Performance at Scale with Amazon ElastiCache

Nate Wiger

May 2015

Contents

Abstract

In-memory caching improves application performance by storing frequently accessed data items in memory, so that they can be retrieved without access to the primary data store. Properly leveraging caching can result in an application that not only performs better, but also costs less at scale. Amazon ElastiCache is a managed service that reduces the administrative burden of deploying an in-memory cache in the cloud. Beyond caching, an in-memory data layer also enables advanced use cases, such as analytics and recommendation engines. This whitepaper lays out common ElastiCache design patterns, performance tuning tips, and important operational considerations to get the most out of an in-memory layer.

Introduction

An effective caching strategy is perhaps the single biggest factor in creating an app that performs well at scale. A brief look at the largest web, gaming, and mobile apps reveals that all apps at significant scale have a considerable investment in caching. Despite this, many developers fail to exploit caching to its full potential. This oversight can result in running larger database and application instances than needed. Not only does this approach decrease performance and add cost, but also it limits your ability to scale.

The in-memory caching provided by Amazon ElastiCache improves application performance by storing critical pieces of data in memory for fast access. You can use this caching to significantly improve latency and throughput for many read-heavy application workloads, such as social networking, gaming, media sharing, and Q&A portals. Cached information can include the results of database queries, computationally intensive calculations, or even remote API calls. In addition, compute-intensive workloads that manipulate data sets, such as recommendation engines and high-performance computing simulations, also benefit from an in-memory data layer. In these applications, very large datasets must be accessed in real-time across clusters of machines that can span hundreds of nodes. Manipulating this data in a disk-based store would be a significant bottleneck for these applications.

Amazon ElastiCache is a web service that makes it easy to deploy, operate, and scale an in-memory cache in the cloud. Amazon ElastiCache manages the work involved in setting up an in-memory service, from provisioning the AWS resources you request to installing the software. Using Amazon ElastiCache, you can add an in-memory caching layer to your application in a matter of minutes, with a few API calls. Amazon ElastiCache integrates with other Amazon web services such as Amazon Elastic Compute Cloud (Amazon EC2) and Amazon Relational Database Service (Amazon RDS), as well as deployment management solutions such as AWS CloudFormation, AWS Elastic Beanstalk, and AWS OpsWorks.

In this whitepaper, we'll walk through best practices for working with ElastiCache. We'll demonstrate common in-memory data design patterns, compare the two open source engines that ElastiCache supports, and show how ElastiCache fits into real-world application architectures such as web apps and online games. By the end of this paper, you should have a clear grasp of which caching strategies apply to your use case, and how you can use ElastiCache to deploy an in-memory caching layer for your app.

ElastiCache Overview

The Amazon ElastiCache architecture is based on the concept of deploying one or more cache clusters for your application. Once your cache cluster is up and running, the service automates common administrative tasks such as resource provisioning, failure detection and recovery, and software patching. Amazon ElastiCache provides detailed monitoring metrics associated with your cache nodes, enabling you to diagnose and react to issues very quickly. For example, you can set up thresholds and receive alarms if one of your cache nodes is overloaded with requests. You can launch an ElastiCache cluster by following the steps in the Getting Started section of the *Amazon ElastiCache User Guide.*

It's important to understand that Amazon ElastiCache is not coupled to your database tier. As far as Amazon ElastiCache nodes are concerned, your application is just setting and getting keys in a slab of memory. That being the case, you can use Amazon

ElastiCache with relational databases such as MySQL or Microsoft SQL Server; with NoSQL databases such as Amazon DynamoDB or MongoDB; or with no database tier at all, which is common for distributed computing applications. Amazon ElastiCache gives you the flexibility to deploy one, two, or more different cache clusters with your application, which you can use for differing types of datasets.

Alternatives to ElastiCache

In addition to using ElastiCache, you can cache data in AWS in other ways, each of which has its own pros and cons. Let's briefly review some of the alternatives:

- Amazon CloudFront content delivery network (CDN)—this approach is used to cache web pages, image assets, videos, and other static data at the edge, as close to end users as possible. In addition to using CloudFront with static assets, you can also place CloudFront in front of dynamic content, such as web apps. The important caveat here is that CloudFront only caches rendered page output. In web apps, games, and mobile apps, it's very common to have thousands of fragments of data, which are reused in multiple sections of the app. CloudFront is a valuable component of scaling a website, but it does not obviate the need for application caching.
- Amazon RDS Read Replicas—some database engines, such as MySQL, support the ability to attach asynchronous read replicas. Although useful, this ability is limited to providing data in a duplicate format of the primary database. You cannot cache calculations, aggregates, or arbitrary custom keys in a replica. Also, read replicas are not as fast as in-memory caches. Read replicas are more interesting for distributing data to remote sites or apps.
- On-host caching—a simplistic approach to caching is to store data on each Amazon EC2 application instance, so that it's local to the server for fast lookup. Don't do this. First, you get no efficiency from your cache in this case. As application instances scale up, they start with an empty cache, meaning they end up hammering the data tier. Second, cache invalidation becomes a nightmare. How are you going to reliably signal 10 or 100 separate EC2 instances to delete a given cache key? Finally, you rule out interesting use cases for in-memory caches, such as sharing data at high speed across a fleet of instances.

Let's turn our attention back to ElastiCache, and how it fits into your application.

Memcached vs. Redis

Amazon ElastiCache currently supports two different in-memory key-value engines. You can choose the engine you prefer when launching an ElastiCache cache cluster:

- Memcached—a widely adopted in-memory key store, and historically the gold standard of web caching. ElastiCache is protocol-compliant with Memcached, so

popular tools that you use today with existing Memcached environments will work seamlessly with the service. Memcached is also multithreaded, meaning it makes good use of larger Amazon EC2 instance sizes with multiple cores.

- Redis—an increasingly popular open-source key-value store that supports more advanced data structures such as sorted sets, hashes, and lists. Unlike Memcached, Redis has disk persistence built in, meaning you can use it for long-lived data. Redis also supports replication, which can be used to achieve Multi-AZ redundancy, similar to Amazon RDS.

Although both Memcached and Redis appear similar on the surface, in that they are both in-memory key stores, they are actually quite different in practice. Because of the replication and persistence features of Redis, ElastiCache manages Redis more as a relational database. Redis ElastiCache clusters are managed as stateful entities that include failover, similar to how Amazon RDS manages database failover.

Conversely, because Memcached is designed as a pure caching solution with no persistence, ElastiCache manages Memcached nodes as a pool that can grow and shrink, similar to an Amazon EC2 Auto Scaling group. Individual nodes are expendable, and ElastiCache provides additional capabilities here such as automatic node replacement and Auto Discovery.

When deciding between Memcached and Redis, here are a few questions to consider:

- Is object caching your primary goal, for example to offload your database? If so, use Memcached.
- Are you interested in as simple a caching model as possible? If so, use Memcached.
- Are you planning on running large cache nodes, and require multithreaded performance with utilization of multiple cores? If so, use Memcached.
- Do you want the ability to scale your cache horizontally as you grow? If so, use Memcached.
- Does your app need to atomically increment or decrement counters? If so, use either Redis or Memcached.
- Are you looking for more advanced data types, such as lists, hashes, and sets? If so, use Redis.
- Does sorting and ranking datasets in memory help you, such as with leaderboards? If so, use Redis.
- Are publish and subscribe (pub/sub) capabilities of use to your application? If so, use Redis.
- Is persistence of your key store important? If so, use Redis.

- Do you want to run in multiple AWS Availability Zones (Multi-AZ) with failover? If so, use Redis.

Although it's tempting to look at Redis as a more evolved Memcached due to its advanced data types and atomic operations, Memcached has a longer track record and the ability to leverage multiple CPU cores.

Because Memcached and Redis are so different in practice, we're going to address them separately in most of this paper. We will focus on using Memcached as an in-memory cache pool, and using Redis for advanced datasets such as game leaderboards and activity streams.

ElastiCache for Memcached

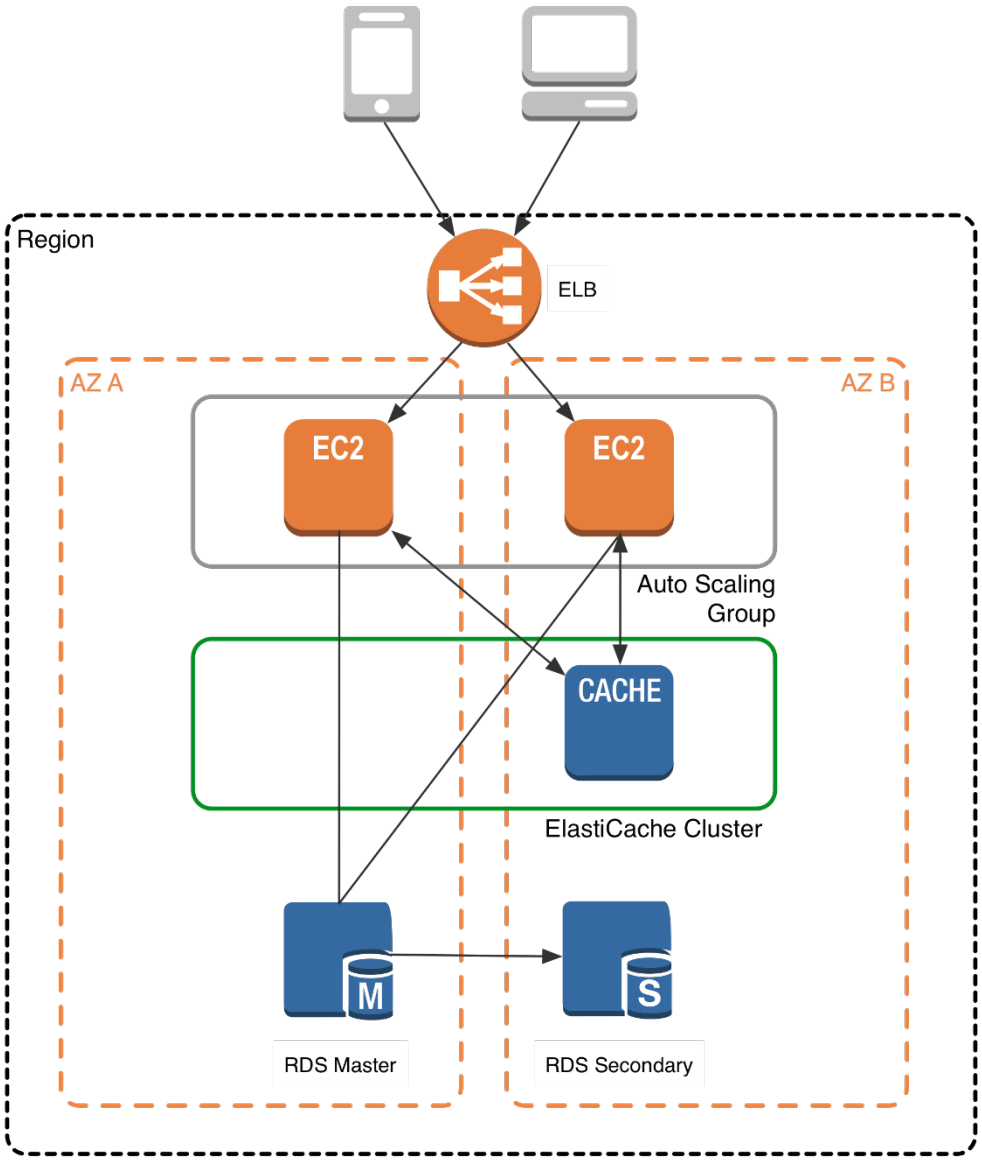
The primary goal of caching is typically to offload reads from your database or other primary data source. In most apps, you have hot spots of data that are regularly queried, but only updated periodically. Think of the front page of a blog or news site, or the top 100 leaderboard in an online game. In this type of case, your app can receive dozens, hundreds, or even thousands of requests for the same data before it's updated again. Having your caching layer handle these queries has several advantages. First, it's considerably cheaper to add an in-memory cache than to scale up to a larger database cluster. Second, an in-memory cache is also easier to scale out, because it's easier to distribute an in-memory cache horizontally than a relational database.

Last, a caching layer provides a request buffer in the event of a sudden spike in usage. If your app or game ends up on the front page of Reddit or the App Store, it's not unheard of to see a spike that is 10 to 100 times your normal application load. Even if you auto-scale your application instances, a 10x request spike will likely make your database very unhappy.

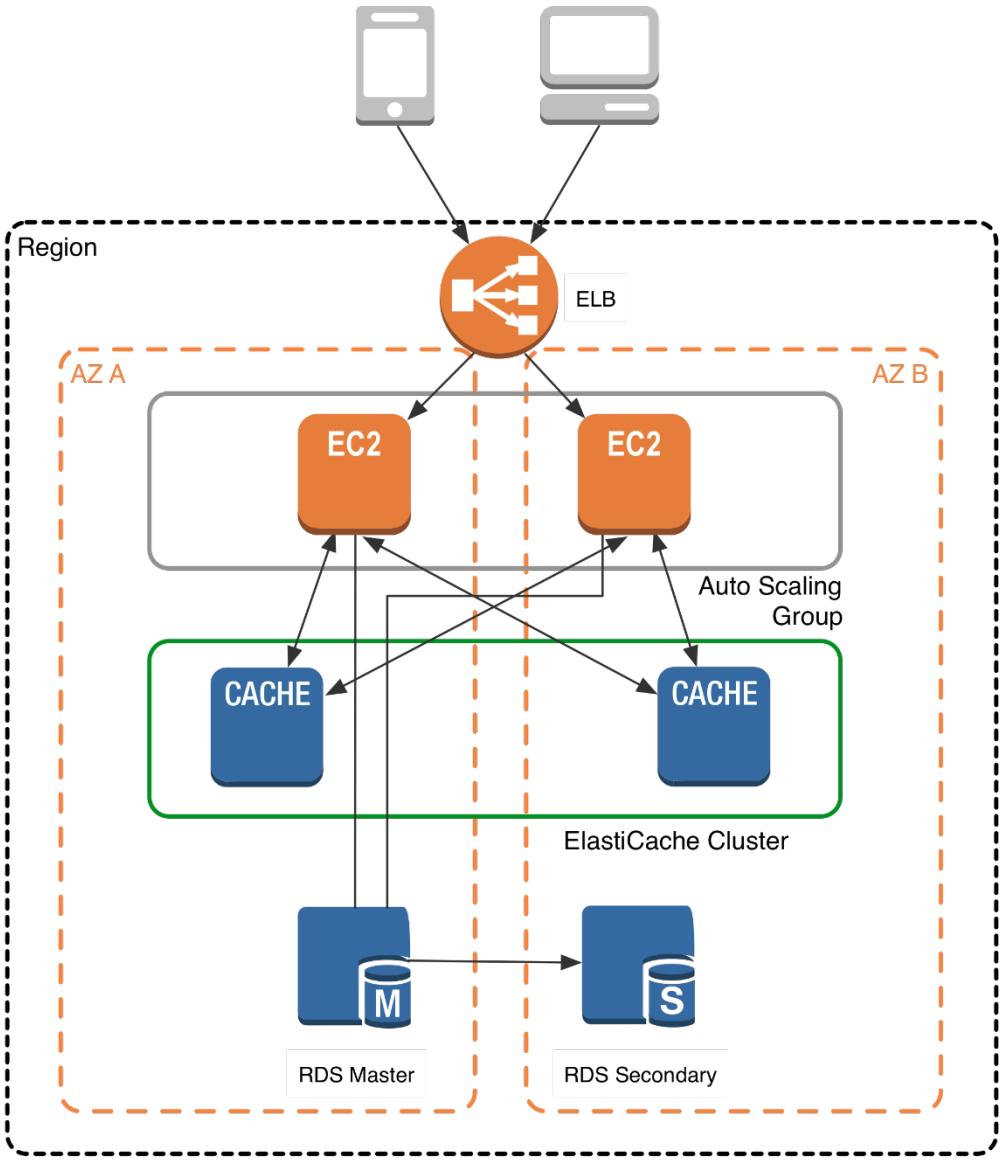
Let's focus on ElastiCache for Memcached first, because it is the best fit for a caching-focused solution. We'll revisit Redis later in the paper, and weigh its advantages and disadvantages.

Architecture with ElastiCache for Memcached

When you deploy an ElastiCache Memcached cluster, it sits in your application as a separate tier alongside your database. As mentioned previously, Amazon ElastiCache does not directly communicate with your database tier, or indeed have any particular knowledge of your database. A simplified deployment for a web application looks something like this:

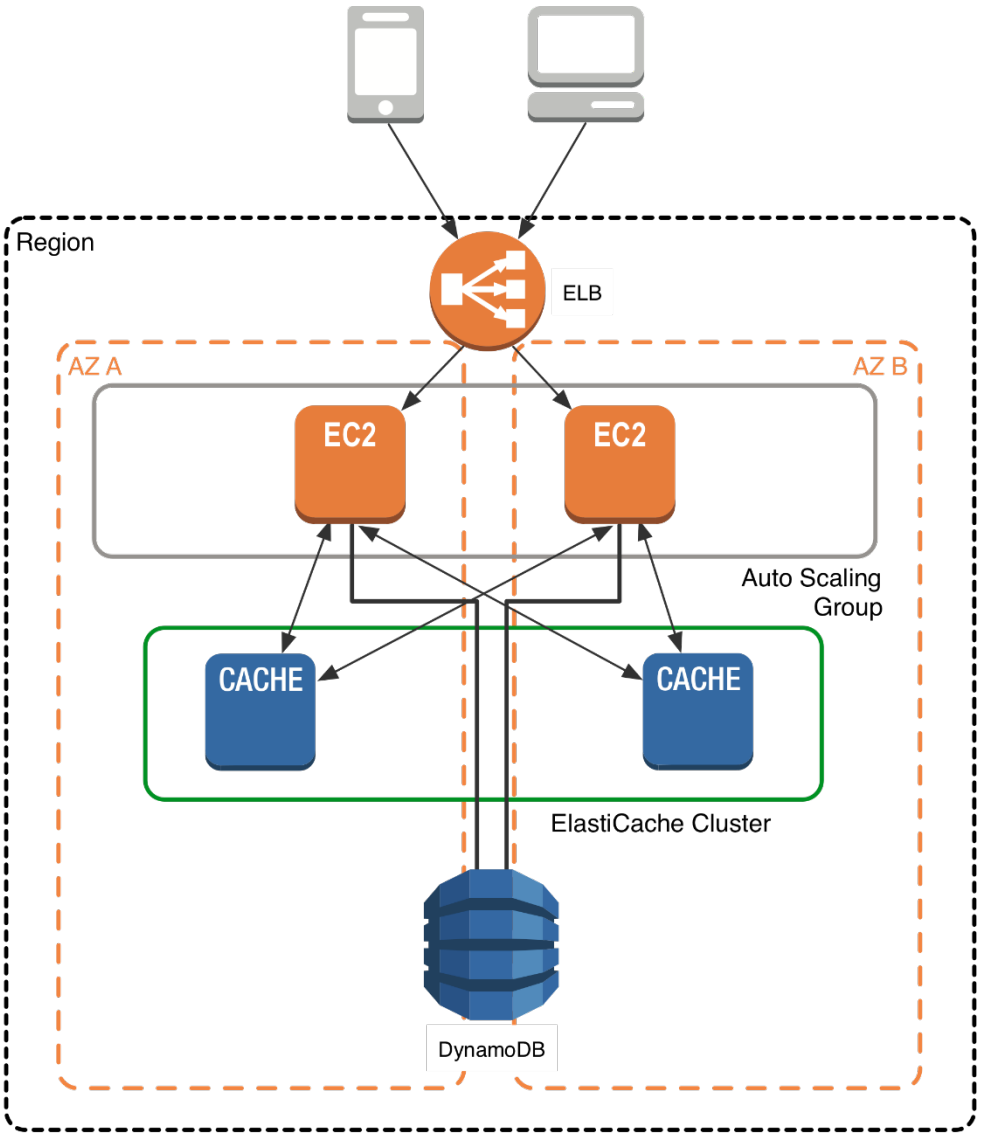


In this architecture diagram, the Amazon EC2 application instances are in an Auto Scaling group, located behind a load balancer using Elastic Load Balancing, which distributes requests among the instances. As requests come into a given EC2 instance, that EC2 instance is responsible for communicating with ElastiCache and the database tier. For development purposes, you can begin with a single ElastiCache node to test your application, and then scale to additional cluster nodes by modifying the ElastiCache cluster. As you add additional cache nodes, the EC2 application instances are able to distribute cache keys across multiple ElastiCache nodes. The most common practice is to use client-side sharding to distribute keys across cache nodes, which we will discuss later in this paper.



When you launch an ElastiCache cluster, you can choose the Availability Zone(s) that the cluster lives in. For best performance, you should configure your cluster to use the same Availability Zones as your application servers. To launch an ElastiCache cluster in a specific Availability Zone, make sure to specify the **Preferred Zone(s)** option during cache cluster creation. The Availability Zones that you specify will be where ElastiCache will launch your cache nodes. We recommend that you select **Spread Nodes Across Zones**, which tells ElastiCache to distribute cache nodes across these zones as evenly as possible. This distribution will mitigate the impact of an Availability Zone disruption on your ElastiCache nodes. The trade-off is that some of the requests from your application to ElastiCache will go to a node in a different Availability Zone, meaning latency will be slightly higher. For more details, refer to Creating a Cache Cluster in the *Amazon ElastiCache User Guide*.

As mentioned at the outset, ElastiCache can be coupled with a wide variety of databases. Here is an example architecture that uses Amazon DynamoDB instead of Amazon RDS and MySQL:



:

This combination of DynamoDB and ElastiCache is very popular with mobile and game companies, because DynamoDB allows for higher write throughput at lower cost than traditional relational databases. In addition, DynamoDB uses a key-value access pattern similar to ElastiCache, which also simplifies the programming model. Instead of using relational SQL for the primary database but then key-value patterns for the cache, both the primary database and cache can be programmed similarly. In this architecture

pattern, DynamoDB remains the source of truth for data, but application reads are offloaded to ElastiCache for a speed boost.

Selecting the Right Cache Node Size

ElastiCache supports many different types of cache nodes. Because the newest node types support the latest-generation CPUs and networking capabilities, we recommend choosing a cache node from the M3 or R3 families. Of these, M3 has smaller sizes and R3 instances provide proportionately more RAM per CPU core, enhanced networking performance, and the cheapest price per GB of RAM. All of these features can save money as your nodes grow in memory size. As of the time of writing, ElastiCache supports cache nodes with over 200 GB of memory!

You can get a ballpark estimate of the amount of cache memory you'll need by multiplying the size of items you want to cache by the number of items you want to keep cached at once. Unfortunately, calculating the size of your cached items can be trickier than it sounds. You can arrive at a slight overestimate by serializing your cached items and then counting characters. Here's an example that flattens a Ruby object to JSON, counts the number of characters, and then multiplies by 2 because there are typically 2 bytes per character:

```
irb(main):010:0> user = User.find(4)
irb(main):011:0> user.to_json.size * 2
=> 580
```

In addition to the size of your data, Memcached adds approximately 50–60 bytes of internal bookkeeping data to each element. The cache key also consumes space, up to 250 characters at 2 bytes each. In this example, it's probably safest to overestimate a little and guess 1–2 KB per cached object. Keep in mind that this approach is just for illustration purposes. Your cached objects can be much larger if you are caching rendered page fragments or if you use a serialization library that expands strings.

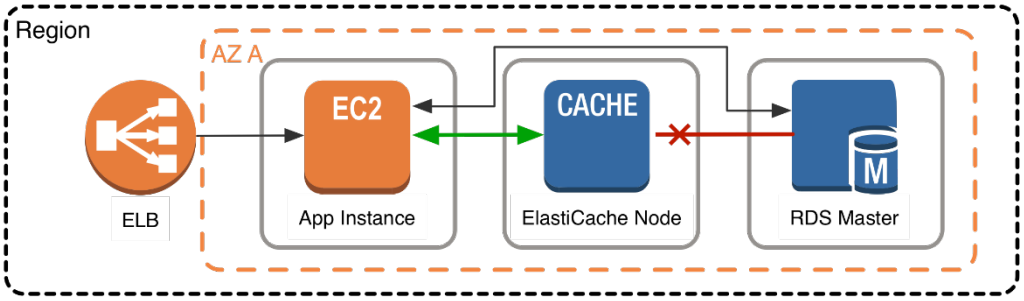
Because Amazon ElastiCache is a pay-as-you-go service, make your best guess at the node instance size, and then adjust after getting some real-world data. Make sure your application is set up for consistent hashing, which will enable you to add additional Memcached nodes to scale your in-memory layer horizontally. For additional tips, refer to Cache Node Considerations for Memcached and Cache Node Considerations for Redis in the *Amazon ElastiCache User Guide.*

Security Groups and VPC

Like other Amazon web services, ElastiCache supports security groups. You can use security groups to define rules that limit access to your instances based on IP address

and port. ElastiCache supports both subnet security groups in Amazon Virtual Private Cloud (Amazon VPC) and classic Amazon EC2 security groups. We strongly recommend you deploy ElastiCache and your application in Amazon VPC, unless you have a specific need otherwise (such as for an existing application). Amazon VPC offers several advantages, including fine-grained access rules and control over private IP addressing. For an overview of how ElastiCache integrates with Amazon VPC, refer to ElastiCache with VPC in the *Amazon ElastiCache User Guide.*

When launching your ElastiCache cluster in VPC, launch it in a private subnet with no public connectivity for best security. Neither Memcached nor Redis has any serious authentication or encryption capabilities. Following is a simplified version of our previous architecture diagram that includes an example VPC subnet design.



To keep your cache nodes as secure as possible, only allow access to your cache cluster from your application tier, as shown preceding. ElastiCache does not need connectivity to or from your database tier, because your database does not directly interact with ElastiCache. Only application instances that are making calls to your cache cluster need connectivity to it.

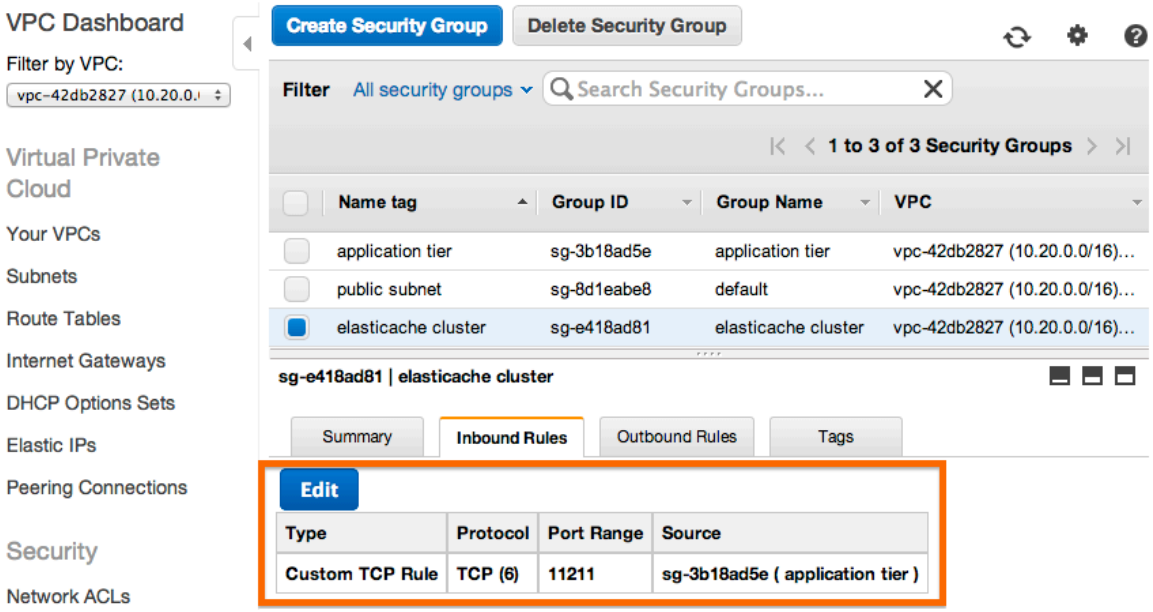
The way ElastiCache manages connectivity in Amazon VPC is through standard VPC subnets and security groups. To securely launch an ElastiCache cluster in Amazon VPC, follow these steps:

1. Create VPC private subnet(s) that will house your ElastiCache cluster, in the same VPC as the rest of your application. A given VPC subnet maps to a single Availability Zone. Given this mapping, create a private VPC subnet for each Availability Zone where you have application instances. Alternatively, you can reuse another private VPC subnet that you already have. For more information, refer to VPC Subnets in the *Amazon Virtual Private Cloud User Guide*.
2. Create a VPC security group for your new cache cluster. Make sure it is also in the same VPC as the preceding subnet. For more details, refer to VPC Security Groups in the *Amazon Virtual Private Cloud User Guide*.
3. Create a single access rule for this security group, allowing inbound access on port 11211 for Memcached or on port 6379 for Redis.

4. Create an ElastiCache subnet group that contains the VPC private subnet(s) you chose preceding. This subnet group is how ElastiCache knows which VPC subnets to use when launching the cluster. For instructions, see Creating a Cache Subnet Group in the *Amazon ElastiCache User Guide.*

5. When you launch your ElastiCache cluster, make sure to place it in the correct VPC, and choose the correct ElastiCache subnet group. For instructions, refer to Creating a Cache Cluster in VPC in the *Amazon ElastiCache User Guide.*

A correct VPC security group for your cache cluster should look like the following. Notice the single ingress rule allowing access to the cluster from the application tier:



To test connectivity from an application instance to your cache cluster in VPC, you can use netcat, a command-line Linux utility. Choose one of your cache cluster nodes, and attempt to connect to the node on either port 11211 (Memcached) or port 6379 (Redis):

```
$ nc -z -w5 my-cache-2b.z2vq55.001.usw2.cache.amazonaws.com
11211
$ echo $?
0
```

If the connection is successful, netcat will exit with status 0. If netcat appears to hang, or exits with a nonzero status, check your VPC security group and subnet settings.

Caching Design Patterns

With a ElastiCache cluster deployed, let's now dive into how to best apply caching in your application.

How to Apply Caching

Caching is applicable to a wide variety of use cases, but fully exploiting caching requires some planning. When you are deciding whether to cache a piece of data, consider the following questions:

- Is it safe to use a cached value? The same piece of data can have different consistency requirements in different contexts. For example, during online checkout, you need the authoritative price of an item, so caching might not be appropriate. On other pages, however, the price might be a few minutes out of date without a negative impact on users.
- Is caching effective for that data? Some applications generate access patterns that are not suitable for caching—for example, sweeping through the key space of a large dataset that is changing frequently. In this case, keeping the cache up to date could offset any advantage caching could offer.
- Is the data structured well for caching? Simply caching a database record can often be enough to offer significant performance advantages. However, other times, data is best cached in a format that combines multiple records together. Because caches are simple key-value stores, you might also need to cache a data record in multiple different formats, so you can access it by different attributes in the record.

You don’t need to make all of these decisions up front. As you expand your usage of caching, keep these guidelines in mind when deciding whether to cache a given piece of data.

Consistent Hashing (Sharding)

In order to make use of multiple ElastiCache nodes, you need a way to efficiently spread your cache keys across your cache nodes. The naïve approach to distributing cache keys, often found in blogs, looks like this:

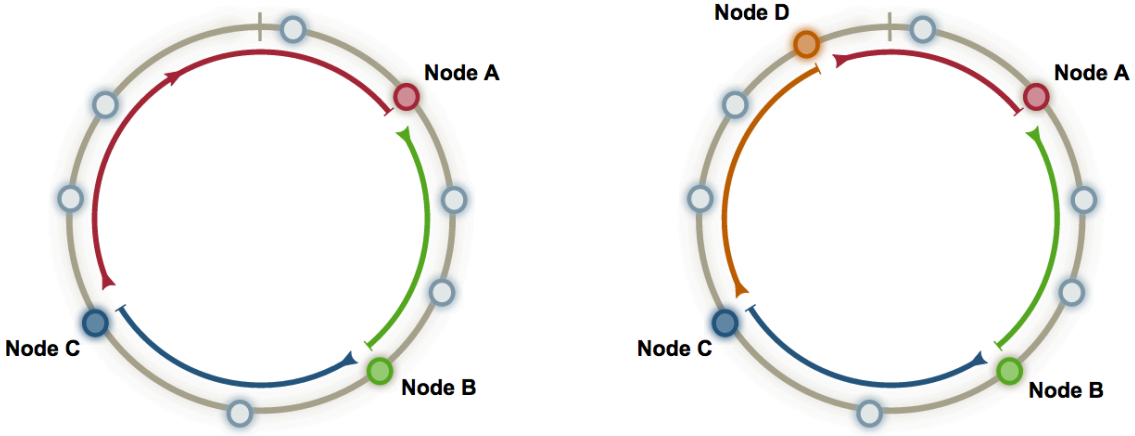
```
cache_node_list = [
  'my-cache-2a.z2vq55.0001.usw2.cache.amazonaws.com:11211',
  'my-cache-2a.z2vq55.0002.usw2.cache.amazonaws.com:11211'
]
cache_index = hash(key) % length(cache_node_list)
cache_node  = cache_node_list[cache_index]
```

This approach applies a hash function (such as CRC32) to the key to add some randomization, and then uses a math modulo of the number of cache nodes to distribute the key to a random node in the list. This approach is easy to understand, and most importantly for any key hashing scheme it is deterministic in that the same cache key always maps to the same cache node.

Unfortunately, this particular approach suffers from a fatal flaw due to the way that modulo works. As the number of cache nodes scales up, most hash keys will get remapped to new nodes with empty caches, as a side effect of using modulo. You can calculate the number of keys that would be remapped to a new cache node by dividing the old node count by the new node count. For example, scaling from 1 to 2 nodes remaps ½ your cache keys; scaling from 3 to 4 nodes remaps ¾ of your keys; and scaling from 9 to 10 nodes remaps 90 percent of your keys to empty caches. Ouch.

This approach is bad for obvious reasons. Think of the scenario where you're scaling rapidly due to a spike in demand. Just at the point when your application is getting overwhelmed, you add an additional cache node to help alleviate the load. Instead you effectively wipe 90 percent of your cache, causing a huge spike of requests to your database. Your dashboard goes red and you start getting those alerts that nobody wants to get.

Luckily, there is a well-understood solution to this dilemma, known as *consistent hashing*. The theory behind consistent hashing is to create an internal hash ring with a pre-allocated number of partitions that can hold hash keys. As cache nodes are added and removed, they are slotted into positions on that ring. The following illustration, taken from Benjamin Erb's thesis on Scalable Web Architectures, illustrates consistent hashing graphically.



The downside to consistent hashing is that there's quite a bit of math involved—at least, it's more complicated than a simple modulo. In a nutshell, you preallocate a set of random integers, and assign cache nodes to those random integers. Then, rather than

using modulo, you find the closest integer in the ring for a given cache key, and use the cache node associated with that integer. A concise yet complete explanation can be found in the article Consistent Hashing, by Tom White.

Luckily, many modern client libraries include consistent hashing. Although you shouldn't need to write your consistent hashing solution from scratch, it's still important you be aware of consistent hashing, so you can ensure it's enabled in your client. For many libraries, it's still not the default behavior, even when supported by the library.

Client Libraries

Mature Memcached client libraries exist for all popular programming languages. Any of the following Memcached libraries will work with Amazon ElastiCache:

Language	Library
Ruby	Dalli, Dalli::ElastiCache
Python	Memcache Ring, django-elasticache
Node.js	node-memcached
PHP	ElastiCache AutoDiscover Client
Java	ElastiCache AutoDiscover Client, spymemcached
C#/.NET	ElastiCache AutoDiscover Client, Enyim Memcached

For Memcached with Java, .NET, or PHP, we recommend using the Amazon ElastiCache client library, because it supports Auto Discovery of new ElastiCache nodes as they are added to the cache cluster. For Java, this library is a simple wrapper around the popular spymemcached library that adds Auto Discovery support. For PHP, it is a wrapper around the built-in Memcached PHP library. For .NET, it is a wrapper around Enyim Memcached.

Note that Auto Discovery works for only Memcached, not Redis. When ElastiCache repairs or replaces a cache node, the Domain Name Service (DNS) name of the cache node will remain the same, meaning your application doesn't need to use Auto Discovery to deal with common failures. You only need Auto Discovery support if you dynamically scale the size of your cache cluster on the fly, while your application is running. Dynamic scaling is only required if your application load fluctuates significantly. For more details, see Cluster Scaling and Auto Discovery.

As mentioned, you should choose a client library that includes native support for consistent hashing. Many of the libraries in the preceding table support consistent hashing, but we recommend you check the documentation, because this support can change over time. Also, you might need to enable consistent hashing by setting an option in the client library.

In PHP, for example, you need to explicitly set **Memcached::OPT_LIBKETAMA_COMPATIBLE** to **true** to enable consistent hashing:

```
$cache_nodes = array(
    array('my-cache-
2a.z2vq55.0001.usw2.cache.amazonaws.com', 11211),
    array('my-cache-
2a.z2vq55.0002.usw2.cache.amazonaws.com', 11211)
);
$memcached = new Memcached();
$memcached->setOption(Memcached::OPT_LIBKETAMA_COMPATIBLE,
true);
$memcached->addServers($cache_nodes);
```

This code snippet tells PHP to use consistent hashing by using libketama. Otherwise, the default in PHP is to use modulo, which suffers from the drawbacks outlined preceding.

Next, let's look at some common and effective caching strategies. If you've done a good amount of caching before, some of this might be old hat.

Be Lazy

Lazy caching, also called lazy population or cache-aside, is the most prevalent form of caching. Laziness should serve as the foundation of any good caching strategy. The basic idea is to populate the cache only when an object is actually requested by the application. The overall application flow goes like this:

1. Your app receives a query for data, for example the top 10 most recent news stories.
2. Your app checks the cache to see if the object is in cache.
3. If so (a *cache hit),* the cached object is returned, and the call flow ends.
4. If not (a *cache miss),* then the database is queried for the object. The cache is populated, and the object is returned.

This approach has several advantages over other methods:

- The cache only contains objects that the application actually requests, which helps keep the cache size manageable. New objects are only added to the cache as needed. You can then manage your cache memory passively, by simply letting Memcached automatically evict (delete) the least-accessed keys as your cache fills up, which it does by default.
- As new cache nodes come online, for example as your application scales up, the lazy population method will automatically add objects to the new cache nodes when the application first requests them.
- Cache expiration, which we will cover in depth later, is easily handled by simply deleting the cached object. A new object will be fetched from the database the next time it is requested.
- Lazy caching is widely understood, and many web and app frameworks include support out of the box.

Here is an example of lazy caching in Python pseudocode:

```
# Python
def get_user(user_id):
    # Check the cache
    record = cache.get(user_id)
    if record is None:
       # Run a DB query
       record = db.query("select * from users where id =
?",user_id)
       # Populate the cache
       cache.set(user_id, record)
    return record

# App code
user = get_user(17)
```

You can find libraries in many popular programming frameworks that encapsulate this pattern. But regardless of programming language, the overall approach is the same.

You should apply a lazy caching strategy anywhere in your app where you have data that is going to be read often, but written infrequently. In a typical web or mobile app, for example, a user's profile rarely changes, but is accessed throughout the app. A person might only update his or her profile a few times a year, but the profile might be accessed dozens or hundreds of times a day, depending on the user. Because Memcached will automatically evict the less frequently used cache keys to free up memory, you can apply lazy caching liberally with little downside.

Write On Through

In a write-through cache, the cache is updated in real time when the database is updated. So, if a user updates his or her profile, the updated profile is also pushed into the cache. You can think of this as being proactive to avoid unnecessary cache misses, in the case that you have data that you absolutely know is going to be accessed. A good example is any type of aggregate, such as a top 100 game leaderboard, or the top 10 most popular news stories, or even recommendations. Because this data is typically updated by a specific piece of application or background job code, it's straightforward to update the cache as well.

The write-through pattern is also easy to demonstrate in pseudocode:

```
# Python
def save_user(user_id, values):
    # Save to DB
    record = db.query("update users ... where id = ?",
                      user_id, values)
    # Push into cache
    cache.set(user_id, record)
    return record

# App code
user = save_user(17, {"name": "Nate Dogg"})
```

This approach has certain advantages over lazy population:

- It avoids cache misses, which can help the application perform better and feel snappier.
- It shifts any application delay to the user updating data, which maps better to user expectations. By contrast, a series of cache misses can give a random user the impression that your app is just slow.
- It simplifies cache expiration. The cache is always up-to-date.

However, write-through caching also has some disadvantages:

- The cache can be filled with unnecessary objects that aren't actually being accessed. Not only could this consume extra memory, but unused items can evict more useful items out of the cache.
- It can result in lots of cache churn if certain records are updated repeatedly.
- When (not if) cache nodes fail, those objects will no longer be in the cache. You need some way to repopulate the cache of missing objects, for example by lazy population.

As might be obvious, you can combine lazy caching with write-through caching to help address these issues, because they are associated with opposite sides of the data flow. Lazy caching catches cache misses on reads, and write-through caching populates data on writes, so the two approaches complement each other. For this reason, it's often best to think of lazy caching as a foundation that you can use throughout your app, and write-through caching as a targeted optimization that you apply to specific situations.

Expiration Date

Cache expiration can get really complex really quickly. In our previous examples, we were only operating on a single user record. In a real app, a given page or screen often caches a whole bunch of different stuff at once—profile data, top news stories, recommendations, comments, and so forth, all of which are being updated by different methods.

Unfortunately, there is no silver bullet for this problem, and cache expiration is a whole arm of computer science. But there are a few simple strategies that you can use:

- Always apply a time to live (TTL) to all of your cache keys, except those you are updating by write-through caching. You can use a long time, say hours or even days. This approach catches application bugs, where you forget to update or delete a given cache key when updating the underlying record. Eventually, the cache key will auto-expire and get refreshed.
- For rapidly changing data such as comments, leaderboards, or activity streams, rather than adding write-through caching or complex expiration logic, just set a short TTL of a few seconds. If you have a database query that is getting hammered in production, it's just a few lines of code to add a cache key with a 5 second TTL around the query. This code can be a wonderful Band-Aid to keep your application up and running while you evaluate more elegant solutions.
- A newer pattern, *Russian doll caching*, has come out of work done by the Ruby on Rails team. In this pattern, nested records are managed with their own cache keys, and then the top-level resource is a collection of those cache keys. Say you have a news webpage that contains users, stories, and comments. In this approach, each of those is its own cache key, and the page queries each of those keys respectively.
- When in doubt, just delete a cache key if you're not sure whether it's affected by a given database update or not. Your lazy caching foundation will refresh the key when needed. In the meantime, your database will be no worse off than it was without Memcached.

For a good overview of cache expiration and Russian doll caching, refer to The performance impact of "Russian doll" caching, a post in the Basecamp *Signal vs Noise* blog.

The Thundering Herd

Also known as *dog piling,* the thundering herd effect is what happens when many different application processes simultaneously request a cache key, get a cache miss, and then each hits the same database query in parallel. The more expensive this query is, the bigger impact it has on the database. If the query involved is a top 10 query that requires ranking a large dataset, the impact can be a significant hit.

One problem with adding TTLs to all of your cache keys is that it can exacerbate this problem. For example, let's say millions of people are following a popular user on your site. That user hasn't updated his profile or published any new messages, yet his profile cache still expires due to a TTL. Your database might suddenly be swamped with a series of identical queries.

TTLs aside, this effect is also common when adding a new cache node, because the new cache node's memory is empty. In both cases, the solution is to prewarm the cache by following these steps:

1. Write a script that performs the same requests that your application will. If it's a web app, this script can be a shell script that hits a set of URLs.
2. If your app is set up for lazy caching, cache misses will result in cache keys being populated, and the new cache node will fill up.
3. When you add new cache nodes, run your script before you attach the new node to your application. Because your application needs to be reconfigured to add a new node to the consistent hashing ring, insert this script as a step before triggering the app reconfiguration.
4. If you anticipate adding and removing cache nodes on a regular basis, prewarming can be automated by triggering the script to run whenever your app receives a cluster reconfiguration event through Amazon Simple Notification Service (Amazon SNS).

Finally, there is one last subtle side effect of using TTLs everywhere. If you use the same TTL length (say 60 minutes) consistently, then many of your cache keys might expire within the same time window, even after prewarming your cache. One strategy that's easy to implement is to add some randomness to your TTL:

```
ttl = 3600 + (rand() * 120)  /* +/- 2 minutes */
```

The good news is that only sites at large scale typically have to worry about this level of scaling problem. It's good to be aware of, but it's also a good problem to have.

Cache (Almost) Everything

Finally, it might seem as if you should only cache your heavily hit database queries and expensive calculations, but that other parts of your app might not benefit from caching. In practice, in-memory caching is widely useful, because it is much faster to retrieve a flat cache key from memory than to perform even the most highly optimized database query or remote API call. Just keep in mind that cached data is stale data by definition, meaning there may be cases where it's not appropriate, such as accessing an item's price during online checkout. You can monitor statistics like cache misses to see whether your cache is effective, which we will cover in Monitoring and Tuning later in the paper.

ElastiCache for Redis

So far, we've been talking about ElastiCache for Memcached as a passive component in our application—a big slab of memory in the cloud. Choosing Redis as our engine can unlock more interesting possibilities for our application, due to its higher-level data structures such as lists, hashes, sets, and sorted sets.

Deploying Redis makes use of familiar concepts such as clusters and nodes. However, Redis has a few important differences compared with Memcached:

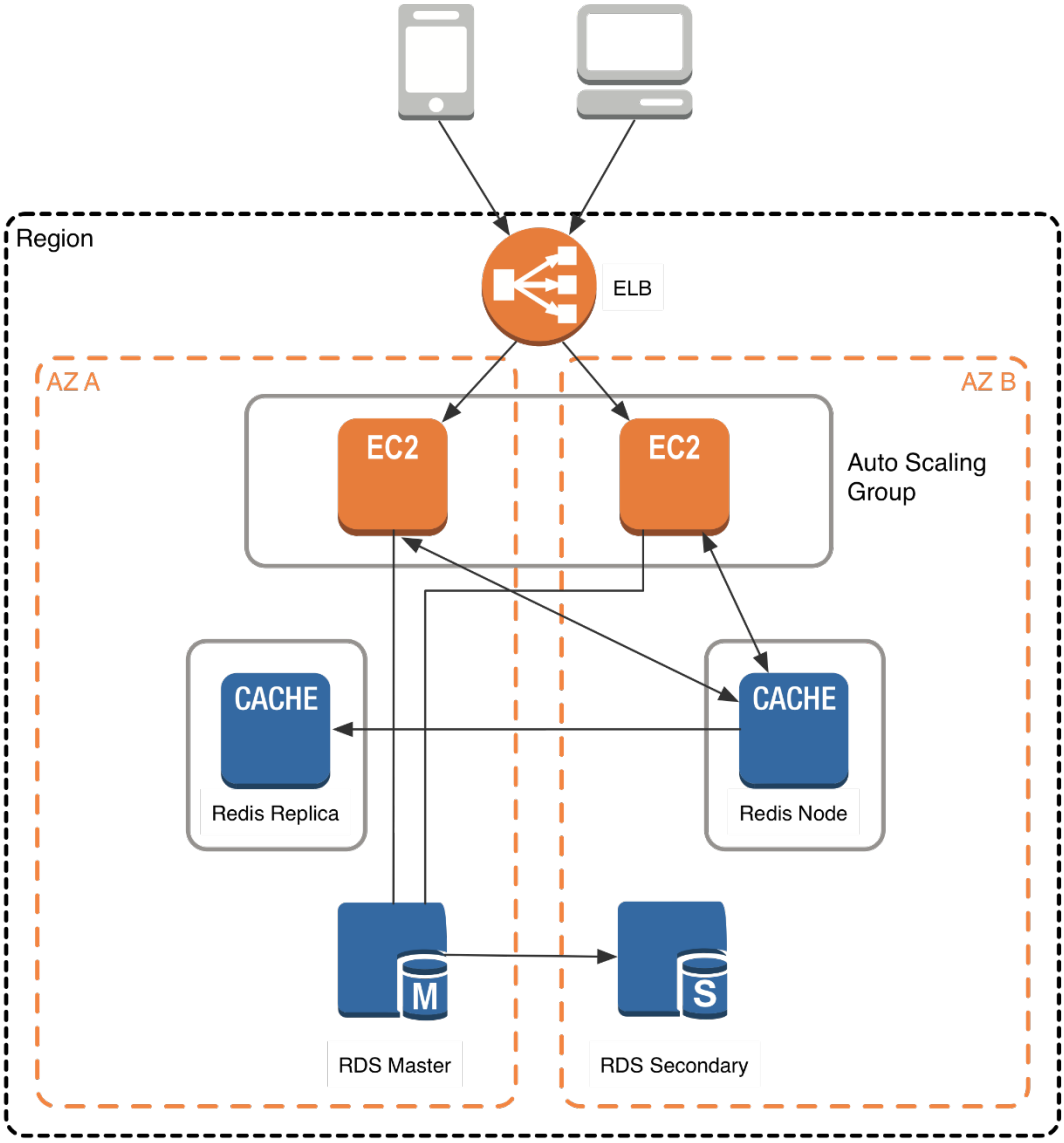
- Redis data structures cannot be horizontally sharded. As a result, Redis ElastiCache clusters are always a single node, rather than the multiple nodes we saw with Memcached.
- Redis supports replication, both for high availability and to separate read workloads from write workloads. A given ElastiCache for Redis primary node can have one or more replica nodes. A Redis primary node can handle both reads and writes from the app. Redis replica nodes can only handle reads, similar to Amazon RDS Read Replicas.
- Because Redis supports replication, you can also fail over from the primary node to a replica in the event of failure. You can configure ElastiCache for Redis to automatically fail over by using the Multi-AZ feature.
- Redis supports persistence, including backup and recovery. However, because Redis replication is asynchronous, you cannot completely guard against data loss in the event of a failure. We will go into detail on this topic in our discussion of Multi-AZ.

Architecture with ElastiCache for Redis

As with Memcached, when you deploy an ElastiCache for Redis cluster, it is an additional tier in your app. Unlike Memcached, ElastiCache clusters for Redis only contain a single primary node. After you create the primary node, you can configure one or more replica nodes and attach them to the primary Redis node. An ElastiCache for

Redis replication group consists of a primary and up to five read replicas. Redis asynchronously replicates the data from the primary to the read replicas.

Because Redis supports persistence, it is technically possible to use Redis as your only data store. In practice, customers find that a managed database such as Amazon DynamoDB or Amazon RDS is a better fit for most use cases of long-term data storage.

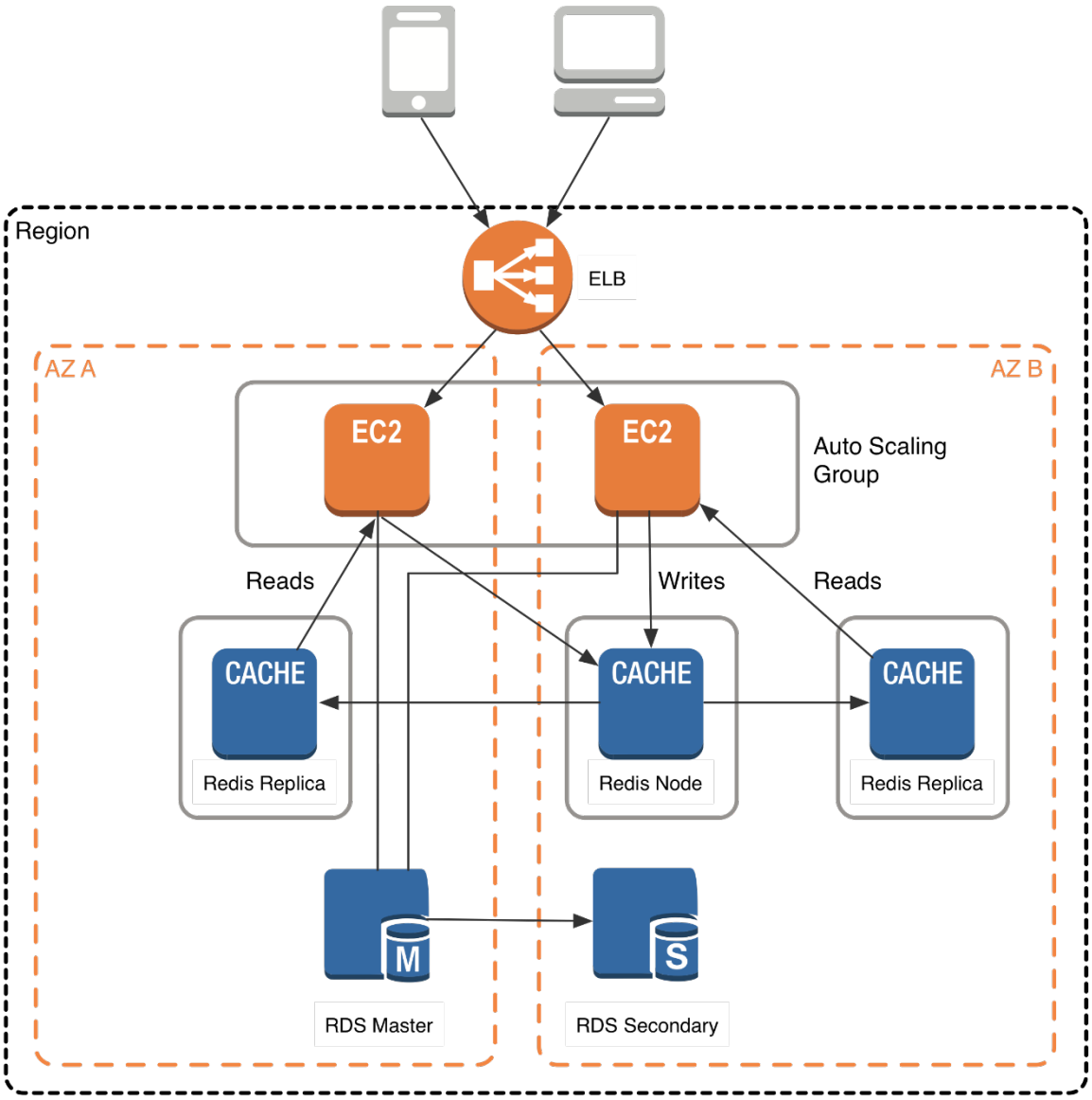


ElastiCache for Redis has the concept of a *primary endpoint,* which is a DNS name that always points to the current Redis primary node. If a failover event occurs, the DNS entry will be updated to point to the new Redis primary node. To take advantage of this functionality, make sure to configure your Redis client so that it uses the primary endpoint DNS name to access your Redis cluster.

Keep in mind that the number of Redis replicas you attach will affect the performance of the primary node. Resist the urge to spin up lots of replicas just for durability. One or two replicas in a different Availability Zone are sufficient for availability. When scaling read throughput, monitor your application's performance and add replicas as needed. Be sure to monitor your ElastiCache cluster's performance as you add replica nodes. For more details, refer to Monitoring and Tuning.

Distributing Reads and Writes

Using read replicas with Redis, you can separate your read and write workloads. This separation lets you scale reads by adding additional replicas as your application grows. In this pattern, you configure your application to send writes to the primary endpoint. Then you read from one of the replicas, as shown in the following diagram. With this approach, you can scale your read and write loads independently, so your primary node only has to deal with writes.



The main caveat to this approach is that reads can return data that is slightly out of date compared to the primary node, because Redis replication is asynchronous. For example, if you have a global counter of "total games played" that is being continuously incremented (a good fit for Redis), your master might show 51,782. However, a read from a replica might only return 51,775. In many cases, this is just fine. But if the counter is a basis for a crucial application state, such as the number of seconds remaining to vote on the most popular pop singer, this approach won't work.

When deciding whether data can be read from a replica, here are a few questions to consider:

- Is the value being used only for display purposes? If so, being slightly out of date is probably okay.

- Is the value a cached value, for example a page fragment? If so, again being slightly out of date is likely fine.
- Is the value being used on a screen where the user might have just edited it? In this case, showing an old value might look like an application bug.
- Is the value being used for application logic? If so, using an old value can be risky.
- Are multiple processes using the value simultaneously, such as a lock or queue? If so, the value needs to be up-to-date and needs to be read from the primary node.

In order to split reads and writes, you will need to create two separate Redis connection handles in your application: one pointing to the primary node, and one pointing to the read replica(s). Configure your application to write to the DNS primary endpoint, and then read from the other Redis nodes.

Multi-AZ with Auto-Failover

During certain types of planned maintenance, or in the unlikely event of ElastiCache node failure or Availability Zone failure, Amazon ElastiCache can be configured to automatically detect the failure of the primary node, select a read replica, and promote it to become the new primary. ElastiCache auto-failover will then update the DNS primary endpoint with the IP address of the promoted read replica. If your application is writing to the primary node endpoint as recommended earlier, no application change will be needed.

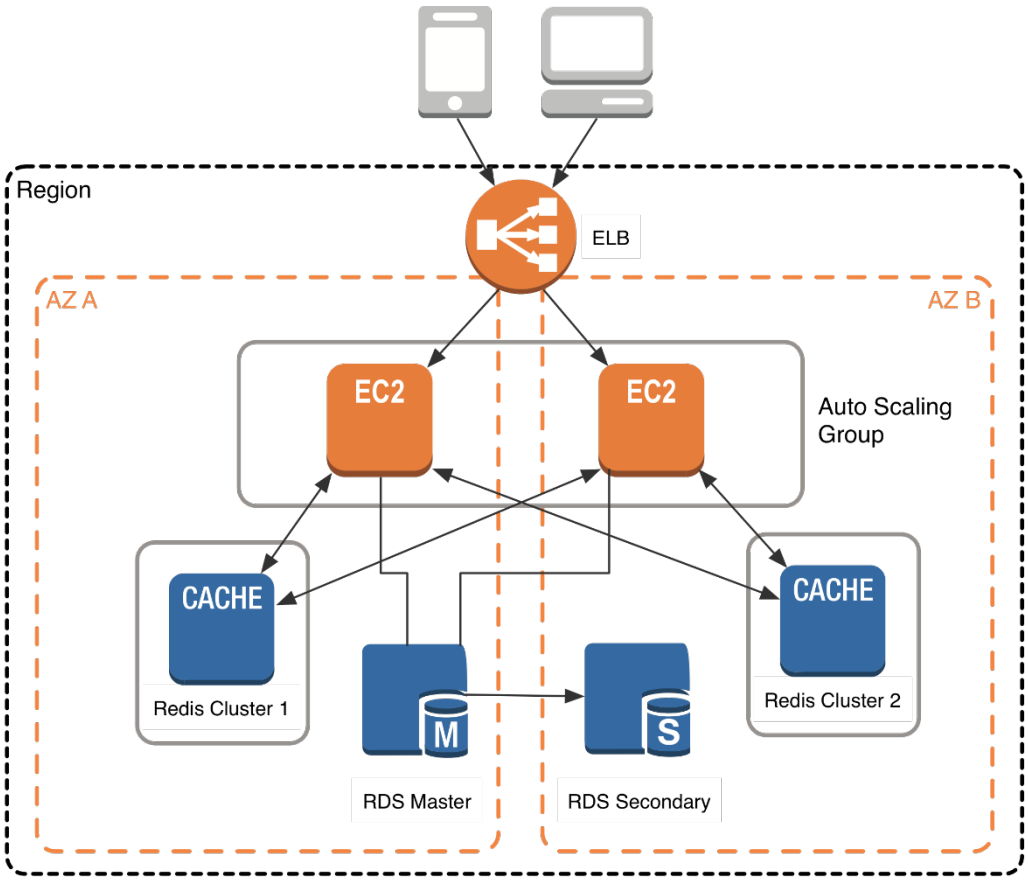
Depending on how in-sync the promoted read replica is with the primary node, the failover process can take several minutes. First, ElastiCache needs to detect the failover, then suspend writes to the primary node, and finally complete the failover to the replica. During this time, your application cannot write to the Redis ElastiCache cluster. Architecting your application to limit the impact of these types of failover events will ensure greater overall availability.

Unless you have a specific need otherwise, all production deployments should use Multi-AZ with auto-failover. Keep in mind that Redis replication is asynchronous, meaning if a failover occurs, the read replica that is selected might be slightly behind the master. Bottom line: Some data loss might occur if you have rapidly changing data. This effect is currently a limitation of Redis replication itself. If you have crucial data that cannot be lost (for example, transactional or purchase data), we recommend you also store that in a durable database such as Amazon DynamoDB or Amazon RDS.

Sharding with Redis

Redis has two categories of data structures: simple keys and counters, and multidimensional sets, lists, and hashes. The bad news is the second category cannot be sharded horizontally. But the good news is that simple keys and counters can.

In the simplest case, you can treat a single Redis node just like a single Memcached node. Just like you might spin up multiple Memcached nodes, you can spin up multiple Redis clusters, and each Redis cluster is responsible for part of the sharded dataset.



In your application, you'll then need to configure the Redis client to shard between those two clusters. Here is an example from the Jedis Sharded Java Client:

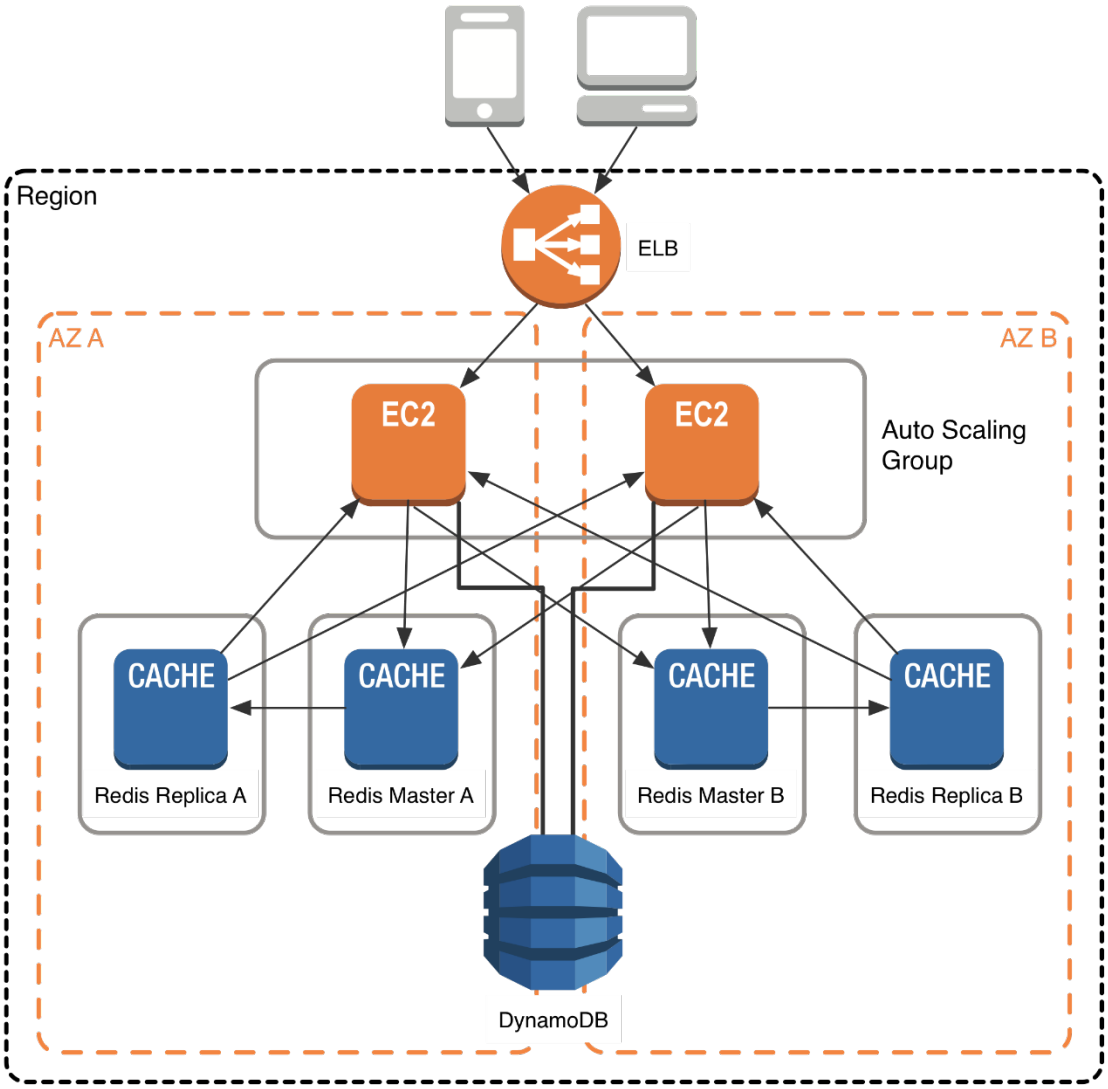
```
List<JedisShardInfo> shards = new
ArrayList<JedisShardInfo>();

shards.add(new JedisShardInfo("redis-cluster1", 6379));
shards.add(new JedisShardInfo("redis-cluster2", 6379));

ShardedJedisPool pool = new ShardedJedisPool(shards);
ShardedJedis jedis = pool.getResource();
```

You can also combine horizontal sharding with split reads and writes. In this setup, you have two or more Redis clusters, each of which stores part of the keyspace. You configure your application with two separate sets of Redis handles, a write handle that

points to the sharded masters and a read handle that points to the sharded replicas. Following is an example architecture, this time with Amazon DynamoDB rather than MySQL, just to illustrate that you can use either one:



For the purpose of simplification, the preceding diagram shows replicas in the same Availability Zone as the primary node. In practice, you should place the replicas in a different Availability Zone. From an application perspective, continuing with our Java example, you configure two Redis connection pools as follows:

```
List<JedisShardInfo> masters = new
ArrayList<JedisShardInfo>();
masters.add(new JedisShardInfo("redis-masterA", 6379));
masters.add(new JedisShardInfo("redis-masterB", 6379));
```

```
ShardedJedisPool write_pool = new
ShardedJedisPool(masters);
```

```
ShardedJedis write_jedis = write_pool.getResource();

List<JedisShardInfo> replicas = new
ArrayList<JedisShardInfo>();
replicas.add(new JedisShardInfo("redis-replicaA", 6379));
replicas.add(new JedisShardInfo("redis-replicaB", 6379));

ShardedJedisPool read_pool = new
ShardedJedisPool(replicas);
ShardedJedis read_jedis = read_pool.getResource();
```

In designing your application, you need to make decisions as to whether a given value can be read from the replica pool, which might be slightly out of date, or from the primary write node. Be aware that reading from the primary node will ultimately limit the throughput of your entire Redis layer, because it takes I/O away from writes.

Using multiple clusters in this fashion is the most advanced configuration of Redis possible. In practice, it is overkill for most applications. However, if you design your application so that it can leverage a split read/write Redis layer, you can apply this design in the future, if your application grows to the scale where it is needed.

Advanced Datasets with Redis

Let's briefly look at some use cases that ElastiCache for Redis can support.

Game Leaderboards

If you've played online games, you're probably familiar with top 10 leaderboards. What might not be obvious is that calculating a top *n* leaderboard in near real time is actually quite complex. An online game can easily have thousands of people playing concurrently, each with stats that are changing continuously. Re-sorting these users and reassigning a numeric position is very computationally expensive.

Sorted sets are particularly interesting here, because they simultaneously guarantee both the uniqueness and ordering of elements. Redis sorted set commands all start with Z. When an element is inserted in a Redis sorted set, it is reranked in real time and assigned a numeric position. Here is a complete game leaderboard example in Redis:

```
ZADD "leaderboard" 556  "Andy"
```

```
ZADD "leaderboard" 819  "Barry"
ZADD "leaderboard" 105  "Carl"
ZADD "leaderboard" 1312 "Derek"

ZREVRANGE "leaderboard" 0 -1
1) "Derek"
2) "Barry"
3) "Andy"
4) "Carl"

ZREVRANK "leaderboard" "Barry"
2
```

When a player's score is updated, the Redis command ZADD overwrites the existing value with the new score. The list is instantly re-sorted, and the player receives a new rank. For more information, refer to the Redis documentation on ZADD, ZRANGE, and ZRANK.

Recommendation Engines

Similarly, calculating recommendations for users based on other items they've liked requires very fast access to a large dataset. Some algorithms, such as Slope One, are simple and effective but require in-memory access to every item ever rated by anyone in the system. Even if this data is kept in a relational database, it has to be loaded in memory somewhere to run the algorithm.

Redis data structures are a great fit for recommendation data. You can use Redis counters used to increment or decrement the number of likes or dislikes for a given item. You can use Redis hashes to maintain a list of everyone who has liked or disliked that item, which is the type of data that Slope One requires. Here is a brief example of storing item likes and dislikes:

```
INCR "item:38923:likes"
HSET "item:38923:ratings" "Susan" 1
INCR "item:38923:dislikes"
HSET "item:38923:ratings" "Tommy" -1
```

From this simple data, not only can we use Slope One or Jaccardian similarity to recommend similar items, but we can use the same counters to display likes and dislikes in the app itself. In fact, a number of open source projects use Redis in exactly this manner, such as Recommendify and Recommendable. In addition, because Redis supports persistence, this data can live solely in Redis. This placement eliminates the

need for any data loading process, and also offloads an intensive process from your main database.

Chat and Messaging

Redis provides a lightweight pub/sub mechanism that is well-suited to simple chat and messaging needs. Use cases include in-app messaging, web chat windows, online game invites and chat, and real-time comment streams (such as you might see during a live streaming event). Two basic Redis commands are involved, PUBLISH and SUBSCRIBE:

```
SUBSCRIBE "chat:114"
PUBLISH "chat:114" "Hello all"
   ["message", "chat:114", "Hello all"]
UNSUBSCRIBE "chat:114"
```

Note that unlike other Redis data structures, pub/sub messaging doesn't get persisted to disk. Redis pub/sub messages are not written as part of the RDB or AOF backup files that Redis creates. If you want to save these pub/sub messages, you will need to add them to a Redis data structure, such as a list. For more details, refer to Using Pub/Sub for Asynchronous Communication in the *Redis Cookbook.*

Also, because Redis pub/sub is not persistent, you can lose data if a cache node fails. If you're looking for a reliable topic-based messaging system, consider evaluating Amazon SNS.

Queues

Although we offer a managed queue service in the form of Amazon Simple Queue Service (Amazon SQS) and we encourage customers to use it, you can also use Redis data structures to build queuing solutions. The Redis documentation for RPOPLPUSH covers two well-documented queuing patterns. In these patterns, Redis lists are used to hold items in a queue. When a process takes an item from the queue to work on it, the item is pushed onto an "in-progress" queue, and then deleted when the work is done. Open source solutions such as Resque use Redis as a queue; GitHub uses Resque.

Redis does have certain advantages over other queue options, such as very fast speed, once and only once delivery, and guaranteed message ordering. However, pay careful attention to ElastiCache for Redis backup and recovery options (which we will cover shortly) if you intend to use Redis as a queue. If a Redis node terminates and you have not properly configured its persistence options, you can lose the data for the items in your queue. Essentially, you need to view your queue as a type of database, and treat it appropriately, rather than as a disposable cache.

Client Libraries and Consistent Hashing

As with Memcached, you can find Redis client libraries for the currently popular programming languages. Any of these will work with ElastiCache for Redis:

Language	Redis
Ruby	Redis-rb, Redis Objects
Python	Redis-py
Node.js	node_redis
PHP	phpredis
Java	Jedis
C#/.Net	ServiceStack.Redis

Unlike with Memcached, it is *un*common for Redis libraries to support consistent hashing. Redis libraries rarely support consistent hashing because the advanced data types that we discussed preceding cannot simply be horizontally sharded across multiple Redis nodes. This point leads to another, very important one: Redis as a technology cannot be horizontally scaled easily. Redis can only scale up to a larger node size, because its data structures must reside in a single memory image in order to perform properly.

Note that Redis Cluster was just released in April 2015 in Redis version 3.0. It aims to provide scale-out capability with certain data types. The stability and robustness of this version remains to be seen over the next months. Redis Cluster currently only supports a subset of Redis functionality, and has some important caveats about possible data loss. For more details, refer to the Redis Cluster specification.

Monitoring and Tuning

Before we wrap up, let's spend some time talking about monitoring and performance tuning.

Monitoring Cache Efficiency

To start with, refer to the Amazon ElastiCache documentation for CloudWatch Metrics with ElastiCache and Which Metrics Should I Monitor? Both sections are excellent

resources for understanding how to measure the health of your ElastiCache cluster, by using metrics that ElastiCache publishes to Amazon CloudWatch. Most importantly, watch CPU usage. A consistently high CPU usage indicates a node is overtaxed, either by too many concurrent requests, or by performing dataset operations in the case of Redis.

For Redis, because the engine is single-threaded, you will need to multiply the CPU percentage by the number of cores to get an accurate measure of CPU usage. Once Redis maxes out a single CPU core, that instance is fully utilized, and a larger instance is needed. Suppose you're using an EC2 instance with four cores and the CPU is at 25 percent. This situation actually means that the instance is maxed out, because Redis is essentially pegging one CPU at 100 percent.

In addition to CPU, here is some additional guidance for monitoring cache memory utilization. Each of these metrics is available in CloudWatch for your ElastiCache cluster:

- **Evictions**—both Memcached and Redis manage cache memory internally, and when memory starts to fill up they evict (delete) unused cache keys to free space. A small number of evictions shouldn't alarm you, but a large number means your cache is running out of space.
- **CacheMisses**—the number of times a key was requested but not found in the cache. This number can be fairly large if you're using lazy population as your main strategy. If this number is remaining steady, it's likely nothing to worry about. However, a large cache miss number combined with a large eviction number can indicate your cache is thrashing due to lack of memory.
- **BytesUsedForCacheItems**—this value is the actual amount of cache memory that Memcached or Redis is using. Both Memcached and Redis, like all software, will attempt to allocate as much system memory as possible, even if it's not used by actual cache keys. Thus, monitoring the system memory usage on a cache node doesn't tell you how full your cache actually is.
- **SwapUsage**—in normal usage, neither Memcached nor Redis should be performing swaps.

A well-tuned cache node will show cache bytes used almost equal to the "max memory" parameter. In steady state, most cache counters will increase, with cache hits increasing faster than misses. You will probably also see a low number of evictions. However, a rising number of evictions indicates cache keys are getting pushed out of memory, which means you can benefit from larger cache nodes with more memory.

The one exception to the evictions rule is if you follow a strict definition of Russian doll caching, which says you should never cause cache items to expire, but instead let Memcached and Redis evict unused keys as needed. If you follow this approach, keep a closer eye on cache misses and bytes used to detect potential problems.

Watching For Hot Spots

If you are using consistent hashing to distribute cache keys across your cache nodes, in general your access patterns should be fairly even across nodes. However, you still need to watch out for *hot spots*, which are nodes in your cache that receive higher load than other nodes. This pattern is caused by *hot keys*, which are cache keys that are accessed more frequently than others. Think of a social website, where you have some users that might be 10,000 times more popular than an average user. That user's cache keys will be accessed much more often, which can put an uneven load onto the cache nodes that house that user's keys.

If you see uneven CPU usage among your cache nodes, you might have a hot spot. This pattern often appears as one cache node having a significantly higher operation count than other nodes. One way to confirm this is by keeping a counter in your application of your cache key gets and puts. You can push these as custom metrics into CloudWatch, or another monitoring service. Don't do this unless you suspect a hot spot, however, because logging every key access will decrease the overall performance of your application.

In the most common case, a few hot keys will not necessarily create any significant hot spot issues. If you have a few hot keys on each of your cache nodes, then those hot keys are themselves evenly distributed, and are producing an even load on your cache nodes. If you have three cache nodes and each of them has a few hot keys, then you can continue sizing your cache cluster as if those hot keys did not exist. In practice, even a well-designed application will have some degree of unevenness in cache key access.

In extreme cases, a single hot cache key can create a hot spot that overwhelms a single cache node. In this case, having good metrics about your cache, especially your most popular cache keys, is crucial to designing a solution. One solution is to create a mapping table that remaps very hot keys to a separate set of cache nodes. Although this approach provides a quick fix, you will still face the challenge of scaling those new cache nodes. Another solution is to add a secondary layer of smaller caches in front of your main nodes, to act as a buffer. This approach gives you more flexibility, but introduces additional latency into your caching tier.

The good news is that these concerns only hit applications of a significant scale. We recommend being aware of this potential issue and monitoring for it, but not spending time trying to engineer around it up front. Hot spots are a fast-moving area of computer science research, and there is no one-size-fits-all solution. As always, our team of Solutions Architects is available to work with you to address these issues if you encounter them. For more research on this topic, refer to papers such as Relieving Hot Spots on the World Wide Web and Characterizing Load Imbalance in Real-World Networked Caches.

Memcached Memory Optimization

Memcached uses a slab allocator, which means that it allocates memory in fixed chunks, and then manages those chunks internally. Using this approach, Memcached can be more efficient and predictable in its memory access patterns than if it used the system **malloc()**. The downside of the Memcached slab allocator is that memory chunks are rigidly allocated once and cannot be changed later. This approach means that if you choose the wrong number of the wrong size slabs, you might run out of Memcached chunks while still having plenty of system memory available.

When you launch an ElastiCache cluster, the **max_cache_memory** parameter is set for you automatically, along with several other parameters (for a list of default values, refer to Parameters for Memcached section in the *Amazon ElastiCache User Guide).* The key parameters to keep in mind are **chunk_size** and **chunk_size_growth_factor**, which together control how memory chunks are allocated.

Redis Memory Optimization

Redis has a good write-up on memory optimization that can come in handy for advanced use cases. Redis exposes a number of Redis configuration variables that will affect how Redis balances CPU and memory for a given dataset. These directives can be used with ElastiCache for Redis as well.

Redis Backup and Restore

Redis clusters support persistence by using backup and restore. When Redis backup and restore is enabled, ElastiCache can automatically take snapshots of your Redis cluster and save them to Amazon Simple Storage Service (Amazon S3). The ElastiCache documentation includes very good coverage of this functionality in the topic Managing Backup and Restore.

Because of the way Redis backups are implemented in the Redis engine itself, you need to have more memory available that your dataset consumes. This requirement is because Redis forks a background process that writes the backup data. To do so, it makes a copy of your data, using Linux copy-on-write semantics. If your data is changing rapidly, this approach means those data segments will be copied, consuming additional memory. For more details, refer to Amazon ElastiCache Backup Best Practices.

For production use, we strongly recommend that you always enable Redis backups, and retain them for a minimum of 7 days. In practice, retaining them for 14 or 30 days will provide better safety in the event of an application bug that ends up corrupting data. Even if you plan to use Redis primarily as a performance optimization or caching layer, persisting the data means you can prewarm a new Redis node, which avoids the thundering herd issue that we discussed earlier. To create a new Redis cluster from a

backup snapshot, refer to Restoring a Snapshot to a New Cluster in the *Amazon ElastiCache User Guide.*

You can also use a Redis snapshot to scale up to a larger Amazon EC2 instance type. To do so, follow this process:

1. Suspend writes to your existing ElastiCache cluster. Your application can continue to do reads.
2. Take a snapshot by following the procedure in the Creating a Manual Snapshot section in the *Amazon ElastiCache User Guide.* Give it a distinctive name that you will remember.
3. Create a new ElastiCache Redis cluster, and specify the snapshot you took preceding to seed it.
4. Once the new ElastiCache cluster is online, reconfigure your application to start writing to the new cluster.

Currently, this process will interrupt your application's ability to write data into Redis. If you have writes that are only going into Redis and that cannot be suspended, you can put those into Amazon SQS while you are resizing your ElastiCache cluster. Then, once your new ElastiCache Redis cluster is ready, you can run a script that pulls those records off Amazon SQS and writes them to your new Redis cluster.

Cluster Scaling and Auto Discovery

Scaling your application in response to changes in demand is one of the key benefits of working with AWS. Many customers find that configuring their client with a list of node DNS endpoints for ElastiCache works perfectly fine. But let's look at how to scale your ElastiCache Memcached cluster while your application is running, and how to set up your application to detect changes to your cache layer dynamically.

Auto Scaling Cluster Nodes

Amazon ElastiCache does not currently support using Auto Scaling to scale the number of cache nodes in a cluster. To change the number of cache nodes, you can use either the AWS Management Console or the AWS API to modify the cluster. For more information, refer to Managing Cache Cluster Nodes in the *Amazon ElastiCache User Guide.*

In practice, you usually don't want to regularly change the number of cache nodes in your Memcached cluster. Any change to your cache nodes will result in some percentage of cache keys being remapped to new (empty) nodes, which means a performance impact to your application. Even with consistent hashing, you will see an impact on your application when adding or removing nodes.

Auto Discovery of Memcached Nodes

The Amazon ElastiCache client libraries for Java, .NET, and PHP support Auto Discovery of new ElastiCache Memcached nodes. For Ruby, the open source library dalli-elasticache provides autodiscovery support, and django-elasticache is available for Python Django. In other languages, you'll need to implement autodiscovery yourself. Luckily, this implementation is very easy.

The overall Auto Discovery mechanism is outlined in the How Auto Discovery Works section in the *Amazon ElastiCache User Guide.* Basically, ElastiCache adds a special Memcached configuration variable called `cluster` that contains the DNS names of the current cache nodes. To access this list, your application connects to your cache cluster configuration endpoint, which is a hostname ending in `cfg.region.cache.amazonaws.com`. Once you retrieve the list of cache node host names, your application configures its Memcached client to connect to the list of cache nodes, using consistent hashing to balance across them. Here is a complete working example in Ruby:

```
require 'socket'
require 'dalli'

socket = TCPSocket.new(
          'my-cache-
2a.z2vq55.cfg.usw2.cache.amazonaws.com', 11211
         )
socket.puts("config get cluster")

header   = socket.gets
version  = socket.gets
nodelist = socket.gets.chomp.split(/\s+/).map{|l|
l.split('|').first }
socket.close

# Configure Memcached client
cache = Dalli::Client.new(nodelist)
```

Using Linux utilities, you can even do this from the command line using netcat, which can be useful in a script:

```
ec2-host$ echo "config get cluster" | \
nc my-cache-2a.z2vq55.cfg.usw2.cache.amazonaws.com 11211 |
\
grep 'cache.amazonaws.com' | tr ' ' '\n' | cut -d'|' -f 1
```

```
my-cache-2a.z2vq55.0001.usw2.cache.amazonaws.com

my-cache-2a.z2vq55.0002.usw2.cache.amazonaws.com
```

Using Auto Discovery, your Amazon EC2 application servers can locate Memcached nodes as they are added to a cache cluster. However, once your application has an open socket to a Memcached instance, it won't necessarily detect any changes to the cache node list that might happen later. To make this a complete solution, two more things are needed:

- The ability to scale cache nodes as needed
- The ability to trigger an application reconfiguration on the fly

Cluster Reconfiguration Events from Amazon SNS

Amazon ElastiCache publishes a number of notifications to Amazon SNS when a cluster change happens, such as a configuration change or replacement of a node. Because these notifications are sent through Amazon SNS, you can route them to multiple endpoints, including email, Amazon SNS, or other Amazon EC2 instances. For a complete list of Amazon SNS events that ElastiCache publishes, refer to ElastiCache SNS Events in the *Amazon ElastiCache User Guide.*

If you want your application to dynamically detect nodes that are being added or removed, you can use these notifications as follows. Note that the following process is *not* required to deal with cache node failures. If a cache node fails and is replaced by ElastiCache, the DNS name will remain the same. Most client libraries should automatically reconnect once the cache node becomes available again.

The two most interesting events that ElastiCache publishes, at least for the purposes of scaling our cache, are **ElastiCache:AddCacheNodeComplete** and **ElastiCache:RemoveCacheNodeComplete**. These events are published when cache nodes are added or removed from the cluster. By listening for these events, your application can dynamically reconfigure itself to detect the new cache nodes. The basic process for using Amazon SNS with your application is as follows:

1. Create an Amazon SNS topic for your ElastiCache alerts, as described in the Creating an SNS Topic section in the *Amazon ElastiCache User Guide.*
2. Modify your application code to subscribe to this Amazon SNS topic. All of your application instances will listen to the same topic. The AWS PHP blog has an in-depth posting with code samples: Receiving Amazon SNS Messages in PHP.

3. When a cache node is added or removed, you will receive a corresponding Amazon SNS message. At that point, your application needs to be able to rerun the Auto Discovery code we discussed preceding to get the updated cache node list.
4. Once your application has the new list of cache nodes, it also reconfigures its Memcached client accordingly.

Again, this workflow is not needed for cache node recovery—only if nodes are added or removed dynamically, *and* you want your application to dynamically detect them. Otherwise, you can simply add the new cache nodes to your application's configuration, and restart your application servers. To accomplish this with zero downtime to your app, you can leverage solutions such as zero-downtime deploys with Elastic Beanstalk.

Conclusion

Proper use of in-memory caching can result in an application that performs better and costs less at scale. Amazon ElastiCache greatly simplifies the process of deploying an in-memory cache in the cloud. By following the steps outlined in this paper, you can easily deploy an ElastiCache cluster running either Memcached or Redis on AWS, and then use the caching strategies we discussed to increase the performance and resiliency of your application. You can change the configuration of ElastiCache to add, remove, or resize nodes as your application's needs change over time, in order to get the most out of your in-memory data tier.