Chapter 14. MySQL in the Cloud

“No need to worry, it’s in the cloud” is a phrase we often hear. It reminds us of a story about a woman who was worried that, after her iPhone drowned in the toilet, she’d lost all her years’ worth of family and travel photos. To her surprise, when she bought a new phone, the device “recovered” all the photos. She was using the iCloud backup solution from Apple to back up her device content to the cloud. (The other surprise may have been the service subscription bill she hadn’t realized she was paying.)

As computer engineers, we don’t have the luxury of taking risks with regard to whether our data will be recovered or not. Cloud storage is a scalable and reliable solution. In this chapter, we will look at a few options that companies have for using MySQL in the cloud. These range from database-as-a-service (DBaaS) options that are easily scalable and provide automatic backup and high availability features to more traditional choices like EC2 instances, which provide more fine-grained control. In general, startup companies, where the core business is not technology, prefer to use DBaaS options since they are easier to implement and work with. On the other hand, companies that need more strict control over their data might prefer to use an EC2 instance or their own cloud infrastructure.

Database-as-a-Service (DBaaS)

DBaaS is an outsourcing option where companies pay a cloud provider to launch and maintain a cloud database for them. Payment is usually per-usage, and the data owners can access their application data as they please. A DBaaS provides the same functionalities as a standard relational or non-relational database. It’s often a good solution for companies trying to avoid configuring, maintaining, and upgrading their databases and servers (although this is not always true). DBaaS lives in the realm of software-as-a-service (SaaS), like platform-as-a-service (PaaS) and infrastructure-as-a-service (IaaS), where products like databases become services.

Amazon RDS for MySQL/MariaDB

The most popular DBaaS is Amazon RDS for MySQL. Getting started with RDS is almost like configuring a new car on a website. You choose the main product and add the options you want until it looks the way you like and then launch. Figure 14-1 shows the products available. In this case, we will go for MySQL (the MariaDB version has similar settings for deployment).

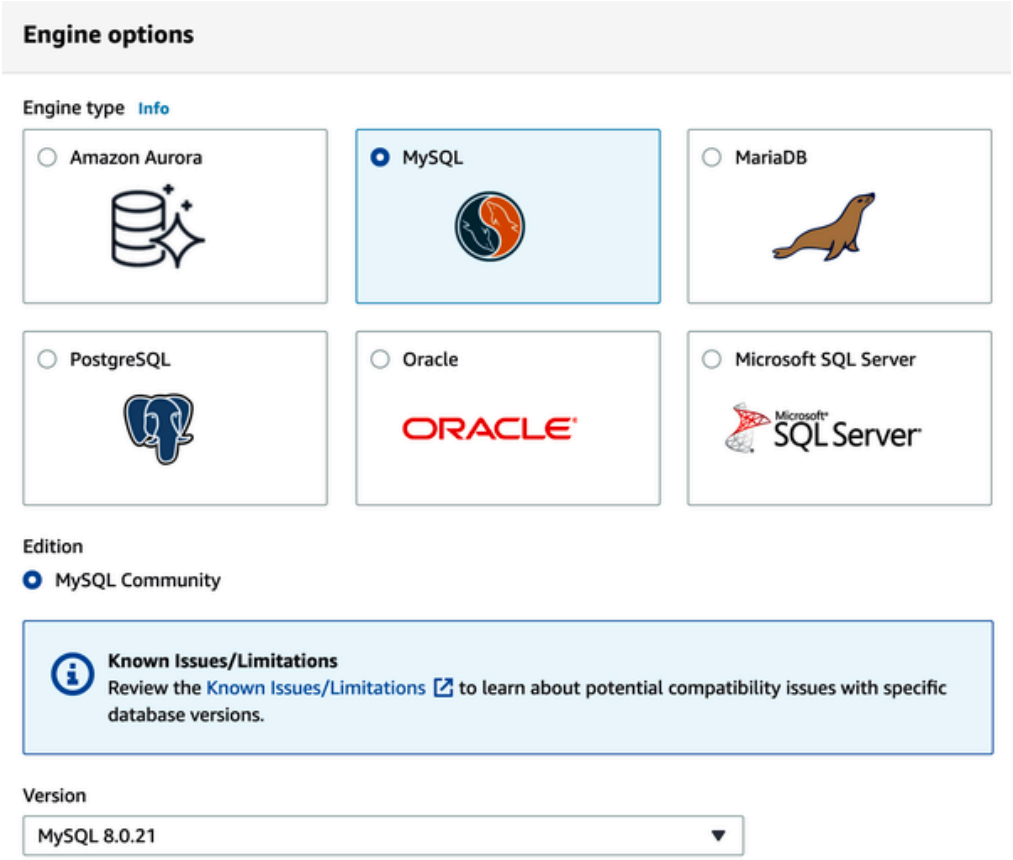

Figure 14-1. Choosing the product

We can also choose the version—here, we've selected 8.0.21. Next, we need to set the master user (similar to `root`) and its password. Make sure to pick a strong password, especially if you will expose your database to the world. Figure 14-2 shows how to define the username and the password for the master user.

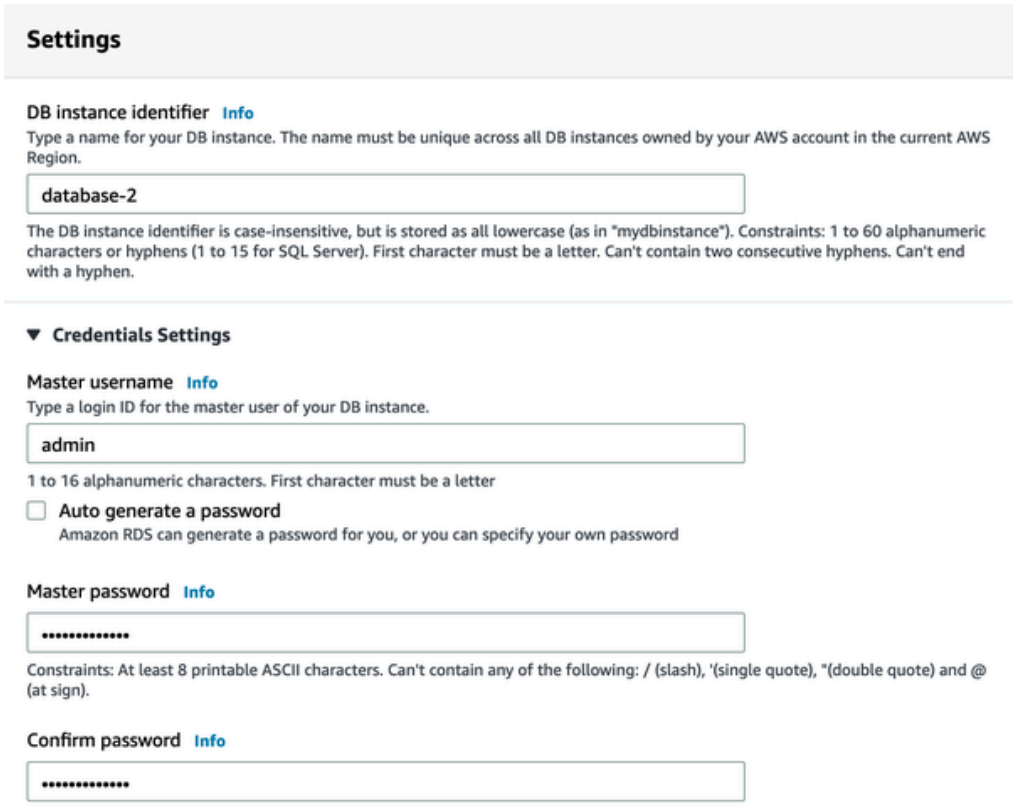

Figure 14-2. Configuring the master user's username and password

Next is the instance size, which will impact directly on the final price. We will pick a top-level configuration to give you an idea of how costly using a DBaaS can be. Figure 14-3 shows the instance classes that are available; there are several options, with varying costs.

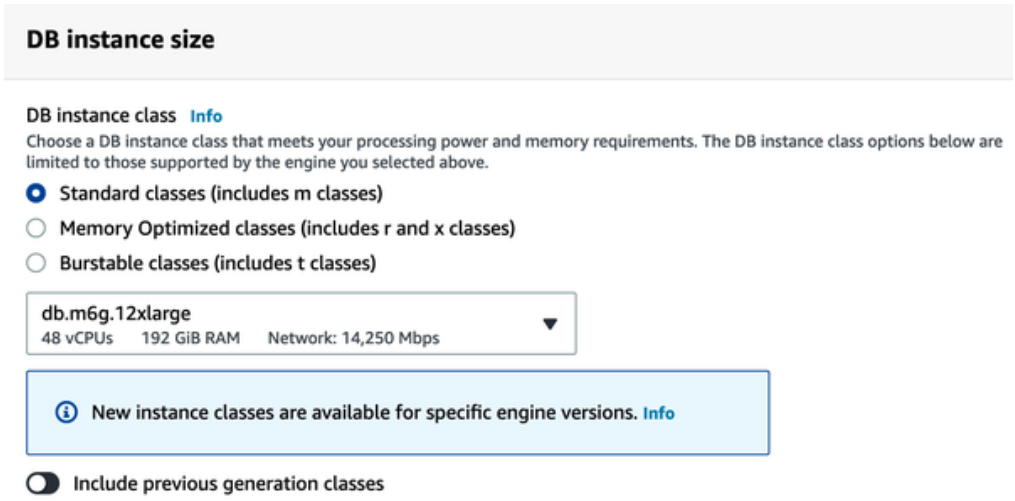

Figure 14-3. Choosing an instance class

Another option that can directly affect the billing is the storage options. Naturally, higher performance (more IOPS) and more storage lead to a higher cost. Figure 14-4 illustrates the choices. You can also select whether to enable autoscaling.

The next option is an important choice: do you want to use multi-AZ deployment or not? The multi-AZ option is all about high availability. When

you provision a multi-AZ DB instance, Amazon RDS automatically creates a primary DB instance and synchronously replicates the data to a standby instance in a different Availability Zone (AZ). The AZs are physically distinct and have independent infrastructure, which increases overall availability.

If you don't want to use multi-AZ deployment, RDS will install a single instance. In the event of failure, it will spin up a new one and remount its data volume. This process takes some time, during which your database will not be available. Even big cloud providers are not bulletproof, and disasters can happen, so having a standby server is always recommended. Figure 14-5 shows how to configure the replica.

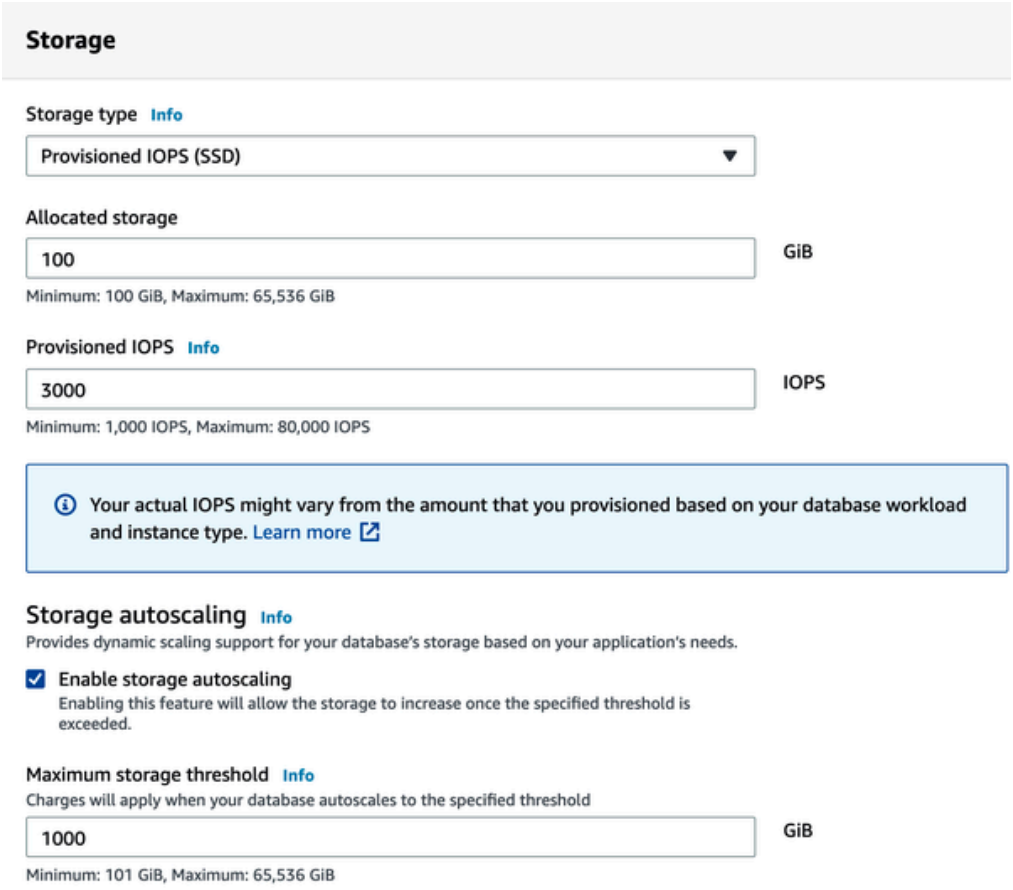

Figure 14-4. Configuring storage size and its IOPS performance

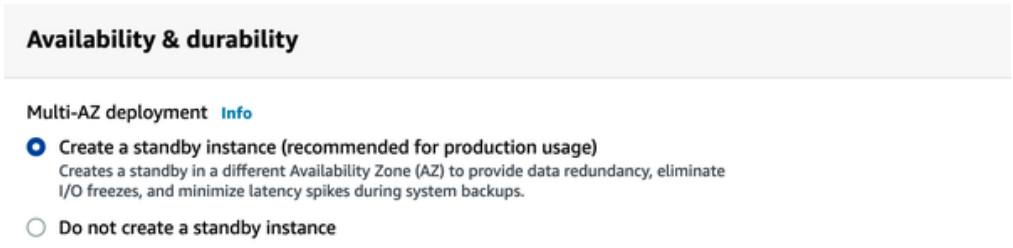

Figure 14-5. Configuring a standby replica

The next part is setting up a general networking configuration. We recommend configuring RDS to use a private network, which only the

application servers and developers' IPs can access. Figure 14-6 shows the network options.

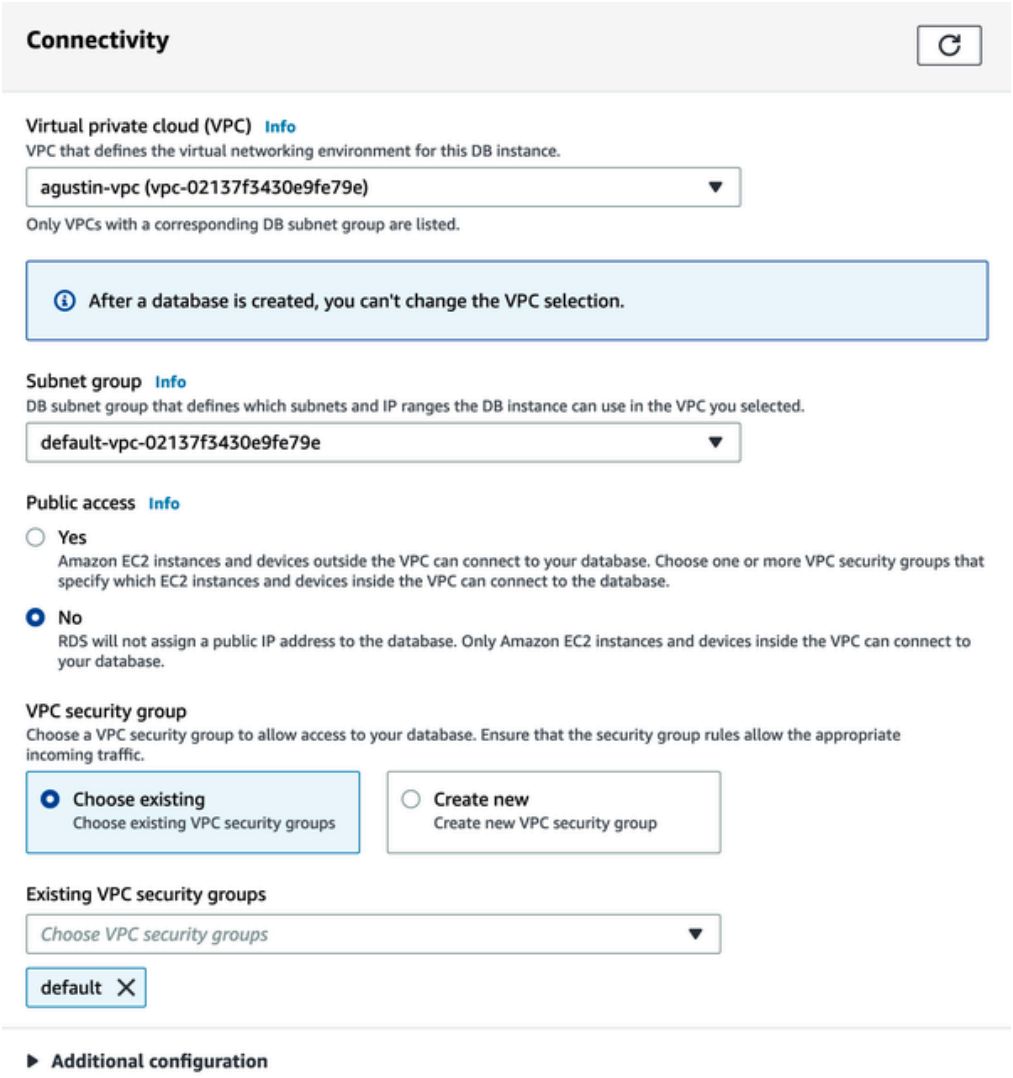

Figure 14-6. Configuring the network settings

Finally, alas, come the estimated costs. Figure 14-7 shows how much you will pay per month for your configured choices.

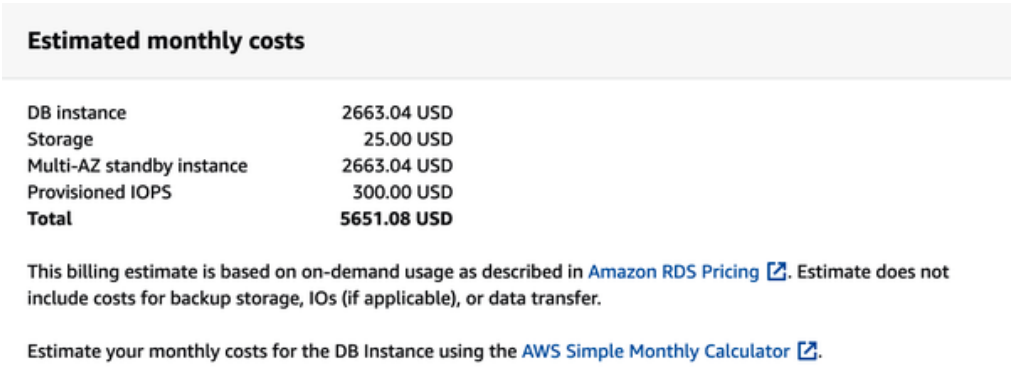

Figure 14-7. The bill can reach the stars under certain configurations!

Google Cloud SQL for MySQL

Google Cloud SQL offers managed database services comparable to Amazon RDS (and Azure), but with slight differences. The Google Cloud options for MySQL are more straightforward because there are fewer options to choose from. For example, you cannot choose between MySQL and MariaDB, or choose the MySQL minor version (only the major version). As shown in Figure 14-8, you can get started by either creating a new instance or migrating an existing database to Google Cloud.

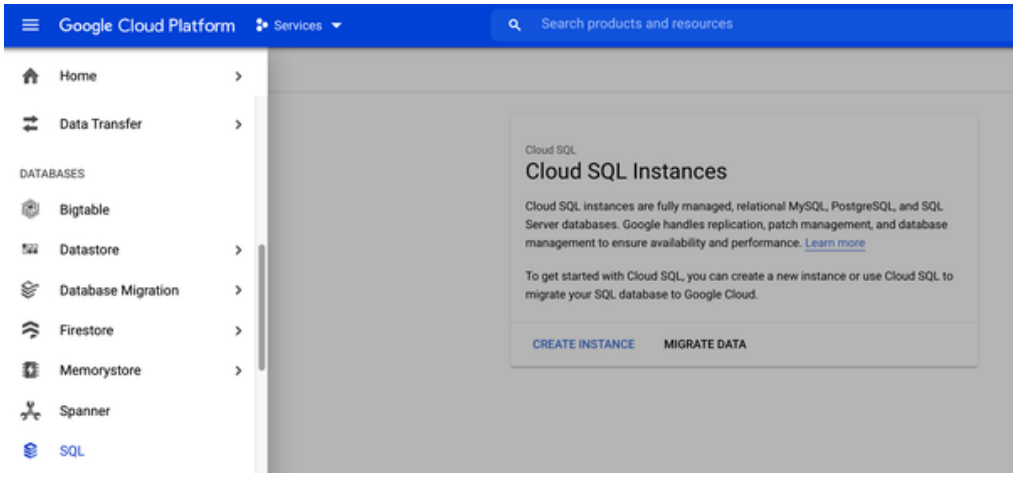

Figure 14-8. Google Cloud SQL

When creating a new instance, you have to fill in a few options. The first step is to choose the product. Figure 14-9 show the options available for MySQL.

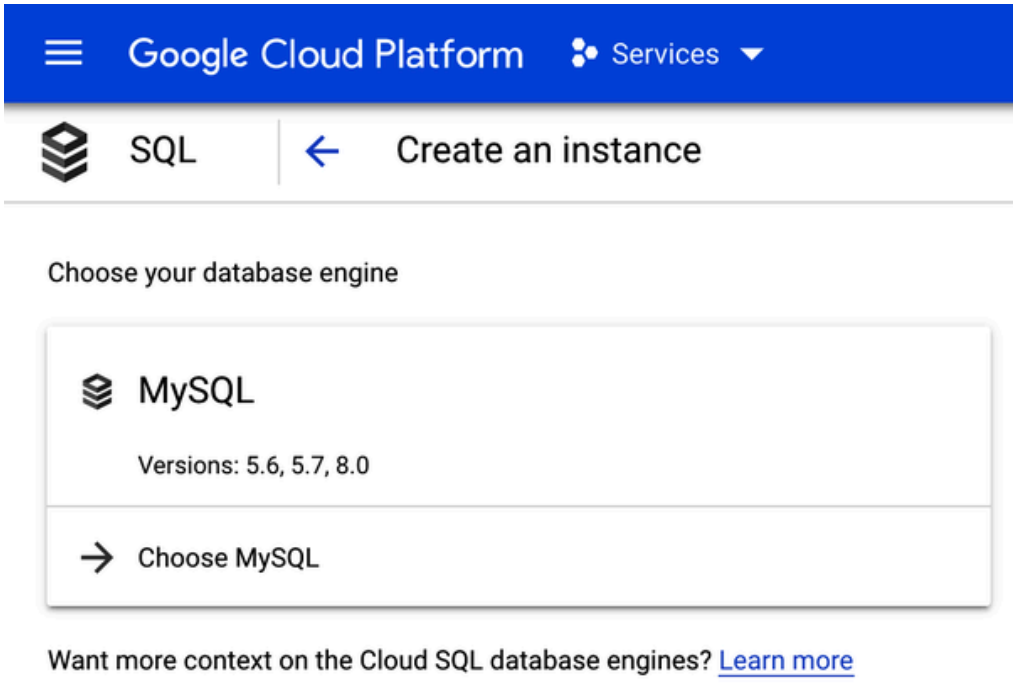

Figure 14-9. Choosing the product

After picking MySQL, you'll need to specify the instance name, `root` password, database version, and location. Figure 14-10 shows how to configure these settings.

Figure 14-10. Setting basic configuration

Next are the settings that can impact the performance, and of course, the cost—it's crucial to find the right balance here. Figure 14-11 shows the storage, memory, and CPU options available.

Figure 14-11. Configuring the machine type and storage

Now the instance is ready to launch in the Google Cloud.

Azure SQL

The last of the top three cloud service providers is Azure SQL. Figure 14-12 shows the database products available in Azure. You'll want to select "Azure Database for MySQL servers."

Figure 14-12. Choosing MySQL in Azure

Azure offers two options, to go with a simple server or a more robust solution with high availability in the setup. Figure 14-13 shows the difference between the two options.

Figure 14-13. Choosing a single server or a flexible server

Those choices are followed by similar configurations regarding the service performance and costs. Figure 14-14 shows the MySQL managed services options.

Figure 14-14. Configuring our MySQL managed service instance

Amazon Aurora

Amazon Aurora is a MySQL- and PostgreSQL-compatible relational database solution provided by Amazon under a commercial license. It offers similar features to MySQL, plus a few extra features developed by Amazon.

Two of these features are worth mentioning. First is Aurora Parallel Query (PQ), a feature that parallelizes some of the I/O and computation involved in processing data-intensive queries.

Aurora PQ works by doing a full table scan (the storage level performs the parallel reads). When we use a parallel query, the query does not use the InnoDB buffer pool. Instead, it pushes query processing down to the storage layer and parallelizes it.

The advantage is that moving the processing closer to the data reduces network traffic and latency. However, the feature is not a silver bullet and does not work well for all cases—it works best for analytical queries that need to run over large portions of data.

The PQ feature is not available for all AWS instances. For instances that support this feature, their instance class determines the number of parallel queries that can be active at a given time. The following instances are the ones that support the PQ feature:

- `db.r*.large` : 1 concurrent parallel query session
- `db.r*.xlarge` : 2 concurrent parallel query sessions
- `db.r*.2xlarge` : 4 concurrent parallel query sessions
- `db.r*.4xlarge` : 8 concurrent parallel query sessions
- `db.r*.8xlarge` : 16 concurrent parallel query sessions
- `db.r4.16xlarge` : 16 concurrent parallel query sessions

The other notable feature is the Amazon Aurora Global Database, designed for applications with a global footprint. It allows a single Aurora database to span multiple AWS regions, with fast replication to enable low-latency global reads and disaster recovery from region-wide outages. The Aurora Global Database uses storage-based replication using the dedicated Amazon infrastructure across its data centers worldwide.

MySQL Cloud Instances

A *cloud instance* is nothing but a virtual server. The different cloud providers have different names for these: Amazon Elastic Compute Cloud (EC2) instances, Google Compute Engine instances, and Azure Virtual Machines.

All of them offer different instance types according to the user's business needs, varying from shallow, basic configurations to astounding limits. For example, the Compute Engine `m2-megamem-416` machine type is a monster that has 416 CPUs and 5,888 GB of RAM.

The MySQL installation process for these instances is the standard one described in Chapter 1. In this case, the most significant advantage of using cloud instances compared to DBaaS solutions is the freedom of customizing

MySQL and the operating system according to your needs without the limitations that managed databases have.

MySQL in Kubernetes

The most recent option available to deploy MySQL instances is Kubernetes. Kubernetes and the OpenShift platform have added a way to manage containerized systems, including database clusters. The management is achieved by controllers declared in configuration files. These controllers provide automation to create objects, such as a container or a group of containers called a *pod*, to listen for a specific event and perform a task.

This automation adds complexity to the container-based architecture and stateful applications, such as databases. A Kubernetes *operator* is a particular type of controller introduced to simplify complex deployments. The operator extends the Kubernetes API with custom resources.

There are many good books written on how Kubernetes works. To keep this section as concise as possible, we will discuss the significant components relevant to the Percona Kubernetes Operator. For a quick introduction to Kubernetes, you can check out the documentation from the Linux Foundation. Figure 14-15 shows the Percona XtraDB Cluster components in Kubernetes.

Figure 14-15. The Percona XtraDB Cluster components in Kubernetes

The following section describes how to deploy the Percona Kubernetes Operator for Percona XtraDB Cluster, which is considered production-ready. Other operators are also available. For example:

- Oracle provides a Kubernetes Operator for MySQL InnoDB Cluster. At the time of this writing, the operator is in a preview state, so it is not recommended for production.
- MariaDB has an operator, but at the time of writing it is in the alpha stage, so please check its maturity before using it in production.
- Presslabs has released an operator that deploys MySQL instances along with orchestrator and backup functionalities. This operator is production-ready.

Deploying Percona XtraDB Cluster in Kubernetes

This section will walk you through the steps of deploying a Kubernetes cluster in the Google Cloud using the Google Cloud SDK and the Percona Kubernetes Operator for PXC:

1. Install the Google Cloud SDK.
   The SDK provides tools and libraries for interacting with Google Cloud products and services. Download the binary appropriate for your platform and install it. Here's an example for macOS:

```
$ wget https://dl.google.com/dl/cloudsdk/channels/ra
     google-cloud-sdk-341.0.0-darwin-x86_64.tar.gz
$ tar -xvf google-cloud-sdk-341.0.0-darwin-x86_64.ta
$ cd google-cloud-sdk/
$ ./install.sh
```

2. Install `kubectl` with `gcloud`.
   With `gcloud` installed, install the `kubectl` component using the following command:

```
$ gcloud components install kubectl
```

3. Create the Kubernetes cluster.
   To create the Kubernetes cluster, first you need to authenticate in the Google Cloud service:

```
$ gcloud auth login
```

Once authenticated, create the cluster. The command accepts a lot of parameters, but in this case, we will go with the basics to create a Kubernetes cluster:

```
$ gcloud container clusters create --machine-type n1
    --num-nodes 3 --zone us-central1-b --project sup
    --cluster-version latest vinnie-k8
```

NOTE

The account needs to have the necessary privileges to create the cluster. Also, you need to replace the project and cluster names used here with your own names. You may also be required to edit the zone location, set to `us-central1-b` in this example.

The parameters used here are just a small subset of everything available—you can see all the options by running `gcloud container clusters --help`. For this case, we've just requested a cluster with three nodes of *`n1-standard-4`* type instances.

This process may take a while, especially if there are a lot of nodes. The output will look like this:

```
Creating cluster vinnie-k8 in us-central1-b... Clust
health-checked (master is healthy)...done.
Created [https://container.googleapis.com/v1/project
zones/us-central1-b/clusters/vinnie-k8].
To inspect the contents of your cluster, go to:
https://console.cloud.google.com/kubernetes/workload
us-central1-b/vinnie-k8?project=support-211414
kubeconfig entry generated for vinnie-k8.
+----------+--------------+------------------+----
| NAME     | LOCATION     | MASTER_VERSION   | MAS
+----------+--------------+------------------+----
| vinnie-k8 | us-central1-b | 1.19.10-gke.1000 | 34.
+----------+--------------+------------------+----
...+---------------------------------+-----------+--
...| MACHINE_TYPE NODE_VERSION       | NUM_NODES | S
...+---------------------------------+-----------+--
```

```
...| n1-standard-4  1.19.10-gke.1000 | 3          | R
...+-----------------------------------+-----------+--
```

And we can check the pods of our Kubernetes cluster in the Google Cloud:

```
$ kubectl get nodes


NAME                                          STATUS
VERSION
gke-vinnie-k8-default-pool-376c2051-5xgz   Ready
v1.19.10-gke.1000
gke-vinnie-k8-default-pool-376c2051-w2tk   Ready
v1.19.10-gke.1000
gke-vinnie-k8-default-pool-376c2051-wxd7   Ready
v1.19.10-gke.1000
```

It is also possible to use the Google Cloud interface to deploy the cluster, as shown in Figure 14-16.

Figure 14-16. From the main menu, select Kubernetes Engine, then Clusters

To create a new cluster, choose the CREATE option shown at the top of Figure 14-17.

Figure 14-17. Create the Kubernetes cluster by clicking CREATE

The final step is to install the PXC operator. The documentation for deploying the operator has very detailed instructions. We'll follow the recommended steps here.

First, configure Cloud Identity and Access Management (Cloud IAM) to control access to the cluster. The following command will give you the ability to create Roles and RoleBindings:

```
$ kubectl create clusterrolebinding cluster-admin-bindi
    cluster-admin --user $(gcloud config get-value core
```

The return statement confirms the creation:

```
clusterrolebinding.rbac.authorization.k8s.io/cluster-ac
```

Next, create a namespace and set the context for the namespace:

```
$ kubectl create namespace learning-mysql
$ kubectl config set-context $(kubectl config current-c
   --namespace=learning-mysql
```

Now, clone the repository and change to the directory:

```
$ git clone -b v1.8.0 \
    https://github.com/percona/percona-xtradb-cluster-c
$ cd percona-xtradb-cluster-operator
```

Deploy the operator with the following command:

```
$ kubectl apply -f deploy/bundle.yaml
```

This should return the following confirmation:

```
customresourcedefinition.apiextensions.k8s.io/perconaxt
percona.com created
customresourcedefinition.apiextensions.k8s.io/perconaxt
pxc.percona.com created
customresourcedefinition.apiextensions.k8s.io/perconaxt
pxc.percona.com created
customresourcedefinition.apiextensions.k8s.io/perconaxt
percona.com created
role.rbac.authorization.k8s.io/percona-xtradb-cluster-c
serviceaccount/percona-xtradb-cluster-operator created
rolebinding.rbac.authorization.k8s.io/service-account-p
cluster-operator created
deployment.apps/percona-xtradb-cluster-operator createc
```

The operator has been started, and you can confirm this by running:

```
$ kubectl get pods
```

Now, create the Percona XtraDB Cluster:

```
$ kubectl apply -f deploy/cr.yaml
```

This step can take some time. After that, you will see all the pods running:

```
$ kubectl get pods
NAME                                  READY   STATUS    RE
cluster1-haproxy-0                    2/2     Running   0
cluster1-haproxy-1                    2/2     Running   0
cluster1-haproxy-2                    2/2     Running   0
cluster1-pxc-0                        3/3     Running   0
cluster1-pxc-1                        3/3     Running   0
cluster1-pxc-2                        3/3     Running   0
```

```
percona-xtradb-cluster-operator-   1/1     Running   0
77bfd8cdc5-d7zll
```

During the previous steps, the operator has generated several secrets, including the password for the `root` user, which you will need to access the cluster. To get the generated secrets, run the following command:

```
$ kubectl get secret my-cluster-secrets -o yaml
```

You will see output like this:

```
apiVersion: v1
data:
  clustercheck: UFZjdjk0SU4xWGtBSTR2VlVJ
  monitor: ZWZja01mOWhBTXZ4bTB0bUZ4eQ==
  operator: Vm10R0IxbHA4cVVZTkxqVVI4Mg==
  proxyadmin: VXVFbkx1S3RmUTEzVlNOd1c=
  root: eU53aWlKT3ZXaXJaeG16OXJK
  xtrabackup: V3VNNWRnWUdIblVWaU1OWGY=
...
secrets/my-cluster-secrets
  uid: 9d78c4a8-1926-4b7a-84a0-43087a601066
type: Opaque
```

The actual password is base64-encoded, so you'll need to run the following command to get the `root` password:

```
$ echo 'eU53aWlKT3ZXaXJaeG16OXJK' | base64 --decode
yNwiiJOvWirZxmz9rJ
```

Now that you have the password, to check connectivity with the cluster you can create a client pod:

```
$ kubectl run -i --rm --tty percona-client --image=perc
    --restart=Never -- bash -il
```

Then connect to MySQL:

```
$ mysql -h cluster1-haproxy -uroot -pyNwiiJOvWirZxmz9rJ
```

Note that the operator comes with HAProxy, which is a load balancer (we will discuss load balancing in the next chapter).