

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

UGP REPORT

---

## Auto Scaling of Key Value stores

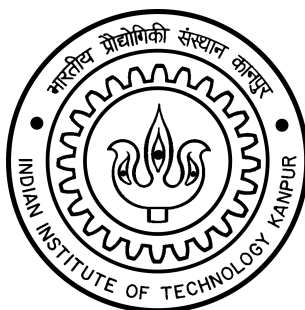
---

*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, lightweight 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 context 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 guarantee 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 [2]. In layman terms, when you realize 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 memory, 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 the 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 [1]. Apart from the 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 persistence 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 [1]. There is also support to run these operations in atomic fashion with the help of transactions.

One intriguing property of redis which makes it different from other classical key-value stores is: it gives support to mention expiration time for keys. This feature comes handy when we are using redis for cache purpose, older 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 [1].

Every redis cluster node requires two TCP ports, the lower one is used to serve 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 particular 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 the 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 the 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 information about the 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 information like node timeout etc.

### Redirection and Resharding

We assume that a client knows 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 receiving a request, the redis node will find the hash slot to which this key belongs, if the hash slot is served by the 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 a MOVED error [1], along with 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 the 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 the 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 be done in redis cluster without any downtime.

## Partitioning [1]

Partitioning refers to the way how we shard data among different nodes of redis cluster. Note that redis cluster provides 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 needs to be made.
2. **Proxy assisted partitioning:** This scheme suggests that the client sends the request to a proxy, which analyzes the request and forwards it to the correct node. On receiving response, it then returns it to the client. eg: Twemproxy
3. **Query Routing:** We can send the 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 partitioning schemes 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 instances of the proxy, and the client would try to connect with them sequentially on the unsuccessful connection it would try to connect to the next instance. It maintains persistent server connections, enables pipelining of requests and responses, shard data automatically across multiple servers keep copy of the node configurations [3].

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 keys (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 an 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 built 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
```

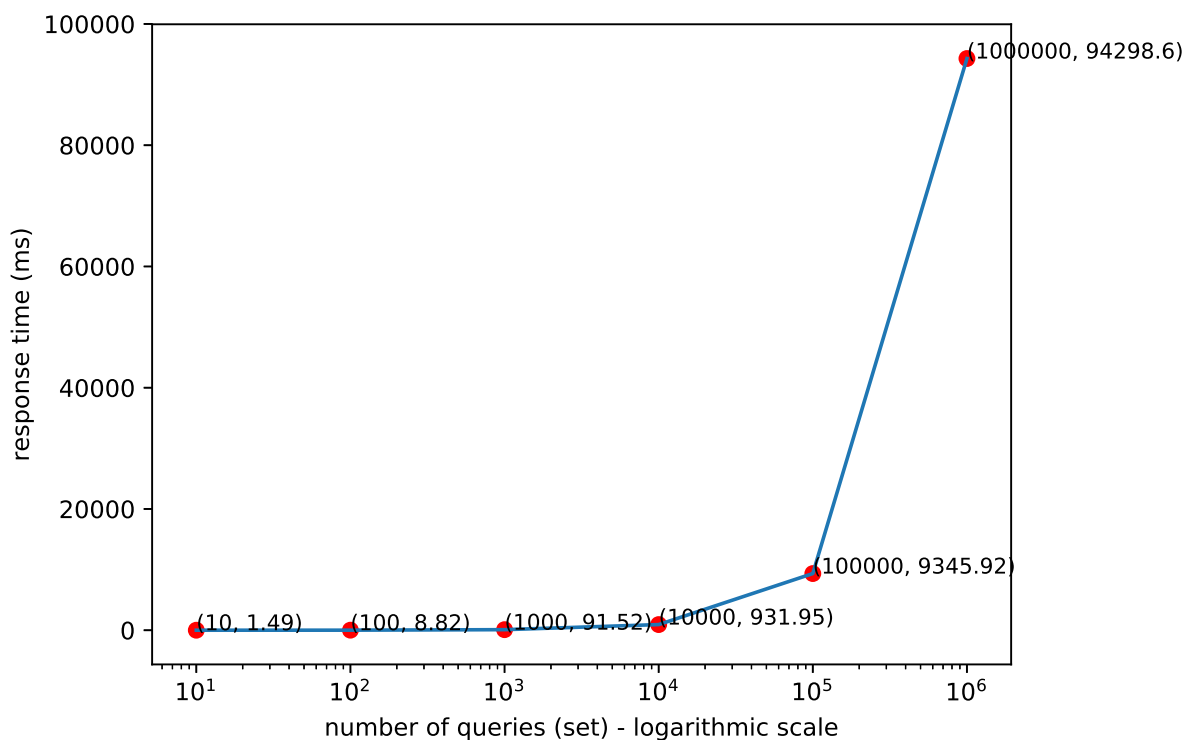
After creating the network, we need to run each of the instances in this network. One important thing to note is that we need to give the default port (6379) in all of them, this is because Docker network, itself allocated unique IP:PORT to each of the container in the network.

```
$ 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).

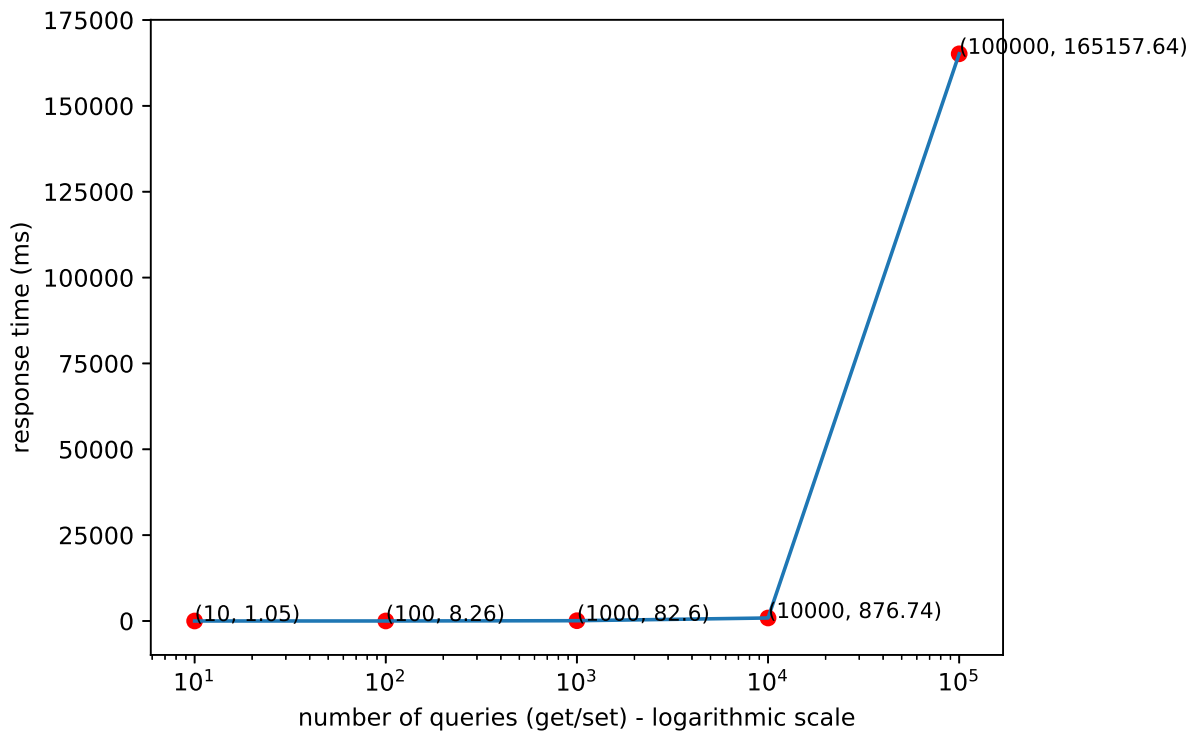
### Single redis instance

#### Single redis instance, with no memory/cpu restriction



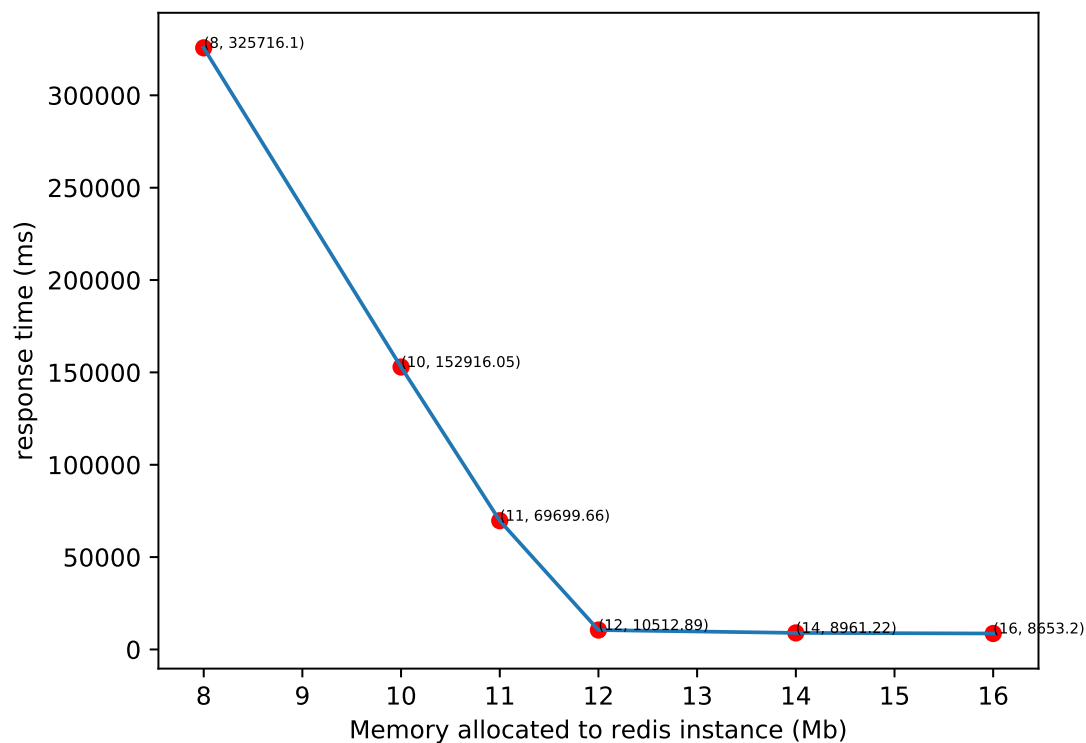
This graph is on the logarithmic scale, on a linear scale we observe a linear relationship between the response time and the number of queries.

## Single redis instance, with max main memory = 10Mb



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.

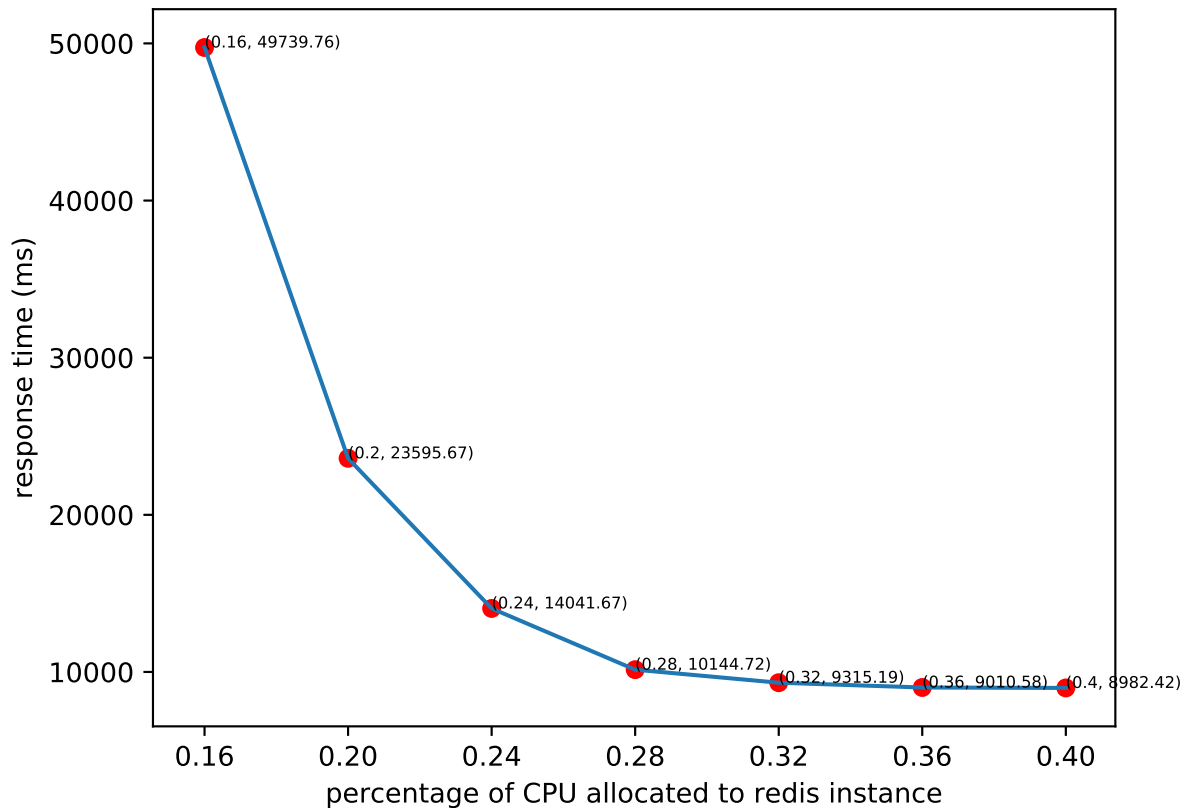
## $10^5$ queries on redis instance, with varying main memory





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

## $10^5$ queries on redis instance, with varying CPU

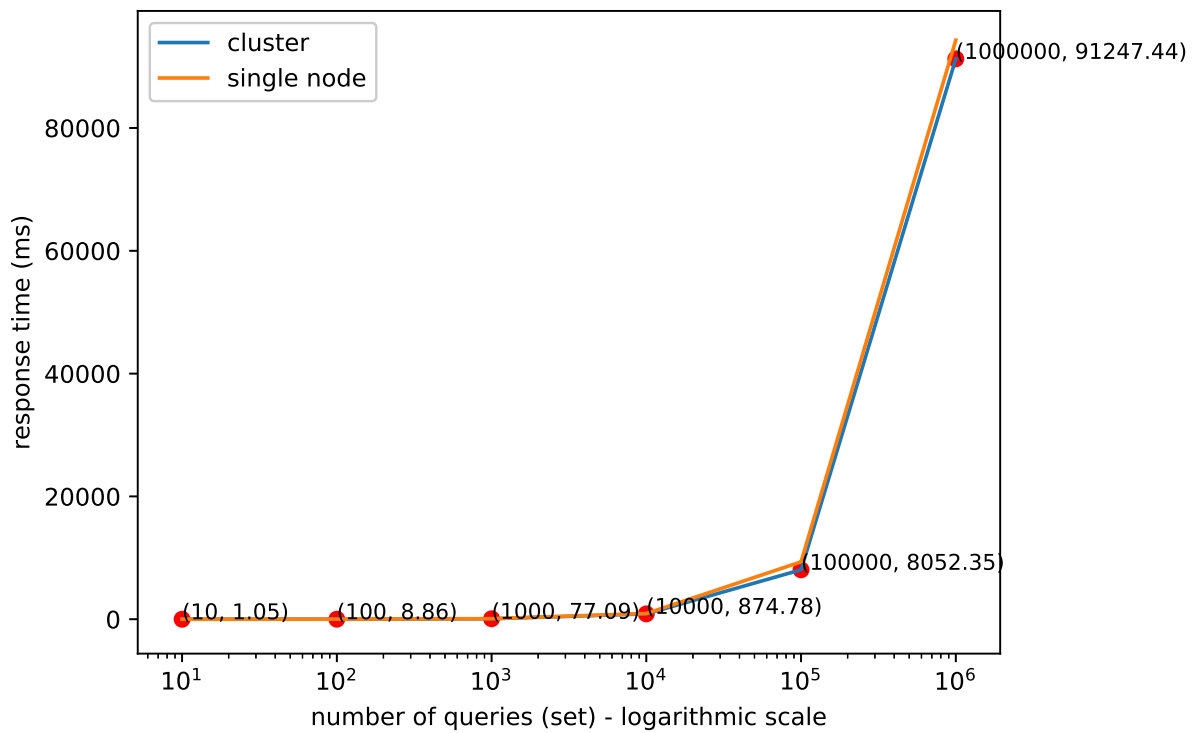


We cannot have fractional CPUs, above values just signify CPU period, and the CPU quota. For example CPUs="1.5", means that the node is guaranteed at most one and a half of the CPUs, which is the equivalent of setting CPU-period="100000" and CPU-quota="150000"

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.

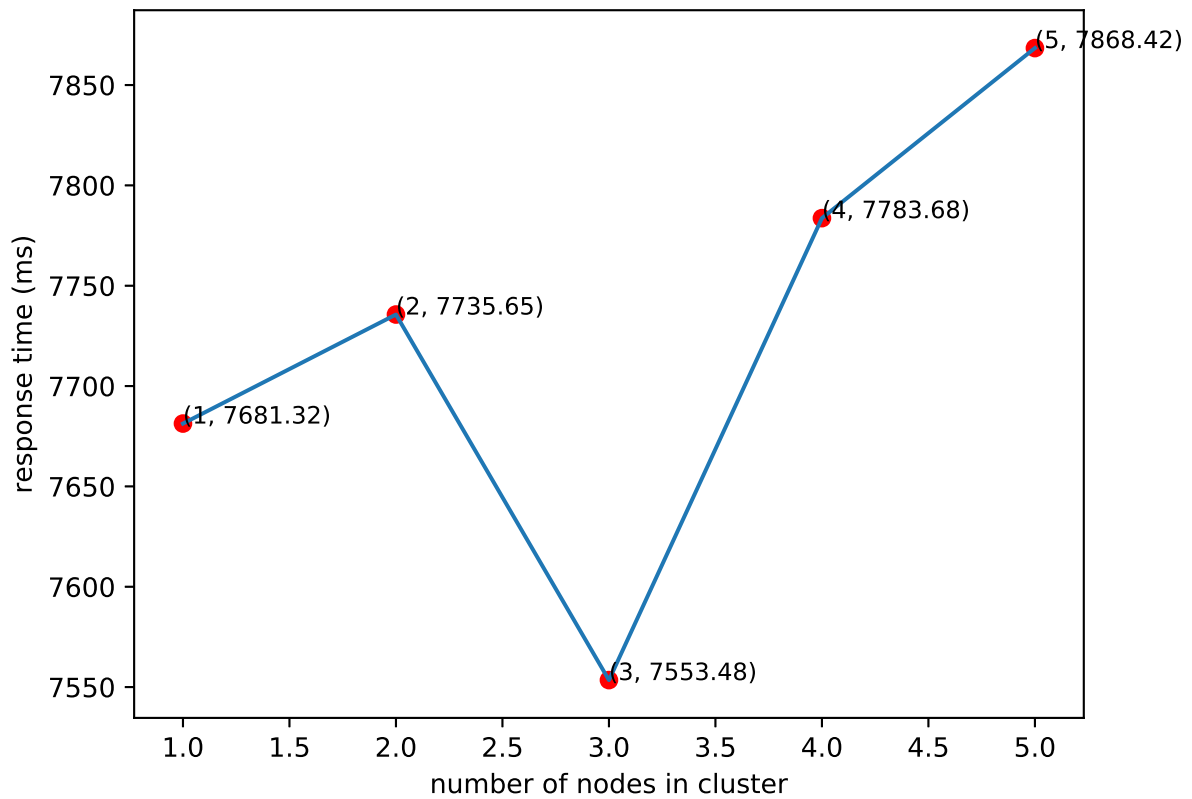
## Redis Cluster experiments

### Single redis instance, compared with 6 Node cluster



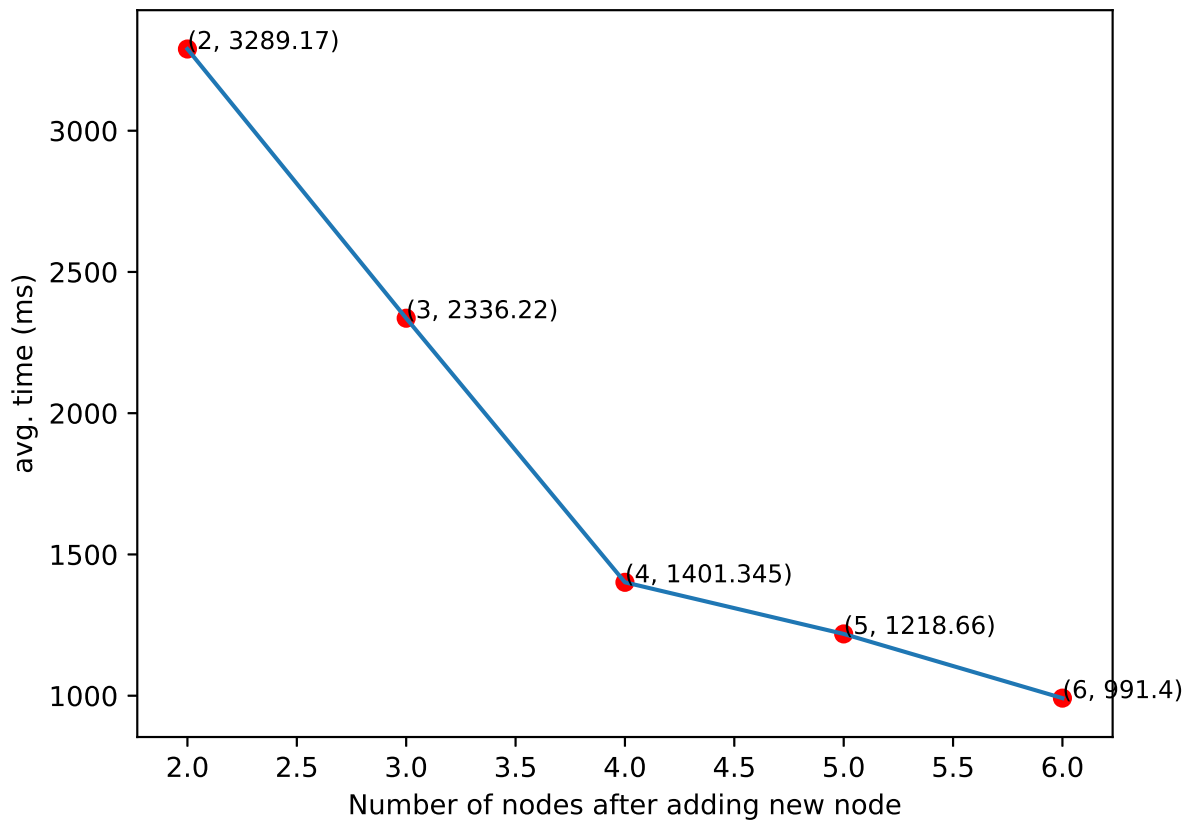
The above experiment is performed with no memory/CPU restriction, we can observe that the cluster based system performs a bit better (though the effect is negligible).

## $10^5$ queries on different node configurations



In this experiment, we try to find the number of nodes in a cluster that would give the best performance. All the resources are the same in each of the configurations. We observe that there is no direct correlation between the number of nodes and the response rate. This is because increasing number of nodes would create an overhead for Twemproxy but would reduce the load on each individual instance, so there is a kind of trade-off.

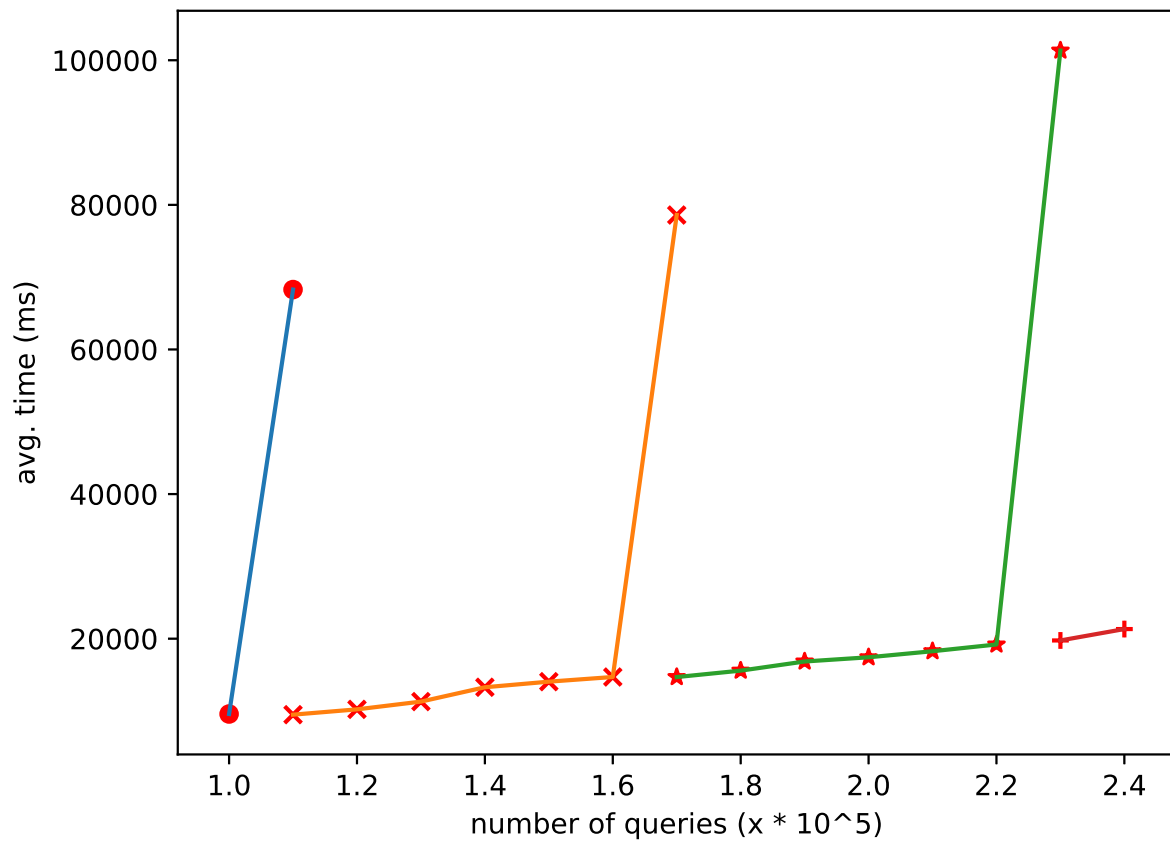
## $10^5$ keys present, setup time for adding new node



In this experiment, we try to find the startup time for a give node in different cluster configuration. Note that the number of keys present in the system is constant throughout. Initially, we just have a single node in the cluster holding all the keys, then we gradually add a new node in the cluster and measure the downtime (not exactly downtime, as the cluster can still respond to requests).

An interesting finding of this experiment is that the startup time is inversely proportional to number of nodes in the cluster, which is quite intuitive as the number of keys needed to be transferred will reduce by a factor of the number of nodes.

## Varying queries and number of nodes



In this survey, we try to maintain a threshold of response time, and if the response time gets above this threshold value, we increase the node count by one. Each node in the system has limited memory available (6 Mb). So we can see jumps in the case when memory the limit is reached in the cluster. For the above example response time threshold is 20 sec.

## Current Picture

### Machine learning based solution [4]

Note that scaling up/down can be modeled as a classification problem since it is a binary decision problem. So we can use all the state of the art algorithms, to refine the model. In their research Sabidur et al try to use network load to dynamically reach to traffic changes. They used different ML classifiers, which were fed by past decisions taken by the system, and the seasonal (trends, etc) behavior of the network traffic load to generate dynamic scaling decisions. This model is based on online learning algorithms since online data is fed into the classifier.

Different models used in the paper are:

- Reduced Error Pruning Tree.
- Multi-Layer Perceptron.
- DNNs.

They have used a variety of features, some of them are startup time, resource utilization factors, last 7 day's scaling up/down mean, etc.

### Static metric based approach [5]

In their paper, they present on a simple auto-scaler, which doesn't take any season trends into account. It simply increases the number of nodes if the time elapsed since the last scale up, is greater than provisioning interval. A similar simple auto-scaler is used for scaling down, but with a different epoch compared to scaling up.

They also present a "Single Instance Binding Autoscaler", which basically aims at making the scale up/down process idempotent. Since multiple nodes can fire the autoscaler to scale up, but for a particular instance of time, only single node should be added.

### Performance models [6]

These models aim to precisely for distributed stream processing systems that can predict the execution performance of a job even before deployment. For collecting this information, they generally perform benchmark testing and use statistics to make the predicted performance time closer to actual values. These models can be used to optimally schedule logically specified jobs onto available physical hardware. These models and the derived execution schedules can be refined online to dynamically adapt to unpredictable changes in the runtime environment or auto-scale with variations in job load.

## Future Works

Some of the things that can be added further are:

- Implement dynamic scaling feature in Twemproxy. Currently does not support dynamics scaling of redis cluster, if we increase node count in the cluster, we have to restart twem.
- Add a service on top of Twemproxy to apply scaling logic (based on statistics received from Twem).
- Test some state of the art machine learning algorithms like Meta learning, Siamese nets, etc.

## References

- [1] Redis documentation, [redis.io](https://redis.io)
- [2] Bondi, Andre. *Characteristics of scalability and their impact on performance*. WOSP, 2000.
- [3] Twemproxy, developed by twitter, [github.com/twitter/Twemproxy](https://github.com/twitter/Twemproxy)
- [4] Auto-Scaling Network Resources using Machine Learning to Improve QoS and Reduce Cost, Sabidur Rahman, Tanjila Ahmed, Minh Huynh, Massimo Tornatore, Biswanath Mukherjee. Aug'18.
- [5] Containers Orchestration with Cost-Efficient Autoscaling in Cloud Computing Environments, Maria A. Rodriguez, Rajkumar Buyya, Dec'18.
- [6] Trevor: Automatic configuration and scaling of stream processing pipelines, Manu Bansal, Eyal Cidon, Arjun Balasingam, Aditya Gudipati, Christos Kozyrakis, Sachin Katti, Dec'18.