# Optimisations on AWS Elastic Cache (Redis) Usage

Rahul Banerjee

***Abstract*—AWS elastic cache (Redis) is one the most popular mechanism for cloud developers to achieve both excellent caching as well as persistency of data. As part of our recent project of building personalised ad decision system, we needed to support stringent performance requirement of 125000 requests per second and 95% of requests must be served under 250ms. We used Elastic cache (Redis) for both caching and persistency and found number of optimisation techniques which helped us to achieve the performance requirement. These techniques we used ensured that cost for Elastic cache (and the whole project) is much lower than initial estimate. The techniques involved choosing appropriate CPU instances for Elastic cache, using right cache structure and whole host of changes on Redis client side to make this end-to-end system high performance, and cost effective.**

***Index Terms*—AWS elasticcache, computer science, distributed computing, redis, software architecture and design.**

## I. Problem Statement

We are building a personalised ad decision system (i.e., for a given request from an OTT subscriber, system will decide which ad is most suited for him/her in this given ad break) which uses AWS Elastic cache (Redis) for caching important pieces of information. These set of information helps system to decide most suited Ad for requester. For every request, ad decision system needs to make several Redis calls (both GET and SET).

AWS Elastic cache [5] is quite expensive solution as can be seen at pricing page [1]. With the stringent requirements for Ad decision component in terms of transactions per second (needed to support 125000 requests per second) and performance (95% of requests must be completed by 250 ms), continually scaling up elastic cache infrastructure to meet performance requirement would have been extremely expensive. During testing we also figured out having a very large ElasticCache module was not meeting our performance requirement!

Also, during implementation phase, new requirements started flowing in and few of them ensured that access to Redis cluster need to be serialized (mutually exclusive). This would further have adverse impact on performance.

## II. Environment

Component software was written in JavaScript (Node version v14.15.0), and library used for interaction with Redis cluster is ioredis [2].

Redis cluster is part of AWS ElasticCache ecosystem. Following is specification of cluster

- Redis engine version 6.0.5
- Number of shards: 1
- Number of replicas: 2
- Multi AZ support: enabled

Component software works from Kubernetes pods which are hosted on EC-2 machines in EKS cluster. Redis cluster and pods reside within same VPC.

The data being stored into Redis cluster are always JSON document (key value pairs).

## III. Solution

Before we start discussing solution, a brief introduction to Redis. Redis [14] is an open source (BSD licensed), in-memory data structure store, can be used as a database, cache, and message broker. Redis provides data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes, and streams. Redis has built-in replication, Lua scripting, LRU eviction, transactions, and different levels of on-disk persistence.

Redis key space is split into 16384 slots, effectively setting an upper limit for the cluster size of 16384 master nodes (however the suggested max size of nodes is in the order of ~ 1000 nodes). Each master node in a cluster handles a subset of the 16384 hash slots. The base algorithm used to map keys to hash slots is the following

HASH_SLOT = CRC16(key) mod 16384

There are two main parts of system that was looked while developing solution

- Optimising Redis cluster
- Optimising access to Redis cluster

Following sections describe these in more details.

### A. Optimising Redis Cluster

First and foremost, it is very important to understand what type of data (size, both for individual entry and cumulative) is being stored and how often it will be accessed (it is also important that access is get or set). Also, with the requirement of mutual exclusion on data usage for certain type of data has triggered some curious discussion. Is mutual exclusion required for all the types of data, or we can get away with isolating the data requiring mutual exclusion to isolate?

Amazon supports many CPU instance types [6] to run elastic cache cluster on. CPU instance types pertain to different categories, some being optimized for memory (R series), some are fine tuned to maximize CPU performance (C series), and some are more applicable for general purpose computing (M series). From reading of AWS documentation [9] and experimenting it was clear that we needed to use memory optimized CPU instances for Redis cluster. AWS Memory optimized CPU instances are useful for following

Manuscript received September 1, 2021; revised October 28, 2021.

Rahul Banerjee is with Principal Engineer, Synamedia, Bangalore, India (e-mail: rahulbec@gmail.com).

use cases

- High-performance, relational (MySQL) and NoSQL (MongoDB, Cassandra) databases.
- Distributed web scale cache stores that provide in-memory caching of key-value type data (Memcached and Redis).
- In-memory databases using optimized data storage formats and analytics for business intelligence (for example, SAP HANA).
- Applications performing real-time processing of big unstructured data (financial services, Hadoop/Spark clusters).
- High-performance computing (HPC) and Electronic Design Automation (EDA) applications.

Once we have decided memory optimized CPU instances are way to go ahead, it was clear that we needed to use the latest instance series R6g for best performance. Now the next and most important question was how do we structure the cluster such that it is optimized both in terms of cost and performance?

We have classified data being stored into three different categories as following

- Large and most frequently read and written. Size of data hosted in this cluster would increase over the time. On average the size of value associated with each key is highest here. Frequency of access to this data can be millions of times per second during prime-time TV viewing. So, cluster responsible for hosting this category data must be able to support high network bandwidth throughput and must include CPU capable of handling and disposing such high number of requests. Also, the cluster must have sufficient storage to persist data. Mutual exclusion needs to be applied for this category of data.
- Large (but smaller than previous category), frequently read but written only once per day. Size of value associated with each key remains constant and moderate. So, cluster hosting this category data requires moderate storage, CPU requirement is lower than the previous category data and storage required by this category data is much lesser (going to by calculation, can go up to 50 Gbytes in average case). Also, no mutual exclusion deemed necessary for this type of data.
- Small and relatively less frequently read (say once in few minutes) but written not so often (say once every few hours). Size of data set remain relatively constant. Thus, for cluster hosting this category of data requires moderate network bandwidth, CPU, and storage requirement.

At this point, it was clear that instead of trying to find single elastic cache cluster to meet requirements for above mentioned data categories, it is better to use specifically tailored cluster for each category of data. Based on these three categories, we have created three different clusters with different Redis node types (CPU instance type).

- For the first category, using node type r6g.16xlarge (largest and most expensive node type allowed today on elastic cache)
- For the second category, using node type r6g.2xlarge
- For the third category, using node type r6g.xlarge

Capabilities of each of the chosen elastic cache node type (and on demand pricing for it) have been depicted at following table. These details are as per AWS page [1] and for EU-WEST-1 (Ireland) region (pricing do change over different AWS regions)

| Node type | Number of virtual CPUs | Memory | Network performance | Price per hour |
|---|---|---|---|---|
| cache.r6g.16xlarge | 64 | 419.09 GBytes | 25 Gbps | USD 7.327 |
| cache.r6g.2xlarge | 8 | 52.82 Gbytes | Up to 10 Gbps | USD 0.916 |
| cache.r6g.xlarge | 4 | 26.32 Gbytes | Up to 10 Gbps | USD 0.458 |

While considering the cost, need to remember that to maintain high availability, best (and AWS recommended) practice is to have one main node and two replicas pertaining to different availability zones. We are following this principle, and thus the cost seen on last column must be multiplied by 3 to come up with actual cost. (of course, we are planning to use reserved instances on production deployment and thus cost is somewhat lower to hat is quoted on table).

Basically, now we have optimised cluster available for each type of data (and its access) from both cost and memory/network requirement perspective.

Instead of this segregation, if we maintained a single cluster for all types of data, the cluster would have to be of node type r6g.16xlarge and had to horizontally scale (i.e., to add more shards. Each shard means basically 3 more nodes), which would have solution extremely expensive.

For each of the cluster, if in an unforeseen scenario, if we run out of storage space, we plan to do horizontal scaling (i.e. adding more shard).

Using this right cluster strategy, node type and latest Redis version (version 5) helped us to reach the best possible performance and cost strategy that can be achieved within the ambit of requirements.

In future, we plan to keep looking at upcoming new node types (and Redis versions) made available by AWS for elastic cache, which hopefully helps us to achieve more bang per buck!

### *B. Optimising Access to Redis Cluster*

One of the most important optimizations we considered was to reduce payload size associated with each Redis access. Reduced payload size ensures that less memory copying happening between user and kernel space (both on EC-2 machine and Redis cluster size) as well as using lesser network bandwidth between pods and cluster. To achieve this, we have decided that most frequently accessed and large payloads in and out of Redis cluster would be compressed using zlib. We have chosen zlib because it provided a right amount of balance between CPU usage and compression ratio. Using compression also helped us more efficient usage of elastic cache cluster storage space (thus improving performance while reducing cost!). We have considered using zstd and brotli algorithms for compressing data. While these algorithms provided better compression ratio, but they are more CPU hungry and thus was reducing transaction per seconds for each pod.

Pipelining is something Redis documentation [4] highly

recommends. Creating pipeline of multiple commands and then sending them to cluster on single TCP connection produced big performance gain. With this round-trip time for each command reduces significantly. Pipelining is not just a way to reduce the latency cost associated with the round trip time, it greatly improves the number of operations you can perform per second in each Redis server. This is the result of the fact that, without using pipelining, serving each command is very cheap from the point of view of accessing the data structures and producing the reply, but it is very costly from the point of view of doing the socket I/O. This involves calling the read() and write() system call, that means going from user land to kernel land. The context switch is a huge speed penalty. When pipelining is used, many commands are usually read with a single read() system call, and multiple replies are delivered with a single write() system call. Because of this, the number of total queries performed per second initially increases almost linearly with longer pipelines, and eventually can reach 10 times the baseline obtained without pipelining.

In ioredis, pipelining can be enabled by setting enable Auto Pipelining option to true. In auto pipelining mode, all commands issued during an event loop are enqueued in a pipeline automatically managed by ioredis. At the end of the iteration, the pipeline is executed and thus all commands are sent to the server at the same time.

Another very important aspect to achieve good performance is how to achieve appropriate balance of load between master and replica nodes within cluster. This can be done by directing read queries to replica nodes, while write commands always gets processed by master node on cluster. With this change, requests (and thus the network traffic in and out of nodes) are more balanced between nodes within cluster, and stops a single node being overworked (both in terms of CPU and network bandwidth usage) in high load scenarios, and potentially can save us from scaling cluster up (thus reducing cost)! Once this change is made, even on highest load scenario, we always have found that all nodes within this cluster are well within range of CPU and network bandwidth usage.

Important thing to remember here is that master node asynchronously copies that latest written data to replica nodes, and it typically takes few seconds to propagate the latest change for a key's value from master to replica. While going for this option, data propagation delay from master to replica nodes and its impact must be considered. For our use cases, this was never a problem

In ioredis, this feature can be enabled by setting option scaleReads to "slave".

Some other implementation specific optimisations we used are as following

- Combining multiple Redis requests to one is another change that was made to ensure lesser interaction with cluster (and as explained earlier, much lesser number of system calls). This again reduces the overheads associated with socket I/Os and reduces/removes round-trip time of making several calls. This required changes in the way data was structured, but again amounted to considerable performance improvement.
- Since the data being stored into Redis here was always in form of JSON document, shortening the key name lengths reduced payload size significantly.
- For compressed payload, do not further serialise the buffer (i.e., don't use JSON.stringify). Redis cluster can store base-64 buffer, and this ensures payload size is lower (vis-a-vis JSON stringified version of compressed data).

With all the above optimisations, we achieved our performance objective with fraction of initially anticipated cost! So, it was really win-win situation.

### *C. Mutual Exclusion on Redis Cluster Access*

However, one more challenge joined late in the party is need for mutual exclusion to maintain data coherency across multiple read/write cycle. Maintaining data coherency on a distributed system like ours in Redis can be quite challenging. Redis proposes a detailed and failproof mutual exclusion mechanism as describe its website [7].

Crux of it is as follows

- Before accessing a particular key (k), create another unique key (u_k) associated with it and assign a unique value to it. All Redis clients accessing the cluster must successfully create (using setnx command) this unique key (u_k) (akin to acquire lock) before accessing key (k). So, there shall be one-to-one mapping between k and u_k.
- However, value being set by each client to u_k shall be unique for a given client and that shall not match with any other client dealing with u_k.
- So, let's say two clients A and B trying to access key k. Thus, both would try to create key u_k with setnx command. setnx command ensures if the key is already present, it will fail, else it will create the key and set the requested value. Assuming client A tried first, and creates key u_k, B's attempt to create u_k would fail, and B needs to wait till it can acquire the lock (i.e., create u_k and set its own value using setnx command). A will thus complete required work with key k and then it reads the value of key u_k and assuming it sees the value is same as what it set during create, would delete key u_k. B then can create u_k and proceed. While creating u_k, a time to live (ttl) would be associated with this key.
- What happens if for some reason, client A is stuck (or worse crashed) post locking u_k? Well, post the expiry of u_k's time to live, client B will be able to create u_k (thus acquire lock and gets access to key k).
- Another use case is let's say client A has acquired lock (i.e., created u_k and set value A1 to it for example) and then it got stuck while some processing and time to live associated with u_k is over. So, in this case client B will be able to create u_k (thus acquire lock, and set value say B1 to u_k) and thus access key k. Now while B is busy working post acquiring lock, say A get invoked again, and it tries to delete u_k. If it can delete u_k, then it will create number of further use cases of problematic for mutual exclusion! This is where unique value for u_k for each client comes handy! Before deletion, A would read the value and check if value equals to A1. If not, A will figure out it does not hold the lock anymore and thus would not delete the key.

So above theory about Redis lock mechanism ensures that there is no way a client will remain stuck forever while waiting to acquire lock. At the same time, it ensures a very good level of mutual exclusion.

In our project it became quite a challenge as now for every access to Redis cluster, we need to issue few extra Redis calls to achieve mutual exclusion which has serious performance impact.

We tried with official NodeJS package [13] node redlock recommended by Redis distributed lock [7], but it increased average transaction/request time on load scenarios to 4 times without it! So clearly, we could ill afford it.

While going over internet, we could see a paper from Alibaba [8], which also echoed the same finding as ours, but they solved it using their proprietary wrapper over Redis, thus not generic solution (i.e., who are not using this proprietary solution) for others.

Also, we tried with Lua script comprising of several Redis command to make it atomic did not get us to desired performance level.

Finally, we have fixed this with pipelining! Since multiple client's accessing same key is relatively rare scenario, we decided to go for least penalty approach for average use cases. We structured the keys (k and u_k) in such a manner that both will be pertaining to same Redis hash slot. Once this is done, we can create a Redis pipeline comprising of following

- setnx command for key u_k (which will create u_k and set specific value to u_k).
- get value of key K.

If first command on pipeline (setnx command) indicates that u_k already exist, we ignore value returned on second step, and wait for setnx command to succeed (by periodic retries). However, it most cases this is not the case, and thus when pipeline returns, we are good to go!

With this change, we have coupled both set of Redis calls under same command. Since we anyway needed to get value of key K for every request, we achieve most important part of mutual exclusion with almost no extra cost! This helped us to achieve performance goals with mutual exclusion on maximum load scenario!

For pipelining to work, keys being subject to command on pipeline must pertain to same Redis hash slot. Details about Redis hash slot can be found on Reference [10]. To achieve this, we needed to change key naming pattern within our software for key, so that both key and corresponding lock key (for example u_k referred earlier in this document) could remain in same hash slot.

## IV. Conclusion

Most of the principles described above shall be applicable for any application working with ElasticCache or for that matter with Redis engine outside ElasticCache too. Things like using pipelining, choosing right node type for the cluster (only applicable for ElasticCache), reducing payload associated with keys, reducing number of calls to Redis Engine shall be always used while interacting with Redis. In fact, reducing payload size associated with system calls and reduction of system calls are surefire optimization technique on any Linux system, irrespective of if it is large distributed system running on cloud or a tiny, embedded software running on small local device.

Having said that, understanding the nature of data being stored to Redis and how frequently it is accessed is a key thing. If for an application only one type of data is being stored, there is absolutely no need to have different clusters.

Similarly, while reading from replica node has, its own benefit, if an application needs the value of a key immediately after writing (and this value need to be latest), reading from replica is not recommended! In such scenario, it is better to direct read and write commands to master node.

One of our findings about mutual exclusion is that it might be better to define a key naming structure which allows achieving mutual exclusion through pipelining mechanism as explained earlier. If this provision is not kept, and requirement for mutual exclusion comes later in software development lifecycle (and in worse case post the solution is in production), we may end up a scenario where the old keys (named with existing scheme) stored on cluster is not compatible with required new scheme! It is a very difficult position to be in!

While this paper solely focuses on the changes in and around Redis usage from software perspective which helped us to achieve the target performance, there are few other aspects that we optimised and got good results too. For example, using right Kubernetes worker CPU type (on which our software runs) for the job in hand really added great value. AWS provides a series of compute optimized node types [12], which we found particularly fit for our purpose. As always, while deciding a particular CPU instance type, optimization needed to achieve both in cost and performance side.

For our use case, we needed node types, which includes compote optimized CPUs with high network throughput (as it interacts largely with Redis cluster almost throughout its lifetime). With large amount of testing, we have found node type c5a was sufficient to achieve required performance.

Now that we are very close to production deployment, we do see that most of the cost (around 75%) of our system stems from Redis cluster and worker nodes (on which our software runs). So, squeezing every bit of performance while always being aware of cost aspect really helped us to not only create a system which meets every stringent performance requirement but also remain a reasonably cost-effective solution.

## Conflict of Interest

The author declares no conflict of interest

## References


[1] Amazon ElasticCache Pricin. [Online]. Available: https://aws.amazon.com/elasticache/pricing/

[2] Ioredis documentation. [Online]. Available: https://github.com/luin/ioredis

[3] Right sizing Elasticcache Cluster. [Online]. Available: https://aws.amazon.com/blogs/database/five-workload-characteristics-to-consider-when-right-sizing-amazon-elasticache-redis-clusters/

[4] Redis Pipelining. [Online]. Available: https://redis.io/topics/pipelining

[5] Amazon Elastic Cache. [Online]. Available: https://aws.amazon.com/elasticache/

[6] Amazon elastic cache supported node types. [Online]. Available: https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/CacheNodes.SupportedTypes.html

[7] Redis Distributed locking. [Online]. Available: https://redis.io/topics/distlock
[8] Redis lock use case. [Online]. Available: https://www.alibabacloud.com/help/doc-detail/146758.htm
[9] AWS memory optimized instances. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/memory-optimized-instances.html
[10] Redis cluster spec. [Online]. Available: https://redis.io/topics/cluster-spec
[11] Kubernetes. [Online]. Available: https://kubernetes.io/
[12] AWS EC2 compute optimized nodes. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/compute-optimized-instances.html
[13] Node redlock. [Online]. Available: https://github.com/mike-marcacci/node-redlock
[14] Redis IO. [Online]. Available: https://redis.io/



**Rahul Banerjee** was born at Shibpur, Howrah, West Bengal, India on 17th September 1981. He has completed his Bachelor of Engineering on information technology from Bengal Engineering College (currently known as Indian Institute of Engineering Science and Technology) located at Shibpur, Howrah, India.

He is currently working at Synamedia Bangalore as Principal Engineer. Prior to Synamedia, he has worked with Wipro, Marvell Semiconductor and TATA ELXI Limited.