Dissecting Two Popular Key Value Stores: Redis and Memcached

Noshin Nawar Sadat
University of Waterloo
Waterloo, Canada
nnsadat@uwaterloo.ca

Ishank Jain
University of Waterloo
Waterloo, Canada
ishank.jain@uwaterloo.ca

ABSTRACT

We study the performances of two very popular open source key-value stores Redis and Memcached. For this purpose, we use different customized workloads from the Yahoo! Cloud Serving Benchmark (YCSB). We provide performance analysis and suggest design changes for the two systems for enhanced performance.

CCS CONCEPTS

• **Information systems → Key-value stores**.

KEYWORDS

Cache memories, Key-value stores, Memcached, Performance Analysis and Design Aids, Redis, YCSB

ACM Reference Format:
Noshin Nawar Sadat and Ishank Jain. . Dissecting Two Popular Key Value Stores: Redis and Memcached. In . ACM, New York, NY, USA, 12 pages.

1 INTRODUCTION

NoSQL frameworks are important part of websites and cloud services and are widely used. For example, Dynamo is used at Amazon [20]; Redis is used at GitHub, Weibo; and Memcached is used at Facebook, LinkedIn and Twitter [23] [29]. There are different types of storages in NoSQL database. For instance, key-value store, column store, document store and graph databases.

A typical use case for key-value store is as a layer in the information retrieval system: a cache for expensive-to-obtain values, indexed by unique keys. These systems can store the information that is less expensive or quicker to cache than re-acquire. For example, generally accessed results of database queries or the results of complex computations that require temporary storage and distribution. When performance of a web application or service needs to be improved, caching is one of the first step that is considered.

Moreover, key-value stores are for the most part network-enabled, allowing sharing of data over multiple server machines and offering usefulness of scaled-out shared memory without the requirement for any special hardware. Caching and sharing the data among many front-end servers allow system architects to plan for simple and linear scaling, adding more key-value stores to the cluster as the data grows [9].

With the generous number of open-source and promptly accessible key-value store, web applications engineer experience the confusion of choosing among many such options. They have to benchmark these systems to know which one is more suitable for the required application. Benchmarking in this respect refers to a performance evaluation of key-value store solutions proposed or in use.

Whenever it comes to choosing key-value stores, Memcached or Redis are typically the first places to turn[12]. Systems such as Redis and Memcached keep there data exclusively in-memory to help avoid cost of input-output operation. Both the system are fast, but they offer different functions and have different storage mechanisms. All these factors led to our decision to evaluate the two system.

In this project, we studied and summarized the architecture as well as storage mechanisms of Redis and Memcached. While reviewing the related literature we did not come across any prior work that provided information on both the systems in detail. In our experiments we evaluated their performances using Yahoo! Cloud Serving Benchmark (YCSB). It is an independent, highly customizable benchmarking tool designed to evaluate cloud-serving stores and is used as a de-facto benchmarking tool for key-value stores [10].

To summarize, our contributions are:

- We have briefly summarized the architecture and storage mechanism of Redis and Memcached.
- We ran experiments using varying workloads on both Redis and Memcached to analyze their performances.
- Based on our analysis of the results and also study of the architecture of the two systems, we have suggested some design changes in Redis and Memcached.

2 ARCHITECTURE DESIGN

In this section we will discuss how Redis and Memcached store and exchange data between clients and server machines. We will also discuss some of the common operations and how communication takes place between client and server machines during those operations.

2.1 Redis

Redis is an in-memory remote database that offers high performance and a unique data model to produce a platform for solving problems [16]. Redis supports different kinds of data structures, such as strings, bitmaps, maps, sorted sets, HyperLogLogs, lists and streams. In addition to that, Redis supports various features like replication that allows to scale read performance; in-memory persistent storage on disk, and client-side sharding to scale write performance. These allow Redis to scale from a small system to a distributed system, allowing it to handle hundreds of gigabytes of data and thousands of requests every second.

To handle memory management, Redis stores the memory size in the memory block header following memory allocation. Redis stores the memory block size in the header and the memory occupied by

University of Waterloo, Waterloo, ON, April, 2019
© 2019

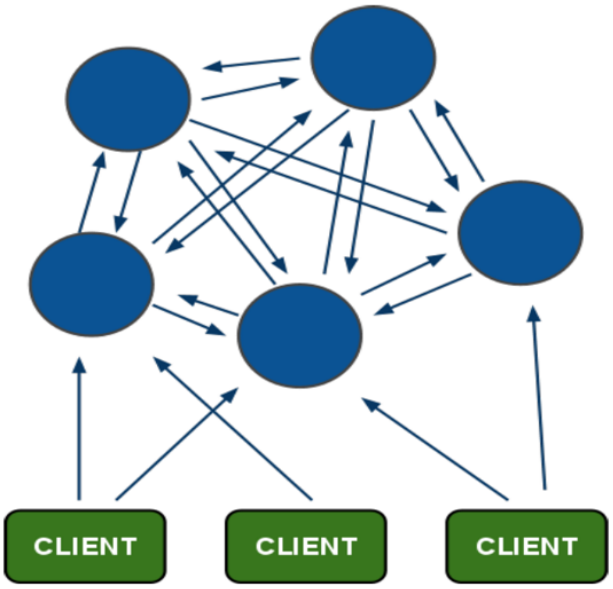

Figure 1: Overview of a typical client-server structure for Redis distributed storage service.

the size is determinable, i.e., the system returns the length of the size of specific data type, and then the return pointer. When there is a need to release memory, the system passes return pointer to the memory management program. Through the return pointer, the program calculates the value of return pointer and then passes the return pointer to release the memory [18]. This mechanism optimizes storage space without wasting space by creating fixed size blocks, but instead dynamically allotting space.

Figure 1 [30] gives a typical topology of a distributed Redis storage service. In the figure we can observe how the system can be scaled out with good performance and how the system can ensure data safety. There are usually multiple machines that are providing data storage services together, which are called "Nodes" [17]. Each node is responsible for storing parts of the data and different clients can visit these nodes to read and write the data independently. To ensure data safety in case on node failure, Redis supports master-slave replication. Slave nodes hold the replica of the data at the master node and is used to retrieve data in case of failure.

Redis uses a key-slot-node mapping strategy to maintain the data distribution status. Figure 2 [17] shows how the keys are mapped to each node. In the hash stage, for a given key, Redis will calculate its hash value. In Redis, all hash values are evenly assigned to each slot and there are 16,382 slots in total, all slots can be dynamically assigned to different nodes. This allows a node is under heavy workloads to move parts of its slots to other nodes or even new nodes to improve the system's overall performance. To avoid single-node failure, Redis follows a fully decentralized design, which uses the Gossip protocol [21] to maintain these important information on all nodes. As shown in figure 1, all nodes in the system are fully connected and know the current state of the system. Nodes also send random PING messages to other nodes and wait to receive

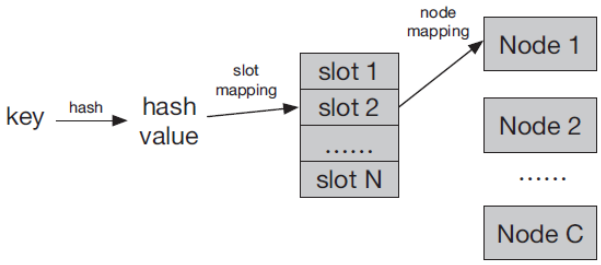

Figure 2: Mapping from key to node.

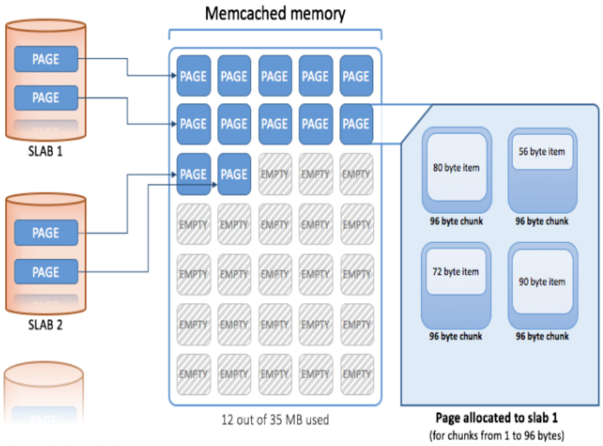

Figure 3: Memcached architecture.

PONG messages to confirm that the cluster is in proper working state.

In Redis, when a node receives a key-value request from a client, it will check whether this key belongs to the node by checking the key-slot-node mapping available on the node. If yes, it will begin processing this request and return the results back to the client. Otherwise, it will return a MOVED error which contains the information of the right node for this request. After receiving the error message, the client will know which node it should forward the request to.

In Redis, all the commands in a transaction are serialized and executed sequentially. A request issued by one client will not be served in the middle of the execution of a Redis transaction. This mechanism guarantees that the commands are executed as a single isolated operation. If there are race conditions and another client tries to modify the result of value in the time between call to WATCH and call to EXEC, the transaction will fail. The operation is repeated hoping that next time there will be no new race condition. This form of locking is called optimistic locking [4].

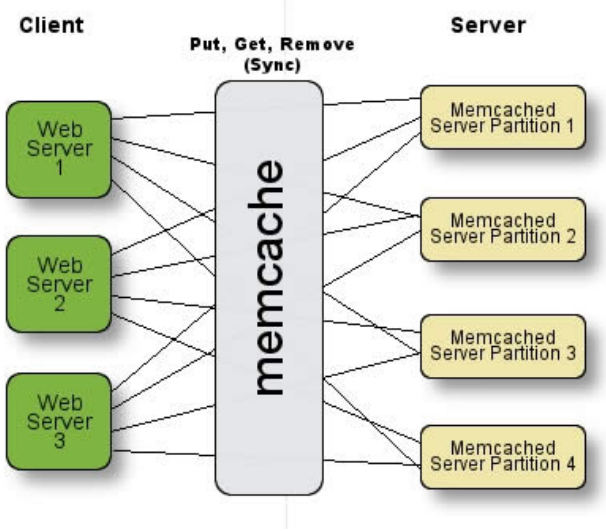

Figure 4: Memory Allocation in Memcached.

2.2 Memcached

Memcached is a high-performance, distributed memory object caching system, generic in nature, but originally intended for use in speeding up dynamic web applications by alleviating database load [22]. Memcached uses a client-server architecture. In this architecture servers maintain a key-value associative array; the clients populate this array and query it by key. In Memcached the keys are up to 250 bytes long and values can be at most 1 megabyte in size [5], which can be manually increased in the latest versions. Memcached does not allocate the memory for the data on an item by item basis. It uses slab allocation (figure 3) [1] to optimize memory usage and prevent memory fragmentation when information expires from the cache. With slab allocation, memory is reserved in blocks of 1MB, the slab is divided up into a number of blocks of equal size. When we try to store a value into the cache, Memcached checks the size of the value that we are adding to the cache and determines which slab contains the right size allocation for the item. If a slab with the item size already exists, the item is written to the block within the slab. If the new item is bigger than the size of any existing blocks, then a new slab is created, divided up into blocks of a suitable size. If an existing slab with the right block size already exists, but there are no free blocks, a new slab is created. If we update an existing item with data that is larger than the existing block allocation for that key, then the key is re-allocated into a suitable slab.

A Memcached cluster 4 [9] provides a distributed hash table for storing small objects (up to 1 MB), providing a simple set/get interface. Each object's key is used to determine which individual Memcached server within the cluster will store the object. Generally, a hash function is chosen to distribute the keys evenly across the cluster [25]. In the cluster individual Memcached servers do not communicate with each other, as each server is responsible for its own independent range of keys. Because the servers do not interact, the performance of a single Memcached server can be used to generalize the behavior of an entire cluster.

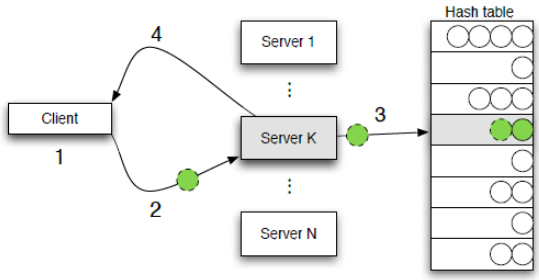

Figure 5: Write path.

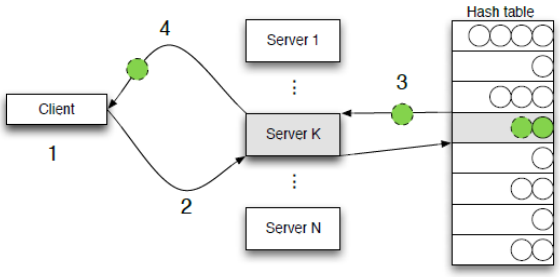

Figure 6: Read path.

Memcached provides all the basic primitives that hash tables provides such as insertion, deletion, and lookup - as well as more complex operations. The interface provides the following operations [9]:

- STORE: this function stores (key; value) in the table.
- ADD: this function adds (key; value) to the table iff the lookup for key fails.
- DELETE: this function deletes (key; value) from the table based on key.
- REPLACE: this function replaces (key; value1) with (key; value2) based on (key; value2).
- CAS: this function does atomic compare-and-swap of (key; value1) with (key; value2).
- GET: this function retrieves either (key; value) or a set of(key; value).

The first four operations mentioned above are write operations and follow the same code path as for STORE (Figure 5) [9]. Write operations are always transmitted over the TCP protocol to ensure retries in case of a communication error. STORE requests that exceed the server's memory capacity incur a cache eviction based on the least-recently-used (LRU) algorithm [9].

GET requests follow a similar code path (Figure 6) [9] [Many-Core Key-Value Store]. In Memcached, client side holds the key-value table and directly address the server holding the data. If the key to be retrieved is actually stored in the table (a hit), the (key, value) pair is returned to the client. Otherwise (a miss), the server notifies the client of the missing key. One notable difference from

the write path, however, is that clients can opt to use the faster but less-reliable UDP protocol for GET requests. It is worth noting that GET operations can take multiple keys as an argument. In this case, Memcached returns all the key-value pairs that were successfully retrieved from the table. The benefit of this approach is that it allows aggregating multiple GET requests in a single network packet, reducing network traffic and latencies.

3 EXPERIMENTAL SETUP

In this section, we will discuss in detail our entire experimental setup and design choices and the following section will be devoted to the results we found through our experiment and our own analysis of the results we got.

3.1 Evaluation Metrics

[19] had proposed two benchmarking tiers for evaluation of cloud serving systems in their paper, based on which YCSB was designed:

- `Tier-1 (Performance)`: This tier involved evaluating the latency of operations (read, update, insert, scan and delete) as well as the throughputs (operations/second) of the systems. For our experiment, we only tested for the latency and throughputs for read, insert and update operations.
- `Tier-2 (Scaling)`: This tier involved studying the affects of having more machines added to the database system. They had proposed two different methods to see this affect - one was to simply compare performances of small and large cluster and the other was to see how the systems fared when machines were added while it was running (elastic speedup). For our experiment, we used the first method and not the second one. This is because in Memcached, the nodes do not have information of the other nodes in the cluster. Hence, while the Memcached cluster is running, we would have to make changes on the client side if more new machines are added. On the other hand, for Redis, we only need to add the socket address of the new node and the rest will be handled by the Redis cluster nodes. Hence, that would be more of a comparison between Memcached client and Redis servers.

Furthermore, YCSB also allows us to change the number of concurrent clients, which are separate threads that try to send requests to the database servers at once. This can help evaluate how the systems perform when the number of concurrent requests are increased gradually.

Moreover, while we were adjusting the sizes of workload databases, we noticed that Redis was taking up more memory space to store the data compared to Memcached. We thought that would be an important metric to evaluate the two systems too.

Therefore, in our experiment, we evaluated Redis and Memcached based on the following metrics:

(1) *Average Latency (in µs) :* We evaluated the latency of read, update and insert operations for the two systems when faced with varying workloads.
(2) *Average Throughput (operations/second) :* We evaluated the number of overall operations per second the systems performed when faced with varying workloads.
(3) *Memory Usage (in MB) :* We calculated the memory usage of the two systems given the same database size.
(4) *Scaling Out :* We evaluated the change in performances of the two systems when scaled from a single node setup to a three node cluster. We chose three node cluster size because it was stated that to have a Redis cluster perform as expected, it should have at least three master nodes [3].
(5) *Number of concurrent clients:* We evaluated the change in performances of the two systems when they were accessed by more than one concurrent clients at a time. These threads were generated by YCSB client. We increased the number of concurrent threads in multiples of 12 up to 48 as our server machines had 12 cores and we wanted to test how they performed in overloaded conditions.

Results of all these metrics were generated at the YCSB client which executed each benchmarking workload.

3.2 System Configuration

We performed our experiments in two different modes to test the scalabilities of the two systems:

(1) `Single Node Mode` : Here, the systems were installed in one server machine and the YCSB client was installed in another machine.
(2) `Cluster Mode` : Here, the systems were installed in three server machines and the YCSB client was installed in another machine.

For both Redis and Memcached, we tried to use the bare minimum configurations. Since Redis nodes in a cluster have information about other nodes in the cluster, we had to make some changes to its configuration files. Moreover, Redis also supports replication and persistence by default, which Memcached doesn't. Hence, we had to disable these features for Redis in order to make the comparison more fair.

The machines we used as servers, had the following configurations:

- Ubuntu version : 16.04.6 LTS (GNU/Linux 4.4.0-137-generic x86_64)
- Total memory : 15GB
- Cores : 12
- Processor : AMD Opteron(tm) Processor 4332 HE
- Ethernet : NetXtreme II BCM5716 Gigabit Ethernet

The machine we used as client had the same configurations as above, except that their total memory was 7.76 GB. The same machines were used to perform the experiments for both the systems so that their inter-connections did not have any impact on the performances of the systems.

We would like to mention here that our experiment was a part of a course project at the University of Waterloo and our choice of machines were mostly governed by what could be provided to us from the University.

3.3 Workloads

The workloads in the YCSB project [14] can be modified easily, which was part of the several reasons why we chose to use this benchmark. Since our goal was to evaluate the latencies of reads, inserts and updates, we prepared six different types of workloads in our experiment. The summary of which is provided in Table 1. Our

Table 1: Types of Workloads Used

Workloads	Description	Initial Database Size	Number of operations
Workload A	50% Reads and 50% Updates	10.2 GB	2,500,000
Workload B	95% Reads and 5% Updates	10.2 GB	2,500,000
Workload C	5% Reads and 95% Updates	10.2 GB	2,500,000
Workload D	90% Reads and 10% Inserts	5.1 GB	1,250,000
Workload E	10% Reads and 90% Inserts	5.1 GB	1,250,000
Workload F	50% Reads and 50% Inserts	5.1 GB	1,250,000

Table 2: Memory Usage

Scale	Memory Usage (in MB) Memcached	Redis
Single Node	10,876	14,740
Per node in Cluster	3,630	4958

choice to use these specific mixes of reads, inserts and updates in the workloads was in order to test different possible combinations of workloads in which the two systems performed better.

All the workloads worked with the same kind of database that included a single table of records. Each record had 16 fields in total and each field had the size of 255 bytes. The former was chosen in order to keep its number neither too small nor too big, whereas the latter was chosen because often field sizes of databases which hold strings are set to 255 characters as a safe choice [27]. Therefore, the total size of each record is 4,080 kB. Each record in the table are identified by a primary key which is a string and the value of each field is a string of ASCII characters which are generated randomly.

The order of the operations of a workload are randomly chosen and the choice of the record which is read, updated or inserted is also determined randomly by the YCSB client. The YCSB client supports Zipfian, Uniform, Latest and Multinomial distributions. We chose the Zipfian distribution for our experiment because it provides the closest simulation of real-world data streams [11] [10] and also is set as the default distribution of the client.

In our experiment, the three operations perform the following tasks:

- `Insert` : Insert a new record with 16 fields.
- `Update` : Update a single field of a record, where the new valuse is of size 255 bytes.
- `Read` : Read all 16 fields of a record.

Before running each workload, the databases were pre-loaded with some data. We wanted to load the databases to their highest potential but wanted to avoid overloading them because then, the systems would start evicting old data to make space for new data. Hence, we tried to find the maximum amount of data the systems could hold by gradually increasing the size of the database.

We found that a single node server of Redis required 44.5% extra memory to store data, whereas Memcached only required 6.6% of extra memory (See Table 2). Since the RAM sizes of our machines were 15GB, we had to set our initial database size to 10.2 GB for workloads A, B and C. These workloads did not involve inserts and thus, did not increase the size of the database further. However, workloads D, E and F did. And so, for the latter three, we had set the initial database size to half of 10.2 GB. To achieve these, the number of records were set to 2,500,000 for the first three workloads and 1,250,000 for the latter three. We wanted to keep a one to one mapping of the number of records and number of operations for easier debugging, hence we chose the number of operations equal to the number of records.

4 EVALUATION

According to [12], YCSB client starts calculating latencies right after a read, insert or update method is called at the database interface layer at the client side. So, the average latencies that were generated by the YCSB client, included the latency in the database interface layer (the YCSB client code), the database engine operations and the network latency. However, through our experiment, we noticed that the average latency also includes the time a request has to wait until it gets processed by the system.

4.1 Workload Analysis

Workload A. From figure 7, it can be seen that Memcached has a lower read latency and also slightly lower update latency compared to Redis in both single node mode and cluster mode for workload A. Since Redis is single threaded, it can only handle one operation at a time, so even if it is faster for read and update operations within the system (see tables 3 and 4), most of the threads have to wait until their turn comes. Moreover, if multiple clients try to increment the key at about the same time there will be a race condition. In Redis, which operates with one thread at a time this can lead to optimistic locking. The throughput of Memcached gets better compared to Redis not only because it is optimized for multi-threaded operations but also, it has lower latencies for this workload. Moreover, in cluster mode, for any kind of operation request in Redis, if the contacted node does not own the hash slot for the current record sent, then it asks clients to contact the node in the cluster which actually owns the hash slot. In case of reads, this requires maximum four communications in total between client and the cluster nodes - to request the node whose IP address is

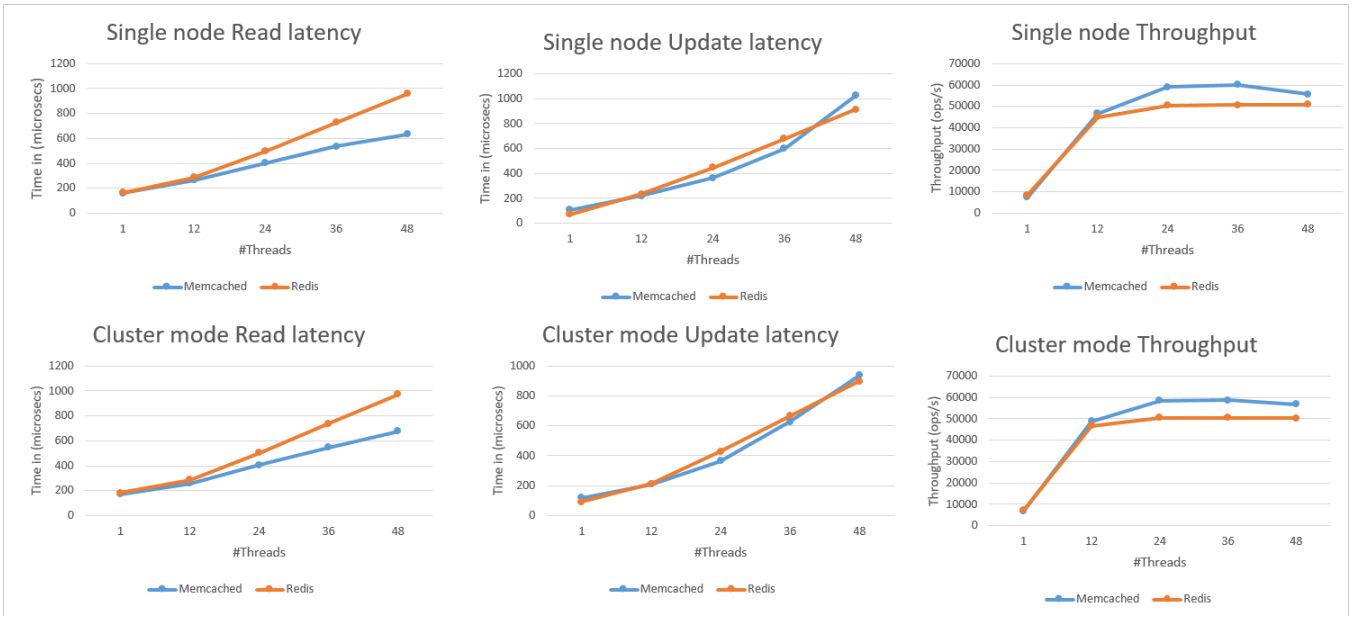

Figure 7: Latency and Throughput results of the two systems for Workload A

Figure 8: Latency and Throughput results of the two systems for Workload B

known, contacted node sends information on the node that actually can handle the request, client re-sending the request to correct node and then, correct node replying. In case of updates and inserts, this communication number can be maximum of three with the last step not being required. This re-direction does not occur in Memcached because the client already knows which node in the cluster actually can handle the request.

Workload B. Figure 8 shows latencies and throughputs for workload B and they look similar to those of workload A except the read latencies of both the systems increased while update latencies increased for Redis and decreased for Memcached. Since this is a read heavy workload, a lot more time is spent on communication over the network for both the systems and the systems had to wait longer to push the data out into the network, leading to latencies in reads. On the other hand, with less number of updates, Memcached

Figure 9: Latency and Throughput results of the two systems for Workload C

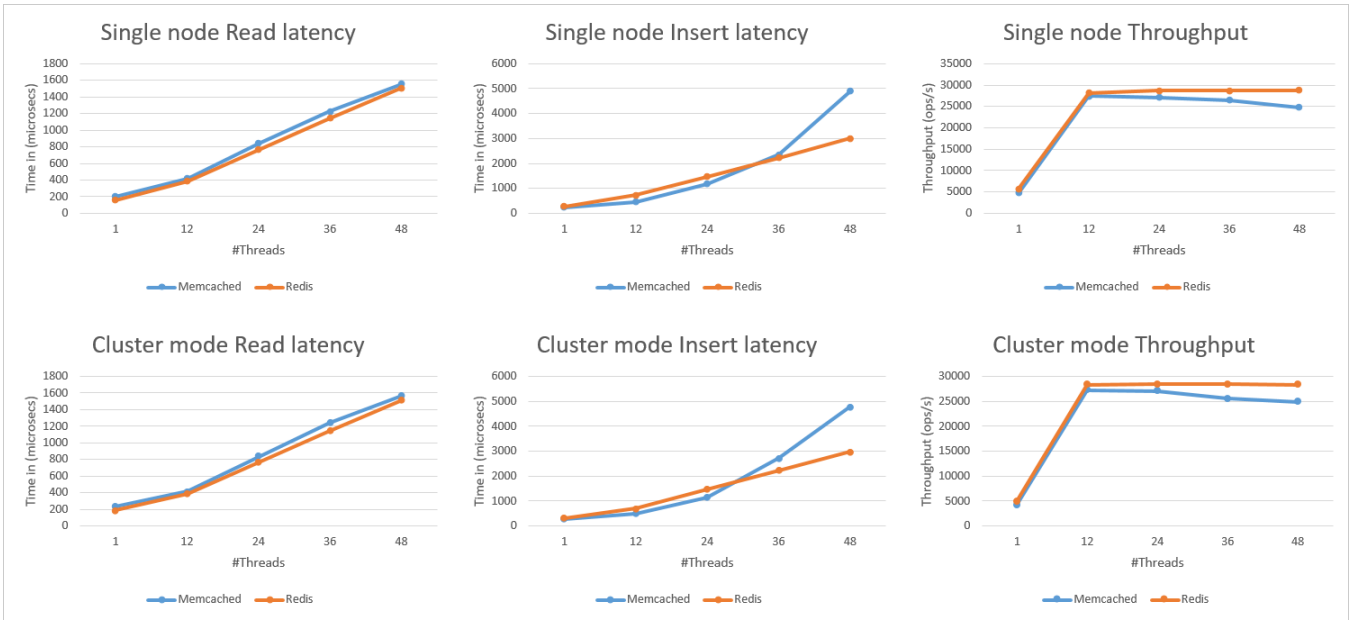

Figure 10: Latency and Throughput results of the two systems for Workload D

resources went through less number of locking for writes which led to the decrease in update latency. Redis' update latency might have increased because of Zipfian distribution which could have led to increased number of requests for a node that was unknown to the client. All these resulted in drastic decrease of throughput of Redis while that of Memcached remained the same. After all, throughput has an inverse relationship with latency [19].

Workload C. For workload C (as shown in figure 9), the performance of Memcached deteriorates. This is because it is an update heavy workload which requires larger number of locks on the pages for making updates and the fact that the records are accessed with a Zipfian distribution, certain pages become more in demand compared to other pages which negatively affects the performance of Memcached. We were also able to detect this phenomenon when we ran a 100% update workload on Memcached. We observed that

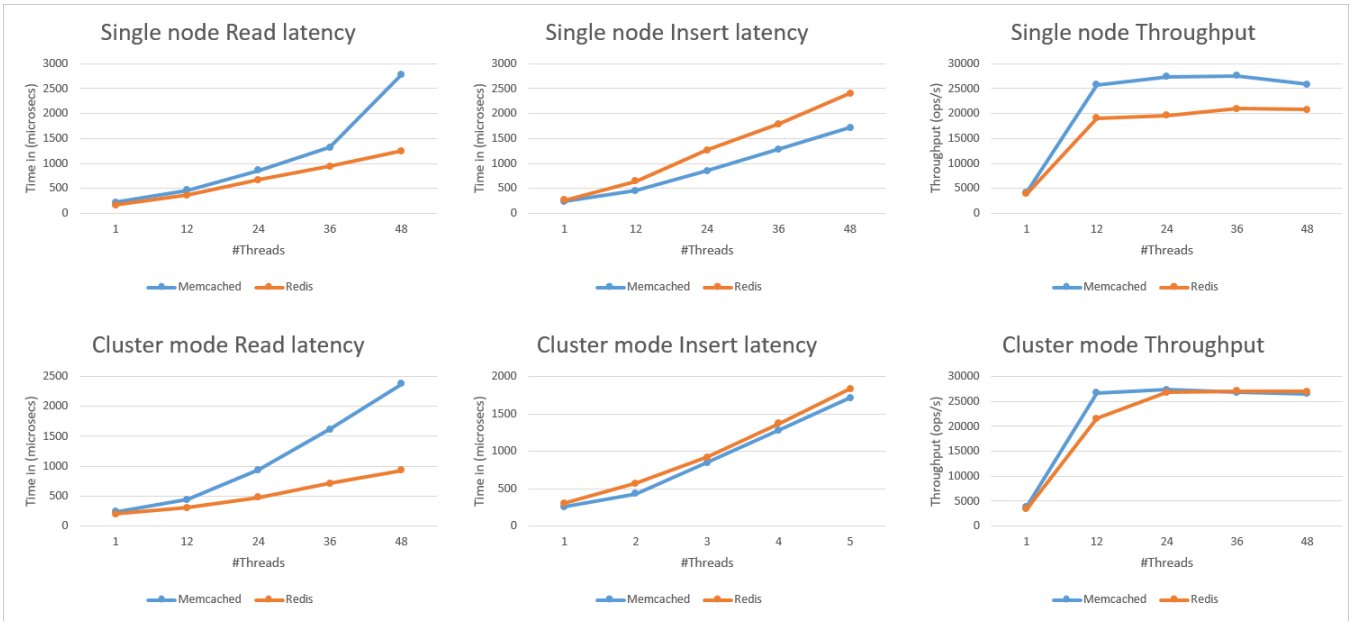

Figure 11: Latency and Throughput results of the two systems for Workload E

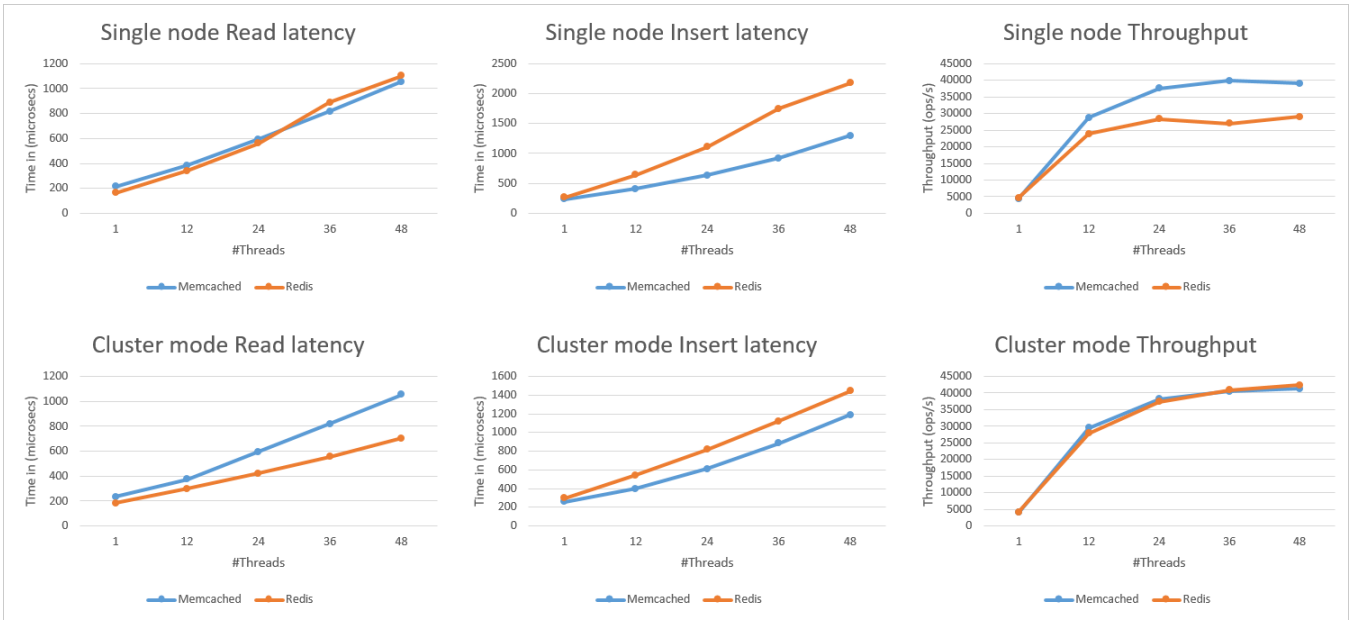

Figure 12: Latency and Throughput results of the two systems for Workload F

when dealing with multiple threads, Memcached used the futex() system call a large number of times which takes up considerable amount of time. This system call is mainly used to block accesses on shared memory. This was not the case for Redis.

Workload D. The performances of both the systems for workload D (as shown in figure 10) is similar to that of workload B except that Memcached's performance deteriorates. Compared to updates, the futex() system calls are used considerably higher number of times during inserts in Memcached. Moreover, during inserts, Memcached has to look up for pages that can hold the chunks and if not, they have to create new slabs. This leads to increased latency and thus lower throughput.

Workload E. The performances of the systems for workload E (as shown in figure 11) show that Memcached performs better in inserts

while worse in reads compared to Redis. Redis is slower for inserts because a lot of its incoming threads have to wait for their turns. Moreover, in cluster mode, if the key value is different from the node which is computing it then it has to be sent to the correct node. On the other hand, in case of Memcached the key is computed at client end and then sent to server machines and there also is no need for redirection which is reflective from the numbers in the graph. Moreover, as this is an insert heavy workload, it involves a lot of locking which affects the reads of Memcached.

Workload F. (see figure 12) is composed of equal numbers of reads and inserts and the performances of the two systems are similar to that in Workload E and for similar reasons.

4.2 Scaling Analysis

We observed for both the systems that the increase in number of threads, increased the latencies. However, the throughputs almost saturated after 12 threads. Again, when we scaled our systems from single node mode to cluster node mode, there was very little improvement in performances for both the systems. We believe, this is because the client machine used in our experiment was the bottleneck as it had only 12 cores and could only perform 12 threads or a bit more at a time.

This indicates that even though we can perform a variety of experiments with a single YCSB client, its experiments are still bounded by the client machine configuration, which can add to the overall latency of any workload. To overcome this, we could run multiple client machines in parallel as instructed in [13].

4.3 Other Observations

Overloading the Server Machines. Our goal of using up to 48 concurrent YCSB clients was in order to overload the cluster machines, which had 36 cores in total, and see how the performances of the systems got affected because of it. We observed that depending on the workloads, a change in the number of threads from 36 to 48, led to variation of 2,000-3,200 op/sec. Several of the times, they remained constant. The latencies for all the cases increased, which was expected since more incoming requests means more wait time.

System Calls. When we were looking into the system calls and their durations for both Redis and Memcached with 100% reads, 100% inserts and 100% updates workloads, we noticed that the Memcached server process took a considerable amount of time to handle the workload. Table 3 and 4 show the average latencies for each operation, the system call duration per operation on the Redis/Memcached server side and that on the YCSB client side. It can be seen that for Redis, the system call durations for both the server and client side is almost negligible compared to the total average latency, which is not the case for Memcached. Therefore, for Redis, almost all part of the latency is due to the multiple communications per operation over the network and also the amount of time the threads have to wait until their time comes to being processed.

The main differences we observed between the types of system calls used by the two system were:

- Memcached used sendmsg() to write data into the sockets whereas Redis used the write() system call.
- Memcached spent a lot of time (almost 99%) waiting for resources to free up whereas most of the time in Redis was dedicated to write() system call.
- Memcached also had the futex() system calls for handling shared memory access which was absent in Redis

5 RECOMMENDATIONS

For the current settings in the Redis each time when the client sends a key to a server machine, the server machine will check whether this key has a place with it or not. If not, it will send a MOVED error back to the client side which could then redirect the client to the correct server machine. So if on every missed chance we can cache this mapping status on the client machine, then next time this client can be ready to find the right server machine effectively. In addition to this, instead of sending the error back to client the server machine should send the request to the correct server machine with updated destination IP address. This will reduce the number of communications among the servers and clients. Since all the server machines in the Redis cluster are already connected to each other, there will be no need for new hardware.

For Memcached, the default maximum record size is 1MB. In Memcached 1.4.2 and later, we can change the maximum size of an record using the -I command line option. If we try to store value larger than the default maximum size, the value is just truncated to ensure high speed. Also, if an record is larger than the maximum record size, it must be manually split. We suggest adding a mechanism for dynamically splitting the values so that the data is not truncated and can be optimally aplit and stored in the system. We suggest that the value be broken in the maximum block sizes that are available on the page so that time is not wasted on paging [26].

6 RELATED WORK

Comparison of Key-value stores. Prior works have used YCSB to compare or provide detailed evaluation of variety of systems such as MongoDB and ElasticSearch [7]; Redis and SSDB [28]; Cassandra, HBase, PNUTS and MySQL [19]; Cassandra [6]; and more. We used these works to design our experimental setup using YCSB.

Comparison of Redis and Memcached. We found a few existing works that compared Redis and Memcached performances [8] [24] [15]. The work which is closest to our project [8] used YCSB to compare Redis, Memcached and Aerospike, but with uniform distribution for the different operation requests, which is not how regular requests are sent to key-value stores. Moreover, we found that when they performed the experiment, YCSB had no binding available for Redis and so they had created their own binding for it. We also noticed that their Redis client code was designed exactly like the Memcached client code. That is, they were handling the hash slots on the client side for Redis as well. This meant the client already knew which Redis node was responsible for handling which slots. However, normally, Redis cluster does not work that way. Only the server machines in the cluster are aware of the hash slots, not the clients. Furthermore, they mentioned that they had compared the systems for read heavy, write heavy and balanced workloads. Yet, they had used update operations as writes in their experiments. In YCSB client, the update operation only involves

Table 3: System Call Durations for Redis

Setup	100% Reads			100% Updates			100% Inserts		
	Average Latency	Redis Latency	YCSB Latency	Average Latency	Redis Latency	YCSB Latency	Average Latency	Redis Latency	YCSB Latency
Single Node (Thread 1)	236.43	0.36	13.912	146.97	0.216	8.792	412.93	0.528	17.624
Single Node (Thread 12)	977.55	0.2	23.008	862.38	0.192	15.288	1978.92	0.16	31.65
Cluster (Thread 12)	477.77	0.044	35.348	375.95	0.044	16.96	1282.57	0.128	35.82

Table 4: System Call Durations for Memcached

Setup	100% Reads			100% Updates			100% Inserts		
	Average Latency	Memcached Latency	YCSB Latency	Average Latency	Memcached Latency	YCSB Latency	Average Latency	Memcached Latency	YCSB Latency
Single Node (Thread 1)	280.248	65.37	39.43	162.43	49.128	28.8	290.05	109.672	30.272
Single Node (Thread 12)	526.6	52.744	61.332	362.05	34.73	41.54	500.15	62.49	50.072
Cluster (Thread 12)	462.97	32.8	39.912	241.15	18.4	23.064	474.81	46.4	41.28

updating a single field of a record whereas insert operations involve adding a completely new record to the database. They did not differentiate between these two operations which involve completely different work flows for both the systems. [24] compared Redis, Memcached, Cassandra, H2 and MongoDB with their own benchmarking sotware. Yet, they did not provide much information about their software to allow us to be convinced with their work. Finally, [15] used Memtier benchmark for their experiment which was designed by Redis Labs and thus had a chance of being biased.

Benchmarks for Comparison. We had also studied other benchmarks, like Memtier [31] and TPC-C [2] for our experiment. However, for reasons mentioned above, we did not use Memtier. On the other hand, TPC-C queries are designed for specific applications so generalization of their results is not possible. Hence, we avoided using it.

7 CONCLUSIONS

In this paper, we have studied and tried to summarize the architecture as well as memory management mechanisms of Redis and Memcached. We have also looked into their methods of handling requests in a single node mode or in cluster mode. With the insights from our study, we designed an experiment to evaluate the performances of the two systems using Yahoo! Cloud Serving Banchmark (YCSB).

We chose Redis and Memcached for our experiments since they both are very popular, easy to use and fast, despite having completely different memory management policies. YCSB was chosen as it is a de-facto benchmarking tool for cloud serving key-value stores and also highly customizable which was what we needed to analyze and confirm our theories about their performances.

We observed some key facts about the two systems through our experiments. We observed that Redis, took up less time when handling requests compared to Memcached. Yet, in several cases, due to it's single threaded nature, the average latency of operations would increase as the operation requests would have to wait longer time to be processed. Also, in cluster mode, Redis requires longer time due to more number of network communications per operation. On the other hand, Memcached was slower in processing requests mainly due to having to spend large amounts of time waiting for locked resources to be released.

ACKNOWLEDGMENTS

We would like to thank Dr. Tamer M. Ozsu for his continuous support and guidance towards us for completing this project. We would also like to thank Edward Chrzanowski for providing us with the infrastructure to complete this experiment.

REFERENCES

[1] [n. d.]. https://docs.oracle.com/cd/E17952_01/mysql-5.1-en/ha-memcached-using-memory.html#ha-memcached-fig-slabs
[2] [n. d.]. C - Homepage. http://www.tpc.org/tpcc/
[3] [n. d.]. Redis cluster tutorial. https://redis.io/topics/cluster-tutorial
[4] [n. d.]. *Transactions. https://redis.io/topics/transactions
[5] 2019. Memcached. https://en.wikipedia.org/wiki/Memcached
[6] Veronika Abramova, Jorge Bernardino, and Pedro Furtado. 2014. Evaluating cassandra scalability with YCSB. In *International Conference on Database and Expert Systems Applications*. Springer, 199–207.
[7] Yusuf Abubakar, Thankgod Sani Adeyi, and Ibrahim Gambo Auta. 2014. Performance evaluation of NoSQL systems using YCSB in a resource austere environment. *Performance Evaluation* 7, 8 (2014), 23–27.
[8] Anthony Anthony and Yaganti Naga Malleswara Rao. [n. d.]. Memcached, Redis, and Aerospike Key-Value Stores Empirical Comparison. ([n. d.]).

[9] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. 2011. Many-core key-value store. In *2011 International Green Computing Conference and Workshops*. 1–8. https://doi.org/10.1109/IGCC.2011.6008565

[10] Edwin F Boza, Cesar San-Lucas, Cristina L Abad, and Jose A Viteri. 2017. Benchmarking Key-Value Stores via Trace Replay. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 183–189.

[11] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, Scott Shenker, et al. 1999. Web caching and Zipf-like distributions: Evidence and implications. In *Ieee Infocom*, Vol. 1. INSTITUTE OF ELECTRICAL ENGINEERS INC (IEEE), 126–134.

[12] Brianfrankcooper. [n. d.]. brianfrankcooper/YCSB. https://github.com/brianfrankcooper/YCSB/wiki/Running-a-Workload

[13] Brianfrankcooper. [n. d.]. brianfrankcooper/YCSB. https://github.com/brianfrankcooper/YCSB/wiki/Running-a-Workload-in-Parallel

[14] Brianfrankcooper. 2018. brianfrankcooper/YCSB. https://github.com/brianfrankcooper/YCSB/

[15] Wenqi Cao, Semih Sahin, Ling Liu, and Xianqiang Bao. 2016. Evaluation and Analysis of In-Memory Key-Value Systems. *2016 IEEE International Congress on Big Data (BigData Congress)* (2016). https://doi.org/10.1109/bigdatacongress.2016.13

[16] Josiah L. Carlson. 2013. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA.

[17] S. Chen, X. Tang, H. Wang, H. Zhao, and M. Guo. 2016. Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis. In *2016 IEEE Trustcom/BigDataSE/ISPA*. 1660–1667. https://doi.org/10.1109/TrustCom.2016.0255

[18] Alibaba Cloud and Alibaba Cloud. 2018. Redis vs. Memcached: In-Memory Data Storage Systems. https://medium.com/@AlibabaCloud/redis-vs-memcached-in-memory-data-storage-systems-3395279b0941

[19] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 205–220.

[21] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '87)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/41840.41841

[22] dormando. [n. d.]. a distributed memory object caching system. https://memcached.org/about

[23] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.

[24] Abdullah Talha Kabakus and Resul Kara. 2017. A performance evaluation of in-memory databases. *Journal of King Saud University - Computer and Information Sciences* 29, 4 (Oct 2017), 520âĂŞ525.

[25] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC '97)*. ACM, New York, NY, USA, 654–663. https://doi.org/10.1145/258533.258660

[26] Oleku konko. 2013. olekukonko/MultipartCache. https://github.com/olekukonko/MultipartCache

[27] Patrick McElhaney. [n. d.]. List of standard lengths for database fields. https://stackoverflow.com/questions/20958/list-of-standard-lengths-for-database-fields

[28] Omoruyi Osemwegie, Kennedy Okokpujie, Nsikan Nkordeh, Charles Ndujiuba, Samuel John, and Uzairue Stanley. 2018. Performance Benchmarking of Key-Value Store NoSQL Databases. *International Journal of Electrical and Computer Engineering (IJECE)* 8, 6 (2018), 5333–5341.

[29] Jure Petrovic. 2008. Using memcached for data distribution in industrial environment. In *Third International Conference on Systems (icons 2008)*. IEEE, 368–372.

[30] Play-With-Docker. [n. d.]. play-with-docker/play-with-docker.github.io. https://github.com/play-with-docker/play-with-docker.github.io/blob/master/_posts/2017-12-30-redis-cluster.markdown

[31] RedisLabs. 2018. RedisLabs/memtierbenchmark. https://github.com/RedisLabs/memtierbenchmark

A FURTHER EXPLORATION

There were a lot more things we had planned to carry out for our project which we were unable to complete due to time constraints. A short description of those are explained below:

Scaling out. In this project, we used three node cluster for our experiments, which did not allow us to see a better demonstration of Zipfian distribution workloads. This is because with less number of nodes, the data will be less spread out.

Parallel Client Machines. As mentioned in section 4.2, we posit that the client machine was a bottleneck for our experiments and so to test that theory, we would like to re-run the experiments by scaling out the client machines.

Test Eviction Policies. Both Redis and Memcached have eviction policies to remove old data to make space for new ones. Redis offers six such policies while Memcached provides just one. Memcached has fixed slab and page sizes, which makes it obvious that evicting old data will not lead to memory fragmentation. However, that is not the case with Redis, which adds data to memory as it goes. This might lead to memory fragmentation and the system might run our of memory. This is something we want to test for both the systems to confirm.

Run more iterations. In our experiment, we ran the workloads only once per system per different parameters due to limited time. However, in this type of experiments which involves communication over the network, it is better to run multiple iterations of the workloads and average out the results in order to rule out scenarios of abnormal results. This is because the numbers can vary from iteration to iteration due to network lags and I/O operations. We would like to perform this step in future.

Other Variants of Workloads. YCSB allows testing for two more kinds of operations - Scan and Delete. We were not able to complete them in time but wish to perform tests by varying these operation proportions in order to have a complete evaluation of the two systems.