



Google Cloud Platform **IN ACTION**

JJ Geewax

Foreword by Urs Hözle

 MANNING

Google Cloud Platform in Action

Google Cloud Platform in Action

JJ GEEWAX



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2018 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- ⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

The photographs in this book are reproduced under a Creative Commons license.

 Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Christina Taylor
Revieweditor: Aleks Dragosavljevic
Technical development editor: Francesco Bianchi
Project manager: Kevin Sullivan
Copy editors: Pamela Hunt and Carl Quesnel
Proofreaders: Melody Dolab and Alyson Brener
Technical proofreader: Romin Irani
Typesetter: Dennis Dalinnik
Illustrator: Jason Alexander
Cover designer: Marija Tudor

ISBN: 9781617293528

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – DP – 23 22 21 20 19 18

brief contents

PART 1	GETTING STARTED	1
1	■ What is “cloud”? 3	
2	■ Trying it out: deploying WordPress on Google Cloud 24	
3	■ The cloud data center 38	
PART 2	STORAGE	51
4	■ Cloud SQL: managed relational storage 53	
5	■ Cloud Datastore: document storage 89	
6	■ Cloud Spanner: large-scale SQL 117	
7	■ Cloud Bigtable: large-scale structured data 158	
8	■ Cloud Storage: object storage 199	
PART 3	COMPUTING	241
9	■ Compute Engine: virtual machines 243	
10	■ Kubernetes Engine: managed Kubernetes clusters 306	
11	■ App Engine: fully managed applications 337	
12	■ Cloud Functions: serverless applications 385	
13	■ Cloud DNS: managed DNS hosting 406	

PART 4 MACHINE LEARNING425

- 14 ■ Cloud Vision: image recognition 427**
- 15 ■ Cloud Natural Language: text analysis 446**
- 16 ■ Cloud Speech: audio-to-text conversion 463**
- 17 ■ Cloud Translation: multilanguage machine translation 473**
- 18 ■ Cloud Machine Learning Engine: managed machine learning 485**

PART 5 DATA PROCESSING AND ANALYTICS519

- 19 ■ BigQuery: highly scalable data warehouse 521**
- 20 ■ Cloud Dataflow: large-scale data processing 547**
- 21 ■ Cloud Pub/Sub: managed event publishing 568**

contents

*foreword xvii
preface xix
acknowledgments xxi
about this book xxiii
about the cover illustration xxvii*

PART 1 GETTING STARTED1

I	What is “cloud”? 3
1.1	What is Google Cloud Platform? 4
1.2	Why cloud? 4 <i>Why not cloud?</i> 5
1.3	What to expect from cloud services 6 <i>Computing</i> 6 ▀ <i>Storage</i> 7 ▀ <i>Analytics (aka, Big Data)</i> 8 <i>Networking</i> 8 ▀ <i>Pricing</i> 9
1.4	Building an application for the cloud 9 <i>What is a cloud application?</i> 9 ▀ <i>Example: serving photos</i> 10 <i>Example projects</i> 12
1.5	Getting started with Google Cloud Platform 13 <i>Signing up for GCP</i> 13 ▀ <i>Exploring the console</i> 14 <i>Understanding projects</i> 15 ▀ <i>Installing the SDK</i> 16

1.6	Interacting with GCP	18
	<i>In the browser: the Cloud Console</i>	18
	<i>On the command line: gcloud</i>	20
	<i>In your own code: google-cloud-*</i>	22

2 Trying it out: deploying WordPress on Google Cloud 24

2.1	System layout overview	25
2.2	Digging into the database	26
	<i>Turning on a Cloud SQL instance</i>	27
	<i>Securing your Cloud SQL instance</i>	28
	<i>Connecting to your Cloud SQL instance</i>	30
	<i>Configuring your Cloud SQL instance for WordPress</i>	30
2.3	Deploying the WordPress VM	31
2.4	Configuring WordPress	33
2.5	Reviewing the system	36
2.6	Turning it off	37

3 The cloud data center 38

3.1	Data center locations	39
3.2	Isolation levels and fault tolerance	42
	<i>Zones</i>	42
	<i>Regions</i>	42
	<i>Designing for fault tolerance</i>	43
	<i>Automatic high availability</i>	45
3.3	Safety concerns	45
	<i>Security</i>	46
	<i>Privacy</i>	47
	<i>Special cases</i>	48
3.4	Resource isolation and performance	48

PART 2 STORAGE 51

4 Cloud SQL: managed relational storage 53

4.1	What's Cloud SQL?	54
4.2	Interacting with Cloud SQL	54
4.3	Configuring Cloud SQL for production	60
	<i>Access control</i>	60
	<i>Connecting over SSL</i>	61
	<i>Maintenance windows</i>	66
	<i>Extra MySQL options</i>	67
4.4	Scaling up (and down)	68
	<i>Computing power</i>	69
	<i>Storage</i>	69
4.5	Replication	71
	<i>Replica-specific operations</i>	75

4.6	Backup and restore	75
	<i>Automated daily backups</i>	76 ▪ <i>Manual data export to Cloud Storage</i> 77
4.7	Understanding pricing	81
4.8	When should I use Cloud SQL?	83
	<i>Structure</i> 83 ▪ <i>Query complexity</i> 84 ▪ <i>Durability</i> 84	
	<i>Speed (latency)</i> 84 ▪ <i>Throughput</i> 84	
4.9	Cost	85
	<i>Overall</i> 85	
4.10	Weighing Cloud SQL against a VM running MySQL	87

5 *Cloud Datastore: document storage* 89

5.1	What's Cloud Datastore?	90
	<i>Design goals for Cloud Datastore</i> 91 ▪ <i>Concepts</i> 92	
	<i>Consistency and replication</i> 96 ▪ <i>Consistency with data locality</i> 99	
5.2	Interacting with Cloud Datastore	101
5.3	Backup and restore	107
5.4	Understanding pricing	110
	<i>Storage costs</i> 110 ▪ <i>Per-operation costs</i> 110	
5.5	When should I use Cloud Datastore?	111
	<i>Structure</i> 111 ▪ <i>Query complexity</i> 112 ▪ <i>Durability</i> 112	
	<i>Speed (latency)</i> 112 ▪ <i>Throughput</i> 113 ▪ <i>Cost</i> 113	
	<i>Overall</i> 113 ▪ <i>Other document storage systems</i> 115	

6 *Cloud Spanner: large-scale SQL* 117

6.1	What is NewSQL?	118
6.2	What is Spanner?	118
6.3	Concepts	118
	<i>Instances</i> 119 ▪ <i>Nodes</i> 120 ▪ <i>Databases</i> 120 ▪ <i>Tables</i> 120	
6.4	Interacting with Cloud Spanner	121
	<i>Creating an instance and database</i> 122 ▪ <i>Creating a table</i> 125	
	<i>Adding data</i> 127 ▪ <i>Querying data</i> 127 ▪ <i>Altering database schema</i> 131	
6.5	Advanced concepts	132
	<i>Interleaved tables</i> 133 ▪ <i>Primary keys</i> 136 ▪ <i>Split points</i> 137	
	<i>Choosing primary keys</i> 138 ▪ <i>Secondary indexes</i> 139	
	<i>Transactions</i> 145	

- 6.6 Understanding pricing 152
- 6.7 When should I use Cloud Spanner? 153
 - Structure* 154 ▪ *Query complexity* 154 ▪ *Durability* 154
 - Speed (latency)* 154 ▪ *Throughput* 154 ▪ *Cost* 155
 - Overall* 155

7 *Cloud Bigtable: large-scale structured data* 158

- 7.1 What is Bigtable? 159
 - Design goals* 159 ▪ *Design nongoals* 161
 - Design overview* 162
- 7.2 Concepts 162
 - Data model concepts* 163 ▪ *Infrastructure concepts* 168
- 7.3 Interacting with Cloud Bigtable 173
 - Creating a Bigtable Instance* 173 ▪ *Creating your schema* 175
 - Managing your data* 177 ▪ *Importing and exporting data* 181
- 7.4 Understanding pricing 184
- 7.5 When should I use Cloud Bigtable? 185
 - Structure* 185 ▪ *Query complexity* 186 ▪ *Durability* 186
 - Speed (latency)* 186 ▪ *Throughput* 186 ▪ *Cost* 187
 - Overall* 187
- 7.6 What's the difference between Bigtable and HBase? 190
- 7.7 Case study: InstaSnap recommendations 191
 - Querying needs* 191 ▪ *Tables* 192 ▪ *Users table* 192
 - Recommendations table* 195 ▪ *Processing data* 196
- 7.8 Summary 198

8 *Cloud Storage: object storage* 199

- 8.1 Concepts 200
 - Buckets and objects* 200
- 8.2 Storing data in Cloud Storage 201
- 8.3 Choosing the right storage class 204
 - Multiregional storage* 204 ▪ *Regional storage* 205
 - Nearline storage* 205 ▪ *Coldline storage* 206
- 8.4 Access control 207
 - Limiting access with ACLs* 207 ▪ *Signed URLs* 213
 - Logging access to your data* 217
- 8.5 Object versions 219

- 8.6 Object lifecycles 223
- 8.7 Change notifications 225
 - URL restrictions* 227
- 8.8 Common use cases 228
 - Hosting user content* 228 ▪ *Data archival* 229
- 8.9 Understanding pricing 230
 - Amount of data stored* 231 ▪ *Amount of data transferred* 232
 - Number of operations executed* 233 ▪ *Nearline and Coldline pricing* 234
- 8.10 When should I use Cloud Storage? 236
 - Structure* 236 ▪ *Query complexity* 236 ▪ *Durability* 236
 - Speed (latency)* 237 ▪ *Throughput* 237 ▪ *Overall* 237
 - To-do list* 237 ▪ *E*Exchange* 238 ▪ *InstaSnap* 238

PART 3 COMPUTING.....241

9

Compute Engine: virtual machines 243

- 9.1 Launching your first (or second) VM 244
- 9.2 Block storage with Persistent Disks 245
 - Disks as resources* 246 ▪ *Attaching and detaching disks* 247
 - Using your disks* 250 ▪ *Resizing disks* 252 ▪ *Snapshots* 253
 - Images* 258 ▪ *Performance* 259 ▪ *Encryption* 261
- 9.3 Instance groups and dynamic resources 264
 - Changing the size of an instance group* 269 ▪ *Rolling updates* 270 ▪ *Autoscaling* 274
- 9.4 Ephemeral computing with preemptible VMs 276
 - Why use preemptible machines?* 277 ▪ *Turning on preemptible VMs* 278 ▪ *Handling terminations* 278 ▪ *Preemption selection* 279
- 9.5 Load balancing 280
 - Backend configuration* 282 ▪ *Host and path rules* 285
 - Frontend configuration* 286 ▪ *Reviewing the configuration* 287
- 9.6 Cloud CDN 289
 - Enabling Cloud CDN* 290 ▪ *Cache control* 293
- 9.7 Understanding pricing 294
 - Computing capacity* 294 ▪ *Sustained use discounts* 295
 - Preemptible prices* 298 ▪ *Storage* 298 ▪ *Network traffic* 299

9.8 When should I use GCE?	301
Flexibility	301 ▪ Complexity
Cost	302 ▪ Overall
E*Exchange	303 ▪ InstaSnap
Performance	302 ▪ To-Do List
InstaSnap	304

10 Kubernetes Engine: managed Kubernetes clusters 306

10.1 What are containers?	307
Configuration	307 ▪ Standardization
10.2 What is Docker?	310
10.3 What is Kubernetes?	310
Clusters	312 ▪ Nodes
Services	313 ▪ Pods
10.4 What is Kubernetes Engine?	315
10.5 Interacting with Kubernetes Engine	315
Defining your application	315 ▪ Running your container
locally	317 ▪ Deploying to your container registry
Setting up your Kubernetes Engine cluster	320 ▪ Deploying
your application	321 ▪ Replicating your application
Using the Kubernetes UI	325
10.6 Maintaining your cluster	327
Upgrading the Kubernetes master node	327 ▪ Upgrading
cluster nodes	329 ▪ Resizing your cluster
10.7 Understanding pricing	332
10.8 When should I use Kubernetes Engine?	332
Flexibility	332 ▪ Complexity
Cost	333 ▪ Performance
E*Exchange	334 ▪ Overall
InstaSnap	334 ▪ To-Do List
Scaling	335 ▪ Choosing instance configurations

11 App Engine: fully managed applications 337

11.1 Concepts	338
Applications	339 ▪ Services
Instances	341 ▪ Versions
11.2 Interacting with App Engine	343
Building an application in App Engine Standard	344
On App Engine Flex	353
11.3 Scaling your application	361
Scaling on App Engine Standard	362 ▪ Scaling on App
Engine Flex	367 ▪ Choosing instance configurations
InstaSnap	368

11.4	Using App Engine Standard's managed services	371
	<i>Storing data with Cloud Datastore</i>	371
	<i>Caching ephemeral data</i>	372
	<i>Deferring tasks</i>	374
	<i>Splitting traffic</i>	375
11.5	Understanding pricing	379
11.6	When should I use App Engine?	380
	<i>Flexibility</i>	380
	<i>Complexity</i>	381
	<i>Performance</i>	381
	<i>Cost</i>	381
	<i>Overall</i>	382
	<i>To-Do List</i>	382
	<i>E*Exchange</i>	382
	<i>InstaSnap</i>	383

12 *Cloud Functions: serverless applications* 385

12.1	What are microservices?	385
12.2	What is Google Cloud Functions?	386
	<i>Concepts</i>	388
12.3	Interacting with Cloud Functions	391
	<i>Creating a function</i>	391
	<i>Deploying a function</i>	392
	<i>Triggering a function</i>	394
12.4	Advanced concepts	395
	<i>Updating functions</i>	395
	<i>Deleting functions</i>	396
	<i>Using dependencies</i>	396
	<i>Calling other Cloud APIs</i>	399
	<i>Using a Google Source Repository</i>	401
12.5	Understanding pricing	403

13 *Cloud DNS: managed DNS hosting* 406

13.1	What is Cloud DNS?	407
	<i>Example DNS entries</i>	409
13.2	Interacting with Cloud DNS	410
	<i>Using the Cloud Console</i>	410
	<i>Using the Node.js client</i>	414
13.3	Understanding pricing	418
	<i>Personal DNS hosting</i>	418
	<i>Startup business DNS hosting</i>	418
13.4	Case study: giving machines DNS names at boot	419

PART 4 MACHINE LEARNING 425

14 *Cloud Vision: image recognition* 427

14.1	Annotating images	428
	<i>Label annotations</i>	429
	<i>Faces</i>	432
	<i>Text recognition</i>	435
	<i>Logo recognition</i>	437
	<i>Safe-for-work detection</i>	440
	<i>Combining multiple detection types</i>	441

- 14.2 Understanding pricing 443
- 14.3 Case study: enforcing valid profile photos 443

15 *Cloud Natural Language: text analysis 446*

- 15.1 How does the Natural Language API work? 447
- 15.2 Sentiment analysis 448
- 15.3 Entity recognition 452
- 15.4 Syntax analysis 455
- 15.5 Understanding pricing 457
- 15.6 Case study: suggesting InstaSnap hash-tags 459

16 *Cloud Speech: audio-to-text conversion 463*

- 16.1 Simple speech recognition 465
- 16.2 Continuous speech recognition 467
- 16.3 Hinting with custom words and phrases 468
- 16.4 Understanding pricing 469
- 16.5 Case study: InstaSnap video captions 469

17 *Cloud Translation: multilanguage machine translation 473*

- 17.1 How does the Translation API work? 475
- 17.2 Language detection 477
- 17.3 Text translation 479
- 17.4 Understanding pricing 481
- 17.5 Case study: translating InstaSnap captions 481

18 *Cloud Machine Learning Engine: managed machine learning 485*

- 18.1 What is machine learning? 485
 - What are neural networks? 486 ▪ What is TensorFlow? 488*
- 18.2 What is Cloud Machine Learning Engine? 491
 - Concepts 492 ▪ Putting it all together 495*
- 18.3 Interacting with Cloud ML Engine 498
 - Overview of US Census data 498 ▪ Creating a model 499*
 - Setting up Cloud Storage 501 ▪ Training your model 503*
 - Making predictions 506 ▪ Configuring your underlying resources 509*

- 18.4 Understanding pricing 514
 Training costs 514 ▪ *Prediction costs* 516

PART 5 DATA PROCESSING AND ANALYTICS.....519

19 *BigQuery: highly scalable data warehouse* 521

- 19.1 What is BigQuery? 521
 Why BigQuery? 522 ▪ *How does BigQuery work?* 522
 Concepts 525
- 19.2 Interacting with BigQuery 528
 Querying data 528 ▪ *Loading data* 533
 Exporting datasets 542
- 19.3 Understanding pricing 544
 Storage pricing 544 ▪ *Data manipulation pricing* 545
 Query pricing 545

20 *Cloud Dataflow: large-scale data processing* 547

- 20.1 What is Apache Beam? 549
 Concepts 550 ▪ *Putting it all together* 555
- 20.2 What is Cloud Dataflow? 556
- 20.3 Interacting with Cloud Dataflow 557
 Setting up 557 ▪ *Creating a pipeline* 559 ▪ *Executing a pipeline locally* 560 ▪ *Executing a pipeline using Cloud Dataflow* 561
- 20.4 Understanding pricing 565

21 *Cloud Pub/Sub: managed event publishing* 568

- 21.1 The headache of messaging 569
- 21.2 What is Cloud Pub/Sub? 569
- 21.3 Life of a message 569
- 21.4 Concepts 572
 Topics 572 ▪ *Messages* 572 ▪ *Subscriptions* 574
 Sample configuration 575
- 21.5 Trying it out 576
 Sending your first message 576 ▪ *Receiving your first message* 578
- 21.6 Push subscriptions 581

- 21.7 Understanding pricing 583
21.8 Messaging patterns 584
Fan-out broadcast messaging 584 ▪ *Work-queue messaging* 587
index 589

foreword

In the early days of Google, we were a victim of our own success. People loved our search results, but handling more search traffic meant we needed more servers, which at that time meant physical servers, not virtual ones. Traffic was growing by something like 10% every week, so every few days we would hit a new record, and we had to ensure we had enough capacity to handle it all. We also had to do it all from scratch.

When it comes to our infrastructural challenges, we've largely succeeded. We've built a system of data centers and networks that rival most of the world, but until recently, that infrastructure has been exclusively for us. Google Cloud Platform represents the natural extension of our infrastructural achievements over the past 15 years or so by allowing everyone to benefit from the efficiency of Google's data centers and the years of experience we have running them.

All of this manifests as a collection of products and services that solve hard technical problems (think data consistency) so that you don't have to, but it also means that instead of solving the hard technical problem, you have to learn how to use the service. And while tinkering with new services is part of daily life at Google, most of the world expects things to "just work" so they can get on with their business. For many, a misconfigured server or inconsistent database is not a fun puzzle to solve—it's a distraction.

Google Cloud Platform in Action acts as a guide to minimize those distractions, demonstrating how to use GCP in practice while also explaining how things work under the hood. In this book, JJ focuses on the most important aspects of GCP (like Compute Engine) but also highlights some of the more recent additions to GCP (like Kubernetes

Engine and the various machine-learning APIs), offering a well-rounded collection of all that GCP has to offer.

Looking back, Google Cloud Platform has grown immensely. From App Engine in 2008, to Compute Engine in 2012, to several machine-learning APIs in 2017, keeping up can be difficult. But with this book in hand, you're well equipped to build what's next.

URS HÖLZLE
SVP, Technical Infrastructure
Google

****preface****

I was lucky enough to fall in love with building software all the way back in 1997. This started with toy projects in Visual Basic (yikes) or HTML (yes, the `<blink>` and `marquee` tags appeared from time to time), and eventually moved on to “real work” using “more mature languages” like C#, Java, and Python. Throughout that time the infrastructure hosting these projects followed a similar evolution, starting with free static hosting and moving on to the “grown-up” hosting options like virtual private servers or dedicated hosts in a colocation facility. This certainly got the job done, but scaling up and down was frustrating (you had to place an order and wait a little bit), and the minimum purchase was usually a full calendar year.

But then things started to change. Somewhere around 2008, cloud computing became available using Amazon’s new Elastic Compute Cloud (EC2). Suddenly you had way more control over your infrastructure than ever before thanks to the ability to turn computers on and off using web-based APIs. To make things even better, you paid only for the time when the computer was actually running rather than for the entire year. It really was amazing.

As we now know, the rest is history. Cloud computing expanded into generalized cloud infrastructure, moving higher and higher up the stack, to provide more and more value as time went on. More companies got involved, launching entire divisions devoted to cloud services, bringing with them even more new and exciting products to add to our toolbox. These products went far beyond leasing virtual servers by the hour, but the principle involved was always the same: take a software or infrastructure problem, remove the manual work, and then charge only for what’s used. It just so

happens that Google was one of those companies, applying this principle to its in-house technology to build Google Cloud Platform.

Fast-forward to today, and it seems we have a different problem: our toolboxes are overflowing. Cloud infrastructure is amazing, but only if you know how to use it effectively. You need to understand what's in your toolbox, and, unfortunately, there aren't a lot of guidebooks out there. If Google Cloud Platform is your toolbox, *Google Cloud Platform in Action* is here to help you understand all of your tools, from high-level concepts (like choosing the right storage system) to the low-level details (like understanding how much that storage will cost).

acknowledgments

As with any large project, this book is the result of contributions from many different people. First and foremost, I must thank Dave Nagle who convinced me to join the Google Cloud Platform team in the first place and encouraged me to go where needed—even if it was uncomfortable.

Additionally, many people provided similar support, encouragement, and technical feedback, including Kristen Ranieri, Marc Jacobs, Stu Feldman, Ari Balogh, Max Ross, Urs Hölzle, Andrew Fikes, Larry Greenfield, Alfred Fuller, Hong Zhang, Ray Colline, JM Leon, Joerg Heilig, Walt Drummond, Peter Weinberger, Amnon Horowitz, Rich Sanzi, James Tamplin, Andrew Lee, Mike McDonald, Jony Dimond, Tom Larkworthy, Doron Meyer, Mike Dahlin, Sean Quinlan, Sanjay Ghemawatt, Eric Brewer, Dominic Preuss, Dan McGrath, Tommy Kershaw, Sheryn Chan, Luciano Cheng, Jeremy Sugerman, Steve Schirripa, Mike Schwartz, Jason Woodard, Grace Benz, Chen Goldberg, and Eyal Manor.

Further, it should come as no surprise that a project of this size involved technical contributions from a diverse set of people at Google, including Tony Tseng, Brett Hesterberg, Patrick Costello, Chris Taylor, Tom Ayles, Vikas Kedia, Deepti Srivastava, Damian Reeves, Misha Brukman, Carter Page, Phaneendar Vemuru, Greg Morris, Doug McErlean, Carlos O’Ryan, Andrew Hurst, Nathan Herring, Brandon Yarbrough, Travis Hobrla, Bob Day, Kir Titovsky, Oren Teich, Steren Gianni, Jim Caputo, Dan McClary, Bin Yu, Milo Martin, Gopal Ashok, Sam McVeety, Nikhil Kothari, Apoorv Saxena, Ram Ramanathan, Dan Aharon, Phil Bogle, Kirill Tropin, Sandeep Singhal, Dipti Sangani, Mona Attarian, Jen Lin, Navneet Joneja, TJ Goltermann, Sam Greenfield,

Dan O'Meara, Jason Polites, Rajeev Dayal, Mark Pellegrini, Rae Wang, Christian Kemper, Omar Ayoub, Jonathan Amsterdam, Jon Skeet, Stephen Sawchuk, Dave Gramlich, Mike Moore, Chris Smith, Marco Ziccardi, Dave Supplee, John Pedrie, Jonathan Amsterdam, Danny Hermes, Tres Seaver, Anthony Moore, Garrett Jones, Brian Watson, Rob Clevenger, Michael Rubin, and Brian Grant, along with many others. Many thanks go out to everyone who corrected errors and provided feedback, whether in person, on the MEAP forum, or via email.

This project simply wouldn't have been possible with the various teams at Manning who guided me through the process and helped shape this book into what it is now. I'm particularly grateful to Mike Stephens for convincing me to do this in the first place, Christina Taylor for her tireless efforts to shape the content into great teaching material, and Marjan Bace for pushing to tighten the content so that we didn't end with a 1,000-page book.

Finally, I'd like to thank Al Scherer and Romin Irini, for giving the manuscript a thorough technical review and proofread, and all the reviewers who provided feedback along the way, including Ajay Godbole, Alfred Thompson, Arun Kumar, Aurélien Marocco, Conor Redmond, Emanuele Origgi, Enric Cecilla, Grzegorz Bernas, Ian Stirk, Javier Collado Cabeza, John Hyaduck, John R. Donoghue, Joyce Echessa, Maksym Shcheglov, Mario-Leander Reimer, Max Hemingway, Michael Jensen, Michał Ambroziewicz, Peter J. Krey, Rambabu Posa, Renato Alves Felix, Richard J. Tobias, Sopan Shewale, Steve Atchue, Todd Ricker, Vincent Joseph, Wendell Beckwith, and Xinyu Wang.

about this book

Google Cloud Platform in Action was written to provide a practical guide for using all of the various cloud products and APIs available from Google. It begins by explaining some of the fundamental concepts needed to understand how cloud works and proceeds from there to build on these concepts one product at a time, digging into the details of how different products work and providing realistic examples of how they can be used.

Who should read this book

Google Cloud Platform in Action is for anyone who builds software products or deals with hosting them. Familiarity with the cloud is not necessary, but familiarity with the basics in the software development toolbox (such as SQL databases, APIs, and command-line tools) is important. If you've heard of the cloud and want to know how best to use it, this book is probably for you.

How this book is organized: a roadmap

This book is broken into five sections, each covering a different aspect of Google Cloud Platform. Part 1 explains what Google Cloud Platform is and some of the fundamental pieces of the platform itself, with the goal of building a good foundation before digging into specific cloud products.

- Chapter 1 gives an overview of the cloud and what Google Cloud Platform is. It also discusses the different things you might expect to get out of GCP and walks you through signing up, getting started, and interacting with Google Cloud Platform.

- Chapter 2 dives right into the details of getting a real GCP project running. This covers setting up a computing environment and database storage to turn on a WordPress instance using Google Cloud Platform's free tier.
- Chapter 3 explores some details about data centers and explains the core differences when moving into the cloud.

Part 2 covers all of the storage-focused products available on Google Cloud Platform. Because so many different options for storing data exist, one goal of this section is to provide a framework for evaluating all of the options. To do this, each chapter looks at several different attributes for each of the storage options, summarized in Table 1.

Table 1 Summary of storage system attributes

Aspect	Example question
Structure	How normalized and formatted is the data being stored?
Query complexity	How complicated are the questions you ask about the data?
Speed	How quickly do you need a response to any given request?
Throughput	How many queries need to be handled concurrently?
Price	How much will all of this cost?

- Chapter 4 looks at how you can minimize the management overhead when running MySQL to store relational data.
- Chapter 5 explores document-oriented storage, similar to systems like MongoDB, using Cloud Datastore.
- Chapter 6 dives into the world of NewSQL for managing large-scale relational data using Cloud Spanner to provide strong consistency with global replication.
- Chapter 7 discusses storing and querying large-scale key-value data using Cloud Bigtable, which was originally designed to handle Google's search index.
- Chapter 8 finishes up the section on storage by introducing Cloud Storage for keeping track of arbitrary chunks of bytes with high availability, high durability, and low latency content distribution.

Part 3 looks at all the various ways to run your own code in the cloud using cloud computing resources. Similar to the storage section, many options exist, which can often lead to confusion. As a result, this section has a similar goal of setting up a framework for evaluating the various computing services. Each chapter looks at a few different aspects of each service, explained in table 2. As an extra, this section also contains a chapter on Cloud DNS, which is commonly used to give human-friendly names to all the computing resources that you'll create in your projects.

Table 2 Summary of computing system attributes

Aspect	Example question
Flexibility	How restricted am I when building using this computing platform?
Complexity	How complicated is it to fully understand the system?
Performance	How well does the system perform compared to dedicated hardware?
Price	How much will all of this cost?

- Chapter 9 looks in depth at the fundamental way of running computing resources in the cloud using Compute Engine.
- Chapter 10 moves one level up the stack of abstraction, exploring containers and how to run them in the cloud using Kubernetes and Kubernetes Engine.
- Chapter 11 moves one level further still, exploring the hosted application environment of Google App Engine.
- Chapter 12 dives into the world of service-oriented applications with Cloud Functions.
- Chapter 13 looks at Cloud DNS, which can be used to write code to interact with the internet's distributed naming system, giving friendly names to your VMs or other computing resources.

Part 4 switches gears away from raw infrastructure and focuses exclusively on the rapidly evolving world of machine learning and artificial intelligence.

- Chapter 14 focuses on how to bring artificial intelligence to the visual world using the Cloud Vision API.
- Chapter 15 explains how the Cloud Natural Language API can be used to enrich written documents with annotations along with detecting the overall sentiment.
- Chapter 16 explores turning audio streams into text using machine speech recognition.
- Chapter 17 looks at translating text between multiple languages using neural machine translation for much greater accuracy than other methods.
- Chapter 18, intended to be read along with other works on TensorFlow, generalizes the heavy lifting of machine learning using Google Cloud Platform infrastructure under the hood.

Part 5 wraps up by looking at large-scale data processing and analytics, and how Google Cloud Platform's infrastructure can be used to get more performance at a lower total cost.

- Chapter 19 explores large-scale data analytics using Google's BigQuery, showing how you can scan over terabytes of data in a matter of seconds.

- Chapter 20 dives into more advanced large-scale data processing using Apache Beam and Google Cloud Dataflow.
- Chapter 21 explains how to handle large-scale distributed messaging with Google Cloud Pub/Sub.

About the code

This book contains many examples of source code, both in numbered listings and inline with normal text. In both cases, source code is formatted in a fixed-width font like this to separate it from ordinary text. Sometimes **boldface** is used to highlight code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this was not enough, and listings include line-continuation markers (➡). Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

Book forum

Purchase of Google Cloud Platform in Action includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to <https://forums.manning.com/forums/google-cloud-platform-in-action>. You can also learn more about Manning's forums and the rules of conduct at <https://forums.manning.com/forums/about>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the author

JJ Geewax received his Bachelor of Science in Engineering in Computer Science from the University of Pennsylvania in 2008. While an undergrad at UPenn he joined Invite Media, a platform that enables customers to buy online ads in real time. In 2010 Invite Media was acquired by Google and, as their largest internal cloud customer, became the first large user of Google Cloud Platform. Since then, JJ has worked as a Senior Staff Software Engineer at Google, currently specializing in API design, specifically for Google Cloud Platform.

about the cover illustration

The figure on the cover of Google Cloud Platform in Action is captioned, “Barbaresque Enveloppe Iana son Manteaul.” The illustration is taken from a collection of dress costumes from various countries by Jacques Grasset de Saint-Sauveur (1757–1810), titled *Costumes de différents pays*, published in France in 1797. Each illustration is finely drawn and colored by hand. The rich variety of Grasset de Saint-Sauveur’s collection reminds us vividly of how culturally apart the world’s towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by their dress.

The way we dress has changed since then, and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns, regions, or countries. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Grasset de Saint-Sauveur’s pictures.

Part 1

Getting started

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

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

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

What is “cloud”?



This chapter covers

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

The term “cloud” has been used in many different contexts and it has many different definitions, so it makes sense to define the term—at least for this book.

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

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

In many ways, cloud is the next layer of abstraction in computer infrastructure, where computing, storage, analytics, networking, and more are all pushed higher

up the computing stack. This structure takes the focus of the developer away from CPUs and RAM and toward APIs for higher-level operations such as storing or querying for data. Cloud services aim to solve your problem, not give you low-level tools for you to do so on your own. Further, cloud services are extremely flexible, with most requiring no provisioning or long-term contracts. Due to this, relying on these services allows you to scale up and down with no advanced notice or provisioning, while paying only for the resources you use in a given month.

1.1 What is Google Cloud Platform?

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

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

Although Google Cloud Platform is similar to other cloud providers, it has some differences that are worth mentioning. First, Google is “home” to some amazing people, who have created some incredible new technologies there and then shared them with the world through research papers. These include MapReduce (the research paper that spawned Hadoop and changed how we handle “Big Data”), Bigtable (the paper that spawned Apache HBase), and Spanner. With Google Cloud Platform, many of these technologies are no longer “only for Googlers.”

Second, Google operates at such a scale that it has many economic advantages, which are passed on in the form of lower prices. Google owns immense physical infrastructure, which means it buys and builds custom hardware to support it, which means cheaper overall prices, often combined with improved performance. It’s sort of like Costco letting you open up that 144-pack of potato chips and pay 1/144th the price for one bag.

1.2 Why cloud?

So why use cloud in the first place? First, cloud hosting offers a lot of flexibility, which is a great fit for situations where you don’t know (or can’t know) how much computing power you need. You won’t have to overprovision to handle situations where you might need a lot of computing power in the morning and almost none overnight.

Second, cloud hosting comes with the maintenance built in for several products. This means that cloud hosting results in minimal extra work to host your systems compared to other options where you might need to manage your own databases, operating

systems, and even your own hardware (in the case of a colocated hosting provider). If you don't want to (or can't) manage these types of things, cloud hosting is a great choice.

1.2.1 Why not cloud?

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

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

Figure 1.1 shows a visual representation of a usage and cost pattern that would be a bad fit for cloud hosting. Notice how the growth of computing needs (the bottom line) steadily increases, and the company is provisioning extra capacity regularly to stay ahead of its needs (the top, wavy line).

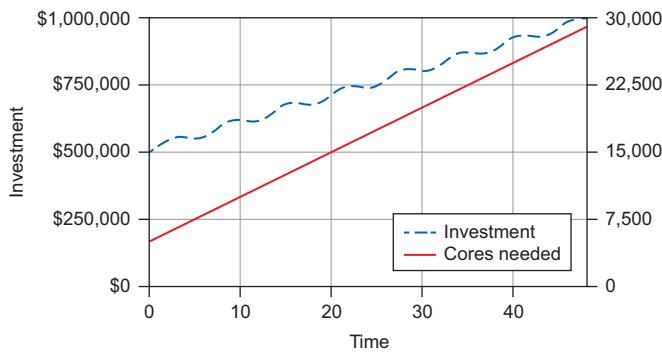


Figure 1.1 Steady growth in resource consumption

Compare this with figure 1.2, which shows a more typical company of the internet age, where growth is spiky and unpredictable and tends to drop without much notice. In this case, the company bought enough computing capacity (the top line) to handle a spike, which was needed up front, but then when traffic fell (the bottom line), it was stuck with quite a bit of excess capacity.

In short, if you have the expertise to run your own data centers (including the plans for disasters and other failures, and the recovery from those potential disasters), along with steady growing computing needs (measured in cores, storage, networking

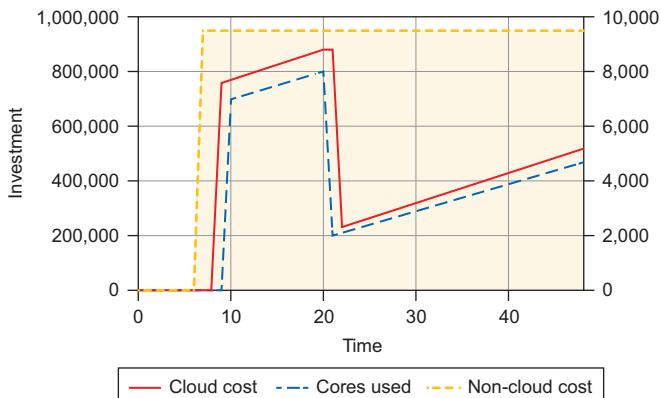


Figure 1.2 Unexpected pattern of resource consumption

consumption, and so on), cloud hosting might not be right for you. If you’re anything like the typical company of today, where you don’t know what you need today (and certainly don’t know what you’ll need several years from today), and don’t have the expertise in your company to build out huge data centers to achieve the same economies of scale that large cloud providers can offer, cloud hosting is likely to be a good fit for you.

1.3 **What to expect from cloud services**

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

1.3.1 **Computing**

You’ve already learned a little bit about how cloud computing is fundamentally different from virtual private, colocated, or on-premises hosting. Let’s take a look at what you can expect if you decide to take the plunge into the world of cloud computing.

The first thing you’ll notice is that provisioning your machine will be fast. Compared to colocated or on-premises hosting, it should be significantly faster. In real terms, the typical expected time from clicking the button to connecting via secure shell to the machine will be about a minute. If you’re used to virtual private hosting, the provisioning time might be around the same, maybe slightly faster.

What’s more interesting is what is missing in the process of turning on a cloud-hosted virtual machine (VM). If you turn on a VM right now, you might notice that there’s no mention of payment. Compare that to your typical virtual private server (VPS), where you agree on a set price and purchase the VPS for a full year, making monthly payments (with your first payment immediately, and maybe a discount for up-front payment). Google doesn’t mention payment at this time for a simple reason:

they don't know how long you'll keep that machine running, so there's no way to know how much to charge you. It can determine how much you owe only either at the end of the month or when you turn off the VM. See table 1.1 for a comparison.

Table 1.1 Hosting choice comparison

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

1.3.2 Storage

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

The answers to these questions change what you build in significant ways, so much so that you might end up building entirely different products if you were the one building a storage service. Ultimately, the abstraction provided by a storage service gives you the ability to configure your storage mechanisms for various levels of performance, durability, availability, and cost.

But these systems come with a few trade-offs. First, the failure aspects of storing data typically disappear. You shouldn't ever get a notification or a phone call from someone saying that a hard drive failed and your data was lost. Next, with reduced-availability options, you might occasionally try to download your data and get an error telling you to try again later, but you'll be paying much less for storage of that class than any other. Finally, for virtual disks in the cloud, you'll notice that you have lots of choices about how you can store your data, both in capacity (measured in GB) and in performance (typically measured in input/output operations per second [IOPS]). Once again, like computing in the cloud, storing data on virtual disks in the cloud feels familiar.

On the other hand, some of the custom database services, like Cloud Datastore, might feel a bit foreign. These systems are in many ways completely unique to cloud hosting, relying on huge, shared, highly scalable systems built by and for Google. For

example, Cloud Datastore is an adapted externalization of an internal storage system called Megastore, which was, until recently, the underlying storage system for many Google products, including Gmail. These hosted storage systems sometimes required you to integrate your own code with a proprietary API. This means that it'll become all the more important to keep a proper layer of abstraction between your code base and the storage layer. It still may make sense to rely on these hosted systems, particularly because all of the scaling is handled automatically.

1.3.3 **Analytics (aka, Big Data)**

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

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

1.3.4 **Networking**

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

In the world of cloud computing some of these assumptions remain unchanged. The interesting parts come up when you start developing the need for more advanced features, such as faster-than-normal network connections, advanced firewalling abilities (where you only allow certain IPs to talk to certain ports), load balancing (where requests come in and can be handled by any one of many machines), and SSL certificate management (where you want requests to be encrypted but don't want to manage the certificates for each individual virtual machine).

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

1.3.5 Pricing

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

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

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

1.4 Building an application for the cloud

So far this chapter has been mainly a discussion on what cloud is and what it means for developers looking to rely on it rather than traditional hosting options. Let's switch gears now and demonstrate how to deploy something meaningful using Google Cloud Platform.

1.4.1 What is a cloud application?

In many ways, an application built for the cloud is like any other. The primary difference is in the assumptions made about the application's architecture. For example, in a traditional application, we tend to deploy things such as binaries running on particular servers (for example, running a MySQL database on one server and Apache with mod_php on another). Rather than thinking in terms of which servers handle which things, a typical cloud application relies on hosted or managed services whenever possible. In many cases it relies on containers the way a traditional application would rely on servers. By operating this way, a cloud application is often much more flexible and able to grow and shrink, depending on the customer demand throughout the day.

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

1.4.2 Example: serving photos

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

If that's a bit too technical, try thinking about it from an architectural standpoint. The old way of doing this was to store the image data in your relational database, and then whenever someone wanted to see the profile photo, you'd retrieve it from the database and return it through your web server, as shown in figure 1.3.

In case it wasn't clear, this is bad for a variety of reasons. First, storing binary data in your database is inefficient. It does work for transactional support, which profile photos probably don't need. Second, and most important, by storing the binary data of a photo in your database, you're putting extra load on the database itself, but not using it for the things it's good at, like joining relational data together.

In short, if you don't need transactional semantics on your photo (which here, we don't), it makes more sense to put the photo somewhere on a disk and then use the static serving capabilities of your web server to deliver those bytes, as shown in figure 1.4. This leaves the database out completely, so it's free to do more important work.

This structure is a huge improvement and probably performs quite well for most use cases, but it doesn't illustrate anything special about the cloud. Let's take it a step further and consider geography for a moment. In your current deployment, you have

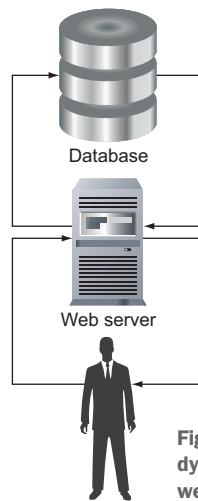


Figure 1.3 Serving photos dynamically through your web server

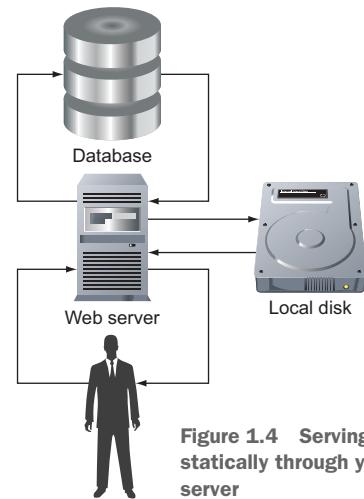


Figure 1.4 Serving photos statically through your web server

a single web server living somewhere inside a data center, serving a photo it has stored locally on its disk. For simplicity, let's assume this server lives somewhere in the central United States. This means that if someone nearby (for example, in New York) requests that photo, they'll get a relatively zippy response. But what if someone far away, like in Japan, requests the photo? The only way to get it is to send a request from Japan to the United States, and then the server needs to ship all the bytes from the United States back to Japan.

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

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

In this case, when someone uploads a photo to your server, you'd resize it and edit it however you want, and then forward the final image along to Google Cloud Storage, using its API client to ship the bytes securely. See figure 1.5. After that, all you'd do is refer to the photo using the Cloud Storage URL, and all of the problems from before are taken care of.

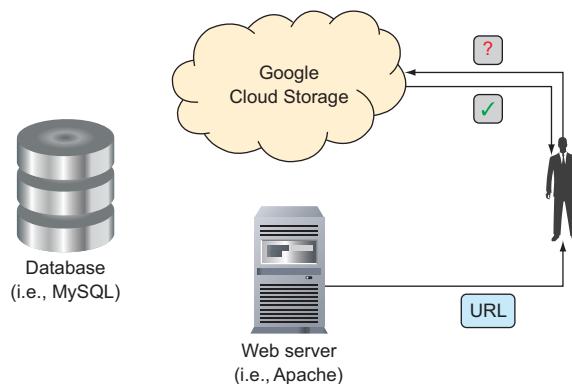


Figure 1.5 Serving photos statically through Google Cloud Storage

This is only one example, but the theme you should take away from this is that cloud is more than a different way of managing computing resources. It's also about using

managed or hosted services via simple APIs to do complex things, meaning you think less about the physical computers.

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

1.4.3 Example projects

Let's explore a few concrete examples of projects you might work on.

To-Do List

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

Throughout this book, we rely on this example to illustrate how you might use Google Cloud for your personal projects, which quite often involve storing and retrieving data and serving either API or web requests to users. You'll notice that the focus of this example is building something "real," but it won't cover all of the edge cases (and there may be many) or any of the more advanced or enterprise-grade features. In short, the To-Do List is a useful demonstration of doing something real, but incredibly simple, with cloud infrastructure.

InstaSnap

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

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

E*EXCHANGE

E*Exchange is our example of more grown-up application development that tends to come with growing from a small or mid-sized company into a larger, more mature, more heavily capitalized company, which means audits, Sarbanes-Oxley, and all the other

(potentially scary) requirements. To make things more complicated, E*Exchange is an application for trading stocks in the United States, and, therefore, will act as an example of applications operating in more highly regulated industries, such as finance.

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

1.5 Getting started with Google Cloud Platform

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

1.5.1 Signing up for GCP

Before you can start using any of Google's Cloud services, you first need to sign up for an account. If you already have a Google account (such as a Gmail account), you can use that to log in, but you'll still need to sign up specifically for a cloud account. If you've already signed up for Google Cloud Platform (see figure 1.6), feel free to skip

The screenshot shows the Google Cloud Platform homepage. At the top, there is a navigation bar with links for Why Google, Products, Solutions, Launcher, Pricing, Customers, Documentation, Support, and Partners. On the right side of the navigation bar are buttons for 'TRY IT FREE' and 'CONTACT SALES'. Below the navigation bar, the main heading reads 'Build What's Next' and 'Better software. Faster.' followed by a bulleted list: '✓ Use Google's core infrastructure, data analytics and machine learning.', '✓ Secure and fully featured for all enterprises.', and '✓ Committed to open source and industry leading price-performance.'. There are two buttons at the bottom of this section: 'TRY IT FREE' and 'CONTACT SALES'. Below this section, there are three columns of content: 'Forrester Research' (Google Cloud is named the Insight PaaS Leader by Forrester), 'GCP Region Expansion' (Run workloads in even more locations around the world. Our newest regions: Frankfurt, São Paulo and Mumbai.), and 'Let's Talk About AI' (Join the Cloud OnAir: The Journey From Big Data to AI global event on December 5.). Each column has a 'LEARN MORE' button. At the very bottom, the heading 'Why Google Cloud Platform?' is centered above three decorative icons: a stylized 'A', a wavy line, and a series of colored bars.

Figure 1.6 Google Cloud Platform

ahead. First, navigate to <https://cloud.google.com>, and click the button that reads “Try it free!” This will take you through a typical Google sign-in process. If you don’t have a Google account yet, follow the sign-up process to create one.

If you’re eligible for the free trial, you’ll see a page prompting you to enter your billing information. The free trial, shown in figure 1.7, gives you \$300 to spend on Google Cloud over a period of 12 months, which should be more than enough time to explore all the things in this book. Additionally, some of the products on Google Cloud Platform have a free tier of usage. Either way, all the exercises in this book will remind you to turn off any resources after the exercise is finished.

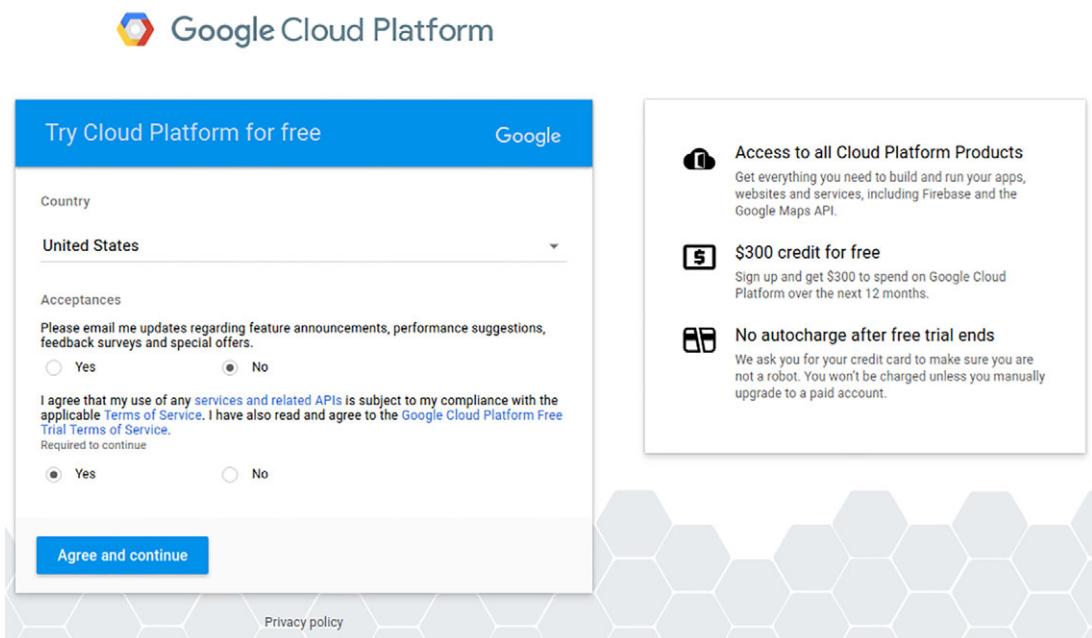


Figure 1.7 Google Cloud Platform free trial

1.5.2 Exploring the console

After you’ve signed up, you are automatically taken to the Cloud Console, shown in figure 1.8, and a new project is automatically created for you. You can think of a project like a container for your work, where the resources in a single project are isolated from those in all the other projects out there.

On the left side of the page are categories that correspond to all the different services that Google Cloud Platform offers (for example, Compute, Networking, Big Data, and Storage), as well as other project-specific configuration sections (such as authentication, project permissions, and billing). Feel free to poke around in the console to familiarize yourself with where things live. We’ll come back to all of these

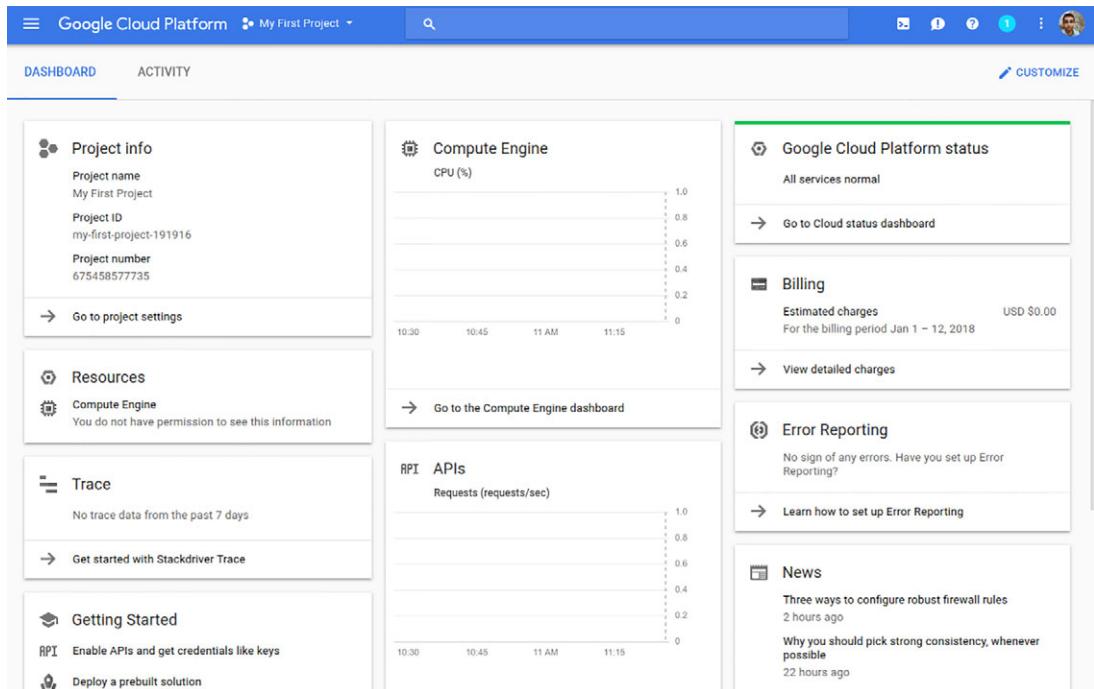


Figure 1.8 Google Cloud Console

things later as we explore each of these areas. Before we go any further, let's take a moment to look a bit closer at a concept that we threw out there: projects.

1.5.3 Understanding projects

When we first signed up for Google Cloud Platform, we learned that a new project is created automatically, and that projects have something to do with isolation, but what does this mean? And what are projects anyway? Projects are primarily a container for all the resources we create. For example, if we create a new VM, it will be “owned” by the parent project. Further, this ownership spills over into billing—any charges incurred for resources are charged to the project. This means that the bill for the new VM we mentioned is sent to the person responsible for billing on the parent project. (In our examples, this will be you!)

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

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

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

1.5.4 *Installing the SDK*

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

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

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

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

As you can see, upgrading components is pretty simple: run `gcloud components update`, and the SDK handles everything. After you have everything installed, you have to tell the SDK who you are by logging in. Google made this easy by connecting your terminal and your browser:

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

[A long link is here]

Created new window in existing browser session.
```

You should see a normal Google login and authorization screen asking you to grant the Google Cloud SDK access to your cloud resources. Now when you run future gcloud commands, you can talk to Google Cloud Platform APIs as yourself. After you click Allow, the window should automatically close, and the prompt should update to look like this:

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

[A long link is here]

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

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

You're now authenticated and ready to use the Cloud SDK as yourself. But what about that warning message? It says that even though you're logged in and all the gcloud commands you run will be authenticated as you, any code that you write may not be. You can make any code you write in the future automatically handle authentication by using application default credentials. You can get these using the gcloud auth sub-command once again:

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

[Another long link is here]

Created new window in existing browser session.

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

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

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

1.6 *Interacting with GCP*

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

1.6.1 *In the browser: the Cloud Console*

Let’s start by navigating to the Google Compute Engine area of the console: click the Compute section to expand it, and then click the Compute Engine link that appears. The first time you click this link, Google initializes Compute Engine for you, which should take a few seconds. Once that’s complete, you should see a Create button, which brings you to a page, shown in figure 1.9, where you can configure your virtual machine.

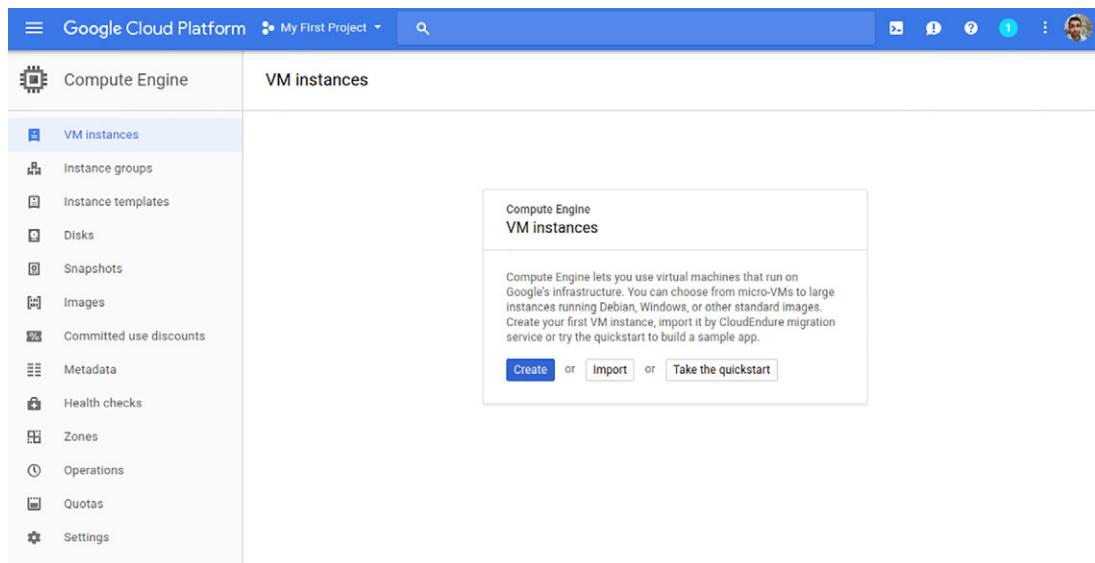


Figure 1.9 Google Cloud Console, where you can create a new virtual machine

On the next page, a form (figure 1.10) lets you configure all the details of your instance, so let’s take a moment to look at what all of the options are.

First there is the instance Name. The name of your virtual machine will be unique inside your project. For example, if you try to create “instance-1” while you already have an instance with that same name, you’ll get an error saying that name is already taken. You can name your machines anything you want, so let’s name our instance “learning-cloud-demo.” Below that is the Zone field, which represents where the machine should live geographically. Google has data centers all over the place, so you

← Create an instance

Name ?
learning-cloud-demo

Zone ?
us-east1-b

Machine type
Customize to select cores, memory and GPUs.

\$24.67 per month estimated
Effective hourly rate \$0.034 (730 hours per month)

1 vCPU 3.75 GB memory Customize

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

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

Identity and API access ?

Service account ?
Compute Engine default service account

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

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

Allow HTTP traffic
 Allow HTTPS traffic

Management, disks, networking, SSH keys

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

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

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

Figure 1.10 Form where you define your virtual machine

can choose from several options of where you want your instance to live. For now, let's put our instance in us-central1-b (which is in Iowa).

Next is the Machine Type field, where you can choose how powerful you want your cloud instances to be. Google has lots of different sizing options, ranging from

f1-micro (which is a small, not powerful machine) all the way up to n1-highcpu-32 (which is a 32-core machine), or a n1-highmem-32 (which is a 32-core machine with 208 GB of RAM). As you can see, you have quite a few options, but because we’re testing things out, let’s leave the machine type as n1-standard-1, which is a single-core machine with about 4 GB of RAM.

Many, many more knobs let you configure your machine further, but for now, let’s launch this n1-standard-1 machine to test things out. To start the virtual machine, click Create and wait a few seconds.

TESTING OUT YOUR INSTANCE

After your machine is created, you should see a green checkmark in the list of instances in the console. But what can you do with this now? You might notice in the Connect column a button that says “SSH” in the cell. See figure 1.11.

Name	Zone	Recommendation	Internal IP	External IP	Connect
<input checked="" type="checkbox"/> learning-cloud-demo	us-east1-b		10.142.0.2	35.227.93.212	<input type="button" value="SSH"/>

Figure 1.11 The listing of your VM instances

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

1.6.2 On the command line: `gcloud`

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

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

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

CONNECTING TO YOUR INSTANCE

Now that you can see your instance, you probably are curious about how to connect to it like we did with the SSH button. Type `gcloud compute ssh learning-cloud-demo` and choose the zone where you created the machine (us-central1-b). You should be connected to your machine via SSH:

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

Updated [https://www.googleapis.com/compute/v1/projects/glass-arcade-111313].
Warning: Permanently added '104.154.94.41' (ECDSA) to the list of known hosts.
Linux learning-cloud-demo 3.16.0-0.bpo.4-amd64 #1 SMP Debian 3.16.7-ckt11-1+deb8u3-bpo70+1 (2015-08-08) x86_64
```

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

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

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

1.6.3 In your own code: google-cloud-*

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

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

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

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

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

Listing 1.1 Showing all VMs (script.js)

```
const gce = require('@google-cloud/compute')({
  projectId: 'your-project-id'                                     ↪ Make sure to change
});                                                               this to your project ID!
const zone = gce.zone('us-central1-b');

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

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

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

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

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

Listing 1.2 Showing and stopping all VMs

```
const gce = require('@google-cloud/compute')({  
  projectId: 'your-project-id'  
});  
const zone = gce.zone('us-central1-b');  
  
console.log('Getting your VMs...');  
  
zone.getVMs().then((data) => {  
  data[0].forEach((vm) => {  
    console.log('Found a VM called', vm.name);  
    console.log('Stopping', vm.name, '...');  
    vm.stop((err, operation) => {  
      operation.on('complete', (err) => {  
        console.log('Stopped', vm.name);  
      });  
    });  
  });  
});
```

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

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

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

Summary

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

Trying it out: deploying WordPress on Google Cloud

This chapter covers

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

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

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

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

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

2.1 System layout overview

Before we get down to the technical pieces of turning on machines, let's start by looking at what we need to turn on. We'll do this by looking at the flow of an ideal request through our future system. We're going to imagine a person visiting our future blog and look at where their request needs to go to give them a great experience. We'll start with a single machine, shown in figure 2.1, because that's the simplest possible configuration.

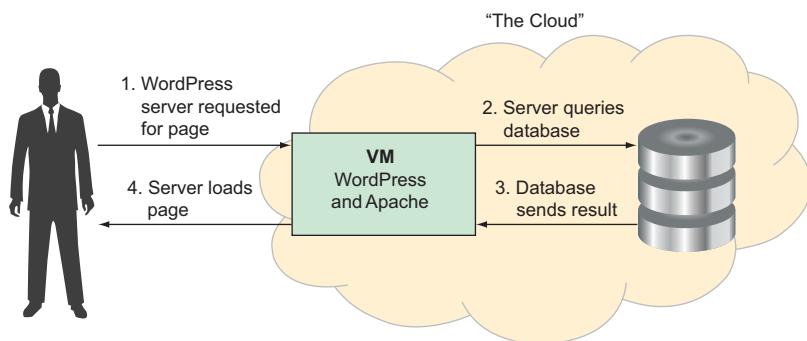


Figure 2.1 Flow of a future request to a VM running WordPress

As you can see here, the flow is

- 1 Someone asks the WordPress server for a page.
- 2 The WordPress server queries the database.
- 3 The database sends back a result (for example, the content of the page).
- 4 The WordPress server sends back a web page.

Simple enough, right? What happens as things get a bit more complex? Although we won't demonstrate this configuration here, you might recall in chapter 1 where we discussed the idea of relying on cloud services for more complicated hosting problems like content distribution. (For example, if your servers are in the United States, what's the experience going to be like for your readers in Asia?) To give an idea of how this might look, figure 2.2 shows a flow diagram for a WordPress server using Google Cloud Storage to handle static content (like images).

In this case, the flow is the same to start. Unlike before, however, when static content is requested, it doesn't reuse the same flow. In this configuration, your WordPress server modifies references to static content so that rather than requesting it from the WordPress server, the browser requests it from Google Cloud Storage (steps 5 and 6 in figure 2.2).

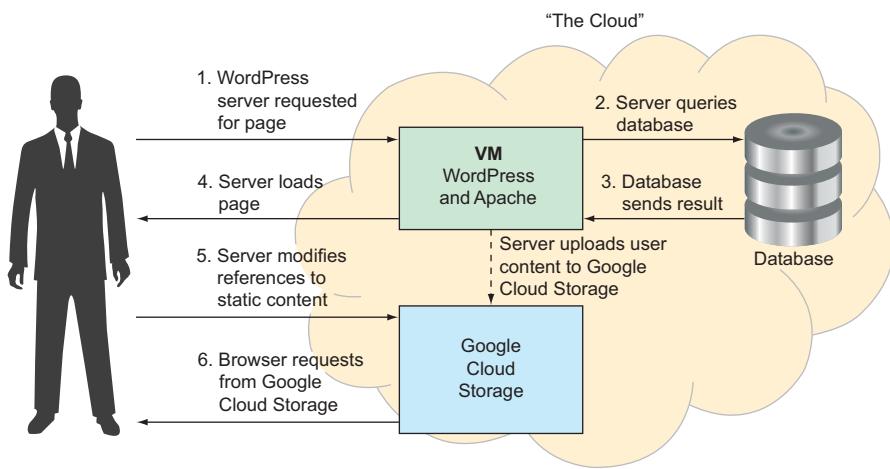


Figure 2.2 Flow of a request involving Google Cloud Storage

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

2.2 *Digging into the database*

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

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

All of the major cloud-hosting providers offer this sort of service—for example, Amazon has Relational Database Service (RDS), Azure has SQL Database service, and Google has Cloud SQL service. Managing a database via Cloud SQL is quicker

and easier than configuring and managing the underlying virtual machine and its software, so we’re going to use Cloud SQL for our database. This service isn’t always going to be the best choice (see chapter 4 for much more detail about Cloud SQL), but for our WordPress deployment, which is typical, Cloud SQL is a great fit. It looks almost identical to a MySQL server that you’d configure yourself, but is easier and faster to set up.

2.2.1 Turning on a Cloud SQL instance

The first step to turning on our database is to jump into the Cloud Console by going to the Cloud Console (cloud.google.com/console) and then clicking SQL in the left-side navigation, underneath the Storage section. You’ll see the blue Create instance button, shown in figure 2.3.

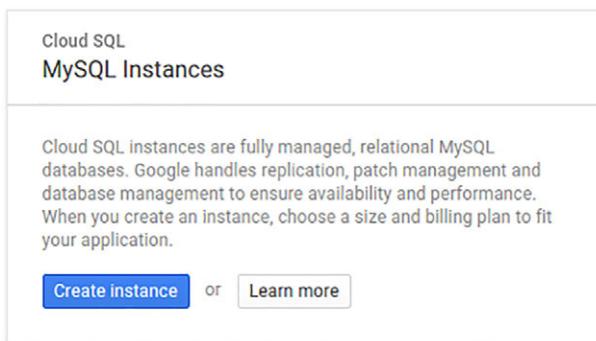


Figure 2.3 Prompt to create a new Cloud SQL instance

When you select a Second Generation instance (see chapter 4 for more detail on these), you’ll be taken to a page where you can enter some information about your database. See figure 2.4. The first thing you should notice is that this page looks a little bit like the one you saw when creating a virtual machine. This is intentional—you’re creating a virtual machine that Google will manage for you, as well as install and configure MySQL for you. Like with a virtual machine, you need to name your database. For this exercise, let’s name the database `wordpress-db` (also like VMs, the name has to be unique inside your project, so you can have only one database with this name at a time).

Next let’s choose a password to access MySQL. Cloud Console can automatically generate a new secure password, or you can choose your own. We’ll choose `my-very-long-password!` as our password. Finally, again like a VM, you have to choose where (geographically) you want your database to live. For this example, we’ll use `us-central1-c` as our zone.

To do any further configuration, click Show configuration options near the bottom of the page. For example, we might want to change the size of the VM instance for our database (by default, this uses a `db-n1-standard-1` type instance) or increase

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

Root password
Set a password for the root user. [Learn more](#)
 [Generate](#)
 No password

Location [?](#)
For better performance, keep your data close to the services that need it.

Estimated monthly total	Hourly rate
\$51.89	\$0.071
~730 hours per month	

db-n1-standard-1 machine \$70.45 / month
10 GB SSD, with backups \$2.58 / month
Committed use discount [?](#) -\$21.13 / month

[Less](#)

Show configuration options

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

Figure 2.4 Form to create a new Cloud SQL instance

the size of the underlying disk (by default, Cloud SQL starts with a 10 GB SSD disk). You can change all the options on this page later—in fact, the size of your disk automatically increases as needed—so let’s leave them as they are and create our instance. After you’ve created your instance, you can use the `gcloud` command-line tool to show that it’s all set with the `gcloud sql` command:

```
$ gcloud sql instances list
NAME      REGION   TIER          ADDRESS        STATUS
wordpress-db - db-n1-standard-1 104.197.207.227 RUNNABLE
```

TIP Can you think of a time that you might have a large persistent disk that will be mostly empty? Take a look at chapter 9 if you’re not sure.

2.2.2 Securing your Cloud SQL instance

Before you go any further, you should probably change a few settings on your SQL instance so that you (and, hopefully, only you) can connect to it. For your testing phase you will change the password on the instance and then open it up to the world. Then, after you test it, you’ll change the network settings to allow access only from your Compute Engine VMs. First let’s change the password. You can do this from the command line with the `gcloud sql users set-password` command:

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

In this example, you reset the password for the root user across all hosts. (The MySQL wildcard character is a percent sign.) Now let's (temporarily) open the SQL instance to the outside world. In the Cloud Console, navigate to your Cloud SQL instance. Open the Authorization tab, click the Add network button, add "the world" in CIDR notation (0.0.0.0/0, which means "all IPs possible"), and click Save. See figure 2.5.

wordpress-db

MySQL Second Generation master

OVERVIEW USERS DATABASES AUTHORIZATION

ⓘ You have not authorized any external networks to connect to your Cloud SQL instance. External applications can still connect to the instance through the Cloud SQL Proxy. [Learn more](#)

⚠ You have added 0.0.0.0/0 as an allowed network. This prefix will allow any IPv4 client to pass the network firewall and make login attempts to your instance, including clients you did not intend to allow. Clients still need valid credentials to successfully log in to your instance.

Authorized networks

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

New network

Name (Optional)
None

Network
Use CIDR notation. ↗
0.0.0.0/0

Done Cancel

+ Add network

App Engine authorization

All apps in this project are authorized by default. To authorize apps in other projects, follow the steps below.

Apps in this project: All authorized.

Authorize apps in other projects

Save Discard changes

Figure 2.5 Configuring access to the Cloud SQL instance

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

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

2.2.3 Connecting to your Cloud SQL instance

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

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

On Windows or Mac, you can download the package from the MySQL website: <http://dev.mysql.com/downloads/mysql/>. After installation, connect to the database by entering the IP address of your instance (you saw this before with `gcloud sql instances list`). Use the username "root", and the password you set earlier. Here's this process on Linux:

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

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

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

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

mysql>
```

Next let's run a few SQL commands to prepare your database for WordPress.

2.2.4 Configuring your Cloud SQL instance for WordPress

Let's get the MySQL database prepared for WordPress to start talking to it. Here's a basic outline of what we're going to do:

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

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

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

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

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

Next you need to give this new user the right level of access to do things to the database (like create tables, add rows, run queries, and so on):

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

Finally let's tell MySQL to reload the list of users and privileges. If you forget this command, MySQL would know about the changes when it restarts, but you don't want to restart your Cloud SQL instance for this:

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

That's all you have to do on the database! Next let's make it do something real.

Quiz

How does your database get backed up? Take a look at chapter 4 on Cloud SQL if you're not sure.

2.3 Deploying the WordPress VM

Let's start by turning on the VM that will host our WordPress installation. As you learned, you can do this easily in the Cloud Console, so let's do that once more. See figure 2.6.

Take note that the check boxes for allowing HTTP and HTTPS traffic are selected because we want our WordPress server to be accessible to anyone through their browsers. Also make sure that the Access Scopes section is set to allow default access. After that, you're ready to turn on your VM, so go ahead and click Create.

[← Create an instance](#)

Name 

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

Machine type
Customize to select cores, memory and GPUs.

1 vCPU 3.75 GB memory

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

Boot disk   New 50 GB standard persistent disk
Image: Ubuntu 16.04 LTS

Identity and API access 

Service account 

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

Firewall  Add tags and firewall rules to allow specific network traffic from the Internet

Allow HTTP traffic
 Allow HTTPS traffic

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

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

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

Figure 2.6 Creating a new VM instance

Quiz

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

Take a look at chapter 3 if you're not sure.

2.4 Configuring WordPress

The first thing to do now that your VM is up and running is to connect to it via SSH. You can do this in the Cloud Console by clicking the SSH button, or use the Cloud SDK with the gcloud compute ssh command. For this walkthrough, you'll use the Cloud SDK to connect to your VM:

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

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

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

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

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

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

```
jjg@wordpress:~$
```

After you're connected, you need to install a few packages, namely Apache, MySQL Client, and PHP. You can do this using apt-get:

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

When prompted, confirm by typing Y and pressing Enter. Now that you have all the prerequisites installed, it's time to install WordPress. Start by downloading the latest version from wordpress.org and unzipping it into your home directory:

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

You'll need to set some configuration parameters, primarily where WordPress should store data and how to authenticate. Copy the sample configuration file to wp-config.php, and then edit the file to point to your Cloud SQL instance. In this example, I'm using Vim, but you can use whichever text editor you're most comfortable with:

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

After editing wp-config.php, it should look something like the following listing.

Listing 2.1 WordPress configuration after making changes for your environment

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

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

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

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

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

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

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

After you have your configuration set (you should need to change only the database settings), move all those files out of your home directory and into somewhere that

Apache can serve them. You also need to remove the Apache default page, `index.html`. The easiest way to do this is using `rm` and then `rsync`:

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

Now navigate to the web server in your browser (for example, <http://104.197.86.115> in this specific example), which should end up looking like figure 2.7.

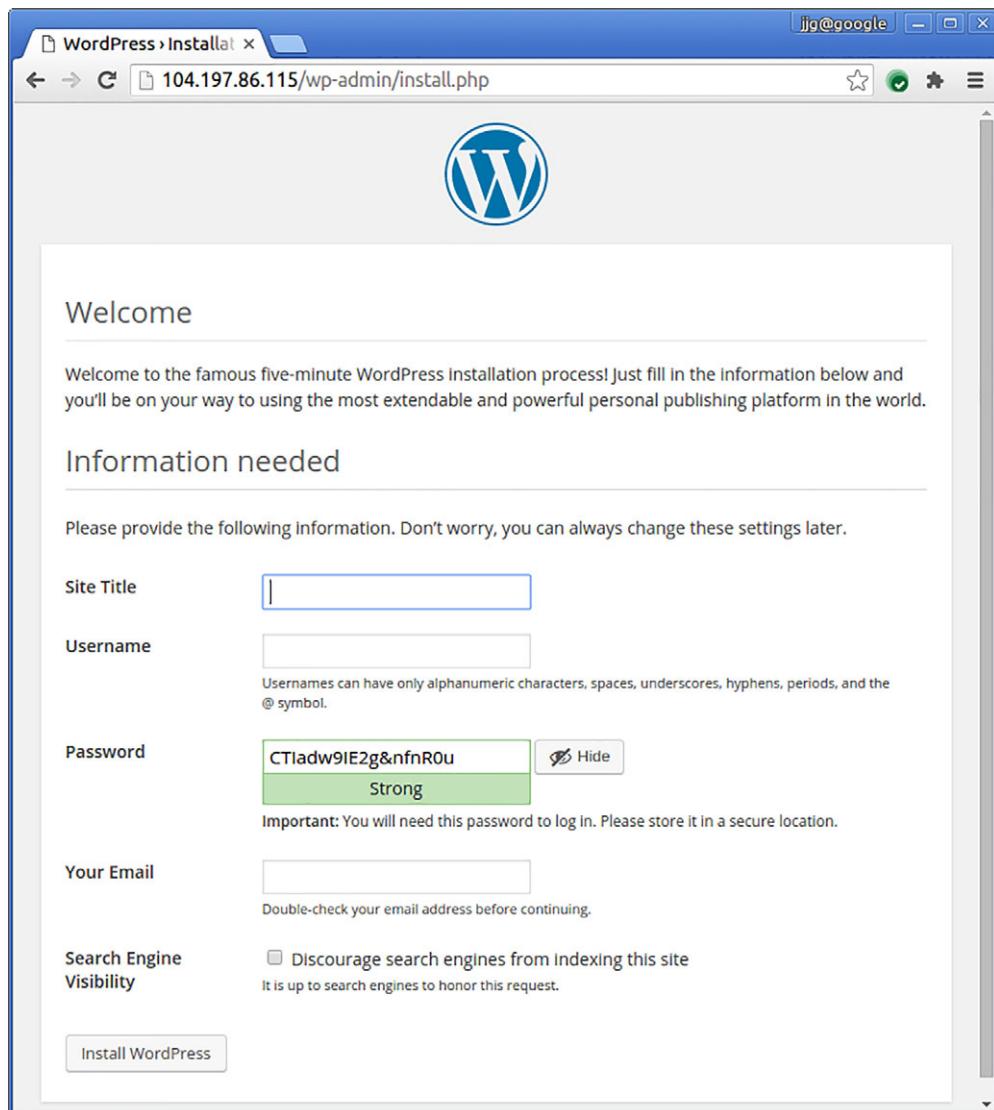


Figure 2.7 WordPress is up and running.

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

2.5 **Reviewing the system**

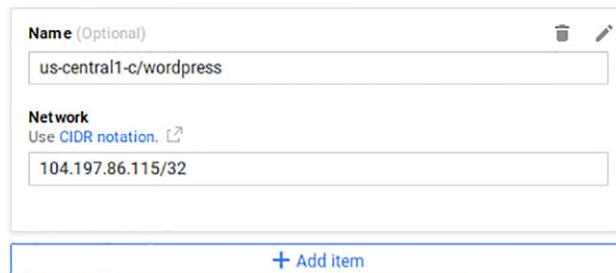
So what did you do here? You set up quite a few different pieces:

- You turned on a Cloud SQL instance to store all of your data.
- You added a few users and changed the security rules.
- You turned on a Compute Engine virtual machine.
- You installed WordPress on that VM.

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

To do this, go to the Cloud Console, and navigate to your Cloud SQL instance. On the Access Control tab, edit the Authorized Network, changing `0.0.0.0/0` to the IP address followed by `/32` (for example, `104.197.86.115/32`), and rename the rule to read `us-central1-c/wordpress` so you don't forget what this rule is for. When you're done, the access control rules should look like figure 2.8.

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



The screenshot shows the 'Authorized Networks' section of the Cloud SQL instance configuration. It includes fields for 'Name (Optional)' containing 'us-central1-c/wordpress', 'Network' with 'Use CIDR notation.' checked, and the value '104.197.86.115/32'. A blue button at the bottom left says '+ Add item'.

Figure 2.8 Updating the access configuration for Cloud SQL

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

2.6 Turning it off

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

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

So why wouldn't we always stop instances rather than delete them? The catch with stopping is that you have to keep your persistent disks around, and those cost money. You won't be paying for CPU cycles on a stopped instance, but the disk that stores the operating system and all your configuration settings needs to stay around. You are billed for your disks whether or not they're attached to a running virtual machine. In this case, if you're done with your WordPress installation, the right choice is probably deleting rather than stopping it. When you click delete, you should notice that the confirmation prompt reminds you that your disk (the boot disk) will also be deleted. See figure 2.9.

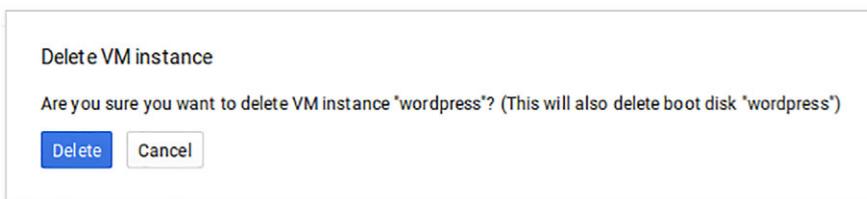


Figure 2.9 Deleting the VM when we're finished

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

Summary

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

The cloud data center

This chapter covers

- What data centers are and where they are
- Data center security and privacy
- Regions, zones, and disaster isolation

If you've ever paid for web hosting before, it's likely that the computer running as your web host was physically located in a data center. As you learned in chapter 1, deploying in the cloud is similar to traditional hosting, so, as you'd expect, if you turn on a virtual machine in, or upload a file to, the cloud, your resources live inside a data center. But where are these data centers? Are they safe? Should you trust the employees who take care of them? Couldn't someone steal your data or the source code to your killer app?

All of these questions are valid, and their answers are pretty important—after all, if the data center was in somebody's basement, you might not want to put your banking details on that server. The goal of this chapter is to explain how data centers have evolved over time and highlight some of the details of Google Cloud Platform's data centers. Google's data centers are pretty impressive (as shown in figure 3.1), but this isn't a fashion show. Before you decide to run mission-critical stuff in a data center, you probably want to understand a little about how it works.

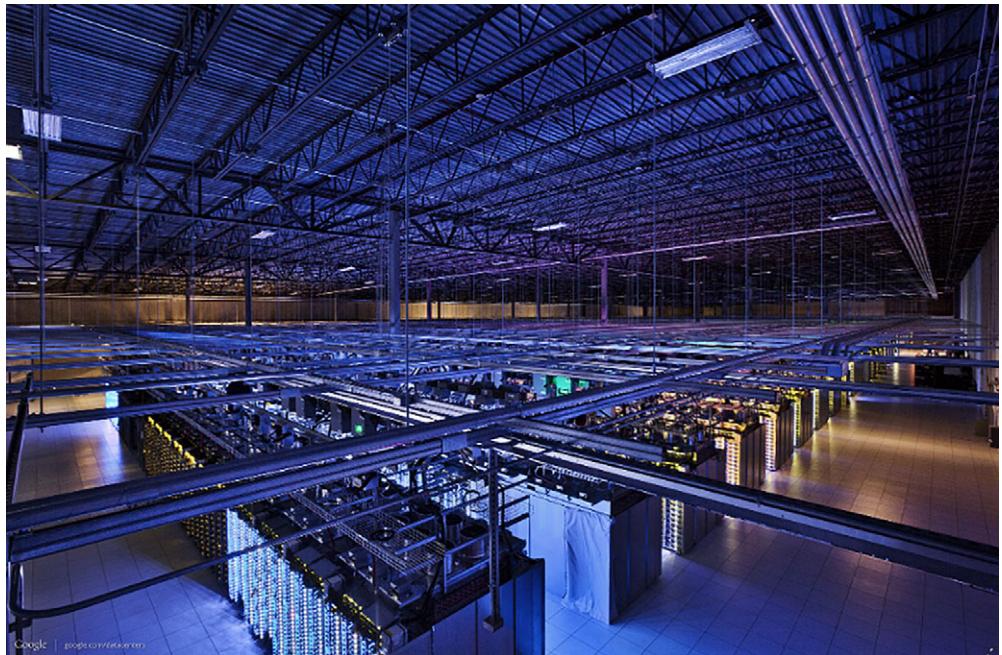


Figure 3.1 A Google data center

Keep in mind that many of the things you'll read in this chapter about data centers are industrywide standards, so if something seems like a great feature (such as strict security to enter the premises), it probably exists with other cloud providers as well (like Amazon Web Services or Microsoft Azure). I'll make sure to call out things that are Google-specific so it's clear when you should take note. I'll start by laying out a map to understand Google Cloud's data centers.

3.1 Data center locations

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

To choose the right place, you first need to know what your choices are. As of this writing, Google Cloud operates data centers in 15 different regions around the world, including in parts of the United States, Brazil, Western Europe, India, East Asia, and Australia. See figure 3.2.



Figure 3.2 Cities where Google Cloud has data centers and how many in each city (white balloons indicate “on the way” at the time of this writing.)

This might not seem like a lot, but keep in mind that each city has many different data centers for you to choose from. Table 3.1 shows the physical places where your data resources can exist.

Table 3.1 Zone overview for Google Cloud

Region	Location	Number of data centers
Total		44
Eastern US	South Carolina, USA	3
Eastern US	North Virginia, USA	3
Central US	Iowa, USA	4
Western US	Oregon, USA	3
Canada	Montréal, Canada	3
South America	São Paulo, Brazil	3
Western Europe	London, UK	3
Western Europe	Belgium	3
Western Europe	Frankfurt, Germany	3
Western Europe	Netherlands	2
South Asia	Mumbai, India	3
South East Asia	Singapore	2
East Asia	Taiwan	3

Table 3.1 Zone overview for Google Cloud (continued)

Region	Location	Number of data centers
North East Asia	Tokyo, Japan	3
Australia	Sydney, Australia	3

How does this stack up to other cloud providers, as well as traditional hosting providers? Table 3.2 will give you an idea.

Table 3.2 Data center offerings by provider

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

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

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

NOTE I cover a third reason based on legal concerns in section 3.3.3.

If your requirements are less strict, the locations of data centers shouldn't make too much of a difference in choosing a cloud provider. Still, it's important to understand your latency requirements and how geographical location might affect whether you meet them or not (figure 3.3).

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

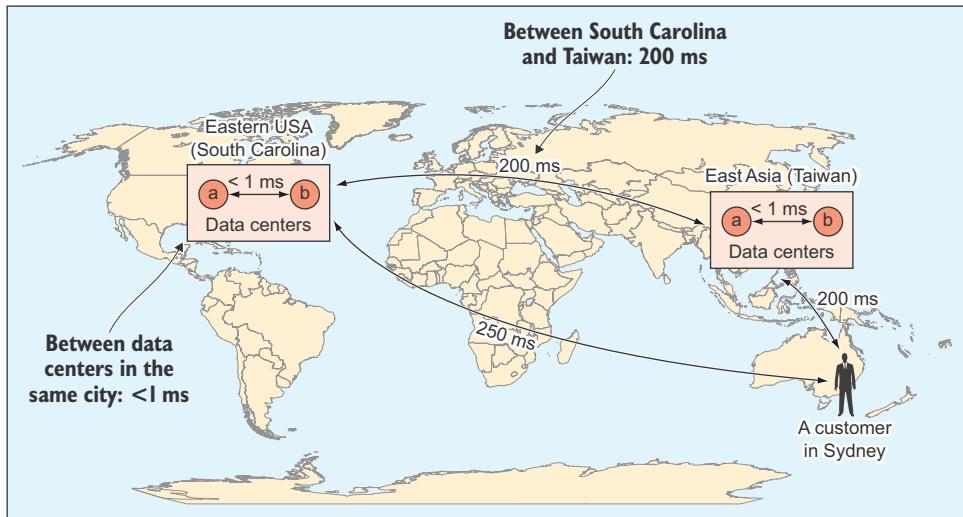


Figure 3.3 Latencies between different cities and data centers

3.2 *Isolation levels and fault tolerance*

Although I've talked about cities, regions, and data centers, I haven't defined them in much detail. Let's start by talking about the types of places where resources can exist.

3.2.1 *Zones*

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

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

3.2.2 *Regions*

Moving up the stack, a collection of zones is called a *region*, and this corresponds loosely to a city (as you saw in table 3.1), such as Council Bluffs, Iowa, USA. If you turn on two resources in the same region but different zones, say `us-east1-b` and `us-east1-c`, the resources will be somewhat close together (meaning the latency between them will be

shorter than if one resource were in a zone in Asia), but they're guaranteed to not be in the same physical facility.

In this case, although your two resources might be isolated from zone-specific failures (like a power outage), they might not be isolated from catastrophes (like a tornado). See figure 3.4. You might see regions abbreviated by dropping the last letter on the zone. For example, if the zone is us-central1-a, the region would be us-central1.

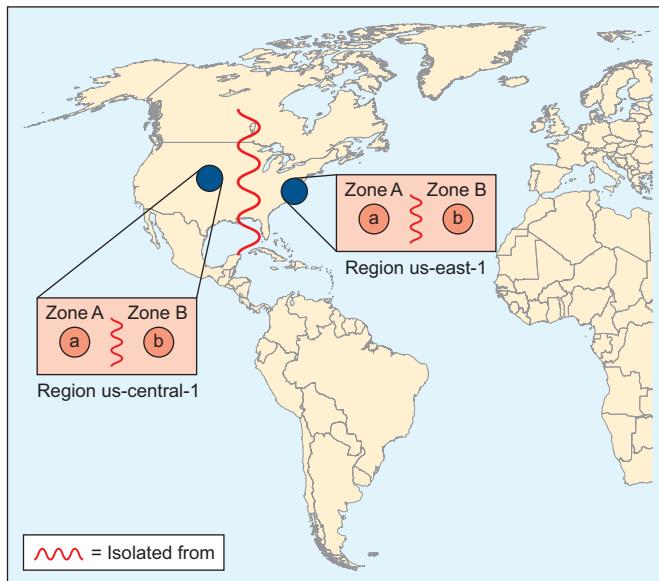


Figure 3.4 A comparison of regions and zones

3.2.3 Designing for fault tolerance

Now that you understand what zones and regions are, I can talk more specifically about the different levels of isolation that Google Cloud offers. You might also hear these described as *control planes*, borrowing the term from the networking world. When I refer to isolation level or the types of control plane, I'm talking about what thing would have to go down to take your service down with it. Services are available, and can be affected, at several different levels:

- **Zonal**—As I mentioned in the example, a service that's *zonal* means that if the zone it lives in goes down, it also goes down. This happens to be both the easiest type of service to build—all you need to do is turn on a single VM and you have a zonal service—and the least highly available.
- **Regional**—A regional service refers to something that's replicated throughout multiple zones in a single region. For example, if you have a MongoDB instance living in us-east1-b, and a hot-failover living in us-east1-c, you have a regional service. If one zone goes down, you automatically flip to the instance in the other zone. But if an earthquake swallows the entire city, both zones will

go down with the region, taking your service with it. Although this is unlikely, and regional services are much less likely to suffer outages, the fact that they're geographically colocated means you likely don't have enough redundancy for a mission-critical system.

- *Multiregional*—A multiregional service is a composition of several different regional services. If some sort of catastrophe occurs that takes down an entire region, your service should still continue to run with minimal downtime (figure 3.5).
- *Global*—A global service is a special case of a multiregional service. With a global service, you typically have deployments in multiple regions, but these regions are spread around the world, crossing legal jurisdictions and network providers. At this point, you typically want to use multiple cloud providers (for example, Amazon Web Services alongside Google Cloud) to protect the service against disasters spanning an entire company.

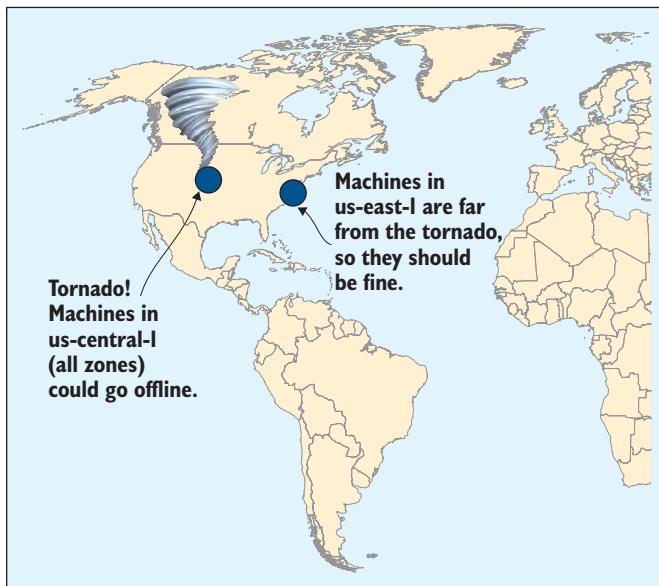


Figure 3.5 Disasters like tornadoes are likely to affect a single region at a time.

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

The important thing when building your service isn't primarily using the most highly available configuration, but knowing what your levels of fault tolerance and isolation are at any time. Armed with that knowledge, if any part of your system becomes absolutely critical, you at least know which pieces will need redundant deployments and where those new resources should go. I'll talk much more about redundancy and high availability when I discuss Compute Engine in chapter 9.

3.2.4 Automatic high availability

Over the years, certain common patterns have emerged that show where systems need to be highly available. Based on these patterns, many cloud providers have designed richer systems that are automatically highly available. This means that instead of having to design and build a multiregional storage system yourself, you can rely on Google Cloud Storage, which provides the same level of fault isolation (among other things) for your basic storage needs.

Several other systems follow this pattern, such as Google Cloud Datastore, which is a multiregional nonrelational storage system that stores your data in five different zones, and Google App Engine, which offers two multiregional deployment options (one for the United States and another for Europe) for your computing needs. If you run an App Engine application, save some data in Google Cloud Storage, or store records in Google Cloud Datastore, and an entire region explodes, taking down all zones with it, your application, data, and records all will be fine and remain accessible to you and your customers. Pretty crazy, right?

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

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

3.3 Safety concerns

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

3.3.1 Security

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

If you store a photo on a hard drive in your home, you know exactly where the photo is—on your desk. Alternatively, if you upload a photo to a cloud service like Google Cloud Storage or Amazon's S3, the exact location of the data is a bit more complicated to determine, but you can at least pinpoint the region of the world where it lives. On the other hand, the entire photo is unlikely to live in only one place—different pieces of multiple copies of the file likely are stored on lots of disk drives. What do you get for this trade-off? Is more ambiguity worth it? When you use a cloud service to do something like store your photos, you're paying for quite a bit more than the disk space; otherwise, the fee would be a flat rate per byte rather than a recurring monthly fee.

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

When talking about securing resources, you typically have three goals:

- *Privacy*—Only authorized people should be able to access the resources.
- *Availability*—The resources should never be inaccessible to authorized people.
- *Durability*—The resources should never be corrupted or go missing.

In more specific terms with you and your photo, that would be

- *Privacy*—No one besides you should be able to look at your photo.
- *Availability*—You should never be told “Not right now, try again later!” when you ask to look at your photo.
- *Durability*—You should never come back and find your photo deleted or corrupted.

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

You can lump the next thing that can go wrong into a large group called unexpected disasters. This includes natural disasters, such as earthquakes, fires, and floods,

but in the case of storing data at home, it also includes more common accidents, such as power surges, hard drive failures, and kids spilling water on electronic equipment.

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

All of these types of accidents break the availability and durability goals, and some of them break the privacy goals. So how do cloud providers plan for these problems? Couldn't you do this yourself? The typical way cloud providers deal with these problems comes down to a few tactics:

- *Secure facilities*—Any facility housing resources (like hard drives) should be a high-security area, limiting who can come and go and what they can take with them. This is to prevent theft as well as sabotage.
- *Encryption*—Anything stored on disks should be encrypted. This is to prevent theft compromising data privacy.
- *Replication*—Data should be duplicated in many different places. This is to prevent a single failure resulting in lost data (durability) as well as a network outage limiting access to data (availability). This also means that a catastrophe (such as a fire) would only affect one of many copies of the data.
- *Backup*—Data should be backed up off-site and can be easily restored on request. This is to prevent a software bug accidentally overwriting all copies of the data. If this happens, you could ask for the old (correct) copy and disregard the new (erroneous) copy.

As you might guess, providing this sort of protection in your own home isn't just challenging and expensive—by definition it requires you to have more than one home! Not only would you need advanced security systems, you'd need full-time security guards, multiple network connections to each of your homes, systems that automatically duplicated data across multiple hard drives, key management systems for storing your encryption keys, and backups of data on rolling windows to different locations. I can comfortably say that this isn't something I'd want to do myself. Suddenly, a few cents per gigabyte per month doesn't sound all that bad.

3.3.2 Privacy

What about the privacy of your data? Google Cloud Storage might keep your photo in an encrypted form, but when you ask for it back, it arrives unencrypted. How can that be? The truth here is that although data is stored in encrypted form and transferred between data centers similarly, when you ask for your data, Google Cloud does have

the encryption key and uses it when you ask for your photo. This also means that if Google were to receive a court order, it does have the technical ability to comply with the order and decrypt your data without your consent.

To provide added security, many cloud services provide the ability to use your own encryption keys, meaning that the best Google can do is hand over encrypted data, because it doesn't have the keys to decrypt it. If you're interested in more details about this topic, you can learn more in chapter 8, where I discuss Google Cloud Storage.

3.3.3 **Special cases**

Sometimes special situations require heightened levels of security or privacy; for example:

- Government agencies often have strict requirements.
- Companies in the U.S. healthcare industry must comply with HIPAA regulations.
- Companies dealing with the personal data of German citizens must comply with the German BDSG.

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

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

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

- It's still possible to use cloud hosting.
- You may be slightly limited as to which services you can use.

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

3.4 **Resource isolation and performance**

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

This concept is fascinating, but it omits one important thing: Do two virtual half computers run as fast as one physical whole computer? This leads to further

questions, such as whether one person using a virtual half computer could run a CPU-intensive workload that spills over into the resources of another person using a second virtual half computer and effectively steal some of the CPU cycles from the other person. What about network bandwidth? Or memory? Or disk access? This issue has come to be known as the noisy neighbor problem (figure 3.6) and is something everyone running inside a cloud data center should understand, even if superficially.



Figure 3.6 Noisy neighbors can impinge on those nearby.

The short answer to those questions is that you'll only get perfect resource isolation on *bare metal* (nonvirtualized) machines.

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

In Google's case, all of the cloud services ultimately run on top of a system called Borg, which, as you can read in *Wired* magazine from March 2013, "is a way of efficiently parceling work across Google's vast fleet of ... servers." Because Google uses the same system internally for other services (such as Gmail and YouTube), resource isolation (or perhaps better phrased as *resource fairness*) is a feature that has almost a decade of work behind it and is constantly improving. More concretely, for you this means that if you purchase 1 vCPU worth of capacity on Google Compute Engine, you should get the same number of computing cycles, regardless of how much work other VMs are trying to do.

Summary

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

Part 2

Storage

N

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

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

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

Cloud SQL: managed relational storage

This chapter covers

- What is Cloud SQL?
- Configuring a production-grade SQL instance
- Deciding whether Cloud SQL is a good fit
- Choosing between Cloud SQL and MySQL on a VM

Relational databases, sometimes called SQL (pronounced like *sequel*) databases, are one of the oldest forms of structured data storage, going back to the 1980s. The term *relational database* comes from the idea that these databases store related data and then allow you to combine it to ask complex questions, such as “How old are this year’s top five highest paid employees?”

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

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

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

4.1 **What's Cloud SQL?**

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

Cloud SQL is perfectly compatible with MySQL, so if you currently use MySQL anywhere in your system, Cloud SQL is a viable option for you. Also, integrating with Cloud SQL involves nothing more than changing the hostname in your configuration to point at a Cloud SQL instance.

Configuration and performance tuning will be identical for Cloud SQL and your own MySQL server, so I won't get into those topics. Instead, this chapter will explain how Cloud SQL automates some of the more tedious tasks, like upgrading to a newer version of MySQL, running recurring backups, and securing your Cloud SQL instance so it only accepts connections from people you trust.

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

4.2 **Interacting with Cloud SQL**

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

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

When filling out the form (figure 4.1), be sure to pick a region that's nearby, so your queries won't be traveling around the world and back. In this example, you'll create the instance in us-east1. Once you click Create, Google will get to work setting up your Cloud SQL instance.

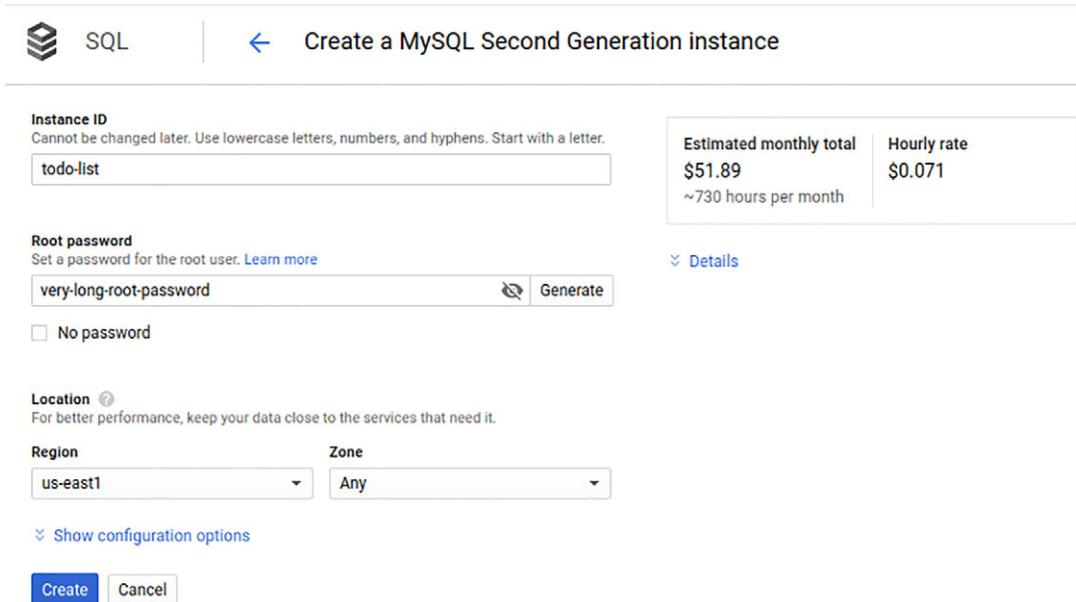


Figure 4.1 Creating a new Cloud SQL instance with your nonrequirements

Before talking to your database, you need to make sure you have access. MySQL uses password authentication, so to grant additional access, all you have to do is create new users. You can do this inside the Cloud Console by clicking on the Cloud SQL instance and choosing the Users tab (figure 4.2).

The screenshot shows the 'Instance details' page for the 'todo-list' instance. At the top are navigation links: 'Instance details' (with a back arrow), 'EDIT', 'IMPORT', 'EXPORT', and 'RESTART'. Below this is a green checkmark next to 'todo-list' and the text 'MySQL Second Generation master'. A horizontal menu bar below the instance name includes 'OVERVIEW', 'USERS' (which is underlined, indicating it is selected), 'DATABASES', 'AUTHORIZATION', 'SSL', and 'BACKUP'. Under the 'USERS' tab, there is a section titled 'MySQL user accounts' with the sub-instruction: 'User accounts enable users and applications to connect to your Cloud SQL instance.' It includes a 'Create user account' button and a table with one row: 'User name' 'root' and 'Host name' '% (any host)'. There is also a vertical ellipsis button.

Figure 4.2 The Access Control section with the Users tab selected

Here you can create a new user or change the root user's password, but make sure you keep track of the username and password that you create. You can do a lot of other things too, but I'll get into those in more detail later.

After you've created a user, it's time to switch environments completely, from the browser over to the command line. Open up a terminal, and start by checking whether you can see your Cloud SQL instance using the instances list command that lives in gcloud sql:

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

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

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

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

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

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

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

mysql>

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

The first thing you have to do is create a database for your app, which you can do by using the CREATE DATABASE command, as follows:

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

Now you can create a few tables for your To-Do Lists. If you're not familiar with relational database schema design, don't worry—nothing here is super-advanced.

First, you'll create a table to store your To-Do Lists, which will look something like table 4.1. This translates into the MySQL schema shown in listing 4.1.

Table 4.1 To-Do Lists table (todolists)

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

Listing 4.1 Defining the todolists table

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

Run that against the database you created, as shown in the following listing.

Listing 4.2 Creating the todolists table in your database

```
mysql> use todo;
Database changed

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

Now create the example lists I mentioned in table 4.1 so you can see how things work, as shown in the next listing.

Listing 4.3 Adding some sample To-Do Lists

```
msgy1> INSERT INTO todolists (`name`) VALUES ("Groceries"),
    ->   ("Christmas shopping"),
    ->   ("Vacation plans");
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

You can use a SELECT query to check if the lists are there, as follows.

Listing 4.4 Looking up your To-Do Lists

```
mysql> SELECT * FROM todolists;
+----+-----+
| id | name |
+----+-----+
| 1  | Groceries |
| 2  | Christmas shopping |
```

```
| 3 | Vacation plans |
+----+-----+
3 rows in set (0.02 sec)
```

Lastly, do the same thing again, but this time for to-do items for each checklist. The example data will look something like what's shown in table 4.2. That translates into the MySQL schema shown in listing 4.5.

Table 4.2 To-do items table (todoitems)

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

Listing 4.5 Creating the todoitems table

```
> CREATE TABLE `todoitems` (
    ->   `id` INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->   `todolist_id` INT(11) NOT NULL REFERENCES `todolists`.`id`,
    ->   `name` varchar(255) NOT NULL,
    ->   `done` BOOL NOT NULL DEFAULT '0'
    -> ) ENGINE = InnoDB;
Query OK, 0 rows affected (0.03 sec)
```

Then you can add the example to-do items, as follows.

Listing 4.6 Adding example items to the todoitems table

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

Next you can do things like ask for all the groceries that you still have to buy that sound like “egg,” as shown in the following listing.

Listing 4.7 Querying for groceries left to buy that sound like “egg”

```
mysql> SELECT `todoitems`.`name` from `todoitems`, `todolists` WHERE
->   `todolists`.`name` = "Groceries" AND
->   `todoitems`.`todolist_id` = `todolists`.`id` AND
->   `todoitems`.`done` = 0 AND
->   SOUNDEX(`todoitems`.`name`) LIKE CONCAT(SUBSTRING(SOUNDEX("egg"), 1,
2), "%");
```

```
+-----+  
| name |  
+-----+  
| Eggs |  
| Egg salad |  
+-----+  
2 rows in set (0.02 sec)
```

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

To delete a Cloud SQL instance, click Delete in the Cloud Console (figure 4.3). After that, you'll need to confirm you're deleting the right database, as shown in figure 4.4. (I wouldn't want you to delete the wrong one!)

The screenshot shows the 'Instance details' page for a MySQL Second Generation master instance named 'wordpress-db'. At the top, there are several buttons: EDIT, IMPORT, EXPORT, RESTART, STOP, DELETE, and CLONE. The 'DELETE' button is highlighted with a blue border. Below the buttons, the instance name 'wordpress-db' is listed with its type and status. A note at the bottom indicates it's a MySQL Second Generation master.

Figure 4.3 Deleting your Cloud SQL instance

The screenshot shows a 'Delete instance' dialog box. It contains a warning message: 'All the data in your instance will be deleted and cannot be retrieved. This action cannot be undone.' Below this, it asks to type the instance name to delete: 'To delete instance "todo-list", type your instance name: todo-list'. A text input field contains 'todo-list'. At the bottom are two buttons: a blue 'Delete' button and a white 'Cancel' button. To the right of the dialog, there are two time-related buttons: '12h' and '1 d'.

Figure 4.4 Confirming the instance you meant to delete

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

4.3 Configuring Cloud SQL for production

Now that you've learned how to turn on a Cloud SQL instance, it's time to go through what it takes to run Cloud SQL in a production-grade environment. Before I continue, it might be worthwhile to clarify that for the purposes of this chapter (and most of this book), when I say *production* I mean the environment that's safe for you to run a business in. In a production environment, you'd have things like reliable backups, failover procedures, and proper security practices. Now let's jump in by looking at one of the most obvious topics: access control.

4.3.1 Access control

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

If your system is spread out across many providers (maybe you have some VMs running in Amazon's EC2, some in Microsoft's Azure, and some in Google Compute Engine), the simplest thing to do is assign a static IP to these machines and then specifically limit access to those in the Authorization section when looking at the Cloud SQL instance. For example, if you have a VM running using the IP address `104.120.19.32`, you could allow access from that exact IP using CIDR notation, which would be `104.120.19.32/32` (figure 4.5). (The `/32` here means “This must be an exact match.”) These types of limits happen at the network level, which means that MySQL won't even hear about these requests coming in. This is a good thing because unless you've allowed access to an IP, your database appears completely invisible.

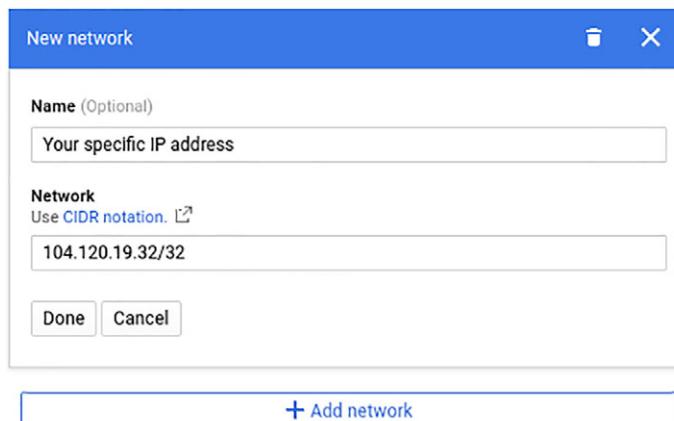


Figure 4.5 Setting access to a specific IP address

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

For example, the default network uses a special subnet for assigning internal IP addresses (`10.240.0.0/16`), which means that your machines will all have IPs matching this CIDR expression (for example, `10.240.0.1`). To limit access to these machines, you can use `10.240.0.0/16` (where `/16` means the last two numerals are wildcards).

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

4.3.2 Connecting over SSL

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

Having this security is important. Most of the time, you think of SSL as a thing for websites, but if you securely send your credit card number to a web server, and the web server then insecurely sends it to a database, you have a big problem. How do you make sure the connection to your databases is encrypted?

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

- The server's CA certificate
- A client certificate
- A client private key

Once you have them, the MySQL client knows what to do with them to establish a secure connection, so you don't need to do much more. To get these three things, start off by viewing your instance in the Cloud Console and jump into the SSL tab (figure 4.6).

The screenshot shows the 'Instance details' page for a MySQL Second Generation master instance named 'wordpress-db'. The 'SSL' tab is selected. A prominent warning message states: 'Unsecured connections are allowed to connect to this instance.' with a button to 'Allow only SSL connections'. Below this, a note about the server certificate expiration on Feb 5, 2020, is displayed, along with buttons to 'View Server CA Certificate' and 'Reset SSL Configuration'. A section for 'Client Certificates' includes a 'Create a Client Certificate' button.

Figure 4.6 Cloud SQL's SSL options

To get the server's CA certificate, click the aptly named View Server CA Certificate button. You'll see a pop-up appear (figure 4.7), and you can either copy and paste the certificate or download it as server-ca.pem using the link above the text box.

After that, you need to get the client certificate and private key. To do so, click the Create a Client Certificate button and type in a name for your certificate. Typically you'd name the certificate after the server that's using it to access your database. For example, if you'll use this certificate on your production web servers to read and write to the database, you might call it webserver-production (figure 4.8).

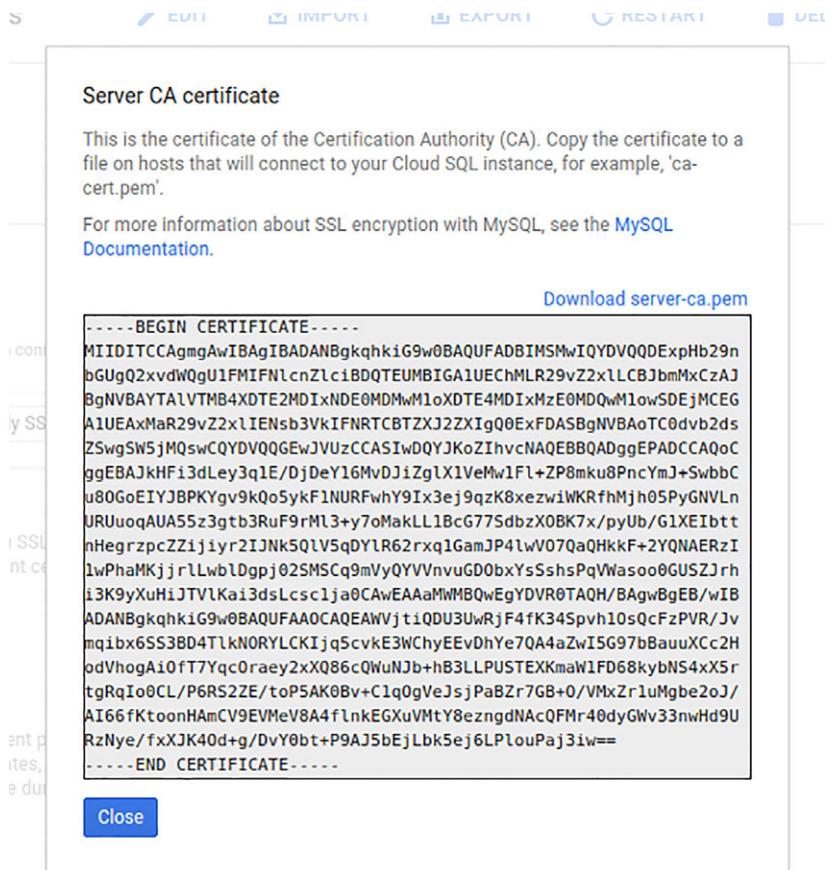


Figure 4.7 Cloud SQL's Server CA Certificate

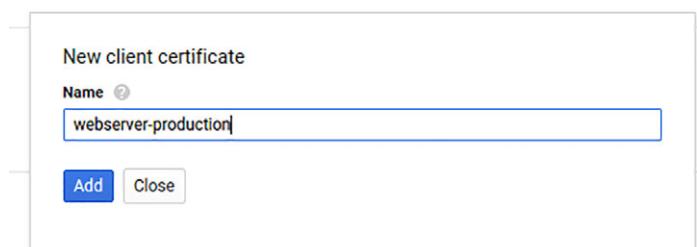


Figure 4.8 Creating a new client certificate

Once you click Add, you'll see a second pop-up showing the client certificate and private key (figure 4.9). As before, you can either copy and paste or click the download links, but at the end of this, you should have both client-key.pem and client-cert.pem.

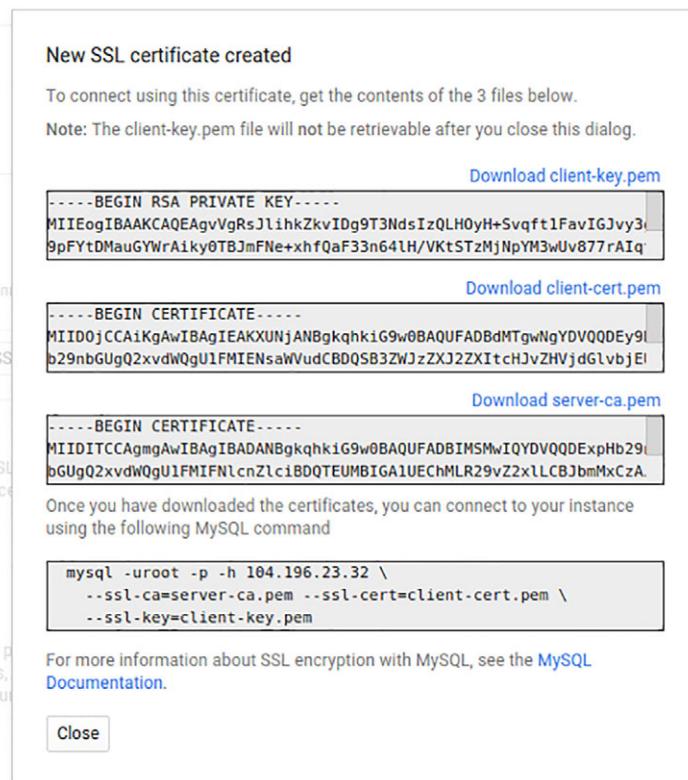


Figure 4.9 Certificate created and ready to use

WARNING Although you can come back later to get server-ca.pem and client-cert.pem files if you lose them, you can't get the client-key.pem file if you lose it. If you do lose it, you'll need to create a new certificate.

Once you have all three files, you can try things out by running the MySQL command provided in the figure 4.9 pop-up:

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

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

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

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

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

mysql>

To double-check that your connection is encrypted, you can use MySQL's SHOW STATUS command, as follows:

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

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

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

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

Listing 4.8 Connecting to MySQL from Node.js

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

const connection = mysql.createConnection({
  host: '104.196.23.32',
  ssl: {
    ca: fs.readFileSync(__dirname + '/server-ca.pem'),
    cert: fs.readFileSync(__dirname + '/client-cert.pem'),
    key: fs.readFileSync(__dirname + '/client-key.pem')
  }
});
```

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

4.3.3 **Maintenance windows**

One area we all tend to forget about during development is the need to upgrade software once in a while. Servers can't live forever without any maintenance, like security patches or upgrades to newer versions, and taking care of those things can be a pain. Luckily, this is one of the things that Cloud SQL handles for you. But you might want to give it some guidance. You might want to tell Google when it's OK to do things like system upgrades, so your customers don't notice the database disappearing or getting slower in the middle of the day.

Cloud SQL lets you set a specific day of the week and time of the day (in one-hour windows) that's an acceptable window for Google to do maintenance. You need to set them because, obviously, Google doesn't know what your business is. The maintenance window is probably different for apps like E*Exchange (where late at night on the weekends is a good time for maintenance) versus apps like InstaSnap (where slightly early morning on weekdays is a good time for maintenance).

To set this window, jump over to the Cloud Console to your Cloud SQL instance's details page, and toward the bottom you'll see a Maintenance Schedule section (figure 4.10) with a link to edit the schedule.

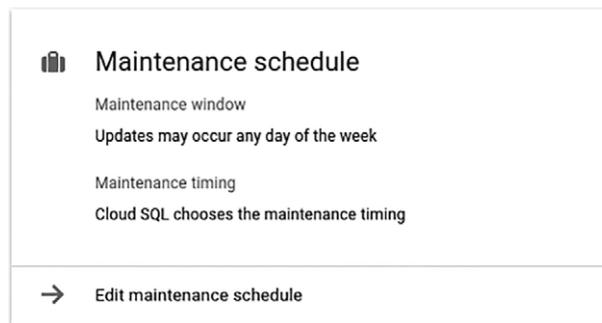


Figure 4.10 Cloud SQL instance details page with a maintenance schedule card

On the editing page (figure 4.11), you'll notice a section called Maintenance Window, which may have been left as Any Window (which tells Google that it's OK to perform maintenance on your Cloud SQL instance at any time on any day); this is unlikely to be what you want!

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

After you pick a day, you can pick a single-hour window that works for you. Keep in mind that this time is in your local time zone, not UTC, so if you're in New York

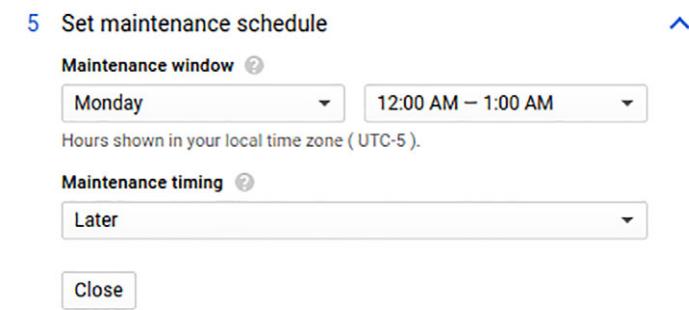


Figure 4.11 Choosing a maintenance window

(as I am), 8:00 a.m. means 8:00 a.m. Eastern time, which is either 12:00 or 13:00 UTC, depending on the time of year. (This difference is due to daylight savings time.)

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

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

The maintenance schedule options let you configure when you want updates, but what about when you want to tweak MySQL's configuration parameters?

4.3.4 Extra MySQL options

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

Tuning MySQL for maximum performance is an enormous topic, so if you're interested in getting the most from your Cloud SQL (or MySQL) database, you may want to pick up a copy of *High Performance MySQL, Third Edition* by Peter Zaitsev, et al (a classic O'Reilly book on the topic). The purpose of this section is to clarify how you'd set all of the parameters on Cloud SQL, as you would on your own MySQL database.

As an example, let's say that you're creating large in-memory temporary tables. By default, there's a limit to how big those tables can be, which is 16 MB. If you end up

going past that limit, MySQL automatically converts that in-memory table to an on-disk MyISAM table. If you know you have more than enough memory (for example, you’re running with 104 GB of RAM) and you’re often going past this limit, you may find that you get better performance by raising the limit from 16 MB to something more in line with your system, say 256 MB.

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

Click the Edit button again on the Cloud SQL instance details page, and choose Add Database Flags from the configuration options section (figure 4.12). In this section, you can choose from a bunch of MySQL configuration flags and set custom values for these options.

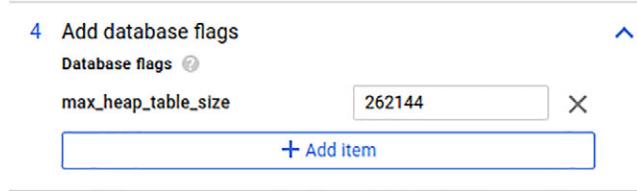


Figure 4.12 Changing the `max_heap_table_size` for your Cloud SQL instance

In your case, you want to change the `max_heap_table_size` to 256 MB (262144 KB). Once you’ve set the value, clicking Save will update the parameter.

You should be able to change almost any of the configuration options you’d see in my.cnf, with a few exceptions related to where your data lives, SSL certificate locations, and other similar things that Cloud SQL manages carefully.

4.4 Scaling up (and down)

In general, there’s nothing wrong with starting out on a small VM type (maybe a single-core VM) and then moving to a larger, more powerful VM later on.

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

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

- Computing power (for example, the VM instance type)
- Disk performance (for example, the size of the disk, because size and performance are tied)

I’ll start by discussing changing the amount of computing power behind your Cloud SQL instance.

4.4.1 Computing power

Go to the Cloud SQL instance details page and click the Edit button at the top. Once you’re there, you’ll notice that you can now change the machine type (figure 4.13). If you started with a single-core machine (db-n1-standard-1), you can change the machine type to a larger machine (for example, db-n1-standard-2) and click Save.

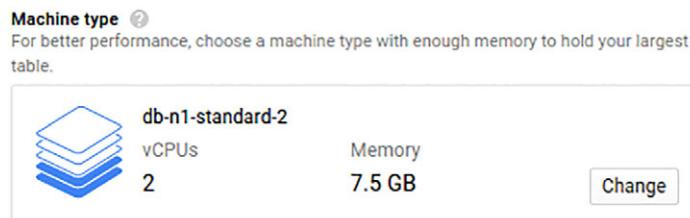


Figure 4.13 Changing the machine type

When you click Save, you’ll have to restart your database (figure 4.14), so there’s a little bit of downtime (typically a few minutes), but that’s all you have to do. When your database comes back up, it’ll be running on the larger (or smaller) machine type.

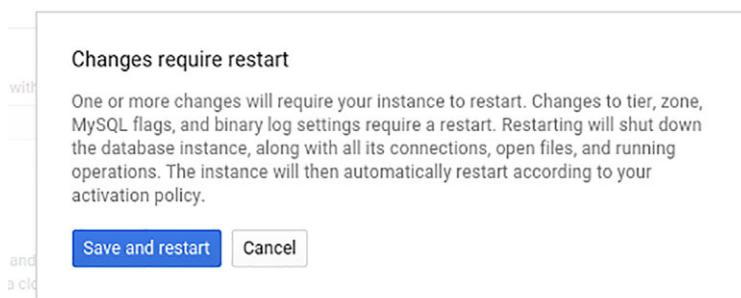


Figure 4.14 Changing the machine type requires a restart.

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

4.4.2 Storage

As you’ll learn about in more detail in chapter 9, disk size and performance are tied together. A larger disk not only can store more bytes, it provides more IOPS to access those bytes. For that reason, if you only plan to have 10 GB of data, but you plan to access it heavily, you might want to allocate far more than 10 GB. You can read all

about this in chapter 9. The key thing to remember here is that you may find yourself in a situation where you’re running low on disk space, or where your data isn’t growing in size, but it’s being accessed more frequently and needs more IOPS capacity. In either situation, the answer’s the same: make your disk bigger.

By default, disks used as part of Cloud SQL have automatic growth enabled. As your disk gets full, Cloud SQL will automatically increase the size available. But if you want to grow a mostly empty disk to increase performance, doing so involves a pretty simple process that once again starts with the Edit button.

On the Edit Instance page, under the Configuration Options, you should see a section called Configure Machine Type and Storage. Inside there, the Storage Capacity section is free for you to change, so increasing the size (and performance) of your disk is as easy as changing the number in the text box to your target size (figure 4.15).



Figure 4.15 Changing the disk size under Storage Capacity

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

Note that you can increase the size of your database, but you can’t decrease it. If you try to make the available storage smaller, regardless of how much space you’ve used, you’ll get an error saying you can’t do that (figure 4.16). Keep that in mind when you change your disk size, as going backwards involves extra work.



Figure 4.16 Disk size can only increase.

This explains how to scale your Cloud SQL instance up and down, but what about high availability? Let’s look at how you can use Cloud SQL to make sure your database stays running even in the face of accidents and other disasters.

4.5 Replication

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

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

A read replica is a clone of your Cloud SQL instance that follows the primary or master instance, pulling in any changes made to the master (figure 4.17). The read replica is strictly read-only, which means that it will reject any queries that modify data (such as `INSERT` or `UPDATE` queries). As a result, read replicas are useful when your application does a lot more reads than writes, because you can turn on a bunch of read replicas and route some of the read-only traffic to those instances. In effect, those instances allow you to scale horizontally (where you add more instances as a way of increasing capacity) rather than only vertically (where you make your machine bigger to increase capacity).

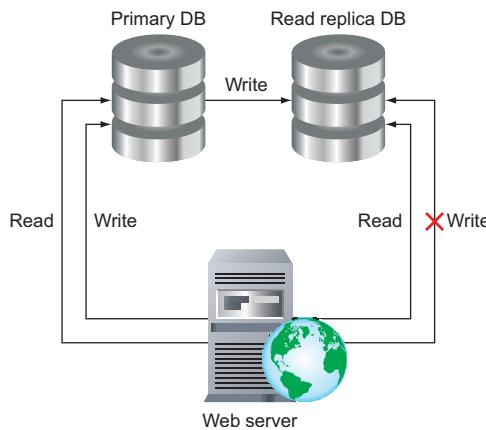


Figure 4.17 Read replicas follow the primary database.

A failover replica is similar to a read replica, except its primary job is to be ready as a replacement primary instance in case of some sort of disaster (figure 4.18). You can think of a failover replica like an alternate on a sports team, ready to replace a player if they are injured.

To create these replicas, all you have to do is click in the Cloud Console. Start first by creating a failover replica.

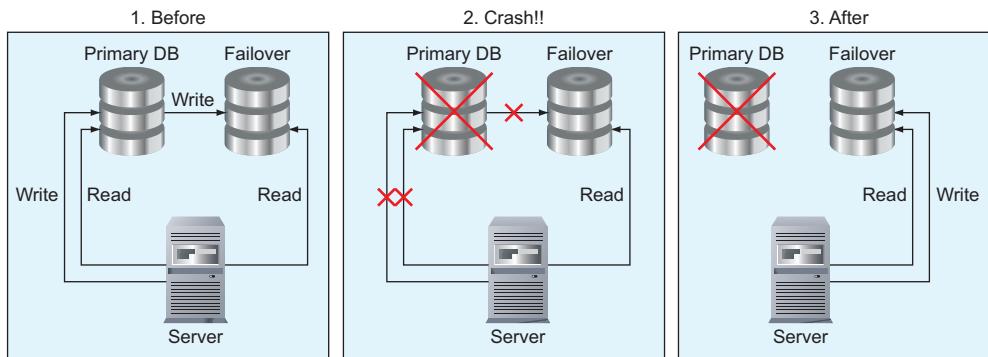


Figure 4.18 Failover replicas step in when the primary database has a problem.

Navigate over to the list of SQL instances, and you should notice a button that says Add Failover (figure 4.19).

Screenshot of the Google Cloud Platform SQL Instances list page. The top navigation bar shows "JJG Cloud Research". Below it, there are tabs for "SQL" and "Instances", with "Instances" selected. A "CREATE INSTANCE" button is visible. The main table lists one instance:

Instance ID	Type	IP address	Storage used	Storage type	Failover	Location	⋮
<input checked="" type="checkbox"/> todo-list	Second Generation	104.196.23.32	1 GB of 50 GB (2.6%)	SSD	Add Failover	us-east1	⋮

Figure 4.19 The list of SQL instances

When you click Add Failover, you'll see a form that looks a lot like creating a new SQL instance—because it is—with one extra option (figure 4.20). Notice that you can choose a different zone within the same region. For example, with the current instance, the region is locked to us-east1, but you can choose a different zone, such as us-east1-b, or leave it as Any, which tells Google you don't care which zone the instance lives in.

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

Machine type ⓘ
For better performance, choose a machine type with enough memory to hold your largest table.

	db-n1-standard-2
vCPUs	Memory
2	7.5 GB
Change	

Network throughput ⓘ
500 of max 15,000 MB/s

Storage type ⓘ
Choice is permanent.
SSD

Storage capacity ⓘ
Failover capacity is fixed at master capacity at the time you added the failover.
Cannot edit.

50	GB
----	----

Enable automatic storage increase
Adds storage capacity whenever space is low. Up to 25 GB per increase. All increases are permanent. [Learn more](#)

Disk throughput ⓘ **IOPS (16KB operations)** ⓘ

Read: 24 MB/s	Max: 240 MB/s	Read: 1,500 IOPS	Max: 4,000 IOPS
			

Write: 24 MB/s	Max: 151.5 MB/s	Write: 1,500 IOPS	Max: 5,000 IOPS
			

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

Anyone (0.0.0.0/0)	
--------------------	---

[+ Add network](#)

[▼ Show advanced options](#)

Create **Cancel**

Figure 4.20 Form for creating a failover replica

In this example, you'll choose us-east1-c for your failover replica and click Create. Once that VM is created, you should see the replica underneath the primary instance in a hierachal representation (figure 4.21).

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

Figure 4.21 The list of SQL instances, including a failover

To create a read replica, the process is similar. In the list of instances, choose Create Read Replica from the contextual menu, as you can see in figure 4.22.

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

⋮

- Create read replica
- Create clone
- Delete

Figure 4.22 The list of SQL instances with the contextual menu

At that point, you can continue as you did with the failover replica, with one important addition: you can use a different instance type! This means that you can create a more powerful (or less powerful) read replica if need be. You also can provide it with a larger disk size, if you suspect that you'll need more disk capacity over time. Then click Create to turn on your read replica. Afterwards, your instance list should look something like figure 4.23.

The screenshot shows the Google Cloud Platform interface for managing SQL instances. At the top, there's a blue header bar with icons for navigation, settings, and search, followed by the text 'JJG Cloud Research'. Below this is a navigation bar with 'SQL' selected and 'Instances' and 'CREATE INSTANCE' buttons. The main area is a table listing three instances:

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

Figure 4.23 The list of SQL instances, including both types of replicas

4.5.1 Replica-specific operations

In addition to the typical operations you can do on a Cloud SQL instance (for example, restart it, edit it, and so on), a couple of operations are only possible with read replicas: promoting and disabling replication. Disabling replication does exactly what it says it does: it pauses the stream of data between the primary and the replica, effectively freezing the database as it is in the moment that replication is disabled. This can be handy if you're worried about a bug that might change your replica inadvertently or if you want to freeze the data in a certain way for development. If you choose to re-enable replication, the replica will resume pulling data from the primary instance and eventually come into sync with it.

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

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

4.6 Backup and restore

When I talk about backups in the planning stages, most people's eyes gloss over, but when disaster strikes, suddenly their attitude changes entirely. Cloud SQL does a solid job of making backups simple so that you don't have to think about them until you need them. Lots of different backup methods are available, but let's start by looking at the simplest: automated daily backups.

4.6.1 Automated daily backups

The simplest, quickest, and probably most useful backup for Cloud SQL is the automatic one that occurs daily at a time you specify when you create the Cloud SQL instance. Although you can disable this backup (for example, if you’re running a test database), it’s probably a bad idea to turn it off for anything that stores data you care at all about.

To set this, all you have to do is choose a backup window when creating your Cloud SQL instance (figure 4.24). (You can always change this setting later on.)

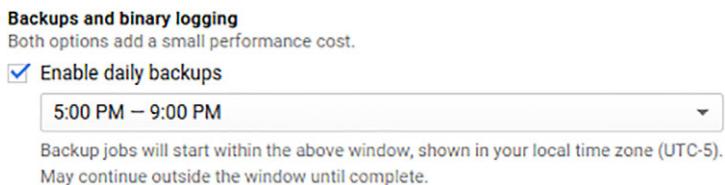


Figure 4.24 Setting the automated backup window

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

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

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

Coming full circle, restoring backups involves a simple single command, using the due time as the unique identifier for which backup to restore from. The following snippet shows how you might restore your database to a previous backup:

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

$ gcloud sql instances restore-backup todo-list
➡ --due-time=2016-01-15T16:19:00.094Z
```

```
Restoring Cloud SQL instance...done.  
Restored [https://www.googleapis.com/sql/v1beta3/projects/your-project-id-  
here/instances/todo-list].
```

WARNING If your instance has replicas attached (for example, read replicas or failover replicas), you must delete them before restoring from a backup.

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

4.6.2 Manual data export to Cloud Storage

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

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

To get started, go to the instance details page for your Cloud SQL instance, and click the Export button at the top of the page. This will present you with a dialog box where you can set some options for the data export (figure 4.25).

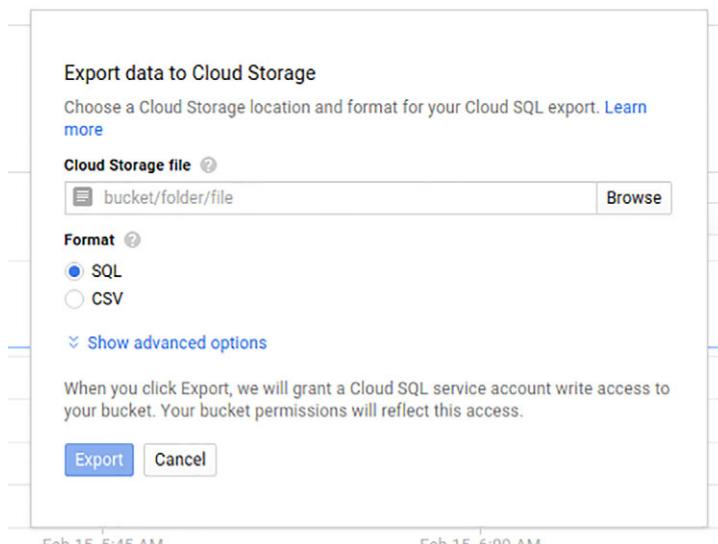


Figure 4.25 The data export configuration dialog box

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

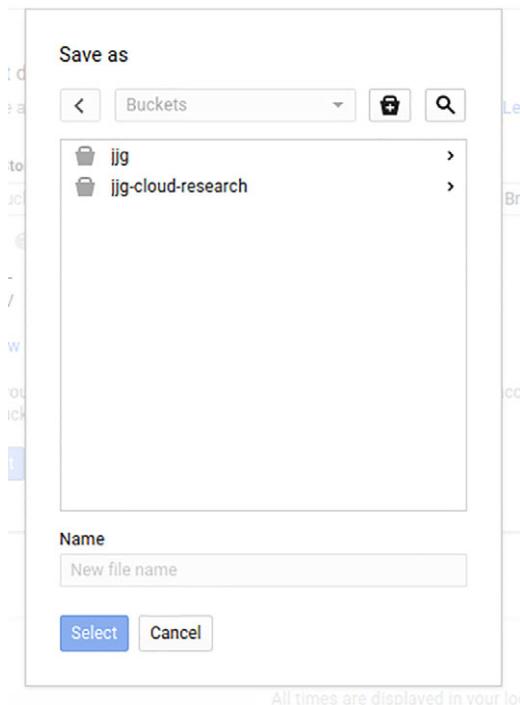


Figure 4.26 Dialog box for choosing a location for your export

Click the Browse button next to the field for the file path, and at the top of the new dialog that opens up (figure 4.26), you should see a small icon that looks like a bucket with a plus sign in the center. When you click this, you'll see a dialog where you can choose the Name for your bucket, as well as the Storage Class and Location (figure 4.27). I go through the differences between all of the storage classes later on, but in general, backups are a good fit for the Nearline storage class, as it's less expensive for infrequently accessed data.

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

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

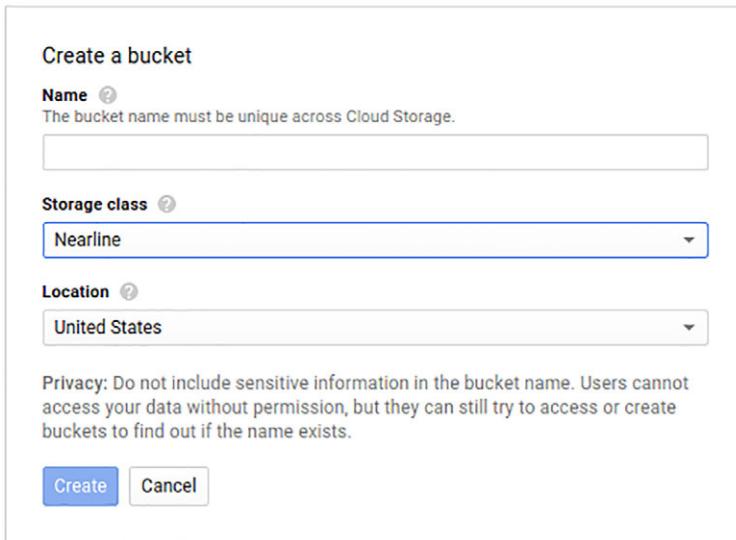


Figure 4.27 Dialog box for creating a bucket

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

Lastly, if you plan to use your data export as a complete backup (you intend to revert to the data stored exactly as it is in the export), make sure to choose the SQL format (not CSV), which includes all of your table definitions along with your schema, rather than the data alone. With an export in SQL format, the output is the SQL statements required to bring the database into the state that exists when the export is executed.

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

Once you click Select, you'll be brought back to the export dialog, which should show your export path with a green check mark next to it (figure 4.28). Click Export to start things off.

This could take a few minutes, depending on how much data is in your Cloud SQL instance, but you can check on the status by clicking the Operations tab on the instance details page. When the operation is complete, you'll see a row confirming that the export succeeded (figure 4.29).

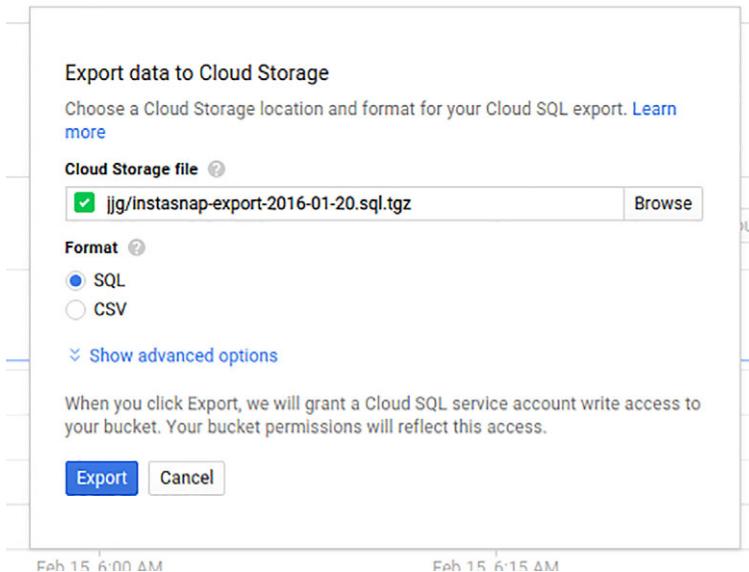


Figure 4.28 Dialog box for Export Data to Cloud Storage

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

Figure 4.29 The Operations list showing the successful export

To confirm that your export worked, you can open your bucket in the Cloud Storage browser (figure 4.30). If you browse to your bucket, you'll see the export available there, along with its size and other details.

Now that you have an export on Cloud Storage, let's walk through how to restore it into your Cloud SQL instance. Start by clicking Import on the instance details page, and you should see a dialog that looks similar to the one you used when creating the data export (figure 4.31). From there, browse to the export file that you created, click Import, and you're all done.

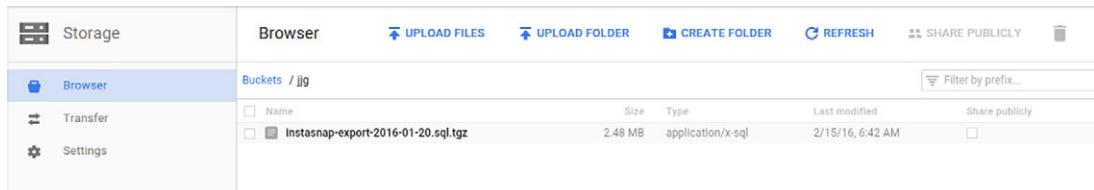


Figure 4.30 Your export will be visible in the Cloud Storage browser.

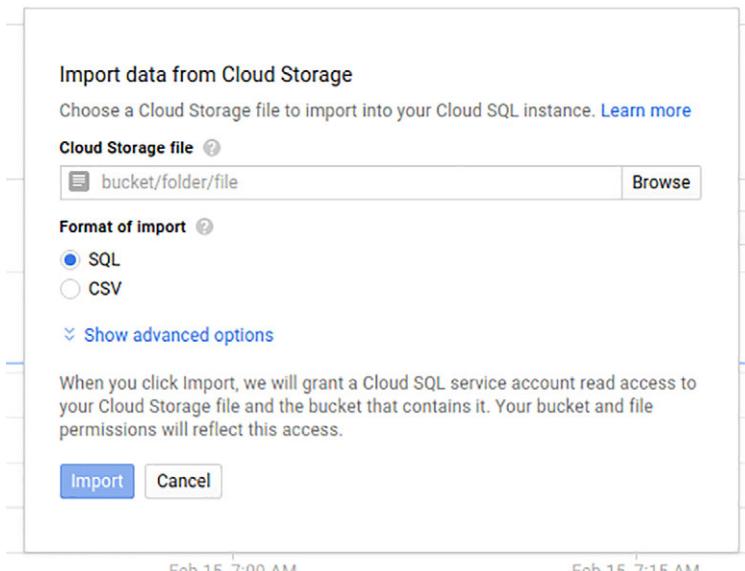


Figure 4.31 The data import dialog box

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

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

4.7 Understanding pricing

As you read in chapter 1, Google Cloud considers two basic principles of pricing for computing resources: computing time and storage. The prices for Cloud SQL follow

these same principles, with a slight markup on CPU time for managing the MySQL binary and configuration for you.

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

To make this clearer, take a look at the comparison in table 4.3. This comparison doesn't include all of the different configurations for Cloud SQL instances, but it covers a representative spectrum of the more common options.

Table 4.3 Different sizes of Cloud SQL instances and costs

ID	CPU Cores	Memory	Hourly price	Monthly price	Effective hourly price
g1-small	1	1.70 GB	\$0.0500	\$25.20	\$0.0350
n1-standard-1	1	3.75 GB	\$0.0965	\$48.67	\$0.0676
n1-standard-16	16	60 GB	\$1.5445	\$778.32	\$1.0810
n1-highmem-16	16	104 GB	\$2.0120	\$1,014.05	\$1.4084

You may be wondering how these numbers compare to running your own VM option discussed earlier. Let's start by looking at a comparison of these two options (table 4.4), focusing exclusively on the cost of computing power rather than storage, because it's the same for both options. Also, let's assume you'll run your database for a full month—that'll make the numbers a bit easier to relate to.

Table 4.4 Cloud SQL vs Compute Engine monthly cost

ID	CPU Cores	Memory	Cloud SQL	Compute Engine	Extra cost
g1-small	1	1.70 GB	\$25.20	\$13.68	\$11.52
n1-standard-1	1	3.75 GB	\$48.67	\$25.20	\$23.47
n1-standard-16	16	60 GB	\$778.32	\$403.20	\$375.12
n1-highmem-16	16	104 GB	\$1,104.05	\$506.88	\$597.17

As you can see, because the cost of Cloud SQL is directly proportional to the hourly cost, as you scale up to larger and larger VM types, your absolute cost difference grows. Although this might not mean much for smaller-scale deployments (\$13 dollars versus \$11 dollars isn't a big deal), it starts to become a bigger deal as you add more and more machines. For example, if you were running 20 machines of the largest

type in your table, you'd be paying \$12,000 in extra cost for your Cloud SQL instances *every month!* That's \$144,000 annually, which means you may be better off hiring someone to manage your databases and switching to Compute Engine VMs.

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

4.8 When should I use Cloud SQL?

Before you decide whether Cloud SQL is a good fit, let's look at a summary of Cloud SQL using the scorecard in figure 4.32. Keep in mind that because Cloud SQL is almost the same thing as MySQL, this scorecard is identical to the one for running your own MySQL server on a virtual machine in a cloud service like Compute Engine or Amazon's EC2, or using Amazon's RDS mentioned earlier.

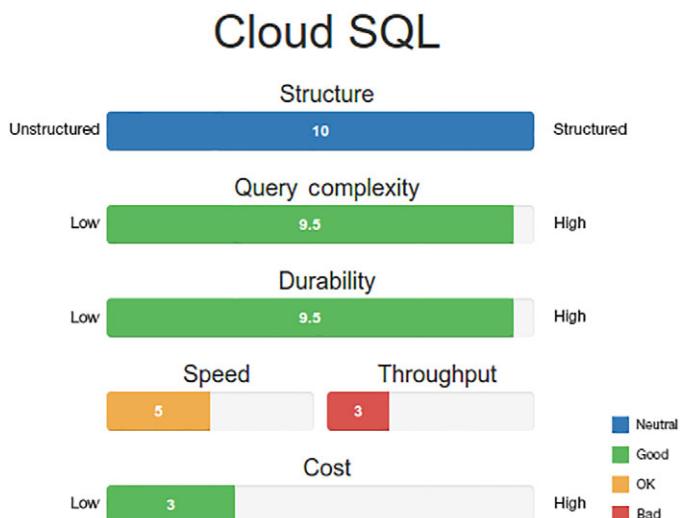


Figure 4.32 Scorecard for Cloud SQL

As you may have noticed, this scorecard presents a few interesting things. Let's go through it point by point to understand why the scores came out this way.

4.8.1 Structure

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

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

4.8.2 **Query complexity**

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

4.8.3 **Durability**

Durability is another area where relational databases shine. If you're looking for something that *really* means it when it says, "I saved your data to disk," relational databases are a great choice. Although you should still dig deep on tuning MySQL for the level of durability you need, the general consensus is that relational storage systems (like MySQL) are capable of providing a high level of durability. Furthermore, because Cloud SQL runs on top of Compute Engine and stores all the data on Persistent Disk, you benefit from the higher levels of durability and availability that Persistent Disk offers. For more details on Persistent Disk, check out chapter 9.

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

4.8.4 **Speed (latency)**

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

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

4.8.5 **Throughput**

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

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

4.9 Cost

As we learned before in the section on pricing, Cloud SQL uses Compute Engine under the hood and follows a similar cost pattern. Cloud SQL's costs also are on the same level as running any database yourself on Compute Engine (such as your own MySQL instance), with a bit of overhead for the automatic maintenance and management that Cloud SQL provides. As a result, Cloud SQL comes in very low on the cost scale for data sets that are suitable for a MySQL database. For larger data sets that require significantly more computing power, you may want to explore running your own MySQL cluster on Compute Engine machines and using the cost savings to hire a full-time administrator.

4.9.1 Overall

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

To-Do List

As you'll recall, the To-Do List application was intended as a good starter app for learning new systems. Let's go through the various aspects of this application and see how it lines up with Cloud SQL as a possible storage option. See table 4.5.

Table 4.5 To-Do List application storage needs

Aspect	Needs	Good fit?
Structure	Structure is fine; not necessary, though.	Sure
Query complexity	We don't have that many fancy queries.	Definitely
Durability	High—We don't want to lose stuff.	Definitely
Speed	Not a lot.	Definitely
Throughput	Not a lot.	Definitely
Cost	Lower is better for toy projects.	Mostly

Based on table 4.5, it seems like Cloud SQL is a pretty good option for the To-Do List database. What about something more complicated, like E*Exchange?

E*EXCHANGE

E*Exchange was an online trading platform where people could buy and sell stocks with the click of a button. Let's look through the list and see how Cloud SQL stacks up against the requirements for this application. See table 4.6.

Table 4.6 E*Exchange storage needs

Aspect	Needs	Good fit?
Structure	Yes, reject anything suspect; no mistakes.	Definitely
Query complexity	Complex—We have fancy questions to answer.	Definitely
Durability	High—We can't lose stuff.	Sure
Speed	Things should be pretty fast.	Probably
Throughput	High—Lots of people may be using this.	Maybe
Cost	Lower is better, but willing to pay top dollar.	Definitely

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

Additionally, this example assumes that the only data being stored here is customer data, such as balances, bank account information, and portfolios. Trading data, which is likely to be much larger than the customer data, wouldn't be well suited for relational storage, but instead would fit better in some sort of data warehouse. We'll explore some options for this type of data in chapter 19, where I discuss large-scale analytics using BigQuery.

Although Cloud SQL might be a good place to start if E*Exchange had moderate amounts of data, if that data grew into tens to hundreds of gigabytes, the company might have to migrate to a different storage system or risk frustrating its customers with downtime or slow-loading pages.

INSTASNAP

InstaSnap was a super high-traffic application that caught on with celebrities all over the world—meaning lots of concurrent requests. As I mentioned, that aspect alone would be likely to disqualify something like Cloud SQL from the list of possibilities, but let's run through the scorecard. See table 4.7.

Table 4.7 InstaSnap storage needs

Aspect	Needs	Good fit?
Structure	Not really—Structure is pretty flexible.	Not really
Query complexity	Mostly lookups; no highly complex questions.	Not really
Durability	Medium—Losing things is inconvenient.	Sure
Speed	Queries must be very fast.	Not really (with lots of data)

Table 4.7 InstaSnap storage needs (continued)

Aspect	Needs	Good fit?
Throughput	Very high—Kim Kardashian uses this.	Not really
Cost	Lower is better, but willing to pay top dollar.	Definitely

It looks like Cloud SQL is a bad fit for something of this scale, particularly when the most valuable features of a relational storage system like MySQL aren't even necessary. For a product like InstaSnap, the structure of the data isn't that important, nor are the durability and transactional semantics. In a sense, if you used Cloud SQL, you would sacrifice the high performance that you desperately need in exchange for transactions, high durability, and high consistency that you don't care that much about. Cloud SQL isn't a great fit for something like InstaSnap, so if your needs are similar to InstaSnap's, consider some of the other storage options I'll present.

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

4.10 Weighing Cloud SQL against a VM running MySQL

Google built Cloud SQL with a specific target audience in mind: people who just want MySQL and don't care all that much about customizing their instance. If you were only planning to turn on a VM, install MySQL, and change the password, Cloud SQL was made for you.

As I discussed in chapter 1, one of the primary motivations for shifting toward the cloud was to reduce your overall TCO (total cost of ownership). Cloud SQL does this not necessarily by reducing the cost of the hardware, but by reducing your maintenance and management costs. For example, if you were running your own VM running MySQL, you'd need to find the time to upgrade your operating system and MySQL version for any new security patches that happen to come out (or accept the risk of your data being compromised, but I'll assume you'd never do that).

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

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

Sometimes you'll have more specific requirements for your database, and in those situations, you may end up needing more flexibility than Cloud SQL can provide. The most common scenario is requiring a different relational database, such as PostgreSQL or Microsoft's SQL Server. Right now, Cloud SQL only handles MySQL, so if you need any other relational database flavor, Cloud SQL isn't a good fit. Although MySQL is a reasonable choice, other database systems have some impressive features (such as

PostgreSQL 9.5’s native JSON type support), and if you want or need those features for whatever reason, the better fit is likely to be running your database on a VM and managing it yourself.

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

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

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

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

You may be thinking now, “This is all great, but how much will this cost me?” Let’s dig into that.

Summary

- Relational databases are great for storing data that relates to other data using foreign key references, such as a customer database.
- Cloud SQL is MySQL in a box that runs on top of Compute Engine.
- When choosing your storage capacity, don’t forget that size is directly related to performance. It’s OK (and expected) to have lots of empty space.
- When you have enough Cloud SQL instances to justify hiring a DBA, it might make sense to manage MySQL yourself on Compute Engine instances.
- Always configure Cloud SQL to encrypt traffic using an SSL certificate to avoid eavesdropping on the internet.
- Don’t worry if you chose too slow of a VM. You can always change the computing power later. You also can increase the storage space, but it’s more work to decrease it if you overshoot.
- Use failover replicas if you want your system to be up even when a zone goes down.
- Enable daily backups if you want to be sure to never lose data.

Cloud Datastore: document storage

This chapter covers

- What's document storage?
- What's Cloud Datastore?
- Interacting with Cloud Datastore
- Deciding whether Cloud Datastore is a good fit
- Key distinctions between hosted and managed services

Document storage is a form of nonrelational storage that happens to be different conceptually from the relational databases discussed in chapter 4. With this type of storage, rather than thinking of tables containing rows and keeping all of your data in a rectangular grid, a document database thinks in terms of collections and documents. These *documents* are arbitrary sets of key-value pairs, and the only thing they must have in common is the document type, which matches up with the collection. For example, in a document database, you might have an Employees collection, which might contain two documents:

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

Comparing this to a traditional table of similar data (table 5.1), you'll see that the grid format will look quite different from a document collection's jagged format (table 5.2).

Table 5.1 Grid of employee records

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

Table 5.2 Jagged collection of employees

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

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

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

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

The reason for this omission will vary from system to system, but one reason is that a *missing* property isn't the same thing as a property with a null value, so the only documents considered are those that explicitly have a key called `favoriteColor`. You might be wondering, where did behavior like this come from?

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

5.1

What's Cloud Datastore?

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

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

5.1.1 Design goals for Cloud Datastore

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

DATA LOCALITY

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

RESULT-SET QUERY SCALE

Another requirement with this database is that it'd be frustrating if your inbox got slower as you receive more email. To deal with this, you'd probably want to index emails as they arrive so that when you want to search your inbox, the time it takes to run any query (for example, searching for specific emails or listing the last 10 messages to arrive) would be proportional only to the number of *matching* emails (not the total number of emails).

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

AUTOMATIC REPLICATION

Finally, you have to worry about the fact that sometimes servers die, disks fail, and networks go down. To make sure that people can always access their email, you need to put email data in lots of places so it's always available. Any data written should be replicated automatically to many physical servers. That way, your email is never on a single computer with a single hard drive. Instead, each email is distributed across lots of places. This distribution can be difficult to achieve if you start from traditional database software, but Google's underlying storage systems are well suited to this requirement, and Cloud Datastore takes care of it.

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

5.1.2 Concepts

You learned a little bit about how document storage is pretty different from relational storage, but I didn't dive into the specifics of Cloud Datastore's take on these differences. Let's look at the important pieces, and I'll discuss how they fit together.

KEYS

The most primitive concept to learn first is the idea of a *key*, which is what Cloud Datastore uses to represent a unique identifier for anything that has been stored. The closest thing to compare this to in the relational database world is the unique ID you often see as the first column in tables, but Datastore keys have two major differences from table IDs.

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

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

The second major difference is that keys themselves can be hierarchical, which is a feature of the concept of data locality I mentioned before. Your keys can have *parent keys*, which colocate your data, effectively saying, "Put me near my parent." An example of a nested (or hierarchical) key would be `Employee:1:Employee:2`, which is a pointer to employee #2.

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

This gets tricky when you realize that there isn't always a great reason for nested keys of the same kind, but instead you might want to nest subentities inside each other. Such nesting is perfectly acceptable, because keys can refer to multiple kinds in their path or the hierarchy, and the kind (type) of the data is the kind of the bottom-most piece.

For example, you might want to store your employee records as children of the company they work for, which could be `Company:1:Employee:2`. The kind of this key is `Employee`, and the parent key is `Company:1` (whose kind is `Company`). This key refers to employee #2, and because of its parent (`Company:1`), it'll be stored near all other employees of the same company; for example, `Company:1:Employee:44` will be nearby.

Also note that although you've only seen numerical IDs in the examples, you also can specify keys as strings, such as `Company:1:Employee:jbond` or `Company:apple.com:Employee:stevejobs`.

ENTITIES

The primary storage concept in Cloud Datastore is an *entity*, which is Datastore's take on a document. From a technical perspective, an entity is nothing more than a collection of properties and values combined with a unique identifier called a *key*.

An entity can have properties of all the basics, also known as *primitives*, such as

- Booleans (true or false)
- Strings ("James Bond")
- Integers (14)
- Floating-point numbers (3.4)
- Dates or times (2013-05-14T00:01:00.234Z)
- Binary data (0x0401)

Here's an example entity with only primitive types:

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

In addition to the basic types, Datastore exposes some more advanced types, such as

- Lists, which allow you to have a list of strings
- Keys, which point to other entities
- Embedded entities, which act as subentities

The following example entity includes more advanced types:

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

The team property is an embedded entity, which itself could be structured like any other entity stored in Datastore.

The manager property is a key that points to another entity, which is as close to a foreign key as you can get.

The groups property is a list of strings, but could easily be a list of integers, keys, and so on.

This configuration has a few unique properties:

- A reference to another key is as close as you can get to the concept of foreign keys in relational databases.
- There's no way to enforce that a reference is valid, so you have to keep references up to date; for example, if you delete the key, update the reference.

- Lists of values typically aren't supported in relational databases, which typically use pivot tables to store a *has many* relationship. In Datastore, a list of primitives is the natural way to express this.
- In relational databases, you typically use a foreign key to store other structured data. In Datastore, if that structured data doesn't need its own row in a table, you can embed that data directly inside another entity using *embedded entities*. Embedded entities are useful. In some ways they're like anonymous functions in JavaScript, where you've put the contents of the function inline rather than naming them as a function and calling them by name.

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

OPERATIONS

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

- **get**—Retrieve an entity by its key.
- **put**—Save or update an entity by its key.
- **delete**—Delete an entity by its key.

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

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

INDEXES AND QUERIES

Now that you have a handle on the fundamentals of document storage, I need to discuss the two concepts that pull it all together: *indexes* and *queries*. In a typical database, a query is nothing more than a SQL statement, such as `SELECT * FROM employees`. In Datastore, this is possible using GQL (a query language much like SQL). A more structured way of representing a query is also available, and you'll learn about that in section 5.3. What's interesting, though, is that although Datastore may look like it can speak SQL, there are quite a few queries that Datastore can't answer. Furthermore, relational databases tend to treat indexes as a way of *optimizing* a query, whereas Datastore uses indexes to make a query possible (table 5.3).

Table 5.3 Queries and indexes, relational vs Datastore

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

So what's an index? And what type of queries go from impossible to possible with an index? You may find the answer surprising. Anytime you're filtering (for example, using a `WHERE` clause) in your query, you're relying on an index, which is there to ensure that the query scales with the result set.

Imagine if every time you needed to find all emails from Steve (`steve@apple.com`), you had to go through all of your emails, checking each one's `sender` property looking for "Steve". This clearly would work, but it means that the more email you get, the longer this query takes to run, which is obviously bad. The way you fix this problem is by creating an index that stays up to date whenever information changes and that you can scan through to find matching emails. An index is nothing more than a specially ordered and maintained data set to make querying fast. For example, with your email, an index over the `sender` field might look like table 5.4.

Table 5.4 An index over the `sender` field

Sender	Key
<code>eric@google.com</code>	<code>GmailAccount:me@gmail.com:Email:8495</code>
<code>steve@apple.com</code>	<code>GmailAccount:me@gmail.com:Email:2441</code>

This index pulls out the `sender` field from emails and allows you to query over all emails with a certain `sender` value. It also provides you with a guarantee that when the query finishes, all matching results have been found. The query for all emails from Steve (`SELECT * FROM Email WHERE sender = 'steve@apple.com'`) relies on the index to find the first entry that matches; then it continues scanning until it finds an entry that doesn't match (`tom@example.com`). As you can see, the more emails from Steve, the longer this query takes, but emails from other people (which *don't* match the query you're running) have no affect at all on how long this query takes to run.

This raises the obvious question: Do I have to create an index to do a simple filtering query? Luckily, no! Datastore automatically creates an index for each property (called simple indexes) so that those simple queries are possible by default. But if you want to do matching on multiple properties together, you may need to create an index. For example, finding all email from Steve where Eric is cc'd might require an index that looks like the following listing:

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

To make sure this query scales with the result set (of matching emails), you'd need an index on both `sender` and `cc` that might look like table 5.5.

Table 5.5 An index over the sender and cc fields

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

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

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

5.1.3 Consistency and replication

As you learned earlier, a distributed storage system for something like Gmail needs to meet two key requirements: to be always available and to scale with the result set. This means that not only does data need to be replicated, but you also need to create and maintain indexes for your queries.

Data replication, though complicated to implement, is somewhat of a solved problem, with many protocols around, each with their own trade-offs. One protocol that Cloud Datastore happens to use involves something called a two-phase commit.

In this method, you break the changes you want saved into two phases: a preparation phase and a commit phase. In the preparation phase, you send a request to a set of replicas, describing a change and asking the replicas to get ready to apply it. Once all of the replicas confirm that they've prepared the change, you send a second request instructing all replicas to apply that change. In the case of Datastore, this second (commit) phase is done asynchronously, where some of those changes may hang around in the prepared but not yet applied state.

This arrangement leads to eventual consistency when running broad queries where the entity or the index entry may be out of date. Any strongly consistent query (for example, a get of an entity) will first push a replica to execute any pending commits of the resource and then run the query, resulting in a strongly consistent result.

As you can see, maintaining entities and indexes in a distributed system is a much more complicated task, because the same save operation also would need to include

the saves to any indexes that the change affects. (And remember that the indexes need to be replicated, so they need to be updated in multiple places as well.)

This means that Datastore would have two options:

- Update the entity and the indexes everywhere synchronously, confirming the operation will take an unreasonably long time, particularly as you create more indexes.
- Update the entity itself and the indexes in the background, keeping request latency much lower because there's no need to wait for a confirmation from all replicas.

As mentioned, Datastore chose to update data asynchronously to make sure that no matter how many indexes you add, the time it takes to save an entity is the same. As a result, when you use the put operation, under the hood Datastore will do quite a bit of work (figure 5.1):

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

And then later, whenever a strongly consistent query runs:

- Ensure all pending changes to the affected entity group are applied.
- Execute the query.

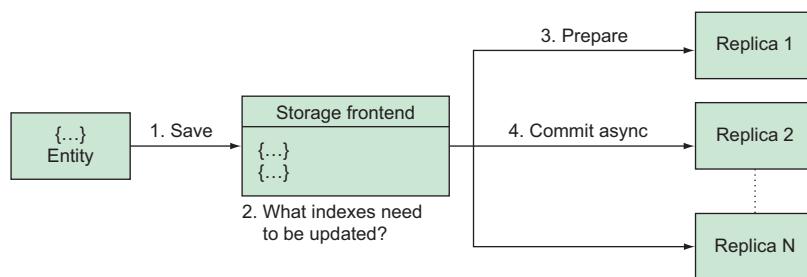


Figure 5.1 Saving an entity in Cloud Datastore

It also means that when you run a query, Datastore uses these indexes to make sure your query runs in time that's proportional to the number of matching results found. This means that a query will do the following (figure 5.2):

- Send the query to Datastore.
- Search the indexes for matching keys.
- For each matching result, get the entity by its key in an eventually consistent way.
- Return the matching entities.

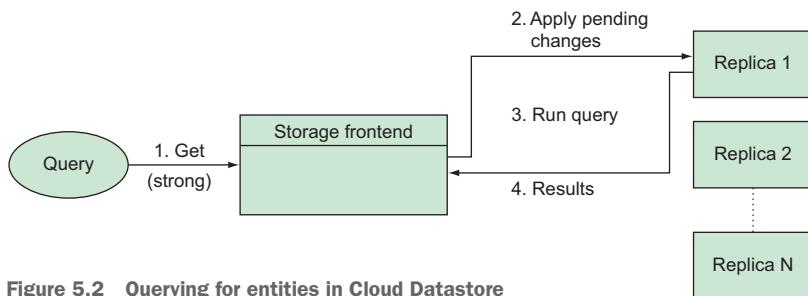


Figure 5.2 Querying for entities in Cloud Datastore

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

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

For example, imagine that you've just added a new Employee entity to Cloud Datastore, as shown in the following listing.

Listing 5.1 Example Employee entity

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

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

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

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

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

Your queries are eventually consistent, specifically because the indexes that Datastore uses to find those entities are updated in the background. Note that this also applies when your entities are *modified*. For example, imagine that the indexes have reached a level of consistency, and when you look for all employees with blue as their favorite color, Employee 1 is returned.

Now imagine that you change this employee's favorite color, as follows.

Listing 5.2 Employee entity with a different favorite color

```
{
  "_key": "Employee:1",
  "name": "James Bond",
  "favoriteColor": "red"
}
```

If you run your query again, depending on which updates have been committed, you may see different results, described in table 5.6.

Table 5.6 Summary of the different possible results

Entity updates	Index updated	Employee matches	Favorite color
Yes	Yes	No	Doesn't matter
No	Yes	No	Doesn't matter
Yes	No	Yes	red
No	No	Yes	blue

In short, the three possibilities are

- The employee won't be in the results.
- The query still sees the employee as matching the query (`favoriteColor = blue`), and correctly so, so it ends up in the results.
- The query still sees the employee as matching the query (`favoriteColor = blue`), so it ends up in the results, even though the entity doesn't actually match! (`favoriteColor = red`)

This must seem strange for anyone working day to day with a SQL database. You may also be asking yourself, “How on earth can you build something with this?”

It's important to remember that systems like this were designed with services like Gmail in mind, which have different requirements than a typical SQL-backed web application. So how does this type of system benefit customers like Gmail? This brings us to the next big topic: combining querying with data locality to get strong consistency.

5.1.4 Consistency with data locality

I talked earlier about data locality as a tool for putting many pieces of data near each other (for example, you group all of a single account's emails close together), but I didn't clarify why that might matter.

Now that you understand the concept of eventual consistency (that your queries run over indexes rather than your data, and those indexes are eventually updated in

the background), you can combine it with the concept of data locality so you can build real things that will enable you to query without wondering whether the data is accurate.

Let's start with a hugely important fact: queries inside a single entity group are *strongly consistent* (not *eventually consistent*). If you recall, an *entity group*, defined by keys sharing the same parent key, is how you tell Datastore to put entities near each other. If you want to query over a bunch of entities that all have the same parent key, your query will be *strongly consistent*.

Telling Datastore where you want to query over in terms of the locality gives it a specific range of keys to consider. It needs to make sure that any pending operations in that range of keys are fully committed prior to executing the query, resulting in strong consistency. If you ask Datastore for all Apple employees who have blue as their favorite color, for example, it knows exactly which keys could be in the result set, and before executing the query it can first make sure no operations involving those keys are pending. That means the results will always be up to date.

The following listing goes back to the previous example with Apple employees.

Listing 5.3 Apple employee with favorite color of blue

```
{
  "__key__": "Company:apple.com:Employee:jonyive",
  "name": "Jony Ive",
  "favoriteColor": "blue"
}
```

Now let's change Jony's favorite color, as follows.

Listing 5.4 Updating the favorite color to red

```
{
  "__key__": "Company:apple.com:Employee:jonyive",
  "name": "Jony Ive",
  "favoriteColor": "red"
}
```

As you learned before, running a query across all employees may not accurately reflect your data, but if you query over all Apple employees, you're guaranteed to get the latest data:

```
SELECT * FROM Employees WHERE favoriteColor = "blue" AND
__key__ HAS ANCESTOR Key(Company, 'apple.com')
```

Because this query is limited to a single entity group, the results will always be consistent with the data, which is referred to as being strongly consistent. This begs the obvious question: Why don't I just put everything in a single entity group? Won't I always have strong consistency then?

Although technically true, that doesn't make it a good idea. The reason for this is that a single entity group can only handle a certain number of requests simultaneously—in the range of about 10 per second. If you put everything in one entity group, you'd be trading off eventual consistency and getting pretty low throughput overall in return. If you value strong consistency enough that you'd be willing to throw away the scalability of Datastore, you should probably be using a regular relational database instead.

Now that you have some idea of how Cloud Datastore works, let's kick the tires a bit to see what it's like to use it in your app.

5.2 Interacting with Cloud Datastore

Before you can use Cloud Datastore, you may need to enable it in the Cloud Console. To do so, start by searching for “Cloud Datastore API” in the Cloud Console main search box, which should yield only one result. Click that to get to a page that should have a big Enable button (figure 5.3). (If you only see the ability to Disable the API, you're already set.)

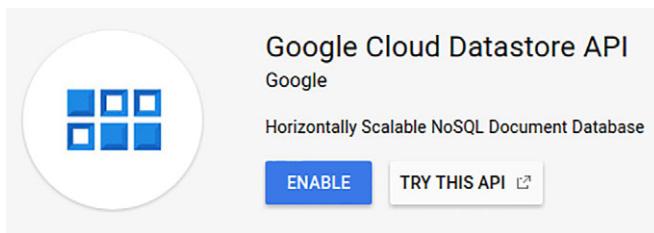


Figure 5.3 Dialog box for enabling the Cloud Datastore API

Once you've enabled the API, jump to the Datastore UI from the left navigation. Then we'll go back to the To-Do List example and explore how it might look in Cloud Datastore.

You'll start by creating the `TodoList` entity. Notice that, unlike with a SQL database, you'll first create some data, rather than defining a schema. This is the nature of document-oriented databases, and although it might seem strange at first, it's typical for nonrelational storage. You should see a big, blue Create Entity button when you first visit the Datastore page, so start by clicking that.

Next, as shown in figure 5.4, leave your entity in the `[default]` namespace (I'll discuss namespaces a bit later), make it a `TodoList` kind, and let Datastore automatically assign a numerical ID. After that, give your `TodoList` entity a name. To do so, click the Add Property button, set the name of the property to `name`, leave the property type set to `String`, and fill in the value of the property (in this case, the name of the list). In this example, the list is called `Groceries`. Also note that because you may

The screenshot shows the Google Cloud Platform Datastore interface. At the top, there are navigation icons and a search bar. Below that, the title 'Datastore' and the action 'Create entity' are displayed. The form fields are as follows:

- Namespace:** [default]
- Kind:** TodoList
- Key identifier:** Numeric ID (auto-generated)
- Properties:**

Name	Type	Value	Indexed
name	String	Groceries	<input checked="" type="checkbox"/>
- Add property:** + Add property
- Buttons:** Save (blue), Cancel

Figure 5.4 Creating the Groceries TodoList

want to search based on this name, you'll leave the property indexed (marked by the check box).

Click Save, and you should see a newly created TodoList entity in your browser (figure 5.5).

Let's take a moment now and look at how to interact with this entity in your own code. If you followed the tutorial in chapter 1, you should already have all the right

The screenshot shows the Google Cloud Platform Datastore interface with the 'Entities' tab selected. The main area displays the following information:

- Entities:** A table header with columns for Entity ID, Kind, and Status.
- CREATE ENTITY:** A button to create a new entity.
- REFRESH:** A button to refresh the list.
- DELETE:** A button to delete selected entities.
- Dashboard:** A link to the dashboard.
- Entities:** The currently selected link in the sidebar.
- Indexes:** A link to indexes.
- Admin:** A link to admin tools.
- Query by kind:** A dropdown set to 'TodoList'.
- Query by GQL:** A text input field.
- Kind:** A dropdown set to 'TodoList'.
- Filter entities:** A button to filter entities.
- Results:** A table showing one entity:

<input type="checkbox"/> Name/ID	name
<input type="checkbox"/> Id=5629499534213120	Groceries

Figure 5.5 Your TodoList entity

tools installed, but to get the library for Cloud Datastore, you'll need the `@google-cloud/datastore` package, which you can install by running `$ npm install @google-cloud/datastore@0.4.0`. Once you have that settled, let's look at how you can query for all of the lists in your Datastore instance.

The following listing shows a quick Node.js script that asks Datastore for all of the `TodoList` entities and prints them to the screen.

Listing 5.5 Querying Cloud Datastore for all `TodoList` entities

```
const datastore = require('@google-cloud/datastore')({
  projectId: 'your-project-id'
});

const query = datastore.createQuery('TodoList');           ← Creates the
                                                          Query object

datastore.runQuery(query)                                ← Runs the query and
  .on('error', console.error)                            registers listeners to
  .on('data', (entity) => {                           handle data as it's found
    console.log('Found TodoList:\n', entity);
  })
  .on('end', () => {
    console.log('No more TodoLists');
});
```

NOTE If you get an error saying “Not Authorized,” make sure you’ve run `gcloud auth application-default login` and have authenticated successfully.

The output of this script should be something like the following:

```
Found TodoList:
{ key:
  Key {
    namespace: undefined,
    id: 5629499534213120,
    kind: 'TodoList',
    path: [Getter] },
  data: { name: 'Groceries' } }
No more TodoLists
```

As you can see, your grocery list is returned with the name you stored. Now try creating a `TodoItem` using the hierarchical key structure I described earlier. In the example shown in the following listing, your grocery list items will have keys that use the list as their parent.

Listing 5.6 Creating a new `TodoItem`

```
const datastore = require('@google-cloud/datastore')({
  projectId: 'your-project-id'
});
```

```

const entity = {
  key: datastore.key(['TodoList', 5629499534213120, 'TodoItem']),
  data: {
    name: 'Milk',
    completed: false
  }
};

datastore.save(entity, (err) => {
  if (err) {
    console.log('There was an error...', err);
  } else {
    console.log('Saved entity:', entity);
  }
});

```

The number here is the ID that you got before when querying for TodoList.

When you run this script, you should see output that looks something like the following:

```

Saved entity: { key:
  Key {
    namespace: undefined,
    kind: 'TodoItem',
    parent:
      Key {
        namespace: undefined,
        id: 5629499534213120,
        kind: 'TodoList',
        path: [Getter] ,
        path: [Getter],
        id: 5629499534213120 },
    data: { name: 'Milk', completed: false } }

```

Take special notice of the key property, which has a parent key pointing to your TodoList entity. Also note that the key has an automatically generated ID for you to reference later. Now you can add a few more items to the grocery list with a script, as in the following listing, but this time you'll save several of them in a single API call.

Listing 5.7 Adding more items to TodoList

```

const itemNames = ['Eggs', 'Chips', 'Dip', 'Celery', 'Beer'];
const entities = itemNames.map((name) => {
  return {
    key: datastore.key(['TodoList', 5629499534213120, 'TodoItem']),
    data: {
      name: name,
      completed: false
    }
  };
});

datastore.save(entities, (err) => {           ←———— Saves list of items
  if (err) {
    console.log('There was an error...', err);
  }
});

```

```

    } else {
      entities.forEach((entity) => {
        console.log('Created entity', entity.data.name, 'as ID',
          entity.key.id);
      })
    }
  );
}

```

When you run this script, you should see that your entities were created and given IDs:

```

Created entity Eggs as ID 5707702298738688
Created entity Chips as ID 5144752345317376
Created entity Dip as ID 6270652252160000
Created entity Celery as ID 4863277368606720
Created entity Beer as ID 5989177275449344

```

Now you can go back to the Cloud Console and query for all of the items in your grocery list. As you might recall, you do this by querying for the items that are descendants of the `TodoList` entity (they have this entity as an ancestor), and you express this in GQL as follows:

```

SELECT * FROM TodoItem
WHERE __key__ HAS ANCESTOR Key(TodoList, 5629499534213120)

```

If you run this query using the GQL tool in the Cloud Console, you should see that all of your grocery items are in your list (figure 5.6).

Now check one of these items off the list, and then see if you can ask for only the uncompleted ones. Start by clicking on the item in the query results and changing the `completed` field from `False` to `True` (figure 5.7). Then click Save.

The screenshot shows the Google Cloud Platform Datastore interface. At the top, there are buttons for 'CREATE ENTITY', 'REFRESH', and 'DELETE'. Below that, a navigation bar has 'Entities' selected and 'Query by kind' and 'Query by GQL' tabs, with 'Query by GQL' being active. A text input box contains the GQL query: `SELECT * FROM TodoItem WHERE __key__ HAS ANCESTOR Key(TodoList, 5629499534213120)`. To the right of the input box are buttons for 'Number of columns to display' (set to 50) and a dropdown menu. At the bottom, there are buttons for 'Run query' and 'Clear query', and a link to 'GQL query help'. The main area displays a table of query results:

<input type="checkbox"/> Name/ID	completed	name
<input type="checkbox"/> Id=4863277368606720	false	Celery
<input type="checkbox"/> Id=5144752345317376	false	Chips
<input type="checkbox"/> Id=5629499534213120	false	Milk
<input type="checkbox"/> Id=5707702298738688	false	Eggs
<input type="checkbox"/> Id=5989177275449344	false	Beer
<input type="checkbox"/> Id=6270652252160000	false	Dip

Figure 5.6 The list of items to buy at the grocery store

The screenshot shows the 'Edit entity' interface for a TodoItem. At the top, there are buttons for REFRESH and DELETE. Below that, entity metadata is displayed: Namespace: [default], Kind: TodoItem, Key: TodoList id:5629499534213120 > TodoItem id:5989177275449344, Key literal: Key(TodoList, 5629499534213120, TodoItem, 5989177275449344), and URL-safe key: ahZzfmjdjcGihLWRhdGFzdG9yZS10ZXN0cioLEghUb2RvTGldBiAgICAgICAgwL EghUb2RvSXRLbRiAgICAgOTRCgwL. The main area is titled 'Properties' and contains two rows: 'completed' (Boolean, True, Indexed checked) and 'name' (String, Beer, Indexed checked). A blue button '+ Add property' is visible. At the bottom are 'Save' and 'Cancel' buttons.

Figure 5.7 Crossing Beer off the list

Now let's go back to the code and see how you might query for all of the things you still need to buy at the grocery store. Notice that the query object has three important pieces, which are noted in the following listing.

Listing 5.8 Querying for all uncompleted TodoItem entities in your list

```
const datastore = require('@google-cloud/datastore')({
  projectId: 'your-project-id'
});

const query = datastore.createQuery('TodoItem')
  .hasAncestor(datastore.key(['TodoList', 5629499534213120]))
  .filter('completed', false); ← The filter for completed = false

datastore.runQuery(query)
  .on('error', console.error)
  .on('data', (entity) => {
    console.log('You still need to buy:', entity.data.name);
  });
  ↑ The kind you're querying (TodoItem)
  ↑ The parent key (the TodoList entity)
```

When you run this, you should see that everything you added before is on the list, except for the Beer item, which you marked as completed:

```
You still need to buy: Celery
You still need to buy: Chips
You still need to buy: Milk
You still need to buy: Eggs
You still need to buy: Dip
```

Now that we've explored a bit about how to interact with Cloud Datastore, let's look at how you might go about backing up and restoring your data.

5.3 Backup and restore

Backups are one of those things that you tend to not need until you *really* need them, particularly when you accidentally delete a bunch of data. Cloud Datastore backups are a bit unusual in that they're not exactly backups in the sense that you've gotten used to them. Datastore's eventually consistent queries make it difficult to get the overall state of the data at a single point in time. Instead, asking for all the data tends to be more of a *smear* over the time that it takes the query to run.

What does this mean? First, Datastore's backup ability is more of an export that's able to take a bunch of data from a regular Datastore query and ship it off to a Cloud Storage bucket. But because a regular Datastore query is only eventually consistent, the data exported to Cloud Storage could be equally inconsistent. For example, if you were to create a new entity every second, a backup of the data after 10 seconds could end up storing exactly the 10 entities...or more than 10. More confusingly, you might end up seeing fewer than 10!

Because of this effect, it's important to remember that exports are not a snapshot taken at a single point in time. Instead, they're more like a long-exposure photograph of your data. To minimize the effect of this long exposure, it's possible to disable Datastore writes beforehand and then re-enable them once the export completes. With all that in mind, let's look at how you can export your data.

NOTE As of this writing, this feature of Datastore is Beta, meaning that the commands you'll run will start with gcloud beta.

First, you'll need a Cloud Storage bucket (see listing 5.9), which I explain in chapter 8. For now, consider it a place that'll hold your exported data, which you interact with using the gsutil command that comes with the Cloud SDK command-line tool.

Listing 5.9 Creating a Cloud Storage bucket

```
$ gsutil mb -l US gs://my-data-export
Creating gs://my-data-export/...
```

Once you've created the bucket, you can disable writes to your Datastore instance via the Cloud Console, using the Admin tab in the Datastore section (figure 5.8).

After that, you can trigger an export of your data into your bucket using the datastore export subcommand, shown in listing 5.10.

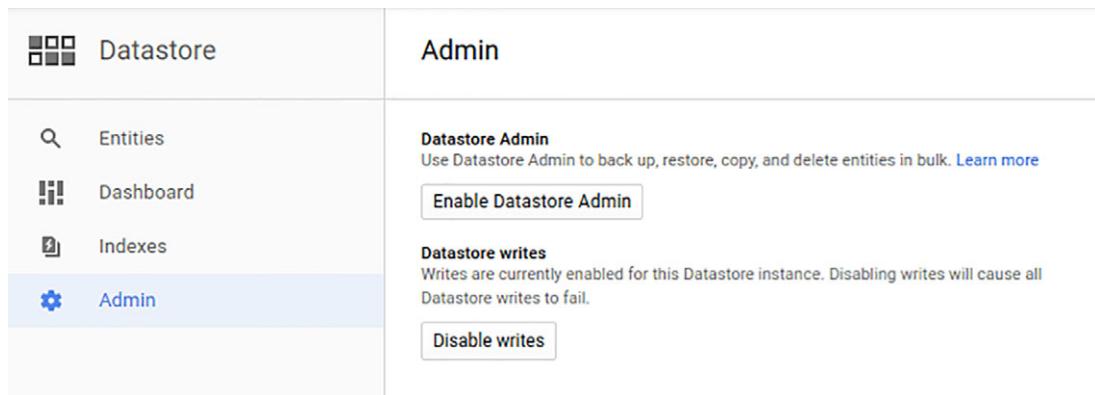


Figure 5.8 Disabling writes to Datastore using the Cloud Console

Listing 5.10 Exporting data to Cloud Storage

```
$ gcloud beta datastore export gs://my-data-export/export-1
Waiting for [projects/your-project-id-here/operations/
    ASA1MTIwNzE4OTIJGnRsdWFmZWQHEmxhcnRuZWNzdS1zYm9qLW5pbWRhFAosEg] to
    finish...done.
metadata:
  '@type':
    type.googleapis.com/google.datastore.admin.v1beta1.ExportEntitiesMetadata
  common:
    operationType: EXPORT_ENTITIES
    startTime: '2018-01-16T14:26:02.626380Z'
    state: PROCESSING
    outputUrlPrefix: gs://my-data-export/export-1
  name: projects/your-project-id-here/operations/
    ASA1MTIwNzE4OTIJGnRsdWFmZWQHEmxhcnRuZWNzdS1zYm9qLW5pbWRhFAosEg
```

Once that completes, you can verify that the data arrived in your bucket, again using the gsutil tool, as follows.

Listing 5.11 Viewing the size of the export data

```
$ gsutil du -sh gs://my-data-export/export-1
32.2 KiB    gs://my-data-export/export-1
```

You can see here that the data has arrived in your bucket, taking up about 32 kilobytes of space.

Now that you can see the export is complete, I can start talking about the other half of this puzzle: restoring.

Similar to how backing up is more like exporting, restoring is more like importing, which raises a couple of topics worth mentioning. First, importing entities will use all the same IDs as before, which will overwrite any entities that use those IDs. If any accidental ID collisions occur, those entities will be overwritten. This should only be a problem if you choose your own IDs, but it's worth knowing. Second, because this is

an import rather than a restore, any entities that you created after the previous export (and therefore that are unaffected by the import) will still remain. The import can edit and create entities, but will never delete any entities.

To run an import, you can do the same thing you did with the export, remembering first to disable writes ahead of time. The only difference this time is that instead of pointing to a directory where the data will live, you'll need to point to the metadata file that was created during the export. You can find this metadata file using the gsutil command once again, as shown in the following listing.

Listing 5.12 Listing the objects created by the export

```
$ gsutil ls gs://my-data-export/export-1  
gs://my-data-export/export-1/export-1.overall_export_metadata  
gs://my-data-export/export-1/all_namespaces/
```

Lists the objects that were created by the export job

The export metadata created, which you'll reference during an import

Now that you have the path to the metadata file for the export, you can trigger an import job using the gcloud command similar to before, as follows.

Listing 5.13 Importing data from a previous export

```
$ gcloud beta datastore import gs://my-data-export/export-1/export-  
1.overall_export_metadata  
Waiting for [projects/your-project-id-here/operations/  
AiA4NjUwODEzOTIJGnRsdWFmZWQHEmxhcnRuZWNzdS1zYm9qLW5pbWRhFAosEg] to  
finish...done.  
metadata:  
  '@type':  
    type.googleapis.com/google.datastore.admin.v1beta1.ImportEntitiesMetadata  
  common:  
    operationType: IMPORT_ENTITIES  
    startTime: '2018-01-16T16:26:17.964040Z'  
    state: PROCESSING  
  inputUrl: gs://my-data-export/export-1/export-1.overall_export_metadata  
  name: projects/your-project-id-here/operations/  
AiA4NjUwODEzOTIJGnRsdWFmZWQHEmxhcnRuZWNzdS1zYm9qLW5pbWRhFAosEg
```

At this point, if you had made changes to any of the entities (or deleted any entities), those entities would be reverted to how they were at the time of the export. But if you had created new entities, they'd be left entirely alone, because an import doesn't affect entities it hasn't seen before.

Now that you have a good grasp of using Cloud Datastore, let's look in more detail at how much all of this will cost you.

5.4 Understanding pricing

Google determines Cloud Datastore prices based on two things: the amount of data you store and the number of operations you perform on that data. Let's look at the easy part first: storage.

5.4.1 Storage costs

Data stored in Cloud Datastore is measured in GB, costing \$0.18 per GB per month as of this writing. That might sound pretty straightforward, but it's a bit more complicated than it looks. In addition to your data (the property values on your entities), the total storage size for billing purposes of a single entity includes the kind name (for example, Person), the key name (or ID), all property names (for example, favorite-Color), and 16 extra overhead bytes. Furthermore, all properties have simple indexes created, where each index entry includes the kind name, the key name, the property name, the property value, and 32 extra overhead bytes. Finally, don't forget that Cloud Datastore includes indexes for both ascending and descending order.

In short, long names (indexes, properties, and keys) tend to explode in size, so you'll have far more total data than the actual data stored. For lots of detail about how Google computes the total storage size, take a look at the online storage reference: <http://mng.bz/BcIr>. Knowing this is particularly important if you expect to have a lot of entities and indexes to query over those entities.

Now let's talk about the other pricing aspect, which in retrospect is much more straightforward: operations.

5.4.2 Per-operation costs

Operations, in short, are any requests that you send to Cloud Datastore, such as creating a new entity or retrieving data. Cloud Datastore charges based on how many entities are involved in a given operation, at different rates for different types of operations, so some operations (such as updating or creating an entity) cost more than others (such as deleting an entity). The price breakdown is shown in table 5.7.

Table 5.7 Operation pricing breakdown

Operation type	Cost per 100,000 entities
Read	\$0.06
Write	\$0.18
Delete	\$0.02

Unlike storage totals, this type of pricing has few gotchas. For example, if you retrieve 100,000 of your entities, your bill will be 6 cents. Similarly, updating and deleting those entities will cost you 18 and 2 cents, respectively. The only thing to worry about is queries that involve retrieving each entity in the query. If you run a query selecting

all of your entities, that'll count as a read operation on each entity returned to you. If all you want is to look at the *key* of your entities, you can use a keys-only query, which is a free operation.

Now that you have a grasp on how Datastore pricing works, it's time to think about when Cloud Datastore is a good fit for your projects.

5.5 When should I use Cloud Datastore?

Let's start with a scorecard to summarize some of the strong and weak points of Cloud Datastore. Notice that the two places where Datastore shines are durability and throughput, and that cost is entering into the danger zone. See figure 5.9.

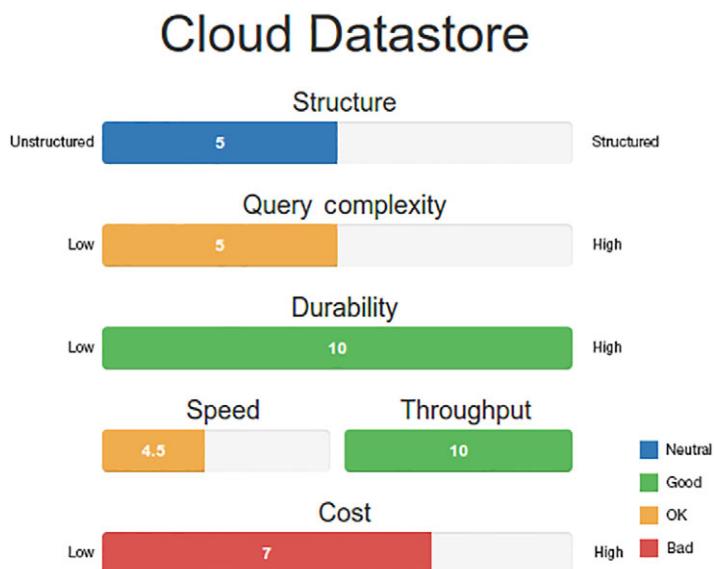


Figure 5.9 Cloud Datastore scorecard

5.5.1 Structure

As you learned, unlike relational databases, Cloud Datastore excels at managing semi-structured data where attributes have types, but it provides no single schema across all entities (or *documents*) of the same kind. You might choose to design your system such that entities of a single kind are homogeneous, but that's up to you to enforce in your application code.

Along with the document-style storage, Datastore also allows you to express the locality of your data using hierarchical keys (where one key is prefixed with the key of its parent). This can be confusing but reflects the desire to segment data between units of isolation (for example, a single user's emails). This aspect of Datastore, which enables automatic replication of your data, is what allows it to be so highly available as

a storage system. Although this setup provides many benefits, it also means that queries across all the data will be eventually consistent.

5.5.2 **Query complexity**

As with any nonrelational storage system, Cloud Datastore doesn't support the typical relational aspects (for example, the `JOIN` operator). It does allow you to store keys that act as pointers to other stored entities, but it provides no management for these values. Most notably, it has no referential integrity and no ability to cascade or limit changes involving referenced entities. When you delete an entity in Cloud Datastore, anywhere you pointed to that entity from elsewhere becomes an invalid reference.

Furthermore, certain queries require that you have indexes to enable them, which is somewhat different from a relational database, where indexes are helpful but not necessary to run specific queries. Some of these limitations are the consequence of the structural requirements that went into designing Cloud Datastore, whereas other limitations enable consistent performance for all queries.

5.5.3 **Durability**

Durability is where Cloud Datastore starts to excel. Because Megastore was built on the premise that you can never lose data, everything is automatically replicated and not considered saved until saved in several places. Although you have various levels of self-management for replication when using a relational database (even Cloud SQL requires that you configure your replicas), Datastore handles this entirely on its own, meaning that the only setting for durability is *as high as possible*.

This arrangement, combined with the indexes aspect discussed previously, has an unfortunate side effect of global queries being only eventually consistent. Because your data needs to replicate to several places before being called saved, at times a query across all data may return stale results because it takes additional time for the indexes to be updated alongside the data.

5.5.4 **Speed (latency)**

Compared to many in-memory storage systems (for example, Redis), Cloud Datastore won't be as fast for the simple reason that even SSDs are slower than RAM. Compared to a relational database system like PostgreSQL or MySQL, Cloud Datastore will be in the same ballpark, with one primary difference: as your SQL database gets larger or receives more requests at the same time, it'll likely get slower. As you learned in this chapter, Cloud Datastore's latency stays the same regardless of the level of concurrency, and the time a query takes to run scales with the size of the result set rather than the amount of data that needs to be sifted through.

The key thing to take away from this section is that Cloud Datastore certainly won't be blazing fast like in-memory NoSQL storage systems, but it'll be on par with other relational databases and will remain consistent as you increase your levels of concurrency as well as the size of your data.

5.5.5 Throughput

Cloud Datastore's throughput benefits from running on Google's infrastructure as a fully managed storage service, so it can accommodate as much traffic as you care to throw at it. Because your data is automatically spread out across different groups (unless you specifically say not to do so), the pessimistic locking that comes with relational databases like MySQL doesn't apply; instead, you're able to scale up to many concurrent write operations.

This scalability also means that if you ever grow so large that even Google has trouble supporting your traffic, it's a simple matter of adding more servers on Google's side to keep up. Compare this to MySQL's throughput story. With MySQL, you can deal with reads using read-replicas, but scaling up the number of concurrent write operations executing is quite a challenge. Cloud Datastore makes this something you don't have to worry about.

5.5.6 Cost

Cloud Datastore's costs are unique in that they tend to grow in somewhat surprising ways. At smaller scales, for example, storing a few gigabytes, your total cost of storage and querying could be around \$50 a month, which is pretty reasonable. As you add more and more data, and query that data more and more frequently, overall costs can skyrocket—primarily because of indexes.

In exchange for the enormous cost, you get the benefit of never worrying that your data will be unavailable. You might be paying a lot to store and access the data, but when your application is featured on a TV show and the whole world starts accessing it, everything will just work, and you'll certainly get your money's worth out of those indexes.

5.5.7 Overall

Now that you have an idea of where Cloud Datastore starts to do well, let's take our example applications and see whether Datastore is a good fit.

To-Do List

As a starter app, your To-Do List definitely won't need the high levels of throughput that Datastore can provide, but being a fully managed offering, it brings some interesting things to the table. See table 5.8.

Table 5.8 To-Do List application storage needs

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

Table 5.8 To-Do List application storage needs (continued)

Aspect	Needs	Good fit?
Throughput	Not a lot.	Sure
Cost	Lower is better for toy projects.	Definitely

In short, Cloud Datastore is an acceptable fit, but it's a bit of overkill on the scalability side. This is sort of like giving your grandmother a Lamborghini. It'll get her to the grocery store fine, but she probably won't be drag racing on her way there.

If this To-Do List app could become something enormous, then Datastore is a safe bet to go with because it means that scaling to handle tons of traffic is something you don't need to worry about too much.

E*EXCHANGE

E*Exchange, the online trading platform, is a bit more complex compared to the To-Do List app. Specifically, the main difference is in the complexity of the queries that customers are likely to need. See table 5.9.

Table 5.9 E*Exchange storage needs

Aspect	Needs	Good fit?
Structure	Yes, reject anything suspect—No mistakes.	Maybe
Query complexity	Complex—We have fancy questions to answer.	No
Durability	High—We can't lose stuff.	Definitely
Speed	Things should be relatively fast.	Probably
Throughput	High—Lots of people may be using this.	Definitely
Cost	Lower is better, but willing to pay top dollar.	Definitely

Looking at table 5.9, Cloud Datastore is probably not the best fit for E*Exchange if used on its own. For example, Cloud Datastore doesn't enforce strict schema requirements, but E*Exchange wants clear validation of any data entering the system. To do this, you'd have to enforce that schema in your application rather than relying on the database. So although it's possible to do it, it's not built into Datastore.

Furthermore, you learned that Datastore can't do extremely complex queries, specifically things like joining two separate tables together. This means that, again, Datastore on its own is unlikely to be a good fit.

Finally, Datastore's eventually consistent queries will be challenging to design around for a system that requires highly accurate and up-to-date information like E*Exchange. Although you could certainly design around this consistency model, it'd be quite a bit of work.

If E*Exchange was hoping to benefit from Datastore's high durability, replication, and throughput abilities, it'd likely make the most sense to store the raw data in Datastore while using some sort of data warehouse or analytical storage engine for running the more complex queries. E*Exchange would store each single trade as an entity, which would scale to extremely high throughput and always maintain high durability, while storing the analytical data in something like BigQuery (see chapter 19) or one of the many time-series databases, such as HBase, InfluxDB, or OpenTSDB.

It's also important to mention that because Datastore offers full ACID (atomicity, consistency, isolation, durability) transaction semantics, you never have to worry about multiple updates accidentally ending up in a half-committed state. For example, transferring shares would be an atomic transaction that would decrease the seller's balance and increase the buyer's balance, and you don't have to worry that one of those changes would be committed while another was lost because of some sort of failure.

INSTASNAP

InstaSnap, the popular social media application, has a few requirements that seem to fit well and only a couple that are a bit off. See table 5.10.

Table 5.10 InstaSnap storage needs

Aspect	Needs	Good fit?
Structure	Not really—Structure is pretty flexible.	Definitely
Query complexity	Mostly lookups; no highly complex questions.	Definitely
Durability	Medium—Losing things is inconvenient.	Sure
Speed	Queries must be very fast.	Maybe
Throughput	Very high—Kim Kardashian uses this.	Definitely
Cost	Lower is better, but willing to pay top dollar.	Definitely

The biggest issue for an app like InstaSnap is the single-query latency, which needs to be extremely fast. This is yet another place where Datastore on its own isn't the best fit, but, if you use it in conjunction with some sort of in-memory cache like Memcached, this problem goes away entirely. Additionally, although InstaSnap's durability needs aren't all that serious, the fact that Datastore provides higher levels than needed isn't such a big deal.

In short, InstaSnap is a pretty solid fit because of the relatively simple queries combined with the enormous throughput requirements. As a matter of fact, SnapChat (the real app) uses Datastore as one of its primary storage systems.

5.5.8 Other document storage systems

As a document storage system, Cloud Datastore is one of many options: from the other hosted services, like Amazon's DynamoDB, to the many open source alternatives,

like MongoDB or Apache HBase. (You'll learn more about HBase's parent system, Bigtable in chapter 7.) You have plenty of systems to choose from, each with its own benefits and drawbacks. In some cases, a system can act a bit like a document-storage system in certain configurations, even if it wasn't designed for that.

Table 5.11 attempts to summarize the characteristics of several of the document storage systems and suggest when you might want to choose one over another.

Table 5.11 Brief comparison of document storage systems

Name	Cost	Flexibility	Availability	Durability	Speed	Throughput
Cloud Datastore	High	Medium	High	High	Medium	High
MongoDB	Low	High	Medium	Medium	Medium	Medium
DynamoDB	High	Low	Medium	Medium	High	Medium
HBase	Medium	Low	Medium	High	High	High
Cloud Bigtable	Medium	Low	High	High	High	High

Notice that although it's possible to configure systems like HBase and MongoDB for high availability, when that happens, cost will go up significantly. You can read more about scaling such systems in chapter 7, section 7.7. First, though, now that you have a grasp on how Datastore stacks up, we'll take a look at pricing in chapter 6 to see what the overall cost is.

Summary

- Document storage keeps data organized as heterogeneous (jagged) documents rather than homogeneous rows in a table.
- Using document storage effectively may involve duplicating data for easy access (denormalizing).
- Document storage is great for storing data that may grow to huge sizes and experience huge amounts of traffic, but it comes at the cost of not being able to do fancy queries (for example, *joins* that you do in SQL).
- Cloud Datastore is a fully managed storage system with automatic replication, result-set query scale, full transactional semantics, and automatic scaling.
- Cloud Datastore is a good fit if you need high scalability and have relatively straightforward queries.
- Cloud Datastore charges for operations on entities, meaning the more data you interact with, the more you pay.

Cloud Spanner: large-scale SQL

This chapter covers

- What is NewSQL?
- What is Spanner?
- Administrative interactions with Cloud Spanner
- Reading, writing, and querying data
- Interleaved tables, primary keys, and other advanced topics

So far we've looked at relational (SQL) databases and nonrelational (NoSQL) databases and learned about some of the trade-offs of each. SQL databases generally provide richer queries, strong consistency, and transactional semantics but have trouble handling massive amounts of traffic. NoSQL databases tend to trade some or all of these in exchange for horizontal scalability, which allows the system to easily handle more traffic by adding more machines to the cluster. Obviously, the choice you make between SQL and NoSQL will depend on your business needs, but wouldn't it be nice if you didn't have to make that choice?

6.1 What is NewSQL?

What if you could have rich querying, transactional semantics, strong consistency, *and* horizontal scalability? These types of systems are sometimes referred to as NewSQL databases.

NewSQL databases look and act a lot like SQL databases, but they have the scaling properties of NoSQL databases. For example, a NewSQL database may require that data locality be expressed in the schema somehow, but you can still query your data using familiar `SELECT * FROM ...` syntax. Let's explore a bit of Google's history in this area and see what came out in an attempt to solve this problem.

6.2 What is Spanner?

For a long time, many of Google's needs were no different than those of any other business, where data was structured and relational and fit comfortably in MySQL. As the size of the data stored grew out of control, it became a problem. The first step to fixing this was to push the off-the-shelf databases beyond where they were designed to perform, sharding data and hiring lots of database administrators to fine-tune the system. This helped but didn't solve the problem, and the data kept growing.

Unfortunately, using one of the in-house storage systems (like Megastore) wouldn't work because the features needed were things like transactional or relational semantics as well as strong consistency, and those features were traded first when designing things like Megastore. What was needed was a system that combined the scalability of nonrelational storage with the features of a traditional MySQL database, leading to Spanner.

Spanner is a NewSQL database that offers many of the features of a relational database (like schemas and `JOIN` queries) with many of the scaling properties of a nonrelational database (like being able to add more machines). In the case of failures (or exceptionally large loads), Spanner can split and redistribute data across more machines, even if they're in entirely separate data centers. Through dynamic resizing and shuffling of data chunks, the system is prepared for all types of disasters.

Spanner also offers strongly consistent queries so you'll never have a stale version of the data. Following the pattern of Google Cloud Platform, Google has taken the Spanner database, which at first was available only to Google engineers, and made it available to anyone using Google Cloud Platform as a hosted storage system, much like Cloud Datastore or Cloud Bigtable. Let's dive right into some of the concepts to see how you go about using Cloud Spanner.

6.3 Concepts

As with any storage system, you should understand a few underlying concepts before getting started. In this section we'll explore a few of those, starting with the infrastructural concept of an instance, and then dive into the data-model concepts like tables and keys. Along the way, we'll touch on some of the more theoretical concepts like

split points and transactions, which are relevant when digging into how to use Spanner to get the best performance possible. Let's dive right in with instances.

6.3.1 Instances

In its most basic form, a Cloud Spanner *instance* acts as an infrastructural container that holds a bunch of databases (see figure 6.1). It also manages multiple discrete units of computing power, which are ultimately responsible for serving your Spanner data. Spanner instances feature two aspects: a data-oriented aspect and an infrastructural aspect. Let's start by exploring the data-oriented side of things.

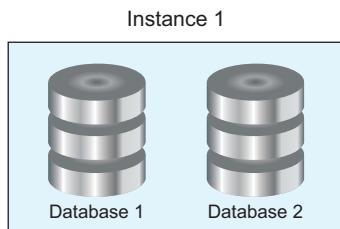


Figure 6.1 At a high level, instances are containers for databases.

When you want to run a query and receive results, an instance acts as nothing more than a database container, similar to a Cloud SQL instance. When you run a query, you route it to the instance itself, and Spanner does the heavy lifting. What about the infrastructural side?

Unlike a single MySQL instance, Spanner instances are automatically replicated. Rather than choosing a specific zone where the instance will live, you choose a configuration that maps to some combination of different zones. For example, the `regional-us-central1` configuration represents some combination of zones inside the `us-central1` region (see figure 6.2). Spanner instances do have geographical homes, but the location is much more general than the home of, say, a Compute Engine VM.

Now that you understand this dual nature of instances, let's look more deeply at the physical component that makes up the computing power of an instance: a node.

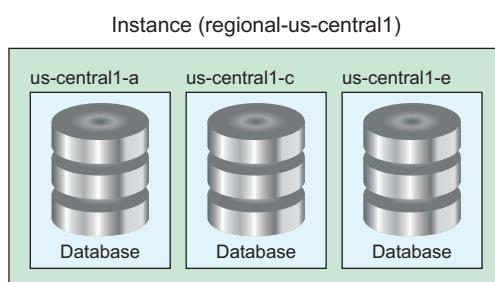


Figure 6.2 Instance configurations determine the zones that data is replicated to.

6.3.2 Nodes

In addition to acting like containers of databases and being replicated across multiple different zones, Spanner instances are made up of a specific number of nodes that can be used to serve instance data. These nodes live in specific zones and are ultimately responsible for handling queries. Because Spanner instances are fully replicated, you have identical replicas in each of the different zones (see figure 6.3), which ensures that if any zone has an outage, your data will continue serving without any problems.

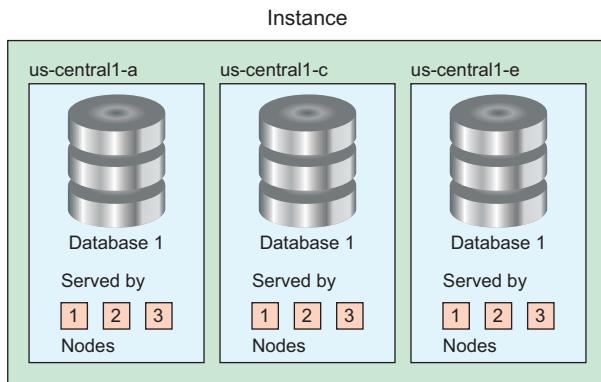


Figure 6.3 Instances have the same number of nodes in every replica.

If you have a three-node instance in a regional configuration (replicated across three zones), you have a total of nine nodes because each replica is a copy of both the data and the serving capacity. Although this might seem like overkill, recall that Spanner's guarantees are focused on rich querying, globally strong consistency, and high availability and performance. Notably missing from this is low cost—Spanner overcomes many of these issues by throwing more resources at the problem. Now that you understand instances and the replication configurations, let's explore how databases work.

6.3.3 Databases

Databases are primarily containers of tables. Typically a single database acts as a container of data for a single product, which makes things like limiting access permissions or atomically dropping all data easy. We'll also use databases to make schema changes and query for data. Let's dig a tiny bit deeper and discuss what Spanner tables are and how they work.

6.3.4 Tables

In most ways, Spanner tables are similar to other relational databases, but with some important differences. Let's start by talking about what's the same, and then we'll explore the differences later in the chapter.

Tables have a schema, which looks a lot like those of any other relational database. Tables have columns, which have types (such as `INT64`) and modifiers (such as `NOT NULL`) that define the shape of your data. Like in a relational database, adding data that doesn't match the type defined in the schema results in an error. Tables have a few other constraints, such as a maximum cell size of 10 MiB, but in general, Spanner tables shouldn't be surprising. To demonstrate how similar Spanner tables can be to those in other databases, let's look at an example and compare the two schema definitions. In the next listing you'll see a table for storing employee records, which is valid for defining a table in MySQL.

Listing 6.1 Storing employee IDs and names

```
CREATE TABLE employees (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    start_date DATE
);
```

Here's an example of creating the same table in Cloud Spanner.

Listing 6.2 Storing employee IDs and names in Spanner

```
CREATE TABLE employees (
    employee_id INT64 NOT NULL,
    name STRING(100) NOT NULL,
    start_date DATE
) PRIMARY KEY (employee_id);
```

As you can see, these tables are almost identical, with some small differences in data type names and location of the primary key directive. Because you have enough background information to take Spanner for a test drive, let's explore how to use it and then come back later to explore some of the more advanced topics.

6.4 Interacting with Cloud Spanner

Before you can store any data in Cloud Spanner, you first have to create some of the infrastructural resources. You'll start by doing that in the Cloud Console. As always, you start by enabling the Cloud Spanner API. In the Cloud Console, enter Cloud Spanner API in the main search box at the top of the page. One result should appear. Click that to open a page, shown in figure 6.4, with an Enable button. After you click that, you should be good to go.

Once that's done, head over to the Spanner interface by clicking Spanner in the Storage section in the left-side navigation.

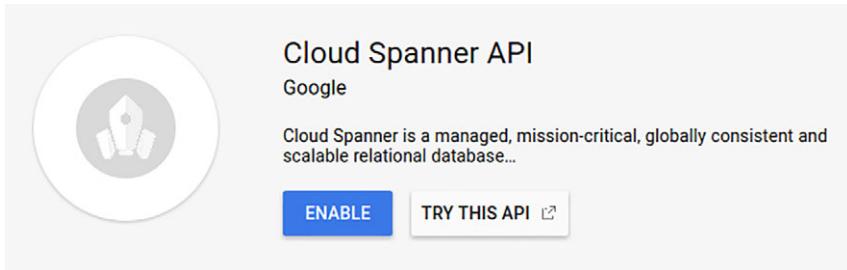


Figure 6.4 Enable the Cloud Spanner API

6.4.1 *Creating an instance and database*

When you first start Spanner, you don't have any databases, so you see a prompt asking you to create a Spanner instance. See figure 6.5.

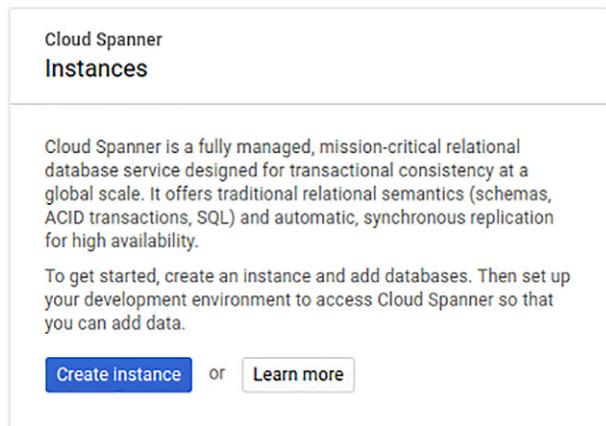


Figure 6.5 The prompt you'll see on your first visit to the Spanner UI

NOTE Though Cloud Spanner is powerful, it can also be expensive. This means that if you turn on an instance in this tutorial, don't forget to turn it off afterward or you may get a bigger bill than you expected!

When you click Create instance, a form opens where you can choose some of the details for your Spanner instance. For this example, call the instance "Test Instance." When you type the name into the first field, you should notice that a simplified version of the name automatically appears in the field for the instance ID. The first field is the display name that you'll see in the UI, and the second field is the official ID of the instance that you'll need when addressing it in any API calls.

After that, you need to choose the configuration. As you learned earlier, Spanner configurations are sort of like Compute Engine zones and concern availability. Like with a VM, you're going to be accessing the Spanner instance from your local

machine, so it's a good idea to choose a configuration geographically near you. Additionally, when you're using your instance in production, you should generally have the VMs accessing Spanner in the same region as the instance itself. If you deploy your Spanner instance in the us-central1 configuration, you'll want to put your VMs in us-central1 zones (such as us-central1-a).

Last, for the purposes of this test—unless you're looking to run a benchmark or performance test—leave the number of nodes set to one. Under the hood, this will result in having three node replicas spread across three different zones (one node in each zone), which is plenty of capacity for your test. See figure 6.6.

[← Create an instance](#)

Instance name
For display purposes only.

Instance ID
Unique identifier for instance. Permanent.

Configuration
Determines where your data and nodes are located. Affects cost, performance, and replication. This choice is permanent. Select a configuration to view its details.

Regional
 Multi-region

Nodes
Add nodes to increase data throughput and queries per second (QPS). Affects billing.

Node guidance

Cost
Storage cost depends on GB stored per month. Nodes cost is an hourly charge for the number of nodes in your instance. [Learn more](#)

Nodes cost \$0.90 per hour	Storage cost \$0.30 per GB/month
--------------------------------------	--

Create **Cancel**

Figure 6.6 Creating a Spanner instance

When you click Create, the instance should appear and a page where you can view your new (but empty) instance opens, as shown in figure 6.7. Now that you have your instance, you have to create a new database. To do that, click the Create database button. A form where you can choose a database name and fill in a schema opens, as shown in figure 6.8.

The screenshot shows the 'Instance details' section for a Cloud Spanner instance named 'Test Instance'. At the top, there are buttons for 'CREATE DATABASE', 'EDIT', 'DELETE', and 'PERMISSIONS'. Below this, the 'Overview' tab is selected, showing the instance ID 'test-instance' and configuration 'us-central1'. A summary table provides real-time metrics: 1 node, 0% CPU utilization (mean), 0 operations (Read: 0/s, Write: 0/s), 0 B/s throughput (Outbound: 0 B/s, Inbound: 0 B/s), and 0 B total storage. Under the 'Databases' section, it says 'No databases yet. Create a database to get started.' A prominent blue 'Create database' button is at the bottom left, along with a link to 'Spanner documentation'.

Figure 6.7 Viewing your newly created instance

The screenshot shows the 'Create a database' wizard. Step 1, 'Name your database', is completed with a green checkmark. It asks for a permanent name and has a 'test-database' input field. Step 2, 'Define your database schema', is partially completed. It allows adding tables and indexes via 'Edit as text' (radio button selected) and '+ Add table' and '+ Add index' buttons. At the bottom are 'Create' and 'Cancel' buttons.

Figure 6.8 Creating your first database

This is a two-step process where you first choose a name for the database, and you then can create some tables for your database. For now, leave the database completely empty. Enter the name `test-database` and then click Create. A page where you can view your new (but empty) database appears. See figure 6.9.

The screenshot shows the 'Database details' page for a database named 'test-database'. At the top, there are navigation links: 'QUERY', 'CREATE TABLE', 'DELETE', and 'PERMISSIONS'. Below the title, there are tabs for 'Overview' (which is selected) and 'Monitor'. Under 'Overview', there are four performance metrics boxes: 'CPU utilization (mean)' at 1.49%, 'Operations' (Read: 0/s, Write: 0/s), 'Throughput' (Outbound: 0 B/s, Inbound: 0 B/s), and 'Total storage' at 0 B. Below these metrics, there's a section titled 'Tables' with the sub-instruction 'No tables yet. Create a table to get started.' followed by a blue 'Create table' button.

Figure 6.9 Viewing your newly created database

Now that you have an empty database, let's move on to the schema side of things and create a new table.

6.4.2 Creating a table

As you learned, Spanner tables are similar to other relational databases, but we'll save the differences for later when we discuss more advanced topics. To start, you're going to create a simple employee information table which has the two fields you used in our earlier example: a unique ID (primary key) for the employee, and the employee's name.

To get started, click the Create table button, and a form where you can create the table opens. The Cloud Console makes it easy to create a new table with a helpful schema-building tool. Because you're going to learn about more advanced concepts later, use the Edit as text option and paste in the schema for your `employees` table, as shown in figure 6.10.

After you click Create, a page opens where you can see the details of your table, such as the schema, any indexes (currently you have none), and a preview of the data (which will be empty now). See figure 6.11.

You've now created an instance, a database belonging to the instance, and a table belonging to the database. But what good is an empty table? Let's move onto the interesting part: loading it up with some data.

[← Create a table in test-database](#)

Edit as text

DDL statements

Add Spanner Database Definition Language SQL statements below. Separate statements with a semicolon. [Learn more](#) ↗

```
1 CREATE TABLE employees (
2   employee_id INT64 NOT NULL,
3   name STRING(100) NOT NULL,
4   start_date DATE
5 ) PRIMARY KEY (employee_id);
```

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

Figure 6.10 Creating your employees table

Table details[+ CREATE INDEX](#)[EDIT](#)[DELETE](#)**employees**[Schema](#) [Indexes](#) [Preview](#)

Column	Type	Nullable
employee_id	INT64	No
name	STRING(100)	No
start_date	DATE	Yes

[Show equivalent DDL](#)

Figure 6.11 Viewing your newly created table

6.4.3 Adding data

One of the key differences between Spanner and other relational databases is the way you modify data. In a typical database, like MySQL, you use an `INSERT SQL` query to add new data and an `UPDATE SQL` query to update existing data. Spanner doesn't support those two commands, however, which shows its NoSQL influences.

Instead of inserting data using the query interface, you write to Cloud Spanner via a separate API, which is more similar to a nonrelational key-value system, where you choose a primary key and then set some values for that key. To demonstrate, use the `@google-cloud/spanner` Node.js package to add some employee data to your `employees` table in Spanner, as shown in the following listing. You can install this using `npm`, by running `npm install @google-cloud/spanner@0.7.0`.

Listing 6.3 Script to add some employees to your table

```
const spanner = require('@google-cloud/spanner')({
  projectId: 'your-project-id'
});

const instance = spanner.instance('test-instance');
const database = instance.database('test-database');
const employees = database.table('employees');

→ employees.insert([
  {employee_id: 1, name: 'Steve Jobs', start_date: '1976-04-01'},
  {employee_id: 2, name: 'Bill Gates', start_date: '1975-04-04'},
  {employee_id: 3, name: 'Larry Page', start_date: '1998-09-04'}
]).then((data) => {
  console.log('Saved data!', data);
});
```

Remember to replace the project ID here with your own project ID.

Create a pointer to the database that you created in the Cloud Console.

Create a pointer to the table that you created earlier.

Insert several rows of data, each row being its own JSON object.

If everything worked, you'll see output confirming that the data was saved, as well as the time stamp of the change being persisted:

```
> Saved data! [ { commitTimestamp: { seconds: '1489763847', nanos: 466238000 } } ]
```

Now that we've seen how to get data into Spanner, let's look at how to get it out of Spanner.

6.4.4 Querying data

There are two ways that you can query data. First, you can use Spanner's Read API to query a single table. These queries can be either lookups of a specific key (or set of keys) or a table scan with some filters applied. This method is probably the best fit to retrieve the three rows you added.

You can also execute a SQL query on the database, which allows you to query multiple tables using joins and other advanced filtering techniques that you've come to know in other databases. In this case, you don't need to do anything complex so this would be overkill, but we'll demonstrate it anyway. Start by using the Read API, by calling `table.read()` in the Node.js client library to fetch one of the rows you added by the primary key, as shown in the next listing.

Listing 6.4 Using Spanner's Read API to retrieve a row by its key

```
const spanner = require('@google-cloud/spanner')({
  projectId: 'your-project-id'
});
const instance = spanner.instance('test-instance');
const database = instance.database('test-database');
const employees = database.table('employees');
const query = {
  columns: ['employee_id', 'name', 'start_date'],
  keys: ['1']
};

employees.read(query).then((data) => {
  const rows = data[0];
  rows.forEach((row) => {
    console.log('Found row:');
    row.forEach((column) => {
      console.log(' - ' + column.name + ': ' + column.value);
    });
  });
});
```

After running this, you can see that the row you added was stored correctly:

```
Found row:
- employee_id: 1
- name: Steve Jobs
- start_date: Wed Mar 31 1976 19:00:00 GMT-0500 (EST)
```

But what if you wanted to get all of the rows in the database? Generally, this is a bad idea, but because you're trying to check whether the three rows you added were stored successfully, you can use a special `all` flag on the query, shown next.

Listing 6.5 Retrieving all rows

```
const spanner = require('@google-cloud/spanner')({
  projectId: 'your-project-id'
});

const instance = spanner.instance('test-instance');
const database = instance.database('test-database');
const employees = database.table('employees');
const query = {
  columns: ['employee_id', 'name', 'start_date'],
```

```
    keySet: {all: true}
};

employees.read(query).then((data) => {
  const rows = data[0];
  rows.forEach((row) => {
    console.log('Found row:');
    row.forEach((column) => {
      console.log(' - ' + column.name + ': ' + column.value);
    });
  });
});
```

After running this code, you will see all of the data that you added come back as the results:

```
Found row:
- employee_id: 1
- name: Steve Jobs
- start_date: Wed Mar 31 1976 19:00:00 GMT-0500 (EST)
Found row:
- employee_id: 2
- name: Bill Gates
- start_date: Thu Apr 03 1975 20:00:00 GMT-0400 (EDT)
Found row:
- employee_id: 3
- name: Larry Page
- start_date: Thu Sep 03 1998 20:00:00 GMT-0400 (EDT)
```

Now that you've tried the Read API, let's look at the more generic SQL-querying API. The first notable difference when querying is that you query a database rather than a specific table because the query might involve other tables (for instance, if you JOIN two tables together). Additionally, instead of sending a structured object to represent the query, you send a string containing your SQL query.

Start by sending a simple query to retrieve all of the employees with a SQL query, as shown in the next listing. As you might expect, the query itself is straightforward and identical to what it would be when querying something like MySQL.

Listing 6.6 Executing a SQL query against Spanner

```
const spanner = require('@google-cloud/spanner')({
  projectId: 'your-project-id'
});

const instance = spanner.instance('test-instance');
const database = instance.database('test-database');
const query = 'SELECT employee_id, name, start_date FROM employees';

database.run(query).then((data) => {
  const rows = data[0];
  rows.forEach((row) => {
    console.log('Found row:');
```

```

    row.forEach((column) => {
      console.log(' - ' + column.name + ': ' + column.value);
    });
  });
}
);

```

After running this, you'll see the same output as the previous run, showing all of the employees and the columns involved:

```

Found row:
- employee_id: 1
- name: Steve Jobs
- start_date: Wed Mar 31 1976 19:00:00 GMT-0500 (EST)
Found row:
- employee_id: 2
- name: Bill Gates
- start_date: Thu Apr 03 1975 20:00:00 GMT-0400 (EDT)
Found row:
- employee_id: 3
- name: Larry Page
- start_date: Thu Sep 03 1998 20:00:00 GMT-0400 (EDT)

```

Now, filter this down to only Bill Gates. To do that, you need to add a `WHERE` clause in your SQL statement. You'll also structure things so that you can correctly inject parameters into the SQL query—a generally good practice to avoid SQL injection attacks. Any variable data you use in a query should always be properly escaped, as the following listing shows.

Listing 6.7 Using parameter substitution on a SQL query

```

const spanner = require('@google-cloud/spanner')({
  projectId: 'your-project-id'
});

const database = spanner.instance('test-instance').database('test-database');
const query = {
  sql: 'SELECT employee_id, name, start_date FROM employees
    WHERE employee_id = @id',
  params: {
    id: 2
  }
};

database.run(query).then((data) => {
  const rows = data[0];
  rows.forEach((row) => {
    console.log('Found row:');
    row.forEach((column) => {
      console.log(' - ' + column.name + ': ' + column.value);
    });
  });
}
);

```

After running this, you'll see only one row in the results, including Bill Gates:

```
Found row:  
- employee_id: 2  
- name: Bill Gates  
- start_date: Thu Apr 03 1975 20:00:00 GMT-0400 (EDT)
```

Now let's look at what happens when you decide you want to store different information in your tables and have to change your schema.

6.4.5 **Altering database schema**

As your applications grow and evolve over time, you may find the need to change the structure of the data that you store. Like any other relational database, Spanner supports schema alterations, but you must be aware of a few caveats. Let's run through some of the things that are easy and obvious, and then we'll look at some of the more complicated changes.

First, the most basic change to a database is adding a new table. As you've seen already, this type of operation (`CREATE TABLE`) works as you'd expect. Similarly, deleting entire tables (`DROP TABLE`) works as expected, though there is a limitation related to child tables, which we explore later in the chapter.

You can modify tables in many of the ways you'd expect, though a few prerequisites exist for what types of changes are allowed. First, the new column can't be a primary key. This should be obvious because you can have only one primary key, and it's required when you create the table. Next, the new column can't have a `NOT NULL` requirement. This is because you may already have data in the table, and those existing rows clearly don't have a value for the new column and need to be set to `NULL`.

Columns themselves can also be modified, with similar limitations involved when adding new columns. You can perform three different types of column alterations:

- Change the type of a column from `STRING` to `BYTES` (or `BYTES` to `STRING`).
- Change the size of a `BYTES` or `STRING` column, so long as it's not a primary key column.
- Add or remove the `NOT NULL` requirement on a column.

In these situations, the limitations are related to data validation. For example, if you try to apply a `NOT NULL` limitation to a column that currently has rows where that column is set to `NULL`, the schema alteration fails because the data won't fit with the altered column definition. Because all of the data must be checked against the new schema definition, these types of alterations can take a long time, so it's not a great idea to do these often.

Let's take this for a spin, but this time, you'll use the Cloud SDK's command-line tool (`gcloud`) to execute your queries and alter your schema. A simple and common task is to increase the length of a string column, so take your `employees` table and increase the length of the `name` column from 100 characters to the maximum supported, which is

denoted by a special value: MAX (with a maximum limit per column of 10 MiB). The query you need to run is shown next.

Listing 6.8 SQL query to support longer employee names

```
ALTER TABLE employees ALTER COLUMN name STRING(MAX) NOT NULL;
```

To run this, you'll use the `gcloud spanner` subcommand and request alterations using Spanner's DDL (data definition language), as shown in the following listing.

Listing 6.9 Using the Cloud SQL to execute the schema alteration

```
$ gcloud spanner databases ddl update test-database \
--instance=test-instance \
--ddl="ALTER TABLE employees ALTER COLUMN name STRING(MAX) NOT NULL"
DDL updating...done.
```

If you go back the Cloud Console to look at your table, shown in figure 6.12, you'll see that the column has a new maximum length.

employees

[Schema](#) [Indexes](#) [Preview](#)

Column	Type	Nullable
employee_id	INT64	No
name	STRING(MAX)	No
start_date	DATE	Yes

[Show equivalent DDL](#)

Figure 6.12 The employees table after the alteration has been applied

Now we've covered the basics you should know about Spanner. But none of the things we've described does anything more than demonstrate how Spanner is similar to a traditional relational database like MySQL. To understand where Spanner shines, we'll need to explore more, so let's dive right into the advanced concepts that show the real power of Spanner.

6.5 Advanced concepts

Although the basic concepts you've learned so far are enough to get you going with Cloud Spanner, to use it effectively and at the enormous scale for which it was designed, you'll need to understand quite a bit more about how it blends a traditional relational database with a large-scale distributed storage system. Let's start by looking at the schema-level concept of interleaving tables with one another.

6.5.1 Interleaved tables

In a typical relational database, such as MySQL, the data itself is flat. When you store a row, it tends to have a unique identifier and then some data, but the only hierarchical relationship is between the row and the table (the row belongs to the table). Cloud Spanner supports additional relational aspects, which are sometimes explained as relationships between tables themselves, with one table belonging to another. This might sound weird at first, so we'll take a brief detour to explore one of the problems that comes up when databases experience heavy loads.

When you have a large amount of data or a large number of requests for data, sometimes a single server can't handle it. One of the first steps to fix this is to create read replicas, which duplicate data and act as alternative servers to query for the data. This solution is often the best one for systems that have heavy read load (lots of people asking for the data) and relatively light write load (modifications to the data), because read replicas do what their name says: act as duplicate databases that you can read from (see figure 6.13). All changes to the data still need to be routed through the primary server, which means it's still the bottleneck of your database.

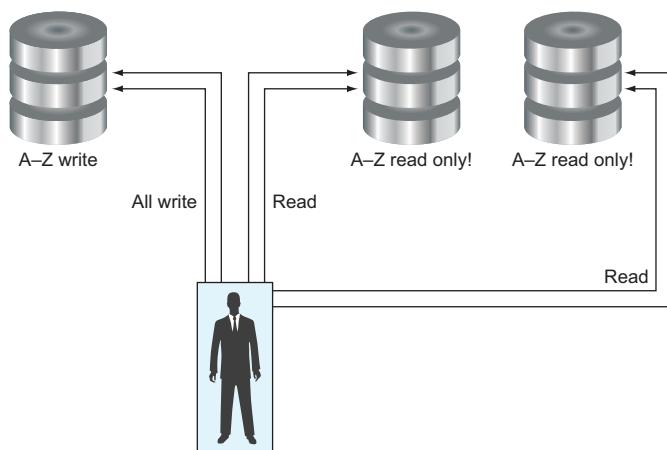


Figure 6.13 Using a read replica means one database is responsible for all writes.

What happens if you have a lot of modifications? Or if your database is getting so large that it won't easily fit on a single server? In that case, a read replica is unlikely to fix the problem for you, because it needs to duplicate all of the data.

In this situation, a common solution is to shard the data across multiple machines. Instead of creating many different machines, each with a full copy of the data—but only one capable of modifying that data—you instead chop up the data into distinct pieces and delegate responsibility for different chunks to different machines (see figure 6.14). For example, given an employees table that stores employee information,

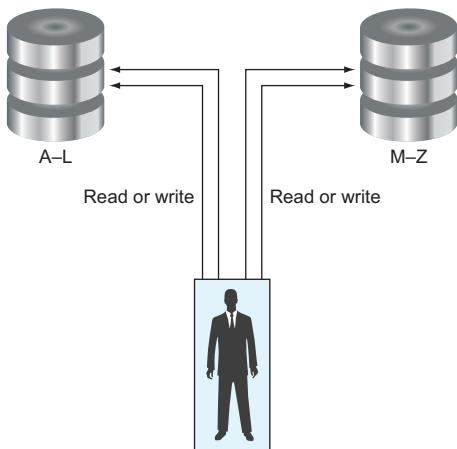


Figure 6.14 Using data shards splits the read and write responsibility

you might put data for employees with names in the range A through L on one server and M through Z on another server. By doing this, you've doubled your capacity as long as someone doing the querying can figure out how to find the right data. To make this concrete, before this sharding, a query for two employees (say, Steve Jobs and Mark Zuckerberg) would have been handled by a single machine. If the database is split as described earlier, these two queries would be handled by two different machines.

That example sounds easy because we focused on a single table (`employees`). But you also need to make sure that, for example, paycheck information, insurance enrollment, and other employee data in different tables are similarly chopped up. In addition, you'd want to make sure that all of the data is consistently split, particularly when you want to run a `JOIN` across those two tables. If you want to get an employee's name and the sum of their last 10 paychecks, having the paycheck data on one machine and the employee data on another would mean that this query is incredibly difficult to run.

Even worse, what about when you need even more serving capacity? Doing this process again to split the range into three pieces (say, A through F, G through O, and P through Z) is a pain, and you don't want to have to do this whenever your query load changes. Even more perplexing is that this design assumes all users have the same amount of traffic asking for their data. What if it turned out that two users (say, the Kardashians) are responsible for 80% of the traffic? In that case, it might make sense to give each of those their own server and then segregate the rest of the data evenly as described earlier.

Wouldn't it be nice if your database could figure this out for you? That way, instead of chopping up your data manually, you could rely on it being dynamically split up and shifted around to ensure your resources are being used optimally. Spanner does this with interleaved tables.

Splitting up the data is easy for Spanner to do. In fact, Bigtable has supported this capability for quite some time. What's unique is the idea of being able to provide hints to Spanner of where it should do the splitting, so that it doesn't do crazy things like put an employee's paycheck and insurance information on two separate machines.

You use interleaving tables to tell Spanner which data should live near and move with other data, even if that data is split across multiple tables. In the previous example, the `employees` table might be a parent table, and the others (storing paycheck or insurance information) would be interleaved within the `employees` table as child tables. Note also that the `employees` table has no more parents, so it's considered a root table.

Let's look at this a bit more concretely to see how it works by using some demonstration tables. In a traditional layout, storing employees and their paycheck amounts would involve separate tables, with a foreign key pointing from the `paychecks` table to the `employees` table (in this case, the User ID column). See table 6.1.

Table 6.1 Typical structure to store employee IDs and paycheck amounts

Employees		Paychecks			
ID	Name	ID	User ID	Date	Amount
1	Tom	1	3	2016-06-09	\$3,400.00
2	Nicole	2	1	2016-06-09	\$2,200.00

As you learned, if you went to shard these tables by ID, it's possible that the paycheck information for a user (say, Nicole) would end up on one machine, but her employee record would end up elsewhere. This is an issue.

In Spanner, you can fix this by interleaving the two tables together. Where you want to convey that an employee and their corresponding paychecks should be located near each other and move around together, your data would look somewhat different, as shown in table 6.2.

Table 6.2 Employee IDs interleaved with paychecks

ID	Name	Date	Amount
Employees(1)	Tom		
Employees(2)	Nicole		
Paychecks(2, 2)		2016-06-09	\$2,200.00
Employees(3)	Kristen		
Paychecks(3, 1)		2016-06-09	\$3,400.00

An equivalent representation with the IDs separated would look something like table 6.3.

Table 6.3 Alternative key style of employees interleaved with paychecks

Employee ID	Paycheck ID	Name	Date	Amount
1		Tom		
2		Nicole		
2	2		2016-06-09	\$2,200.00
3		Kristen		
3	1		2016-06-09	\$3,400.00

As you can see, related data is put together, even though this means that data from two different tables aren't separated. This layout also means that the ID fields become condensed, so let's look in more detail at what those keys are.

6.5.2 Primary keys

Though not required in a typical relational database, it's good practice to give each row what's called a *primary key*. Often this key is numeric (though that isn't required). The value has a uniqueness constraint, meaning that duplicate values aren't permitted, so the primary key can be used for indexing and addressing a single row. In Spanner, the primary key is required, but rather than being a single field, it can comprise multiple fields, as you saw in the previous example of the interleaved tables.

In the next listing, let's look at the same example (employees and paychecks), but instead of relying on an example table, we'll take a peek at the underlying SQL-style query that defines the schema and see what each piece does.

Listing 6.10 Example schema for the employees and paychecks tables

```

CREATE TABLE employees (
    employee_id INT64          NOT NULL,
    name        STRING(1024)    NOT NULL,
    start_date  DATE            NOT NULL
) PRIMARY KEY(employee_id);

```

Define the ID for each employee.
Call it `employee_id` (rather than `id`) for clarity in the future.


```

CREATE TABLE paychecks (
    employee_id   INT64 NOT NULL,
    paycheck_id   INT64 NOT NULL,
    effective_date DATE NOT NULL,
    amount_cents  INT64 NOT NULL
) PRIMARY KEY(employee_id, paycheck_id),
INTERLEAVE IN PARENT employees ON DELETE CASCADE;

```

Define that the `employee_id` field is the primary key for this table. This means that it must be unique and used to identify a given row.

In the paychecks table, track the employee's ID as well as the ID of the paycheck, similar to how you had the fields defined in a typical relational database.

Unlike in a typical relational database, rather than defining a foreign key relationship (pointing from `employee_id` in `paychecks` to `employee_id` in `employees`), make the relationship a part of the compound primary key.

To clarify that the `paychecks` table should be kept near the `employees` table, use the `INTERLEAVE IN PARENT` statement and specify that if an employee is deleted, the `paychecks` should also be deleted.

This example shows two tables: `employees` and `paychecks`. Each employee has an ID and a name, whereas each paycheck has an ID, a pointer to the employee (the employee's ID), a date, and an amount. This should feel familiar, but there are two important things to notice:

- Primary keys can be defined as a combination of two IDs (e.g., `employee_id` and `paycheck_id`).
- When interleaving tables, the parent's primary key must be the start of the child's primary key (for instance, the `paychecks` primary key must start with the `employee_id` field) or you'll get an error.

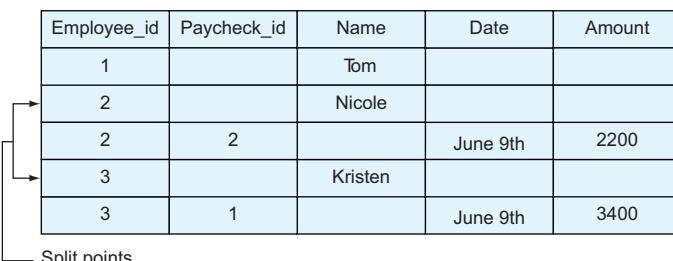
Now recall the idea of sharding data into chunks and splitting it across servers. We said that by interleaving tables the related data would be kept together, but we didn't dive into how that works. Let's take a moment to walk through how data is divided up using something called split points, because this method can have some important performance implications.

6.5.3 Split points

As the name suggests, split points are the exact positions at which data in a table might be split into separate chunks and potentially handed off to another machine to cope with request load or data size. So far we've said where we don't want data to be split and demonstrated that in our schema by interleaving the paycheck data with the employee data. By using a compound primary key in the `paychecks` table, you've said that all paychecks of each employee should be kept alongside the record for the parent employee.

Notice, however, that you haven't clarified how exactly data *can* be split. You've never said which employees can be separated and handed off. Spanner makes a big assumption: if you didn't say that things must stay together, they can and may be split. These points that you haven't specifically prohibited, which lie between two rows in a root table, are called split points.

Let's look at your example table of employees and paychecks again and see where the split points are. Recall that a root table is a table without a parent, which in this case is your `employees` table. Split points occur between every two different primary keys belonging to the root table, so split points exist before every unique employee ID, as shown in figure 6.15.



The diagram shows a table with five rows. The first four rows have Employee_id values of 1, 2, 2, and 3 respectively. The fifth row has an Employee_id value of 3. A vertical bracket on the left side of the table is labeled "Split points". Two arrows point from this bracket to the empty space before the second and third rows, indicating that split points occur before every unique employee ID.

Employee_id	Paycheck_id	Name	Date	Amount
1		Tom		
2		Nicole		
2	2		June 9th	2200
3		Kristen		
3	1		June 9th	3400

Split points

Figure 6.15 Split points between every unique employee ID

Notice that all records with the same employee ID at the start of the primary key will be kept together, but each chunk of records can be shifted around as necessary. For example, it's possible that employees 1, 2, and 3 could be on different machines, but paycheck 2 will be on the same machine as employee 2, and paycheck 1 will be on the same machine as employee 3.

NOTE If you read chapter 5, you should notice some similarities. In this case, Datastore has the same concept but talks about *entity groups* as the indivisible chunks of data, whereas Spanner talks about the points *between* the chunks and calls them *split points*.

This leads us to one final topic on this tricky business of interleaving tables, split points, and primary keys: choosing a good primary key.

6.5.4 Choosing primary keys

You might ask, “Choosing a primary key? Why not use numbers?” And you’re not crazy. Choosing primary keys isn’t something you typically do in a relational database. For example, MySQL offers a way to specify that fields should be automatically incremented, so if you omit the field, it will be substituted by the highest value incremented by one. But Spanner works differently.

Spanner keeps all of the data in the database sorted lexicographically by primary key, keeping sequential data together. Although it divides data only on split points between these chunks (for example, between different employees), employees 10 and 11 will be next to each other (unless Spanner has decided to divide them up at the split point between the two).

This might seem like no big deal, but it’s powerful because you can distribute your writes evenly across the key space (and therefore across your Spanner infrastructure) by choosing keys that are evenly distributed. But you can effectively cripple yourself if you choose keys that all happen to hit a single Spanner node. In the next listing, let’s look at a classic example of a terrible primary key to use: timestamps.

Listing 6.11 Example schema using a timestamp

```
CREATE TABLE events (
    event_time TIMESTAMP NOT NULL,
    event_type STRING(64) NOT NULL
) PRIMARY KEY(event_time);
```

Let’s imagine that you had millions of sensors broadcasting events and the total request rate was one write every microsecond (that’s 60,000 writes per second). Spanner should be able to handle that, right? Not so fast. Think about what happens when Spanner tries to deal with this scenario.

First, lots of traffic is coming to a single node because each event is only one microsecond away from the previous one. To deal with this overload, Spanner picks a split point (in this case, between any two events because this is a root table) and chops the

data in half. Half of the data will have IDs as timestamps *before* the split point and the other half *after* the split point. Now more traffic comes in. Can you guess which side will be responsible for the new rows?

All the new rows are guaranteed to have IDs as timestamps after the split point, because time continues to count upward! This means you're right back where you started with a single node handling all of the traffic. If you do this same process again, you'll notice that it continues to *not* fix the problem. This problem, which happens quite often, is called hot-spotting—you've created a hot spot that's the focus of all the requests.

The moral of this story is that when writing new data, you should choose keys that are evenly distributed and never choose keys that are counting or incrementing (such as A, B, C, D, or 1, 2, 3). Keys with the same prefix and counting increments are as bad as the counting piece alone (for example, sensor1-<timestamp> is as bad as using a timestamp). Instead of using counting numbers of employees, you might want to choose a unique random number or a reversed fixed-size counter. A library, such as Groupon's `locality-uuid` package (see <https://github.com/groupon/locality-uuid.java>), can help with this.

Now that you understand all of these concepts of data locality, choosing primary keys, split points, and interleaving tables, let's explore how and why you might want to use indexes on your tables.

6.5.5 Secondary indexes

For many of us, indexes are something we add later when our database gets slow. Though that description is somewhat accurate (and often practical), indexes are an important performance tool for a database. Let's take a moment to review how indexes work, and then we'll dig into how Spanner uses them to speed up queries.

Indexes tell your database to maintain some alternative ordering of data in addition to the data already stored in the database. For example, instead of storing the list of employees sorted by their primary keys, you might want the database to store a list of employees sorted by their name as well.

If you have data sorted by a column that you intend to filter on (for example, `WHERE name = "Joe Gagliardi"`), the search on that column can be done much more quickly. Searching an ordered list is much faster than searching an unordered list for a variety of reasons.

Imagine I asked you to find everyone in the phone book with the name “Richard Feynman (Feynman, Richard). Easy, right? This is because the phone book’s primary key is (`last name, first name`). Imagine instead that you had to find everyone in the phone book with the first name Richard and a phone number ending in 5691. This query would likely take a while because the phone book doesn’t have an index for those fields. To do this query, you’d have to scan through all of the records in the phone book, which might take a while. Why wouldn’t you index everything? Wouldn’t that make all of your queries faster?

Although indexes can make queries of your data run more quickly, those indexes also need to be updated and maintained. Searching for a specific person by name might be faster thanks to the index on the employee names. Whenever you update an employee’s name (or create a new employee), however, you need to update the row in the table along with the data in each index that references the name column. If you don’t, the data will get out of sync and strange things might happen, such as a query returning a matching row that ends up not matching after all.

If you added an index on employee names to make those lookups and filters faster, updating a name would now involve writes to two different resources: the table itself and the index you created. In short, you’re exchanging slightly more work being done at write time for much less work needing to be done at read time.

Indexes also take up extra space. Though the size at first may be no big deal, as you add more and more data the total space consumed can become significant. Imagine how large the phone book would be if it had both the regular data (by last name) and the index from the previous example (first name and phone number). You might not have to store all the pictures in the index, but it would certainly have exactly the same number of entries as those that are in the phone book.

How do you decide when to add an index? This can get complicated—there are entire books on the subject—but in general you should start by looking at the queries you need to run against the database. Although the shape of your data will influence the schema of your tables, it’s the queries you run that will influence the indexes you need. If you understand the queries you’re running, you can see exactly what types of indexes you need and add them as needed (or remove them when they become unnecessary). It’s best to walk through this using more clear examples, so let’s look at Spanner’s take on secondary indexes and expand the example from earlier with employees and paychecks.

Spanner’s idea of secondary indexes is close to other common relational databases. Without them, Spanner queries execute completely but may be slower than usual, and with them, writes have extra work to do, but queries should get faster. A couple of key differences stem from the concept of interleaved tables that we explored previously. Let’s start by looking at some of the similarities.

In the current database schema (with a paychecks table interleaved in an employees table), you’ll want to do lookups and searches by an employee’s name. Running this query, however, will involve a full table scan (looking at every row to be sure that you’ve found all matches) as shown in figure 6.16. To see this, you can run a query that does a name lookup from the Cloud Console and look at the Explanation tab to see that the query starts off with a table scan.

Make this faster by creating an index on the name column, using a DDL statement, as shown in the following listing.

Listing 6.12 Schema alteration to add an index to the employees table

```
CREATE INDEX employees_by_name ON employees (name)
```

Query database: test-database

The screenshot shows a Cloud Spanner query editor interface. At the top, there is a code editor containing the following SQL query:

```
1 select employee_id from employees where name = "Larry Page"
```

Below the code editor are three buttons: "Run query" (highlighted in blue), "Clear query", and "SQL query help". To the right, it says "Run query: Ctrl + Enter".

Under the query results, there are two tabs: "Results table" and "Explanation" (which is currently selected). Below these tabs, there is a summary table with four columns:

Total elapsed time 16.24 msecs	CPU time 14.19 msecs	Rows returned 0	Rows scanned 0
-----------------------------------	-------------------------	--------------------	-------------------

At the bottom of the interface, there are links for "Operator reference" and "Guided tour".

Figure 6.16 Finding employees by name without an index results in a table scan.

You can use the gcloud command like you did earlier to create the index, as the next listing shows.

Listing 6.13 Create the index at the command line

```
$ gcloud spanner databases ddl update test-database \
--instance=test-instance \
--ddl="CREATE INDEX employees_by_name ON employees (name)"
DDL updating...done.
```

After the index is created, you should see it in the Cloud Console (see figure 6.17).

The fun part comes from rerunning that same query to find a specific employee by name. As shown in figure 6.18, the results now rely on your newly created index rather than on scanning through the entire table.

Something strange happens when you alter this query to ask for more than the employee ID. If you run a query for `SELECT * FROM employees WHERE name = "Larry Page"`, the explanation says that you're back to using the table scan. What happened? Why didn't it use the index that you have?

employees_by_nameTable indexed: [employees](#)**Columns indexed**

Column	Sort order
name	Ascending

[Show equivalent DDL](#)

Figure 6.17 The newly created index on employee names

Query database: test-database

```
1 select employee_id from employees where name = "Larry Page"
```

[Run query](#)
[Clear query](#)
[SQL query help](#)
Run query: Ctrl + Enter

Results table	Explanation		
Total elapsed time 3.16 msecs	CPU time 1.46 msecs	Rows returned 1	Rows scanned 1

[Operator reference](#) | [Guided tour](#)

Operator	Rows returned	Executions	Latency
▀ Distributed union	1	1	0 ms
↑ Local distributed union	1	1	0 ms
↑ Serialize Result	1	1	0 ms
‡ Index Scan: employees_by_name ▾	1	1	0 ms

Figure 6.18 Spanner uses the new index to execute the query.

Your index was specific about exactly what data is being stored—in this case, the primary key (that's always stored) and the name. If all you want is the primary key and the name (which is all your first query asked for), then the index is sufficient. If you ask for data that isn't in the index, using the index itself won't be any faster because after you've found the right primary keys that match your query, you still have to go back to the original table to get the other data (in this case, the start_date).

Let's imagine that you often run a query that asks for the `start_date` of an employee where you filter based on a name: `SELECT name, start_date FROM employees WHERE name = "Larry Page"`. To make that query fast, you have to pay a storage penalty. To rely on an index to handle the lookup, you also need to ask the index to store the `start_date` field, even though you don't want to filter on it. Spanner does this by adding a simple `STORING` clause at the end of the DDL when creating the index, as shown in the following listing.

Listing 6.14 Creating an index, which stores additional information

```
CREATE INDEX employees_by_name ON employees (name) STORING (start_date)
```

After you add this index, running a query like the one in listing 6.14 uses the newly created index (see figure 6.19). In contrast, a query filtering on a specific ID (such as `SELECT name, start_date FROM employees WHERE employee_id = 1`) will still rely on a table scan, but that's the fastest kind of scan because it's a primary key lookup.

Now that you have your feet wet creating and modifying indexes, let's look at how this relates to the previous topics of interleaved tables. Like you can interleave one table into another, indexes can similarly be interleaved with a table. You end up with a

Query database: test-database

The screenshot shows the Google Cloud Spanner Query interface. At the top, there is a code editor containing the following SQL query:

```
1 SELECT name, start_date FROM employees WHERE name = "Larry Page"
```

Below the code editor are three buttons: "Run query" (highlighted in blue), "Clear query", and "SQL query help". To the right of the buttons is the text "Run query: Ctrl + Enter".

Under the query results, there are two tabs: "Results table" and "Explanation" (underlined). The "Explanation" tab is selected, showing the following table:

Total elapsed time 1.69 msecs	CPU time 0.74 msecs	Rows returned 1	Rows scanned 1
----------------------------------	------------------------	--------------------	-------------------

Below the table are two links: "Operator reference" and "Guided tour".

The "Explanation" section displays the execution plan with the following rows:

Operator	Rows returned	Executions	Latency
Distributed union	1	1	0 ms
Local distributed union	1	1	0 ms
Serialize Result	1	1	0 ms
Index Scan: employees_by_name_with_start_date	1	1	0 ms

Figure 6.19 Spanner now can rely on the index for the entire query.

local index that's applied within each row of the parent table. This is a bit tricky to follow, so let's look at some examples where you want to see paycheck amounts.

If you want to look at the paychecks sorted by amount, as shown in the next listing, the query would be across all employees, so this query would be what's called *global*.

Listing 6.15 Querying for paychecks across all employees

```
SELECT amount_cents FROM paychecks ORDER BY amount_cents
```

If you wanted the same information but only for a specific employee, the query is only across the paychecks belonging to a single employee, as shown in the listing 6.16. Because the paychecks table is interleaved into the employees table, you can think of this query as local because it's scanning only a subset of rows, whittled down by your employee criteria, which you've already designated as rows you want to keep near one another.

Listing 6.16 Querying for paychecks of a single employee

```
SELECT amount_cents FROM paychecks
  WHERE employee_id = 1 ORDER BY amount_cents
```

If you were to look at the explanation of both of these queries, you'd see that they both involve a table scan over the paychecks table. What indexes would make these faster?

For your first global query, having an index across the paychecks table on the amount_cents column would do the trick. But for the second one, you want to take advantage of the fact that paycheck entries are interleaved in employee entries. To do this, you can interleave the index in the parent table and get a local index that will work when you look within rows in a child table that are filtered by a row in a parent table.

In this case, the two indexes would look quite similar, the difference being an additional row in the index (employee_id) and the fact that the index itself would be interleaved with employee records, like the paycheck records themselves. See the following listing.

Listing 6.17 Create two indexes, one global and one local

```
CREATE INDEX paychecks_by_amount ON paychecks(amount_cents);

CREATE INDEX paychecks_per_employee_by_amount_interleaved
  ON paychecks(employee_id, amount_cents),
  INTERLEAVE IN employees;
```

If you were to rerun the same query, the explanation would say that this time the query relied on your interleaved index.

Why would you care about interleaving the index in the employees table? Why not create the index on those fields and leave out that `INTERLEAVE IN` part? Technically,

that's a valid index; however, it loses out on the benefits of colocating related rows near to each other. Updates to a paycheck record may be handled by one server, and the corresponding (required) update to the index may be handled by another server. By interleaving the index with the table in the same way that paycheck records are interleaved, you guarantee that the two records will be kept together and keep updates to both close by one another, which improves overall performance.

As you can see, indexes are incredibly powerful, but they can be a double-edged sword. On the one hand, they can make your queries much faster by virtue of having your data in exactly the format you need. On the other hand, you must be willing to pay the cost of having to update them as your data changes and store additional data as needed to avoid further table scans.

Figuring out what indexes are most useful can be tricky, and entire books are devoted to how best to index your data. The good news is that when you run queries against Spanner, it will automatically pick the one that it thinks will be the fastest unless you specifically force it to use an index. You can do this with the `force_index` option on the statement; for example, `SELECT amount_cents FROM paychecks@ {force_index=paychecks_by_amount}`. Generally it's better to allow Spanner to choose the best way of running queries. Now that we've gone through the basics of indexing in Spanner, let's explore something equally important: transactional semantics.

6.5.6 Transactions

If you've worked with a database (or any storage system), you should be familiar with the idea of a transaction and the acronym that tends to define the term: ACID. Databases that support ACID transactional semantics are said to have *atomicity* (either all the changes happen or none of them do), *consistency* (when the transaction finishes, everyone gets the same results), *isolation* (when you read a chunk of data, you're sure that it didn't change out from under you), and *durability* (when the transaction finishes, the changes are truly saved and not lost if a server crashes). These semantics allow you to focus on your application and not on the fact that multiple people might be reading and writing to your database at the same time.

Without support for transactions, all sorts of problems can occur, from the simple (such as seeing a duplicate entry in a query) to the horrifying (you deposit money in a bank account and your account isn't credited). Being a full-featured database, Spanner supports ACID transactional semantics, even going as far as supporting distributed transactions (although at a performance cost). Spanner supports two types of transactions: read-only and read-write. As you might guess, read-only transactions aren't allowed to write, which makes them much simpler to understand, so we'll start there.

READ-ONLY TRANSACTIONS

Read-only transactions let you make several reads of data in your Spanner database at a specific point in time. You never have to worry about getting a "smear" of the data spread across multiple times. For example, imagine that you need to run one query,

do some processing on that data, and then query again based on the output of that processing. By the time that you run the second query, it's possible that the underlying data has changed (for example, some rows may have been updated or deleted), and your queries might not make sense anymore! With read-only transactions, you can be sure that the data hasn't changed because you're always reading data at a specific point in time.

A read-only transaction doesn't hold any locks on your data and, therefore, doesn't block any other changes that might be happening (such as someone deleting all the data or adding more data). To demonstrate how this works, let's look at a sample querying your employee data in the next listing.

Listing 6.18 Querying data from inside and outside a transaction

This is a helper function that gets the row counts from two connections: one from the transaction provided and the other from the database outside of the transaction.

```
const spanner = require('@google-cloud/spanner')({
  projectId: 'your-project-id'
});
const instance = spanner.instance('test-instance');
const database = instance.database('test-database', {max: 2});
```

Because the client uses a session pool to manage concurrent requests, make sure that you're using more than a single session (in this case, you'll use two).

```
const printRowCounts = (database, txn) => {
  const query = 'SELECT * FROM employees';
  return Promise.all([database.run(query), txn.run(query)]).then((results) => {
    const inside = results[1][0], outside = results[0][0];
    console.log('Inside transaction row count:', inside.length);
    console.log('Outside transaction row count:', outside.length);
  });
}

database.runTransaction({readOnly: true}, (err, txn) => {
  printRowCounts(database, txn).then(() => {
    const table = database.table('employees');
    return table.insert({
      employee_id: 40,
      name: 'Steve Ross',
      start_date: '1996-01-23'
    });
  }).then(() => {
    console.log(' --- Added a new row! --- ');
  }).then(() => {
    printRowCounts(database, txn);
  });
});
```

Start by creating a read-only transaction.

Count all the rows from both inside and outside the transaction.

From outside of the transaction, create a new employee in your table.

Count all the rows again from both inside and outside the transaction.

In this script, you're demonstrating how your transaction maintains an isolated view of the world, despite new data showing up from other people accessing (and writing to) the database. To be more specific, the inside counts should always remain the same

(“inside” being the row count as seen by queries run from the `txn` object), regardless of what’s happening outside. Queries from outside the transaction, however, should see the newly added row when running the query. To see that this works, run the previous script. You should see output that looks like this:

```
$ node transaction-example.js
Inside transaction row count: 3
Outside transaction row count: 3
    --- Added a new row!
Inside transaction row count: 3
Outside transaction row count: 4
```

As you can see, your inside counts always stayed the same (at 3), whereas the outside counts (the query run from outside our transaction) see the new row after it’s committed. This demonstrates that read-only transactions act as containers for reads at a point frozen in time. Additionally, because a read-only transaction holds no locks on any of the data, you can create as many as you want and everything should work as expected. Because of these properties, sometimes it makes sense to think of a read-only transaction as an additional filter on your data, as the following listing shows.

Listing 6.19 Example of the implicit restriction of queries run at a specific time

```
SELECT <columns> FROM <table> WHERE <your conditions> AND
    run_query_frozen_at_time = <time when you started your transaction>
```

This concept of freezing time is easy to understand and has almost none of those pesky what-if scenarios. But read-write transactions are more complicated, so let’s take a look at how they work.

READ-WRITE TRANSACTIONS

As the name suggests, read-write transactions are transactions that both read and modify data stored in Spanner. These transactions tend to be the important ones that prevent you from doing things like losing an ATM deposit by operating on data that changed, so it’s important to understand how they work and how to use them correctly.

Imagine you found a mistake in employee 40’s paycheck—it’s \$100 less than it should be. To make this change using Spanner’s API, you need to do the following two things:

- 1** Read the amount of the paycheck.
- 2** Update the amount of the paycheck to `amount + $100`.

This might seem boring, but in a distributed system where you may have lots of people all doing things at once (some of them potentially conflicting with what you want to do), this task can become quite difficult. To see this, let’s imagine that two jobs are running at once to update paychecks. One job is fixing an error where all paychecks were \$100 less than expected, and another is fixing an error where a \$50 fee wasn’t

taken out. If you run these jobs serially (one after another), everything should work fine. Also, if you combine these jobs (turn them into one job that adds \$50), things will also work out fine. But those options aren't always available, so for this example, let's imagine them running side by side.

The problems begin to arise when both jobs happen to operate on the same paycheck at almost the same time. In those scenarios, it's possible that one job will overwrite the work of the other, resulting in either *only* a \$100 paycheck increase or *only* a \$50 paycheck decrease, rather than both (see figure 6.20).

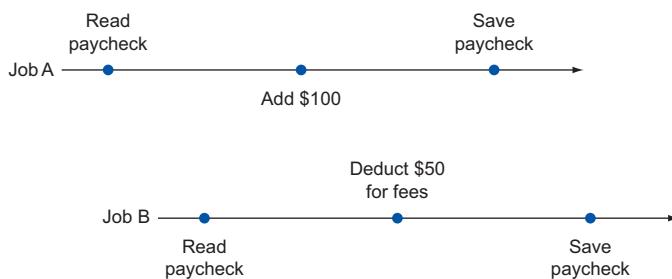


Figure 6.20 Example of the fee-deducting job overwriting the \$100-increase job

To fix this, you need to lock certain areas of the data to tell other jobs, “Don’t mess with this—I’m using it.” This is where Spanner’s read-write transactions save the day. Read-write transactions provide a way of locking exactly what you need, even when there’s a close overlap of data. In the time line described earlier, job A’s write would complete, and when job B tries to save the changes, it will see a failure and be instructed to retry.

Read-write transactions also guarantee atomicity, which means that the writes done inside the transaction either all happen at the same time or don’t happen at all. For example, if you wanted to transfer \$5 from one paycheck to another, you perform two operations: deduct \$5 from paycheck A, and add \$5 to paycheck B. If those two don’t happen atomically, it means that one part of the process could happen and be saved without its corresponding partner, which would result in either disappearing money (\$5 deducted but not transferred) or free money (\$5 added but not deducted).

Additionally, reads inside a read-write transaction see all data that has been committed before the transaction itself commits. If someone else modifies a paycheck after your transaction starts, everything will work as expected as long as you read the data after that other transaction commits. To see this in action, let’s look at two examples of overlapping transactions, one failing and one succeeding.

Transactions guarantee that all reads happen at a single point in time (as I explained in the section on read-only transactions), but they also guarantee that a transaction fails if any of the data read became stale during the life of the transaction. In this case,

if you read some data at the start of a transaction, and another transaction commits a change to that same data, the transaction will fail no matter what, regardless of what data you end up writing. In figure 6.21, transaction 2 is attempting to write the record of employee B based on a read of paycheck A. Between the read and the write, paycheck A is modified by transaction 1, meaning that paycheck A's data is out of date, and as a result the transaction must fail.

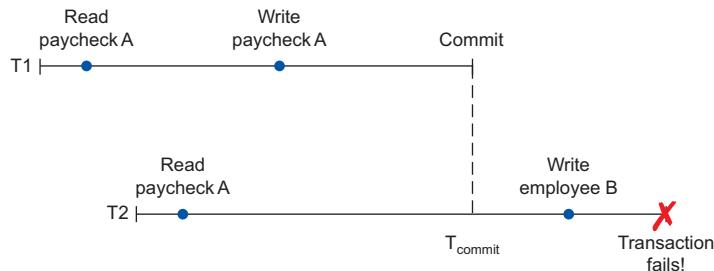


Figure 6.21 Transactions fail if any of the data read becomes stale.

On the other hand, transactions are smart enough to ensure that reading *any* data won't force your transaction to fail. If you were to read some data at the start of your transaction, then another transaction modifies some unrelated data, and then you read the data that was modified, your transaction can still commit successfully. See figure 6.22.

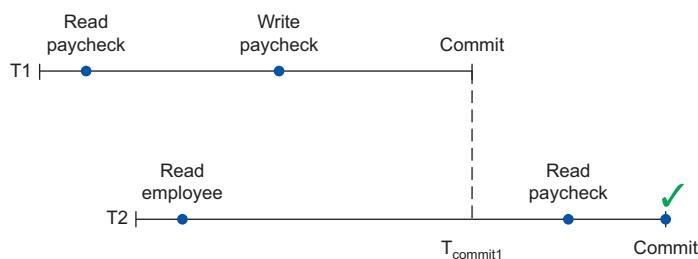


Figure 6.22 Reading data after it's been changed doesn't cause transaction failures.

To make things even better, data is locked on a cell level (a row and a column), which means that transactions modifying different parts of the same row won't conflict with one another. For example, if you read and update only the date of paycheck A in one transaction and then read and update only the amount of paycheck A in another transaction, even if the two overlap, they'll be able to succeed. See figure 6.23.

To see this in action, you're going to write some code that illustrates successful cell-level locking, as well as some that demonstrates failure, as shown in the following listing.

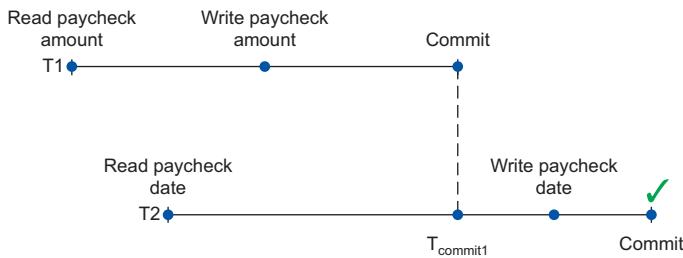


Figure 6.23 Example of cell-level locking avoid conflicts.

Listing 6.20 Non-overlapping read-write transactions touching the same row

This helper function prints out the data for employee 1 that's committed in Spanner (it doesn't include any uncommitted data).

```
const spanner = require('@google-cloud/spanner')({  
  projectId: 'your-project-id'  
});  
const instance = spanner.instance('test-instance');  
const database = instance.database('test-database', {max: 5});  
const table = database.table('employees');  
  
Promise.all([database.runTransaction(), database.runTransaction()]).then( ←  
  (txns) => {  
    const txn1 = txns[0][0], txn2 = txns[1][0]; ←  
    const printCommittedEmployeeData = () => {  
      const allQuery = {keys: ['1'], columns: ['name', 'start_date']};  
      return table.read(allQuery).then((results) => {  
        console.log('table:', results[0]);  
      });  
    }  
    const printNameFromTransaction1 = () => { ←  
      const nameQuery = {keys: ['1'], columns: ['name']};  
      return txn1.read('employees', nameQuery).then((results) => {  
        console.log('txn1:', results[0][0]);  
      });  
    }  
    const printStartDateFromTransaction2 = () => { ←  
      const startDateQuery = {keys: ['1'], columns: ['start_date']};  
      return txn2.read('employees', startDateQuery).then((results) => {  
        console.log('txn2:', results[0][0]);  
      });  
    }  
    const changeNameFromTransaction1 = () => { ←  
      txn1.update('employees', {  
        employee_id: '1',  
      })  
    }  
  }  
);  
Start by creating two transactions, both read-write.  
The results of the Promise.all() call are the two transaction objects.  
This helper function reads only the name of the employee through the first transaction (txn1).  
This helper function reads only the start date of the employee through the second transaction (txn2).  
This helper function changes only the name of the employee and commits the first transaction (txn1).
```

```
        name: 'Steve Jobs (updated)'  
    });  
    return txn1.commit().then((results) => {  
        console.log('txn1:', results);  
    });  
});  
  
const changeStartDateFromTransaction2 = () => {  
    txn2.update('employees', {  
        employee_id: '1',  
        start_date: '1976-04-02'  
    });  
    return txn2.commit().then((results) => {  
        console.log('txn2:', results);  
    });  
}  
  
printCommittedEmployeeData()  
    .then(printNameFromTransaction1)  
    .then(printStartDateFromTransaction2)  
    .then(changeNameFromTransaction1)  
    .then(changeStartDateFromTransaction2)  
    .then(printCommittedEmployeeData)  
    .catch((error) => {  
        console.log('Error!', error.message);  
    });  
};
```

This helper function changes only the start date of the employee and commits the second transaction (txn2).

This is the control flow, which ensures that these different functions are executed in order, so you can be sure of the overlap described.

As you learned earlier, despite these two transactions modifying the exact same row, the locking is at the cell level, so these two transactions don't overlap one another at all. This is specifically because there was no overlap in the cells read or modified. To see that this works as expected, if you run the script in listing 6.26, you'll see output looking something like this:

```
$ node run-transactions.js
table: [ [ { name: 'name', value: 'Steve Jobs' },
  { name: 'start_date', value: 1976-04-01T00:00:00.000Z } ] ]
txnl: [ { name: 'name', value: 'Steve Jobs' } ]
txn2: [ { name: 'start_date', value: 1976-04-01T00:00:00.000Z } ]
txnl: [ { commitTimestamp: { seconds: '1490101784', nanos: 765552000 } } ]
txn2: [ { commitTimestamp: { seconds: '1490101784', nanos: 817660000 } } ]
table: [ [ { name: 'name', value: 'Steve Jobs (updated)' },
  { name: 'start date', value: 1976-04-02T00:00:00.000Z } ] ]
```

This is pretty neat, but what if the second transaction also *read* the name value? Then the control flow would look something like the next listing.

Listing 6.21 Looking at the name causes the transaction to fail

```
const printNameAndStartDateFromTransaction2 = () => {
  const startDateQuery = {
```

```

keys: ['1'], columns: ['name', 'start_date'] };
return txn2.read('employees', startDateQuery).then((results) => {
  console.log('txn2:', results[0][0]);
})
}
/*
 * ... */

printCommittedEmployeeData()
  .then(printNameFromTransaction1)
  .then(printNameAndStartDateFromTransaction2) ←
  .then(changeNameFromTransaction1)
  .then(changeStartDateFromTransaction2)
  .then(printCommittedEmployeeData);

```

This helper function is almost identical to printStartDateFromTransaction2; however it also includes the name column.

Instead of printing only the start date, you'll also print the name value.

You've read an outdated version of the name value from the second transaction, so after the first transaction commits, the second will fail because you can't be sure that the second transaction didn't make any bad decisions based on stale data. The error result is shown next:

```

table: [ [ { name: 'name', value: 'Steve Jobs (updated)' },
  { name: 'start_date', value: 1976-04-02T00:00:00.000Z } ] ]
txn1: [ { name: 'name', value: 'Steve Jobs (updated)' } ]
txn2: [ { name: 'name', value: 'Steve Jobs (updated)' },
  { name: 'start_date', value: 1976-04-02T00:00:00.000Z } ]
txnl: [ { commitTimestamp: { seconds: '1490116055', nanos: 805223000 } } ]
Error! Transaction was aborted. It was wounded by a higher priority
transaction due to conflict on key [1], column name in table employees.

```

Transactional semantics and concurrency are both complicated, so there's far more information than we can go into in this chapter. Spanner's online documentation is pretty detailed, though, and worth a read. The general guideline when it comes to transactions is to be specific about the data you want from Spanner and put critical pieces that must be atomic inside transactions. Spanner can do the right thing to make sure that your queries execute both safely (correctly) and optimally (as fast and at the highest levels of concurrency possible).

Now let's move on from these more advanced topics and take a quick look at how much all of this will cost you.

6.6 ***Understanding pricing***

Cloud Spanner pricing has three different components: computing power, data storage, and network cost. Network cost is not typical in most Spanner configurations. Let's start by looking at the computing power.

Similar to Cloud SQL, Cloud Spanner is billed by the total number of nodes created and priced on an hourly basis, with variations in price depending on the location (for example, Asia tends to be more expensive than the central United States). Unlike Cloud SQL, the replication that happens under the hood is baked into the overall hourly price.

Spanner currently runs at \$0.90 US per node per hour (for a US-based instance), with a recommendation of a three-node instance for anything that needs production-level availability. All configurations are currently replicated across three different zones, meaning that in total, a three-node instance is nine total nodes (three-node replicas each in three different zones). To put this in perspective, the total monthly computing power cost for a three-node Cloud Spanner instance in the central United States works out to around \$2,000 US per month.

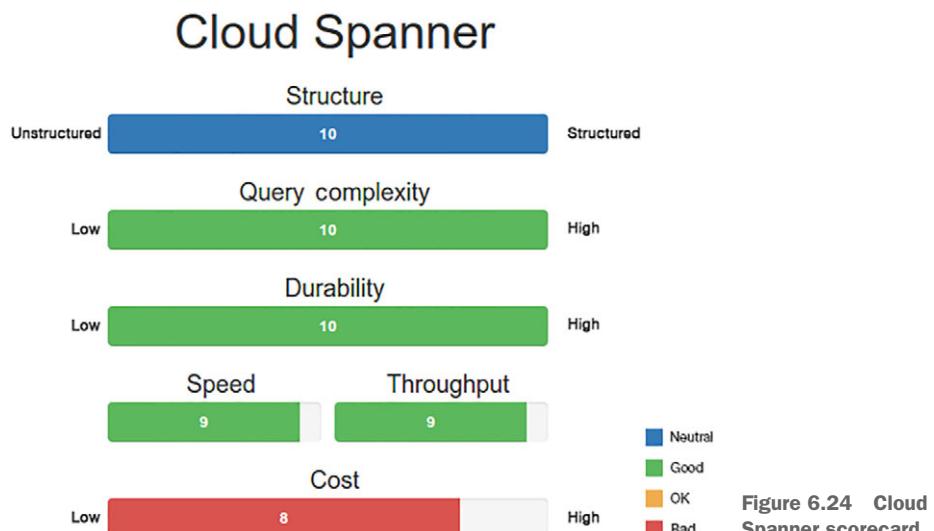
In addition to computing power, the data stored in Spanner is charged at a rate of \$0.30 US per month. Unlike Compute Engine's persistent disks, Spanner's storage space is measured based on how much data you have rather than a specific block of data that you've provisioned. For example, a Spanner database holding 1 TB of data would cost around \$300 per month.

Last, any data sent from Spanner to the outside world (to machines outside of Google's network) or between separate regions (from a Spanner instance in `asia-east1` to a Compute Engine VM in `us-central1-c`) is charged at global network rates, which varies by location but is currently \$0.01 US per GB in the United States.

Generally, when you're using Spanner, your queries send data to and from Compute Engine or App Engine instances in the same region, meaning that the network cost is complete free (those don't leave the Google Cloud network). This cost can become meaningful if you try to run an export of your data outside of Google Cloud or send lots of queries across multiple regions. Now that you understand how billing works, let's look at what factors make Spanner a good or bad fit for your projects.

6.7 When should I use Cloud Spanner?

Let's start by looking at the score card shown in figure 6.24, which summarizes the various criteria that you might care about.



6.7.1 Structure

Spanner is a full-featured SQL-style database—you define columns that have specific types and data is rejected if it doesn’t fit properly. This also includes NOT NULL modifiers, which makes certain columns required, so on the scale of how structured I’d consider Spanner, it’s as high as possible (alongside Cloud SQL or any other SQL database).

Spanner also imposes additional structure that’s not possible with traditional SQL databases—the ability to interleave a child table into a parent table. In a sense, if this scale could go any higher, Spanner would be right there at the top.

6.7.2 Query complexity

Spanner not only tops the charts on overall structure, it’s also up there when it comes to query complexity. Not only can you do single-key lookups and specify which columns you’re interested in, you can do arbitrarily complex SQL statements involving joins across tables, fancy groupings, and advanced filtering of rows. This level of query complexity is generally not available in other databases that are focused on providing high performance and availability, such as Cloud Bigtable, making this a powerful feature.

6.7.3 Durability

Like Cloud Datastore, Cloud Spanner is replicated across multiple different zones, which ensures that data, once persisted, doesn’t go anywhere. This is made explicit with the transactional semantics such that not only is there no need to worry about data loss, it’s also clear exactly when a transaction has been committed. You always have a consistent view of the world about what data is committed and what isn’t.

6.7.4 Speed (latency)

When it comes to overall query latency, Spanner’s key lookups are extremely fast. For other more complex queries, obviously there will be some additional latency, but in general, most queries to Spanner should complete within a few milliseconds. What’s impressive is that Spanner latency can be kept consistently fast even as request load increases so long as the number of nodes is turned on to handle the load. Should the latency increase, the fix is to turn on more nodes, which will split up the work and keep queries fast.

6.7.5 Throughput

Unlike object storage systems or file systems, Spanner’s benefit is not in how many bytes it can ship over the wire in a given amount of time (throughput), but in how quickly it can respond to a given query (latency). Further, the data stored in Spanner is generally large in overall size but smaller on a per-query basis. Although overall system throughput may be sufficiently large, the per-query throughput is not typically measured. (How often have you thought about how many MB per second could be sent out of your

MySQL instance?) Spanner as a whole is capable of large overall throughput and scores highly on the scale, in line with other systems like Cloud Bigtable.

6.7.6 Cost

With the overall cost being around \$650 US per node per month, and the general guideline being to have at least three nodes for any production traffic, Spanner comes in at the high end of the price range, costing almost \$2,000 US per month for the minimum suggested configuration—certainly more than a single small VM running a database (either unmanaged through GCE or managed using Cloud SQL), which costs around \$50 US per month.

The primary difference is the number of nodes, which is triple what is shown due to Spanner's full replication across three different zones. Looking at that in numbers, the true cost for a single node in a single zone is \$0.30 per hour, which comes to about \$200 per node per month. This adjustment puts you in the same overall price range as a four-core Cloud SQL machine (db-n1-standard-4), so if you were deploying a nine-node cluster of these machines, your monthly costs would come to \$1,750 per month, which is in the same range as Cloud Spanner. When you might use a large SQL cluster to handle your database traffic, Spanner would cost around the same amount and handle all of the management and replication for you automatically. In short, though Spanner does rank highly on the overall cost scale, this is primarily because you're getting so much computing power, which is masked by replication.

6.7.7 Overall

Now that you can see how Cloud Spanner works and where it shines, let's look through your sample applications (the To-Do List, InstaSnap, and Exchange) and see how they each stack up.

To-Do List

As you learned earlier, the To-Do List application is certainly not in need of either of the performance characteristics of Cloud Spanner (neither the super low latency or the high throughput). The structure offered will come in handy, but it seems like the other aspects all end up being overkill. See table 6.4.

Table 6.4 To-Do List application storage needs

Aspect	Needs	Good fit?
Structure	Structure is fine; not necessary though.	Overkill
Query complexity	Not many fancy queries.	Overkill
Durability	High; we don't want to lose stuff.	Definitely
Speed	Not a lot.	Overkill
Throughput	Not a lot.	Overkill
Cost	Lower is better for all toy projects.	Overkill

Overall, Cloud Spanner is an acceptable fit if you don't care about your bank account. For all of the performance-related aspects as well as the querying abilities, using Spanner for this project is a bit like swatting a fly with a sledge hammer. If the To-Do List application became enormous, where everyone in the world were using it, then Cloud Spanner would become a much better fit because a traditional SQL database may start falling over after the first billion users.

E*EXCHANGE

E*Exchange, the online trading platform, is a bit more complex compared to To-Do List, specifically when it comes to the queries that need to be run and the transactional semantics needed to ensure that concurrent users don't overwrite one another. As shown in table 6.5, for this application, Cloud Spanner is a slightly better fit.

Table 6.5 E*Exchange storage needs

Aspect	Needs	Good fit?
Structure	Yes, reject anything suspect; no mistakes.	Definitely
Query complexity	Complex; we have fancy questions to answer.	Definitely
Durability	High; we cannot lose stuff.	Definitely
Speed	Things should be pretty fast.	Definitely
Throughput	High; we may have lots of people using this.	Definitely
Cost	Lower is better, but willing to pay top dollar.	Definitely

Looking through this, it looks like Cloud Spanner is a pretty great fit for E*Exchange, offering the advanced querying and transactional semantics that you need for the project but also keeping queries fast (low latency) even under heavy load (high throughput).

INSTASNAP

InstaSnap, the popular social media photo-sharing application, has a few requirements that seem to fit well and only a couple that are a bit off. See table 6.6.

Table 6.6 InstaSnap storage needs

Aspect	Needs	Good fit?
Structure	No, structure is pretty flexible.	Overkill
Query complexity	Mostly lookups; no highly complex questions.	Overkill
Durability	Medium; losing things is inconvenient.	Overkill
Speed	Queries must be fast.	Definitely
Throughput	High; Kim Kardashian uses this.	Definitely
Cost	Lower is better, but willing to pay top dollar.	Definitely

As you can see, some of the querying features are overkill for InstaSnap because it's more key-value oriented. The ability to remain fast as more and more people start using the app, however, makes Spanner less overkill than it was for other simple apps (such as To-Do List).

The primary concern for InstaSnap is about single-query latency as the request load goes through the roof (when a famous person posts a photo and the whole world wants to see it at the same time), and in this scenario Cloud Spanner does well, and will do even better with a cache, like Memcache, around to help out.

Summary

- Spanner is a relational database (like MySQL) with the scaling abilities of a non-relational database (like Cassandra or MongoDB).
- Spanner has the ability to automatically split data into chunks based on hints you provide, which allows it to evenly spread request load across many different servers, keeping query latency low even under heavy load.
- Spanner is always deployed in a replicated regional configuration, with multiple complete replicas in several zones.
- Spanner is generally a good fit when you need the features of a SQL database, but the scalability of a nonrelational system.



Cloud Bigtable: large-scale structured data

This chapter covers

- What is Bigtable? What went into its design?
- How to create Bigtable instances and clusters
- How to interact with your Bigtable data
- When is Bigtable a good fit?
- What's the difference between Bigtable and HBase?

Over the years, the amount of data stored has been growing considerably. One reason is that businesses have become more interested in the history of data changes over time than in a snapshot at a single point. Storing every change to a given value takes up much more space than a single instance of a value. In addition, the cost of storing a single byte has dropped significantly. Following this practice has led to engineering projects focused on discovering more uses for all of this just-in-case data such as machine learning, pattern recognition, and prediction engines.

These new uses require storage systems that can provide fast access to extremely large datasets, while also maintaining the ability to update these datasets continuously. One of these systems is Google's Bigtable, first announced in 2006, which has been reimplemented as the open source project Apache HBase. Based on the success

of HBase, Google launched Cloud Bigtable as a managed cloud service to address the growing need for these large-scale storage systems. Let's explore what Bigtable is and dig into some of the technical details that went into building it.

7.1 What is Bigtable?

Bigtable began as the storage system for the web search index at Google and has become one of the main technologies backing many of the other storage systems at Google, such as Megastore and Cloud Datastore. It was built to solve a specific but complex problem: How do you store and continuously update petabytes of data, with incredibly high throughput, low latency, and high availability?

The obvious question is why you can't toss all of this into MySQL. MySQL falls over quickly in attacking this problem, so Google came up with an interesting way of using a globally sorted key-value map, which automatically rebalances data based on service use to reach the performance and scale requirements needed. Let's look more closely at the design goals (and nongoals) that went into building Cloud Bigtable and how they affect whether you should use Bigtable in your own applications.

7.1.1 Design goals

Because the primary use case for Bigtable was the web search index, let's look specifically at those requirements. Google's web search index is one of those things that must be always on and always fast, so it should come as no surprise that many of the requirements are related to both performance and scale—which come at the cost of sacrificing many of the nice-to-have features common in modern databases.

LARGE AMOUNTS OF (REPLICATED) DATA

The search index will obviously be enormous, with overall sizes measured in petabytes, which means that it's far too large for a single server to manage. This is also a benefit, however. One of the hidden requirements for the index would be that it's distributed across many different servers, each one being in some sense commodity hardware (aka cheap). This problem is further exacerbated by the need to ensure that the data itself is stored in more than one place—after all, hard drives and servers can fail and you wouldn't want a chunk of the data to disappear (even temporarily) due to occasional hardware failure.

LOW LATENCY, HIGH THROUGHPUT

Regardless of the size of the data to be stored, the search index clearly sees a ton of traffic, potentially millions of queries every second. If the search index starts failing as more and more requests come in at the same time, folks will take their searches elsewhere.

Each search request needs to return a result quickly, measured in milliseconds. When you include all of the other things that need to happen to achieve that deadline, this leaves relatively little time to query the database—likely only a few milliseconds. Anything more than “get the data at this address” will exceed the time deadline.

RAPIDLY CHANGING DATA

New web pages are added all the time and the search index will be updated by a web crawler frequently. Regardless of the number of queries asking to find web pages (number of reads per second), the system must handle lots of updates at the same time (number of writes per second).

Although these writes likely have a less extreme latency requirement (for example, they can take longer than a user-facing search request), if these updates take too long, they will start to pile up and the index will slip out of date. Though a single write request can take longer to finish, the total number of write operations that can be done in a given period of time needs to be a large number.

HISTORY OF DATA CHANGES

Because the data being stored will change rapidly over time, you want a way to easily see the data as it was at a particular point in time. The client can do this manually by constructing keys with timestamps to signify which version of the data you’re referring to. Letting the storage system track change history, however, keeps your clients thin and simple. In some ways, you can think of this as a third dimension to data—typically databases have a row and column position (two dimensions), but to see history of the data in a row, you’ll need a third dimension: time. See figure 7.1.

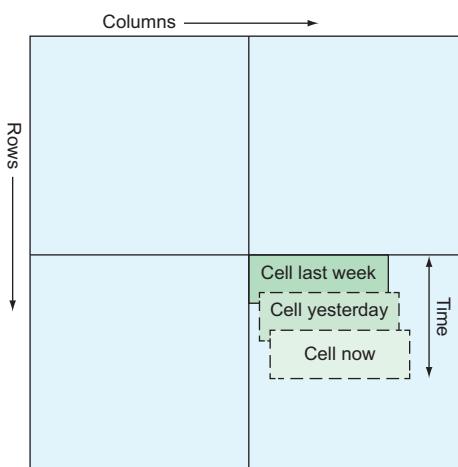


Figure 7.1 Time as a third dimension in Bigtable

With this ability, you’ll be able to ask for the latest value in a row, as well as all the values that this row has had over time.

STRONG CONSISTENCY

Next up is the need for strong consistency, which means anyone querying the index will never see stale data. Updates either happen everywhere or don’t happen.

If the system didn’t have this property (and was eventually consistent instead), it’d be possible for someone to search for the same thing in two browser windows and see different results—definitely not good.

ROW-LEVEL TRANSACTIONS

In addition to always presenting a consistent view of the world, this system would need to allow atomic read-modify-write sequences or risk two updates overwriting each other. The system must expose a way to return an error if someone else has changed a row's data while you're attempting to work on it.

Figure 7.2 shows what you want to happen when two competing writes overlap during a transaction on a single row.

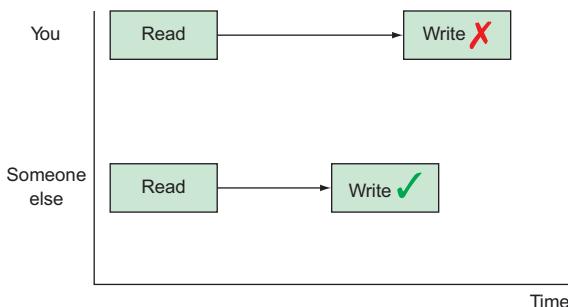


Figure 7.2 What should happen if two clients overwrite each other

Although this is definitely a requirement for a single row, it's unlikely that we'll have multiple rows in the search index that would require an atomic update across them. This means that although this system would need to provide atomic writes for a single row to avoid write contention, it wouldn't need general transactional semantics across multiple rows.

SUBSET SELECTION

Finally it's important to remember that you don't always want to request all of the data stored for a given set of results, so it'd be nice if the system had a way of asking for only a specific set of properties, such as a specific set of column families, columns, or timestamps, which would allow you to ask for things like only the two most recent values.

Being able to limit the pieces the storage system should return, allows you to store more data in one chunk and request only small bits of that large chunk.

7.1.2 Design nongoals

Quite a few things are not necessarily required—they'd be lovely to have, but you can do without them if it makes the other aspects possible. In the case of Bigtable, to achieve the enormous scale of the datasets combined with the throughput and latency requirements, you would need to drop most of the nice-to-have features such as secondary indexes (such as the ability to run queries like `SELECT * FROM users WHERE name = "Jim"`), multirow transactional semantics, and many of the other things you've come to expect from databases.

7.1.3 Design overview

What came out of all of these requirements was a unique storage system that did things quite differently from most of the nonrelational systems that existed at the time (2006). As the name suggests, Bigtable is a large table of data with some important differences from the tables you've come to know. Though in many ways it can act like a traditional table, the storage model of Bigtable is much more like a jagged key-value map than a grid. In fact, the authors of the research paper describing Bigtable called it "a sparse, distributed, persistent, multi-dimensional sorted map" (the key word at this point being "map"). Put visually, this design looks something like figure 7.3.

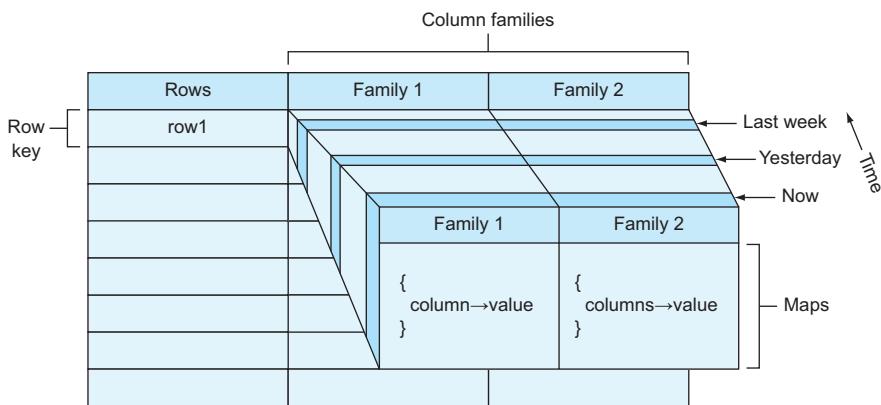


Figure 7.3 Bigtable design overview

In short, Bigtable is less like a relational database and a bit more like a big key-value store that distributes data across lots of servers while keeping all the keys in that map sorted. Thanks to that global sorting, Bigtable allows you to do both key lookups (as you would in any key-value store) as well as scans over key ranges and key prefixes.

Lastly, hidden in this list of features is the idea that the map is multidimensional. In this case, the extra dimension attached to all data stored in Bigtable is a timestamp, which effectively allows you to go back in time and view data as it was at a previous point. This unique set of features is what makes Bigtable so powerful.

7.2 Concepts

Although Bigtable is incredibly robust, it will require you to think a bit differently about the structure and access patterns of your data, similar in some ways to our previous discussion of Cloud Datastore. Generally, you'll have to think ahead about what types of questions you'll want to ask about your data, because the ability to answer different questions is determined frequently by the way in which you structure that data.

7.2.1 Data model concepts

Let's kick things off by looking at the concepts you'll need to understand, starting with the data model concepts shown in figure 7.4: tables, rows, column families, and column qualifiers.

Rows	Table					
	Column Family 1			Column Family 2		
	col1	col2	col3	col1	col2	col3
row1						
row2						
row3						
.						
!						

Figure 7.4 Data model concept hierarchy

Note that although the data model concepts apply most specifically to Cloud Bigtable, they're also relevant if you use HBase, which was designed following the publication of the Bigtable paper in 2006. The second section of this chapter, which focuses on the infrastructural details of Cloud Bigtable as a managed service, applies almost exclusively to Cloud Bigtable.

ROW KEYS

Although Bigtable may look a bit like a relational database at a glance, data stored is much more like a key-value store (as we saw with Cloud Datastore), where the key used to find content is called the row key. This key can be anything you want, but as you'll read later, you should choose the format of this key carefully.

If you're familiar with relational databases, you've likely seen the term PRIMARY KEY somewhere in SQL to denote that a specific column is both unique and used to identify a given row. In Bigtable the row key is used for the same purpose, and you can think of it as the address of a given chunk of data. Bigtable allows you to quickly find data using a row key, but it doesn't allow you to find data using any secondary indexes (they don't exist). Therefore, even though in a relational system you're able to do lookups based on other columns (for example, `SELECT * FROM users WHERE name = 'Jim'`), you can't do this kind of lookup in Bigtable.

ROW KEY SORTING

As mentioned earlier, choosing how to structure and format row keys is important for a few different reasons:

- Row keys are always unique. If you have collisions, you'll overwrite data.
- Row keys are lexicographically sorted across the entire table. High traffic to lots of keys with the same prefix could result in serious performance problems.

- Row key prefixes and ranges can be used in queries to make the query more efficient. Poorly structured keys will require inefficient full-table scans of your data.

These reasons mean that some of the more common key formats aren't a good fit for Bigtable. For example, in MySQL or PostgreSQL, primary keys typically take the format of a sequence of numbers (1, 2, 3, 4, ...). Using an incrementing sequence for your row key means that you may have collisions, are likely to encounter performance problems, and are precluded from doing queries by key-range because it's not super-useful to say "Please give me users 1 through 20."

The most subtle issue in choosing a row key is choosing a format that's both useful to you as the application developer and efficient for Bigtable to store, so take your time in choosing a key, and make sure that whatever you choose fits the criteria described earlier. To simplify this, let's walk through some examples of row key formats that would be a particularly good fit for Bigtable:

- *String IDs (hashes)*—If your identifier is an opaque ID (such as Person #52), use a hash of that value (such as 'person_' + crc32(52)). The hash ensures that the row key is both a fixed length and evenly distributed (lexicographically) throughout the key space.

The underlying assumption here is that although writes to the entire system (for example, all person_rows) may need to handle an extremely heavy load, writes to a single person's row (such as person_2b3f81c9) are much more likely to be a tiny fraction of the overall load. Because all the rows will be evenly distributed, Bigtable can optimize where each row lives and evenly distribute the load across lots of machines. This is discussed in more detail later.

- *Timestamps*—It's often the case that you'll need to retrieve data based on a point in time, which makes timestamps an obvious choice. *Do not use a timestamp* as the key itself (or the start of the key)! Doing so ensures that all write traffic will always be concentrated in a specific area of the key space, which would force all traffic to be handled by a small number of machines (or even a single machine).

A good rule of thumb is to prefix time-series data with another key that's useful for querying. For example, if you're storing stock price information over time, consider prefixing the row key with the hash of the stock ticker symbol (such as stock_c318f29c#1478519731 which is 'stock_' + crc32(GOOG) + '#' + NOW()).

- *Combined values*—Sometimes rows contain information relating two different concepts, for example, a tag of a person in a post involves the person tagged and the person doing the tagging. In these cases, you can combine the two keys into a single key to simplify looking up all messages between two people.

For example, if Alice tagged Bob in a post, you might use a key that looks like post_6ef2e5a06af0517f, which is 'post_' + crc32(alice) + crc32(bob). You can then store the post content in the row data.

- *Hierarchical structured content*—The last format, similar to Java package-naming formats, is to use a reverse hierarchy prefix format. The most common example of this is reverse domain name such as com.manning.gcpia (gcpia.manning.com reversed), which makes row key ranges convenient. You can ask for everything belonging to manning.com (prefixed with com.manning.) or everything belonging to gcpia.manning.com (prefixed with com.manning.gcpia.).

This format works well with anything hierarchical because the reversed hierarchy allows filtering by providing a longer (more specific) prefix. The assumption is that specific rows would follow the guidelines discussed previously using hashed final values to ensure relatively even key distribution.

Note that nonreversed hierarchical representation doesn't put related rows next to one another (for example, gcpia.manning.com is not lexicographically next to forum.manning.com), so it's not a good idea to use the nonreversed format.

Now that you have a better grasp on what row keys are, let's look at the data that they point to and how that's structured.

COLUMNS AND COLUMN FAMILIES

In many key-value stores, the data that's pointed at by a particular key is a completely unstructured piece of data. It could be a bunch of bytes representing an image, or it could be a JSON document storing a user profile, but the storage system typically has no understanding (or need to understand) what the value is. Even though you don't define secondary indexes in Bigtable, it does allow you to define certain aspects resembling a schema, which makes it easy to specify which bits of data to retrieve. This schema acts as extra criteria to define the structure of a key-value map that will ultimately hold your data.

In Bigtable the keys of this map are called *column qualifiers* (sometimes shortened to *columns*), which are often dynamic pieces of data. Each of these belongs to a single *family*, which is a grouping that holds these column qualifiers and act much more like a static column in a relational database. This unique combination of static and dynamic data may seem strange, and at first it is, so don't be worried. Unlike normalized SQL databases, column qualifiers can be anything you want and can be thought of as data—something you'd never do in a relational database. Using this type of structure also means that when you visualize data in Bigtable as a table, most of the cells will be empty, or you call a sparse map.

VISUALIZING YOUR BIGTABLE DATA

To make this more concrete, let's imagine a to-do list where you're storing items that have been completed. If you don't recall, the To-Do List application was a simple tracker of items that each user wants to complete. The app also tracks when each item is completed and any notes recorded at that time. To store this same data in Bigtable, you'd have a completed column family (this is the static part) where each individual column qualifier corresponds to the ID of the item (these are the dynamic keys of

each row). In addition, you may need to add another column qualifier to store optional notes that you write down when completing the item. See table 7.1.

Table 7.1 Visualizing To-Do List

Row key	Completed			
237121cd (user-3)	item-1	item-1-notes	item-2	item-2-notes "Right on time"
4d4aa3c4 (user-1)	true		true	"2 days late!"
946ce0c9 (user-2)				

Although this setup looks similar to any other relational database table at first, when you look more closely it starts to seem strange and inefficient. Why aren't the completed items stored in a table with a user ID, an item ID, and a completed field? Why are notes stored on that particular row? Why are there so many empty spaces? Isn't that inefficient?

First, notice the row key is acting as your address, or the lookup key of where to find the data. Although it looks like a relational table, your data is probably better thought of as a key-value store. (For example, there's no way to query for all users with item-1 completed.) Second, each row stores only the data present in that row, so there's no penalty for those empty spaces—it's a sparsely populated table. Finally, the column qualifiers (such as item-1) can be thought of as a dynamic value adding further detail to the completed column family. The static part (similar to the column name in a relational database) is the completed column family. Inside that column family is an arbitrary set of dynamic data, but you happen to visualize that as somewhat of a subtable with more columns and values for those columns.

You could also visualize this is as a key-value store where the keys are further maps of data, a bit like how you first saw Cloud Datastore. See table 7.2.

Table 7.2 Visualizing To-Do List as maps

User	Data (completed column family)
237121cd (user-3)	{item-2: true, item-2-notes: "Right on time"}
4d4aa3c4 (user-1)	{item-1: true, item-2: true, item-2-notes: "2 days late!"}
946ce0c9 (user-2)	{}

The completed column family is a static key in the map, so you could look at this differently by adding some hierarchy, as shown in tables 7.3 and 7.4.

Table 7.3 Visualizing To-Do List as maps with hierarchy

User	Data
237121cd (user-3)	{item-2: {completed: true, notes: "Right on time"}}}
4d4aa3c4 (user-1)	{item-1: {completed: true}, item-2: {completed: true, notes: "2 days late!"}}
946ce0c9 (user-2)	{}

Table 7.4 Visualizing To-Do List as maps with a different hierarchy

User	Data
237121cd (user-3)	{completed: {item-2: true, item-2-notes: "Right on time"}}}
4d4aa3c4 (user-1)	{completed: {item-1: true, item-2: true, item-2-notes: "2 days late!"}}
946ce0c9 (user-2)	{}

These are all different ways of looking at the same data, but for the rest of the chapter you'll use the format in table 7.1, which is similar to what you'll see in documentation for Bigtable as well as HBase.

Notice how in the final visualization, all of the data is grouped first with a key called completed. Even though you only have that single column family, you could imagine further column families in the table, perhaps a followers column family that shows who's watching the items you complete. With that, it becomes a bit more clear why column families are useful—you're able to ask specifically for the chunks of data you want. If you wanted to see what items were completed by a user, you'd ask for only the completed column family which would return only the data in the completed key. If you wanted to see followers, you could ask for the followers column family, which would give you only that subset of data. Column families are helpful groupings that, in some ways, can be thought of as the keys pointing to maps of arbitrary maps of more data.

Now that you understand what columns and column families are, let's explore the different layouts of tables, which give you the ability to query your data in different ways.

TALL VS. WIDE TABLES

So far our earlier example used what we'll call a *wide* table, which is a table having relatively few rows but lots of column families and qualifiers. In the example, each row stores the data about a particular user and contains quite a few potential column qualifiers (one for each item that user completes). This allows you to ask useful questions such as “What items has user-020b8668 completed?” What if you wanted to know whether a user has completed a particular item? This is where a tall table would come into play.

As you might guess, a *tall* table is one with relatively few column families and column qualifiers but quite a few rows, each one corresponding somehow to a particular

data point. In this case the row key would be a combination of values, as we discussed in the previous section, which is easy to calculate given the data you’re interested in. For example, using a tall table to store whether a given item was completed, you might use a combined hash of the user and the item, which might look like `4d4aa3c4931ea52a` (`crc32(user-1) + crc32(item-1)`). In the column data you might store the notes in a similarly named completed column family. This would make your tall table look something like table 7.5.

Table 7.5 Tall table version of To-Do List

Row Key	Completed	
	item-id	notes
<code>4d4aa3c4931ea52a</code> (user-1, item-1)	item-1	
<code>4d4aa3c44a38e627</code> (user-1, item-2)	item-2	"2 days late!"
<code>b516476c4a38e627</code> (user-3, item-2)	item-2	"Right on time"

This table style contains quite a few differences compared to what we’ve discussed so far. The first and most obvious difference is that rather than growing wider as more items are completed, it will grow longer (or taller) instead. In addition to looking different on paper, this tall version of your table allows you to query quickly to ask the question, “Did user-1 complete item-1?” In this case, that’s a matter of computing the proper row key (`crc32(user-1) + crc32(item-1)`) and checking if the row exists.

Finally, this table also allows you to ask the question, “What items did user-1 complete?” but answers it in a different way. In the wide table example, looking up the complete items was a single get for a given user, but in this tall table example, to find the list of items completed for a user, you would execute a scan based on a row prefix, in this case, `crc32(user-1)`. This prefix would return all rows starting with that value, which you can then iterate through to find all of the items that were completed (one per row).

Although these two tables do ultimately allow you to ask similar questions, it would appear that the tall version allows you to be a bit more specific at the cost of more single-entry lookups to get bulk information. If you’re going to be asking bulk-style questions (such as “What did user X do?”), a wide table may be a better fit; if you intend to ask more specific questions (“Did user X do thing Y?”), a tall table is likely a better fit. Now that we’ve gone deeper into the data-modeling concepts, let’s switch back to the infrastructural world and see how exactly you turn on and use Cloud Bigtable.

7.2.2 *Infrastructure concepts*

As discussed earlier, Cloud Bigtable acts as a managed service, which means that you don’t have to manage individual virtual machines like you would if you were running your own HBase cluster. Automated management features some new concepts that

you'll need to understand. Unfortunately, Bigtable is one of the more confusing services, particularly when it comes to how replication is handled. Another tricky area is that Bigtable itself has a concept of a tablet, which isn't directly exposed via the Cloud Bigtable API. To keep things as simple as possible, let's start by first looking at the hierarchy of concepts that you can manage yourself: instances, clusters, and nodes. See figure 7.5.

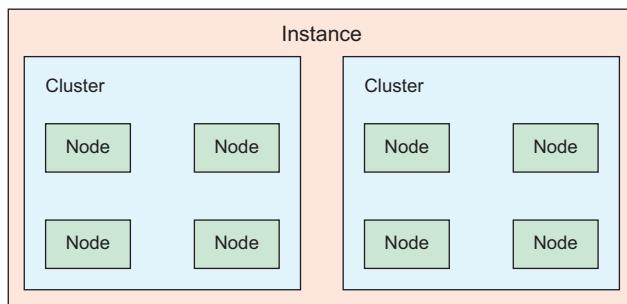


Figure 7.5 Hierarchy of instances, clusters, and nodes

As you can see, the basic structure here is that an instance is the top-most concept and can contain many clusters, and each cluster contains several nodes (with a minimum of three).

INSTANCES

Think of an instance as the primary resource you refer to when thinking about your Bigtable deployment, similar to how you'd think of the database server when deploying a MySQL cluster (with a primary and a read-slave). When you write data to Bigtable, you'd refer to writing it to a specific Bigtable instance.

Unlike a MySQL cluster where you always write data to the primary, in Bigtable you send your data to the instance, which ensures that those changes are propagated to all the other clusters. Although you can address specific clusters directly if needed, it shouldn't be necessary because Bigtable should route your queries to the closest cluster and, therefore, should be reliably fast. Instances are globally scoped, meaning that they remain addressable regardless of whether a particular zone is experiencing an outage.

CLUSTERS

Before we go into too much detail about clusters, let's start with an important caveat: though figure 7.5 shows multiple clusters per instance, this is currently not yet possible—you're limited to a single cluster per instance. That said, Bigtable will almost certainly support replication with multiple clusters per instance in the future. Given that impending launch of the feature, let's look at how clusters function with the assumption that you'll soon be able to maintain many of them inside a single instance.

Clusters, unfortunately (or fortunately, depending on who you ask), are boring. They’re a grouping for a bunch of nodes, each of which is responsible for handling some subset of queries sent to a Bigtable instance. Each cluster has a unique name, a location (zone), and some performance settings such as the type of disk storage to use as well as the number of nodes to run. Clusters themselves have an hourly computing cost, as well as a monthly storage cost to reflect the amount of data stored in that particular cluster. Each cluster holds a copy of your data, so more clusters would imply higher availability of your data with the obvious trade-off of higher costs. As you’d expect, your hourly computing cost goes up as you add more nodes, with the benefit that you’ll never hit a bottleneck of “too many nodes,” as has been known to happen with other systems such as HBase.

NODES

Nodes are even more boring than clusters for one important reason: from our perspective, they’re invisible. Although we talk about nodes as discrete individual entities, in reality you’ll never experience them that way except for seeing them on your bill. Although you can think of a cluster as a grouping together of multiple nodes, the nodes themselves are hidden from you in the API. You can communicate only with the particular cluster that’s responsible for routing your request to a particular node.

This structure allows the cluster to ensure that requests are spread evenly across the nodes and also allows the cluster to rebalance data to maintain this even distribution. If nodes themselves were addressable, the cluster wouldn’t be able to move data around as freely, which could lead to a case where a single node held all the hot data, driving down performance during busy times. This leads us to the Bigtable concept of a *tablet*, which we haven’t yet discussed, but it’s important to understand when you’re concerned about performance.

TABLETS

Tablets are a way of referencing chunks of data that live on a particular node. The cool thing about tablets is that they can be split, combined, and moved around to other nodes to keep access to data spread evenly across the available capacity. As with nodes, you’ll never address tablets directly, so you won’t see these in the API, but you can influence how data is written to tablets through the choice of your keys. For example, writing lots of data quickly over a long period of time to keys with two distinct prefixes (such as `machine_` and `sensor_`) will typically lead to the data being on two distinct tablets (such as `machine_` prefixed data wouldn’t be on the same tablet as `sensor_` prefixed data). Let’s take a quick look at the progression of data as you add more (and query more) over time.

When you first start writing data, your Bigtable cluster will likely put most of the data on a single node, shown in figure 7.6.

As more tablets accumulate on a single node, the cluster may relocate some of those tablets onto another node to redistribute the data in a more balanced fashion, shown in figure 7.7.

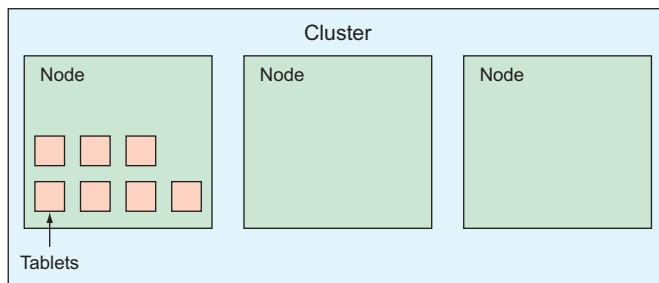


Figure 7.6 When starting, Bigtable might put data on a single node.

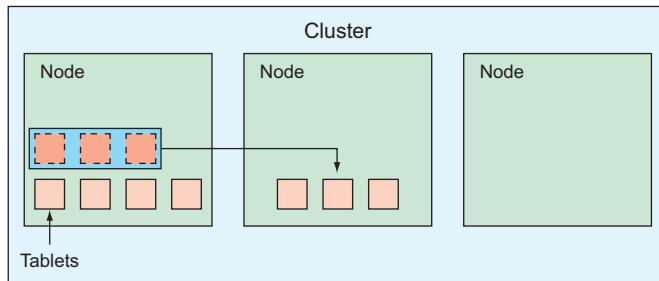


Figure 7.7 Bigtable redistributes tablets to spread data more evenly across nodes.

As more data is written over time, it's possible that some tablets are more frequently accessed than others. In figure 7.8, three tablets are responsible for 35% of all the read queries on the entire system.

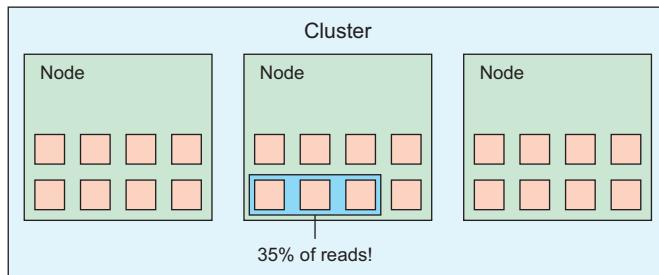


Figure 7.8 Sometimes a few tablets are responsible for a high percentage of traffic.

In scenarios like these, where a few hot tablets are colocated on a single node, Bigtable rebalances the cluster by shifting some of the less frequently accessed tablets to other nodes that have more capacity to ensure that each of the three nodes sees about one-third of the total traffic, shown in figure 7.9.

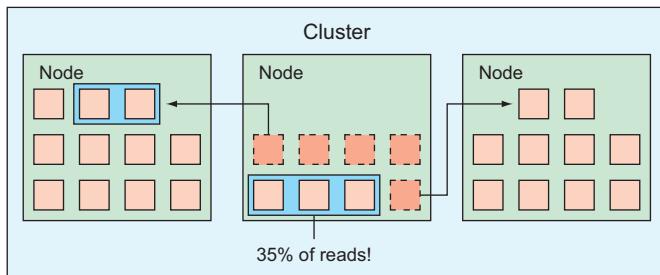


Figure 7.9 Bigtable shifts data away from hot tablets.

It's also possible that a *single* tablet could become too hot (it's being written to or read from far too frequently). Moving the tablet as it is to another node doesn't fix the problem. Instead, Bigtable may *split* this tablet in half and then rebalance the tablets as we saw earlier, shifting one of the halves to another node. See figures 7.10 and 7.11.

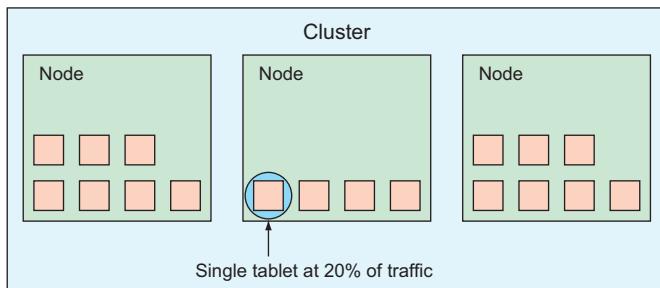


Figure 7.10 Sometimes a single tablet is responsible for a high percentage of traffic.

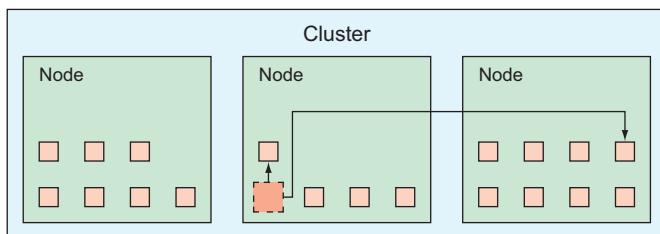


Figure 7.11 Bigtable splits tablets and shifts them to other nodes.

If you're a bit confused by the inner workings of Bigtable, that's OK. Although it's great to understand nodes, tablets, and re-balancing of data, Bigtable itself is a complex storage system, and understanding every nuance is incredibly difficult. In general,

the most important thing you can do when using Bigtable is to choose row keys carefully so that they don't concentrate traffic in a single spot. If you do that, Bigtable should do the right thing and perform well with your dataset. By now you should have a pretty decent understanding of how all the parts fit together, so let's take a minute to walk through how to manage your own Bigtable instance.

7.3 *Interacting with Cloud Bigtable*

As you saw previously, Bigtable has a simple hierarchy involving instances, clusters, and nodes, and the data model for each of these is simple as well, involving tables, rows, column families, and column qualifiers. But so far we've only talked about these. Let's take a look at how to create these different resources, using a combination of the Cloud Console and some command-line tools, starting with creating an instance.

7.3.1 *Creating a Bigtable Instance*

Before you can do anything with Cloud Bigtable, you first have to create a new instance. As we discussed earlier, currently you're limited to a single cluster per instance, which can be a bit off-putting when you're expecting a true hierarchy. For now, let's not worry about that and go ahead with creating your instance.

Start by navigating to the Bigtable section of the Cloud Console using the left-side navigation. Then you can click the Create instance button on the top to open a form with several fields to fill out. It's important to start by filling in the first field (instance name) right away. When you do that and click elsewhere on the form, the next two fields (instance ID and cluster ID) complete themselves automatically, as you can see in figure 7.12. A cluster ID can be automatically computed from the instance name,

The screenshot shows the 'Create instance' page in the Google Cloud Platform Bigtable interface. At the top, there's a logo and the word 'Bigtable'. To its right is a back arrow and the text 'Create instance'. Below this, a descriptive text states: 'A Cloud Bigtable instance is a container for your cluster. Choose the instance and cluster properties below.' Under the heading 'Instance properties', there are three input fields:

- Instance name:** A placeholder text 'For display purposes only.' followed by a text input field containing 'test-instance'.
- Instance ID:** A placeholder text 'ID is permanent. Use lowercase letters, numbers, or hyphens.' followed by a text input field containing 'test-instance'.
- Cluster ID:** A placeholder text 'ID is permanent. Use lowercase letters, numbers, or hyphens.' followed by a text input field containing 'test-instance-cluster'.

Below these fields is a section titled 'Cluster properties'.

Figure 7.12 Bigtable instance identifiers

and because there's currently a limit of one cluster per instance, this is not a useful field (though it will be eventually).

Next you'll need to choose a zone, as shown in figure 7.13. As discussed previously, a cluster is a zonal resource, which means that its availability is subject to that of the zone. This is where your data will live and is, therefore, permanent for the cluster, so you should aim to choose a zone that's near any VMs that need to read or write Bigtable data. This zone should be the same as where all of your VMs live.

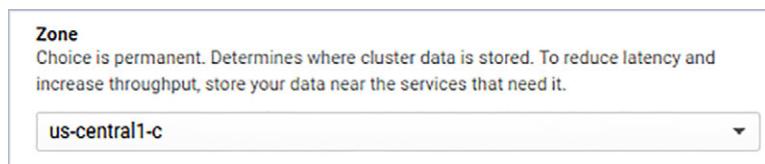


Figure 7.13 Bigtable zone setting

The next two pieces of information concern the performance of your Bigtable instance. See figure 7.14. The first is computing throughput where you set how many nodes you want to keep running. You can change this number later, so don't worry too much about it, but the minimum you can choose is three. As you might expect, the number of nodes you have will increase the read, write, and scan capacity of your instance in a mostly linear way. If you're getting started, a good first choice is to leave this set to three and expand your instance with more nodes later if you need the extra capacity.

A screenshot of a configuration page for a Bigtable instance. It shows the following sections:

- Nodes (3 – 30) ?**: A text input field containing the value "3". Below it is explanatory text: "Add nodes to increase data throughput and queries per second (QPS). Contact us to request more than 30 nodes."
- Storage type ?**: A radio button group. The option "SSD (Recommended)" is selected, with the note "Most popular choice. Lower latency. Higher QPS and data throughput.". The other option, "HDD", is described as "May be preferable for very large data sets (>10 TB). Much cheaper per GB. Best for infrequently read data that can tolerate up to 200ms latencies."
- Performance**: A note stating "The storage type and the number of nodes in your cluster determine performance."
- A summary table showing performance metrics:

Reads	Writes	Scans
30,000 QPS @ 6ms	30,000 QPS @ 6ms	660 MB/s

Figure 7.14 Bigtable performance characteristics

The next piece related to performance is the type of disk to use to store your data. A solid-state disk is going to have much better performance in both latency and throughput, which makes a big difference in your Bigtable instance. For this reason, unless you specifically know that the SSD storage type is overkill for your use case and you know that standard disk (HDD) is acceptable, you should plan to leave this set to SSD.

For comparison, the table 7.6 shows the performance differences of the different storage types for Cloud Bigtable.

Table 7.6 Storage type comparison for Cloud Bigtable

Attribute	SSD (recommended)	HDD	Comparison
Read throughput	10,000 QPS/node	500 QPS/node	20x better with SSD
Read latency	6 ms	200 ms	33x better with SSD
Write throughput	10,000 QPS/node	10,000 QPS/node	Same
Write latency	6 ms	50 ms	33x better with SSD
Scan throughput	220 MB/s	180 MB/s	1.2x better with SSD

7.3.2 Creating your schema

As you've learned, your schema will have long-lasting effects, so it's something you should try to get right ahead of time. Unlike a traditional relational database, updating the schema later isn't quite as simple as running an `ALTER TABLE` statement. Instead, you must update every single row you have stored to fit your new schema, similar to how you would with any other key-value storage system. Although this is certainly possible by adding code complexity (such as by adding code that understands multiple schema versions and, when saving, rewrites data in the newest version), it's still worthwhile to invest time up front to push any schema changes off into the distant future.

Although it's a terrible example of how to use Bigtable, let's use the To-Do List project to test what it's like interacting with Bigtable. To refresh your memory, you'll be using the tall table format, where your row key is a combination of hashes for the user and the item with a single Column Family called `completed`. In this case, the column qualifiers themselves are known in advance, which isn't always the case (as you saw in the wide table format). Your table will look a bit like table 7.7.

Table 7.7 Tall table version of To-Do List

Row key	Completed	
4d4aa3c4931ea52a (user-1, item-1)	item-id	notes
4d4aa3c44a38e627 (user-1, item-2)	item-1	"2 days late!"
b516476c4a38e627 (user-3, item-2)	item-2	"Right on time"

Before you can write some code to interact with Cloud Bigtable, you need to install the `@google-cloud/bigtable` client by running `npm install @google-cloud/bigtable@0.9.1`. After the client is installed, you can test it by listing out the instances and clusters, shown in the next listing.

Listing 7.1 Listing instances and clusters

```
const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

console.log('Listing your instances and clusters:');

bigtable.getInstances().then((data) => {
  const instances = data[0];
  for(let i in instances) {
    let instance = instances[i];
    console.log('- Instance', instance.id);
    instance.getClusters().then((data) => {
      const clusters = data[0];
      const cluster = clusters[0];
      console.log(' - Cluster', cluster.id);
    });
  }
});
```

Annotations for Listing 7.1:

- An annotation points to the call to `getInstances()` with the text: "Use `.getInstances()` to iterate through the list of available instances."
- An annotation points to the call to `getClusters()` with the text: "Use `.getClusters()` to iterate through the list of clusters in an instance."

When you run this after creating an instance, you should see something like the following:

```
Listing your instances and clusters:
- Instance projects/your-project-id/instances/test-instance
  - Cluster projects/your-project-id/instances/test-instance/clusters/test-
    instance-cluster
```

In this case, it appears exactly as you expect, with one instance and one cluster belonging to that instance. Now let's look at how to create your table and schema in the next listing.

Listing 7.2 Creating a table

Column families are defined as a list of strings.

```
const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

const instance = bigtable.instance('test-instance');

instance.createTable('todo', {
  families: ['completed']
}).then((data) => {
```

Annotations for Listing 7.2:

- An annotation points to the line `const instance = bigtable.instance('test-instance');` with the text: "This time you construct an instance object using its ID rather than trying to look it up via the API as you did previously."
- An annotation points to the call to `createTable()` with the text: "Notice that you create a table with `instance.createTable()` rather than `cluster.createTable()`. As noted earlier, the instance is the owner of tables."

```
const table = data[0];
console.log('Created table', table.id);
});
```

After running this, you should see output looking something like this:

```
Created table projects/your-project-id/instances/test-instance/tables/todo
```

That's it! You've now created a table called `todo` with a single column family called `completed`. But a schema alone isn't all that useful, so let's look at how you can manage the data that goes in your table.

7.3.3 Managing your data

As with any storage system, there are two sides to managing data: one going in (writing), the other going out (reading or querying). Start by adding some new rows to your `todo` table, and then you'll see how you can query to get these rows back. As you may recall, a single row in your tall table has a row key that's a concatenated hash of the user ID and item ID, and your columns are statically defined as `item-id` and `notes`.

NOTE You'll use CRC32 as the hash for this exercise, which means you'll need to install the library by running `npm install fast-crc32c`.

The code to add a row completing an item would look something like the following.

Listing 7.3 Inserting data into Bigtable

```
const crc = require('fast-crc32c');
const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

const instance = bigtable.instance('test-instance');
const table = instance.table('todo');

const userId = 'user-84';
const itemId = 'item-24';
const notes = 'This was a few days later than expected';

const userHash = crc.calculate(userId).toString(16);
const itemHash = crc.calculate(itemId).toString(16);
const key = userHash + itemHash;

const entries = [
  {
    key: key,
    data: {
      completed: {
        'item-id': itemId,
        'notes': notes
      }
    }
}
```

As you constructed the instance from its ID, this time you do the same with the table that you've created, using the ID to create a table reference.

This is the data you plan to store, put into variables so it's easy to read.

You determine the row key by hashing both values and then concatenating them. In this case, the row key is `c4ae6082` combined with `8900c74c`.

The list of entries has to be in a particular format, specifically with the row key in a field called `key` and the data in a field called `data`.

```

        }
    }
];

table.insert(entries, (err, insertErrors) => {
  console.log('Added rows:', entries);
});

```

After you run this code, you should see a confirmation that the data was added:

```
Added rows: [ { key: 'c4ae60828900c74c',
  data: { completed: [Object] },
  method: 'insert' } ]
```

Now that you have some data, added let's look at how you'd retrieve this row, starting with a single key lookup. This works by constructing the row key as you did before, and looking up the row with that key, as shown in the following listing.

Listing 7.4 Retrieving data by key

```

const crc = require('fast-crc32c');
const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

const instance = bigtable.instance('test-instance');
const table = instance.table('todo');

const userId = 'user-84';
const itemId = 'item-24';

const userHash = crc.calculate(userId).toString(16);
const itemHash = crc.calculate(itemId).toString(16);
const key = userHash + itemHash;

const row = table.row(key);
row.get().then((data) => {
  const row = data[0];
  console.log('Found row', row.id, row.data.completed);
});

```

As before, you compute the row key using a hash of the user and item IDs.

Like you did with instance and table, you use the row key to create a row reference.

Obviously there's more data in the row object, but to make it easy to read you're printing the completed column family to the console.

Finally, you use the .get() method on your row object to attempt to retrieve the row from Bigtable.

If you run this code (and you created the row previously), you should see output that looks something like the following. Note that the timestamps will be different:

```
Found row c4ae60828900c74c { 'item-id':
  [ { value: 'item-24',
    labels: [],
    timestamp: '1479145189752000',
    size: 0 } ],
  }
```

```
notes:
[ { value: 'This was a few days later than expected',
  labels: [],
  timestamp: '1479145189752000',
  size: 0 } ] }
```

Now that you understand how to retrieve a single row, let's look at a more powerful type of query that shows off the benefits of using a tall table. Try adding some more data and then iterating over the items completed by a particular user, as shown in the next listing.

Listing 7.5 Inserting a bunch of rows

```
const crc = require('fast-crc32c');
const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

const instance = bigtable.instance('test-instance');
const table = instance.table('todo');

const getRowEntry = (userId, itemId, notes) => {
  const userHash = crc.calculate(userId).toString(16);
  const itemHash = crc.calculate(itemId).toString(16);
  const key = userHash + itemHash;
  return {
    key: key,
    data: {
      completed: {
        'item-id': itemId,
        'notes': notes
      }
    }
};
const rows = [
  ['user-1', 'item-1', undefined],
  ['user-1', 'item-2', 'Late!'],
  ['user-1', 'item-3', undefined],
  ['user-1', 'item-5', undefined],
  ['user-2', 'item-2', 'Partially complete'],
  ['user-2', 'item-5', undefined],
  ['user-84', 'item-5', 'On time'],
  ['user-84', 'item-20', 'Done 2 days early!'],
  ['user-84', 'item-21', 'Done but needs review'],
];
const entries = rows.map((row) => {
  return getRowEntry.apply(null, row);
});
table.insert(entries, console.log);
```

You'll use a helper function called `getRowEntry` to take a few pieces of information and return an object in the format that `table.insert` expects.

To make the data easier to read, write it as an array of rows, sort of like a CSV file in the format of [userId, itemId, notes].

Take the CSV-style data, and get back properly formatted row entries.

Add the data to Bigtable.

Running this snippet should give you a `null`, `[]` in your console, meaning you've added the entries and had no errors with any of the rows. Now let's figure out which items were completed by a particular user. You'll rely on the fact that you chose our row keys to be in the format of `crc(userId) + crc(itemId)` combined with the ability of Bigtable to easily scan across rows with fixed start and end points. You'll start with any key "greater than" (or lexicographically "after") `crc(userId)` and stop with the next key (`crc(userId) + 1`), as shown in the following listing.

Listing 7.6 Scanning rows for user-2

```
const crc = require('fast-crc32c');
const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

const instance = bigtable.instance('test-instance');
const table = instance.table('todo');

const userId = 'user-2';
const userHash = crc.calculate(userId).toString(16);

table.createReadStream({
  start: userHash,
  end: (parseInt(userHash, 16)+1).toString(16)
}).on('data', (row) => {
  console.log('Found row', row.id, row.data.completed);
}).on('end', () => {
  console.log('End of results.');
});
```

As noted earlier, you start at the userHash key. A callout arrow points from this text to the line `start: userHash`.

You're going to scan over all the rows pertaining to user-2. A callout arrow points from this text to the line `end: (parseInt(userHash, 16)+1).toString(16)`.

Here you use the createReadStream method, which allows you to use Javascript's event emitter style .on() handlers. A callout arrow points from this text to the `.on('data', ...)` and `.on('end', ...)` lines.

Because you know userHash to be a string representing a hexadecimal number, you know that incrementing the number by one is where you should stop scanning. A callout arrow points from this text to the line `end: (parseInt(userHash, 16)+1).toString(16)`.

When you run this short snippet, if you added the data listed previously, you should see the two items completed by user-2 as well as any notes that were stored:

```
Found row 79c375855dc6587 {
  'item-id':
    [ { value: 'item-5',
        labels: [],
        timestamp: '1479145268897000',
        size: 0 } ],
  notes: []
}
Found row 79c37588116016c {
  'item-id':
    [ { value: 'item-2',
        labels: [],
        timestamp: '1479145268897000',
        size: 0 } ],
  notes:
    [ { value: 'Partially complete',
        labels: [],
        timestamp: '1479145268897000',
        size: 0 } ]
}
End of results.
```

Now that you understand how to read and write data to and from Bigtable, let's talk briefly about how to manage imports and exports.

7.3.4 Importing and exporting data

As with any storage system it's important to have a back-up strategy for a variety of reasons. The obvious one is in the case of data corruption or physical drive failures, but this shouldn't be a concern with managed services on Google Cloud Platform. There are many other cases not related to this, one of the most common being invalid deployments that write corrupt or incorrect data, protecting you from yourself. To deal with potential issues, Bigtable offers the ability to both export data and reimport data using Hadoop sequence files as the format.

Hadoop, as you may remember, is Apache's open source version of Google MapReduce and is commonly used alongside HBase, Apache's open source version of Bigtable. Thanks to the similarity of these systems, Bigtable can rely on the Hadoop file format, which makes it easy for you to export and import data not only to Cloud Bigtable but also to HBase if you happen to use that.

NOTE Importing and exporting data in Bigtable is currently done by using Google Cloud Dataproc, a managed Hadoop service. You don't need to know anything about Hadoop or Dataproc to import or export data.

Unlike the other import and export operations we've gone through, Bigtable has a unique problem: it's a ton of data. Because Bigtable can (and often does) store petabytes worth of data, asking a single machine to copy all of it somewhere is not exactly going to be a fast process. Therefore, to import or export quickly you'll rely on the magic of distributed systems and turn on many machines under the hood to make this happen.

Managing machines can be a bit of a distraction when all you want to do is export some data from Bigtable. Luckily a managed service called Google Cloud Dataproc can handle the hard work for you, and all you need to do is run a single command. Also, because you're dealing with potentially enormous amounts of data, it's probably best to put that data in Google Cloud Storage. How does this all fit together? The general process looks something like this:

- Download the import/export package from GitHub.
- Compile the package (using Maven).
- Turn on a Dataproc cluster.
- Submit the import/export job to your cluster.
- Turn off the Dataproc cluster.

Let's start by going through the preparation work you'll need for both imports and exports.

NOTE If you don't have Java set up on your machine, you can always use the Google Cloud Shell, which is available in the Cloud Console in the top right-hand corner of the screen, next to the search box, and comes with all the tools preinstalled and configured.

The first thing you need to do is download the import/export package from GitHub and jump into the Dataproc example, shown next:

```
$ git clone https://github.com/GoogleCloudPlatform/cloud-bigtable-examples.git
$ cd cloud-bigtable-examples/java/dataproc-wordcount
```

Next you need to compile the package. To do this, you'll use Maven (`mvn`), which is a popular build manager for Java. (If you're using Ubuntu, you can install Maven by running `apt-get install maven`.) When compiling, you'll pass in both the project ID and the instance ID that you'll be talking to. Note that the format for passing in data via the command line is to use `-D` with no space following it, which might look strange to non-Java developers:

```
$ mvn clean package -Dbigtable.projectID=your-project-id \
    -Dbigtable.instanceID=your-bigtable-instance-id
```

**Make sure to substitute
in your project ID and
Bigtable instance ID
when running this
command.**

After the build command finishes, you'll be left with a JAR file in the `target` directory, which is what will do the heavy lifting to import and export data. Let's look first at how you'll export the data that you added to your `todo` table.

First you need to decide where to put this data. The easiest and recommended choice is to use a Google Cloud Storage bucket, so you'll create one. Because your Bigtable cluster is in the `us-central1-c` zone, let's make sure your bucket lives in the same region. You can do this with the `gsutil` command, as shown in the next listing.

Listing 7.7 Create a new bucket in the same location as your Bigtable instance

```
$ gsutil mb -l us-central1 gs://my-export-bucket
Creating gs://my-export-bucket/...
```

Now you can create a Dataproc cluster in the same zone as your Bigtable instance and deploy your export operation to the cluster, as the following listing shows.

Listing 7.8 Create a Dataproc cluster, and submit an export job to it

**If you're only testing, it might save some money to
use a single-node Dataproc cluster. If you are
exporting a lot of data, leave this flag off.**

```
$ gcloud dataproc clusters create my-export-cluster --zone us-central1-c \
    --single-node

$ gcloud dataproc jobs submit hadoop --cluster my-export-cluster \
    --jar target/wordcount-mapreduce-0-SNAPSHOT-jar-with-dependencies.jar \
    -- \
    export-table todo gs://my-export-bucket/todo-export-2016-11-01
```

Make sure that the `--` is separated from the `export-table`. The double dashes by themselves tell `gcloud` to forward these flags to the Java code.

After running these two commands—which might take a little while, don’t worry—your data should be available as Hadoop sequence files in the bucket you created. You can verify this by listing the contents of the bucket using gsutil, as shown in the next listing.

Listing 7.9 List the contents of your bucket to see exported data

```
$ gsutil ls gs://my-export-bucket/todo-export-2016-11-01/  
gs://my-export-bucket/todo-export-2016-11-01/  
gs://my-export-bucket/todo-export-2016-11-01/_SUCCESS  
gs://my-export-bucket/todo-export-2016-11-01/part-m-00000
```

Now let’s look at how you might reimport the same sequence files into a table. To do this, you can use the same Dataproc cluster and JAR file that we built, but you make a few tweaks to the parameters. You also need to make sure there’s a table ready to accept the imported data, which you can do quickly using the following code.

Listing 7.10 Use Node.js to create the table to hold imported data

```
const bigtable = require('@google-cloud/bigtable')({  
  projectId: 'your-project-id'  
});  
  
const instance = bigtable.instance('test-instance');  
instance.createTable('todo-imported', {  
  families: ['completed']  
});
```

In this example, you’re calling the new table `todo-imported`.

Note that you must specify the same column families again. Skipping this will lead to errors during the import process.

After you have the table set up, you submit a job to Dataproc to load the data from your bucket and import it into your newly created table, as shown in the following listing.

Listing 7.11 Submit the import job to Cloud Dataproc

```
$ gcloud dataproc jobs submit hadoop --cluster my-export-cluster \  
--class com.google.cloud.bigtable.mapreduce.Driver \  
--jar target/wordcount-mapreduce-0-SNAPSHOT-jar-with-dependencies.jar -- \  
import-table todo-imported gs://my-export-bucket/todo-export-2016-11-01
```

Note that you’ve changed `export-table` to `import-table`, and the table name changed from `todo` to `todo-imported`. Also, although the value for the data location is the same, this time that data is being used as source data rather than as a destination of exported data.

And that’s it. At this point you should have a pretty strong understanding of both the theory underlying Bigtable as well as the operational aspects of using it. Let’s take a moment to look at how much all of these things will cost.

7.4 Understanding pricing

Similar to Cloud SQL (see chapter 4), Cloud Bigtable splits pricing into a couple of areas: compute costs (hourly rate for running nodes), storage costs (monthly rate for GB stored), and network costs (per GB rate for data sent outside the same region). These costs vary depending on the location in which Bigtable is running, making the pricing model pretty straightforward. A few things are worth mentioning, however.

First, the minimum size of an instance is three nodes, so the minimum hourly rate for any production instance is technically three times the per-node hourly rate. Next, storage can be either on solid-state drives (SSDs) or standard hard disks (HDDs), and each of these has different prices. Your choice of how to store data affects your monthly per-GB cost. Finally, networking costs are charged only for out-bound (egress) traffic and even then only when the traffic is leaving the region where the instance lives. If you send data only from a Bigtable instance to a Compute Engine instance in the same zone (or even different zones in the same region), for example, that traffic is entirely free. To make things a bit easier for US-based instances, if you happen to send traffic between different regions that are both inside the United States, traffic is billed at a discounted rate of \$0.01 per GB sent. Table 7.8 shows an overview of the costs broken down by the different locations where you can run Bigtable instances.

Table 7.8 Bigtable pricing for some locations

Location	Compute (per node-hour)	HDD (per GB-month)	SSD (per GB-month)
Iowa (US)	\$0.65 (\$1.95 minimum)	\$0.026	\$0.17
Singapore	\$0.72 (\$2.16 minimum)	\$0.029	\$0.19
Taiwan	\$0.65 (\$1.95 minimum)	\$0.026	\$0.17

Let's take a look at an example Bigtable instance running in Iowa starting with three nodes and about 100 GB of data on SSDs. Then you'll look at growing that to ten with 10 TB of data. To start, your three nodes in Iowa will cost \$0.65 per node-hour (\$1.95 per hour), meaning the total monthly cost is about \$1,400 per month for the nodes ($\$0.065 * 3 \text{ nodes} * 24 \text{ hours per day} * 30 \text{ days per month}$). On top of that, you have 100 GB of data stored on SSDs, adding an additional \$17 ($\$0.17 * 100 \text{ GB per month}$). In this case, the storage cost is a rounding error on top of the compute cost, so you can round this off to around \$1,400 per month for this cluster.

If you were to expand this to ten nodes and 10 TB of data, your numbers would jump up a bit, as you'd expect. The compute cost is now \$4,680 per month ($\$0.65 * 10 \text{ nodes} * 24 \text{ hours per day} * 30 \text{ days per month}$), and the storage cost jumps to \$1,700 per month ($\$0.17 * 10,000 \text{ GB per month}$). This brings your grand total for this (pretty large) Bigtable instance to about \$6,400 per month. Note that you're assuming all traffic is staying inside the same region (for example, we're interacting with Bigtable

using Compute Engine instances nearby), so there's no egress network cost for serving data around the world.

At this point you should have a good grasp about how much Bigtable costs, but you may still be wondering, "Why would I use Bigtable over something else?" or more specifically, "When is it a good fit for my project?" Let's spend a bit more time going through the benefits and drawbacks of Bigtable, which may help inform your decision about whether to use it in your project.

7.5 When should I use Cloud Bigtable?

To get an overview, let's look at the scorecard shown in figure 7.15 for Bigtable and go through the attributes point by point.

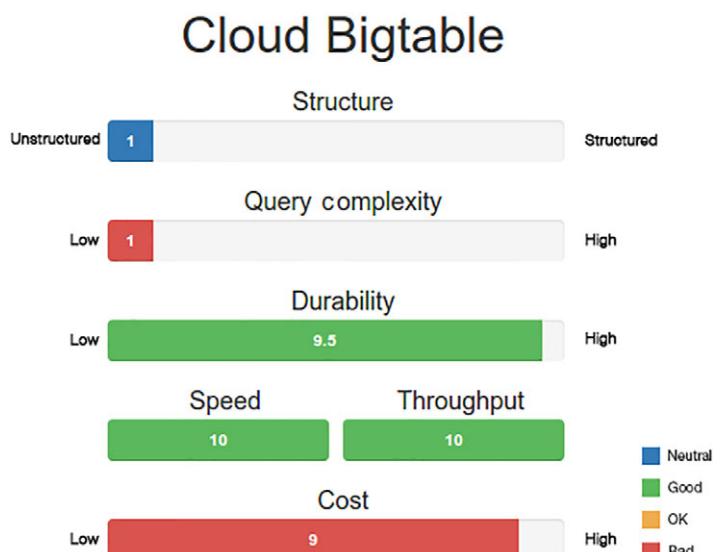


Figure 7.15 Scorecard for Cloud Bigtable

7.5.1 Structure

As you've learned throughout the chapter, Bigtable is loosely structured when compared to the other storage systems we've seen. Although it does require specific column family names, the column qualifiers can be dynamic and created on the fly, meaning the column qualifiers can themselves store data.

In many ways, the structured aspect of Bigtable applies more to the concepts than it does to the data. Inside that conceptual framework, the column qualifiers and the values can be anything you want them to be. This freedom, however, means that you lose out on many of the more advanced features that you might be used to in other storage systems.

7.5.2 **Query complexity**

If a strict key-value storage system (such as Memcache) is an example of a system that offers the minimal query complexity possible, Bigtable should be considered a hair above that. As you saw earlier, Bigtable can mimic the key-value querying by constructing a row key and asking for the data with that row key, but it allows you to do something critical that services like Memcache don't: scan the key space.

In most key-value systems, you can request a given key but have no way of asking for all keys matching a specific prefix (or even "all keys"). In Bigtable you're able to specify a range of keys to return, making it important to choose row keys that serve this purpose. In some ways, this is a bit like being able to choose one and only one index for your data. Therefore, many things you're used to with relational databases are not possible:

- Querying based on data inside a row (SELECT * FROM employees WHERE name = 'Jimmy' AND age > 20)
- Computing new values based on data (SELECT AVERAGE(age) FROM employees)
- Joining sets of data together in a query (SELECT * FROM employees, employers WHERE employees.employer_id = employer.id)

7.5.3 **Durability**

Because all Bigtable data is stored on persistent disk, the chances of losing any stored data are extraordinarily low. But like any storage system, in addition to worrying about the underlying storage system (the physical disks), you have to consider the software system's persistence model.

In Bigtable's case, the system is built to shard data across multiple machines (and multiple tablets) so that the load is spread evenly across the system. Also, Bigtable's row-level atomicity means that when writing a row, the write either persists or fails, so losing data isn't something to worry about.

7.5.4 **Speed (latency)**

One of the main reasons to use Bigtable is its performance. The whole reason you're not able to run fancy, complex queries or operate atomically on more than a single row means that things like reading a single row are incredibly fast (typically below 10 ms, even with thousands of writes per second). Though some in-memory storage systems are capable of this, few can maintain this level of speed without sacrificing durability or concurrency (for example, throughput). The system is able to keep this latency low because it automatically moves your data around, so choosing a row key is important and may have adverse effects on performance if done poorly.

7.5.5 **Throughput**

As we hinted previously, throughput on Bigtable is best in class for storage systems. The same aspects of data redistribution that help to keep latency low also help keep

throughput high. Because Bigtable uses SSD disks, random reads and writes are extremely fast, and many of them happen concurrently. By combining the high performance of the low-level storage with the even load balancing across tablets, Bigtable clusters as a whole can handle extraordinarily large levels of throughput, with measurements starting in the tens of thousands of requests per second.

Further, adding more capacity to the cluster is as simple as adding more nodes. Because Bigtable will shift data to nodes that are underused, adding more nodes is the same as having empty nodes with no traffic to them. As you'd expect, Bigtable notices these empty and idle nodes, shift tablets to them based on the traffic to those tablets. At the end you have a larger cluster with traffic evenly balanced across each node, improving your overall throughput.

7.5.6 Cost

Bigtable's primary benefit above all else is its performance. Unlike some of the other storage systems discussed so far, Cloud Bigtable has no free tier and has a minimum cluster size of three nodes, which translates to about \$1,400 per month as a minimum. This is quite a change from the \$30 per month minimum for Cloud SQL.

In short, because of this high initial and on-going cost for Cloud Bigtable, you should use it only when you absolutely need it due to the scale you expect to see. If you can make do with something else (for example, MySQL), it's probably going to be a better fit.

7.5.7 Overall

As you might notice, most of the value from Bigtable comes from performance with both speed and throughput topping the charts. Aside from the performance, Bigtable acts much like any other key-value store, with almost no structure (you have a row key that points to mostly unstructured data) and little supported query complexity (you ask for a row key, or sequence of row keys, and get back subsets of data). If you're still wondering why you'd want to use Cloud Bigtable, don't worry, because you're not alone. Bigtable is incredibly powerful, but the lack of common features (such as secondary indexes) tends to be a big drawback for most projects. Why might you want to use Bigtable?

First and foremost, Bigtable should always be on the list of options whenever you have a large dataset. In this case, large typically means terabytes or more. If your data is only in the gigabyte range (which is typical for a database storing user information), you're probably better off with something else.

Second, Bigtable is great for usage sustained over a long period of time. In this case, a long period of time is measured in hours or days rather than seconds or minutes. If you use Bigtable to store and query data only infrequently, you're probably better off with some other analytical storage system.

Third, Bigtable is likely to be a good fit if you need extraordinarily high levels of throughput. In this case, extraordinarily high means tens to hundreds of thousands

of queries every second. If you need only a few queries per second, you have many options and may want to start with another system.

Finally, if you need basic access to your data in the form of lookups and simple scans across keys, then Bigtable may be a good fit. If you need more than this (like secondary indexes), you're probably better off using a relational database. To make this more concrete, let's look briefly at our example applications and see whether Cloud Bigtable might be a good fit.

To-Do List

As we mentioned already, the To-Do List application, which stores history of items to complete, along with when someone finished the items, definitely won't need the levels of performance offered by Bigtable and is primarily application-focused data rather than analytical data. This means that even though we used it as our example, it isn't a good fit for Cloud Bigtable, as shown in table 7.9.

Table 7.9 To-Do List application storage needs

Aspect	Needs	Good fit?
Structure	Structure is fine; not necessary, though.	Sure
Query complexity	We don't have that many fancy queries.	Not really
Durability	High; we don't want to lose stuff.	Definitely
Speed	Not a lot.	Overkill
Throughput	Not a lot.	Overkill
Cost	Lower is better for all toy projects.	Overkill

In short, Cloud Bigtable is acceptable on a few of the storage needs, not a great fit when it comes to the queries you'd want to run, and completely overkill for your performance requirements. You certainly could use Bigtable to store To-Do List data, but it's going to be way more expensive than you need. You'll likely be frustrated as your application grows in complexity far more than it does in scale, and you realize that you need to run more advanced queries over a relatively small amount of data.

E*EXCHANGE

As we saw before, E*Exchange, the online trading platform that allows people to trade stocks and bonds online, requires far more complicated queries for customer data, which is one aspect that Bigtable is particularly bad at. See table 7.10.

Table 7.10 E*Exchange storage needs

Aspect	Needs	Good fit?
Structure	Yes, reject anything suspect;; no mistakes.	Not really
Query complexity	Complex; we have fancy questions to answer.	Definitely not

Table 7.10 E*Exchange storage needs (continued)

Aspect	Needs	Good fit?
Durability	High; we cannot lose stuff.	Definitely
Speed	Things should be pretty fast.	Probably overkill
Throughput	High; we may have lots of people using this.	Probably overkill
Cost	Lower is better, but willing to pay top dollar.	Definitely

Although Bigtable happens to fit the application's durability requirements, the performance requirements are yet again overkill. Additionally, the query complexity needed by an online trading platform is difficult to handle with a storage system like Bigtable. Finally, the need for data validation and structure at the storage layer is not what Bigtable is designed for, so these features aren't available. This means that Bigtable isn't great for the trading platform's business-level data. What about the stock trading data?

We didn't discuss this before, but what if E*Exchange wanted to store historical stock trading data? This data will have lots of small events, including the stock symbol, the time, the trade amount, and the price paid. And there are millions (or more) of these every day, even if counting only larger orders that are filled. Would this aspect of E*Exchange be a good fit for Cloud Bigtable? See table 7.11.

Table 7.11 E*Exchange stock trading storage needs

Aspect	Needs	Good fit?
Cost	Lower is better, but willing to pay top dollar.	Definitely
Durability	Medium; a few items can be lost.	Definitely
Query complexity	Simple lookups and scans.	Definitely
Speed	Things should be pretty fast.	Probably overkill
Structure	Not really.	Definitely
Throughput	High; we have tons of traffic.	Definitely

It seems like the stock trading data might be an excellent fit for Cloud Bigtable, even though single row latency might be overkill.

INSTASNAP

InstaSnap, the popular social media application that lets people post images and follow and like others images, has a few requirements that seem to fit well and only a couple that are a bit off, as shown in table 7.12.

Table 7.12 InstaSnap storage needs

Aspect	Needs	Good fit?
Structure	Not really; structure is pretty flexible.	Definitely
Query complexity	Mostly lookups; no highly complex questions.	Definitely
Durability	Medium; losing things is inconvenient.	Sure
Speed	Queries must be fast.	Definitely
Throughput	High; Kim Kardashian uses this.	Definitely
Cost	Lower is better, but willing to pay top dollar.	Definitely

As we saw when evaluating InstaSnap earlier, the biggest issue is the single query latency, which needs to be extremely fast and Bigtable happens to excel at. The performance requirements are certainly met by Bigtable, and the fact that most of the queries are simple lookups or scans means that Bigtable's query complexity limitations shouldn't be a cause for concern. In short, though InstaSnap could potentially run using something providing more complex queries (such as Cloud Datastore), as the service grows larger and larger, something like Cloud Bigtable is likely to be the better overall fit.

7.6 **What's the difference between Bigtable and HBase?**

If you're familiar with HBase, you should know how Cloud Bigtable is different. First, a few of the advanced features aren't available with Bigtable, such as co-processors, where HBase allows you to deploy some Java code to be run on the server with your HBase instance. Bigtable is written in C, so it would be tricky to connect HBase co-processors (written in Java) to the Bigtable service (written in C).

Second, due to an underlying design difference between Bigtable and HBase, Bigtable (currently) is able to scale more easily to a larger number of nodes and, as a result, can handle more overall throughput for a given instance. HBase's design requires a master node to handle fail-overs and other administrative operations, which means that as you add more and more nodes (in the thousands) to handle more and more requests, the master node will become a performance bottleneck. Cloud Bigtable, though similar to HBase in many respects, doesn't have this same design limitation and will scale to arbitrarily large cluster sizes without introducing this same performance bottleneck.

Last are the typical cloud-like benefits, in particular the automatic upgrade of binaries (you don't have to upgrade Bigtable like you do on HBase nodes), as well as easy and stable resizing of your cluster (you can change how much serving capacity you have with zero downtime), and the obvious "pay for what you use" principle applies to data storage.

7.7 Case study: InstaSnap recommendations

As you may recall, InstaSnap was your sample application that allows users to post images and share them with their followers and has some potentially large scaling requirements (after all, some celebrities might use InstaSnap). To demonstrate one way InstaSnap could use Bigtable, let's imagine that you want to build a recommendation system for InstaSnap.

The code for querying a table for data is not all that complicated, but choosing the right table schema can be pretty complicated. As a result, rather than zooming in on code samples, let's focus on designing a system that can make recommendations and the tables that will store this data. To start, let's look at the various components and how they talk to each other. See figure 7.16.

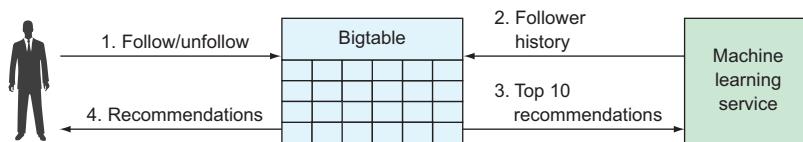


Figure 7.16 Overview of InstaSnap's recommendation pipeline

First, you need to store an overview for each user that tracks who they follow as well as who follows them. Based on these signals, you could construct some sort of machine-learning service that would use that information to notice various overlapping patterns and ultimately come up with a list of recommendations. To make this more concrete, imagine that you followed George Clooney on InstaSnap. The machine-learning service could notice that the majority of people who follow George Clooney also happen to follow Leonardo DiCaprio. Based on this information, it seems likely that Leo might be good a suggestion.

To keep things relatively simple and focused on Bigtable, we're going to assume that this machine-learning service is somewhat magical and uses the “A follows B” data to come up with recommendations. Given that, let's look at how you might design a schema so that InstaSnap can store all the necessary data in Bigtable.

7.7.1 Querying needs

Before you start, you need to figure out how you want to query your data. This machine-learning service has a few questions it needs to ask to get the data it needs about a given user, such as the following:

- Who does user X follow?
- Who follows user X ?

In short, it seems like you need provide lists of followers, both of a particular user and by a particular user. If you provide the machine-learning service with these answers on-demand, it can probably use that information to come up with some

recommendations. Additionally, the recommendation results need to be stored back in Bigtable, and InstaSnap will need to ask, “Who’s recommended based on following user X?” Now that you have an idea of the questions you’d want to ask, let’s look at some possible schemas and decide which fits best.

7.7.2 Tables

Based on the questions you need answered, it looks like there should be a total of three tables:

- User’s followers and followees (users)
 - Who does user X follow?
 - Who follows user X?
- Recommendation results (recommendations)
- If I follow user X, who else is recommended?

Let’s start by looking at the users table, which will let you figure out who a user follows as well as who follows that user.

7.7.3 Users table

When it comes to storing followers, you could use either a tall table or a wide table. Let’s look at the differences, starting with a tall table. As you learned earlier, a tall table has lots of rows to represent data and accomplishes this by adding information to the row key. Then, to get lists of related information that spans many rows, you use a prefix scan over the rows. Table 7.13 shows how you might store some rows representing one user following another user.

Table 7.13 Followers represented as a tall table

Row key	Follows (column family)	
	Username	
14ccc4ac79c3758	user-2	
79c3758f5f7b45b	user-3	
f5f7b45b14ccc4ac	user-1	
f5f7b45b79c3758	user-2	

Recall that you generate the row key by hashing both the follower and the followee and concatenating the results. For example, user-1 following user-2 would have a row key of $\text{crc32c}(\text{user-1}) + \text{crc32c}(\text{user-2})$, which turns out to be 14ccc4ac79c3758. As expected, this table structure makes it easy to ask the question “Does user-1 follower user-2?” All you have to do is compute the hashes and retrieve the row. If the row exists, then the answer is yes.

You can also request all the people that a user follows using a prefix scan—compute the hash of the user you’re interested in, and use that value as the prefix. For example, finding the users that `user-1` follows would be a prefix scan of `crc32c(user-1)`, which comes out to `14ccc4ac`. Finally, it’s easy to add and remove followers by adding and removing rows corresponding to the mappings.

What about finding all the followers of a given user? How do you do this? It turns out that with this type of tall table, finding everyone following a given user can’t be done with a simple table scan. You can do a prefix scan, which asks, “Who does the prefix follow?” but there’s no way to do a suffix scan, which asks, “Who follows the suffix?” If you think about it, even if a suffix scan existed, the row keys are in lexicographical order, so the idea of scanning based on a suffix runs against what Bigtable was designed to do. You’re stuck with this so far, so let’s check a few other options to answer this question.

One option that would work with a tall table is to store two rows for the bidirectional relationship. You store one row saying “A follows B” and another row saying “B is followed by A,” using a special token between the `crc32c` hashes of A and B to denote “follows” or “is followed by.” This might look like table 7.14.

Table 7.14 Followers represented as a tall table

Row key	Follows (column family)
	Username
<code>14ccc4ac > 79c3758</code>	<code>user-2</code>
<code>14ccc4ac < f5f7b45b</code>	<code>user-3</code>
<code>79c3758 > f5f7b45b</code>	<code>user-3</code>
<code>79c3758 < 14ccc4ac</code>	<code>user-1</code>
<code>79c3758 < f5f7b45b</code>	<code>user-3</code>
<code>f5f7b45b > 14ccc4ac</code>	<code>user-1</code>
<code>f5f7b45b > 79c3758</code>	<code>user-2</code>
<code>f5f7b45b < 79c3758</code>	<code>user-2</code>

In this table you can see that you’ve constructed a strange-looking, but completely valid, row key that stores rows for both “A follows B” (`crc32c(a) > crc32c(b)`), and “B is followed by A” (`crc32c(b) < crc32c(a)`). If you want to ask, “Who does A follow?,” you do a prefix scan on `crc32c(a) >`, and if you want to ask, “Who follows A,?” you do a slightly different prefix scan on `crc32c(a) <`. The value stored in the row is always the unknown side of the query, which in this case is the user on the right side of the arrow. Although you know the value that you hashed to run the prefix scan, you can’t go backward from a hash to the user.

This table schema will certainly work, but it isn't space-efficient because it's technically storing twice the number of rows to convey the same information. The row `crc32c(a) > crc32c(b)` (`A` follows `B`) conveys the same information as `crc32c(b) < crc32c(a)` (`B` is followed by `A`). Because none of the tall table schemas look like a perfect fit, let's look at a wide table to see if it works any better.

In this case, a wide table might store a row key for each user and then a column family to store other users being followed. Inside that column family, each user being followed gets its own column with a placeholder value. This might look like table 7.15.

Table 7.15 Followers represented as a wide table

Row key	Follows (column family)		
	user-1	user-2	user-3
user-1		1	
user-2			1
user-3	1	1	

This table structure makes it easy to ask, "Who does A follow?" by asking for the row for the user and the `Follows` column family. All the keys in the returned map will be the people that A follows. Likewise, it's easy to ask, "Does A follow B?" because you'd ask for the row for the user and a specific column inside the `Follows` column family, because it will have the flag value set for the target user (in this case, B).

But what about finding everyone followed by a single user? ("Which users follow A?") It looks like this schema is going to run into the same problem as before where going one direction ("Who does A follow?") is fast and easy, but the other direction ("Who follows A?") is tricky. Let's see if you can tweak this schema to handle both directions. You could add a second column family that represents the inverse relationship ("B is followed by A") and store followers in that map as well. Then you'd ask for that column family to answer the other side of the question ("Who follows A?"). This would make your new schema look like table 7.16.

Table 7.16 Bidirectional followers represented as a wide table

Row key	Follows (column family)			Followed by (column family)		
	user-1	user-2	user-3	user-1	user-2	user-3
user-1		1				1
user-2			1	1		1
user-3	1	1			1	

The Follows column family (the left side of the table) helps answer the question “Who does A follow?” by storing a sparse map with flag values set. For example, “Who does user-1 follow?” would return `{"user-2": 1}`. The Followed by column family (the right side of the table) answers the question “Who follows A?” by storing the same style of sparse map. For example, “Who follows user-2?” would return `{"user-1": 1, "user-3": 1}`.

What are the downsides of this schema? If you use this wide table, you’ll need to update two rows for every follow and unfollow action. For example, if `user-3` wants to unfollow `user-2`, you need to do the following two actions:

- 1 Update row `user-3` and delete the column `user-2` from the follows column family.
- 2 Update row `user-2` and delete the column `user-3` from the followed by column family.

This presents a bit of an issue because Bigtable doesn’t support the ability to change multiple rows in a single transaction. How big of a problem is this?

The failure condition of only one of the two actions happening (but not both) would be strange but not critical. If you ended up in this bad state, depending on the question you ask, you might get different results. For example, this would mean that “Does A follow B?” might say “Yep!”, but asking “Is A followed by B?” might say “Nope!” One easy fix for this is to always ask this question the same way (that is, always ask, “Does A follow B?” and never ask, “Is B followed by A?”).

Your next problem concerns listing followers in both directions. If you have a failure and end up in this bad state, then looking at the list of people that A follows might show B in that list, but looking at the list of people followed by B might not show A in that list. In the grand scheme of things, this seems like it’d be a tough consistency issue to spot, so much so that you’d have to go specifically looking for this problem, and even then it’d be tough for a human to notice. Given that, this doesn’t seem like a big deal.

Overall, it seems like the wide table is probably going to be easier to manage, so let’s next look at how data-recommendation data might be stored in Bigtable.

7.7.4 Recommendations table

The recommendations table brings everything together. In short, it’s the table that stores the output of your machine-learning job, so that you can come up with a set of recommendations when someone on InstaSnap follows someone new. It turns out to be pretty simple.

When you’re presenting some recommendations of who else to follow, you’ve had a follow event, meaning your question would be phrased as “Given I’ve followed user X, whom else should I follow?” Your queries are user-based, which makes for an easy row key (the same as you had with your Users table).

The column family would be called Recommendations, with a column for each user that’s recommended, with a score set as the value rather than a simple flag. An example of how this might look is shown in table 7.17.

Table 7.17 Recommendations table example

Row key	Recommendations (column family)		
	user-1	user-2	user-3
user-1		0.5	
user-2			0.4
user-3	0.4	0.6	

Using this table design, you might ask, “I followed user-1. Whom else should I follow?” This translates to asking for the user-1 row of the recommendations table. The results would be `{"user-2": 0.5}`, and the InstaSnap application would show that as a suggested recommendation. The application could sort through the list of users by their values, prioritizing the more highly recommended users over others. Further, to keep the table clean, the machine-learning job would overwrite stale data every time the recommendation job runs.

7.7.5 Processing data

Because it’s likely that running a deep learning algorithm isn’t exactly a quick operation, you should probably design your system so that the learning happens periodically, and then requests for suggestions would be pulled from cached results of the previous run. You can use Bigtable as the middle man in this process. At a high level, referring to figure 7.16, getting recommendations would fall in the following two steps:

- 1 Every so often, the machine-learning job starts and comes up with a set of follow recommendations.
- 2 Whenever a user follows someone new, show them a set of recommendations related to that action (“You might also be interested in ...”).

You can use Bigtable as an intermediary where it reads follower data from Bigtable as designed earlier, computes recommendations, and then stores the results of that computation back in Bigtable. Then, when you need to show recommendations to a user, it’s a simple read from those results in Bigtable. Let’s look at each step and see what the code might look like when interacting with Bigtable as an intermediary.

First, the machine-learning job retrieves lists of followers. This can be on a single-row basis (for example, `getFollowers('user-1')`) or as a full-table scan if the job is reprocessing the recommendations. Let’s start with a simple way of grabbing the followers for a given user, shown in the next listing.

Listing 7.12 Getting followers of a single user

```
const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});
```

```

const instance = bigtable.instance('test-instance');
const table = instance.table('users');

const getFollowers = (userId) => {
  const row = table.row(userId);
  return row.get(['followed-by']).then((data) => {
    return Object.keys(data);
  });
}

The row is nothing more than the
user ID, so you can jump right to it.

```

Start by pointing to the Users table.

You ask specifically for the column family storing the followers, and return the keys (remember, the values are placeholders).

Next you need to provide a way to scan through the table. Because you're doing a full-table scan, you should partition the search space so that multiple VMs can all pull data out of Bigtable. You can use the `sampleRowKeys()` method to give you the borders of tablets to help you decide where to split the data, as shown in the next listing.

Listing 7.13 Finding the split points and returning them as key range filters

```

const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

const instance = bigtable.instance('test-instance');
const table = instance.table('users');

const getKeyRanges = () => {
  return table.sampleRowKeys().then((data) => {
    const ranges = [];
    const currentRange = {start: null, end: null};
    for (let splitPoint in data[0]) {
      currentRange.start = currentRange.end;
      currentRange.end = splitPoint.key;
      ranges.push(currentRange);
    }
    return ranges;
  })
}

```

First, use the `sampleRowKeys` method to find the split points (the borders) that you can use to split how you consume the rows in the table.

Take the end of the previous range, make it the start of the next, and make the new split point the end of the current range.

Add the range to the list of results.

As you can see, this method will ask Bigtable for the split points (or the borders) to use when splitting the work of asking for all of the data in the table, and return it as a list of ranges. After this, it's a matter of using the `createReadStream` method to scan between those ranges.

Listing 7.14 Scanning the table in chunks

```

const bigtable = require('@google-cloud/bigtable')({
  projectId: 'your-project-id'
});

```

```

const instance = bigtable.instance('test-instance');
const table = instance.table('users');

getKeyRanges().then((ranges) => {
  for (let range in ranges) {
    runOnWorkerMachine(() => {
      table.createReadStream({
        start: range.start, end: range.end
      }).on('data', (row) => {
        addRowToMachineLearningModel(row);
      });
    });
  }
});

```

Start by fetching the split points for the table.

When creating the read stream, use the start and end keys of the range as provided from the getKeyRanges method.

Finally, you perform some magic that adds the new row to the model to be used when making recommendations with machine learning.

Here you use the fictitious runOnWorkerMachine, which would take the method provided and forward it to a separate worker (perhaps by broadcasting a message to perform the work).

In this case, despite the need to use a couple of fake methods (runOnWorkerMachine and addRowToMachineLearningModel), you can see how you would scan through the table using multiple consumers of data.

Summary

- Bigtable is a large-scale data storage system, originally built for Google's web search index.
- It was designed to handle large amounts of replicated, rapidly changing data and can be queried quickly (low latency) with high concurrency (high throughput), while maintaining strong consistency throughout.
- Cloud Bigtable is a fully managed version of Google's Bigtable, exposing almost all of the features available in Google's original version.
- Bigtable is likely a good fit if you have a large amount of data and primarily access it using key lookups or key scans but not a great fit if you need secondary indexes or relational queries.



Cloud Storage: object storage

This chapter covers

- What is object storage?
- What is Cloud Storage?
- Interacting with Cloud Storage
- Access control and lifecycle configuration
- Deciding whether Cloud Storage is a good fit

If you've ever built an application that involves storing an image (such as a user's profile photo), you've run into the problem of deciding where to put that photo. Chances are that to keep making progress on your project, you went with the easiest place: right in your database or on your local filesystem. This works for a little while, but if your website becomes popular, the disk that holds all of these images and videos might get overwhelmed. This is the exact problem that object storage services aim to solve.

In addition to storing data correctly, a primary design goal of these systems is to reduce complexity of the underlying disks and data centers and instead provide a simple API for uploading and retrieving files, a bit like key-value storage for large values with automatic replication and caching around the world.

Of all the cloud services that exist today, object storage tends to be one of the most common and most standardized. For example, Google Cloud Storage and Amazon S3 have the same concepts and are capable of speaking the same XML API. Although object storage systems share many similarities, they tend to have slight differences in the pricing model, replication strategy, or storage class.

Google Cloud Storage is the default object storage system on Google Cloud Platform (GCP), so let's look at the key concepts that you need to understand to store your data.

8.1 Concepts

Cloud Storage, like many other object storage systems (such as Amazon's S3), uses two key concepts: buckets and objects.

8.1.1 Buckets and objects

You can think of a *bucket* as a container that stores your data. The bucket has a globally unique name, rather than one unique to your project, as well as a few other options you can set, such as the geographical location and the storage class (both discussed later). In many ways, you can think of buckets as "disks," in the sense that you can choose what type of disk you want (for example, SSD, regular disk, replicated disk across the United States, and so on) and where you want that disk to live (for example, Europe or the United States).

The big difference is that this "disk" is extraordinarily large—there's no limit to how many bytes can end up in a bucket. The only limit is that each file in the bucket must not be larger than 5 terabytes. Additionally, this "disk" doesn't have the same failure semantics as a typical physical disk. The bucket itself is replicated and spread across many physical disks to maintain high levels of durability and availability.

Objects are the files that you put inside a bucket. They have a unique name inside the bucket, and as on typical file systems, slashes (/) are treated specially so that you can browse directories like on any traditional Linux system. Later we'll discuss some other advanced features (for example, storing the generation of an object), but objects themselves are straightforward: named chunks of bytes that you can retrieve on demand.

LOCATIONS

Like VMs that you turned on in Compute Engine, buckets can have locations as well. Rather than always defined a specific zone (for example, us-central1-a), however, buckets exist either at the regional level (for example, us-central1) or spread across multiple regions (for example, "United States" or "Asia"). VMs can only exist in a single place, but data can be copied and live in multiple places simultaneously. Why might you choose these different locations for your data? It depends on what you need.

If you need your data to be always available, even if lightning strikes all of the data centers in the us-central1 region, you probably want to create a multiregional bucket (for example, set the location to "United States"). A multiregional bucket is by

definition replicated across several regions, which means that even a complete outage of all data centers in a single region can't stop your data from being available.

If you're concerned about latency between your VMs and your data on GCS, you might want to choose a specific region (for example, us-east1) for your data. If you make a request from your VM in us-east1-a to a bucket located in "United States," that request could end up going to either us-east1 or us-central1, so the data may end up taking the long way to you. If you're unsure where you'll put your VMs (or if you'll even have any VMs accessing your data at all), you might want a multiregional bucket to ensure data is always closest to where you or your customers are.

Finally, as you'll learn later in the section on pricing, if you make a mistake and put a bucket far away from your VMs, you'll end up paying a premium for reading your data due to cross-region network transfer fees. This can range from being obvious (for example, a bucket in "Asia" and your VMs in us-central1-a) to the much more subtle (for example, a bucket in us-central1 and your VMs in us-east1-b), so it's important to be careful or you may accidentally put your data far away from where you need it.

8.2 Storing data in Cloud Storage

As always, you have many ways to get started with Cloud Storage, so we'll walk through a few different ways, starting with the Cloud Console, then moving on to the command line with the Cloud SDK (`gsutil`), and then using your own code in Node.js (`@google-cloud/storage`).

Before you can start storing data, you first have to create a bucket. Because bucket names need to be globally unique, you won't be able to use the same bucket name used here, so feel free to append your name to the bucket to keep it unique. Start by heading over to the Cloud Console and choosing Storage from the left navigation. You should see a prompt to create a bucket that looks like figure 8.1.

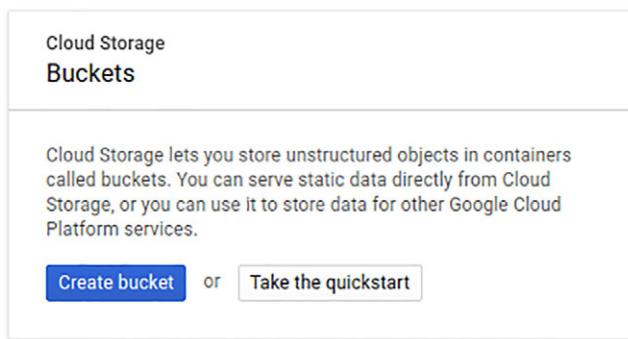


Figure 8.1 Your first visit to the Cloud Storage UI

When you click Create bucket, you'll see a field for the name of the bucket, as well as drop-down selectors for the storage class and location. For now, leave the drop-downs

as they are (we'll discuss those later), and enter a unique name for your bucket. Here you're using `my-first-bucket-jjg`. See figure 8.2.

The screenshot shows the 'Create a bucket' dialog. At the top left is a back arrow icon and the title 'Create a bucket'. Below the title is a 'Name' field containing 'my-first-bucket-jjg'. A note below the name field states: 'Must be unique across Cloud Storage. Privacy: Do not include sensitive information in your bucket name. Others can discover your bucket name if it matches a name they're trying to use.' The 'Default storage class' section contains five options: 'Multi-Regional' (selected), 'Regional', 'Nearline', 'Coldline', and 'Standard'. Each option has a description and a note about its best use case. Below this is a 'Multi-Regional location' section with a dropdown menu set to 'United States'. At the bottom are 'Create' and 'Cancel' buttons.

← Create a bucket

Name ?
Must be unique across Cloud Storage. Privacy: Do not include sensitive information in your bucket name. Others can discover your bucket name if it matches a name they're trying to use.

my-first-bucket-jjg

Default storage class ?

Multi-Regional
Use to stream videos and host hot web content.
Best for data accessed frequently around the world.

Regional
Use to store data and run data analytics.
Best for data accessed frequently in one part of the world.

Nearline
Use to store rarely accessed documents.
Best for data accessed less than once per month.

Coldline
Use to store very rarely accessed documents.
Best for data accessed less than once per year.

Multi-Regional location
Redundant across 2+ regions within your selected location.

United States

Specify labels

Create Cancel

Figure 8.2 Create your first bucket.

Now let's explore Cloud Storage with the command line.

NOTE Cloud Storage currently has a separate command-line tool called `gsutil`. Even though it's under a different command, it's still installed and updated with the Cloud SDK. If you don't see the command on your machine, try running `gcloud components install gsutil`.

First, try listing the buckets available to you with `gsutil ls`, as shown in the following listing. (Don't forget to make sure you're authenticated with `gcloud auth login`.)

Listing 8.1 Listing your buckets with gsutil

```
$ gsutil ls
gs://my-first-bucket-jjg/
```

Now upload a simple text file with gsutil. If you have a file laying around that you want to upload, feel free to use that. If you don't, create a small text file for this example.

Listing 8.2 Uploading your first file

```
$ echo "This is my first file!" > my_first_file.txt
$ cat my_first_file.txt
This is my first file!

$ gsutil cp my_first_file.txt gs://my-first-bucket-jjg/
Copying file://my_first_file.txt [Content-Type=text/plain]...
Uploading   gs://my-first-bucket-jjg/my_first_file.txt:           23 B/23 B
```

Now look in your bucket in the Cloud Console to see if it worked, as shown in figure 8.3.

Browser	UPLOAD FILES	UPLOAD FOLDER	CREATE FOLDER	REFRESH	SHARE PUBLICLY	
Buckets / my-first-bucket-jjg						Filter by prefix...
<input type="checkbox"/> Name	Size	Type	Last modified	Share publicly		
<input type="checkbox"/> my_first_file.txt	23 B	text/plain	5/22/16, 1:06 PM	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 8.3 Checking that your file was uploaded

As you can see, the file (called an object in this context) made its way into your newly created bucket. Now access Cloud Storage from your own code. To do this, you'll need the `@google-cloud/storage` package, which you can install by running `npm install @google-cloud/storage@0.2.0`. When that's ready, you can test the waters by listing the contents of a bucket, shown in the following listing.

Listing 8.3 Listing the contents inside a bucket

```
const storage = require('@google-cloud/storage')({
  projectId: 'your-project-id'
});
const bucket = storage.bucket('my-first-bucket-jjg');
bucket.getFiles()
  .on('data', (file) => {
    console.log('Found a file called', file.name);
  })
  .on('end', () => {
    console.log('No more files!');
});
```

Make sure to plug in your bucket name and your project ID before you run the script. Afterward, you should see output that looks something like this:

```
Found a file called my_first_file.txt  
No more files!
```

What about uploading files? You're going to upload a new file. First, create `my_second_file.txt` by adding some text to a new file (for example, `echo "This is my second file!" >my_second_file.txt`), and then write a script that uploads the file, as shown in the next listing.

Listing 8.4 Script to upload a file to Cloud Storage

```
const storage = require('@google-cloud/storage')({  
  projectId: 'your-project-id'  
});  
const bucket = storage.bucket('my-first-bucket-jjg');  
bucket.upload('my_second_file.txt', (err, file) => {  
  if (err) {  
    console.log('Whoops! There was an error:', err);  
  } else {  
    console.log('Uploaded your file to', file.name);  
  }  
});
```

If you run this script, you should see a message saying the file was uploaded. After this, if you rerun the script to list files, you should see the new file listed in the results:

```
Uploaded your file to my_second_file.txt
```

Now that you understand how to interact with Cloud Storage, let's jump back to some of the topics we skipped over earlier, such as the class of storage for your buckets.

8.3 Choosing the right storage class

Just as there are different types of hard drives (for example, SSD or magnetic), Cloud Storage offers different types of buckets that you can configure in Cloud Storage. These storage classes come with different performance characteristics (both latency and availability), as well as different prices. Different use cases require different features, so Cloud Storage offers a few choices that are likely to best match the your situation.

Let's start by running through the most common one: multiregional storage.

8.3.1 Multiregional storage

Multiregional storage is the most commonly used option and the one likely to fit the needs of most applications. The flip side is that it's also the most expensive of the options available because it replicates data across several regions inside the chosen location. (The current location options are United States, Europe, and Asia.)

If you don't know exactly from where you'll be requesting your data, multiregional storage provides the best latency available due to Google's ability to cache data at the nearest edge to the requester. In addition, because the data is replicated across several different regions, it can offer the highest availability.

Multiregional storage is likely the best choice for content frequently served to lots of different destinations, such as website content, streaming videos, and mobile application data. Generally, if your users are going to wait on this data (and you want them to get it quickly), you probably want to use multiregional storage.

8.3.2 **Regional storage**

In many ways, the regional storage class is like a slimmed-down version of the multiregional storage class. Instead of replicating data across several regions inside an area (for example, "United States"), this class replicates the data across different zones inside a single region (for example, "Iowa"). Because this class doesn't spread data as far apart, it offers slightly lower availability, and latency to destinations far away from the region chosen (for example, sending data from the Iowa region to Belgium) might be slightly higher.

In exchange for this, data stored in the regional storage class costs about 20% less per GB stored, making it attractive if you happen to know where your data will be needed in the future.

8.3.3 **Nearline storage**

Nearline storage attempts to closely match the data archival use case by making a few key trade-offs that you shouldn't notice if you're using the data as intended. For example, Nearline storage offers slightly lower availability as well as higher latency to the first byte. Nearline focuses on the scenario where you don't need your data all that often, and when you do, you can wait a bit for the download to start.

In exchange for these differences, data stored in the Nearline storage class has a slightly different pricing model. This model is explored in much more detail in section 8.10, but the key difference is that in addition to the other pricing components you'll learn about, per-operation cost is slightly higher (for example, overhead of running a "get"), data retrieval is not free like it is with regional or multiregional storage, and there's a 30-day minimum cliff for data in this class. On the other hand, the cost for data in this class is around 60% less per GB stored, which means it's a great deal when it matches your system's needs.

On the other hand, if you need to make frequent changes to your data or even retrieve the data more than monthly, this storage class will end up being much more expensive than the other options. It's typically a poor choice for anything customer-facing (such as downloads on a website).

8.3.4 Coldline storage

Coldline storage is targeted at the extreme end of the data-archival spectrum—the data used primarily in the case of a serious disaster. For example, you might need to restore your database backups monthly for one reason or another, making that data a great fit for Nearline. If there's a security breach of some sort, however, and you're calling in the FBI to investigate, they might want all transaction logs for the past year. That data would be a much better fit for the Coldline storage class because you probably aren't calling the FBI monthly, but you still want the data around just in case.

Outside of this, Coldline is similar to Nearline in that it has similar per-operation costs as well as data-retrieval costs. Instead of a 30-day minimum storage duration, however, Coldline storage has a 90-day minimum and is about 30% cheaper than Nearline on a per-GB basis, making it about 70% cheaper than multiregional storage. If you happen to fit the mold for Coldline storage, using this class can save you quite a bit of money.

In general, Coldline is a great choice for scenarios that seem to fit into Nearline, but taken to an extreme. You'd want to use Coldline in scenarios where you have data that you rarely need (for example, once per year) but want to make sure that it is there when you do need it. In exchange for not needing the data often, you get a much lower price of storing it. See table 8.1 for a summary.

Table 8.1 Overview of storage classes

	Multiregional	Regional	Nearline	Coldline
Cost per GB	\$0.026	\$0.02	\$0.01	\$0.007
SLA	99.95%	99.9%	99.0%	99.0%
Data-retrieval costs	No	No	Yes	Yes
Per-operation costs	Normal	Normal	Higher	Higher
Minimum duration	None	None	30 days	90 days
Typical use case	Website data	Analytical data	Archival	Disaster archival

Generally, because the cost difference for small amounts of data (10s of GB), your safest bet is to use multiregional storage whenever you're unsure how often you'll need to access your data or how quickly you'll need it. As you start storing more data, you should take a look at your access patterns, keeping an eye specifically for whether you're accessing data in one single place or infrequently. If you see all data being accessed from a single zone (or region), it's worth looking at regional storage for the cost savings. Additionally, if you find you're not accessing certain data often, it may be a good idea to investigate using Nearline (or even Coldline if the access is *really* infrequent).

Regardless of this, all of your data is replicated and saved across Google's data centers, so you shouldn't worry about *losing* it. The storage classes are specifically about the performance (how long it takes Google to start sending you the file after you

request it) and availability, as well as the overall price per GB, but never about the *durability*. Your data's safety is the same, with a 99.999999999% durability guarantee (that's 11 total nines in case you didn't want to count).

Now that you understand some of the fundamentals, let's dig a bit deeper into the more advanced concepts. These might not seem important at first, but as you begin using Cloud Storage in more real-life scenarios, these features will become far more interesting.

8.4 Access control

I've talked about Cloud Storage being a safe place to put all of your data but haven't explained much about how to control who's able to access or modify the data after it's stored.

8.4.1 Limiting access with ACLs

So far I've discussed interacting with your data while authorized as a service account (the thing in key.json in your code examples) or as yourself (you start by typing gcloud auth login). How does it work when you want to allow others to access your data? How do you restrict who can do what?

Before we get into more detail, it might be worthwhile to say that by default everything you create is locked down to be accessible by only those people who have access to your project. If you're working alone in your project, all of your data is restricted to only you. When you add someone else for other parts of your project, they also will have access to your data in Cloud Storage. For example, if you add someone as another owner of the project, they'll be able to control your Cloud Storage data (buckets and objects) like you can, so be careful about who you add. Let's dive into some of the specific things you should understand to control who can access your data.

Cloud Storage allows fine-grained access control of your buckets and objects through a security mechanism called Access Control Lists (ACLs). These lists do exactly what you expect by letting you say which accounts can do which operations (for example, read or write).

These operations are conveyed by three roles, which mean different things for buckets and objects. See table 8.2 for an explanation.

Table 8.2 Description of roles for Cloud Storage

Role	Meaning (buckets)	Meaning (objects)
Readers	Bucket readers can list the objects in a bucket.	Object readers can download the contents of an object.
Writers	Bucket writers can list, create, overwrite, and delete objects from a bucket.	(This doesn't apply because you can't have object writers.)
Owners	Bucket owners can do everything readers and writers can do, as well as update metadata such as ACLs.	Object owners can do everything readers can do, as well as update metadata such as ACLs.

As you might expect, you control access to your objects by assigning these roles to different actors (for example, a particular user). Let's start by looking at the ACL for your bucket in the Cloud Console. You can do this by clicking the vertical three-dot button on the far right in your list of buckets and selecting Edit bucket permissions. See figure 8.4.



Figure 8.4 Choose from the menu.

When you click Edit bucket permissions you should see something like figure 8.5.

ENTITY	NAME	ACCESS
Project	editors-648816988819	Owner
Project	viewers-648816988819	Reader
Project	owners-648816988819	Owner

Figure 8.5 Edit bucket permissions.

As you can see, the default access on the bucket is based on the project, with project editors and owners having Owner access and project viewers having Reader access. Adding access to a specific person is as easy as entering their email address and choosing the access level. For example, figure 8.6 shows what it looks like to grant Reader access to your-email@gmail.com.

Adding access to a specific user means they'll need to log in with Google's traditional login, so they'll need to have a Google account.

The screenshot shows the 'jjg-personal permissions' page in the Google Cloud Storage console. It displays a table of access entries:

ENTITY	NAME	ACCESS
Project	editors-648816988819	Owner
Project	viewers-648816988819	Reader
Project	owners-648816988819	Owner
User	your-email@gmail.com	Reader

A blue button labeled '+ Add item' is visible below the table. At the bottom are 'Save' and 'Cancel' buttons.

Figure 8.6 Granting Reader access

In addition to adding access to individuals, Cloud Storage also allows you to control access based on a few other things:

- User `allUsers`, as you might expect, refers to anyone. If you give Reader access to the `allUsers` entity, the resource will be readable by anyone who asks for it.
- User `allAuthenticatedUsers` is similar to `allUsers`, but refers to anyone who's logged in with their Google account.
- Groups (for example, `mygroup@googlegroups.com`) refer to all members of a specific Google Group. This allows you to grant access once and then control further access based on group membership.
- Domains (for example, `mydomain.com`) refer to a Google Apps managed domain name. If you use Google Apps, this is a quick way to limit access to only those who are registered as users in your domain.

As I hinted earlier, in addition to setting permissions on your bucket, you can also set these similar permissions on your individual objects, but doing so might raise questions about how the two lists interact. For example, what happens if you're an Owner for the bucket, but the object is readable by only a single person (not you)?

The answer is quite simple: each of the permissions conveys specific activities that are allowed, so there's no hierarchy of permissions that trickle down. For example, imagine that you have Owner access to a bucket but only have Reader access to an object. In this scenario, although you can manipulate any data inside the bucket, you

can't update the metadata for the object itself. If you wanted to change the metadata, you'd have to re-create the object so that you'd have the requisite permissions.

DEFAULT OBJECT ACLS

In addition to granting permissions on both buckets and objects, Cloud Storage allows you to decide up front what ACLs should be set on newly created objects in the form of a bucket's default object ACLs. This process follows the same pattern as a single object ACL (you can have various Readers and Owners), but you define the ACL at the bucket level and then apply it to all objects when they're created. For example, if you define your default object ACL to have `allUsers` as a Reader, all objects that you upload will be publicly readable as you create them.

Note that default object ACLs are a template applied when you create an object. This doesn't modify existing objects in any way.

PREDEFINED ACLS

As you might expect, a few common scenarios entail quite a bit of clicking (or typing) to get configured. To make this easier, Cloud Storage has predefined ACLs that you can set using the `gsutil` command-line tool. When you want to do common things like make an object publicly readable or private to the project, you can do this with a few keystrokes. Upload a file to Cloud Storage and make it publicly readable, as shown in the following listing.

Listing 8.5 Set a predefined ACL

```
$ gsutil mb gs://my-public-bucket           ← Start by creating
Creating gs://my-public-bucket/...            a new bucket.

$ echo "This should be public" > public.txt
$ gsutil cp public.txt gs://my-public-bucket   ← Then create a new
Copying file://public.txt [Content-Type=text/plain]...
Uploading   gs://my-public-bucket/public.txt:          23 B/23 B
```

After that, you should look at the default ACL file. To get the ACL that GCS created by default, run `gsutil acl get gs://my-public-bucket/public.txt`. You should see something like the following.

Listing 8.6 Stored ACL

```
[  
  {  
    "entity": "project-owners-243576136738",  
    "projectTeam": {  
      "projectNumber": "243576136738",  
      "team": "owners"  
    },  
    "role": "OWNER"  
  },  
  {  
    "entity": "project-editors-243576136738",  
    "projectTeam": {  
      "projectNumber": "243576136738",  
      "team": "editors"  
    },  
    "role": "EDITOR"  
  },  
  {  
    "entity": "project-viewers-243576136738",  
    "projectTeam": {  
      "projectNumber": "243576136738",  
      "team": "viewers"  
    },  
    "role": "VIEWER"  
  }]
```

Notice how by default the ACL has owners, editors, and viewers preset.

```

"projectTeam": {
  "projectNumber": "243576136738",
  "team": "editors"
    },
  "role": "OWNER"
  },
  {
  "entity": "project-viewers-243576136738",
  "projectTeam": {
    "projectNumber": "243576136738",
    "team": "viewers"
      },
  "role": "READER"
  },
  {
  "entity": "user-
    00b4903a978dc75fbff509edb5b5658a3c6972b0ef52feca6618b156ced45d8",
  "entityId":
    "00b4903a978dc75fbff509edb5b5658a3c6972b0ef52feca6618b156ced45d8",
  "role": "OWNER"
  }
}

```

Notice how by default the ACL has owners, editors, and viewers preset.

Now access that file over the public internet (for example, not through gsutil) and then update the ACL to be public after that fails, as shown in the next listing.

Listing 8.7 Inspect and update the ACL

```

$ curl https://my-public-bucket.storage.googleapis.com/public.txt
<?xml version='1.0' encoding='UTF-8'?><Error>
  <Code>AccessDenied</Code><Message>Access denied.</Message>
  <Details>Anonymous users does not have storage.objects.get
  </Details></Error>

$ gsutil acl set public-read gs://my-public-bucket/public.txt
Setting ACL on gs://my-public-bucket/public.txt...

$ curl https://my-public-bucket.storage.googleapis.com/public.txt
This should be public!

```

As you can see in this example, if you look at the ACL that was created by default, it shows the project roles as well as the owner ID. When you try to access the object through curl, it's rejected with an XML Access Denied error as expected. Then you can set the predefined ACL (public-read) with a single command, and after that the object is visible to the world. This behavior isn't limited to public-read. Table 8.3 shows more of the predefined ACLs in order of the likelihood that you'll use them.

Table 8.3 Pre-defined ACL definitions

Name	Meaning
private	Removes any permissions besides the single owner (creator)
project-private	The default for new objects, which gives access based on roles in your project
public-read	Gives anyone (even anonymous users) reader access
public-read-write	Gives everyone (even anonymous users) reader and writer access
authenticated-read	Gives anyone logged in with their Google account reader access
bucket-owner-read	Used only for objects (not buckets); gives the creator owner access and the bucket owners read access
bucket-owner-full-control	Gives object and bucket owners the owner permission

It's important to point out that by using a predefined ACL, you are *replacing* the existing ACL. If you have a long list of users who have special access and you apply any of the predefined ACLs, you overwrite your existing list. Be careful when applying predefined ACLs, particularly if you've spent a long time curating ACLs in the past. You should also try to use Group and Domain entities often rather than specific User entities because group membership won't be lost by setting a predefined ACL.

ACL BEST PRACTICES

Now that you understand quite a bit about ACLs, it seems useful to spend a bit of time describing a few best practices of how to manage ACLs and choose the right permissions for your buckets and objects. Keep in mind this is a list of guidelines and not rules, so you should feel comfortable deviating from them if you have a good reason:

- *When in doubt, give the minimum access possible.* This is a general security guideline but definitely relevant to controlling access to your data on Cloud Storage. If someone needs permission only to read the data of an object, give them Reader permission only. If you give out more than this, don't be surprised when someone else borrowing their laptop accidentally removes a bunch of ACLs from the object.

In general, remember that you can always grant more access if someone should need it. You can't always undo things that a malicious or absent-minded user did.

- *The Owner permission is powerful, so be careful with it.* Owners can change ACLs and metadata, which means that unless you trust someone to grant further access appropriately, you shouldn't give them the Owner permission.

Following on the previous principle, when in doubt, give the Writer permission instead. Your data doesn't have an undo feature, so you should trust not only that any new Owners will do the right thing, but also that they're careful enough to make sure that no one else can do the wrong thing, either accidentally or purposefully.

- *Allowing access to the public is a big deal, so do it sparingly.* It's been said before that after something is on the internet, it's there forever. This is certainly true about your data after you expose it to the world. When using the allUsers or allAuthenticatedUsers (and therefore the public-read or authenticated-read) tokens, recognize that this is the same as publishing your content to the world. We'll also discuss a concern about this when we cover pricing later in this chapter.
- *Default ACLs happen automatically, so choose sensible defaults.* It's easy to miss when an overly open default ACL is set precisely because you don't notice until you look at the newly created object's ACL. It's also easy to break the rule about giving out the minimum access when you have a relatively loose ACL as your default. In general, it's best to use one of the more strict predefined ACLs as your object default, such as project-private or bucket-owner-full-control if you're on a small team and private or bucket-owner-read if you're on a larger team.

Now that you understand how to control access in the general sense, let's look at how to handle those one-off scenarios where you want to grant access to a single operation.

8.4.2 Signed URLs

It turns out that sometimes you don't want to add someone to the ACL forever, but rather want give someone access for a fixed amount of time. You're not so concerned about authenticating the user with their Google account, but you've authenticated them with your own login system and want to say "This person has access to view this data." Luckily, Cloud Storage provides a simple way to do so with signed URLs.

Signed URLs take an intent to do an operation (for example, download a file) and sign that intent with a credential that has access to do the operation. This allows someone with no access at all to present this one-time pass as their credential to do exactly what the pass says they can do. Let's run through a simple example, like creating a signed URL to download a text file from GCS. To start, you'll need a private key, so jump over to the IAM & Admin section, and select Service accounts from the left-side navigation, shown in figure 8.7.

Then create a new service account, making sure to have Google generate a new private key in JSON format. In this case, use the name gcs-signer as the name for this account, as shown in figure 8.8.

When you create the service account, notice that it added the account to the list but also started a download of a JSON file. Don't lose this file because it's the only

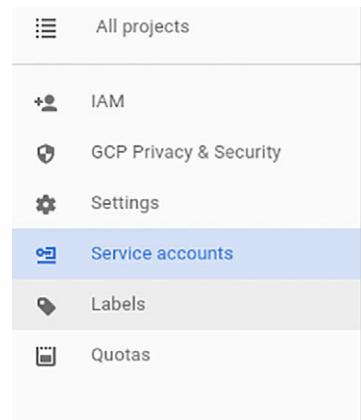


Figure 8.7 Choose Service accounts from the left-side navigation.

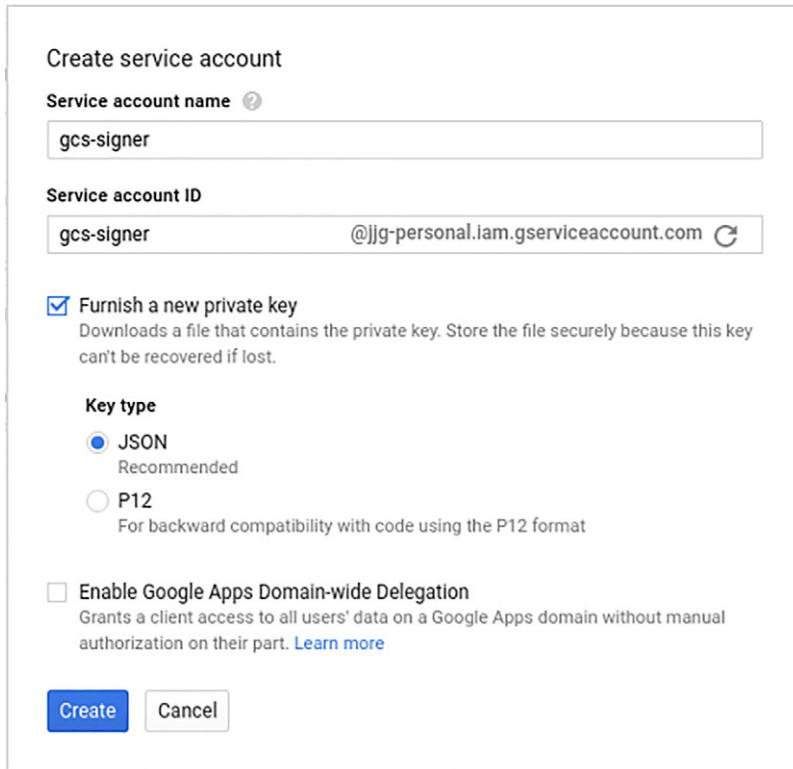


Figure 8.8 Create a new service account.

copy of the private key for your account (Google doesn't keep a copy of the private key for security reasons). Now quickly upload a file that you're sure is private, as shown in the next listing.

Listing 8.8 Uploading a file that is private by default

```
$ echo "This is private." > private.txt
$ gsutil cp private.txt gs://my-example-bucket/
Copying file://private.txt [Content-Type=text/plain]...
Uploading   gs://my-example-bucket/private.txt:                                17 B/17 B

$ curl https://my-example-bucket.storage.googleapis.com/private.txt
<?xml version='1.0' encoding='UTF-
8'?><Error><Code>AccessDenied</Code><Message>Access
denied.</Message><Details>Anonymous users does not have
storage.objects.get access to object my-example-
bucket/private.txt.</Details></Error>
```

Finally you should make sure the service account you created has access to the file. (Remember, the service account can sign only for things that it's able to do, so if it

doesn't have access to the file, its signature is worthless.) You can grant access to your new service account by using `gsutil acl ch` ("ch" standing for "change"), as the following listing shows.

Listing 8.9 Grant access to a service account

```
$ gsutil acl ch -u gcs-signer@your-project-
→ id.iam.gserviceaccount.com:R gs://my-example-bucket/private.txt
Updated ACL on gs://my-example-bucket/private.txt
```

Notice that the ACL you changed was of the form `-u service-account-email:R`. Service accounts are treated like users, so you use the `-u` flag, then you use the email address based on the name of the service account, and finally use `:R` to indicate that Reader privileges are added. Now that you have the right permissions, you have to provide the right parameters to `gsutil` to build a signed URL. See table 8.4.

Table 8.4 Parameters for signing a URL with `gsutil`

Parameter	Flag	Meaning	Example
Method	<code>-m</code>	The HTTP method for your request	GET
Duration	<code>-d</code>	How long until the signature expires	1h (one hour)
Content type	<code>-t</code>	The content type of the data involved (used only when uploading)	image/png

In this example you want to download (GET) a file called `private.txt`. Let's assume that the signature should expire in 30 minutes (`30m`). This means the parameters to `gsutil` would be as shown in the next listing.

Listing 8.10 `gsutil` command to sign a URL

```
$ gsutil signurl -m GET -d 30m key.json gs://my-example-bucket/private.txt
URL      HTTP Method      Expiration      Signed URL
gs://my-example-bucket/private.txt      GET      2016-06-21 07:07:35
https://storage.googleapis.com/my-example-
bucket/private.txt?GoogleAccessId=gcs-signer@your-project-
id.iam.gserviceaccount.com&Expires=1466507255&Signature=
→ ZBufnbBAQOz1oS8ethq%2B519C7YmVHVbNM%2F%2B43z9XDcsTgpWoc
→ bAMmJ2ZhugI%2FZWE665mxD%2BJL%2BJzVSy7BAD7qFWTok0vDn5a0
→ sq%2Be78nCuJmgE01DTERQpnXSvbcohtOyV1Fr8p3StKU0ST1wKoN1ceh
→ fRXWD45fEMMFmchPhkI8M8ASwai%2FVNZOXP5HxtZvZaco47NTClB5k9
→ uKBL1MEg65RAbBTt5huHRGO6XkYgnYKDY87rs18HSEL4dMauUZpaYC4Z
→ Pb%2FSBpWAMOneaXpTHlh4cKXXN1rQ03MUf5w3sKKJBsUWB10xoAsf3H
→ pdnnrFjW5sUZUQu1RRTqHyztc4Q%3D%3D
```

It's a bit tough to read but the last piece of output is a URL that will allow you to read the file `private.txt` from any computer for the next 30 minutes. After that, it expires, and you'll go back to getting the Access Denied errors we saw before. To test this, you can try getting the file with and without the signed piece, as shown in the next listing.

Listing 8.11 Retrieving our file

```
$ curl -S https://storage.googleapis.com/my-example-bucket/private.txt
<?xml version='1.0' encoding='UTF-
8'?><Error><Code>AccessDenied</Code><Message>Access
denied.</Message><Details>Anonymous users does not have storage.objects.get
access to object my-example-bucket/private.txt.</Details></Error>

$ curl -S "https://storage.googleapis.com/my-example-
bucket/private.txt?GoogleAccessId=gcs-signer@your-project-
id.iam.gserviceaccount.com&Expires=1466507255&Signature=
↳ ZBufrnbBAQOz1oS8ethq%2B519C7YmVHVbNM%2F%2B43z9XDcsTgpWoC
↳ bAMmJ2ZhugI%2FZWE665mxD%2BJL%2BJzVSy7BAD7qFWTok0vDn5a0
↳ sq%2Be78nCJmgE01DTERQpnXSvbc0htOyVlFr8p3StKU0ST1wKoN1ceh
↳ fRXWD45fEMMFmchPhkI8M8ASwai%2FVNZOxp5HxtZvZacO47NTClB5k9
↳ uKBL1MEg65RABBTt5huHRGO6XkYgnyKDY87rs18HSEL4dMauUZpaYC4Z
↳ Pb%2FSBpWAMOneaXpTHlh4cKXXNlrQ03MUf5w3sKKJBsUWB10xoAsf3Hp
↳ dnmrFjW5sUZUQu1RRTqHyztc4Q%3D%3D"
This is private.
```

Note that you added quotes around the URL (because there are extra parameters that would be interpreted by the command line).

You might be thinking that this is great when you happen to be sitting at your computer, but isn't the more common scenario where you have content in your app that you want to share temporarily with users? For example, you might want to serve photos, but you don't want them always available to the public to discourage things like hotlinking. Luckily this is easy to do in code, so let's look at a short example snippet in Node.js.

The basic premise is the same, but you'll do it in JavaScript rather than on the command line with gsutil, as the next listing shows.

Listing 8.12 Sign a URL to grant specific access

```
const storage = require('@google-cloud/storage')({
  projectId: 'your-project-id',
  keyFilename: 'key.json'
});
const bucket = storage.bucket('my-example-bucket');
const file = bucket.file('private.txt');

file.getSignedUrl({
  action: 'read', // This is equivalent to HTTP GET.
  expires: new Date().valueOf() + 30*60000, // This says "30 minutes from now"
}, (err, url) => {
  console.log('Got a signed URL:', url);
});
```

When you run this you should see something like this.

```
Got a signed URL: https://storage.googleapis.com/my-example-
bucket/private.txt?GoogleAccessId=gcs-signer@your-project-
```

```

id.iam.gserviceaccount.com&Expires=1466508154&Signature=LW0AqC4
↳ E31I7c1JgMhuljeJ8WC01qnazEeqE%2B2ikSPmzThauAqht5fxo2WYfL%2F
↳ 5MnbBF%2BFUDj1gsESjwb2Ar%2F5EoRDFY2O9GRE50IuOhAoWK3kbiQ4sIUR
↳ xmSF%2BZymU1Nou1BEEPxahgeQNIcy1snkjF7pqpEU9fKjTcwzKfTBcYx7n
↳ 3irIW27IYJx4JQ8146bFFweiHei%2B7fVzK081fP5XY%2BM2kCovfeWSb8K
↳ cLPZ8501tW9g8Xmo%2Fvf3rZpwF27rgV4UPDwz247Fn7UAm17T%2B%2FmEe
↳ ANY1RoQtb8I1hnHl1Ota36iWKOV17GQ%2FYh7F2JsDhJxZTwXkIR51zSR8n
↳ D2Q%3D%3D

```

If you wanted to render this new value as the image `src` attribute, you could do that instead of using a `console.log` statement.

Now that you understand how to change the access restrictions on data, let's also look at how to track who's accessing your data.

8.4.3 Logging access to your data

If you're managing sensitive data (for example, you're storing employee records), you probably want to track when this data is accessed. Cloud Storage makes this simple by allowing you to set that a specific bucket should have its access logged. Use the Cloud Storage API to specify a logging configuration that says where the logs should end up (the `logBucket`) and whether Cloud Storage should prefix the beginning of the log files (the `logObjectPrefix`).

You're going to interact with your logging configuration using the `gsutil` command-line tool, as shown in the next listing.

Listing 8.13 Interacting with the logging configuration using gsutil

After that, grant access to the “logger” account (`cloud-storage-analytics@google.com`) that will be responsible for putting the logs into that bucket.

```
$ gsutil logging get gs://my-example-bucket
gs://my-example-bucket/ has no logging configuration.
```

Start by checking the logging configuration for a bucket (`my-example-bucket`) and then configure logging on it.

```
$ gsutil mb -l US -c multiRegional gs://my-example-bucket-logs
Creating gs://my-example-bucket-logs/...
```

To do this, create a bucket that will hold all of the logs (`my-example-bucket-logs`).

```
↳ $ gsutil acl ch -g cloud-storage-analytics@google.com:W \
gs://my-example-bucket-logs
```

```
$ gsutil logging set on -b gs://my-example-bucket-logs \
-o example-prefix gs://my-example-bucket
Enabling logging on gs://my-example-bucket/...
```

Finally, configure the logging details, telling Cloud Storage to place all access logs into the newly created bucket.

```
$ gsutil logging get gs://my-example-bucket
{
  "logBucket": "my-example-bucket-logs",
  "logObjectPrefix": "example-prefix"
}
```

To check that it worked, you can use the `gsutil` `logging get` command to show the configuration you saved and make sure it's all accurate.

After you have your configuration set, Cloud Storage stores all access logs in the logging bucket every hour that activity occurs. The log files themselves will be named based on your prefix, a timestamp of the hour being reported, and a unique ID (for example, 1702e6). For example, a file from your logging configuration might look like `example-prefix_storage_2016_06_18_07_00_00_1702e6_v0`. Inside each of the log files are lines of comma-separated fields (you've probably seen .csv files before), with the schema shown in table 8.5.

Table 8.5 Schema of access log files

Field (type)	Description
<code>time_micros</code> (int)	The time that the request was completed, in microseconds since the Unix epoch.
<code>c_ip</code> (string)	The IP address from which the request was made.
<code>c_ip_type</code> (integer)	The type of IP in the <code>c_ip</code> field (1 for IPv4, and 2 for IPv6).
<code>c_ip_region</code> (string)	Reserved for future use.
<code>cs_method</code> (string)	The HTTP method of this request.
<code>cs_uri</code> (string)	The URI of the request.
<code>sc_status</code> (integer)	The HTTP status code the server sent in response.
<code>cs_bytes</code> (integer)	The number of bytes sent in the request.
<code>sc_bytes</code> (integer)	The number of bytes sent in the response.
<code>time_taken_micros</code> (integer)	The time it took to serve the request in microseconds.
<code>cs_host</code> (string)	The host in the original request.
<code>cs_referer</code> (string)	The HTTP referrer for the request.
<code>cs_user_agent</code> (string)	The User Agent of the request; for requests made by lifecycle management, the value is GCS Lifecycle Management.
<code>s_request_id</code> (string)	The request identifier.
<code>cs_operation</code> (string)	The Google Cloud Storage operation.
<code>cs_bucket</code> (string)	The bucket specified in the request; if this is a list buckets request, this can be null.
<code>cs_object</code> (string)	The object specified in this request; this can be null.

Note that each of the fields in the access log entry is prefixed by a `c`, `s`, `cs`, or `sc`. These prefixes are explained in table 8.6.

Although uncommon, log entries could have duplicates, so you should use the `s_request_id` field as a unique identifier if you ever need to be completely confident that an entry is not a duplicate.

Table 8.6 Access log field prefix explanation

Prefix	Stands for...?	Meaning
c	client	Information about the client making a request
s	server	Information about the server receiving the request
cs	client to server	Information sent from the client to the server
sc	server to client	Information sent from the server to the client

Now that you have a grasp of access control, let's move on to a slightly more advanced topic: versioning.

8.5 Object versions

Similar to version control (like Git, Subversion, or Mercurial), Cloud Storage has the ability to turn on versioning, where you can have objects with multiple revisions over time. Also, when versioning is enabled, you can revert back to an older version like you can with files in a Git repository.

The biggest change when object versioning is enabled is that overwriting data doesn't truly overwrite the original data. Instead, the previous version of the object is archived and the new version marked as the active version. If you upload a 10 MB file called `data.csv` into a bucket with versioning enabled and then re-upload the revised 11 MB file of the same name, you'll end up with the original 10 MB file archived in addition to the new file, so you're storing a total of 21 MB (not 11 MB).

In addition to versions of objects, Cloud Storage also supports different versions of the metadata on the objects. In the same way that an object could be archived and a new generation added in its place, when metadata (such as ACLs) is changed on a versioned object, the metadata gets a new metageneration to track its changes. In any version-enabled bucket, every object will have a generation (tracking the object version) along with a metageneration (tracking the metadata version). As you might imagine, this feature is useful when you have object data (or metadata) that changes over time, but you still want to have easy access to the latest version. Let's explore how to set this up and then demonstrate how you can do some of these common tasks that I mentioned.

As you learned, object versioning is a feature that's enabled on a bucket, so the first thing you need to do is enable the feature, as the next listing shows.

Listing 8.14 Enable object versioning

```
$ gsutil versioning set on gs://my-versioned-bucket
Enabling versioning for gs://my-versioned-bucket/...
```

Now check that versioning is enabled and then upload a new file, as shown in the following listing.

Listing 8.15 Check versioning is enabled and upload a text file

```
$ gsutil versioning get gs://my-versioned-bucket
gs://my-versioned-bucket: Enabled

$ echo "This is the first version!"> file.txt
$ gsutil cp file.txt gs://my-versioned-bucket/
Copying file://file.txt [Content-Type=text/plain]...
Uploading  gs://my-versioned-bucket/file.txt:
➥          27 B/27 B
```

Now look more closely at the file by using the `ls -la` command, as shown in the listing 8.16. The `-l` flag shows the “long” listing, which includes some extra information about the file, and the `-a` flag shows noncurrent (for example, archived) objects along with extra metadata about the object such as the generation and metageneration.

Listing 8.16 Listing objects with -la flags

```
$ gsutil ls -la gs://my-versioned-bucket
27 2016-06-21T13:29:38Z gs://my-versioned-
➥ bucket/file.txt#1466515778205000 metageneration=1
TOTAL: 1 objects, 27 bytes (27 B)
```

As you can see, the metageneration (or the version of the metadata) is obvious (`metageneration=1`). The generation (or version) of the object isn’t as obvious, but it’s that long number after the # in the filename; in this example, 1466515778205000. As you learned earlier, when versioning is enabled on a bucket, new files of the same name archive the old version before replacing the file, so try that and then look again at what ends up in the bucket, as shown in the following listing.

Listing 8.17 Upload a new version of the file

```
$ echo "This is the second version."> file.txt
$ gsutil cp file.txt gs://my-versioned-bucket/
Copying file://file.txt [Content-Type=text/plain]...
Uploading  gs://my-versioned-bucket/file.txt:
➥          28 B/28 B

$ gsutil ls -l gs://my-versioned-bucket
28 2016-06-21T13:39:11Z gs://my-versioned-bucket/file.txt
TOTAL: 1 objects, 28 bytes (28 B)

$ gsutil ls -la gs://my-versioned-bucket
27 2016-06-21T13:29:38Z gs://my-versioned-
➥ bucket/file.txt#1466515778205000 metageneration=1
28 2016-06-21T13:39:11Z gs://my-versioned-
➥ bucket/file.txt#1466516351939000 metageneration=1
TOTAL: 2 objects, 55 bytes (55 B)
```

Notice how when listing objects without the `-a` flag you see only the latest generation, but when listing with it, you can see all generations. The total data stored in the first

operation appears to be 28 bytes; however, when listing everything (with the `-a`) flag, the total data stored is 55 bytes. Finally, when you look at the latest version, it should appear to be the more recent file you uploaded:

```
$ gsutil cat gs://my-versioned-bucket/file.txt
This is the second version.
```

If you want to look at the previous version, you can refer to the specific generation you want to see. Try looking at the previous version of your file:

```
$ gsutil cat gs://my-versioned-bucket/file.txt#1466515778205000
This is the first version!
```

As you can see, versioned objects are like any other object, but have a special tag on the end referring to the exact generation. You can treat them as hidden objects, but they're still objects, so you can delete prior versions:

```
$ gsutil rm gs://my-versioned-bucket/file.txt#1466515778205000
Removing gs://my-versioned-bucket/file.txt#1466515778205000...
$ gsutil ls -la gs://my-versioned-bucket
28 2016-06-21T13:39:11Z gs://my-versioned-bucket/file.txt#1466516351939000
    metageneration=1
TOTAL: 1 objects, 28 bytes (28 B)
```

You'll see some surprising behavior when deleting objects from versioned buckets because deleting the file itself doesn't delete other generations. For example, if you were to delete your file (`file.txt`), "getting" the file would return a 404 error. The exact generation of the file would still exist, however, and you could read that file by its specific version. Let's demonstrate this by continuing our example:

```
$ gsutil ls -la gs://my-versioned-bucket/
28 2016-06-21T13:54:26Z gs://my-versioned-
➥ bucket/file.txt#1466517266796000 metageneration=1
TOTAL: 1 objects, 28 bytes (28 B)

$ gsutil rm gs://my-versioned-bucket/file.txt
Removing gs://my-versioned-bucket/file.txt...
```

At this point, you've deleted the latest version of the file so you expect it to be gone. Look at the different views to see what happened:

```
$ gsutil ls -l gs://my-versioned-bucket/
$ gsutil ls -la gs://my-versioned-bucket/
28 2016-06-21T13:54:26Z gs://my-versioned-
➥ bucket/file.txt#1466517266796000 metageneration=1
TOTAL: 1 objects, 28 bytes (28 B)

$ gsutil cat gs://my-versioned-bucket/file.txt
CommandException: No URLs matched: gs://my-versioned-bucket/file.txt

$ gsutil cat gs://my-versioned-bucket/file.txt#1466517266796000
This is the second version.
```

Notice that whereas the file appears to be gone, a prior version still exists and is readable if referred to by its exact generation ID! This capability allows you to restore your previous versions if needed, which you can do by copying the previous generation into place. Look at how to restore the second version of our file:

```
$ gsutil cp gs://my-versioned-bucket/file.txt#1466517266796000 gs://my-
    versioned-bucket/file.txt
Copying gs://my-versioned-bucket/file.txt#1466517266796000
↳ [Content-Type=text/plain]...
Copying      gs://my-versioned-bucket/file.txt:
↳          28 B/28 B

$ gsutil cat gs://my-versioned-bucket/file.txt
This is the second version.
```

You might think you brought the old version back to life, but look at the directory listing to see if that's true:

```
$ gsutil ls -la gs://my-versioned-bucket
28  2016-06-21T13:54:26Z  gs://my-versioned-
↳  bucket/file.txt#1466517266796000  metageneration=1
28  2016-06-21T13:59:39Z  gs://my-versioned-
↳  bucket/file.txt#1466517579727000  metageneration=1
TOTAL: 2 objects, 56 bytes (56 B)
```

You created a new version by restoring the old one, so technically you now have two files with the same content in your bucket. If you want to remove the file along with all of its previous versions, pass the `-a` flag to the `gsutil rm` command:

```
$ gsutil rm -a gs://my-versioned-bucket/file.txt
Removing gs://my-versioned-bucket/file.txt#1466517266796000...
Removing gs://my-versioned-bucket/file.txt#1466517579727000...
$ gsutil ls -la gs://my-versioned-bucket/
```

As you can see, by using the `-a` flag you can get rid of all the previous versions of an object in one swoop.

Specific object generations can be treated as individual objects in the sense that you can operate on them like any other object. They have special features in that they're automatically archived when you overwrite (or delete) the object, but as far as usage goes, archived versions shouldn't scare you any more than hidden files on your computer (and coincidentally, you use the same commands to view those files on most systems).

You might be wondering how to keep your bucket from growing out of control. For example, it's easy to decide you're done with a file (and all of its versions), but how do you decide when you're done with a version? How old is too old? And isn't it obnoxious to have to continuously clean old versions of objects in your bucket? Let's look at how to deal with this problem next.

8.6 Object lifecycles

As you add more objects to your buckets, it's easy for you to accumulate a bunch of less-than-useful data in the form of old or out-of-date objects. This problem can be compounded when you have versioning enabled on your bucket because old versions will build up based on changes and they won't be as noticeable if you happen to be browsing your buckets for files that can be deleted.

To deal with this accumulation problem, Cloud Storage allows you to define a way for you to conditionally delete data automatically so that you don't have to remember to clean your bucket every so often. You'll hear this concept referred to elsewhere as *lifecycle management* because it's a definition of when an object is at the end of its life and should be deleted.

You can define a couple of conditions to determine when objects should be automatically deleted in your bucket:

- *Per-object age* (Age)—This is equivalent to fixing a number of days to live (sometimes referred to as a TTL). When you have an age condition, you're effectively saying delete this object N days after its creation date.
- *Fixed date cut-off* (CreatedBefore)—When setting a lifecycle configuration, you can specify that any objects with a creation date before the configured one be deleted. This setting is an easy way to throw away any created before a fixed date.
- *Version history* (NumberOfNewVersions)—If you have versioning enabled on your bucket, this condition allows you to delete any objects that are the N th oldest (or older) version of a given object. This is like saying, "I only need the last five revisions, so remove anything older than that." Note that this is related not to timing but to the volatility (number of changes) to the object.
- *Latest version* (IsLive)—This setting allows you to delete only the archived (or nonarchived) versions, effectively allowing you to discard all version history if you want to make a fresh start.

To apply a configuration, you have to assemble these conditions into a JSON file as a collection of rules. Then you apply the configuration to the bucket. Inside each rule, all of the conditions are AND-ed together, and if they all match, then the object is deleted.

Let's look at an example lifecycle configuration where you want to delete any object older than 30 days, shown in the next listing.

Listing 8.18 Delete objects older than 30 days

```
{  
  "rule": [  
    {  
      "action": {"type": "Delete"},  
      "condition": {"age": 30}  
    }  
  ]  
}
```

Imagine you like that rule but also want to delete objects older than 30 days, as well as any objects that have more than three newer versions. To do this, you use two different rules, which are applied separately.

Listing 8.19 Delete things older than 30 days or with at least three newer versions.

```
{  
  "rule": [  
    {  
      "action": {"type": "Delete"},  
      "condition": {"age": 30}  
    },  
    {  
      "action": {"type": "Delete"},  
      "condition": {  
        "isLive": false,  
        "numNewerVersions": 3  
      }  
    }  
  ]  
}
```

Note that inside a single rule, the conditions are AND-ed in the sense that both of the conditions must be met. Each individual rule is applied separately, however, which effectively means the rules are OR-ed in the sense that if any rule matches the file will be deleted.

Now that you understand the format of a lifecycle configuration policy, you can set these rules on your buckets. For the purpose of demonstration, let's choose a policy that's easy to test, such as deleting anything that has at least one newer version, as shown in the following listing.

Listing 8.20 Delete anything with at least one newer version

```
{  
  "rule": [  
    {  
      "action": {"type": "Delete"},  
      "condition": {  
        "isLive": false,  
        "numNewerVersions": 1  
      }  
    }  
  ]  
}
```

Start by saving this demonstration policy to a file called `lifecycle.json`. Then apply this policy to your versioned bucket from earlier, as the following listing shows.

Listing 8.21 Interacting with the lifecycle configuration using gsutil

```
$ gsutil lifecycle get gs://my-versioned-bucket
gs://my-versioned-bucket/ has no lifecycle configuration.

$ gsutil lifecycle set lifecycle.json gs://my-versioned-bucket
Setting lifecycle configuration on gs://my-versioned-bucket/...

$ gsutil lifecycle get gs://my-versioned-bucket
{"rule": [{"action": {"type": "Delete"}, "condition": {"isLive": false, "numNewerVersions": 1}}]}
```

If you upload some files, you might notice that they aren't immediately deleted according to the configuration you defined. This might seem strange, but keep in mind that the clean-up happens on a regular interval, not immediately.

Although the object might not be deleted immediately, you aren't billed for storing objects that satisfy the lifecycle configuration but haven't been deleted yet. If you access a file that isn't yet deleted, you will be billed for those operations and bandwidth. After an object should be deleted, you're no longer charged for storage, but you're charged for any other operations.

Now that you understand how to keep your data tidy, let's look at how you might connect Cloud Storage to your app in an event-driven way.

8.7 Change notifications

So far all of the interaction with Cloud Storage has been “pull”—the interaction was initiated by you contacting Cloud Storage, either uploading or downloading data. Wouldn't it be nice if we could use some of these features like access control policies and signed URLs to allow users to upload or update files and have Cloud Storage notify you when things happen? This is possible by setting up change notifications.

If you couldn't guess from the name, change notifications allow you to set a URL that will receive a notification whenever objects are created, updated, or deleted. Then you can do whatever other processing you might need based on the notification.

A common scenario is to have a bucket acting like an inbox that accepts new files and then processes those files into a known location. For example, you might have a bucket called `incoming-photos`, and whenever an image is uploaded, you process the image into a bunch of different sizes as thumbnails and store those for use later on. This method lends itself nicely to using signed URLs for allowing one-time passes to upload files into the incoming bucket.

This process, shown in figure 8.9, works by setting up a notification channel that acts as the conduit between an event happening in your bucket and a notification being sent to your servers:

- 1 A user sends a request to your web server for a signed URL effectively asking, “Can I upload a file?”
- 2 The server responds with a signed URL granting the user access to put a file into the bucket.

- 3 The user then uploads their image into the bucket
- 4 When that file is saved, a notification channel sends a request to let the server know that a new file has arrived.

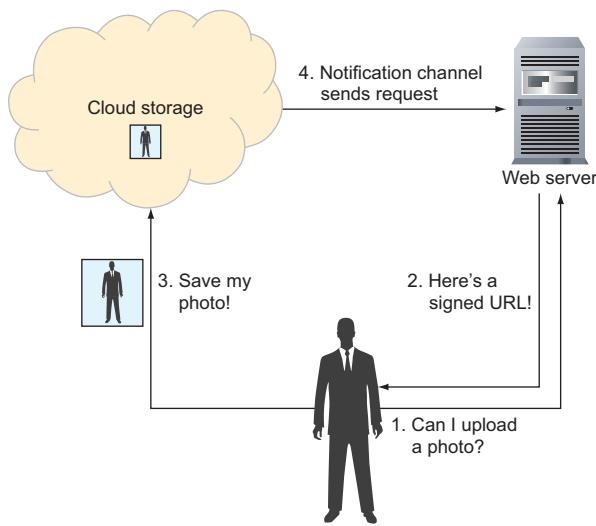


Figure 8.9 Common object notification flow

Setting up a notification channel is easy to do with the `gsutil` command-line tool. Use the `watchbucket` subcommand, and provide the following three pieces of information:

- 1 The URL that should be notified
- 2 The bucket that you want to watch
- 3 The ID for the channel that you're creating, which should be unique for the bucket

Based on those things, setting up a watch command should look like this.

```
$ gsutil notification watchbucket -i channel-id
  ➔ https://mydomain.com/new-image gs://my-bucket
```

In this example, the channel ID is `channel-id`, and the bucket is `my-bucket`, which effectively says to send a request to the specified URL (<https://mydomain.com/new-image>) whenever any changes happen inside `my-bucket`. You can also specify a channel token to act as a unique password of sorts so you can be sure that any requests sent are from Google and not from somewhere else.

After you set up a channel, you'll start to receive POST requests from Cloud Storage for the various events that occur in your bucket. These requests have a variety of parameters that arrive in the form of HTTP headers. See table 8.7.

Table 8.7 Parameters in a notification request

Header name	Meaning (example)
X-Goog-Channel-Id	The channel ID of the notification (for example, <code>channel-id</code>)
X-Goog-Channel-Token	The token of the notification (for example, <code>my-secret-channel-token</code>)
X-Goog-Resource-Id	The ID of the resource being modified (for example, <code>my-bucket/file.txt</code>)
X-Goog-Resource-State	The event prompting this notification (for example, <code>sync</code> , <code>exists</code> , <code>not_exists</code>)
X-Goog-Resource-Url	The URL corresponding to the resource ID (for example, https://www.googleapis.com/storage/v1/b/BucketName/o/file.txt)

Corresponding to the X-Goog-Resource-State header, each state corresponds to a different event, effectively saying what happened to trigger the event. Only three distinct states exist, corresponding to four different events:

- 1 *Sync* (`sync`)—The first event you'll receive; a sync event happens when you create the notification channel. This event lets you know that the channel is open, so you can use it to initialize anything on the server side.
- 2 *Object deletions* (`not_exists`)—Whenever an object is deleted, you'll get a request with a state saying `not_exists`. It's less likely that you'll need this event, but it's available nonetheless.
- 3 *Object creations and updates* (`exists`)—When an object is either created or updated, you'll get an event with the resource state set to `exists`. Along with the headers you get in every request, you'll also get the object metadata in the body of the request.

8.7.1 URL restrictions

Unfortunately, when you try to run the command to watch a bucket with a custom URL, you'll find out that there are a few gotchas about which URLs are allowed. Let's look briefly at why this is and how you can go about resolving any issues.

SECURITY

First, notice that in the example the URL starts with `https` and not `http`. Google wants to make sure that no one can spy on changes happening in buckets, so the notification URL must be at an encrypted endpoint. No matter what URL you put in there, if it starts with `http`, it will be rejected as invalid.

Though this will certainly be frustrating when you're testing, thanks to the wonderful people over at Let's Encrypt, setting up SSL certificates that work is surprisingly easy. Take a look at <https://letsencrypt.org/getting-started> for a summary of how to get SSL set up for your system—this should take only a few minutes.

WHITELISTED DOMAINS

In addition to requiring that your notification endpoint is secure, you also need to prove that you own a domain before using it as an endpoint for object change notifications. This prevents you from using Google Cloud to make requests of servers you don't own (either intentionally or accidentally). For example, what would stop you from setting up loads of endpoints all pointing at <https://your-competitor.com/dos-attack>?

Regardless of your intentions, you'll need to prove that you're authorized to send traffic to the domain before Cloud Storage will start sending notifications there, which means you have to whitelist it. You can whitelist a domain in a few ways, but the easiest is to use Google Domains for managing your domain name. You can do this by registering or transferring a domain into <https://domains.google.com>.

If that isn't an option (and for many, it won't be), you can also prove ownership through Google Webmaster Central by setting a DNS record or special HTML metatag. To get started with this, visit <https://google.com/webmasters/tools/>, which will guide you through the process. After your Google account is registered as an owner of the domain name, Cloud Storage considers your domain to be whitelisted, and your notification URL can use the domain name in question.

8.8 Common use cases

Now that you understand the building blocks common to object storage, let's explore some of the common use cases, specifically how you can put these building blocks together to do real-life things such as hosting profile pictures, websites, or archiving your data in case of a disaster.

8.8.1 Hosting user content

One of the most common scenarios is safely storing user content, such as profile photos, uploaded videos, or voice recordings. Using the concept of signed URLs, described in section 8.5.2, you can set up a simple system for processing user-uploaded content, such as the photos stored in InstaSnap.

As you learned in the section about signing URLs, though you want to accept user-created content, you don't want to give anyone in the world general access to your Cloud Storage buckets because that could lead to some scary things (for example, people deleting data or looking at data they shouldn't). It's wasteful for users to first send their content to your server and ask your server to forward it along to Cloud Storage. Ideally, you'd allow customers to send their content directly into your bucket—with a few limitations.

To accomplish this, Cloud Storage provides a way to create policy tokens, which are kind of like permission slips that children get in school to attend outside functions. You generate a policy document saying what a user can upload and then digitally sign that policy and send the signature back to the user. For example, a policy might convey something like "this person can upload up a png image up to 5 MB in size."

Then the user uploads their content to Cloud Storage and also passes along the policy and the signature of the policy. Cloud Storage checks that the signature is valid and that the operation the user is trying to do is covered by the policy. If it is, the operation completes. Figure 8.10 shows this from a flow-diagram perspective.

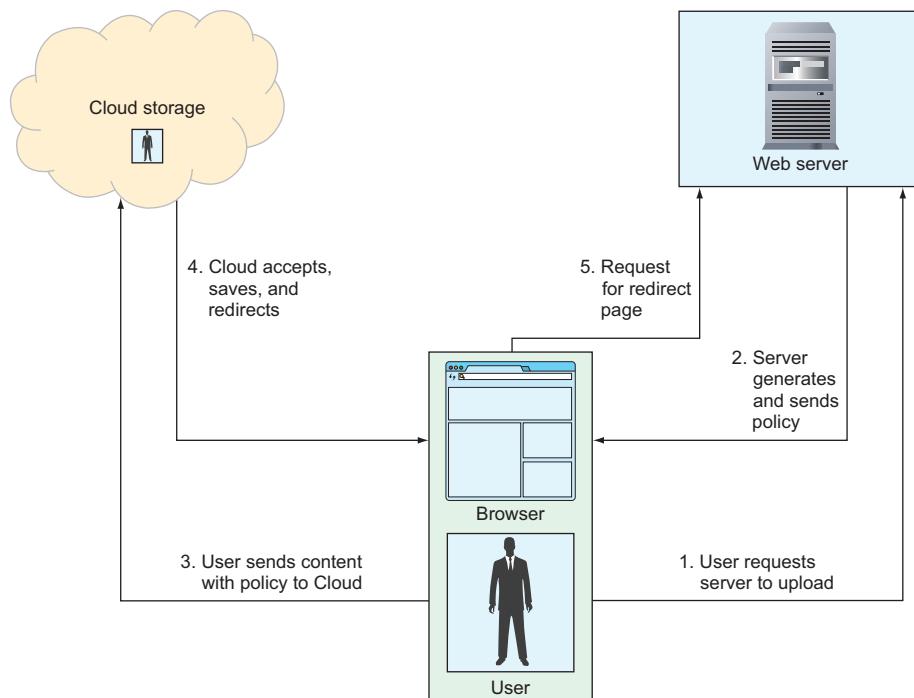


Figure 8.10 Uploading content using a policy signature

The steps are

- 1 The user makes a request to your web server, asking “Can I upload?” (This could be when a user navigates to an upload page.)
- 2 The server generates a policy and sends it back (along with the signature).
- 3 The user sends the content (for example, an image) along with the policy and signature to Cloud Storage using a standard HTML `<form>`.
- 4 Cloud Storage accepts and saves the image and then redirects the user to another web page.

8.8.2 Data archival

As you’ve heard a few times, specifically when we were discussing Nearline and Coldline storage, Cloud Storage can be a cost-effective way to archive your data. Whether

access logs, processed data, or old movies you've converted from DVDs, Cloud Storage cares only about making sure your data stays safe.

Given that archived data is much less frequently accessed, the Nearline and Coldline storage classes are ideal options. You won't often need to download this data, and, therefore, your bill at the end of the month will be much lower than if you'd chosen multiregional storage. Let's look briefly at how you might use Cloud Storage to archive your logs.

Logs are usually text files that a running process (such as a web server) appends to over time and cycles to a new filename every so often (sometimes based on the size of the file, sometimes based on timestamps). With Cloud Storage, your goal is to get those files off of your machine's storage and into a Cloud Storage bucket. Typically, your logging system packages your logs into a gzipped format when it makes the cut, so all you need to do is set up a schedule task to upload the right files to your bucket.

For example, you can use the `gsutil` command's `rsync` functionality as part of your systems crontab to synchronize your MySQL logs to Cloud Storage every day at 3 a.m.¹:

```
0 3 * * * gsutil -m rsync /var/log/mysql gs://my-log-archive/mysql
```

This command will synchronize your local log files into a Google Cloud Storage bucket, which avoids uploading data that you've already saved and copies any newly created (or modified) files all in a single command. Now let's move on to see how pricing works for Cloud Storage.

8.9 *Understanding pricing*

We've spent a lot of time discussing what Cloud Storage is, the features it comes with, and how you put those features together to do real things. But how do you pay for it? And how much does it cost? Let's spend some time walking through the different ways things cost money, and then we'll take a few common examples and look at how much each of these costs.

Cloud Storage pricing is broken into several different components:

- Amount of data stored
- Amount of data transferred (also known as network traffic)
- Number of operations executed (for example, number of GET operations)

In addition, the Nearline and Coldline storage classes have two extra components that we'll discuss in more detail later:

- Amount of data retrieved (in addition to served)
- 30-day (or 90-day) minimum storage

¹ We're ignoring time zones for the purposes of this conversation.

8.9.1 Amount of data stored

Data storage is the simplest and most obvious component of your Cloud Storage bill and should remind you of other storage providers like Drop Box. Every month, Cloud Storage charges you based on the amount of data you keep in your bucket measured in gigabytes per month, prorated on how long the object was stored. If you store an object for 15 out of 30 days, your bill for a single 2 GB object will be $2 \text{ (GB)} * 0.026 \text{ (USD)} * 15/30 \text{ (months)}$, which is 31 cents. And if you store it for only 1 hour (1/24th of one day) out of a 31-day month, your data storage cost will be $2 \text{ (GB)} * 0.026 \text{ (USD)} * (1/24) \text{ days / 31 (days in the month)}$, which is effectively zero (\$0.000069892). The data storage component gets even cheaper if you change to different storage classes such as Nearline or Coldline.

First, let's look at prices for the multiregion locations, which currently are the United States, the EU, and Asia. These three locations allow multiregion, Nearline, and Coldline storage classes split across multiple regions inside the location. See table 8.8.

Table 8.8 Pricing by storage class in multiregion locations per GB stored

Class	Price per GB per month
Multi-regional	2.6 cents (\$0.026)
Nearline	1 cent (\$0.01)
Coldline	0.7 cents (\$0.007)

For single-region locations, only regional, Nearline, and Coldline storage classes are supported. As you might guess, the prices for these vary from one location to the next, as shown in table 8.9 for a few common locations.

Table 8.9 Pricing by storage class (and location) per GB stored

Location	Regional	Nearline	Coldline
Oregon (US)	\$0.02	\$0.01	\$0.007
South Carolina (US)	\$0.02	\$0.01	\$0.007
London (UK)	\$0.023	\$0.016	\$0.013
Mumbai (India)	\$0.023	\$0.016	\$0.013
Singapore	\$0.02	\$0.01	\$0.007
Sydney (Australia)	\$0.023	\$0.016	\$0.013
Taiwan	\$0.02	\$0.01	\$0.007

These costs are strictly for the amount of data that you store in Cloud Storage. Any redundancy offered to provide high levels of durability is included in the regular price.

This storage cost might not seem like much, but when you look at the cost for larger and larger amounts of data, the cost differences can start to be material. Let's look at table 8.10, which shows a quick summary of storing increasing amounts of data for one month in the different storage classes.

Table 8.10 Monthly storage cost for different classes

Class	10 GB	100 GB	1 TB	10 TB	100 TB	1 PB
Multiregional	\$0.26	\$2.60	\$26.00	\$260.00	\$2,600.00	\$26,000.00
Regional (Iowa)	\$0.20	\$2.00	\$20.00	\$200.00	\$2,000.00	\$20,000.00
Nearline	\$0.10	\$1.00	\$10.00	\$100.00	\$1,000.00	\$10,000.00
Coldline	\$0.07	\$0.70	\$7.00	\$70.00	\$700.00	\$7,000.00

Notice that if you're storing large amounts of data (for example, a petabyte), using a different storage class such as Nearline can be significantly cheaper than multiregional for the data storage component of your bill.

METADATA IS DATA TOO! In addition to storing your data, any metadata you store on your object will be counted as though it were part of the object itself.

This means that if you store an extra 64 characters in metadata, you should expect an extra 64 bytes of storage to appear on your bill.

But your data doesn't sit still—it needs to be sent around the internet, so let's look at how much that costs.

8.9.2 Amount of data transferred

In addition to paying for data storage, you'll also be charged for sending that data to customers or to yourself. This cost is sometimes called network egress, which refers to the amount of data being sent out of Google's network. For example, if you download a 1 MB file from your Cloud Storage bucket onto your office desktop, you'll be charged for egress network traffic at Google's normal rates.

Because networking is dependent on geography (different places in the world have different amounts of network cable), network costs will vary depending on where you are in the world. In Google's case, mainland China and Australia are the two regions in the world that currently cost more than everywhere else.

Additionally, as you send more data in a given month beyond a terabyte, you'll get a reduced rate in the ballpark of 5% to 10%. In table 8.11, you can see how the prices stack up. It's most likely that an average user would fall into the first column (serving up to 1 TB of data per month), and if based in the United States and targeting US-based customers, the last row will be the most common. In the average US-focused case, network charges will come to 12 cents per Gigabyte served.

Table 8.11 Egress network prices per GB

Region	First TB / mo	Next 9 TB / mo	Beyond 10 TB / mo
China (not Hong Kong)	\$0.23	\$0.22	\$0.20
Australia	\$0.19	\$0.18	\$0.15
Anywhere else (for example, the United States)	\$0.12	\$0.11	\$0.08

To put this into context, if you download a 1 MB file from your Cloud Storage bucket to your office desktop in New York City, you'll be charged $0.001 \text{ (GB)} * 0.12 \text{ (USD)}$, or \$0.0001 to download the file. If you download that same file 1,000 times, your total cost will come to $1 \text{ (GB)} * 0.12 \text{ (USD)}$, or \$0.12. If you happen to go on vacation to Australia and do the same thing, your bill becomes $1 \text{ (GB)} * 0.19 \text{ (USD)}$, or \$0.19. One big exception to this component of your bill is in-network traffic.

In Google Cloud, network traffic that stays inside the same region is free of charge. If you create a bucket in the United States and then transfer data from that bucket to your Compute Engine instance in the same region, you won't be charged anything for that network traffic. On the flip side, if you have data stored in a bucket in Asia and download it to a Compute Engine instance in `us-central1-a`, you'll pay for that network traffic, whereas downloading it to an instance in `asia-east1-c` would be free.

8.9.3 Number of operations executed

Last, in addition to charges that depend on the amount of data you're storing or sending over the internet, Cloud Storage charges for a certain subset of operations you might perform on your buckets or objects. The no-free operations have two classes: a "cheap" class (for example, getting a single object) costing 1 cent for every 10,000 operations, and an "expensive" class (for example, updating an object's metadata), which costs 10 cents for every 10,000 operations. A good way to think of whether an operation is one of the cheap ones or one of the expensive ones is to look at whether it modifies any data in Cloud Storage. If it's writing data, it's likely one of the expensive operations, though there are exceptions. See table 8.12.

Table 8.12 Types of operations

Type	Cheap operations (\$0.01 per 10k)	Expensive operations (\$0.10 per 10k)
Read	<ul style="list-style-type: none"> ■ <code>*.get</code> ■ <code>*AccessControls.list</code> 	<ul style="list-style-type: none"> ■ <code>buckets.list</code> ■ <code>objects.list</code>
Write	Any notifications sent to your callback URL	<ul style="list-style-type: none"> ■ <code>*.insert</code> ■ <code>*.patch</code> ■ <code>*.update</code> ■ <code>objects.compose</code> ■ <code>objects.copy</code> ■ <code>objects.rewrite</code> ■ <code>objects.watchAll</code> ■ <code>*AccessControls.delete</code>

Notice that a few operations are missing from these lists because they're free. The free operations are (as you might expect) focused on deleting:

- channels.stop
- buckets.delete
- objects.delete

Let's move on and look in more detail at how Nearline and Coldline pricing works.

8.9.4 Nearline and Coldline pricing

As mentioned in sections 8.4.3 and 8.4.4, data in the Nearline and Coldline storage classes has a significantly cheaper data storage cost, but there can be drawbacks if the data is frequently accessed. In addition to the storage, network, and operations cost that you've learned about so far, Nearline and Coldline also include an extra cost for data retrieval, which is currently \$0.01 per GB retrieved on Nearline and \$0.05 per GB for Coldline. This is sort of like an internal networking cost that applies no matter where your destination is, which means that even downloading inside the same region from Cloud Storage to a Compute Engine instance will cost \$0.01 or \$0.05 per GB retrieved.

This might seem strange, but keep in mind that Nearline and Coldline were designed primarily for archiving, so in exchange for making it much cheaper to store your data safely, these classes add back that per-GB amount only if you retrieve your data. To put this in a more quantitative context, storing your 1 GB in multiregional storage (\$0.026 per month) is effectively the same cost as storing 1 GB in Nearline (\$0.01 per month) and accessing that 1 GB exactly 1.6 times every month (for example, retrieving 1.6 GB throughout the month, costing \$0.016). This means that your break-even point for storage depends on whether you retrieve 1.6 times the amount of data stored.

To drive this point home, imagine that you have 10,000 user-uploaded images totaling 1 GB and need to decide where to store these images. Let's also imagine that you're archiving these images and, therefore, plan to download all of them only once per year. Let's further make the assumption that the download will be to a Compute Engine instance in the same region, which lets you ignore network egress costs. See table 8.13.

Table 8.13 Pricing comparison (yearly access)

Class	Storage	Retrieval	Total
Nearline	\$1.20 (= 10 GB * \$0.01 per GB per month * 12 months)	\$0.10 (= 10 GB * 1 download per year * \$0.01 per GB downloaded)	\$1.30
Coldline	\$0.84 (= 10 GB * \$0.007 per GB per month * 12 months)	\$0.50 (= 10 GB * 1 download per year * \$0.05 per GB downloaded)	\$1.34
Multiregional	\$3.12 (= 10 GB * \$0.026 per GB per month * 12 months)	\$0.00 (= 10 GB * 1 download per year * \$0.00 per GB downloaded)	\$3.12

If you download your data only once per year, you're going to pay less if you use Nearline to store your data.

If you access your data frequently (for example, each image is downloaded at least once per week), this changes the pricing dynamic quite a bit, as shown in table 8.14.

Table 8.14 Pricing comparison (weekly access)

Class	Storage	Retrieval	Total
Nearline	\$1.20 (= 10 GB * \$0.01 per GB per month * 12 months)	\$5.20 (= 10 GB * 52 downloads per year * \$0.01 per GB downloaded)	\$6.40
Coldline	\$0.84 (= 10 GB * \$0.007 per GB per month * 12 months)	\$26.00 (= 10 GB * 52 downloads per year * \$0.05 per GB downloaded)	\$26.84
Multiregional	\$3.12 (= 10 GB * \$0.026 per GB per month * 12 months)	\$0.00 (= 10 GB * 52 downloads per year * \$0.00 per GB downloaded)	\$3.12

In this scenario, you'll end up paying around twice as much to use Nearline, with the cost driven almost exclusively by the data retrieval cost.

If you happen to never need to access the data, you can see how Coldline shines in table 8.15.

Table 8.15 Pricing comparison (no access)

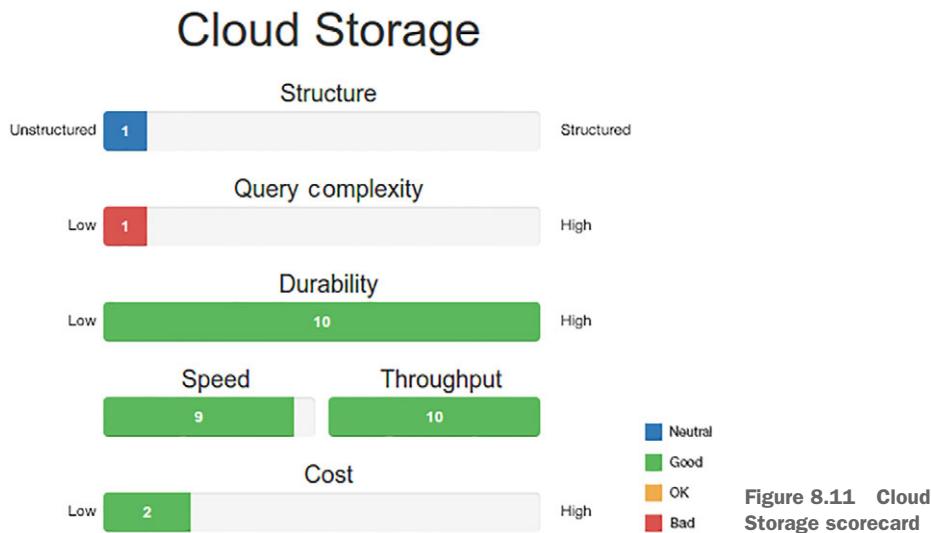
Class	Storage	Retrieval	Total
Nearline	\$1.20 (= 10 GB * \$0.01 per GB per month * 12 months)	\$0.00 (= 10 GB * 0 downloads per year * \$0.01 per GB downloaded)	\$1.20
Coldline	\$0.84 (= 10 GB * \$0.007 per GB per month * 12 months)	\$0.00 (= 10 GB * 0 downloads per year * \$0.05 per GB downloaded)	\$0.84
Multi-regional	\$3.12 (= 10 GB * \$0.026 per GB per month * 12 months)	\$0.00 (= 10 GB * 0 downloads per year * \$0.00 per GB downloaded)	\$3.12

This likely gives you some insight into when Nearline or Coldline storage may be good choices for your system. When in doubt, if you're saving stuff for a rainy day, Nearline might be the better choice, depending on how often it rains. If you're using the data in your application and serving it to users, multiregional (or regional) storage is probably a better fit.

8.10 When should I use Cloud Storage?

Unlike other storage systems, Cloud Storage is complementary to your system in more than one way. In a sense, using object storage is a bit more like a check box than one of the multiple-choice options.

As a result, this section will summarize Cloud Storage briefly using the same scorecard as the other services, shown in figure 8.11. I'll focus more on how Cloud Storage complements your other storage systems rather than whether Cloud Storage is a good fit at all.



8.10.1 Structure

Cloud Storage is by definition an *unstructured* storage system and is, therefore, meant to be used purely as a key-value storage system with no ability to handle any queries besides “give me the object at this key.”

Although you can technically query Cloud Storage for a list of objects based on a prefix, that querying ability should be treated more as an administrative function and not something to be used as a feature in your application.

8.10.2 Query complexity

Due to the complete lack of structure and the pure key-value nature of Cloud Storage, there is no ability to run queries of any complexity. That said, you shouldn't be using Cloud Storage if you need to query your data.

8.10.3 Durability

Durability is an aspect where Cloud Storage is strong, offering a 99.999999999% durability guarantee (that's 11 nines). Even with the cheaper options (Nearline or Coldline),

your data is automatically replicated in several different places by default, so it's as safe as you can possibly make it, on par with Cloud Datastore or Persistent Disks in Compute Engine.

This is done using erasure coding—a form of error correction that chops up data into lots of pieces and stores that data redundantly on many disks spread out across lots of failure domains (considering both network failure and power failure). For example, even if two disks fail with your data on them, your data is still safe and hasn't been lost.

8.10.4 Speed (latency)

Latency is one area where Cloud Storage allows you to choose what to expect for your application. By default, multiregional storage is sufficiently fast to bring the latency (measured as time to the first byte) into the milliseconds. If you're less interested in first-byte latency and more interested in saving money, you can choose either Nearline or Coldline storage if you're dealing more with archival or infrequently accessed data.

If you need the speed, it's there. If you don't, you can save some money.

8.10.5 Throughput

This is another strong area for Cloud Storage. Because Cloud Storage is optimized for throughput, you effectively can treat it as a never-ending resource for throughput. It's obviously not an infinite resource for throughput—after all, there's only so much network cable in the world—but Google automatically manages capacity on a global scale to make sure that you never get stuck in need of a faster download.

8.10.6 Overall

As mentioned earlier, instead of focusing on the typical storage needs of each application and seeing how this service stacks up, this section will focus on the ways that each application can use Cloud Storage and how good of a fit it is.

8.10.7 To-do list

The To-Do List probably won't have much use for Cloud Storage, specifically because most of the data stored is textual rather than binary. If you want to support image uploads in To-Do List, Cloud Storage is a great place to put that data. See table 8.16.

Table 8.16 To-Do List use for storage classes

Storage class	Use case
Multiregional	Storing customer image uploads (for example, profile pictures)
Regional	Storing larger customer attachments (for example, Excel files)
Nearline	Archiving database backups
Coldline	Archiving request logs

Given To-Do List serves customer data, you'll most likely want to use the multiregional storage class for your bucket. If you're trying to pinch pennies, regional could technically work. We don't know where users will be, so those who happen to be far away from the data may see worse overall performance.

8.10.8 E*Exchange

E*Exchange is much less likely to need attachments but might still need to store trading history in an archive or tax documents as PDF files. In these cases, the best choice will likely be multiregional for user-facing downloads, Nearline for trading reports, and Coldline for trade logs in case the SEC wants to perform an annual audit. If the exchange runs some analysis over trading data, because you know where the computation will happen, regional storage may be a good choice here. See table 8.17.

Table 8.17 E*Exchange storage needs

Storage class	Use case
Multiregional	Customer tax documents as PDF files
Regional	Data analysis jobs
Nearline	Customer trading reports
Coldline	Systemwide audit logs

8.10.9 InstaSnap

InstaSnap is a user-facing app you can find and is also focused mostly on customer-uploaded images, with image latency being important. Because of this, multiregional storage with the lowest latency and highest availability is likely the right choice. You also might want to archive database backups using Nearline and user access logs using Coldline. See table 8.18.

Table 8.18 InstaSnap storage needs

Storage class	Use case
Multiregional	Storing customer-uploaded images
Regional	No obvious use case
Nearline	Weekly database backups
Coldline	Archive user-access logs

Summary

- Google Cloud Storage is an object storage system that allows you to store arbitrary chunks of bytes (objects) without worry about disk drives, replication, and so on.
- Cloud Storage offers several storage classes, each with its own trade-offs (for example, lower cost for lower availability).
- Although Cloud Storage is mainly about storing chunks of data, it also provides extra features like automatic deletion for old data (lifecycle management), storing multiple versions of data, advanced access control (using ACLs), and notification of changes to objects and buckets.
- Unlike other storage systems you've learned about, Cloud Storage complements the others and, as a result, is typically used in addition to those rather than instead of them.

Part 3

Computing

N

ow that we've gone through ways to store data, it's time to think about the various computing options we can use to interact with that data.

Similar to storage systems, quite a few computing options are available, each with its own benefits and drawbacks. Each of these options allows you to express the computational work to be done using different layers of abstraction, from the lowest level (working with a virtual machine) all the way up to a single JavaScript function running in the cloud.

In this part of the book, we'll look at the various computing environments and dig down into how they all work. Some of these might feel familiar if you've worked with any sort of server before (for example, Compute Engine in chapter 9, which just hands you a virtual server), whereas others might seem foreign (for example, App Engine in chapter 11, which is a full-featured hosting environment), but it's important to understand the differences to make an informed decision when it comes time to build your next project.

And finally, as an added bonus, in chapter 13 we'll explore how you can use Cloud DNS to give human-readable names to all the computing resources you end up creating over time.

Compute Engine: virtual machines

This chapter covers

- What are virtual machines (VMs)?
- Using persistent storage with virtual machines
- How auto-scaling works
- Spreading traffic across multiple machines with a load balancer
- Compute Engine's pricing structure

As you've learned, virtual machines are chopped-up pieces of a single physical system that are shared between several people. This isn't a new idea—even 10 years ago this was how virtual private servers were sold—but the idea certainly has gotten more advanced with Cloud hosting platforms like Google Cloud Platform and Amazon Web Services. For example, it's now possible to decouple the virtual machine from the physical machine, so physical machines can be taken offline for maintenance while the virtual machine is live-migrated elsewhere, all without any downtime or significant changes in performance.

Advances like these enable even more neat features like automatic scaling, where the hosting provider can automatically provision more or fewer virtual machines based on incoming traffic or CPU usage, but these features sometimes

can be tricky to understand and configure. To make things less tricky, the goal of this chapter is to get you comfortable with virtual machines, explain some of their interesting performance characteristics (particularly those that might seem counterintuitive), and walk you through the more advanced features (like automatic scaling).

Keep in mind that Google Compute Engine (GCE) is an enormous system with almost 40 API resources, meaning it'd be possible to write an entire book on just GCE. Because I need to fit the topic into this book, I'll instead focus on the most common and useful things you can do with GCE and dive deep into the details of a few of those things where necessary. Let's jump right in by creating a virtual machine on GCE.

9.1 Launching your first (or second) VM

You already learned how to launch a VM from the Cloud Console (see chapter 2); now try launching one from the command line using `gcloud`.

NOTE If you don't have `gcloud` installed yet, check out <https://cloud.google.com/sdk> for instructions on how to get set up.

The first thing you'll need to do is authenticate using `gcloud auth login`. Then you should make sure you have your project set as default by using `gcloud config set project your-project-id-here`. After that, you can create a new instance in the `us-central1-a` zone and connect to it using the `gcloud compute ssh` command:

```
$ gcloud compute instances create test-instance-1
  --zone us-central1-a
Created [https://www.googleapis.com/compute/v1/projects/
  your-project-id-here/zones/us-central1-a/instances/
  test-instance-1].
NAME          ZONE          MACHINE_TYPE      STATUS
test-instance-1  us-central1-a  n1-standard-1    RUNNING
```

The diagram illustrates the command flow. The first two lines of the command are grouped together with a bracket and labeled "Creates the new instance". The third line of the command is grouped with the output table and labeled "Connects to the instance over SSH".

```
$ gcloud compute ssh --zone us-central1-a test-instance-1
Warning: Permanently added 'compute.1446186297696272700' (ECDSA) to
  the list of known hosts.
# ... Some welcome text here ...
jjg@test-instance-1:~$
```

If this all seems a bit too easy, that's kind of the point. The goal of cloud computing is to simplify physical infrastructure so you can focus on building your software rather than dealing with the hardware it runs on. That said, there's far more to GCE than turning on VMs.

Let's look at an overview of all the components in a fully autoscaling system built using GCE. This system is capable of expanding or contracting (creating or destroying VMs) based on the load on it at any given time (figure 9.1).

NOTE This may look scary at first. Don't worry! We'll walk through the pieces involved, and by the end of the chapter you should understand all of them!

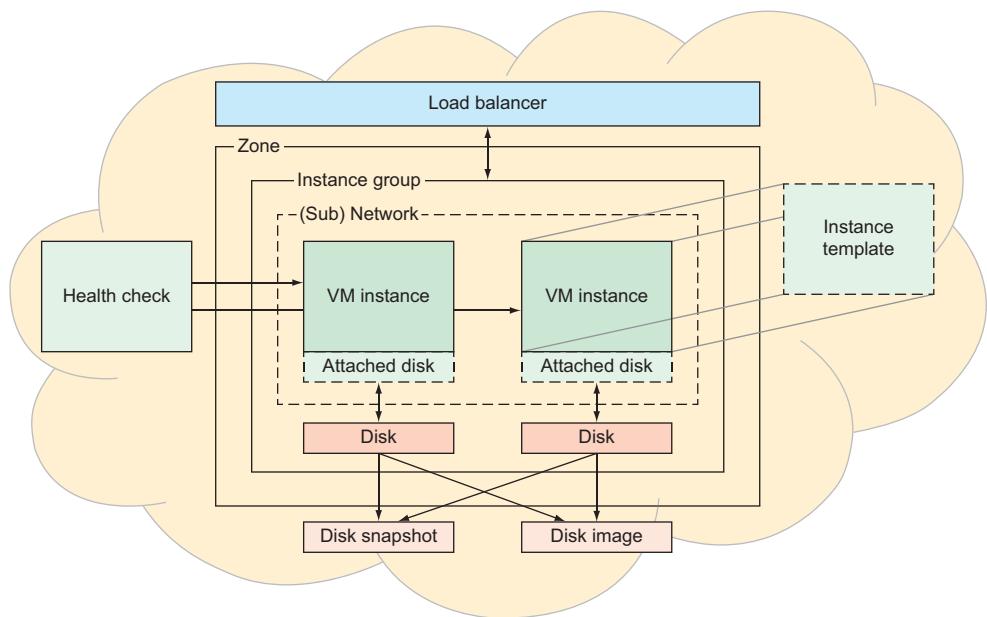


Figure 9.1 A complete overview of GCE

As you can see, quite a lot is going on in this diagram. Why on earth do you need so much stuff? The short version is, you don't! If all you need is a simple VM that you can SSH into and run a server or two, you've learned all you need to learn. The longer version is that you may eventually want to do more advanced things, like customize your virtual machines or balance server requests across a set of many machines. GCE gives you ways to do all of these things, but they're a bit more complicated than typing a single `gcloud` command, so you need to understand a few concepts first. To help you with that, let's move on to the next phase of customizing your deployment by starting with a simplified version of the scary diagram that's much easier to digest (figure 9.2). In this diagram, you can clearly see that disk storage makes up the base of your instance, so let's explore what these disks are and how they work.

9.2 Block storage with Persistent Disks

A persistent disk is a bit like an external hard drive. As with a hard drive, you can get a persistent disk in varying sizes (for example, 100 GB or 1 TB), and you can plug into it virtually using any computer to see the data stored on it, similar to how you could physically plug a hard drive into any computer. This might sound like a basic must-have thing, but originally this storage was entirely ephemeral—whenever you restarted a machine, all of the data stored on the local disk would be completely gone, which could be anywhere from frustrating to dangerous.

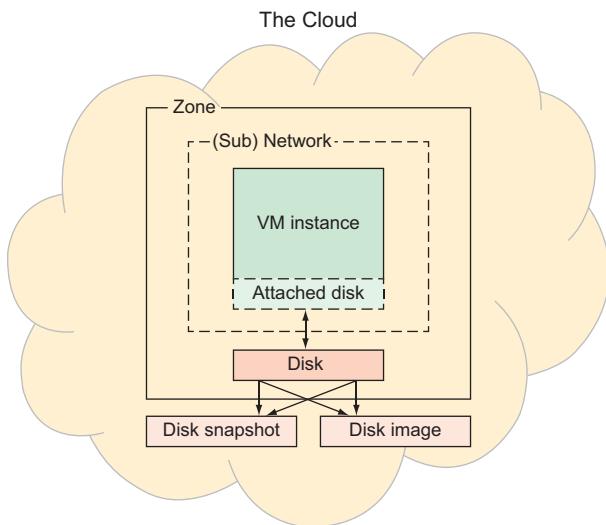


Figure 9.2 A simpler overview of GCE

To fix the issue, cloud hosting providers came up with a storage service that looked and acted like a regular disk but was replicated and highly available. In Google's case, this is called Persistent Disk. Let's look further into the details of how these disks work.

9.2.1 Disks as resources

So far, I've only dealt with disks as part of creating virtual machines, but they aren't limited to that use. You can create and manage disks separately from VMs, and you can even attach and detach them from instances while they're running! Although until now we've only looked at disks when they were attached to a VM, they can be in many other states. Let's go through the lifecycle of a disk and all of the things you can do with one.

At any time, a persistent disk can be in one of three states:

- Unattached—You've created the disk, but it's not mounted on any VMs.
- Attached in read-only mode—The VM can only read from the disk.
- Attached in read-write mode—The VM can both read and write to the disk.

I've only talked about disks in the attached-read-write state because that's been the default when creating a VM. Let's widen the focus and explore how all these states work and how you transition between them.

Figure 9.3 shows how you can transition between the various disk states. As you can see, the default value when creating a disk in GCE is the unattached state, which means the disk exists but isn't in use by any VMs. You might think of this disk as being archived somewhere, ready for use sometime later.

You can attach disks to a VM in two different modes (read-only and read-write) that are pretty self-explanatory, but it's worth noting that the read-write mode is exclusive,

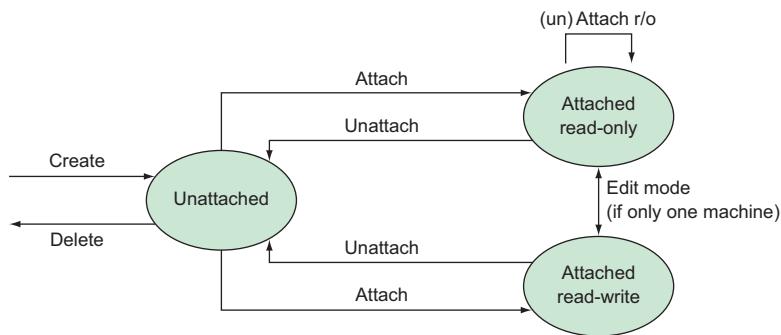


Figure 9.3 Disk states and transitions

whereas read-only mode isn't. You can attach a single disk in read-only mode to as many VMs as you want, but if a disk is attached in read-write mode to a VM, you can't attach it to any other VMs, regardless of the mode.

Now you have a grasp of the disk states and the rules for attaching them. Let's explore how you go about using the disks.

9.2.2 Attaching and detaching disks

To make these ideas a bit more concrete, you're going to create a disk and take it through the different states. To do so, assume you have two VMs that exist already, called `instance-1` and `instance-2`. The details of the VMs aren't important, so don't get hung up on those. You can list them to see what you have:

```
$ gcloud compute instances list
NAME      ZONE      MACHINE_TYPE      STATUS
instance-1 us-central1-a  g1-small      RUNNING
instance-2 us-central1-a  g1-small      RUNNING
```

In the Cloud Console, jump into the Compute Engine section and choose Disks in the left-side navigation. Then click Create Disk, which will bring you to a page that should look familiar (figure 9.4). The first thing that should jump out at you is that you need to choose a name for your disk, which, like a VM instance, must be unique. Also like a VM, disks live inside specific zones, which means the uniqueness of the name is specific to a single zone.

TIP Although you can have two disks with the same name in different zones, it's generally not a good idea because it's easy to mix them up.

When it comes to choosing a location, remember that to be attached to an instance, a disk must live in the same zone as that instance; otherwise, you would risk latency spikes when accessing data. Next, you'll need to choose a disk type, which is focused mainly on performance, with standard disks acting a lot like traditional hard drives and SSD disks acting like solid-state drives. The right choice will depend on your

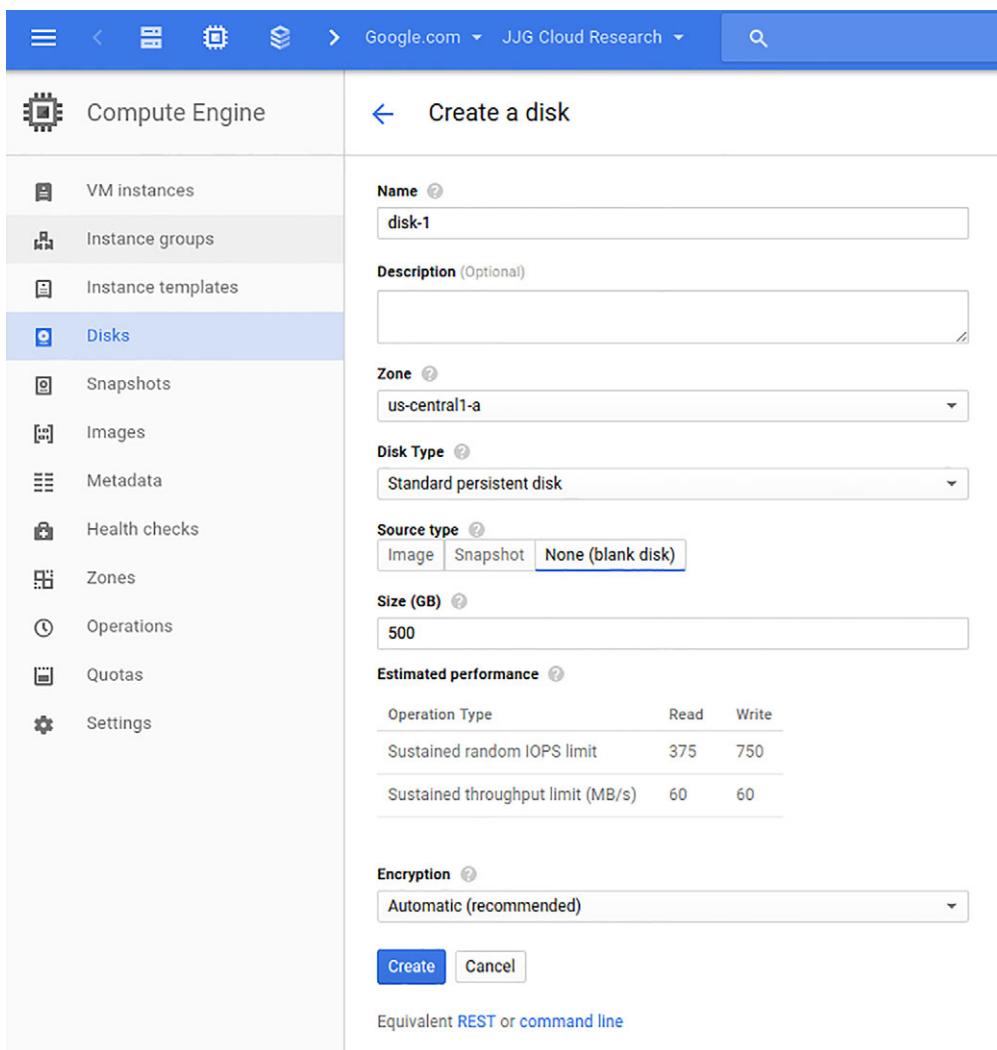


Figure 9.4 Creating your disk

access patterns. SSDs have much faster random operations, and traditional drives are adequate for large sequential operations. After that, leave the Source Type as None (Blank Disk) to create an empty disk resource, and choose any size you want for the disk; for example, 500 GB.

As I discussed in section 4.5.2 of chapter 4, disk size and performance are directly related, such that larger disks can handle more input/output operations per second (IOPS). Typically, applications that don't store a lot of data but have heavy access patterns (lots of reads and writes) will provision a larger disk, not for the storage capacity but for the performance characteristics. You'll also notice that as you enter a size (in

GB), you can see the estimated performance below the field. Finally, a field allows you to choose what type of encryption to use. For now, leave this as-is—I'll discuss disk encryption later on.

Now you can use the command line to look at your disks like you did with looking at the running instances:

```
$ gcloudcompute disks list
NAME      ZONE      SIZE_GB      TYPE      STATUS
disk-1    us-central1-a  500        pd-standard  READY
instance-1 us-central1-a   10        pd-standard  READY
instance-2 us-central1-a   10        pd-standard  READY
```

If you're wondering why three disks (two of which have names starting with "instance") are listed instead of one, remember that when you create an instance, GCE also has to create a disk, and it comes with a preset value of 10 GB for the total storage size. The two other disks you're seeing here (`instance-1` and `instance-2`) are the disks that were automatically created when you turned on your instances.

Now that you have a newly created disk, what can you do with it? Referring to the state diagram shown in figure 9.3, this disk is in the unattached state. You can move it through the other states by first attaching it as a read-only disk to `instance-1`, then `instance-2`. To start, go back to the Cloud Console and look at `instance-1`. If you scroll down the page a bit, you'll see a section called Additional Disks, which has the informative statement "None" listed there (figure 9.5).

Boot disk and local disks			
Name	Size (GB)	Type	Mode
instance-1	10	Standard persistent disk	Boot, read/write
<input checked="" type="checkbox"/> Delete boot disk when instance is deleted			
Additional disks			
None			
Network			
default			
Firewalls			

Figure 9.5 No additional disks

To attach your disk to this instance, choose Edit from the top of the page, then click +Add Item under that Additional Disks heading. You can choose your new disk (`disk-1`) from the list, but be certain you choose to attach it in Read Only mode! (See figure 9.6.) Then scroll to the bottom and click Save, and you should see `disk-1` is attached to your instance.

Now your disk is in the attached-read-only state, which means that it can continue to be attached to other VMs, but if you were to try to write to this disk from `instance-1`,

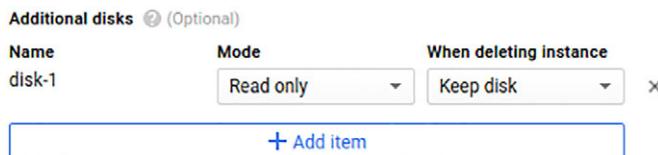


Figure 9.6 Attach an additional disk

the operation would fail with an error. This comes in handy when you have information on a persistent disk that you don't intend to modify from a VM. It prevents disasters if you accidentally run a script or type `rm -rf` in the wrong place.

Now you can attach that same disk to `instance-2`, again in read-only mode. This time, you can do so using the `attach-disk` subcommand. Before you do that, though, try attaching your disk to `instance-2` in read-write mode to confirm that it throws an error:

```
$ gcloud compute instances attach-disk instance-2
  --zone us-central1-a --disk disk-1 --mode rw
ERROR: (gcloud.compute.instances.attach-disk) Some requests did not succeed:
- The disk resource 'disk-1' is already being used by 'instance-1'
```

Attaching the disk in
read-write mode fails
because it's already
attached elsewhere.

```
$ gcloud compute instances attach-disk instance-2
  --zone us-central1-a --disk disk-1 --mode ro
Updated [https://www.googleapis.com/compute/v1/projects/
  your-project-id-here/zones/us-central1-a/instances/instance-2].
```

Attaching the disk
in read-only mode
succeeds as expected.

After this, if you go back to the Cloud Console and look at your list of disks, you'll see that `disk-1` is in use by both `instance-1` and `instance-2`. Now that disks are attached, how do you start saving data on them?

9.2.3 Using your disks

So far, we've looked at creating and managing disks but you haven't read any data from them, or written any to them. It turns out that under the hood, when you attach a disk to an instance, it's kind of like plugging your external hard drive into the VM. As with any brand-new drive, before you can do anything else, you have to mount the disk device and then format it. In Ubuntu, you can do this with the `mount` command as well as by calling the `mkfs.ext4` shortcut to format the disk with the `ext4` file system.

First, you have to get your disk into read-write mode, which you can do by detaching the disk from both instances and then reattaching the disk to `instance-1` in read-write mode:

```
$ gcloud compute instances detach-disk instance-1
  --zone us-central1-a --disk disk-1
$ gcloud compute instances detach-disk instance-2
  --zone us-central1-a --disk disk-1
```

Detaches the disk
from both instances

```
$ gcloud compute instances attach-disk instance-1
  ↵ --zone us-central1-a --disk disk-1 --mode rw
```



Reattaches the disk in read-write mode

Once you have disk-1 attached to instance-1 only, and in read-write mode, SSH into instance-1 and look at the disks, which are conveniently located in /dev/disk/by-id/. In the following snippet, you can see that disk-1 has a friendly alias as /dev/disk/by-id/google-disk-1, which you can use to point at the Linux device:

```
jjg@instance-1:~$ ls -l /dev/disk/by-id
total 0
lrwxrwxrwx 1 root root 9 Sep  5 19:48 google-disk-1 -> ../../sdb
lrwxrwxrwx 1 root root 9 Aug 31 11:36 google-instance-1 -> ../../sda
lrwxrwxrwx 1 root root 10 Aug 31 11:36 google-instance-1-part1 -> ../../sda1
lrwxrwxrwx 1 root root 9 Sep  5 19:48 scsi-0Google_PersistentDisk_
  ↵ disk-1 -> ../../sdb
lrwxrwxrwx 1 root root 9 Aug 31 11:36 scsi-0Google_PersistentDisk_
  ↵ instance-1 -> ../../sda
lrwxrwxrwx 1 root root 10 Aug 31 11:36 scsi-0Google_PersistentDisk_
  ↵ instance-1-part1 -> ../../sda1
```

WARNING Don't run this against a disk that has data on it, because it deletes all data on the disk!

Start by formatting the disk using its device ID (/dev/disk/by-id/google-disk-1). In the example shown in the following snippet, you'll use some extended options (passed in via the -E flag) that the disk team at Google recommends. Once the disk is formatted, you'll mount it like any other hard drive:

```
jjg@instance-1:~$ sudo mkfs.ext4 -F -E
  ↵ lazy_itable_init=0,lazy_journal_init=0,discard
  ↵ /dev/disk/by-id/google-disk-1
mke2fs 1.42.12 (29-Aug-2014)
Discarding device blocks: done
Creating filesystem with 1441792004k blocks and 36044800 inodes
Filesystem UUID: 37d0454e-e53f-49ab-98fe-dbed97a9d2c4
Superblock backups stored on blocks:
      32768, 98304, 163840, 229376, 294912, 819200, 884736,
  ↵ 1605632, 2654208,
      4096000, 7962624, 11239424, 20480000, 23887872, 71663616,
  ↵ 78675968,
      102400000

Allocating group tables: done
Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done

jjg@instance-1:~$ sudo mkdir -p /mnt/disks/disk-1
jjg@instance-1:~$ sudo mount -o discard,defaults
  ↵ /dev/disk/by-id/google-disk-1 /mnt/disks/disk-1
```

At this point, the disk is ready, but `root` still owns it, which could be irritating because you can't write to it without taking on superuser privileges, so it may be worthwhile to change the owner to yourself. Then you can start writing data to the disk like you would on a regular desktop. Here's how you change the owner and write data to the disk:

```
jjg@instance-1:~$ cd /mnt/disks/disk-1
jjg@instance-1:/mnt/disks/disk-1$ sudomkdir workspace
jjg@instance-1:/mnt/disks/disk-1$ sudochownjjg:jjg workspace/
jjg@instance-1:/mnt/disks/disk-1$ cd workspace
jjg@instance-1:/mnt/disks/disk-1/workspace$ echo "This is a test" > test.txt
jjg@instance-1:/mnt/disks/disk-1/workspace$ ls -l
total 4
-rw-r--r-- 1 jjgjjg 15 Sep 12 12:43 test.txt
jjg@instance-1:/mnt/disks/disk-1/workspace$ cat test.txt
This is a test
```

You've seen how to interact with this disk. Now let's look at a common problem: running out of space.

9.2.4 Resizing disks

You might want to resize a disk for a variety of reasons. In addition to running out of space, you might recall that the size of the disk directly correlates to its speed: the bigger the disk, the faster it is. How do you resize the disk itself?

First, you have to increase the size of the virtual disk in the Cloud Console by clicking Edit on the disk and then typing in the new size (figure 9.7).

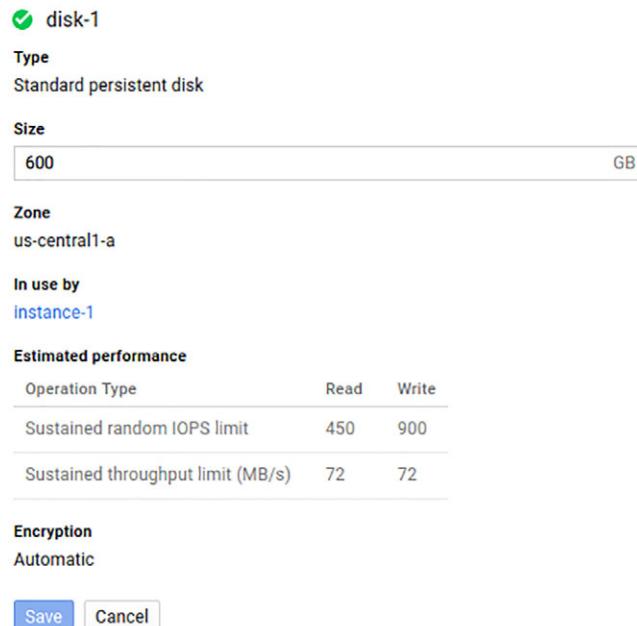


Figure 9.7 Resizing your disk in the Cloud Console

WARNING Keep in mind that you can always make a disk larger by increasing the size, but you can't make a disk smaller. You should be particularly careful when increasing the size of your disk. To undo that increase, you'll need to create a new, smaller disk and copy your data over, which will cost more money and be time-consuming.

Once you've done that, you have to resize your file system (in the previous example, this was the ext4 file system) to fill up the newly allotted space. To do this, you can use the `resize2fs` command on an unmounted disk:

```
jjg@instance-1:~$ sudo umount /mnt/disks/disk-1
jjg@instance-1:~$ sudo e2fsck -f /dev/disk/by-id/google-disk-1
e2fsck 1.42.12 (29-Aug-2014)
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
/dev/disk/by-id/google-disk-1: 13/36044800 files (0.0%
➥ non-contiguous), 2312826/144179200 blocks
jjg@instance-1:~$ sudo resize2fs /dev/disk/by-id/google-disk-1
resize2fs 1.42.12 (29-Aug-2014)
Resizing the filesystem on /dev/disk/by-id/google-disk-1 to
➥ 157286400 (4k) blocks.
The filesystem on /dev/disk/by-id/google-disk-1 is now 157286400
➥ (4k) blocks long.
```

If you get an error about the target being busy, make sure you're not doing anything with the disk, then wait a bit. A background process could be preventing the disk from being unmounted.

Now you can remount the disk, and you should see that it has expanded to fill the available space on the virtual device:

```
jjg@instance-1:~$ sudo mount -o discard,defaults
➥ /dev/disk/by-id/google-disk-1 /mnt/disks/disk-1
jjg@instance-1:~$ df -h | grep disk-1
/dev/sdb      591G   70M  561G  1% /mnt/disks/disk-1
```

That's how you manage disks. Now let's explore some of the aspects of disks that are unique to virtualized devices like Google's Persistent Disks, starting with the concept of disk snapshots.

9.2.5 Snapshots

Have you ever wanted to freeze your computer at a point in time and be able to jump right to that checkpoint? Maybe because you accidentally deleted a file? Although snapshots weren't intended solely for those *oops* moments, they can act as those checkpoints for the data on your disk, allowing you to jump around in time by restoring a snapshot to a disk instance. And because snapshots act like checkpoints rather than copies of your disk, they end up costing you much less than a full backup.

This works because snapshots use *differential storage*, storing only what's changed from one snapshot to the next. For example, if you create a snapshot, change one block of data, and then take another snapshot, the second snapshot will only store the

difference (or *delta*) between the two snapshots (in this case, only the one block), rather than an entire copy.

Storage savings aside, snapshots act mostly like regular disks in that you can create and delete them at any time. Unlike a regular disk, you can't read or write from one directly. Instead, once you have a snapshot of a disk, you can create a new disk based on the content from the snapshot.

To see how this works, do a quick experiment (figure 9.8) using `disk-1` that walks through the lifecycle of a snapshot in the following steps:

- 1 Start with `disk-1` attached to your instance.
- 2 Create a snapshot (`snapshot-1`) from `disk-1`.
- 3 Change some data on the mounted copy of `disk-1`.
- 4 Create a new instance based on your snapshot and mount it to your instance.

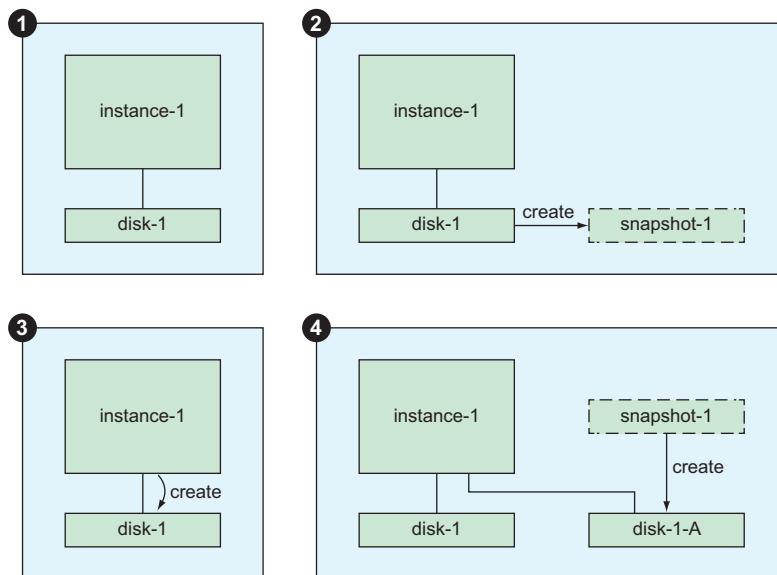


Figure 9.8 Visualizing your experiment

By doing this, you'll have two versions of `disk-1` attached to the VM. One of the disks will be the old version reflecting the snapshot from step 1, and the other the current version with the data you modified in step 2.

For this experiment, start by taking a snapshot of `disk-1`. Look at the list of disks and click on `disk-1`. Then click **Create Snapshot** at the top of the page, which should bring you to a form to create a new snapshot from your disk (figure 9.9).

Click **Create** and wait a few seconds. You'll be sent over to a page that lists the snapshots in your project (figure 9.10).

[←](#) Create a snapshot

Name [?](#)
disk-1

Description (Optional)

Source disk [?](#)
disk-1

Encryption [?](#)
Automatic (recommended)

Integrate volume shadow copy service [?](#)
 Enable VSS

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

Equivalent REST or [command line](#)

Figure 9.9 Creating a new snapshot

Snapshots		CREATE SNAPSHOT	REFRESH	DELETE
<hr/>				
<hr/>				
<input type="checkbox"/>	Name ^	Source disk	Creation time	Disk size
<input checked="" type="checkbox"/>	disk-1	disk-1	Sep 12, 2016, 5:44:21 PM	600 GB
				3.25 MB

Figure 9.10 A list of your snapshots

Now that you have a new snapshot, you can change some data on your current disk-1:

```
jjg@instance-1:~$ cd /mnt/disks/disk-1/workspace/
jjg@instance-1:/mnt/disks/disk-1/workspace$ echo "This is changed
➡ after the snapshot!" > test.txt
jjg@instance-1:/mnt/disks/disk-1/workspace$ cat test.txt
This is changed after the snapshot!
```

Now imagine you forgot what test.txt used to have written in it and want to go back in time. To do this, create a disk instance from your snapshot; then you can mount it to your machine like any other disk. Start by navigating to the list of disks and choosing Create Disk. Instead of creating a blank disk like you did before, this time you're going to choose Snapshot as the source type, then choose disk-1 as your source snapshot. The rest of the fields should look similar to the other times you've created a disk (figure 9.11).

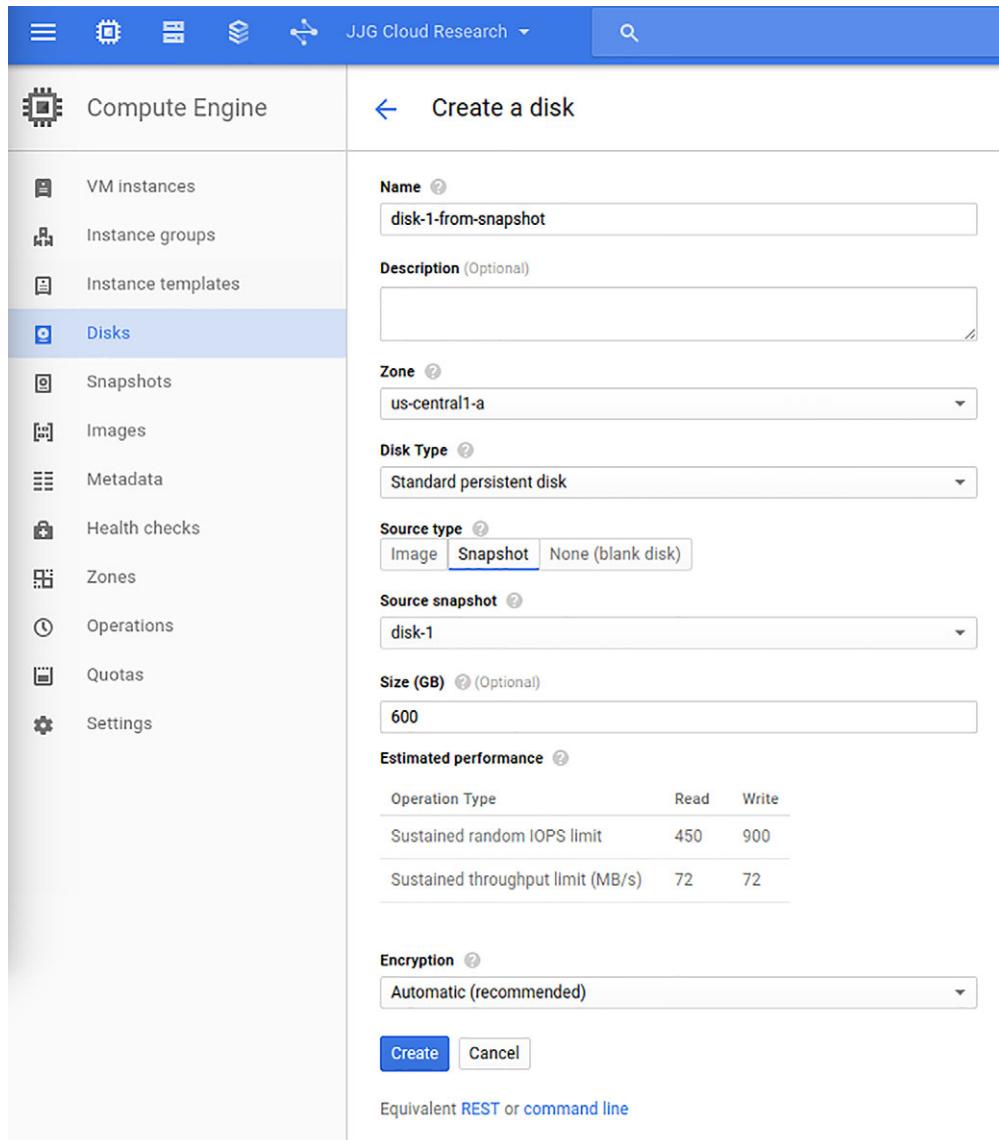


Figure 9.11 Creating a disk instance from a snapshot

WARNING Don't forget to choose us-central1-a for the zone. Otherwise, you won't be able to mount your disk to your VM!

When you click Create, you'll be brought to the list of disks. You should see your newly created disk (in your case, disk-1-from-snapshot) in the list (figure 9.12).

<input type="checkbox"/> Name ^	Type	Size	Zone	In use by	<input type="button"/>
<input type="checkbox"/> disk-1	Standard persistent disk	600 GB	us-central1-a		<input type="button"/>
<input type="checkbox"/> disk-1-from-snapshot	Standard persistent disk	600 GB	us-central1-a		<input type="button"/>
<input type="checkbox"/> instance-1	Standard persistent disk	50 GB	us-central1-c	wordpress	<input type="button"/>
<input type="checkbox"/> instance-2	SSD persistent disk	10 GB	us-central1-c		<input type="button"/>

Figure 9.12 List of disks

Now you can attach that disk to your VM (this time, you'll do it from the command line), and then you can mount the disk on the remote machine:

```
$ gcloud compute instances attach-disk instance-1
  --zone us-central1-a --disk disk-1-from-snapshot
  Updated [https://www.googleapis.com/compute/v1/projects/
  your-project-id-here/zones/us-central1-a/instances/instance-1] .

$ gcloud compute ssh --zone us-central1-a instance-1
# ...
Last login: Mon Sep  5 19:46:06 2016 from 104.132.34.72
jjg@instance-1:~$ sudo mkdir -p /mnt/disks/disk-1-from-snapshot
jjg@instance-1:~$ sudo mount -o discard,defaults
  /dev/disk/by-id/google-disk-1-from-snapshot
  /mnt/disks/disk-1-from-snapshot
```

Now you have both disks mounted to the same machine, where disk-1-from-snapshot holds the data you had before you modified it, and disk-1 holds the data from afterward. To see the difference, print out the contents of your test.txt file for each disk:

```
jjg@instance-1:~$ cat /mnt/disks/disk-1-from-snapshot/
  workspace/test.txt
This is a test
jjg@instance-1:~$ cat /mnt/disks/disk-1/workspace/test.txt
This is changed after the snapshot!
```

SNAPSHOT CONSISTENCY

But what happens if you’re writing to your disk, and you take a snapshot in between two important disk operations? Using the analogy of a bank transfer, what happens if you take the snapshot right between the bank deducting \$100 from your account and crediting \$100 to your friend’s account? (See figure 9.13.)

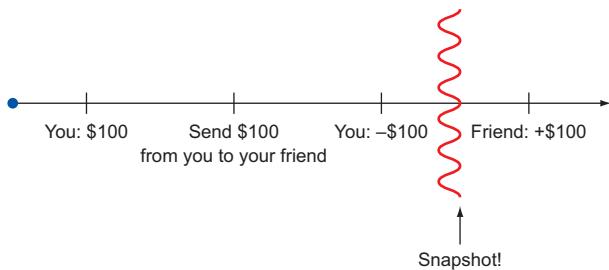


Figure 9.13 Bad timing for a snapshot

This issue is fundamentally low-level in that the problem arises because computers tend to cache things in memory instead of always writing the data to your hard drive. To avoid the types of problems that can come up when you take a snapshot at a bad time, you have to tell your virtual machine to flush any data that’s stored in memory but not yet on the disk. That only gets you so far; anything running on the machine might continue storing data in memory but not flushing it to the disk.

The end result is that to avoid a potentially disastrous snapshot, you should shut down your applications that are writing data (for example, stop your MySQL server binary), flush your disk buffers (using the `sync` command), freeze the disk (using `fsfreeze`) and only then take the snapshot:

```
jjg@instance-1:~$          ←———— Stops your applications
jjg@instance-1:~$ sudo sync
jjg@instance-1:~$ sudo fsfreeze -f /mnt/disks/disk-1           ←———— Creates your snapshot
jjg@instance-1:~$ 
jjg@instance-1:~$ sudo fsfreeze -u /mnt/disks/disk-1
```

While your disk is frozen (after calling `fsfreeze`), any attempts to write to the disk will wait until the disk is unfrozen. If you don’t halt your applications (for example, your MySQL server), they’ll hang until you unfreeze the file system.

Snapshots can protect your data over time, and now you understand how to use them. Now I’ll take a brief detour to talk about disk *images* and why you might want to use them.

9.2.6 Images

Images are similar to snapshots in that both can be used as the source of content when you create a new disk. The primary difference is that images are meant as starting templates for your disk, whereas snapshots are meant as a form of backup to pinpoint your disk’s content at a particular time. Every time you create a new VM from a base

operating system, you're using an image under the hood. The primary difference is that an image doesn't rely on differential storage like snapshots do, which means it may be more expensive to keep around.

In general, images are a good starting point for your VMs, and although you can create custom images, the curated list that Google provides should cover the common scenarios. Because of this, I'm not going to dig into the details of creating custom images, but you can find a tutorial on how to do this in the GCE documentation (<http://mng.bz/LKS7>). Now that you know a bit about images, let's switch gears from the mechanics of storing data and start looking at disk performance.

9.2.7 Performance

As I briefly discussed in earlier chapters, persistent disks are designed to abstract away the details of managing physical disks (for example, things like RAID arrays). You also learned that sometimes it makes sense to create a disk that's larger than you need for storage if you want to meet performance requirements. Although this can feel counterintuitive (and wasteful), rest assured that it's common and expected to create a disk that's larger than you need to get the performance that you need. That said, several classes of persistent disk (Standard, SSD, and Local SSD) are available, each with slightly different performance characteristics (table 9.1). Let's take a moment to go through each of them and explore when you might want to use the different classes.

Table 9.1 Disk performance summary

Type	Random IOPS per GB		Throughput (MB/s) per GB		Cost per GB
Standard	Read (max) 0.75 (3k max)	Write (max) 1.5 (15k max)	Read (max) 0.12 (180 max)	Write (max) 0.12 (120 max)	\$0.04
SSD	30 (25k max)	30 (25k max)	0.48 (240 max)	0.48 (240 max)	\$0.17
Local SSD	267 (400k max)	187 (280k max)	1.0 (1.5k max)	0.75 (1k max)	\$0.218

A few interesting things are hiding in this table that you should look at. To start, it's clear that Local SSD disks provide the most performance by far. Don't get too excited though, because these are local disks, meaning they aren't replicated and should be considered ephemeral rather than persistent. Put differently, if your machine goes away, so does all the data on local disks. Because Local SSDs are so different from the others, let's instead focus on the two truly persistent types: Standard and SSD.

SSD and Standard disks have two performance profiles, which I can summarize quickly. Standard disks are great if you need lots of space and don't need super-high performance, whereas SSDs are great for super-fast reads and writes. To put this in perspective, the graphs in figures 9.14 and 9.15 show a comparison between SSD and Standard disks in relation to read operation capacity and disk size.

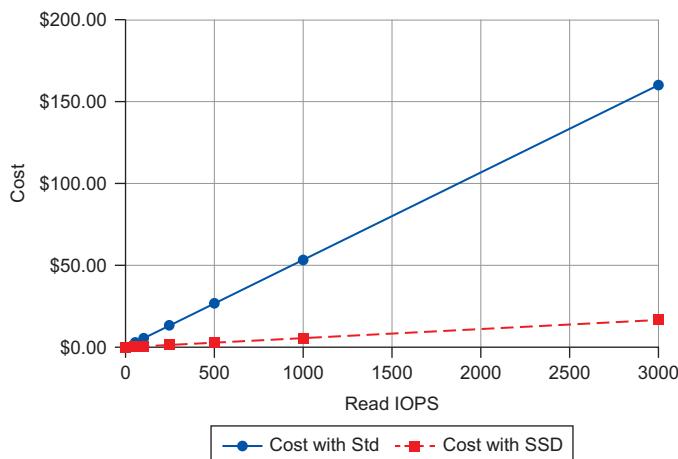


Figure 9.14 Graph of the cost by gigabyte stored

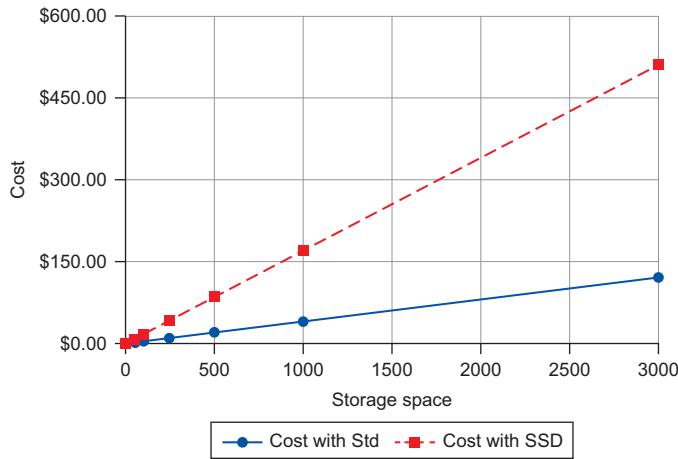


Figure 9.15 Graph of the cost by read IOPS

In the first graph, you can see the comparative cost to achieve a given level of read IOPS. Notice how the cost for additional IOPS with a Standard disk (circles) is far more than with an SSD (squares). In the second graph, you can see the comparative cost to store a given amount of data (in GB). Notice how the trend line is almost exactly reversed. This time, with the SSD (squares), each additional gigabyte of storage costs far more than with a standard disk (circles). Ultimately, when it comes time to create a disk that has the performance level you need, you have to look at both your throughput needs (how many GB/s do you need to read and write?) and your random-access needs (how many operations per second do you need?) and decide what disk type and size fits best.

Now that you understand disk performance a bit better, it's a good idea to jump back to that drop-down box I told you to ignore that talks about disk encryption. I'll briefly explain what that is and how it works.

9.2.8 Encryption

As you might guess, storing data in the cloud brings different risks than storing data locally on your home computer. Instead of worrying about someone breaking into your house, you have to worry about unauthorized access to your data via other means. You don't have to worry about fires at your house; instead, you need to worry about fires in a Google data center—it's a double-edged sword.

When you hear “unauthorized access to your data,” you probably tend to imagine a hacker in a foreign country trying to steal some private data. A less commonly imagined scenario is a Google employee copying and then accessing or selling your data, which could be equally bad. To help prevent such a scenario, Google encrypts the data stored on your disks, so even if someone were to copy the bytes directly, they'd be useless without the encryption keys. By default, Google comes up with its own random encryption key for your disk and stores that in a secure place with access logged, but if you're worried that Google's storage (and access logging) for the keys encrypting your disks isn't trustworthy enough, you can elect to keep these keys for yourself and give Google the key only when you need to decrypt the disk (such as when you first attach it to a VM).

To make this more concrete, let's quickly walk through the process of creating an encrypted disk where you manage the keys yourself. You'll start the process by getting a random key to use. To do this, you'll use `/dev/urandom` in Linux combined with the `tr` command to put a random chunk of bytes into a file called `key.bin`. To see what these bytes look like, use the `hexdump` command:

```
$ head -c 32 /dev/urandom | tr '\n' = >key.bin
$ hexdumpkey.bin
0000000 2a65 92b2 aa00 414b f946 29d9 c906 bf60
0000010 7069 d92f 80c8 4ad1 b341 0b7c 4d4f f9d6
0000020
```

At this point, you can decide either to use RSA encryption to wrap your key or leave the key as is. In the world of cryptography, *wrapping a key* involves encrypting it with a public encryption key so that it can only be decrypted by the corresponding private key. In this case, it's the way you ensure your secret is only able to be decrypted by Google Cloud Platform systems. As with most situations in security, storing things in plain text is typically bad, so it's recommended that you wrap your keys. But for the purposes of this example, you'll leave your key alone. (You can read more about how to wrap your keys in the GCE documentation at <http://mng.bz/6bYK>.) All you have left to do is to put the key in base64 format, which you can do with the `base64` command in Linux:

```
$ base64 key.bin
ZSqykgCqS0FG+dkpBslgv2lwL9nIgNFKQbN8C09N1vk=
```

At this point, you create a disk like you usually would, but choose Customer Supplied from the Encryption drop-down and leave the box for Wrapped Key unchecked

(because you're leaving the key as plain text). See figure 9.16. To finish up, click Create, and your disk should appear.

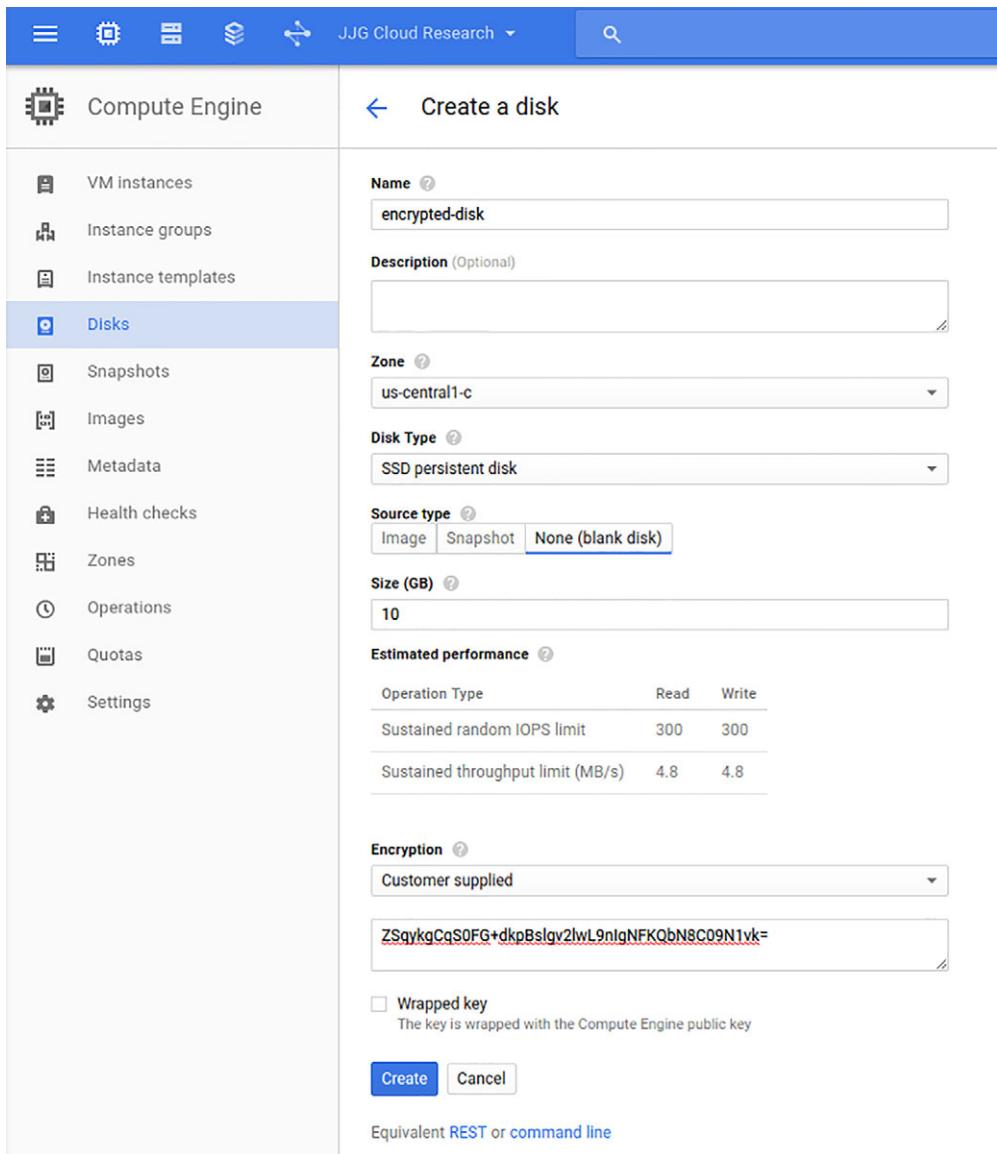


Figure 9.16 Creating an encrypted disk

Now that you have a disk that's encrypted with your key, let's look at how you attach that disk. To start, navigate to an existing instance from before and click Edit. As before, as shown in figure 9.17, in the section where you attach new disks, choose the

newly created “encrypted-disk” from the drop-down. You should notice something new this time, which is a section saying that the disk is encrypted and you’ll need to provide the key from before!

Boot disk and local disks			
Name	Size (GB)	Type	Mode
instance-3	100	SSD persistent disk	Boot, read/write

Delete boot disk when instance is deleted

Additional disks (Optional)

Name	Mode	When deleting instance
encrypted-disk	Read/write	Keep disk

Encryption ⓘ
This disk is encrypted. To attach it, please enter the encryption key.

SHA-256 hash
60c605b206fad279c42604e422db50aed13099fa3b1436c7698e41484abb1c4f

Wrapped key
The key is wrapped with the Compute Engine public key

[+ Add item](#)

Figure 9.17 Attaching an encrypted disk

Once you’ve pasted in the encryption key, scroll down and click Save. If you use the wrong key, you’ll see an error message like the one shown in figure 9.18. Google figures this out based on the hash of the key, which is stored along with the disk, so you can be sure the key provided is the right one without storing the actual key.



Figure 9.18 Invalid encryption key error message

As you can see, dealing with encrypted disks is similar to dealing with an unencrypted disk, with the main difference being that you have to provide some extra information (the key) when you attach an encrypted disk to an instance. Once the disk is mounted to the instance, it’ll act like a regular disk that I’ve talked about before, so everything

you learned previously still applies. Now that you understand everything there is to know about disks, I can talk a bit more about the computing features that I brought up earlier, which will show you why cloud computing is in a league above traditional VPS hosting.

9.3 **Instance groups and dynamic resources**

Given that you have a firm grip on how disks and instances interact, it's time to explore one of the more unique aspects of cloud computing: autoscaling. By autoscaling, I mean the ability to expand or contract the number of VMs running to handle requests based on how much traffic is being sent to them, which, for example, may show up as CPU usage. To make this a bit clearer, I'll use a concrete example of a system that experiences a request load that varies over the course of the day, as shown in figure 9.19.

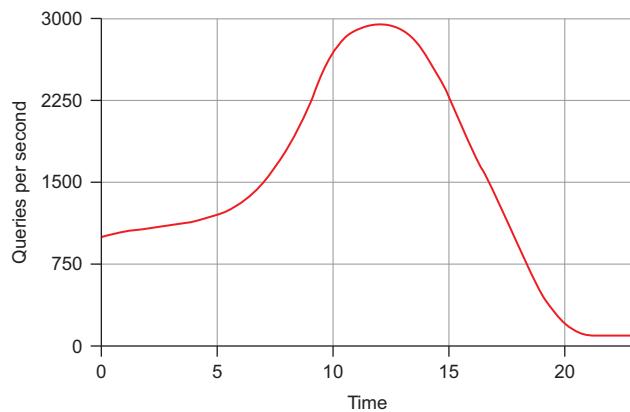


Figure 9.19 Queries per second throughout the day

As you can see, at the start of the day, the system sees around 1,000 queries every second, growing quickly until about noon, and it only slows as it approaches 3,000 queries per second. Then it steadily falls off to about 100 queries per second.

In a perfect world, this system would have exactly the right amount of capacity available to handle the number of requests needed. If you needed three machines at the start of the day, you'd need somewhere around three times that amount toward the middle of the day, and no more than one at the end. Currently, and unfortunately, all you've seen so far with GCE is the wasteful version of this graph, where you turn on the exact number of machines that you need to handle the worst part of the day. Those machines would be sitting idle for around half the day, as you can see in figure 9.20.

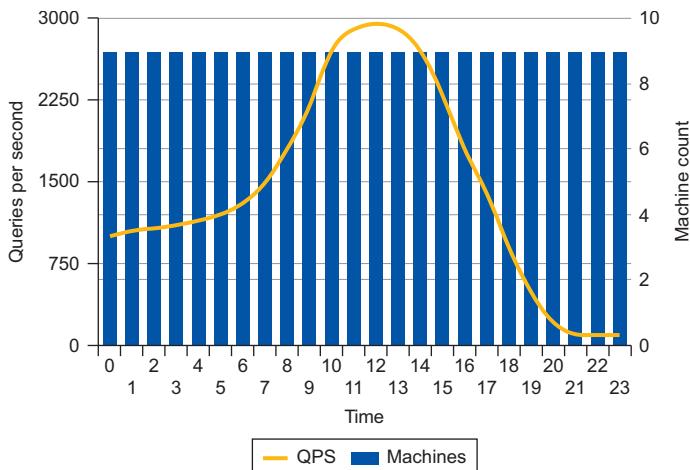


Figure 9.20 Machines provisioned for the worst part of the day

To make real life a bit more like the perfect world, where your system grows and shrinks to meet your demands, GCE's setup can use the concept of autoscaling. To put this visually, the number of machines added to this graph ideally would look something like figure 9.21.

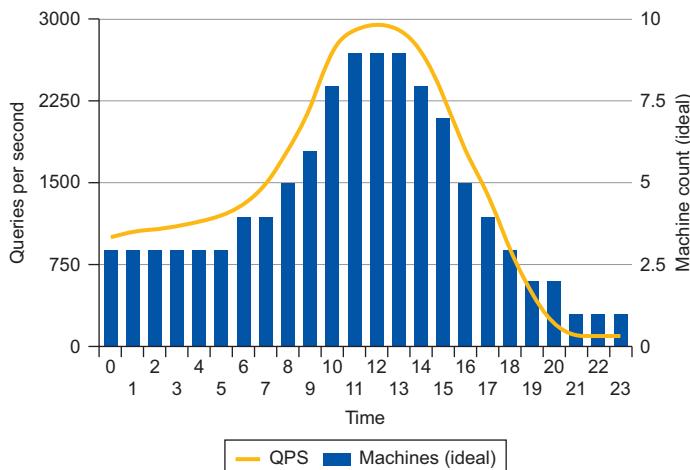


Figure 9.21 Machines required to handle requests

Because computing power is reactive, you might not be able to get as close as you'd like to the line, like the one in figure 9.21, but you can get much closer than the block shape shown in figure 9.20, where a specific number of machines is always on regardless of the traffic sent to them. Figure 9.22 shows a fairly realistic rendering of what you might achieve.

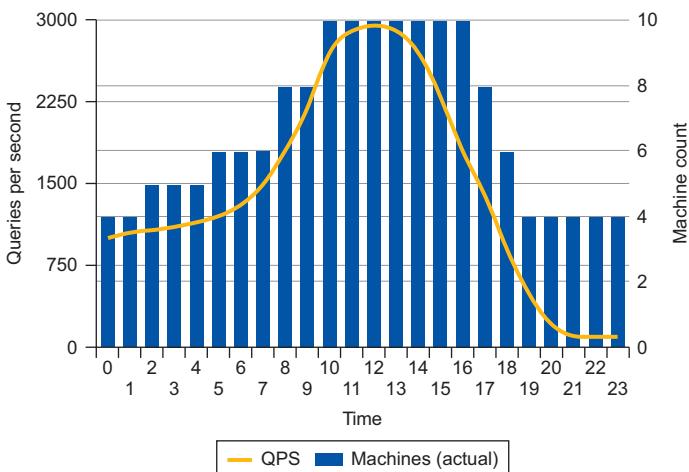


Figure 9.22 Machines that might handle requests with autoscaling

So how would this work? The main idea here is based on templates, where you teach GCE how to turn on instances configured to your liking. Once it knows how to turn on your instances, it can monitor the overall CPU usage of the currently running instances and decide whether to turn on more, turn off some that are currently running, or do nothing. To put all of this together, you need to understand two new concepts: instance groups and instance templates.

Instance templates are like the recipe for your instances. They contain all of the information needed to turn on a new VM instance that looks exactly how you want it to. Higher up the chain, an instance group acts as a container for these managed instances and, given a template and some configuration parameters, is the thing that decides whether to turn on more, turn off some, or leave things alone.

Let's dive right in, first by creating an instance template. To do this, navigate to the Compute Engine section of the Cloud Console, and choose Instance Templates from the left-hand navigation. From there, click the Create Instance Template button at the top, where you'll see a screen that looks similar to the one you saw when creating a single GCE instance (figure 9.23). Start by naming your instance (for example, `first-template`), and then click the Create button at the bottom.

[← Create an instance template](#)

Describe a VM instance once and then use that template to create groups of identical instances [Learn more](#)

Name [?](#)
first-template

Machine type
 3.75 GB memory [Customize](#)

Containers
 Deploy your container images on Compute Engine VMs

Show costs for location [US](#) [▼ Details](#)

Boot disk [?](#)
 New 10 GB standard persistent disk
Image
Debian GNU/Linux 8 (jessie) [Change](#)

Identity and API access [?](#)
Service account [?](#)
 [▼](#)

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

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

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

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

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

Figure 9.23 Creating your first instance template

When that completes, you'll see a list of templates, and your newly created one will be in the list. If you click on the template, you'll be brought to a details page that should feel familiar. To use this instance template as the basis for the nodes in an instance group, click the Create Instance Group button at the top. The page that comes up is where you decide how to apply the template to create instances (figure 9.24).

[← Create a new instance group](#)

Use an instance group when configuring a load-balancing backend service or to group VM instances. [Learn more](#)

Name [?](#)

Description (Optional)

Location
Multi-zone groups span multiple zones which assures higher availability [Learn more](#)

Single-zone
 Multi-zone

Zone [?](#)

Specify port name mapping (Optional)

Group type
 Managed instance group
Managed instance group contains identical instances, created from an instance template, and supports autoscaling, autohealing, rolling updating, load balancing and more. VM instances are stateless and disks are deleted on VM deletion or recreation.
[Learn more](#)
 Unmanaged instance group
Unmanaged instance group is best for load balancing dissimilar instances, which you can add and remove arbitrarily. Autoscaling, autohealing, and rolling updating are not supported. [Learn more](#)

Instance template [?](#)

Autoscaling [?](#)

Number of instances

Autohealing
VMs in the group are recreated as needed. You can use a health check to recreate a VM if the health check finds the VM unresponsive. If you do not select a health check, VMs are recreated only when stopped. [Learn more](#)

Health check

Initial delay [?](#)
 seconds

[▼ Advanced creation options](#)

Create **Cancel**

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

Figure 9.24 Creating your first instance group

Start by naming the group itself, and then set the group to be a single-zone group in us-central1-c. (Note that you can choose a regional configuration by selecting Multi-zone and choosing the region to host the instances.) Leave the group type as a managed instance group and make sure that the instance template is set to the one you just created. Finally, change the number of instances from 1 to 3 (leaving the Autoscaling setting at Off), and then click the Create button at the bottom of the page.

It'll take a few seconds to finish, but you should eventually see your instance group fully deployed (figure 9.25).

The screenshot shows a table listing an instance group. The columns are: Name, Zone, Creation time, Instances, Template, Autoscaling, and In use by. The single row contains: first-group, us-central1-c, Jun 5, 2017, 7:27:57 AM, 3, first-template, Off. There are buttons for CREATE INSTANCE GROUP, REFRESH, EDIT, and DELETE at the top.

Instance groups						
<input type="checkbox"/>	Name	Zone	Creation time	Instances	Template	Autoscaling
<input checked="" type="checkbox"/>	first-group	us-central1-c	Jun 5, 2017, 7:27:57 AM	3	first-template	Off

Figure 9.25 Listing of your instance groups

If you click on the instance group, you'll see a list of the three instances that you created using the template from before. Now that you have an instance group, let's explore what you can do with it, starting with growing and shrinking your group.

9.3.1 **Changing the size of an instance group**

The cool part about instance groups is that you can easily change the size of the group, and GCE will do all the heavy lifting. This makes growing and shrinking the group straightforward. Because you have an instance group that you defined with a specific number of instances, you can easily shrink your instance group by deleting some of them.

For example, if you want to shrink down to a single instance, all you have to do is check the boxes next to two of the three instances, then from the top right corner choose the context menu. (It looks like three dots.) From that menu, click Delete Instance, as shown in figure 9.26, and you should see loading icons next to the two instances you selected. After a minute or so, you should see the instances disappear—the group is now made up of a single instance.

To grow the instance again, you need to click the pencil icon to reach the Edit form, and then change the number of instances back to three. After a few seconds, you should see your new instances being created again! Now let's make things even more interesting by looking at how to upgrade your instance group.

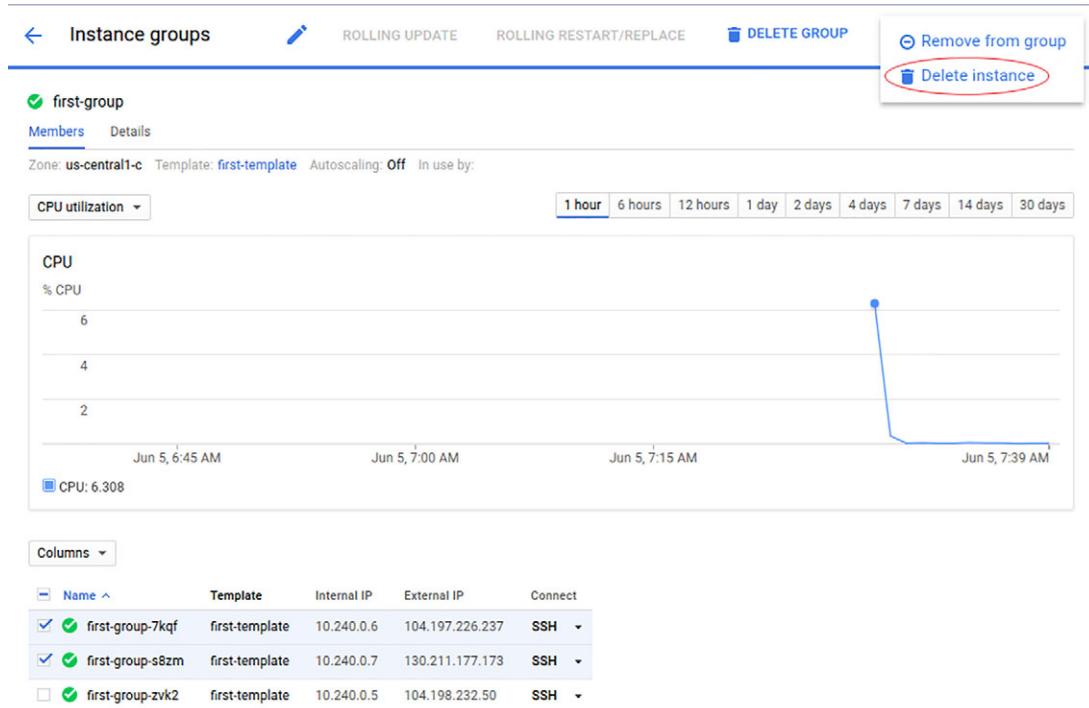


Figure 9.26 Deleting two instances from the group

9.3.2 Rolling updates

Sometimes you might have a new software package that you want to deploy across a bunch of machines, but you want to do it in stages rather than all at once. You might want to upgrade, say, half of the instances, while leaving the other half alone in case the newest version runs into any problems. Instance groups can do this using something called rolling updates, which we'll explore next.

To see how this works, start by creating a new instance template that turns on a simple Apache web server. Go back to the page to create an instance template (figure 9.27), and do the same thing you did before, but with two key changes:

- 1 Choose an Ubuntu 16.04 boot disk (instead of Debian 8).
- 2 Make sure to check the Allow HTTP Traffic box.

Also, under the Management tab of this page you'll see a section called Startup Script. In that box, install Apache using the `apt-get` command for Ubuntu:

```
#!/bin/bash
sudo apt-get install -y apache2
```

Once you've created the new template, you can use a rolling update to phase it into use. Go back to the page where you can view the details of the instance group you created

[← Create an instance template](#)

Describe a VM instance once and then use that template to create groups of identical instances [Learn more](#)

Name [?](#)
apache-template

Machine type
 3.75 GB memory [Customize](#)

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

Boot disk [?](#)
 New 10 GB standard persistent disk
Image
Ubuntu 16.04 LTS [Change](#)

Identity and API access [?](#)

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

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

Firewall [?](#)
Add tags and firewall rules to allow specific network traffic from the internet
 Allow HTTP traffic
 Allow HTTPS traffic

[Management](#) [Disks](#) [Networking](#) [SSH keys](#)

Description (Optional)

Labels [?](#) (Optional)
 [+ Add label](#)

Automation

Startup script (Optional)
You can choose to specify a startup script that will run when your instance boots up or restarts. Startup scripts can be used to install software and updates, and to ensure that services are running within the virtual machine. [Learn more](#)

```
#!/bin/bash
sudo apt-get install -y apache2
```

Metadata (Optional)
You can set custom metadata for an instance or project outside of the server-defined

Figure 9.27 Creating your new instance template

already and choose Rolling Update from the top of that page. Once there (figure 9.28), you'll migrate two of your instances to use your new Apache-enabled template.

← Update first-group

A rolling update is a gradual update of the instances in the instance group to the target configuration of templates. [Learn more](#)

Current configuration

first-template : 100% (3 instances)

New configuration
Select 1-2 template(s), using the target size to specify percentage or number of instances per template

Template	Target size
first-template	Currently 1 out of 3 instances
apache-template	2 instance(s)

+ Add item

Update mode
 Proactive
Starts the update immediately
 Opportunistic
Only updates when new instances are created or the instance group is resized. [Learn more](#)

Maximum surge
Maximum number (or percentage) of temporary instances to add while updating. [Learn more](#)
1 instance(s)

Maximum unavailable
Maximum number (or percentage) of instances that can be offline at the same time while updating. [Learn more](#)
1 instance(s) out of 3 instances

Minimum wait time
Time to wait between consecutive instance updates. [Learn more](#)
30 s

You are deploying template "first-template" to 1 instance and template "apache-template" to 2 instances in instance group "first-group".
1 instance will be taken offline at a time and 1 instance will be temporarily added to support the update.
The managed instance group will wait 30 seconds after an updated instance is healthy before considering the instance as ready and proceeding to the next instance.

Update **Cancel**

Figure 9.28 Configuring your rolling update

First, click Add Item to create a new row, and then choose the new Apache template (called apache-template in the image) from the target template. Next, set the Target Size to two instances. Before you click Update, let's look at a few of the other parameters here. First, you can choose when you want the update to happen. GCE can start it either immediately (Proactive), where it begins turning off the currently running matches, or whenever there's a good opportunity (Opportunistic), such as an outage of a machine. For this example, you'll use the proactive mode. When handling an upgrade, old instances are only turned off after the replacement is ready. As a result, you'll go above your maximum instance count during a rolling upgrade. To avoid turning on twice the number of instances you have, you can choose to set how big the surge over the limit will be. For this example, you'll use one instance.

Next, you can control how much work should happen concurrently. Or you can limit how many instances (or what percentage of instances) can be down at a given time. For this example, only upgrade one at a time, but if you have a larger cluster, you may want to do this in a larger group (for example, 10 at a time).

Finally, you can configure how long to wait between updates. You may have some extra configuration happening on your instance, and it may take a few minutes to get up and running. If you have that scenario, you may want to set the wait time to a safe amount that will allow your newly created instance to boot up and become active in the cluster. For the Apache example, you can use 30 seconds to make sure Apache installed and is running.

After you click Update, you should see a progression of your instances turning off and on. The end result will be that two of the instances use the Apache template, and one of them has been left alone (figure 9.29).

Columns ▾					
	Name ^	Template	Internal IP	External IP	Connect
<input type="checkbox"/>	<input checked="" type="checkbox"/> first-group-nsjg	apache-template	10.240.0.6	130.211.151.73 ↗	SSH ▾
<input type="checkbox"/>	<input checked="" type="checkbox"/> first-group-qjch	apache-template	10.240.0.9	104.154.247.150 ↗	SSH ▾
<input type="checkbox"/>	<input checked="" type="checkbox"/> first-group-zvk2	first-template	10.240.0.5	104.198.232.50	SSH ▾

Figure 9.29 Your instances after the rolling update has completed

To do a complete update, go through the same process again, this time setting the end state of 100% of instances using the Apache template. Now that you've seen a rolling update, let's explore how autoscaling works.

9.3.3 Autoscaling

Autoscaling builds on the principles you saw with a rolling update but looks at various measures of health to decide when to replace an instance or grow and shrink the cluster as a whole. For example, if a single instance becomes unresponsive, the instance group can mark it as dead and replace it with a new one. Additionally, if instances become overloaded (for example, the CPU usage goes above a certain percentage), the instance group can increase the size of the pool to accommodate the unexpected load on the system. Conversely, if all instances are far under the CPU limit for a set amount of time, the instance group can retire some of the instances to remove unnecessary cost.

The underlying idea behind this isn't all that complicated, but configuring the instance groups and templates to do this kind of thing is a bit trickier. Currently, the instance group you have is configured to always be three instances. Start by changing this to scale between 1 and 10 instances whenever CPU usage goes above a certain threshold.

Start by looking at the instance group you created and click the Edit button (which looks like a pencil icon). On that page (figure 9.30), you'll notice an Autoscaling drop-down that you previously set to Off. When you flip this to On, you're able to choose how exactly you want to scale the group, but to start, use CPU usage.

Leave the Autoscale Based On option to CPU Usage, but change the Target CPU Usage to 50%. Next, make sure the Minimum and Maximum instances are set to 1 and 10, respectively. Finally, let's look at the Cool-Down Period setting. The cool-down period is the minimum amount of time that the group should wait before deciding that an instance is above the threshold. In your configuration, if you have a spike of CPU usage that lasts less than the cool-down period, it won't trigger a new machine being created. In a real-life scenario, it probably makes sense to leave this set to at least 60 seconds, but for this example, you can make the cool-down period a bit shorter so you'll see changes to the group more quickly.

After you click Save and wait a few seconds, you should notice that two of your instances have disappeared! The instance group noticed that CPU usage was low and removed the instances that weren't needed anymore.

You can try making things go the other way now by making the remaining instance very busy so the group sees the CPU usage jump. SSH into the remaining instance and run the command shown in the following snippet, which uses the `stress` library to force the CPU to do some extra work:

```
$ sudo apt-get install stress  
$ stress -c 1
```

While that's running, keep an eye on the CPU graph in the Cloud Console. You should notice it starting to increase rapidly, and as it gets higher and stays that way for

[←](#) Instance groups [EDIT GROUP](#) [ROLLING UPDATES](#)

Edit first-group

Zone
us-central1-c

Specify port name mapping (Optional)

Instance template [?](#)

⚠ The template applies only to newly created VMs. After saving these settings, use [this gcloud command](#) to recreate existing VMs using the new template

Autoscaling [?](#)

Autoscale based on [?](#)
For best results read [Configuring autoscaling instance groups](#)

Target CPU usage [?](#)
Scaling dynamically creates or deletes VMs to meet the group target. [Learn more](#)
 %

Minimum number of instances [?](#)

Maximum number of instances [?](#)

Cool-down period [?](#)
 seconds

Autohealing
VMs in the group are recreated as needed. You can use a health check to recreate a VM if the health check finds the VM unresponsive. If you do not select a health check, VMs are recreated only when stopped. [Learn more](#)

Health check

Initial delay [?](#)
 seconds

[Save](#) [Cancel](#)

Figure 9.30 Configuring autoscaling

a few seconds, you should see at least one completely new instance get created! See figure 9.31.

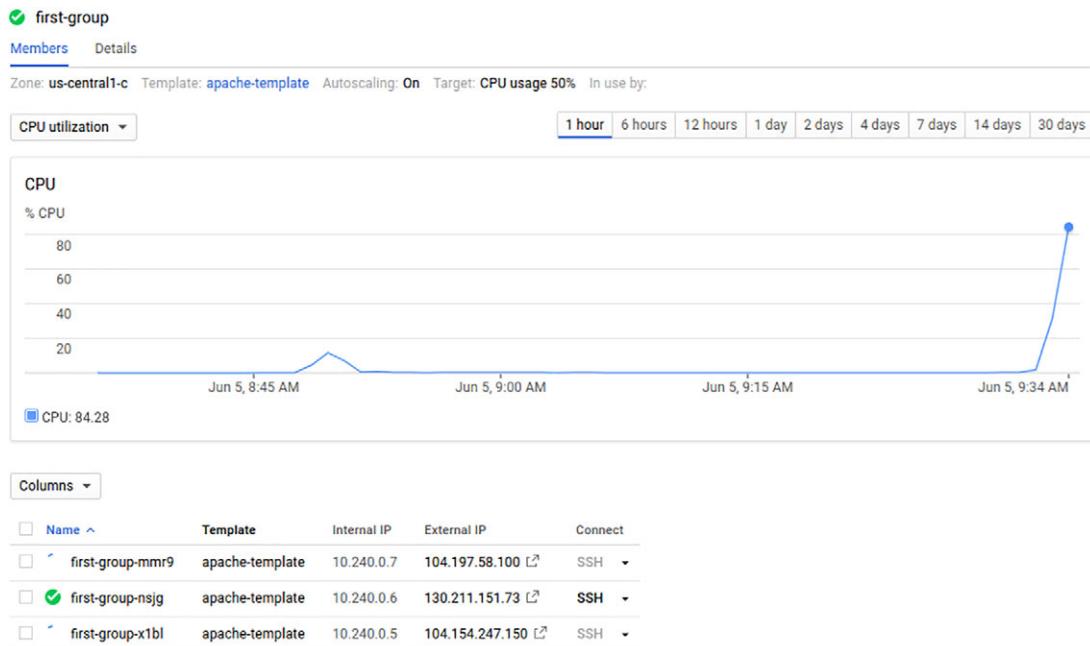


Figure 9.31 CPU usage making new machines appear

If you leave the stress tool running, even more instances will be created to handle the (apparent) large load on that one instance. To close things up, press **CTRL+C** to kill the stress tool. In a few minutes, you should see the group shrink back down to one instance. If you look at the Autoscaled size graph (figure 9.32), you can see how your cluster has grown and shrunk based on the CPU usage.

As you can see, once you teach GCE how to turn on instances configured the way you need to run your application, it can handle the rest based on how busy the VMs become. In addition to allowing you to automatically scale your compute capacity, using these templates opens the door to a new form of computing that can significantly reduce your costs, using preemptible VMs. Let's take a look at how that works and when you should consider it as a viable option for your workloads.

9.4 Ephemeral computing with preemptible VMs

So far, all of the machines I've talked about have been virtual, but also relatively persistent. Once you create the VM, it continues to run until you tell it to stop. But GCE also has another type of machine that's *ephemeral*, meaning it might disappear at any moment and never lives for more than 24 hours. In short, Google gives you a discount

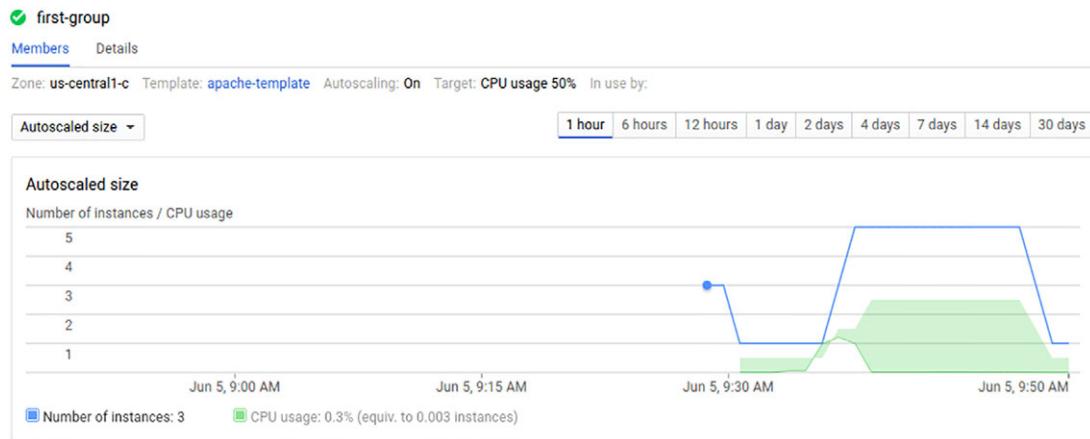


Figure 9.32 Autoscaled size graph

on the regular per-hour price in exchange for being able to reclaim the machine at any time and resell it if someone else shows up willing to pay the full price.

9.4.1 Why use preemptible machines?

So far, the most common use for preemptible machines is large-batch workloads, where you have lots of worker machines that process a small piece of an overall job. The reason is simple: preemptible machines are cheap, and if one of those machines is terminated without notice, the job might complete a bit slower but won't be canceled altogether.

Imagine you have a job that you want to split into four chunks. You could use a regular VM to orchestrate the process, and then four preemptible VMs to do the work. By doing this, any one of the chunks could be canceled at any point, but you could retry that chunk of work on another preemptible VM. By doing this, you can use cheaper computing capacity to work on the job. In exchange for your savings, the workers (VMs) may get killed and need replacement (figure 9.33).

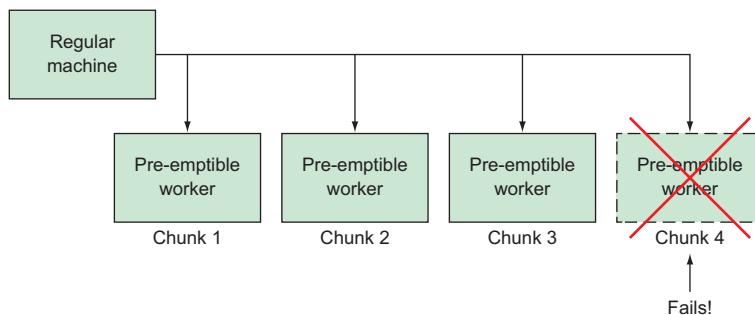


Figure 9.33 Four virtual workers, with one getting killed

Now that I've discussed why preemptible machines exist and what they're good for, the next obvious question is, how does it all work? To understand that, let's explore the two new things you have to think about: turning on machines as preemptible and handling requests for machines to be terminated.

9.4.2 Turning on preemptible VMs

Creating a preemptible VM is simple. When you're creating your VM (or your instance template), in the Advanced section (the link that says "Management, disks, networking, SSH keys") (figure 9.34), you'll notice under Availability Policy that you can set Preemptibility. Changing this from Off to On makes your VM preemptible.

Availability policy

Preemptibility

A preemptible VM costs much less, but lasts only 24 hours. It can be terminated sooner due to system demands. [Learn more](#)

On

Automatic restart

Compute Engine can automatically restart VM instances if they are terminated for non-user-initiated reasons (maintenance event, hardware failure, software failure, etc.)

Off

On host maintenance

When Compute Engine performs periodic infrastructure maintenance it can migrate your VM instances to other hardware without downtime

Terminate VM instance

Figure 9.34 The dialog box where you can make a VM preemptible

As you'd guess, it comes with the side effect that both automatic restarts and live migration during host maintenance are disabled. This shouldn't be a problem, because when designing for preemptible machines, you're already expecting that the machine can disappear at any given moment. Now that you understand how to create a preemptible machine, let's jump to the end of its life and see how to handle the inevitable terminations from GCE.

9.4.3 Handling terminations

Preemptible machines can be terminated anytime (and will definitely be terminated within 24 hours), and understanding how to gracefully handle these terminations is critically important. Luckily, GCE doesn't sneak up on you with these, but instead gives you a reasonable notification window to let you know that it'll terminate your VM. You can listen for that notification and finish up any pending work before the VM is gone.

The easiest way to listen for it is by setting a shutdown script—sort of the opposite of what you did with your instance templates' startup scripts. Once the termination is triggered, GCE gives the VM 30 seconds to finish up, and then sends a firm termination signal (the equivalent of pressing your machine's power button) and switches the machine to terminated. This gives your shutdown script 30 seconds to do its work. After 30 seconds, the plug gets pulled.

To set a shutdown script, you can use the metadata section of the instance (or instance template) with a key appropriately called `shutdown-script` (figure 9.35).

Metadata (Optional)

You can set custom metadata for an instance or project outside of the server-defined metadata. This is useful for passing in arbitrary values to your project or instance that can be queried by your code on the instance. [Learn more](#)

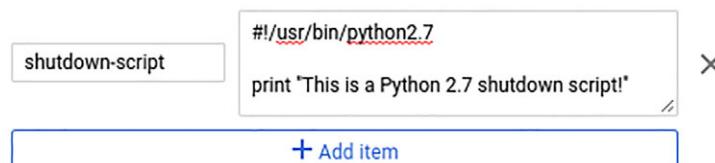


Figure 9.35 Setting a shutdown script when creating a VM

If you have a shutdown script stored remotely on Cloud Storage, you can link to it using metadata with the key `shutdown-script-url` and a URL starting with `gs://` (figure 9.36).

Metadata (Optional)

You can set custom metadata for an instance or project outside of the server-defined metadata. This is useful for passing in arbitrary values to your project or instance that can be queried by your code on the instance. [Learn more](#)

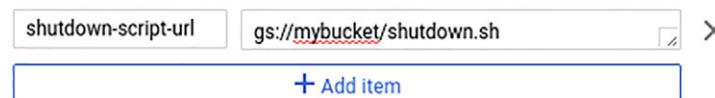


Figure 9.36 Setting a shutdown script URL when creating a VM

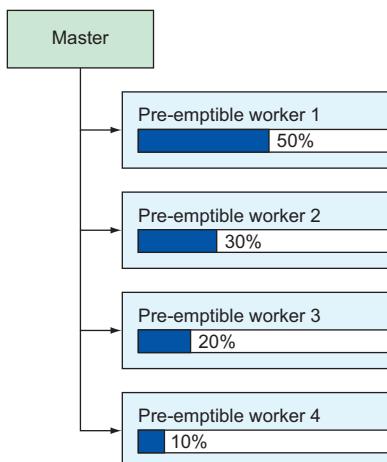
Because termination of a preemptible machine triggers a regular shutdown script, you can test your scripts by stopping the machine. How often will this termination happen? Let's look at how GCE decides which VMs to terminate first.

9.4.4 Preemption selection

When selecting a VM to terminate, GCE chooses the youngest one, which might seem counterintuitive. To see why it does this, let's think about what the best case is for choosing to terminate a VM.

Imagine you have a job where each VM needs to download a large (5 GB) file. If you boot your VMs in order and they get right to work downloading the file, at any

given time, the first VM will have more download progress than the second, third, and fourth. Now imagine that Google needs to terminate one of these VMs. Which one is most convenient to terminate? Which VM causes the least amount of wasted work if it has to be terminated and start over? Obviously, it's the one that started last and has downloaded the least amount of data (figure 9.37).



Pre-emptible worker 4 has the least download progress, so the least work will be wasted if it's terminated.

Figure 9.37 Selecting which machine to terminate

Because GCE will terminate the youngest machine, one concern is *thrashing*, where machines continually get terminated and never make any progress. Although this is definitely possible, GCE will distribute terminations evenly at a global level. If it wants to reclaim 100 VMs, it'll take those 100 from lots of customers rather than a single one. As a result, you should only see repeated terminations during extreme circumstances. For the rare cases where thrashing does happen, remember that GCE doesn't charge for VMs that are forcibly terminated within their first 10 minutes. With that last aspect of ephemeral computing covered, let's explore how to balance requests across multiple machines using a load balancer.

9.5 Load balancing

Load balancing is a relatively old topic, so if you've run any sort of sizable web application, you're probably at least familiar with the concept. The underlying principle is that sometimes the overall traffic that you need to handle across your entire system is far too much for a single machine. As a result, instead of making your machines bigger and bigger to handle the traffic, you use more machines and rely on a load balancer to split (balance) the traffic (load) across the available resources (figure 9.38).

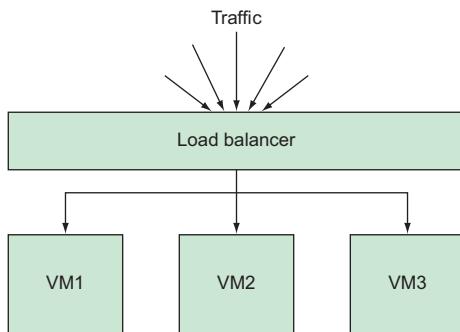


Figure 9.38 Load balancers spread traffic across all of the available resources.

Traditionally, if you needed a load balancer (and you weren't handling many millions of hits per second), you'd turn on a VM and install some load balancing software, which could be anything from HAProxy to Squid, or even NGINX. But because this is such a common practice, Google Cloud Platform offers a fully managed load balancer that does all of the things that traditional software load balancers can do.

Because load balancers take incoming requests at the front and spread them across some set of machines at the back, any load balancer will have both frontend and backend configurations. These configurations decide how the load balancer will listen for new requests and where it will send those requests as they come in. These configurations can range anywhere from super-simple to extremely complex. A simple example will help you to understand better how this all works.

Imagine you have a calculator web application that allows users to type a calculation into a box—say, $2+2$ —and then the application prints out the correct answer—4, in this case. This calculator isn't that useful, but the important thing to note is that it doesn't need to store anything, so you say it's stateless.

Now imagine that for some reason, the calculator has gotten so popular that the single VM handling calculations is overwhelmed by traffic. In this scenario, you'd create a second VM running the exact same software as the first; then you'd create a load balancer to spread the traffic evenly across the two machines (figure 9.39). The load

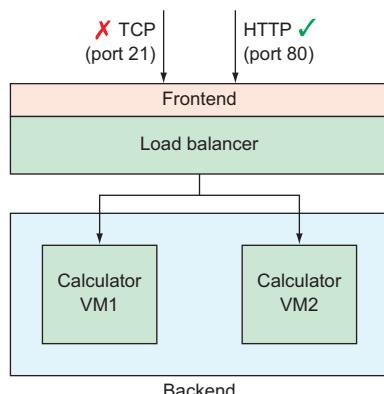


Figure 9.39 A calculator application using a load balancer

balancer frontend would listen for HTTP traffic and forward all requests to the backend, which is made up of the two VMs that handle calculations. Once the calculating is done, the VMs respond with the result, and the load balancer forwards the response back over the connection used to make the request.

It's important to note that the VMs have no idea a load balancer is involved, because from their perspective, requests come in and responses go out, so the load balancer looks like any other client making requests. And the most convenient part of all of this is that as traffic grows even more, you can turn on another VM and configure the load balancer backend to also send requests to the new VM. If you wanted to automate this even further, you could configure the load balancer to use an autoscaling instance group for its backend, so you wouldn't have to worry about adding new capacity and registering it with the load balancer.

Let's run through the process of setting up a load balancer and splitting traffic across several VMs. To do this, you'll repurpose the VM instance group you learned about in section 9.4 as the backend of your load balancer. To create the load balancer, choose Network Services from the left-side navigation in the Cloud Console, and choose Load Balancing after that. On that page, you should see a prompt that allows you to create a new load balancer by clicking a button.

When you click to create a new load balancer, you'll see a few choices of the types of load balancing you can do. For this exercise, you'll use HTTP(S) load balancing because you're trying to balance HTTP requests across your web application.

Click Start Configuration and you'll see a new page that has a place to choose a name (use `first-load-balancer`) and three steps toward configuring the new load balancer: BackendConfiguration, Host and Path Rules, and FrontendConfiguration. Because you want to configure the load balancer to take HTTP requests and send them to your instance group as a backend, start by setting your backend configuration.

9.5.1 **Backend configuration**

The first thing you need to do is create what's called a backend service. This service represents a collection of backends (typical VMs running in GCE) that currently refer only to instance groups. To do this, click the drop-down that says Create or Select Backend Services & Backend Buckets. From there, choose Backend Services > Create a Backend Service, which opens a new form where you can configure your new service (figure 9.40).

NOTE You may be wondering why you have this extra level of indirection and why you have to create a service to contain a single instance group. The simple answer is that even though you only have one instance group now, you may want to add more groups later behind the load balancer.

Backend services allow the load balancer to always point at one thing so you can add backends (instance groups) to, and remove them from, the service.

Create backend service

Name [?](#)

[Description](#)

Protocol: HTTP Named port: http Timeout: 30 seconds [Edit](#)

Backends

New backend ✖

Instance group [?](#)

Port numbers [?](#)

Balancing mode [?](#)
 Utilization
 Rate

Maximum CPU utilization [?](#)
 %

Maximum RPS (Optional) [?](#)
Max total RPS. Leave blank for unlimited RPS

Capacity [?](#)
 %

[Less](#)

[+ Add backend](#)

Health check [?](#)

port: 80, timeout: 5s, check interval: 5s, unhealthy threshold: 2 attempts

Session affinity [?](#)

Affinity cookie TTL [?](#)

seconds

[Advanced configurations](#)

Cloud CDN [?](#)

 Enable Cloud CDN

Figure 9.40 Creating a new backend service

Continue your naming pattern and call this `first-backend-service`, and then choose your `first-group` as the backend. You'll notice quite a few extra options that should feel similar to the autoscaling configuration of the instance group. Although they're definitely similar, they have different purposes.

In an instance group, you used things like target CPU usage to tell GCE when it should turn on more instances. In a load balancer, these things describe when the system should consider the backend to be over capacity. If the backend as a whole goes over these limits, the load balancer will consider it to be unhealthy and stop sending requests to it. As a result, you should choose these targets carefully to make sure you don't have the load balancer unnecessarily returning errors that say the system is over capacity when it isn't.

Leave these settings at the default values for now. Next, you'll notice that you need to set a *health check* before you can save your backend service. Let's dig into what these are and how they work.

CREATING A HEALTH CHECK

Now that you've almost finished configuring your backend service, you'll need to create something called a health check, which you can do directly from the drop-down menu on the New Backend form. Health checks are similar to the measurements that are taken to decide whether you need more VMs in an instance group, but are less about the metrics of the virtual machine (such as CPU usage) and instead focus on asking the application itself if everything is OK. They're more like the nurse asking you if you feel OK, whereas the other checks about things like CPU utilization are like the nurse checking your temperature.

These health checks can be a simple static response page to show that the web server is up and running, or something more advanced, like a test of whether the database connection is working properly. You can create a simple TCP check to see that port 80 is indeed open (figure 9.41). Name it `tcp-80`. You'll leave the other settings (such as how long to wait between checks, how long to wait before timing out, and more) as they are.

When you create the health check, your backend service will be ready. Click Create to see the summary of your backend service, and you can move on to the host and path rules.

Name 

Description (Optional)

Protocol  Port 

Proxy protocol 

Request (Optional)  Response (Optional) 

Health criteria

Define how health is determined: how often to check, how long to wait for a response, and how many successful or failed attempts are decisive

Check interval  Timeout  seconds seconds

Healthy threshold  Unhealthy threshold  consecutive successes consecutive failures

Figure 9.41 Creating a new health check to test port 80

9.5.2 Host and path rules

Although you only created a single backend this time, nothing's stopping you from creating multiple different backends and using them all together. To do that, you'd rely on a set of rules to decide how different incoming requests would be routed to the various backends. For example, you might have a situation where a specific backend had to handle the more complicated requests. In that case, you'd create a separate backend for them, route the relevant requests to that backend, and route all other requests elsewhere.

In the current example, you have just one backend and want all requests to be routed directly to it, so you don't need to do anything in this section. Instead, you can move on to configuring the frontend.

9.5.3 Frontend configuration

For this example, you want to handle normal HTTP traffic, which is the default setting (figure 9.42). The only potential issue is that the IP address for your load balancer may change from time to time. (That's what it means when it says the IP address is Ephemeral). If you were setting this service up to have a domain name like mycalculator.com, you'd create a new static IP address (by choosing Create IP Address from the drop-down); that IP address would always be the same, so you could add it to a DNS entry. For now (and because this is a demonstration), you'll stick with an ephemeral IP for your load balancer, and you can leave the name field blank.

Frontend configuration

Specify an IP address, port and protocol. This IP address is the frontend IP for your clients requests. For SSL, a certificate must also be assigned.

The screenshot shows a modal dialog titled "New Frontend IP and port". It has a blue header bar with a trash icon and a close button. The main area contains fields for "Name (Optional)" (containing "lowercase, no spaces"), "Add a description", "Protocol" (set to "HTTP"), "IP version" (set to "IPv4") and "IP address" (set to "Ephemeral"), and "Port" (set to "80"). At the bottom are "Done" and "Cancel" buttons, and a large blue "Add Frontend IP and port" button at the very bottom.

Figure 9.42 Your simple frontend configuration

As you can see, it's possible to create multiple frontend configurations if you want to listen on multiple ports or multiple protocols. For this demonstration, you'll stick to boring old HTTP on port 80. Click Done. Now you can jump to the final step, where you can review all the configurations you've set up to verify that they're correct.

9.5.4 Reviewing the configuration

As you can see in figure 9.43, you have an instance group called `first-group`, which has no host- or path-specific rules and is exposed on an ephemeral IP address on port 80. You also have a health check called `tcp-80`, which will be used to figure out which of the instances in the group are able to handle requests.

Review and finalize

Backend

Backend services

1. first-backend-service

Endpoint protocol: **HTTP** Named port: **http** Timeout: **30 seconds** Health check: **tcp-80** Session affinity: **None**

Cloud CDN: **disabled**

▼ Advanced configurations

Instance group	Zone	Autoscaling	Balancing mode	Capacity
first-group	us-central1-c	Off	Max CPU: 80%	100%

Host and path rules

Hosts	Paths	Backend
All unmatched (default)	All unmatched (default)	first-backend-service

Frontend

Protocol	IP:Port	Certificate
HTTP	EPHEMERAL:80	—

Figure 9.43 A summary of your load balancer configuration

Clicking Create (not shown) will start the process of creating the load balancer and the health checks, assigning an ephemeral IP, and getting ready for your load balancer to start receiving requests. Figure 9.44 shows the result.

After a couple of minutes (while the health checks determine that the backend is ready), visiting the address of your load balancer in a browser should show you the Apache 2 default page. Seeing this page will show you that the request was routed to one of your VMs in the instance group.

Now imagine tons of people sending requests to the load balancer. As the number of requests goes up and the CPU usage of a given VM increases, the autoscaling

The screenshot shows the Google Cloud Platform Load Balancing interface. At the top, there are buttons for 'CREATE LOAD BALANCER' and 'REFRESH'. Below this, a navigation bar has 'Load balancers' selected, with other options 'Backends' and 'Frontends'. The main area displays a 'Load balancer' card for 'first-load-balancer'. The card includes tabs for 'Details' (selected), 'Monitoring', and 'Caching'. Under 'Frontend', it shows an 'IP:Port' entry of 'HTTP 35.186.226.157:80'. Under 'Host and path rules', there's a table with columns 'Hosts', 'Paths', and 'Backend', showing a single row for 'All unmatched (default)'. Under 'Backend', it lists 'Backend services' with one entry: '1. first-backend-service'. This service has an endpoint protocol of 'HTTP', a named port of 'http', a timeout of '30 seconds', a health check of 'tcp-80', and session affinity set to 'None'. It also mentions 'Cloud CDN: disabled' and has an 'Advanced configurations' section. A table for 'Instance group' shows one entry: 'first-group' in 'us-central1-c' zone, healthy count 0 / 0, target CPU usage 50%, max CPU 80%, and capacity 100%.

Figure 9.44 Your newly created load balancer

instance group will turn on more VMs, as you learned earlier. The big difference is that because you're routing all requests through your load balancer, as more requests come in, the load balancer will automatically balance them across all of the VMs that the instance group turned on. You now have a truly autoscaling system that will handle all the requests you could possibly throw at it, and it will grow and shrink and distribute the request load automatically!

Now that you have the functionality working, it might make sense to look at ways to optimize this configuration. One common low-hanging fruit you can pick off is figur-

ing out how to avoid duplicating work by caching results whenever possible, so let's see how you might use that principle to make your system more efficient.

9.6 Cloud CDN

In any application, it's likely that lots of identical requests will go to the servers running the application. Sometimes identical requests also yield identical responses. Although this scenario is most common with static content like images, it can apply to dynamic requests as well. To avoid duplicating effort, it turns out that Google Cloud Platform has something called Google Cloud CDN that can automatically cache responses from backend services and is designed to work with GCE and the load balancer that you just created.

Cloud CDN sits between the load balancer and the various people making requests to the service and attempts to short-circuit requests. As you can see in figure 9.45, a

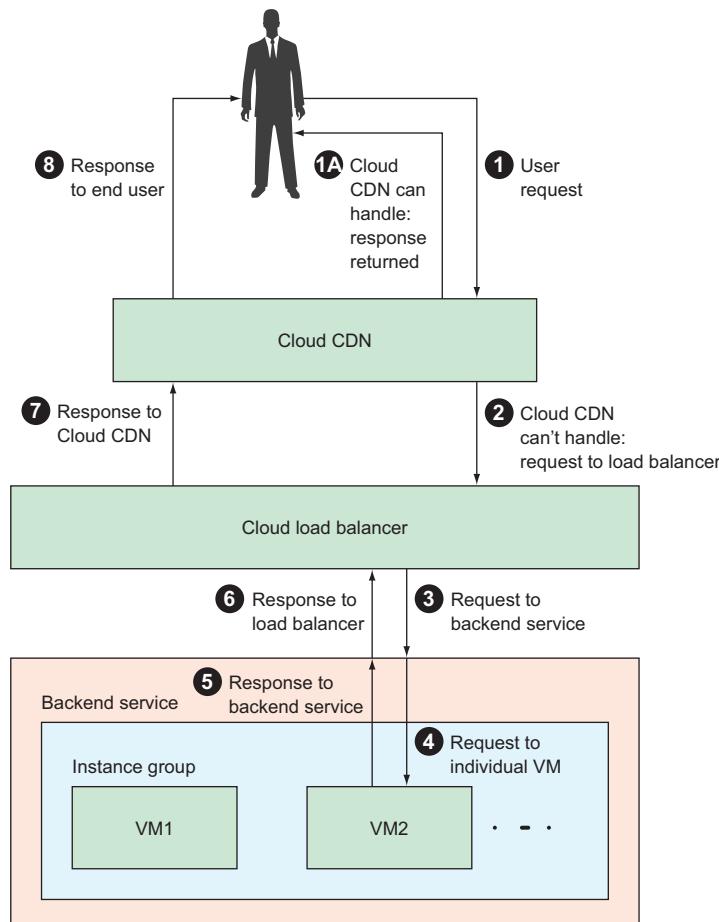


Figure 9.45 The flow of a request with Cloud CDN enabled

request starts from a user (1), and if Cloud CDN can handle the request, it returns a response immediately (1A). The load balancer, backend service, instance group, and VM instances never even see the request. If Cloud CDN doesn't handle the request, the request follows the traditional path of visiting the load balancer (2), then the backend service (3), which routes the request to an individual VM (4). Then the response flows back over the same path to the load balancer (5, 6). After the load balancer returns the response, instead of going directly to the end user, as you saw previously, the response flows through Cloud CDN (7) and from there back to the end user (8). This allows Cloud CDN to inspect the response and determine if it can be cached so that Cloud CDN can handle future identical requests via the short-circuit route (1 and 1A).

In addition to this sequence, if a request appears that it *could be* cached but the given Cloud CDN endpoint doesn't have a response to send back, it can ask other Cloud CDN endpoints if they've handled the same request before and have a response (figure 9.46). If the response is available somewhere else in Cloud CDN, the local instance can return that value as well as storing it locally for the future.

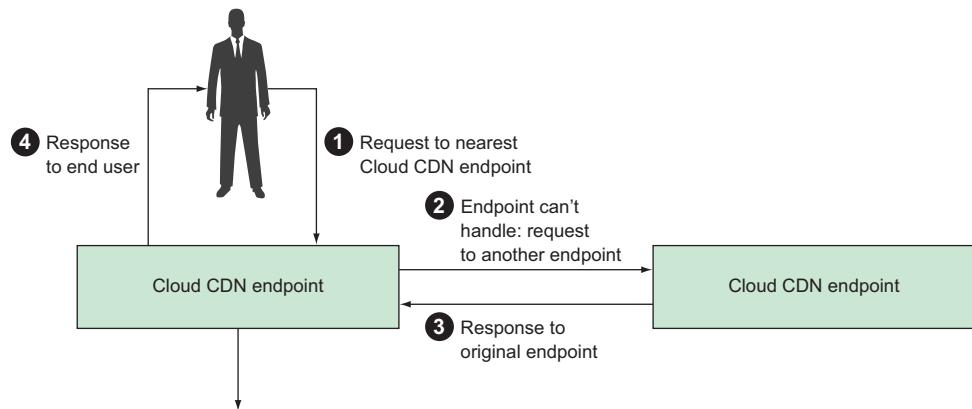


Figure 9.46 Cloud CDN looking to other caches for a response

As you can see, if a request can be cached but happened to be cached elsewhere, the request will flow to the nearest Cloud CDN endpoint (1) and over to another Cloud CDN (2) that has a response. The response will then flow back to the original endpoint (3), where it's stored locally and ultimately returned back to the user (4). Let's look at how you can enable Cloud CDN for the load balancer you created previously.

9.6.1 Enabling Cloud CDN

In the left-side navigation of the Cloud Console, choose Cloud CDN from the Network Services section. Once there, you'll see a form prompting you to add a new origin for Cloud CDN, and you'll click Add Origin.

Choosing the load balancer you created previously (figure 9.47) will populate a list of backend services that Cloud CDN can cache.

The screenshot shows a 'Network services' sidebar with 'Load balancing', 'Cloud DNS', and 'Cloud CDN'. The 'Cloud CDN' item is selected and highlighted with a blue background. To the right, the 'Add origin to Cloud CDN' dialog is open. It has a header with a back arrow and the title. Below the header is a descriptive text: 'Choose a load balancer as origin. Responses from the origin will be cached by Cloud CDN.' A section titled 'Origin' contains a dropdown menu labeled 'Select origin' with a 'Filter' input field. Below the dropdown is a link 'Create a load balancer...'. A list titled 'Load balancer' shows one entry: 'first-load-balancer'. The entire dialog is enclosed in a light gray border.

Figure 9.47 Choosing the correct load balancer to cache using Cloud CDN

Here you also can add some extra configuration to the specific backend by clicking Configure (not shown). This allows you to customize the way pages are cached by saying, for example, that pages served over HTTP should be cached separately from pages served over HTTPS. By clicking Add, you enable Cloud CDN on the load balancer for the selected backend services (figure 9.48).

The screenshot shows the same 'Add origin to Cloud CDN' dialog as in figure 9.47, but with additional configuration options. The 'Origin' dropdown now contains the value 'first-load-balancer'. Below it, a section titled 'Backend services' is expanded. It contains the text 'Cloud CDN will cache responses from the checked backend services'. Underneath are two checkboxes: 'Name' (which is checked) and 'Cache key'. Below these checkboxes is a table row with three columns: 'first-backend-service' (with a checked checkbox), 'Default' (with a radio button), and 'Configure' (with a link). At the bottom of the dialog are two buttons: 'Add' (in a blue box) and 'Cancel'.

Figure 9.48 Ensuring the right backends are selected to be cached

The screenshot shows the Cloud CDN interface. On the left, a sidebar lists 'Network services' with options for 'Load balancing', 'Cloud DNS', and 'Cloud CDN'. The 'Cloud CDN' option is selected and highlighted with a blue background. The main area is titled 'Cloud CDN' with 'ADD ORIGIN' and 'REFRESH' buttons. Below this, there's a table with columns for 'Origin name', 'Backends', and 'Cache hit ratio'. A row is shown for 'first-load-balancer' with 'first-backend-service (Enabled)' listed under 'Backends' and 'n/a' under 'Cache hit ratio'. There's also a 'More' button represented by three dots.

Figure 9.49 A listing of load balancers that Cloud CDN is actively caching

At this point, you can see in the list that Cloud CDN and a specific set of backends are caching your load balancer (`first-load-balancer`) (figure 9.49).

You also can see that Cloud CDN is enabled by looking at the details of your load balancer and noting the Cloud CDN: Enabled annotation listed under the backend service in figure 9.50.

The screenshot shows the 'Load balancer' details page. At the top, there are buttons for 'CREATE LOAD BALANCER' and 'REFRESH'. Below this, a navigation bar has 'Load balancers' selected. The main content area shows a 'first-load-balancer' entry. It includes tabs for 'Details', 'Monitoring', and 'Caching'. Under 'Frontend', it shows an 'HTTP' entry with port 35.186.226.157:80. Under 'Host and path rules', it shows 'All unmatched (default)' for both 'Hosts' and 'Paths' pointing to 'first-backend-service'. Under 'Backend', it shows a single entry for '1. first-backend-service' with details: Endpoint protocol: HTTP, Named port: http, Timeout: 30 seconds, Health check: tcp-80, Session affinity: None, and Cloud CDN: enabled. It also shows an 'Advanced configurations' section and a table for 'Instance group' with one entry: 'first-group' in 'us-central1-c' zone, healthy status 0 / 1, target CPU usage 50%, max CPU 80%, and capacity 100%.

Figure 9.50 The load balancer showing that Cloud CDN is enabled

9.6.2 Cache control

How does Cloud CDN decide what pages it can and can't cache? By default, Cloud CDN will attempt to cache all pages that are *allowed*. This definition mostly follows IETF standards (such as RFC-7234), meaning that the rules are what you'd expect if you're familiar with HTTP caching in general. For example, the following all must be true for Cloud CDN to consider a response to be cacheable:

- Cloud CDN must be enabled.
- The request uses the GET HTTP method.
- The response code was "successful" (for example, 200, 203, 300).
- The response has a defined content length or transfer encoding (specified in the standard HTTP headers).

In addition to these rules, the response also must explicitly state its caching preferences using the Cache-Control header (for example, set it to public) and must explicitly state an expiration using either a Cache-Control: max-age header or an Expires header.

Furthermore, Cloud CDN will actively not cache certain responses if they match other criteria, such as

- The response has a Set-Cookie header.
- The response size is greater than 10 MB.
- The request or response has a Cache-Control header indicating it shouldn't be cached (for example, set to no-store).

In addition, as I noted earlier, you can configure whether you want to distinguish between URLs based on the scheme (for example, HTTP versus HTTPS), query string (for example, stuff after the ? in a URL), and more to get fine-grained control over how different responses are cached. The moral of the story here is that Cloud CDN will follow the rules that most browsers and load-balancing proxy servers follow with regard to caching but will do the caching work in a location much closer to the end user than your VMs typically will be.

Finally, at times you may have cached something and need to forcibly uncache it. You want the request to go to the backend service rather than having the cache handle it. This is a common scenario when, for example, you deploy new static files, such as an updated style.css file, and don't want to wait for the content to expire from the cache.

To do this, you can use the Cloud Console and click the Cache Invalidation tab. Here (figure 9.51) you can enter a pattern to match against (such as /styles/*.css), and all matching cache keys will be evicted. On a subsequent request for these files, they'll be fetched first from the backend service and then cached as usual.

At this point, you should have a good grasp of most of the things that GCE can do. With that in mind, it's time to look at how much all of this costs to use. As you might guess, pricing can be a bit complicated, given the various ways you can use GCE.

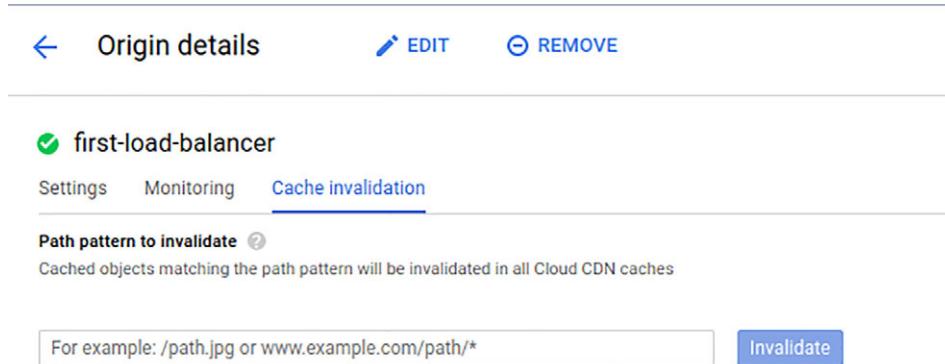


Figure 9.51 Invalidating a particular cached URL

NOTE Before we jump into looking at how much everything costs, now is a great time to turn off any resources you created while reading this chapter so you don't end up getting charged unnecessarily!

9.7 *Understanding pricing*

The basic features of GCE have straightforward prices, whereas some of the more advanced features can get complicated, and even more complicated when you consider an important discount available for sustained use. I'll start by talking about the simple parts, then move into the more complicated aspects of GCE pricing.

You need to consider three factors for pricing with GCE:

- 1 Computing capacity using CPUs and memory
- 2 Storage using persistent disks
- 3 Network traffic leaving Google Cloud

9.7.1 *Computing capacity*

The most common way of using GCE is with a predefined instance type, such as n1-standard-1, which you used in chapter 1. By turning on an instance of a particular predefined type, you're charged a specific amount every hour for the use of the computing capacity. That capacity is a set amount of CPU time, which is measured in vCPUs (a virtual CPU measurement), and memory, which is measured in GB. Each predefined type has a specific number of vCPUs, a specific amount of memory, and a fixed hourly cost. Table 9.2 shows a brief summary of common instance types and how much they cost on an hourly and monthly basis in the us-central1 region. As expected, more compute power and memory mean more cost.

If one of these instance types doesn't quite fit your needs—for example, if you need a lot of memory but little CPU (or vice-versa)—other predefined machine types are available. They have a pricing structure similar to table 9.2.

Table 9.2 Cost and details for some common instance types

Instance type	vCPUs	Memory	Hourly cost	Monthly cost (approximate)
n1-standard-1	1	3.75 GB	\$0.0475	\$25
n1-standard-2	2	7.5 GB	\$0.0950	\$50
n1-standard-8	8	30 GB	\$0.3800	\$200
n1-standard-16	16	60 GB	\$0.7600	\$400
n1-standard-64	64	240 GB	\$3.0400	\$1,500

For the truly unusual scenarios where no predefined types fit, you can design your own custom machine profile. For example, imagine you want to store a huge amount of data in memory but don't need the machine to do anything other than act as a cache. In that case, you might want to have a lot of memory but not a lot of CPU. Of the predefined types, your choices are limited (either too little memory or too much CPU). To handle situations like these, you can customize machine types to the right size with a specific number of vCPUs and memory, where each CPU costs about \$0.033 per hour, and each GB of memory costs \$0.0045 per hour.

If you've been doing the math along the way, you may notice that these numbers don't quite add up as you'd expect. For example, I said your n1-standard-8 instance costs \$0.38 per hour. The difference is obvious when you look at the cost per month, which is listed as \$200, but \$0.38 per hour times 24 hours per day times ~30 days per month is about \$270, not \$200! It turns out that GCE gives you a discount when you use VMs for a sustained period of time.

9.7.2 Sustained use discounts

Sustained use discounts are a bit like getting a discount for buying in bulk. What makes them particularly cool is that you don't have to commit to buying anything. They work by looking back over the past month, figuring out how many VM-hours you used, and computing the overall hourly price based on that, with a bulk discount if you used VMs for a long period of time. Think of it as a bit like paying less on your electricity bill as you consume more throughout the month. As you use more electricity, the per-unit cost drops until you're paying wholesale prices.

Sustained use discounts have three tiers, with a maximum net discount of 30% for the month. The way it works is by applying a new base rate for the second, third, and fourth quarter of each month. After a VM has been running for 25% of the month, the following 25% of the month is billed at 20% off the regular rate. The next 25% is billed at 40% off, and the final 25% is billed at 60% off. When you put this all together, running 100% of the month means you end up paying 30% less than you would have without the discount. See figure 9.52.

In this chart, the top line is the normal cost, which follows a straight line. The actual cost follows the bottom line, where the slope of the curve decreases over time.

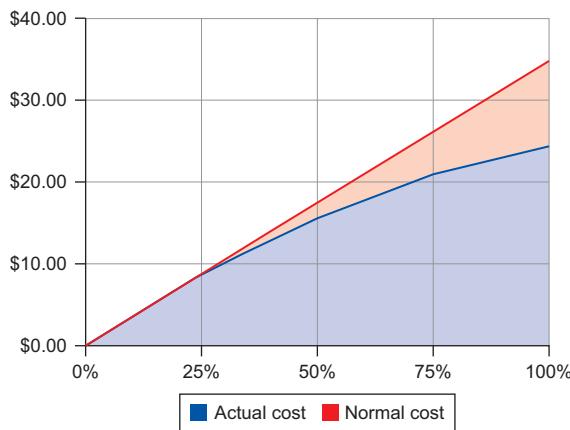


Figure 9.52 Sustained use discount vs. normal cost

At the end of the month, the actual cost line ends up being about 30% lower. This example is straightforward, but what if you have two VMs that you run for half of the month each? Or what if you have a custom machine type? What about when you have autoscaling turned on? It turns out that this all gets pretty complicated, so rather than trying to enumerate all of these examples, it might be better to communicate the underlying principle Google uses when doing these calculations.

First, GCE tries to infer a consistent amount of usage, even if you reconfigure your machines frequently. It looks at the number of VM instances that were running and tries to combine them into a denser configuration to figure out the minimum number of simultaneously running VM instances. Using this condensed graph of inferred instances, GCE will try to calculate the maximum discount possible given the configuration. If you’re terrified of that, you’re not alone, so I’ll make it clearer with a picture (figure 9.53).

In the example in figure 9.53, you can see that you had five instances running over the course of the month, with some different overlaps—for example, VM 4 was running at the same time as VM 3 and VM 5. First, GCE condenses or *flattens* the images to get the minimum number of slots you’d need to handle this usage scenario. In the example, rather than turning on VM 3 in week 3 as you did, you could’ve recommissioned VM 1 to do the

Actual use			
Week 1	Week 2	Week 3	Week 4
VM 1			
	VM 2		
		VM 3	
			VM 4
			VM 5

Inferred instances			
Week 1	Week 2	Week 3	Week 4
VM 1		VM 3	
	VM 2	VM 4	
		VM 5	

Discount computation			
Week 1	Week 2	Week 3	Week 4
VM 1		VM 3	
VM 2		VM 4	
VM 5			

Figure 9.53 Inferred instances and discount computation

same work. For calculating cost, GCE will flatten the two VMs together and treat them as a single run of sustained use, even though you turned one machine off and another one on.

Once GCE has the inferred instances, it slides everything to the start of the month, as you can see in figure 9.53. It uses this final condensed and shifted graph to apply the discounted rate for each segment, and then it computes how much of a discount it can apply.

The more time that multiple VMs are running concurrently, the less condensation GCE can do when calculating a discount. That means not all VM hours cost the same. Running a single VM for a full month (~730 hours) might cost the same as running 730 machines for one hour each, but only if all of those machines run in order (turn one off and turn another one on at the same time). If you run 730 machines all for the exact same hour, you won't see any discounts at all, so the overall cost will be 30% more expensive. For example, figure 9.54 shows you running more instances at the same time (only VM 2 and VM 4 don't overlap at all), so GCE can't condense as much and therefore applies a smaller discount.

Finally, before I move onto storage costs, it's important to remember that all of the cost numbers so far have been based on resources based in the United States. GCE offers lots of regions where you can run VMs, and the prices differ from region to region. The reason behind this is mostly variable costs to Google (in the form of electricity, property, and so on), but it also tends to relate to available capacity. Aside from the overall cost, the costs for the different resources (such as predefined machine types, custom machine type vCPUs and memory, and extended memory) vary quite a bit from one region to the next. For example, table 9.3 shows a few prices per vCPU and GB of memory in different regions.

Table 9.3 Prices per vCPU based on location

Resource	Iowa	Sydney	London
vCPU	\$0.033174	\$0.04488	\$0.040692
GB memory	\$0.004446	\$0.00601	\$0.005453

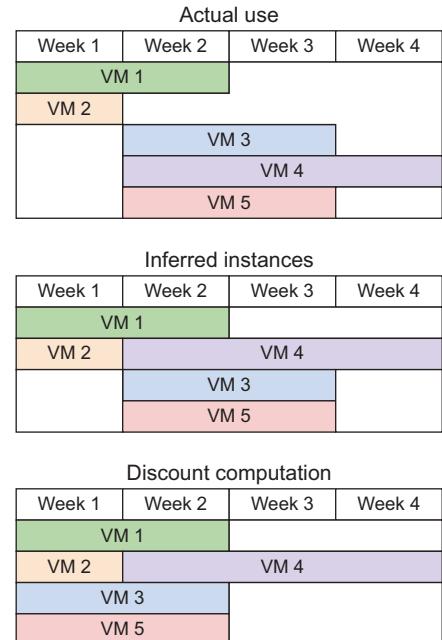


Figure 9.54 Less condensation is possible when there's more overlap.

To put that in perspective, a VM in London might cost around 25% more than the same VM in Iowa. (For example, an n1-standard-16 costs about \$388 per month in Iowa but about \$500 per month in London.) In general, it'll be cheapest to run your VMs in US-based regions (like Iowa), and you typically should only run them in other regions if you have a meaningful reason for doing so, such as needing low latency to your customers in Australia or having concerns about data living outside the EU.

9.7.3 Preemptible prices

In addition to regular list prices for VMs and sustained use discounts, preemptible VMs have special price reductions in exchange for the restrictions on these instances. As always, these prices vary from location to location, but the structure remains the same with per-hour prices for the use of the instance. Table 9.4 shows some example prices for a few instance types in three popular locations.

Table 9.4 Preemptible instance hourly prices for a few locations

Instance type	Iowa	Sydney	London
n1-standard-1	\$0.01	\$0.01349	\$0.01230
n1-standard-2	\$0.02	\$0.02698	\$0.02460
n1-standard-4	\$0.04	\$0.05397	\$0.04920
n1-standard-8	\$0.08	\$0.10793	\$0.09840
n1-standard-16	\$0.16	\$0.21586	\$0.19680
n1-standard-32	\$0.32	\$0.43172	\$0.39360
n1-standard-64	\$0.64	\$0.86344	\$0.78720

As you can see, these prices do indeed vary by location, but they're around 80% cheaper than the standard hourly prices. If you're cost-conscious, it might make sense to see if you can find a way to make preemptible instances work for your project.

You've made it through the hard part. Now let's finish by looking at the easier aspects of GCE pricing: storage and networking.

9.7.4 Storage

Compared to VM pricing, storage pricing is a piece of cake. As you learned earlier, you can use a few classes of persistent disk storage with your VM instances, each with different performance capabilities. Each of these classes has a different cost (with SSD disks costing more than standard storage), and the rates tend to differ depending on the region, like VM prices. Table 9.5 shows some of the rates per GB per month of disk storage for the same regions I discussed before (Iowa, Sydney, and London).

Table 9.5 Data storage rates based on location and disk type

Disk type	Iowa	Sydney	London
Standard	\$0.040	\$0.054	\$0.048
SSD	\$0.170	\$0.230	\$0.204

To put that in perspective, a solid-state persistent disk in London might cost around 20% more than the same disk in Iowa. (For example, a 1 TB SSD costs about \$170 per month in Iowa but about \$200 per month in London.) In general, as with VMs, it'll be cheapest to keep your data in US-based regions, and you typically should keep persistent disks in other regions only if you have a good geographical reason to do so. Additionally, as you can see, the price of SSDs far outstrips the cost of a standard disk, meaning you should use SSDs only if you have a strong performance need to do so.

For example, if you have a 1 TB SSD disk in Sydney that you want to back up, you might want to consider uploading the important data somewhere else (like a Cloud Storage bucket), as it will cost you about \$200 to have it as a disk (or \$35 if you save it as a snapshot only), but only about \$15 if you store it on Cloud Storage, or even as low as \$10 if you use Nearline storage described in section 8.4.3.

We've gone through how much it costs to store data on persistent disks. Now let's look at the final piece of the puzzle: networking costs.

9.7.5 Network traffic

Typically, when you build something using GCE, you don't intend for it to live entirely in a vacuum with no communication with the outside world. On the contrary, most VMs you create will be sending data back to customers, like images or videos or other web pages. Although the incoming data is always free, unfortunately, sending this data around the world isn't. As with VMs, network cables around the world have varying costs, and outgoing (or *egress*) networking costs vary depending on where they exit from Google's network. You can guess by now that sending data out of places like Iowa will cost less than sending it from a place like Sydney. Sending data from one Google zone to another isn't free, either, because it's a *fast pipe* across long distances on Google-owned network cables.

To understand costs for networking, you need to look at both where the traffic comes from and where it's going. Google uses its own network infrastructure to make sure your data gets to its destination as quickly as possible, which, as you'd expect, costs more to go a greater distance. For example, getting a packet from Iowa to New York City is far less costly than getting a packet from Iowa to Australia.

That said, the networking prices for traffic to Australia or mainland China are the same regardless of the source, but that could change down the line. For all other locations (everywhere except those two), the cost varies depending on where the VM is sending the data from.

Additionally, in the same way that buying bulk from Costco gets you a discount, traffic prices go down as you send more data. For GCE, the cost per GB of traffic has three pricing tiers: one for the first TB (which should be most of us), another price for the next 9 TB (more than 1 TB, up to 10 TB), and then a final bulk price for all data after the first 10 TB. Table 9.6 lists some example prices from the same regions as before (Iowa, Sydney, and London) when sending data to most locations.

Table 9.6 Network prices per GB of data for most locations

Price group	Iowa	Sydney	London
First TB	\$0.12	\$0.19	\$0.12
Next 9 TB	\$0.11	\$0.18	\$0.11
Above 10 TB	\$0.08	\$0.15	\$0.08

For sending data to those two special places (Australia and mainland China), the prices are currently the same, regardless of the origin (table 9.7).

Table 9.7 Network prices per GB of data from anywhere to special places (mainland China and Australia)

Price group	To mainland China	to Australia
First TB	\$0.23	\$0.19
Next 9 TB	\$0.22	\$0.18
Above 10 TB	\$0.20	\$0.15

To put this in perspective, imagine you have a 50 MB video file that you want to serve on your website. Assume you get 10,000 people watching the video, so that's 10,000 hits of 50 MB each for a total of 500 GB of data altogether. Typically, that number would be enough to figure out the cost, but, as you learned earlier, you need to consider where the hits are coming from, because the destination of your 50 MB video costs more for certain places than it does for others. You also need to know where the VM serving the video is running, because the source of the data matters as well!

Imagine the video is on a VM in Iowa (so you'll use the Iowa egress cost table above), and 10% of the hits are from Australia, 10% are from mainland China, and the other 80% of the hits are coming from elsewhere in the world, such as New York, Hong Kong (not part of mainland China), and London. Your cost calculations can be broken down as shown in table 9.8.

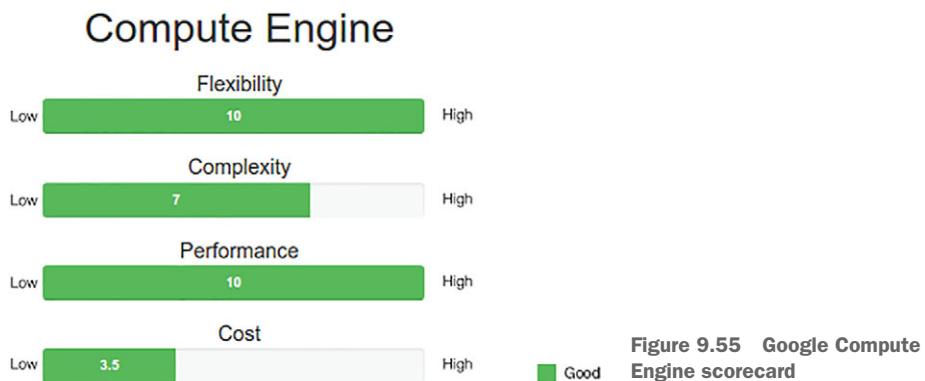
Table 9.8 Breakdown of cost calculations based on location

Location	GB served	Cost per GB	Total cost
China	50 GB	\$0.23	\$11.50
Australia	50 GB	\$0.19	\$9.50
Elsewhere	400 GB	\$0.12	\$48.00
Total			\$69.00

As you can see, most of the data transfer costs for the 500 GB you sent around the world came from all of the other locations, but the special destinations held a disproportionate amount of the total cost. In this case, mainland China destinations were 10% of the traffic but resulted in about 16% of the total cost. There's not a lot you can do to limit who downloads what, short of restricting access to certain regions by IP address, but it's worth keeping in mind that network costs to special destinations can add quite a bit to your total if you happen to handle a lot of traffic from those places.

9.8 When should I use GCE?

To figure out whether GCE is a good fit, let's start by looking at the scorecard (figure 9.55), which summarizes the various computing aspects you might care about.

**Figure 9.55 Google Compute Engine scorecard**

9.8.1 Flexibility

The first thing to note with GCE is that it's as flexible as you can get in a cloud computing environment. It focuses specifically on providing general purpose infrastructure for you to build on.

Although you do have access to fancier things, like autoscaling instance groups and load balancing, those are extras that you can use if they fit your needs. If, for example, you found that you needed some special load balancing feature, you could opt to not use a hosted load balancer and instead turn on your own VM to run the

load balancing software. That said, GCE has limits, but those limits tend to be standard across all cloud hosting providers. For example, it's currently not possible to bring your own hardware into a cloud data center, which means you won't be able to run your own hardware-based load balancer (such as one of the products from F5).

9.8.2 **Complexity**

As you can see from the length of this chapter, GCE is far from simple. If all you want is a virtual machine that runs your software, you could've stopped reading this chapter a long time ago. On the other hand, if you want to use the other, more powerful features of GCE, things can get much more difficult quickly. For example, to take advantage of its autoscaling capability, you first have to understand how instance templates work, then load balancers, and finally health checks. Without those, it's not possible to get the full benefit of an autoscaled system. Put more simply, because you can get going relatively quickly with GCE, the overall difficulty is somewhat lower than that of other computing systems that require you to learn everything before being even trivially useful.

9.8.3 **Performance**

When it comes to performance, GCE scores particularly well. Being as close to bare metal as you'll get in Google Cloud means that you have the fewest possible abstraction layers between your code and the physical CPU doing the work. In other computing systems (for example, App Engine Standard or Heroku), more layers of abstraction exist between the physical CPU and your code, which means they'll be slightly less efficient and therefore have slightly worse performance.

This isn't to say that other managed computing platforms aren't useful or are materially inefficient, but the nature of their design (being higher up the stack) means more work has to happen, so CPU cycles that could be spent on your code are spent instead on other things.

9.8.4 **Cost**

Finally, GCE is relatively low on the cost scale, given that you're only paying for raw virtual machines and disks. Additionally, computing resources are discounted as you use them throughout the month, so you can get large discounts on resources without having to reserve them ahead of time. Notice in particular that GCE's rates are hourly, meaning your costs should be much easier to estimate compared to a fully managed service like Cloud Datastore, which depends on how many requests you make to the service.

9.8.5 **Overall**

Now that you can see how GCE works, let's look at how you might use it for each of the sample applications (the To-Do List, InstaSnap, and E*Exchange) to see how they stack up. It's important to note that GCE will work for each of the examples, so

I'll discuss whether you might want to use just the basics or some of the more advanced features.

9.8.6 To-Do List

The To-Do List app, being a small toy and unlikely to see tons of traffic, probably won't need any of the advanced features of GCE, like autoscaling or preemptible VMs. This is because the traffic patterns you expect are nothing more than going from zero (no one using the app) to a moderate amount (a few people using it at peak hours). If you use GCE, you're buying into a guaranteed price and have to learn and configure quite a bit to use the automatic scaling features. Also, you have no way to lay dormant for the time when the app has no traffic.

As a result, although you could use GCE, it's likely you'll only turn on a single VM and leave that running around the clock. For this type of hobby project that won't have a lot of traffic in total, or a lot of volatility in the traffic patterns, a fully managed system like App Engine (which you'll learn about later) might be a better fit.

Table 9.9 To-Do List application computing needs

Aspect	Needs	Good fit?
Flexibility	Not all that much	Overkill
Complexity	Simpler is better.	Not so good
Performance	Low to moderate	Slightly overkill during nonpeak time
Cost	Lower is better.	Not ideal, but not awful either

Overall, as you can see in table 9.9, GCE is an acceptable fit if you only use the basic aspects of the platform. But it's likely to cost more than necessary and unlikely that the application will make use of the more advanced features available.

9.8.7 E*Exchange

E*Exchange, the online trading platform, has more complex features, as well as much more flexibility in what makes a good fit for running the computing resources (table 9.10).

Table 9.10 E*Exchange computing needs

Aspect	Needs	Good fit?
Flexibility	Quite a bit	Definitely
Complexity	Fine to invest in learning	OK
Performance	Moderate	Definitely
Cost	Nothing extravagant	Definitely

First, an application like this needs quite a bit more flexibility than the To-Do List. For example, rather than handling requests to look at and modify to-do items, you may need to run background computation jobs to collect statistics and email them to users as reports. This is a fine fit for GCE, which is able to handle general computing needs like this.

When it comes to complexity, if you’re building something as complex as this trading application, you probably have the time to invest in learning about the system’s more complex features. It’s not necessarily a bad fit to have a complex system to understand.

Next, your performance needs aren’t extraordinarily large, but they aren’t tiny either. It seems plausible that you may want to use some of the larger instance types so that viewing pages in a browser feels quick and snappy. Similarly, your budget for an application like this isn’t necessarily enormous, but you do have a reasonable budget to spend on computing resources. As a result, because GCE’s prices are pretty reasonable, it’s unlikely the bill will come to anything extravagant for your application serving web pages and running reports.

All of this means that E*Exchange is a pretty good fit to use GCE. GCE offers the right balance of flexibility with a relatively low cost and solid performance.

9.8.8 InstaSnap

InstaSnap, the popular social media photo sharing application, has a few requirements that seem to fit well and a few others that are a bit off (table 9.11).

Table 9.11 InstaSnap computing needs

Aspect	Needs	Good fit?
Flexibility	A lot	Definitely
Complexity	Eager to use advanced features	Mostly
Performance	High	Definitely
Cost	No real budget	Definitely

As you can imagine, for InstaSnap you need a lot of flexibility and performance, which is a great fit for GCE. You’re also willing to pay for the best stuff, and all the venture capital funding you get means you have a pretty big budget, making it fit here also.

You also are interested in the bleeding edge of cool features, such as autoscaling and managed load balancing, and GCE’s a good fit there. That said, you’ll learn later that although GCE’s advanced features are great, some other systems offer even more advanced orchestration, such as Kubernetes Engine. Although GCE’s a reasonably good fit, better options may be available.

Summary

- Virtual machines are virtualized computing resources, a bit like slices of a physical computer somewhere.
- GCE offers virtual machines for rent priced by the hour (billable by the second) as well as persistent replicated disks to store data for the machines.
- GCE can automatically turn machines on and off based on a template, allowing you to automatically scale your system up and down.
- With highly scalable workloads where worker VMs can turn on and off quickly and easily, preemptible VMs can reduce costs significantly, with the caveat that machines can live no longer than 24 hours and may die at any time.
- GCE is best if you want fine-grained control of your computing resources and want to be as close to the physical infrastructure as possible.

Kubernetes Engine: managed Kubernetes clusters

This chapter covers

- What containers, Docker, and Kubernetes do
- How Kubernetes Engine works and when it's a good fit
- Setting up a managed Kubernetes cluster using Kubernetes Engine
- Upgrading cluster nodes and resizing a cluster

A common problem in software development is the final packaging of all your hard work into something that's easy to work with in a production setting. This problem is often neglected until the last minute because we tend to keep our focus on building and designing the software itself. But the final packaging and deployment are often as difficult and complex as the original development. Luckily many tools are available to address this problem, one of which relies on the concept of a *container* for your software.

10.1 What are containers?

A container is an infrastructural tool aimed at solving the software deployment problem by making it easy to package your application, its configuration, and any dependencies into a standard format. By relying on containers, it becomes easy to share and replicate a computing environment across many different platforms. Containers also act as a unit of isolation, so you don't have to worry about competing for limited computing resources—each container is isolated from the others.

If all of this sounds intimidating, don't worry: containers are pretty confusing when you're starting to learn about them. Let's walk through each piece, one step at a time, starting with configuration.

10.1.1 Configuration

If you've ever tried to deploy your application and realized you had a lot more dependencies than you thought, you're not alone. This issue can make one of the benefits of cloud computing (easily created fresh-slate virtual machines) a bit of a pain! Being engineers, we've invented lots of ways of dealing with this over the years (for example, using a shell script that runs when a machine boots), but configuration remains a frustrating problem. Containers solve this problem by making it easy to set up a clean system, describe how you want it to look, and keep a snapshot of it once it looks exactly right. Later, you can boot a container and have it look exactly as you described.

You may be thinking about the Persistent Disk snapshots you learned about in chapter 9 and wondering why you shouldn't use those to manage your configuration. Although that's totally reasonable, it suffers from one big problem: those snapshots only work on Google! This problem brings us to the next issue: standardization.

10.1.2 Standardization

A long time ago (pre-1900s) (figure 10.1), if you wanted to send a table and some chairs across the ocean from England to the United States, you had to take everything to a ship and figure out how to fit it inside, sort of like playing a real-life game of Tetris. It was like packing your stuff into a moving van—just bigger—and you shared the experience with everyone else who was putting their stuff in there too.

Eventually, the shipping industry decided that this way of packing things was silly and started exploring the idea of containerization. Instead of packing things like puzzle pieces, people solve the puzzle themselves using big metal boxes (*containers*) before they even get to a boat. That way, the boat crew only ever deals with these standard-sized containers and never has to play Tetris again. In addition to reducing the time it took to load boats, standardizing on a specific type of box with specific dimensions meant the shipping industry could build boats that were good at holding containers (figure 10.2), devise tools that were good at loading and unloading containers, and charge prices based on the number of containers. All of this made shipping things easier, more efficient, and cheaper.



Figure 10.1 Shipping before containers



Figure 10.2 Shipping using containers

Software containers do for your code what big metal boxes did for shipping. They act as a standard format representing your software and its environment and offer tools to run and manage that environment so it works on every platform. If a system understands containers, you can be sure that when you deploy your code there, it'll work. More concretely, you can focus specifically on getting your code into a container, playing Tetris up front instead of when you're trying to deploy to production. One last piece here needs mentioning, and it comes as a by-product of using containers: isolation.

10.1.3 Isolation

One thing you might notice in the first shipping picture (figure 10.1) is that transporting stuff before containers looked a bit risky, because your things might get crushed by other, heavier things. Luckily, inside a container for shipping or for your code, you only worry about your own stuff. For example, you might want to take a large machine and chop it into two pieces: one for a web server and another for a database. Without a container, if the database were to get tons of SQL queries, the web server would have far fewer CPU cycles to handle web requests. But using two separate containers makes this problem go away. Physical containers have walls to prevent a piano from crushing your stuff, and software containers run in a virtual environment with similar walls that allow you to decide exactly how to allocate the underlying resources.

Furthermore, although applications running on the same virtual machine may share the same libraries and operating system, they might not always do so. When applications running on the same system require different versions of shared libraries, reconciling these demands can become quite complicated. By containerizing the application, shared libraries aren't shared anymore, meaning your dependencies are isolated to a single application (figure 10.3).

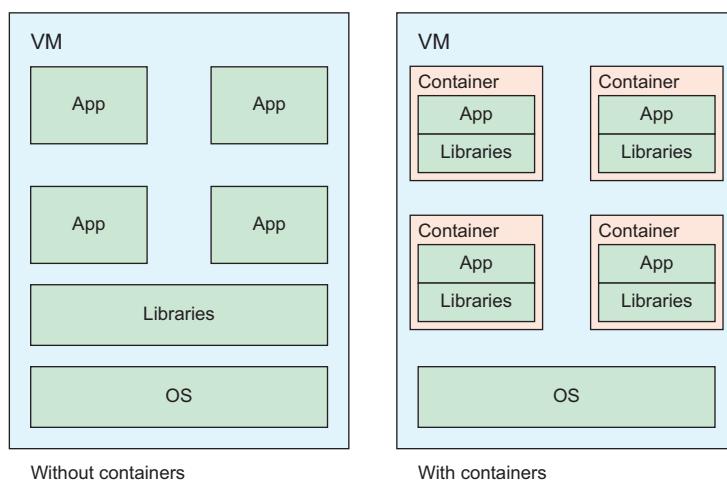


Figure 10.3 Applications without containers vs. with containers

Now you understand the benefits of configuration, standardization, and isolation. With those benefits in mind, let's jump up a layer in the stack and think about the ship that will hold all of these containers, and the captain who will be steering the ship.

10.2 What is Docker?

Many systems are capable of running virtualized environments, but one has taken the lead over the past few years: Docker. Docker is a tool for running containers and acts a bit like a modern container ship that carries all of the containers from one place to another. At a fundamental level, Docker handles the lower-level virtualization; it takes the definitions of container images and executes the environment and code that the containers define.

In addition to being the most common base system for running containers, Docker also has become the standard for how you define container images, using a format called a Dockerfile. Dockerfiles let you define a container using a bunch of commands that do anything from running a simple shell command (for example, `RUN echo "Hello World!"`) all the way to more complex things like exposing a single port outside the container (`EXPOSE 8080`) or inheriting from another predefined container (`FROM node:8`). I'll refer back to the Dockerfile format throughout this chapter, and although you should understand what this type of file is trying to accomplish, don't worry if you don't feel comfortable writing one from scratch. If you get deeper into containers, entire books on Docker are available that can help you learn how to write a Dockerfile of your own.

NOTE If you want to follow along with the code and deployment in this chapter, you should install the Docker runtime on your local machine, which is available at <http://docker.com/community-edition> for most platforms.

10.3 What is Kubernetes?

If you start using containers, it becomes natural to split things up based on what they're responsible for (figure 10.4). For example, if you were creating a traditional web application, you might have a container that handles web requests (for example, a web app server that handles browser-based requests), another container that handles caching frequently accessed data (for example, running a service like Memcached), and another container that handles more complex work, like generating fancy reports, shrinking pictures down to thumbnail size, or sending e-mails to your users.

Managing where all of these containers run and how they talk to one another turns out to be tricky. For example, you might want all of the web app servers to have Memcached running on the same physical (or virtual) machine so that you can talk to Memcached over localhost rather than a public IP. As a result, there are a bunch of systems that try to fix this problem, one of which is Kubernetes.

Kubernetes is a system that manages your containers and allows you to break things into chunks that make sense for your application, regardless of the underlying

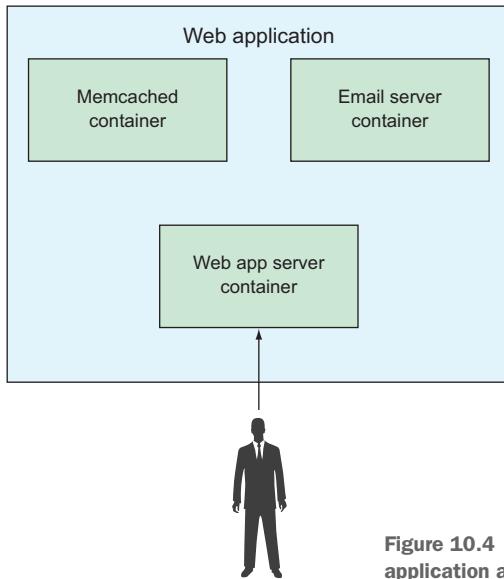


Figure 10.4 Overview of a web application as containers

hardware that runs the code. It also allows you to express more complex relationships, like the fact that you want any VMs that handle web requests to have Memcached on the same machine. Also, because it's open source, using it doesn't tie you to a single hosting provider. You can run it on any cloud provider, or you can skip out on the cloud entirely by using your own hardware. To do all of this, Kubernetes builds on the concept of a container as a fundamental unit and introduces several new concepts that you can use to represent your application and the underlying infrastructure. We'll explore them in the next several subsections.

NOTE Kubernetes is an enormous platform that has been evolving for several years and becoming more and more complex as time goes on, meaning it's too large to fit everything into a single chapter. As a result, I'm going to focus on demonstrating how you can use Kubernetes. If you want to learn more about Kubernetes, you might want to check out Marko Luksa's book, *Kubernetes in Action* (Manning, 2017).

Because there's so much to cover about Kubernetes, let's start by looking at a big, scary diagram showing most of the core concepts in Kubernetes (figure 10.5). We'll then zoom in on the four key concepts: clusters, nodes, pods, and services.

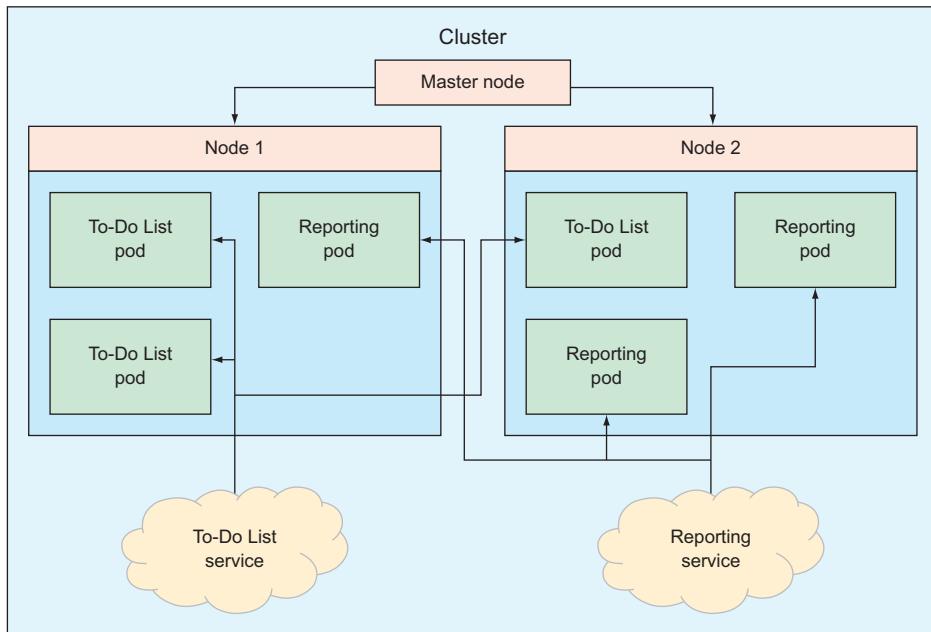


Figure 10.5 An overview of the core concepts of Kubernetes

10.3.1 Clusters

At the top of the diagram, you'll see the concept of a *cluster*, which is the thing that everything else I'm going to talk about lives inside of. Clusters tend to line up with a single application, so when you're talking about the deployment for all of the pieces of an application, you'd say that they all run as part of its Kubernetes cluster. For example, you'd refer to the production deployment of your To-Do List app as your To-Do List Kubernetes cluster.

10.3.2 Nodes

Nodes live inside a cluster and correspond to a single machine (for example, a VM in GCE) capable of running your code. In this example cluster, two different nodes (called Node 1 and Node 2) are running some aspects of the To-Do List app. Each cluster usually will contain several nodes, which are collectively responsible for handling the overall work needed to run your application.

It's important to stress the collective aspect of nodes, because a single node isn't necessarily tied to a single purpose. It's totally possible that a given node will be responsible for many different tasks at once, and that those tasks might change over time. For example, in the diagram, you have both Node 1 and Node 2 handling a mix of responsibilities, but this might not be the case later on when work shuffles around across the available nodes.

10.3.3 Pods

Pods are groups of containers that act as discrete units of functionality that any given node will run. The containers that make up a pod will all be kept together on one node and will share the same IP address and port space. As a result, containers on the same pod can communicate via `localhost`, but they can't both bind to the same port; for example, if Apache is running on port 80, Memcached can't also bind to that same port. The concept of a pod can be a bit confusing, so to clarify, let's look at a more concrete example and compare the traditional version with the Kubernetes-style version.

A LAMP stack is a common deployment style that consists of running Linux (as the operating system), Apache (to serve web requests), MySQL (to store data), and PHP (to do the work of your application). If you were running such a system in a traditional environment (figure 10.6), you might have a server running MySQL to store data, another running Apache with `mod_php` (to process PHP code), and maybe one more running Memcached to cache values (on either the same machine as the Apache server or a separate one).

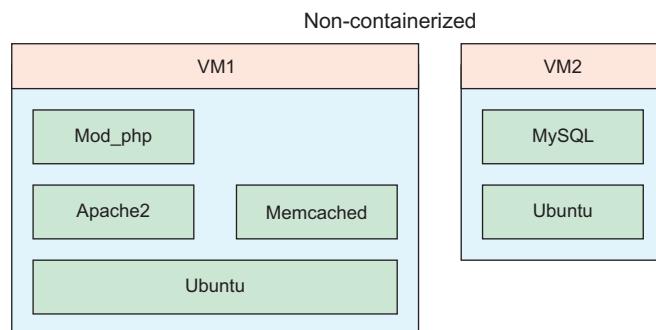


Figure 10.6 Noncontainerized version of a LAMP stack

If you were to think of this stack in terms of containers and pods, you might rearrange things a bit, but the important idea to note is leaving VMs (and nodes) out of the picture entirely. You might have one pod responsible for serving the web app (which would be running Apache and Memcached, each in its own container), and another pod responsible for storing data (with a container running MySQL) (figure 10.7).

These pods might be running on a single VM (or node) or be split across two different VMs (or nodes), but you don't need to care about where a pod is running. As long as each pod has enough computing resources and memory, it should be irrelevant. The idea of using pods is that you can focus on what should be grouped together, rather than how it should be laid out on specific hardware (whether that's virtual or physical).

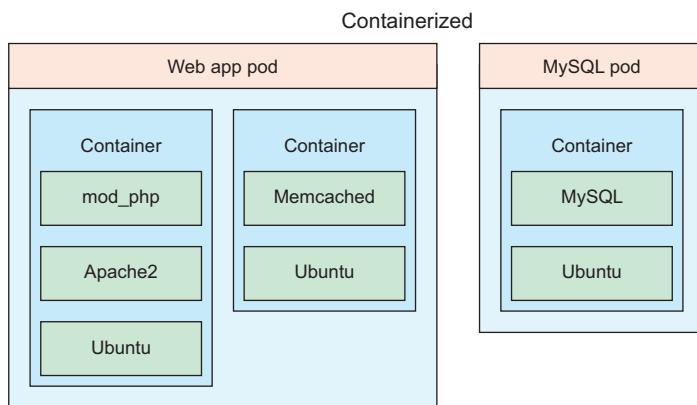


Figure 10.7 Containerized version of a LAMP stack

Looking at this from the perspective of the To-Do List app, you had two different pods: the To-Do List web app pod and the report-generation pod. An example of the To-Do List pod is shown in figure 10.8, which is similar to the LAMP stack I described, with two containers: one for web requests and another for caching data.

Although the ability to arrange different functionality across lots of different physical machines is neat, you may be worried about things getting lost. For example, how do you know where to send web requests for your To-Do List app if it might live on a bunch of different nodes?

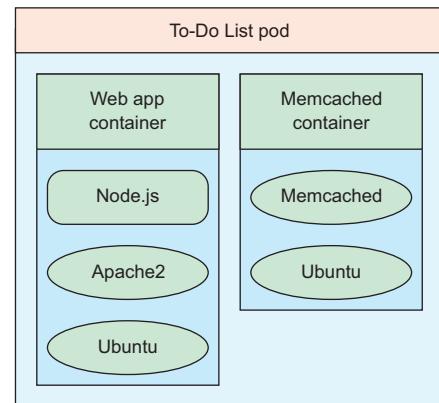


Figure 10.8 The To-Do List pod

10.3.4 Services

A *service* is the abstract concept you use to keep track of where the various pods are running. For example, because the To-Do List web app service could be running on either (or both) of the two nodes, you need a way to find out where to go if you want to make a request to the web app. This makes a service a bit like an entry in a phone book, providing a layer of abstraction between someone's name and the specific place where you can contact them. Because things can jump around from one node to another, this phone book needs to be updated quite often. By relying on a service to keep track of the various pieces of your application (for example, in the To-Do List, you have the pod that handles web requests), you never worry about

where the pod happens to be running. The service can always help route you to the right place.

At this point, you should understand some of the core concepts of Kubernetes, but only in an abstract sense. You should understand that a service is a way to help route you to the right pod and that a pod is a group of containers with a particular purpose, but I've said nothing at all about how to create a cluster or a pod or a service. That's OK! I'll take care of some of that later on. In the meantime, I've reached the point where I can explain what exactly Kubernetes Engine is. All this talk about containers and Kubernetes and pods has finally paid off!

10.4 What is Kubernetes Engine?

Kubernetes is an open source system, so if you want to create clusters and pods and have requests routed to the right nodes, you have to install, run, and manage the Kubernetes system yourself. To minimize this burden, you can use Kubernetes Engine, which is a hosted and managed deployment of Kubernetes that runs on Google Cloud Platform (using Compute Engine instances under the hood).

You still use all of the same tools that you would if you were running Kubernetes yourself, but you can take care of the administrative operations (such as creating a cluster and the nodes inside it) using the Kubernetes Engine API.

10.5 Interacting with Kubernetes Engine

To see how this all works, you can define a simple Kubernetes application and then see how you can deploy it to Kubernetes Engine.

10.5.1 Defining your application

You'll start by defining a simple Hello World Node.js application using Express.js. You should be familiar with Express, but if you're not, it's nothing more than a Node.js web framework. A simple application might look something like the following listing, saved as index.js.

Listing 10.1 Simple Hello World Express application

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
    res.send('Hello world!');
});

app.listen(8080, '0.0.0.0', () => {
    console.log('Hello world app is listening on port 8080.');
});
```

This web application will listen for requests on port 8080 and always reply with the text "Hello world!" You also need to make sure you have your Node.js dependencies

configured properly, which you can do with a package.json file like the one shown in the following listing.

Listing 10.2 package.json for your application

```
{  
  "name": "hellonode",  
  "main": "index.js",  
  "dependencies": {  
    "express": "~4"  
  }  
}
```

How would you go about containerizing this application? To do so, you'd create a Dockerfile, as shown in the next listing, which will look like a start-up script for a VM, but a bit strange. Don't worry, though—you're not supposed to be able to write this from scratch.

Listing 10.3 An example Dockerfile

```
FROM node:8  
WORKDIR /usr/src/app  
COPY package.json .  
RUN npm install  
COPY . .  
EXPOSE 8080  
CMD ["node", "index.js"]
```

Let's look at each line of the listing and see what it does:

- 1 This is the base image (node:8), which Node.js itself provides. It gives you a base operating system that comes with Node v8 preinstalled and ready to go.
- 2 This is the equivalent of cd to move into a current working directory, but it also makes sure the directory exists before moving into it.
- 3 The first COPY command does exactly as you'd expect, placing a copy of something from the current directory on your machine in the specified directory on the Docker image.
- 4 The RUN command tells Docker to execute a given command on the Docker image. In this case, it installs all of your dependencies (for example, express) so they'll be present when you want to run your application.
- 5 You use COPY again to bring the rest of the files over to the image.
- 6 EXPOSE is the same as opening up a port for the rest of the world to have access. In this case, your application will use port 8080, so you want to be sure that it's available.
- 7 The CMD statement is the default command that will run. In this case, you want to start a Node.js process running your service (which is in index.js).

Now that you've written a Dockerfile, it might make sense to test it locally before trying to deploy it to the cloud. Let's take a look at how to do that.

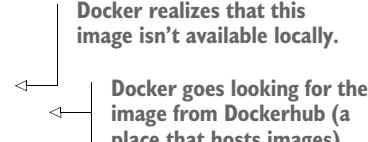
10.5.2 Running your container locally

Before you can run a container on your own machine, you'll need to install the Docker runtime. Docker Community Edition is free, and you can install it for almost every platform out there. For example, there's a .deb package file for Ubuntu available on <http://docker.com/community-edition>.

As you learned earlier, Docker is a tool that understands how to run containers that you define using the Dockerfile format. Once you have Docker running on your machine, you can tell it to run your Dockerfile, and you should see your little web app running. To test whether you have Docker set up correctly, run `docker run hello-world`, which tells Docker to go find a container image called "hello-world." Docker knows how to go find publicly available images, so it'll download the `hello-world` image automatically and then run it. The output from running this image should look something like this:

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
b04784fba78d: Pull complete
Digest: sha256:f3b3b28a45160805bb16542c9531888519430e9e6d6fffc09d72261b0d26ff74f
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
# ... More information here ...
```



To run your image, you have to take your Dockerfile that describes a container and build it into a container image. This is a bit like compiling source code into a runnable binary when you're writing code in a compile language like C++ or Java. Make sure the contents of your Dockerfile are in a file called Dockerfile; then you'll use `docker build` to create your image and tag it as `hello-node`:

```
$ docker build --tag hello-node .
Sending build context to Docker daemon 1.345MB
Step 1/7 : FROM node:8
Step 2/7 : WORKDIR /usr/src/app
Step 3/7 : COPY package.json .
Step 4/7 : RUN npm install
Step 5/7 : COPY .
Step 6/7 : EXPOSE 8080
Step 7/7 : CMD node index.js
Successfully built 358ca555bbf4
Successfully tagged hello-node:latest
```

You'll see a lot happening under the hood, and it'll line up one-to-one with the commands you defined in the Dockerfile. First, it'll go looking for the publicly available

base container that has Node v8 installed, and then it'll set up your work directory, all the way through to running the index.js file that defines your web application. Note that this is only building the container, not running it, so the container itself is in a state that's ready to run but isn't running at the moment.

If you want to test out that things worked as expected, you can use the docker run command with some special flags:

```
$ docker run -d -p 8080:8080 hello-node
```

Here, the -d flag tells Docker to run this container image in the background, and the -p 8080:8080 tells Docker to take anything on your machine that tries to talk to port 8080 and forward it onto your container's port 8080.

The following line shows the result of running your container image:

```
485c84d0f25f882107257896c2d97172e1d8e0e3cb32cf38a36aee6b5b86a469
```

This is a unique ID that you can use to address that particular image (after all, you might have lots of the same image running at the same time).

To check that your image is running, you can use the docker ps command, and you should see the hello-node image in the list:

```
$ docker ps --format "table {{.ID}}\t{{.Image}}\t{{.Status}}"
CONTAINER ID        IMAGE               STATUS
485c84d0f25f      hello-node          Up About a minute
```

As you can see, the container is using the hello-node image and has only been running for about a minute. Also note that the container ID has been shortened to the first few letters of the unique ID from running the docker run command. You can shorten this even further, as long as the ID doesn't match more than one container, so for this exercise, I'll refer to this container as 485c. You told Node to print to the console when it started listening for requests.

You can check the output of your container so far by entering this line:

```
$ docker logs 485c
```

The output here is exactly what you'd expect:

```
Hello world app is listening on port 8080.
```

Now try connecting to the container's Node.js server using curl:

```
$ curl localhost:8080
```

You should see this:

```
Hello world!
```

Like magic, you have a Node.js process running and serving HTTP requests from inside a container being run by the Docker service on your machine. If you wanted to stop this container, you could use the docker stop command:

```
$ docker stop 485c
485c
$ docker ps --format "table {{.ID}}"
CONTAINER ID
```

Here, once you stop the docker container, it no longer appears in the list of running containers shown using docker ps.

Now that you have an idea of what it feels like to run your simple application as a container using Docker, let's look at how you could switch from using your local Docker instance to a full-fledged Kubernetes cluster (which itself uses Docker under the hood). I'll start with how you package up your containerized application and deploy it to your private container registry.

10.5.3 Deploying to your container registry

At this point, you've built and run a container locally, but if you want to deploy it, you'll need it to exist on Google Cloud. You need to upload your container to run it on Kubernetes Engine. To allow you to do this, Google offers a private per-project container registry that acts as storage for all of your various containers.

To get started, you first need to tag your image in a special format. In the case of Google's container registry, the tag format is gcr.io/your-project-id/your-app (which can come with different versions on the end, like :v1 or :v2). In this case, you need to tag your container image as gcr.io/your-project-id/hello-node:v1. To do this, you'll use the docker tag command. As you'll recall, you called the image you created hello-node, and you can always double-check the list of images using the docker images command:

```
$ docker images --format "table {{.Repository}}\t{{.ID}}"
REPOSITORY           IMAGE ID
hello-node          96001025c6a9
```

Re-tag your hello-node Docker image:

```
$ docker tag hello-node gcr.io/project-id/hello-node:v1
```

Once you've retagged the image, you should see an extra image show up in the list of available Docker images. Also notice that the :v1 part of your naming shows up under the special TAG heading in the following snippet, making it easy to see when you have multiple versions of the same container:

```
$ docker images --format "table {{.Repository}}\t{{.Tag}}"
REPOSITORY           TAG
gcr.io/project-id/hello-node    v1
hello-node            latest
```

You could always build your container with this name from the start, but you'd already built this container beforehand.

Now all that's left is to upload the container image to your container registry, which you can do with the gcloud command-line tool:

```
$ gcloud docker -- push gcr.io/project-id/hello-node:v1
The push refers to a repository [gcr.io/project-id/hello-node]
b3c1e166568b: Pushed
7da58ae04482: Pushed
2398c5e9fe90: Pushed
e677efb47ea8: Pushed
aaccb8d23649: Pushed
348e32b251ef: Pushed
e6695624484e: Pushed
da59b99bbd3b: Pushed
5616a6292c16: Pushed
f3ed6cb59ab0: Pushed
654f45ecb7e3: Pushed
2c40c66f7667: Pushed
v1: digest:
sha256:65237913e562b938051b007b8cbc20799987d9d6c7af56461884217ea047665a size:
2840
```

You can verify that this worked by going into the Cloud Console and choosing Container Registry from the left-side navigation. Once there, you should see your hello-node container in the listing, and clicking on it should show the beginning of the hash and the v1 tag that you applied (figure 10.9).

The screenshot shows the Google Cloud Container Registry interface. On the left, there is a sidebar with options: Container Registry (selected), Build triggers, and Build history. The main area has a header with back, refresh, show pull command, and delete buttons. Below the header, it says 'Container Registry / project-id / hello-node'. There is a filter bar labeled 'Filter by name or tag'. A table lists a single container entry:

Name	Tags	Virtual size	Uploaded
65237913e562	v1	250.2 MB	5 minutes ago

Figure 10.9 Container Registry listing of your hello-node container

You've uploaded your container to Google Cloud. Now you can get your Kubernetes Engine cluster ready.

10.5.4 Setting up your Kubernetes Engine cluster

Similar to how you needed to install Docker on a local machine to run a container, you'll need to set up a Kubernetes cluster if you want to deploy your containers to Kubernetes Engine. Luckily, this is a lot easier than it might sound, and you can do it from the Cloud Console, like you'd turn on a Compute Engine VM. To start, choose Kubernetes Engine from the left-side navigation of the Cloud Console. Once there, you'll see a prompt to create a new Kubernetes cluster. When you click on that, you'll

see a page that should look similar to the one for creating a new Compute Engine VM (figure 10.10).

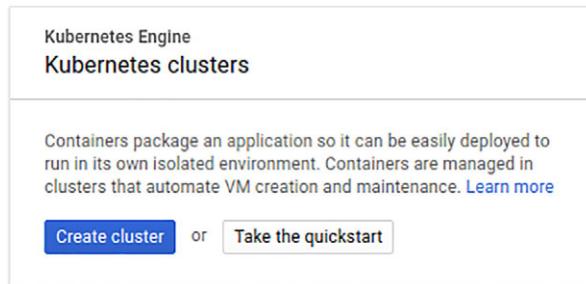


Figure 10.10 Prompt to create a new Kubernetes Engine cluster

Because you're only trying to kick the tires of Kubernetes Engine, you can leave everything set to the defaults. You'll use the us-central1-a zone, a single vCPU per machine, and a size of three VMs for the whole cluster. (Remember, you can always change these things later.) The only thing you should do is pick a name for your cluster in this example, like `first-cluster`. Once you've verified that the form shows what you expect, click the Create button, and then wait a few seconds while Google Kubernetes Engine (GKE) actually creates the VMs and configures Kubernetes on the new cluster of machines.

Once you have your cluster created and marked as running, you can verify that it's working properly by listing your VMs. Remember that a GKE cluster relies on Compute Engine VMs under the hood, so you can look at them like any other VM running:

```
$ gcloud compute instances list --filter "zone:us-central1-a name:gke-*" |  
    awk '{print $1}'  
NAME  
gke-first-cluster-default-pool-e1076aa6-c773  
gke-first-cluster-default-pool-e1076aa6-mdcd  
gke-first-cluster-default-pool-e1076aa6-xhxp
```

You have a cluster running and can see that three VMs that make up the cluster are running. Now let's dig into how to interact with the cluster.

10.5.5 Deploying your application

Once you've deployed your container and created your cluster, the next thing you need to do is find a way to communicate with and deploy things to your cluster. After all, you have a bunch of machines running doing nothing! Because this cluster is made up of machines running Kubernetes under the hood, you can use the existing tools for talking to Kubernetes to talk to your Kubernetes Engine cluster. In this case, the tool you'll use to talk to your cluster is called `kubectl`.

NOTE Keep in mind that some of the operations you'll run using `kubectl` will always return quickly, but they're likely doing some background work under

the hood. As a result, you may have to wait a little bit before moving on to the next step.

In case you’re not super-familiar with Kubernetes (which is expected), to make this process easy, Google Cloud offers a fast installation of kubectl using the gcloud command-line tool. All you have to do to install kubectl is run a simple gcloud command:

```
$ gcloud components install kubectl
```

NOTE If you’ve installed gcloud using a package manager (like apt-get for Ubuntu), you might see a recommendation from gcloud saying to use the same package manager to install kubectl (for example, apt-get install kubectl).

Once you have kubectl installed, you need to be sure that it’s properly authenticated to talk to your cluster. You can do this using another gcloud command that fetches the right credentials and ensures that kubectl has them available:

```
$ gcloud container clusters get-credentials --zone us-central1-a first-cluster
Fetching cluster endpoint and auth data.
kubeconfig entry generated for first-cluster.
```

Once you’ve set up kubectl, you can use it to deploy a new application using your container image under the hood. You can do this by running kubectl run and using kubectl get pods to verify that the tool deployed your application to a pod:

```
$ kubectl run hello-node --image=gcr.io/your-project-id-here/hello-node:v1 --port 8080
deployment "hello-node" created
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
hello-node-1884625109-sjq76 1/1 Running 0 55s
```

You’re now almost done, with one final step before you can check whether things are working as expected. Remember that EXPOSE 8080 command in your Dockerfile? You have to do something similar with your cluster to make sure the ports you need to handle requests are properly exposed. To do this, you can use the kubectl expose command:

```
$ kubectl expose deployment hello-node --type=LoadBalancer --port 8080
service "hello-node" exposed
```

Under the hood, Kubernetes Engine will configure a load balancer like you learned about in chapter 9. Once this is done, you should see a load balancer appear in the Cloud Console that points to your three VM instances that make up the cluster (figure 10.11).

At this point, you may be thinking that pods are the way you keep containers together to serve a common purpose, and not something that you’d talk to individually.

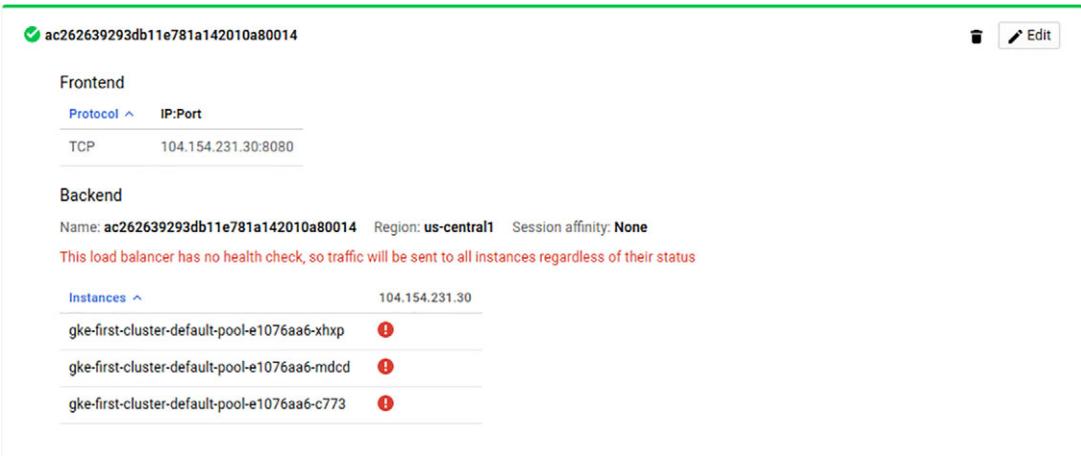


Figure 10.11 Automatically created load balancer in the Cloud Console

And you're right! If you want to talk to your application, you have to use the proper abstraction for this, which is known as a *service*.

You can look at the available services (in this case, your application) using `kubectl get services`:

```
$ kubectl get service
NAME      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
hello-node  10.23.245.188  104.154.231.30  8080:31201/TCP  1m
kubernetes  10.23.240.1    <none>          443/TCP      10m
```

Notice at this point that you have a generalized Kubernetes service (which handles administration), as well as the service for your application. Additionally, your application has an external IP address that you can use to see if everything worked by making a simple request to the service:

```
$ curl 104.154.231.30:8080
Hello world!
```

And sure enough, everything worked exactly as expected. You now have a containerized application running using one pod and one service inside Kubernetes, managed by Kubernetes Engine. This alone is pretty cool, but the real magic happens when you need to handle more traffic, which you can do by replicating the application.

10.5.6 Replicating your application

Recall that using Compute Engine, you could easily turn on new VMs by changing the size of the cluster, but to get your application running on those machines, you needed to set them to run automatically when the VM turned on, or you had to

manually connect to the machine and start the application. What about doing this with Kubernetes? At this point in your deployment, you have a three-node Kubernetes cluster, with two services (one for Kubernetes itself, and one for your application), and your application is running in a single pod. Let's look at how you might change that, but first, let's benchmark how well your cluster can handle requests in the current configuration.

You can use any benchmarking tool you want, but for this illustration, try using Apache Bench (ab). If you don't have this tool installed, you can install it on Ubuntu by running `sudo apt-get install apache2-utils`. To test this, you'll send 50,000 requests, 1,000 at a time, to your application, and see how well the cluster does with handling the requests:

```
$ ab -c 1000 -n 50000 -qSd http://104.154.231.30:8080/
This is ApacheBench, Version 2.3 <$Revision: 1604373 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 104.154.231.30 (be patient).....done
# ...

Concurrency Level:      1000
Requests per second:   2980.93 [#/sec]  (mean)
Time per request:     335.465 [ms]  (mean)

# ...
```

What if you could scale your application up to take advantage of more of your cluster? It turns out that you can do so with one command: `kubectl scale`. Here's how you scale your application to run on 10 pods at the same time:

```
$ kubectl scale deployment hello-node --replicas=10
deployment "hello-node" scaled
```

Immediately after you run this command, looking at the pods available will show that you're going from 1 available up to 10 different pods:

```
$ kubectl get pods
NAME          READY   STATUS        RESTARTS   AGE
hello-node-1884625109-8ltzb  1/1    ContainerCreating  0          3m
hello-node-1884625109-czn7q  1/1    ContainerCreating  0          3m
hello-node-1884625109-dzs1d  1/1    ContainerCreating  0          3m
hello-node-1884625109-gw6rz  1/1    ContainerCreating  0          3m
hello-node-1884625109-kvh9v  1/1    ContainerCreating  0          3m
hello-node-1884625109-ng2bh  1/1    ContainerCreating  0          3m
hello-node-1884625109-q4wm2  1/1    ContainerCreating  0          3m
hello-node-1884625109-r5msp  1/1    ContainerCreating  0          3m
hello-node-1884625109-sjq76  1/1    Running         0          1h
hello-node-1884625109-tc2lr  1/1    ContainerCreating  0          3m
```

After a few minutes, these pods should come up and be available as well:

```
$ kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
hello-node-1884625109-8ltzb 1/1     Running   0          3m
hello-node-1884625109-czn7q 1/1     Running   0          3m
hello-node-1884625109-dzs1d 1/1     Running   0          3m
hello-node-1884625109-gw6rz 1/1     Running   0          3m
hello-node-1884625109-kvh9v 1/1     Running   0          3m
hello-node-1884625109-ng2bh 1/1     Running   0          3m
hello-node-1884625109-q4wm2 1/1     Running   0          3m
hello-node-1884625109-r5msp 1/1     Running   0          3m
hello-node-1884625109-sjq76 1/1     Running   0          1h
hello-node-1884625109-tc2lr 1/1     Running   0          3m
```

At this point, you have 10 pods running across your three nodes, so try your benchmark a second time and see if the performance is any better:

```
$ ab -c 1000 -n 50000 -qSd http://104.154.231.30:8080/
This is ApacheBench, Version 2.3 <$Revision: 1604373 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 104.154.231.30 (be patient).....done
# ...
Concurrency Level:      1000
Requests per second:   5131.86 [#/sec]  (mean)
Time per request:      194.861 [ms]  (mean)
# ...

Your newly scaled-up
cluster handled about
5,000 requests per second.

It completed most
requests in around
200 milliseconds.
```

At this point, you may be wondering if a UI exists for interacting with all of this information. Specifically, is there a UI for looking at pods in the same way that there's one for looking at GCE instances? There is, but it's part of Kubernetes itself, not Kubernetes Engine.

10.5.7 Using the Kubernetes UI

Kubernetes comes with a built-in UI, and because Kubernetes Engine is just a managed Kubernetes cluster, you can view the Kubernetes UI for your Kubernetes Engine cluster the same way you would any other Kubernetes deployment. To do so, you can use the `kubectl proxy` command-line tool to open up a tunnel between your local machine and the Kubernetes master (figure 10.12). That will allow you to talk to, say, `http://localhost:8001`, and a local proxy will securely route your request to the Kubernetes master (rather than a server on your local machine):

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

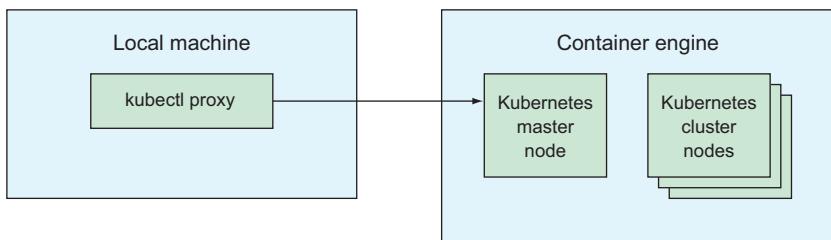


Figure 10.12 Proxying local requests to the Kubernetes master

Once the proxy is running, connecting to <http://localhost:8001/ui/> will show the full Kubernetes UI, which provides lots of helpful management features for your cluster (figure 10.13).

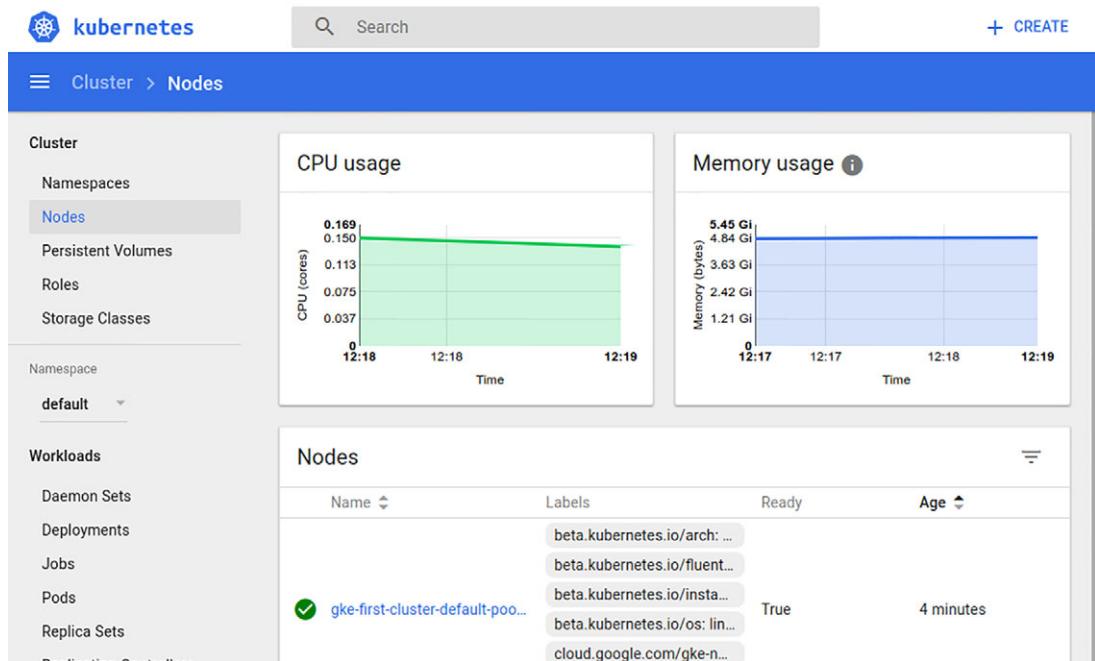


Figure 10.13 Kubernetes UI using kubectl proxy

You've now seen how Kubernetes works at a simplified level. The part that's important to remember is that you didn't have to configure or install Kubernetes at all on your cluster because Kubernetes Engine did it all for you. As I mentioned before, Kubernetes is a huge system, so this chapter isn't about teaching you everything there is to know about it. For example, you can see how to access the Kubernetes UI, but I'm not going into any detail about what you can do using the UI. Instead, the goal of this

chapter is to show you how Kubernetes works when you rely on Kubernetes Engine to handle all of the administrative work.

If you’re interested in doing more advanced things with Kubernetes, such as deploying a more advanced cluster made up of lots of pods and databases, now’s the time to pick up a book about it, because Kubernetes Engine is nothing more than a managed Kubernetes deployment under the hood. That said, quite a few things are specific to Kubernetes Engine and not general across Kubernetes itself, so let’s look briefly at how you can manage the underlying Kubernetes cluster using Kubernetes Engine and the Google Cloud tool chain.

10.6 **Maintaining your cluster**

New versions of software come out, and sometimes it makes sense to upgrade. For example, if Apache releases new bug fixes or security patches, it makes quite a bit of sense to upgrade to the latest version. The same goes for Kubernetes, but remember, because you’re using Kubernetes Engine, instead of deploying and managing your own Kubernetes cluster, you need a way of managing that Kubernetes cluster via Kubernetes Engine. As you might guess, this is pretty easy. Let’s start with upgrading the Kubernetes version.

Your Kubernetes cluster has two distinct pieces that Kubernetes Engine manages: the master node, which is entirely hidden (not listed in the list of nodes), and your cluster nodes. The cluster nodes are the ones you see when listing the active nodes in the cluster. If Kubernetes has a new version available, you’ll have the ability to upgrade the master node, all the cluster nodes, or both. Although the upgrade process is similar for both types of nodes, you’ll have different things to worry about for each type, so we’ll look at them separately, starting with the Kubernetes master node.

10.6.1 **Upgrading the Kubernetes master node**

By default, as part of Google managing them, master nodes are automatically upgraded to the latest supported Kubernetes version after it’s released, but if you want to jump to the latest supported version of Kubernetes, you can choose to manually upgrade your cluster’s master node ahead of schedule. When an update is available for Kubernetes, your Kubernetes Engine cluster will show a link next to the version number that you can click to change the version. For example, figure 10.14 shows the link that displays when an upgrade is available for your master node.

When you click the Upgrade link, you’ll see a prompt that allows you to choose a new version of Kubernetes. As the prompt notes, you need to keep a few things in mind when changing the version of Kubernetes (figure 10.15).

First, upgrading from an older version to a new version on the Kubernetes Engine cluster’s master node is a one-way operation. If you decide later that you don’t like the new version of Kubernetes (maybe there’s a bug no one noticed or an assumption that doesn’t hold anymore), you can’t use this same process to go back to the previous version. Instead, you’d have to create a new cluster with the old Kubernetes version and

The screenshot shows the 'Kubernetes clusters' page in the Google Cloud Platform interface. A cluster named 'first-cluster' is selected. The 'Details' tab is active, displaying the following information:

Master version	1.7.12-gke.1	Upgrade available
Endpoint	104.198.26.195	Show credentials
Client certificate	Enabled	
Kubernetes alpha features	Disabled	
Total size	3	
Master zone	us-central1-a	

Figure 10.14 When an upgrade for Kubernetes is available on Kubernetes Engine

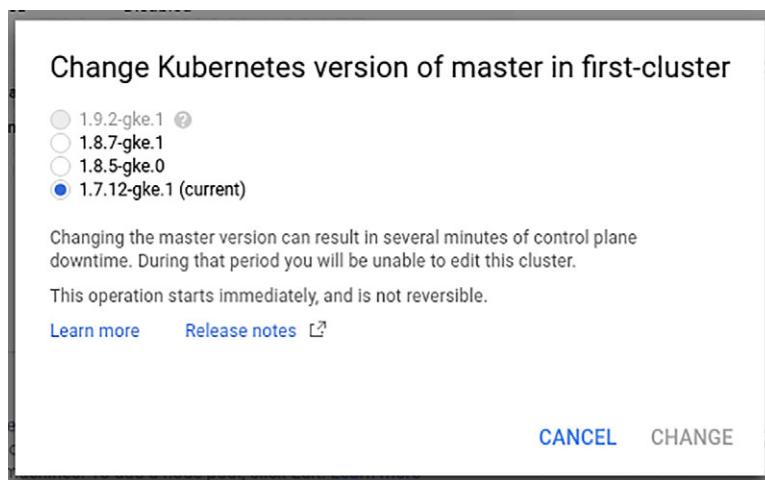


Figure 10.15 Prompt and warning for upgrading your Kubernetes master node

redeploy your containers to that other cluster. To protect yourself against upgrade problems and avoid downtime, it's usually a good idea to try out a separate cluster with the new version to see if everything works as you'd expect. After you've tested out the newer version and found that it works as you expected, it should be safe to upgrade your existing cluster.

Next, changing the Kubernetes version requires that you stop, upgrade, and restart the Kubernetes control plane (the service that `kubectl` talks to when it needs to scale or deploy new pods). While the upgrade operation is running, you'll be unable to edit your cluster, and all `kubectl` calls that try to talk to your cluster won't work. If you sud-

denly receive a spike of traffic in the middle of the upgrade, you won't be able to run `kubectl scale`, which could result in downtime for some of your customers.

Finally, don't forget that manually upgrading is an optional step. If you wait around for a bit, your Kubernetes master node will automatically upgrade to the latest version without you noticing. But that isn't the case for your cluster nodes, so let's look at those in more detail.

10.6.2 Upgrading cluster nodes

Unlike the master node, cluster nodes aren't hidden away in the shadows. Instead, they're visible to you as regular Compute Engine VMs similar to managed instance groups. Also, unlike with the master node, the version of Kubernetes that's running on these managed VMs isn't automatically upgraded every so often. It's up to you to decide when to make this change. You can change the version of Kubernetes on your cluster's nodes by looking in the Cloud Console next to the Node Version section of your cluster and clicking the Change link (figure 10.16).

Node Pools

Node pools are separate instance groups running Kubernetes in a cluster. You may add node pools in different zones for higher availability, or add node pools of different type machines. To add a node pool, click Edit. [Learn more](#)

Name	default-pool
Size	3
Node version	1.6.7
Node image	Container-Optimized OS (cos)
Machine type	n1-standard-1 (1 vCPU 3.75 GB memory)

Figure 10.16 Cloud Console area for changing the version of cluster nodes

You may be wondering why I'm talking about changing the node version rather than upgrading. The reason is primarily because unlike with the master node version, this operation is sometimes reversible (though not always). You can downgrade to 1.5.7 and then decide to upgrade back to 1.6.4. When you click the Change link, you'll see a prompt that allows you to choose the target version and explains quite a bit about what's happening under the hood (figure 10.17).

First, because there's always at least one cluster node (unlike the master node, which is always a single instance), you change the Kubernetes version on the cluster nodes by applying a rolling update to your cluster, meaning the machines are modified one at a time until all of them are ready. To do this, Kubernetes Engine will first make the node unscheduleable. (No new pods will be scheduled on the node.) It'll then *drain* any pods on the node (terminate them and, if needed, put them on

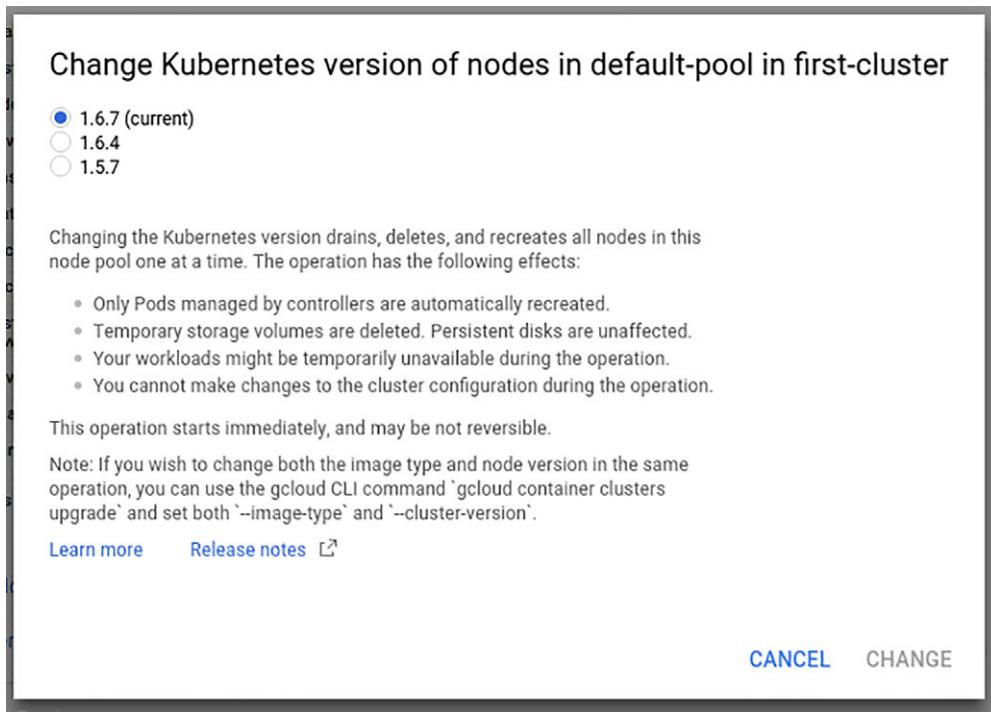


Figure 10.17 Prompt to change the version of cluster nodes

another node). The fewer nodes you have, the more likely it is you'll experience some form of downtime. For example, if you have a single-node cluster, your service will be unavailable for the duration of the downtime—100% of your nodes will be down at some point. On the other hand, if you have a 10-node cluster, you'll be down by 10% capacity at most (1 of the 10 nodes at a single instance).

Second, notice that the choices available in this prompt (figure 10.17) aren't the same as those in the prompt for upgrading the master node (figure 10.15). The list is limited in this way because the cluster nodes must be compatible with the master node, which means not too far behind it (and never ahead of it). If you have a master node at version 1.6.7, you can use version 1.6.4 on your cluster nodes, but if your master node uses a later version, this same cluster node version might be too far behind. As a result, it's a good idea to upgrade your cluster nodes every three months or so.

Third, unlike with the master node, which is hidden from your view, you may have come to expect any data stored on the cluster nodes to be there forever. In truth, unless you explicitly set up persistent storage for your Kubernetes instance, the data you've stored will be lost when you perform an upgrade. The boot disks for each cluster node are deleted and new ones created for the new nodes. Any other nonboot disks (and nonlocal disks) will be persisted. You can read more about connecting Google Cloud persistent storage in the Kubernetes documentation (or one of the many

books on Kubernetes that are available). Look for a section on storage volumes and the `gcePersistentDisk` volume type.

Fourth, and finally, similarly to upgrading the master node, while the version change on cluster nodes is in progress, you won't be able to edit the cluster itself. In addition to the downtime you might experience because of nodes being drained of their pods, the control plane operations will be unavailable for the duration of the version change.

10.6.3 Resizing your cluster

As with scaling up the number of pods using `kubectl scale`, changing the number of nodes in your cluster is easy. In the Cloud Console, if you click Edit on your cluster, you'll see a field called Size, which you originally set to three when you created the cluster.

Changing this number will scale the number of nodes available in your cluster, and you can set the size either to a larger number, which will add more nodes to provide more capacity, or to a smaller number, which will shrink the size of your cluster. If you shrink the cluster, similarly to a version change on the cluster nodes, Kubernetes Engine will first mark a node as unscheduleable, then drain it of all pods, then shut it down. As an example, figure 10.18 shows what it's like to change your cluster from three nodes to six.

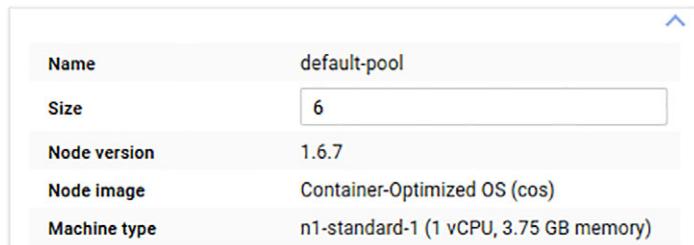


Figure 10.18 Resizing your cluster to six nodes

You also can do this using the `gcloud` command-line tool. For example, the following snippet resizes the cluster from six nodes back to three:

```
$ gcloud container clusters resize first-cluster --zone us-central1-a --size=3
Pool [default-pool] for [first-cluster] will be resized to 3.
```

```
Do you want to continue (Y/n)? Y
```

```
Resizing first-cluster...done.
```

```
Updated [https://container.googleapis.com/v1/projects/your-project-id-here/
→ zones/us-central1-a/clusters/first-cluster].
```

Because we're about to move on from maintenance, you may want to spin down your Kubernetes cluster. You can do so by deleting the cluster, either in the Cloud Console

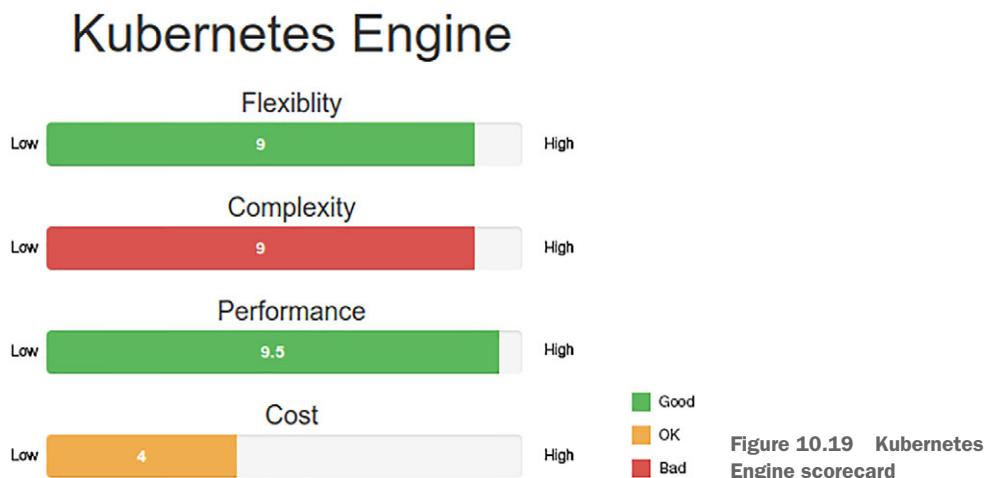
or using the command-line tool. Make sure you move any data you might need to a safe place before deleting the cluster. With that, it's time to move on to looking at how pricing works.

10.7 Understanding pricing

As with some of the other services on Google Cloud Platform, Kubernetes Engine relies on Compute Engine to provide the infrastructure for the Kubernetes cluster. As a result, the cost of the cluster itself is based primarily on the cluster nodes. Because these are simply Compute Engine VMs, you can refer back to chapter 9 for information on how much each node costs. In addition to the cost of the nodes themselves, remember the Kubernetes master node, which is entirely hidden from you by Kubernetes Engine. Because you don't have control over this node explicitly, there's no charge for overhead on it.

10.8 When should I use Kubernetes Engine?

You may be wondering specifically how Kubernetes Engine stacks up against other computing environments, primarily Compute Engine. Let's use the standard scorecard for computing to see how Kubernetes Engine compares to the others (figure 10.19).



10.8.1 Flexibility

Similar to Compute Engine, Kubernetes Engine is quite flexible, but it's not the same as having a general-purpose set of VMs that run whatever you want. For example, you're required to specify your environment using container images (with Dockerfiles), rather than custom start-up scripts or GCE disk images. Although this is technically a limitation that reduces flexibility, it isn't a bad thing to formalize how you define your application in terms of a container. Although Kubernetes Engine is slightly more restrictive, that might be a good thing.

Kubernetes Engine has other limitations as well, such as the requirement that your cluster nodes' Kubernetes version be compatible with the version of your master node, or the fact that you lose your boot disk data when you upgrade your nodes. Again, although these things are technically restrictions, you shouldn't consider them deal breakers. For most scenarios, Kubernetes Engine isn't any less flexible than Compute Engine, and it provides quite a few benefits, such as the ability to scale both nodes and pods up and down. As a result, if you look past the requirement that you define your application using containers, Kubernetes Engine is pretty free of major restrictions when you compare it with Compute Engine. The big difference comes is when you start talking about complexity.

10.8.2 Complexity

As you've seen, computing environments can be complicated, and Kubernetes Engine (which relies on Kubernetes under the hood) is no different. It has a great capacity for complexity, but benefiting from that complexity involves high initial learning costs. Similarly, although a car is a lot more complex than a bicycle, once you learn how to drive the car, the benefits become clear.

Because I've only scratched the surface of what Kubernetes is capable of, you may not have a full understanding of how complex the system as a whole can be—it's far more complicated than "turn on a VM." Putting this into realistic context, if you wanted to deploy a simple application with a single node that would never need to grow beyond that node, Kubernetes Engine is likely overkill. If, on the other hand, you wanted to deploy a large cluster of API servers to handle huge spikes of traffic, it'd probably be worth the effort to understand Kubernetes and maybe rely on Kubernetes Engine to manage your Kubernetes cluster.

10.8.3 Performance

Unlike using raw VMs like Compute Engine, Kubernetes has a few layers of abstraction between the application code and the actual hardware executing that code. As a result, the overall performance can't be as good as a plain old VM, and certainly not as good as a nonvirtualized system. Kubernetes Engine's performance won't be as efficient as something like Compute Engine, but efficiency isn't everything. Scalability is another aspect of performance that can have a real effect.

Although you might need more nodes in your cluster to get the same performance as using nonvirtualized hardware, you can more easily change the overall performance capacity of your system with Kubernetes Engine than you can with Compute Engine or nonvirtualized machines. As a result, if you know your performance requirements exactly, and you're sure they'll stay exactly the same over time, using Kubernetes Engine would be providing you with scalability that you don't need. On the other hand, if you're unsure of how much power you need and want the ability to change your mind whenever you want, Kubernetes Engine makes that easy, with a slight reduction in overall efficiency.

Because this efficiency difference is so slight, it should only be an issue when you have an enormous deployment of hundreds of machines (where the slight differences add up to become meaningful differences). If your system is relatively small, you shouldn't even notice the efficiency differences.

10.8.4 Cost

Kubernetes Engine is no more costly than the raw Compute Engine VMs that power the underlying Kubernetes cluster. Additionally, it doesn't charge for cluster management using a master node. As a result, using it is actually likely to be cheaper than running your own Kubernetes Cluster using Compute Engine VMs under the hood.

10.8.5 Overall

How do you choose between Computer Engine and Kubernetes Engine, given that they're both flexible, perform similarly, and are priced similarly, but using Kubernetes Engine requires you to learn and understand Kubernetes, which is pretty complex? Although this is all true, the distinguishing factor tends to be how large your overall system will be and how much you want your deployment configuration to be represented as code. The benefits of using Kubernetes Engine over other computing platforms aren't about the cost or the infrastructure but about the benefits of Kubernetes as a way of keeping your deployment procedure clear and well documented.

As a result, the general rule is to use Kubernetes Engine when you have a large system that you (and your team) will need to maintain over a long period of time. On the other hand, if you need a few VMs to do some computation and plan to turn them off after a little while, relying on Compute Engine might be easier. To make this more concrete, let's walk through the three example applications that I've discussed and see which makes more sense to deploy using Kubernetes Engine.

10.8.6 To-Do List

The sample To-Do-List app is a simple tool for tracking To-Do-Lists and whether they're done or not. As a result, it's unlikely to need to scale up because of extreme amounts of load. As a result, Kubernetes Engine is probably a bit overkill for its needs (table 10.1).

Table 10.1 To-Do-List application computing needs

Aspect	Needs	Good fit?
Flexibility	Not all that much	Overkill
Complexity	Simpler is better.	Not so good
Performance	Low to moderate	Slightly overkill during nonpeak time
Cost	Lower is better.	Not ideal, but not awful either

Overall, the To-Do-List app, although it can run on Kubernetes, is probably not going to make use of all the features and will require a bit more learning than is desirable for such an application. As a result, something simpler, like a single Compute Engine VM, might be a better choice.

10.8.7 E*Exchange

E*Exchange, the online stock trading platform (table 10.2), has many more complex features, and you can divide each of them into many different categories. For example, you may have an API server that handles requests to the main storage layer, a separate service to handle sending emails to customers, another that handles a web-based user interface, and still another that handles caching the latest stock market data. That's quite a few distinct pieces, which might get you thinking about each piece as a set of containers, with some that might be grouped together into a pod.

Table 10.2 E*Exchange computing needs

Aspect	Needs	Good fit?
Flexibility	Quite a bit	Definitely
Complexity	Fine to invest in learning	Definitely, if it makes things easy
Performance	Moderate	Definitely
Cost	Nothing extravagant	Definitely

Because E*Exchange was a reasonable fit for Compute Engine, it's likely to be a good fit for Kubernetes Engine. It also turns out that the benefits of investing in learning Kubernetes and deploying the services using Kubernetes Engine might save quite a bit of time and simplify the overall deployment process for the application. Unlike the To-Do List, this application has quite a few distinct pieces, each with its own unique requirements. Using multiple different pods for the pieces allows you to keep them all in a single cluster and scale them up or down as needed. Overall, Kubernetes Engine is probably a great fit for the E*Exchange application.

10.8.8 InstaSnap

InstaSnap, the social media photo sharing application (table 10.3), lies somewhere in the middle of the two previous examples in terms of overall system complexity. It probably doesn't have as many distinct systems as E*Exchange, but it definitely has more than the simple To-Do List. For example, it might use an API server that handles requests from the mobile app, a service for the web-based UI, and perhaps a background service that handles processing videos and photos into different sizes and formats.

That said, the biggest concern for InstaSnap is performance and scalability. You may need the ability to increase the resources available to any of the various services if a spike in demand (which happens often) occurs. This requirement makes InstaSnap

a great fit for Kubernetes Engine, because you can easily resize the cluster as a whole as well as the number of pod replicas running in the cluster.

Table 10.3 InstaSnap computing needs

Aspect	Needs	Good fit?
Flexibility	A lot	Definitely
Complexity	Eager to use advanced features	Definitely
Performance	High	Mostly
Cost	No real budget	Definitely

As you can see in table 10.3, even though you don't have as many distinct services as E*Exchange, Kubernetes Engine is still a great fit for InstaSnap, particularly when it comes to using the advanced scalability features. Although the performance itself is slightly lower, based on more abstraction happening under the hood, this requirement has little effect on the choice of a computing platform. You can always add more machines if you need more capacity (which is OK due to the "No real budget" need for cost).

Summary

- A container is an infrastructural tool that makes it easy to package up code along with all dependencies down to the operating system.
- Docker is the most common way of defining a container, using a format called a Dockerfile.
- Kubernetes is an open source system for orchestrating containers, helping them act as a cohesive application.
- Kubernetes Engine is a hosted and fully managed deployment of Kubernetes, minimizing the overhead of running your own Kubernetes cluster.
- You can manage your Kubernetes Engine cluster like any other Kubernetes cluster, using kubectl.

App Engine: fully managed applications

This chapter covers

- What is App Engine, and when is it a good fit?
- Building an application using the Standard and Flex versions
- Managing how your applications scale up and down
- Using App Engine Standard's managed services

As you've learned, there are many available computing platforms, representing a large variety in terms of complexity, flexibility, and performance. Whereas Compute Engine was an example of low-level infrastructure (a VM), App Engine is a fully managed cloud computing environment that aims to consolidate all of the work needed when deploying and running your applications. In addition to being able to run code as you would on a VM, App Engine offers several services that come in handy when building applications.

For example, if you had a to-do list application that required storing lists of work you needed to finish, it wouldn't be unusual for you to need to store some data, send emails, or schedule a background job every day (like recomputing your to-do list completion rate). Typically, you'd need to do all of this yourself by turning on a database, signing up for an email sending service, running a queuing system

like RabbitMQ, and relying on Linux's cron service to coordinate it all. App Engine offers a suite of hosted services to do this so you don't have to manage it yourself.

App Engine is made up of two separate environments that have some important differences. One environment is built using open source tools like Docker containers. The other is built using more proprietary technology that allows Google to do interesting things when automatically scaling your app, although it imposes quite a few limitations on what you can do with your code. Both environments are under the App Engine umbrella, but they're pushing against the boundaries of what you could consider a single product. As a result, we'll look at them together in one chapter, but in a few places, we'll hit a fork in the road. At that point, it'll make sense to split the two environments apart.

The App Engine Standard Environment, released in early 2008, offers a fully managed computing environment complete with storage, caching, computing, scheduling, and more. But it's limited to a few programming languages. In this type of environment, your application tends to be tailored to App Engine, but it benefits from living in an environment that's always autoscaling. App Engine handles sudden spikes of traffic sent to your application gracefully, and periods when your application is inactive don't cost you any money.

App Engine Flexible Environment (often called App Engine Flex) provides a fully managed environment with fewer restrictions and somewhat more portability, trading some scalability in exchange. App Engine Flex is based on Docker containers, you're not limited to any specific versions of programming languages, and you can still take advantage of many of the other benefits of App Engine, such as the hosted cron service.

If you're confused about which environment is right for you, this chapter will help clarify things. We'll first explore some of the organizational concepts, then go further into the details, and finally look at how to choose whether App Engine is right for you. If it turns out that App Engine is a great fit, we'll explore how to choose which of the two environments is best to meet your needs.

11.1 Concepts

Because App Engine is a hosted environment, the API layer has a few more organizational concepts that you'll need to understand to use App Engine as a computing platform. App Engine uses four organizational concepts to understand more about your application: applications, services, versions, and instances (figure 11.1).

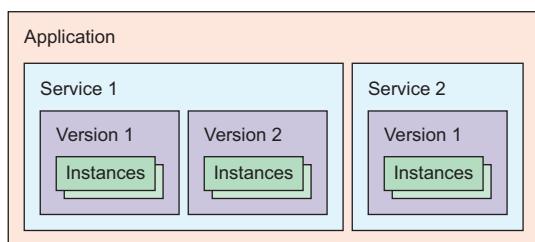


Figure 11.1 An overview of App Engine concepts

NOTE Keep in mind that although App Engine offers two environments, the concepts across both environments are the same (or very similar). You won't have to relearn these concepts to switch between environments.

In addition to looking at your application in terms of its components, App Engine keeps track of the versions of those components. For example, in your to-do list application, you might break the system into separate components: one component representing the web application itself and another responsible for recomputing statistics every day at midnight (figure 11.2). After that, revising a component from time to time (for example, fixing a bug in the web application) might bring about a new version of that component.

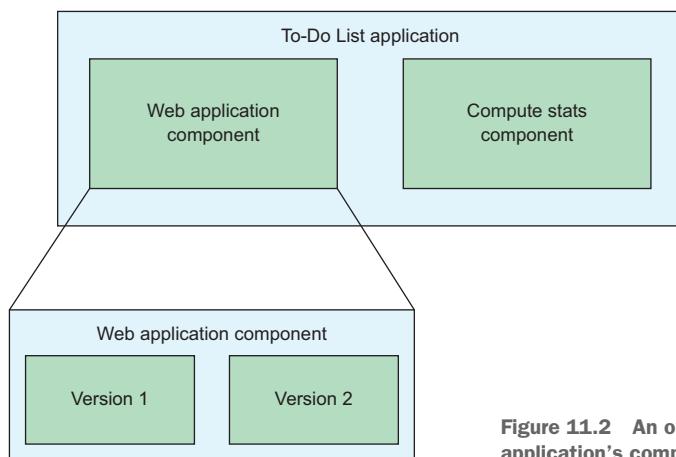


Figure 11.2 An overview of a to-do list application's components and versions

App Engine uses and understands all of these things (figure 11.1), and we'll explore them in more detail in this section.

Let's start at the top by looking at the idea of an App Engine *application*.

11.1.1 Applications

The basic starting place for using App Engine to host your work is the top-level application. Each of your projects is limited to one application, with the idea that each project should have one purpose. Like a project acts as a container for your Compute Engine VMs, the application acts as a container for your code, which may be spread across multiple services. (I'll discuss that in the next section.)

The application also has lots of settings. You can configure and change some of them easily, whereas others are permanent and locked to the application once set. For example, you can always reconfigure an SSL certificate for your application, but once you've chosen the region for your application, you can't change that.

NOTE The location of your application also impacts how much it costs to run, which we'll explore toward the end of the chapter.

To see how this works, click the App Engine section in the left-side navigation of the Cloud Console. If you haven't configured App Engine before, you'll be asked to choose a language (for example, Python); then you'll land on a page where you choose the location of your application (figure 11.3). This particular setting controls where the physical resources for your application will live and is an example of a setting that, once you choose it, you can't change.

The screenshot shows the 'App Engine' section of the Google Cloud Platform console. At the top, there's a header with the title 'Your first app with Python' and two tabs: '1 Select a location' (which is active) and '2 Deploy'. Below the tabs, a question asks 'In which region would you like to serve your app?'. A note explains that the app will be served from the selected region, with lower latency for users closer to the region. The main part of the screen is a world map with several regions highlighted by blue markers. The regions labeled are North America, Europe, Asia, South America, Oceania, and Africa. The map also shows the Atlantic Ocean, Indian Ocean, and Pacific Ocean. At the bottom of the map, there's a 'Google' logo and links for 'Map data ©2017' and 'Terms of Use'. Below the map, there's a dropdown menu labeled 'Select a region' with 'us-central' selected. A large blue 'Next' button is at the bottom.

Figure 11.3 Choosing a location for an App Engine application

Outside of that, the more interesting parts, such as services, are those that an App Engine application contains. Let's move on to looking at App Engine services.

11.1.2 Services

Services on App Engine provide a way to split your application into smaller, more manageable pieces. Similar to microservices, App Engine services act as independent components of computing, although they typically share access to the various shared App Engine APIs. For example, you can access the same shared cron API from any of the various services you might have as part of your application.

For example, imagine you're building a web application that tracks your to-do list. At first it might involve only simple text, but as you grow, you may want to add a feature that sends email reminders to finish something on your list. In that case, rather than trying to add the email reminder feature to the main application, you might define it as a separate service inside your application. Because its job is completely isolated from the main job of storing to-do items, it can live as a separate service and avoid cluttering the main application (figure 11.4).

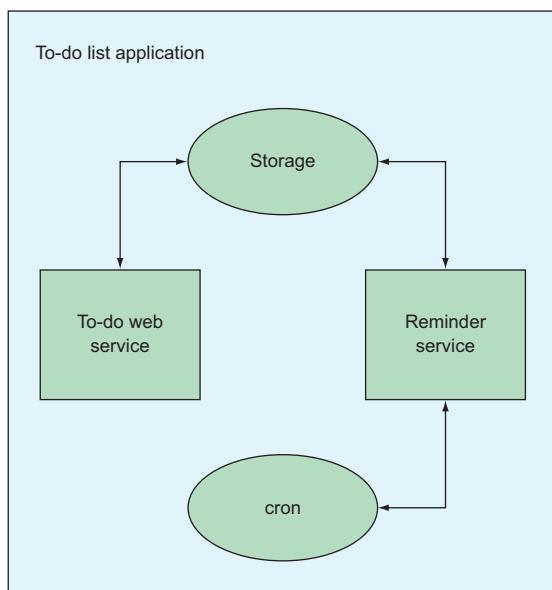


Figure 11.4 A to-do list application with two services

The service itself consists of your source code files and extra configuration, such as which runtime to use (for App Engine Standard). Unlike with applications (which have a one-to-one relationship with your project), you can create (deploy) as well as delete services. The first set of source code that you deploy on App Engine will create a new service, which App Engine will register as the default service for your application. When you make a request for your application without specifying the service, App Engine will route the request to this new default service.

Services also act as another container for revisions of your application. In the to-do list example, you could deploy new versions of your reminder service to your application

without having to touch the web application service. As you might guess, being able to isolate changes between related systems can be useful, particularly with large applications built by large teams, where each team owns a distinct piece of the application. Let's continue and look at how versions work.

11.1.3 Versions

Versions themselves are a lot like point-in-time snapshots of a service. If your service is a bunch of code inside a single directory, a version of your service corresponds to the code in that directory at the exact time that you decided to deploy it to App Engine. A neat side effect of this setup is that you can have multiple versions of the same service running at the same time.

Similar to how the first App Engine service you deploy becomes the default service for your entire application, the code that you deploy in that first service becomes the default *version* of that service. You can address an individual version of any given service, like you can address an individual service of your application. For example, you can see your web application by navigating to `webapp.my-list.appspot.com` (or explicitly, `default.webapp.my-list.appspot.com`), or you can view a newly deployed version (perhaps called version v2, as in figure 11.5) by navigating to `v2.webapp.my-list.appspot.com`.

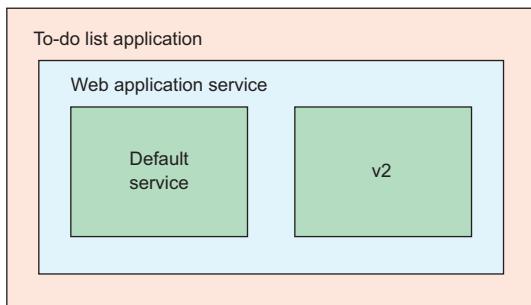


Figure 11.5 Deploying a new version of the web application service

I've covered the organizational concepts of applications, services, and versions. Now let's take a moment to look at an infrastructural one: instances.

11.1.4 Instances

Although we've looked at the organizational concepts in App Engine, you haven't seen how App Engine goes about running your code. Given what you've learned so far, it should come as no surprise that App Engine uses the concept of an instance to mean a chunk of computing capacity for your application. Unlike the concepts I've covered in this chapter so far, you'll find a couple of slight differences in the instances depending on whether you're using the Standard or Flexible environment, and they're worth exploring in a bit more detail (figure 11.6).

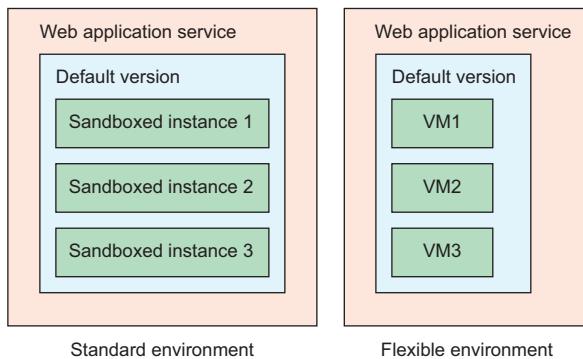


Figure 11.6 App Engine instances for Standard vs. Flexible environments

In App Engine Standard, these instances represent an abstract chunk of CPU and memory available to run your application inside a special sandbox. They scale up and down automatically based on how many requests are being sent to your application. Because they're lightweight sandbox environments, your application can scale from zero to thousands of instances quickly. You can choose the type of App Engine instance to use for your application from a list of available types that have varying costs and amounts of CPU and memory.

Because App Engine Flex is built on top of Compute Engine and Docker containers, it uses Compute Engine instances to run your code, which comes with a couple important caveats. First, because Compute Engine VMs take some time to turn on, Flex applications must always have at least a single VM instance running. As a result, Flex applications end up costing money around the clock. Because of the additional startup time, if you see a huge spike of traffic to your application, it might take a while to scale up to handle the traffic. During this time, existing instances could become overloaded, which would lead to timeouts for incoming requests.

It's also important to remember that App Engine instances are specific to a single version of your service, so a single instance only handles requests for the specific version of the service that received them. As a result, if you host lots of versions concurrently, those versions will spawn instances as necessary to service the traffic. If they're running inside App Engine Flex, each version will have at least one VM running at all times.

That finishes the summary of the concepts involved in App Engine. Now let's get down to business and look at how to use it.

11.2 Interacting with App Engine

At this point, you should have a decent understanding of the underlying organizational concepts that App Engine uses (such as services or versions), but that's not all that helpful until you do something with them. To that end, you'll create a simple "Hello, world!" application for App Engine, deploy it, and verify that it works. You can build the application for App Engine Standard first.

11.2.1 Building an application in App Engine Standard

As I discussed previously, App Engine Standard is a fully managed environment where your code runs inside a special sandbox rather than a full virtual machine, like it would on Compute Engine. As a result, you have to build your “Hello, world!” application using one of the approved languages. Of the languages available (PHP, Java, Python, and Go), Python seems a good choice (it’s powerful and easy to read), so for this section, you’re going to switch to using Python to build your application.

NOTE Don’t worry if you aren’t familiar with Python. I’ll annotate any Python code that isn’t super obvious to explain what it does.

One thing to keep in mind is that white space (for example, spaces and tabs) is important in Python. If you find yourself with syntax errors in your Python code, it could be that you used a tab when you meant to use four spaces, so be careful!

Before you get into building your application code, you first need to make sure you have the right tools installed. You’ll need them to deploy your code to App Engine.

INSTALLING PYTHON EXTENSIONS

To develop locally using App Engine (and specifically using App Engine Standard’s Python runtime), you’ll need to install the Python extensions for App Engine, which you can do using the `gcloud components` subcommand. This package contains the local development server, various emulators, and other resources and libraries you need to build a Python App Engine application:

```
$ gcloud components install app-engine-python
```

TIP If you installed the Cloud SDK using a system-level package manager (like `apt-get`), you’ll get an error message saying to use that same package manager and to run the command to install the Python extensions.

CREATING AN APPLICATION

With everything installed, you can get to the real work of building your application. Because you’re only testing out App Engine, you can start by focusing on nothing more than a “Hello, world!” application that sends a static response back whenever it receives a request. You’ll start your Python app by using the `webapp2` framework, which is compatible with App Engine.

NOTE You can use other libraries and frameworks as well (such as Django or Flask), but `webapp2` is the easiest to use with App Engine.

The next listing shows a simple `webapp2` application that defines a single request handler and connects it to the root URL (`/`). In this case, whenever you send a GET HTTP request to this handler, it sends back “Hello from App Engine!”

Listing 11.1 Defining your simple web application

```
import webapp2

classHelloWorld(webapp2.RequestHandler):
    defget(self):
        self.response.write('Hello from App Engine!');

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
])
```

You can put this code into a file called main.py; then you'll move over to defining the configuration of your App Engine application. The way you tell App Engine how to configure an application is with an app.yaml file. YAML (Yet Another Markup Language) has an easily readable syntax for some structured data that looks a lot like Markdown, and the app.yaml name is a special file that App Engine looks for when you deploy your application. It contains settings about the runtime involved, handlers for URL mappings, and more. You can see the app.yaml file that you'll use in the following listing.

Listing 11.2 Defining app.yaml

```
runtime: python27
api_version: 1
threadsafe: true

handlers:
- url: .*
  script: main.app
```

Tells App Engine to run your code inside the Python 2.7 sandbox

Tells App Engine which version of the API you're using. (Currently, there's only one version for Python 2.7, so this should be set to 1.)

Tells App Engine you've written your code to be threadsafe and App Engine can safely spawn multiple copies of your application without worrying about those threads tangling with each other

Section that holds the handlers that map URL patterns to a given script

Points to the main.py file, but the “app” suffix tells App Engine to treat main.py as a web server gateway interface (WSGI) application (which says to look at the app variable in main.py)

A regular expression that's matched against requests—if a request URL matches, the script will be used to handle the request.

At this point, you have everything you need to test out your application. It's time to try running it locally and making sure it works as you want.

TESTING THE APPLICATION LOCALLY

To run your application, you'll use the App Engine development server, which was installed as dev_appserver.py when you installed the Python App Engine extensions. Navigate to the directory that contains your app.yaml file (and the main.py file) and run dev_appserver.py pointing to the current directory (.). You should see some

debug output that says where the application itself is available (usually on localhost on port 8080). Once the development server is running with your application, you can test that it did the right thing by connecting to <http://localhost:8080/>:

```
$ curl http://localhost:8080/
Hello from App Engine!
```

So far so good! Now that you've built and tested your application, it's time to see whether you can deploy it to App Engine. After all, running the application locally isn't going to work when you have lots of incoming traffic.

DEPLOYING TO APP ENGINE STANDARD

Deploying the application to App Engine is easy because you have all the right tools installed. To do so, you can use the `gcloud app deploy` subcommand, confirm the place you're deploying to, and wait for the deployment to complete:

The project ID and the application ID are the same (because they have a one-to-one relationship).

\$ gcloud app deploy
Initializing App Engine resources...done.
Services to deploy:

descriptor: [/home/jjg/projects/appenginehello/app.yaml]
source: [/home/jjg/projects/appenginehello]
target project: [your-project-id-here]
target service: [default]
target version: [20171001t160741]
target url: [https://your-project-id-here.appspot.com]

Verifies the configuration of what you're planning to deploy

If no service name is set (which is the case here), App Engine uses the default service.

If no version name is specified, App Engine generates a default version number based on the date.

After deploying your application, it'll be available at a URL in the appspot.com domain.

Do you want to continue (Y/n)? Y

Beginning deployment of service [default]...
Some files were skipped. Pass `--verbosity=info` to see which ones.
You may also view the `gcloud log` file, found at
[/home/jjg/.config/gcloud/logs/2017.10.01/16.07.33.825864.log].

Asks you to confirm the deployment parameters to avoid accidentally deploying the wrong code or to the wrong service

Uploading 2 files to Google Cloud Storage

File upload done.
Updating service [default]...done.
Waiting for operation [apps/your-project-id-here/operations/1fad9f55-35bb
↳ -45e2-8b17-3cc8cc5b1228] to complete...done.
Updating service [default]...done.
Deployed service [default] to [https://your-project-id-here.appspot.com]

You can stream logs from the command line by running:
\$ gcloud app logs tail -s default

To view your application in the web browser run:
\$ gcloud app browse

Once this is completed, you can verify that everything worked either by using the curl command or through your browser by sending a GET request to the target URL from the deployment information. The curl command yields the following:

```
$ curl http://your-project-id-here.appspot.com
Hello from App Engine!
```

You also can verify that SSL works with your application by connecting using https:// as the scheme instead of plain http://:

```
$ curl https://your-project-id-here.appspot.com
Hello from App Engine!
```

Lastly, because the service name is officially “default,” you can address it directly at default.your-project-id-here.appspot.com:

```
$ curl http://default.your-project-id-here.appspot.com
Hello from App Engine!
```

You can check inside the App Engine section of the Cloud Console and see how many requests have been sent, how many instances you currently have turned on, and more. An example of what this might look like is shown in figure 11.7.

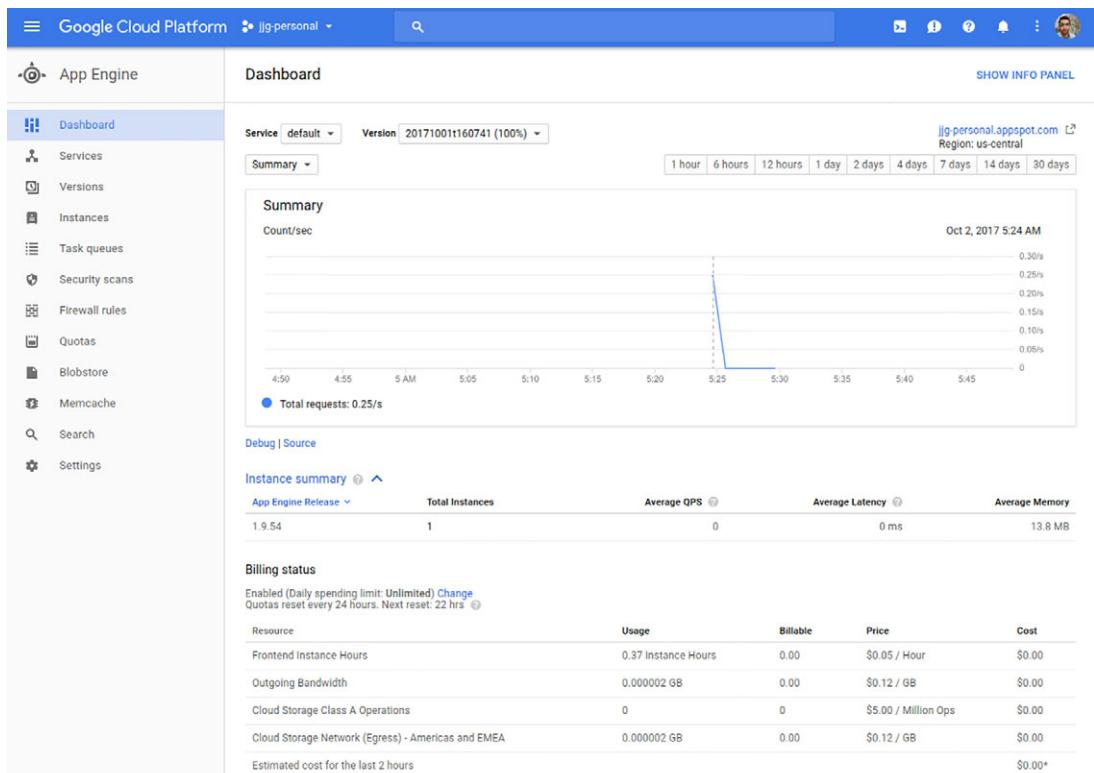


Figure 11.7 The App Engine overview dashboard in the Cloud Console

How do you create new services? Let's take a moment and explore how to deploy code to a service other than the default.

DEPLOYING ANOTHER SERVICE

In some ways, you can think of a new service as a new chunk of code, and you need to make sure you have a safe place to put this code. Commonly, the easiest way to set this up is to separate code chunks by directory, where the directory name matches up with the service name.

To see how this works, make two new directories called default and service2, and copy the app.yaml and main.py files into each directory. This effectively rearranges your code so you have two copies of both the code and the configuration in each directory.

To see this more clearly, here's how it should look when you're done:

```
$ tree
.
├── default
│   ├── app.yaml
│   └── main.py
└── service2
    ├── app.yaml
    └── main.py

2 directories, 4 files
```

Now you can do a few things to define a second service (and clarify that the current service happens to be the default):

- 1 Update both app.yaml files to *explicitly* pick a service name, so default will be called default.
- 2 Update service2/main.py to print something else.
- 3 Redeploy both services.

After you update both app.yaml files, they should look like the following two listings.

Listing 11.3 Updated default/app.yaml

```
runtime: python27
api_version: 1
threadsafe: true
service: default
    ←
handlers:
  - url: /*
    script: main.app
```

Explicitly states that the service involved is the default one—this has no real effect in this case, but it clarifies what this app.yaml file controls.

Listing 11.4 Updated service2/app.yaml

```
runtime: python27
api_version: 1
threadsafe: true
service: service2
    ←
Chooses a new service name, which can be any ID-style string that you want
```

```
handlers:
- url: /.*
  script: main.app
```

As you can see, you've made it explicit that each app.yaml file controls a different service. You've also made sure the service name matches the directory name, meaning it's easy to keep track of all of the different source code and configuration files.

Next, you can update service2/main.py, changing the output so you know it came from this other service. Doing this might make your application look like the following listing.

Listing 11.5 The service2 "Hello, world!" application in Python

```
import webapp2

classHelloWorld(webapp2.RequestHandler):
    defget(self):
        self.response.write('Hello from service 2!');

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
])
```

Makes it clear that service2 is responding.

Finally, you can deploy your new service by running gcloud app deploy and pointing at the service2 directory instead of the default directory:

```
$ gcloud app deploy service2
Services to deploy:
descriptor:      [/home/jjg/projects/appenginehello/service2/app.yaml]
source:          [/home/jjg/projects/appenginehello/service2]
target project:  [your-project-id-here]
target service:   [service2]
target version:  [20171002t053446]
target url:      [https://service2-dot-your-project-id
                  -here.appspot.com]
# ... More information here ...
```

Points the gcloud deployment tool to your service2 directory

As a result, the deployment tool looks for the copied app.yaml file, which states a different service name.

Figures out the service name should be service2 as you defined it

Because the service name isn't "default," you get a separate URL where you can access your code.

Like before, your new application service should be live. And at this point, your system conceptually looks a bit like figure 11.8.

You can verify that the deployment worked by navigating to the URL again in a browser, or you can use the command line:

```
$ curl https://service2-dot-your-project-id-here.appspot.com
Hello from service 2!
```

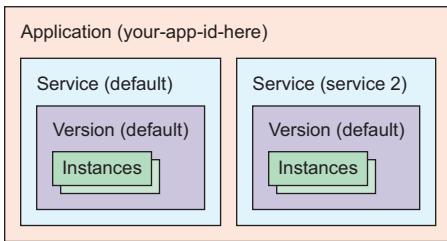


Figure 11.8 Organizational layout
of your application so far

If this URL looks strange to you, that's not unusual. The syntax of <service>.dot-<application> is definitely new. This exists because of how SSL certificates work. App Engine ensures that *.appspot.com is secured but doesn't allow additional dots nested deeper in the DNS hierarchy. When accessing your app over HTTP (not HTTPS), you technically can make a call to <service>.<application>.appspot.com, but if you were to try that with HTTPS, you'd run into trouble:

```
$ curl http://service2.your-project-id-here.appspot.com
Hello from service 2!
$ curl https://service2.your-project-id-here.appspot.com
# Error
```

The result is an error code due to the SSL certificate not covering the domain specified.

You've seen how to deploy a new service. Now let's look at a slightly less adventurous change by deploying a new version of an existing service.

DEPLOYING A NEW VERSION

Although you may only create new services once in a while, any update to your application will probably result in a new version, and updates happen far more often. So how do you update versions? Where does App Engine store the versions?

To start, confirm how your application is laid out. You can inspect your current application either in the Cloud Console or from the command line:

```
$ gcloud app services list
SERVICE  NUM VERSIONS
default   1
service2  1

$ gcloud app versions list
SERVICE  VERSION      SERVING_STATUS
default   20171001t160741  SERVING
service2  20171002t053446  SERVING
```

As you can see, you currently have two services, each with a single default version specified. Now imagine that you want to update the default service, but this update should create a new version and not overwrite the currently deployed version of the service.

To update the service in this way, you have two options. The first is to rely on App Engine's default version naming scheme, which is based on the date and time when

the deployment happened. When you deploy your code, App Engine creates a new version automatically for you and never overwrites the currently deployed version, which is helpful when you accidentally deploy the wrong code! The other is to use a special flag (-v) when deploying your code, and the result will be a new version named as you specified.

If you want to update your default version, you can make your code changes and deploy it like you did before. In this example, you'll update the code to say, "Hello from version 2!" Once the deployment completes, you can verify that everything worked as expected by trying to access the URL as before:

```
$ curl https://your-project-id-here.appspot.com
Hello from version 2!
```

This might look like you've accidentally blasted out the previous version, but if you inspect the list of versions again, you'll see that the previous version is still there and serving traffic:

```
$ gcloud app versions list --service=default
SERVICE  VERSION          TRAFFIC_SPLIT  SERVING_STATUS
default   20171001t160741  0.00          SERVING
default   20171002t072939  1.00          SERVING
```

Notice that the traffic split between the two versions has shifted, and all traffic is pointing to the later version, with zero traffic being routed to the previous version. I'll discuss this in more detail later on. If the version is still there, how can you talk to it? It turns out that just as you can access a specific service directly, you can access the previous version by addressing it directly in the format of <version>.<service>.your-project-id-here.appspot.com (or using -dot- separators for HTTPS):

```
$ curl http://20171001t160741.default.your-project-id-here.appspot.com
Hello from App Engine!
$ curl https://20171001t160741-dot-default-dot-your-project-id
➥ -here.appspot.com
Hello from App Engine!
```

It's completely reasonable if you're worried about a new version going live right away. You do have a way to tell App Engine that you want the new version deployed but don't want to immediately route all traffic to the new version. You can update the code again to change the message and deploy another version, without it becoming the live version immediately. To do so, you'll set the `promote_by_default` flag to false:

```
$ gcloud config set app/promote_by_default false
Updated property [app/promote_by_default] .

$ gcloud app deploy default
Services to deploy:

descriptor:      [/home/jjg/projects/appenginehello/default/app.yaml]
source:         [/home/jjg/projects/appenginehello/default]
```

```

target project: [your-project-id-here]
target service: [default]
target version: [20171002t074125]
target url: [https://20171002t074125-dot-your-project-id
➥ -here.appspot.com]

(add --promote if you also want to make this service available from
[https://your-project-id-here.appspot.com] )

# ... More information here ...

```

At this point, the new service version should be deployed but not live and serving requests. You can look at the list of services to verify that, as follows, or check by making a request to the target URL as you did before:

```

$ gcloud app versions list --service=default
SERVICE VERSION TRAFFIC_SPLIT SERVING_STATUS
default 20171001t160741 0.00 SERVING
default 20171002t072939 1.00 SERVING
default 20171002t074125 0.00 SERVING

$ curl http://your-app-id-here.appspot.com/
Hello from version 2!

```

You also can verify that the new version was deployed correctly by accessing it the same way you did before:

```

$ curl http://20171002t074125.default.your-project-id-here.appspot.com
Hello from version 3, which is not live yet!

```

Once you see that the new version works the way you expect, you can safely promote it by migrating all traffic to it using the Cloud Console. To do this, you browse to the list of versions, check the version you want to migrate traffic to, and click Migrate Traffic at the top of the page (figure 11.9).

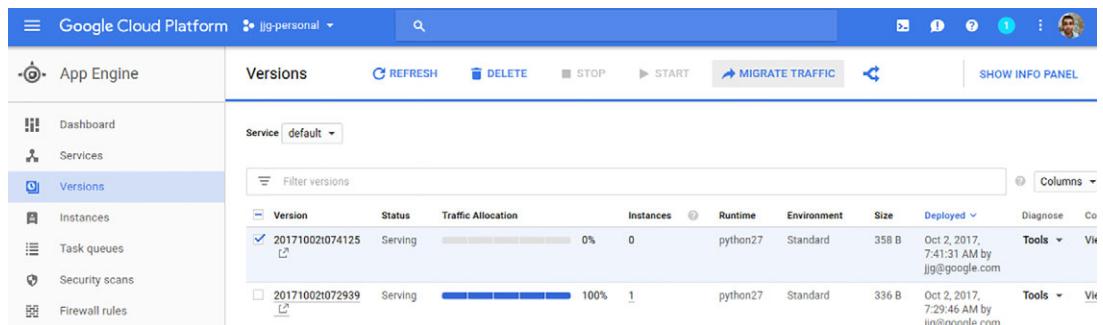


Figure 11.9 Checking the box for the version and clicking Migrate Traffic

When you click the button, you'll see a pop up where you can confirm that you want to route all new traffic to the selected version (figure 11.10).

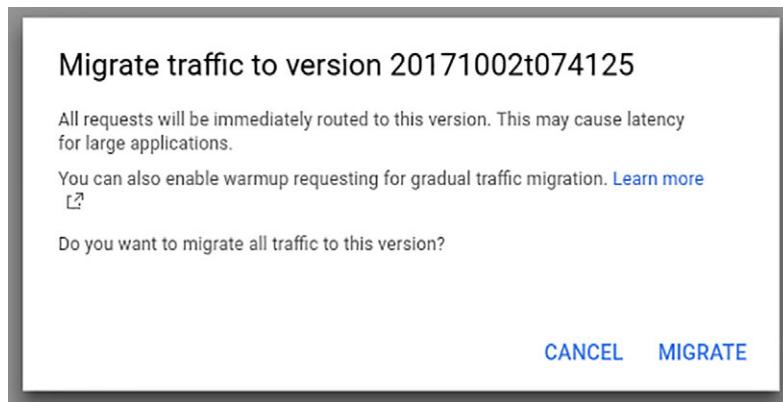


Figure 11.10 Pop up to confirm you want to migrate traffic to the new version

When this is complete, you'll see that 100% of traffic is being sent to your new version:

```
$ gcloud app versions list --service=default
SERVICE  VERSION          TRAFFIC_SPLIT  SERVING_STATUS
default   20171001t160741  0.00          SERVING
default   20171002t072939  0.00          SERVING
default   20171002t074125  1.00          SERVING

$ curl https://your-project-id-here.appspot.com
Hello from version 3, which is not live yet!
```

Obviously, it's
live now!

You've seen how deployment works on App Engine Standard Environment. Now let's take a detour and look at how things work in the Flexible Environment.

11.2.2 On App Engine Flex

As I discussed previously, whereas App Engine Standard is limited to some of the popular programming languages and runs inside a sandbox environment, App Engine Flex is based on Docker containers, so you can use any programming language you want. You get to switch back to Node.js when building your "Hello, world!" application. Let's get started!

CREATING AN APPLICATION

Similarly to the example I used when building an application for App Engine Standard, you'll start by building a "Hello, world!" application using Express (a popular web development framework for Node.js).

NOTE You can use any web framework you want. Express happens to be popular and well documented, so I'll use that for the example.

First, create a new directory called default-flex to hold the code for this new application, alongside the other directories you have already. After that, you should initialize the application using `npm` (or `yarn`) and add `express` as a dependency:

```
$ mkdir default-flex
$ cd default-flex
$ npm init
# ...
Wrote to /home/jjg/projects/appenginehello/default-flex/package.json:

{
  "name": "appengineflexhello",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}

$ npm install express
# ...
```

Now that you've defined your package and set the dependencies you need, you can write a simple script that uses Express to handle HTTP requests. This script, shown in the following listing, which you'll put inside `app.js`, will take any request sent to `/` and send "Hello, world!" as a response.

Listing 11.6 Defining a simple “Hello, world!” application in Node.js

```
'use strict';

const express = require('express');
const app = express();                                ← Uses the Express framework for Node.js

app.get('/', (req, res) => {
  res.status(200).send('Hello from App Engine Flex!').end();   ← Sets the response code to 200 OK and returns a short "Hello, world!" type message

});
```

```
const PORT = process.env.PORT || 8080;           ← Tries to read a port number from the environment, but defaults to 8080 if it's not set
app.listen(PORT, () => {
  console.log(`App listening on port ${PORT}`);
  console.log('Press Ctrl+C to quit.');
});
```

Once you've written this script, you can test that your code works by running it and then trying to connect to it using `curl`, as shown here, or your browser:

```
$ node app.js
App listening on port 8080
Press Ctrl+C to quit.

$ curl http://localhost:8080
Hello from App Engine Flex!
```

This is executed from a separate terminal on the same machine.

Now that you’re sure the code works, you can get back to work on deploying it to App Engine. Like before, you’ll need to also define a bit of configuration that explains to App Engine how to run your code. Like with App Engine standard, you’ll put the configuration options in a file called `app.yaml`. The main difference here is that because a Flex-based application is based on Docker, the configuration you put in `app.yaml` is far less involved:

```
runtime: nodejs
env: flex
service: default
```

A new parameter called env explains to App Engine that you intend to run your application outside of the standard environment.

For this example, you'll deploy the Flex service on top of the Standard service.

As you can see, this configuration file is far simpler than the one you used when building an application to App Engine Standard. App Engine Flex only needs to know how to take your code and put it into a container. Next, instead of setting up routing information, you need to tell App Engine how to start your server. App Engine Flex’s `nodejs` runtime will always try to run `npm start` as the initialization command, so you can set up a hook for this in your `package.json` file that executes `node app.js`, as shown in the following listing.

NOTE This format I’m demonstrating is a feature of `npm` rather than App Engine. All App Engine does is call `npm start` when it turns on the container.

Listing 11.7 Adding a start script to package.json

```
{
  "name": "appengineflexhello",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "dependencies": {
    "express": "^4.16.1"
  },
  "scripts": {
    "start": "node app.js"
  }
}
```

That should be all you need. The next step here is to deploy this application to App Engine.

DEPLOYING TO APP ENGINE FLEX

As you might guess, deploying an application to App Engine Flex is similar to deploying one to App Engine Standard. The main difference you'll notice at first is that it takes a bit longer to complete the deployment. It takes more time primarily because App Engine Flex builds a Docker container from your application code, uploads it to Google Cloud, provisions a Compute Engine VM instance, and starts the container on that instance. This is quite a bit more to do than if you're using App Engine Standard, but the process itself from your perspective is the same, and you can do it with the `gcloud` command-line tool:

```
$ gcloud app deploy default-flex
Services to deploy:

descriptor:      [/home/jjg/projects/appenginehello/default-flex/app.yaml]
source:          [/home/jjg/projects/appenginehello/default-flex]
target project:  [your-project-id-here]
target service:  [default]
target version:  [20171002t104910]
target url:      [https://your-project-id-here.appspot.com]

(add --promote if you also want to make this service available from
[https://your-project-id-here.appspot.com] )

# ... More information here ...
```

TIP If you followed along with all of the code examples in the previous section, you might want to undo the change you made to the `promote_by_default` configuration setting by running `gcloud config set app/promote_by_default true`. Otherwise, when you attempt to visit your newly deployed application, it'll still be served by the previous version you deployed.

Now that App Engine Flex has deployed your application, you can test that it works the same way you tested your application on App Engine Standard:

```
$ curl https://your-project-id-here.appspot.com/
Hello from App Engine Flex!
```

What might surprise you at this point is that you actually deployed a new version of the default service, which uses a completely different runtime. You have two versions running right now, one using the Standard environment and the other using the Flexible environment. You can see this by listing the versions of the default service again:

```
$ gcloud app versions list --service=default
SERVICE  VERSION      TRAFFIC_SPLIT  SERVING_STATUS
default   20171001t160741  0.00        SERVING
default   20171002t072939  0.00        SERVING
default   20171002t074125  0.00        SERVING
default   20171002t104910  1.00        SERVING
```

The previously live
default version on
App Engine Standard

Your new version
running on App Engine
Flex alongside the
other versions

You can access your previous service from App Engine Standard by addressing it directly, as you did before:

```
$ curl http://20171002t074125.default.your-project-id-here.appspot.com/
Hello from version 3, which is not live yet!
```

As you might expect, deploying other services and versions on Flex is identical to how you just learned using App Engine Standard. To demonstrate this, you can deploy one last new service named `service3`, putting all of your code side by side. To start, you'll copy and paste the code for `default-flex` into `service3`:

```
$ tree -L 2 .
.
├── default
│   ├── app.yaml
│   └── main.py
├── default-flex
│   ├── app.js
│   ├── app.yaml
│   ├── node_modules
│   └── package.json
└── service2
    ├── app.yaml
    └── main.py
└── service3
    ├── app.js
    ├── app.yaml
    ├── node_modules
    └── package.json
```

At this point, you'll have to make two changes. The first is to the `app.yaml` file, where you should change the service name to `service3`. The second is to update your `app.js` code so it says, “Hello from service 3!” The updated contents for both files are shown in the following two listings.

Listing 11.8 Updated app.yaml for the new service

```
runtime: nodejs
env: flex
service: service3
```

You want this to deploy as
a new service, so you call
it `service3`.

Listing 11.9 Updated app.js for the new service

```
'use strict';

const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.status(200).send('Hello from service 3!'); ←
});
```

Updates the response
in your application to
state that it's coming
from the new service

```
const PORT = process.env.PORT || 8080;
app.listen(PORT, () => {
  console.log(`App listening on port ${PORT}`);
  console.log('Press Ctrl+C to quit.');
});
```

Once you've done that, you can move into the parent directory and deploy the new service with gcloud app deploy service3 (because you named the directory service3):

```
gcloud app deploy service3
Services to deploy:

descriptor:      [/home/jjg/projects/appenginehello/service3/app.yaml]
source:          [/home/jjg/projects/appenginehello/service3]
target project:  [your-project-id-here]
target service:   [service3]
target version:  [20171003t062949]
target url:      [https://service3-dot-your-project-id-here.appspot.com]

# ... More information here ...
```

And now you can test that everything works by using curl, as follows, or your browser to talk to all of the various services you've deployed:

```
$ curl http://service3.your-app-id-here.appspot.com
Hello from service 3!

$ curl http://service2.your-app-id-here.appspot.com
Hello from service 2!

$ curl http://your-app-id-here.appspot.com
Hello from App Engine Flex!
```

You've now deployed multiple services on App Engine Flex. Let's try digging even deeper into deploying services with custom runtime environments.

DEPLOYING CUSTOM IMAGES

So far, you've always relied on the built-in runtimes (in this case, nodejs), but because App Engine Flex is based on Docker containers (which I discussed in chapter 10), technically you can use any container you want! To see how this works, try building an entirely different type of "Hello, world!" application relying on a typical Apache web server. To do this, the first thing you should do is create another directory named after your service. In this case, you can call it custom1 and put it right next to the other directories for the other services in your application.

Once you've created the directory, you'll need to define a Dockerfile that defines your application, as shown in the following listing. Because you're only trying to demonstrate how this works, keep it simple and stick with using Apache to serve a static file.

Listing 11.10 A Dockerfile to run your application

```

FROM ubuntu:16.04           ← The base image (and
                            operating system) will
                            be a plain version of
                            Ubuntu 16.04.

RUN apt-get update && apt-get install -y apache2      ← Updates packages and
                                                          installs Apache

# Set Apache to listen on port 8080
# instead of 80 (what App Engine expects)
RUN sed -i 's/Listen 80/Listen 8080/' /etc/apache2/ports.conf
RUN sed -i 's/:80/:8080/' /etc/apache2/sites-enabled/000-default.conf

# Add our content
COPY hello.html /var/www/html/index.html      ← Copies the hello.html file—you'll write
                                                that shortly—over the Apache static
                                                content directory and makes sure it's
                                                readable by the world

RUN chmoda+r /var/www/html/index.html

EXPOSE 8080                  ← Tells Docker to expose
                                port 8080 to the outside

CMD ["apachectl", "-D", "FOREGROUND"]          ← Starts the Apache service
                                                in the foreground

```

NOTE Don't worry if you don't understand all of this completely. If you're interested in using custom runtime environments like this, you should probably read up on Dockerfile syntax, but it's not a requirement to use App Engine.

Now that you have a simple Dockerfile that serves some content, the next thing you'll need to do is update your app.yaml file to rely on this custom runtime. To do that, you'll replace nodejs in your previous definition with custom. This tells App Engine to look for a Dockerfile and use that instead of the built-in nodejs Dockerfile you used previously:

```

runtime: custom
env: flex
service: custom1

```

The last thing you'll need to do is define the static file you want to serve, which is pretty easy. Write some simple HTML that says, “Hello from Apache!” as follows.

Listing 11.11 A simple “Hello, world” HTML file

```

<html>
  <body>
    <h1>Hello from Apache!</h1>
  </body>
</html>

```

At this point, your directory should look something like this:

```
$ tree . -L 2
.
├── custom1
│   ├── app.yaml
│   ├── Dockerfile
│   └── hello.html
├── default
│   ├── app.yaml
│   └── main.py
├── default-flex
│   ├── app.js
│   ├── app.yaml
│   ├── node_modules
│   └── package.json
├── service2
│   ├── app.yaml
│   └── main.py
└── service3
    ├── app.js
    ├── app.yaml
    ├── node_modules
    └── package.json
```

You're treating this custom runtime environment like any of your other services.

Notice that your new service has no code in it and is made up entirely of the Dockerfile, some static content, and the App Engine configuration. Although this is valid, it's unlikely that a real App Engine application would be this simple.

You can test that your code works using Docker. First, you build the container image using `docker build custom1`, and then you start the container and verify that Apache is doing what you expect:

```
$ docker build custom1
Sending build context to Docker daemon 4.608 kB
Step 1 : FROM ubuntu:16.04
16.04: Pulling from library/ubuntu
Builds the container, which will package everything up into a single Docker image

# ... Lots of work here ...
Successfully built 431cf4c10b5b
Runs the container using the ID of the image and links your local machine's port 8080 to the container's port 8080

$ docker run -d -p 8080:8080 431c
e149d89e7f619f0368b0e205d9a06b6d773d43d4b74b61063d251e0df3d49f66

$ docker ps --format "table {{.ID}}\t{{.Status}}"
CONTAINER ID        STATUS
e149d89e7f61        Up 17 minutes
Verifies that the container is running

$ curl localhost:8080
<html>
  <body>
    <h1>Hello from Apache!</h1>
  </body>
</html>
```

Connects to the container over HTTP on port 8080 and shows that it serves the correct HTML content that you wrote

Now that you've checked that your application works, you can deploy it to App Engine using the same deploy command you've used before:

```
$ gcloud app deploy custom1
Services to deploy:

descriptor:      [/home/jjg/projects/appenginehello/custom1/app.yaml]
source:          [/home/jjg/projects/appenginehello/custom1]
target project:  [your-project-id-here]
target service:   [custom1]
target version:  [20171003t123015]
target url:      [https://custom1-dot-your-project-id-here.appspot.com]

# ... More information here ...
```

To verify that the deployment worked, you can make the same request again to `custom1.your-project-id-here.appspot.com` and see that the result is the HTML file you wrote previously. The moral of the story here is that you can run and scale on App Engine Flex anything that fits into a Docker container. Anything you can run on your local machine also can run on App Engine, like you saw with Compute Engine and Kubernetes Engine.

WARNING If you deployed an application using App Engine Flex, compute resources are running under the hood regardless of whether anyone is sending requests to your application.

If you don't want to be billed for these resources, make sure to stop any running Flex versions. You can do this in the App Engine section of the Cloud Console by choosing Versions in the left-side navigation, checking the boxes for running versions, and clicking the Stop button at the top of the page.

Now that you've seen all the many ways that you can build applications for App Engine, there's one topic that I've sort of glossed over: scaling. How exactly do you control how many instances are running at a given time? Let's take some time to run through that on both App Engine Standard and App Engine Flex.

11.3 **Scaling your application**

So far, you've sort of assumed that all the scaling on App Engine was taken care of for you. Although that's mostly true, you do have lots of ways you can configure both the scaling style and the underlying instances that run your code.

As I discussed previously, when it comes to scaling and instance configuration, App Engine Standard and App Engine Flex have a few differences. We'll look at each of them individually. Keep in mind that you can only choose one type of scaling for any given service, so it's important to understand how they all work and to choose the one that best suits what your application needs. Let's look in more detail at how you can control how scaling works on both of the App Engine environments.

11.3.1 Scaling on App Engine Standard

So far, you've deployed all of your services without any mention of how to scale them—you've relied on the default options. But App Engine Standard has quite a few scaling options that you can fine-tune so they fit your needs. Let's start by looking at the default option that's also one of the main features of App Engine: automatic scaling.

AUTOMATIC SCALING

The default scaling option is automatic, so App Engine will decide when to turn on new instances for you based on a few metrics, such as the number of concurrent requests, the longest a given request should wait around in a queue to be handled, and the number of instances that can be idle at any given time. These all have default settings, but you can change them in `app.yaml`. Let's start by looking at the simplest of the settings: idle instances.

Idle instances

App Engine Standard instances are only chunks of CPU and memory rather than a full virtual machine (and each of these instances costs money, which we'll look at later). Because of this arrangement, App Engine Standard provides a way for you to decide the minimum and maximum number of instances that can sit idle waiting for requests before being turned off.

In a way, this setting is a bit like choosing how many buffer instances to keep around that aren't actively in use. For example, imagine you deploy a service that gets enough traffic to keep three instances busy (figure 11.11), followed by a period when only one instance is busy. If you set the minimum idle instances to 2 (and maximum to 3), you'll end up with one busy instance and two idle instances that sit around doing nothing except waiting for more requests (figure 11.12).

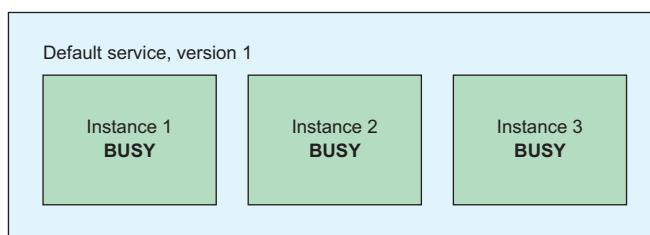


Figure 11.11 A service keeping all three instances busy

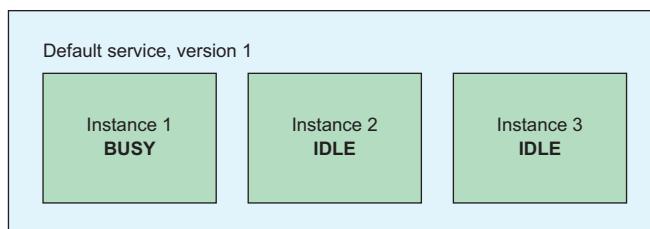


Figure 11.12 The same service with two idle instances

On the other hand, if you set the minimum and maximum idle instances both to 1, then App Engine will turn off instances until there's exactly one sitting idle waiting for requests. In this case, that would mean you'd have one busy instance and one idle instance (figure 11.13).

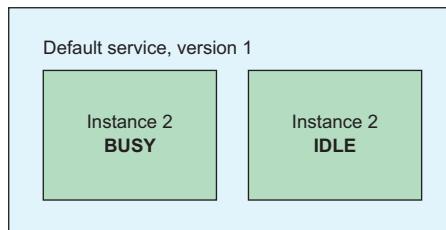


Figure 11.13 The service with one idle instance kept around and the other terminated

In this example scenario, you'd update your app.yaml file with a category called `automatic_scaling` and fill in the `min_idle_instances` and `max_idle_instances` settings. The following listing shows what the first configuration I described for figure 11.11 might look like in real life.

Listing 11.12 Updated app.yaml with scaling based on idle instances

```
runtime: python27
api_version: 1
service: default
automatic_scaling:
  min_idle_instances: 2
  max_idle_instances: 3
```

Pending latency

When you make a request to App Engine, Google's frontend servers handle the request and ultimately route it to a specific instance running your service. But App Engine keeps a queue of requests in case some of them aren't ready to be handled yet. If no instance is available to respond to a given request immediately as it arrives, it can sit in a queue for a bit until an instance becomes available. Figure 11.14 shows an example of how you might think of the flow of a request through your App Engine service.

The flow begins with (1) lots of people making requests to the service. App Engine immediately queues up requests to be processed (2) in a standard work queue-style service, and ultimately individual instances handle them (3) to do the work the requests require. Hidden in this flow is an important metric to keep in mind: how long a request sits in that queue. Because App Engine can turn on more instances, if a request is sitting in a queue for too long, it seems like a good idea to turn on an instance. Doing so will help get through the queue of work more quickly. To help you set that up, App Engine lets you choose the minimum and/or maximum amount of time that any given request should spend sitting in this queue, which is called *pending latency*.

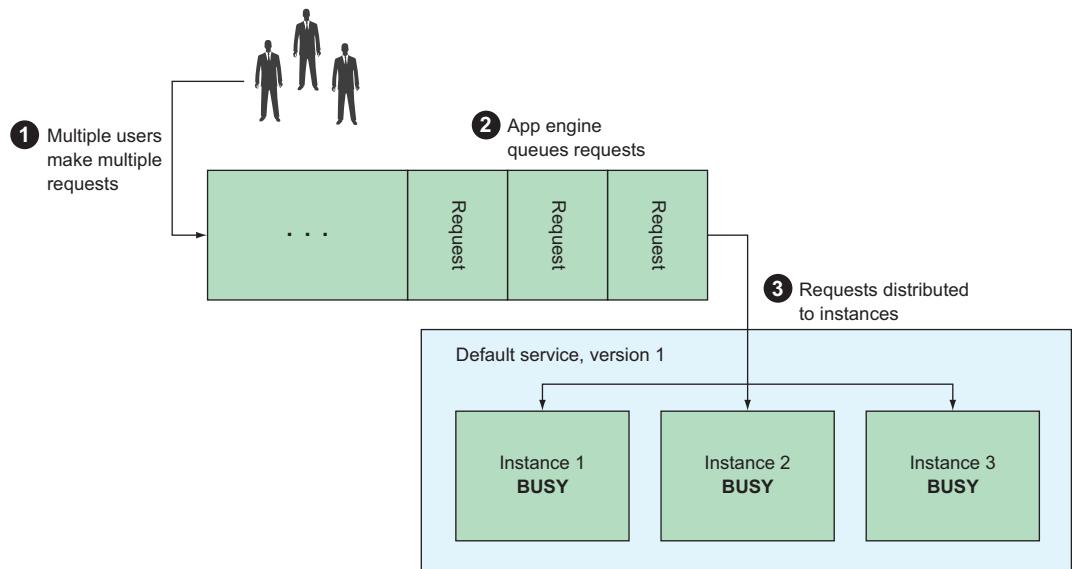


Figure 11.14 Requests queue up before being routed to an instance

If requests are spending more than the maximum pending latency in the queue, you should turn on more instances. For example, you might set it to 10 seconds, and App Engine will keep an eye on this metric, turning on new instances whenever the typical request spends more than 10 seconds in the queue. In general, a lower maximum pending latency means that App Engine will start instances more frequently.

The minimum pending latency is the way you set a lower bound when telling App Engine when it's OK to turn on more instances. When you set a minimum pending latency, it tells App Engine to not turn on a new instance if requests aren't waiting at least a certain amount of time. For example, you might set this to five seconds to ensure that you don't turn on new instances to handle requests that are only waiting a few seconds.

These settings are a bit like setting how stretchy a spring is. Lower values for both minimum and maximum mean the spring is super-stretchy (App Engine will expand capacity quickly to handle requests), and higher values mean the spring is much stiffer (App Engine will tolerate requests sitting in the queue for a while). In the example I described, with a minimum of 5 seconds and a maximum of 10 seconds of pending latency in the queue, the configuration would look like the following listing.

Listing 11.13 Updated app.yaml with scaling defined based on pending latency

```
runtime: python27
api_version: 1
service: default
automatic_scaling:
  min_pending_latency: 5s
  max_pending_latency: 10s
```

How these two settings interact can be confusing, so figure 11.15 shows the different cutoff points. As you can see, up until the minimum pending latency mark, App Engine will keep looking for an available instance and won't create a new one because of a request. After that minimum pending latency point has passed, App Engine will keep looking for an available instance but will consider itself free to create a new one if it makes sense to do so. If the request is still sitting in the queue by the maximum latency time, App Engine (if allowed by other parameters) will create a new instance to handle the request.

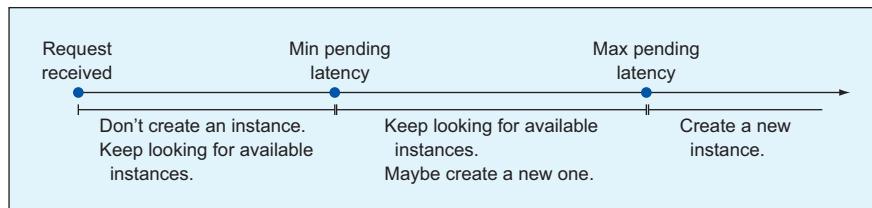


Figure 11.15 Time line of what actions are possible based on minimum and maximum pending latency

Requests spending more than the maximum amount of time in the queue will trigger App Engine to turn on more instances, acting as a sort of gas pedal that spurs more scaling. On the other hand, the minimum latency time acts more like a brake on scaling that prevents App Engine from turning on instances before they're needed.

Let's move on to the final metric you can control as part of automatic scaling. It has to do with the concurrency level of a given instance.

Concurrent requests

Because instances can handle more than one request at a time, the number of requests happening at once (level of concurrency) is another metric to use when autoscaling. App Engine allows you to set a concurrency level as a way to trigger turning more instances on or off, meaning you can set a target for how many requests an instance can handle at the same time before it's considered too busy. Obviously, a higher value here will try to send more requests to a single instance, which could overload it, but a super-low value here will leave the instances underutilized.

By default, App Engine will aim to handle eight requests concurrently on your instances, and although you can crank this all the way up to 80, it's worth testing and monitoring your instances to tune this number. Like the other settings, you change the concurrent request parameter with a setting inside your app.yaml file, as shown in the following listing.

Listing 11.14 Updated app.yaml file with scaling based on concurrent requests

```
runtime: python27
api_version: 1
service: default
```

```
automatic_scaling:  
  max_concurrent_requests: 10
```

I've covered the most common scaling options. Let's quickly look through the other, simpler scaling configurations.

BASIC SCALING

Basic scaling is another option that App Engine Standard provides and is sort of like a slimmed-down version of automatic scaling. Basic scaling has only two options to control, which are the maximum number of instances and how long to keep an idle instance around before turning it off. As you might guess, the maximum instances is a limit on the number of instances running at any given time, which is helpful if you're worried about being surprised by a large bill at the end of the month.

The next setting has to do with the termination policy for the instances handling requests to your service. As you learned before, it's likely that as traffic to your application fluctuates, instances that App Engine created to handle spikes of traffic might go idle during a lull. When you use basic scaling, you're able to specify how long those instances should sit idle before App Engine turns them off. By default, instances will sit idle for no longer than five minutes, but you're free to make that longer or shorter, depending on your needs. For example, to set basic scaling with no more than 10 instances, and a maximum idle time of three minutes, you can update your app.yaml file to look like the following listing.

Listing 11.15 Updated app.yaml file with basic scaling configured

```
runtime: python27  
api_version: 1  
service: default  
basic_scaling:  
  max_instances: 10  
  idle_timeout: 3m
```

As you can see, basic scaling does mean basic, because this type of scaling offers few options and none of them are particularly complicated. Let's look at an even simpler form of scaling: manual.

MANUAL SCALING

Manual scaling is a bit of a misnomer, because it's almost like no scaling at all. In this configuration, you tell App Engine exactly how many instances you want running at a given time. When you do this, all requests are routed to this pool of machines, which may become overwhelmed to the point where requests time out. As a result, this type of scaling should be for situations where you have a strict budget and don't care much whether your application is always available to your customers.

If you've decided that you want manual scaling, you choose the number of instances you want and update your app.yaml file. The example in the following listing shows what it would look like if you wanted to have exactly 10 instances running at all times for your service.

Listing 11.16 Updated app.yaml file showing manual scaling

```
runtime: python27
api_version: 1
service: default
manual_scaling:
  instances: 10
```

That covers how you scale your App Engine Standard services. Now let's look at how scaling works when using App Engine Flexible Environment.

11.3.2 Scaling on App Engine Flex

Because 1136.320 App Engine Flex is based on Docker containers and Compute Engine VMs, the scaling configurations should feel similar to the way we looked at autoscaling Compute Engine instances with instance groups and instance templates. Similar to App Engine Standard, Flex has two scaling options: automatic and manual. The only difference is that Flex is lacking the “basic” scaling option. Let's start by looking at the more common scaling method: automatic.

AUTOMATIC SCALING

App Engine Flex is capable of scaling your services up and down like you did previously when we looked at automatic scaling of Compute Engine instances. In addition, the parameters you can configure for App Engine Flex services are similar to how you handle Compute Engine's instance groups. Because all of the options are pretty straightforward, I'll run through them quickly and then demonstrate how they work together.

First, you can control the number of VM instances that can be running at any given time. As you've learned, at least one instance must be running at all times, but it's recommended to have a minimum of two instances to keep latency low in the face of traffic spikes (which happens to be the default). You also can set a maximum number of instances to avoid your application scaling out of control. By default, Flex services are limited to 20 instances, but you can increase or decrease that limit.

Next, you can control how App Engine decides whether additional instances are needed, which it does by looking at the CPU utilization and comparing it to a target. If the CPU usage is above the target, you'll get more instances, and if it's below the target, App Engine will turn some instances off. By default, App Engine Flex services will aim for a 50% utilization (0.5) across all of the running instances.

Aiming for a target is great, but turning instances on and off isn't an immediate action, which could cause some problems. Turning on a new instance might take a few seconds, so turning off an instance immediately after the utilization is low might not make a lot of sense. Luckily, App Engine Flex has a way to control how aggressively it terminates instances when the overall utilization comes in below the target amount, which is called the cooldown period. This setting controls how long to hold off after the utilization drops before terminating instances (by default, two minutes). As you'd expect, a

higher value here means you'll typically have excess capacity, whereas a lower value may lead to periods where requests queue up, waiting for available capacity.

We've gone through all of the automatic scaling settings for App Engine Flex. Now let's look at a sample configuration where you want to have somewhere between three and eight instances, with a CPU utilization target of 70% and a five-minute cooldown period, as follows.

Listing 11.17 Updated app.yaml showing a configuration of automatic scaling for Flex

```
runtime: nodejs
env: flex
service: default
automatic_scaling:
  min_num_instances: 3
  max_num_instances: 8
  cool_down_period_sec: 300 # 5 minutes * 60 = 300
  cpu_utilization:
    target_utilization: 0.7
```

MANUAL SCALING

Like App Engine Standard's manual scaling options, App Engine Flex has an option to decide up front exactly how many VM instances to run for your service. The syntax is identical, as shown in the following listing with an example of four VM instances.

Listing 11.18 Updated app.yaml file showing manual scaling

```
runtime: nodejs
env: flex
service: default
manual_scaling:
  instances: 4
```

I've covered all of the scaling options. Now it's time to look at what exactly you're scaling.

11.3.3 Choosing instance configurations

So far, I've talked about the number of instances and how to scale them, and I've asked you to think of instances as chunks of CPU and memory (either sandboxes or virtual machines), but we haven't looked at the details of the instances themselves. Let's explore what these instances are and how to choose instance configurations that suit your application, starting with App Engine Standard.

APP ENGINE STANDARD INSTANCE CLASSES

Because App Engine Standard involves running your code in a special sandbox environment, you'll need a way of configuring the computing power of that environment. To do so, you'll use a setting called `instance_class` in your `app.yaml` file. You can view the full list of instance class options in the App Engine documentation, but a few common options are listed in table 11.1.

Table 11.1 Resources for various App Engine instance classes

Name	Memory	CPU
F1	128 MB	600 MHz
F2	256 MB	1.2 GHz
F4	512 MB	2.4 GHz
F4_1G	1024 MB	2.4 GHz

By default, automatically scaled services use F1 instances. If you wanted to increase the instance class from the default F1 up to the F2 type, you could update your configuration, as follows.

Listing 11.19 Updated app.yaml file configuring a different instance class

```
runtime: python27
api_version: 1
service: default
instance_class: F2
```

In general, the best way to choose an instance class is to experiment (similar to how you'd choose the scaling parameters, such as minimum/maximum pending latency). After changing instance classes, you can look at performance characteristics using benchmarking tools to see what fits best.

Don't forget to adjust your concurrent requests scaling parameter when you're changing the instance class. Typically, larger classes can handle more concurrent requests (and vice-versa for small classes). This would mean that when you make the change in listing 11.19 to use F2 instances, you might also want to double the limit of concurrent requests for your service, as follows.

Listing 11.20 Adjusting instance class and concurrent request limits together

```
runtime: python27
api_version: 1
service: default
instance_class: F2
automatic_scaling:
  max_concurrent_requests: 16
```

Another general rule for choosing an instance class is that having more resources typically doesn't reduce the overall latency of requests (because of the typical pattern involving lots of I/O). Instead, it allows a single instance to handle more requests at the same time. If you're hoping to make a single request faster, instance class isn't guaranteed to fix that for you.

APP ENGINE FLEX INSTANCES

How does App Engine Flex let you define instances? Because App Engine Flex is based on Docker containers and Compute Engine instances, you get quite a bit more freedom when choosing virtual hardware. Although Compute Engine has specific instance types that you can customize to suit your projects, App Engine Flex sticks to the idea of declaring the resources you need and allowing App Engine itself to provision machines that match those needs.

Instead of saying, “I want this machine type,” you say, “I need at least two CPUs and at least 4 GB of RAM.” App Engine takes that and provisions a VM for your service that has *at least* those resources. (It may have more than that.) If you wanted to configure your service in the way I described (two CPUs, 4 GB of RAM), you’d update your app.yaml file to express this using a resources heading, as follows.

Listing 11.21 Updated app.yaml file configuring Compute Engine instance memory and CPU

```
runtime: nodejs
env: flex
service: default
resources:
  cpu: 2           | Sets the desired
  memory_gb: 4.0  | configuration
```

By default (if you leave these fields out entirely), you’ll get a single-core VM with 0.6 GB of RAM, which should be enough for relatively simple web applications. If you find your service is handling lots of memory-intensive work or computational work that can be easily parallelized and split across more cores, adding more memory or more CPU is likely a good idea.

As with Compute Engine, memory and CPU are related and are limited so they don’t stray too far from each other. For these instances, RAM in GB can be anywhere from 90% to 650% of the number of CPUs, but App Engine uses some of the memory (about 0.4 GB) for overhead on your instance. For the two CPUs you requested before, your VMs are limited to anywhere from 1.8 GB to 13 GB of RAM, so you can only access between 1.4 GB (1.8 - 0.4) and 12.6 GB (13 - 0.4) in this configuration.

In addition to setting the CPU and memory targets, you can choose the size of your boot disk and attach other temporary file system disks. (If your Docker image is larger than the default limit of 10 GB, you’ll need to increase this size to fit your image.)

WARNING Although App Engine Flex instances have a boot disk, you should consider this disk *temporary* because it’ll disappear anytime an instance is turned off.

Because different disk sizes have different performance characteristics, it might make sense to increase the size of your boot disk if your Docker image has lots of local data that you want to load up quickly. In the following listing, you can see how you might increase the size of the boot disk to 20 GB from the default of 10 GB.

Listing 11.22 Updating app.yaml to increase the size of instance boot disks

```
runtime: nodejs
env: flex
service: default
resources:
  disk_size_gb: 20
```



Doubles the boot disk size to 20 GB,
which will both store more data and
provide higher performance

At this point, we've explored in depth the computing environment that App Engine provides (both Standard and Flexible environments) and all of the infrastructural considerations to keep in mind when building applications on App Engine. What we haven't done is looked in detail at how you might write your services to make use of all the hosted services on App Engine. Let's spend some time exploring a few of App Engine's managed services and how you might use them to build out an application.

11.4 Using App Engine Standard's managed services

If you were building an application that stores data, you'd need to build the application itself, and then make sure you had a database server running as well that could hold the persistent data. App Engine aims to help make building applications easier by providing services (like storing data) that just work, so you don't have to worry about the surrounding infrastructure.

App Engine offers a lot of services and many ways to use them. If you're interested in digging into the details of each and every service, you may want to explore a book on Google App Engine to supplement this chapter. For now, I'll focus on a few of the important services, covering briefly how you can use them. I'll be limiting the discussion here to App Engine Standard, because App Engine Flex is just Compute Engine VMs. Let's get started by looking at the most common thing an application needs to do: store data.

11.4.1 Storing data with Cloud Datastore

As you learned in part 2 of this book, you can go about storing data using many methods, and Google Cloud Platform has many services available to help you do so. Even better, you can access these services from inside App Engine. Instead of telling you how each of the storage systems works (because each of them fills a whole chapter), I'll focus on how you might connect to the services from inside your App Engine application.

As you learned in chapter 5, Cloud Datastore is a nonrelational storage system that stores documents and provides ways to query them. To make life easy, it comes pre-baked into App Engine, with APIs built into the runtime. In the case of Python, App Engine Standard provides a Datastore API package called `ndb`, which acts as an ORM (object-relational mapping) tool. You won't even scratch the surface of what `ndb` can do, but listing 11.23 shows how you might define a `TodoList` model and interact with entities in Datastore. It starts by defining a `model`, which is the type of an entity, creates a new to-do list, queries the available lists, and then deletes the list it created.