11 Caching data in memory: Amazon ElastiCache and MemoryDB

This chapter covers

- The benefits of a having a caching layer between your application and data store
- Defining key terminology, such as cache cluster, node, shard, replication group, and node group
- Using /operating an in-memory key-value store
- Performance tweaking and monitoring ElastiCache clusters

Imagine a relational database being used for a popular mobile game where players' scores and ranks are updated and read frequently. The read and write pressure to the database will be extremely high, especially when ranking scores across millions of players. Mitigating that pressure by scaling the database may help with load but not necessarily the latency or cost. Also, caching relational databases tends to be more expensive than caching data stores.

A proven solution used by many gaming companies is leveraging an in-memory data store such as Redis for both caching and ranking player and game metadata. Instead of reading and sorting the leaderboard directly from the relational database, they store an in-memory game leaderboard in Redis, commonly using a sorted set, which will sort the data automatically when it's inserted, based on the score parameter. The score value may consist of the actual player ranking or player score in the game.

Because the data resides in memory and does not require heavy computation to sort, retrieving the information is incredibly fast, leaving little reason to query directly from the relational database. In addition, any other game and player metadata, such as player profile, game-level information, and so on, that requires heavy reads can also be cached using this in-memory layer, thus alleviating heavy read traffic to and from the database.

In this solution, both the relational database and in-memory layer will store updates to the leaderboard: one will serve as the primary database and the other as the working and fast processing layer. When implementing a caching layer, you can employ a variety of caching techniques to

keep the data that's cached fresh, which we'll review later. Figure 11.1 shows that the cache sits between your application and the database.

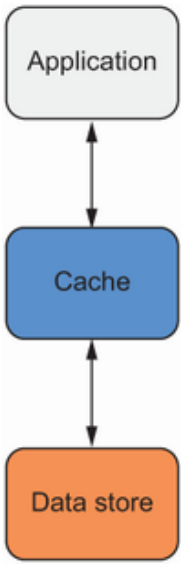

Figure 11.1 The cache sits between the application and the database.

A cache comes with the following benefits:

- Serving read traffic from the caching layer frees resources on your primary data store, for example, for write requests.
- It speeds up your application because the caching layer responds more quickly than your data store.
- It allows you to downsize your data store, which is typically more expensive than the caching layer.

Most caching layers reside in memory, which is why they are so fast. The downside is that you can lose the cached data at any time because of a hardware defect or a restart. Always keep a copy of your data in a primary data store with disk durability, like the relational database in the mobile game example. Alternatively, Redis has optional failover support. In the event of a node failure, a replica node will be elected to be the new primary and will already have a copy of the data. On top of that, some in-memory databases also support writing data to persistent storage to be able to restore the data after a reboot or outage.

In this chapter, you will learn how to implement an in-memory caching layer to improve the performance of a web application. You will deploy a complex web application called Discourse, a modern forum software application that uses Redis for caching. You will also learn how to scale a cache cluster and how to monitor and tweak performance.

Depending on your caching strategy, you can either populate the cache in real time or on demand. In the mobile game example, on demand means

that if the leaderboard is not in the cache, the application asks the relational database and puts the result into the cache. Any subsequent request to the cache will result in a cache hit, meaning the data is found in the cache. This will be true until the duration of the TTL (time-to-live) value on the cached value expires. This strategy is called *lazy-loading* data from the primary data store. Additionally, we could have a job running in the background that queries the leaderboard from the relational database every minute and puts the result in the cache to populate the cache in advance.

The lazy-loading strategy (getting data on demand) is implemented like this:

1. The application writes data to the data store.
2. If the application wants to read the data, at a later time it makes a request to the caching layer.
3. If the caching layer does not contain the data, the application reads from the data store directly and puts the value into the cache, and also returns the value to the client.
4. Later, if the application wants to read the data again, it makes a request to the caching layer and finds the value.

This strategy comes with a problem. What if the data is changed while it is in the cache? The cache will still contain the old value. That's why setting an appropriate TTL value is critical to ensure cache validity. Let's say you apply a TTL of five minutes to your cached data: this means you accept that the data could be out of sync by up to five minutes with your primary database. Understanding the frequency of change for the underlying data and the effects out-of-sync data will have on the user experience is the first step of identifying the appropriate TTL value to apply. A common mistake some developers make is assuming that a few seconds of a cache TTL means that having a cache is not worthwhile. Remember that within those few seconds, millions of requests can be offloaded from your data store, speeding up your application and reducing database pressure. Performance testing your application with and without your cache, along with various caching approaches, will help fine-tune your implementation. In summary, the shorter the TTL, the higher the load on the underlying data store. The higher the TTL, the more out of sync the data gets.

The write-through strategy (caching data up front) is implemented differently to tackle the synchronization problem, as shown here:

1. The application writes data to the data store and the cache (or the cache is filled asynchronously, for example, by a background job, an AWS Lambda function, or the application).
2. If the application wants to read the data at a later time, it makes a request to the caching layer, which always contains the latest data.
3. The value is returned to the client.

This strategy also comes with a problem. What if the cache is not big enough to contain all your data? Caches are in memory, and your data store's disk capacity is usually larger than your cache's memory capacity. When your cache exceeds the available memory, it will evict data or stop accepting new data. In both situations, the application stops working. A global leaderboard will most likely fit into the cache. Imagine that a leaderboard is 4 KB in size and the cache has a capacity of 1 GB (1,048,576 KB). But what about introducing leaderboards per team? You can only store 262,144 (1,048,576/4) leaderboards, so if you have more teams than that, you will run into a capacity issue. Figure 11.2 compares the two caching strategies.

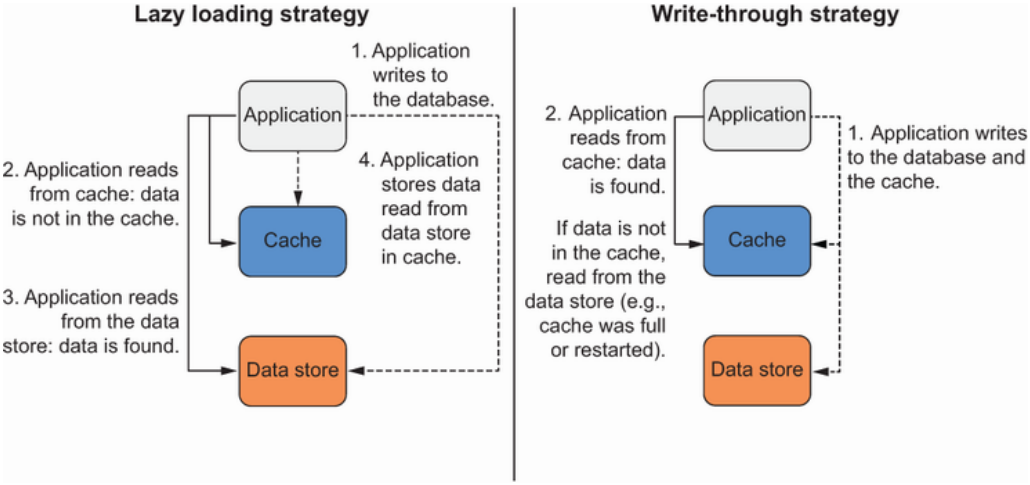

Figure 11.2 Comparing the lazy-loading and write-through caching strategies

When evicting data, the cache needs to decide which data it should delete. One popular strategy is to evict the least recently used (LRU) data. This means that cached data must contain meta information about the time when it was last accessed. In case of an LRU eviction, the data with the oldest timestamp is chosen for eviction.

Caches are usually implemented using key-value stores. Key-value stores don't support sophisticated query languages such as SQL. They support retrieving data based on a key, usually a string, or specialized commands, for example, to extract sorted data efficiently.

Imagine that in your relational database is a `player` table for your mobile game. One of the most common queries is `SELECT id, nickname FROM player ORDER BY score DESC LIMIT 10` to retrieve the top 10 players. Luckily, the game is very popular, but this comes with a technical challenge. If many players look at the leaderboard, the database becomes very busy, which causes high latency or even timeouts. You have to come up with a plan to reduce the load on the database. As you have already learned, caching can help. What technique should you employ for caching? You have a few options.

One approach you can take with Redis is to store the result of your SQL query as a string value and the SQL statement as your key name. Instead of using the whole SQL query as the key, you can hash the string with a hash function like `md5` or `sha256` to optimize storage and bandwidth, shown in figure 11.3. Before the application sends the query to the database, it takes the SQL query as the key to ask the caching layer for data—step 2. If the cache does not contain data for the key (step 3), the SQL query is sent to the relational database (step 4). The result (step 5) is then stored in the cache using the SQL query as the key (step 6). The next time the application wants to perform the query, it asks the caching layer (step 7), which now contains the cached table (step 8).

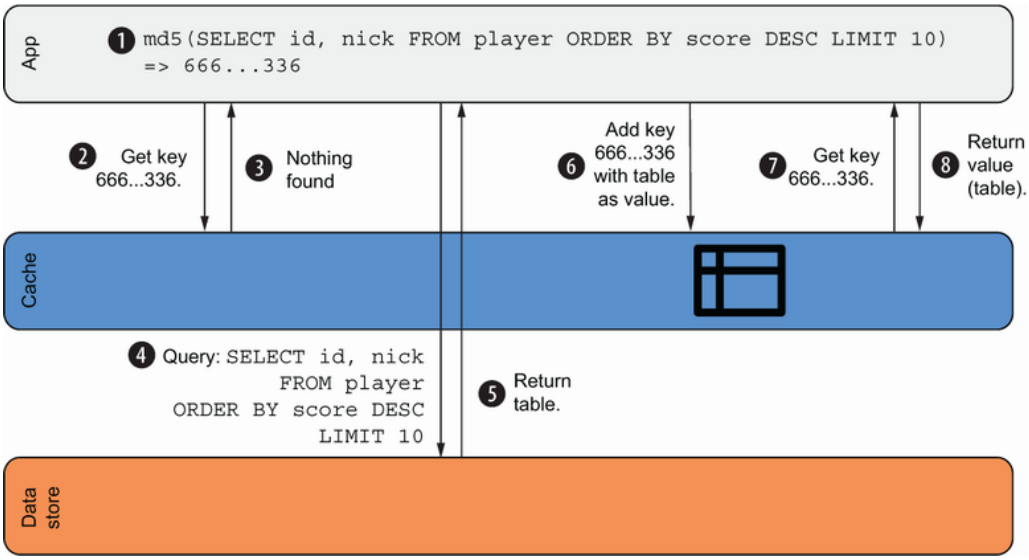

Figure 11.3 SQL caching layer implementation

To implement caching, you only need to know the key of the cached item. This can be an SQL query, a filename, a URL, or a user ID. You take the key and ask the cache for a result. If no result is found, you make a second call to the underlying data store, which knows the truth.

With Redis, you also have the option of storing the data in other data structures such as a sorted set. If the data is stored in a sorted set, retriev-

ing the ranked data will be very efficient. You could simply store players and their scores and sort by the score. An equivalent SQL command would be as follows:

```
ZREVRANGE "player-scores" 0 9
```

This would return the 10 players in a sorted set named “player-scores,” ordered from highest to lowest.

The two most popular implementations of in-memory key-value stores are Memcached and Redis. Table 11.1 compares their features.

Table 11.1 Comparing Memcached and Redis features

	Memcached	Redis
Data types	Simple	Complex
Data manipulation commands	12	125
Server-side scripting	No	Yes (Lua)
Transactions	No	Yes

Examples are 100% covered by the Free Tier

The examples in this chapter are completely covered by the Free Tier. As long as you don’t run the examples longer than a few days, you won’t pay anything. Keep in mind that this applies only if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you’ll clean up your account at the end of the sections.

Amazon ElastiCache offers Memcached and Redis clusters as a service. Therefore, AWS covers the following aspects for you:

- *Installation*—AWS installs the software for you and has enhanced the underlying engines.
- *Administration*—AWS administers Memcached/Redis for you and provides ways to configure your cluster through parameter groups. AWS also detects and automates failovers (Redis only).
- *Monitoring*—AWS publishes metrics to CloudWatch for you.

- *Patching*—AWS performs security upgrades in a customizable time window.
- *Backups*—AWS optionally backs up your data in a customizable time window (Redis only).
- *Replication*—AWS optionally sets up replication (Redis only).

Next, you will learn how to create an in-memory cluster with ElastiCache that you will later use as an in-memory cache for a gaming backend.

11.1 Creating a cache cluster

In this chapter, we focus on the Redis engine because it's more flexible. You can choose which engine to use based on the features that we compared in the previous section. If there are significant differences to Memcached, we will highlight them.

11.1.1 Minimal CloudFormation template

Imagine you are developing an online game. To do so, you need to store sessions that keep the current game state of each user, as well as a high-score list. Latency is important to ensure the gamers enjoy a great gaming experience. To be able to read and write data with very little latency, you decide to use the in-memory database Redis.

First, you need to create an ElastiCache cluster using the Management Console, the CLI, or CloudFormation. You will use CloudFormation in this chapter to manage your cluster. The resource type of an ElastiCache cluster is `AWS::ElastiCache::CacheCluster`. The required properties follow:

- `Engine`—Either `redis` or `memcached`
- `CacheNodeType`—Similar to the EC2 instance type, for example, `cache.t2.micro`
- `NumCacheNodes`—1 for a single-node cluster
- `CacheSubnetGroupName`—Reference subnets of a VPC using a dedicated resource called a subnet group
- `VpcSecurityGroupIds`—The security groups you want to attach to the cluster

A minimal CloudFormation template is shown in the next listing. The first part of the template contains the ElastiCache cluster.

Listing 11.1 Minimal CloudFormation template of an ElastiCache Redis single-node cluster

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'AWS in Action: chapter 11 (minimal)'
Parameters:                                                    ①
  VPC:
    Type: 'AWS::EC2::VPC::Id'
  SubnetA:
    Type: 'AWS::EC2::Subnet::Id'
  SubnetB:
    Type: 'AWS::EC2::Subnet::Id'
Resources:
  CacheSecurityGroup:                                          ②
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: cache
      VpcId: !Ref VPC
      SecurityGroupIngress:
      - IpProtocol: tcp                                        ③
        FromPort: 6379
        ToPort: 6379
        CidrIp: '0.0.0.0/0'
  CacheSubnetGroup:                                            ④
    Type: 'AWS::ElastiCache::SubnetGroup'
    Properties:
      Description: cache
      SubnetIds:
      - Ref: SubnetA                                           ⑤
      - Ref: SubnetB
  Cache:
    Type: 'AWS::ElastiCache::CacheCluster'                     ⑥
    Properties:
      CacheNodeType: 'cache.t2.micro'                          ⑦
      CacheSubnetGroupName: !Ref CacheSubnetGroup
      Engine: redis                                            ⑧
      NumCacheNodes: 1                                         ⑨
      VpcSecurityGroupIds:
      - !Ref CacheSecurityGroup
```

① Defines VPC and subnets as parameters

② The security group to manage which traffic is allowed to enter/leave the cluster

③ Redis listens on port 6379. This allows access from all IP addresses, but because the cluster has only private IP addresses, access is possible only from inside the VPC. You will improve this in section 11.3.

④ Subnets are defined within a subnet group (same approach is used in RDS).

⑤ List of subnets that can be used by the cluster

⑥ The resource to define the Redis cluster

⑦ The node type cache.t2.micro comes with 0.555 GiB memory and is part of the Free Tier.

⑧ ElastiCache supports redis and memcached. We are using redis because we want to make use of advanced data structures supported only by Redis.

⑨ Creates a single-node cluster for testing, which is not recommended for production workloads because it introduces a single point of failure

As mentioned already, ElastiCache nodes in a cluster use only private IP addresses. Therefore, you can't connect to a node directly over the internet. The same is true for other resources as RDS database instances. Therefore, create an EC2 instance in the same VPC as the cluster for testing. From the EC2 instance, you can then connect to the private IP address of the cluster.

11.1.2 Test the Redis cluster

To be able to access the ElastiCache cluster, you'll need a virtual machine. The following snippet shows the required resources:

```
Resources:
  # [...]
  InstanceSecurityGroup:                                          ①
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'vm'
      VpcId: !Ref VPC
  Instance:                                                       ②
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: !FindInMap [RegionMap, !Ref 'AWS::Region', AMI]
      InstanceType: 't2.micro'
      IamInstanceProfile: !Ref InstanceProfile
      NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeleteOnTermination: true
        DeviceIndex: 0
        GroupSet:
```

```
          - !Ref InstanceSecurityGroup
          SubnetId: !Ref SubnetA
  Outputs:
    InstanceId:                                                    ③
      Value: !Ref Instance
      Description: 'EC2 Instance ID (connect via Session Manager)'
    CacheAddress:                                                  ④
      Value: !GetAtt 'Cache.RedisEndpoint.Address'
      Description: 'Redis DNS name (resolves to a private IP address)'
```

① The security group does allow outbound traffic only.

② The virtual machine used to connect to your Redis cluster

③ The ID of the instance used to connect via the Session Manager

④ The DNS name of Redis cluster node (resolves to a private IP address)

Next, create a stack based on the template to create all the resources in your AWS account using the Management Console: **http://mng.bz/09jm**. You have to fill in the next three parameters when creating the stack:

- `SubnetA`—You should have at least two options here; select the first one.
- `SubnetB`—You should have at least two options here; select the second one.
- `VPC`—You should have only one possible VPC here—your default VPC. Select it.

Want to have a deeper look into the CloudFormation template?

Where is the template located?

You can find the template on GitHub. Download a snapshot of the repository at **https://github.com/AWSinAction/code3/archive/main.zip**. The file we're talking about is located at chapter11/redis-minimal.yaml. On S3, the same file is located at **http://mng.bz/9Vra**.

The following example guides you through storing a gamer's session, as well as populating and reading a highscore list:

1. Wait until the CloudFormation stack reaches status `CREATE_COMPLETE`.
2. Select the stack and open the Outputs tab. Copy the EC2 instance ID and cache address.

3. Use the Session Manager to connect to the EC2 instance, and use the Redis CLI to interact with the Redis cluster node as described in the following listing.
4. Execute the following commands. Don't forget to replace `$CacheAddress` with the value from the CloudFormation outputs:

```
$ sudo amazon-linux-extras install -y redis6     ①
$ redis-cli -h $CacheAddress                     ②
> SET session:gamer1 online EX 15                ③
OK
> GET session:gamer1                             ④
"online"
> GET session:gamer1                             ⑤
(nil)                                            ⑥
> ZADD highscore 100 "gamer1"                    ⑦
(integer) 1
> ZADD highscore 50 "gamer2"                     ⑧
(integer) 1
> ZADD highscore 150 "gamer3"                    ⑨
(integer) 1
> ZADD highscore 5 "gamer4"                      ⑩
(integer) 1
> ZRANGE highscore -3 -1 WITHSCORES              ⑪
1) "gamer2"                                      ⑫
2) "50"
3) "gamer1"
4) "100"
5) "gamer3"
6) "150"
> quit                                           ⑬
```

① Installs the Redis CLI

② Connects to the Redis cluster node, and replaces $CacheAddress with the output from the CloudFormation stack

③ Stores a session for user gamer1 with a time-to-live of 15 seconds

④ Gets the session for user gamer1

⑤ Wait 15 seconds and fetch the session for gamer1 again.

⑥ The response is empty, because the key was automatically deleted after 15 seconds.

⑦ Adds gamer1 with a score of 100 to the sorted set named highscore

⑧ Adds gamer2 with a score of 50 to the sorted set named highscore

⑨ Adds gamer3 with a score of 150 to the sorted set named highscore

⑩ Adds gamer4 with a score of 5 to the sorted set named highscore

⑪ Gets the top three gamers from the highscore list

⑫ The list is sorted ascending and includes the user followed by its score.

⑬ Quits the Redis CLI

You've successfully connected to the Redis cluster and simulated a session store and highscore list used by a typical gaming backend. With this knowledge, you could start to implement a caching layer in your own application. But as always, there are more options to discover.

Cleaning up
It's time to delete the CloudFormation stack named `redis-minimal`. Use the AWS Management Console or the following command to do so:

```
$ aws cloudformation delete-stack --stack-name redis-minimal
```

Continue with the next section to learn more about advanced deployment options with more than one node to achieve high availability or increase the available memory by using sharding.

11.2 Cache deployment options

Which deployment option you should choose is influenced by the following four factors:

- *Engine*—Which in-memory database do you want to use: Memcached or Redis?
- *Backup/Restore*—Does your workload require data persistence, which means being able to backup and restore data?
- *Replication*—Is high availability important to your workload? If so, you need to replicate to at least one other node.
- *Sharding*—Does your data exceed the maximum memory available for a single node, or do you need to increase the throughput of your system?

Table 11.2 compares the deployment options for Memcached and Redis.

Next, we'll look at deployment options in more detail.

Table 11.2 Comparing ElastiCache and MemoryDB engines and deployment options

Service	Engine	Deployment Option	Backup/Restore	Replication	Sharding
ElastiCache	Memcached	Default	No	No	Yes
	Redis	Single Node	Yes	No	No
		Cluster Mode **disabled**	Yes	Yes	No
		Cluster Mode **enabled**	Yes	Yes	Yes
MemoryDB	Redis	Default	Yes	Yes	Yes

11.2.1 Memcached: Cluster

An Amazon ElastiCache for Memcached cluster consists of 1–40 nodes. Sharding is implemented by the Memcached client, typically using a consistent hashing algorithm, which arranges keys into partitions in a ring distributed across the nodes. The client decides which keys belong to which nodes and directs the requests to those partitions. Each node stores a unique portion of the key-space in memory. If a node fails, the node is replaced, but the data is lost. You cannot back up the data stored in Memcached. Figure 11.4 shows a Memcached cluster deployment. Remember that a VPC is a way to define a private network on AWS. A subnet is a way to separate concerns inside the VPC. The cluster nodes are distributed among multiple subnets to increase availability. The client communicates with the cluster node to get data and write data to the cache.

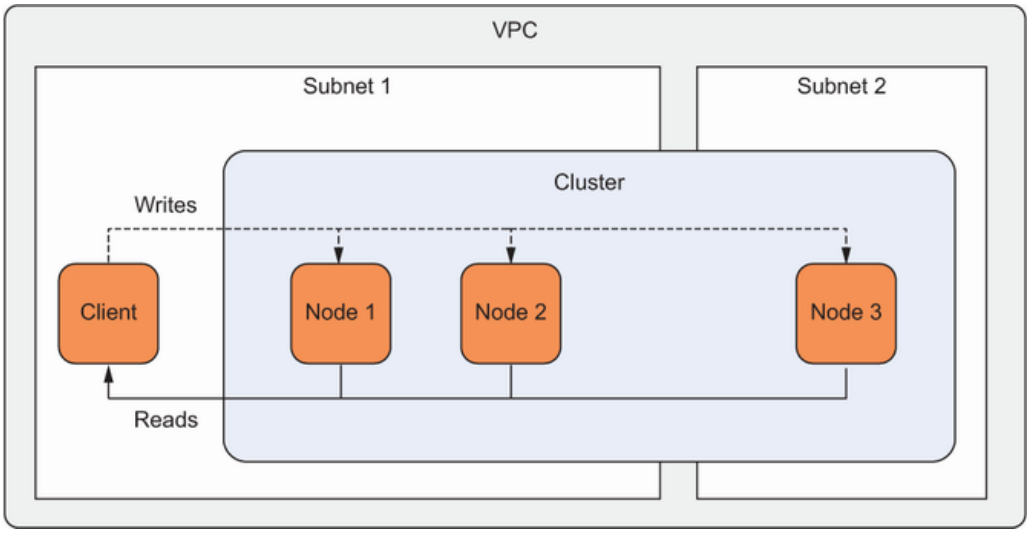

Figure 11.4 Memcached deployment option: Cluster

Use a Memcached cluster if your application requires a simple in-memory store and can tolerate the loss of a node and its data. For instance, the SQL cache example in the beginning of this chapter could be implemented using Memcached. Because the data is always available in the relational database, you can tolerate a node loss, and you need only simple commands (`GET`, `SET`) to implement the query cache.

11.2.2 Redis: Single-node cluster

An ElastiCache for Redis single-node cluster always consists of one node. Sharding and high availability are not possible with a single node. But Redis supports the creation of backups and also allows you to restore those backups. Figure 11.5 shows a Redis single-node cluster.

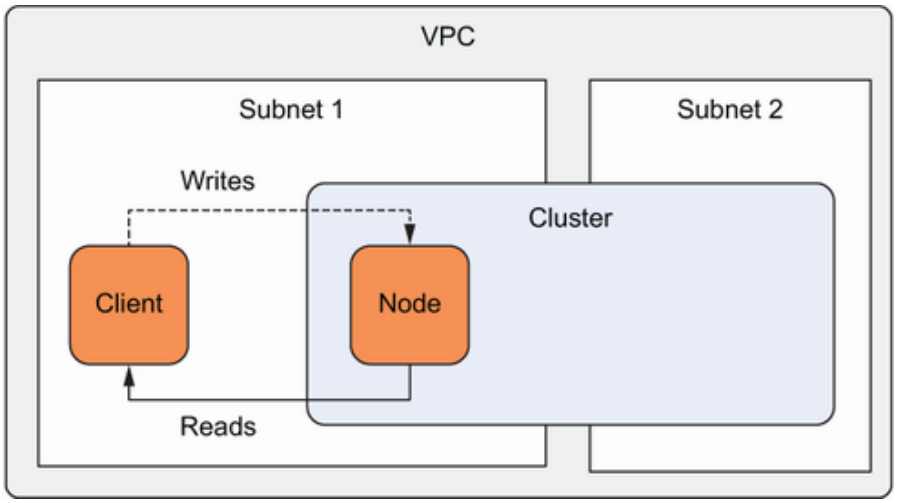

Figure 11.5 Redis deployment option: Single-node cluster

A single node adds a single point of failure (SPOF) to your system. This is probably something you want to avoid for business-critical production systems.

11.2.3 Redis: Cluster with cluster mode disabled

Things become more complicated now, because ElastiCache uses two terminologies. We've been using the terms *cluster*, *node*, and *shard* so far, and the graphical Management Console also uses these terms. But the API, the CLI, and CloudFormation use a different terminology: *replication group*, *node*, and *node group*. We prefer the *cluster*, *node*, and *shard* terminology, but in figures 11.6 and 11.7, we've added the *replication group*, *node*, and *node group* terminology in parentheses.

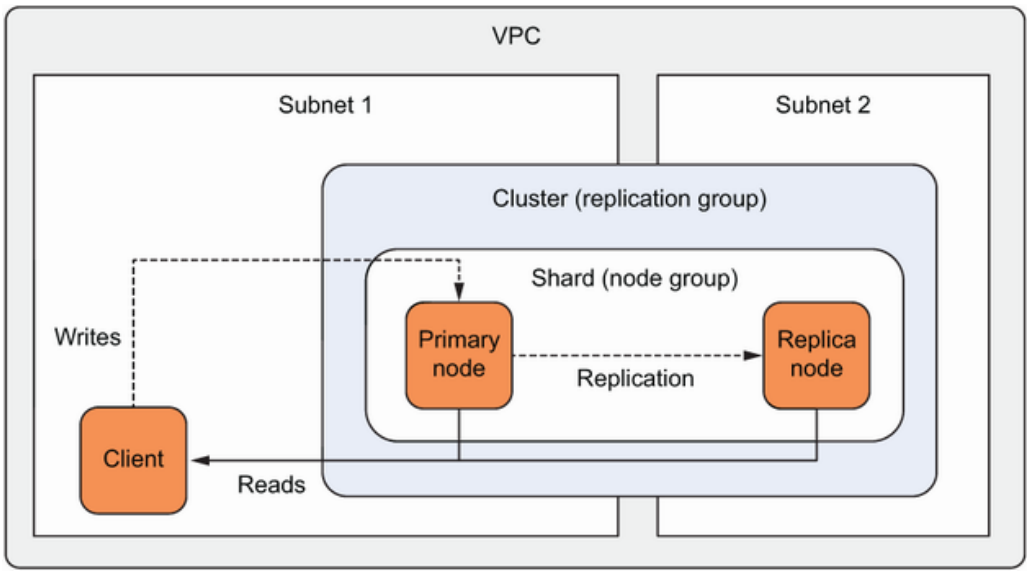

Figure 11.6 Redis deployment option: Cluster with cluster mode disabled

A Redis cluster with cluster mode disabled supports backups and data replication but not sharding. This means there is only one shard containing all the data. The primary node is synchronized to one to five replica nodes.

Use a Redis cluster with cluster mode disabled when you need data replication and all your cached data fits into the memory of a single node. Imagine that your cached data set is 4 GB in size. In that case, the data fits into the memory of nodes of type `cache.m6g.large`, which comes with 6.38 GiB, for example. There is no need to split the data into multiple shards.

11.2.4 Redis: Cluster with cluster mode enabled

A Redis cluster with cluster mode enabled, shown in figure 11.7, supports backups, data replication, and sharding. You can have up to 500 shards per cluster. Each shard consists of at least a primary node and optionally replica nodes. In total, a cluster cannot exceed 500 nodes.

Figure 11.7 Redis deployment option: Cluster with cluster mode enabled

Use a Redis cluster with cluster mode enabled when you need data replication and your data is too large to fit into the memory of a single node. Imagine that your cached data is 22 GB in size. Each cache node has a capacity of 4 GB of memory. Therefore, you will need six shards to get a total capacity of 24 GB of memory. ElastiCache provides up to 437 GB of memory per node, which totals to a maximum cluster capacity of 6.5 TB (15 * 437 GB).

Additional benefits of enabling cluster mode

With cluster mode enabled, failover speed is much faster, because no DNS is involved. Clients are provided a single configuration endpoint to discover changes to the cluster topology, including newly elected primaries. With cluster mode disabled, AWS provides a single primary endpoint, and in the event of a failover, AWS does a DNS swap on that endpoint to one of the available replicas. It may take ~1–1.5 minutes before the application is able to reach the cluster after a failure, whereas with cluster mode enabled, the election takes less than 30 seconds.

On top of that, increasing the number of shards also increases the maximum read/write throughput. If you start with one shard and add a second shard, each shard now only has to deal with 50% of the requests (assuming an even distribution).

Last but not least, as you add nodes, your blast radius decreases. For example, if you have five shards and experience a failover, only 20% of your data is affected. This means you can't write to this portion of the key space until the failover process completes (~15–30 seconds), but you can still read from the cluster, given you have a replica available. With cluster mode disabled, 100% of your data is affected, because a single node consists of your entire key space. You can read from the cluster but can't write until the DNS swap has completed.

You learned about the different deployment options for ElastiCache. There is one more thing we want to bring to your attention: AWS offers another in-memory database called MemoryDB, which you will learn about next.

11.2.5 MemoryDB: Redis with persistence

Even though ElastiCache for Redis supports snapshots and transaction logs for persistence, AWS recommends using ElastiCache as a secondary data store. AWS also provides an alternative called MemoryDB, a proprietary in-memory database with Redis compatibility and distributed transaction log. MemoryDB writes data to disk and reads from memory. By default, MemoryDB stores data among multiple data centers using a distributed transaction log. Therefore, AWS recommends using MemoryDB as a primary database. Keep in mind that data persistence comes with higher write latency—think milliseconds instead of microseconds.

So, MemoryDB is a good fit when a key-value store is all you need, but data persistency is essential, such as the following situations:

- *Shopping cart*—Stores the items a user intends to check out
- *Content management system (CMS)*—Stores blog posts and comments
- *Device management service*—Stores status information about IoT devices

MemoryDB has been generally available since August 2021. We haven't used MemoryDB for production workloads yet, but the approach sounds very promising to us.

Want to give MemoryDB a try? We have prepared a simple example for you. Create a CloudFormation stack by using the Management Console: **http://mng.bz/jAJy**.

While you wait for the stack to reach state `CREATE_COMPLETE`, let's have a look into the CloudFormation template necessary to spin up a MemoryDB cluster, shown here:

```
Resources:
  # [...]
  CacheSecurityGroup:                                    ①
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: cache
      VpcId: !Ref VPC
  CacheParameterGroup:                                   ②
    Type: 'AWS::MemoryDB::ParameterGroup'
    Properties:
      Description: String
      Family: 'memorydb_redis6'                          ③
      ParameterGroupName: !Ref 'AWS::StackName'
  CacheSubnetGroup:                                      ④
```

```
    Type: 'AWS::MemoryDB::SubnetGroup'
    Properties:
      SubnetGroupName: !Ref 'AWS::StackName'
      SubnetIds:
      - !Ref SubnetA                                    ⑤
      - !Ref SubnetB
  CacheCluster:                                         ⑥
    Type: 'AWS::MemoryDB::Cluster'
    Properties:
      ACLName: 'open-access'                            ⑦
      ClusterName: !Ref 'AWS::StackName'
      EngineVersion: '6.2'                              ⑧
      NodeType: 'db.t4g.small'                          ⑨
      NumReplicasPerShard: 0                            ⑩
      NumShards: 1                                      ⑪
      ParameterGroupName: !Ref CacheParameterGroup
      SecurityGroupIds:
      - !Ref CacheSecurityGroup
      SubnetGroupName: !Ref CacheSubnetGroup
      TLSEnabled: false                                 ⑫
```

① The security group controlling traffic to the cache cluster

② The parameter group allows you to configure the cache cluster.

③ However, we are going with the defaults for a Redis 6–compatible cluster here.

④ The subnet groups specifies the subnets the cluster should use.

⑤ We are using two subnets for high availability here.

⑥ Creates and configures the cache cluster

⑦ Disables authentication and authorization to simplify the example

⑧ The Redis engine version

⑨ We are using the smallest available node type.

⑩ We are disabling replication to minimize costs for the example.

⑪ A single shard is enough for testing purposes. Adding shards allows you to scale the available memory in the cluster.

⑫ Disables encryption in transit to simplify the example

Want to have a deeper look into the CloudFormation template?

Where is the template located?

You can find the template on GitHub. Download a snapshot of the repository at **https://github.com/AWSinAction/code3/archive/main.zip**. The file we're talking about is located at chapter11/memorydb-minimal.yaml. On S3, the same file is located at **http://s3.amazonaws.com/awsinaction-code3/chapter11/memorydb-minimal.yaml**.

We are reusing the gaming example from section 11.1. Hopefully, your CloudFormation stack `memorydb-minimal` reached status `CREATE_COMPLETE` already. If so, open the Outputs tab and copy the EC2 instance ID as well as the cache address. Next, use the Session Manager to connect to the EC2 instance. Afterward, use the steps you used previously in section 11.1 to play with Redis. Make sure to replace `$CacheAddress` with the value from the CloudFormation outputs:

```
$ sudo amazon-linux-extras install -y redis6      ①
$ redis-cli -h $CacheAddress                      ②
> SET session:gamer1 online EX 15                 ③
OK
> GET session:gamer1                              ④
"online"
> GET session:gamer1                              ⑤
(nil)                                             ⑥
> ZADD highscore 100 "gamer1"                     ⑦
(integer) 1
> ZADD highscore 50 "gamer2"                      ⑧
(integer) 1
> ZADD highscore 150 "gamer3"                     ⑨
(integer) 1
> ZADD highscore 5 "gamer4"                       ⑩
(integer) 1
> ZRANGE highscore -3 -1 WITHSCORES               ⑪
1) "gamer2"                                       ⑫
2) "50"
3) "gamer1"
4) "100"
5) "gamer3"
6) "150"
> quit                                            ⑬
```

① Installs the Redis CLI

② Connects to the Redis cluster node. Don't forget to replace $CacheAddress with the output from the CloudFormation stack.

③ Stores a session for user gamer1 with a time-to-live of 15 seconds

④ Gets the session for user gamer1

⑤ Waits 15 seconds and gets the session for gamer1 again

⑥ The response is empty, because the key was automatically deleted after 15 seconds.

⑦ Adds gamer1 with a score of 100 to the sorted set named highscore

⑧ Adds gamer2 with a score of 50 to the sorted set named highscore

⑨ Adds gamer3 with a score of 150 to the sorted set named highscore

⑩ Adds gamer4 with a score of 5 to the sorted set named highscore

⑪ Gets the top three gamers from the highscore list

⑫ The list is sorted ascending and includes the user followed by its score.

⑬ Quits the Redis CLI

MemoryDB behaves like ElastiCache for Redis, but it comes with persistence guarantees, which make it possible to use it as the primary database.

> Cleaning up
> It's time to delete the CloudFormation stack named `memorydb-minimal`. Use the AWS Management Console or the following command to do so:
>
> ```
> $ aws cloudformation delete-stack --stack-name memorydb-minimal.
> ```

You are now equipped to select the best-fitting in-memory database engine and deployment option for your use case. In the next section, you will take a closer look at the security aspects of ElastiCache to control access to your cache cluster.

11.3 Controlling cache access

Controlling access to data stored in ElastiCache is very similar to the way it works with RDS (see section 11.4). ElastiCache is protected by the following four layers:

- *Identity and Access Management (IAM)*—Controls which IAM user, group, or role is allowed to administer an ElastiCache cluster.
- *Security groups*—Restricts incoming and outgoing traffic to ElastiCache nodes.
- *Cache engine*—Redis >6.0 supports authentication and authorization with role-based access control (RBAC). Memcached does not support authentication.
- *Encryption*—Optionally, data can be encrypted at rest and in transit.

11.3.1 Controlling access to the configuration

Access to the ElastiCache service is controlled with the help of the IAM service. The IAM service is responsible for controlling access to actions like creating, updating, and deleting a cache cluster. IAM doesn't manage access inside the cache; that's the job of the cache engine. An IAM policy defines the configuration and management actions a user, group, or role is allowed to execute on the ElastiCache service. Attaching the IAM policy to IAM users, groups, or roles controls which entity can use the policy to configure an ElastiCache cluster. You can get a complete list of IAM actions and resource-level permissions supported at **http://mng.bz/WM9x**.

SECURITY WARNING It's important to understand that you don't control access to the cache nodes using IAM. Once the nodes are created, security groups control the access on the network layer. Redis optionally supports user authentication and authorization.

11.3.2 Controlling network access

Network access is controlled with security groups. Remember the security group from the minimal CloudFormation template in section 11.1 where access to port 6379 (Redis) was allowed for all IP addresses. But because cluster nodes have only private IP addresses this restricts access to the VPC, as shown here:

```
Resources:
  # [...]
  CacheSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: cache
      VpcId: !Ref VPC
      SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 6379
```

```
          ToPort: 6379
          CidrIp: '0.0.0.0/0'
```

You should improve this setup by working with two security groups. To control traffic as tightly as possible, you will not allow traffic from certain IP addresses. Instead, you create two security groups. The client security group will be attached to all EC2 instances communicating with the cache cluster (your web or application servers). The cache cluster security group allows inbound traffic on port 6379 only for traffic that comes from the client security group. This way you can have a dynamic fleet of clients who are allowed to send traffic to the cache cluster, as shown next. You used the same approach in section 5.4:

```
Resources:
  # [...]
  ClientSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'cache-client'
      VpcId: !Ref VPC
  CacheSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: cache
      VpcId: !Ref VPC
      SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 6379
        ToPort: 6379
        SourceSecurityGroupId: !Ref ClientSecurityGroup          ①
```

① Only allows access from the ClientSecurityGroup

Attach the `ClientSecurityGroup` to all EC2 instances that need access to the cache cluster. This way, you allow access only to the EC2 instances that really need access.

Keep in mind that ElastiCache nodes always have private IP addresses. This means that you can't accidentally expose a Redis or Memcached cluster to the internet. You still want to use security groups to implement the principle of least privilege.

11.3.3 Controlling cluster and data access

Unfortunately, ElastiCache for Memcached does not provide user authentication. However, you have two different ways to authenticate users with

ElastiCache for Redis:

- Basic token-based authentication
- Users with RBAC

Use token-based authentication when all clients are allowed to read and manipulate all the data stored in the cluster. Use RBAC if you need to restrict access for different users, for example, to make sure the frontend is only allowed to read some of the data and the backend is able to write and read all the data. Keep in mind that when using authentication, you should also enable encryption in-transit to make sure to not transmit secrets in plain text.

In the next section, you'll learn how to use ElastiCache for Redis in a real-world application called Discourse.

11.4 Installing the sample application Discourse with CloudFormation

Small communities, like football clubs, reading circles, or dog schools, benefit from having a place where members can communicate with each other. Discourse is an open source software application for providing modern forums to your community. The forum software is written in Ruby using the Rails framework. Figure 11.8 gives you an impression of Discourse. Wouldn't that be a perfect place for your community to meet? In this section, you will learn how to set up Discourse with CloudFormation. Discourse is also perfectly suited for you to learn about ElastiCache because it requires a Redis in-memory database acting as a caching layer.

Figure 11.8 Discourse: A platform for community discussion

Discourse requires PostgreSQL as its main data store and uses Redis to cache data and process transient data. In this section, you'll create a CloudFormation template with all the components necessary to run Discourse. Finally, you'll create a CloudFormation stack based on the template to test your work. The necessary components follow:

- *VPC*—Network configuration
- *Cache*—Security group, subnet group, cache cluster
- *Database*—Security group, subnet group, database instance
- *Virtual machine*—Security group, EC2 instance

You'll start with the first component and extend the template in the rest of this section.

11.4.1 VPC: Network configuration

In section 5.5, you learned all about private networks on AWS. If you can't follow the next listing, you could go back to section 5.5 or continue with the next step—understanding the network is not key to get Discourse running.

Listing 11.2 CloudFormation template for Discourse: VPC

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'AWS in Action: chapter 11'
Parameters:                                          ①
  AdminEmailAddress:
    Description: 'Email address of admin user'
    Type: 'String'
Resources:
  VPC:                                               ②
    Type: 'AWS::EC2::VPC'
    Properties:
      CidrBlock: '172.31.0.0/16'
      EnableDnsHostnames: true
  InternetGateway:                                   ③
    Type: 'AWS::EC2::InternetGateway'
    Properties: {}
  VPCGatewayAttachment:                              ④
    Type: 'AWS::EC2::VPCGatewayAttachment'
    Properties:
      VpcId: !Ref VPC
      InternetGatewayId: !Ref InternetGateway
  SubnetA:                                           ⑤
    Type: 'AWS::EC2::Subnet'
    Properties:
      AvailabilityZone: !Select [0, !GetAZs '']
      CidrBlock: '172.31.38.0/24'
      VpcId: !Ref VPC
  SubnetB: # [...]                                   ⑥
  RouteTable:                                        ⑦
    Type: 'AWS::EC2::RouteTable'
    Properties:
      VpcId: !Ref VPC
  SubnetRouteTableAssociationA:                      ⑧
    Type: 'AWS::EC2::SubnetRouteTableAssociation'
    Properties:
      SubnetId: !Ref SubnetA
      RouteTableId: !Ref RouteTable
```

```
  # [...]
  RouteToInternet:                                    ⑨
    Type: 'AWS::EC2::Route'
    Properties:
      RouteTableId: !Ref RouteTable
      DestinationCidrBlock: '0.0.0.0/0'
      GatewayId: !Ref InternetGateway
    DependsOn: VPCGatewayAttachment
```

① The email address of the Discourse admin must be valid.

② Creates a VPC in the address range 172.31.0.0/16

③ We want to access Discourse from the internet, so we need an internet gateway.

④ Attaches the internet gateway to the VPC

⑤ Creates a subnet in the address range 172.31.38.0/24 in the first availability zone (array index 0)

⑥ Creates a second subnet in the address range 172.31.37.0/24 in the second availability zone (properties omitted)

⑦ Creates a route table that contains the default route, which routes all subnets in a VPC

⑧ Associates the first subnet with the route table

⑨ Adds a route to the internet via the internet gateway

The following listing adds the required network access control list.

Listing 11.3 CloudFormation template for Discourse: VPC NACLs

```
Resources:
  # [...]
  NetworkAcl:                                ①
    Type: AWS::EC2::NetworkAcl
    Properties:
      VpcId: !Ref VPC
  SubnetNetworkAclAssociationA:              ②
    Type: 'AWS::EC2::SubnetNetworkAclAssociation'
    Properties:
      SubnetId: !Ref SubnetA
      NetworkAclId: !Ref NetworkAcl
  # [...]
```

```
  NetworkAclEntryIngress:                    ③
    Type: 'AWS::EC2::NetworkAclEntry'
    Properties:
      NetworkAclId: !Ref NetworkAcl
      RuleNumber: 100
      Protocol: -1
      RuleAction: allow
      Egress: false
      CidrBlock: '0.0.0.0/0'
  NetworkAclEntryEgress:                     ④
    Type: 'AWS::EC2::NetworkAclEntry'
    Properties:
      NetworkAclId: !Ref NetworkAcl
      RuleNumber: 100
      Protocol: -1
      RuleAction: allow
      Egress: true
      CidrBlock: '0.0.0.0/0'
```

① Creates an empty network ACL

② Associates the first subnet with the network ACL

③ Allows all incoming traffic on the network ACL. (You will use security groups later as a firewall.)

④ Allows all outgoing traffic on the network ACL

The network is now properly configured using two public subnets. Let's configure the cache next.

11.4.2 Cache: Security group, subnet group, cache cluster

You will add the ElastiCache for Redis cluster now. You learned how to describe a minimal cache cluster earlier in this chapter. This time, you'll add a few extra properties to enhance the setup. The next code listing contains the CloudFormation resources related to the cache.

Listing 11.4 CloudFormation template for Discourse: Cache

```
Resources:
  # [...]
  CacheSecurityGroup:                                    ①
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: cache
      VpcId: !Ref VPC
```

```
    CacheSecurityGroupIngress:                                    ②
      Type: 'AWS::EC2::SecurityGroupIngress'
      Properties:
        GroupId: !Ref CacheSecurityGroup
        IpProtocol: tcp
        FromPort: 6379                                            ③
        ToPort: 6379
        SourceSecurityGroupId: !Ref InstanceSecurityGroup         ④
    CacheSubnetGroup:                                             ⑤
      Type: 'AWS::ElastiCache::SubnetGroup'
      Properties:
        Description: cache
        SubnetIds:
        - Ref: SubnetA
        - Ref: SubnetB
    Cache:                                                        ⑥
      Type: 'AWS::ElastiCache::CacheCluster'
      Properties:
        CacheNodeType: 'cache.t2.micro'
        CacheSubnetGroupName: !Ref CacheSubnetGroup
        Engine: redis
        EngineVersion: '6.2'                                      ⑦
        NumCacheNodes: 1
        VpcSecurityGroupIds:
        - !Ref CacheSecurityGroup
```

① The security group to control incoming and outgoing traffic to/from the cache

② To avoid a cyclic dependency, the ingress rule is split into a separate CloudFormation resource.

③ Redis runs on port 6379.

④ The InstanceSecurityGroup resource is not yet specified; you will add this later when you define the EC2 instance that runs the web server.

⑤ The cache subnet group references the VPC subnets.

⑥ Creates a single-node Redis cluster

⑦ You can specify the exact version of Redis that you want to run. Otherwise, the latest version is used, which may cause incompatibility issues in the future. We recommend always specifying the version.

The single-node Redis cache cluster is now defined. Discourse also requires a PostgreSQL database, which you'll define next.

11.4.3 Database: Security group, subnet group, database instance

PostgreSQL is a powerful, open source relational database. If you are not familiar with PostgreSQL, that's not a problem at all. Luckily, the RDS service will provide a managed PostgreSQL database for you. You learned about RDS in chapter 10. The following listing shows the section of the template that defines the RDS instance.

Listing 11.5 CloudFormation template for Discourse: Database

```
Resources:
  # [...]
  DatabaseSecurityGroup:                                          ①
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: database
      VpcId: !Ref VPC
  DatabaseSecurityGroupIngress:
    Type: 'AWS::EC2::SecurityGroupIngress'
    Properties:
      GroupId: !Ref DatabaseSecurityGroup
      IpProtocol: tcp
      FromPort: 5432                                              ②
      ToPort: 5432
      SourceSecurityGroupId: !Ref InstanceSecurityGroup           ③
  DatabaseSubnetGroup:                                            ④
    Type: 'AWS::RDS::DBSubnetGroup'
    Properties:
      DBSubnetGroupDescription: database
      SubnetIds:
      - Ref: SubnetA
      - Ref: SubnetB
  Database:                                                       ⑤
    Type: 'AWS::RDS::DBInstance'
    DeletionPolicy: Delete
    Properties:
      AllocatedStorage: 5
      BackupRetentionPeriod: 0                                    ⑥
      DBInstanceClass: 'db.t2.micro'
      DBName: discourse                                           ⑦
      Engine: postgres                                            ⑧
      EngineVersion: '12.10'                                      ⑨
      MasterUsername: discourse                                   ⑩
      MasterUserPassword: discourse                               ⑪
      VPCSecurityGroups:
      - !Sub ${DatabaseSecurityGroup.GroupId}
      DBSubnetGroupName: !Ref DatabaseSubnetGroup
    DependsOn: VPCGatewayAttachment
```

① Traffic to/from the RDS instance is protected by a security group.

② PostgreSQL runs on port 5432 by default.

③ The InstanceSecurityGroup resource is not yet specified; you'll add this later when you define the EC2 instance that runs the web server.

④ RDS also uses a subnet group to reference the VPC subnets.

⑤ The database resource

⑥ Disables backups; you want to turn this on (value > 0) in production.

⑦ RDS created a database for you in PostgreSQL.

⑧ Discourse requires PostgreSQL.

⑨ We recommend always specifying the version of the engine to avoid future incompatibility issues.

⑩ PostgreSQL admin username

⑪ PostgreSQL admin password; you want to change this in production.

Have you noticed the similarity between RDS and ElastiCache? The concepts are similar, which makes it easier for you to work with both services. Only one component is missing: the EC2 instance that runs the web server.

11.4.4 Virtual machine: Security group, EC2 instance

Discourse is a Ruby on Rails application, so you need an EC2 instance to host the application. Because it is very tricky to install the required Ruby environment with all its dependencies, we are using the recommended way to run Discourse, which is running a container with Docker on the virtual machine. However, running containers is not in scope of this chapter. You will learn more about deploying containers on AWS in chapter 18.

Discourse requires SMTP to send email

The forum software application Discourse requires access to an SMTP server to send email. Unfortunately, operating an SMTP server on your own is almost impossible, because most providers will not accept emails or flag them as spam. That's why we are using the Simple Email Service

(SES) provided by AWS. SES requires verification before sending emails. Start with enabling sandbox mode by adding your email address to the allowlist as follows:

1. Open SES in the AWS Management Console.
2. Select Verified Identities from the menu.
3. Click Create Identity.
4. Choose the identity type Email Address.
5. Type in your email address.
6. Click Create Identity.
7. Check your inbox, and click the verification link.

Please note: you need to request production access to ensure SES is delivering emails to any address. To follow our example, however, this isn't needed.
The next listing defines the virtual machine and the startup script to install and configure Discourse.

Listing 11.6 CloudFormation template for Discourse: Virtual machine

```
Resources:
  # [...]
  InstanceSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'vm'
      SecurityGroupIngress:
      - CidrIp: '0.0.0.0/0'
        FromPort: 80
        IpProtocol: tcp
        ToPort: 80
      VpcId: !Ref VPC
  Instance:
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: !FindInMap [RegionMap, !Ref 'AWS::Region', AMI]
      InstanceType: 't2.micro'
      IamInstanceProfile: !Ref InstanceProfile
      NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeleteOnTermination: true
        DeviceIndex: 0
        GroupSet:
        - !Ref InstanceSecurityGroup
        SubnetId: !Ref SubnetA
      BlockDeviceMappings:
      - DeviceName: '/dev/xvda'
```

```
          Ebs:
            VolumeSize: 16
            VolumeType: gp2
        UserData:
          'Fn::Base64': !Sub |
            #!/bin/bash -x
            bash -ex << "TRY"
              # [...]

              # install and start docker
              yum install -y git
              amazon-linux-extras install docker -y
              systemctl start docker

              docker run --restart always -d -p 80:80 --name discourse \
                -e "UNICORN_WORKERS=3" \
                # [...]
                -e "RUBY_ALLOCATOR=/usr/lib/libjemalloc.so.1" \
                public.ecr.aws/awsinaction/discourse:3rd /sbin/boot

              docker exec discourse /bin/sh -c /
                "cd /var/www/discourse && rake db:migrate"
              docker restart discourse
            TRY
            /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} /
              --resource Instance --region ${AWS::Region}
      CreationPolicy:
        ResourceSignal:
          Timeout: PT15M
      DependsOn:
      - VPCGatewayAttachment
```

① Allows HTTP traffic from the public internet

② The virtual machine that runs Discourse

③ Increases the default volume size from 8 GB to 16 GB

④ Installs and starts Docker

⑤ Creates and launches a Docker container

⑥ Runs the database migration script to initialize or migrate the database

⑦ Restarts the container to make sure the database migration is in effect

You've reached the end of the template. All components are defined now. It's time to create a CloudFormation stack based on your template to see

whether it works.

11.4.5 Testing the CloudFormation template for Discourse

Let's create a stack based on your template to create all the resources in your AWS account, as shown next. To find the full code for the template, go to /chapter11/discourse.yaml in the book's code folder. Use the AWS CLI to create the stack. Don't forget to replace `$AdminEmailAddress` with your e-mail address:

```
$ aws cloudformation create-stack --stack-name discourse \
➥ --template-url https:/ /s3.amazonaws.com/\
➥awsinaction-code3/chapter11/discourse.yaml \
➥ --parameters \
➥ "ParameterKey=AdminEmailAddress,ParameterValue=$AdminEmailAddress" \
➥ --capabilities CAPABILITY_NAMED_IAM
```

Where is the template located?

You can find the template on GitHub. You can download a snapshot of the repository at **https://github.com/AWSinAction/code3/archive/main.zip**. The file we're talking about is located at chapter11/discourse.yaml. On S3, the same file is located at **https://s3.amazonaws.com/awsinaction-code3/chapter11/discourse.yaml**.

The creation of the stack can take up to 20 minutes. After the stack has been created, get the public IP address of the EC2 instance from the stack's output by executing the following command:

```
$ aws cloudformation describe-stacks --stack-name discourse \
➥ --query "Stacks[0].Outputs[1].OutputValue"
```

Open a web browser and insert the IP address in the address bar to open your Discourse website. Figure 11.9 shows the website. Click Register to create an admin account.

Figure 11.9 Discourse: First screen after a fresh install

You will receive an email to activate your account. Note that this email might be hidden in your spam folder. After activation, the setup wizard is started, which you have to complete to configure your forum. After you complete the wizard and have successfully installed Discourse, don't

delete the CloudFormation stack because you'll use the setup in the next section.

Congratulations—you have deployed Discourse, a rather complex web application, using Redis as a cache to speed up loading times. You've experienced that ElastiCache is for in-memory databases, like RDS is for relational databases. Because ElastiCache is a relevant part of the architecture, you need to make sure you monitor the cache infrastructure to avoid performance issues or even downtimes. You will learn about monitoring ElastiCache in the following section.

11.5 Monitoring a cache

CloudWatch is the service on AWS that stores all kinds of metrics. Like other services, ElastiCache nodes send metrics to CloudWatch as well. The most important metrics to watch follow:

- `CPUUtilization`—The percentage of CPU utilization.
- `SwapUsage`—The amount of swap used on the host, in bytes. *Swap* is space on disk that is used if the system runs out of physical memory.
- `Evictions`—The number of nonexpired items the cache evicted due to the memory limit.
- `ReplicationLag`—This metric is applicable only for a Redis node running as a read replica. It represents how far behind, in seconds, the replica is in applying changes from the primary node. Usually this number is very low.

In this section we'll examine those metrics in more detail and give you some hints about useful thresholds for defining alarms on those metrics to set up production-ready monitoring for your cache.

11.5.1 Monitoring host-level metrics

The virtual machines running underneath the ElastiCache service report CPU utilization and swap usage. CPU utilization usually gets problematic when crossing 80–90%, because the wait time explodes. But things are more tricky here. Redis is single-threaded. If you have many cores, the overall CPU utilization can be low, but one core can be at 100% utilization. Use the `EngineCPUUtilization` metric to monitor the Redis process CPU utilization. Swap usage is a different topic. You run an in-memory cache, so if the virtual machine starts to swap (move memory to disk), the performance will suffer. By default, ElastiCache for Memcached and Redis is configured to limit memory consumption to a value smaller than what's physical available (you can tune this) to have room for other

resources (e.g., the kernel needs memory for each open socket). But other processes (such as kernel processes) are also running, and they may start to consume more memory than what's available. You can solve this issue by increasing the memory of the cache, either by increasing the node type or by adding more shards.

Queuing theory: Why 80–90%?

Imagine you are the manager of a supermarket. What should be the goal for daily utilization of your cashiers? It's tempting to go for a high number, maybe 90%. But it turns out that the wait time for your customers is very high when your cashiers are used for 90% of the day, because customers don't arrive at the same time at the queue. The theory behind this is called *queuing theory*, and it turns out that wait time is exponential to the utilization of a resource. This applies not only to cashiers but also to network cards, CPU, hard disks, and so on. Keep in mind that this sidebar simplifies the theory and assumes an M/D/1 queuing system: Markovian arrivals (exponentially distributed arrival times), deterministic service times (fixed), one service center. To learn more about queuing theory applied to computer systems, we recommend *Systems Performance: Enterprise and the Cloud* by Brendan Gregg (Prentice Hall, 2013) to get started.

When you go from 0% utilization to 60%, wait time doubles. When you go to 80%, wait time has tripled. When you to 90%, wait time is six times higher, and so on. If your wait time is 100 ms during 0% utilization, you already have 300 ms wait time during 80% utilization, which is already slow for an e-commerce website.

You might set up an alarm to trigger if the 10-minute average of the `EngineCPUUtilization` metric is higher than 80% for one out of one data points, and if the 10-minute average of the `SwapUsage` metric is higher than 67108864 (64 MB) for one out of one data points. These numbers are just a rule of thumb. You should load-test your system to verify that the thresholds are high or low enough to trigger the alarm before application performance suffers.

11.5.2 Is my memory sufficient?

The `Evictions` metric is reported by Memcached and Redis. If the cache is running out of memory and you try to add a new key-value pair, an old key-value pair needs to be deleted first. This is called an eviction. By default, Redis deletes the least recently used key, but only for keys that de-

fine a TTL. This strategy is called `volatile-lru`. On top of that, the following eviction strategies are available:

- `allkeys-lru`—Removes the least recently used key among all keys
- `volatile-random`—Removes a random key among keys with TTL
- `allkeys-random`—Removes a random key among all keys
- `volatile-ttl`—Removes the key with the shortest TTL
- `noeviction`—Does not evict any key

Usually, high eviction rates are a sign that you either aren't using a TTL to expire keys after some time or that your cache is too small. You can solve this issue by increasing the memory of the cache, either by increasing the node type or by adding more shards. You might set an alarm to trigger if the 10-minute average of the `Evictions` metric is higher than 1000 for one out of one data points.

11.5.3 Is my Redis replication up-to-date?

The `ReplicationLag` metric is applicable only for a node running as a read replica. It represents how far behind, in seconds, the replica is in applying changes from the primary node. The higher this value, the more out of date the replica is. This can be a problem because some users of your application will see very old data. In the gaming application, imagine you have one primary node and one replica node. All reads are performed by either the primary or the replica node. The `ReplicationLag` is 600, which means that the replication node looks like the primary node looked 10 minutes before. Depending on which node the user hits when accessing the application, they could see 10-minute-old data.

What are reasons for a high `ReplicationLag`? There could be a problem with the sizing of your cluster—for example, your cache cluster might be at capacity. Typically this will be a sign to increase the capacity by adding shards or replicas. You might set an alarm to trigger if the 10-minute average of the `ReplicationLag` metric is higher than 30 for one consecutive period.

Cleaning up

It's time to delete the running CloudFormation stack, as shown here:

```
$ aws cloudformation delete-stack --stack-name discourse
```

11.6 Tweaking cache performance

Your cache can become a bottleneck if it can no longer handle the requests with low latency. In the previous section, you learned how to monitor your cache. In this section, you learn what you can do if your monitoring data shows that your cache is becoming the bottleneck (e.g., if you see high CPU or network usage). Figure 11.10 contains a decision tree that you can use to resolve performance issues with ElastiCache. The strategies are described in more detail in the rest of this section.

Figure 11.10 An ElastiCache decision tree to resolve performance issues

Three strategies for tweaking the performance of your ElastiCache cluster follow:

- *Selecting the right cache node type*—A bigger instance type comes with more resources (CPU, memory, network) so you can scale vertically.
- *Selecting the right deployment option*—You can use sharding or read replicas to scale horizontally.
- *Compressing your data*—If you shrink the amount of data being transferred and stored, you can also tweak performance.

11.6.1 Selecting the right cache node type

So far, you used the cache node type `cache.t2.micro`, which comes with one vCPU, ~0.6 GB memory, and low to moderate network performance. You used this node type because it's part of the Free Tier. But you can also use more powerful node types on AWS. The upper end is the `cache.r6gd.16xlarge` with 64 vCPUs, ~419 GB memory, and 25 Gb network.

As a rule of thumb: for production traffic, select a cache node type with at least 2 vCPUs for real concurrency, enough memory to hold your data set with some space to grow (say, 20%; this also avoids memory fragmentation), and at least high network performance. The `cache.r6g.large` is

an excellent choice for a small node size: 2 vCPUs, ~13 GB, and up to 10 GB of network. This may be a good starting point when considering how many shards you may want in a clustered topology, and if you need more memory, move up a node type. You can find the available node types at **https://aws.amazon.com/elasticache/pricing/**.

11.6.2 Selecting the right deployment option

By replicating data, you can distribute read traffic to multiple nodes within the same replica group. By replicating data from the primary node to one or multiple replication nodes, you are increasing the number of nodes that are able to process read requests and, therefore, increase the maximum throughput. On top of that, by sharding data, you split the data into multiple buckets. Each bucket contains a subset of the data. Each shard is stored on a primary node and is optionally synchronized to replication nodes as well. This further increases the number of nodes available to answer incoming reads—and even writes.

Redis supports the concept of replication, where one node in a node group is the primary node accepting read and write traffic, while the replica nodes only accept read traffic, which allows you to scale the read capacity. The Redis client has to be aware of the cluster topology to select the right node for a given command. Keep in mind that the replicas are synchronized asynchronously. This means that the replication node eventually reaches the state of the primary node.

Both Memcached and Redis support the concept of sharding. With sharding, a single cache cluster node is no longer responsible for all the keys. Instead the key space is divided across multiple nodes. Both Redis and Memcached clients implement a hashing algorithm to select the right node for a given key. By sharding, you can increase the capacity of your cache cluster.

As a rule of thumb: when a single node can no longer handle the amount of data or the requests, and if you are using Redis with mostly read traffic, then you should use replication. Replication also increases the availability at the same time (at no extra cost).

It is even possible to increase and decrease the number of shards automatically, based on utilization metrics, by using Application Auto Scaling. See **http://mng.bz/82d2** to learn more.

Last but not least, replicating data is possible not only within a cluster but also between clusters running in other regions of the world. For example,

if you are operating a popular online game, it is necessary to minimize latency by deploying your cloud infrastructure to the United States, Europe, and Asia. By using ElastiCache Global Datastore, you are able to replicate a Redis cluster located in Ireland to North Virginia and Singapore, for example.

11.6.3 Compressing your data

Instead of sending large values (and also keys) to your cache, you can compress the data before you store it in the cache. When you retrieve data from the cache, you have to uncompress it on the application before you can use the data. Depending on your data, compressing data can have a significant effect. We saw memory reductions to 25% of the original size and network transfer savings of the same size. Please note that this approach needs to be implemented in your application.

As a rule of thumb: compress your data using an algorithm that is best suited for your data. Most likely the `zlib` library is a good starting point. You might want to experiment with a subset of your data to select the best compression algorithm that is also supported by your programming language.

On top of that, ElastiCache for Redis also supports data tiering to reduce costs. With data tiering enabled, ElastiCache will move parts of your data from memory to solid-state disks—a tradeoff between costs and latency. Using data tiering is most interesting if your workload reads only parts of the data frequently.

Summary

- A caching layer can speed up your application significantly, while also lowering the costs of your primary data store.
- To keep the cache in sync with the database, items usually expire after some time, or a write-through strategy is used.
- When the cache is full, the least frequently used items are usually evicted.
- ElastiCache can run Memcached or Redis clusters for you. Depending on the engine, different features are available. Memcached and Redis are open source, but AWS added engine-level enhancements.
- Memcached is a simple key-value store allowing you to scale available memory and throughput by adding additional machines—called shards—to the cluster.
- Redis is a more advanced in-memory database that supports complex data structures like sorted sets.

- MemoryDB is an alternative to ElastiCache and provides an in-memory database with strong persistence guarantees. MemoryDB comes with Redis compatibility.
- CloudWatch provides insights into the utilization and performance of ElastiCache.