

Deploying a Highly Available Distributed Caching Layer on Oracle Cloud Infrastructure using Memcached & Redis

ORACLE WHITEPAPER | FEBRUARY 2018 | VERSION 1.0





Table of Contents

Purpose of this Whitepaper	1
Scope & Assumptions	1
Introduction	1
Memcached vs Redis	2
Architecting Memcached on Oracle Cloud Infrastructure	3
Deployment planning	3
Instance shape selection	3
Building Memcached cluster on Oracle Cloud Infrastructure	3
Caching Strategies	5
Cache-aside	5
Read-Through	6
Write-Through	7
Write-Back	9
Scaling Memcached - Consistent Hashing	10
Architecting Redis on Oracle Cloud Infrastructure	11
Deployment planning	12
Instance shape selection	12
Building Redis cluster on Oracle Cloud Infrastructure	13
Non-clustered multi-AD deployment	13
Clustered multi-AD deployment	14
Backup and Restore of Redis on Oracle Cloud Infrastructure	15
Conclusion	16
References	16





Purpose of this Whitepaper

This white paper is intended for Oracle customers and partners who are building solutions that will be deployed on Oracle Cloud Infrastructure (OCI). In this white paper we shall compare the two most widely used in-memory cache engines: Redis and Memcached. We shall understand the use cases for in-memory caching, discuss common caching design patterns, performance considerations and the best practices for deploying them on to the Oracle Cloud Infrastructure.

By the end of this paper, you should have a clear grasp of which caching strategies apply to your use case and how to architect them using Redis or Memcached. Finally, we shall learn how to deploy and scale the in-memory caching layer for your application on Oracle Cloud Infrastructure.

Scope & Assumptions

Caching can be leveraged at every layer of the technology stack ranging from operating systems, Networking (CDN), Domain Name System (DNS), web applications, and databases, but we will not be going over each one of them. The scope of this paper is strictly limited to in-memory shared caching layer associated with large scale distributed systems as opposed to instance-specific private caches.

There are a number of products and topics that are beyond the scope of this document. While not all topics are listed, Virtual Cloud Network (VCN), Identity Access Management (IAM), security lists, Load Balancer are examples of additional components needed to architect an in-memory caching solution on Oracle Cloud Infrastructure. Readers of this document are suggested to familiarize themselves with these services, the links to which are in the References section.


Introduction

In computing, a cache is a high-speed data layer which stores the result of an operation, so that the future requests are fetched faster rather than being accessed from the relatively slower primary data store. Caching improves application performance by storing frequently accessed data items in memory, so that they can be quickly retrieved without accessing the primary data store. Properly leveraging caching can result in an application that not only performs better, but also costs less at scale.

Caching is one of the most effective techniques to speed up a website and has become a staple of modern web architectures. Effective caching strategies will allow you to get the most out of your website, ease the pressure on your database, and offer a better experience for users. An effective caching strategy is perhaps the single biggest factor in creating an application that performs well at scale. For an application that repeatedly accesses data from a primary data store, caching is most effective when any of the following criteria are met:

- Computation intensive queries performed on the primary data store
- Relatively static data resides on the primary data store
- A relatively slow primary data store
- Limited support for concurrent connections to the primary data store

Caching is effective for data that is read frequently but modified infrequently. The cache shouldn't be used as an authoritative source of critical information. It's good practice to fall back to a more persistent primary data



store for applications which cannot afford to lose any information. Let's now look into the in-memory caching solutions offered by Memcached and Redis.

Memcached vs Redis

Although Redis and Memcached are both in-memory key-value stores and appear to be very similar, under the hood they are quite different.

Memcached is a widely adopted in-memory key-value store. It is fast, flexible and lightweight. Its manageable design promotes fast deployment and solves many problems related to large data caches. Its APIs provide access to a very large hash table distributed across multiple machines. Memcached server is multi-threaded, meaning it makes good use of the larger instance shapes which offer multiple cores.

Redis is a popular open-source in-memory data structure store which supports a wide array of data structures and not just key-value pairs. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries. Redis has replication built-in which can provide easy redundancy across multiple Availability Domains. It can also support different levels of disk persistence. Redis is single threaded, so it cannot effectively benefit from multi-core instances.

Because of the built-in replication and disk persistence options in Redis, it makes more sense to treat it more as a non-relational database with support for in-memory caching. Conversely, Memcached is a pure light-weight caching solution. Here are a few deciding factors which help you choose one over the other.

Use Memcached

- If just caching is your primary goal and nothing else
- If you are caching relatively small and static data, such as HTML code fragments. Memcached's internal memory management, is more efficient in the simplest use cases.
- If you are running on large instance sizes and require multi-thread performance with utilization of multiple-cores. Memcached being multithreaded, can easily scale up by giving it more computational resources.

Use Redis

- If you are looking for more advanced data sets like lists, hashes, sets, sorted sets, bitmaps and hyperloglogs
- If your application requires Publish Subscribe (pub/sub) functionality
- If you are looking for additional persistence to disk (with eventual consistency) in addition to just caching

Use Redis or Memcached

- If you are planning to horizontally scale out your cache layer as you grow. In Memcached it is possible to do this just by adding more nodes while in Redis this can be done by clustering which is built-in.
- If your caching needs a Check and Set operation to maintain strong consistency. Memcached supports this out of the box, whereas Redis does this using a similar operation, which provides optimistic locking.

Architecting Memcached on Oracle Cloud Infrastructure

Deployment planning

The primary goal of Memcached deployment is to offload the reads as much as possible from your database. It is generally a good practice to have extra cache space than what your data would require, to have a substantial buffer in case there is a traffic spike so that the database does not get overwhelmed. A successful Memcached deployment also requires the administrator to weigh multiple factors: initial volume size of data, future data growth, and redundancy options.

Instance shape selection

Memcached cluster can contain one or more instances. You can attain your needed cluster memory capacity by having a few large instances or many smaller instances. There are tradeoffs for both approaches:

Selecting fewer larger instances: In this case, the loss of one instance can have a significant impact on the backend database. On the plus side, the large sized instances on Oracle Cloud Infrastructure provide very high Network I/O, up to 25Gbps. Given the multi-threaded nature of Memcached, you can also effectively utilize multiple cores provided in large sized instances. A good example of a high performance, large sized instance suitable for Memcached is the VM.DenseIO1.4 instance type, which provides 4 OCPU cores and 60GB of memory with enhanced Network I/O.

Selecting many smaller instances: In this case, the loss of an instance does not significantly burden the database, but at the cost of providing poor network I/O. Obviously, poor network I/O can cause performance problems. Based on the cache access patterns, this can also cause problems like [TCP Incast](#) which can degrade the performance of the cache. A good example of a high performance small sized instance shape suitable for Memcached deployment is the VM.Standard2.2, which offers two OCPU cores, giving you the option to leverage multi-threading and 30GB of instance memory.

Building Memcached cluster on Oracle Cloud Infrastructure

When deploying Memcached on a public cloud, the common design strategy involves deploying the Memcached instances across multiple Availability Domains (ADs) within a region. In Oracle Cloud Infrastructure, ADs are independent datacenters, isolated from each other, fault tolerant, and very unlikely to fail simultaneously. This ensures High Availability of your cache clusters. Also, since the latency between ADs is on the order of milliseconds, this should not significantly affect the performance of the cache in any way.

Use security lists and private subnets to limit the access of our cache instances from the Internet. Since Memcached or Redis have no serious authentication or encryption capabilities, it's a best practice to launch these instances in private subnets. There are various caching design patterns involving Memcached, some of the patterns might not need direct access to the database tier, as your database does not directly interact with these instances. Only application tier instances make calls to the Memcached instances. We shall discuss this in greater detail in the caching strategies section. Let's walkthrough the deployment steps:

- Create a VCN big enough to house the application servers, Memcached servers and database servers. Refer to **VCN Overview and Deployment Guide** in references for more information on how to create a VCN and the associated best practices.
- Create two public subnets in two separate Availability Domains, say AD1 and AD2 to act as the DMZ subnets to host the bastion instances and the public Load Balancer. In Oracle Cloud Infrastructure, the subnets in a VCN are bound to a specific AD. We create one subnet each in each Availability

Domain. For more information on Availability Domains and subnets, refer to **Overview of OCI Networking** in the references section.

- Create a public load balancer pair to load balance the traffic between the application servers and place it in the DMZ subnets. OCI's public load balancer come as an Active/Standby pair by design to provide high availability. Active and Standby should be placed in separate subnets and in separate Availability Domains. Refer to **Overview of OCI Load Balancing** for more information.
- Create two private subnets in AD1 and AD2 to house the application servers. Make sure both the subnets are still in the same VCN.
- Create two private subnets each in AD1 and AD2 to house the Memcached instances.
- Create two other private subnets each in AD1 and AD2 to house the database servers. You can choose the OCI's Database service to create any flavor of Oracle's Databases for you, based on your application requirement. Refer to **Overview of OCI Database** in references for more information.
- Create appropriate route rules and security rules for the subnets created. Remember to create a security rule on your cache servers' subnet to allow inbound access on port 11211 for TCP and UDP, for application servers and/or database servers based on the appropriate caching strategy used, to interact with the cache instances. For more information, to create route and security rules, refer to **Overview of Security lists and Route tables** in References section.

Once the Memcached clusters are successfully deployed on the OCI, you should configure the cache clusters with the right kind of caching algorithm based on your use cases. After deciding on your caching strategy, you need to apply the right set of security list rules to facilitate the inbound and outbound communication with the cache clusters. We shall discuss that next.

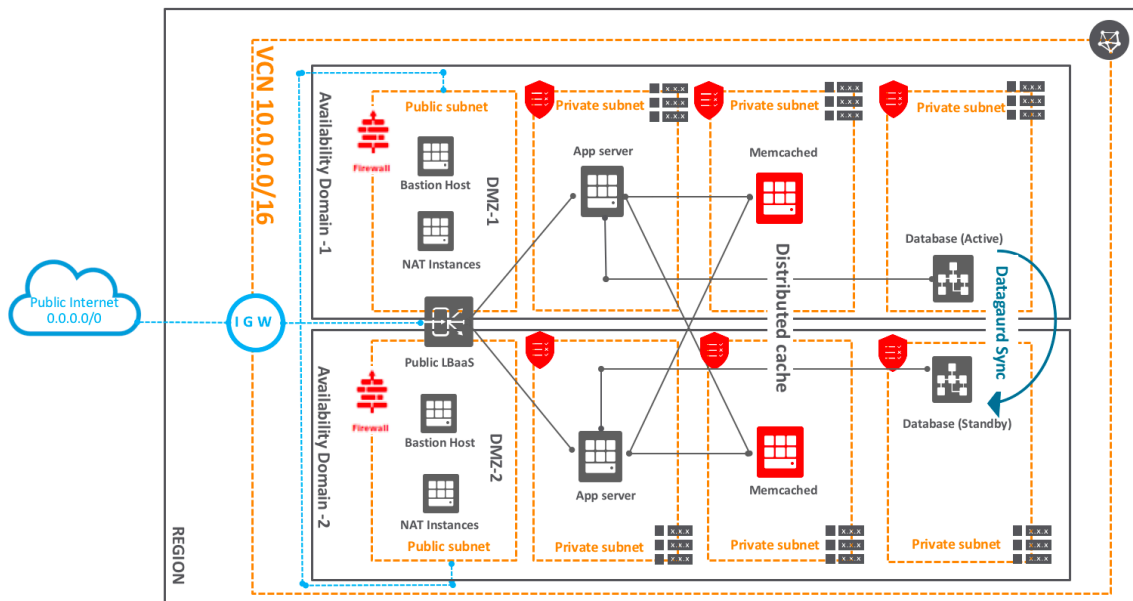


Figure 1 – Distributed Memcached architecture on OCI

Caching Strategies

There are several strategies and choosing the right one can make a big difference. Your caching strategy depends on the data and data access patterns. In other words, how the data is written and read. For instance:

- Is the application write-heavy? (e.g. time based logs)
- Is data written once and read multiple times? (e.g. User Profile)
- Is data returned always unique? (e.g. search queries)

Let's take a look at a various caching strategies.

Cache-aside

This is perhaps the most commonly used caching approach, especially for caches involving Memcached. The cache sits on the side and the application directly talks to both the cache and the database. Cache-aside strategy loads data "lazily," that is, only when it is first read.

- The application first checks the cache.
- If the data is found in cache, it's a cache hit. The data is read and returned to the client.
- If the data is not found in cache, it's a cache miss. The application has to do some extra work of querying the database to read the data, return it to the client and store the data in cache so subsequent reads for the same data results in a cache hit.

Cache-aside caches are usually general purpose and work best for read-heavy workloads. Systems using cache-aside are resilient to cache failures. If the cache cluster goes down, the system can still operate by going directly to the database. (Although, it doesn't help much if cache goes down during peak load. Response times can become terrible and in worst case, the database can stop working.)

Another benefit is that the data model in cache can be different than the data model in database. For example, the response generated as a result of multiple queries can be stored against the same request id on cache.

When cache-aside is used, the most common write strategy is to write data to the database directly. When this happens, cache may become inconsistent with the database. To deal with this, developers generally use time to live (TTL) and continue serving stale data until TTL expires. If data freshness must be guaranteed, developers either invalidate the cache entry or use an appropriate write strategy, as we'll explore later. Here are the security list rules needed for implementing cache-aside cache on OCI.

Stateful Security List rules needed for implementing Cache-aside cache on OCI

Direction	Source subnet	Protocol	Source Port	Destination Port	Description
INGRESS	<App subnet>	TCP	ALL	11211	Allows inbound access to Memcached
INGRESS	<App subnet>	UDP	ALL	11211	Allows inbound access to Memcached

Direction	Stateful Rule
Ingress	Default deny all ports
Ingress	Allow UDP & TCP 11211, App subnet

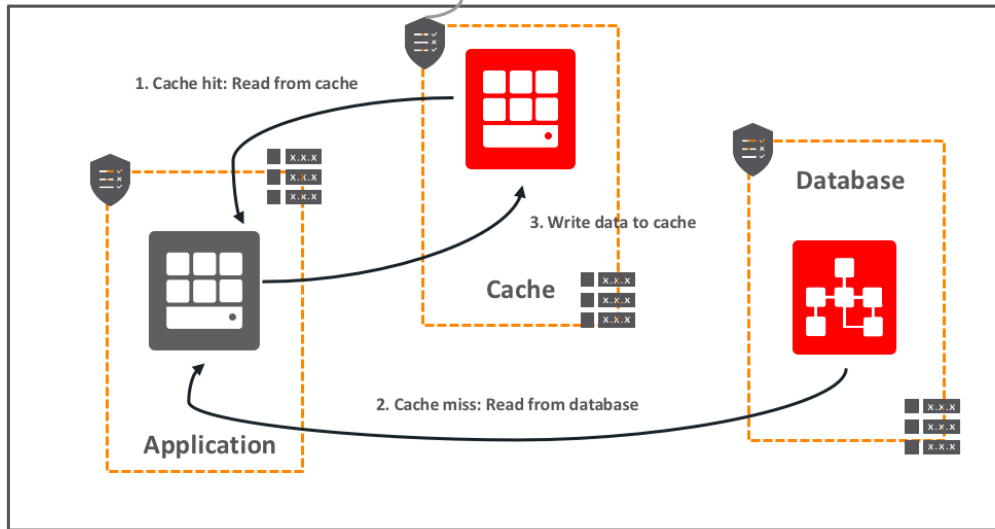


Figure 2 – Cache-aside

Python pseudo code for implementing cache-aside cache

```
def get_image(image_id):
    record = memcache.get(image_id) # fetch from cache
    if record is None: # if MISS, fetch from DB
        record = db.query(select * from images where id = ?", image_id)
        memcache.set(image_id, record, 500) # Write to cache and set TTL value
    return record
```

Read-Through

Both cache-aside and read-through strategies load data “lazily,” that is, only when it is first read.

While read-through and cache-aside are very similar, there are at least two key differences:

- In cache-aside, the application is responsible for fetching data from the database and populating the cache. In read-through, this logic is usually supported by the Memcached library.
- Unlike cache-aside, the data model in read-through cache cannot be different than that of the database.

Read-through caches work best for read-heavy workloads when the same data is requested many times. For example, a news story. The disadvantage is that when the data is requested the first time, it always results in

a cache miss and incurs the extra penalty of loading data to the cache. This is referred to as Thundering herd. For information on Thundering herd, check the references section of this whitepaper. Developers deal with this by “warming” or “pre-heating” the cache by issuing queries manually. Just like cache-aside, it is also possible for data to become inconsistent between cache and the database. The solution lies in the write strategy, as we’ll see next. Here are the security list rules needed for implementing Read-Through cache.

Stateful Security List rules needed for implementing Read-Through cache on OCI

DIRECTION	SOURCE SUBNET	PROTOCOL	SOURCE PORT	DESTINATION PORT	DESCRIPTION
INGRESS	<App subnet>	TCP	ALL	11211	Allows inbound access to Memcached
INGRESS	<App subnet>	UDP	ALL	11211	Allows inbound access to Memcached
EGRESS	<cache subnet>	TCP	ALL	3306	Allows outbound access to MYSQL DB subnet

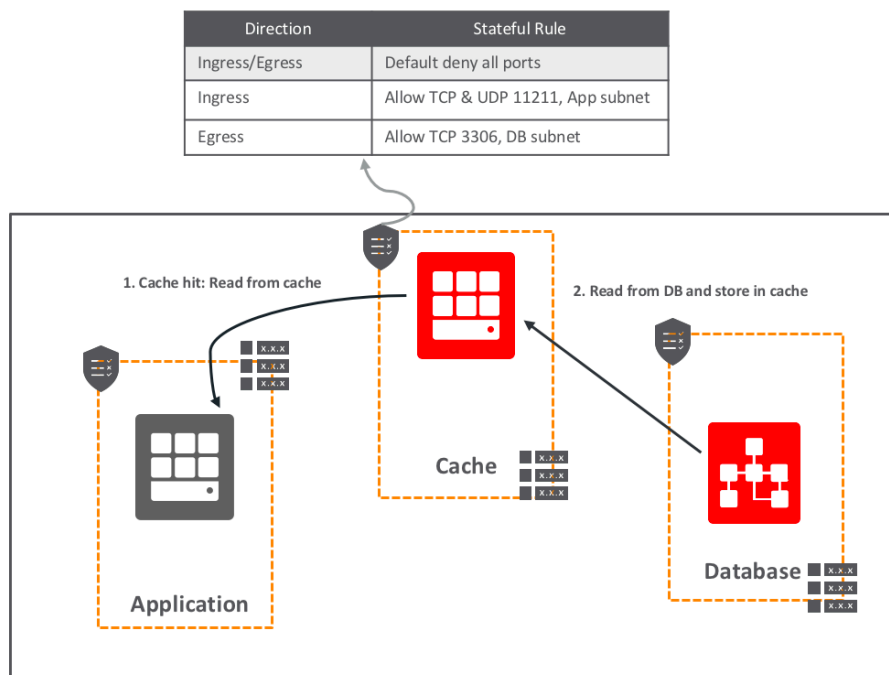


Figure 3 – Read-Through cache

Write-Through

In this case, when the application writes to the database, it also simultaneously writes to the cache. On their own, write-through caches don’t seem to do much. In fact, they introduce extra write latency because data is written to the cache first and then to the main database. But when paired with cache-aside caches, we get all

the benefits of read-through and we also get data consistency guarantee, freeing us from using cache invalidation techniques.

Stateful Security List rules needed for implementing Write-Through cache subnet on OCI

DIRECTION	SOURCE SUBNET	PROTOCOL	SOURCE PORT	DESTINATION PORT	DESCRIPTION
INGRESS	<App subnet>	TCP	ALL	11211	Allows inbound access to Memcached
INGRESS	<App subnet>	UDP	ALL	11211	Allows inbound access to Memcached

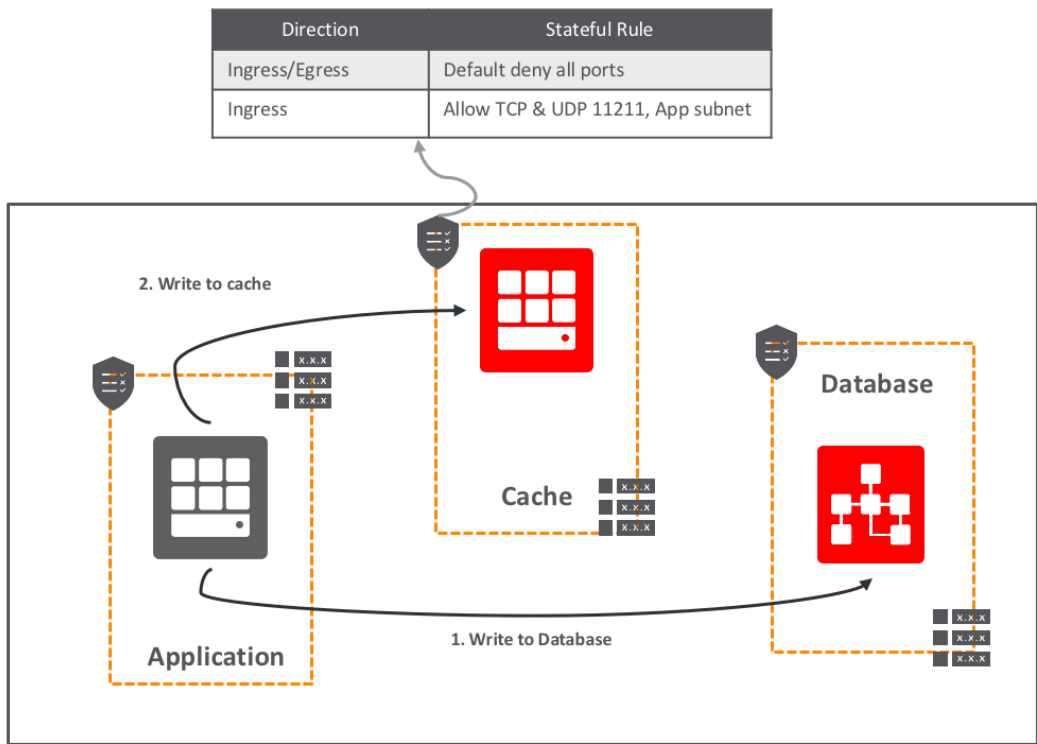


Figure 4 – Write –Through cache

Python pseudo code for implementing write-through cache.

```
def save_image(image_id, values):
    record = db.set("update images .. where id = x", image_id, values) #write to DB
    cache.set(image_id, record, 500) #write to cache and set the TTL
    return record
```

Write-Back

Here, the application writes data to the cache which acknowledges immediately and after some delay, it writes the data back to the database. This is sometimes called write-behind as well.

Write-back caches improve write performance and are good for write-heavy workloads. When combined with read-through, it works well for mixed workloads, where the most recently updated and accessed data is always available in cache. This is particularly used with Redis (not so much with Memcached) to better absorb spikes during peak load. The main disadvantage is that if there's a cache failure, the data may be permanently lost. Hence, with Memcached, this may not be a suitable caching strategy.

Stateful Security List rules needed for implementing Write-Back cache subnet on OCI

DIRECTION	SOURCE SUBNET	PROTOCOL	SOURCE PORT	DESTINATION PORT	DESCRIPTION
INGRESS	<App subnet>	TCP	ALL	11211	Allows inbound access to Memcached
INGRESS	<App subnet>	UDP	ALL	11211	Allows inbound access to Memcached
EGRESS	<cache subnet>	TCP	ALL	3306	Allows outbound access to MYSQL DB subnet

Direction	Stateful Rule
Ingress/Egress	Default deny all ports
Ingress	Allow TCP & UDP 11211, App subnet
Egress	Allow TCP 3306, DB subnet

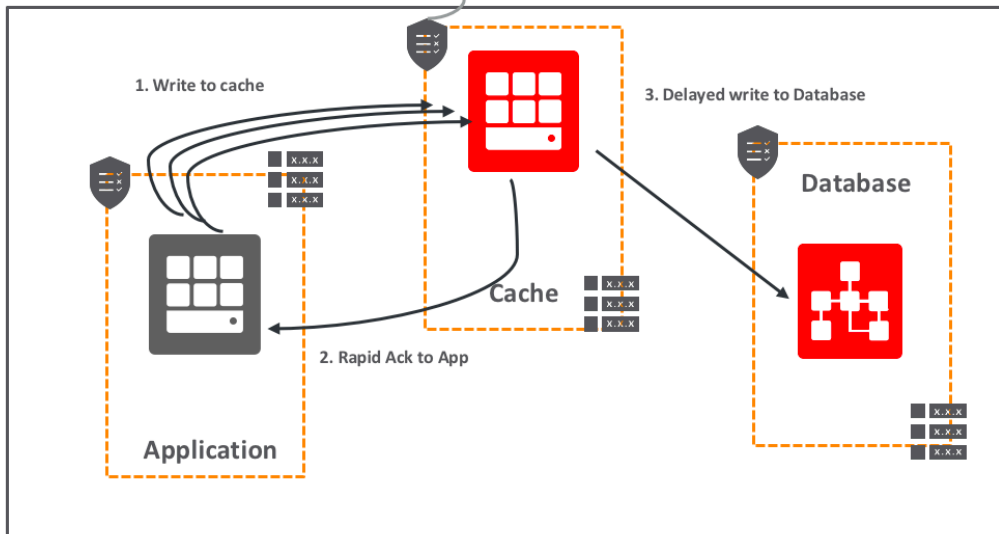


Figure 5 – Write-back cache


Scaling Memcached - Consistent Hashing

The vanilla version of Memcached does not have native support for persistence, replication and fault-tolerance as it is just a light-weight caching service. Although there are many extensions of Memcached which do offer these services (e.g. membrain, membase), these defeat the purpose of a lightweight cache. Once you start scaling to multiple instances of memcached across multiple nodes on the cloud, it is very important that the keys are consistently distributed so that the traffic load is evenly distributed among the instances. One of the most important features needed to create a scalable memcached infrastructure is consistent hashing. Consistent hashing is a great tool for partitioning data when things are scaled horizontally. It is widely used for scaling in-memory caches.

Memcached is a simple hash table where a unique key maps to a value. This is simple when you have a single memcached instance, but what do you do if you have more than one? Which instance should the key/value pair be stored on? The naive approach is to hash the key to an integer and do a modulo based on the size of the instance set. For a given key, this should return the same, random instance from the set every time.

```
idx = Zlib.crc32(key) % servers.size
```

This works great -- if the set of allocated instances never changes. Where things can go wrong is when there is a surge in traffic and you plan on adding a new instance to the set. When this is done, the module value for most of the key hashes also changes. This means that the Memcached-client chooses the "wrong" instance,



which ends up in a lookup miss and hence that expensive operation you are caching needs to be performed again, thereby thrashing the backend database. You essentially get a storm of recalculation as your cache contents shift from their old server to their new server.

Let's see how consistent hashing solves this problem. Instead of using a modulo, consistent hashing uses a predefined continuum of values which map onto an instance. We select N random integers (where N is around 100 or 200) for each server and sort those values into an array of N * instance.size values. To look up an instance for a key, we find the closest value \geq the key hash and use the associated instance. The values form a virtual circle; the key hash maps to a point on that circle and then we find the instance clockwise from that point. For more information on how *consistent hashing* works, refer to [David Karger's original paper on Consistent Hashing](#)

Architecting Redis on Oracle Cloud Infrastructure

Unlike Memcached, Redis is more than a simple key-value store in-memory cache. Redis can be referred to as a key-value database where values can contain more complex data types, with atomic operations defined on those data types. Redis combines the concepts of in-memory caching with the persistence, sharding, and a master-slave architecture of a traditional database. Redis can also hold complicated data-structures like lists, hashes, sets, sorted sets in-memory. Given the rich features offered by Redis out of the box, there can be a wide variety of deployment options. Before we start architecting a Redis cluster on Oracle Cloud Infrastructure, let's briefly review the characteristics of Redis:

- **Single Instance Architecture:** Redis runs as a single threaded application, called Redis Server. Redis server is responsible for storing data in memory. It handles all kinds of management and forms the major part of architecture. The Redis client can be the Redis console client or any application that uses the Redis API. As we saw, Redis stores everything in primary memory. Primary memory is volatile so we will lose all stored data once we restart our Redis server. Therefore, we need a way for our data to persist.
- **Persistence:** There are two ways to make Redis persist data: Redis Database File (RDB), or Append Only File (AOF). RDB is a snapshot style persistence format which makes a copy of all the data in memory and stores them in secondary storage. This happens at specified intervals, so there is chance that you will lose data written after RDB's last snapshot. AOF is a change-log style persistence format, which logs all the write operations received by the server. Therefore, every operation is persisted. The problem with using AOF is that it writes to disk for every operation and it is an expensive task. Also, the size of the AOF file is larger than the RDB file.
- **Backup and Recovery:** Redis does not provide any mechanism for data store backup and recovery. Therefore, if there is a storage crash or any other kind of disaster then all data will be lost. You can use RDB or AOF snapshots and store them in Oracle Cloud Infrastructure's Object Storage, which provides durable storage. We will discuss this once we start architecting a Redis cluster.
- **Partial High Availability:** Redis supports replication, both for high availability and to separate read workloads from write workloads. Redis asynchronously replicates its data to one or more nodes, called read replicas. This is similar to a master-slave architecture, with the Redis primary node being the master which handles both reads and writes and the read-replicas only handle reads. All the slaves contain exactly the same data as master. In case of a failure of master node (crash of master with loss of data on disk), Redis gives the ability to convert a slave into a master. There are many monitoring solutions available to perform this action, the most commonly used is [Redis Sentinel](#) which can handle service discover and automatic failover of Redis instances.

- **Maximum High Availability:** Clustering, although complicated provides the highest level of Availability for Redis instances. Redis achieves clustering by partitioning and replication of data. Partitioning involves sharding your data into multiple Redis instances such that every instance only contains a subset of the keys. This helps by taking some load off a particular instance as the data volume grows, and also reduces impact in case of a node failure. Redis supports primitive types of partitioning like range partitioning and hash partitioning. It natively does not support consistent hashing because its data structures such as multi-dimensional sets, lists and hashes cannot be horizontally sharded. But if you are using Redis just for storing simple key-value pairs, then you can leverage consistent hashing.

Deployment planning

A successful Redis deployment requires the administrator to weigh multiple factors:

- **Type of workload:** Write heavy workloads inherently tend to require significantly more memory and in case of Redis, it requires additional memory while taking AOF snapshots, or syncing a cluster primary with replica(s) failing over, or promoting a replica to cluster primary after a failover. It is a best practice to select the memory size of the instance to be twice as needed for data alone.
- **Memory:** Amount of memory currently needed and also factoring in the rate of growth of your data.
- **Mode of operation:** If you are implementing Redis in a non-clustered mode, take into consideration that the instance should have sufficient memory to accommodate all the data plus the overhead as described above. If you are implementing in a clustered mode, in addition to accommodating memory for snapshot/sync overhead, plan ahead to have sufficient amount of memory for your shards. This is because, for Redis versions 3.x and below, cluster re-sharding is a non-trivial process and is prone to downtime.

Instance shape selection

There are multiple factors which directly impact the performance of Redis and play a significant role in instance shape selection. For more information refer to [Redis benchmarking](#).

- **Network bandwidth:** In many situations, Redis throughput is directly impacted by network well before being limited by the CPU. Redis suggests having at least 10Gbps NIC for optimal performance.
- **CPU:** Being single threaded, Redis favors fast CPUs with large caches and not many cores. Use of older hypervisors can only worsen this problem.
- **Virtualization:** Redis runs significantly slower in a virtualized environment as compared to the bare-metal instances. Using any kind of nested virtualization only worsens this.

Factoring all the above requirements, Redis gives optimal performance on non-virtualized, network-optimized, high-memory instances. If you have significant amount of write-heavy workloads requiring extreme performance, you can use Oracle's BM.Standard1.36 bare metal instances, which can not only offer an un-virtualized environment but also provide 10Gbps of Network I/O, 256GB of memory with 36 OCPUs with an option to run multiple Redis servers on the single instance. For regular workloads, you can use VM.DenseIO instances which provide very good performance for a lower price, but with virtualization overhead.

Building Redis cluster on Oracle Cloud Infrastructure

A common design pattern for Redis involves directing very write-heavy small data workloads to Redis, while directing big blobs of data to SQL or an eventually consistent on-disk database. Sometimes Redis is also used to keep an in memory copy of a subset of the same data stored in the on-disk database. Redis provides [ACID](#) properties to some extent, although its implementation is non-trivial. If your application has strict ACID requirements, it's best to fall back to a true RDBMS like Oracle Cloud Infrastructure's Database service to achieve guaranteed atomicity, consistency, isolation and durability. Now let's walkthrough two common patterns of deploying and scaling Redis instances on OCI.

Non-clustered multi-AD deployment

Using read replicas with Redis, you can separate your read and write workloads. This separation lets you scale reads by adding replicas as your application grows. In this scenario, we deploy Redis nodes in two separate Availability Domains (AD), which provides high availability in case of an accidental failure of an entire AD. We place a Redis Master, which handles the writes from the Application servers, in one AD and multiple read replicas (slaves) which handle reads, in both the AD. We also provision Redis Sentinel, so that in an event of failure of the Master, Sentinel quickly does the failover by converting one of the read replicas to Master. Let's walkthrough the deployment steps:

- Create a VCN big enough to house your application servers, Redis servers and database servers. Refer to **VCN overview and deployment guide** for more information on how to create a VCN and the associated best practices.
- Create two public subnets in two separate Availability Domains, say AD1 and AD2 to act as the DMZ subnets to host the Bastion instances and the public load balancer. In Oracle Cloud Infrastructure, the subnets in a VCN are bound to a specific AD. Create one subnet each in each AD.
- Create a public load balancer pair to load balance the traffic between the application servers and place it in the DMZ subnets. By design, OCI's public load balancer come as an Active/Standby pair to provide high availability. Active and Standby should be placed in separate subnets and in separate Availability Domains.
- Create two private subnets in AD1 and AD2 to house the application servers. Make sure both the subnets are still in the same VCN.
- Create two private subnets each in AD1 and AD2 to house the Redis instances. Place the Redis Master in one Availability Domain and spread the read-replicas across AD1 and AD2. Finally, setup Sentinel on these instances to provide active monitoring of the Redis instances.
- Create two other private subnets each in AD1 and AD2 to house the database servers. You can choose the OCI's Database service to create any flavor of Oracle's Databases for you, based on your application requirements.
- Create appropriate Route table rules and Security List rules for the subnets created. Remember to create a security rule on your Redis servers' subnet to allow inbound access on TCP port 6379 for application servers and/or database servers to interact with the Redis instances.

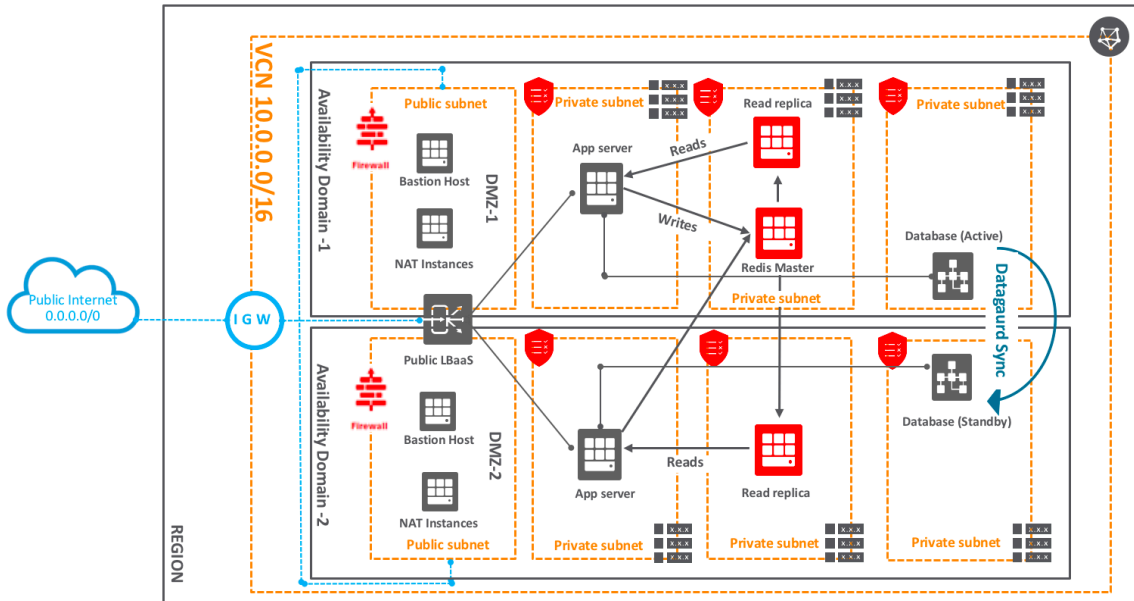


Figure 6 – Redis non-clustered multi-AD deployment on OCI

Clustered multi-AD deployment

Clustered architecture involves partitioning and replication of data across multiple Redis instances. This is particularly useful when the volume of data to be stored in memory by a single Redis instance is very large. The data gets sharded to multiple master nodes such that at any given time a particular master only stores a subset of keys. Just like the non-clustered approach, the data in the master can be replicated to multiple read-replicas to distribute the reads and writes. In this scenario, we place Redis Master nodes in two separate Availability Domains (AD) each storing a subset of actual data. You can determine the sharding logic based on your use case. Within each AD, we place multiple read replicas to handle the reads from the master. Let's walk through the deployment steps:

- Create a VCN big enough to house the application servers, Redis servers and database servers. Refer to **VCN overview and deployment guide** for more information on how to create a VCN and the associated best practices.
- Create two public subnets in two separate ADs, say AD1 and AD2 to act as the DMZ subnets to host the Bastion instances and the public Load balancer. In Oracle Cloud Infrastructure, the subnets in a VCN are bound to a specific AD. Create one subnet each in each AD.
- Create a public load balancer pair to load balance the traffic between the application servers and place it in the DMZ subnets. By design, OCI's Public Load balancer come as an Active/Standby pair to provide high availability. Active and Standby should be placed in separate subnets and in separate AD. Refer to OCI's Load Balancing for more information.
- Create two private subnets in AD1 and AD2 to house the Application servers. Make sure both the subnets are still in the same VCN.
- Create two private subnets each in AD1 and AD2 to house the Redis instances. Place one Redis Master in one AD and another Master in the second AD. Create read-replicas of the masters in the

same AD. Finally, setup Sentinel on these instances to provide active monitoring of the Redis instances and to failover master in case of failure.

- Create two other private subnets each in AD1 and AD2 to house the Database servers. You can choose the OCI's Database service to create any flavor of Oracle's Databases for you, based on your application requirement.
- Create appropriate Route rules and Security rules for the subnets created. Remember to create a security rule on your Redis servers' subnet to allow inbound access on TCP port 6379 for application servers and/or Database servers to interact with the Redis instances.

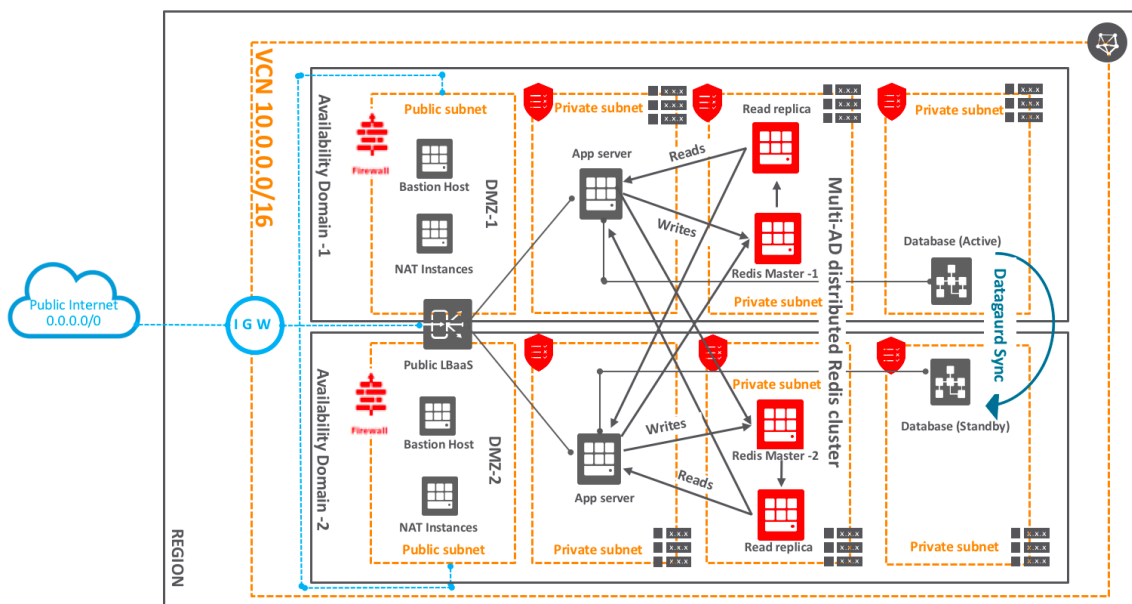



Figure 7 – Redis clustered multi-AD deployment on OCI

Backup and Restore of Redis on Oracle Cloud Infrastructure

As discussed earlier, persistence is built-in to Redis in the form of AOF and RDB. Redis periodically backs up data to disk either by taking point-in-time snapshots at specific intervals or by asynchronously writing to an append only file on disk for every write performed. As the volume of backup increases on disk, there should be a way to save these backups to more durable storage so that even in case of an instance failure, the backups are not lost.

Oracle Cloud Infrastructure's Object Storage is a highly durable and extremely scalable object store which can be used as a centralized store for saving Redis backup snapshots. Object Storage also offers APIs which are compatible with other provider's object store like S3 on AWS so that the Redis backups can be migrated to Object Storage on Oracle's Cloud Infrastructure. For more information on Object Storage, refer to Oracle Cloud Infrastructure documentation.

Note: If you plan on implementing backups of Redis, remember that Redis consumes extra memory during backups. This is because Redis forks a background process that writes the backup data using standard copy



on write semantics. Please refer to Redis deployment planning section of this paper for more information on memory requirements.

Conclusion

This white paper provides guidance for designing a highly available distributed caching layer on Oracle Cloud Infrastructure, using Memcached and Redis. By following the steps outlined in this paper, you can easily deploy either Memcached or Redis clusters, and then use the caching strategies we discussed to increase the performance and resiliency of your application. The paper also highlights the use cases, best practices and several scaling methodologies associated with a distributed in-memory caching layer.

References





- [Overview of Oracle Cloud Infrastructure](#)
- [Overview of Oracle Cloud Infrastructure Networking](#)
- [Overview of Oracle Cloud Infrastructure Security Lists](#)
- [Overview of Oracle Cloud Infrastructure Route tables](#)
- [Overview of Oracle Cloud Infrastructure Load Balancing](#)
- [Overview of Oracle Cloud Infrastructure Database](#)
- [Overview of Oracle Cloud Infrastructure Compute](#)
- [VCN Deployment and best practices guide](#)
- [Memcached Wiki](#)
- [Redis Sentinel](#)
- [Caching strategies](#)
- [TCP Incast](#)
- [Redis Benchmarking](#)
- [The Twelve-factor app](#)
- [Consistent Hashing reference](#)



Oracle Corporation, World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries
Phone: +1.650.506.7000
Fax: +1.650.506.7200

CONNECT WITH US

-  blogs.oracle.com/oracle
-  facebook.com/oracle
-  twitter.com/oracle
-  oracle.com

ORACLE
Cloud Infrastructure

Copyright © 2018, Oracle and/or its affiliates. All rights reserved. This document is provided *for* information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0218

Whitepaper: Deploying a Highly Available Distributed Caching Layer on Oracle Cloud Infrastructure using Memcached & Redis

February 2018
Author: Abhiram Annangi