ElastiCache

Fully managed implementations of two popular in-memory data stores – Redis and Memcached.

ElastiCache is a web service that makes it easy to deploy and run Memcached or Redis protocol-compliant server nodes in the cloud.

The in-memory caching provided by ElastiCache can be used to significantly improve latency and throughput for many read-heavy application workloads or compute-intensive workloads.

Best for scenarios where the DB load is based on Online Analytics Processing (OLAP) transactions.

Push-button scalability for memory, writes and reads.

In-memory key/value store – not persistent in the traditional sense.

Billed by node size and hours of use.

ElastiCache EC2 nodes cannot be accessed from the Internet, nor can they be accessed by EC2 instances in other VPCs.

Cached information may include the results of I/O-intensive database queries or the results of computationally intensive calculations.

Can be on-demand or reserved instances too (but not Spot instances).

ElastiCache can be used for storing session state.

A node is a fixed-sized chunk of secure, network-attached RAM and is the smallest building block.

Each node runs an instance of the Memcached or Redis protocol-compliant service and has its own DNS name and port.

Failed nodes are automatically replaced.

Access to ElastiCache nodes is controlled by VPC security groups and subnet groups (when deployed in a VPC).

Subnet groups are a collection of subnets designated for your Amazon ElastiCache Cluster.

You cannot move an existing Amazon ElastiCache Cluster from outside VPC into a VPC.

You need to configure subnet groups for ElastiCache for the VPC that hosts the EC2 instances and the ElastiCache cluster.

When not using a VPC, Amazon ElastiCache allows you to control access to your clusters through Cache Security Groups (you need to link the corresponding EC2 Security Groups).

ElastiCache nodes are deployed in clusters and can span more than one subnet of the same subnet group.

A cluster is a collection of one or more nodes using the same caching engine.

Applications connect to ElastiCache clusters using endpoints.

An endpoint is a node or cluster’s unique address.

Maintenance windows can be defined and allow software patching to occur.

There are two types of ElastiCache engine:

* Memcached – simplest model, can run large nodes with multiple cores/threads, can be scaled in and out, can cache objects such as DBs.
* Redis – complex model, supports encryption, master / slave replication, cross AZ (HA), automatic failover and backup/restore.

**Use Cases**

The following table describes a few typical use cases for ElastiCache:

|  |  |
| --- | --- |
| **Use Case** | **Benefit** |
| Web session store | In cases with load-balanced web servers, store web session information in Redis so if a server is lost, the session info is not lost, and another web server can pick it up |
| Database caching | Use Memcached in front of AWS RDS to cache popular queries to offload work from RDS and return results faster to users |
| Leaderboards | Use Redis to provide a live leaderboard for millions of users of your mobile app |
| Streaming data dashboards | Provide a landing spot for streaming sensor data on the factory floor, providing live real-time dashboard displays |

***Exam tip:****the key use cases for ElastiCache are offloading reads from a database and storing the results of computations and session state. Also, remember that ElastiCache is an in-memory database and it’s a managed service (so you can’t run it on EC2).*

The table below describes the requirements that would determine whether to use the Memcached or Redis engine:

|  |  |
| --- | --- |
| **Memcached** | **Redis** |
| Simple, no-frills | You need encryption |
| You need to elasticity (scale out and in) | You need HIPAA compliance |
| You need to run multiple CPU cores and threads | Support for clustering |
| You need to cache objects (e.g. database queries) | You need complex data types |
|  | You need HA (replication |
|  | Pub/Sub capability |

**Memcached**

Simplest model and can run large nodes.

It can be scaled in and out and cache objects such as DBs.

Widely adopted memory object caching system.

Multi-threaded.

Scales out and in, by adding and removing nodes.

Ideal front-end for data stores (RDS, Dynamo DB etc.).

Use cases:

* Cache the contents of a DB.
* Cache data from dynamically generated web pages.
* Transient session data.
* High frequency counters for admission control in high volume web apps.

Max 100 nodes per region, 1-20 nodes per cluster (soft limits).

Can integrate with SNS for node failure/recovery notification.

Supports auto-discovery for nodes added/removed from the cluster.

Scales out/in (horizontally) by adding/removing nodes.

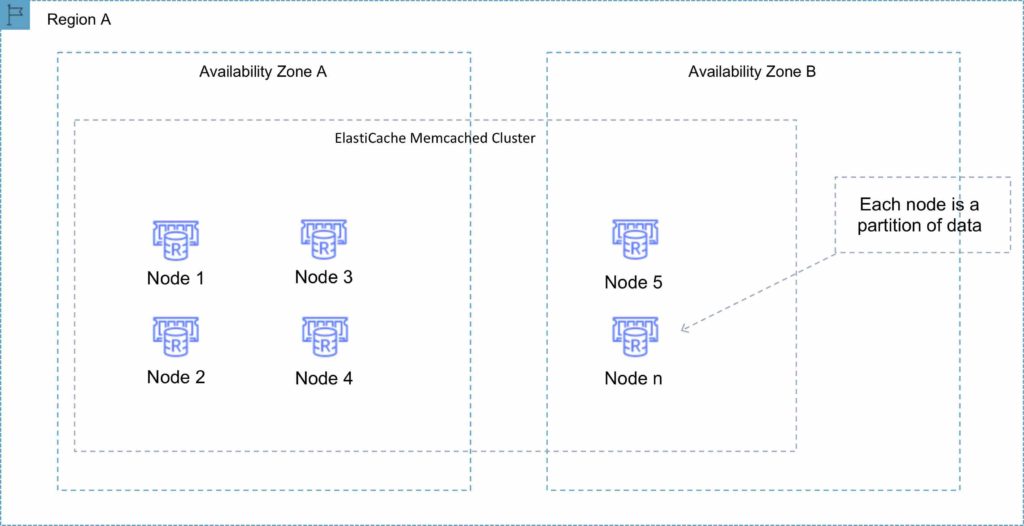
Scales up/down (vertically) by changing the node family/type.

Does not support multi-AZ failover or replication.

Does not support snapshots.

You can place nodes in different AZs.

With ElastiCache Memcached each node represents a partition of data and nodes in a cluster can span availability zones:



**Redis**

Open-source in-memory key-value store.

Supports more complex data structures: sorted sets and lists.

Data is persistent and it can be used as a datastore.

Redis is not multi-threaded.

Scales by adding shards, not nodes.

Supports master / slave replication and multi-AZ for cross-AZ redundancy.

Supports automatic failover and backup/restore.

A Redis shard is a subset of the cluster’s keyspace, that can include a primary node and zero or more read replicas.

Supports automatic and manual snapshots (S3).

Backups include cluster data and metadata.

You can restore your data by creating a new Redis cluster and populating it from a backup.

During backup you cannot perform CLI or API operations on the cluster.

Automated backups are enabled by default (automatically deleted with Redis deletion).

You can only move snapshots between regions by exporting them from ElastiCache before moving between regions (can then populate a new cluster with data).

**Clustering mode disabled**

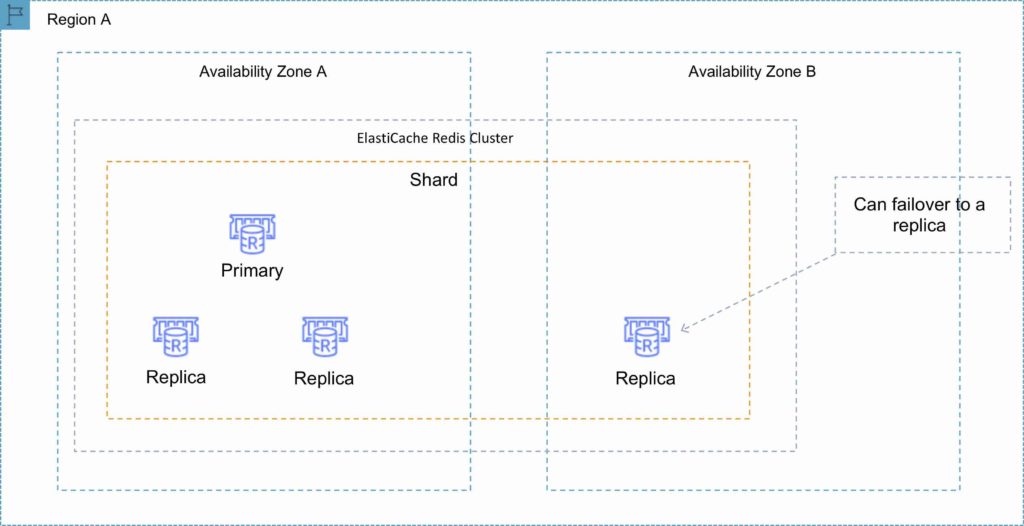
You can only have one shard.

One shard can have one read/write primary node and 0-5 read only replicas.

You can distribute the replicas over multiple AZs in the same region.

Replication from the primary node is asynchronous.

A Redis cluster with cluster mode disabled is represented in the diagram below:



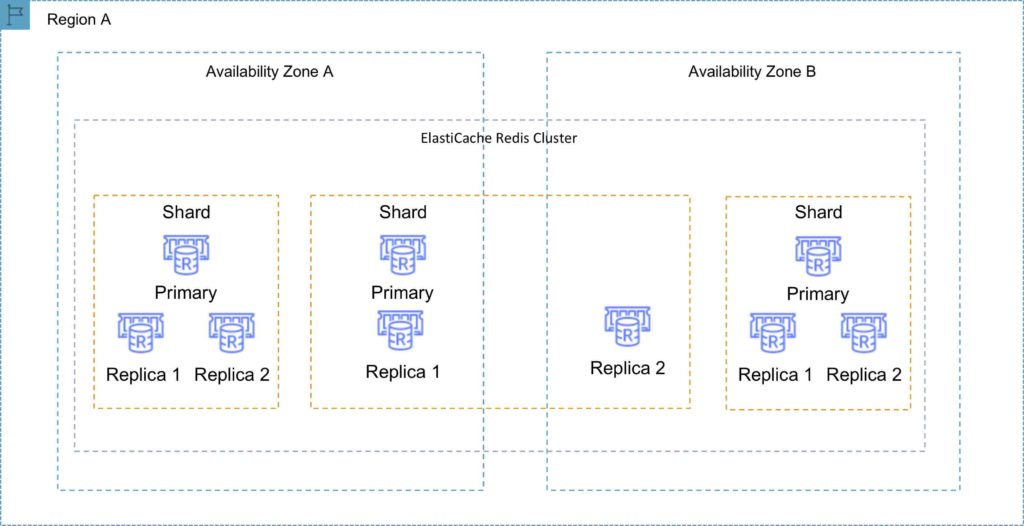
**Clustering mode enabled**

Can have up to 15 shards.

Each shard can have one primary node and 0-5 read only replicas.

Taking snapshots can slow down nodes, best to take from the read replicas.

A Redis cluster with cluster mode enabled is represented in the diagram below:



**Multi-AZ failover**

Failures are detected by ElastiCache.

ElastiCache automatically promotes the replica that has the lowest replica lag.

DNS records remain the same but point to the IP of the new primary.

Other replicas start to sync with the new primary.

You can have a fully automated, fault tolerant ElastiCache-Redis implementation by enabling both cluster mode and multi-AZ failover.

The following table compares the Memcached and Redis engines:

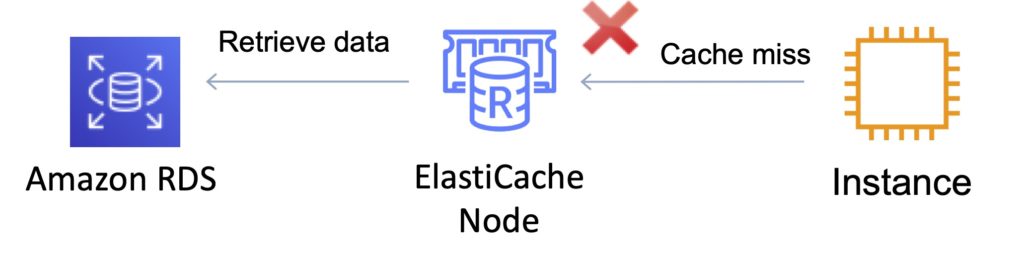
|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Memcached** | **Redis (cluster mode disabled)** | **Redis (cluster mode enabled)** |
| Data persistence | No | Yes | Yes |
| Data types | Simple | Complex | Complex |
| Data partitioning | Yes | No | Yes |
| Encryption | No | Yes | Yes |
| High availability (replication) | No | Yes | Yes |
| Multi-AZ | Yes, place nodes in multiple AZs. No failover or replication | Yes, with auto-failover. Uses read replicas (0-5 per shard) | Yes, with auto-failover. Uses read replicas (0-5 per shard) |
| Scaling | Up (node type); out (add nodes) | Single shard (can add replicas) | Add shards |
| Multithreaded | Yes | No | No |
| Backup and restore | No (and no snapshots) | Yes, automatic and manual snapshots | Yes, automatic and manual snapshots |

**Caching strategies**

There are two caching strategies available: Lazy Loading and Write-Through:

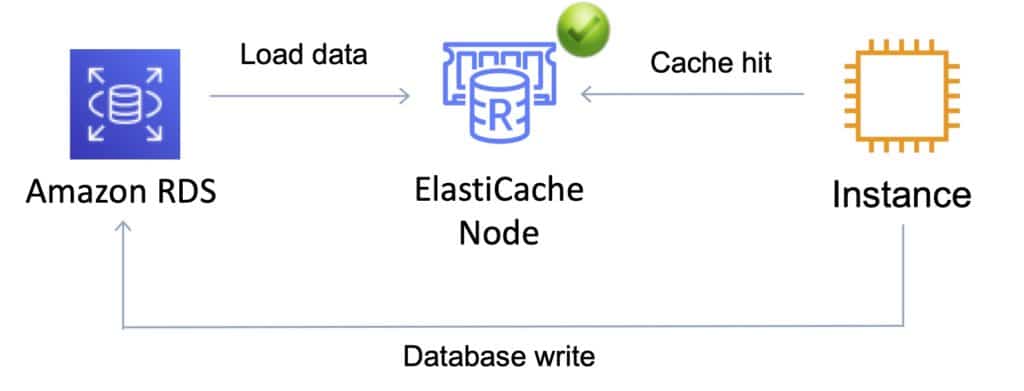
**Lazy Loading**

* Loads the data into the cache only when necessary (if a cache miss occurs).
* Lazy loading avoids filling up the cache with data that won’t be requested.
* If requested data is in the cache, ElastiCache returns the data to the application.
* If the data is not in the cache or has expired, ElastiCache returns a null.
* The application then fetches the data from the database and writes the data received into the cache so that it is available for next time.
* Data in the cache can become stale if Lazy Loading is implemented without other strategies (such as TTL).



**Write Through**

* When using a write-through strategy, the cache is updated whenever a new write or update is made to the underlying database.
* Allows cache data to remain up to date.
* This can add wait time to write operations in your application.
* Without a TTL you can end up with a lot of cached data that is never read.



**Dealing with stale data – Time to Live (TTL)**

* The drawbacks of lazy loading and write through techniques can be mitigated by a TTL.
* The TTL specifies the number of seconds until the key (data) expires to avoid keeping stale data in the cache.
* When reading an expired key, the application checks the value in the underlying database.
* Lazy Loading treats an expired key as a cache miss and causes the application to retrieve the data from the database and subsequently write the data into the cache with a new TTL.
* Depending on the frequency with which data changes this strategy may not eliminate stale data – but helps to avoid it.

***Exam tip:****Compared to DynamoDB Accelerator (DAX) remember that DAX is optimized for DymamoDB specifically and only supports the write-through caching strategy (does not use lazy loading).*

**Monitoring and Reporting**

**Memcached Metrics**

The following [**CloudWatch**](https://digitalcloud.training/amazon-cloudwatch/) metrics offer good insight into ElastiCache Memcached performance:

**CPUUtilization –** This is a host-level metric reported as a percent. Because Memcached is multi-threaded, this metric can be as high as 90%. If you exceed this threshold, scale your cache cluster up by using a larger cache node type, or scale out by adding more cache nodes.

**SwapUsage –** This is a host-level metric reported in bytes. This metric should not exceed 50 MB. If it does, we recommend that you increase the ConnectionOverhead parameter value.

**Evictions –** This is a cache engine metric. If you exceed your chosen threshold, scale your cluster up by using a larger node type, or scale out by adding more nodes.

**CurrConnections –** This is a cache engine metric. An increasing number of CurrConnections might indicate a problem with your application; you will need to investigate the application behavior to address this issue.

**Redis Metrics**

The following CloudWatch metrics offer good insight into ElastiCache Redis performance:

**EngineCPUUtilization –** Provides CPU utilization of the Redis engine thread. Since Redis is single threaded, you can use this metric to analyze the load of the Redis process itself.

**MemoryFragmentationRatio –** Indicates the efficiency in the allocation of memory of the Redis engine. Certain threshold will signify different behaviors. The recommended value is to have fragmentation above 1.0.

**CacheHits –** The number of successful read-only key lookups in the main dictionary.

**CacheMisses –** The number of unsuccessful read-only key lookups in the main dictionary.

**CacheHitRate –** Indicates the usage efficiency of the Redis instance. If the cache ratio is lower than ~0.8, it means that a significant number of keys are evicted, expired, or do not exist.

**CurrConnections –** The number of client connections, excluding connections from read replicas. ElastiCache uses two to four of the connections to monitor the cluster in each case.

**Logging and Auditing**

All Amazon ElastiCache actions are logged by AWS CloudTrail.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

* Whether the request was made with root or IAM user credentials.
* Whether the request was made with temporary security credentials for a role or federated user.
* Whether the request was made by another AWS service.

**Authorization and Access Control**

Access to Amazon ElastiCache requires credentials that AWS can use to authenticate your requests. Those credentials must have permissions to access AWS resources, such as an ElastiCache cache cluster or an Amazon Elastic Compute Cloud (Amazon EC2) instance.

You can use [**identity-based policies**](https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/IAM.IdentityBasedPolicies.html) with Amazon ElastiCache to provide the necessary access.

You can use [**Redis Auth**](https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/auth.html) to require a token with ElastiCache Redis.

The Redis authentication tokens enable Redis to require a token (password) before allowing clients to run commands, thereby improving data security.

**Charges**

Pricing is per Node-hour consumed for each Node Type.

Partial Node-hours consumed are billed as full hours.

There is no charge for data transfer between Amazon EC2 and Amazon ElastiCache within the same Availability Zone.

High Availability for ElastiCache

Memcached:

* Because Memcached does not support replication, a node failure will result in data loss.
* Use multiple nodes to minimize data loss on node failure.
* Launch multiple nodes across available AZs to minimize data loss on AZ failure.

Redis:

* Use multiple nodes in each shard and distribute the nodes across multiple AZs.
* Enable Multi-AZ on the replication group to permit automatic failover if the primary nodes fail.
* Schedule regular backups of your Redis cluster.

<https://digitalcloud.training/aws-resource-access-manager/>

<https://digitalcloud.training/amazon-rds/>

<https://digitalcloud.training/amazon-emr/>