Listing 11.23 Example interaction with Datastore from `ndb` library

```
from google.appengine.ext impor tndb

class TodoList(ndb.Model):
  name = ndb.StringProperty()
  completed = ndb.BooleanProperty()

# Create a new TodoList
my_list = TodoList(name='Groceries', completed=False)
key = my_list.put()

# Find TodoLists by name
lists = TodoList.query(name='Groceries')

# Delete the TodoList by ID
my_list.delete()
```

Imports the ndb library to use it (similar to require in Node.js)

Defines 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 you intend to set on entities)

ndb allows you to set many property types, such as strings, lists, booleans, and more.

Creates a new entity by creating an instance of a model and using the put() method to persist it to Datastore

Queries Datastore for matching entities using the query() method

Deletes the entity by calling .delete() on it

In the listing, a couple of interesting things are worth mentioning. First, I didn't talk about authentication at all. Authentication happens automatically because your code is running inside a managed sandbox environment, so I didn't need to. As a result, you don't have to set which URL to send API requests to, specify which project you're interacting with, or provide any private keys to gain access. By virtue of running inside App Engine, you're guaranteed secure and easy access to your instance of Cloud Datastore. Also, you didn't have to define any special dependencies to use the `ndb` package in your application. 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 definitely should read more about the various libraries available in the language you intend to work in. App Engine has 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 you might cache data temporarily using Memcached.

11.4.2 Caching ephemeral data

In addition to storing data permanently, applications commonly will want to store data temporarily 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 to 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 App Engine Standard provides a hosted Memcached service that you can use with no extra setup at all.

NOTE You may not be familiar with Memcached. This service offers an incredibly simple way to store data temporarily, always using a unique key. Think of it like a big shared Node.js JSON object store that you manipulate by calling `value = get(key)`, `set(key, value)`, and so on.

App Engine's Memcached service acts like a true Memcached service, so the API you use to communicate with it should feel familiar if you've ever used Memcached yourself. The following listing shows some code that writes, reads, and then deletes a key from App Engine's Memcached service.

Listing 11.24 Example interaction with App Engine Standard's Memcached service

```
from google.appengine.api import memcache

memcache.set('my-key', 'my-value')
memcache.get('my-key')
memcache.delete('my-key')
```

Imports the App Engine memcached library

Sets keys using the set(key, value) method

Retrieves keys using get(key)

Removes the key by using delete(key)

Although the API to talk to App Engine's Memcached service is the same as a regular Memcached instance running on a VM, it isn't a true Memcached binary running in the same way. Instead, it's a large shared service that acts like Memcached. As a result, you need to keep a few things in mind.

First, your Memcached instance will be limited to about 10,000 operations per second. If your application gets a lot of traffic, you may need to think about using your own Memcached cluster of VMs inside Compute Engine. Additionally, you may find that certain keys in Memcached receive more traffic than others. For example, if you use a single key to count the number of visitors to your site, App Engine will have a hard time distributing 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, you 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 that, the service will reject the request. Additionally, because Memcached supports batch or bulk operations, where you set multiple keys at once, the most data you can send in one of those requests (the size of the keys combined with the size of the values) is 32 MB.

Finally, you must consider the shared nature (by default) of the Memcached service and how that affects the lifetime of your keys. Because the Memcached service is shared by everyone (though it's 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.

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, Memcached 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 switch from caching to queueing and 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 doesn't need to be done right away but instead could be delayed. This work might be sending an email or recalculating some difficult result, but typically it's something that takes a while and can be done in the background. To handle this, you may end up building your own system to handle work to be done later (for example, storing the work in a database and having a worker process handle it) or using 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 system works, imagine you have a web application with a profile page that stores a user's email address. If they want to change that email address, you might want to send a confirmation email to the new address to prove that they control the email they provided. In this case, sending an email might take a while, so you wouldn't want to sit around waiting for it to be sent. Instead, you'd want to schedule the work to be done, and once it was confirmed as "scheduled," you could send a response telling the user that they should get an email soon.

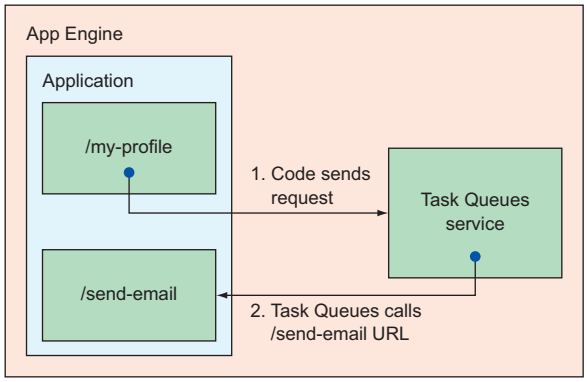


Figure 11.16 An application that uses Task Queues to schedule future work

As shown in figure 11.16, first the code that updates the email 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 you scheduled, and your code will pick up the baton, doing the email sending work. In Python code, this might look something like the following listing.

Listing 11.25 An example application that uses Task Queues to schedule work for later

```
import webapp2
from google.appengine.api import taskqueue

classMyProfileHandler(webapp2.RequestHandler):
  defget(self):
    self.response.write('This is your profile.')

  defpost(self):
    task = taskqueue.add(
            url='/send-email',
            params={'email': self.request.get('email')})

classSendEmailHandler(webapp2.RequestHandler):
  defpost(self):
    some_email_library.send_email(
      email=self.request.get('email'))

app = webapp2.WSGIApplication([
  ('/my-profile', MyProfileHandler),
  ('/send-email', SendEmailHandler),
])
```

Imports the task queue libraries for Python

Renders a full HTML page where users can change their email

When someone makes a POST request to the /my-profile URL, the previous method won't handle it; this one will.

You 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.

The Task Queues service will make the request as you scheduled, so it's your job to define what happens at that point. In this case, you'd send an email to the desired recipient.

Makes sure incoming requests are routed to the correct handlers

The Task Queues service is incredibly powerful and has far more features than I could cover in one 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.

You can find all of these things and more 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's unique as well as useful: traffic splitting.

11.4.4 Splitting traffic

As you saw, when deploying new versions of services, it's possible to trigger a deployment without making the new version live yet. This arrangement 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 you wanted to slowly test out new versions, shifting traffic from one version to another over the course of the day?

For example, in figure 11.17, you can see a hard switch-over from version A to version B, where 100% of the traffic originally sent toward version A immediately jumps over toward version B.

With traffic splitting, you can control what percentage of traffic goes to which version. You could be in a state where 100% of traffic is sent to version A and

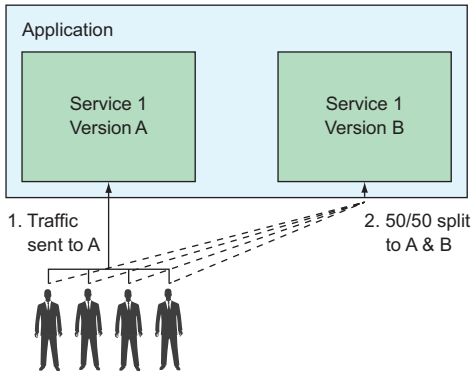


Figure 11.17 A hard switch-over of all traffic from version A to version B

transition to a state where 50% remains on version A and 50% is migrated to version B (figure 11.18).

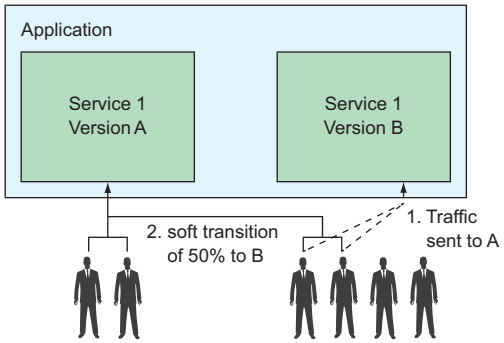


Figure 11.18 A soft transition of 50% of traffic to version B

You may have a lot of reasons for wanting 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 those 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. I'll talk about A/B testing here because it most clearly illustrates the functionality that App Engine's traffic splitting offers.

> **NOTE** It may be helpful to set the `promote_by_default` flag back to `false` to avoid the automatic hard switch-over during a typical deployment.

To demonstrate this, you can deploy two versions of a service (called `trafficsplit`) using the `--version` flag to name them `version-a` and `version-b`. Start by deploying `version-a`, which is your "Hello, world!" application tweaked so it says, "Hello from version A!" After that, you can deploy a second version, called `version-b`,

which you modify slightly to say, "Hello from version B!" Once you're done deploying, you should be able to access both versions by their names, with `version-a` being the default:

```
$ 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, you have one version that's the default (`version-a`) and another that's deployed but not yet the default (`version-b`). If you click the Versions heading in the left-side navigation and choose `trafficsplit` from the service dropdown, you can see the current split (or allocation) of traffic is 100% to `version-a` and 0% to `version-b` (figure 11.19).

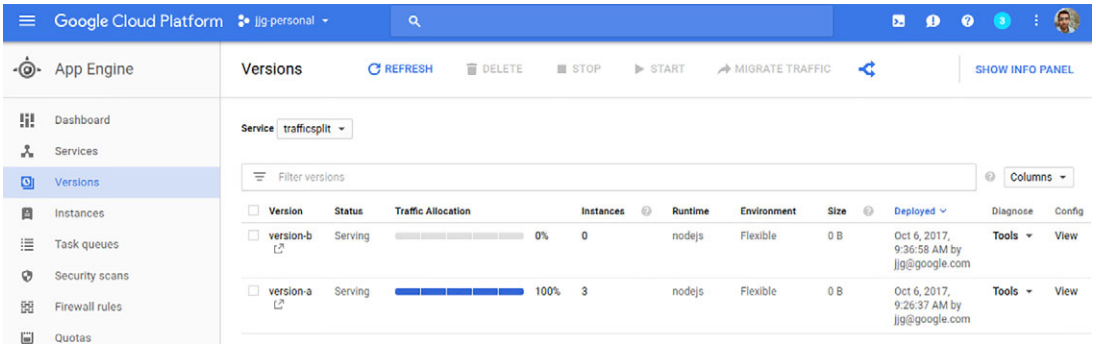


Figure 11.19 Available versions and their traffic allocations

If you wanted to split 50% of the traffic currently going to `version-a`, you could do this by clicking the Split Traffic icon (which looks like a road sign forking into two arrows), which brings you to a form where you can configure how to split the traffic (figure 11.20).

For the purposes of this demonstration, you'll choose the Random strategy when deciding which requests go to which versions. Generally, the Cookie strategy is best for user-facing services, so the same user won't see a mix of versions; instead, they'll stick with a single version per session. After that, you'll add `version-b` to the traffic allocation list and route 50% of the traffic to that version. Once that's done, click Save. Viewing the same list of versions for your service now should show that half of the traffic is heading toward `version-a` and the other half toward `version-b` (figure 11.21).

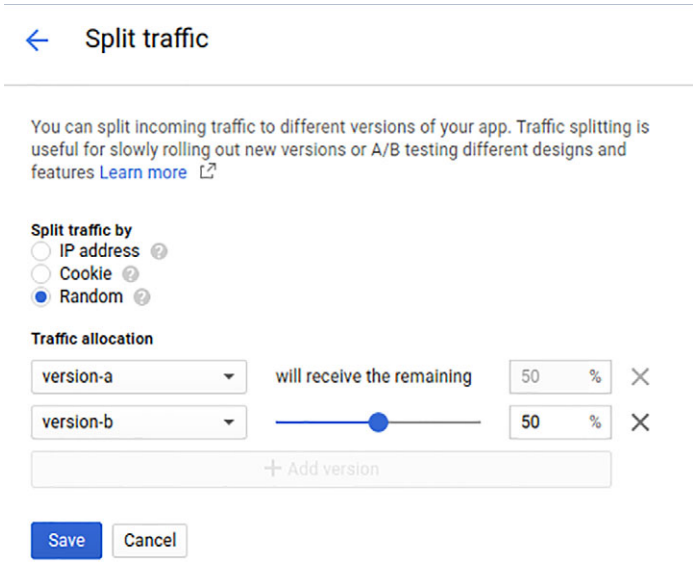


Figure 11.20 The form where you can choose how to split traffic between versions

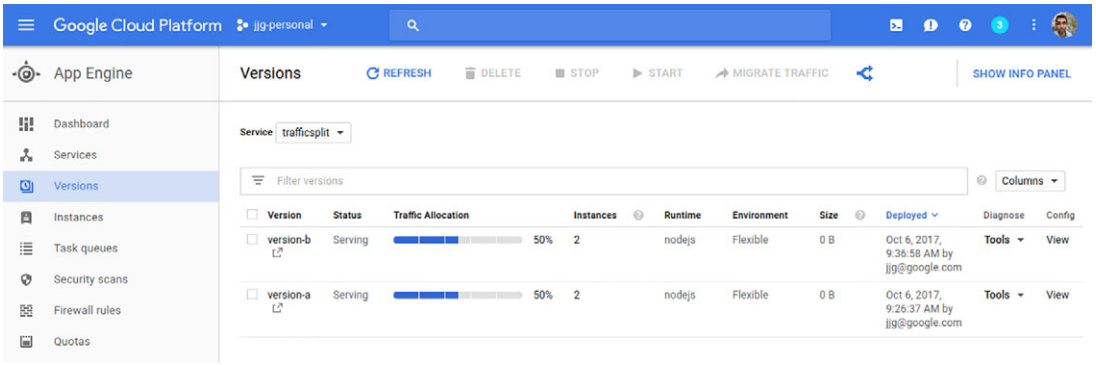


Figure 11.21 The list of versions with traffic split evenly between them

To check whether this worked, you can make a few requests to the default URL for your service and see how you flip-flop between answers from the various versions:

```
$ 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, App Engine is capable of many more things—enough to fill an entire book—so if you're interested in learning about all of the features of App Engine, it's definitely worth picking up a book focusing exclusively on that topic. On the other hand, this chapter doesn't have enough space to talk about everything, so it's time to switch gears and look at how much it costs to run your applications on App Engine.

11.5 Understanding pricing

Because App Engine has many services, each with its own pricing scheme, instead of going through every single service and looking at how much it costs, we'll see how the computational aspects of App Engine are priced and look at costs for a few of the services that I discussed in this chapter, starting with computing costs.

Because App Engine Flex is built on top of Compute Engine instances, the costs are identical to Compute Engine, which I discussed in depth in section 9.7. App Engine Standard, on the other hand, uses a sandbox with different instance types. But it still follows the same principle: App Engine Standard instances are priced on a per-hour basis, which varies 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 per hour (35% more). Table 11.2 shows prices for the various instance types in Iowa.

Table 11.2 Cost for various App Engine instance types

Instance type	Cost (per hour)
F1	$0.05
F2	$0.10
F4	$0.20
F4_1G	$0.30

In addition to the cost for computing resources, App Engine charges for outgoing network traffic, like the other computing environments you've seen. For App Engine Flex, the cost is again equivalent to the cost for Compute Engine network traffic. For App Engine Standard, a flat rate per GB 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 (for example, Task Queues or Memcached) don't 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 shared Memcached caching has no charge for data cached. To learn more about this pricing, it's worth looking through the details, which you can find at https://cloud.google.com/appengine/pricing. Now that I've covered how much everything costs,

I'll zoom out and discuss the big picture of when to use App Engine and, if you do use it, which environment is the best fit.

11.6 When should I use App Engine?

To figure out whether or not App Engine's a good fit, let's start by looking at its scorecard, which gives you a broad overview of App Engine's characteristics. But because App Engine's environments are almost like entirely separate computing platforms, it seems worthwhile to have a separate scorecard for the different options (figures 11.22 and 11.23).

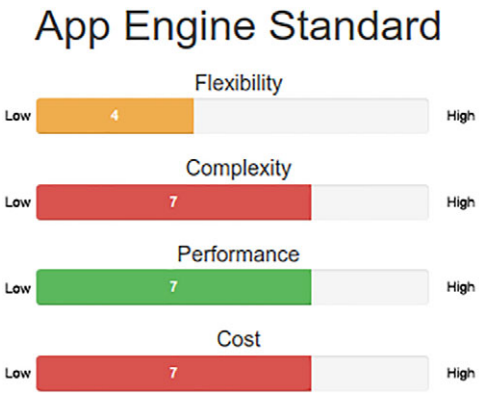


Figure 11.22 The scorecard for App Engine Standard

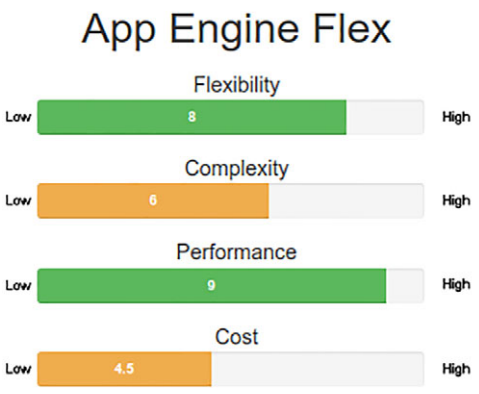


Figure 11.23 The scorecard for App Engine Flex

11.6.1 Flexibility

The first thing to notice about App Engine is that whereas App Engine Flex offers levels of flexibility (what code you can run) similar to those of Compute Engine or Kubernetes Engine, App Engine Standard is far more limited. Its limitations are due to App Engine Standard's reliance on a sandbox runtime to execute code, which limits it 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 there's much less of a difference.

For example, you have different ways to configure how scaling works (such as manual or automatic scaling) for both environments, and you can choose specific instance configurations for both environments. Overall, App Engine Standard is relatively inflexible with regard to instance configuration, whereas App Engine Flex allows you to control almost all details of the resources that will be running your code.

11.6.2 Complexity

When it comes to the complexity of the two environments, the difference is fairly substantial, despite the overall moderate scores in this area. With App Engine Standard, you have a lot to learn, specifically with regard to the runtime environments and the limitations that come with them. For example, when you were building a "Hello, world!" application in Python, you relied on the `webapp2` framework, which works quite well with App Engine. If you wanted to use a different Python web framework (such as Django or Flask), you'd have to do a bit of work to ensure that everything ran 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 because you don't need to understand all of the scaling details like instance groups. It's also slightly less complex than Kubernetes Engine, because you don't have 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

Because 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 App Engine Flex, only a small bit of overhead consumes any of the CPU time on the instances running your code. Because App Engine Standard executes your code in a sandbox environment, you see a different performance profile. This poor showing is primarily due to the runtime itself having extra work to do to ensure that code executes safely, so doing intense computational work may not be the best fit for App Engine Standard.

11.6.4 Cost

As I mentioned in section 5 of this chapter, App Engine Flex has pricing that's almost identical to the pricing for Compute Engine, making it quite reasonable. Because you're paying for Compute Engine instances, the rates themselves are the same, but App Engine (by default) controls the scaling. As a result, you may overprovision, which would lead 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`), you saw that an `F1` instance costs $0.05 per hour, but in that same region, an `n1-standard-1` Compute Engine instance costs slightly less ($0.0475 per hour). Additionally, the Compute Engine instance has 3.75 GB of memory available, whereas the App Engine Standard `F1` instance has only 128 MB. Also, the one 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 comparison is hard to make, though generally it seems that a Compute Engine instance will tend to outperform an equivalently sized App Engine Standard instance.

On the other hand, it's worth noting that App Engine Standard has both a permanent free tier and the ability to scale down to zero (costing no money at all when an application isn't in use), whereas Compute Engine, Kubernetes Engine, and App Engine Flex don't have these advantages. This feature alone makes App Engine Standard a clear winner for toy or hobby applications that don't see a lot of steady traffic.

11.6.5 Overall

Now that you've seen how App Engine compares, let's look at the example applications and see whether it might be a good choice.

11.6.6 To-Do List

The first example application I 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 they completed the tasks. Because this application is unlikely to see a lot of traffic (and is a common getting-started toy project), App Engine Standard might be a great fit, from the perspective of cost. Let's look at how App Engine Standard stacks up (table 11.3).

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 good fit, particularly in the cost category because it can scale down to zero. App Engine Flex, on the other hand, is a bit of overkill in a few areas and not quite a perfect fit when it comes to the cost goal.

11.6.7 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 efficient use of computing resources to avoid overpaying for computing power. Additionally, this application represents a real business that's quite different from a toy project like a to-do list. Table 11.4 shows 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 the ability to scale to zero (because this application is unlikely to ever be without any traffic at all). Although the cost of App Engine Standard is acceptable, App Engine Flex seems like a much better fit. App Engine Flex provides the needed flexibility, performance, and cost, with a reasonable fit when it comes to the learning curve of getting up to speed on using it. Overall, whereas App Engine Standard doesn't quite fit, App Engine Flex would be a fine choice for running the E*Exchange application.

11.6.8 InstaSnap

InstaSnap, the 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 moderate. As a result, finding a good system 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 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 table 11.5, InstaSnap's demands for performance and flexibility (given that it wants to try everything under the sun) rule out App Engine Standard right away. Compare that to App Engine Flex and you see 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, and the learning curve is certainly not a deterrent to adopting App Engine Flex.

> **NOTE** SnapChat 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 the company would have chosen to start there (or Kubernetes Engine).

But the desire to use bleeding-edge features makes something like Kubernetes and Kubernetes Engine a better fit for this project than App Engine Flex. The reason is that Kubernetes is open source, so 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.

Summary

- App Engine is a fully managed cloud computing environment that simplifies the overhead needed for all applications (such as setting up a cache service).
- App Engine has two 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 runtimes, 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 you can configure 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 runtime directly (for example, 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 microservices?
- What is Google Cloud Functions?
- Creating, deploying, updating, triggering, and deleting functions
- Managing function dependencies
- How pricing works for Google Cloud Functions

12.1 What are microservices?

A "microservice 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 microservice). Each microservice can stand on its own, whereas a traditional application has many parts that are intertwined with one another, incapable of running independently.

For example, when creating a typical application, you'd start a project and then start adding controllers to handle the different parts of the application. When building the To-Do List application, you might start by adding the ability to sign up and log in, 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.

Microservices take a hatchet to this design, as shown in figure 12.1, 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, you'd have a microservice responsible for signing up, another for logging in, and others for searching, adding items, creating lists, and so on. In a sense, you can think of this as a very fine-grained, service-oriented architecture (commonly known as SOA).

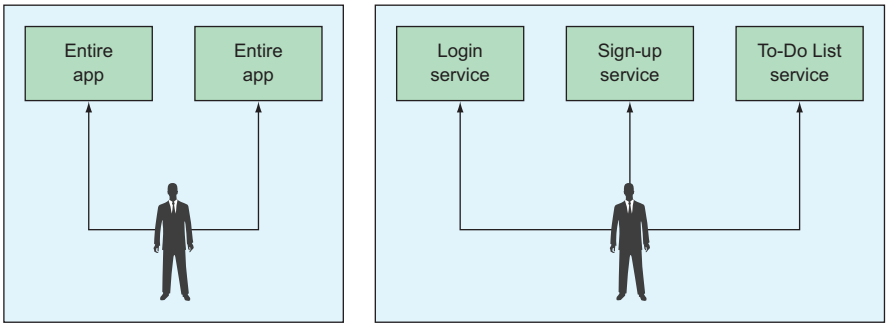


Figure 12.1 Microservice architecture compared to a traditional application

Why would you want to have this type of architecture? What's the benefit over a typical "monolithic" application?

One of the biggest benefits is that each service is only loosely coupled to any other services. Because each microservice can run on its own, development (particularly testing) is narrow and constrained, so 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 straightforward. Further, because each piece must fulfill a contract (for instance, the login service must set a cookie or return a secure login token), the implementation under the hood doesn't matter so long as that contract is fulfilled by the service. What would be major changes in a monolithic application (such as rewriting a piece in a different language) is pretty simple: just rewrite the microservice, and make sure it upholds the same contractual obligations.

Entire books tell of the benefits to using a microservice architecture, so let's jump ahead and look at how Google Cloud Platform makes it easy to design, build, deploy, and run microservices on GCP.

12.2 What is Google Cloud Functions?

As you learned in chapter 9, the first step toward enabling cloud computing has been the abstraction of physical infrastructure in favor of virtual infrastructure. Instead of

worrying about installing and running a physical computer, 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, as shown in figure 12.2. This also happens to fit well with microservice architectures, because the goal there is to design lots of standalone pieces, each responsible for a single part of an application.

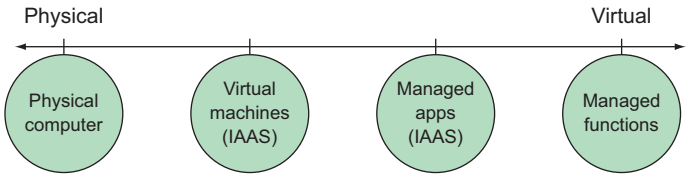


Figure 12.2 The spectrum of computing from physical to virtual

With Cloud Functions, instead of thinking about virtual servers (like Compute Engine), containers (like Kubernetes Engine), or even "applications" (like App Engine), you think only about single functions that run in an entirely serverless environment. 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 microservices that we discussed earlier.

Although 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 (see figure 12.3). In the typical flow of an application, most requests are triggered by users making requests,

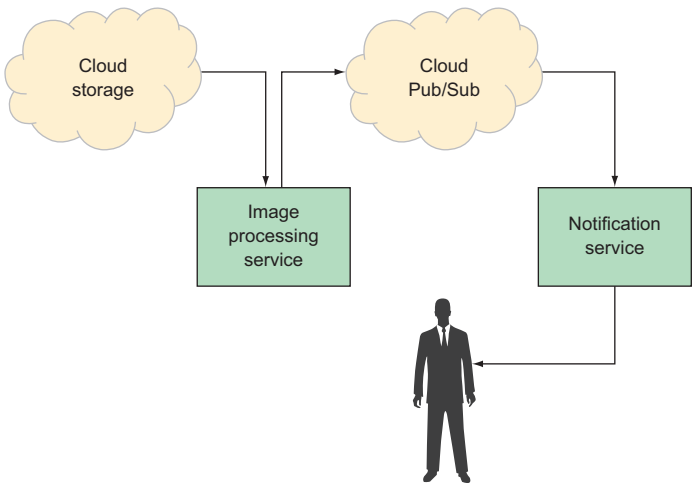


Figure 12.3 Using other cloud services' events as glue between microservices

usually over HTTP (for example, a user somewhere logs in to 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.

All of these events can be monitored by different triggers, which can then run different functions—it's this unique ability to knit different pieces together that makes Cloud Functions so interesting. Cloud Functions allows you to associate small pieces of code to different events and have that code run whenever those events happen. For example, you could hook up a function so that it runs whenever a customer uploads a file into a Cloud Storage bucket, and that function might automatically tag the image with labels from the Cloud Vision API. Now that we've gone through what microservices are and what makes Cloud Functions unique, let's dig into the underlying building blocks needed to do something with Cloud Functions.

12.2.1 Concepts

Cloud Functions is the overarching name for a category of concepts, one of them being a function. But a function 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 go into detail in a moment, but before doing that, let's look at how these different parts fit together, starting from the bottom up (as shown in figure 12.4).

Figure 12.4 Overview of different concepts

Events are things that can happen (for example, 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. Creating a trigger is like saying, "Make sure to run function *X* whenever a new GCS Object is created." Additionally, because 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 microservices together to build complex applications out of lots of simple pieces. See figure 12.5.

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 likely familiar with is an HTTP request, but they can also come from other places such as Google Cloud Storage or Google Cloud Pub/Sub. Every event comes with attributes, which

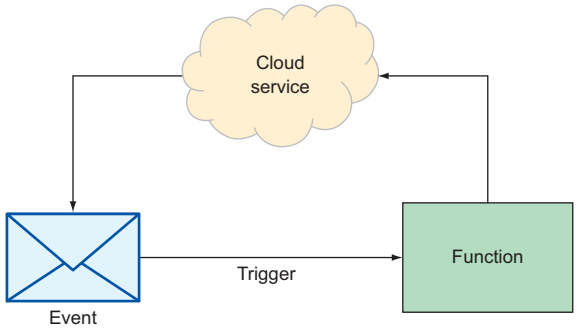


Figure 12.5 Building complex applications out of simple concepts

you can use when building your function, including the basic details of an event (such as a unique ID, the type of the event, and a timestamp of when the event occurred), the resource targeted by the event, and the payload of data specific to the event. Although the data attached to an event depends on the type of the event, common data types are used to minimize the learning curve. For example, HTTP events have an Express Request object in the data field, whereas events from Cloud Storage have the affected Cloud Storage Object in the data field.

Even though events from different sources share quite a bit in common, they fall into two categories. Events based on HTTP requests are synchronous events (the requester is waiting for a response), whereas those coming from other services such as Cloud Pub/Sub are asynchronous (they run in the background). This distinction is important because the code you write to list for synchronous events will be slightly different from that for asynchronous events. 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 write them.

FUNCTIONS

The idea of a microservice architecture is to split different responsibilities of an application into separate services that run on their own. In the world of Cloud Functions, the function itself is the equivalent of a single microservice in an application. A function should be responsible for a single thing and have no problem standing on its own.

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, such as how long to run before timing out and returning an error (defaulting to one minute, but configurable up to nine minutes) and the amount of memory to allocate for a given request (defaulting to 256 MB).

The key part of any Cloud Function is the code that you're able to write. Google Cloud Functions lets you write these functions in JavaScript, but depending on whether you're dealing with a synchronous event (an HTTP request) or an asynchronous event

(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) => {
  if (req.get('content-type') !== 'text/plain') {
    res.status(400).send('I only know how to echo text!');
  } else {
    res.status(200).send(req.body);
  }
};
```

This function is named echoText and mapped to the same name when exported.

Here you can read the request header for content type and show an error for non–plain text requests.

If the request was plain text, you can echo the body back in the response.

If you're at all familiar with web development in JavaScript, this function shouldn't be a surprise. If you're not, the idea 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 is closed and the request considered completed.

What 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, shown in the following listing.

Listing 12.2 A Cloud Function that logs a message from Cloud Pub/Sub

```
exports.logPubSubMessage = (event, callback) => {
  const msg = event.data;
  console.log('Got message ID', msg.messageId);
  callback();
};
```

Background functions are provided with an event and a callback rather than a request and a response.

The Pub/Sub message itself is stored in the event data.

Just like a regular message, the event ID is attached and accessible.

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 you can read from and do things with, and when you're done, you call the `callback` provided. The obvious question 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 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. You use triggers to 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. 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 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, you write the function itself in JavaScript. After that, you deploy it to Google Cloud Functions, and in the process, you'll define what exactly triggers it (such as HTTP requests, Pub/Sub messages, or Cloud Storage notifications). Then you'll verify that everything works by making some test calls and then some live calls. You'll 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 toward working with Cloud Functions is to write your function. Because this will be a synchronous function (rather than a background function), you'll write it in the request and response style as you saw earlier. Start by creating a new directory called echo and, in that directory, a new file called index.js. Then 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;
  } elseif (contentType == 'application/json') {
    responseContent.echo = req.body.data;
  } else {
    responseContent.echo = JSON.stringify(req.body);
  }

  res.status(200).send(responseContent);
};
```

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. You now have a function on your local file system, but you have to get it in the cloud. Let's move along and look at how to deploy your function.

12.3.2 Deploying a function

Deploying a function you wrote locally is the one step of the process where you'll need to do a little setup. More specifically, you'll need a Cloud Storage bucket, which is where the content of your functions will live. Additionally, if you haven't already, you'll need to enable the Cloud Functions API in your project. Start with enabling the Cloud Functions API. To do this, navigate to the Cloud Console and enter 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 (shown in figure 12.6).

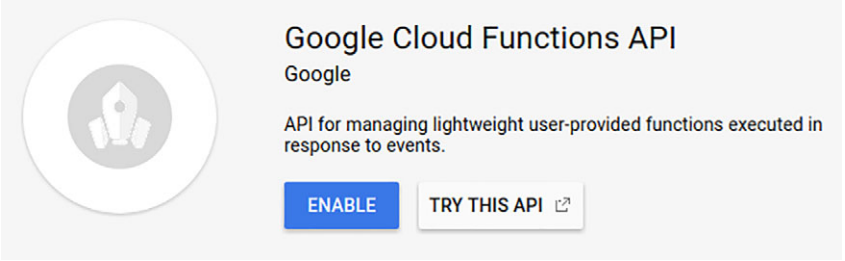


Figure 12.6 Enable the Cloud Functions API

Next, you need to create your bucket. For this example, you'll use the Cloud Console. Start by navigating to the Cloud Console and choose Storage from the left-side navigation. A list of buckets you already have appears. To create a new one, click the Create bucket button. In this example, as shown in figure 12.7, you'll leave the bucket as multiregional in the United States (take a look at chapter 8 for more details on these options).

After you have a bucket to hold your functions, you'll use the `gcloud` tool to deploy your function from the parent directory, as the next listing shows.

Listing 12.4 Command to deploy your new function

```
$ tree
.
└── echo
    └── index.js

1 directory, 1 file
$ gcloud beta functions deploy echo --source=./echo/ \
  --trigger-http --stage-bucket=my-cloud-functions
```

Your directory tree should show the echo directory with your index.js file living inside.

Make sure to change the bucket name to match your bucket name.

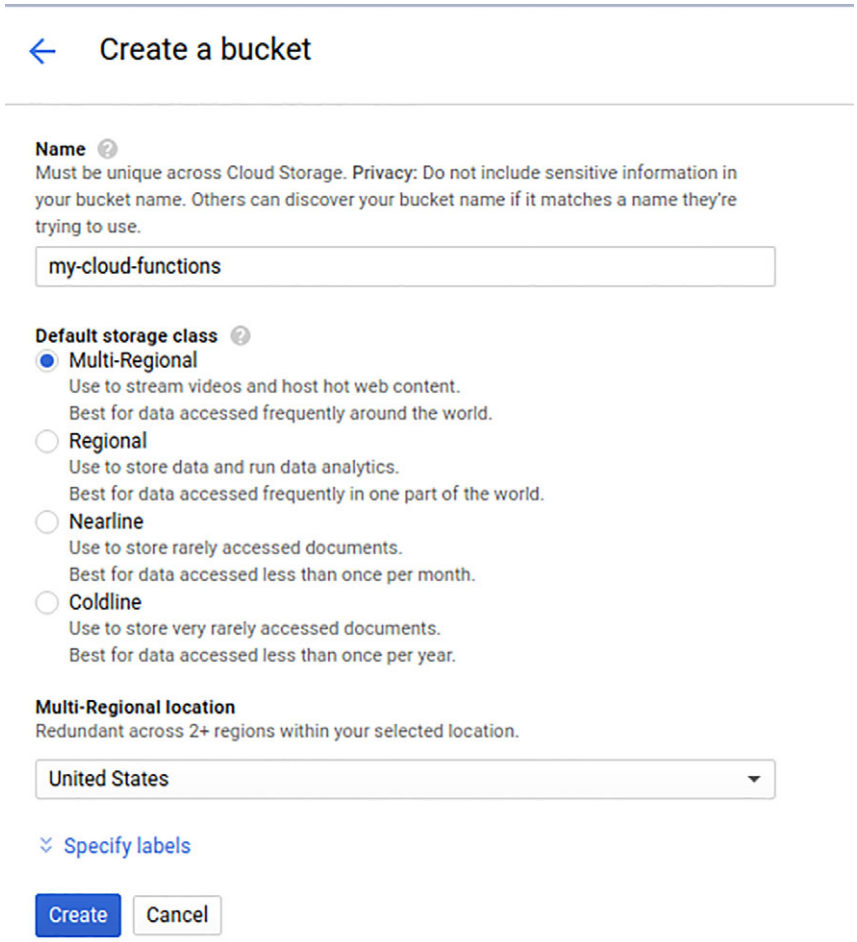


Figure 12.7 Create a new bucket for your cloud function

This command tells Cloud Functions to create a new function handle called `echo` from the file that you noted in echo/index.js and from the function that you exported (which was called `echo`). This also says to trigger the function from HTTP requests and to put the function itself into your staging bucket.

After running this function, you should see the following output:

```
$ 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/ampnLWNsb3VkLXJlc2VhcmNoL3VzLWNlbnRyYWwxL2VjaG8vaVFZMTM5bk9jcUk
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 the functions you have locally and uploading them to your Cloud Storage bucket. After 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 (https://us-central1-your-project-id-here.cloudfunctions.net/echo). Let's take your new function out for a spin.

12.3.3 Triggering a function

Your newly deployed Cloud Function is triggered via HTTP, so it comes with a friendly URL to trigger the function. Try that out using `curl` in the command line, as shown in the next listing.

Listing 12.5 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 ran and returned what you expected! But what about functions triggered by something besides HTTP? It would be a pain if you had to do the thing that would trigger the event (such as 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 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, execute the same thing using `gcloud` next.

Listing 12.6 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!"}'
```

Now you have a 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 to build more complicated (and full-featured) applications with your functions, starting with updating an existing function.

12.4 Advanced concepts

Although the section happens to be called "advanced" concepts, most of these are 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 as redeploying. For example, tweak your `echo` function from earlier by adding a second parameter in the response content, just to show that you made a change. Inside your `echo` function, start off `responseContent` with an extra field, as shown in the following listing.

Listing 12.7 Adding a new parameter to your response content

```
let responseContent = {
  from: 'Cloud Functions',
  version: 1
};
```

If you were to redeploy this function and then call it again, you should see the modified response, shown in the next listing.

Listing 12.8 Redeploying 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/ampnLWNsb3VkLXJlc2VhcmNoL3VzLWNlbnRyYWwx
➥   L2VjaG8vUDB2SUM2dzhDeG8
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-afkbzeghcygu.zip
status: READY
timeout: 60s
updateTime: '2017-05-22T22:17:27Z'

$ gcloud beta functions call echo --data='{"data": "Test!"}'
executionId: nwluxpwmef9l
result: '{"from":"Cloud Functions","version":1,
➥   "echo":"Test!"}'
```

As you can see, the new parameter (version) is returned after redeploying.

Note also that if you were to list the items in your Cloud Functions bucket (in the example, `my-cloud-functions`), you'd see the previously deployed functions. Now you have a safe backup for your deployments in case you ever accidentally deploy the wrong one. Now that you've seen how to update (redeploy) 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. You may find yourself needing to delete a function you'd previously deployed. This is easily done using the `gcloud` tool, shown next, deleting the `echo` function that you built previously.

Listing 12.9 Deleting your `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, think of this as deregistering 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 your `echo` function, 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

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 via the Node Package Manager (NPM). It would be annoying if you had to download and redeploy duplicates of these packages to run your function. Let's see how Cloud Functions deals with these types of dependencies.

Imagine that in your `echo` function you wanted to include the Moment JavaScript library so that you can properly format dates, times, and durations. When developing your typical application, you'd use `npm` to do this and maintain the packaging details by running `npm install --save moment`. But what do you do with Cloud Functions? You can use those same tools to ensure your dependencies are handled properly. To

see this in action, start by initializing your package (using `npm init`) and then installing Moment inside the echo directory that you created previously.

Listing 12.10 Initializing your 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:

```
{
  "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 your package is ready, modify your `echo` function to also say how much time has passed since Christmas in 2016, as shown in the following listing.

Listing 12.11 Using the dependency on Moment.js

```
const moment = require('moment');

exports.echo = (req, res) => {
  let now = moment();
  let christmas2016 = moment('2016-12-25');

  let responseContent = {
    from: 'Cloud Functions',
    christmas2016: moment.duration(christmas2016 - now).humanize(true)
  };

  let contentType = req.get('content-type');

  if (contentType == 'text/plain') {
    responseContent.echo = req.body;
  } elseif (contentType == 'application/json') {
    responseContent.echo = req.body.data;
  } else {
    responseContent.echo = JSON.stringify(req.body);
  }

  res.status(200).send(responseContent);
};
```

Start by requiring the dependency as you always would.

Use the library as you would in a typical application.

Calculate the humanized difference between now and Christmas 2016.

When this is done, redeploy the function with the new code and dependencies, as follows.

Listing 12.12 Redeploying your 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 you have successfully uses the Moment package to say that (as of this writing), Christmas 2016 was five months ago! Now that you can see that using other libraries works 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

Applications are rarely completely stateless (they have no need to store any data). As you can imagine, 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's a great time to do that. If you're not interested in the particulars of Spanner but want to follow along with an example showing how to talk to another Cloud API, that's fine, too. To demonstrate reading and writing data from Spanner, start by creating an instance, a database, and then a table. For the first two, take a look at chapter 6 on Cloud Spanner. For the table, 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 is to install (and add to your dependencies) a library to generate UUID values (`uuid`) and the Google Cloud Spanner Client for Node.js (`@google-cloud/spanner`). You can install these easily using `npm`, as shown in the next listing.

Listing 12.13 Install (and add dependencies for) the Spanner client library and UUID

```
$ npm install --save uuid @google-cloud/spanner
```

After those are installed, you'll update your code, making two key changes. First, whenever you echo something, you'll log the content to Cloud Spanner by creating a new row in the `logs` table. Second, in each echo response, you'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 following code does this, while still pinning to the `echo` function.

Listing 12.14 Your new Spanner-integrated `echo` function

```
const uuid4 = require('uuid/v4');
const Spanner = require('@google-cloud/spanner');

const spanner = Spanner();
```

Start by importing your two new dependencies.

This call creates a new Spanner client.

```
const getDatabase = () => {
  const instance = spanner.instance('my-instance');
  return instance.database('my-db');
};

const createLogEntry = (data) => {
  const table = getDatabase().table('logs');
  let row = {log_id: uuid4(), log_data: data};
  return table.insert(row);
};

const countLogEntries = () => {
  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) => {
  let contentType = req.get('content-type');
  if (contentType == 'text/plain') {
    return req.body;
  } elseif (contentType == 'application/json') {
    return req.body.data;
  } else {
    returnJSON.stringify(req.body);
  }
};

exports.echo = (req, res) => {
  let body = getBodyAsString(req);
  returnPromise.all([
    createLogEntry('Echoing: ' + body),
    countLogEntries()
  ]).then((data) => {
    res.status(200).send({ echo: body, logRowCount: data[1] });
  });
};
```

getDatabase returns a handle to the Cloud Spanner database. Make sure to update these IDs to the IDs for your instance and database.

createLogEntry is the function that logs the request data to a new row in the logs table.

Here you use the UUID library to generate a new ID for the row.

countLogEntries executes a query against your database to count the number of rows in the logs table.

getBodyAsString is a helper function of the logic you used to have in your old echo function, to retrieve what should be echoed back.

Because these two promises are independent (one adds a new row, another counts the number of rows), you 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 as planned, as the next listing shows

Listing 12.15 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

Deploying a function that you've declared locally involves using the Cloud SDK (`gcloud`) to package 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, like a slimmed-down version of what's offered by GitHub, Bitbucket, or GitLab. They're also a place where you can store the code for your Cloud Functions. To see how these work, migrate your `echo` function from a local file into a hosted source repository and then redeploy from there. The first thing you do is create a new repository from the Cloud Console by choosing Source Repositories from the left-side navigation (toward the bottom under the Tools section). From the list of existing repositories (which should include a default repository), click the Create Repository button. When prompted for a name, call this repository "echo." See figure 12.8.

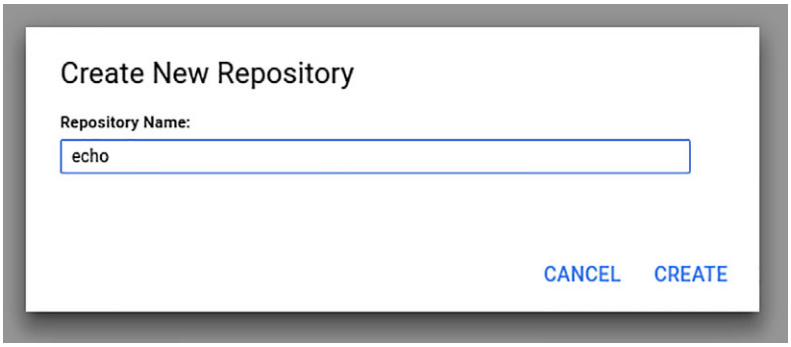


Figure 12.8 Create a new source repository

After you create the new repository, you'll see a few ways to configure the empty repository, including the full URL that points to the newly created repository (something like https://source.developers.google.com/projects/your-project-id-here/repos/echo). Helpers for common providers exist (such as mirroring a repository from GitHub), but to get started, clone your newly created (and empty) repository into the directory with your function and its dependencies. First, initialize your directory as a new Git repository. After that, configure some helpers to make sure authentication is handled

by the Cloud SDK. Finally, add a new remote endpoint to your Git repository. After you have that set up, you can push to the remote like any other Git repository, as shown in the following listing.

Listing 12.16 Initializing a new source repository with your code

```
$ git init
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

$ git config credential.helper gcloud.sh

$ git add index.js package.json
$ 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
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
```

Start by initializing the current directory as a local Git repository.

Add the new source repository's URL as a Git remote location.

Using Git's configuration, tell it to use the Cloud SDK for authentication when interacting with the remote repository.

Add and commit your files to the Git repository.

Finally, push all of your local changes to the new google remote that you 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, as shown in figure 12.9.

The code for your function is officially stored on a Cloud Source Repository, which means that if you wanted to redeploy it, you could use this repository as the source. You can use the Cloud SDK (`gcloud`) once again but with slightly different parameters.

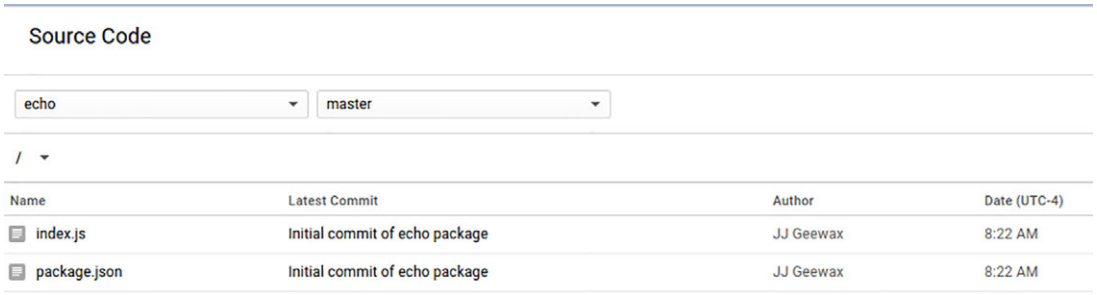


Figure 12.9 Your newly pushed source repository

Listing 12.17 Deploying from the source repository

```
$ gcloud beta functions deploy echo \
>   --source=https://source.developers.google.com/
    ➥ projects/your-project-id-here/repos/echo \
>   --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/ampnLWNsb3VkLXJlc2VhcmNoL3VzLWNl
    ➥ bnRyYWwxL2VjaG8vendQSGFSVFR2Um8
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"}'
```

Make sure you substitute your own project ID in this URL.

And that's it—you've redeployed 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 costs.

12.5 Understanding pricing

Following on the tradition of using Google Cloud Platform, Cloud Functions only charges only for what you use, and in this case it's incredibly granular. Unlike some of the other products, several different aspects 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 is also the most straightforward: the number of invocations (for example, requests) sent to your function. This number is measured in millions of requests and is currently billed at $0.40 per million, meaning each request costs $0.0000004 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. Outbound traffic, however, costs $0.12 per GB. 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 to make things look a bit more like Compute Engine (for more on

GCE, see chapter 9). You may remember that when you deploy your function, an extra parameter controls how much memory is given to the function for each request. The amount of memory you specify also determines the amount of CPU capacity provided to your function. You effectively have five different computing profiles to choose from, each with a different overall cost. See table 12.1.

Table 12.1 Cost of 1 million requests, 100 ms per request

Memory	CPU	Price of 1 million requests, 100 ms 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 12.18 Formula for calculating the cost of 1 million 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 GHZ * 0.0000100 + 0.0000025 * 0.128 GB) = $ 0.232`. Now 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 100 ms.

If things weren't confusing and complicated enough, Cloud Functions comes with a perpetual free tier, which means that some chunks of the resources you use are completely free. With Cloud Functions, 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*—5 GB of egress traffic per month

Summary

- Microservices allow you to build applications in separate standalone pieces of functionality.
- Cloud Functions is one way to deploy and run microservices 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 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 startup

DNS is a hierarchical distributed storage system that tracks 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 stores a set of "resource records," which are the mappings from names to numbers, and splits these records across a hierarchy of "zones." These zones provide a way to delegate responsibility for owning and updating subsets of records. For example, 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 that allow for ownership over certain groups of records from someone higher up the food chain to someone lower. In a sense, this is like each company in the Yellow Pages being responsible for what shows up inside their individual box in the phone book. The publisher of the 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 its box with whatever content it wants.

Because DNS is a distributed system and expected to be only eventually consistent (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 can 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. As you might guess, running your own DNS server is a bit of a pain and falls in the category of problems that Cloud services are best at fixing, which brings us to Google Cloud DNS.

13.1 What is Cloud DNS?

Google Cloud DNS is a managed service that acts as a DNS server and can answer DNS queries 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 is to expose an API that 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 capability is important because BIND, although battle-tested over the years and proven to be quite reliable, is somewhat inconvenient to run and maintain and doesn't support a modern API to make changes to DNS records. Instead, updating records involves modifying files on the machine running the BIND service, followed by reloading the contents into the process's memory.

How does Cloud DNS work? To start, 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, similar to in a true DNS server like BIND. See figure 13.1.

Each zone contains a collection of record sets, and each record set contains a collection of records. These records are where the useful data is stored, whereas the other resources are focused on categorization of this data. See figure 13.2.

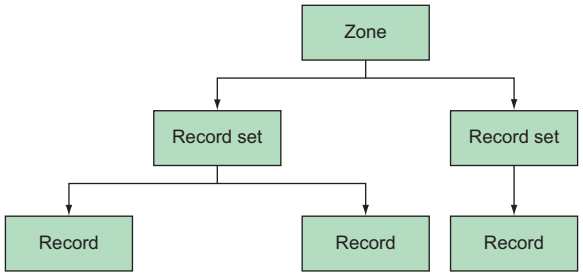


Figure 13.1 Hierarchy of Cloud DNS concepts

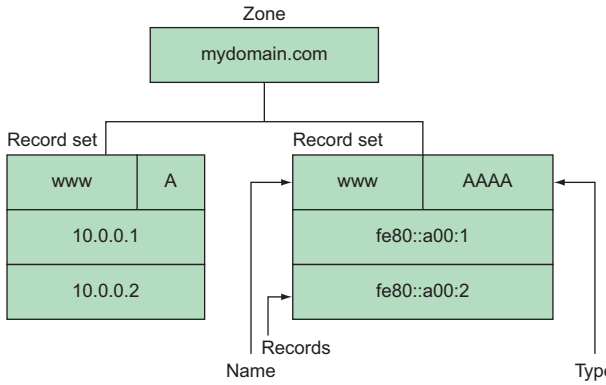


Figure 13.2 Example hierarchy of DNS records

Where a zone is defined by nothing more than a name (e.g., `mydomain.com`), a record set stores 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. We have the ability to store multiple records for a single given subdomain and type. For example, this structure allows you to store several IP addresses for `www.mydomain.com` by setting multiple records in a record set of type `A`—similar to having multiple phone numbers listed for your business in the phone book (see figure 13.3).

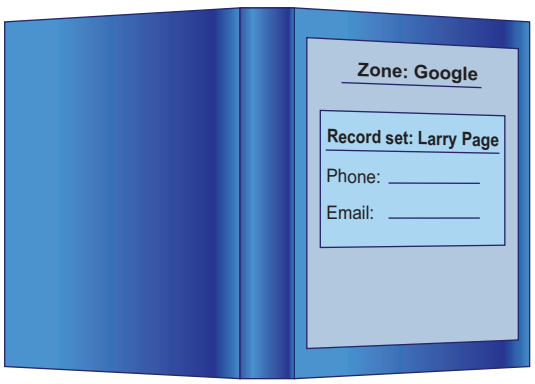


Figure 13.3 DNS records as a phone book

Using the phone book analogy once again, a zone is like the section delegated to a company that was described earlier (for example, Google, Inc.), a record set is equivalent to a single person working at the company (for example, Larry Page), and each record is a different contact method for the person (for example, two phone numbers, an email address, and a physical address).

13.1.1 Example DNS entries

Let's look at an example domain, `mydomain.com`, containing some sample records. 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 table 13.1, the domain has three distinct subdomains—`ns1`, `docs`, and `www`—each entry with at least one record.

Table 13.1 DNS entries by record set

Zone	Subdomain	Record set	Record
mydomain.com	ns1	A	10.0.0.1
	www	A	10.0.0.1
			10.0.0.2
		docs	CNAME

In a regular DNS server like BIND, you manage these as "zone files," which are text files stating in a special format the exact DNS records. The next example 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.
```

Exposing an API to update these remotely and then reloading the DNS server is a nontrivial amount of work, which is even more difficult if you want it to be always available.

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 at how this works next.

13.2 Interacting with Cloud DNS

Cloud DNS is 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 earlier. To begin, we have to enable the Cloud DNS API. In the Cloud Console, type "Cloud DNS API" in the search box at the top. You should see one result in the list. After clicking that, you should land on a page with an Enable button, shown in figure 13.4. Click that and you are good to go.

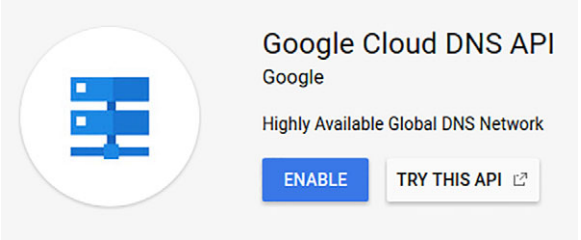


Figure 13.4 Enable the Cloud DNS API

Now that the API is enabled, let's continue using the UI to work with Cloud.

13.2.1 Using the Cloud Console

Let's start our exploration of Cloud DNS by creating a zone. To do this, in the left-side navigation select Network services in the Networking section. As shown in figure 13.5, a Cloud DNS item appears and will take you to the UI for Cloud DNS. This page allows you to manage your zones and records for Cloud DNS. To start, let's create the zone for `mydomain.com`.

How are we going to control the DNS records for a domain that we clearly don't own (because `mydomain.com` is taken)? Remember the concept of delegation that we described earlier? 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. You do this at the domain registrar level, where you can set which name server to use for a domain that you currently own.

Because we definitely don't own `mydomain.com`, what we're doing now is like writing up an advertisement for the plumber in the Yellow Pages. Instead of sending it to the phone book to publish, we'll glue it into the phone book, meaning it'll only be seen by us. You 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 to make it official.

Clicking Create Zone opens a form where you enter three different values: a unique ID for the zone, the domain name, and an optional description. See figure 13.6.

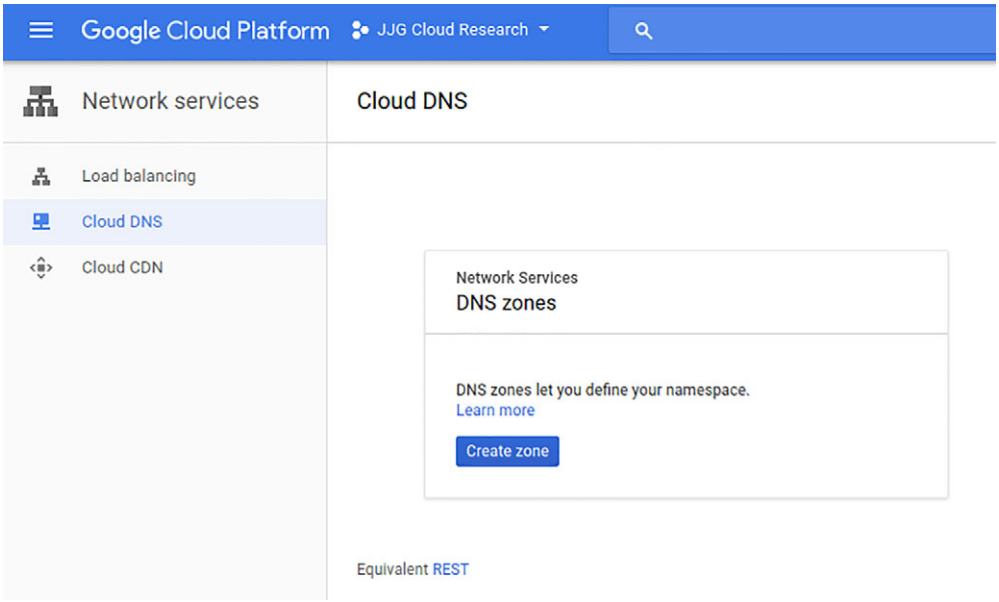


Figure 13.5 Managing Cloud DNS entries from the UI

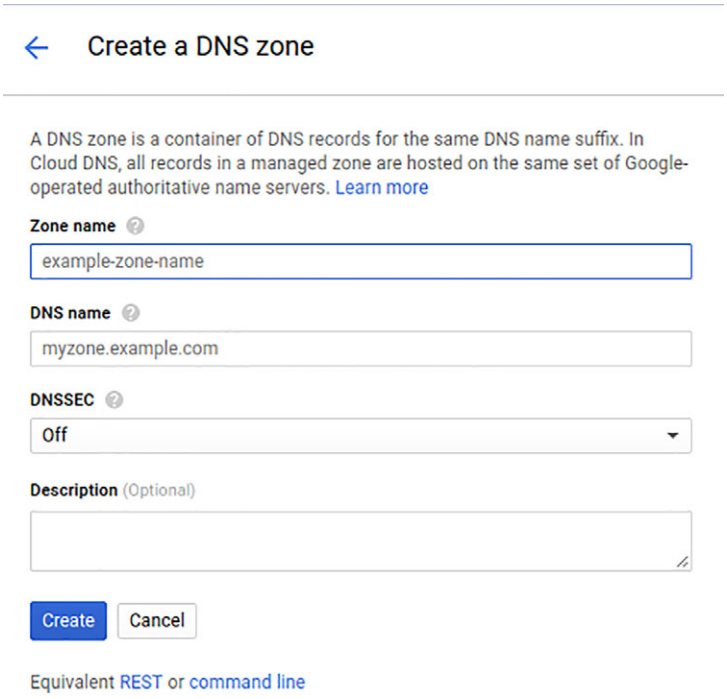


Figure 13.6 Form to create a new zone

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, they serve different purposes. The zone name is a unique ID inside Google Cloud that 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 subgroup of records for which this zone acts as a delegate. In our example, the DNS name will be `mydomain.com`, which indicates that this zone will be responsible for every subdomain of `mydomain.com` (such as `www.mydomain.com` or `anything.else.mydomain.com`). To create the example zone I described, let's use `mydomain-dot-com` as the zone name and `mydomain.com` as the DNS name, as shown in figure 13.7.

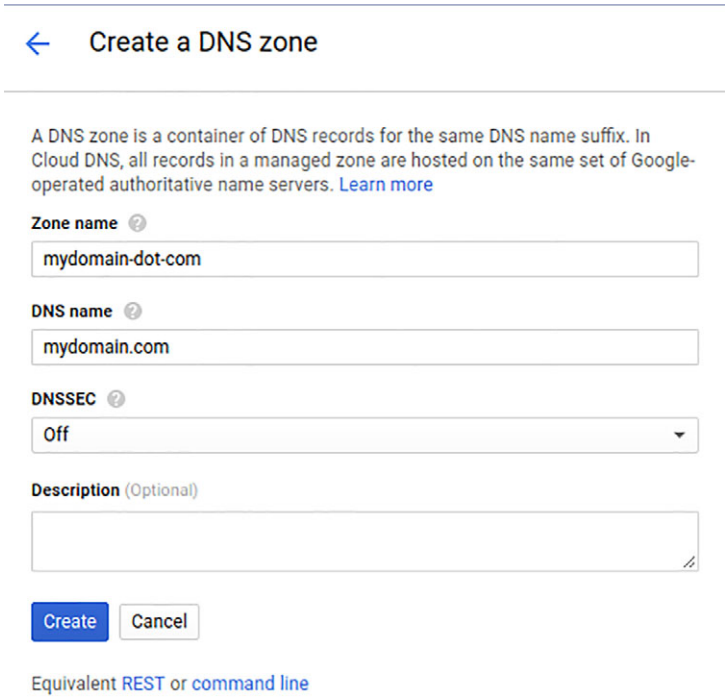


Figure 13.7 Creating our example zone

After you click Create, a screen that lets you manage the records for the zone opens. You may be surprised to see some record sets already in the list! Don't worry—these records are the default (and necessary) `NS` records that state that no further delegations of zones exists and anything inside `mydomain.com` should be handled by Google Cloud DNS name servers (for example, `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. A form opens where you'll enter the DNS name of the record set (for example, `demo.mydomain.com`), as

well as a list of records (for example, an `A` record of `192.168.0.1`), shown in figure 13.8. To add more records to the set, click Add Item.

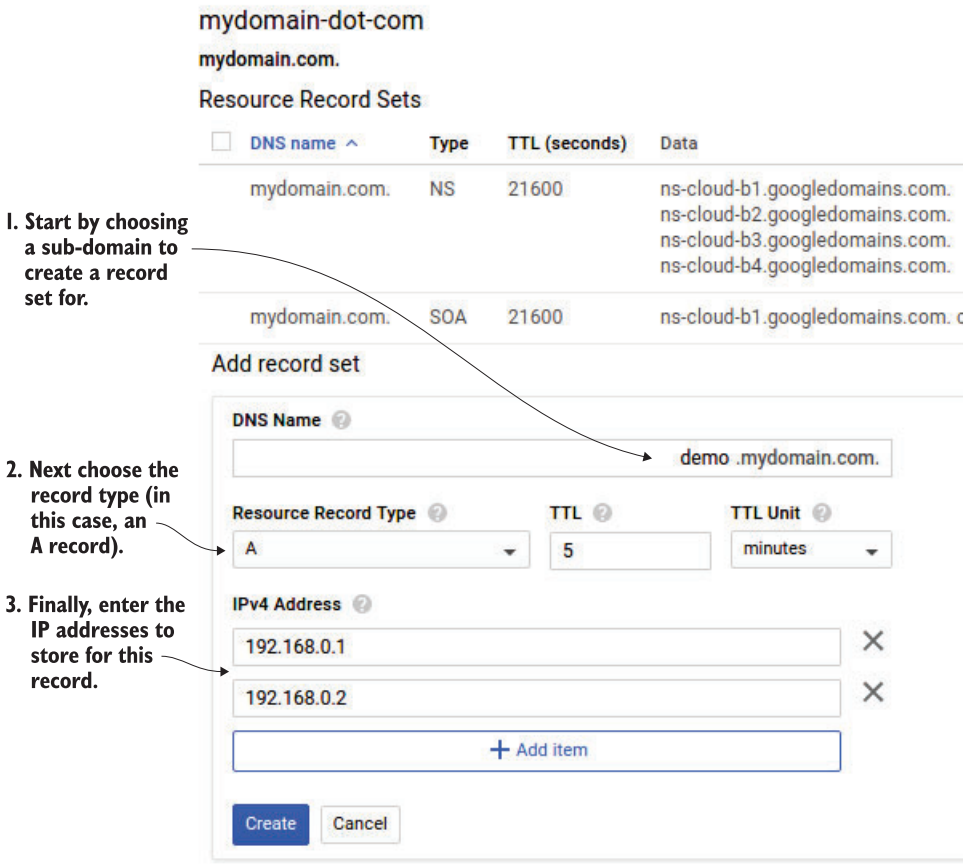


Figure 13.8 Add `demo.mydomain.com` `A` records

When you click Create, the records are added to the list. To check whether it worked, we can make a regular DNS query for `demo.mydomain.com`. We need to specify during the lookup, however, that we are interested only in "our version" of this DNS record, so 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. We will use the Linux terminal utility called `dig`, aimed at a specific DNS server.

Listing 13.2 Asking Google Cloud DNS for the records we added

```
$ dig demo.mydomain.com @ns-cloud-b1.googledomains.com

# ... 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
```

Make sure to use the right DNS server here. In this example, it's ns-cloud-b1.googledomains.com, but it could be something else for your project (for example, 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 "ANSWER" 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:

```
$ 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
```

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 (for example, `ns-cloud-b1.googledomains.com`). Now let's move on to accessing this API from inside Node.js so we can benefit from the purpose of Cloud DNS.

13.2.2 Using the Node.js client

Before you get started writing some code to talk to Cloud DNS, you'll first need to install the Cloud DNS client library by running `npm install @google-cloud/dns@0.6.1`. Next we explore how the Cloud DNS API works under the hood. Unlike some other APIs, the way we update records on DNS entries is by 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 (for example, 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 problematic. We'll create a few records and then use `zone.createChange` to apply changes to a zone, shown next.

Listing 13.3 Adding new records to our zone

```
const dns = require('@google-cloud/dns')({
  projectId: 'your-project-id'
});
const zone = dns.zone('mydomain-dot-com');

const addRecords = [
  zone.record('a', {
    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) => {
  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);
});
```

Start by creating a Zone object using the unique name (not DNS name) from before in the Cloud Console.

Here we create a list of the records we're going to add.

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.

Here we use the zone.createChange method to apply a mutation that adds our records defined earlier.

If you run this snippet, you should see output looking something like this:

```
> Change created at 2017-02-15T10:57:26.139Z as Change ID 6
Change status is currently pending
```

That the change is 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 whether these new records have been applied in the UI by refreshing the page, which should show our new records in the list, as shown in figure 13.9.

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

Figure 13.9 Newly added records in the Cloud DNS UI

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` subcommand. For example, let's look at the newly updated list of our DNS records for the `mydomain-dot-com` zone. As mentioned earlier, 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 13.4 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 tool can be incredibly handy if you happen to have an existing BIND server that you want to move to Cloud DNS, using the `gcloud dns` subcommand'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 next. Notice that I've changed a few of the addresses involved, but the record names are all the same (`ns1`, `www`, and `docs`).

Listing 13.5 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 13.6 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, shown in the following listing.

Listing 13.7 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
```

As before we can check the status either by looking in the UI or using the `gcloud` command to "describe" the change, shown next.

Listing 13.8 Viewing the status of our DNS change

```
$ gcloud dns record-sets changes describe 8 --zone mydomain-dot-com | grep
     status
status: done
```

Because this reports that our change has been applied, we can now look at our updated records with the `record-sets list` directive.

Listing 13.9 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  10.0.0.91,10.0.0.92s
```

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 cost.

13.3 Understanding pricing

As with most things in Google Cloud, Cloud DNS charges only 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.

Although the pricing table is tiered, at most you'll end up paying 20 cents per managed zone per month and 40 cents per million queries per month. As you create more zones and more queries, the per-unit prices go down dramatically. For example, although your first billion queries will be billed at 40 cents per million, after that queries are billed at 20 cents per million. Further, though 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 startup business' DNS hosting.

13.3.1 Personal DNS hosting

In a typical personal configuration, you see no more than 10 different domains being managed. It would 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. See table 13.2 for a pricing summary.

> **NOTE** The *unique* part is important because it's likely that other DNS servers will cache the results, meaning a DNS query usually happens only 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
Total			$6.00 per month

So what about a more "professional" situation, such as a startup that needs DNS records for various VMs and other services?

13.3.2 Startup business DNS hosting

In a typical startup, it's common to have 20 different domains floating around to cover issues like separating user-provided content from the main service domain, vanity domain redirects, and so on. In addition, the traffic to the various domains may have several unique users and shorter TTL values to allow modifications to propagate more quickly, resulting in more overall DNS queries. In this situation it's possible to have more than 50 million monthly DNS queries to handle. Let's estimate that this will be 20 zones and 50 million DNS queries per month. See table 13.3 for a pricing summary.

Table 13.3 Startup business 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
Total			$24.00 per month

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 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 we can access them using a custom domain name.

13.4 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. 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 a public-facing IP address (e.g.,`104.14.10.29`), and other times it's a special DNS name (for example, `ec2-174-32-55-23.compute-1.amazonaws.com`).

Both of those examples are not all that pretty and are definitely difficult to remember. Wouldn't it be nice if we could talk to new servers with a name that was part of our domain (for example, `mydomain.com`)? For example, new web servers would automatically turn on and register themselves as something like `web7-uc1a.mydomain.com`. As you have 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:

- The instance name (for example, `instance-4`)
- The Compute Engine zone (for example, `us-central1-a`)
- The public-facing IP address (for example, `104.197.171.58`)

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 13.10 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 you try running your helper method (`getInstanceDetails()`) from a running GCE instance, you should see output looking something like the following:

```
> getInstanceDetails().then(console.log);
Promise { <pending> }
> { name: 'instance-4',
  zone: 'us-central1-f',
  ip: '104.197.171.58' }
```

Now let's write a quick startup script that uses this metadata to automatically register a friendly domain name.

Listing 13.11 Startup 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:

```
Change created at 2017-02-17T11:38:04.829Z as Change ID 13
Change status is currently pending
```

You can then verify that the change was applied using the `getRecords()` method.

Listing 13.12 Listing all DNS records for your 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 {                          <-- Here you can see that your record was applied properly.
    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] } ] ]
```

Finally, you should verify that this worked from the perspective of a DNS consumer. To do that, you use the `dig` command like you did earlier, specifically checking for your record. Note that you can do this from any computer (and it might be best to test this from outside your GCE VM, because the goal is to be able to find your VM easily from the outside world).

Listing 13.13 Viewing your newly created (nonauthoritative) 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 previously, these records won't be authoritative until the registrar specifically points to Cloud DNS as the name server, 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 work automatically. When that's done, you can use the code shown in listing 13.13 as a startup script for your VMs, and they will register themselves in Cloud DNS once the boot process is completed.

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 against which we can program.
- Cloud DNS charges prices based on the number of zones (domain names) and the number of DNS lookup requests.

Part 4

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 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 (such as identifying things in photographs or translating text between languages). We'll finish 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

For humans, image recognition 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 find it tough to explain clearly what exactly constitutes that understanding.

It's difficult to get a computer to recognize an image. Things that are hard to define are typically tricky to express as code, and understanding an image falls in that category. As with many definition problems, we get around this by choosing a 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, as shown in figure 14.1, 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 aren't limited to *things* like "dog" but can be other attributes, such as colors like "green." Often, the complexity is subtle and can be frustrating. For example, humans easily recognize a mirror, but because a mirror shows itself by duplicating whatever else is in the picture, 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. Millions of people couldn't decide on the color of a dress by looking at a picture. This shows two things: image recognition is super complicated and, therefore, somewhat amazing, and image recognition is not 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. Let's look at how you can use the Cloud Vision API to start recognizing (or annotating) images.

14.1 Annotating images

The general flow for annotating images is a simple request-response pattern (see figure 14.2), where you send an image along with the desired annotations you're interested in to the Cloud Vision API, and the API sends 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 send your image and get back some details about it.

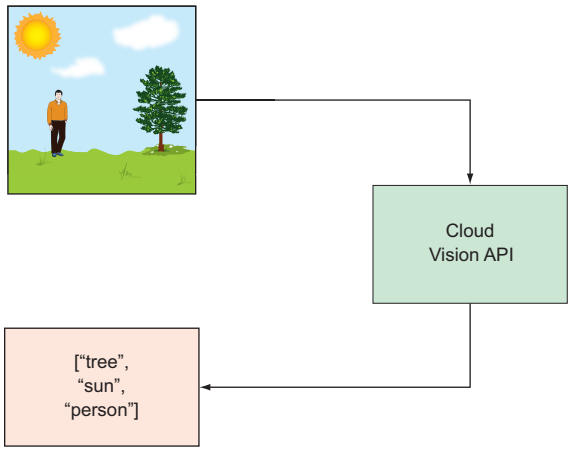


Figure 14.2 Request-response flow for Cloud Vision

Because there's no state to maintain, specify which annotation types you're interested in, and the result will be limited to those. You can specify details for each type of annotation, but we'll explore these one at a time. Because we've already given a few examples of label annotations, let's start there.

14.1.1 Label annotations

Labels are a quick textual description of a concept that Cloud Vision recognized in the image. As you learned, 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. 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 your image. You'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 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 shown in figure 14.3. Make sure to choose Service Account Actor as the role to limit what this particular service account can do. Also, don't forget to

acquire a new private key (by checking the box). After you click Create, the download automatically starts with a .json file. It's this file that we'll refer to as key.json in the following examples.

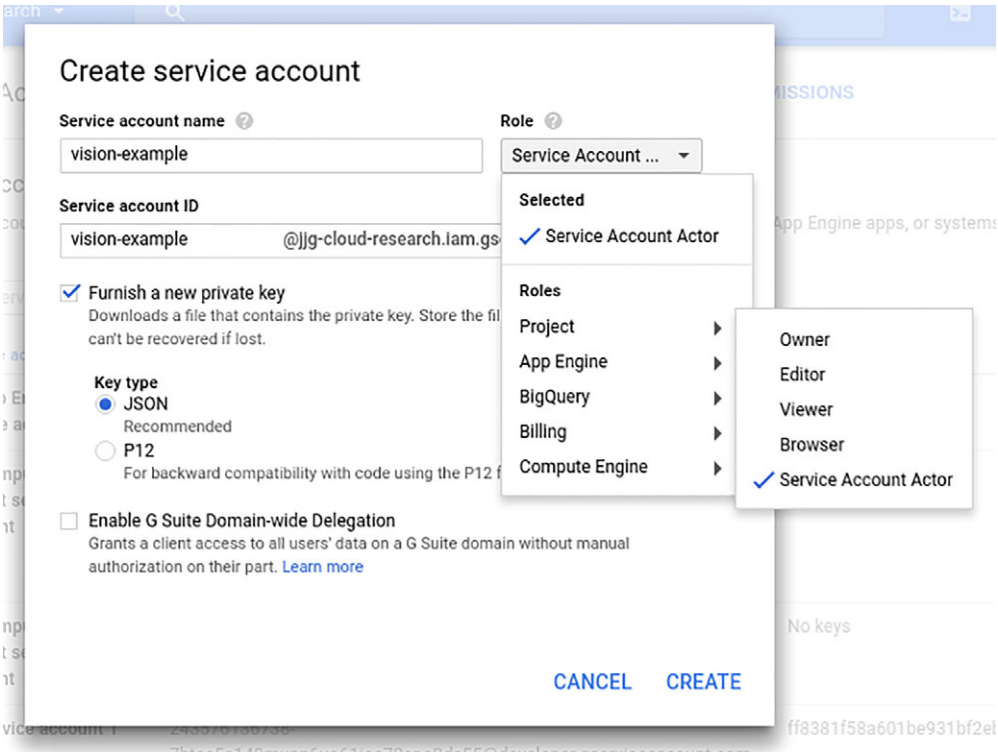


Figure 14.3 Create a new service account.

After you have your key file, you need to install the client library. To do this, run `npm install @google-cloud/vision@0.11.5` to pull down a specific version of the Node.js client library. Next, you'll need to enable the Cloud Vision API using the Cloud Console. To do this, in the search bar at the top of the Cloud Console, enter Cloud Vision API. You'll see a single result. Select that result, then click Enable on the next page (see figure 14.4), and you're good to go.

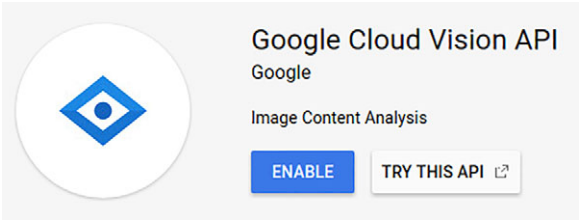


Figure 14.4 Enable the Cloud Vision API.

Now that that's done, you can move on to recognizing an image. In this example, you'll use the dog image from earlier, 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',
  keyFilename: 'key.json'
});

vision.detectLabels('dog.jpg').then((data) => {
  console.log('labels: ', data[0].join(', '));
});
```

Make sure to specify your project ID here.

In this case, you'll need to point to the service account key file that you downloaded before.

Use the detectLabels method to get label annotations on the image.

If you run this, you should see the following output:

```
> 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 use only labels that have a certain confidence level? What if you wanted to ask Cloud Vision, "Show me only 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 in the next listing.

Listing 14.2 Enabling 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) => {
  const labels = data[0];
  labels.forEach((label) => {
    console.log(label);
  });
});
```

Notice the {verbose: true} modifier in the detectLabels call.

Go through each label (which is an object), and print it out.

When you run this code, you should see something more detailed than the label value, as the following listing shows.

Listing 14.3 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/039ndd', score: 69.060057 }
{ desc: 'dog like mammal', mid: '/m/01z5f', score: 68.510407 }
```

These label values are the same, but they also include two extra fields: `mid` and `score`. The `mid` value is an opaque ID for the label that 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 things with only 75% confidence or above, your code to do this might look something like the next listing.

Listing 14.4 Show only 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; })
    .map((label) => { return label.desc; });
  console.log('Accurate labels:', labels.join(', '));
});
```

First, use JavaScript's .filter() method to remove any labels with low scores.

Next, keep around only the description rather than the whole object.

After running this, you should see only the labels with confidence greater than 75%, which turns out to be dog, mammal, and vertebrate:

```
> 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.

14.1.2 Faces

Detecting faces, in many ways, is a lot like detecting labels. Rather than getting "what's in this picture," however, you'll get details about faces in the image, as specifics about where each face is, and where each facial feature is (for example, the left eye is at this position). Further, you're also able to discover details about the emotions of the face in the picture, including things like happiness, anger, and surprise, as well as other facial attributes such as whether the person is wearing a hat, whether the image is blurred, and the tilt of the image.

As with the other image recognition aspects, many of these things are expressed as scores, confidences, and likelihoods. As we mentioned earlier, even we don't know for sure whether someone is sad in an image (perhaps they're only pensive). The API will express how similar a facial expression is to others for which it was trained that those were sad. Let's start with a simple test to detect whether an image has a face. For example, you may be curious if the dog image from earlier counts as a face, or whether the Vision API only considers humans. See the following listing.

Listing 14.5 Detecting whether 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) => {
  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:

```
> Nope! There's no face in that image.
```

Well, that was boring. Try looking at a face and all the various annotations that come back on the image. Figure 14.5 shows 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)

In the next listing you'll look at the image of what you think is a happy kid and check whether the Cloud Vision API agrees with your opinion.

Listing 14.6 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) => {
    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');
  });
});
```

Here you use the joy and anger attributes of the face to see.

When you run this little snippet, you'll see that you're very sure that there is a face, and that the face is happy (if you try this same script against the picture of the dog, you'll see that the dog's face doesn't count):

```
> 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 there'd be only likelihoods and not certainties. In this case, the @google-cloud/vision client library for Node.js is making some assumptions for you, saying "If the likelihood is `LIKELY` or `VERY_LIKELY`, then 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 you want to say with certainty that the kid is happy only if it's *very* likely.

Listing 14.7 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;
  const faces = data[0];

  faces.forEach((face, i) => {
    const rawFace = rawFaces[i];
    console.log('How sure are we that there is a face?', face.confidence + '%');
    console.log('Are we certain the face looks happy?',
```

You can grab the faceAnnotations part of the response in your data attribute.

The faces should be in the same order, so face 1 is face annotation 1.

```
                rawFace.joyLikelihood == 'VERY_LIKELY' ? 'Yes' : 'Not really');
    console.log('Are we certain the face looks angry?',
                rawFace.angerLikelihood == 'VERY_LIKELY' ? 'Yes' : 'Not really');
  });
});
```

You need to look specifically at the joyLikelihood attribute and check that its value is VERY_LIKELY (and not LIKELY).

After running this, the likelihood of the face being joyful turns out to be `VERY_LIKELY`, so the API is confident that this is a happy kid (with which I happen to agree):

```
> 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.

14.1.3 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 to create an image of the document, but they wanted to be able to edit that document in a word processor. Many companies found a way to recognize the words and convert the document from an image to text that you could treat like any other electronic document. Although you might not use the Cloud Vision API to recognize a scanned document, it can be helpful when you're shopping at the store and want to recognize the text on the label. You're going to try doing 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 you ask it to detect the text, as shown in the next listing.

Listing 14.8 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, ' '));
});
```

Use the detectText() method to find text in your image.

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 the following:

```
> The label says: BROOKLYN COWBOY WINERY
```

Figure 14.7 Thank-you card

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 text. It isn't perfect because there always seems to be some subjective aspect to putting text together to be useful. Let's see what happens with a particularly interesting greeting card, shown in figure 14.7. It's easy for us humans to understand what's written on this card ("Thank you so much" from "Evelyn and Sebastian"), but for a computer, this card presents some difficult aspects. 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." Even if a computer can recognize the text in the fancy font, the order of the words is something that takes more than only recognizing text. It's 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, shown in the next listing.

Listing 14.9 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) => {
  const textAnnotations = data[0];
  textAnnotations.forEach((item) => {
    console.log(item);
  });
});
```

You 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 understand only the "Evelyn & Sebastian" text at the bottom and doesn't find anything else in the image, as shown in the following listing.

Listing 14.10 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 by trying to recognize some popular logos.

14.1.4 Logo recognition

As you've certainly noticed, logos often tend to be combinations of text and art, which can be tricky for a computer to identify in an image. Sometimes a detecting text will come up with the right answer (for example, 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. The logo might look a bit like the thank-you card we saw earlier, or it might not including text at all (for example, 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, shown in figure 14.8.

Figure 14.8 The FedEx logo

You can detect this similar to how you detected labels and text, as the next listing shows.

Listing 14.11 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(', '));
});
```

Turn on verbose mode here to include the bounding box coordinates you see below.

In this case, you find the right logo as expected:

```
> Found the following logos: FedEx
```

Now run the same code again with a more complicated logo, shown in figure 14.9.

Figure 14.9 The Tostitos logo

Running the same code again on this logo figures out what it was!

```
> Found the following logos: Tostitos
```

But what about a logo with no text and an image, like that in figure 14.10?

If you run the same code yet again, you get the expected output:

```
> Found the following logos: Starbucks
```

Figure 14.10 Starbucks' logo

Finally, let's look at figure 14.11, which is an image containing many logos.

Figure 14.11 Pizza Hut and KFC next to each other

In this case, running your logo detector will come out with two results:

```
> 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."

14.1.5 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. Even if we were able to come up with an absolute number quantifying "how inappropriate" an image is, each workplace would need to make its own decisions about whether something is appropriate.

We're not even lucky enough to have that capability. Even the Supreme Court of the United States 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. Here we'll look at the Cloud Vision API and some of the things it can discover. I hope you'll be comfortable relying on this fuzziness because it's the same vision algorithm that filters out unsafe images when you do a Google search for images.

> **NOTE** As you might guess, I won't be using pornographic or violent demonstration images. Instead I 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 I mentioned was pornography, known by the API as "adult" content. This likelihood is whether 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). Although medical images and adult images can overlap, many images 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 whether an image depicts any form of violence. 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 violent).

The final aspect of safe search is called *spoof detection*. As you might guess, this practice detects whether an image appears to have been altered somehow, particularly if the alternations lead to the image looking offensive. This change 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 you'll investigate whether you should consider it safe for work. Obviously you should, but let's see if Cloud Vision agrees in the next listing.

Listing 14.12 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 you can see in the result:

```
> { adult: false, spoof: false, medical: false, violence: false }
```

As you 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, you need to use the `verbose` mode that you saw earlier, as shown in the next listing.

Listing 14.13 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 unlikely:

```
> { adult: 'VERY_UNLIKELY',
  spoof: 'VERY_UNLIKELY',
  medical: 'VERY_UNLIKELY',
  violence: 'VERY_UNLIKELY' }
```

We've looked at what each detection does, but what if you want to detect multiple things at once? Let's explore how you can combine multiple types of detection into a single API call.

14.1.6 Combining multiple detection types

The Cloud Vision API was designed to allow multiple types of detection in a single API call, and what you been doing when you call `detectText`, for example, is specifically asking for only a single aspect to be analyzed. Let's look at how you can use the generic `detect` method to pick up multiple things at once. The photo in figure 14.12 is of a protest outside a McDonald's where employees are asking for higher wages. Let's see what's detected when you ask the Cloud Vision API to look for logos in listing 14.14, as well as violence and other generic labels.

Figure 14.12 McDonald's protest

Listing 14.14 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 out that although some labels and logos occur in the image, the crowd doesn't seem to trigger a violence categorization:

```
> Does this image have logos? McDonald's
Are there any labels for the image? crowd
Does this image show violence? No
```

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.

14.2 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're 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 you did in the protest image where you asked for logos, safe search, and labels, that action would cost the same as making one request for each of those features. The only benefit from running multiple detections at once is in latency, not price.

The good news is you can use a specific Cloud Vision API tier with the first 1,000 requests per month absolutely free. The examples we 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). Remember that a request is defined as asking for one feature (which means asking for logos and labels on one image is two requests). 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.

14.3 Case study: enforcing valid profile photos

As you might recall, InstaSnap is a cool application that allows you to upload images and share them with your friends. We've talked about where you might store the images (it seemed like Google Cloud Storage was the best fit), but what if you want to make sure that a profile photo has a person in it? Or at least show a warning if it doesn't? Let's look at how you might do this using the Cloud Vision API. After reading this far you should be familiar with the detection type that you'll need here: faces. The flow of how this might work in your application is shown in figure 14.13.

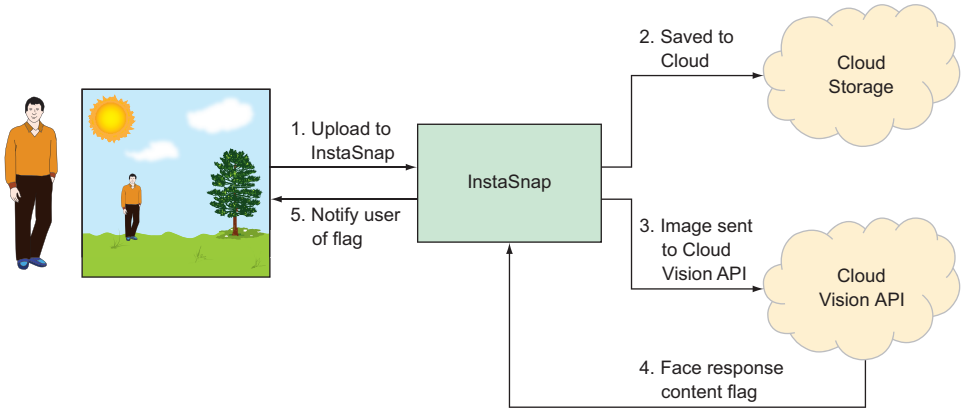


Figure 14.13 Flow of enforcing valid profile photos

As you can see here, the idea is that a user would start by uploading a potential profile photo to your InstaSnap application (1). Once received, it would be saved to Cloud Storage (2). Then you'll send it to the Cloud Vision API (3) to check whether it has any faces in it. You'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. If someone wants their profile picture to be of their cat, that's fine—you only want to warn them about it.

You'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 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 14.15 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);
  });
}
```

You'll use this imageHasFace method later to decide whether to show a warning.

After you have this helper method, you can look at how to plug it into your request handler that's called when users upload new profile photos. Note that the following code is a piece of a larger system so it leaves some methods undefined (such as `uploadToCloudStorage`).

Listing 14.16 Adding the verification step into the flow

Start by defining the request handler for an incoming profile photo. This method follows the standard request/response style used by libraries like Express.

Use a generic object to store the API response as you build it up throughout the flow of promises.

```
const handleIncomingProfilePhoto = (req, res) => {
  const apiResponse = {};
  const url = req.user.username + '-profile-' + req.files.photo.name;
  return uploadToCloudStorage(url, req.files.photo)
    .then( () => {
      apiResponse.url = url;
      return imageHasFace(url);
    })
    .then( (hasFace) => {
```

To kick things off, first upload the photo itself to your Cloud Storage bucket. This method is defined elsewhere but is easy to write if you want.

After the image is stored in your bucket, usey our helper function, which returns a promise about whether the image has a face in it.

```
      apiResponse.hasFace = hasFace;
    })
    .then( () => {
      res.send(apiResponse);
    });
}
```

Based on the response from your helper function, you set a flag in your API response object that says whether the image has a face. In the application, you can use this field to decide whether to show a warning to the user about their profile photo.

Finally, send the response back to the client.

And now you 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.

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.
- Because 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 hashtags

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, you 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 you 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 toward steering a vehicle, but about 100 years ago, it probably meant directing horses. In the United States, Denver has a sports team with the same name, so this could refer to a team that Joe coaches (for

example, "Joe drives his Broncos to victory"). Looking at the term *Bronco* on Wikipedia reveals 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 can't say with certainty whether it means that Joe forces his bronco horses to his workplace, or 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 can't 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 complex and still an active area of research. The Cloud Natural Language API attempts to simplify this 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—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 do and see how you might use it in real life, starting with looking at sentiment.

15.1 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. See figure 15.1.

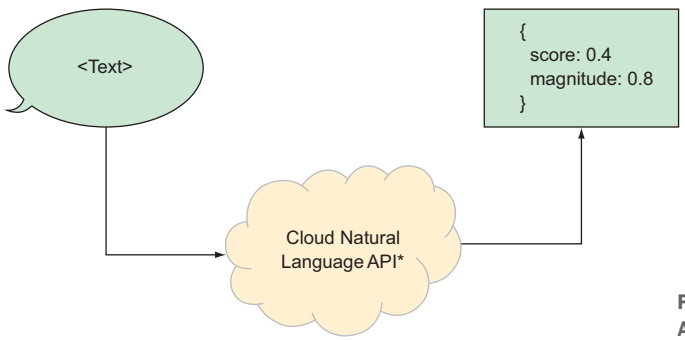


Figure 15.1 Natural Language API flow overview

As of this writing, the NL API can annotate three features of input text: syntax, entities, and sentiment. Let's look briefly at each of these to get an idea of what they mean:

- *Syntax*—Much like diagramming sentences in grade school, the NL API can parse a document into sentences, finding "tokens" along the way. These tokens would have a part of speech, canonical form of the token, and more.
- *Entities*—The NL API can parse the syntax of a sentence. After it does that, it can also look at each token individually and do a lookup in Google's knowledge graph to associate the two. For example, if you write a sentence about a famous

person (such as Barack Obama), you're able to find that a sentence is about Barack Obama and have a pointer to a specific entity in the knowledge graph. Furthermore, 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 mentioned in passing.

- *Sentiment*—Perhaps the most interesting aspect of the NL API is the ability to understand the emotional content involved in a chunk of text and recognize that a given sentence expresses positive or negative emotion and in what quantity. 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"—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.

15.2 Sentiment analysis

One interesting aspect of "understanding" is recognizing the sentiment or emotion of what is said. As humans, we can generally tell whether a given sentence is happy or sad, but asking a computer to do this is still a relatively new capability. 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 because they note a good thing about the car as well as a bad thing. 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 track both the sentiment itself as well as the magnitude of the overall sentiment that went into coming up with the final sentiment result. Table 15.1 contains 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

Sentence	Sentiment	Magnitude
"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, 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, add the two vectors to get a final vector as shown in figure 15.2. It should become clear that the magnitude dimension of the vector will be a sum of both, even if the sentiment dimensions cancel each other out and return a mostly neutral overall sentiment.

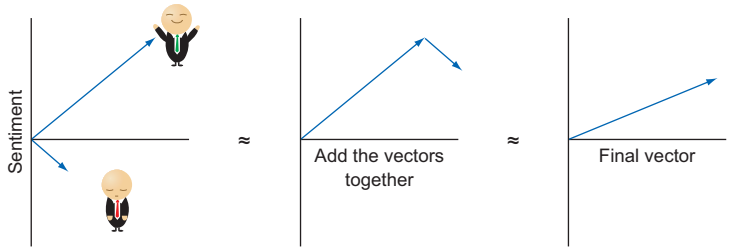


Figure 15.2 Combining multiple sentiment vectors into a final vector

In cases where the score is significant (for example, not close to neutral), the magnitude isn't helpful. But 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 positivity and negativity neutralize one another. When you send 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. As shown in figure 15.3, 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.

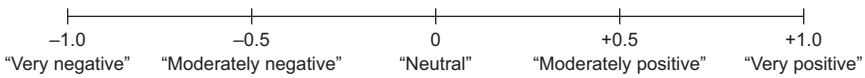


Figure 15.3 Sentiment scale from -1.0 to +1.0

In cases where the 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 because 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, 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 to use this API.

To start, 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 one result, and if you click it, you land on a page with a big Enable button, shown in figure 15.4. Click that, and you're ready to go.

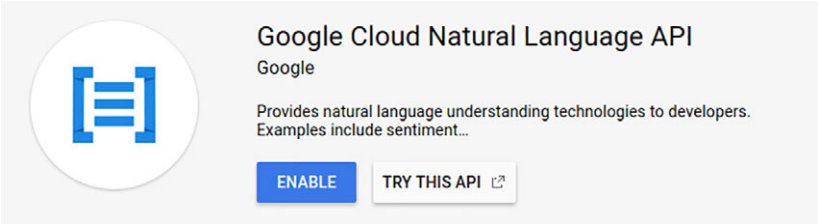


Figure 15.4 Enable the Natural Language API.

Now you'll need to install the client library for Node.js. To do this, run `npm install @google-cloud/language@0.8.0` and then you can start writing some code in the next listing.

Listing 15.1 Detecting sentiment for a sample sentence

```
const language = require('@google-cloud/language')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

language.detectSentiment('This car is really pretty.').then((result) => {
console.log('Score:', result[0]);
});
```

Don't forget that the project ID must match the credentials in your service account.

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:

```
> 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 halfway!) between totally negative (`1.0`) and totally positive (`-1.0`). It's 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 look at one of those sentences that were overall neutral, shown in the following listing.

Listing 15.2 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'll see exactly what we predicted: a score of zero. How you we tell the difference between content that is "neutral" overall but highly emotional and something truly neutral? Let's compare two inputs while increasing the verbosity of the request, as shown in the next listing.

Listing 15.3 Representing 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!',
  'This is a car.'
];

inputs.forEach((content) => {
  language.detectSentiment(content, {verbose: true})
    .then((result) => {
      const data = result[0];
      console.log([
        'Results for "' + content + '":',
        '  Score:     ' + data.score,
        '  Magntiude: ' + data.magnitude
      ].join('\n'));
    });
});
```

This input is emotional but should overall be close to neutral.

This sentence is unemotional and should be overall close to neutral.

Make sure to request "verbose" output, which includes the magnitude in addition to the score.

When you run this, you should see something like the following:

```
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 had quite a bit of emotion. Additionally, it seems that what you 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 to another area of analysis and look at how to recognize key entities in a given input.

15.3 Entity recognition

Entity recognition determines whether input text contains any special entities, such as people, places, organizations, works of art, or anything else you'd consider a proper noun. It works by parsing the sentence for tokens and comparing those tokens against the entities that Google has stored in its knowledge graph. This process 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, if you're interested in doing things like suggesting tags or metadata about textual input, you can use entity detection to determine 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, as the next listing shows.

Listing 15.4 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:

```
> { people: [ 'Barack Obama' ],
  goods: [ 'iPhone' ],
  organizations: [ 'Blackberry' ],
  places: [ 'Hawaii' ] }
```

As you can see, the Natural Language API detected four distinct entities: Barack Obama, iPhone, Blackberry, and Hawaii. This ability can be helpful if you're trying to discover whether famous people or a specific place is mentioned in a given sentence.

But were all of these terms equally important in the sentence? It seems to me that "Barack Obama" was far more prominent in the sentence than "Hawaii."

The Natural Language API can distinguish between differing levels of prominence. It attempts to rank things according to how important they are in the sentence so that, for example, you could consider only the most important entity in the sentence (or the top three). To see this extra data, use the verbose mode when detecting entities as shown here.

Listing 15.5 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};

language.detectEntities(content, options).then((result) => {
  console.log(result[0]);
});
```

Use {verbose: true} to get more context on the annotation results.

When you run this code, rather than 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"):

```
> { 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] } ] }
```

What if you want to specifically get the most salient entity in a given sentence? 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 examples in the next listing and ask the API to decide which entity is deemed most important.

Listing 15.6 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) => {                          <-- Sort entities by
      return -(a.salience - b.salience);                   decreasing salience
    });                                                    (largest salience first).
    console.log(
      'For the sentence "' + content + '"',
      '\n  The most important entity is:', entities[0].name,
      '(' + entities[0].salience + ')');
  });
});
```

After running this code, you can see 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 get an indicator only of what appears, rather than how important it is to the sentence, as shown next:

```
> 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 (0.44054606556892395)
```

Let's take it up a notch and see what happens when you 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 support languages other than English—it currently includes both Spanish (`es`) and Japanese (`jp`). Run an entity analysis on our sample Spanish sentence, which translates to "Hugo Chavez was a dictator of Venezuela." See the following listing.

Listing 15.7 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,
  language: 'es'
}).then((result) => {
  console.log(result[0]);
});
```

Turn on verbose mode to see the salience rankings.

Here you use the BCP-47 language code for Spanish (es). If you leave this empty, the API will try to guess which language you're using.

When you run this code, you should see something like the following:

```
> { 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.

15.4 Syntax analysis

You may recall your elementary school English teacher asking 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 dependency graphs, which allow you to see the core of the sentence and push modifiers and other nonessential information to the side. For example, let's take the following sentence:

> The farmers gave their kids fresh vegetables.

Diagramming this sentence the way our teachers showed us might look something like figure 15.5.

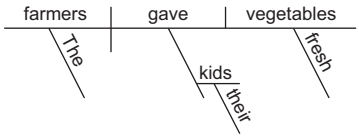


Figure 15.5 Diagram of a sample sentence

Similarly, the Natural Language API can provide a dependency graph given the same sentence as input. The API offers the ability to build a syntax tree 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 made sense. 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 in figure 15.6.

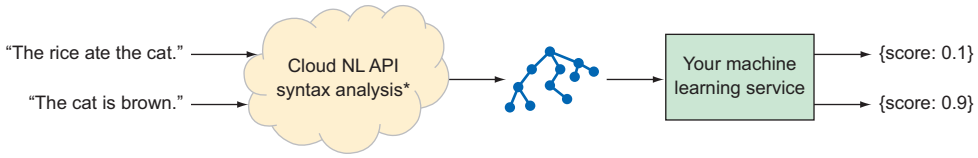


Figure 15.6 Pipeline for an example sense-detection service

You probably wouldn't use this API directly in your applications, but it could be 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, tokenizing the sentence, 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 in the following listing.

Listing 15.8 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 code will give you a table of the dependency graph, which should look like table 15.2.

Table 15.2 Comparing sentences with similar sentiment and different magnitudes

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 figure 15.7.

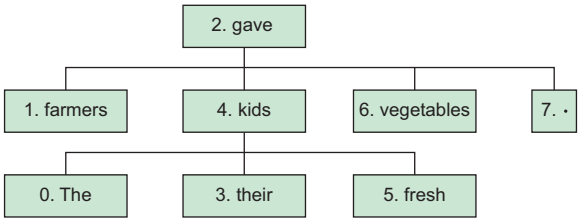


Figure 15.7 Dependency graph represented as a tree

Now that you understand the different types of textual analysis that the Natural Language API can handle, let's look at how much it will cost.

15.5 Understanding pricing

As with most Cloud APIs, the Cloud Natural Language API charges based on the 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. All of our examples so far would be billed as a single unit, but if you send a long document for entity recognition, it'd be billed as the number of 1,000 character chunks needed to fit the entire document (`Math.ceil(document.length / 1000.0)`).

This type of billing is easiest when you 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, different types of analysis cost different amounts, with entity recognition leading the pack at $0.001 each. 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 amounts by 1,000 makes for much more manageable numbers, coming to $1 per thousand requests for most entity recognition and sentiment analysis operations. Also note that when you combine two types of analysis (for example, a single request for sentiment and entities), the cost is the combination (for example, $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 (about 2,500 characters), and sentiment analysis over 2,000 short tweet-like snippets every day. The cost breakdown is summarized in table 15.4.

Table 15.4 Pricing example for Cloud Natural Language API

Item	Quantity	1k character "chunks"	Cost per unit	Total per month
Entity detection (long-form)	1,000	3,000	$0.001	$3.00
Sentiment analysis	60,000	60,000	$0.001	$60.00
Total				$63.00

Note specifically that the long-form documents ballooned into three times the number of chunks because they're about 2,500 characters (which needs three chunks), and that your sentiment analysis requests were defined as 2,000 *daily* rather than *monthly*, resulting in a thirty-times multiplier. Now that you've seen the cost structure and all the different types of analysis offered by the Natural Language API, you'll try putting a couple of them together into something that might provide some value to users: hash-tagging suggestions.

15.6 Case study: suggesting InstaSnap hash-tags

As you may recall, our sample application, InstaSnap, is an app that allows people to post pictures and captions and share them with their friends. Because 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 you 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, you'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, you'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 you'd display a list of suggested tags to the user. See figure 15.8 for an overview of this process.

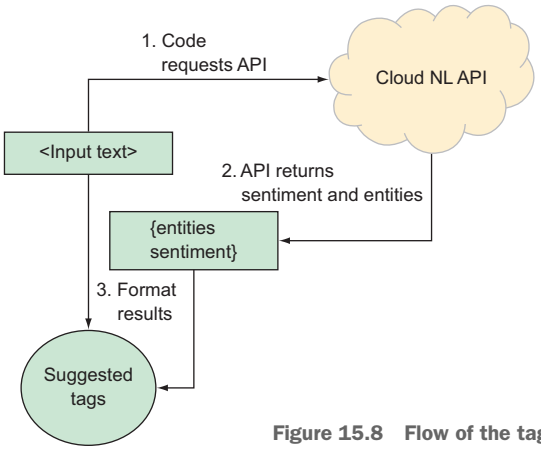


Figure 15.8 Flow of the tagging suggestion process

Let's start by looking at the code to request both sentiment and entities in a single API call, shown in the following listing.

Listing 15.9 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!';
constdocument = 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);
});
```

Here you're assuming Elon Musk has finally managed to land on Mars (and uses InstaSnap).

To handle multiple annotations at once, create a "document," and operate on that.

If you run this snippet, you should see output that looks familiar:

```
> Sentiment was { score: 0.4000000059604645, 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 you can do to apply some tags, starting with entities first. For most entities, you can 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. You 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 you have some suffixes that are positive and some negative.

What about for the sentiment? You're can come up with some happy and sad tags and use those when the sentiment passes certain thresholds. Then you can make a `getSuggestedTags` method that does all the hard work, as the following listing shows.

Listing 15.10 Your 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';
```

Come up with a list of possible suffixes for each category of entity.

Store a list of emotional tags for each category (positive, negative, mixed, or neutral).

Use the sentiment analysis results to choose a tag from the category.

Don't forget to check the magnitude to distinguish between "mixed" and "neutral".

```
  } 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];
    } 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;
};
```

Use a try/catch block in case you don't happen to have tag choices for each particular combination.

Now that you have that method, your code to evaluate it and come up with some suggested tags should look simple, as you can see in the next listing.

Listing 15.11 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!';
constdocument = 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);
  console.log('The suggested tags are', suggestedTags);
  console.log('The suggested caption is',
  '"' + caption + ' ' + suggestedTags.join(' ') + '"');
});
```

Here you 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:

```
> The suggested tags are [ '#Yay', '#SpaceX4Life', '#MarsIsHome' ]
The suggested caption is "SpaceX lands on Mars! Fantastic! #Yay #SpaceX4Life
     #MarsIsHome"
```

Summary

- The Natural Language API is a 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 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

When we talk about speech recognition, we generally mean taking an audio stream (for example, an MP3 file of a book on tape) and turning it into text (in this case, back into the actual written book). This process sounds straightforward, but as you may know, language is a particularly tricky human construct. For instance, the psychological phenomenon called the McGurk effect changes what we *hear* based on what we *see*. In one classic example, the sound "ba" can be perceived as "fa" so long as we see someone's mouth forming an "f" sound. As you might expect, an audio track alone is not always enough to completely understand what was said.

This confusion 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, so 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 could use the context of "drove" to guess that this word was "car."

This phenomenon leads to some 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, but the first child said, "I bring you gold," the second said, "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 guess based on the context. Figure 16.1 shows another humorous misheard name.

Figure 16.1 Understanding based on context ("Who is Justin Bieber?")

What does this mean for you? In general, you should treat the results from a given audio file as 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 yet, but it 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.

16.1 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—an audio stream. The simplest way of recognizing the textual content in an audio file is to send the audio file (for example, a .wav file) to the Cloud Speech API for processing. The output of the audio will be what was said in the audio file.

First, you'll need to tell the Cloud Speech API the format of the audio, because many different formats exist, each with its own compression algorithms. Next, the API needs to know the sample rate of the file. This important aspect of digital signal processing tells the audio processor the clock time covered by each data point (higher sample rates are closer to the raw analog audio). To make sure the API "hears" the audio at the right speed, it must know the sample rate.

> **TIP** Although you probably don't know the sample rate of a given recording, the software that created the recording likely added a metadata tag to the file stating the sample rate. 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 so the API knows which lingual model to use when recognizing content in the audio file. Let's dig into some of the code to do this, using a premade recording of an audio file stored on Google Cloud Storage to start. The audio format properties of this file are shown in figure 16.2.

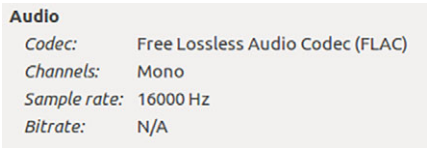


Figure 16.2 Audio format properties of the premade recording

Before you get going, you'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 Cloud Speech API. This should only have one result, which opens a page with an Enable button, as shown in figure 16.3. Click this, and you'll be all set.

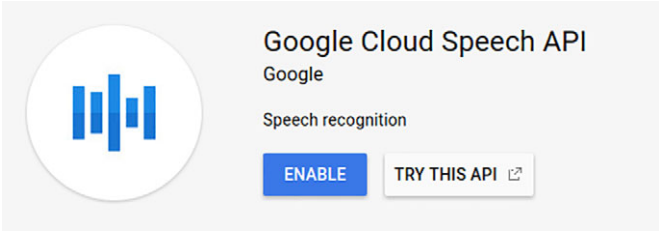


Figure 16.3 Enabling the Cloud Speech API

Now that the API is enabled, you'll install the client library. To do this, run `npm install @google-cloud/speech@0.8.0`. Now write some code that recognizes the text in this file, as shown in the following listing.

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';
const config = {
  encoding: 'FLAC',
  sampleRate: 16000
};
speech.recognize(audioFilePath, config).then((response) => {
  const result = response[0];
  console.log('This audio file says: "' + result + '"');
});
```

Notice that this file lives on Google Cloud Storage rather than as a local audio file.

In this API, the configuration is required because 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, you should see some interesting output:

```
> This audio file says: "how old is the Brooklyn Bridge"
```

One important thing to notice is how long the recognition took. The reason is 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. Therefore, extraordinarily long audio files (for example, more than a few seconds) shouldn't be processed like this. Another important thing to notice is that there's no concept of confidence in this result. How sure is the Cloud Speech API that the audio says that exact phrase? To get that type of information, you can use the `verbose` flag, as the following listing shows.

Listing 16.2 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
};
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)');
});
```

Here you set the verbose option to true.

When you run this code, you should see output that loooks something like the following:

```
> This audio file says: "how old is the Brooklyn Bridge"
➥ (with 98% confidence)
```

How do you deal with longer audio files? What about streaming audio? Let's look at how the Cloud Speech API deals with continuous recognition.

16.2 Continuous speech recognition

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 a large audio file, which is too big to treat as one big blob, so instead you have to break it up into smaller chunks. This is also true when you're trying to recognize streams that are live (not prerecorded), because these streams keep going until you decide to turn them off. To handle this, the Speech API allows asynchronous recognition, which will accept chunks of data, recognize them along the way, and return a final result after the audio stream is completed. Let's look at how to do that with your same file, but treated as chunks, as shown in the next listing.

Listing 16.3 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) => {
  const operation = result[0];
  operation.on('complete', (results) => {
   console.log('This audio file says: "' + results[0].transcript + '"',
   '(with ' + Math.round(results[0].confidence) + '% confidence)');
  });
});
```

Instead of demanding that the Speech API recognize some text immediately, we "start recognizing," which kicks off a streaming version of recognition.

When the operation completes, it returns the recognized transcript as the result.

The result of this startRecognition method is a "long-running operation," which will emit events as the recognition process continues.

As you can see, this example looks similar to the previous examples; however there are some important differences, as the annotations show. If you run this code, you should see the exact same result as before, shown in the next listing.

Listing 16.4 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.

16.3 Hinting with custom words and phrases

Because 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 (for example, 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 API's ranking system for each request. To demonstrate how this works, let's see if you can throw in a new suggestion that might make the Speech API misspell "Brooklyn Bridge." In the following example, you update your `config` with some additional context and then rerun the script.

Listing 16.5 Speech recognition with suggested phrases

```
const config = {
  encoding: 'FLAC',
  sampleRate: 16000,
  verbose: true,
  speechContext: { phrases: [
    "the Brooklynne Bridge"
  ]}
};
```

Here you suggest the phrase "the Brooklynne Bridge" as a valid phrase for the Speech API to use when recognizing.

If you were to run this script, you'd see that the Speech API does indeed use the alternate spelling provided:

```
> This audio file says: "how old is the brooklynne bridge" (with 90% confidence)
```

> **NOTE** As with all of the machine-learning APIs you've learned in this book, results vary over time as the underlying systems learn more and get better. If the output of the code isn't exactly what you see, don't worry! It means that the API has improved since this writing.

Notice, however, that the confidence is somewhat lower than before. This is because two relatively high-scoring results would have come back: "Brooklyn Bridge" and (your suggestion) "brooklynne bridge." These two competing possibilities make the Speech API less confident in its choice, although it's 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, recognized profanity will be "starred out" except for the first letter (for example, "s***"). Now that you have a grasp of some of the advanced customizations, let's talk briefly about how much this will cost.

16.4 Understanding pricing

Following the pattern of the rest of Google Cloud Platform, the Cloud Speech API will charge you only 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—you won't be charged at all. Beyond that it costs 2.4 cents per minute of audio sent.

Because 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). A 5-second audio file is billed as one-quarter minute ($0.006), and a 46-second audio field is billed as a full minute ($0.024). Finally, let's move on to a possible use of the Cloud Speech API: generating hashtag suggestions for InstaSnap videos.

16.5 Case study: InstaSnap video captions

As you may recall, InstaSnap is your 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 previous chapter on the Cloud Natural Language API, you saw how you could generate suggested hashtags based on the caption of a photo. Wouldn't it be neat if you could suggest tags based on what's being said in the video? From a high level, you'll still rely on the Cloud Natural Language API to recognize any entities being discussed (if you aren't familiar with this, check out chapter 15 on the Cloud Natural Language API). Then you'll pull out the audio portion of the video, figure out what's being said, and come back with suggested tags. Figure 16.4 shows the flow of each step, starting at a recorded video and ending at suggested tags:

1. First, the user records and uploads a video (and types in a caption).
2. Here, your servers would need to separate the audio track from the video track (and presumably format it into a normal audio format).
3. Next, you 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 you then combine with the caption that was set in step 1.
5. You then send all of the text (caption and video transcript) to the Cloud Natural Language API.
6. The Cloud NL API will recognize entities and detect sentiment from the text, which you can process to come up with a list of suggested tags.
7. Finally, you send the suggested tags back to the user.

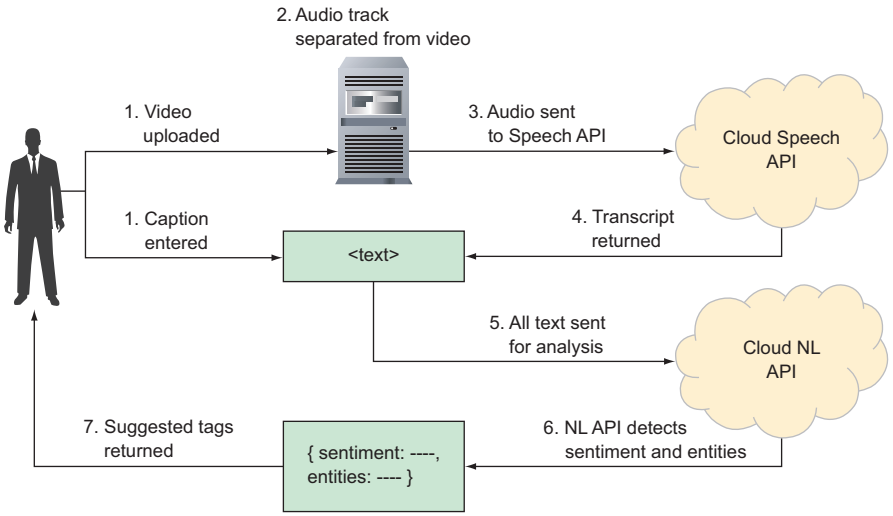


Figure 16.4 Overview of your hashtag suggestion system

If you read the chapter on natural language processing, steps 5, 6, and 7 should look familiar—they're 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. Start by writing a function that will take a video buffer as input and return a JavaScript promise for the transcript of the video, shown in the next listing. Call this function `getTranscript`.

Listing 16.6 Defining a new `getTranscript` function

```
const Q = require('q');
const speech = require('@google-cloud/speech')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

const getTranscript = (videoBuffer) => {
  const deferred = Q.defer();

  extractAudio(videoBuffer).then((audioBuffer,
     ➥ audioConfig) => {
    const config = {
      encoding: audioConfig.encoding, // for example, 'FLAC'
      sampleRate: audioConfig.sampleRate, // for example, 16000
      verbose: true
    };
```

Here you're relying on an open source promise library called Q. You can install it with npm install q.

Use Q.defer() to create a deferred object, which you can resolve or reject in other callbacks.

You're assuming that there's 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).

```
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) => {
      deferred.reject(err);
    });
  }).catch((err) => {
    deferred.reject(err);
  });

  return deferred.promise;
};
```

If there are any errors, reject the deferred object, which will trigger a failed promise.

Here you return the promise from the deferred object, which will be resolved if everything works and rejected if there's a failure.

Now you have a way to grab the audio and recognize it as text, so you can use that along with the code from chapter 15 to do the rest of the work. To make things easier, you'll generalize the functionality from chapter 15 and write a quick method in listing 16.7 that will take any given content and return a JavaScript promise for the sentiment and entities of that content. Call this method `getSentimentAndEntities`. (See chapter 15 for more background if this is new to you.)

Listing 16.7 Defining a `getSentimentAndEntities` function

```
const Q = require('q');
const language = require('@google-cloud/language')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

const getSentimentAndEntities = (content) => {
  constdocument = language.document(content);
  const config = {entities: true, sentiment:true, verbose: true};
  returndocument.annotate(config).then(
    returnnew Q(data[0]);
    ➥ // { sentiment: {...}, entities: [...] }
  });
};
```

Start by creating a NL document.

Then annotate the document with sentiment and entities found.

Finally, return a promise whose value will have properties for both the sentiment and the entities found in the text provided.

Now you have all the tools you need to put your code together. To wrap up, you'll build the final handler function that accepts a video with properties for the video buffer and caption and prints some suggested tags, as shown in the next listing. The function that comes up with the suggested tags (`getSuggestedTags`) is the same one that you wrote in chapter 15.

Listing 16.8 Defining a `getSuggestedTags` function

Here you create a promise to return the sentiment and entities based on the audio content in the uploaded video.

You're relying on Q's allSettled method, which waits until all promises have either succeeded or failed. You should end up with lots of results, some in a fulfilled, which means you can use those results.

```
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) => {
      return getSentimentAndEntities(video.transcript);
    }),
    getSentimentAndEntities(video.caption)
  ]).then((results) => {
    let suggestedTags = [];
    results.forEach((result) => {
      if (result.state === 'fulfilled') {
        const sentiment = result.value.sentiment;
        const entities = result.value.entities;
        const tags = getSuggestedTags(sentiment, entities);
suggestedTags = suggestedTags.concat(
tags);
      }
    });
    console.log('The suggested tags are', suggestedTags);
    console.log('The suggested caption is',
                '"' + caption + ' ' + suggestedTags.join(' ') + '"');
  });
};
```

Next you create a promise to return the sentiment and entities from the caption set when uploading the video.

After all the results are settled (via Q.allSettled), iterate over each and use only those that resolved successfully.

Based on the sentiment and entities from the text, use the function you built in chapter 15 to come up with a list of suggested tags and add them to the list.

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, because you 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").

Summary

- Speech recognition takes a stream of audio and converts it into text, which can be deceivingly complicated due to things like the McGurk effect.
- Cloud Speech is a hosted API that can perform speech recognition on audio files or streams.

17 Cloud Translation: multilanguage 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

If you've ever tried to learn a foreign language, you'll 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 memorizing a simple map from language A to language B (for example, `houseInSpanish = spanish['house']`), shown in figure 17.1.

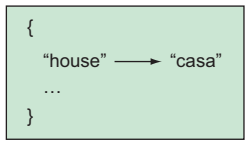


Figure 17.1 Mapping of English to Spanish words

Although this process is challenging for humans, computers are good at it, so this wouldn't be a hard problem to solve. This memorization problem is nowhere near as challenging as a *true* understanding of the language, where you take a "conceptual

representation" in one language and translate it to another, phrasing things in a way that sounds right in the new language. Machine translation aims to solve this problem.

Human languages developed in unique ways. Much like 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 of years into the languages we know today. If you were to hop into 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 completely designed rather than expanding from a single planned city center), but this appears to be the exception rather than the rule.

These issues make the translation problem particularly difficult. Some languages, such as Japanese, feature extraordinarily high levels of complexity. Add to that slang expressions that are ubiquitous, 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 (for example, English in Britain versus America), and, as with anything involving humans, ambiguity of the overall meaning—which has nothing to do with computers!

What started as a simple mapping of words from one language to another has suddenly entered a world where it's not even clear to 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 talking about being "on an airplane *trip*" or "on an airplane flight." Being "in an airplane" conveys the idea of being contained *inside* the airplane. I might use this expression when someone asks why my cell phone reception is so bad. The point would be that I'm stationary while *in*side this airplane.

The distinction is so subtle that if said with a perfect American accent, I probably wouldn't consciously notice the difference, but it might sound a little "off." We're talking about a difference of only a single letter in two prepositions that might translate to the same word in other languages (for example, in Spanish, as *en*).

The fact that an entire Stack Exchange community exists to answer questions about grammar, usage, and other aspects of the English language demonstrates 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 of 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.

17.1 How does the Translation API work?

If this problem is so insurmountable, how does Google Translate work? Is it any different from the Cloud Translation API?

Let's start by looking at the question of how to resolve the complicated problem of understanding vocabularies and grammatical rules. One way would be to try to teach the computer all of the different word pairs (for example, `EnglishToSpanish('home') == 'casa'`) and grammatical words ("English uses subject verb object (SVO) structure"). As we discussed earlier, however, not only is language extraordinarily complex, with exceptions for almost every rule, but it is constantly evolving. You'd be chasing a moving target. Although this method might work with enough effort, it isn't going to be a scalable way of solving the problem.

Another way (and the way that Google Translate uses for many languages) uses something called *statistical machine translation* (SMT). Fundamentally, SMT relies on the same concept as the Rosetta stone, which was a stone engraved with the same inscription in both ancient Greek and Egyptian hieroglyphics. If scholars understood the Greek text, they could use it to decipher the meaning of the Egyptian hieroglyphics. In the case of SMT, rather than Greek and Egyptian on a single stone, the algorithm relies on millions of documents that have equivalents in at least one language pair (for example, English and Spanish).

SMT scans these millions of documents that have translations in several languages (created by human translators) and identifies common patterns across the two documents that are unlikely to be coincidence. The assumption is that if these patterns occur often, it's likely a match between a phrase in the original text and the equivalent phrase in the translated text. The larger the overlap, the closer you get to a true human translation, given that 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 with similar frequency. As you see more and more of this pattern (with matching occurrences of "house" in English and "casa" in Spanish), it becomes likely that when someone wants to know the Spanish equivalent of the word "house," the most correct answer is "casa." 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.

This method has a drawback, however: sentences are translated piece by piece rather than as a whole. If you ask for a translation of an exact sentence that the SMT system has already seen obviously you'll get an exact (human) translation. Unfortunately, it's unlikely that you'll have that exact input and far more likely that your translation will be made up of multiple translations covering several phrases in the sentence. Sometimes this works out fine, but often the translation comes across as choppy due to drawing translations of phrases from different places. If you're

translating a word that hasn't been seen before in any of the training documents, you're out of luck.

For example, translating "I went to the store" from English to Spanish comes across fairly well. Chances are the entire sentence was in a document somewhere, but even if it wasn't, "I went" and "to the store" are likely in those documents, and combining them is pretty natural (see figure 17.2).

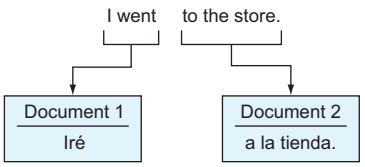


Figure 17.2 Translating based on multiple documents

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 phrases from several places, rather than looking at the sentence as a whole.

This type of result led Google to focus on some newer areas of research, including the same technology underlying the Natural Language API and the Vision API: neural networks.

This type of 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 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 keep the neural network from "forgetting" to translate some parts of the sentence, and has many more technical optimizations to minimize the overall cost of training and storing the neural network handling translation.[1]

What this means is that you end up with smoother translations. For example, that same sentence in German 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,

[1] For more information about Google's Neural Machine Translation system, see https://arxiv.org/pdf/1609.08144v2.pdf.

Spanish, Portuguese, Chinese, Japanese, Korean, and Turkish—a total of eight language pairs) and rely on SMT ("the old way") for 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.

17.2 Language detection

The simplest application of the Translation API is looking at some input text and figuring out what language it is. Though 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, as shown in figure 17.3.

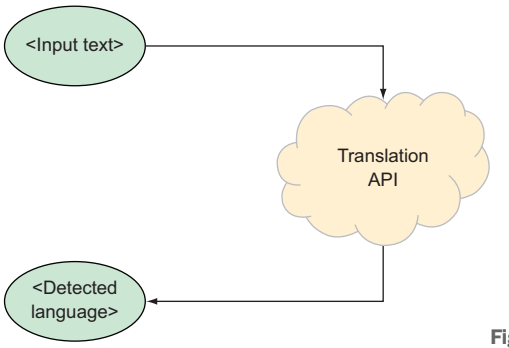


Figure 17.3 Language detection overview

As you might guess, sometimes this is easy (as in the earlier German sentence), and sometimes this isn't quite as easy (particularly with two languages that are similar or sentences that are short). For example, "No" is a sentence in English, but it's also a sentence (with the same meaning) in Spanish. In general, 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 using the other APIs. Enter "Cloud Translation API" in the main search box at the top of the page. This query should come up with one result, which brings you to a page with an Enable button, shown in figure 17.4. After you click that, the API will be enabled and the code samples work as expected.

Before you write any code, you'll need to install the client library. You can do this using `npm` by running `npm install @google-cloud/translate@1.0.0`. When that's done, you'll dive in with some language detection samples in listing 17.1.

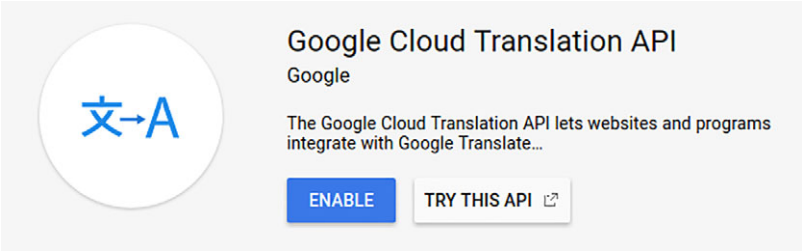


Figure 17.4 Enable button for the Cloud Translation API.

Listing 17.1 Detecting the language of input text

```
const translate = require('@google-cloud/translate')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

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:

```
> 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 a few important things to notice here.

First, and most important for our purposes, the detections were accurate. 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, a confidence level is associated with the result. Like many of the other machine-learning APIs, 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 many 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 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 here, don't worry! The underlying machine-learning algorithms change and improve over time, and the results you get one day may be slightly different later, 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.

17.3 Text translation

Translating text involves a process similar to that for detecting the language. Given some input text and a target output language, the API will return the translated text (if it can), as shown in figure 17.5. Translating text is stateless as well, where you send everything necessary to translate your inputs in the initial request.

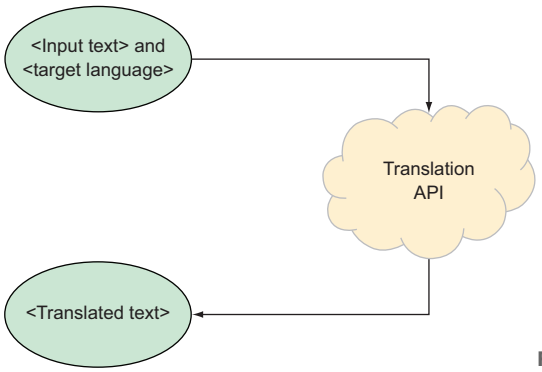


Figure 17.5 Translating text overview

Notice that you specify only the language you want along with the input text—you don't specify the language of the input text. You can tell the Translation API the source language, but if you leave it blank (which many do), it automatically detects the language (for free) as part of the translation.

Given that, let's take those same examples from earlier and try to translate them all to English (`en`) in the next listing.

Listing 17.2 Translating from multiple languages to English

```
const translate = require('@google-cloud/translate')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

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 the sentences translated:

```
> No problem can be solved from the same consciousness that they have arisen.
I am American.
```

Notice a few things missing from what you saw previously when detecting the language.

First, there's no confidence score associated with the translation, so unfortunately, you can't express how confident you are that the translated text is accurate. Although 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 in a single language), the meaning in another language might vary depending on who's doing the translating. Thanks to this ambiguity, a confidence rating wouldn't be that useful.

You might also notice that the source language isn't coming back as a result. If you want that result, you can look at the raw API response, which shows the detected language. The following listing shows how you can get that if needed.

Listing 17.3 Detecting source language when translating

```
const translate = require('@google-cloud/translate')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

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;
```

```
  results.forEach((result, i) => {
    console.log('Sentence', i+1,
                'was detected as', result.detectedSourceLanguage);
  });
});
```

When you run this example, you should see output that shows the detected languages:

```
> 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 straightforward: take some text, get back a detected language, take some text and a target, and get back a translation to the target.

Now let's look briefly at the pricing considerations to take into account.

17.4 Understanding pricing

As with other Cloud APIs, in Translation API you pay for only what you use. 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 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. 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? Whitespace characters are necessary to understand the breaks between words, so they are charged for like any other character (or code point). For billing purposes, "Hello world" is treated as 11 characters due to the space between the two words.

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.

17.5 Case study: translating InstaSnap captions

As you may recall, InstaSnap is your 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 speaks English! In particular, many celebrities are famous worldwide and have fans who want to know what the celebrities are saying in their captions. Let's see if you can use the Translation API to fix this.

Breaking the problem down a bit more, you want to detect if the language of a given caption isn't the same as the user's language. If it isn't, you may want to translate it.

The simple solution is to automatically attempt to translate the text into the user's language—a solution we might call *automatic translation at view-time*.

The problem with this solution is that it will get expensive. For starters, it's unlikely that every one of the captions needs to be translated. Beyond that, even if the caption needed translating, the user might not be interested in that content.

As a result, you should change your design a bit. Instead of trying to translate everything, you could detect the language of text when the caption is created and store that on the post. You can also assume that you know the primary language of each user because they chose one when they signed up for InstaSnap. If you detect language at "post time," you can compare it to the viewer's language and, if they're different, display a button that says "Translate to English" (localized for the viewer's primary language). Then, when the viewer clicks the button, you can request a translation into the viewer's primary language. See figure 17.6 for an overview of this process.

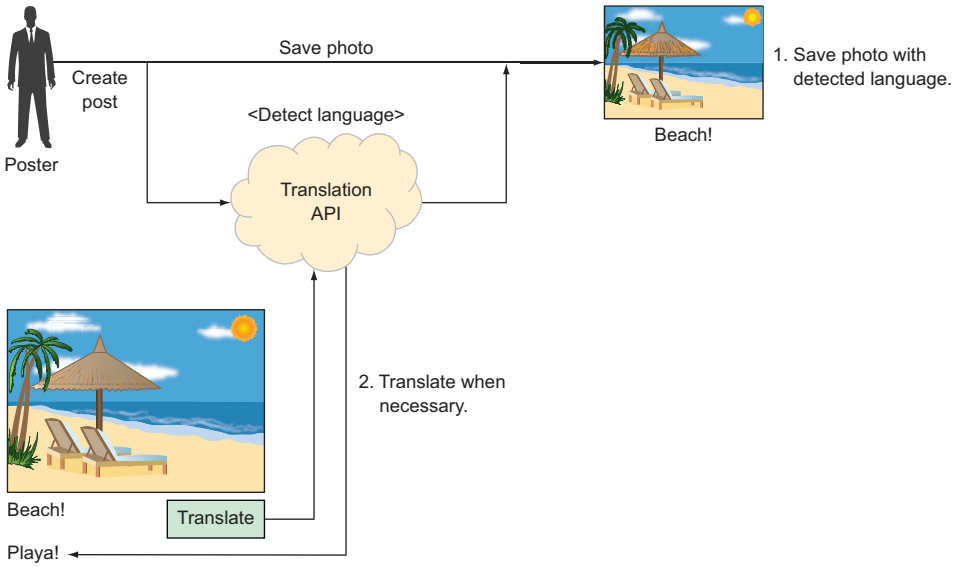


Figure 17.6 Overview of the flow when posting and viewing on InstaSnap

Start by writing some code at upload time to store the detected language, as shown in the next listing. You would call this method after the photo is uploaded.

Listing 17.4 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) => {
```

Given a saved photo, detect the language and save the result.

```
    const result = response[0][0];
    if (result.confidence >0.1) {
      photo.detectedLanguage = result.language;
      photo.save();
    }
  });
};
```

If the confidence is poor, don't do anything.

Next, you can write a function to decide whether to display the Translate button, as the following listing shows.

Listing 17.5 Determining whether to display a translate button

```
const shouldDisplayTranslateButton = (photo, viewer) => {
  if (!photo.detectedLanguage || !viewer.language) {
    return false;
  } else {
    return (photo.detectedLanguage != viewer.language);
  }
}
```

If the two languages are different, this evaluates to true.

If the detected language is empty, you can't do any translating. Similarly, without a target language to translate into, you can't do any translating.

Finally, at view time, you can write a function that will do the translating work, as shown in listing 17.6.

> **NOTE** This code won't run because it uses several "fake" components. It's here to demonstrate how you would wire everything together.

Listing 17.6 Runtime code to handle optional translation of captions

```
const translate = require('@google-cloud/translate')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});

const photoUiComponent = getCurrentPhotoUiComponent();
const photo = getCurrentPhoto();
const viewer = getCurrentUser();
const translateButton = new TranslateUiButton({
  visible: shouldDisplayTranslateButton(photo, viewer),
  onClick: () => {
    photoUiComponent.setCaption('Translating...');
    translate.translate(photo.caption, {to: viewer.language})
      .then((response) => {
        photoUiComponent.setCaption(response[0][0]);
      })
```

You're using a "fake" concept of a Translate button that you can operate on.

You say whether the button is visible by using your previously written function.

If you get a result, you set the photo caption to the translation result.

Before you make the API request, you set the caption to "Translating…" to show that you're doing some work under the hood.

```
      .catch((err) => {
        photoUiComponent.setCaption(photo.caption);
      });
  }
});
```

If there are any errors, you reset the photo caption as it was.

Summary

- Machine translation is the way computers 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.

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?
- Creating and deploying your own ML model

Although we've explored various machine-learning APIs, so far we've focused only on the real-world applications and not on how they work under the hood. In this chapter, we're going to look inside and move beyond these preprogrammed ML problems.

18.1 What is machine learning?

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. Although I'll try to cover some of the core concepts of ML and demonstrate how to write a simple bit of ML code, I'll gloss over the majority of the mathematical theory and most of the calculations. If you're passionate about machine learning, you should absolutely explore other

books that provide more information about the fundamentals of machine learning. With that out of the way, let's explore what exactly is going on inside these ML APIs, such as the speech recognition example shown in figure 18.1.

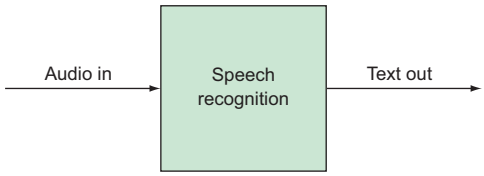


Figure 18.1 Machine learning (speech recognition) as a black-box system

Although many nuances differentiate the types of machine learning, we generally define it as the idea that a system that can be trained with some data and then make predictions based on that training. This behavior is 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 involves 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` → `4` and `40` → `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` → `?`.

We can build systems capable of "learning" using several methods, but the one that has gotten the most interest recently is 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 shown in figure 18.2) to extremely complex (like your brain).

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,

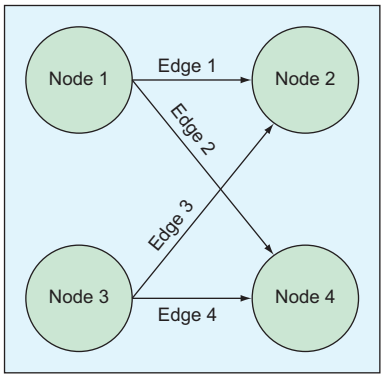


Figure 18.2 Neural network as a directed graph

indicated by the way the arrow is pointing. The line weights determine how much of an input signal is transmitted into an output signal, or 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 from one layer to another until they come out on the other side. See figure 18.3.

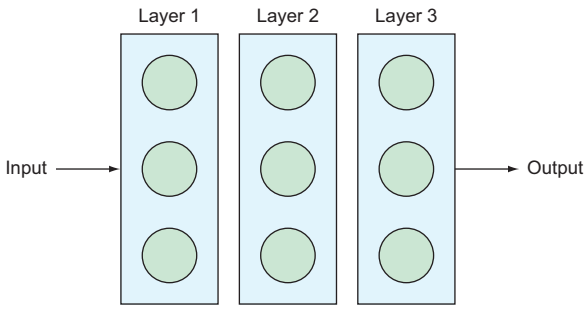


Figure 18.3 The layers of a neural network

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 like a neural network with lots of layers, each consisting of a single node, where each node represents a person in the chain, as shown in figure 18.4. The weights on the edges between each node represent how well the next person can understand the previous person's whispers.

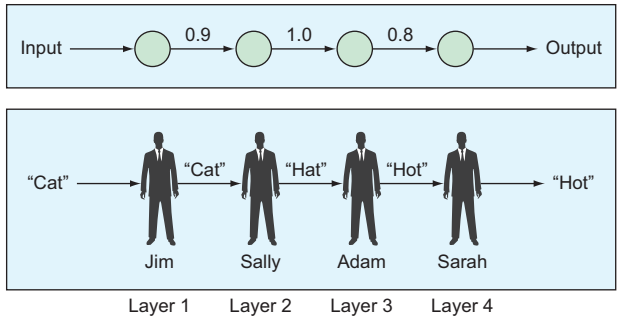


Figure 18.4 A game of Telephone like a neural network's transformations

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 output was from the expected output. Using our analogy of Telephone, this process is 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, the network can 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 to get the best predictive results. We won't get into much detail about hyperparameters, but you should know that they exist and that they typically come from heuristics as well as trial and error.

This explanation 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.

Understanding the concepts doesn't help you do anything, so you need to learn how to do real things with these machine-learning systems. How you do take this concept of a self-adjusting system and do something like figure out whether a cat is in an image? Many libraries make dealing with neural networks and other machine-learning concepts much easier than the diagrams shown earlier. One that 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 easier to express machine-learning concepts (and the underlying math) in code rather than in scary mathematical equations. It provides abstractions to track the different variables, utilities like matrix multiplication, neural network optimization algorithms, and various 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 (because entire books are devoted to this). But to move forward, you need to be familiar with TensorFlow, so let's look at a simple TensorFlow script that can make some predictions. We're not trying to teach you how to write your own TensorFlow scripts, so don't be scared if you don't follow exactly what's happening here. The point is 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 by handwritten numbers. Each image is a square of 28 pixels, and each data point has the image itself as well as the number represented in the image. These images are typically used as a beginner problem in machine learning because it contains both handwritten 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 figure 18.5.

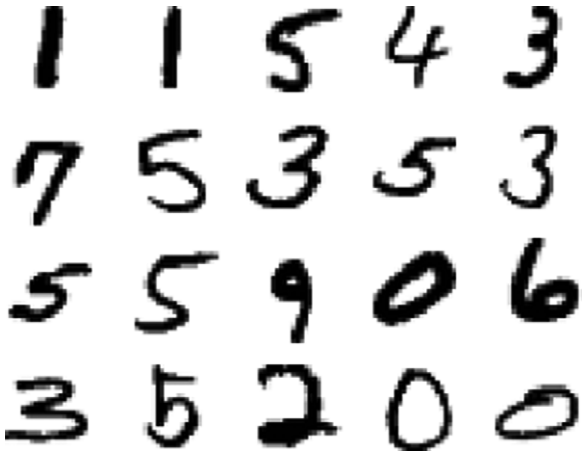


Figure 18.5 MNIST sample hand-written numbers

Because TensorFlow makes it easy to pull in these sample images, you'll use them to build a model that can take a similar image and predict what number is written in the image, as shown in listing 18.1. In a way, you're building a super-slimmed-down version of Cloud Vision's text recognition API, which you learned about in chapter 14. You script will train on the sample training data and then use the evaluation data to test how effective your model is at identifying a number from an image that wasn't used during the training.

Listing 18.1 Example TensorFlow script that recognizes handwritten numbers

```
import tensorflow as tf

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data/', one_hot=True)

# Learning model info
x = tf.placeholder(tf.float32, [None, 28*28])
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])
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)
sess = tf.InteractiveSession()
tf.global_variables_initializer().run()
```

Start by importing the TensorFlow library, which is installed by running pip install tensorflow.

TensorFlow comes with some example datasets, which you import and load into memory here.

Define the structure of your inputs, weights, and biases, and then your model (y), which is a bit like y = mx + b in algebra.

Next you need to measure how far the predicted output is from the "correct" output, which you call crossentropy.

Now that everything is defined, you have to tell TensorFlow to train the model by making adjustments that try to minimize the cross entropy you defined.

```
for _ in xrange(1000):
    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))
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)
```

Evaluate the model by inputting data from mnist.test and looking at how accurate the predictions are.

Finally, you print out the accuracy to see how you did.

To execute the training, you run through 1,000 iterations where at each step you input a new image and adjust based on being told what the correct answer was (this data is in mnist.train).

If you're intimidated by this script, even with the annotations, don't worry: you're not alone. TensorFlow can be complicated, and this example doesn't even use a deep neural network! If you were to run this script, you'd see that it's pretty accurate (over 90%):

```
$ python mnist.py
Successfully downloaded train-images-idx3-ubyte.gz
    ➥ 9912422 bytes.
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)
```

TensorFlow will automatically download all of the training data for you.

Here you can see the output is about 91%.

This script, as short as it is, has managed to recognize handwritten numbers with a 90% accuracy rate, which is pretty cool because you didn't explicitly teach it to recognize anything. Instead, you told it how to handle your input training data (which was an image of a number and the number), then gave it the correct answer (because all of the data is labeled), and it figured out how to make the predictions based on that. So what happens if you 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:

```
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)
```

By increasing the amount of training you see accuracy rise to above 92%.

There are three things important to notice:

- Because you already downloaded the MNIST dataset, you don't download it again.
- The accuracy went up by a couple of points (to 92%) by running more training iterations.
- It took longer to run this script!

If you change the number of iterations even further (say, to 100,000), you 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 presents a problem: How are you supposed to provide adequate training for your ML models, which 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 Machine Learning Engine?

As we've now seen, training machine-learning models can start out being pretty quick, but because 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, although our example was based on data that doesn't change (handwritten numbers typically don't change that often), it's not unusual for a machine-learning model you build to 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. As the data evolves, your machine-learning model should evolve as well, which would require that you retrain your model to get the most up-to-date predictions.

If you were to do this yourself with your computer, the demand for resources would probably end up looking something like figure 18.6, where every so often you need a lot of power to retrain the model, and the rest of the time you don't need 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!

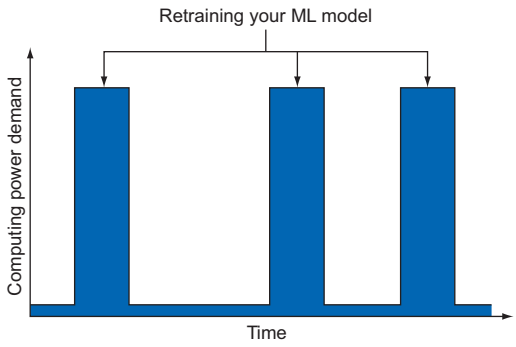


Figure 18.6 Spikes of demand for resources to retrain a machine-learning model

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. 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 your handwritten numbers example from earlier, you can send Cloud ML Engine something like your TensorFlow script, which you can use to train the model, and after that model is trained, you can send inputs to 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 more similar to the other hosted machine-learning APIs like the Vision API that you 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 the code into separate pieces, with different versions as things evolve over time. Let's dig into these different ways of organizing your work, starting with a word we've used quite a bit but never 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 is 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 the example script that recognizes handwritten numbers, the script itself 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. Because the test data is also labeled, you 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 you don't already know. As a result, the lifecycle of a model, shown in figure 18.7, 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.

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. Models are designed to ingest data of a

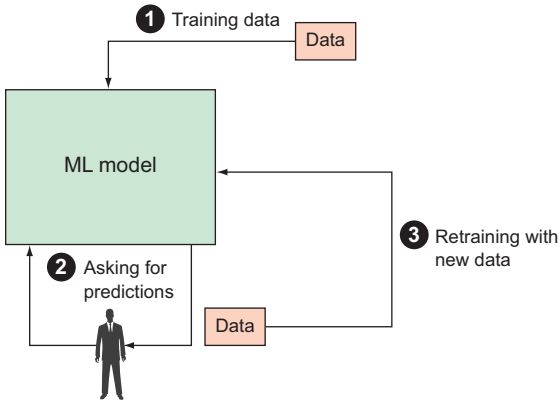


Figure 18.7 Lifecycle of a model

specific format. If you 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 earlier script that recognizes handwritten numbers, the model is designed to understand input data in the form of a grayscale bitmap image of a handwritten number. If you were to send it data in any other format (such as a color image, a JPEG image, or anything else), any results would 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, you'd likely see a bad guess for the number drawn or an error. On the other hand, if you were to use this invalid data during the training process, you'd likely reduce the overall accuracy because the model would be training itself on data that doesn't make much sense.

Coming back to the example of recognizing handwritten numbers, the model in your TensorFlow script was designed to handle a 28-by-28-pixel grayscale 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. The contract for the model you built previously could be thought of as the black box shown in figure 18.8.

Note that in my definition of a model, 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). 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. How would you distinguish between two models that fulfill the same contract but do so in different ways (maybe different training data, maybe different design)?

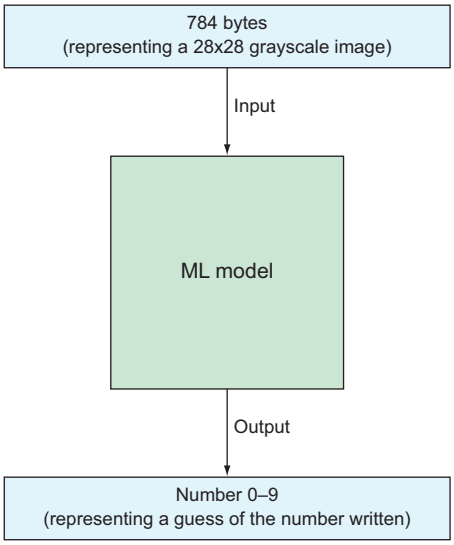


Figure 18.8 Machine-learning model that recognizes handwritten images

VERSIONS

Like a Node.js package, App Engine service, or shared Microsoft Word document, Cloud ML models can support different versions as the inner workings of a model evolve 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 you interact with is a version, but because a model has a default version, you can interact with the model itself, which implicitly means you're interacting with the default version. See figure 18.9.

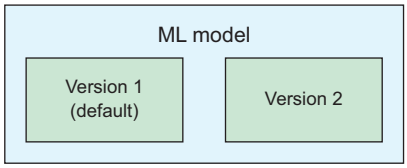


Figure 18.9 Models have many versions and one default version

Having the ability to create many versions of a model allows you to try lots of things when building it and test which of the configurations results in the best predictions for your use case. In the previous example, you might tweak lots of different parameters and see which of the versions is best at predicting the number written in an image. Then you 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. If you were to change the contract of the model (change the input or output formats), you'd 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. You could take the exact model code (similar to the TensorFlow script earlier), 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 haven't talked about where all of this data lives (both the training data and the data that defines the model version itself). Cloud ML Engine uses Google Cloud Storage to track all of the data files that represent the model and also as a staging ground where you can put data for training the model. You can read more about Cloud Storage in chapter 8, and we'll come back to this later, but for now it's sufficient to understand that a model version represents a specific instance of a model that you interact with by training it and using it to make predictions. How do you interact with these models? This is where jobs come into the picture.

Jobs

As you learned earlier, the two key distinguishing features of a model are the ability to be trained and the ability to make predictions based on that training. You also learned that sometimes the amount of data involved in things like training can be exceptionally large, which presents a bit of a problem because you wouldn't want to use an API call that has to upload 5 TB of training data. To deal with this, you rely on a "job," which is a way of requesting work be done asynchronously. After you start one of these jobs, you can check on the progress later and then decide what to do when it 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 the work. In addition, the work that the job does can be run on a variety of different configurations, which you 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 you'll use to interact with your models, whether it's training them to make predictions, making those predictions, or retraining them over time as you 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) fit together.

18.2.2 Putting it all together

Now that we've looked at all of the concepts that ML Engine uses, you need to understand how they get stitched together to do something useful. You've already learned that ML Engine stores data in Cloud Storage, but what does that 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. Models are like pointers to other data that lives in Cloud Storage, shown in figure 18.10.

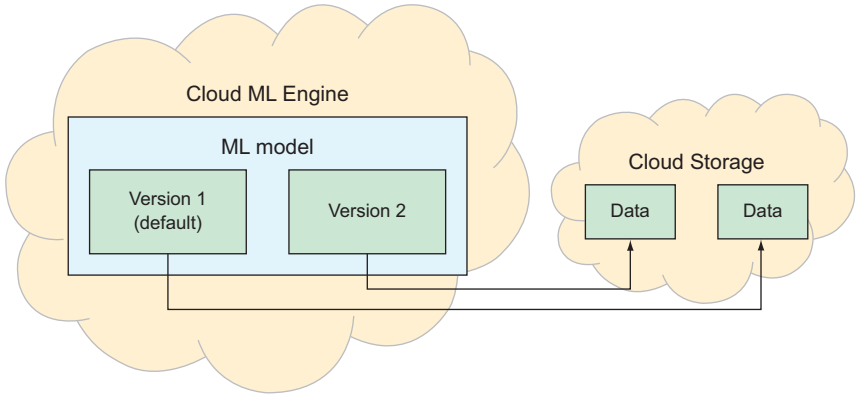


Figure 18.10 Model data is stored in Cloud Storage

How did the model data get there in the first place? As you learned previously, you interact with ML Engine using jobs, so to get model data stored in Cloud Storage, you'd use a training job. When you create the job, you'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 figure 18.11.

First (1), you'd upload the training data to Cloud Storage so that it's always available (you don't have to worry about your computer crashing in the middle of your training job). Next (2), you create a job in ML Engine asking for that data to be used

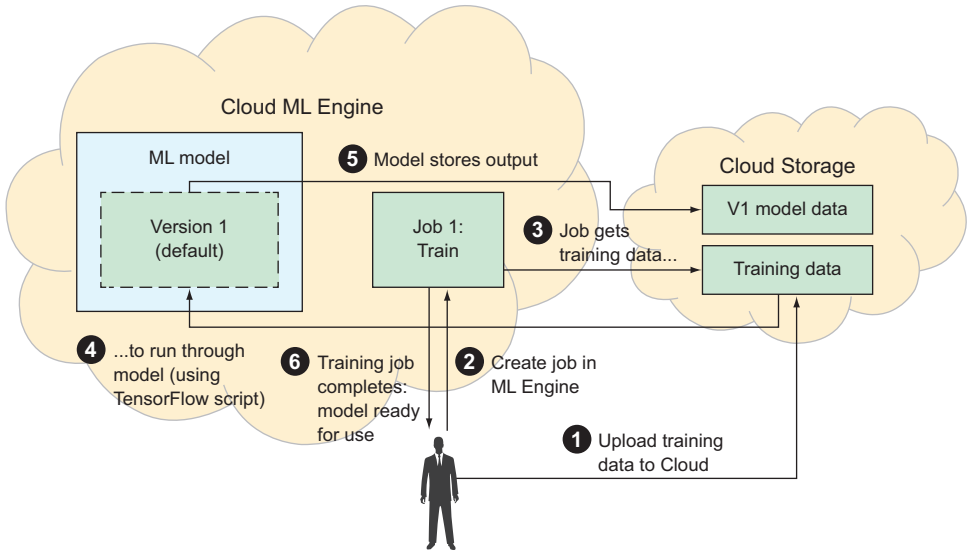


Figure 18.11 Flow of training a model

to train a version of your 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 you'd write (4). After the training is done (5), the mode would store its output back on Cloud Storage so that you can use it for predicting, and the job would complete (6) and let you know that everything worked. When this is all done, you'd end up with a trained model version in Cloud ML Engine with all the data needed being stored in Cloud Storage. After a model has been trained and is ready to make predictions, you can run a prediction job in a similar manner, as shown in figure 18.12.

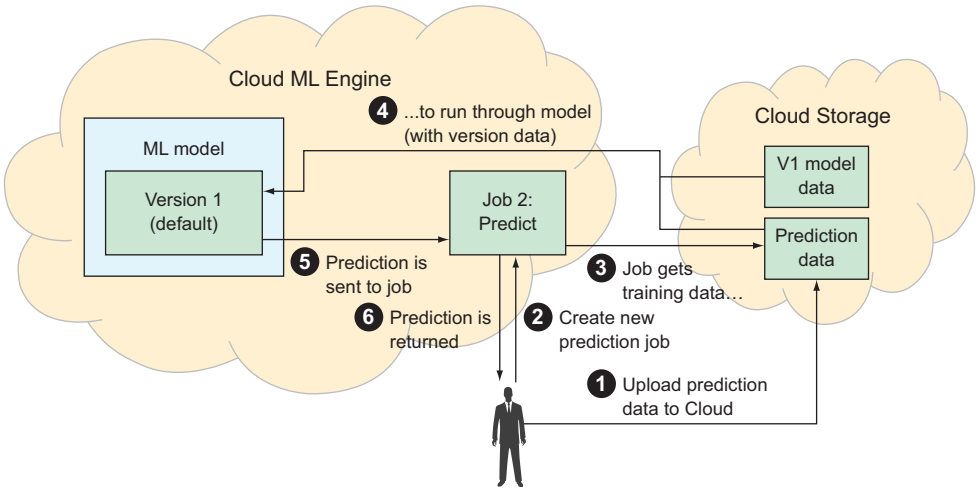


Figure 18.12 Flow of getting predictions based on a model

Like earlier, you'd start by uploading the data you want to make predictions on to Cloud Storage (1) so that it's always available. After that, you'd create a new prediction job on ML Engine (2) specifying where your 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 you (6) with all the details of what happened.

As you can see, the process of generating predictions using custom models is a lot more work than what you'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. If you have a problem that can be easily solved using the prebuilt machine-learning APIs, it's probably a better idea to use those. If you have a machine-learning problem that requires custom work, however, ML Engine aims to minimize the management work you'll

need to do to train and interact with models. Now that you've seen the flow of things when training a model and using it for predictions, let's take a look at what this looks like under the hood.

18.3 Interacting with Cloud ML Engine

To demonstrate the different work flows you learned earlier (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 complicated. 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 you can interact with it.

We're going to gloss over the internals of the model and the data involved and 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), you can find plenty of great books about machine learning out there as well as some about TensorFlow, which are definitely worth reading together with this chapter. Let's look at a common example using real-life data that you 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 a countrywide survey that's done every 10 years that asks general questions about the population such as ages, number of family members, and other basic data. In fact, this survey is how the United States measures the overall population of the country. This data is also available to the public, and you can use some of it to make some interesting predictions. The Census dataset itself is obviously huge, so we'll look at a subset, which includes basic personal information including education and employment details.

> **NOTE** All US Census data you'll use is anonymous, so you're never looking at an individual person.

What does this data look like? A given row in your dataset will contain things like an individual's age, employment situation (for example, private employer, government employer, and so on), level of education, marital status, race, income category (for example, less than or more than $50,000 annual income), and more. Some simplified rows are shown in table 18.1.

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

Table 18.1 Example rows from the US Census data *(continued)*

Age	Employment	Education	Marital status	Race	Gender	Income
38	Private	HS-grad	Divorced	White	Male	<=50K
53	Private	11th	Married	Black	Male	<=50K

You can use all the other data in a row to train a model that can then make predictions about income category based on the other information. You'll train a model that's able to predict whether a person makes more than $50,000 in a year based on their age, employment status, marital status, and so on. You could provide data that looks like table 18.2 and use your ML model to fill in the blanks.

Table 18.2 Example rows with missing data

Age	Employment	Education	Marital status	Race	Gender	Income
40	Private	Bachelors	Married	Black	Male	?
37	Self-emp	HS-grad	Divorced	White	Male	?

How do you do get Cloud ML Engine to fill in these question marks with a guess of what should be there? You'll start by creating a model.

18.3.2 Creating a model

As you learned previously, a model acts as a container of a prediction function that fulfills a specific contract. In this case, when you want to make a prediction, your 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, as shown in figure 18.13.

The process (1) starts by using complete Census data to train your model to predict the income category field based on the rest of the row. After you finish training the model, you 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). Because the model is

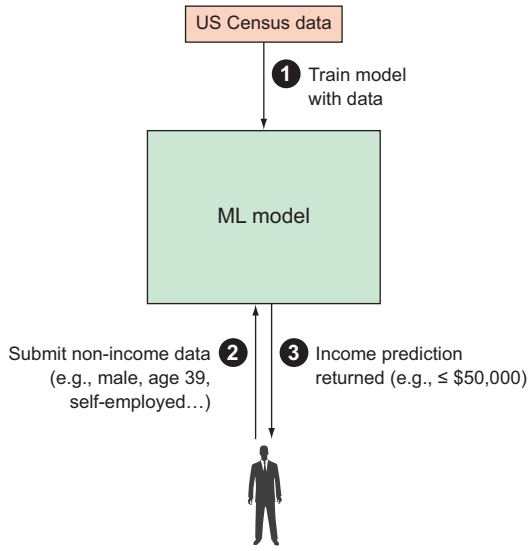


Figure 18.13 Overview of the model flow

only a container, you can create it using the Cloud Console. Choose ML Engine from the left-side navigation (it's under Big Data), and the screen shown in figure 18.14 opens where you can create a new model.

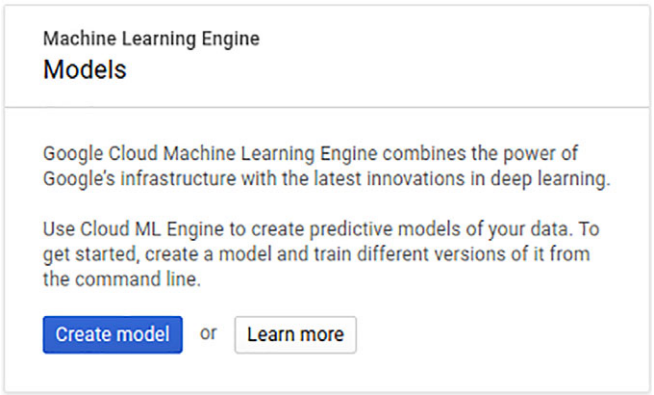


Figure 18.14 Prompt to create a new model

After you click Create model, the short form shown in figure 18.15 opens where you can name and describe the model. For your model you'll use the name `census`, which is how you'll uniquely identify the model from now on.

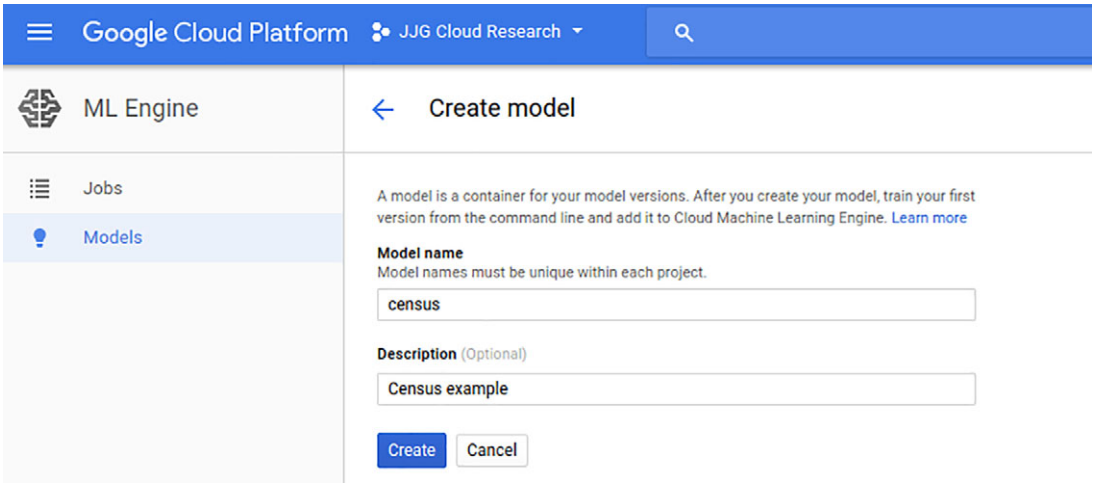


Figure 18.15 Creating your Census model

After you create the model, you can click it and see that there are currently no versions. You can also see this by using the `gcloud` command-line tool to list all models and versions for a given model, shown in the next listing.

Listing 18.2 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). You 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 you can do that, you need to get Cloud Storage set up with all the right code and data that you'll use for training.

18.3.3 Setting up Cloud Storage

Now that your `census` model exists, you have to train it to make some predictions. You'll need a bunch of US Census data to use for training purposes, and you'll need to make sure that the data lives in the right place in Google Cloud Storage. You can download some example data from the US Census dataset using the `gsutil` tool, as shown in the next listing. 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 5 first.

Listing 18.3 Downloading the US Census data set from Cloud Storage

```
$ mkdir data
$ gsutil -m cp gs://cloudml-public/census/data/* data/
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.
```

You'll put your data in a directory called data.

This command copies all of the files from a public bucket into the data directory.

Notice that this dataset is small to start (only about 6 MB), but it should still be able to help you make some reasonably accurate predictions. Also keep in mind that there are two datasets: a data and test. The first (`adult.data.csv`) is the data you'll use to train our model, and the second (`adult.test.csv`) is what you can use to evaluate your model.

Think of the first set as the data you'll use for learning, sort of like example problems that you work through with a teacher in school. The second dataset is more like the final exam at the end of the course where you figure out how well you did. It wouldn't make sense to give you the same problems that you'd already done in class, so these are some new ones that you haven't seen before. The next thing is to create a new bucket in Cloud Storage to hold your 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 you

want to send via a prediction job later on, but for now you'll use it for storing the US Census data.

> **NOTE** You may wonder why you don't rely only on the public Cloud Storage bucket to host the training data. In this example, you want to be sure that the data in question doesn't change out from under you, and the safest way to do that is to keep your own copy available in a bucket that you own and control.

You'll also need to make sure the bucket is located in a single region rather than distributed across the world. You do this to avoid cross-region data transfer costs, which could be large if you have a lot of data and are sending it from a multiregional bucket to your ML Engine jobs. If you had a lot of data stored in a bucket in Asia, for example, then a training job in the United States would involve sending all of that data across the world and back again with the final result. Even though this example is dealing with only a few megabytes, keeping the data near the resources that will do the training means you won't waste any money needlessly sending data all over the place.

For this example, you'll use the `us-central1` region as the home for your bucket as well as for resources you'll use for training later. You create this bucket using the `gsutil` command again, relying on the `-l` flag to indicate that you want your bucket to live in that specific location, as shown in the next listing.

Listing 18.4 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/...
```

After you have both the data you need and the bucket to hold it, you can upload the data using `gsutil` again, as the following listing shows.

Listing 18.5 Uploading a copy of the data to your 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 your bucket, which is located in the `us-central1` region, and we can start looking at how to define and train your model.

18.3.4 Training your model

Now that all the data is in the right place, it's time to start thinking about the code for your model and the job you'll use to train your model using that code and the data you previously uploaded. Start by downloading some of the code.

> **WARNING** As we discussed early in this chapter, the 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 your model is located on GitHub in the @GoogleCloudPlatform/cloudml-samplesrepository. You can clone the repository using `git`, or, if you're not familiar with Git, you can download it as a zip file from https://github.com/GoogleCloudPlatform/cloudml-samples. The example code we're interested in is located in the census directory. See the following listing.

Listing 18.6 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/
```

After you have the same code, you'll need to submit a new training job. As you learned earlier, jobs represent the way you schedule some work to be done that might take a while due to lots of data or computationally intense machine-learning code. Given the size and complexity of what you're trying to do, the training job itself shouldn't take that long. On the other hand, the command you'll need to run to start the training job is pretty complicated, so we'll walk through it piece by piece in the next listing.

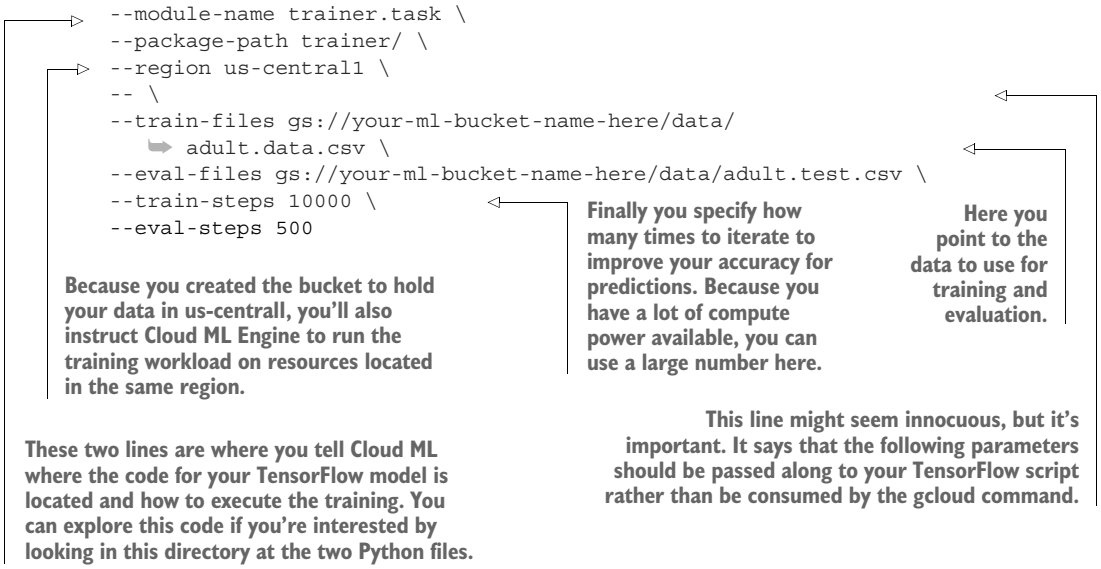
Listing 18.7 Command to submit a new training job

This is instructing Cloud ML Engine to use TensorFlow version 1.2.

Start by submitting a new training job for your census model.

When you run your job, you instruct Cloud ML Engine to put the various output data (the trained model data) in a specific place in Cloud Storage.

```
$ gcloud ml-engine jobs submit training census1 \
  --stream-logs \
  --runtime-version 1.2 \
  --job-dir gs://your-ml-bucket-name-here/census \
```



After running this, you should see quite a bit of output explaining the progress of training, but the whole process shouldn't take 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's trying to access the data (for example, `service-12345678989@cloud-ml.google.com.iam.gserviceaccount.com`).
>
> You can 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, you can use `gsutil` again because you instructed your job to put all of the output data into your Cloud Storage bucket, as shown in the next listing.

Listing 18.8 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/model.ckpt-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
gs://your-ml-bucket-name-here/census/export/variables/
```

This file (saved_ model.pb) is the important one because it contains the model that you can import and use for predictions.

To finish, you need to create a new model version based on the output of your training job. Because the output is located in census/export/saved_model.pb, you can do this using the Cloud Console by creating a new version and pointing it to that specific file. To do this, navigate to the Cloud ML Engine section in the Cloud Console and select your model. Inside that page you'll see some text, shown in figure 18.16, saying that the model currently has no versions yet, along with a link to create one.

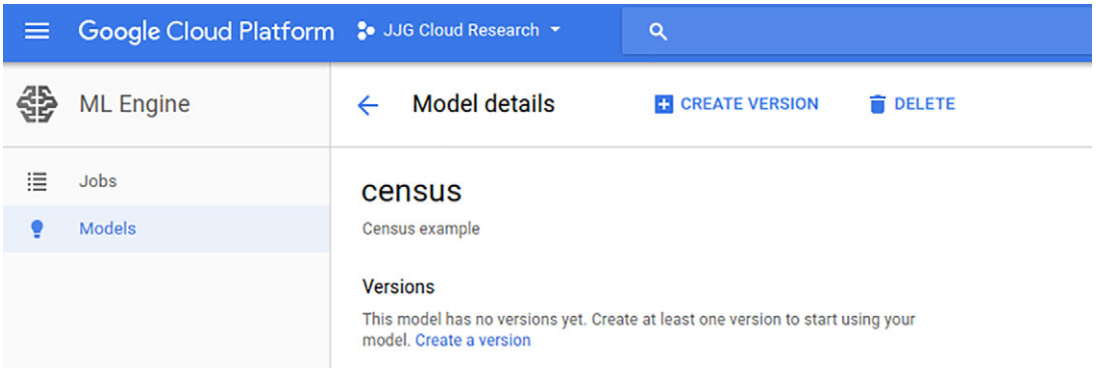


Figure 18.16 The `census` model without any versions yet

Clicking the link will show the form shown in figure 18.17 where you can name the version and choose where the data for the model version lives. Because this is your first version of the census model, use `v1` as the name for the version. As you saw earlier when listing the output from the training job, the model itself is located in the /census/export/ directory of your storage bucket.

After you set that, you can click Create to load the model version and automatically set it as the default version for your `census` model, which you can see by looking at the model details page, shown in figure 18.18.

Now that you finally have a trained model, let's look at how you can use it to make predictions and see how well it does at making them.

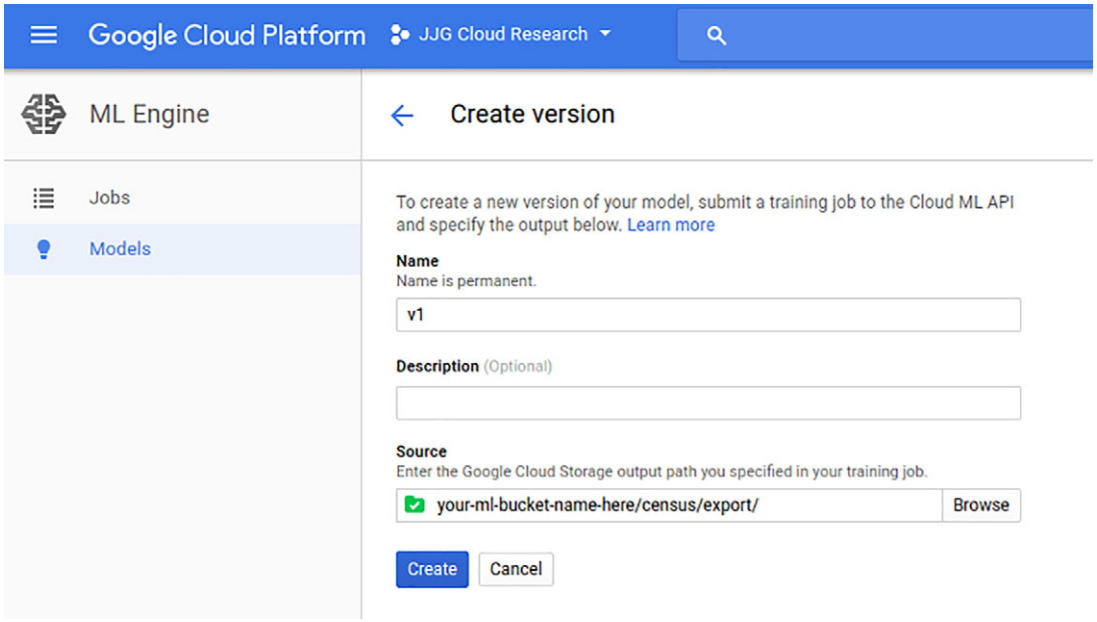


Figure 18.17 Creating a new version from your training output data

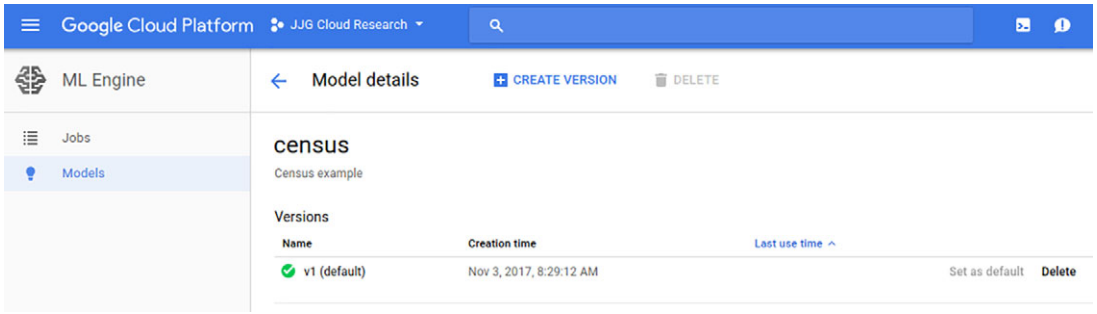


Figure 18.18 The `census` model with v1 as the default version

18.3.5 Making predictions

As you learned earlier, after a model is trained you can use it to make some predictions. In this case, you trained a model on US Census data targeting the "income category" field so that later you can send it details about a person and ask it to predict whether that person is likely to earn more than or less than $50k per year.

The way you do this depends on the number of predictions that you want to make at once. For example, if you want to make a prediction on a single row, you can send

the row directly to the model. If you have lots and lots of rows that you want predictions for, however, it's better to use a prediction job and put the input and output data on Cloud Storage, like you did with training. Let's start by looking at a single row and then move onto multiple rows using prediction jobs. To start, you'll need an incomplete row of data, missing the income category. The GitHub repository has some example data that you can use as a demonstration. Inside census/test.json you'll see a row of data representing a 25-year-old person. In table 18.3, 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 you were to run this data through your predictor, you'd get back some predictions as well as a confidence level, shown next:

```
$ gcloud ml-engine predict \
  --model census --version v1 \
  --json-instances test.json
CONFIDENCE  PREDICTIONS
0.78945      <=50K
```

Here you request using the census model that you created.

In this case you specify a path to the JSON data in a local file.

Your model returns an output made up of a prediction of an income category along with a confidence level.

As you 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 you were to change the age of this same person to 20 years old (instead of 25), the confidence level would go up that the person is earning less than $50k annually, shown next:

```
$ gcloud ml-engine predict --model census --version v1 --json-instances
     ../test2.json
CONFIDENCE  PREDICTIONS
0.825162     <=50K
```

What if you had a lot of instances that you 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 training your model. You'll first upload the data you want to get predictions for to Cloud Storage, and then submit a prediction job asking ML Engine to pull that data and place the output predictions into another location on Cloud Storage. You can use the same file (test.json) again, but modify it to add a

few more rows. In this example, you'll reproduce the same rows and increase the age by 5 years for each row. If you go from 25 up to 65, you'll have 10 rows that you want to make predictions for. First, upload the file to Cloud Storage, shown in the next listing.

Listing 18.9 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 you can submit a prediction job pointing to the uploaded data and ask the output to be placed in a different location, as shown in the following listing.

Listing 18.10 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
```

After the job completes you can look at the output, which will live on Cloud Storage in the prediction1-output directory of your bucket:

You can see the files that resulted by listing the contents of the directory.

```
$ gsutil ls gs://your-ml-bucket-name-here/prediction1-output
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
{"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"}
```

You can print the output of the result using the cat subcommand of gsutil.

As you can see in 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. Your model is more confident about its predictions for a younger person than for an older person. Now that you've seen how to make predictions (both directly and using

a prediction job), you should take a step back and look at what's happening under the hood when interacting with your models.

18.3.6 Configuring your underlying resources

In the jobs that you've run so far, we sort of glossed over the whole idea that there were some computers somewhere doing the computational work. For example, when you submitted your training job, we never discussed anything about the VMs that pulled down the data and the CPU cycles consumed when you 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 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 a predefined configuration of computing resources that are likely to do a good job of handling your training workload. The default scale tiers are a good guess for the typical work done in a machine-learning job.

These configurations have a few pieces that result in different performance profiles. First is the concept of a worker server, which is like a VM that does the computation needed to train a model. Next, if multiple workers exist, you 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 that these servers are responsible for coordinating the efforts of the various worker servers. Finally, these servers can have 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 follow:

- `BASIC`, which is a single worker server that trains a model
- `BASIC_GPU`, which is a single worker server that comes with a GPU attached
- `STANDARD_1`, which uses lots of worker servers but has a single parameter server
- `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, figure 18.19 shows how these different preset scale tiers look.

Setting a specific scale tier is easy: use the `--scale-tier` flag when submitting your training job. If you don't set a scale tier or any other configuration, ML Engine will use the `BASIC` scale tier. For instance, in the earlier example you 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 it's not good if you have a lot of data or a particularly

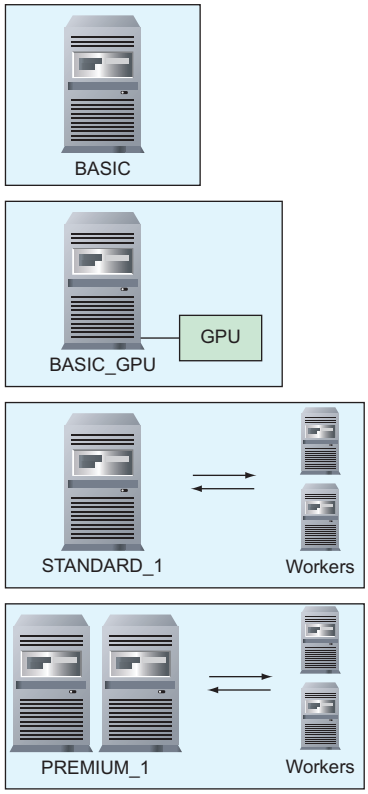


Figure 18.19 Various scale tiers

complex model. If you wanted to configure this explicitly, the command would look something like the following listing.

Listing 18.11 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 \
  -- \
  --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
```

You 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 when you can take advantage of hardware acceleration because the single server will have an NVIDIA Tesla K80 GPU attached.

The next two tiers (`STANDARD_1` and `PREMIUM_1`) are the only ones recommended for real production workloads because they're 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 with one key difference. When there are multiple worker servers, each 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. Workers rely on a parameter server to be the central authority for the cluster of worker servers, which means a bottleneck could result where a parameter server is overwhelmed by all the workers. The `STANDARD_1` tier has 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 supports lots of parameter servers to avoid 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, will likely be the case), ML Engine provides the ability to customize the hardware configuration to the specifics of your jobs. Table 18.4 shows the different machine types that you can use, but before we look at that, there are a few things to note.

First, notice you have only two choices for configuring parameter servers: `standard` and `large_model`. A parameter server 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, you have some reference amount of capacity and larger machine types have rough multiples of that reference amount. Instead of defining the specific amount of memory available from one machine type to the next, you can think of the next step up being roughly twice as much of a resource. For example, the `complex_model_m` (medium) machine type is about twice as much CPU and memory as the `complex_model_s` (small) machine type. See table 18.4.

Table 18.4 Summary of the different machine types

Machine type	Best for	CPU	Memory	GPUs
`standard`	All servers	1x	4x	None
`standard_gpu`	Worker servers	1x	4x	1x K80

Table 18.4 Summary of the different machine types *(continued)*

Machine type	Best for	CPU	Memory	GPUs
`standard_p100`	Worker servers	1x	4x	1x P100
`large_model`	Parameter servers	2x	16x	None
`complex_model_s`	Worker servers	2x	2x	None
`complex_model_m`	Worker servers	4x	4x	None
`complex_model_m_gpu`	Worker servers	4x	4x	4x K80
`complex_model_m_p100`	Worker servers	4x	4x	4x P100
`complex_model_l`	Worker servers	8x	8x	None
`complex_model_l_gpu`	Worker servers	8x	8x	8x K80

How do you use these machine types in your training jobs? Instead of passing all of this information about your underlying resources in the form of command-line arguments, you 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 18.12 Job configuration file

```
trainingInput:
  scaleTier: CUSTOM
  masterType: standard
  workerType: standard_gpu
  parameterServerType: large_model
  workerCount: 10
  parameterServerCount: 2
```

Here you clarify that you want a custom scale tier rather than one of the presets.

You can set the types of the various servers (master, workers, and parameter servers) to anything you want. Here you've used standard for the master, standard_gpu for the worker, and large_model for the parameter server.

In addition to controlling the type of the machine, you can also control how many of each you deploy. The master is always a single server, but you can add more workers and more parameter servers as well. In this example, you use 10 workers and 2 parameter servers.

If you save this information to a file (say, job.yaml), you can then submit a new training job where you leave everything else as before except you don't specify a scale tier and instead refer to the configuration file, as shown in the next listing.

Listing 18.13 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 \
-- \
--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
```

Instead of setting a scale tier, you point the command-line tool at the configuration file that you created earlier.

Now that you've seen how to change the underlying resources for your 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, only one type of server is in use: workers. Unlike training jobs, a prediction job doesn't offer a way to modify the type of machine involved, which means you only ever think in terms of "how many" rather than "of what type." 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 because an element of automatic scaling exists, based on the amount of work submitted in the job. For example, a small job of a few predictions, as you tried previously, might not benefit from more than one node running at a time. A large job of millions of predictions, however, will likely complete more quickly with lots of workers.

As a result, ML Engine will continue to turn on new workers until it either reaches the defined limit or workers run out of work. This ability allows ML Engine to optimize for the fastest completion time of any prediction job within some reasonable limitations. You can easily control this limit by setting the `--max-worker-count` flag. For example, the following snippet shows how you could modify your previous prediction job to use no more than two workers.

Listing 18.14 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 \
  --input-paths gs://your-ml-bucket-name-here/data.json \
  --output-path gs://your-ml-bucket-name-here/prediction2workers-output
```

Here you can set the maximum number of workers to use to two.

This method leaves the open question of how many nodes are used during an online prediction request made directly rather than as part of a batch job. 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 to minimize latency on incoming prediction requests. 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 online prediction requests is scaled entirely automatically, so there's nothing for you to do besides send requests as you

need them. It's important to remember that online prediction shouldn't be used as a replacement for batch prediction. Online 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

ML Engine has two distinct operations that it supports (predicting and training), so there are two different pricing schemes for each of these. Because training is the more complicated of the two, let's start by looking at how much it costs, and then we'll move on to 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, but with a few important differences. First, the table of machine types never 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" of others. Second, we never clarified what types of machines were in use when using one of the preset scale tiers. How does all of this work?

All of the pricing for ML Engine boils down to ML training units consumed, which have a price per hour of use. This price can be chopped into one-minute increments to pay for only what you consume, but like Compute Engine, there's a 10-minute minimum to deal with overhead. 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 pay for exactly 15 minutes. How do you 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 you learned, computing time is measured in ML training units, which themselves have an hourly cost. 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. Table 18.5 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 light on resources, which is why they're much cheaper than others like PREMIUM, which is an order of magnitude more power (and cost).

In the earlier example training job, where you used the default BASIC scale tier, you ended up paying $0.49 per hour because your job was run in the us-central1 region. Assuming you fell under the 10-minute minimum charge, that simple job cost about 8 cents (10 minutes / (60 minutes per hour) * $0.49 per hour = $0.08167). What about the custom deployments that you learned about? Let's look in more detail at the pricing for customized resources.

MACHINE TYPE–BASED PRICING

Like scale tiers, each machine type comes with a cost defined in ML training units, which then follows the same pricing rules that you already learned about. Table 18.6 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 earlier example and calculate how much it would cost on an hourly basis. Recall that in the previous example configuration file we customized the types of all the different machines and set specific numbers of servers. Table 18.7 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
Total	37		

Your example configuration consumes a total of 37 ML training units, which at US-based prices would be $18.13 per hour. Assuming your job completed quickly (under the 10-minute minimum), the job itself would have cost about three dollars (10 minutes / (60 minutes per hour) * $18.13 per hour = $3.02167).

Ultimately, calculating this each time is going to be frustrating. Luckily you can jump right to the end by looking at the job itself either in the command line or the Cloud Console, where you can see the number of ML training units consumed in a given job. Shown in figure 18.20 is the Cloud Console for your example training job.

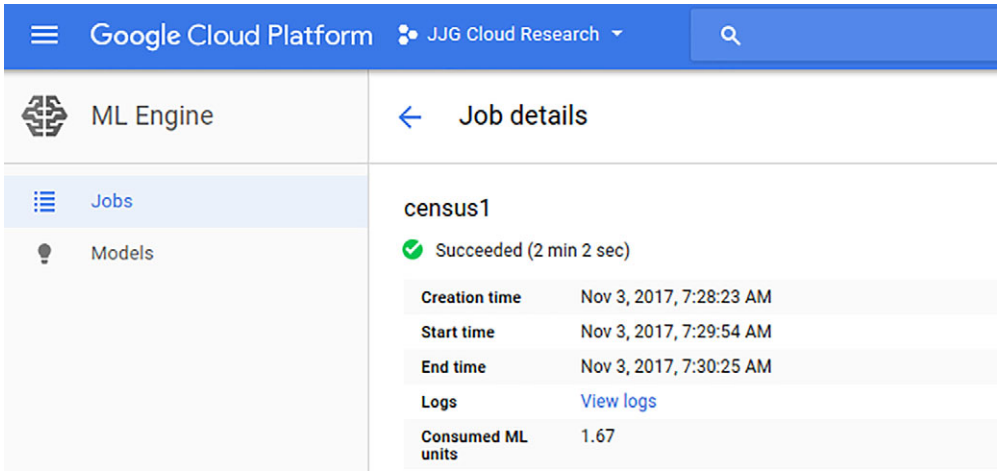


Figure 18.20 Looking at the details of a training job in the Cloud Console

You can also see the same information in the command line by using the `describe` subcommand to request the details of a job. The next listing shows the same information about the job in the command line.

Listing 18.15 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
```

Here you can see that this consumed 1.67 ML training units.

18.4.2 Prediction costs

As you've learned, predicting consumes resources like training, but the prediction work is done entirely by prediction nodes. Although these nodes act like the others, you can't customize them and have much less control over how many of them are running at any given time. As a result, and 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. If you end up consuming 5 minutes' worth of 10 prediction nodes' resources, the cost would come out to about $0.33 (`$0.40 per hour * 5 minutes / 60 minutes per hour * 10 nodes`).

Unlike for training jobs, a flat rate of $0.10 per 1,000 predictions ($0.11 for non-US locations) is available in addition to the hourly-based costs. Further, this cost per prediction applies the same to individual online predictions as well as to each individual prediction in a batch job. Following your earlier example, if a five-minute prediction job covered 10,000 data points, the per-prediction fee would be $1.00 (`$0.10 per chunk of 1,000 predictions * 10 chunks`), bringing the overall cost for the prediction job to a grand total of about $1.33.

At this point you should have a good grasp of 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 as you did with training jobs. The next listing shows the view of your prediction job in the command line where you can clearly see how many prediction node hours were consumed as well as how many predictions were performed.

Listing 18.16 Viewing the details of a prediction job.

```
$ gcloud ml-engine jobs describe prediction1
# ... More information here ...
predictionOutput:
  nodeHours: 0.24
  outputPath: gs://your-ml-bucket-name-here/prediction1-output
  predictionCount: '10'
startTime: '2017-11-03T14:15:41Z'
state: SUCCEEDED
```

Here you can see that you consumed 0.24 prediction node hours.

In this case, those node hours went toward making 10 predictions.

Based on this information, this job cost $0.0001 for the predictions themselves (`10 predictions / 10,000 predictions per chunk * $0.10 per chunk`) and $0.096 for the node hours consumed (`0.24 node hours * $0.40 per hour`), meaning a grand total of $0.0961, which rounds to about 10 cents.

Summary

- Machine learning is the overarching concept that you 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 (for example, 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 important ever since Big Data became a buzz word. As you might guess, processing and analyzing loads of data (measured in terabytes, petabytes, or more) is a 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 quickly, and then move onto Cloud Dataflow where you can take your Apache Beam data-processing pipelines and execute them on Google's infrastructure. Last, 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

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 (for example, 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, recompile, redeploy, and run the job again. This leads us to BigQuery.

19.1 What is BigQuery?

BigQuery is a relational-style cloud database that's capable of querying enormous amounts of data in seconds rather than hours. Because BigQuery uses SQL instead of Java or C++ code, exploring large data sets is both easy and fast. 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. Although

BigQuery is capable of running traditional OLTP-style queries (for example, `UPDATE table SET name = 'Jimmy' where id = 1`), it's most powerful when you use it as an analytical tool for scanning, filtering, and aggregating lots and lots of rows into some meaningful summary data.

19.1.1 Why BigQuery?

You may understand what BigQuery is and what it's used for, but you may be confused about why you might use BigQuery instead of some of the other systems out there. For example, why can't you just use MySQL to explore your data? You can use MySQL for most cases, but as you have to scan over more and more data, MySQL will become overloaded, and performance will degrade. When that happens, it makes sense to start exploring other options.

First you might try to tune MySQL's performance-related parameters so that certain queries run faster. Then you might try to turn on read-replicas so you aren't running super-difficult queries on the same database that handles user-facing requests. Next you might look at using a data warehouse system like Netezza, but the price for those systems can be high (usually millions of dollars), which could be more than you're willing to pay. 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 provides some of the power of traditional data warehouse systems while only charging for what you use. Let's take a quick look at how it works under the hood so you can see why BigQuery can handle scenarios where something like MySQL may struggle.

19.1.2 How does BigQuery work?

You could write an entire book on BigQuery and its underlying technology (and someone has), so this section will cover the inner workings of BigQuery in only a general way. This chapter will be a bit light on underlying theory and advanced concepts and instead will focus on practical usage of BigQuery in an application. If you're interested in 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 (Wiley, 2014).

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). How can BigQuery do what MySQL can't do? Let's start by looking at the problem's two parts. First, if you need to filter billions of rows of data, you need to do billions of comparisons, which require a lot of computing power. Second, you need to do the comparisons on data that's stored somewhere, and the drives that store that data have limits on how quickly it can flow out of them to the computer that's doing those comparisons. Those two problems are the fundamental issues you need to solve, so let's look at how BigQuery tries to address each problem, starting with computing capacity.

Scaling computing capacity

People originally tackled the computation aspect of this problem by using the MapReduce algorithm, where data is chopped into manageable pieces (the map stage) and then reduced to a summary of the pieces (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 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 long time. Using MapReduce, you could speed this up by using 1,000 computers, with each one responsible for counting one one-thousandth of the rows, and then summing up the 1,000 separate counts to get the full count (figure 19.1).

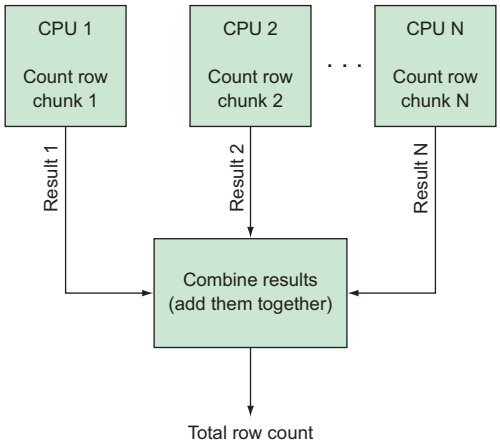


Figure 19.1 Counting a few billion rows by breaking them into chunks

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. BigQuery is 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. Although you sometimes take those disks for granted, they become incredibly important when you start demanding extreme performance out of them. Sometimes you 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 you need 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 and longer. Although the storage capacity of disks has been growing, their bandwidth hasn't necessarily kept up.

When we solved the computational capacity problem by splitting the problem up into many chunks and using lots of CPUs to crunch on each piece in parallel, we never thought about how we'd make sure all of the CPUs had access to the chunks of data. 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 compounded by the fact that the total amount of data you need to query is potentially enormous.

To make this more concrete, most drives, regardless of capacity, typically can sustain hundreds of megabytes per second of throughput. At that rate, pulling all the data off of one 10-terabyte (TB) drive (assuming a 500 MB/s sustained transfer rate) would take about five hours! If 1,000 CPUs all asked for their chunk of data (1,000 chunks of 10 GB each), it'd take about five hours to deliver them, with a best case of about 20 seconds per 10 GB chunk. The single disk acts as a bottleneck because it has a limited data transfer rate.

To fix this, you could split the database across lots of different physical drives (called *sharding*) (figure 19.2) so that when all of the CPUs started asking for their chunks of data, lots of different drives would handle transferring them. No drive alone would be able to ship all the bytes to the CPUs, but the pool of many drives could ship all that data quickly. For example, if you were to take those same 10 TB and split them across 10,000 separate drives, 1 GB would be stored on each drive. Looking at the *fleet* of all the drives, the total throughput available would be around 5,000,000 MB/s (or 5 *TB*/s). Also, each drive could ship the 1 GB it was responsible for in around two seconds. If you followed the example with 1,000 separate CPUs each reading their 10 GB chunk (one one-thousandth of the 10 TB), they'd get the 10 GB in two seconds—each one would read ten 1 GB chunks, with each chunk coming from one of 10 different drives.

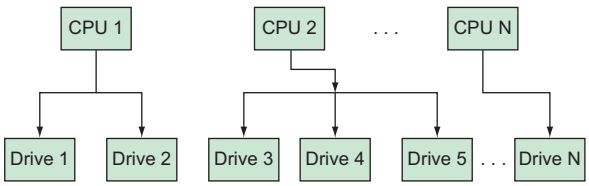


Figure 19.2 Sharding data across multiple disks

As you can see, sharding the data across lots of drives and transporting it to lots of CPUs for processing allows you to potentially read and process enormous amounts of data incredibly quickly. Under the hood, Google is 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 to use it.

19.1.3 Concepts

As you learned already, BigQuery is incredibly SQL-like, so I can draw close comparisons with 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

Like a relational database has databases that contain tables, BigQuery has *datasets* that contain tables (figure 19.3). The datasets mainly act as containers, and the tables, again 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 although datasets act as collections of tables, you have less control over the technical aspects of those tables than you would with a system like MySQL or PostgreSQL.

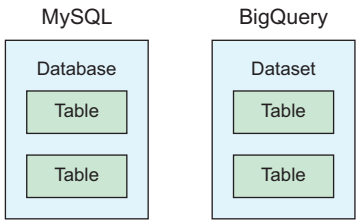


Figure 19.3 A BigQuery dataset and tables compared to a MySQL database and tables

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's a bit more complicated than that when a particular column allows nested or repeated values, but I'll dig into that in more detail later when we look at schemas.

Unlike in a traditional relational database, BigQuery rows typically don't 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. Because BigQuery is intended to be used as an analytical storage and querying system, constraints like uniqueness in even a single column aren't available. This also means that, although data isn't technically immutable, so you *can* change it, because there's no way to deduplicate rows, it's not possible to guarantee that if you request to update data in BigQuery, it will only address the exact row you intended. Otherwise, BigQuery will accept most common SQL-style requests, like `SELECT` statements; `UPDATE`, `INSERT`, and `DELETE` statements with potentially complex `WHERE` clauses; and fancy `JOIN` operations.

Before we move on, I wanted to mention one other interesting BigQuery capability related to tables. Usually with a database, you start by loading data into it, and then later you run queries over the data you put there. Because BigQuery is part of Google Cloud Platform, you can transfer the querying power from BigQuery over to other

storage services. 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. With that knowledge of BigQuery tables in hand, let's look at the schemas that define their structures.

SCHEMAS

As with other SQL databases, BigQuery tables have a structured schema, which in turn has the standard data types you're used to, such as `INTEGER`, `TIMESTAMP`, and `STRING` (sometimes known as `VARCHAR`). Additionally, as with a regular relational database, fields can be required or nullable (like `NULL` or `NOT NULL`). Unlike with a relational database, you define and set schemas as part of an API call rather than running them as a query. Whereas in MySQL you'd execute a query starting with `CREATE TABLE` to define the table's schema, BigQuery doesn't use SQL for requests related to the schema. Instead, you send those types of queries 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. You can represent the schema itself as a list of JSON objects, each with information about a single field. In the following example listing, 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 straightforward, but things get a bit more complicated with some of the other modes and field types. To start with, there's an additional mode called `REPEATED`, which currently isn't common in most relational databases. Repeated fields do as their name implies, taking the type provided and turning it into an array equivalent. A repeated `INTEGER` field acts 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 as long as a single entry of the field matches a given value. Although these methods are nonstandard, they shouldn't feel completely out of place if you think of each row in BigQuery as a separate JSON object.

Next, a field type called `RECORD` 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). Using `RECORD` types like this isn't a common pattern in

standard SQL, where it would be normalized into a separate table (a table of `books`, and the `favorite_book` field would be a foreign key). In BigQuery, this type of inlining or denormalizing is supported and can be useful, particularly if the data (in this case, the book title and author) is never needed in a different context—it's only ever looked at alongside the people who have the book as a favorite.

I'll demonstrate how some of these modes and types work a bit later, but the important thing to remember here is that BigQuery has two nonstandard 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 similar to other relational databases. Next, let's look at the concepts involved in interacting with BigQuery, starting with jobs.

Jobs

Because API requests to BigQuery tend to involve lots of data, it's likely that although a single request will finish quickly, it probably won't finish right away—it may take a few seconds at least. After all, it's difficult to load a terabyte of data into a storage system in a few milliseconds. As a result, BigQuery uses jobs to represent work that will likely take a while to complete.

Instead of making a call to load some data (which might look like `bigquery.loadData('/path/to/1tb_of_data.csv')`), you create a semipersistent resource called a *job* that's 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 (for example, something like `job = bigquery.createJob('SELECT … FROM table WHERE …')`). What can these jobs do? You can accomplish four fundamental operations with jobs:

- Querying for data
- Loading new data into BigQuery
- Copying data from one table to another
- Extracting (or exporting) data from BigQuery to somewhere else (like Google Cloud Storage [GCS])

Although these operations each seem to be doing entirely different things, they're 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, because 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. As a result, you may have several different ways to accomplish the same thing, but all of them use jobs to keep track of work being done.

Finally, because jobs are treated as unique resources, you can perform the typical operations over jobs that you can over things like tables or datasets. For example, you can list all of the jobs you've run, cancel any currently running jobs, or retrieve details

of a job you 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 use BigQuery, starting with querying some shared datasets.

19.2 Interacting with BigQuery

BigQuery, like any other hosted database, is accessible via its API, so you 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. (I'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 you'll start off by trying out some queries. You can kick things off by going to the Cloud Console and choosing BigQuery from the left-side navigation menu. Unlike the APIs you've used so far, you'll be brought to a new page (or tab) focused exclusively on BigQuery. If you then click Public Datasets, you'll land on a page showing off a bunch of these datasets (figure 19.4).

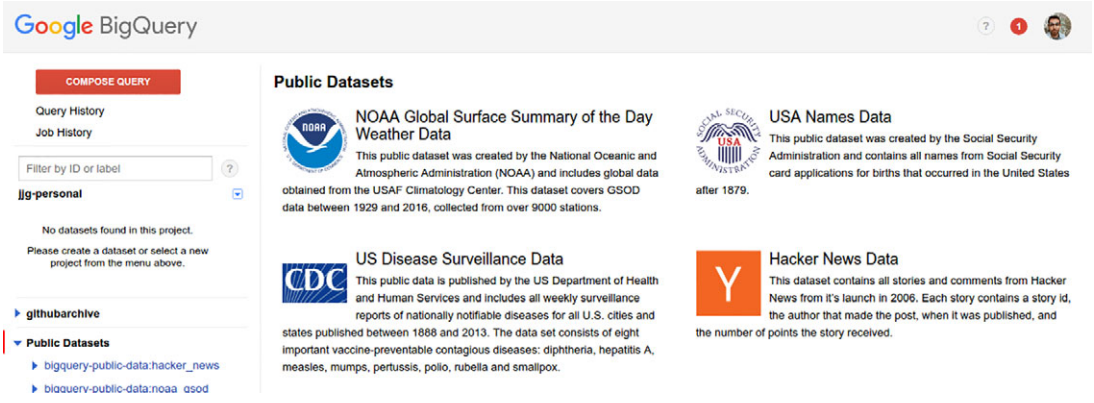


Figure 19.4 BigQuery's public datasets

If you click one of the choices (in this case, try out the yellow taxi dataset), BigQuery will bring you to a summary of the data, which includes both some details of the dataset itself and a list of the tables. If you click on the tables, BigQuery will bring you to a page that shows the most important piece: the schema (figure 19.5).

Here you can see the list of fields available, their data types, and a short description of the data that lives in each field. Notice 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,

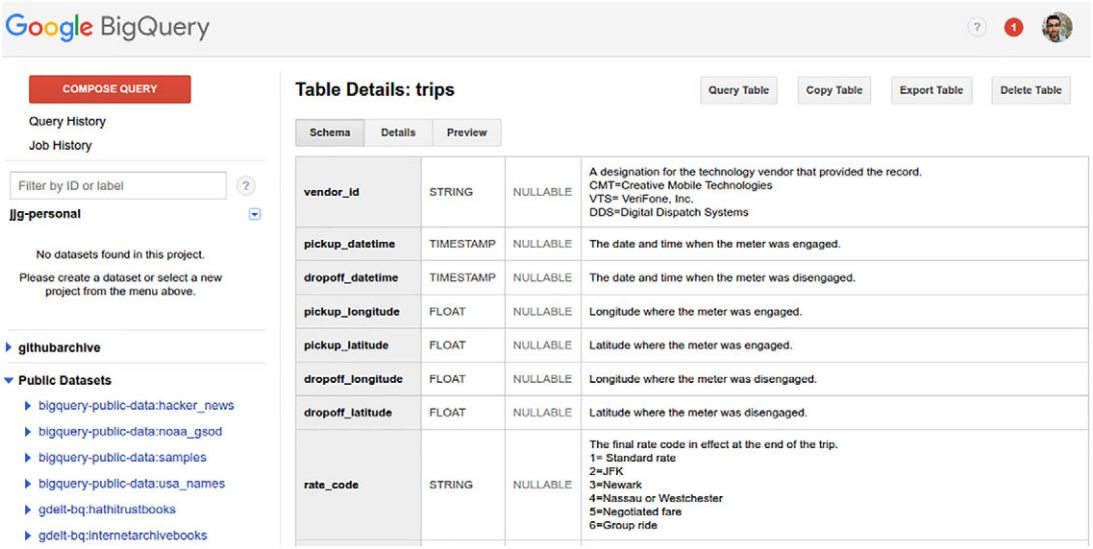


Figure 19.5 The yellow taxi trips schema

you'll be able to see an overview of the table, which in this case shows that it contains about 130 GB in total, spread across over a billion rows. In figure 19.6, you can see the complete Table ID, which is a combination of the project (in this case, `nyc-tlc`), the dataset (`yellow`), and the table (`trips`). Keep this in mind as you run into this format when writing queries.

You can run a few queries that would be interesting, but often the initial worry is "Won't this take a few minutes?" That's a reasonable first thought—after all, querying

Table Info

Table ID	nyc-tlc:yellow.trips
Table Size	130 GB
Long Term Storage Size	130 GB
Number of Rows	1,108,779,463
Creation Time	Sep 25, 2015, 2:29:01 PM
Last Modified	Dec 24, 2015, 10:34:53 AM
Data Location	US
Labels	None Edit

Figure 19.6 The yellow taxi trips table details

1.1 billion records in PostgreSQL would probably take a while—so try starting with a query that any other database could probably handle easily as long as there was an index: the most expensive ride.

To run this query over the table, click the Query Table button at the top right and enter the following:

```
SELECT total_amount, pickup_datetime, trip_distance
  FROM `nyc-tlc.yellow.trips`
  ORDER BY total_amount DESC
  LIMIT 1;
```

In case you're not familiar with SQL, this query asks the table for some details sorted by the total trip cost but only gives you 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 (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 unchecking the Use Legacy SQL box.

When you click Run Query, BigQuery will get to work and should return a result in around two seconds. (In my case, it was 1.7 seconds.) It'll also show you how much data it queried in that time, which in my case was about 25 GB, meaning that BigQuery sifted through about 15 GB per second to give back this result (figure 19.7). That's 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.)

This seems interesting but doesn't show off the power of BigQuery, so you should try something a bit more intricate. You have pickup and drop-off times and locations,

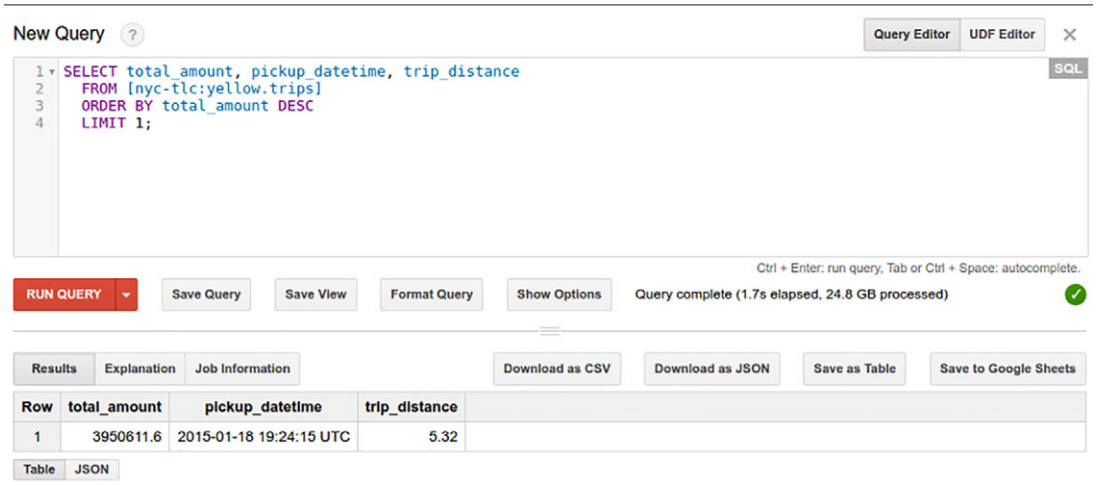


Figure 19.7 BigQuery results of the most expensive trip

so what if you were trying to figure out what was the most common hour of the day that people were picked up? You'd 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 isn't that complicated:

```
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 pickups are most common (6–10 p.m.) and the early morning pickups are least common (3, 4, and 5 a.m.). See figure 19.8.

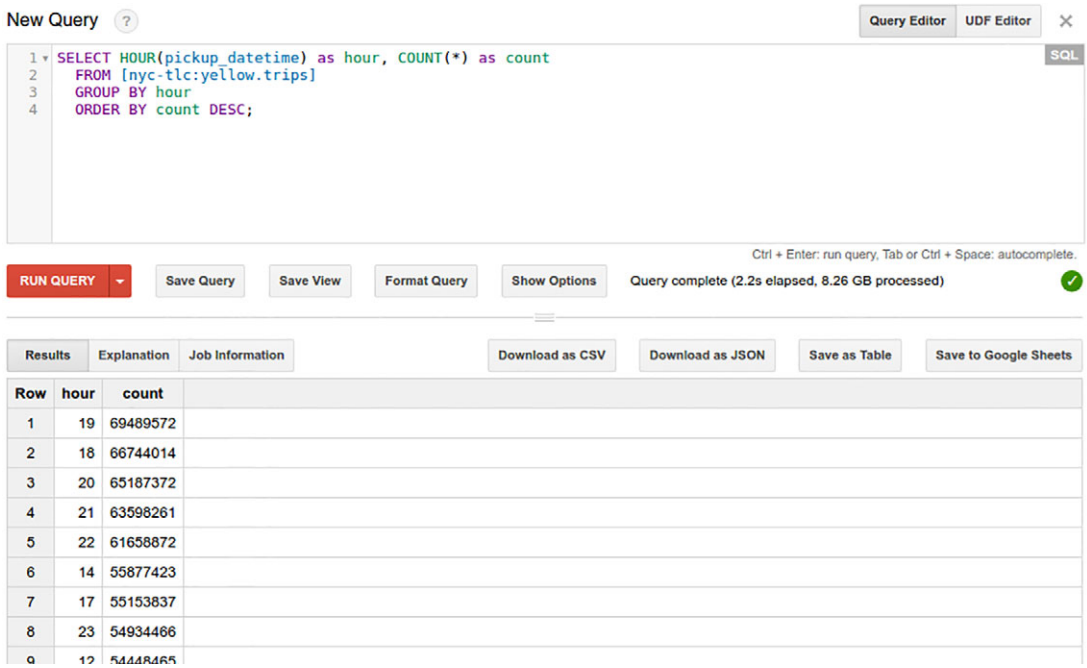


Figure 19.8 Results of querying with a grouping by pickup time

Perhaps you might find more information here if you look at it broken down by the day of the week. Try adding that into the mix:

```
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 query shows that the evening hours are most popular toward the end of the week. (Thursday and Friday at 7 p.m. top the charts.) See figure 19.9.

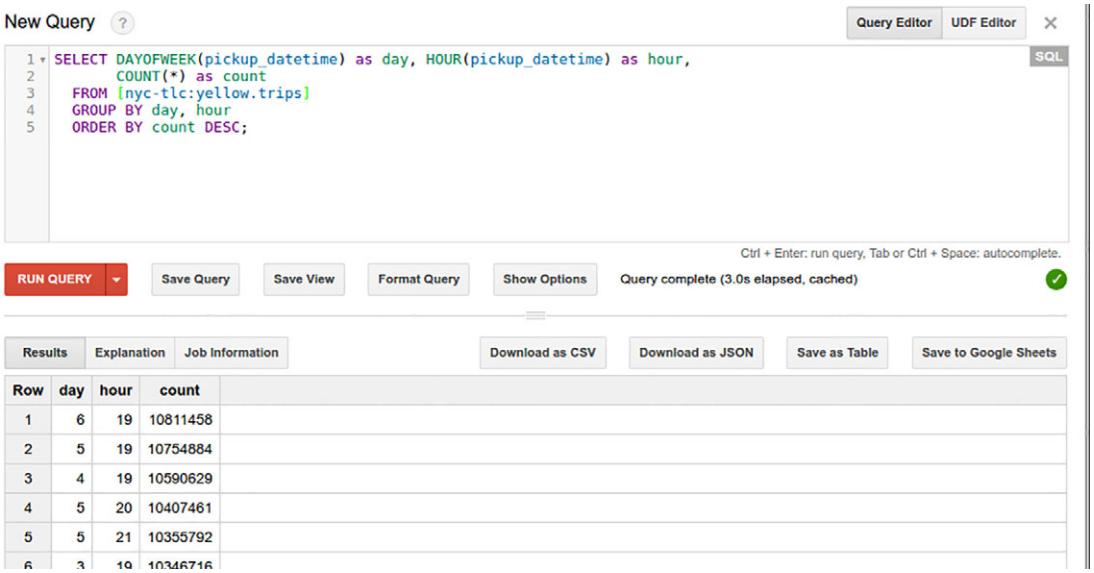


Figure 19.9 Results showing the day and hour with most pickups

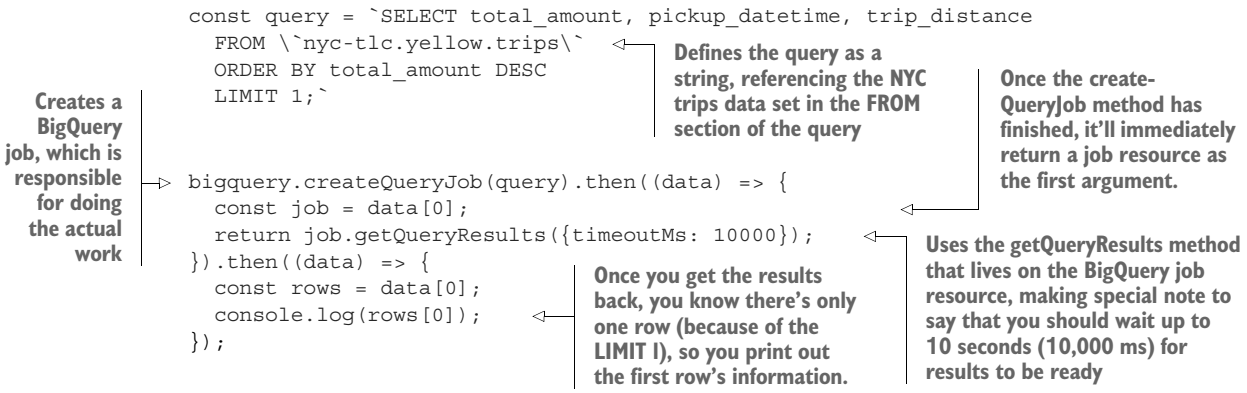
Right now, you might be thinking "So what? MySQL can do all of this." If so, BigQuery has done its job. The whole purpose of BigQuery is to feel like running an analytical query with any other SQL database, but way faster. You'll tend to forget that these queries you're running are scanning over more than a billion records stored in BigQuery, and doing so like it was 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 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, you can write some code that finds the most expensive ride, as in the following listing. If you haven't already, start by installing the Node.js client for BigQuery using `npm install @google-cloud/bigquery@1.0.0`.

Listing 19.2 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'
});
```

Even though you're running a query against a dataset in another project, you'll be creating a job in your project, so you use your own project ID here.



If you run this code, you should see the same output you saw when you tried it in the BigQuery UI, with that crazy trip costing $4 million:

```
{ 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? What if you wanted to find the total cost of all of the trips? Try this:

```
SELECT SUM(total_amount) FROM `nyc-tlc.yellow.trips`;
```

If you replace the `query` value in your code in listing 19.2, 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:

```
{ 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 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 types of operations, one of them being for loading new data. But you have multiple ways of getting data from a source 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 I refer to *bulk loading*, I'm talking about the concept of taking a big chunk of arbitrary data (such as a bunch of CSVs or JSON objects) and loading it into a BigQuery table. In many ways, this is similar to a MySQL `LOAD DATA` query, which you typically use for restoring data that you backed up as a CSV file. As you might guess, you have quite a few options to configure when loading data (data compression, character encoding, and so on), so let's start with the basics.

Imagine you're recreating a table to store the data from taxi rides, similar to the one you've been querying in the shared dataset. To start, you need to create a dataset, then a table, and then you need to set the schema to fit your data. You can do all of these things 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 to BigQuery's interface. On the left-hand side, you should see a "No datasets found" message, so use the little arrow next to your project name and choose Create New Dataset (figure 19.10).

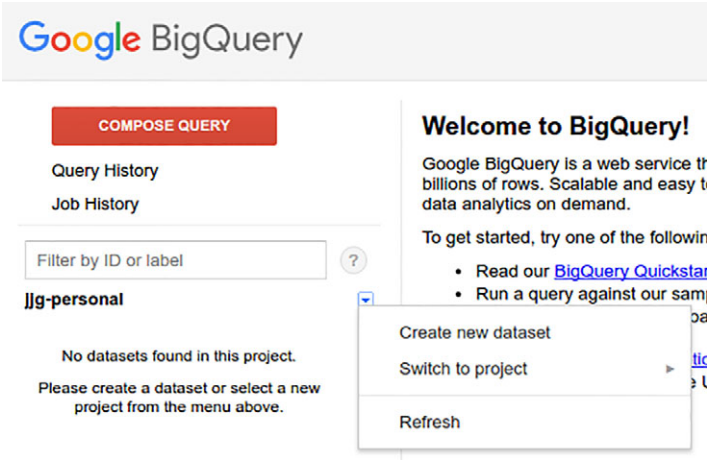


Figure 19.10 Menu showing how to create a new dataset

When you click this, a window will pop up where you can choose the ID for your dataset (figure 19.11). Before you fill it out, note that BigQuery IDs have a tiny difference from the IDs of resources in the rest of Google Cloud Platform: no hyphens. Because 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 (`test_dataset_id` instead of `test-dataset-id`). For this demo, you can call your dataset `taxi_test` (whereas you would've called it `taxi-test` if hyphens were allowed). You

also can choose where the data should live (in the United States or the European Union) and when it should expire. For now, leave both of these options set to the defaults (Unspecified and Never).

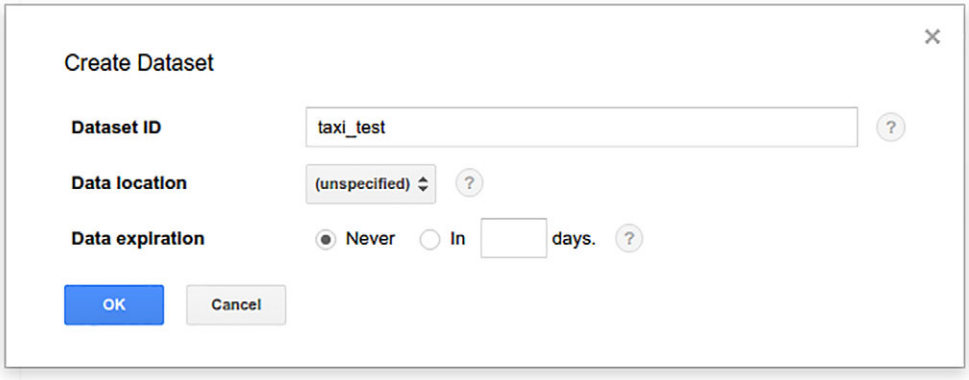


Figure 19.11 Form for creating a new dataset

Click OK, and your dataset should appear right away, so it's time to create your new table. Like with creating a new dataset, use the arrow menu to choose Create a New Table, and you'll see a big form appear (figure 19.12).

The first thing to remember is that tables are mutable, so if you forget a field, it's not the end of the world. But it does mean 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 (like in a regular SQL database). Assume you have a slimmed-down version of the taxi data that has a few fields for the pickup 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`. Here's some example CSV data (which you can use later on if you put it in a file):

```
1493033027,1493033627,8.42
1493033004,1493033943,18.61
1493033102,1493033609,9.17
1493032027,1493033801,24.97
```

Using this information, you can define a table called `trips`, with those three fields under the Schema section. Lastly, if you put the CSV data from those four data points into a file, you can use them in the Source Data section.

Notice that you've checked the File Upload data source in this example, but you also could have uploaded the CSV file to Cloud Storage or Google Drive and used the file hosted there as the source. Additionally, even though you 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.

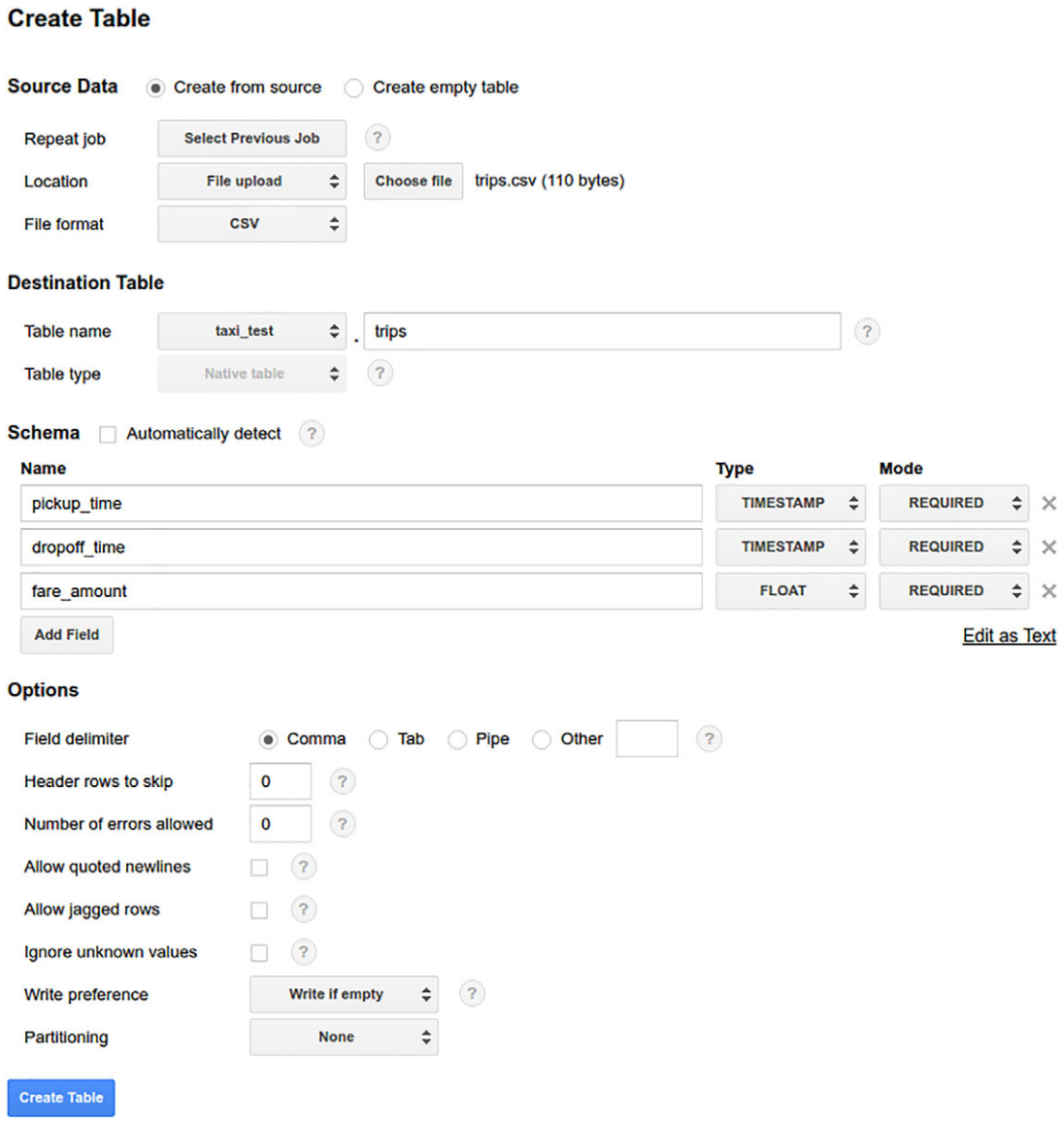


Figure 19.12 Creating the trips table

Listing 19.3 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, BigQuery will immediately create the table with the schema you defined and will create a *load data* job under the hood. Because the data here is tiny, this job should complete quickly, and you should see a result showing that the data loaded successfully into the new table (figure 19.13).

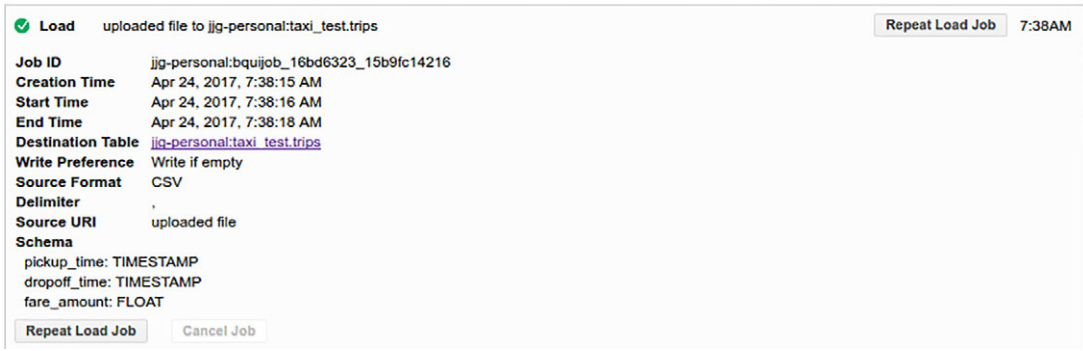


Figure 19.13 Load data job status

Once the data is loaded, you can check on it by running a SQL query. Because you already know how to select all rows, let's look at a fancier query that shows a summary of the cost per minute of your sample trips:

```
SELECT
  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`
LIMIT
  1000;
```

You have to swap this with your own project ID.

Running this query should show how much each trip cost on a per-minute basis, as you can see in figure 19.14.

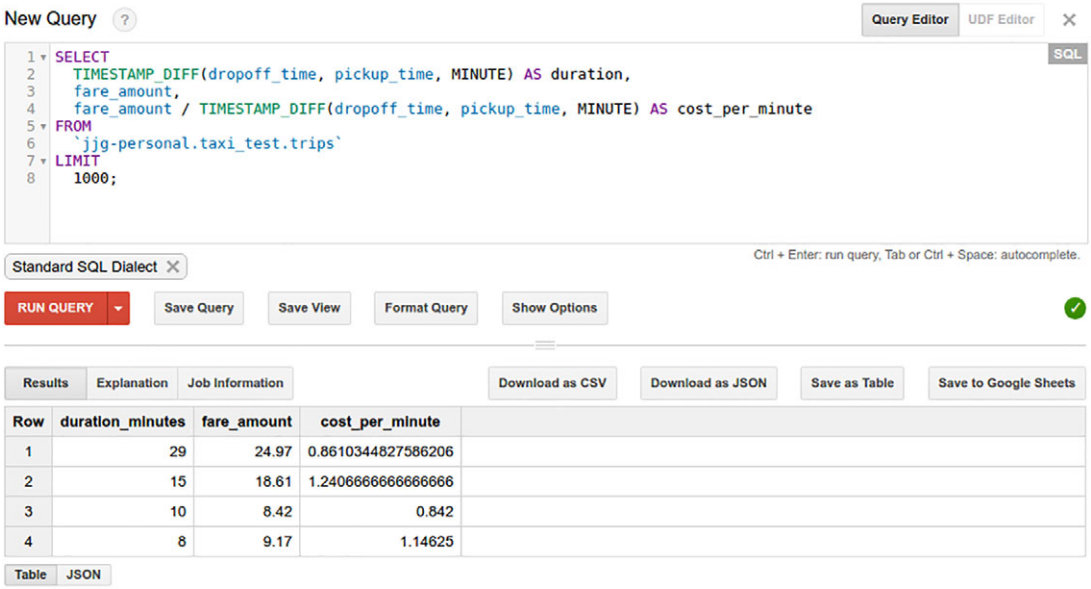


Figure 19.14 Query results for the cost per minute of each trip

The only difference between the job of loading from a sample CSV and a real-life example will be the size of the data, so if you want to try that out, try generating a larger file and loading that. To load from GCS, you can choose Google Cloud Storage from the list of locations and enter the GCS-specific URL into the location box (figure 19.15).

When you click Create Table, BigQuery will pull the data in from GCS and load it into the table. In this case, the loading job takes a few minutes, whereas it only took a few seconds before (figure 19.16). The example file in this demo was about 3.2 GB in total, so a few minutes isn't so bad.

Now you can query the data as before. As you can see in figure 19.17, counting the number of rows shows quite a bit more data, totaling about 120 million rows.

This method of getting data into BigQuery works if you have one chunk of data that isn't 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 make sense, and streaming new data into BigQuery makes for a much better fit. Let's look at how that works by seeing how the taxi trips might stream new rows into your table.

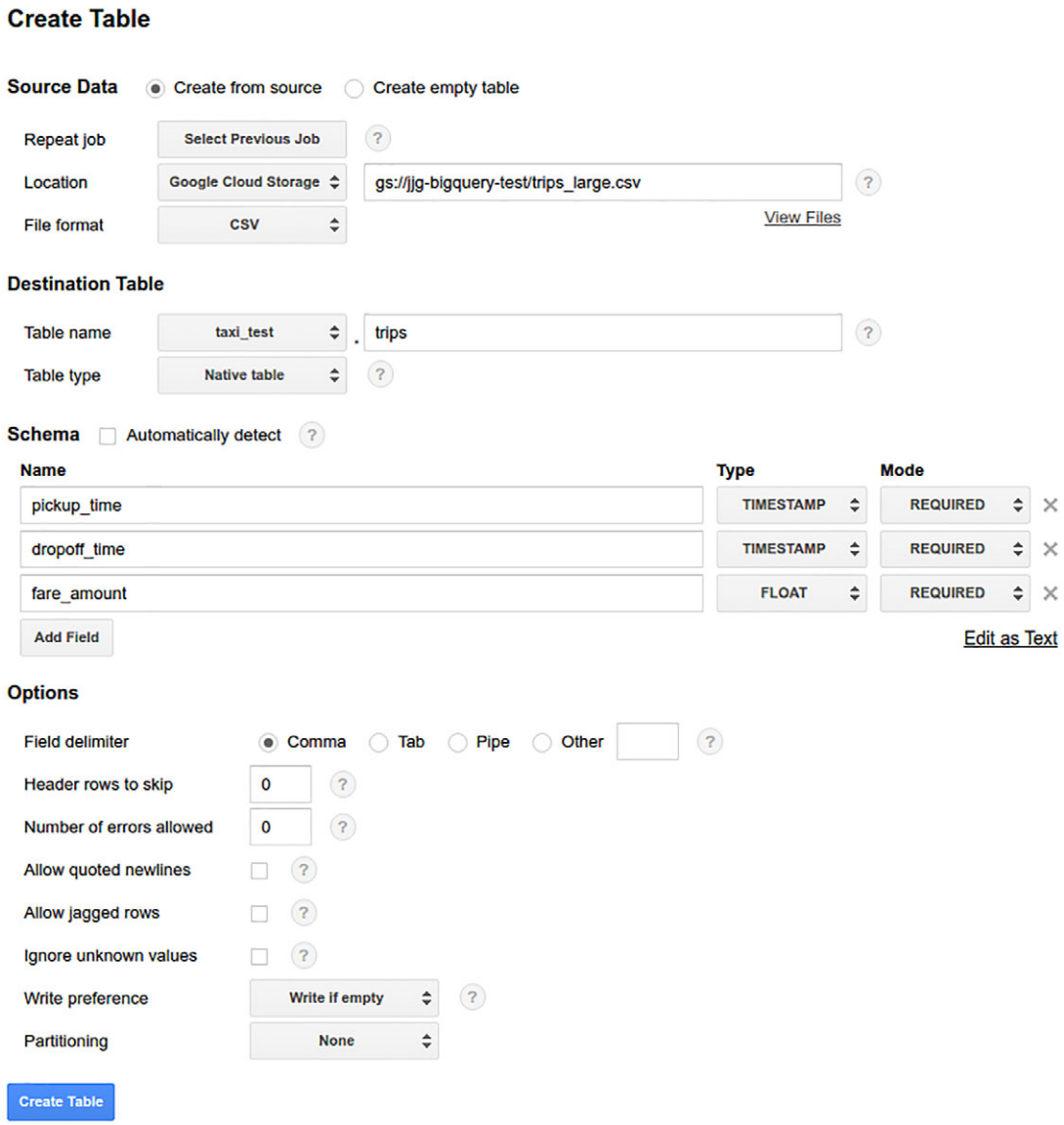


Figure 19.15 Loading a larger file from GCS

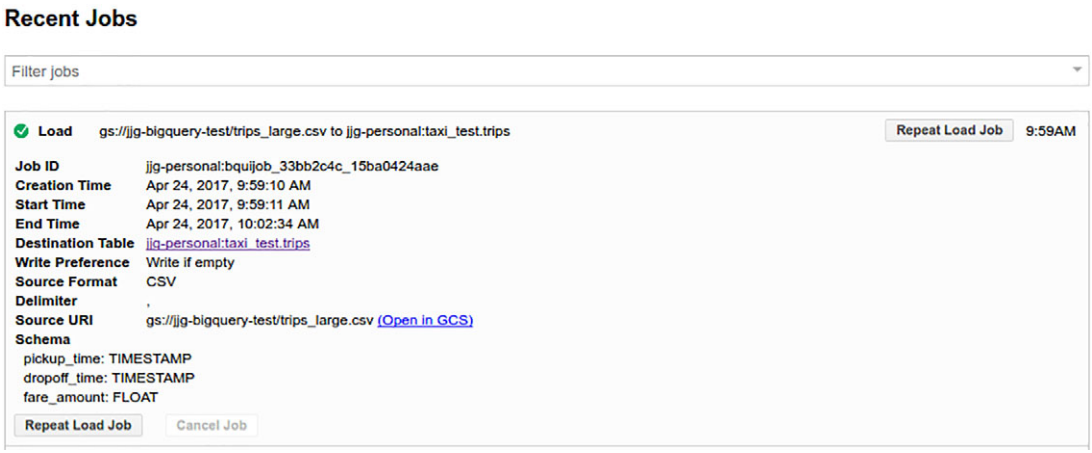


Figure 19.16 Loading job results from a larger file on GCS

Figure 19.17 Total number of rows in the larger file from GCS

STREAMING DATA INTO BIGQUERY

We've explored how to bulk load a big chunk of existing data into BigQuery, but what 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 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, imagine that when a taxi ride is over, you want to make sure you log that the trip happened with the pickup and drop-off times as well as the total fare amount. To do this, you could write a function that takes an object representing the trip and inserts it into your BigQuery trips table, as shown in the following listing.

Listing 19.4 Streaming new data into BigQuery

```
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) => {
  return table.insert({
    pickup_time: trip.pickup_time.getTime() / 1000,
    dropoff_time: trip.dropoff_time.getTime() / 1000,
    fare_amount: trip.fare_amount
  });
}
```

Gets a pointer to the BigQuery table using the .dataset and .table helper methods

Uses the .insert method to load a single row into BigQuery

The assumption here is that the value will be a JavaScript Date type, so you 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 (for example, the connection was cut because of network issues), it would be 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, but on the other hand, you could be missing data points that you thought were stored but were instead lost in transit.

To avoid this problem, 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 simple: if BigQuery has seen the ID before, it'll 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, as shown in the following listing.

Listing 19.5 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 uuid = uuid4();
  return table.insert({
    json: {
      pickup_time: trip.pickup_time.getTime() / 1000,
      dropoff_time: trip.dropoff_time.getTime() / 1000,
      fare_amount: trip.fare_amount
    },
    insertId: uuid
  }, {raw: true});
}
```

Uses a UUID-4 (a random UUID) to act as the insert ID

Specifies the row data in the json property

Sets the insert ID in the insertId property

Tells the client that this is a raw row

Now when you log a trip, if some sort of failure occurs, your client will automatically retry the request. If BigQuery has already seen the retried request, it will ignore it. Also, if the request looked like it failed to you but in reality it worked fine on BigQuery's side, BigQuery will ignore the request rather than adding the same rows all over again.

Note that you're using a random insert ID because a deterministic one (such as a hash) may disregard identical but nonduplicate data. For example, 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 might 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 you shouldn't use it as a way to deduplicate your data. If you need unique data, you should preprocess the data to remove duplicates first, then 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 the trips 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 I've 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 to do some machine learning on it and predict the cost of a trip based on the locations, pickup times, and so on. Pulling this out through the SQL-like interface won't solve the problem for you. Luckily an easier way exists to pull data out of BigQuery: an export job.

Export jobs are 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 reimport it into another table as needed. But 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. Choose the table you want to export and click Export Table (figure 19.18).

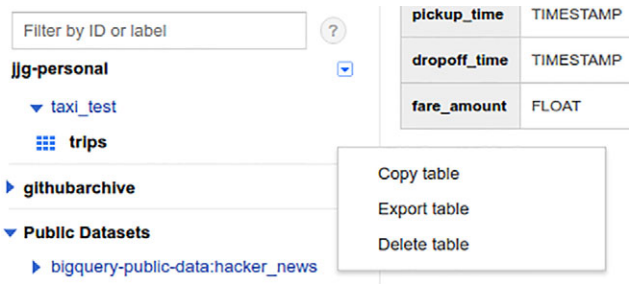


Figure 19.18 Preparing to export data into GCS from BigQuery using Export Table

On the form that shows up (figure 19.19), you can pick the export format and where to put the data afterwards (a filename starting with gs://). You also can choose to compress the data with Gzip compression if you like.

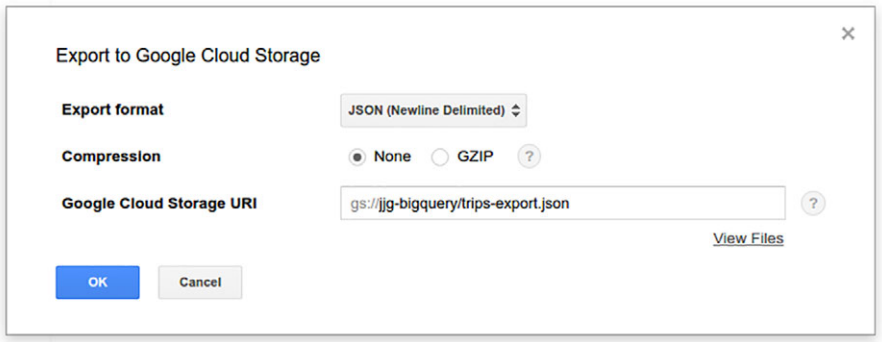


Figure 19.19 Exporting data from BigQuery to GCS

If your data is 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 it with a single file first, and you'll get an error if it's too large.

Once you click OK, 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 19.20).

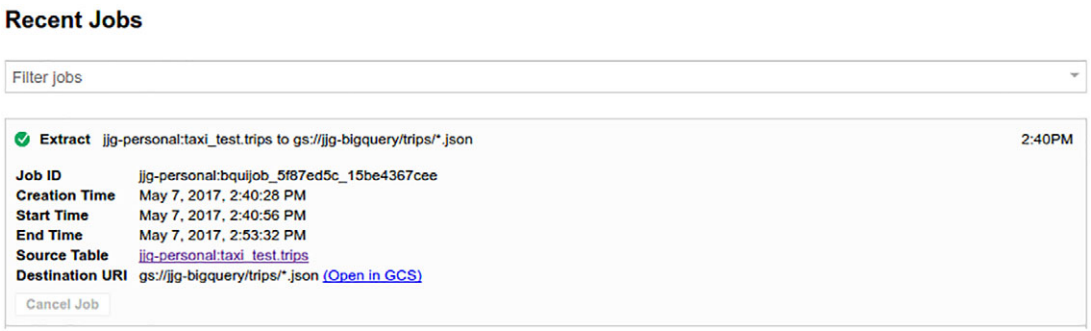


Figure 19.20 The status of the export job

Once the operation 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 copying the data over to an on-premises data warehouse. Now that you'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 it on a regular basis.

19.3 Understanding pricing

Like many of the services on Google Cloud Platform, BigQuery follows the "pay for what you use" pricing model. But it's a bit unclear exactly how much you're *using* in a system like BigQuery, so let's look more closely at the different attributes that cost money. With BigQuery, you're charged for three things:

- Storage of your data
- Inserting new data into BigQuery (one row at a time)
- Querying your data

19.3.1 Storage pricing

Similar to other storage products, the cost of keeping data in BigQuery is measured in GB-months. You're charged not only for the amount of data, but also for how long it's stored.

To make things more complicated, BigQuery has two different classes of pricing, based on how long you keep the data around. BigQuery treats tables you're actively adding new data into as standard storage, whereas it treats tables that you leave alone for 90 days as long-term storage, which costs less. The idea behind this is to give a cost break on data that's older and might otherwise be deleted to save some money. It's

important to remember that although the long-term storage costs less, you'll see no degradation in any aspect of the storage (performance, durability, or availability).

What does all of this cost? Standard storage for BigQuery data is currently $0.02 US per GB-month, and long-term storage comes in at half that price ($0.01 US). If you had two 100 GB tables, and one of them hadn't been edited in 90 days, you'd have a total bill of $3 US for each month you kept the data around ($0.02 * 100 + $0.01 * 100), excluding the other BigQuery costs. That covers the raw storage costs.

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 doesn't 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, except for streaming inserts. For example, 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 with other storage systems, such as Cloud Datastore, streaming inserts are measured based on their size rather than the number of requests. There's no difference between two API calls to insert one row each and one API call that inserts both rows—the total data inserted will cost $0.05 per GB inserted regardless. Given this pricing scheme, you probably should avoid streaming new data into BigQuery if you can bulk load the data all at once at the end of each day (because importing data is completely free). But 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 is the most common one: querying.

19.3.3 Query pricing

Running queries on BigQuery is arguably the most important function of the service, but it's 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 quickly. 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 US per TB processed. For example, a query that scans the entirety of a 1 TB table (for example, `SELECT * FROM table WHERE name = 'Joe'`) will cost $5 in the few seconds it takes to complete!

You should keep a few things in mind when looking at how much queries cost. First, if an error occurs in executing the query, you aren't charged anything at all. But if you cancel a query while it's running, you still may end up being charged for it—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. 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 you may think of data processed in terms of number of rows processed, but with BigQuery, that isn't the case. Because BigQuery is a column-oriented storage system, although the total data processed has to do with the number of rows scanned, the number of columns selected (or filtered) is also considered. For example, imagine you have a table with two columns, both `INTEGER` types. If you were to only look at one of the columns (for example, `SELECT field1 FROM table`), it would cost about half as much as looking at both columns (for example `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 need tons and tons of querying capacity or want the ability to limit how much money you spend on querying data from BigQuery, fixed-rate pricing is available, but it's mainly for people spending a lot of money (for example, $10,000 and up per month).

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 moment's notice.
- Although BigQuery supports many features of OLTP databases, it doesn't have transactions or uniqueness constraints, and you should use it as an analytical data warehouse, not a transactional database.
- Although data in BigQuery is mutable, the lack of uniqueness constraints means it's not always possible to address a specific row, so you should avoid doing so—for example, don't do `UPDATE … WHERE id = 5`.
- When importing or exporting data from BigQuery, GCS typically acts as an intermediate place to keep the data.
- If you need to update your data frequently, BigQuery's streaming inserts allow you to add rows in small chunks, but using them 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 need—for example, avoid `SELECT * FROM table`.

20 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 you use Apache Beam and Cloud Dataflow together to process large sets of data?

You've probably heard the term *data processing* before, likely meaning something like "taking some data and transforming it somehow." More specifically, when we talk about data processing, we tend to mean taking a lot of data (measured in GB at least), potentially combining it with other data, and ending with either an 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 email history in one big pile, and all of your contact information (email 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 email addresses. Once you did that, you could then filter that data down to find only emails that were sent on someone's birthday (figure 20.1).

This idea of taking large chunks of data and combining them with other data (or transforming them somehow) to come out with a more meaningful set of data can be valuable. For example, if you couldn't join the email and contact data like

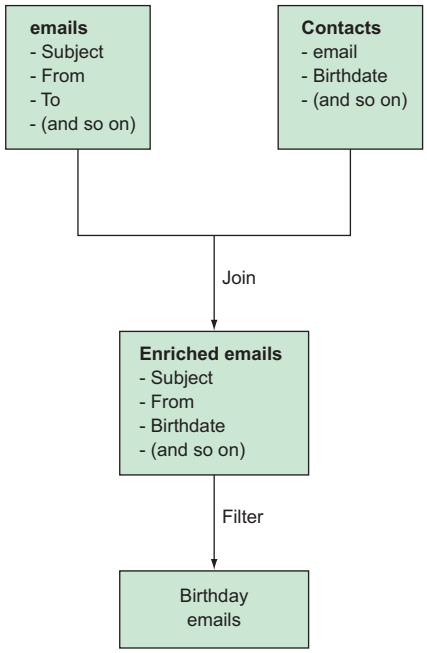


Figure 20.1 Using data processing to combine sets of data for further filtering

this, you'd need to do so manually earlier on. In this case, you'd need to provide the birthdays of all participants in an email thread whenever you sent or received an email, which would be a silly thing to do.

In addition to processing one chunk of data into a different chunk of data, you also can think of data processing as a way to perform *streaming transformations* over data while it's in flight. Rather than treating your email history as a big pile of data and enriching it based on a big pile of contact information, you might instead intercept emails as they arrive and enrich them one at a time. For example, you might load up your contact information, and as each email arrives, you could add in the sender's birthday. By doing this, you end up treating each email 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).

Treating data as a batch or a stream tends to come with benefits and drawbacks. For example, if all you want to do is count the number of emails you receive, storing and querying a big pile of data would take up lots of space and processing time, whereas if you were to rely on a stream, you could increment a counter to keep count whenever new emails arrive. On the flip side, if you wanted to count how many emails you got last week, this streaming counter would be pretty useless, compared to your batch of email data that you could filter through to find only last week's emails and then count the matching ones (see figure 20.3).

So how can you express these ideas of data processing, for both streams and batches of data? How do you write code that represents combining email history and

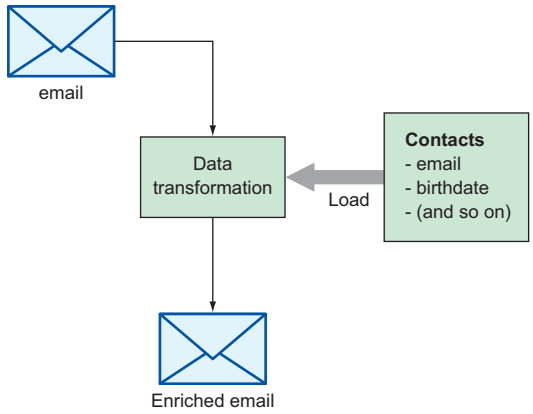


Figure 20.2 Processing data as a stream rather than as a batch

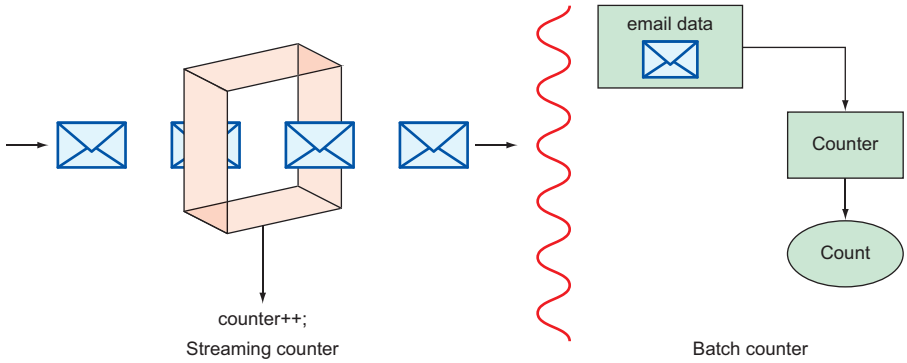


Figure 20.3 Streaming vs. batch counter

contact information to get an enriched email that also contains the sender's birthday? Or how do you keep count of the number of incoming emails? What about counts that only match certain conditions, like those arriving outside of work hours? You can express these things in lots of ways, but we'll look specifically at an open source project called Apache Beam.

20.1 What is Apache Beam?

Having the ability to transform, enrich, and summarize data can be valuable (and fun), but it definitely won't be easy unless you can represent these actions in code. You need a way of saying in your code "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. You can represent pipelines in lots of ways for various purposes, but for handling data processing pipelines, Apache Beam fits 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 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 in a few pages to use Beam with Cloud Dataflow.
>
> If you're excited to learn more about Apache Beam, check out http://beam.apache.org, which has much more information.

20.1.1 Concepts

Before you get into writing a bunch of code, let's start by looking at some of the key concepts needed to understand to be able to express pipelines using Apache Beam. The key concepts we'll look at include the high-level container (a *pipeline*), the data that flows through the pipeline (called *PCollections*), and how you manipulate data along the way (using *transforms*). Figure 20.4 represents these concepts visually.

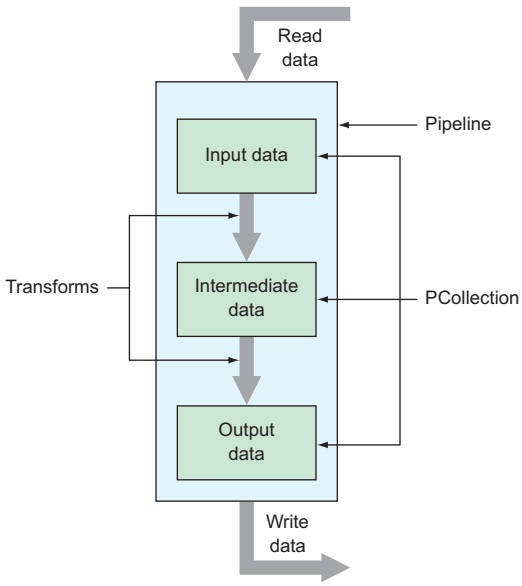


Figure 20.4 The core concepts of Apache Beam

PIPELINES

In Apache Beam, a *pipeline* refers to the high-level container of a bunch of data processing operations. 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. Generally, the pipeline is the first thing you create when you write code that uses Apache Beam. In more technical terms, a pipeline is a *directed acyclic graph* (sometimes abbreviated to *DAG*)—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 figure 20.4, you can see that the chunks of data are like the nodes in a graph, and the big arrows are the edges. The fact that the edges are arrows pointing in a certain direction is what makes this a *directed* graph.

Finally, notice that the pipeline (or graph) in figure 20.4 clearly flows in a single direction and can't get into a loop. This is what we mean by an *acyclic* graph—the pipeline has no cycles to end up in. For example, figure 20.5 shows an acyclic graph using solid lines only. If you were to add the dashed line (from E back to B), the graph could have a loop and keep going forever, meaning it'd no longer be acyclic.

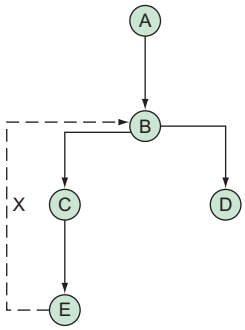


Figure 20.5 A directed acyclic graph with a cyclic option (dashed line)

Pipelines themselves can have lots of 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 you can then read from the command line when you 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—they're 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 how you represent chunks of data as they move through your pipeline.

PCollections

PCollections, so far known only as the nodes in your graph or the data in your pipeline, act as a way to represent intermediate chunks or streams of data as they flow through a pipeline. Because PCollections represent data, you can create them either by reading from some raw data points or by applying some transformation to another PCollection, which I'll discuss more in the next section.

Notice that I only said that a PCollection represents some data, not how it represents that data under the hood. The data could be of any size, ranging from a few rows that you add to your pipeline code to an enormous amount of data distributed across lots of machines. In some cases, the data could even be an infinite stream of incoming data that may never end. For example, you might have a temperature sensor that sends a new data point of the current temperature every second. 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*, you may not know the exact size, but you do know that it does have a fixed and finite size (such as 10 billion items). A bounded PCollection is one that you're sure won't go on forever.

As you'd expect, an *unbounded* PCollection is one that has no predefined finite size and may go on forever. The typical example of an unbounded PCollection is a stream of data that's being generated in real time, such as the temperature sensor I mentioned. Given the fundamental difference between these two types of PCollections, you'll end up treating them a bit differently when running the pipeline, so it's important to remember this distinction.

PCollections also have a few technical requirements that'll affect how you express them in code. First, you always create a PCollection within a pipeline, and it must stay within that pipeline. You can't create a PCollection inside one pipeline and then reference it from another. This shouldn't be too much of an issue because you'll likely be defining one pipeline at a time.

Additionally, PCollections themselves are *immutable*. Once you create a PCollection (for example, by reading in some raw data), you can't change its data. Instead, you rely on a functional style of programming to manipulate your data, where you can create new PCollections by transforming existing ones. We'll get into this in the next section when I talk about transforms, but if you've ever written functional code that manipulates immutable objects, transforming PCollections should seem natural.

Finally, PCollections are more like Python's iterators than lists. You can continue asking for more data from them, but can't necessarily jump to a specific point in the PCollection. Thinking in terms of Python code, it's fine to write `for item in pcollection:` … to iterate through the data in the PCollection, but you couldn't write `item = pcollection[25]` to grab an individual item from it. Now that you have a decent grasp of what PCollections are and some of the technical details about them, let's look at how to work with them using transforms.

TRANSFORMS

Transforms, as the name implies, are the way you take chunks of input data and mutate them into chunks of output data. More specifically, transforms are the way to take PCollections and turn them into other PCollections. Put visually, these are the big arrows between PCollections inside pipelines (figure 20.6), where each transform has some inputs and outputs.

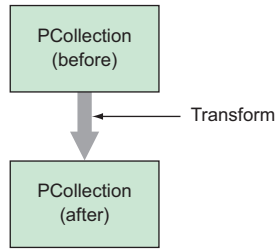


Figure 20.6 A transform between PCollections

Transforms can 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 lot of boilerplate code. For example, the following are all examples of transforms built in to Beam that you might apply to a given PCollection:

- Filter out unwanted data that you're not interested in (such as filtering personal emails out from your email data)
- Split the data into separate chunks (such as splitting emails into those arriving during work hours versus outside work hours)

- Group the data by a certain property (such as grouping emails by the sender's address)
- Join together two (or more) chunks of data (such as combining emails with contact information by email address)
- Enrich the data by calculating something new (such as calculating the number of people cc'd on an email)

These are a few 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 that Beam provides, you can write custom transforms and use them in your pipelines. These transforms have a few interesting properties that are worth mentioning.

First, although many of the transforms I described 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 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, because PCollections are immutable, transforms that you apply to them don't consume the data from an existing PCollection. Put slightly differently, when you apply a transformation to an existing PCollection, you create a new one without destroying the existing one that acted as the data source. You can use the same PCollection as an input to multiple transforms in the same pipeline, which can come in handy when you 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 consumed by iterating over them), which makes it a common area of confusion for people new to Apache Beam.

Finally, although you 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 to execute a pipeline.

Pipeline runners

As the name implies, a *pipeline runner* runs a given pipeline. Although the high-level concept of the system that does the work is a bit boring, the lower-level details are interesting—there can be a great deal of variety in how to apply transforms to PCollections.

Because Apache Beam allows you to define pipelines using the high-level concepts you've learned about so far, you can keep the *definition* of a pipeline separate from the *execution* of that pipeline. Although the definition of a pipeline is specific to Beam, the underlying system that organizes and executes the work is abstracted away from the definition. You could take the same pipeline you defined using Beam and run it across a variety of execution engines, each of which 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 as similar to ORM (object-relational mapping) that you implement with SQL Alchemy in Python or Hibernate in Java. ORMs allow you to define your resources and interact with them as objects in the same language (for example, Python), and under the hood the ORM turns those interactions into SQL queries for a variety of databases (such as MySQL and PostgreSQL). In the same way, Apache Beam allows you to define a pipeline without worrying about where it'll run, and then later execute it using a variety of pipeline runners.

Quite a few pipeline runners are available for Apache beam, with the simplest option being the `DirectRunner`. This runner is primarily a testing tool that'll execute pipelines on your local machine, but it's interesting in that it doesn't take the simplest and most efficient path toward executing your pipeline. Instead, it runs lots of additional checks to ensure the pipeline doesn't rely on unusual semantics that'll break down in other more complex runners.

Unlike the `DirectRunner`, typical 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 as being like a Kubernetes cluster, in that they both execute given input using a set number of machines. This distributed execution allows the work to be spread out across a potentially large number of machines, and you can make any job complete more quickly by adding more machines to the process.

To enable this distribution, pipeline runners will chop the work up into lots of pieces (both at the start of the pipeline and anywhere in between) to make the most efficient use of all the machines available. Although you'd still represent transforms as an arrow between two PCollections, under the hood the work might be split into lots of little pieces with transforms applied across several machines. Put visually, this might look something like figure 20.7.

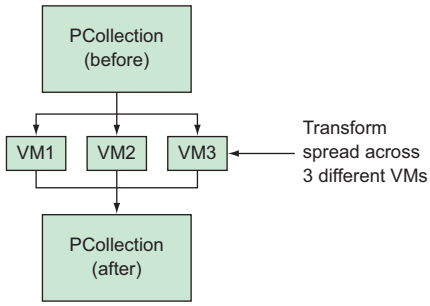


Figure 20.7 A transform applied using multiple VMs

Because this chopping up may mean having to move data around, pipeline runners will do their best to execute computations on as few machines as possible. They do so mostly because data moving over a network is far slower than accessing it from local memory. Minimizing data sent over the network means the processing jobs can complete faster.

Unfortunately, moving data over the network is often unavoidable. On the other hand, other options, such as using a single large machine to do the work, might be even slower despite the time needed to move data from one machine to another over the network. Although shifting data from one place to another may add some overhead, the pipeline runner will likely 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 more specific by writing some code; then we'll look at using one of these pipeline runners.

20.1.2 Putting it all together

Using these three basic concepts (pipelines, PCollections, and transforms) you can build some cool things. Until now I've stuck to high-level descriptions and stayed away from code. Let's dig into the code itself by looking at a short example.

> **WARNING** Because Apache Beam doesn't have bindings for Node.js, the rest of this chapter will use Python to define and interact with Beam pipelines. I'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 you have a digital copy of some text and want to count the number of words that start with the letter *a*. As with many problems like this, you could 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 you have thousands of those texts to analyze? It would probably take even the fastest computers quite a while to do this work. You could instead use Apache Beam to define a pipeline that would do this for you, which would allow you to spread that work across lots of computers and still get the right output. What would this pipeline look like? Let's start by looking graphically at the pipeline (figure 20.8); then you can start writing code.

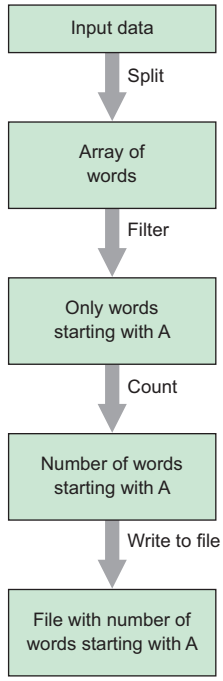


Figure 20.8 Pipeline to count words starting with *a*

Notice that you use multiple steps to take some raw input data and transform it into the output you're interested in. In this case, you 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 you aren't interested in, and then a `Count` transform to reduce the set to the total number of words. Then you finally 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 listing 20.1.

> **NOTE** The following code is accurate but leaves a few variables undefined. It's not expected to run if you copy and paste it exactly as is—you'll need to fill in

some blanks (for example, `input_file`). Don't worry, though. You'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:
  (pipeline
    | beam.io.ReadFromText(input_file)
    | '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)
  )
```

Creates a new pipeline object using Beam

Loads some data from a text input file using beam.io.ReadFromText

Takes the input data and splits it into a bunch of words

Filters out any words that don't start with 'a'

Counts all of those words

Writes that number to an output text file

As you can see, in Apache Beam's Python bindings, you rely on the pipe operator, as you would in a Unix-based terminal, to represent data flowing through the transformations. This allows you to express your intent of how data should flow without getting into the lower level details about how you might divide this problem into smaller pieces, which, as you learned before, is the responsibility of the pipeline runner used to execute the code itself.

At this point, you've learned all the important concepts, looked at an example pipeline, and looked at some corresponding Python code for the pipeline. But what about the pipeline runners? For Apache Beam, quite a few are available, such as Apache Flink and Apache Apex, but one fully managed pipeline runner is the subject of this chapter: Google Cloud Dataflow.

20.2 What is Cloud Dataflow?

As you learned previously, you can use Apache Beam to define pipelines that are portable across lots of pipeline runners. This portability means there are lots of options to choose from when it comes time to run 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. Unlike other pipeline runners, using Cloud Dataflow requires no initial setup of the underlying resources. 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's all taken care of for you, so 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). In their case, running your own Kubernetes cluster requires you to manage the machines that run Kubernetes itself, whereas with Kubernetes Engine, those

machines are provisioned and managed for you. Because Cloud Dataflow is part of Google Cloud Platform, it has the ability to stitch together lots of other services that you've learned about already. For example, you 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 20.9).

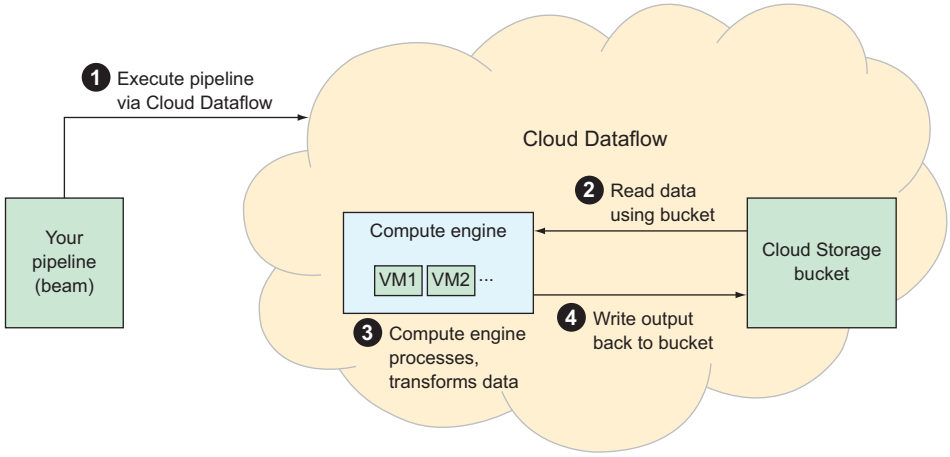


Figure 20.9 Overview of the infrastructure for Cloud Dataflow

Unlike with some of the other runners, Google's systems handle all of this coordination across the various Google Cloud Platform resources for you. You pass in the specifics (such as where input data lives in Cloud Storage) as pipeline parameters, and Cloud Dataflow manages the work of running your pipeline entirely. So how do you use Cloud Dataflow? Let's look at the previous example where you count all the words starting with the letter *a* and see how you can use Cloud Dataflow and Apache Beam to run your pipelines.

20.3 Interacting with Cloud Dataflow

Before you can start using Cloud Dataflow, you'll need to do a bit of initial setup and configuration. Let's look at that first, and then you can jump into creating your pipeline.

20.3.1 Setting up

The first thing you 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 (figure 20.10). Click that, and you should be good to go.

After you enable the API, you'll need to make sure you have Apache Beam installed locally. To do this, you can use `pip`, which manages packages for Python.

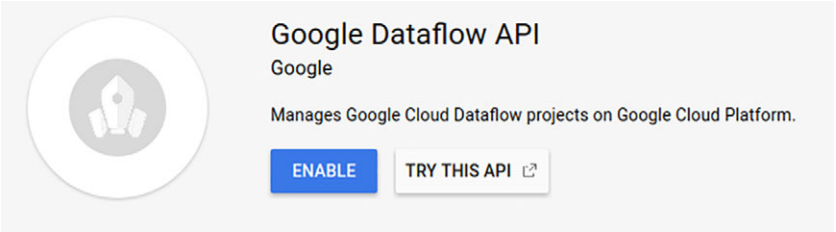


Figure 20.10 Enabling the Cloud Dataflow API

Although the package itself is called `apache-beam`, you want to make sure you get the Google Cloud Platform extras for the package. These extra additions will allow your code to interact with GCP services without any additional code. For example, one of the GCP extras for Apache Beam allows you to refer to files on Google Cloud Storage by a URL starting with gs://. Without this, you'd have to manually pull down the Python clients for each of the services you wanted to use. To get these extras, you'll use the `[]` syntax, which is standard for Python:

```
$ pip install apache-beam[gcp]
```

The next thing you'll need to do is make sure any code you run uses the right credentials and has access to your project on Cloud Dataflow. To do this, you can use the `gcloud` command-line tool (which you installed previously) to fetch credentials that your code will use automatically:

```
$ gcloud auth application-default login
```

When you run that command, you'll see a link to click, 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 you've enabled all of your APIs, have all the software packages you need, and have fetched the right credentials, there's one more thing to do: figure out exactly where you can put your input, output, and (possibly) any temporary data while running your pipeline. After all, the pipeline you defined previously reads input data from somewhere and then writes the output to somewhere. In addition to those two places, you may need a place to store extra data, sort of like a spare piece of paper during a math exam.

Because you're already using Google Cloud Platform for all of this, it makes sense that you'd use one of the storage options such as Google Cloud Storage. To do so, you'll need to create a Cloud Storage bucket:

```
$ gsutil mb -l us gs://your-bucket-id-here
Creating gs://your-bucket-id-here/...
```

This command specifically creates a bucket that uses multiregional replication and is located in the United States. If you're confused by this, take a look at chapter 8. And with that, you have all you need and can finally get to work taking the code in listing 20.1 and turning it into a runnable pipeline!

20.3.2 Creating a pipeline

As you may remember, the goal of the example pipeline is to take any input text document and figure out how many words in the document start with the letter *a* (lowercase or uppercase). You first saw this as a visual representation and then transcribed that into more specific code that relied on Apache Beam to define the pipeline. But you may also recall that the listing left some of the details out, such as where the input files came from and how to execute the pipeline. To make the pipeline run, you'll need to provide these details, as well as add a bit of boilerplate to your pipeline code.

The following updated code 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 your pipeline as options.

Listing 20.2 Your 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>'
BUCKET = 'dataflow-bucket'


def get_pipeline_options(pipeline_args):
  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):

  parser = argparse.ArgumentParser()
  parser.add_argument('--input', dest='input')
```

First, define a bunch of parameters, like your project ID and the bucket where you'll store data. This is the bucket you created in the previous section.

This helper function takes a set of arguments, combines them with some reasonable defaults, and converts those into Apache Beam-specific pipeline options, which you'll use later.

In the main method, you start doing the real work. First, take any arguments passed along the command line. You'll use some of them directly in your code (for example, input), and the rest you'll treat as options for the pipeline itself to use.

At this point, the code should look similar to listing 20.1. One difference is that you pass in some specific options when creating the pipeline object.

```
  parser.add_argument('--output', dest='output',
                      default='gs://%s/dataflow-count' % BUCKET)
  script_args, pipeline_args = parser.parse_known_args(argv)
  pipeline_opts = get_pipeline_options(pipeline_args)

  with beam.Pipeline(options=pipeline_opts) as pipeline:
      (pipeline
       | beam.io.ReadFromText(script_args.input)
       | '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()
```

To get everything rolling, call the main function that you've just defined.

Another difference from the original listing is that you define the location of your input data based on the command-line arguments.

At this point, you 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*. Now how about taking it for a test drive?

20.3.3 Executing a pipeline locally

As you learned before, Apache Beam has a few built-in pipeline runners, one of which is the `DirectRunner`. This runner is 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 your pipeline will run well in a distributed environment (like Google Cloud Dataflow). Because you have that pipeline runner, you can execute the pipeline locally to test it out, first with some sample data and then something a bit larger. To start, you can create a simple file with a few words in it that you can easily count by hand to verify your code is doing the right thing:

```
$ echo "You can avoid reality, but you cannot avoid the consequences of
     avoiding reality." > input.txt
$ python counter.py --input="input.txt" \
  --output="output-" \
  --runner=DirectRunner
```

Here the output is a prefix to use where you put the file.

Use the Direct Runner to execute your pipeline, which runs this entire job locally.

As you can see in the sentence in the snippet, exactly three words start with the letter *a*. Let's check whether your pipeline came up with the same answer. You can see that by looking in the same directory for output inside a file starting with `output-`:

```
$ ls output-*
output--00000-of-00001
$ cat output--00000-of-00001
3
```

It looks like you found the right number of words!

Your pipeline clearly did the trick. But what will it do with a larger amount of data? You're going to use a little Python trick to take that same sentence, repeat it a bunch of times, and test your pipeline again. As before, because you're repeating the input a set number of times, you know the answer is three times that, so it will be easy to check whether your pipeline is still working:

```
$ python -c "print (raw_input() + '\n') * 10**5" < input.txt > input-10e5.txt
$ wc -l input-10e5.txt
100001 input-10e5.txt
$ du -h input-10e5.txt
7.9Minput-10e5.txt
```

Takes out the input sentence from before, repeats it 100,000 times, and saves it back to a file called input-10e5.txt

Using the command-line tool to count the number of lines in the file, you can see that this file has 100,000 lines (plus a trailing newline).

The file is about 8 MB in size.

Now that you have a bit of a larger file (which you know has exactly 300,000 words starting with *a*), you can run your pipeline and test whether it works:

```
$ python counter.py --input=input-10e5.txt --output=output-10e5- \
  --runner=DirectRunner
$ cat output-10e5-*
300000
```

As expected, your pipeline confirms that the file contains exactly 300,000 words that start with the letter *a*.

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 your sentence. Keep in mind that if you do that, you'll probably be waiting around for a little while for the pipeline to finish, because 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 you take this a step further? In this example, all you did was take a local file, run it through your pipeline, and save the output back to the local file system. Let's look at what happens when you move this scenario out of your local world and into the world of Cloud Dataflow.

20.3.4 Executing a pipeline using Cloud Dataflow

Luckily, because you've done all your 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, upload your files to the Cloud Storage bucket you created previously and then use the Cloud Dataflow runner to execute the pipeline:

Uses the gsutil command-line tool to upload your input data to your Cloud Storage bucket. (The -m flag tells GCS to use multiple concurrent connections to upload the file.)

```
$ gsutil -m cp input-10e5.txt gs://dataflow-bucket/input-10e5.txt
$ python counter.py \
  --input=gs://dataflow-bucket/input-10e5.txt \
```

Instructs your pipeline to use the newly uploaded file stored in your Cloud Storage bucket as the input data.

```
--output=gs://dataflow-bucket/output-10e5- \
--runner=DataflowRunner
```

You also want the output result to live in the same Cloud Storage bucket, but you'll use a special prefix to keep track of the variety of files Cloud Dataflow will create during the pipeline execution.

Finally, instead of using the DirectRunner like before, you'll use Cloud Dataflow by specifying to use the DataflowRunner.

Once you press Enter, under the hood Cloud Dataflow will accept the job and get to work turning on resources to handle the computation. As a result, you should see some logging output from Cloud Dataflow that shows it has submitted the job.

> **TIP** If you get an error about not being able to figure out what gs:// means (for example, 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 you keep an eye on the progress of the job? And how will you know when it's done? If you navigate to the Cloud Dataflow UI inside the Cloud Console, you'll see your newly created job in the list (you specified the job name in the pipeline code from listing 20.2) and clicking on it will show a cool overview of the process of your job (figure 20.11).

First, on the left side of the screen, you'll see a graph of your pipeline, which looks similar to the drawing we looked at before you started. This is a good sign; Dataflow has the same understanding of your pipeline that you intended from the start. In this case, you'll notice that most of the work completes quickly, almost too quickly. You never get to see the work in progress because by the time the diagram is updated with the latest status, the work is mostly done. As a result, you only see how each stage moves from Running to Succeeded, and the entire job is over in a few minutes.

> **NOTE** You may be wondering why the job took a few minutes (about five in this case), whereas each stage of the processing took only a few seconds. This is primarily because of the setup 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 you see that all stages take about one minute when summed up, the total runtime (from job submission to completion) adds a bit of time before and after.

On the right 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, you can see the details about the computing and storage resources that the job consumed during its

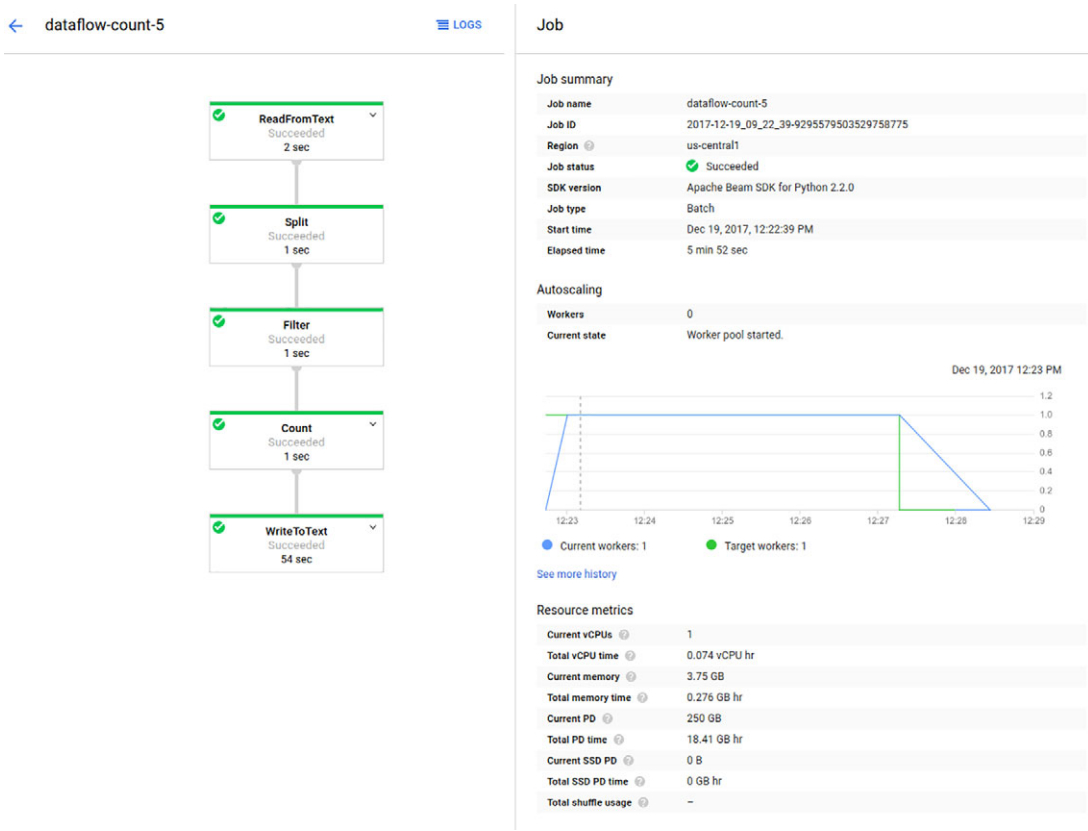


Figure 20.11 Overview of the pipeline job on Cloud Dataflow

lifetime. In this case, it used about 276 MB-hours' worth of memory and less than 0.07 vCPU-hours' worth of compute time.

This is neat, but a five-minute-long job that consumes only a few minutes' worth of computing time isn't that interesting. What happens if you increase the total number of lines to 10 million (`10**7`)? Try that and see what happens:

```
$ python -c "print (raw_input() + '\n') * 10**7"
➥ < input.txt > input-10e7.txt
$ gsutil -m cp input-10e7.txt \
  gs://dataflow-bucket/input-10e7.txt
$ python counter.py \
  --input=gs://dataflow-bucket/input-10e7.txt \
  --output=gs://dataflow-bucket/output-10e7- \
  --runner=DataflowRunner
```

Generates the 10 million lines of text

Uploads it to your Cloud Storage bucket like before

Reruns the same job but uses the 10 million lines of text as the input data

In this case, when you look again at the overview of your new job in the Cloud Console, there's enough data involved so that you 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 (figure 20.12).

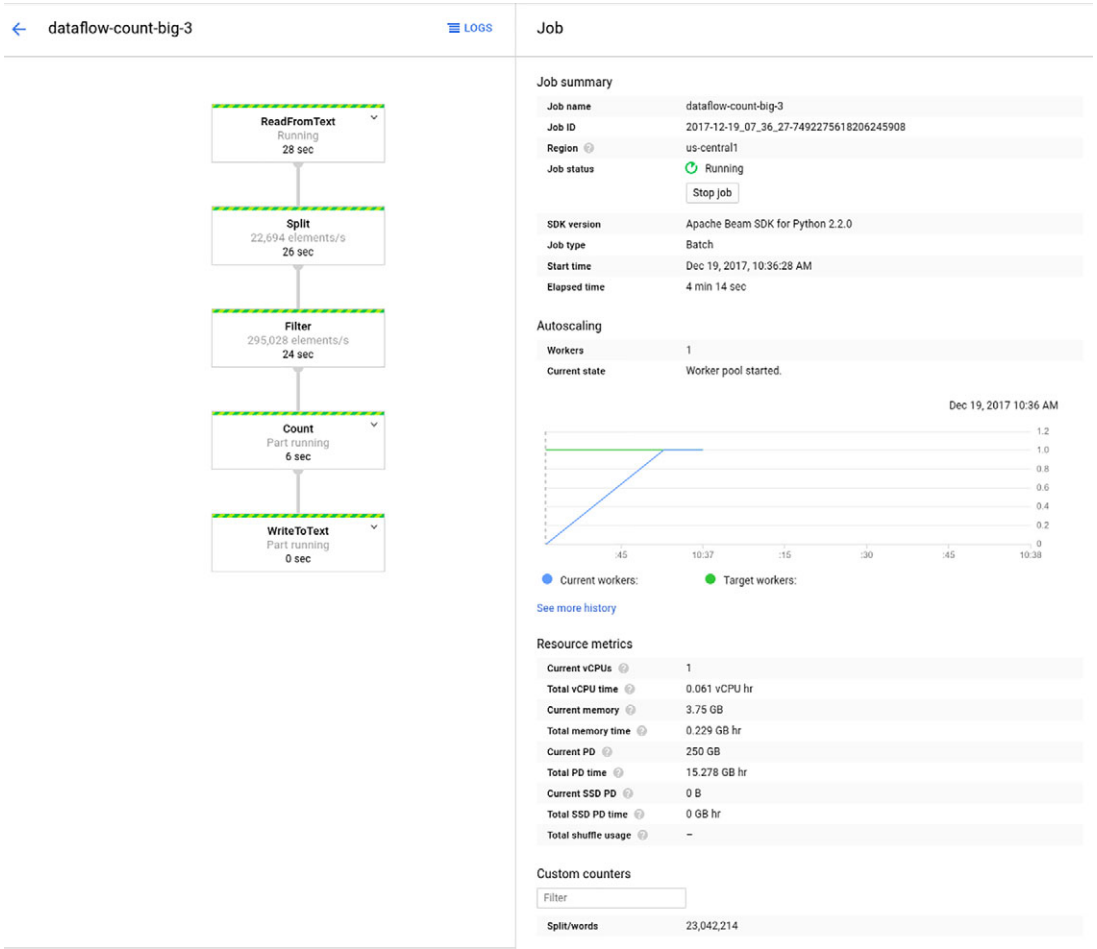


Figure 20.12 Overview of a larger pipeline job in progress

Perhaps the coolest part about this is that you can see how data flows through the pipeline, with each stage being active simultaneously. You can see that each step executes concurrently rather than completing entirely before moving on to 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 your graph (`ReadFromText`), the data is broken up at a high level into manageable-sized chunks using newline tokens. These chunks are then passed along to the `Split` step, which will separate them into words. From there, the data is 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 the final items passed through) and saves the final output back to your Cloud Storage bucket.

Another interesting thing is that you can see how many elements are being processed at each stage on a per-second basis. For example, in figure 20.12, you can see that in the `Split` stage (which, you'll recall, is where a given chunk of text is turned into a list of individual words), you're processing about 22,000 lines per second and outputting them as lists of words.

At the next stage, you're taking in lists of words as elements and filtering out anything that doesn't start with the letter *a*. If you look closely, you'll notice that this stage is processing about 13 times the amount that `Split` is. Why is that? Each line of your input has 13 words in it, so that stage is getting the output from ~22,000 lines per second split into 13 words per line, which comes to about 295,000 words per second.

Once the job completes, Cloud Dataflow writes the total count to your Cloud Storage bucket (as you see in the `WriteToText` stage). Verify this by checking the output files on Cloud Storage to see what the final tally is:

```
$ gsutil cat gs://dataflow-bucket/output-10e7-*
30000000
```

The final output is stored in Cloud Storage, showing the correct result of 30 million.

Because you put in 10 million lines of text, each one with three words starting with the letter *a*, the total comes to 30,000,000.

Additionally, after the job is finished, you have the gift of hindsight, where you can look at the amount of computing resources that your job consumed (figure 20.13). As expected, more data means you might want to use more computing power to process that data, and Cloud Dataflow figured this out as well.

You can see in the graph on the right side that it turns on a second VM to process data a few minutes after starting. This is great, considering you didn't change any code! Instead of you having to think about the number of machines you might need, Cloud Dataflow figured it out for you, scaled up when needed, and scaled down when the work was complete. Now that you've seen how to run your pipeline, all that's left is to look at how much all of this will cost!

20.4 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 expected, 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 are shown in table 20.1.

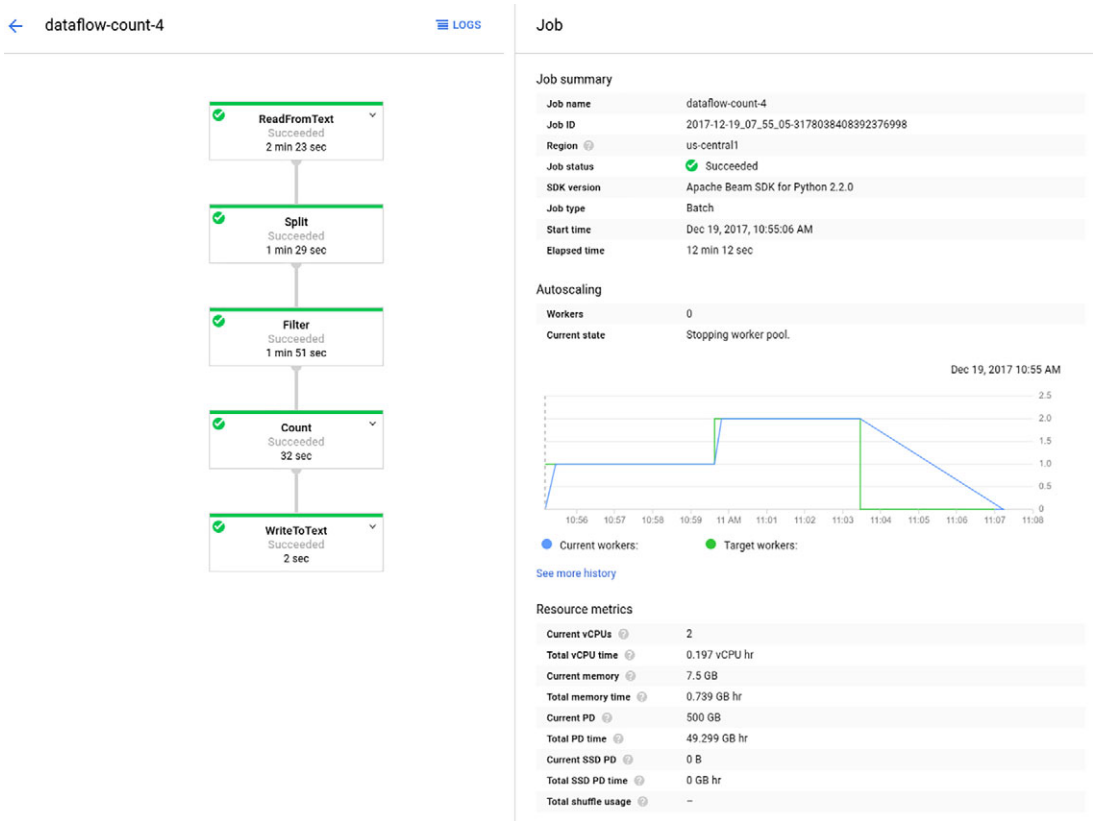


Figure 20.13 Overview of a successful job

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 knowing these handy rates, predicting the total cost ahead of time can be tricky. Each pipeline is different (after all, you're not always trying to count words starting with the letter *a*), and usually the input data varies quite a bit. The number of VMs used in a particular job tends to vary, and it often follows that the amount of disk space and memory used in total will vary as well. Luckily, you can do a couple of things.

First, if your workload is particularly cost-sensitive, you can set a specific number of workers to use (or a maximum number), which will limit the total cost per hour of your job. But this may mean your job could take 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 you 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, you can assume that a run over 10x the amount of data will cost roughly 10x as much. To make this more concrete, in your previous pipeline job that counted the words across 10 million lines of text (as shown in figure 20.12), you 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 your total to ~$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 you processed 100 million lines of text that had a similar distribution of words, the cost for the workload likely would scale linearly to about $0.16.

Summary

- When we talk about data processing, we mean the idea of taking a set of data and transforming it into something more useful for a particular purpose.
- Apache Beam is one of the open source frameworks you can use to represent data transformations.
- Apache Beam has lots of 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

- Distributed messaging systems in general
- When and how to use Cloud Pub/Sub in your application
- How Google calculates Cloud Pub/Sub pricing
- Two examples using common messaging patterns

If you've ever sent an SMS or Facebook message, the concept of messaging should feel familiar and simple. That said, in your day-to-day use of messaging, you have a few requirements that you sometimes take for granted. For example, you expect that messages are

- sent from one specific person (you)
- sent to exactly one specific person (your friend)
- sent and received exactly once (no more, no less)

Just like people, machines often need to communicate with one another, particularly in any sort of large distributed application. As you might expect, machine-to-machine communication tends to have requirements similar to the ones you have.

21.1 The headache of messaging

Meeting these requirements isn't as easy as it looks. Beyond that, you might broadcast messages to a group, which has slightly different requirements (for example, 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. Messaging might seem simple, but it's pretty tricky.

A lot of open source messaging platforms (like Apache Kafka and ZeroMQ) and a variety of standards (like AMQP) are available to handle this trickiness, each with its own benefits and drawbacks, but they all tend to require you to 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. 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) that Google built on top of its internal infrastructure because its messaging needs were similar to those of many other companies. The infrastructure that Google Cloud Pub/Sub uses is the same as the lower level infrastructure that other services internal to Google, such as YouTube and AdWords, use.

Luckily, Google Cloud Pub/Sub uses concepts that are common across many of those open source messaging services I mentioned. Because the concepts have so much overlap, if you're familiar with another messaging system, you should have little trouble using Cloud Pub/Sub. 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 would 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 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." But you can't send a message out into the world without categorizing it in some way. You have to decide what the message is about and publish it specifically to that topic. As a result, this producer first decides on a category (called a *topic*) and then publishes a message specifically to that topic.

Once Cloud Pub/Sub receives the message, it'll assign the message an ID that's unique to the topic and will return that ID to the producer as confirmation that it received the message (figure 21.1). Think of this flow as a bit like calling an office and the receptionist saying, "I'll be sure to pass along the message."

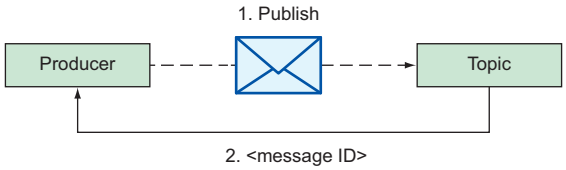


Figure 21.1 The message publishing flow

Now that the message has arrived at Cloud Pub/Sub, a new question arises: who's interested in this message? To figure this out, Cloud Pub/Sub uses a concept of *subscriptions*, which you might create 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 them (figure 21.2). Much like a work queue, subsequent messages sent to the topic will queue up on each subscription so someone might read them later. Think of this as the receptionist photocopying each message once for each department and putting it in each inbox at the front desk.

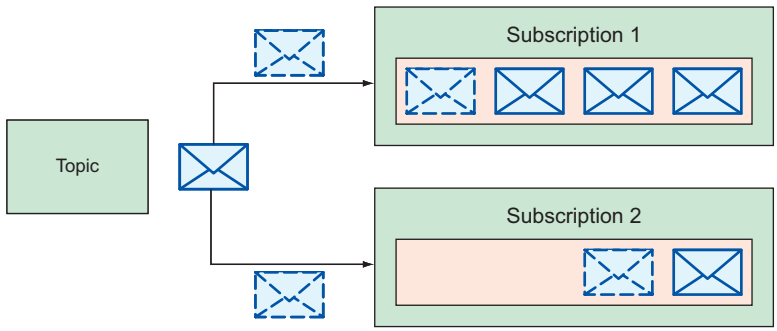


Figure 21.2 The subscription message routing flow

This brings us to the end of the road from the perspective of the producer—after all, Pub/Sub has received the message and it's in the queue of everyone who expressed interest in receiving messages about that topic. But the message still isn't delivered! To understand why, we must shift our focus from the producer of a message to the receiver of that message, which is called a *consumer*.

Once a message lands in the queue of a subscription, it can go one of two ways, depending on how the subscription is configured. Either the subscription can *push* it to an interested consumer, or the message can sit around and wait for that consumer to *pull* it from the subscription. Let's look quickly at these two options.

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 them with the `pull` API method. This is a bit like the receptionist leaving the box of messages on the desk until someone from the department comes to collect them. The difference between these two is shown in figure 21.3—make sure to pay special attention to the direction of each of the arrows.

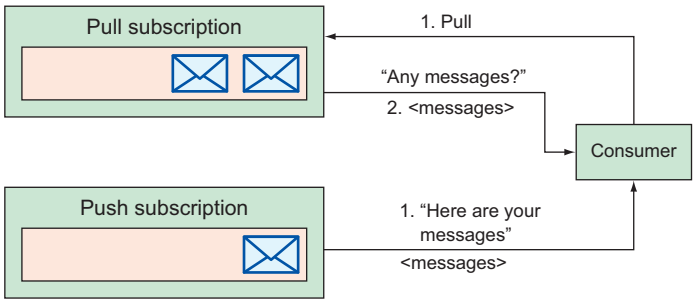


Figure 21.3 Push versus pull subscription flows

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 gets to the consumer, the work must be done, right? Not quite! A final step is required, where the consumer needs to acknowledge that it has received and processed the message, which is called *acknowledgment*.

Just because a consumer gets a message doesn't necessarily mean the system should consider that message processed. For example, it's possible that although the message is delivered to the consumer, their computer crashes in the middle of processing it. In that scenario, you wouldn't want the message to get dropped entirely; you'd want the consumer to be able to pick it up again later when it has recovered from the crash.

To complete the process, consumers must acknowledge to Cloud Pub/Sub that they've processed the message (figure 21.4). They do this by using a special ID they get with the message, called an `ackId`, which is unique to that particular lease on the message, and calling the `acknowledge` API method. Think of this as like a package being delivered to the front desk, and the receptionist asking you to sign for it. This serves as a form of confirmation that not only was the package sent over to you, but you received it and have taken responsibility for it.

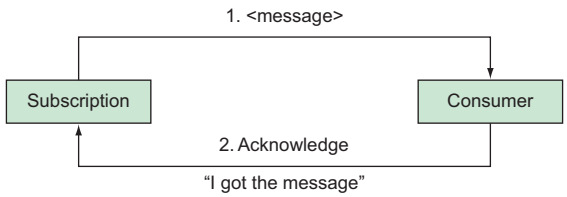


Figure 21.4 Message acknowledge flow

What happens when a consumer crashes before it gets 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 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 of the important concepts individually and explore some of the details about them.

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 concept 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 the example, topics, much like topics of conversation, represent categories of information and are the resource that you publish a message to. Because you publish messages to a specific topic, topics are required when broadcasting messages.

For example, you might have different departments in a company, and although you may always call the main number, you may want to leave messages with different departments depending on your reason for calling. If you're looking to buy something from the company, you may want to leave a message specifically with the sales department. Alternatively, if you 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* that 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 the example I've described, if you work in the support department, you'd 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 I discuss subscriptions. Finally, unlike with most resources I've discussed on Google Cloud Platform, you represent a topic as nothing more than its name. That's all you need, because consumers will handle any customization and configuration, which you'll see in section 21.4.3. Now that you understand topics, let's move on to 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 (for example,

"Someone just signed up in your app!") to a regularly scheduled reminder (for example, "It's midnight; you may want to run a database backup!"). Messages, as you just learned, are always published to a specific topic, which acts as a way to categorize the message, effectively saying what it's 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) that act as metadata about it.

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.

Along with the payload and the attributes that the sender sets, the Cloud Pub/Sub system assigns two additional fields—a message ID and a timestamp of when the message was published—but only when you publish a message. 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 arises: why do you need two places to store the data that you're sending? Why separate the payload from the attributes? Two reasons are involved.

First, the payload is always base-64 encoded, whereas attributes aren't, so to do anything meaningful with the data stored in that field, consumers must decode the payload and process it. As you might expect, if the payload is particularly large, you might have significant performance issues to worry about. For example, imagine sending a large attribute-style map exclusively as a base-64 encoded payload. If message consumers check a field to decide whether they need to pay attention to a message, they would have to decode the entire payload, which could be large. This would obviously be wasteful and is easily fixed by making this particular field a message attribute that *isn't* base-64 encoded, so consumers can check it before doing any decoding work on the payload.

Second, for a variety of reasons, messages may be encrypted before they go to Cloud Pub/Sub. In this case, you have a similar problem to the one I described in the previous paragraph (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 (for example, encrypted messages, each with a `priority` field that could be `low`, `medium`, or `high`). If you sent the priority along with the encrypted payload, the messaging system would have to decrypt the message to decide what type of notification to send to the recipient. If instead you sent the priority in the plain text attributes, the system could inspect the less critical data (such as message priority) without decrypting the message content itself. Now let's look at subscriptions and how they work.

21.4.3 Subscriptions

Subscriptions, sometimes referred to as queues in other messaging systems, represent a desire or intent to listen to (or consume) messages on a specific topic. They also take over most of the responsibility relating to configuration of how consumers will receive messages, which allows for some interesting consumption patterns depending on the particular configuration.

Subscriptions have three important characteristics:

- Each subscription receives a distinct copy of each message sent to its topic, so consumers can access messages from a topic without stepping on the toes of others who are interested in that topic's messages. A consumer reading a message from one subscription has no effect at all on other subscriptions.
- Each subscription sees all of the messages you send on a topic, so you can broadcast messages to a wider audience if more consumers create subscriptions to the topic.
- Multiple consumers can consume messages from the same subscription, so you can use subscriptions to distribute messages from a topic across multiple consumers. Once one consumer consumes a message from a subscription, that message is no longer available on that same topic, so the next consumer will get a different message. This arrangement ensures that no two consumers of that subscription will end up processing the 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 get their messages. As you learned, 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 up subscriptions, let's look briefly at the idea of acknowledging messages that arrive.

ACKNOWLEDGEMENT DEADLINES

I explained earlier that you have to acknowledge you received a message before it's 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 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.

When another consumer of the same subscription (Consumer 2) pulls a message, it gets the message (4) that Consumer 1 didn't acknowledge. It acknowledges receipt of the message (5), which concludes the process of consuming that particular message.

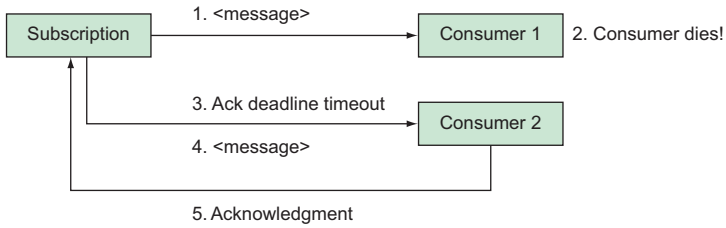


Figure 21.5 Acknowledgment expiration flow

Now that you understand all of the concepts (including how messages must be acknowledged), let's look at 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 a producer is sending three messages (A, B, and C) to two topics (1 and 2). Based on the diagram, four different consumers (1, 2, 3, and 4) ultimately receive these messages.

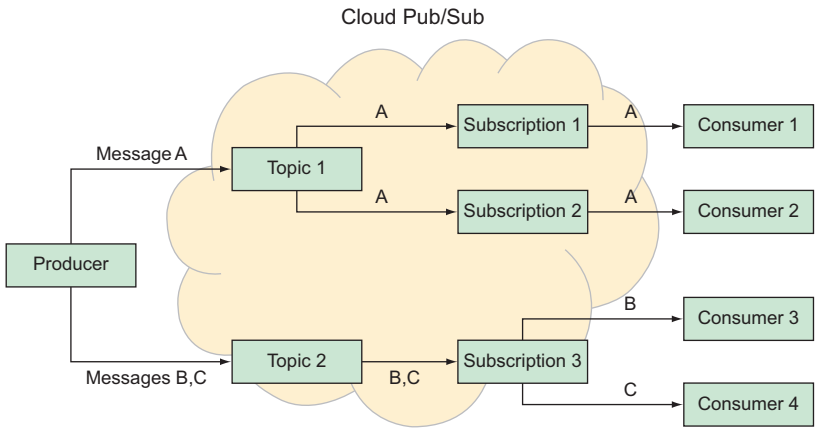


Figure 21.6 Example of sending messages with different subscriptions

Let's start by looking at message A, which the Producer is sending to Topic 1. In this example, two consumers (Consumer 1 and Consumer 2) each have their own subscription. Because subscriptions to a topic get their own copy of all the messages sent to that topic, 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 the Producer sends 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. Because subscriptions only get one copy of each message, and a message is no longer available once a consumer consumes it, the two messages (B and C) will be split. The likely scenario will be that one of them will go to Consumer 3 (B in this example) and the other (C in this example) to Consumer 4. The end result of have multiple consumers to a single subscription is that they end up splitting the work, with each getting some portion of all the messages sent.

But keep in mind that this is the *likely* scenario, not guaranteed. The messages might also be swapped (with Consumer 3 getting message C and Consumer 4 getting message B), or one consumer might get both messages B and C (for example, if one of the consumers is overwhelmed with other work). Now that I've gone over how topics and subscriptions fit together, let's get down to business and use them.

21.5 Trying it out

Before you start writing code to interact with Cloud Pub/Sub, you have to enable the API. To do this, visit the Cloud Console in your browser, and in the main search box at the top type Cloud Pub/Sub API (remember the forward slash). This search should have only one result, and clicking it should bring you to a page with a big Enable button (figure 21.7).

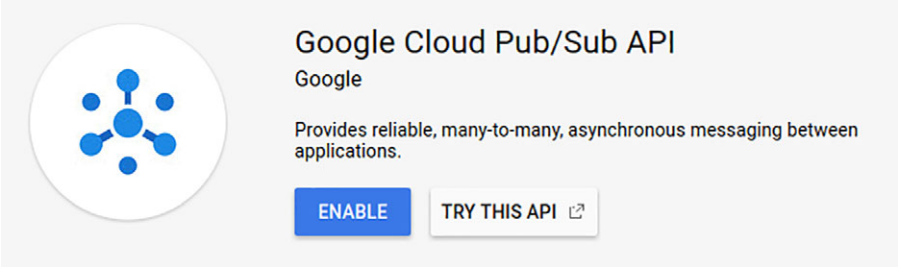


Figure 21.7 The button for enabling the Cloud Pub/Sub API

Once you've enabled the API, you're ready to go. You can 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. Although you can create a topic in code, start here by creating one in the Cloud Console. In the left navigation bar, far toward the bottom under Big Data, click the Pub/Sub entry. The first thing you should see is an empty page with a button suggesting that you create a topic (figure 21.8), so do that.

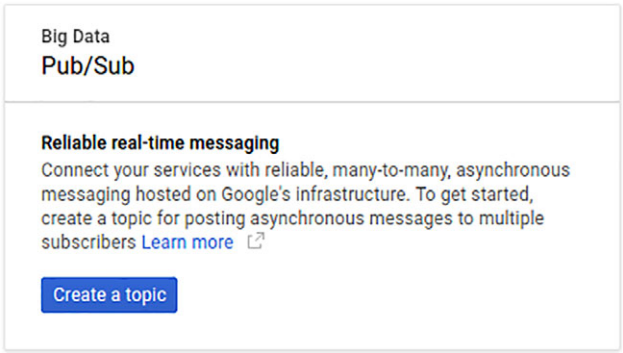


Figure 21.8 The Cloud Pub/Sub page where you can create a topic

After you click the button, you'll see a place to enter a topic name (figure 21.9). Notice that the fully qualified name is a long path starting with projects/. This provides a way to uniquely identify your topic among all of the topics created in Cloud Pub/Sub. Choose a simple name for your first topic: `first-topic`.

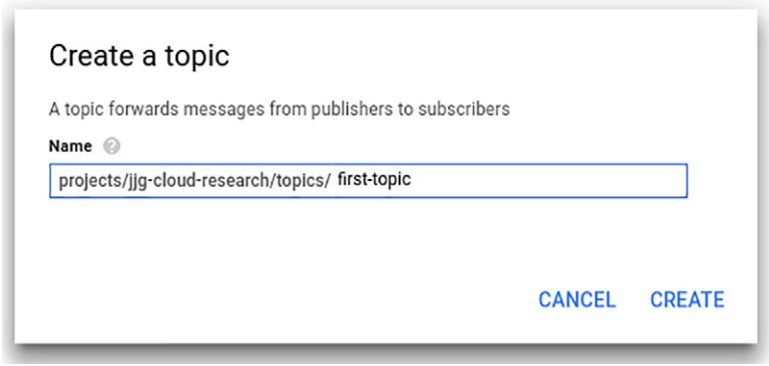


Figure 21.9 Creating a topic in Cloud Pub/Sub

After you click Create, you should see your topic in the list. This means you're ready to start writing code that sends messages! Before you write any code, you'll first need to install the Node.js client library for Cloud Pub/Sub. To do this, you can use `npm` by running `npm install @google-cloud/pubsub@0.13.1`. Once that's done, you can write some code, such as the following listing.

Listing 21.1 Publishing a message

```
constpubsub = require('@google-cloud/pubsub')({
  projectId: 'your-project-id'
});
```

Access the Pub/Sub API using the API client found in the npm package @google-cloud/pubsub.

```
const topic = pubsub.topic('first-topic');

topic.publish('Hello world!').then((data) => {
  constmessageId = data[0][0];
  console.log('Message was published with ID', messageId);
});
```

Because you already created this topic in the Cloud Console, you can access it without checking that it exists.

To publish a message, use the publish method on your topic.

Publishing messages returns a list of message IDs, but you only want the first one.

If you were to run this code, you'd see something like the following:

```
> Message was published with ID 105836352786463
```

The message ID that you're seeing here is an identifier that's guaranteed to be unique within this topic. Later, when I 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 isn't all that valuable, right? (After all, if no one is listening, Cloud Pub/Sub drops the message.) How do you go about receiving them? Let's get to work on receiving some messages from Cloud Pub/Sub.

21.5.2 Receiving your first message

To receive messages from Cloud Pub/Sub, you first need to create a subscription. As you learned before, subscriptions are the way that you consume messages from a topic, and each subscription on a topic gets its own copy of every message sent to that topic. You'll start by using the Cloud Console to create a new subscription to the topic you already created. To do this, head back to the list of topics in the Pub/Sub section, and click the topic name. It should expand to show you that there currently are no subscriptions, but it also should provide a handy New Subscription button at the top right (figure 21.10).

Go ahead and click the button to create a new subscription, and on the following page (figure 21.11) you can keep with the theme and call the subscription

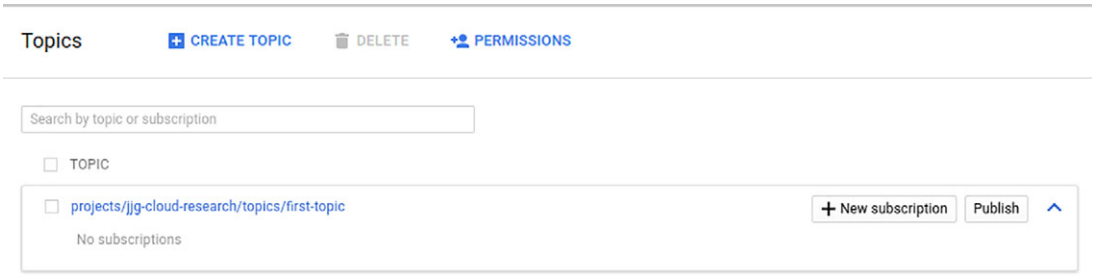


Figure 21.10 The list of topics with a New Subscription button

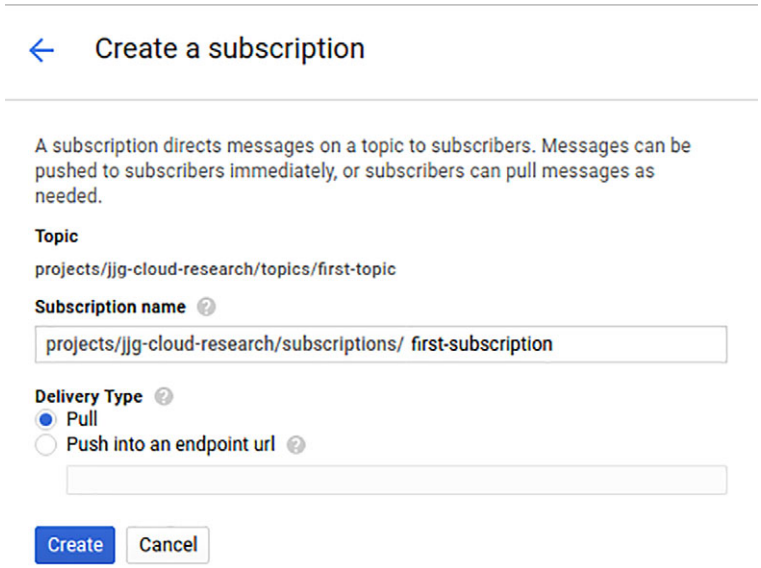


Figure 21.11 Creating a new subscription to your topic

`first-subscription`. Under Delivery Type, leave this as Pull for now. We'll walk through Push subscriptions later on.

Once you click Create, you should be brought back to the page listing all of your topics. If you click on the topic you created, you should see the subscription that you created in the list (figure 21.12).

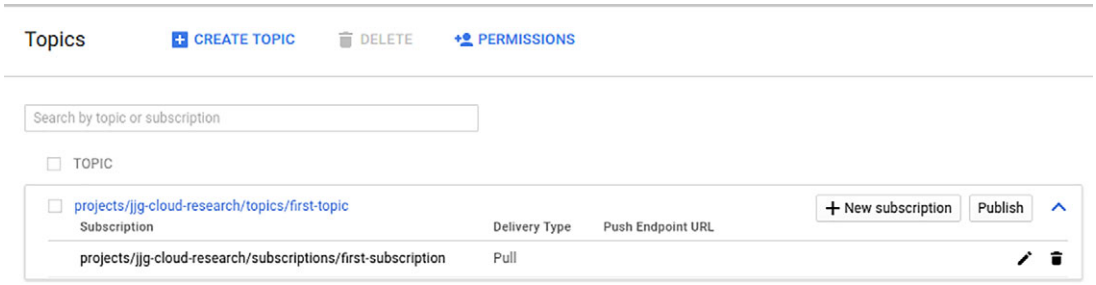


Figure 21.12 Viewing a topic and its subscriptions

Now that you have a subscription, you can go ahead and 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. You have to send a message to your topic and then ask the subscription for any messages received. To do that, start by running the script from

listing 21.1 that publishes a message to your topic. When you run it, you should see a new message ID:

```
> Message was published with ID 105838342611690
```

Because your topic has a subscription this time, you weren't sending a message into the void. Instead, a copy of that message should be waiting for you when you ask your subscription for messages. To do that, you'll use the `pull` method on a subscription, as shown in the following listing.

Listing 21.2 Consuming a message

```
constpubsub = 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);
});
```

Gets a reference to a topic by using its name

Similarly references a subscription using its name

Consumes messages with the pull() method on your subscription

```
> Got message 105838342611690 saying Hello world!
```

Notice here that the message ID is the same as the one you published, and the message content is the same also! Looks good, right? It turns out that you've forgotten an important step: acknowledgment!

If you try to run that same code again in about 10 seconds, you'll get that exact same message, with the same message ID, again. What's happening under the hood is that the subscription knows it gave that message out to the consumer (your script), but the consumer never acknowledged that it got the message. Because of that, the subscription responds with the same message the next time a consumer tries to pull messages. To fix this, you can use a simple method on the message called `ack()`, which makes a separate request to Pub/Sub telling it that you did indeed receive that message. Your updated code will look something like the following listing.

Listing 21.3 Consuming and acknowledging a message

```
constpubsub = 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(() => {
    console.log('Acknowledged message ID', message.id,
    'with ackId', message.ackId);
  });
});
```

message.ack() is bound to the right ackId, meaning you don't need to keep track of lots of IDs.

Notice that the ackId is different from the message ID. Cloud Pub/Sub is set up this way because multiple people may consume the same message.

You should see that running this code receives the same message again, but this time the consumer tells Cloud Pub/Sub that it received the message by sending an acknowledgement:

```
> Got message 105838342611690 saying Hello world!
Acknowledged message ID 105842843456612 with ackId
QV5AEkw4A0RJUytDCypYEU4EISE-
MD5FU0RQBhYsXUZIUTcZCGhRDk9eIz81IChFEQcIFAV8fXFdUXVeWhoHUQ0ZcnxkfDhdRwkAQAV5V
VsRDXptXFc4UA0cenljfW5ZFwQEQ1J8d5qChutoZho9XxJLLD5-MzZF
```

If you were to try pulling again, you'd see that the message has disappeared. 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 you've received you've pulled from the subscription. You've specifically asked a subscription to give you any available messages. As I mentioned earlier, though, another way of consuming messages doesn't necessarily require you to ask for them. Instead of you *pulling* messages from Cloud Pub/Sub, Cloud Pub/Sub can *push* messages to you as they arrive.

These types of subscriptions require you to configure where Cloud Pub/Sub should send push notifications when a new message arrives. Typically, Pub/Sub will make an HTTP call to an endpoint you specify containing the same message data that you saw when using regular pull subscriptions. What does this process look like? How do you handle push notifications?

First, you need to write a handler that accepts an incoming HTTP request with a message body. As you saw before, once the handler receives the message, it's responsible for acknowledging the message, but the way you acknowledge pushed messages is a bit different. With a pull subscription, you make a separate call back to Cloud Pub/Sub, letting it know you've received and processed the message. With a push subscription, you'll rely on HTTP response codes to communicate that. In this case, an HTTP code of 204 (No Content) is your way of saying you've successfully received and processed the message. Any other code (for example, a 500 Server Error or 404 Not Found) is your way of telling Cloud Pub/Sub that some sort of failure occurred.

Put more practically, if you want to handle push subscriptions, you'll need to write a handler that accepts a JSON message and returns a 204 code at the end. A handler like that might look something like the following listing, which uses Express.js.

Listing 21.4 A simple push subscription handler

```
const express = require('express');
const app = express();

app.post('/message', (req, res) => {
  console.log('Got message:', req.message);
  res.status(204).send()
});
```

First you do something with the message. In this case, you log it to the console.

To acknowledge that you've handled the message, you explicitly return a 204 response code.

Now that you've seen what a handler for incoming messages might look like, this only leaves the question of how you 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. You can do this in lots of ways, but I'll show you how to do this using the Cloud Console.

In the Pub/Sub area of the console, you can reuse the topic you created before (`first-topic`) and skip ahead to creating a new subscription. Imagine you've deployed your simple Express.js application with the basic handler to your own domain (for example, `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 for where to push messages. In figure 21.13, I'm using `push-subscription` as the name and https://your-domain.com/message as the URL (note that in listing 21.4, the path is `/message`).

Once you click Create, Cloud Pub/Sub will route incoming messages to this topic to your handler, with no pulling or acknowledging needed.

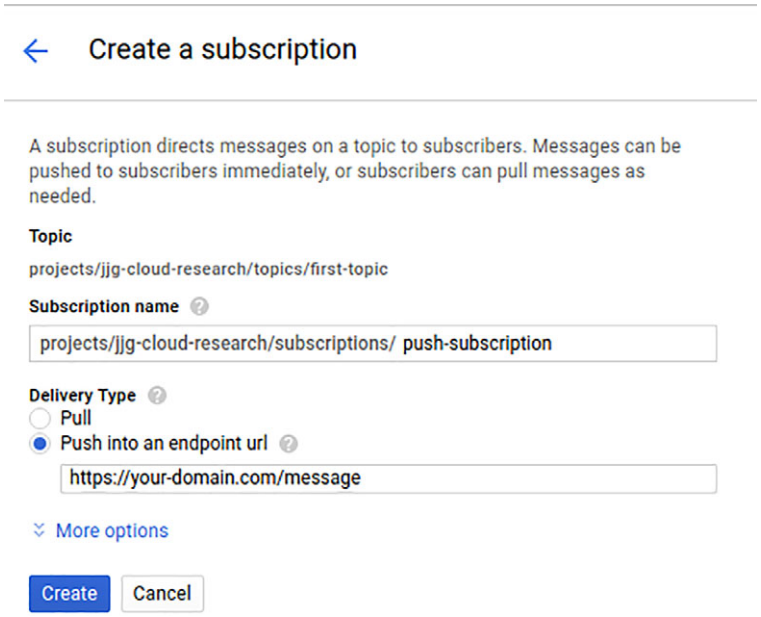


Figure 21.13 Creating a push subscription

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 messages are only sent to domains that you own. To read more about whitelisting your domain for use as a Pub/Sub push endpoint, check out https://cloud.google.com/pubsub/docs/push#other-endpoints.

At this point, you should have a good grasp of 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 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 only charges you for the resources and computation that you actually use. To do this, Pub/Sub bills based on the amount of data you broadcast through the system, at a maximum rate of $0.06 per Gigabyte of data. Although this loosely corresponds to the number of messages, it'll depend on the size of the messages you're sending. For example, I mentioned before how instead of sending an entire video file's content through a message, you might instead send a link to the video. The reason for this is not only that sending large video files isn't exactly what Pub/Sub was designed for, but also that sending a link will cost you a lot less. It's likely it'll be far cheaper to download the video file from somewhere else, such as Google Cloud Storage.

To make this more concrete, let's look at a more specific example. Imagine your system sends five messages every second, and you have 100 consumers interested in those messages. Assume that these messages are tiny. How much will this cost you over the course of a month? First, let's look at how many requests you're making throughout the month. To do that, you'll need to know how many messages you'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 you to 2,592,000 seconds in one of your months.
- Because you're sending five messages per second, you're sending a total of 12,960,000 messages in one of your months.

Now you have to think about what requests you're making to Cloud Pub/Sub. First, you make one `publish` request per message because that's absolutely required to send the message. But you 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. You make one additional request per consumer for each message:

- You send 12,960,000 messages in one of your months.
- Because you have 100 consumers, you make a total of 1,296,000,000 pull (or push) requests to read those messages in one of your months.

This brings your overall total to

- 12,960,000 publish requests, plus
- 1,296,000,000 pull (or push) requests

which comes to a grand total of 1,308,960,000 requests.

The question now becomes how you convert this to data. To do that, it's important to know that the minimum billable amount for each request is 1 KB. Even if your messages are tiny, the total amount of data here is about 1.3 billion KB, or slightly under 1.31 TB. At the rate of $0.06 per GB of data, your bill at the end of the month for these 13 million messages sent to 100 consumers will be $78.60.

21.8 Messaging patterns

Although you've written a bit of code to communicate with Cloud Pub/Sub, the examples have been a bit simplistic in that they assume static resources (topics and subscriptions), when in truth you'll often want to dynamically change your resources. For example, you may want every new machine that boots up to subscribe to some systemwide 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

A fan-out system uses 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, each of which is automatically bidding on items on eBay, and you want to keep track of how much money you've spent overall. Because you have multiple servers all bidding on items, you need a way of communicating to everyone how much money they've 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 to a specific Pub/Sub topic the fact that it has spent money (figure 21.14).

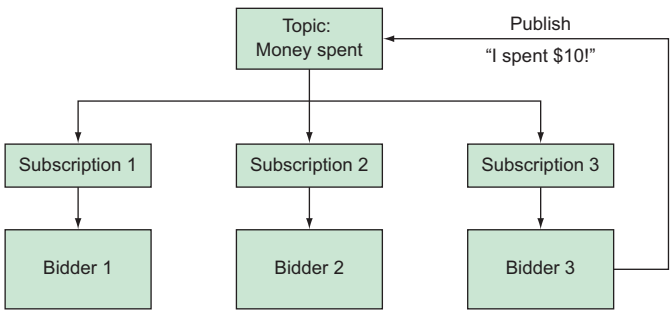


Figure 21.14 Overview of message flow for machines acting as eBay bidders

This enables all the other machines listening on this topic to keep track of how much money was spent in total, and they're immediately notified of any new expenditure.

Using the concepts of Cloud Pub/Sub that you learned about before, this would correspond to a topic called `money-spent`, where each interested consumer (or bidder in the example) would have its own subscription. In this way, each subscriber would be guaranteed to get each message published to the topic. Furthermore, each of these consumers also would be producers, telling the topic when they spend money as it happens in real time.

As you can see, each bidder machine has exactly one subscription to the `money-spent` topic and can broadcast messages to the topic to notify others of money it spends. You can write some code to do all of this, as shown in listing 21.5, starting with a method that you should expect to run whenever one of your eBay bidder instances turns on. To be explicit, this method is ultimately responsible for doing a few key things:

- Getting the total spend count before starting to bid on eBay
- Kicking off the logic that bids on eBay items
- Updating the total amount spent whenever it changes

Listing 21.5 Function to start bidding on eBay items

```
const request = require('request');
constpubsub = require('@google-cloud/pubsub')({
  projectId: 'your-project-id'
});

letmachineId;
const topic = pubsub.topic('money-spent');
constamountSpentUrl = 'http://ebaybidder.mydomain.com:
➥ 8080/budgetAvailable.json';
letamountSpent;

startBidding = () =>{
  request(amountSpentUrl, (err, res, body) => {
    amountSpent = body;

    const subscription = topic.subscription
➥ (machineId + '-queue');

    subscription.on('message', (message) => {
      console.log('Money was spent!', message.data);
      amountSpent += message.data;
      message.ack();
    });

    bidOnItems();
  });
}
```

Uses a well-known name (money-spent) for your topic, with the assumption that it's already been created

Assumes that this is the method you call first when a new bidding machine is turned on, which retrieves the available budget from a central location

Uses a special subscription name so that it'll either be created or reused if it already exists. (The assumption is that you have a unique name [machineId] for each individual bidding VM.) This subscription listens for money being spent.

Method called every time a new message appears on the topic

Updates the amount spent whenever a new message arrives. (Note that the amount is a delta. For example, "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 clearer in listing 21.6.)

Starts bidding on eBay items

Acknowledges that you received the message and processed it

With this code written, you need to devise how you update the amount spent when you bid (or refunded when you get outbid) on items. To do that, let's first think through all the ways this value can change. The obvious way is when you place a bid on an item. If I place a bid of $10 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 to the amount spent, as the default action is that I've 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. When you place a bid, you need to mark money as spent, and when you're outbid on an item, you need to mark that money as recovered or refunded. You can turn those two actions into (pseudo-) code, as shown in the following listing.

Listing 21.6 Functions to update the amount spent locally

```
constpubsub = require('@google-cloud/pubsub')({
projectId: 'your-project-id'
});

letmachineId;
const topic = pubsub.topic('money-spent');

constbroadcastBid = (bid) =>{

returntopic.publish({
    data: bid.amount,
    attributes: {
machineId: machineId,
itemId: bid.item.id
    }
  }, {raw: true});
}

constbroadcastRefund = (bid) =>{
returntopic.publish({
    data: -1 * bid.amount,
    attributes: {
machineId: machineId,
itemId: bid.item.id
    }
  }, {raw: true});
}
```

Broadcasts that some money has been spent on a bid

This is a delta, so, for example, to convey that you spent$3, you need to send a value of 3.00.

For debugging purposes, sends along the machine that sent this message, as well as the eBay item ID in the message attributes

To send message data separate from message attributes, you need to tell your client library that this is a raw message. Otherwise, it'd treat the entire block as the payload.

Reclaims funds that are no longer pledged to a bid (for example, when you lose an auction)

Because you're releasing funds, you flip the sign of the value. When refunded $3.00, you add -3.00 to the amount spent.

Now that you've written the code, let's look at this from a high level to see exactly what's happening. First, each bidding machine turns on and requests the current budget from some central authority. (I won't go into exactly how that works, as it's not relevant here.) After that, each machine 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, the process of bidding on items begins. Whenever you bid on an item, you call the `broadcastBid` function to let others know you've placed a bid. Conversely, if you're ever outbid (or the auction is canceled), you call the `broadcastRefund` function, which will tell other bidders that money you had marked as spent is not spent. Now that you've seen how fan-out works, let's take a look at how you can use Pub/Sub to manage a queue of shared work across multiple workers.

21.8.2 Work-queue messaging

Unlike fan-out messaging, where you deliver each message to lots of consumers, work-queue messaging is a way of *distributing* work across multiple consumers, where, ideally, only one consumer processes each message (figure 21.15).

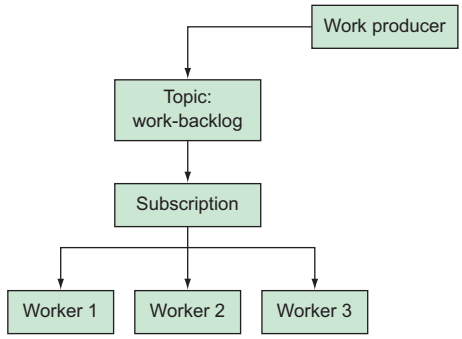


Figure 21.15 Work-queue pattern of messaging

Relying on the eBay bidding example, imagine you want to design a way to instruct all of your bidding machines to bid on a specific list of items. But instead of using a fixed list of items, say you'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 received that message, it would place a bid on the item. How should you lay out your topics and subscriptions?

If you followed the fan-out broadcast style of messaging, each bidding machine would get every message, which would mean each machine would place its own distinct bid on the item. That would get expensive! Here, you could use the work-queue pattern and have a single subscription that each of your bidders would listen to for notifications of messages (figure 21.16). By using this setup, a single machine, rather than every machine, would handle each message.

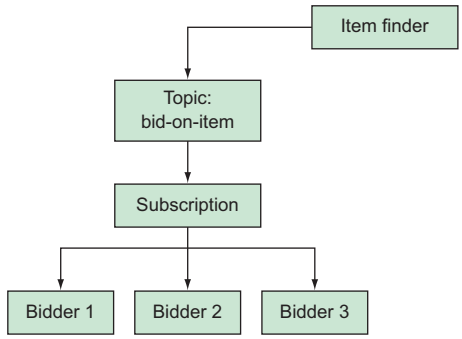


Figure 21.16 Item finder sending messages to bidders

How would this look? First, you'd create a new topic (`bid-on-item`), along with a single pull subscription (`bid-on-item-queue`). After that, you'd modify your bidding machines to consume messages from this new subscription and bid accordingly. Since all the notifications flow through a single subscription, each item will be consumed by

only one bidder on a first-come, first-served basis. Without this form of isolation, you might end up bidding against yourself, which would be a bad scenario. Assuming you create the topic and subscription manually, let's explore what your code would look like, as follows.

Listing 21.7 Functions to update the amount spent locally

```
constpubsub = require('@google-cloud/pubsub')({
projectId: 'your-project-id'
});

const topic = pubsub.topic('bid-on-item');
const subscription = topic.subscription('bid-on-item-queue');

subscription.on('message', (message) => {
message.ack(() => {
bidOnItem(message);
  });
});
```

Because you're dealing with static resources, you can construct references to your topic and subscription by names.

As before, registers a message-handling call-back, which is called as each message is received

Because the bidding process can be long, starts by acknowledging the message

After the acknowledgment succeeds, instructs your bidding machine to bid on the item

Notice that you're erring on the side of accidentally *not* bidding on items if some sort of problem occurs. For example, if the `bidOnItem` method throws an error for some reason, you've already acknowledged the message, so you 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!

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 senders and receivers.
- Producers can send messages to topics, which consumers can then subscribe 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!").
- Producers most commonly use Cloud Pub/Sub for fan-out (broadcast) or work-queue (orchestration) messaging.
- Cloud Pub/Sub charges based on the amount of data you send through the system, meaning larger messages cost more than smaller messages, with a minimum of 1 KB per message.

index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

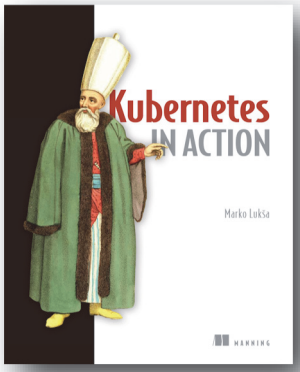
V

W

X

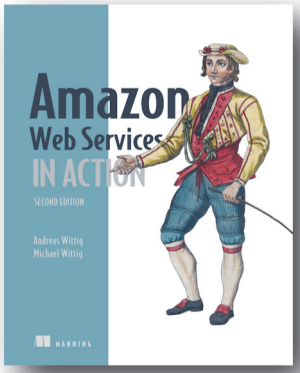
Z

MORE TITLES FROM MANNING



Kubernetes in Action
by Marko Lukša

ISBN: 9781617293726
624 pages
$59.99
December 2017



Amazon Web Services in Action, Second Edition
by Michael Wittig and Andreas Wittig

ISBN: 9781617295119
550 pages
$54.99
September 2018

Learn Amazon Web Services in a Month of Lunches
by David Clinton

ISBN: 9781617294440
328 pages
$39.99
August 2017

For ordering information go to www.manning.com

CLOUD

Google Cloud Platform IN ACTION

JJ Geewax

Thousands of developers worldwide trust Google Cloud Platform, and for good reason. With GCP, you can host your applications on the same infrastructure that powers Search, Maps, and the other Google tools you use daily. You get rock-solid reliability, an incredible array of prebuilt services, and a cost-effective, pay-only-for-what-you-use model. This book gets you started.

Google Cloud Platform in Action teaches you how to deploy scalable cloud applications on GCP. Author and Google software engineer JJ Geewax is your guide as you try everything from hosting a simple WordPress web app to commanding cloud-based AI services for computer vision and natural language processing. Along the way, you'll discover how to maximize cloud-based data storage, roll out serverless applications with Cloud Functions, and manage containers with Kubernetes. Broad, deep, and complete, this authoritative book has everything you need.

What's Inside

- The many varieties of cloud storage and computing
- How to make cost-effective choices
- Hands-on code examples
- Cloud-based machine learning

Written for intermediate developers. No prior cloud or GCP experience required.

JJ Geewax is a software engineer at Google, focusing on Google Cloud Platform and API design.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/google-cloud-platform-in-action

"Demonstrates how to use GCP in practice while also explaining how things work under the hood."

—From the Foreword by Urs Hölzle, SVP, Technical Infrastructure, Google

"Provides powerful insight into Google Cloud, with great worked examples."

—Max Hemingway
DXC Technology

"A great asset when migrating to Google Cloud, not only for developers, but for architects and management too."

—Michał Ambroziewicz, Netsprint

"As an Azure user, I got great insights into Google Cloud and a comparison of both providers. A must-read."

—Grzegorz Bernas
Antaris Consulting

ISBN-13: 978-1-61729-352-8
ISBN-10: 1-61729-352-0
55999
9 781617 293528

MANNING $59.99 / Can $79.99 [INCLUDING eBOOK]