REDIS®* CACHE ON PRODUCTION: AN OVERVIEW AND BEST PRACTICES

In the modern landscape of complex applications, cloud-native technologies empower organizations to build and run scalable applications in public, private, and hybrid clouds.

An in-memory cache has become an essential component for developing loosely coupled systems that are resilient, manageable, and observable with microservices, containers, and immutable infra-services.

Unlike traditional databases, in-memory data stores don't require a trip to disk, reducing engine latency to microseconds, so they can support an order of magnitude more operations and faster response times. The result is blazing-fast performance with average read and write operations taking less than a millisecond and support for millions of operations per second.

Building on earlier experience using caching solutions like Infinispan and Hazelcast, we evaluated various cloud-based and on-premises cache solutions with the following requirements:

- Ability to scale out seamlessly, from a few thousand events per second to multimillion events
- Support for various data types and languages
- Performance metrics for monitoring
- Cache/entry-level time to live (TTL) support

Based on our findings, our BMC Helix SaaS solutions leverage Redis and Redisson Client for faster,

more accurate, and more efficient ways of delivering innovations for the modern enterprise. Redis, which is short for REmote DIctionary Server, is an in-memory cache data structure that enables low latency and high throughput data access.

If you're interested in deploying Redis at your organization, keep reading for some tips and best practices based on what we've learned from our deployment.

In-Memory Caching Service

First, you will need an in-memory caching service that supports Redis, such as one of the cloud and on-premises in-memory caching services below:

- Amazon ElastiCache
- Google Cloud Memorystore
- Azure Redis Cache
- Aiven
- Bitnami

Deployment Types

You should choose your deployment type based on your application use cases, scale, and best practices, while also considering factors such as number of caches, cache size, Pub/Sub workloads, and throughput.

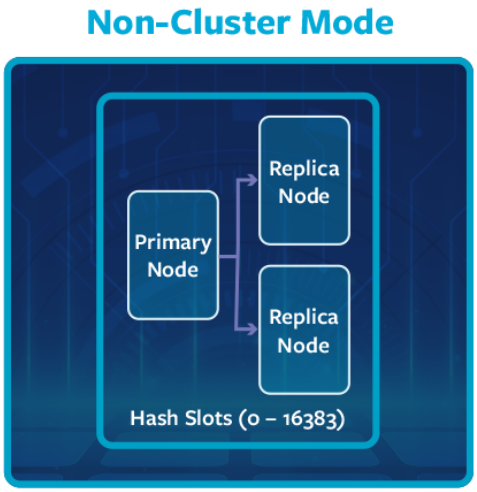

Figure 1. Non-Cluster deployment with single shard contains one primary and two replica nodes.

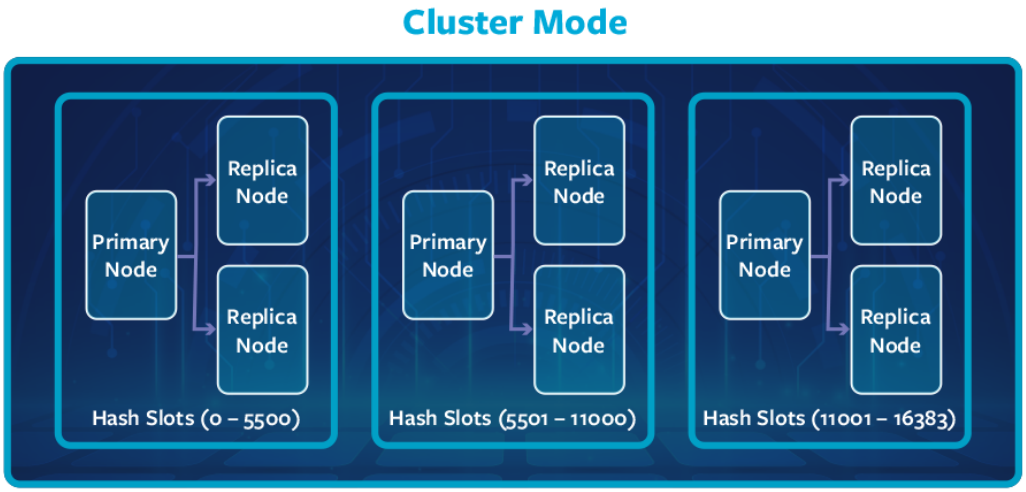

Figure 2. Cluster deployment with three shards and each contains one primary and two replica nodes.

Sharding

A shard is a hierarchical arrangement of one to six nodes, each wrapped in a cluster, that supports replication. Within a shard, one node functions as the read-write primary node, and all the other nodes function as read-only. Below are a few key points about individual shards:

- Up to five replicas per shard (one master plus up to five replica nodes)
- Nodes should be deployed in a shard on multiple availability zones or data centers for fault tolerance
- In case of a master node failure, one of the replicas will become the master

A production deployment may choose from three shards with three nodes per shard (one master and two replicas); each must reside on a different availability zone/data center. The cluster node type (CPU/memory) and Scale-out and Scale-in decisions are based on the cache types, size, and number of operations per second.

Every shard in a Redis cluster is responsible for a subset of the hash slots; so, for example, you may have a cluster with three replication groups (shards), as follows:

- Shard 1 contains hash slots from 0 to 5500
- Shard 2 contains hash slots from 5501 to 11000
- Shard 3 contains hash slots from 11001 to 16383

Redis Client

We zeroed in on Redisson after evaluating the available APIs based on the use cases and data structure requirements. It provides distributed Java data structures on top of Redis for objects, collections, locks, and message brokers and is compatible with Amazon ElastiCache, Amazon MemoryDB for Redis, Azure Redis Cache, and Google Cloud Memorystore.

Redis Client Key Usages

A streaming application that processes millions of metric, event, and log messages per second has various use cases that require low-latency cache operations, which informed our choice of cache type.

RMap is a Redis-based distributed map object for the Java ConcurrentMap interface that's appropriate for:

- Use cases where short-lived caches are required
- Eviction occurs at cache level and not at key/entry level
- Clarity exists on the probable cache size and max insert/retrieve operations

RLocalCacheMap is a near-cache implementation to speed up read operations and avoid network roundtrips. It caches map entries on the Redisson side and executes read operations up to 45 times faster compared to common implementations. The current Redis implementation doesn't have a map entry eviction functionality, so expired entries are cleaned incrementally by org.redisson.eviction.EvictionScheduler. RLocalCacheMap is appropriate for:

- Use cases where the number of cache keys is certain and won't grow beyond a certain limit
- The number of cache hits is high
- The workflow can afford infrequent cache hit misses

RMapCache is a cache object that supports eviction at key level and is appropriate for use cases that require that functionality and situations where ephemeral cache keys must be cleaned periodically.

Redis-based Multimap for Java allows you to bind multiple values per key.

Redis-based RLock is a distributed reentrant lock object for Java.

Monitoring Key Performance Indicators (KPIs)

The following KPIs should be monitored to ensure that the cluster is stable:

- **EngineCPUUtilization:** CPU utilization of the Redis engine thread
- **BytesUsedForCache:** Total number of bytes used by memory for cache
- **DatabaseMemoryUsagePercentage:** Percentage of the available cluster memory in use
- **NetworkBytesIn:** Number of bytes read from the network, monitor-host, shard, and overall cluster level
- **NetworkBytesOut:** Number of bytes sent out from a host, shard, and cluster level
- **CurrConnections:** Number of active client connections
- **NewConnections:** Total accepted connections during a given period

Redis Is Single-Threaded

Redis uses a mostly single-threaded design, which means that a single process serves all the client requests with a technique called multiplexing. Multiplexing allows for a form of implicit pipelining, which, in the Redis sense, means sending commands to the server without regard for the response being received.

Production Issues

As we expanded from a few caches to several caches, we performed vertical and horizontal scaling based on the above key metrics and the cost and recommendations. One critical issue we faced was a warning about high engine CPU utilization, although the application read-write flow was unchanged. That made the whole cluster unresponsive. Scale out and vertical scaling didn't help, and the issue repeated.

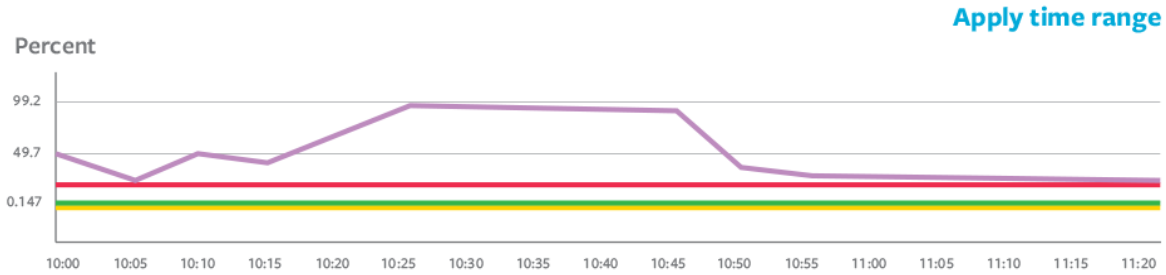

Figure 3. Engine CPU utilization of one of the shards that breached a critical threshold.

Troubleshooting Steps

- Get the big keys: use redis-cli –bigkeys to scan the dataset for big keys and get information about the data types within the dataset
- Review the slow log of slower queries
- Get the latency details with redis-cli –latency

Key Findings

- Publish and subscribe (pub/sub) operations were high on the problematic shard
- One of the hash slots had a large number of keys
- RMapCache seems to be the culprit

Issues with RMapCache

RMapCache uses a custom scheduler to handle the key-level TTLs, which triggers a large number of cache entry cleanups, resulting in huge pub/sub and making the cluster bus busy.

After a client publishes one message on a single node, this node will propagate the same message to other nodes in the cluster through the cluster bus. Currently, the pub/sub feature does not scale well with large clusters. Enhanced input and output (IO) is not able to flush the large buffer efficiently on the cluster bus connection due to high pub/sub traffic. In Redis 7, a new feature called sharded pub/sub has been implemented to solve this problem.

Lessons Learned

1. Choose cache types based on usage patterns:
    - Cache without key-level TTL
    - Cache with key-level TTL
    - Local or near cache

For a cache with key-level TTL, ensure that the cache is partitioned to multiple logical cache units as much as possible to distribute among shards. The number of caches may grow by a few thousand without an issue. Short-lived caches with cache-level TTL are an option.

2. While leveraging the Redisson or other client implementations on top of Redis, be careful with the configuration and impact on the cluster.
   Ensure that the value part is not a collection (if a collection is unavoidable, limit its size).
   Updating an entry on the collection value type has a large impact on the replication.

Conclusion

Looking to provide a real-time enterprise application experience at scale? Based on our usage and experience, we recommend that you check out Redis along with the Redisson Client.

Experience it for yourself with a free, self-guided trial of BMC Helix Operations Management with AIOps, a fully integrated, cloud-native, observability and AIOps solution designed to tackle challenging hybrid-cloud environments.