery combines some of the power of traditional data warehouse systems while only charging for what you actually use. Let's take a quick look at how it works under the hood so that you can see why something like MySQL might fall over, and why BigQuery doesn't.

### **19.1.2 How does BigQuery work?**

As you might guess, you could write an entire book just on BigQuery and the underlying technology (and someone has), so while this section will cover the inner workings of BigQuery in general, if you're interested in even more detail on how BigQuery handles enormous amounts of data so easily, you should check out one of the books focused specifically on BigQuery such as "Google BigQuery Analytics" by Jordan Tigani and Siddartha Naidu. Overall, this means that this chapter will be a bit light on underlying theory and advanced concepts, and instead focus on practical usage of BigQuery in an application.

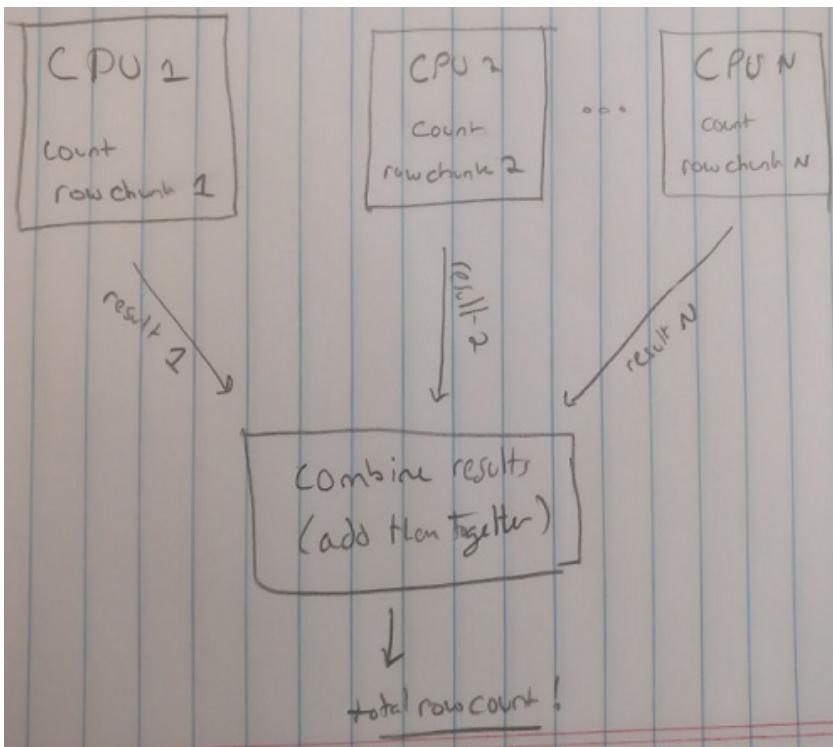
The coolest thing about BigQuery is generally thought to be the sheer amount of data it can handle, while looking mostly like any other SQL database (like MySQL). So how exactly can BigQuery do what MySQL can't do? Let's start by looking at the problem's two parts. First, if we have billions rows of data that we need to filter, that means we need to do billions of comparisons, which requires a lot of computing power. Second, the comparisons have to be done on data that is stored somewhere, and the drives that store that data have limits on how quickly data can flow out of the drive to the computer that is doing those comparisons. These two problems are the fundamental issues we need to solve, so let's look at how BigQuery tries to address each, starting with computing capacity.

#### **SCALING COMPUTE CAPACITY**

The computation aspect of this problem was originally tackled by using the MapReduce algorithm where data is chopped into manageable pieces (the "map" stage)

and then reduced to a summary of the piece (the "reduce" stage). This speeds up the entire process by parallelizing the work to lots and lots of different computers, each working on some subset of the problem. For example, if you had a few billion rows and you wanted to count them, the traditional way to do this would be to run a script on a computer that iterates through all the rows and keeps a counter of the total number of rows, which would take a pretty long time. Using MapReduce you could speed this up by using 1,000 computers where each is responsible for counting 1 one-thousandth of the rows, and then summing up the 1,000 separate counts to get the full count.

**Figure 19.1. Counting a few billion rows by breaking into chunks**



In short, this is what BigQuery does under the hood. Google Cloud Platform has thousands of CPUs in a pool dedicated to handling requests from BigQuery. When you execute a query, it momentarily gives you access to that computing capacity, with each unit of computing power handling a small piece of the data. Once all the little pieces of work are done, BigQuery joins them all back together and gives you a query result. In a sense, BigQuery is sort of like having access to an enormous cluster of machines that you can use for a few seconds at a time to execute your SQL query.

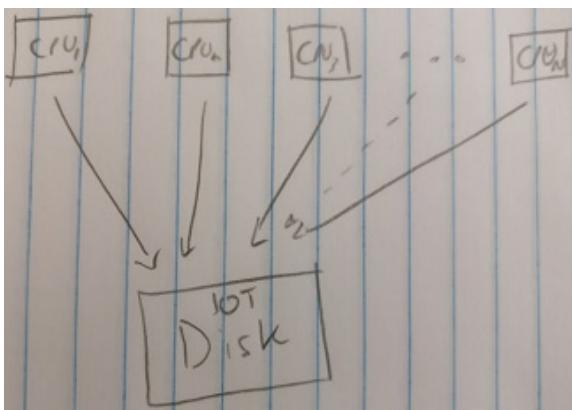
#### SCALING STORAGE THROUGHPUT

As you know, when you store data it ends up on a physical disk somewhere. While we sometimes take those disks for granted, they become incredibly important when we

start demanding extreme performance out of them. Sometimes we fix the problem by changing the type of disk, for example, solid-state disks are better suited to random data access (read the bytes at position 1, then at position 392, then at position 5) whereas mechanical disks are better for sequential data access (read the bytes from position 1 through position 392), but eventually the performance we need just isn't possible with a single disk drive. Also, as disks have gotten bigger and bigger, getting all of the data out of a single disk takes longer longer. This is because while the storage capacity of disks has been growing, the bandwidth limitations of disks haven't necessarily kept up.

When we solved the computational capacity problem above by splitting the problem up into many chunks and using lots of CPUs to crunch on each piece in parallel, we never really thought about how we'll make sure that all of the different CPUs each have access to the chunks of data. In other words, if these thousands of CPUs all requested the data from a single hard drive the drive would get overwhelmed in no time. The problem is further compounded by the fact that the total amount of data we need to query is potentially enormous.

**Figure 19.2. A single disk can't handle requests from tons of CPUs**

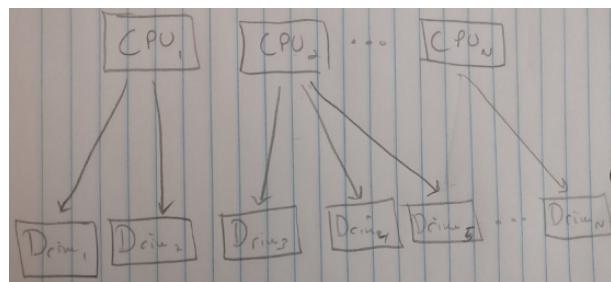


To make this more concrete most drives, regardless of capacity, can typically sustain hundreds of megabytes per second of throughput. This means that pulling all the data off of a single 10 terabyte drive (assuming a 500 MB/s sustained transfer rate) would take about 5 hours! So if 1,000 CPUs all asked for their chunk of data (1000 chunks of 10 GB each), it would take about 5 hours with a best case of about 20 seconds per 10 GB chunk. And all because the single disk acts as a bottle neck because it can only send data to the CPUs at a few hundred megabytes per second.

To fix this, we could split it across lots of different physical drives so that when all of these CPUs start asking for their chunks of data, they are handled by lots of different drives. This would mean that even though no drive alone has the ability to ship all the bytes to the CPUs, the pool of all of the drives is capable of shipping all that data very quickly. For example, If we were to take those same 10 terabytes and split them across 10,000 separate drives, this would result in 1 GB stored on each drive. Looking at "the

fleet" of all the drives, the total throughput available from would be around 5,000,000 MB/s (or 5 TB/s). This also means that each drive could ship the 1 GB they're responsible for in around 2 seconds. If we followed the example above with 1,000 separate CPUs each reading their 10 GB chunk (1 one-thousandth of the 10 TB), they'd get the 10 GB in 2 seconds because each one would read ten 1 GB chunks, with each chunk coming from one of 10 different drives.

**Figure 19.3. Sharding data across multiple disks**



As you can see, sharding the data across lots and lots of drives and transporting it to lots and lots of CPUs for processing means that you can potentially read and process enormous amounts of data incredibly quickly. Under the hood, Google is actually doing this, using a custom-built storage system called Colossus which handles splitting and replicating all of the data, so that BigQuery doesn't have to worry about it. Now that you have a grasp of what BigQuery is doing under the hood, let's look at some of the high-level concepts you'll need to understand in order to use BigQuery.

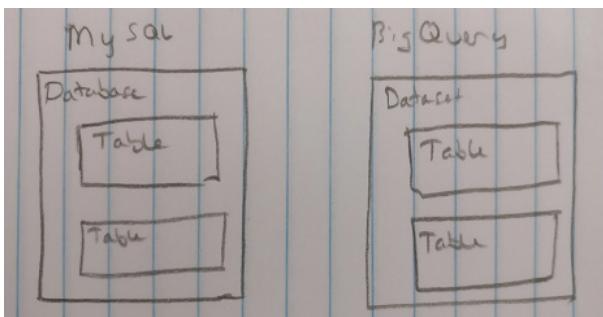
### 19.1.3 Concepts

As you learned already, BigQuery is incredibly SQL-like which means that we can draw very close comparisons to the things you're already familiar with in systems like MySQL. Let's start from the highest level and look at the things that act as containers for data.

#### DATASETS AND TABLES

Just like a relational database has databases which contain tables, BigQuery has datasets which contain tables. The datasets do little else besides act as containers, and the tables, again just like a relational database, are collections of rows. Unlike a relational database, you don't necessarily control the details of the underlying storage systems, so while datasets act as collections of tables, you have a bit less control over the technical aspects of those tables than you would with a system like MySQL or PostgreSQL.

**Figure 19.4. BigQuery datasets and tables compared to MySQL databases and tables**



Each table contained in the dataset is defined by a set schema, so you can think of BigQuery in a traditional grid where each row has cells that fit the types and limits of the columns defined in the schema. It is actually bit more complicated than that where a particular column allows nested or repeated values, however we'll dig into that in more detail later on when we look at schemas.

Unlike a traditional relational database, BigQuery rows typically do not have a unique identifier column, primarily because BigQuery isn't meant for transactional queries where a unique ID is required to address a single row. Since BigQuery is intended to be used as an analytical storage and querying system, constraints like uniqueness in even a single column are not available. This also means that while data is not technically immutable, since there's now way to de-duplicate rows it's not possible to guarantee that a request to update data in BigQuery will only address the exact row that you intended. Otherwise, BigQuery will accept most common SQL-style requests like `SELECT` statements as well as `UPDATE`, `INSERT`, and `DELETE` statements with potentially complex `WHERE` clauses, and fancy `JOIN` operations.

Before we move on there is one final thing to mention about tables that is pretty interesting. Usually with a database, you start by loading data into the database and then later you run queries over the data you loaded, however since BigQuery is part of Google Cloud Platform it's technically possible to transfer the querying power from BigQuery over to other storage services. In other words, in addition to querying data already loaded into a table, BigQuery can run queries over data stored in other storage services in Google Cloud Platform such as Cloud Storage, Cloud Datastore, or Cloud Bigtable, which we'll explore later on. Now that we have a pretty good understanding of BigQuery tables, let's look at the schemas that define their structures.

### SCHEMAS

Just like other SQL databases, BigQuery tables have a structured schema which in turn has the standard data types that you're used to such as `INTEGER`, `TIMESTAMP`, and `STRING` (sometimes known as `VARCHAR`). Additionally, just like a regular relational database fields can be required or nullable (just like `NULL` or `NOT NULL`). Unlike a relational database however, schemas are defined and set as part of an API call rather than run as a query. In other words, where in MySQL you'd execute a query starting

with `CREATE TABLE` to define the schema for the table, BigQuery doesn't use SQL for requests related to the schema. Instead, those types of queries are sent to the BigQuery API itself and the schema is part of that API call.

For example, you might have a table of "people" with fields for each person's name, age, and birth date, but instead of running a query that looks like `CREATE TABLE`, you'd make an API call to the BigQuery service passing along the schema as part of that message. This means the schema itself can be represented as a list of JSON objects, each with information about a single field. In the following example, notice how the `NULLABLE` and `REQUIRED` (SQL's `NOT NULL`) are listed as the "mode" of the field.

#### **Listing 19.1. Example schema for the "people" table.**

```
[{"name": "name",      "type": "STRING",      "mode": "REQUIRED"},  
 {"name": "age",       "type": "INTEGER",     "mode": "NULLABLE"},  
 {"name": "birthdate", "type": "TIMESTAMP",   "mode": "NULLABLE"}]
```

So far this seems pretty straight forward, but things get a bit more complicated with some of the other modes or field types. To start with, there is an additional mode called `REPEATED` which currently isn't very common in most relational databases. Repeated fields do just as their name implies, taking the type provided and turning it into an array equivalent. In other words, a repeated `INTEGER` field acts just like an array of integers. BigQuery comes with special ways of decomposing these repeated fields, such as allowing you to count the number of items in a repeated field, or filtering so long as a single entry of the field matches a given value, so while they are certainly non-standard they shouldn't feel completely out of place if you think of each row in BigQuery as a separate JSON object.

Next, there is a field type called `RECORD` which acts like a JSON object, allowing you to nest rows within rows. For example, the `people` table could have a `RECORD` type field called `favorite_book` which in turn would have fields for the `title` and `author` (which would both be `STRING` types). Following the pattern above, this is not a common pattern in standard SQL because it would instead be normalized into a separate table (a table of books and the `favorite_book` field would be a foreign key), but in BigQuery this type of in-lining or de-normalizing is supported and can be quite useful, particularly if the data (in this case, the book title and author) is never needed in a different context (in other words, only ever looked at alongside the people who have the book as a favorite).

We'll demonstrate how some of these work a bit later, but the important thing to remember here is that BigQuery has two non-standard field modifiers (the `REPEATED` mode and the `RECORD` type) and lacks some of the normalization features of traditional SQL databases (such as `UNIQUE`, `FOREIGN KEY`, and explicit indexes). Aside from those additions and omissions, BigQuery should feel very similar to other relational databases. Next let's look at the concepts involved in interacting with BigQuery, starting with jobs.

## JOBS

Since API requests to BigQuery tend to involve lots of data, it's likely that while a single request will finish quickly, it probably won't finish "right away" (that is, it may take a few seconds at least). After all, it's pretty difficult to load a terabyte of data into a storage system in just a few milliseconds. As a result, BigQuery chose to keep most interactions that aren't immediate scoped to using jobs which represent the work that could take a little while.

In other words, instead of making a call to load some data (which might look like `bigquery.loadData('/path/to/1tb_of_data.csv')`), you create a semi-persistent resource called a *job* which is responsible for executing the work requested, reporting progress along the way, and returning the success or failure result when the work is done or is halted (e.g., something like `job = bigquery.createJob('SELECT ... FROM table WHERE ...')`). So what can these jobs do? There are four fundamental operations that you can accomplish with jobs:

1. Querying for data,
2. Loading new data into BigQuery,
3. Copying data between from one table to another, and
4. Extracting (or exporting) data from BigQuery to somewhere else (like GCS).

While these each seem to be doing entirely different things, they are fundamentally about taking data from one place and putting it in another, potentially with some transformation applied over the data somewhere along the way. For example, since a "query" job can have a separate table as the destination, a "copy" job is sort of like a special type of query job, where the query (in SQL here) is equivalent to `SELECT * FROM table` with a destination table set in the configuration. What this means is that there may be several different ways of accomplishing the same thing, but all these use jobs to keep track of work being done.

Finally, since jobs are treated as unique resources, this means that you can do the typical operations over jobs that you can over things like tables or datasets. This means that you can list all of the jobs you've run, cancel any currently running jobs, or retrieve details of a job created in the past. Compared to the typical relational database, the closest comparison is keeping a query log stored on the server, but that doesn't provide quite the same level of detail. To make this all more concrete, let's look through some examples of how you actually use BigQuery, starting with querying some shared datasets.

## 19.2 *Interacting with BigQuery*

BigQuery, like any other hosted database, is accessible via its API, which means that you really have several convenient ways of talking to it: with the UI in the Cloud Console, on the command-line with the `bq` tool, and using the client library of your choice (we'll discuss the Node.js client in this chapter). Let's start with the simplest by using the UI to run some queries against a shared public dataset.

## 19.2.1 Querying data

As the name suggests, the main purpose of BigQuery is to query your data, so we'll start off by trying out some queries. Let's kick things off by going to the Cloud Console and choosing "BigQuery" from the left-side navigation menu. Unlike the APIs we've used so far, you'll actually be brought to a new page (or tab) focused exclusively on BigQuery. If you then click on "Public Datasets" you'll land on a page showing off a bunch of these datasets.

**Figure 19.5. BigQuery's public datasets**

The screenshot shows the Google BigQuery interface. On the left, there's a sidebar with 'COMPOSE QUERY' and history sections. Below that is a search bar with 'Filter by ID or label' and a dropdown set to 'jj-personal'. A message says 'No datasets found in this project. Please create a dataset or select a new project from the menu above.' Under 'Public Datasets', there are three entries:

- NOAA Global Surface Summary of the Day Weather Data**: This dataset was created by the National Oceanic and Atmospheric Administration (NOAA) and includes global data obtained from the USAF Climatology Center. It covers GSOD data between 1929 and 2016, from over 9000 stations.
- US Disease Surveillance Data**: Published by the US Department of Health and Human Services, it includes weekly surveillance reports of nationally notifiable diseases for all U.S. cities and states between 1888 and 2013. The data set consists of eight important vaccine-preventable contagious diseases: diphtheria, hepatitis A, measles, mumps, pertussis, polio, rubella and smallpox.
- USA Names Data**: Created by the Social Security Administration, it contains all names from Social Security card applications for births after 1879.

If you click on one of the choices (in this case, let's try out the yellow taxi dataset) you'll be brought to a summary of the data which includes both some details of the dataset itself as well as a list of the tables. If you click on the tables, you'll be brought to a page that shows the most important piece: the schema.

**Figure 19.6. Yellow taxi trips schema**

The screenshot shows the Google BigQuery interface. On the left, there's a sidebar with a red 'COMPOSE QUERY' button, 'Query History', 'Job History', and a dropdown for 'jjg-personal' which is expanded to show 'No datasets found in this project.' and a message to 'Please create a dataset or select a new project from the menu above.' Below that is a section for 'githubarchive' and a expanded section for 'Public Datasets' containing links to various public datasets.

The main area is titled 'Table Details: trips'. It has tabs for 'Schema' (selected), 'Details', and 'Preview'. To the right are buttons for 'Query Table', 'Copy Table', 'Export Table', and 'Delete Table'. The 'Schema' tab displays the following table:

Field	Type	Nullable	Description
<code>vendor_id</code>	STRING	NULLABLE	A designation for the technology vendor that provided the record. CMT=Creative Mobile Technologies VTS= VeriFone, Inc. DDS=Digital Dispatch Systems
<code>pickup_datetime</code>	TIMESTAMP	NULLABLE	The date and time when the meter was engaged.
<code>dropoff_datetime</code>	TIMESTAMP	NULLABLE	The date and time when the meter was disengaged.
<code>pickup_longitude</code>	FLOAT	NULLABLE	Longitude where the meter was engaged.
<code>pickup_latitude</code>	FLOAT	NULLABLE	Latitude where the meter was engaged.
<code>dropoff_longitude</code>	FLOAT	NULLABLE	Longitude where the meter was disengaged.
<code>dropoff_latitude</code>	FLOAT	NULLABLE	Latitude where the meter was disengaged.
<code>rate_code</code>	STRING	NULLABLE	The final rate code in effect at the end of the trip. 1= Standard rate 2=JFK 3=Newark 4=Nassau or Westchester 5=Negotiated fare 6=Group ride

Here you can see the list of fields available, the data type, and a short description of the data that lives in that field. Notice in particular that all of these fields are `NULLABLE`, so there's no guarantee that a value will be in there. If you click the "Details" tab at the top, you'll be able to see an overview of the table which in this case shows that this table contains about 130 GB in total spread across over a billion rows. In the image below we can see the complete "Table ID" which is a combination of the project (in this case, `nyc-tlc`) with the dataset (`yellow`), and finally the table (`trips`). Keep this in mind as we run into this format when writing queries.

**Figure 19.7. Yellow taxi trips table details****Table Info**

<b>Table ID</b>	nyc-tlc:yellow.trips
<b>Table Size</b>	130 GB
<b>Long Term Storage Size</b>	130 GB
<b>Number of Rows</b>	1,108,779,463
<b>Creation Time</b>	Sep 25, 2015, 2:29:01 PM
<b>Last Modified</b>	Dec 24, 2015, 10:34:53 AM
<b>Data Location</b>	US
<b>Labels</b>	None <span style="border: 1px solid #ccc; padding: 2px;">Edit</span>

There are a few queries we can run that would be pretty interesting, but often the initial worry is: "Won't this take a few minutes?" This is a reasonable first thought, after all querying 1.1 billion records in PostgreSQL would probably take a while, so let's start with one that any other database could probably handle easily as long as there was an index: the most expensive ride.

**Listing 19.2. Querying the trips table for the most expensive trip**

```
SELECT total_amount, pickup_datetime, trip_distance
  FROM `nyc-tlc.yellow.trips`
 ORDER BY total_amount DESC
 LIMIT 1;
```

To run this query over the table, click the button at the top right that says "Query Table" and enter the query above. In case you're not familiar with SQL, this query basically asks the table for some details sorted by the total trip cost but only gives us the first (most expensive) trip. Before you run this query exactly as it's formatted, you'll need to tell BigQuery **not** to use the "legacy" (i.e., "old") SQL style syntax. The newer syntax uses back-ticks for escaping table names rather than the square brackets from when BigQuery first launched. You can find this setting by clicking "Show Options" and then un-checking the box that says "Use Legacy SQL".

When you click "Run Query", BigQuery will get to work and should return a result in somewhere around 2 seconds (in my case it was 1.7 seconds). It will also show you how much data was queried in that time, which in my case was about 25 GB, meaning that BigQuery sifted through about 15 GB per second in order to give back this result. That's pretty quick, but probably not as scary as the fact that there was a trip that cost someone almost \$4 million. (Even as a New Yorker I have no idea how that happens.)

**Figure 19.8. BigQuery results of the most expensive trip**

The screenshot shows the BigQuery Query Editor interface. At the top, there's a menu bar with 'New Query' and 'Query Editor / UDF Editor'. Below the menu is a code editor window containing the following SQL query:

```

1 SELECT total_amount, pickup_datetime, trip_distance
2   FROM [nyc-tlc:yellow.trips]
3 ORDER BY total_amount DESC
4 LIMIT 1;
    
```

Below the code editor are several buttons: 'RUN QUERY', 'Save Query', 'Save View', 'Format Query', and 'Show Options'. To the right of these buttons is a status message: 'Ctrl + Enter: run query, Tab or Ctrl + Space: autocomplete.' and 'Query complete (1.7s elapsed, 24.8 GB processed)' with a checkmark icon.

At the bottom of the interface are tabs for 'Results', 'Explanation', and 'Job Information'. There are also buttons for 'Download as CSV', 'Download as JSON', 'Save as Table', and 'Save to Google Sheets'. The 'Results' tab is selected, displaying a single row of data:

Row	total_amount	pickup_datetime	trip_distance
1	3950611.6	2015-01-18 19:24:15 UTC	5.32

Below the table are two buttons: 'Table' and 'JSON'.

This seems interesting, but doesn't really show off the power of BigQuery, so let's try something a bit more intricate. We have pick up and drop off times and locations, so what if we were trying to figure out what was the most common hour of the day that people were picked up? This means we have to take the pickup time and group by the "hour" part of that, then sort by the number of trips falling in each hour. In SQL, this is really not that complicated.

#### **Listing 19.3. Query to find trip counts grouped by the time of pick-up**

```

SELECT HOUR(pickup_datetime) as hour, COUNT(*) as count
  FROM `nyc-tlc.yellow.trips`
 GROUP BY hour
 ORDER BY count DESC;
    
```

Running this query shows that the evening pick-ups are most common (that is, 6-10 PM) and the early morning pick-ups are least common (that is, 3, 4, and 5 AM).

**Figure 19.9. Results of querying with a grouping by pick-up time**

The screenshot shows a SQL query interface with the following details:

- Query Editor:** The tab is selected.
- SQL:** The code is written in SQL.
 

```
1 SELECT HOUR(pickup_datetime) as hour, COUNT(*) as count
2   FROM [nyc-tlc:yellow.trips]
3   GROUP BY hour
4 ORDER BY count DESC;
```
- Run Query:** A red button to execute the query.
- Save Options:** Buttons for Save Query, Save View, Format Query, and Show Options.
- Feedback:** "Query complete (2.2s elapsed, 8.26 GB processed)" with a green checkmark.
- Results:** The results are displayed in a table with columns: Row, hour, and count.
- Data:**

Row	hour	count
1	19	69489572
2	18	66744014
3	20	65187372
4	21	63598261
5	22	61658872
6	14	55877423
7	17	55153837
8	23	54934466
9	12	54448465
- Download Options:** Buttons for Download as CSV, Download as JSON, Save as Table, and Save to Google Sheets.

Perhaps there's more information here if we look at it also broken down by the day of the week. Let's try adding that into the mix.

#### **Listing 19.4. Query looking at both time of the day and day of the week**

```
SELECT DAYOFWEEK(pickup_datetime) as day, HOUR(pickup_datetime) as hour,
       COUNT(*) as count
  FROM `nyc-tlc.yellow.trips`
 GROUP BY day, hour
 ORDER BY count DESC;
```

Running this shows that the evening hours are most popular towards the end of the week (that is, Thursday and Friday at 7PM top the charts).

**Figure 19.10. Results showing the day and hour with most pick-ups**

New Query ?

Query Editor UDF Editor X

```
1 SELECT DAYOFWEEK(pickup_datetime) as day, HOUR(pickup_datetime) as hour,
2   COUNT(*) as count
3   FROM [nyc-tlc:yellow.trips]
4   GROUP BY day, hour
5   ORDER BY count DESC;
```

SQL

RUN QUERY Save Query Save View Format Query Show Options Query complete (3.0s elapsed, cached) ✓

Ctrl + Enter: run query, Tab or Ctrl + Space: autocomplete.

Row	day	hour	count
1	6	19	10811458
2	5	19	10754884
3	4	19	10590629
4	5	20	10407461
5	5	21	10355792
6	3	19	10346716

If all you're thinking right now is "So what? MySQL can do all of this just fine." then this means that BigQuery has done its job. The whole purpose of BigQuery is to feel just like running an analytical query with any other SQL database, but way faster. This is because we tend to forget that these queries we're running are scanning over more than a billion records stored in BigQuery, and doing so like it was just a few million records in a MySQL database. To make things even cooler, if you were to increase the size of the data by an order of magnitude (10x what it is today, to 10 billion rows) these queries would all take about the same amount of time as they do now.

Running queries in the UI is fine, but what if you wanted to build something that displayed data pulled from BigQuery? This is where the client library (@google-cloud/bigquery) comes in. To see how it works with BigQuery, let's write some code that finds the most expensive ride. If you haven't already, start by installing the Node.js client for BigQuery using `npm install @google-cloud/bigquery@1.0.0`.

#### **Listing 19.5. Using @google-cloud/bigquery to select the most expensive taxi trip**

```
const BigQuery = require('@google-cloud/bigquery');
const bigquery = new BigQuery({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});
```

1

```

const query = `SELECT total_amount, pickup_datetime, trip_distance
  FROM `nyc-tlc.yellow.trips\`
  ORDER BY total_amount DESC
  LIMIT 1;`          ②

bigquery.createQueryJob(query).then((data) => {      ③
  const job = data[0];    ④
  return job.getQueryResults({timeoutMs: 10000});   ⑤
}).then((data) => {
  const rows = data[0];
  console.log(rows[0]);  ⑥
});

```

- ① Even though we're running a query against a dataset in another project, we'll be creating a job in **our** project, so we use our own project ID here.
- ② Just like before we define the query as a string, referencing the NYC trips data set in the FROM section of the query.
- ③ Here, we're creating a BigQuery job, which is responsible for doing the actual work.
- ④ Once the `createQueryJob` method has finished, it will immediately return a job resource as the first argument.
- ⑤ Use the `getQueryResults` method that lives on the BigQuery job resource, making special note to say that we should wait up to 10 seconds (10,000 ms) for results to be ready.
- ⑥ Once we get the results back, we know there is only one row (due to the `LIMIT 1`), so we print out the first row's information.

If you run this code, you should see the same output we saw when we tried it in the BigQuery UI, with that crazy trip costing \$4 million.

#### **Listing 19.6. Output from selecting the most expensive trip**

```
{
  total_amount: 3950611.6,
  pickup_datetime: { value: '2015-01-18 19:24:15.000' },
  trip_distance: 5.32 }
```

In this case, all of the columns returned had specific names (like `total_amount`), but what about those aggregated columns that aren't explicitly named? In other words, what if we wanted to find the total cost of all of the trips?

#### **Listing 19.7. Query to select the total cost of all trips**

```
SELECT SUM(total_amount) FROM `nyc-tlc.yellow.trips`;
```

If we replace the `query` value in our code above, the results should look something like the following, showing that BigQuery's API will apply some automatically generated field names to the unnamed fields, using the order of the field in the query as an index.

#### **Listing 19.8. Output of selecting the sum of all taxi trips**

```
{ f0_: 14569463158.355078 }
```

As you can see, the "first" field in the query (`SUM(total_amount)`) is named as `f0_` meaning "field 0". Querying public datasets can be fun, but it doesn't seem to be the best use of BigQuery especially when you likely have your own data sitting around that you want to query. Let's take a look at how to put your own data into BigQuery and the different ingestion models that it supports.

### **19.2.2 Loading data**

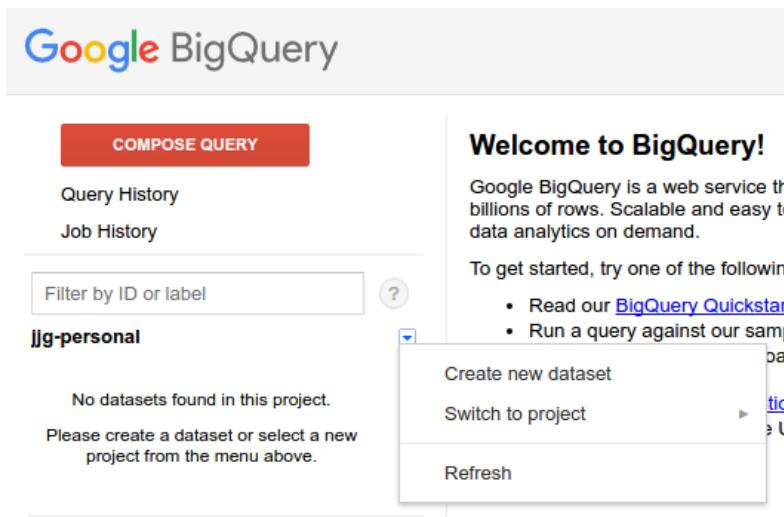
As you learned earlier, BigQuery jobs support multiple different types of operations, one of them being for loading new data. But there are actually multiple ways of getting data from a source somewhere into a BigQuery table. Additionally, as we explored earlier, BigQuery tables themselves may be based on other data sources such as Bigtable, Cloud Datastore, or Cloud Storage. Let's start by looking at how you might take a chunk of data (such as a big CSV file) and load it into BigQuery as a table that you can query.

#### **BULK LOADING DATA INTO BIGQUERY**

When we refer to "bulk loading" we're talking about the concept of taking a big chunk of arbitrary data (such as a bunch of CSVs or JSON objects) and loading them into a BigQuery table. In many ways this is similar to a MySQL `LOAD DATA` query which is typically used for restoring data that was backed-up as a CSV file. As you might guess, there are quite a lot of options to configure when loading data (such as data compression, character encoding) so let's start with the basics.

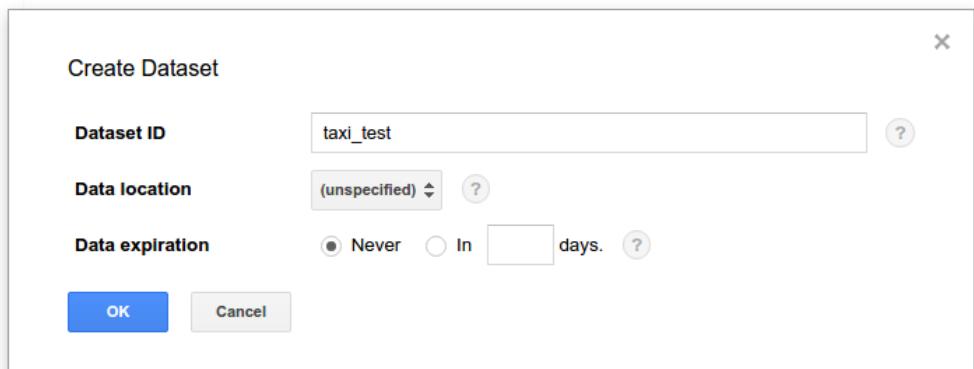
Imagine that we're recreating a table to store the data from taxi rides, similar to the one we've been querying in the shared dataset. To start, we need to create a dataset, then a table, and then set the schema to fit our data. We'll do all of these in one step because each step on its own is pretty basic. The easiest way to do all of this is using the BigQuery UI in the Cloud Console, so start by heading back BigQuery's interface. On the left-hand side, you should see "No datasets found", so use the little arrow next to your project name and choose "Create new dataset".

**Figure 19.11. Menu showing how to create a new dataset**



When you click this, you'll see a window pop up where you can choose the ID for your dataset. Before you go filling it out, it's important to note that BigQuery IDs have a tiny difference from the IDs of resources in the rest of Google Cloud Platform: no hyphens. Since BigQuery dataset (and table) IDs are used in SQL queries, hyphens are prohibited. As a result, it's common to use underscores where you'd usually use hyphens (that is, `test_dataset_id` instead of `test-dataset-id`). For this demo, this means we'll call our dataset `taxis_test` (where we would've called it `taxis-test` if hyphens were allowed). You also can choose where the data should live (in the US or the EU), and when data should "expire". For now, let's leave both of these set to the defaults ("unspecified" and "Never").

**Figure 19.12. Form to create a new dataset**



Click OK and your dataset should appear right away, so it's time to create our new table. Just like with creating a new dataset, use the arrow menu to choose "Create a

new table" and you'll see a big form appear.

The first thing to remember is that tables are mutable, so if you forget a field, it's not the end of the world. It does mean, however, that if you add a field after you've already loaded data, the rows that you have will get a `NULL` value for the new field (just like in a regular SQL database). Let's assume that we have a slimmed down version of taxi data that has a few fields for the pick-up and drop-off times, as well as the fare amount. As you'd guess, the times should be `TIMESTAMP` types, and the fare amount would be a `FLOAT`. Some example CSV data is shown below (which you can use later on if you put it in a file).

**Listing 19.9. Sample trip data in CSV format**

```
1493033027,1493033627,8.42  
1493033004,1493033943,18.61  
1493033102,1493033609,9.17  
1493032027,1493033801,24.97
```

Using this information, we can define a table called `trips`, with those three fields under the Schema section. Lastly, if you put the CSV data from those 4 data points into a file, you can use them in the Source Data section.

**Figure 19.13. Creating the trips table****Create Table**

**Source Data**  Create from source  Create empty table

Repeat job	Select Previous Job	<a href="#">?</a>
Location	File upload	Choose file trips.csv (110 bytes)
File format	CSV	

**Destination Table**

Table name	taxi_test	trips	<a href="#">?</a>
Table type	Native table	<a href="#">?</a>	

**Schema**  Automatically detect [?](#)

Name	Type	Mode
pickup_time	TIMESTAMP	REQUIRED
dropoff_time	TIMESTAMP	REQUIRED
fare_amount	FLOAT	REQUIRED

[Add Field](#) [Edit as Text](#)

**Options**

Field delimiter	<input checked="" type="radio"/> Comma <input type="radio"/> Tab <input type="radio"/> Pipe <input type="radio"/> Other	<a href="#">?</a>
Header rows to skip	0	<a href="#">?</a>
Number of errors allowed	0	<a href="#">?</a>
Allow quoted newlines	<input type="checkbox"/>	<a href="#">?</a>
Allow jagged rows	<input type="checkbox"/>	<a href="#">?</a>
Ignore unknown values	<input type="checkbox"/>	<a href="#">?</a>
Write preference	Write if empty	<a href="#">?</a>
Partitioning	None	

[Create Table](#)

Notice how we used the "File upload" data source in this example, but we could have also uploaded the CSV file to Cloud Storage or Google Drive and used the file hosted there as the source. Additionally, even though we use the UI to define the schema, it's also possible to edit the schema as raw text in the JSON format mentioned previously. If you click "Edit as Text" you should see content looking something like the following.

**Listing 19.10. Schema for the trips table as text**

```
[
  {
    "mode": "REQUIRED",
    "name": "pickup_time",
    "type": "TIMESTAMP"
  },
  {
    "mode": "REQUIRED",
    "name": "dropoff_time",
    "type": "TIMESTAMP"
  },
  {
    "mode": "REQUIRED",
    "name": "fare_amount",
    "type": "FLOAT"
  }
]
```

If you click "Create Table", the table with the schema we defined will be created immediately, and a "load data" job is created under the hood. Since the data here is tiny, this job should complete pretty quickly, and you should see a result showing that the data loaded successfully into the new table.

**Figure 19.14. Load data job status**

The screenshot shows the "Load" status for a job named "uploaded file to jjg-personal:taxi\_test.trips". The job was created at 7:38AM on April 24, 2017. It started at 7:38:16 AM and ended at 7:38:18 AM. The destination table is "jjg-personal:taxi\_test.trips". The job used a CSV source format with a single delimiter (',') and an uploaded file source URI. The schema defined three columns: "pickup\_time" as TIMESTAMP, "dropoff\_time" as TIMESTAMP, and "fare\_amount" as FLOAT. There are "Repeat Load Job" buttons for both the main status bar and the individual job entry.

Job ID	jjg-personal:bquijob_16bd6323_15b9fc14216
Creation Time	Apr 24, 2017, 7:38:15 AM
Start Time	Apr 24, 2017, 7:38:16 AM
End Time	Apr 24, 2017, 7:38:18 AM
Destination Table	<a href="#">jjg-personal:taxi_test.trips</a>
Write Preference	Write if empty
Source Format	CSV
Delimiter	,
Source URI	uploaded file
Schema	pickup_time: TIMESTAMP dropoff_time: TIMESTAMP fare_amount: FLOAT

Once the data is loaded, we can check on it by running a SQL query. Since you already know how to "select all rows", let's look at a fancier query that shows us a summary of the cost per minute of our sample trips.

**Listing 19.11. Query for trip cost per minute**

```
SELECT
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/google-cloud-platform-in-action>

**Licensed to Asif Qamar <asif@asifqamar.com>**

```

    TIMESTAMP_DIFF(dropoff_time, pickup_time, MINUTE) AS duration_minutes,
    fare_amount,
    fare_amount / TIMESTAMP_DIFF(dropoff_time, pickup_time, MINUTE) AS
cost_per_minute
FROM
    `your-project-id-here.taxi_test.trips` ①
LIMIT
    1000;

```

- ① Note that you must swap this with your own project ID.

Running this query should show you how much each trip cost on a per-minute basis as you can see here.

**Figure 19.15. Query results for the cost per minute of each trip**

The screenshot shows the BigQuery web interface. At the top, there's a 'New Query' button and tabs for 'Query Editor' and 'UDF Editor'. Below that is the SQL code for the query. The interface includes standard SQL dialect options, run query buttons, and various export and save options. The results tab displays the query output as a table with four columns: Row, duration\_minutes, fare\_amount, and cost\_per\_minute. The data shows four rows of results.

```

1 ▾ SELECT
2   TIMESTAMP_DIFF(dropoff_time, pickup_time, MINUTE) AS duration,
3   fare_amount,
4   fare_amount / TIMESTAMP_DIFF(dropoff_time, pickup_time, MINUTE) AS cost_per_minute
5 ▾ FROM
6   `jjg-personal.taxi_test.trips`
7 ▾ LIMIT
8   1000;

```

Row	duration_minutes	fare_amount	cost_per_minute
1	29	24.97	0.8610344827586206
2	15	18.61	1.2406666666666666
3	10	8.42	0.842
4	8	9.17	1.14625

Obviously the only difference between the job of loading from a sample CSV and a real-life example will be the size of data, so if you want to try that out, try generating a larger file and loading that. To load from GCS, you can just choose "Google Cloud Storage" from the list of locations and enter the GCS-specific URL into the location box.

**Figure 19.16. Loading a larger file from GCS**

## Create Table

**Source Data**  Create from source  Create empty table

Repeat job	Select Previous Job	<a href="#">?</a>	
Location	Google Cloud Storage	gs://jjg-bigquery-test/trips_large.csv	<a href="#">View Files</a>
File format	CSV		

**Destination Table**

Table name	taxi_test	.	trips	<a href="#">?</a>
Table type	Native table	<a href="#">?</a>		

**Schema**  Automatically detect [?](#)

Name	Type	Mode
pickup_time	TIMESTAMP	REQUIRED
dropoff_time	TIMESTAMP	REQUIRED
fare_amount	FLOAT	REQUIRED

[Add Field](#) [Edit as Text](#)

**Options**

Field delimiter	<input checked="" type="radio"/> Comma	<input type="radio"/> Tab	<input type="radio"/> Pipe	<input type="radio"/> Other	<input type="text"/>	<a href="#">?</a>
Header rows to skip	<input type="text" value="0"/>	<a href="#">?</a>				
Number of errors allowed	<input type="text" value="0"/>	<a href="#">?</a>				
Allow quoted newlines	<input type="checkbox"/>	<a href="#">?</a>				
Allow jagged rows	<input type="checkbox"/>	<a href="#">?</a>				
Ignore unknown values	<input type="checkbox"/>	<a href="#">?</a>				
Write preference	Write if empty				<a href="#">?</a>	
Partitioning	None				<a href="#">?</a>	

[Create Table](#)

When you click "Create Table" the data will be pulled in from GCS and loaded into BigQuery. In this case, the loading job takes a few minutes whereas it only took a few seconds before. That said, the example file in this demo was about 3.2 GB in total, so a few minutes isn't so bad.

**Figure 19.17. Loading job results from a larger file on GCS**

## Recent Jobs

The screenshot shows a 'Recent Jobs' interface. At the top, there is a 'Filter jobs' dropdown. Below it, a job card is displayed for a 'Load' operation:

- Status:** Load (green checkmark)
- Destination:** gs://jjg-bigquery-test/trips\_large.csv to jjg-personal:taxi\_test.trips
- Job ID:** jjg-personal:bquijob\_33bb2c4c\_15ba0424aae
- Creation Time:** Apr 24, 2017, 9:59:10 AM
- Start Time:** Apr 24, 2017, 9:59:11 AM
- End Time:** Apr 24, 2017, 10:02:34 AM
- Destination Table:** [jjg-personal:taxi\\_test.trips](#)
- Write Preference:** Write if empty
- Source Format:** CSV
- Delimiter:** ,
- Source URI:** gs://jjg-bigquery-test/trips\_large.csv ([Open in GCS](#))
- Schema:**
  - pickup\_time: TIMESTAMP
  - dropoff\_time: TIMESTAMP
  - fare\_amount: FLOAT

Buttons at the bottom of the card include 'Repeat Load Job' and 'Cancel Job'.

Now you can query the data just as before. As you can see here, counting the number of rows shows quite a bit more data, totaling about 120 million rows.

**Figure 19.18. Total number of rows in the larger file from GCS**

The screenshot shows the 'Query Editor' tab of the BigQuery interface. A new query is being run:

```
1 SELECT COUNT(*) as trip_count FROM `jjg-personal.taxi_test.trips`
```

Below the query, the results are displayed in a table:

Row	trip_count
1	124382105

At the bottom, there are tabs for 'Table' and 'JSON'.

This method of getting data into BigQuery works if you have one chunk of data that isn't really changing at all, but what if you have new data coming in from your

application, such as user interactions, advertisements shown, or products viewed? In that case, bulk loading jobs don't quite make sense, and streaming new data into BigQuery makes for a much better fit. Let's take a look at how that works by seeing how the taxi trips might stream new rows into our table.

### **STREAMING DATA INTO BIGQUERY**

We've explored how to bulk load a big chunk of existing data into BigQuery, but what about if you want your application to generate new rows that you can search over? Doing this is what BigQuery calls "streaming ingestion" or "streaming data", and it refers specifically to sending lots of single data points into BigQuery over time rather than all of those data points at once. In principle, streaming data into BigQuery is incredibly easy using the client library. All you have to do is point at the table you want to add data to, and (in Node.js) use the `insert()` method.

For example, let's imagine that when a taxi ride is over, we want to make sure that we log the trip happened with the pick-up and drop-off times as well as the total fare amount. To do this, we could write a function that takes an object representing the trip and inserts it into our BigQuery trips table.

#### **Listing 19.12. Streaming new data into BigQuery**

```
const BigQuery = require('@google-cloud/bigquery');
const bq = new BigQuery({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

const dataset = bq.dataset('taxi_test');
const table = dataset.table('trips'); ①

const addTripToBigQuery = (trip) => {
  return table.insert({
    pickup_time: trip.pickup_time.getTime() / 1000, ②
    dropoff_time: trip.dropoff_time.getTime() / 1000, ③
    fare_amount: trip.fare_amount
  });
}
```

- ① Get a pointer to the BigQuery table using the `.dataset` and `.table` helper methods.
- ② Use the `.insert` method to load a single row into BigQuery.
- ③ The assumption here is that the value will be a JavaScript Date type, so we want to convert this to a pure Unix timestamp.

The main problem that comes up when you're loading data in many different requests is how to make sure you don't load the same row twice. As you learned earlier, BigQuery is an analytical database, so there's no way to enforce a uniqueness constraint. This means that if a request failed for some reason (e.g., the connection was cut due to network issues) it'd be very difficult to know whether you should resend the same request. On the one hand you could end up with duplicate values which is never

good, on the other hand you may end up with some missing data points that you thought were stored but were actually lost in transit.

To avoid this BigQuery can accept a unique identifier called `insertId` which acts as a way of de-duplicating rows as they're inserted. The concept behind this ID is pretty simple: if BigQuery has seen the ID before it will treat the rows as "already added" and skip adding them. To do this in code, you have to use the raw format of the rows and choose a specific insert ID, like a UUID.

#### **Listing 19.13. Adding rows and avoiding failures**

```
const uuid4 = require('uuid/v4');
const BigQuery = require('@google-cloud/bigquery');
const bigquery = new BigQuery({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

const dataset = bigquery.dataset('taxi_test');
const table = dataset.table('trips');

const addTripToBigQuery = (trip) => {
  const uid = uuid4(); ①
  return table.insert({
    json: { ②
      pickup_time: trip.pickup_time.getTime() / 1000,
      dropoff_time: trip.dropoff_time.getTime() / 1000,
      fare_amount: trip.fare_amount
    },
    insertId: uid ③
  }, {raw: true}); ④
}
```

- ① We'll use a UUID-4 (a random UUID) to act as the insert ID.
- ② The raw format needs to specify the row data in the `json` property.
- ③ The insert ID is set in the `insertId` property.
- ④ We also need to tell the client that this is a `raw` row.

Now when we log a trip, if there is some sort of failure our client will automatically retry the request. If the retried request happens to have already been seen by BigQuery BigQuery will ignore it. In other words, if the request looked like it failed to us but actually worked just fine on BigQuery's side, BigQuery will simply ignore the request rather than adding the same rows all over again.

Note that we're using a random insert ID because a deterministic one (such as a hash) may disregard identical but non-duplicate data. In other words, if two trips started and ended at the exact same time and cost the exact same amount, a hash of that data would be identical which may lead to the second trip being dropped as a duplicate.

**WARNING**

Remember that BigQuery's insert ID is about avoiding making the exact same request twice and should **not** be used as a way to de-duplicate your data. Instead, it's a way to make sure that no data is added twice when a request is retried. If you need unique data, do that first, and bulk-load the unique data.

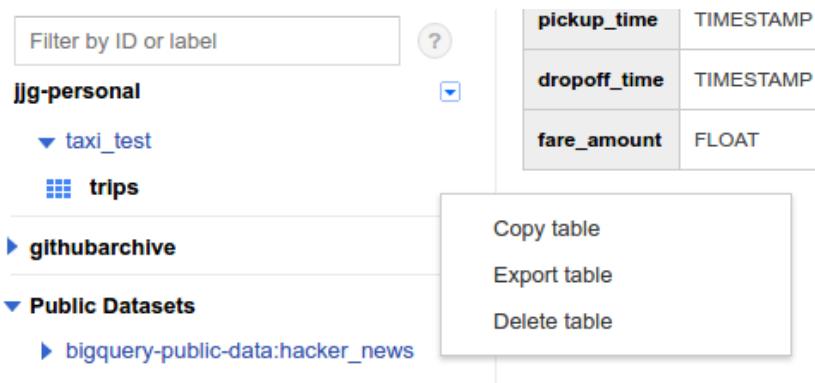
Now all that's left is to call this function at the end of every trip, and the trip information will be added to BigQuery as they happen. This covers the aspect of getting data **into** BigQuery, but what about getting it **out of** BigQuery? Let's take a look at how you can access your data

### 19.2.3 Exporting datasets

So far all we've really talked about is getting data into BigQuery, either through some bulk-loading job or streaming bits of data in, one row at a time. But what about when you want to get your data out? For example, maybe you want to take taxi trip data out of BigQuery in order to do some machine learning on it and predict the cost of a trip based on the locations, pick-up times, etc. Obviously pulling this out through the SQL-like interface won't really solve the problem for you. Luckily there's an easier way to pull data out of BigQuery, which is an export job.

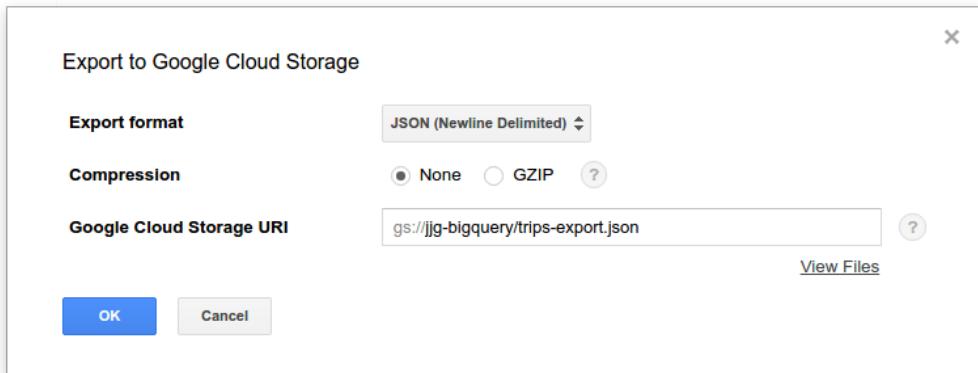
Export jobs are pretty straightforward: they take data out of BigQuery and drop it into Cloud Storage as comma-separated, new-line separated JSON, or Avro. Once there, you can work on it from GCS and re-import it into another table as needed. This means that before you start, you'll need to create a bucket on GCS to store your exported data. Once you have a bucket, exporting is easy from the UI. Just choose the table you want to export, and click on "Export table".

**Figure 19.19. Choose "Export table" from the side menu**



On the form that shows up, you can pick the export format, as well as where to put the data afterwards (a filename starting with `gs://`). You can also choose to compress the data with GZip compression.

**Figure 19.20. Exporting data from BigQuery to Cloud Storage**



If your data happens to be particularly large and won't fit in a single file, you can tell BigQuery to spread it across multiple files by using a glob expression. That is, instead of `gs://bucket/mytable.json` you can use `gs://bucket/mytable/*.json`.

**NOTE** If you aren't sure whether your table is too big, try with a single file first and you'll get an error if it's too large.

Once you click OK, just like in an import job, you'll be brought to the list of running jobs where you can see the status of the export operation.

**Figure 19.21. Status of the export job**

## Recent Jobs

Filter jobs	
<input checked="" type="checkbox"/> Extract	jjg-personal:taxi_test.trips to gs://jjg-bigquery/trips/*.json 2:40PM
<b>Job ID</b>	jjg-personal:bquijob_5f87ed5c_15be4367cee
<b>Creation Time</b>	May 7, 2017, 2:40:28 PM
<b>Start Time</b>	May 7, 2017, 2:40:56 PM
<b>End Time</b>	May 7, 2017, 2:53:32 PM
<b>Source Table</b>	<a href="#">jjg-personal:taxi_test.trips</a>
<b>Destination URI</b>	<a href="#">gs://jjg-bigquery/trips/*.json (Open in GCS)</a>
<a href="#">Cancel Job</a>	

Once this completes, you'll be able to see the files in your GCS bucket. From there, you can download them and manipulate them any way you like, maybe building a machine learning model, or maybe to copy the data over to an on-premises data

warehouse. Now that we've done all sorts of things with BigQuery, it's probably a good idea to look at how much all of this will cost you, particularly if you're going to be using BigQuery on a regular basis.

## **19.3 Understanding pricing**

Like many of the services on Google Cloud Platform, BigQuery is one of those that follows the "pay for what you use" pricing model. However it's a bit unclear exactly how much you're "using" in a system like BigQuery, so let's look more closely about the different attributes that actually cost money. With BigQuery, there are 3 different things that you are charged for:

1. Storage of your data
2. Inserting new data into BigQuery (one row at a time)
3. Querying your data

Let's look first at how much storing data costs.

### **19.3.1 Storage pricing**

Similar to other storage products, the cost of keeping data in BigQuery is measured in GB-months. In other words, you are charged not just for the amount of data, but also for how long it is stored.

To make things more complicated, BigQuery actually has two different classes of pricing, based on how long you keep the data around. Tables where you're actively adding new data are treated as standard storage, but tables that are left alone for 90-days are treated as "long-term storage" which costs less. The idea behind this is to give a cost break on data that is older and might otherwise be deleted to save some money. It's important to remember that even though the long-term storage costs less, there's no degradation in any aspect of the storage (performance, durability, or availability).

So what does all of this cost? Standard storage for BigQuery data is currently \$0.02 USD per GB-month, and long-term storage comes in at half that price (\$0.01 USD). This means that if you had two 100 GB tables, where one of them hasn't been edited in 90 days, you'd have a total bill of \$3 USD for each month you kept the data around ( $\$0.02 * 100 + \$0.01 * 100$ ), excluding the other BigQuery costs of course. Now that we've covered the raw storage costs, let's look at what it costs to get data in and out of BigQuery.

### **19.3.2 Data manipulation pricing**

The next attribute to cover is how much it costs to do things that move data into or out of BigQuery. This includes things like bulk-loading data, exporting data, streaming inserts, copying data, and other metadata operations. This does **not** include query pricing which we'll look at in the next section. The good news for this section is that almost everything is completely free! It turns out that everything except for streaming inserts are free. That means that the bulk load of 1 TB of data into BigQuery is free, as is the export job that then takes that 1 TB of data and moves it into GCS.

Unlike other storage systems like Cloud Datastore, streaming inserts actually are measured based on their size rather than the number of requests. This means that there's really no difference in two API calls to insert one row each compared to one API call that inserts both rows, as the total data inserted will cost \$0.05 per GB inserted. This pricing scheme means that you should probably avoid streaming new data into BigQuery if it's possible for you to bulk load the data all at once at the end of each day (since importing data is completely free). However, if you can't wait for results to be available in queries, streaming inserts is the way to go. This brings us to the final aspect of pricing, which also happens to be the most common one: querying.

### **19.3.3 Query pricing**

Running queries on BigQuery is arguably the most important of the service, but is measured in a way that sometimes confuses people. Unlike an instance that runs and has a maximum capacity, BigQuery's value is in the ability to "spike" and use many thousands of machines to process absurdly large amounts of data very quickly. So instead of measuring how many machines you have on hand, BigQuery measures how much data a given query processes. The total cost of this is \$5 USD per TB processed. This means that a query that scans the entirety of a 1 TB table (e.g., `SELECT * FROM table WHERE name = 'Joe'`), will cost \$5 in those few seconds it takes to scan the entire table!

There are a few things to keep in mind when looking at how much queries cost. Firstly, if there's an error in executing the query, you aren't charged anything at all. However, if you cancel a query while it's running, you still may end up being charged for the query (for example, the query may have been ready by the time you canceled it). When calculating the amount processed, the total is rounded to the nearest MB, but has a minimum of 10 MB. This means that if you run a query that only looks at 1 MB of data, you're charged for 10 MB (\$0.00005).

The final and most important aspect to query pricing is that we tend to think of "data processed" in terms of "number of rows processed", but with BigQuery that isn't the case. Since BigQuery is a column-oriented storage system the total data processed has to do with the number of rows scanned, but also the number of columns selected (or filtered). For example, imagine that you have a table with two columns, both INTEGER types. If you were to only look at one of the columns (e.g., `SELECT field1 FROM table`), it would cost about half as much as looking at both (e.g. `SELECT field1, field2 FROM table`) because you're only looking at about half of the total data.

It may confuse you then to learn that the following two queries cost exactly the same: `SELECT field1 FROM table WHERE field2 = 4` and `SELECT field1, field2 FROM table WHERE field2 = 4`. This is because the two queries both "look" at both fields. In the first, it only processes it as part of the filtering condition, but that still means it needs to be processed. If you happen to need tons and tons of querying capacity or want the ability to limit how much money can be spent on querying data from BigQuery, there is fixed-rate pricing, but it's mainly for people spending a lot of

money (e.g., \$10,000 and up per month).

## 19.4 Summary

- BigQuery acts like a SQL database that can analyze terabytes of data incredibly quickly by allowing you to "spike" and make use of thousands of machines at a moments notice.
- Even though BigQuery supports many features of OLTP databases, it does not have transactions or uniqueness constraints and should be used as an analytical data warehouse.
- While data in BigQuery is mutable, the lack of uniqueness constraints means it's not always possible to address a specific row, so this should be avoided (e.g., don't do `UPDATE ... WHERE id = 5`).
- When importing or exporting data from BigQuery, Cloud Storage typically acts as an intermediate place to keep the data.
- If you have the need for frequent updates to your data, BigQuery's streaming inserts allow you to add rows in small chunks, but this is expensive compared to importing data in bulk.
- BigQuery charges for queries based on the amount of data processed, so only select and filter on the rows that you actually need (e.g., avoid `SELECT * FROM table`).



# Cloud Dataflow: Large-scale data processing

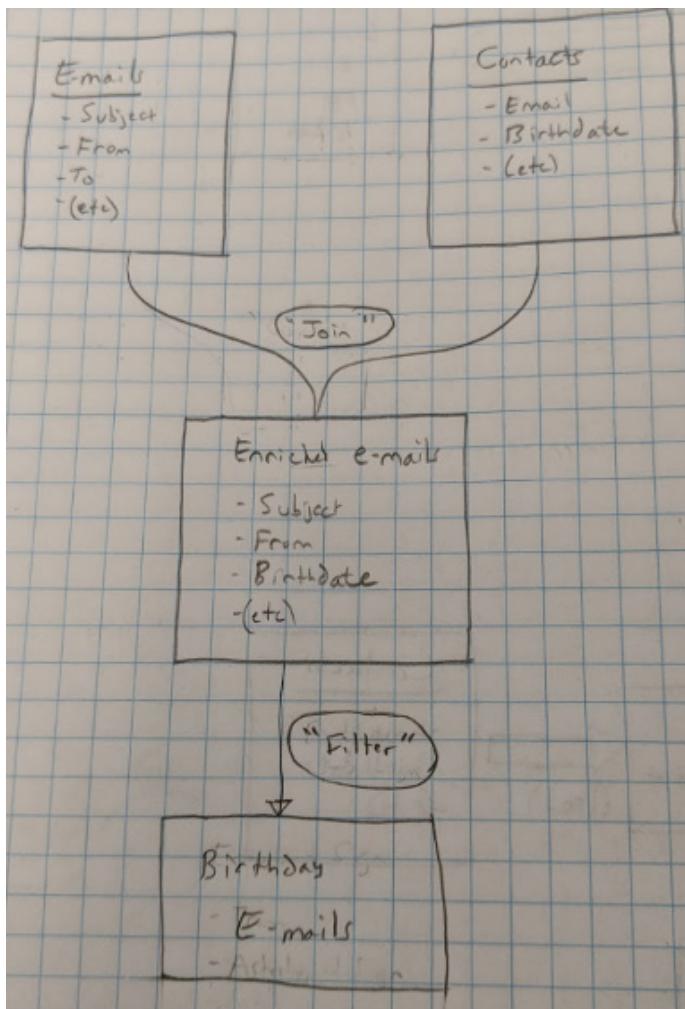
## This chapter covers:

- What do we mean by "data processing"?
- What is Apache Beam?
- What is Cloud Dataflow?
- How can we use Apache Beam and Cloud Dataflow together to process large sets of data?

## 20.1 What is data processing?

You've probably seen or heard the term "data processing" before, likely meaning something like "taking some data and transforming it somehow". Being more specific though, when we talk about data processing we tend to mean taking a whole lot of data (measured in GB at least) potentially combining it with other data, and either ending with some enriched data set of similar size or a smaller data set that summarizes some aspects of the huge pile of data. For example, imagine you had all of your e-mail history in one big pile, and all of your contact information (e-mail addresses and birthdays) in another big pile. Using this idea of data processing you might be able to "join" those two piles together based on the e-mail address. Once you've done that you could then filter that data down to find only e-mails that were sent on someone's birthday.

**Figure 20.1. Using data processing to combine and enrich e-mail history**

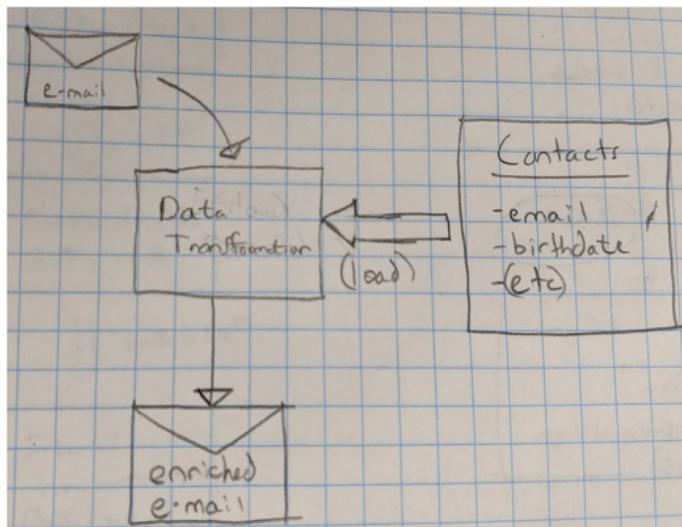


This idea of taking large chunks of data and combining it with other data (or transforming it somehow) to come out with some more meaningful set of data can be very valuable. For example, if we couldn't join the e-mail and contact data like this, we'd need to do so earlier on, which in this case exactly when we send or receive an e-mail. This would mean that we'd need to provide the birthdays of all participants in an e-mail thread whenever we send or receive an e-mail, which would be a pretty silly thing to do.

In addition to processing one chunk of data into a different chunk of data, we can also think of data processing as a way to perform "streaming transformations" over data as it's in flight. In other words, rather than treating your e-mail history as a big pile of data and enriching it based on a big pile of contact information, we might instead

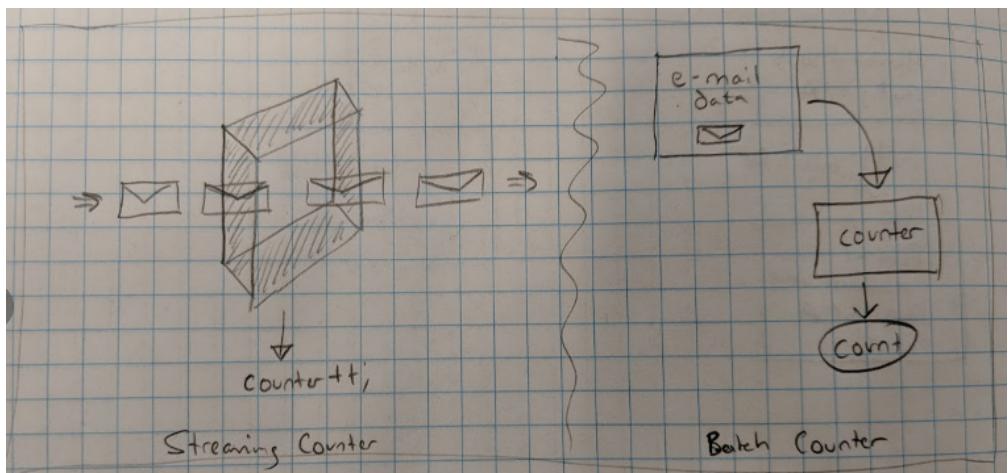
intercept e-mails as they arrive and enrich them one at a time. For example, we might load up our contact information and as each e-mail arrives we could add in the sender's birthday. By doing this, we end up treating each e-mail as one piece of a stream of data rather than looking at a big chunk and treating it as a batch of data.

**Figure 20.2. Processing data as a stream rather than as a batch**



Treating data as a batch or a stream tends to come with different benefits and drawbacks. For example, if all we wanted to do is count the number of e-mails we receive, storing and querying a big pile of data would take up lots of space and processing time whereas if we were to rely on a stream we could simply increment a counter whenever new e-mails arrive to keep count. On the flip side, if we wanted to count how many e-mails we got *last week*, this streaming counter would be pretty useless compared to our batch of e-mail data that we could filter through to find only last week's e-mails and then count the matching ones.

**Figure 20.3. Streaming counter versus batch counter**



So how we can express these ideas of data processing, for both streams and batches of data? How do we write code that represents combining e-mail history and contact information to get an "enriched e-mail" that also contains the sender's birthday? Or how do we keep count of the number of incoming e-mails? What about counts that only match certain conditions like those arriving outside working hours? There are lots of ways to express these different things, but we'll look specifically at an open-source project called Apache Beam.

## 20.2 What is Apache Beam?

The ability to transform, enrich, and summarize data can be really valuable (and fun), but it definitely won't be very easy unless we have a way of representing these actions in code. This means we need a way of saying, "get some data from somewhere", "combine this data with this data", and "add this new field on each item by running this calculation" among other things in our code. There are lots of different ways of representing pipelines for various purposes, but for handling data processing pipelines Apache Beam happens to fit the bill quite well. Beam is a framework with bindings in both Python and Java that allows you to represent a data processing pipeline with actions for inputs and outputs as well as a variety of built-in data transformations.

**NOTE**

Apache Beam is a large open-source project that merits its own book just on pipeline definitions, transformations, execution details, and more. This chapter can't possibly cover everything about Apache Beam, so the goal is to give you enough information to use Beam with Cloud Dataflow in just a few pages.

If you're excited to learn more about Apache Beam, check out [beam.apache.org](https://beam.apache.org) which has much more information.

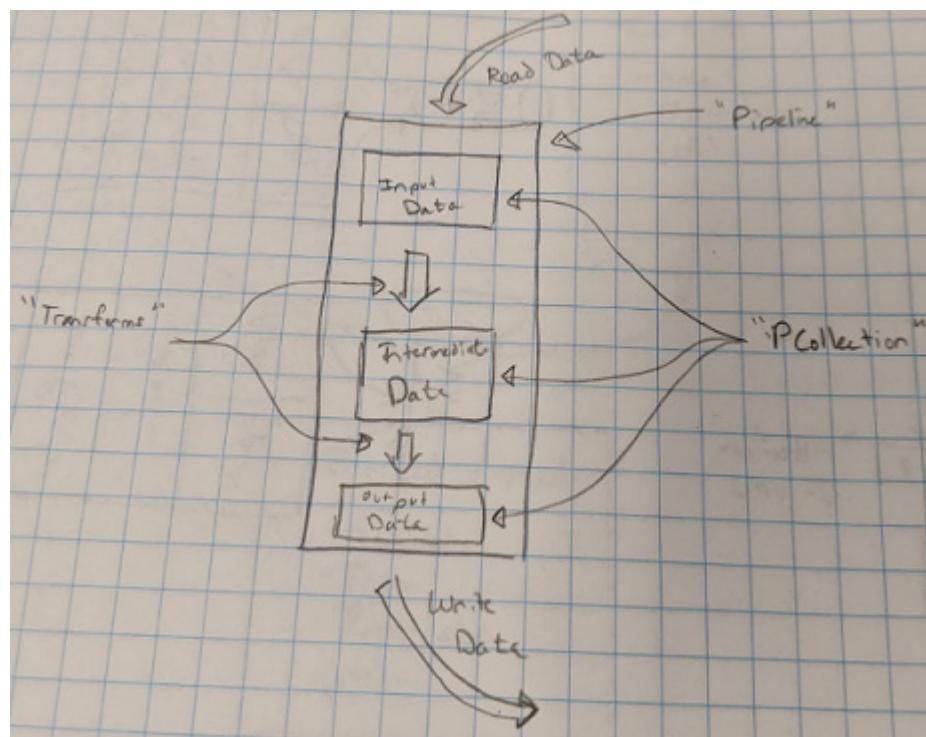
Before we get into writing a bunch of code, let's start by looking at some of the key concepts we'll need to understand in order to express our pipelines using Apache

Beam.

### 20.2.1 Concepts

Just like any other programming framework, before we get to writing code there are a few things that we'll need to understand first. In brief, we'll look at the high level container (a *pipeline*), the data that flows through the pipeline (called *PCollections*), as well as how we manipulate data along the way (using *transforms*). These different concepts are represented visually in the following diagram.

**Figure 20.4. Core concepts of Apache Beam**

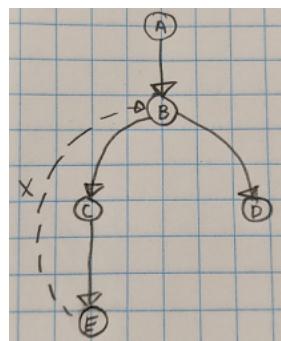


#### PIPELINES

In Apache Beam a *pipeline* refers to the high-level container of a bunch of data processing operations. In other words, pipelines encapsulate all of the input and output data as well as the transformation steps that manipulate data from the input to the desired output. This generally means that the pipeline itself is the first thing you create when you get to writing code that uses Apache Beam. In more technical terms, a pipeline is a directed acyclic graph (sometimes abbreviated to *DAG*), which means it has nodes and edges that flow in a certain direction, and don't provide a way to "repeat" or get into a loop. In the previous diagram, you can see that the chunks of data are like the nodes in a graph and the arrows are the edges. The fact that the edges are arrows pointing in a certain direction is what we mean by a *directed* graph.

Finally, notice that the pipeline (or graph) from before clearly flows in a single direction and cannot get into a loop. This is what we mean by an *acyclic* graph: there are no "cycles" for the pipeline to end up in. For example, the following diagram shows an acyclic graph using solid lines only. If we were to add the dashed line (from E back to B), the graph could have a loop and keep going forever, meaning it would no longer be "acyclic".

**Figure 20.5. Directed acyclic graph**



Pipelines themselves can have lots of different configuration options (such as where the input and output data lives) which allows them to be somewhat customizable. Additionally, Beam makes it easy to define parameter names as well as to set defaults for those parameters, which can then be read from the command-line when you actually run the pipeline, but we'll dig into that a bit later. For now the most important thing to remember is that Beam pipelines are directed acyclic graphs which means they are things that take data and move it from some start point to some finish point without getting into any loops. Now that we've gone through the high-level of a pipeline, we need to zoom in a bit and look at the how we represent chunks of data as they move through our pipeline.

## PCOLLECTIONS

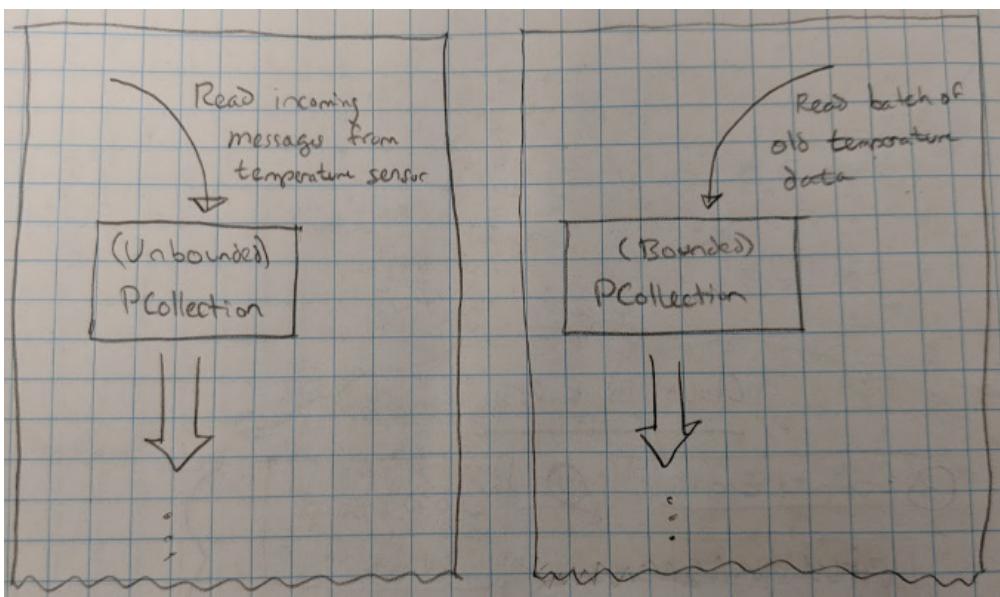
*PCollections*, so far known only as "the nodes" in our graph or "the data" in our pipeline, act as a way to represent intermediate chunks or streams of data as they flow through a pipeline. Since PCollections represent data, we can create them either by reading from some raw data points or by applying some transformation to an existing PCollection, which we'll discuss more in the next section.

Notice that we only said that a PCollection represents some data, but didn't really explain how that data is represented under the hood. It turns out that this data could be of basically any size ranging from just a few rows that we add to our pipeline code, to an enormous amount of data distributed across lots of different machines. In some cases the data could even be an infinite stream of incoming data that may never end! For example, if we had a temperature sensor that sent a new data point every second of the current temperature. This distinction brings us to an interesting property of PCollections called "boundedness".

By definition, a PCollection can be either "bounded" or "unbounded". As you might guess, if a PCollection is *bounded* we may not know the exact size, however we do know that it does have a fixed finite size (such as "10 billion items"). Put slightly differently, a bounded PCollection is one that we are sure will not go on forever.

As you'd expect, it follows that the other type of PCollection is called *unbounded*. An unbounded PCollection is one which has no predefined finite size, instead may just go on forever. The typical example of an unbounded PCollection is a stream of data that is being generated in real-time, such as the temperature sensor mentioned before. Given the fundamental difference of these two types of PCollections, we'll end up treating them a bit differently when running the pipeline, so it's important to remember this distinction.

**Figure 20.6. Bounded versus unbounded PCollections**



PCollections also have a few technical requirements that will affect how we express them in code. First, since a PCollection is always created within a pipeline, they must also stay within a single pipeline. In other words, you can't create a PCollection inside one pipeline and then reference it from another. Hopefully this won't be too much of an issue as you'll likely be defining a single pipeline at a time.

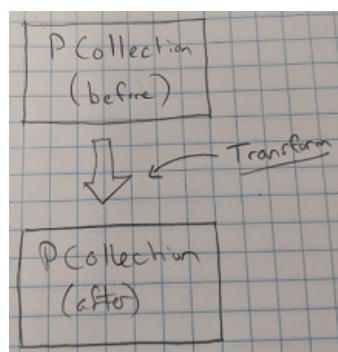
Additionally, PCollections themselves are "immutable". This means that once we create a PCollection (for example, by reading in some raw data), we can't change the PCollection's data again. Instead, we rely on a functional style of programming to manipulate our data, where new PCollections can be created by transforming existing PCollections. We'll get into this in the next section when we talk about transforms, but if you've ever written functional code that manipulates immutable objects, transforming PCollections should seem pretty natural.

Finally, PCollections are more like Python's iterators than lists. This means that we can continue asking for more data from them, but we can't necessarily jump to a specific point in the PCollection. Thinking in terms of Python code, this means that it's fine to write `for item in pcollection: ...` to iterate through the data in the PCollection, however we couldn't write `item = pcollection[25]` to grab an individual item from the PCollection. Now that we have a decent grasp on what PCollections are and some of the technical details about them, let's look at how we work with them using transforms.

### TRANSFORMS

*Transforms*, as the name implies, are the way you take chunks of input data and mutate these into chunks of output data. More specifically, transforms are the way you take PCollections and turn them into other PCollections. Put visually, these are the big arrows between PCollections inside Pipelines, where each transform has some inputs and some outputs.

**Figure 20.7. Visual representation of a transform**



These transforms could do a variety of things, and Beam comes with quite a few built-in transforms to help make it easy to manipulate data in your pipelines without writing a whole lot of boilerplate code. For example, the following are all examples of transforms built in to Beam that we might apply to a given PCollection:

- Filter out unwanted data that we're not interested in (such as filtering personal e-mails out from our e-mail data)
- Split the data into separate chunks (such as splitting e-mails into those arriving during work hours versus outside work hours)
- Group the data by a certain property (such as grouping e-mails by the sender's address)
- Join together two (or more) different chunks of data (such as combining e-mails with contact information by e-mail address)
- Enrich the data by calculating something new (such as calculating the number of people CC'd on an e-mail)

Keep in mind that this is just a short list of examples rather than a complete list of all

the transforms that Apache Beam has to offer out of the box. In fact, in addition to these and many more built-in transforms provided by Beam, you actually have the ability to write your own custom transforms and use them in your pipelines. These transforms have a few pretty interesting properties that are worth mentioning. First, while many of the transforms described above have one PCollection as input and another as output, it's entirely possible that a transform will have more than one input (or more than one output). For example, both the "join" and "split" transformations follow this pattern (the "join" transform takes two different PCollections as inputs and outputs a newly joined PCollection, and the "split" transform takes one PCollection as input and outputs two separate PCollections as outputs).

Next, since PCollections are immutable, transforms that we apply to PCollections don't "consume" the data from an existing PCollection. Put slightly differently, when you apply a transformation to an existing PCollection, a new one is created without destroying the existing one that acted as the data source. This means that we can use the same PCollection as an input to multiple transforms in the same pipeline, which can come in handy when we need the same data for two similar but separate purposes. This flies in the face of how iterators work in a variety of programming languages (including Python whose iterators are in fact consumed by iterating over them), which makes it a common area of confusion in Apache Beam.

Finally, even though we can think conceptually of the new PCollection as "containing the transformed data", the way this works under the hood might vary depending on how the pipeline itself is executed. This leads us to the next topic of interest: how do we actually execute a pipeline?

### **PIPELINE RUNNERS**

As the name implies, a pipeline runner simply *runs* a given *pipeline*. While the high-level concept of the system that does the actual work is a bit boring, the lower-level details are actually pretty interesting as there can be a great deal of variety in how transforms are applied to PCollections.

Since Apache Beam allows us to define pipelines using the high level concepts we've learned about so far, we're able to keep the *definition* of a pipeline separate from the *execution* of that pipeline. In other words, even though the definition of a pipeline is specific to Beam, the underlying system that actually organizes and executes the work is abstracted away from the definition. This means that you could take the same pipeline defined using Beam and run it across a variety of different execution engines which each may have their own strategies, optimizations, and features.

If you've ever written code that has to talk to a SQL database, you can think of this feature of Beam sort of like the same feature of an ORM (Object-relational mapping) such as SQL Alchemy in Python or Hibernate in Java. ORMs allow you define your resources and interact with them as objects in the same language (e.g., Python) and under the hood the ORM turns those interactions into actual SQL queries for a variety of different databases (such as MySQL and PostgreSQL). In this same way Apache Beam allows you to define a pipeline without worrying about where it will run, and

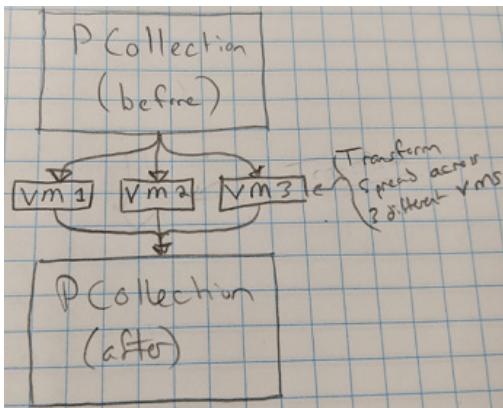
then later execute it using a variety of different "pipeline runners".

There are quite a few pipeline runners available for Apache Beam, with simplest option being the "DirectRunner". This runner is primarily a testing tool that will execute pipelines on your local machine, but it's interesting in that it doesn't just take the simplest and most efficient path toward executing your pipeline. Instead, it runs lots of additional checks to ensure that pipelines don't rely on unusual semantics that will break down in other more complex runners.

Typically, unlike the Direct Runner, pipeline runners are made up of lots of machines all running as one big "cluster". If you read through [chapter 10](#), you can think of a pipeline running cluster sort of like a Kubernetes cluster in that they both execute some given input using some set number of machines. This distributed execution allows the work to be spread out across a potentially large number different machines, which in turn means that it gives you the option to make any job complete more quickly simply by adding more machines to the process.

To do this, these runners will chop the work up into lots of pieces (both at the start of the pipeline and anywhere in between) in order to make the most efficient use of all the machines available. This means that while we would still represent transforms as an arrow between two PCollection, under the hood the work might actually be done by splitting the word list up into lots of little pieces and spreading the actual work of the transform across several different machines. Put visually, this might look something like the following diagram.

**Figure 20.8. Transform applied using multiple VMs**



Since this "chopping" may mean that data will have to be moved around, pipeline runners will do their best to execute computations on as few machines as possible. This is mostly because data moving over a network is far slower than accessing it from local memory. In other words, minimizing data sent over the network means the processing jobs can be completed faster.

Unfortunately, more often than not moving data over the network is simply unavoidable. On the other hand, the other options (such as using a single large machine

to do the work) will be even slower despite the time needed to move data from one machine to another over the network. In short, despite the added time needed to shift data from one place to another, the pipeline runner will likely still land on the division of labor that results in the shortest total time to execute the pipeline. Now that we've gone through the high-level concepts let's get a bit more specific by writing some code, and then we'll look at actually using one of these pipeline runners.

### **20.2.2 Putting it all together**

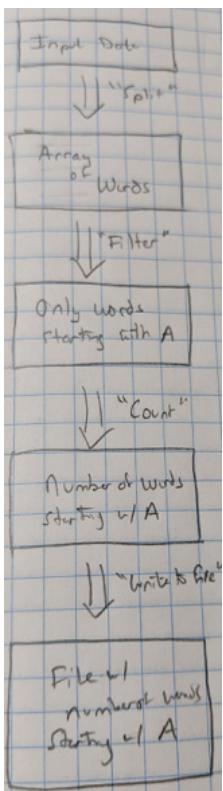
Using these three basic concepts (Pipelines, PCollections, and Transforms) we can build some pretty cool things. However so far we've stuck to high-level descriptions and stayed away from actual code. Let's dig into the code itself by looking at a short example.

**WARNING**

Since Apache Beam doesn't have bindings for Node.js, the rest of this chapter will use Python to define and interact with Beam Pipelines. We'll stay away from the tricky parts of Python so you should be able to follow along, but if you want to get into writing your own pipelines using Beam you'll need to brush up on Python or Java.

Imagine that we had a digital copy of some text and we wanted to count the number of words starting with the letter 'a'. As with many problems like this, we could simply write a script that parses the text and iterates through all the words, but what if the text is a few hundred Gigabytes in size? What if we had thousands of those texts to analyze? It would probably take even the fastest computers quite a while to do this work. We could instead use Apache Beam to define a pipeline that would do this for us, which would allow us to spread that work across lots of different computers and still get the right output. So what would this pipeline look like? Let's start by looking graphically at the pipeline and then we'll start writing code.

**Figure 20.9. Pipeline to count words starting with 'a'**



Notice that we use multiple steps to take some raw input data and transform it into the output we're interested in. In this case we read the raw data in, apply a "Split" transform to turn it into a chunk of words, then a "Filter" transform to remove all words we aren't interested in, then a "Count" transform to reduce the set to just the total number of words, and then finally we write the output to a file. Thinking of a pipeline this way makes it easy to turn it into code, which might look something like the following snippet.

**NOTE**

The following code is accurate but leaves a few variables undefined. This means it's not expected to run if you copy and paste it exactly as is, but you'll need to fill in some blanks (e.g., `input_file`). Don't worry though. We'll have complete examples to work through later in the chapter.

**Listing 20.1. An example Apache Beam pipeline**

```

import re
import apache_beam as beam

with beam.Pipeline() as pipeline: ①
    (pipeline
  
```

```

| beam.io.ReadFromText(input_file) ②
| 'Split' >> (beam.FlatMap(lambda x: re.findall(r'[A-Za-z\']+', x))
|   .with_output_types(unicode))
| 'Filter' >> beam.Filter(lambda x: x.lower().startswith('a'))
| 'Count' >> beam.combiners.Count.Globally()
| beam.io.WriteToText(output_file) ⑥
)

```

- ① We start by creating a new pipeline object using Beam.
- ② Load some data from a text input file using `beam.io.ReadFromText`.
- ③ Take the input data and split it into a bunch of words.
- ④ Filter out any words that don't start with 'a'.
- ⑤ Count all of those words.
- ⑥ Write that number to an output text file.

As you can see, in Apache Beam's Python bindings we rely on the "pipe" operator as we would in a Unix-based terminal to represent data flowing through the different transformations. This allows us to express our *intent* of how data should flow without getting into the lower-level details about how we might divide this problem into smaller pieces which, as we learned before, is the responsibility of the pipeline runner we use to execute the code itself.

At this point, we've learned all the important concepts, looked at an example pipeline visually, and looked at some corresponding Python code for the pipeline itself. But what about the pipeline runners? For Apache Beam there are quite a few such as Apache Flink and Apache Apex, but there's also a fully-managed pipeline runner that happens to be the subject of this chapter: Google Cloud Dataflow.

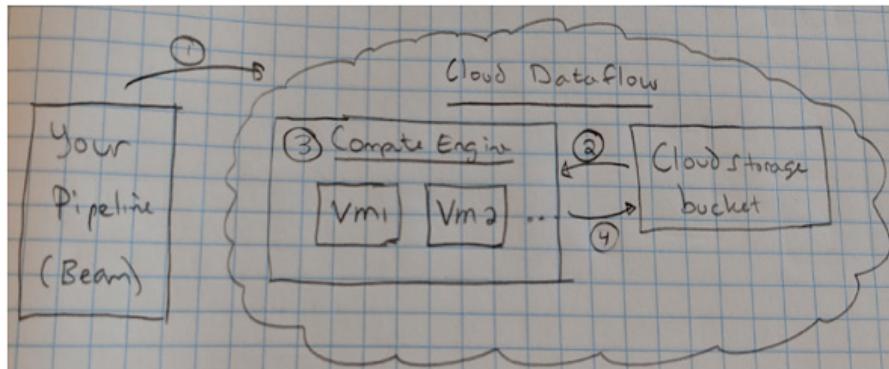
## 20.3 What is Cloud Dataflow?

As we learned previously, we can use Apache Beam to define pipelines that are "portable" across lots of different pipeline runners. This portability means that we have lots of different options to choose from when it comes time to run our Beam pipelines. Google Cloud Dataflow is one of the many options available, but it's special in that it's a fully-managed pipeline runner. This means that, unlike other pipeline runners, using Cloud Dataflow requires no initial set-up of the underlying resources. In other words, most other systems require you to provision and manage the machines first, then install the software itself, and only then can you submit pipelines to execute. With Cloud Dataflow, that is all taken care of for you meaning you can submit your pipeline to execute without any other prior configuration.

You may see a bit of similarity here with Kubernetes and Kubernetes Engine (see [chapter 10](#)) where running your own Kubernetes cluster requires you to manage the machines that run Kubernetes itself, compared with Kubernetes Engine where those machines are provisioned and managed for you. Since Cloud Dataflow is part of Google Cloud Platform, it has the ability to stitch together lots of other services that we've learned about already. For example, we might execute a pipeline using Cloud Dataflow (1), which could read data using a Cloud Storage bucket (2), use Compute

Engine instances to process and transform that data (3), and finally write the output back to another Cloud Storage bucket (4).

**Figure 20.10. Overview of infrastructure for Cloud Dataflow**



Unlike some of the other runners all of this coordination across the various Google Cloud Platform resources is handled for you. Instead the specifics (such as where input data lives in Cloud Storage) are passed in as pipeline parameters and the work of running your pipeline is managed entirely by Cloud Dataflow. So how do we actually use Cloud Dataflow? Let's look at our example from before where we count all the words starting with the letter 'a' and see how we can use Cloud Dataflow and Apache Beam to run our pipelines.

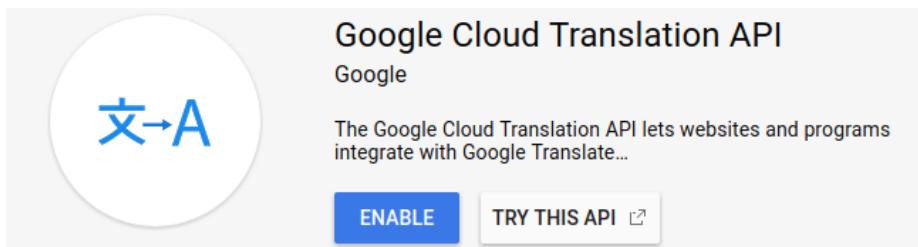
## 20.4 Interacting with Cloud Dataflow

Before we can actually start using Cloud Dataflow, we'll need to do a bit of initial set-up and configuration. Let's look at that first and then we'll jump into creating our pipeline.

### 20.4.1 Setting up

The first thing we should do is enable the Cloud Dataflow API. To do this, navigate to the Cloud Console and in the main search box at the top type "Dataflow API". This query should come up with a single result, and clicking on that should bring you to a page with a big "Enable" button. Click that and you should be good to go.

**Figure 20.11. Enable the Cloud Dataflow API**



After the API is enabled, we'll need to make sure we have Apache Beam installed locally. To do this, we can use pip which manages packages for Python. Although the package itself is called apache-beam, we want to make sure we get the Google Cloud Platform extras for the package. These extra additions will allow our code to interact with GCP services without any additional code. For example, one of the GCP extras for Apache Beam allows us to refer to files on Google Cloud Storage by a URL starting with gs://. Without this, we'd have to manually pull down the Python clients for each of the services we wanted to use. To get these extras, we'll use the [] syntax shown below which is a standard thing for Python.

#### **Listing 20.2. Install Apache Beam with the GCP extras**

```
$ pip install apache-beam[gcp]
```

The next thing we'll need to do is make sure that any code we run uses the right credentials and has access to our project on Cloud Dataflow. To do this we can use the gcloud command-line tool (which we installed previously) to fetch credentials that our code will use automatically.

#### **Listing 20.3. Authenticate using application default credentials**

```
$ gcloud auth application-default login
```

When you run that command, you'll see a link to click on which you can then authenticate with your Google account in your web browser. After that, the command will download the credentials in the background to a place where your code will discover them automatically. Now that all our APIs are enabled, we have all the software packages we need, and fetched the right credentials, we have one more thing to do: figure out exactly where we can put our input, output, and (possibly) any temporary data while we're running our pipeline. After all, the pipeline we defined previous reads input data from somewhere and then writes the output to somewhere. But in addition to those two places there may be extra data that we need to store, sort of like a spare piece of paper during a math exam, and we'll need somewhere for that to live.

Since we're already using Google Cloud Platform for all of this, it makes sense that we'd use one of the storage options such as Google Cloud Storage. To do this, we'll need to create a Cloud Storage bucket, shown below.

#### **Listing 20.4. Create a Cloud Storage bucket to store input, output, and intermediate data**

```
$ gsutil mb -l us gs://your-bucket-id-here
Creating gs://your-bucket-id-here/...
```

This command specifically creates a bucket that uses multi-regional replication and is located in the United States. If you're at all confused by this, take a look at [chapter 8](#).

And with that, we have all we need and can finally get to work taking the snippet of code from before and turning it into an actual runnable pipeline!

## 20.4.2 Creating a pipeline

As you may remember, the goal of our pipeline was to take any input text document, and figure out how many words in the document start with the letter 'a'. We first turned this into a visual representation and then transcribed that into more specific actual code that relied on Apache Beam to define the pipeline. You may also recall, however, that the code snippet left some of the details out such as where the input files came from and how to execute the actual pipeline. To make the pipeline actually run, we'll need to define these as well as add a bit of boilerplate to our pipeline code.

The following updated code snippet adds some helper code, defines a few of the variables that were missing from before, and demonstrates how to parse command-line arguments and pass them into our pipeline as options.

### Listing 20.5. Our complete pipeline code

```
import argparse
import re

import apache_beam as beam
from apache_beam.options import pipeline_options

PROJECT_ID = '<your-project-id-here>'          ①
BUCKET = 'dataflow-bucket'

def get_pipeline_options(pipeline_args):    ②
    pipeline_args = ['--%s=%s' % (k, v) for (k, v) in {
        'project': PROJECT_ID,
        'job_name': 'dataflow-count',
        'staging_location': 'gs://%s/dataflow-staging' % BUCKET,
        'temp_location': 'gs://%s/dataflow-temp' % BUCKET,
    }.items()] + pipeline_args
    options = pipeline_options.PipelineOptions(pipeline_args)
    options.view_as(pipeline_options.SetupOptions).save_main_session = True
    return options

def main(argv=None):                          ③
    # Process command-line arguments, combining with defaults.
    parser = argparse.ArgumentParser()          ④
    parser.add_argument('--input', dest='input')
    parser.add_argument('--output', dest='output',
                        default='gs://%s/dataflow-count' % BUCKET)
    script_args, pipeline_args = parser.parse_known_args(argv)
    pipeline_options = get_pipeline_options(pipeline_args)

    # Construct and run the pipeline.
    with beam.Pipeline(options=pipeline_options) as pipeline: ⑤
        (pipeline
```

```

| beam.io.ReadFromText(script_args.input)           ⑥
| 'Split' >> (beam.FlatMap(lambda x: re.findall(r'[A-Za-z\']+', x))
|   .with_output_types(unicode))
| 'Filter' >> beam.Filter(lambda x: x.lower().startswith('a'))
| 'Count' >> beam.combiners.Count.Globally()
| beam.io.WriteToText(script_args.output)
)

if __name__ == '__main__':
    main()                                     ⑦

```

- ➊ First we need to define a bunch parameters like our project ID, and the bucket where we'll store data. This is the bucket we created in the previous section.
- ➋ This helper function takes a set of arguments, combines them with some reasonable defaults, and converts those into Apache Beam-specific pipeline options, which we'll use later.
- ➌ In the `main` method we actually do the real work.
- ➍ First, we turn take any arguments passed along the command line. We'll use some of these directly in our code (e.g., `input`) and the rest we'll treat as options to be used by the pipeline itself as options.
- ➎ At this point the code should look very similar to the previous snippet. One difference is that we pass in some specific options when creating the pipeline object.
- ➏ Another difference from the original snippet is that we define the location of our input data based on the command-line arguments.
- ➐ To start everything off, we'll call the `main` function that we've just defined.

At this point we have a fully-defined Apache Beam pipeline that can take some input text and will output the total number of words that start with the letter 'a'. So how about we take it for a test drive?

### 20.4.3 Executing a pipeline locally

As we learned before, Apache Beam has a few built-in pipeline runners, one of which is the "DirectRunner". This runner happens to be great for testing not only because it runs the pipeline on your local machine, but also because it checks a variety of things to make sure that your pipeline will run well in a distributed environment (like Google Cloud Dataflow). Since we have that, let's execute our pipeline locally to test it out, first with some sample data and then something a bit larger. To start, let's create a simple file with a few words in it that we can easily count by hand to verify our code is doing the right thing.

#### Listing 20.6. Running the pipeline locally

```

$ echo "You can avoid reality, but you cannot avoid the consequences of avoiding
reality." > input.txt
$ python counter.py --input="input.txt" \
    --output="output-" \ ①
    --runner=DirectRunner ②

```

- ➊ Here the output is a *prefix* to be used where we put the file.

- ② As we noted above, we'll use the Direct Runner to execute our pipeline, which runs this entire job locally.

As you can see in the sentence above, there are exactly 3 words there that start with the letter 'a'. Let's check whether our pipeline came up with the same answer. We can see that by looking in the same directory for output inside a file starting with output-.

#### **Listing 20.7. Viewing the output of the pipeline**

```
$ ls output-*
output--00000-of-00001
$ cat output--00000-of-00001
3 ①
```

- ① It looks like we found the right number of words!

As we can see, our pipeline clearly did the trick. But what about with a larger amount of data? Let's use a little Python trick to take that same sentence, repeat it a whole bunch of times, and then test our pipeline again. As before, since we're repeating the input a set number of times, we know the answer is 3-times that, which means it'll be easy to check whether our pipeline is still working.

#### **Listing 20.8. Create much more data to process**

```
$ python -c "print (raw_input() + '\n') * 10**5" < input.txt > input-10e5.txt ①
$ wc -l input-10e5.txt
100001 input-10e5.txt ②
$ du -h input-10e5.txt
7.9M    input-10e5.txt ③
```

- ① Here we take out input sentence from before, repeat it 100,000 times, and save it back to a file called input-10e5.txt.  
 ② Using the command-line tool to count the number of lines in the file, we can see that this file has 100 thousand lines (plus a trailing newline).  
 ③ As we can see, the file is about 8 MB in size.

Now that we have a bit of a larger file (which we know has exactly 300 thousand words starting with A), we can run our pipeline and test whether it actually works.

#### **Listing 20.9. Execute the pipeline again, this time with more data**

```
$ python counter.py --input=input-10e5.txt --output=output-10e5-
  --runner=DirectRunner
$ cat output-10e5-*
300000 ①
```

- ① As we expected, there are exactly 300,000 words starting with the letter 'a'.

If you want, feel free to try even larger files by increasing the  $10^{**5}$  to something

larger like  $10^{**6}$  (1 million) or  $10^{**7}$  (10 million) copies of our sentence. Keep in mind that if you do that you'll probably be waiting around for a little while for the pipeline to finish as the files themselves will be around 80 MB and 800 MB respectively, which is a decent amount of data to process on a single machine. So how do we take this a step further? In this example, all we did was take a local file, run it through our pipeline, and save the output back to the local file system. Let's look at what happens when we move this out of our local world and into the world of Cloud Dataflow.

#### 20.4.4 Executing a pipeline using Cloud Dataflow

Luckily, since we've done all our setup already, running this pipeline on Cloud Dataflow is as easy as changing the runner and updating the input and output files. To demonstrate this, let's upload our files to the Cloud Storage bucket we created previously and then use the Cloud Dataflow runner to execute the pipeline.

##### **Listing 20.10. Upload our input files to Cloud Storage and run our pipeline on Cloud Dataflow**

```
$ gsutil cp input-10e5.txt gs://dataflow-bucket/input-10e5.txt ①
$ python counter.py \
    --input=gs://dataflow-bucket/input-10e5.txt \ ②
    --output=gs://dataflow-bucket/output-10e5- \ ③
    --runner=DataflowRunner ④
```

- ① First we can use the `gsutil` command-line tool to upload our input data to our Cloud Storage bucket. (The `-m` flag tells GCS to use multiple concurrent connections to upload the file.)
- ② Next, we instruct our pipeline to use the newly uploaded file stored in our Cloud Storage bucket as the input data.
- ③ We also want the output result to live in the same Cloud Storage bucket, but we'll use a special prefix to keep track of the variety of files that will be created during the pipeline execution.
- ④ Finally, instead of using the `DirectRunner` like before, we'll use Cloud Dataflow by specifying to use the `DataflowRunner`.

Once you hit enter, under the hood Cloud Dataflow will accept the job and start getting to work on turning on resources to handle the computation. As a result, you should see some logging output from Cloud Dataflow that shows the job has been submitted.

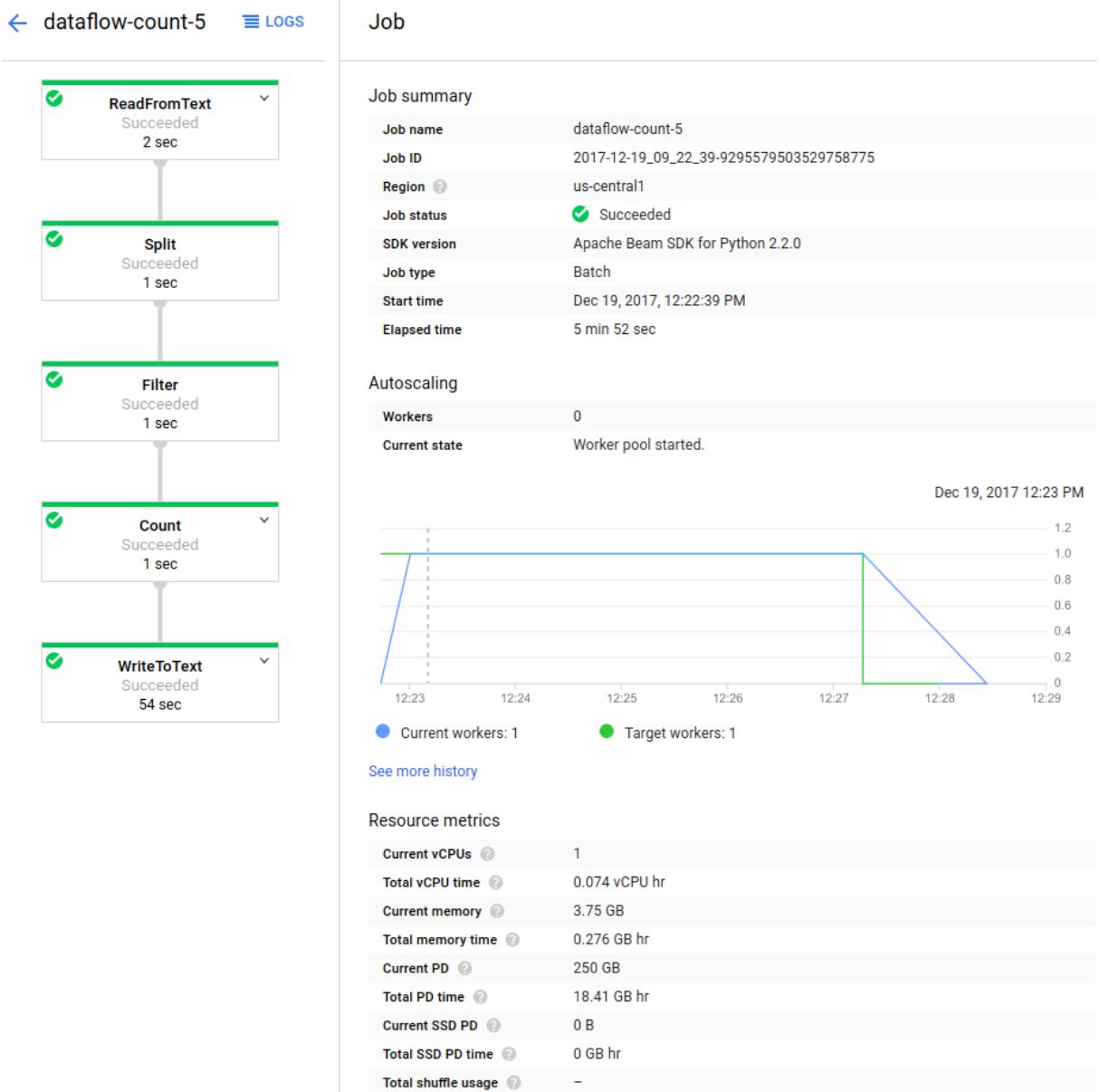
**TIP**

If you run into an error about not being able to figure out what `gs://` means (e.g., `ValueError: Unable to get the Filesystem for path gs://...)`, check that you installed the GCP extras for Apache Beam, usually by running `pip install apache-beam[gcp]`.

You might also notice that the process exits normally once the job is submitted, so how can we keep an eye on the progress of the job? And how will we know when it's done? It turns out that if you navigate to the Cloud Dataflow UI inside the Cloud Console, you'll see your newly created job in the list (the job name was specified in the pipeline

code from before) and clicking on it will show you a pretty cool overview of the process of your job.

**Figure 20.12. Overview of the pipeline job on Cloud Dataflow**



First, on the left side of the screen you'll see a graph of our pipeline which looks quite similar to the drawing that we put together before we even started. This is a good sign; it means that Dataflow has the same understanding of our pipeline as we had intended

from the start. In this case, you'll notice that most of the work completes pretty quickly, almost too quickly. That is, we never really get to see the work in progress since by the time the diagram is updated with the latest status, the work is mostly done. As a result, we really only see how each stage moves from "Running" to "Succeeded" and the entire job is over in just a few minutes.

**NOTE**

You may be wondering why the job took "a few minutes" (about 5 in this case) while each stage of the processing took only a few seconds. This is primarily because of the "set-up time" where VMs need to be turned on, disks provisioned, and software upgraded and installed, and then all the resources turned off afterwards.

As a result, even though we see that all stages take about 1 minute when summed up, the total run time (from job submission to completion) adds a bit of time before and after.

On the right hand side of the screen you can see the job details (such as the region, start time, and elapsed time) as well as some extra details about the resources involved in executing the pipeline. In this case, the work scaled up to a single worker and then back down to zero when the work was over. Just below the job details, we can see the details about the computing and storage resources that were consumed during the lifetime of the job. In this case, there was about 230 MB-hours worth of memory, and less than 0.06 vCPU-hours worth of compute time.

This is neat, but a 5-minute long job that consumes only a few minutes worth of computing time isn't all that interesting. What happens if we increase the total number of lines to 10 million ( $10^{**}7$ )? Let's try that and look at what happens with our job.

**Listing 20.11. Processing our job using even more data**

```
$ python -c "print (raw_input() + '\n') * 10**7" < input.txt > input-10e7.txt ①
$ gsutil -m cp input-10e7.txt gs://dataflow-bucket/input-10e7.txt ②
$ python counter.py \ ③
  --input=gs://dataflow-bucket/input-10e7.txt \
  --output=gs://dataflow-bucket/output-10e7- \
  --runner=DataflowRunner
```

- ① First we need to generate the 10 million lines of text.
- ② Next we'll upload it to our Cloud Storage bucket like before.
- ③ Finally, we re-run the same job but use the 10 million lines of text as the input data.

In this case, when we look again at the overview of our new job in the Cloud Console, there's enough data involved that we can see what it looks like while it's in flight. As the work progresses you'll see each stage of the pipeline show some details about how many elements it's processing, shown here.

**Figure 20.13. Overview of a larger pipeline job in progress**

← dataflow-count-big-3 LOGS
Job

```

graph TD
    A[ReadFromText  
Running  
28 sec] --> B[Split  
22,694 elements/s  
26 sec]
    B --> C[Filter  
295,028 elements/s  
24 sec]
    C --> D[Count  
Part running  
6 sec]
    D --> E[WriteToText  
Part running  
0 sec]
  
```

### Job summary

Job name	dataflow-count-big-3
Job ID	2017-12-19_07_36_27-7492275618206245908
Region	us-central1
Job status	<span>Running</span>
<a href="#">Stop job</a>	
SDK version	Apache Beam SDK for Python 2.2.0
Job type	Batch
Start time	Dec 19, 2017, 10:36:28 AM
Elapsed time	4 min 14 sec

### Autoscaling

Workers	1
Current state	Worker pool started.

Dec 19, 2017 10:36 AM

● Current workers: ● Target workers:

[See more history](#)

### Resource metrics

Current vCPUs	1
Total vCPU time	0.061 vCPU hr
Current memory	3.75 GB
Total memory time	0.229 GB hr
Current PD	250 GB
Total PD time	15.278 GB hr
Current SSD PD	0 B
Total SSD PD time	0 GB hr
Total shuffle usage	—

### Custom counters

Filter	
Split/words	23,042,214

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/google-cloud-platform-in-action>

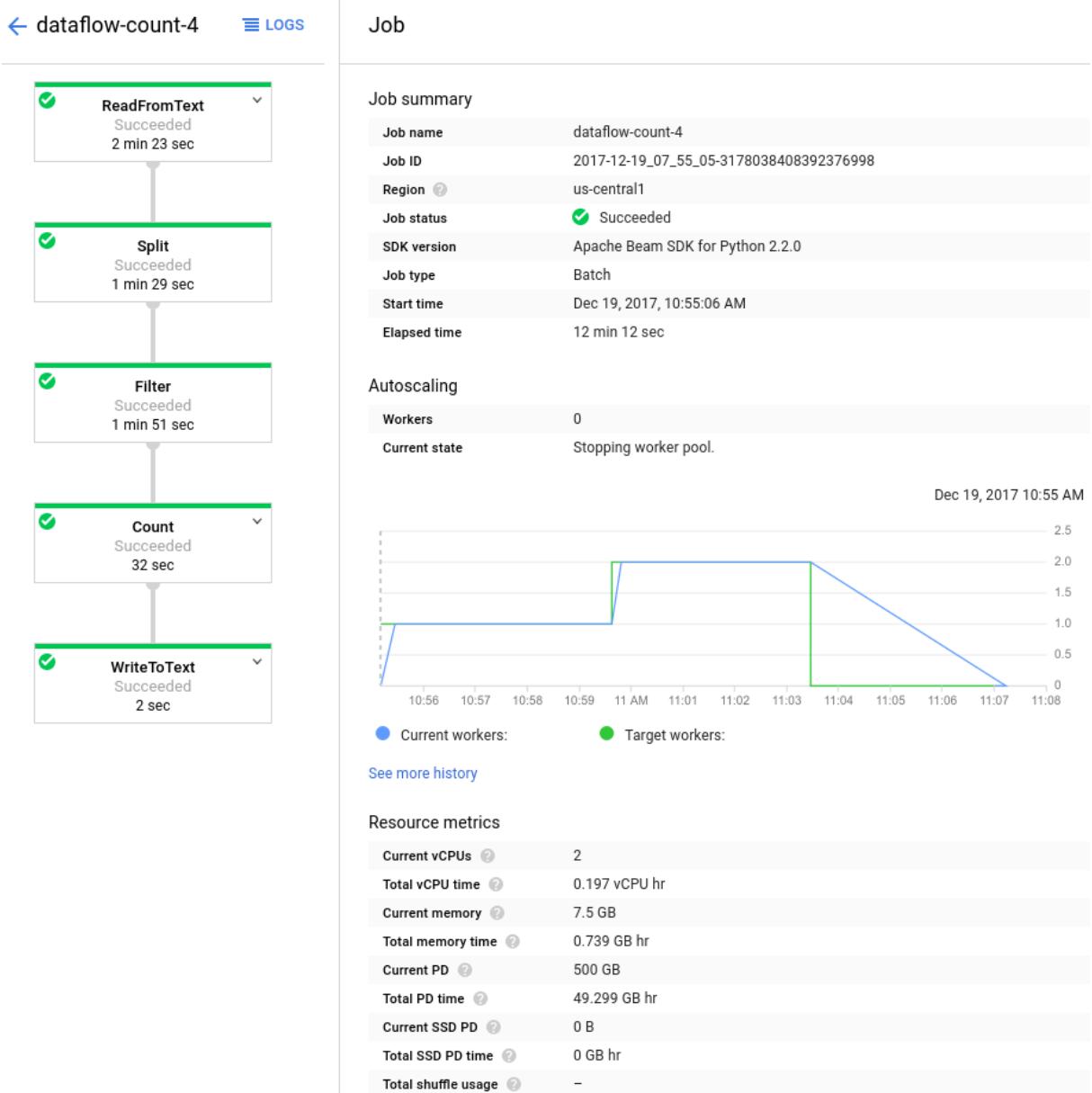
Licensed to Asif Qamar <[asif@asifqamar.com](mailto:asif@asifqamar.com)>

Perhaps the coolest part about this is that we can see how data actually flows through the pipeline with each stage being active simultaneously. In other words, we can actually see that each step executes concurrently rather than each step completing entirely before moving onto the next step. This works because each step is working on chunks of data at a time rather than the entire data set. In the first step in our graph ("ReadFromText") the data is broken up using new-line tokens to separate them, which allows a chunk of lines to be passed along to the "Split" step. From there onwards, chunks are continually moved along like an assembly line with each stage computing something and passing results forward. Finally, the last step aggregates the results (in this case, counting all of final items passed through) and saves the final output back to our Cloud Storage bucket.

Another very interesting thing is that we can see how many elements are being processed at each stage on a per-second basis. For example, in the snapshot above we can see that the "Split" stage (which you'll recall is where we find all matching words in the chunk provided) we're processing about 22,000 lines per second and outputting them as lists of words.

In the next stage we're taking in lists of words as elements and filtering out anything that doesn't start with 'a'. If you look closely, you'll notice that this stage is processing about 13-times that amount. Why is that? It turns out that each line of our input has 13 words in it, so that stage is getting the output from ~22 thousand lines per second split into 13 words per line, which comes to about 295 thousand words per second.

**Figure 20.14. Overview of a successful job**



Once the job completes, the total count is written to our Cloud Storage bucket (as we see in the "WriteToText" stage). We can verify this by checking the output files on Cloud Storage to see what the final tally is, shown below. Since we put in 10-million lines of text, each one with 3 words starting with 'a', there total should come to 30,000,000.

**Listing 20.12. Checking the output of our pipeline job**

```
$ gsutil cat gs://dataflow-bucket/output-10e7-*  
30000000 ①
```

- ① The final output is stored in Cloud Storage, showing the correct result of 30 million.

Additionally, after the job is finished we have the gift of hindsight where we can look at the amount of computing resources consumed by our job. As you might expect, more data means that we might want to use more computing power to process that data, and Cloud Dataflow figured this out as well. We can see in the graph on the right-hand side that it turns on a second VM to process data after a few minutes of starting. This is pretty great considering we didn't change any code! Instead of having to think about the number of machines needed, Cloud Dataflow just figured it out for us, scaled up when needed, and then scaled down when the work was complete. Now that we've seen how to run our pipeline, all that's left is to look at how much all of this will cost!

## 20.5 Understanding pricing

Like many compute-based products (such as Kubernetes Engine or Cloud ML Engine), Cloud Dataflow breaks down the cost of resources by a combination of computation (CPU per hour), memory (GB per hour), and disk storage (GB per hour). As you'd expect, these costs vary from location to location with US-based locations coming in cheapest (\$0.056 per CPU per hour in Iowa) compared to some other locations (\$0.0756 per CPU per hour in Sydney). Prices for a few select locations is shown in the table below.

**Table 20.1. Prices based on location**

Resource	Iowa	Sydney	London	Taiwan
vCPU	\$0.056	\$0.0756	\$0.0672	\$0.059
GB Memory	\$0.003557	\$0.004802	\$0.004268	\$0.004172
GB Standard disk	\$0.000054	\$0.000073	\$0.000065	\$0.000054
GB SSD	\$0.000298	\$0.004023	\$0.000358	\$0.000298

Unfortunately, even with these handy price numbers, predicting the total cost ahead of time can be pretty tricky. Each pipeline is different (after all, we're not trying to always count words starting with 'a') and usually the input data varies quite a bit. This means that the number of VMs used in a particular job tends to vary, and it tends to follow that the amount of disk space and memory used in total will vary as well. Luckily there are a couple of things we can do. First, if your workload is particular cost sensitive you can certainly set a specific number of workers to use (or a maximum number) which will limit the total cost per hour of your job. However, this may mean your job could take quite a long time, and there's no way to force a job to complete in a set amount of time.

Next, if you know how a particular pipeline scales over time you could run a job using

a small input to get an idea of cost and then extrapolate to get a better idea of how much larger inputs might cost. For example, the job where we count words starting with the letter 'a' is likely to scale up linearly where more words of input text take more time to count. Given that, we can assume that a run over 10x the amount of data will cost roughly 10x as much. To make this more concrete, in our previous pipeline job that counted the words across 10,000,000 lines of text we ended up consuming ~0.2 vCPU hours, ~0.75 GB-hours of memory, and ~50 GB-hours of standard disk space. Assuming this job was run in Iowa, this would bring our total to  $\sim \$0.0165$  ( $0.2 * 0.056 + 0.75 * 0.003557 + 50 * 0.000054$ ) or just under 2 cents. As a result, it's not crazy to assume that if we did 100,000,000 lines of text that had a similar distribution of words, we'd likely see that cost about \$0.16 for the workload.

## 20.6 Summary

- When we talk about "data processing" we mean the idea of taking some sets of data and transforming that data into something more useful for a particular purpose.
- Apache Beam is one of the open-source frameworks we can use to represent data transformations.
- Apache Beam has lots of different "runners", one of which is Cloud Dataflow.
- Cloud Dataflow executes Apache Beam pipelines in a managed environment, using Google Cloud Platform resources under the hood.

# 21

## *Cloud Pub/Sub: Managed event publishing*

**This chapter covers:**

- An overview of distributed messaging systems
- When and how to use Cloud Pub/Sub in your application
- How Cloud Pub/Sub pricing is calculated
- Two examples using common messaging patterns

### 21.1 What is messaging?

If you've ever sent an SMS or Facebook message, the concept of messaging should feel pretty familiar and simple. That said, in your day-to-day use of messaging you have a few requirements that we sometimes take for granted. For example, we expect that messages are:

1. sent from one specific person (you)
2. sent to exactly one specific person (your friend)
3. sent and received *exactly* once (no more, no less)

It turns out that getting meeting these requirements is not as easy as it looks. Further, messages might be broadcast to a group, which has slightly different requirements (e.g., messages should be received exactly once by each member of the group). And this communication might be synchronous (like calling someone on the phone) or asynchronous (like leaving a voice mail), each with its own requirements. In other words: messaging might seem simple but it's actually pretty tricky.

To handle this trickiness, there are a lot of open-source messaging platforms (like Apache Kafka and ZeroMQ) and a variety of standards (like AMQP), each with its

own benefits and drawbacks, but all of these tend to require that you turn on some servers and install and manage some software to route all these messages everywhere. And as the number of messages you want to send grows, you'll need to turn on more machines and possibly reconfigure the system to make use of the new computing power. All of this headache is where Cloud Pub/Sub comes in.

## 21.2 What is Cloud Pub/Sub?

Cloud Pub/Sub is a fully-managed messaging system (like Apache Kafka) built on top Google's internal infrastructure, which in turn was created because Google had the same messaging needs as many other companies. This means that the infrastructure used by Google Cloud Pub/Sub is the same as the lower-level infrastructure used by other services internal to Google such as YouTube or AdWords.

Luckily, it also turns out that Google Cloud Pub/Sub uses concepts that seem to be common across many of those open-source messaging services mentioned above. Because the concepts have so much overlap, if you're familiar with another messaging system you should have little trouble using Cloud Pub/Sub. So let's take a quick tour of how messages flow through the Cloud Pub/Sub system.

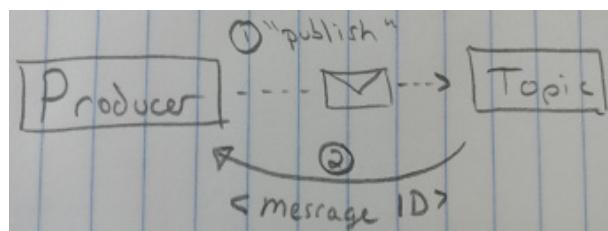
## 21.3 Life of a message

Before we dig all the way down into the low-level details of Cloud Pub/Sub, it might be useful to start with a high-level overview of how Cloud Pub/Sub works in practice. To start, let's look at the flow of a message through the system from start to finish.

At the very beginning, a message producer (also known as a sender) decides it wants to send a message. This is a fancy way of saying "you write code that needs to send messages". However, it turns out that you can't just "send a message" out into the world without categorizing it in some way. Instead, you have to decide what the message is about, and publish it specifically to that "topic". So as a result, this producer first decides on a category (called a *topic*), and then publishes a message specifically to that topic.

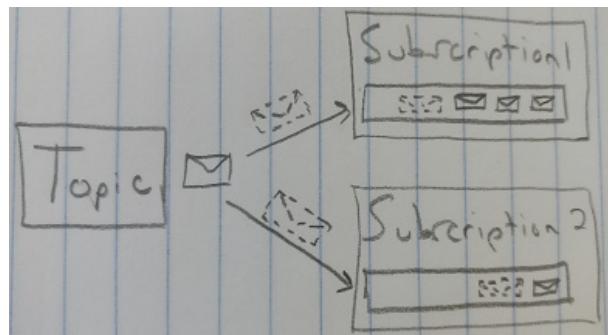
Once the message is received by Cloud Pub/Sub, it'll be assigned an ID that is unique to the topic, and that ID will be returned back to the producer as confirmation that the message was received. Think of this flow a bit like calling an office and the receptionist saying "I'll be sure to pass along the message."

**Figure 21.1. Message publishing flow**



Now that the message has arrived at Cloud Pub/Sub, there's a new question: who is interested in this message? To figure this out, Cloud Pub/Sub uses a concept of *subscriptions* which consumers create in order to say, "I'd like to get messages about this topic!" Specifically, Cloud Pub/Sub looks at all of the subscriptions that already exist on the topic and broadcasts a copy of the message to each of those subscriptions. Much like a work queue, subsequent messages sent to the topic will queue up on each subscription so that they can be consumed later. Think of this as the receptionist photocopying each message once for each department and putting it in a box at the front desk.

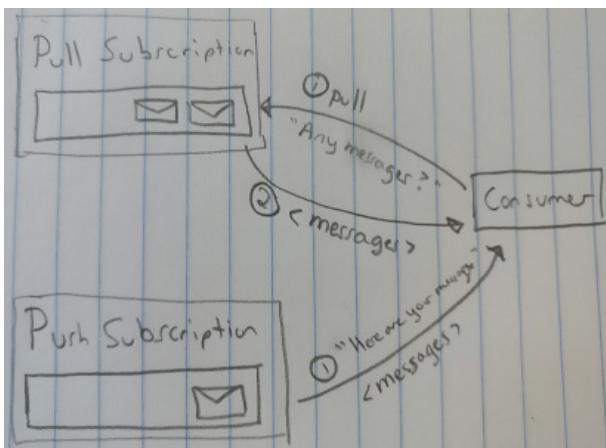
**Figure 21.2. Subscription message routing flow**



This brings us to the end of the road from the perspective of the producer — after all, their message has been received by Pub/Sub and is on the queue of everyone who expressed interest in receiving messages about that topic. But the message is still not "delivered" just yet! To understand this, we must shift our focus from the producer of a message to the consumer (or receiver) of that message.

Once a message lands in the queue of a subscription it can go one of two ways depending on how the subscription is configured: it can either be *pushed* by the subscription to the consumer, or it can sit around and wait to be *pulled* by that consumer. Let's look quickly at each of these.

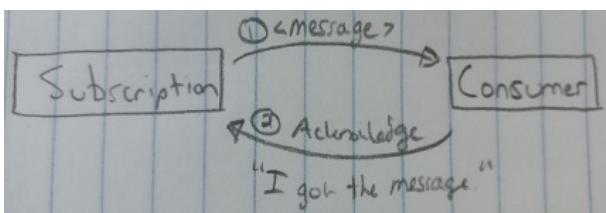
In a push-style subscription, Cloud Pub/Sub will actively make a request to some endpoint, effectively saying "Hi, here's your message!" This is similar to the receptionist walking over to the department with each message as it arrives, interrupting any current work. On the other hand, in a pull-style subscription messages will wait for a consumer on that subscription to ask for messages with the `pull` API method. This is a bit like the receptionist leaving the box of messages on the desk until the someone from the department comes to collect them. The difference between these two is shown visually below, but make sure to pay special attention to the direction of each of the arrows.

**Figure 21.3. Push versus pull subscription flows**

Regardless of how those messages end up on the consumer's side (either pushed by Cloud Pub/Sub or pulled by the consumer), you may be thinking that once the message ends up at the consumer, the work must be done, right? Actually not quite! It turns out that there is one final step required where the consumer needs to acknowledge that the message has actually been received and processed, which is called "acknowledgment".

In short, just because a consumer got a message doesn't necessarily mean that the system should consider that message "processed". For example, it's possible that the message may be delivered to the consumer, but the consumer crashes in the middle of processing the message. In that scenario, you wouldn't want the message to get dropped entirely, but rather picked up again later when the consumer recovers from its crash.

To make this all possible, consumers must acknowledge to Cloud Pub/Sub that they've actually processed the message. They do this by using a special ID they get with the message, called an `ackId`, which is unique for that particular "lease" on the message, and calling the `acknowledge` API method. Think of this a little bit like a package being delivered to the front desk, and the receptionist asking you to sign for it. This effectively serves as a form of confirmation that not only was the package sent over to you, but you actually received it and have taken responsibility for it.

**Figure 21.4. Message acknowledge flow**

So what happens when a consumer crashes before getting around to acknowledging the

message? In the case of Pub/Sub, if you don't acknowledge a message within a certain amount of time (a few seconds by default, but you can customize this for each subscription), the message is put back into the subscription's queue as though it was never sent in the first place. This process effectively gives Pub/Sub messages an "automatic retry" feature. And that's it! Once you've acknowledged the message, the subscription considers the message "dealt with", and you're all done. Now that you've seen how messages flow through the system, let's look a bit more at each one individually and explore some of the details about each.

## 21.4 Concepts

As you've seen in our walk-through, and as with most messaging systems in existence today, Cloud Pub/Sub has three core concepts: topics, messages, and subscriptions. Each of these serves a unique purpose in the process of publishing and consuming messages, so it's important to understand how they all interact with one another. Let's start off with the first thing you'll need as a message producer: topics.

### 21.4.1 Topics

As you saw in our example, topics, much like topics of conversation, represent "categories" of information and are the resource that you actually "publish" a message to. Since you publish messages to a specific topic, this means that topics are actually *required* when broadcasting messages.

For example, you might have different departments in a company, and while you may always call the main number, you may want to leave messages with different departments depending on your reason for calling. If you happen to be looking to buy something from the company you may want to leave a message specifically with the sales department. Alternatively, if you happen to need technical support with something you already bought, you may want to leave your message specifically with the support department. These different departments would correspond to different *topics* which serve to categorize your messages.

This also applies to consumers of messages in that topics also act as a way of segmenting which categories of messages you're interested in. In our example above, if you work in the support department, you would ask for messages that were from customers needing help rather than asking for all the messages the company received that day. You'll see more about this when we discuss subscriptions below. Finally, unlike most resources we've discussed on Google Cloud Platform, a topic is actually represented as nothing more than its name. This is primarily because any customization and configuration will actually be handled by consumers, which you'll see in "[Subscriptions](#)". Now that you understand topics, let's move onto the things you publish to them: messages.

### 21.4.2 Messages

Messages represent the content you want to broadcast to others who might be interested. This could be anything from a notification from a customer action (e.g.,

"someone just signed up in your app!") to a regularly scheduled reminder (e.g., "it's midnight, you may want to run a database back-up!"). Messages, as you just learned above are always published to a specific topic which acts as a way to categorize the message, effectively saying what the message is about. Under the hood, a message is composed of a base-64 encoded *payload* (some arbitrary data), as well as an optional set of plain text *attributes* about the message (represented as a key-value map) which act as metadata about the message.

Sometimes, when the payload would be excessively large, the message might instead refer to information that lives elsewhere. For example, if you're notifying someone that a new video was published, rather than setting the payload of the message to be the full content of the video file (which could be quite large), you might choose to send a link to the video on Google Cloud Storage or YouTube instead.

In addition to payload and attributes which are set by the sender, the Cloud Pub/Sub system assigns two additional fields, but only at the time when you publish a message: a message ID and a timestamp of when the message was published. These fields can be useful when trying to uniquely identify a particular message or to record confirmation times from the Pub/Sub system. One obvious question that arises is: why do you need two places to store the data that you're sending? In other words, why separate the payload from the attributes? There are actually two reasons.

First, the payload is always base-64 encoded whereas attributes are not, which means that in order to do anything meaningful with the data stored in that field, consumers must first decode the payload and then process it. As you might expect, if the payload is particularly large, there could be significant performance issues to worry about. For example, imagine if sending a very large attribute-style map exclusively as a base-64 encoded payload. If message consumers happen to check a field to see whether or not they can ignore a message, then in order to decide whether they need to pay attention to the message, they field have to decode the entire payload, which could be very large. This would obviously be wasteful, and is easily fixed by putting this particular field as a message attribute, which is **not** base-64 encoded and therefore can be checked before doing any actual decoding work on the payload.

Second, for a variety of reasons messages may be encrypted before being sent to Cloud Pub/Sub. In this case, you have a similar problem to the one described above (to check whether to ignore a message, consumers must first decode the payload) as well as a new question of whether a particular consumer is authorized to look into the message payload itself. For example, imagine a secure messaging system with its own priority ranking system (e.g. encrypted messages, each with a *priority* field which could be *low*, *medium*, or *high*). If the priority is sent along with the encrypted payload, the messaging system would be required to decrypt the message simply to decide what type of notification to send to the recipient. If the priority is instead sent in the plain text attributes, the system can inspect the less-critical data (such as message priority) without decrypting the message contents itself. Lastly, let's look at subscriptions and how they work.

### 21.4.3 Subscriptions

Sometimes referred to as queues in other messaging systems, *subscriptions* represent a desire or intent to listen to (or consume) messages on a specific topic. As noted above, they also take over most of the responsibility relating to configuration of how messages will be received, which allows for some interesting consumption patterns depending on the particular configuration.

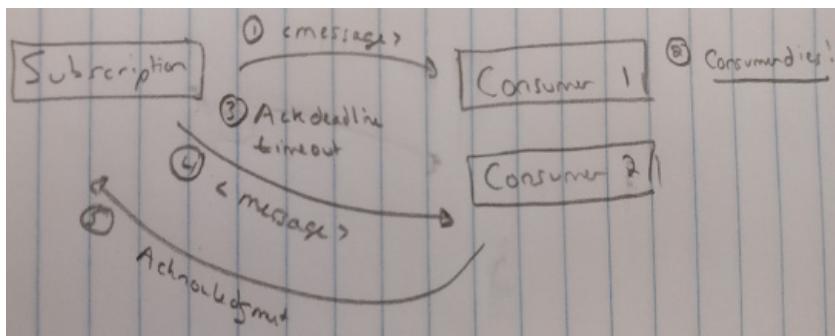
First, since each subscription receives a distinct copy of each message sent to its topic, this means that messages can be consumed from a topic without stepping on the toes of others who are also interested in that topic's messages. In other words, someone reading a message from one subscription has no effect at all on other subscriptions. Next, it follows that since each subscription sees all of the messages sent on a topic, this means that you can broadcast messages to a wider audience simply by having each consumer create a new subscription to the topic. Finally, since multiple people can consume messages from the same subscription, this means that you can use subscriptions to distribute the messages from a topic across multiple consumers, ensuring that no two consumers of that subscription will end up processing the exact same message.

To make all of these scenarios possible, subscriptions come in two flavors (pull and push), which have to do with the way consumers actually get their messages. As you learned above, the difference is whether the subscription waits for a consumer to ask for messages (pull) or actively sends a request to a specific URL when a new message arrives (push). To wrap things up, let's look briefly at the idea of "acknowledging" messages that arrive.

#### ACKNOWLEDGEMENT DEADLINES

We explained earlier how you have to acknowledge you received a message before it is actually treated as "delivered", so let's look at the details of how that works. On each subscription, in addition to the push or pull configuration, you also must specify what's called an *acknowledgment deadline*. This deadline, measured in seconds, acts as a timer of how long to wait before assuming that something has gone wrong with the consumer. Put differently, it's sort of like saying how long the receptionist should wait at your desk for you to sign for your package delivery before trying to deliver it again later.

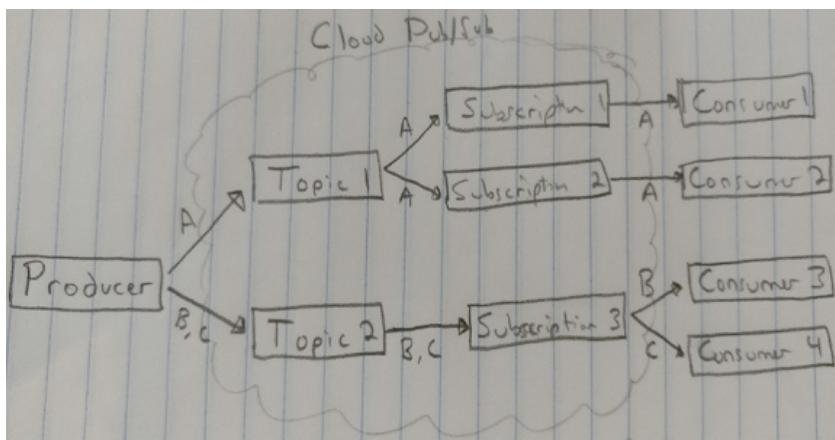
To make this clear, Figure 21.5 shows a scenario where a deadline runs out. In this example, Consumer 1 pulls a message from a subscription (1), but dies somehow before acknowledging the receipt of the message (2). As a result, the acknowledgment deadline runs out (3) and the message is put back on the subscription's queue.

**Figure 21.5. Acknowledgment expiration flow**

This means that when another consumer of the same subscription (Consumer 2) pulls a message, it gets the exact same one (4). And finally, it acknowledges the receipt of the message (5) which concludes the process of consuming that particular message. Now that you understand all of the concepts (including how messages must be acknowledged) let's look at how one example of how subscription configurations can result in different messaging patterns.

#### **21.4.4 Sample configuration**

Figure 21.6 shows an example of different subscription configurations where we're sending three messages (A, B, and C) to two topics (1 and 2). Based on the diagram, these messages are ultimately received by four different consumers (1, 2, 3, and 4).

**Figure 21.6. Example of sending messages with different subscriptions**

Let's start by looking at message A which is being sent to Topic 1. In this example, two different consumers (Consumer 1 and Consumer 2) each have their own subscription. Since subscriptions to a topic get their own copy of all the messages sent to that topic, this means that both of these consumers will see all the messages sent. This results in both Consumer 1 and Consumer 2 being notified of message A. Now

let's look at messages B and C which are sent to Topic 2.

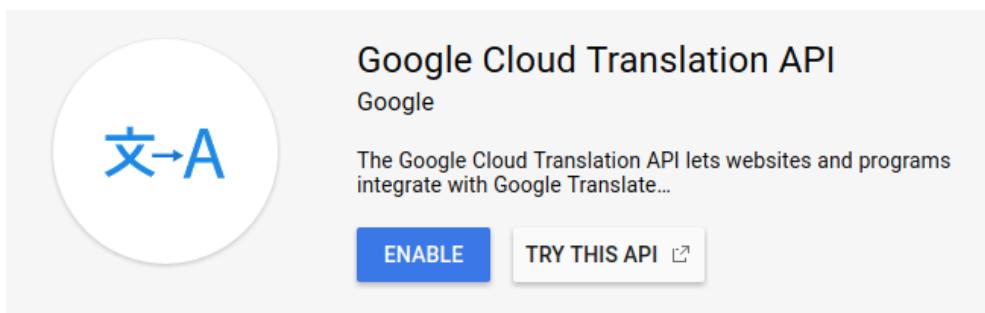
As you can see, the two consumers of Topic 2 (Consumer 3 and Consumer 4) are both using the same subscription (Subscription 3) to consume messages from the topic. Since subscriptions only get one copy of each message, this means that the two messages (B and C) will be split, with the likely scenario being that one of them will go to Consumer 3 (B in this example) and the other (C in this example) to Consumer 4. In other words, the end result of having multiple consumers to a single subscription is that they end up "splitting the work", with each getting some portion of all the messages sent.

Keep in mind, however, that this is simply the *likely* scenario, not guaranteed. It's also possible that the messages will be swapped (with Consumer 3 getting message C and Consumer 4 getting message B), or that one consumer will get both messages B and C (e.g., if one of the consumers is overwhelmed with other work). Now that we've gone over how topics and subscriptions fit together, let's get down to business and use these things.

## 21.5 Trying it out

Before we get started with writing code to interact with Cloud Pub/Sub, we have to first enable the API. To do this, visit the Cloud Console in your browser, and in the main search box at the top type in "Cloud Pub/Sub API" (remember the forward-slash). This search should have only one result, and clicking on it should bring you to a page with a big "Enable" button.

**Figure 21.7. Enable the Cloud Pub/Sub API**



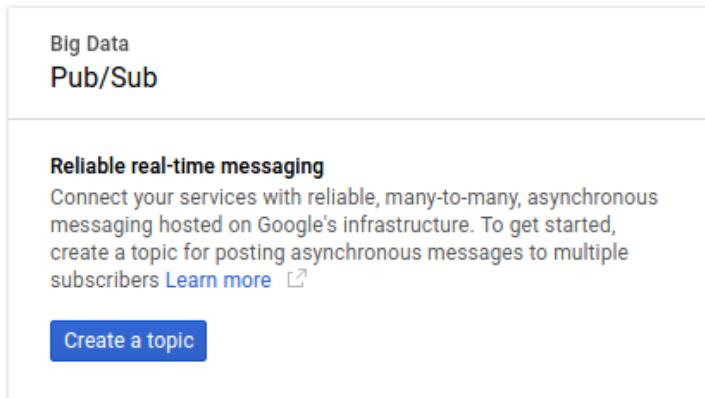
Once the API is enabled, you're ready to go. Let's start off by sending a message.

### 21.5.1 Sending your first message

To broadcast a message using Cloud Pub/Sub, you'll first need a topic. The idea behind this is that when you send a message you want to categorize what the message is about, so you use a topic as a way of communicating that. While you can create a topic in code, let's start by creating one in the Cloud Console. In the left-navigation bar, far towards the bottom under "Big Data", click on the Pub/Sub entry. The first thing you should see is an empty page with a button suggesting that you create a topic, so let's do

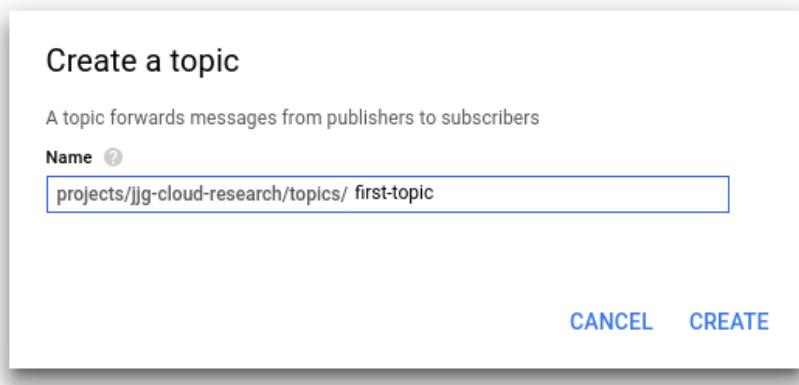
that.

**Figure 21.8. Cloud Pub/Sub's empty-state page where you can create a topic**



After you click the button, you'll see a place to enter in a topic name. Notice that the "fully qualified" name is actually a long path starting with `/projects/`. This is a way of uniquely identifying your topic among all of the topics created in Cloud Pub/Sub. Let's choose a simple name for our first topic: `first-topic`.

**Figure 21.9. Creating a topic in Cloud Pub/Sub**



After you click `create`, you should see your topic in the list. This means we're ready to start writing code that sends messages! Before we write any code, we'll first need to install the Node.js client library for Cloud Pub/Sub. To do this, we can use `npm` by running `npm install @google-cloud/pubsub@0.13.1`. Once that's done, we can get to writing some actual code.

### **Listing 21.1. Publishing a message**

```
const pubsub = require('@google-cloud/pubsub')({ ❶})
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/google-cloud-platform-in-action>

**Licensed to Asif Qamar <asif@asifqamar.com>**

```

    projectId: 'your-project-id'
});

const topic = pubsub.topic('first-topic'); ②

topic.publish('Hello world!').then((data) => {
  const messageId = data[0][0]; ③
  console.log('Message was published with ID', messageId);
}); ④

```

- ① You can access the Pub/Sub API using the API client found in the NPM package `@google-cloud/pubsub`.
- ② Since we already created this topic in the Cloud Console, we can access it without checking that it exists.
- ③ To publish a message, we use the `publish` method on our topic.
- ④ Publishing messages returns a list of message IDs, but we only want the first one.

If you were to run this code, you'd see something like the following.

#### **Listing 21.2. Output of publishing a message**

```
> Message was published with ID 105836352786463
```

The message ID that we're seeing here is an identifier that is guaranteed to be unique within this topic. Later, when we talk about receiving messages, you'll be able to use this ID to tell the difference between two otherwise identical messages. And that's it! You've published your first message!

But sending messages into the void really isn't all that valuable, right? (After all, if no one is listening, the message is effectively dropped by Cloud Pub/Sub.) So how do we go about receiving these? Let's get to work on actually receiving some messages from Cloud Pub/Sub.

### **21.5.2 Receiving your first message**

To receive messages from Cloud Pub/Sub, we first need to create a subscription. As you learned before, subscriptions are the way that we consume messages from a topic, and each subscription on a topic gets its own copy of every message sent to that topic. Let's start by using the Cloud Console to create a new subscription to the topic we already created. To do this, head back to the list of topics in the Pub/Sub section, and click on the topic name. It should expand to show you that there are currently no subscriptions, but also show a handy "New subscription" button at the top right.

**Figure 21.10. The list of topics with a "New subscription" button**

The screenshot shows the 'Topics' section of the Google Cloud Platform interface. At the top, there are buttons for 'CREATE TOPIC', 'DELETE', and 'PERMISSIONS'. Below this is a search bar labeled 'Search by topic or subscription'. A checkbox labeled 'TOPIC' is checked. Underneath, there is a list item for 'projects/jjg-cloud-research/topics/first-topic'. To the right of this item are three buttons: '+ New subscription', 'Publish', and a small upward arrow icon. Below the list item, it says 'No subscriptions'.

Go ahead and click the button to create a new subscription, and on the following page let's call the subscription (keeping with our theme) `first-subscription`. Under "Delivery type", leave this as "Pull" for now. We'll walk through "Push" subscriptions later on.

**Figure 21.11. Creating a new subscription to our topic**

The screenshot shows the 'Create a subscription' form. At the top left is a back arrow and the text 'Create a subscription'. Below this is a descriptive text: 'A subscription directs messages on a topic to subscribers. Messages can be pushed to subscribers immediately, or subscribers can pull messages as needed.' The 'Topic' field is set to 'projects/jjg-cloud-research/topics/first-topic'. The 'Subscription name' field contains 'projects/jjg-cloud-research/subscriptions/ first-subscription'. The 'Delivery Type' section has two options: 'Pull' (selected) and 'Push into an endpoint url'. The 'Create' and 'Cancel' buttons are at the bottom.

Once you click create, you should be brought back to the page listing all of your topics. If you click on the topic we created, you should see the subscription that we just created in the list.

**Figure 21.12. Viewing a topic and its subscriptions**

The screenshot shows the Google Cloud Platform Pub/Sub interface. At the top, there are buttons for 'CREATE TOPIC' (with a plus sign), 'DELETE' (with a trash can), and 'PERMISSIONS'. Below this is a search bar labeled 'Search by topic or subscription'. A checkbox labeled 'TOPIC' is checked. Underneath, a table lists a subscription:

Subscription	Delivery Type	Push Endpoint URL	Action Buttons
<a href="#">projects/jjg-cloud-research/topics/first-topic</a>	Delivery Type	Push Endpoint URL	<a href="#">+ New subscription</a> <a href="#">Publish</a> <a href="#">^</a> <a href="#">Edit</a> <a href="#">Delete</a>
<a href="#">projects/jjg-cloud-research/subscriptions/first-subscription</a>	Pull		

Now that we have a subscription, we can head go write some code to interact with it. Remember that the idea behind a subscription is that it's a way to consume messages sent to a topic. This means that we first have to send a message to our topic, and then ask the subscription for any messages received. To do that, let's start by running the script from above that publishes a message to our topic. When you run it, you should see a new message ID.

#### **Listing 21.3. Running our code to publish a message**

```
> Message was published with ID 105838342611690
```

Since there is a subscription on our topic this time, it means that we weren't sending a message to into the void of nowhere, but instead a copy of that message should be waiting for us when we ask our subscription for messages. To do that, we'll use the `pull` method on a subscription, shown here.

#### **Listing 21.4. Consuming a message**

```
const pubsub = require('@google-cloud/pubsub')({
  projectId: 'your-project-id'
});

const topic = pubsub.topic('first-topic');
const subscription = topic.subscription('first-subscription'); ① ②

subscription.pull().then((data) => { ③
  const message = data[0][0];
  console.log('Got message', message.id, 'saying', message.data);
});
```

- ① Get a reference to a topic simply by using its name.
- ② Similarly reference a subscription using its name.
- ③ Consume messages with the `pull()` method on your subscription.

**Listing 21.5. Output of pulled message**

```
> Got message 105838342611690 saying Hello world!
```

Notice here that the message ID is the same as the one we published, and the message content is the same also! Looks good, right? It turns out that we've actually forgotten a really important step: acknowledgment!

If you try to run that same code again in about 10 seconds, you'll actually get that exact same message again. What's happening under the hood is that the subscription knows it gave that message out to the consumer, but the consumer (our script) never acknowledged that we got it. Because of that, the subscription responds with the message the next time a consumer tries to pull messages. To fix this, we can use simple method on the message called `ack()`, which makes a separate request to Pub/Sub telling it that we did indeed receive that message. Our updated code will look something like the following.

**Listing 21.6. Consume and acknowledge a message**

```
const pubsub = require('@google-cloud/pubsub')({
  projectId: 'your-project-id'
});

const topic = pubsub.topic('first-topic');
const subscription = topic.subscription('first-subscription');

subscription.pull().then((data) => {
  const message = data[0][0];
  console.log('Got message', message.id, 'saying', message.data);
  message.ack().then(() => {           ①
    console.log('Acknowledged message ID', message.id,
               'with ackId', message.ackId);  ②
  });
});
```

- ① `message.ack()` is bound to the right `ackId`, meaning you don't need to keep track of lots of IDs.
- ② Notice that the `ackId` is different from the message ID. This is due to the fact that the same message may be consumed by multiple different people.

In this case, you should see that running this code receives the same message again, but this time tells Cloud Pub/Sub that it received the message by sending an acknowledgement request.

**Listing 21.7. Output of acknowledged message**

```
> Got message 105838342611690 saying Hello world!
Acknowledged message ID 105842843456612 with ackId QV5AEkw4A0RJUytDCypYEU4EISE-
MD5FU0RQBhYsXUZIUTcZCGhRdk9eIz81IChFEQcIFAV8fXFdUXVeWhoHUQ0ZcnxkfDhdRwkAQAV5VVsRDXp
tXFc4UA0cenljfw5ZFwQE01J8d5qChutoZho9XxJLLD5-MzZF
```

If you were to try pulling again, you'd see that the message has disappeared. This is

because Pub/Sub considers it "consumed" from this subscription and therefore won't send it to the same subscription again.

## 21.6 Push subscriptions

So far, all of the messages we've received have been *pulled* from the subscription. In other words, we've specifically asked a subscription to give us any available messages. As mentioned before though, there's another way of consuming messages that doesn't necessarily require you to ask for the messages, where instead of *pulling* messages from Cloud Pub/Sub, Cloud Pub/Sub *pushes* messages to you as they arrive.

These types of subscriptions require you to configure where push notifications should be sent whenever a new message arrives. Typically this means that Pub/Sub will make an HTTP call to an endpoint you specify containing the same message data that we saw when using regular pull subscriptions. So what does this process look like? How do we handle push notifications?

First, we'd need to write a handler that accepts an incoming HTTP request with a message body. As we saw before, once the handler receives the message, it would be responsible for acknowledging it, but the way in which we do this is a bit different. With a pull subscription, we'd make a separate call back to Cloud Pub/Sub letting it know that we've received and processed the message. With a push subscription, we'll rely on HTTP response codes to communicate this. In this case, an HTTP code of 204 ("No Content") is our way of saying that we've successfully received and processed the message. Any other code (e.g., a 500 Server Error or 404 Not Found) is our way of telling Cloud Pub/Sub that there was some sort of failure.

Put more practically, if you want to handle push subscriptions you'll just need to write a handler that accepts a JSON message and returns a 204 code at the end. A handler like this might look something like the following snippet which uses ExpressJS.

### **Listing 21.8. A simple push subscription handler**

```
const express = require('express');
const app = express();

app.post('/message', (req, res) => {
  console.log('Got message:', req.message); ①
  res.status(204).send() ②
});
```

- ① First we do something with the message. In this case, we log it to the console.
- ② To acknowledge that the message was handled, we explicitly return a 204 response code.

Now that we've seen what a handler for incoming messages might look like, this only leaves the question of how we instruct Cloud Pub/Sub to send messages to this handler! As you might guess, this is as easy as creating a new subscription with the URL configured. We can do this in lots of ways, but let's show how we can do this using the Cloud Console.

In the Pub/Sub area of the console, we can re-use the topic we created before (`first-topic`) and skip ahead to creating a new subscription. Let's imagine that we've deployed your simple ExpressJS application with the very basic handler to your own domain (e.g., `your-domain.com`). By browsing into the topic and clicking the "Create Subscription" button at the top, you should land on a form where you can specify a subscription name and a URL of where to push messages. Shown below, I'm using `push-subscription` as the name and [your-domain.com/message](https://your-domain.com/message) as the URL (note that in our code snippet, the path is `/message`).

**Figure 21.13. Creating a push subscription**

A subscription directs messages on a topic to subscribers. Messages can be pushed to subscribers immediately, or subscribers can pull messages as needed.

**Topic**  
projects/jjg-cloud-research/topics/first-topic

**Subscription name** [?](#)  
projects/jjg-cloud-research/subscriptions/push-subscription

**Delivery Type** [?](#)  
 Pull  
 Push into an endpoint url [?](#)  
https://your-domain.com/message

[More options](#)

**Create** **Cancel**

Once you click "Create", incoming messages to this topic will be routed to your handler, with no pulling or acknowledging needed.

**WARNING**

You may get errors about whether you "own" a domain or not. This is entirely normal, and is Google's way of making sure that messages are only sent to domains that are owned by you.

To read more about whitelisting your domain for use as a Pub/Sub push endpoint, check out [cloud.google.com/pubsub/docs/push#other-endpoints](https://cloud.google.com/pubsub/docs/push#other-endpoints).

At this point you should have a good grasp on many of the ways to interact with Cloud Pub/Sub. This makes it a great time to switch gears and start looking at how much all of this costs to actually use by exploring how pricing works for Cloud Pub/Sub.

## 21.7 Understanding pricing

As with many of the Google Cloud APIs, Cloud Pub/Sub is yet another that only charges you for the resources and computation that you actually use. To do this, Pub/Sub bills based on the amount of data that you broadcast through the system, at a maximum rate of \$0.06 per Gigabyte of data. While this loosely corresponds to the number of messages, it is going to be very dependent on the size of the messages you're sending. For example, we mentioned before how instead of sending an entire video file's content through a message, you might instead send just the link to the video. The reason for this is not only that it's not exactly what Pub/Sub was designed for, but also that it will cost you far more (as it's likely the video file will be far cheaper to download from somewhere else, such as Google Cloud Storage).

To make this more concrete, let's look at a more specific example. Imagine that your system sends 5 messages every second, and you have 100 consumers all interested in those messages. Let's also assume that these messages are really tiny. How much will this cost you over the course of a month? First, let's look at what requests are being made throughout the month. To do that, we'll need to know how many messages we're sending, which requires a bit of math.

- There are 86,400 seconds in each day, and for purposes of simplification let's work with the idea that there are 30 days in an average month. This brings us to 259,200 seconds in one of our months.
- Since we're sending 5 messages per second, this means that we're actually sending a total of 1,296,000 messages in one of our months.

Now we actually have to think about what requests we're making to Cloud Pub/Sub. First, we make one publish request per message since that is absolutely required in order to send the message. However we also have to consider the consumer's side of things! Every consumer needs to either make a pull request to ask for each message, or have each message sent to them via a push subscription. This means that we actually make one additional request per consumer for each message.

- There are 1,296,000 messages sent in one of our months.
- Since we have 100 consumers, this means that to read all of these messages, we make a total of 129,600,000 pull (or push) requests to read those messages in one of our months.

This brings our overall total to:

- 1,296,000 publish requests, plus
- 129,600,000 pull (or push) requests

which comes to a grand total of 130,896,000 requests.

The question now becomes how we convert this to data. To do that, it's important to know that the minimum billable amount for each request is 1 KB. This means that even if our messages are tiny, the total amount of data here is about 130 million KB, or just under 131 GB. At the rate of \$0.06 per GB of data, your bill at the end of the month for

these 1.3 million messages sent to 100 different consumers will be \$7.86.

## 21.8 Messaging patterns

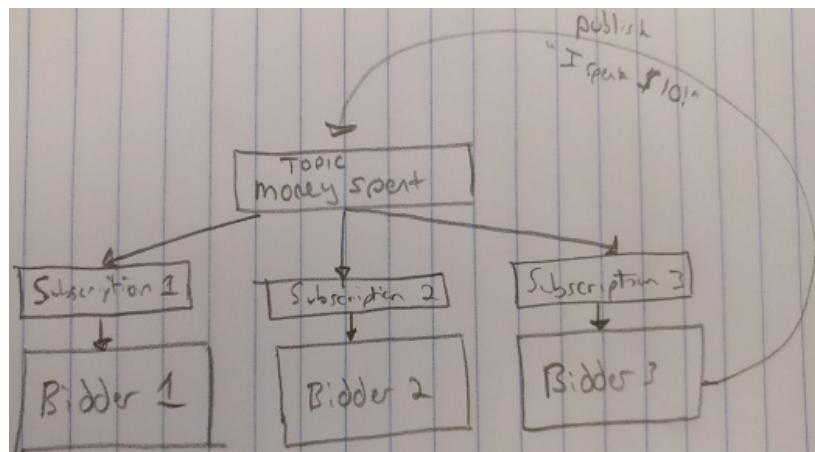
Even though you've written a bit of code to communicate with Cloud Pub/Sub, the examples are a bit simplistic in that they assume static resources (topics and subscriptions) when in truth it's pretty common that you'll want to dynamically change your resources. For example, you may want every new machine that boots up to subscribe to some system-wide flow of events. To make real-life situations a bit more concrete, let's look at two common examples (fan-out messaging and work-queue messaging) and write some code to bring these patterns to life.

### 21.8.1 Fan-out broadcast messaging

Just to refresh, a fan-out system is using Pub/Sub in such a way that any single sender is broadcasting messages to a broad audience. For example, imagine you have a system of many machines with each one automatically bidding on items on eBay, and you want to keep track of how much money you've spent overall. Since you have multiple different servers all bidding on items, you need a way of communicating to everyone how much money has been spent so far, otherwise you'll be stuck polling a single central server for this total. To accomplish this you could use a fan-out message where each server broadcasts the fact that they've spent money to a specific Pub/Sub topic. All the other machines listening on this topic are able to keep track of how much money was spent in total, and are immediately notified of any new expenditures.

Using the concepts of Cloud Pub/Sub that we learned about before, this would correspond to a topic called `money-spent` where each interested consumer (or "bidder" in our example) would have their own subscription. In this way, each subscriber would be guaranteed to get each message published to the topic. Further, each of these same consumers would also be producers, telling the topic about when they spend money as it happens in real time.

**Figure 21.14. Overview of message flow for eBay bidder**



As you can see, each "bidder" machine has exactly one subscription to the `money-spent` topic, and each can broadcast messages to the topic to notify others of the spend. So let's write some code to do all of this, starting with a method that we should expect to run whenever one of our eBay bidder instances turns on. To be explicit, this method is ultimately responsible for doing a few key things:

1. Getting the total spend count before starting to bid on eBay
2. Kicking off the logic that actually bids on eBay items
3. Updating the total amount spent whenever it changes

#### **Listing 21.9. Function to start bidding on eBay items**

```
const request = require('request');
const pubsub = require('@google-cloud/pubsub')({
  projectId: 'your-project-id'
});

let machineId;
const topic = pubsub.topic('money-spent'); ❶
const amountSpentUrl = 'http://ebaybidder.mydomain.com:8080/budgetAvailable.json';
let amountSpent;

startBidding = () => {
  // First get the available budget from a central place.
  request(amountSpentUrl, (err, res, body) => {
    amountSpent = body;

    // Create a subscription to listen for money being spent.
    const subscription = topic.subscription(machineId + '-queue'); ❷
  });
}

subscription.on('message', (message) => { ❸
  console.log('Money was spent!', message.data);
  amountSpent += message.data;
  message.ack(); ❹
}); ❺

// Start bidding on eBay items here!
bidOnItems();
}); ❻
}
```

- ❶ We use a well-known name (`money-spent`) for our topic, with the assumption that it's already been created.
- ❷ We assume that this is the method you call first when a new bidding machine is turned on.
- ❸ We use a special subscription name so that it will either be created or re-used if it already exists. The assumption is that you have a unique name (`machineId`) for each individual bidding VM.
- ❹ This method will be called every time a new message appears on the topic.
- ❺ Whenever a new message arrives, we should update the amount spent. Note that the amount is a **delta** (e.g., "I spent \$2.50" will show up as 2.5, but "I got a refund of \$1.00" would show up as -1). This will become more clear in the next code snippet.
- ❻ Don't forget to acknowledge that we received the message and processed it.

```
// This method will run whenever money was spent by anyone.
subscription.on('message', (message) => { ❻
  console.log('Money was spent!', message.data);
  amountSpent += message.data; ❼
  message.ack(); ❽
}); ❾

// Start bidding on eBay items here!
bidOnItems();
}); ❿
}
```

With this snippet written, we now need to devise how we actually update the amount spent when we bid (or get outbid) on items. To do that, let's first think through all the ways this value can change. The obvious way is when we place a bid on an item. That is, if I place a bid of \$10.00 on a pair of shoes, I am committed to buying that item, so even though I haven't *won* the shoes yet I still need to add that 10 dollars to the amount spent, as the default action is that I have *committed* to spending this amount. That said, if I happen to be outbid or lose the auction somehow, that money is now free to be spent on other things. In other words, when we place a bid, we need to mark money as "spent" and when we're outbid on an item, we need to mark money as "recovered" or "refunded". Let's turn these two actions into (pseudo-) code.

#### **Listing 21.10. Functions to update the amount spent locally.**

```
const pubsub = require('@google-cloud/pubsub')({
  projectId: 'your-project-id'
});

let machineId;
const topic = pubsub.topic('money-spent');

const broadcastBid = (bid) => {
  // Broadcast that the bid.amount has been "spent".
  return topic.publish({
    data: bid.amount,          ①
    attributes: {
      machineId: machineId,  ②
      itemId: bid.item.id
    }
  }, {raw: true});           ③
}

const broadcastRefund = (bid) => {
  // Broadcast that the bid.amount has been refunded/recovered.
  return topic.publish({
    data: -1 * bid.amount,    ④
    attributes: {
      machineId: machineId,
      itemId: bid.item.id
    }
  }, {raw: true});
}
```

- ① As noted before, this is a **delta**, meaning to convey that we *spent* one dollar, we need to send a value of **1.00**.
- ② For debugging purposes, we send along the machine that sent this message, as well as the eBay item ID in the message attributes.
- ③ In order to send message data separate from message attributes, we need to tell our client library that this is a "raw" message. Otherwise, it would treat the entire block as the payload.
- ④ Since we are releasing funds, we leave flip the sign of the value. In other words, when refunded \$1.00, we add **-1.00** to the amount spent.

Now that all the code is written, let's look at this from a high level to see exactly

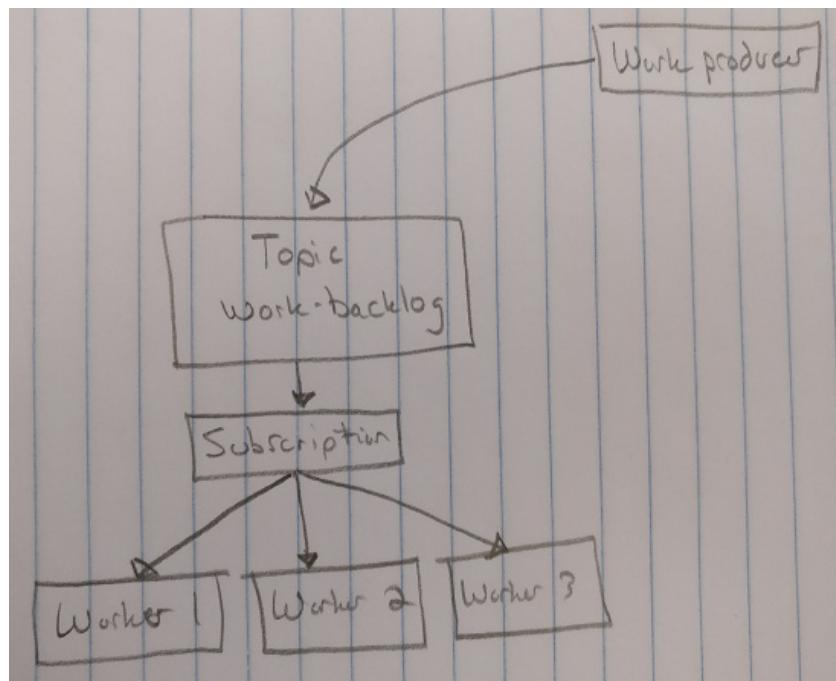
what's happening. First, each bidding machine turns on and requests from some central authority what the current budget is (however we don't say how exactly that works, as it's not really relevant here). After that, these machines immediately either gets or creates a Pub/Sub subscription for itself on the `money-spent` topic. The subscription has a call-back registered which will execute every time a new message arrives, whose main purpose is to update the running balance.

Once that process is complete, it begins the process of bidding on items. Whenever we bid on an item, we should call the `broadcastBid` function to let others know that we've placed a bid. Conversely, if we're ever outbid (or the auction is canceled), we simply call the `broadcastRefund` function which will tell other bidders that money we had marked as spent is actually not spent. Now that we've seen how fan-out works, let's take a quick look at how we can use Pub/Sub to manage a queue of shared work across multiple different workers.

### 21.8.2 Work-queue messaging

Unlike fan-out messaging, where each message is delivered to lots of other consumers, work-queue messaging is a way of distributing work across multiple different consumers, where each message is (ideally) processed only by a single consumer.

**Figure 21.15. Work-queue pattern of messaging**

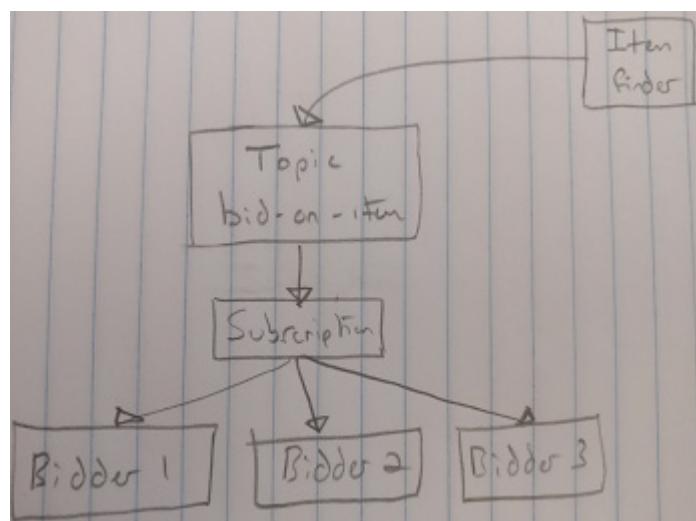


Relying on our eBay bidding example, let's imagine that we want to design a way to instruct all of our bidding machines that they should bid on a specific list of items. However, instead of using a fixed list of items let's say that we're going to continue

adding to the list so it could get incredibly long. Putting this in terms of Cloud Pub/Sub, each message would primarily contain an ID of an eBay item to purchase, and when a bidding machine receives this message, it would place a bid on the item. So how should we lay out our topics and subscriptions?

If we followed the fan-out broadcast style of messaging, each bidding machine would get every message, which would mean that each machine would place their own distinct bid on the item—which would get very expensive! Here, we could use the work-queue pattern and have a single subscription which each of our bidders listen to for notifications of messages. By doing this, each message would be handled by a single machine, rather than every machine.

**Figure 21.16. Item finder sending messages to bidders**



So how would this look? First, we'd create a new topic (`bid-on-item`) along with a single pull subscription (`bid-on-item-queue`). After that, we'd modify our bidding machines to consume messages from this new subscription, and bid accordingly. Let's assume that we create the topic and subscription manually, and explore what our code would look like.

#### **Listing 21.11. Functions to update the amount spent locally.**

```

const pubsub = require('@google-cloud/pubsub')({
  projectId: 'your-project-id'
});

const topic = pubsub.topic('bid-on-item'); ①
const subscription = topic.subscription('bid-on-item-queue');

subscription.on('message', (message) => { ②
  message.ack(() => { ③
    bidOnItem(message); ④
  });
});
  
```

```
});  
});
```

- ➊ Since we're dealing with static resources, we can construct references to our topic and subscription just by names.
- ➋ Just as before, we register a message-handling call-back which is called as each message is received.
- ➌ Since the bidding process can be long, we should start by acknowledging the message.
- ➍ After the acknowledgment succeeds, we instruct our bidding machine to actually bid on the item.

Notice that we are erring on the side of accidentally *not* bidding on items if there is some sort of problem. For example, if the `bidOnItem` method throws an error for some reason, we've already acknowledged the message, so we won't get another notification to go bid on that item again. Compare this to the alternative where you might get the same item twice and bid against yourself. If you were to add a way to check that you aren't already the high bidder, then it might make sense to do the bidding first and only acknowledge the message after the bid succeeds. That said, this is all the code you have to write and you have a single-subscription work-queue messaging system!

## 21.9 Summary

- Messaging is the concept of sending and receiving data as events across processes, which can include sending messages to any number of parties (one-to-one, one-to-many, or many-to-many).
- Cloud Pub/Sub is a fully-managed, highly-available messaging system that handles message routing across lots of different senders and receivers.
- Messages can be sent to topics, which can then be subscribed to by creating subscriptions.
- Messages can either be pulled by the receiver ("Any messages for me?") or pushed by the sender ("There's a message for you!").
- Cloud Pub/Sub is most commonly used for fan-out (broadcast) or work-queue (orchestration) messaging.
- Cloud Pub/Sub charges based on the amount of data sent through the system, meaning larger messages cost more than smaller messages, with a minimum of 1 KB per message.