Indian Institute of Technology, Kanpur

UGP Report

Auto Scaling of Key Value stores

Author:
Raghukul Raman

Supervisor:
Dr. Debadatta Mishra

A report submitted in fulfillment of the requirements for Under Graduate Project (CS396A)

Department of Computer Science and Engineering

Abstract

In the modern day computing where application response is a critical metric, in memory databases are gaining popularity. In-memory databases primarily use main memory for data storage. These databases are becoming a crucial part of many applications due to the availability of less expensive RAM, and the demand for high response time. Some of the cases where in-memory databases perform exceptionally well are Queues, sticky sessions, caching, real-time data analysis, etc.

Generally, these in-memory databases are implemented as key-value stores; some well know key-value stores are Redis, Memcached, Hazelcast, etc. Large scale caching can be done by scaling the instances of these databases. Dynamic scaling of instances can provide cost-effective key-value storage services.This project aims at analyzing and providing an auto-scaling solution to this problem.

We try to collect statistics for different queries on different node configurations and try to find possible bottlenecks in the cluster based system. We analyze by scaling redis horizontally as well as vertically. For analyzing clusters we use an open source tool called Twemproxy (a fast, light-weight proxy for Memcached and redis). We limit resources and measure performance for these configurations. We also try to build a system which can be used to interact with the cluster and efficiently shard the requests among different instances.

Introduction

Key Value Stores

Key value stores are a special kind of database management system, where the data is organized in just 2 columns - a key, and a value. Here the key is mostly text, while the value can be anything and is simply an object. The actual data essentially becomes "value", which is indexed with "key", so whole data becomes schema-less and is just organized in just 2 columns. One important property is that since the data is indexed by just the keys, there is a lot of scope of distributing keys over different instances, making the system highly scalable.

These kind of databases fall into the category of nosql data stores, satisfying the BASE properties (Basically Available, Soft state, Eventually consistent). In the contezt of the CAP theorem (Consistency, Availability, Partition tolerance), these databases fall into the category of CP. Since these databases are generally implemented in cluster mode, we cannot gurantee availability on data.

These databases are generally in-memory which reside in main memory. In-memory databases are faster than disk-optimized databases because disk access is slower than memory access, the internal optimization algorithms are simpler and execute fewer CPU instructions.

Scalability

Scalability is the property of a system to handle a growing amount of work by adding resources to the system [**?**]. In layman terms, when you realise your system is getting slow and is unable to handle the current number of requests, you need to scale the system. Scaling can be done in two ways:

- **Vertical Scaling:** Increasing the resources in the server which we are currently using, i.e increase the amount of CPU, GPU, CPU, etc. Vertical scaling generally costly, also it may require the system to go down for a moment when the scaling takes place (need for downtime).
- **Horizontal Scaling:** Increasing the number of servers (instances). This would lead to decrease in the overall load on the system, if requests are sharded properly. This kind of scaling solution is typically popular in tech industries, as it makes the system fault tolerant (single point of failure reduced). Important advantage of this method is that it can provide administrators with the ability to increase capacity on the fly.

Redis

Redis project started by Salvatore Sanfilippo(*antirez*) is an open source, in-memory data structure store, used as a database, cache and message broker [**?**]. Apart from basic key-value structure it also supports other data structures like: strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes with radius queries and streams. Redis has built in support for replication, on-disk persistance and automated partitioning with Redis cluster. Redis also supports trivial-to-setup master-slave asynchronous replication, with very fast non-blocking first synchronization, auto-reconnection with partial resynchronization on net split [**?**]. There is also support to run these operations in atomic fashion with the help of transactions.

One intruging property of redis which makes it different from other classical key value stores is: it gives support to mention expiration time for keys. This features comes handy when we are using redis for cache purpose, olders keys would be deleted automatically.

Redis Cluster

Redis cluster can be defined as a collection of redis nodes which are able to communicate among themselves, and are able to respond to requests collectively. Redis Cluster provides a way to run a Redis installation where data is automatically sharded across multiple Redis nodes. Redis Cluster also provides some degree of availability during partitions that is in practical terms the ability to continue the operations when some nodes fail or are not able to communicate [**?**].

Every redis cluster node require two TCP ports, the lower one is used to server clients, while the higher one(lower port+10000) is used for Cluster bus (node-to-node communication). Redis cluster use CRC16 hashing system, using which every key is hashed to becomes part of one of 16384 hash slots (these are like buckets which contain keys, after hashing). So each cluster node is responsible for a subset of hash slots, that is keys which hash to a particluar hash slot, would be found in the node serving that hash slot. The user can force multiple keys to be part of the same hash slot by using a concept called hash tags.

Creating a Redis Cluster

To create a redis cluster we need to run the required number of instances of redis (on different ports), and then inform all these nodes about cluster meet. A typical redis config file for cluster node is:

```
redis.conf
port 6379
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
```

After running all the required number of instances(with different ports, if running on same machine), we need to run:

```
$ redis-cli --cluster create 127.0.0.1:7000 127.0.0.1:7001 \
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005
```

Here the IPs mentioned are the host and ports of the redis instances. After running this command we are asked about the hash slot distribution, to be configured among the nodes. And that's it, all the node share infomation about cluster, and other nodes via messaging protocol. So after this process, each node will have a hash slot to node (IP:PORT) mapping, and all other cluster informations like node timeout etc.

Redirection and Resharding

We assume that a client know nothing about the distribution of hash slots in the cluster, so it is equally probable to send a request to any of the node. On recieving a request, the redis node will find the hash slot to which this key belongs, if the hash slot is served by current node, it would simply return the value to the client. Is case hash slot is not served, it would check its internal hash slot to node map, and would respond to the client with an MOVED error, along this error message it would also send the IP of the node serving hash slot to which this key maps. This is pretty handy since client doesn't have to hit and try every node, it would get the correct address after one try. Still there is an overhead, since we need to send a request twice (in worst case). We'll come to this issue in more detail in the partitioning section.

Redis Cluster supports the ability to add and remove nodes while the cluster is running. Adding or removing a node is abstracted into the same operation: moving a hash slot from one node to another. This means that the same basic mechanism can be used in order to rebalance the cluster, add or remove nodes, and so forth. Moving slots in running cluster is not an issue, since if the hash key is present in current node, it would simply return the value, in case that particular key is moved to another node, it would respond with an moved error. Hence resharding can we done in redis cluster without any down time.

Partioning

Partioning refers to the way how we shard data among different nodes of redis cluster. Note that redis cluster provide a sharding scheme based on CRC16, but we can make some custom modification in the sharding scheme at different levels, these are:

1. **Client side partitioning**: In this scheme, redis clients select the node to which read/write request need to be made.
2. **Proxy assisted partitioning**: This scheme suggests that client sends request to a proxy, which analyzes the request and forwards it to the correct node. On recieving response it then returns it to the client. eg: Twemproxy
3. **Query Routing**: We can send request to any node, and the node will forward our request to the correct node. Redis cluster implements a similar kind of scheme, where instead of forwarding, it returns the address to which we should forward.

Among these partioning schems proxy assisted partitioning are most popular.

Twemproxy

Twemproxy is a proxy assisted partitioning implementation, developed for redis and memcached. Note that this proxy is not a single point of failure, since we can run multiple instance of the proxy, and the client would try to connect with them sequentially on unsuccessful connection it would try to connect to the next intance. It maintains persistent server connections, enables pipelining of requests and responses, shard data automatically across multiple servers, keeps copy of node configurations [**?**].

Twemproxy also has support for data collection i.e stats exposed on the stats monitoring port (can be configured). To configure twemproxy for a cluster of nodes we need to provide with a YAML config file like:

conf

```
listen: 127.0.0.1:22122
hash: fnv1a_64
hash_tag: "{}"
distribution: ketama
auto_eject_hosts: false
timeout: 400
redis: true
servers:
 - 127.0.0.1:6380:1 server1
 - 127.0.0.1:6381:1 server2
 - 127.0.0.1:6382:1 server3
 - 127.0.0.1:6383:1 server4
```

Stats are by default exposed on port 22222.

Experiments

For collecting stats about performance of redis/redis cluster by limiting resources, we generate random strings of length 64 as key (to mimic sha2, hashed keys - 256 bit sized), and the values are ranged from 1 to 128 character long. Most of these experiments are performed by running redis images on docker, since docker provides elegant way to limit resources of a container. Since get/set are the most basic commands used, most of our tests are based on these commands (other commands are indeed build on these commands).

Running redis cluster on docker

To run a redis cluster in docker containers we need to run all the redis instances in the same network. For this we first need to create a bridge in docker:

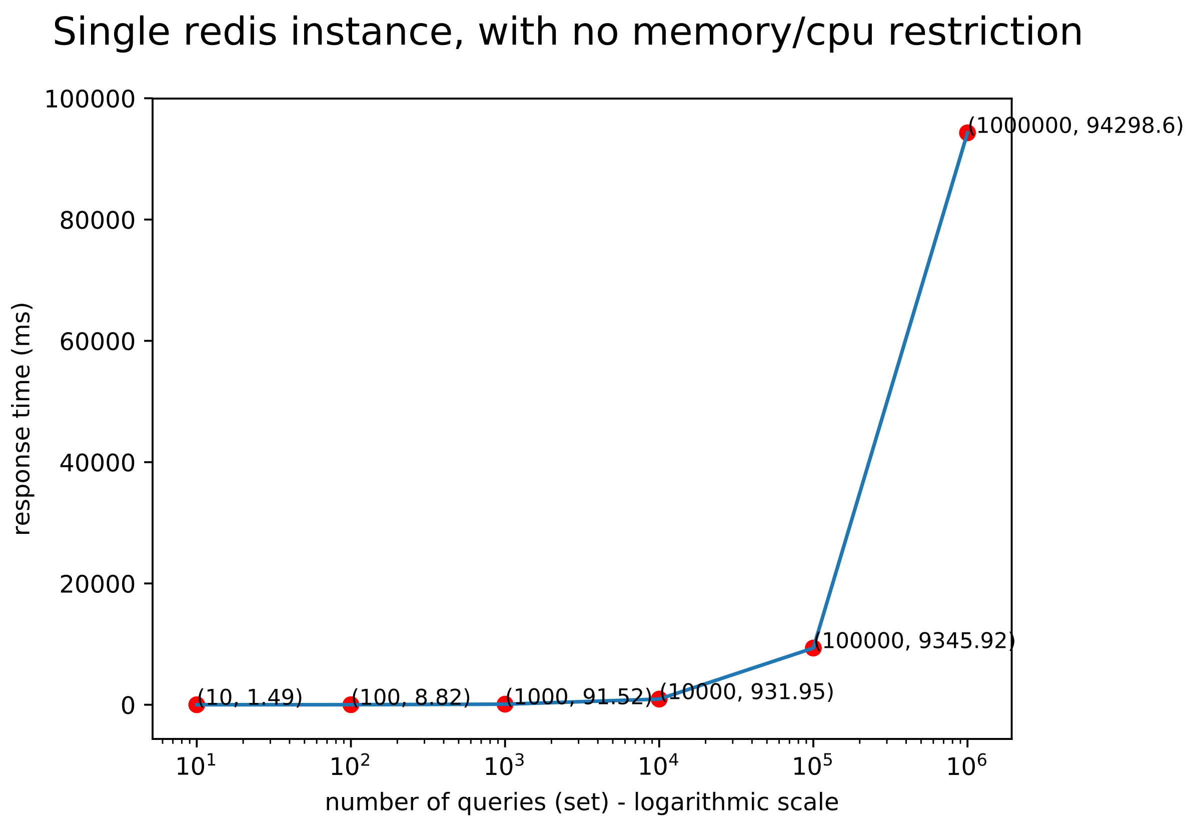
```
$ docker network create redis_cluster
```

after creating the network, we need to run each of the instances in this network. One important thing to note is that we need to give the default port (6379) in all of them, this is because docker network, itself allocated unique IP:PORT to each of the container in the network.

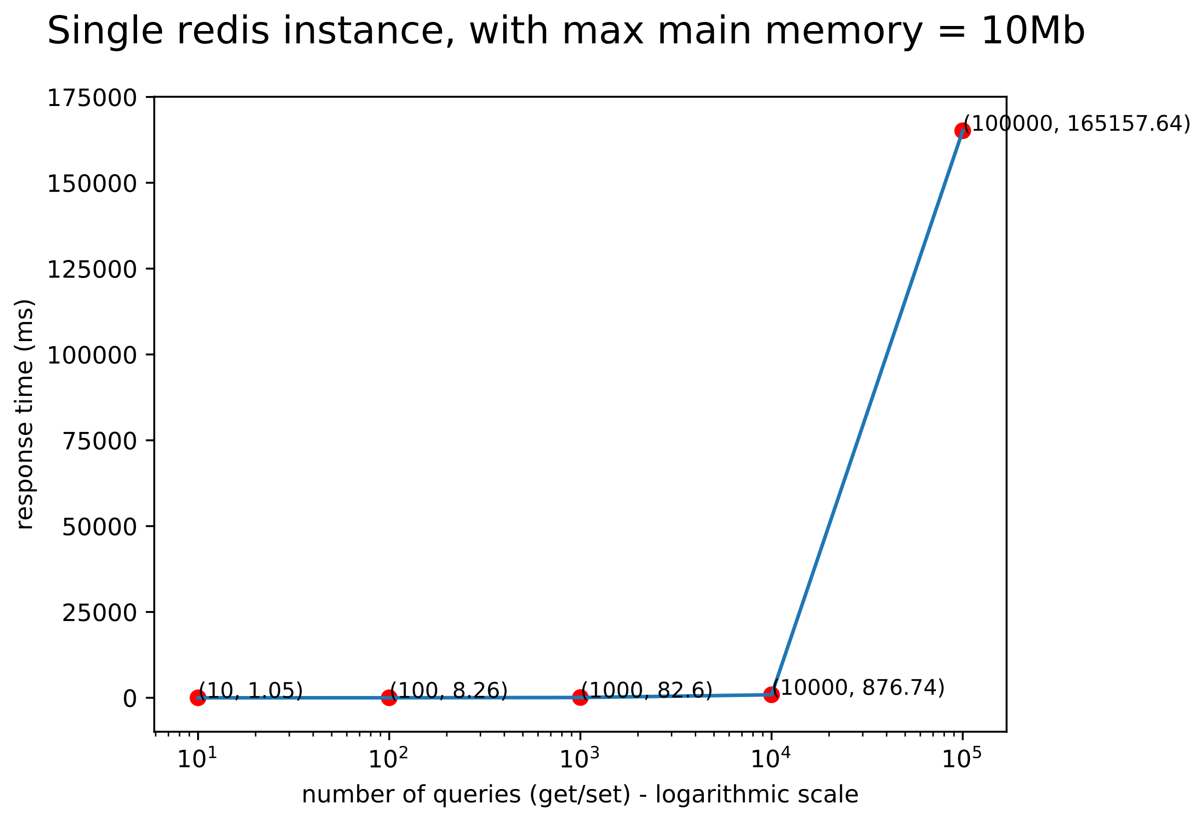
```
$ docker run -d -v --net redis_cluster redis \
    redis-server /path/to/config/$file
```

After this step, we just need to inform all the nodes to form cluster (can be done with redis-cli, as given in the introduction section for creating a redis cluster).

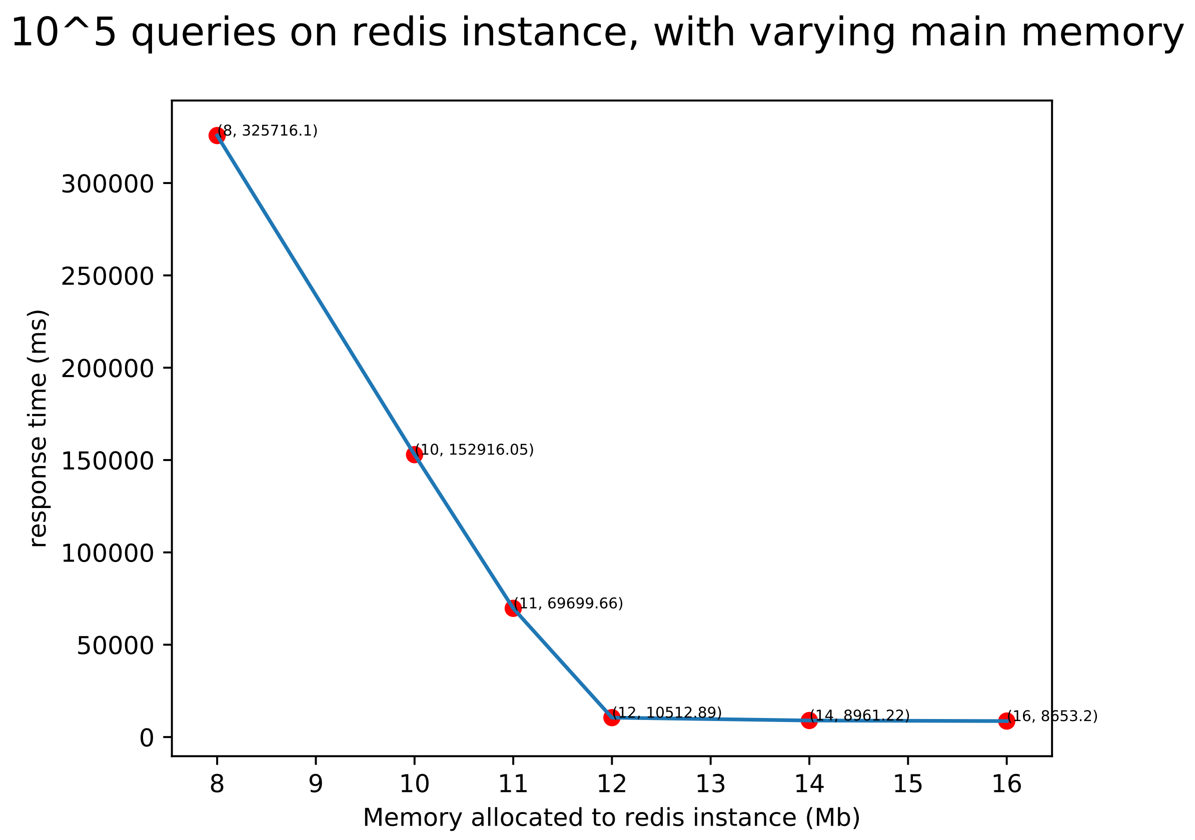
Single redis instance



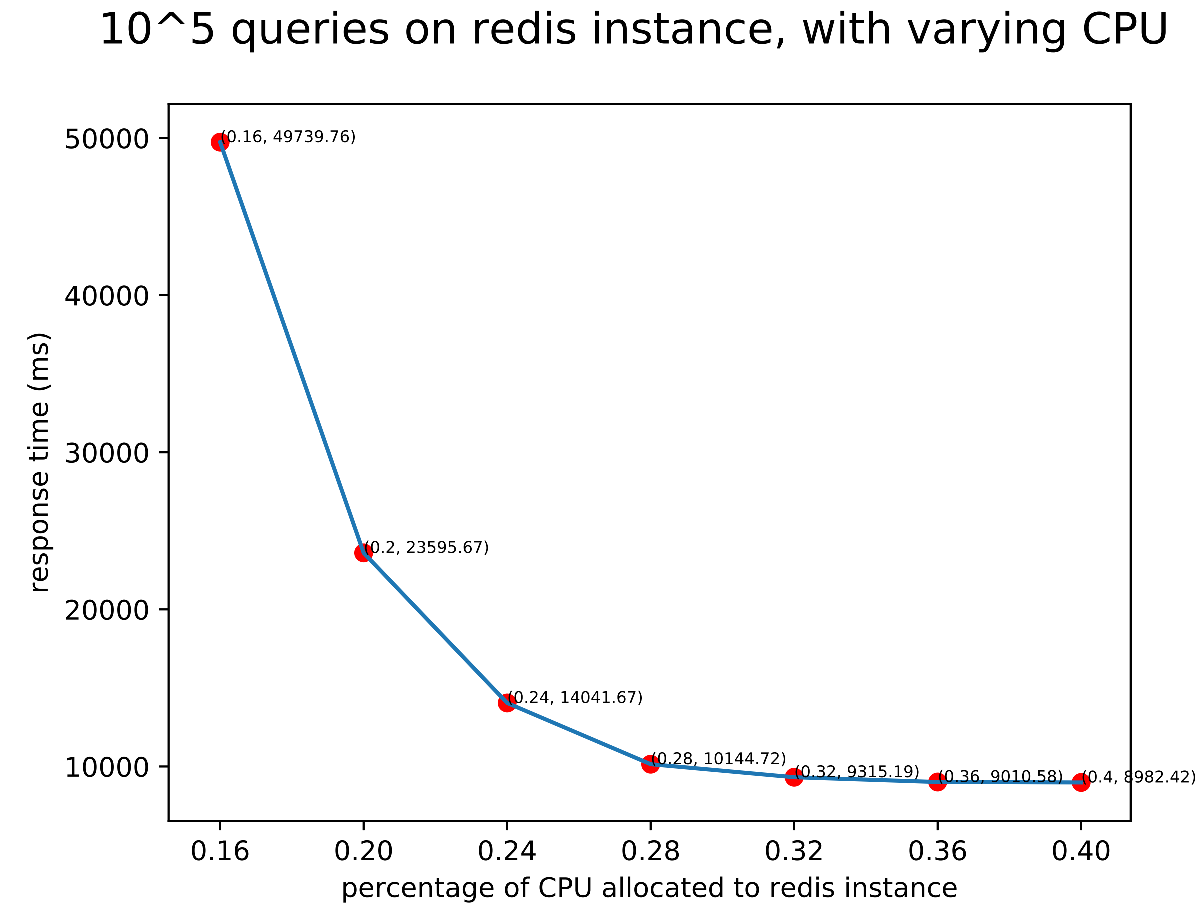
This graph is on logarithmic scale, on a linear scale we observe a linear relation between the time and number of queries.



In the above experiment, we limit the main memory supply of the redis instance to $10Mb$. Comparing this graph with the first graph, we can see that time taken increases drastically for $n = 10^5$ queries



In this experiment we try to capture the effect of main memory on the performace of redis. We can see that increasing memory, lineary decrease the response time, which is quite intutive since the pages swaped out would decrease lineary if we increase memory supply.



We cannot have fractional CPUs, above values just signify cpu period, and the cpu quota. For example cpus="1.5", means that the node is guaranteed at most one and a half of the CPUs, which is the equivalent of setting cpu-period="100000" and cpu-quota="150000"

Unlike main memory CPUs kind of affect the response time non linearly. Above figure suggests that response time quadratically depends on the cpu period/quota.

Redis Cluster experiments

References

[1] Redis documentation, `redis.io`

[2] Bondi, Andre. *Characteristics of scalability and their imapct on performace.* WOSP, 2000.

[3] Twemproxy, developed by twitter, `github.com/twitter/twemproxy`