Evaluation and Analysis of In-Memory Key-Value Systems

Wenqi Cao, Semih Sahin, Ling Liu, Xianqiang Bao
School of Computer Science, Georgia Institute of Technology, Atlanta, Georgia, USA

***Abstract*—This paper presents an in-depth measurement study of in-memory key-value systems. We examine in-memory data placement and processing techniques, including data structures, caching, performance of read/write operations, effects of different in-memory data structures on throughput performance of big data workloads. Based on the analysis of our measurement results, we attempt to answer a number of challenging and yet most frequently asked questions regarding in-memory key-value systems, such as how do in-memory key-value systems respond to the big data workloads, which exceeds the capacity of physical memory or the pre-configured size of in-memory data structures? How do in-memory key value systems maintain persistency and manage the overhead of supporting persistency? why do different in-memory key-value systems show different throughput performance? and what types of overheads are the key performance indicators? We conjecture that this study will benefit both consumer and providers of big data services and help big data system designers and users to make more informed decision on configurations and management of key-value systems and on parameter turning for speeding up the execution of their big data applications.**

1. Introduction

As big data penetrating both business enterprises and scientific computing, in-memory key-value systems are gaining increased popularity in recent years. In contrast to disk-based key-value systems, such as HBase, Mongo, Cassandra, in-memory key-value systems, such as Redis [1], Memcached [2], rely on memory to manage all their data by utilizing virtual memory facility when the dataset to be loaded and processed is beyond the capacity of the physical memory. In memory key-value systems are about 6 orders of magnitude faster than disk-optimized key-value systems [16]. Thus, the ability to keep big data in-memory for processing and performing data analytics can have huge performance advantages.

The recent advances and price reduction in computer hardware and memory technology have further fueled the growth of in-memory computing systems. In-memory key-value systems become one of the most prominent emerging big data technologies and an attractive alternative to disk-optimized key-value systems for big data processing and analytics [1, 2, 15, 16]. However, there is no in-depth performance study with quantitative and qualitative analysis on in-memory key-value systems in terms of the design principle, the effectiveness of memory utilization, the key performance indicators for in-memory computing, and the impact of in-memory systems on big data applications.

In this paper, we present the design and the results of a measurement study on in-memory key-value systems with two objectives: First, we plan to perform a comparative study on two most popular and most efficient in-memory key-value systems, Redis [1] and Memcached [2], in terms of the in-memory data placement and processing techniques. This includes internal data structure in memory, fragmentation, caching, read and write operation performance. Second, we plan to provide experimental evaluation and analysis on the performance impact of different in-memory data structures on different workloads, including a comparison on commonality and difference in the design of Redis and Memcached with respect to memory efficiency. As a result of this study, we attempt to answer a number of challenging and yet most frequently asked questions regarding in-memory key-value systems through extensive and in-depth experimentation and measurements. Example questions include but not limited to: (1) What types of data structures are effective for in-memory key value systems? (2) How do in-memory key-value systems respond to the big data workloads, which exceeds the capacity of physical memory or the pre-configured size of in-memory data structures? (3) What types of persistence models are effective for maintaining persistency of in-memory key value systems? And (4) Why do different in-memory key value systems have different throughput performance and what types of overheads are the key performance indicators? To the best of our knowledge, this is the first in-depth comparative measurement study on in-memory key-value systems. We conjecture that this study will benefit both consumer and providers of big data services. The results of this study can help big data system designers and users make informed decision on configurations and management of key-value systems and on performance turning for speeding up the execution of their big data applications.

2. Overview

2.1. In-Memory Key-Value Systems

The key-value systems achieve high performance through three common systems-level techniques: (i) *parallel request handling*, (ii) *efficient network I/O*, and (iii) *concurrent data access*. Concretely, through hashing, key-value pairs can be distributed almost evenly to each compute node of a key-value system cluster. Client requests are served concurrently by accessing different servers with minimum

performance interference. To deal with the latency of network I/Os, key-value systems employ pipeline to support asynchronous request processing. It allows multiple requests to be sent to the server without waiting for the replies, and batch deliver the replies in a single step. In addition to utilize multiple CPU cores and multithreading for concurrent data access, key-value systems maintains data integrity using mutexes, optimistic locking, or lock-free data structure.

Unlike disk-optimized key-value systems, in-memory key-value systems rely on internal memory data structures to manage all application data and intelligently utilize virtual memory facility to handle large datasets that exceed the size of allocated physical memory. In-memory key-value systems also employ logging and snapshot techniques to provide persistency support. The in-memory key-value systems differ from one another in terms of their memory management strategies, such as memory allocation, virtual memory facility, concurrent access methods, and their persistency implementation models. To handle the volatility of DRAM memory and data loss due to reboot or power outage, in-memory key-value systems need to provide durability support by either implementing some persistent models, such as snapshot, transaction logging, or deploying non-volatile random access memory or other high availability solutions. However, the persistency model also introduces another source of system overhead.

2.2. Redis Overview

Redis is an open source, in-memory data structure store, used as database, cache and message broker. It is implemented in C and adopts jemalloc [7, 10] as its memory allocator since version 2.4, although its early versions use glibc [17] as the default memory allocator. The quality of a memory allocator is determined by how effectively it can reduce the memory usage for storing the same amount of dataset and alleviate the impact of fragmentation. Redis supports four sets of complex data structures to allow application developers to select the best memory data structures based on their dataset sizes and workload characteristics: (1) *Lists* - collections of string elements sorted by the order of insertion; (2) *Sets* - collections of unique, unsorted string elements; (3) *SortedSets* - similar to Sets but every string element is also associated with a floating number value, called score. The elements are sorted by their score; and (4) *Hashes* - which are maps composed of fields associated with values. For each of the four types of data structure, two implementations are provided, each with its unique characteristics. For example, linked list and ziplist are two implementations of the list data structure. Previously, Redis had its own virtual memory facility to handle swap-in and swap-out bypassing the OS virtual memory management. The latest version 4.2 has changed to provide the in-memory dataset to host all data in DRAM and let OS handle all swap-in and swap-out paging requests when the dataset size exceeds the allocated physical memory. We will analyze the reasons for this change in our measurement study. Redis also supports five different persistency models: *nosave*, *snapshot*, *AOF-No*, *AOF-Everysec*, and *AOF-Always*. *nosave* denotes no support for persistency. *snapshot* periodically takes snapshots of all the working dataset hosted in memory and then dumps the snapshots into persistence storage system stored as the latest snapshot files. The three AOF models refer to log-immediate, log-periodical and log-deferred, with Log-Periodical as the tradeoff of persistence and write performance between log-immediate and log-deferred.

2.3. Memcached Overview

Memcached is a general-purpose distributed memory caching system [2]. It is often used to speed up dynamic database-driven websites by caching data and objects in DRAM to improve the performance of reading from an external data source. Data items in Memcached are organized in the form of key and value pair with metadata. Given the varying sizes of data items, a naive memory allocation scheme could result in significant memory fragmentation problem. Memcached addresses this problem by combining glibc with slab allocator [9], since glibc alone cannot handle the memory fragmentation problems. In Memcached, the memory is divided into 1MB pages. Each page is assigned to a slab class, and then is broken down into chunks of a specific size for the given slab class. Upon the arrival of new data items, a search for the slab class of the best fit in terms of memory utilization is initiated. If this search fails, a new slab of the class is allocated from the heap and otherwise the new item is pushed into a chunk in the chosen slab class. Similarly, when an item is removed from the cache, its space is returned to the appropriate slab, rather than the heap. Memory is allocated to slab classes based on the initial workload and its item sizes until the heap is exhausted. Therefore, if the workload characteristics change significantly after the initial phase, the slab allocation may not be appropriate for the workload, which can results in memory underutilization. Memcached only supports key-value pairs without sorting or complex data structures. Also it does not support any persistency model like Redis does, No failure and data loss recovery in Memcached due to reboot and power outage.

Memcached stores key and value pairs along with entry structure within a single allocation, reducing memory overhead, improving access locality, lowering memory allocation pressure (frequency), and reducing fragmentation. However, this may also cause some problems: First, it is not straightforward to extend Memcached to support diverse data structures like Redis, because this fundamental design constrains its extension. Second, with slab allocator, although Memcached no longer needs to allocate memory as frequently as Redis, the use of slab allocator incurs more computation on memory allocation and leads to a more rigid memory allocation policy.

3. Methodology

3.1. Workload Design

YCSB [3] is a popular workload generator for disk-resident key-value systems. Given that in-memory key-value systems are emerging in recent years, only a few benchmarking tools support both Redis and Memcached,

and unfortunately YCSB does not. In our measurement study, all workloads are generated by *Memtier_benchmark* [13] produced by RedisLab, or GIT_KVWR, developed by authors using Jedis interacting with Redis.

Memtier_benchmark is used to generate two types of workloads: (1) each record is 100×10 bytes and (2) each record of 1000×10 bytes. The 1st set of experiments uses the record size of 1KB and the 2nd set of experiments uses the record size of 10KB. For both sets of experiments, *Memtier_benchmark* is used to generate 5GB and 10GB workloads, corresponding to 5,000,000×1KB records and 10,000,000×1KB records respectively.

GIT_KVWR is designed for generating workloads to measure the performance of the manifold data structure options in Redis, since none of the existing benchmarks, including *Memtier_benchmark*, can create workloads to measure the performance for all four types of data structures.

Experimental Platform Setup. All experiments were performed on a server platform with 3.20GHz Intel Core-i5-4460 CPU (4 cores and 4 hyperthreads), which has 32KB L1d caches, 32KB L1i caches, 256KB L2 cache, and 6144KB L3 caches, 16GB 1600MHz DDR3 memory, and a Samsung SATA-3 250 GB SSD. Ubuntu 14.04 with Linux kernel 3.19.0-25-generic is installed. We use Redis 3.0.5 standalone mode with default settings, and Memcached 1.4.25 with the configuration of maximum 1024 connections and maximum 20480MB (20GB) memory allocation.

3.2. Evaluation Models

Our evaluation examines four types of performance overheads commonly observed in key-value systems.

Measuring the overheads of handling large scale datasets. We use the bulk insertion workloads for this type of overhead measurement. For the 5GB (light load) and 10GB (heavy load) workloads generated by *Memtier_benchmark*, we track the throughput and fine-grained CPU and memory consumptions for three types of activities (user-level, kernel system level, I/O level) during insertion phase and generate fine-grained system performance statistics using *SYSSTAT* [4] and *Perf* [12]. Moreover, we also examine the memory swapping overheads during bulk insertion of 10GB workloads by allocating only 8GB physical memory. We examine both swap-in and swap-out activities and compare the physical memory utilization with the amount of swapping memory pages.

Measuring the overheads of different persistency models. Snapshot and logging are the two popular persistency models. Although it is widely recognized that supporting persistency can introduce performance overheads for in-memory key-value systems, very few studies the efficiency of different persistency models and how they impact on the performance of different workloads and how they compare with the scenarios of no persistency support in terms of CPU and memory consumptions. We measure *nosave* in Redis as no-persistency support scenario, *Snapshot*, and three different logging scenarios: *AOF-No* for log-deferred, *AOF-Everysec* for log-periodic, and *AOF-Always* for log-immediate.

Measuring data structure overheads. First of all, there is no detailed documentation on internal data structures for Redis and Memcached. In order to conduct in-depth measurement on overheads of different in-memory data structures, we go through the source code of both Redis and Memcached in order to identify and analyze the detailed layout and implementation of different internal data structures. In addition, we also measure the space overhead, the cache overhead, the overhead of read and write operations on each of the data structures, and the fragmentation overhead of different data structures.

3.3. Evaluation Metrics

We collect six system-level and three application-level characteristics from workloads using *SYSSTAT*, *ltrace*, *Valgrind*, *Perf* and Redis internal benchmarking commands. These characteristics collectively reveal the underlying causality details of various performance overheads, enabling an in-depth understanding of different performance overheads inherent in the in-memory key-value systems. Concretely, the following different workload characteristics are collected under different workloads, different persistency models, and different data structures:

CPU utilization (%): Three metrics are used to measure CPU utilization in user space, kernel space or storage I/O are *%user*, *%system* and *%iowait*, which show the CPU overhead occurred for performing tasks in user space, kernel space or storage I/O.

Memory and Swap usage (KB): These metrics are measuring the dynamics of memory usage and swap events per second.

Swap-in and Swap-out (page/sec): These metrics capture the number of page swap-out events and swap-in events per second, reflecting the overhead of swapping and virtual memory management during memory shortage.

L1 and Last Level Cache miss rate (%): These metrics refer to the miss rate of L1 and the miss rate of LLC cache during read operation execution for different data structures, showing the efficiency of cache line locality.

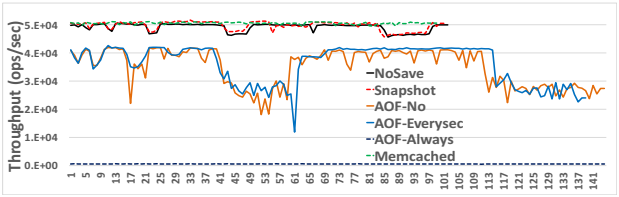
Memory fragmentation rate: this metric reflects the degree of memory fragmentation, which is calculated as the amount of memory currently in use divided by the physical memory actually used (the RSS value).

Heap allocation (byte): we measure both the total amount of memory and the total number of blocks allocated by application in heap, showing the efficiency of application memory allocation policy.

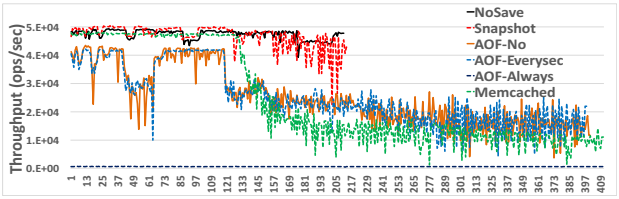
Throughput (ops/sec): this metric shows the number of operations per second for a given workload, reflecting the efficiency of request processing.

Data structure overhead (byte): we measure the maximum and the minimum space overhead of entry insertion, and these two metrics clearly reflect the space overhead of different data structures.

Operation execution time (μsec): this metric captures the CPU time of baseline read/write operation execution for a given data structure under a given workload and dataset, allowing the performance comparison for different data structures, different datasets and workloads.



(a) Light load



(b) Heavy load

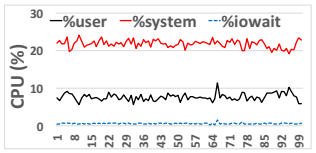
Figure 1: The throughput of load with different overhead.

4. Experimental Results and Analysis

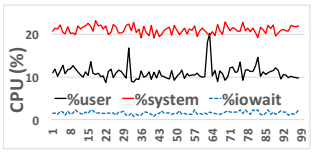
4.1. Overhead for Bulk Insertion of Large Datasets

In this set of experiments, we present the measurement results for dictionary expansion overhead for two scenarios: (i) bulk insertion of dataset within the allocated memory capacity and (ii) bulk insertion of dataset exceeding the allocated memory capacity. One internal threshold is set in Redis and Memcached for controlling when dictionary expansion is necessary. Once the threshold is met or exceeded, the system will start creating a double-size hash bucket and gradually moving data from existing bucket to the new one. To understand the impact of the dictionary expansion overhead on the throughput of Redis workloads, we measure the throughput of both 5GB and 10GB workloads. Figure 1(a) shows five dents, showing that the 5GB workload experienced 5 times of dictionary expansion. Table 1 shows that the average throughput with dictionary expansion is 46597 ops/sec and the average throughput without dictionary expansion is 50206. Thus the throughput degradation due to dictionary expansion is about 7.19% (i.e., 50206-46597)/50206=0.0719).

Another interesting observation from Figure 1(a) is that the Memcached throughput is more stable for the same workload, even with dictionary expansion. This is because Redis is using single thread for request serving, and this single thread has to deal with both data loading and dictionary expansion. However, Memcached uses multi-threading for request serving, and dedicates one exclusive expanding thread, always ready to perform dictionary expansion task, reducing the overhead significantly. Although they both use the similar rehashing strategy – incremental rehashing, Memcached with multi-threading solution clearly outperforms Redis with single-thread solution. Table 2 presents the average throughput detail for Memcached under 5GB (non-swapping) and 10GB (swapping) workloads. Figure 2(a) and Figure 2(b) show that the CPU utilization of Memcached is less stable compared to Redis, with several spikes for CPU *%user* level metric. Figure 1(b) shows that with 10GB workloads, the allocated memory of 8GB can no longer host all the data in memory. When persistency models are turned

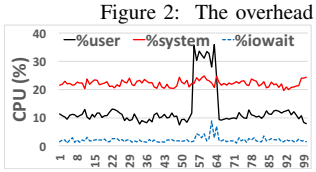


(a) Redis CPU Utilization

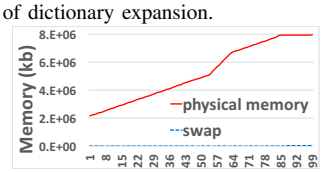


(b) Memcached CPU Utilization

Figure 2: The overhead of dictionary expansion.



(a) CPU Utilization



(b) Memory consumption

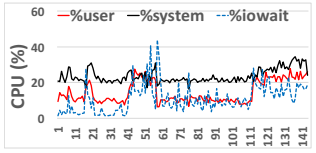
Figure 3: The overhead of *snapshot*.

on, the throughput performance of all persistency models is decreasing sharply at the elapse time of 125th sec or so, compared to no persistency support scenario (NoSave). We will analyze the impact of different persistency models on workload performance in the next subsection.

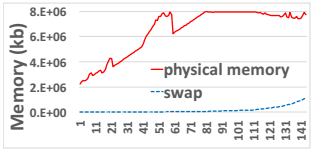
4.2. Overheads of Persistency Models

4.2.1. Snapshot Overhead. Redis performs *snapshot* from elapse time 55th to 63rd second as shown in Figure 1(a). The average throughput overhead is about 1%. The reason for such a small and negligible overhead is that Redis forks a child process by using copy-on-write for the background save purpose, thus loading data and snapshotting can run concurrently. As a result, the performance of snapshot is surprisingly good in Redis. To further understand the overhead of snapshot, we zoom in the period from 55th to 63rd second in Figure 3(a), observing that snapshotting generates spikes for both CPU *%user* and CPU *%iowait*. The spike of CPU consumption to over 30% in user space shows the changes caused by the background RDB (*snapshot*) save process, which is forked from Redis main process, responsible for writing log files to disk and running in user space. The small spike in CPU *%iowait* means that the writing of snapshot files to disk (SSD in our test platform) are causing some I/O waits compared to the time period with no snapshotting. Figure 3(b) shows the sudden increase of memory consumption at 55th second, which is again caused by the background RDB save process. Although Redis adopts copy-on-write for eliminating additional memory consumption, the memory change rate is still significant with 209.51% increase (i.e., (168753-54522)/54522=2.0951).

4.2.2. AOF-No Overhead. *AOF-No* means never *fsync*, and instead relying on the Operating system for flushing log files. We also refer to this model as log-deferred



(a) CPU Utilization



(b) Memory consumption

Figure 4: The overhead of *AOF-No*.

TABLE 1: Overhead of Redis during Bulk Insertion

	Overhead	Throughput	CPU *%user*	CPU *%system*	CPU *%iowait*	Memory/Swap changing (KB/sec)	Swap-in (page/sec)	Swap-out (page/sec)
non-swapping (5GB)	nosave	50206	7.39	21.43	0.66	54522	n/a	n/a
	dict expansion	46597	7.49	21.52	0.67	58558	n/a	n/a
	snapshot	49703	32.17	23.4	3.79	168753	n/a	n/a
	AOF-No	41670	9.88	21.95	2.68	87489	n/a	n/a
	AOF-No + Rewriting	25419	21.61	24.18	17.27	187148	n/a	n/a
	AOF-Everysec	41649	9.545	21.51	3.57	88896	n/a	n/a
	AOF-Everysec + Rewriting	28018	21.89	25.08	18.27	197104	n/a	n/a
swapping (10GB)	nosave	46937	10.48	22.7	3.4	50028	245	12939
	dict expansion	42335	11	22.86	11.05	46701	171	12646
	snapshot	45272	26.04	24.19	23.12	63241	4758	19471
	AOF-No	40842	9.62	21.64	10	44654	221	12380
	AOF-No + Rewriting	19465	16.43	24.81	28.97	50146	2375	13178
	AOF-Everysec	40813	10.02	20.26	7.1	44876	259	12397
	AOF-Everysec + Rewriting	19460	16.48	24.73	28.7	50972	2401	13642

TABLE 2: Overhead of Memcached during Bulk Insertion

	Overhead	Throughput	CPU *%user*	CPU *%system*	CPU *%iowait*	Memory/Swap changing (KB/sec)	Swap-in (page/sec)	Swap-out (page/sec)
non-swapping (5GB)	baseline	50611	10.45	20.69	1.51	59337	n/a	n/a
	dict expansion	50461	15.16	20.92	1.62	60630	n/a	n/a
swapping (10GB)	baseline	34250	12.25	20.05	7.61	32978	203	11990
	dict expansion	13922	6.66	8.61	36.36	13922	2435	13685

model. Compared to *nosave* mode (no persistency support), recall Figure 1(a) and Table 1, we observe that *AOF-No* incurs 17% throughput degradation (i.e., (50206-41670)/50206=0.17). The *AOF* file gets bigger as more write operations are performed, so Redis provides *AOF Rewrite* to rebuild the *AOF* in the background without interrupting its client services. Whenever *BGREWRITEAOF* is issued or *AOF* file is big enough, Redis will generate the shortest sequence of commands needed to rebuild the current dataset in memory. We call this *AOF Rewrite*, which happens from 17^{th} to 20^{th} second and from 41^{st} to 58^{th} second in Figure 1(a), showing a nearly 50% throughput degradation using statistics collected in Table 1. Since during *AOF Rewrite* Redis rewrites all commands into a new temp file, while new commands are stored in *server.aof_rewrite_buf_blocks*. The temp file will finally replace the original *AOF* file and data stored in *server.aof_rewrite_buf_blocks* will open the new *AOF* file as append mode and insert into the end of it. All these tasks create more computation and I/O cost, resulting in significant performance degradation. *AOF Rewrite* combined with dictionary expansion can make the average throughput even worse as shown during the elapse time duration from 41^{st} to 58^{th} second in Figure 1(a). This is also a good example to show that multiple overheads combined together could lead to seriously worse performance result.

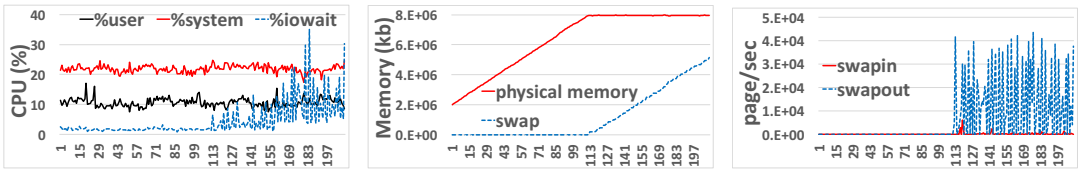
Figure 4(a) and Figure 4(b) further zoom into the overhead of *AOF-No*. By Table 1, for *AOF-No*, its CPU *%user* increases by 33.69%; CPU *%system* increases by 2.43%; and CPU *%iowait* increases by 306.06%. This is because additional operations for *AOF* writes (writing *server.aof_buf data* into disk file) leads to more user space computation. Whenever these writes flushed to disk by operating system, it results in large CPU *%iowait*. For the memory consumption of *AOF-No*, by Table 1, *AOF-No* allocates 60.47% more memory space than *nosave* mode (i.e., (87489-54522)/54522=0.6047), which are needed for storing all executed but not yet persisted commands in *server.aof_buf*. When *AOF-No* is combined with *AOF rewrite*, it allocates 243.25% more memory space than *nosave* mode (i.e., (187148-54522)/54522=2.4325), because the mass of I/O operations created by *AOF rewrite* occupies huge amount of memory buffer and cache space. Figure 4(b) shows the memory consumption changes during the following elapse time durations: 2^{nd}-3^{rd} second, 7^{th}-8^{th} second, 17^{th}-20^{th} second, 41^{st}-58^{th} second, and 112^{th}-141^{st} second.

The overhead *AOF-Everysec* is very similar to that of *AOF-No*, because they both uses almost same *AOF* code, except *AOF-Everysec* executing *fsync()* every second.

4.3. Swapping Overhead

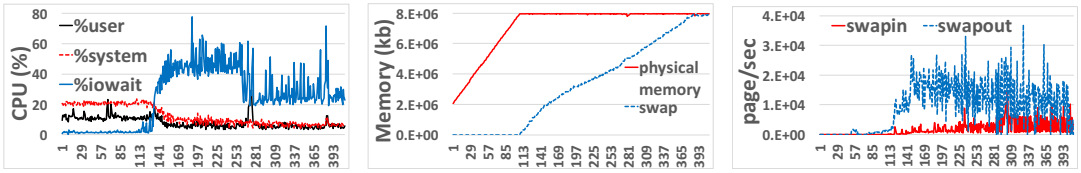
Recall Figure 1(b), the workloads of 10GB dataset under the available 8GB memory allocation will incur swapping during the bulk insertion. We below analyze the swapping overhead under dictionary expansion, snapshot and AOF-No. For ***dictionary expansion***, the throughput degradation of Redis is about 9.15% compared with non-swapping scenario according to Table 1. From Figure 1(b), the degradation occurs from 177^{th} to 202^{nd} second. More interestingly, the impact of swapping on Memcached is even bigger, and the throughput degradation of Memcached is about 72.41%, compared to non-swapping scenario. The throughput of Memcached decreases to about 13,922 ops/sec by Table 1, which is lower than the lowest throughput for the Redis *AOF* model. From Figure 5(c) and Figure 6(c), we observe that after the 111^{th} second, due to the shortage of physical memory, operating system starts to swap-out some data into disk in order to load new data. Bulking loading and *dictionary expansion* fiercely compete memory resource. Also Memcached shows a lot more swap-in pages than Redis, a reason for sharp performance drop of Memcached as shown in Figure 1(b).

For ***snapshot***, the throughput of *snapshot* with swapping is 45272 and the throughput of *nosave* with swapping is 46937. We conclude that snapshot consumes very little system resource, and generates almost same throughput as *nosave* mode. However, snapshotting while expanding dictionary can significantly slow down the system, as shown in Figure 1(b) and Figure 7. In this case, CPU *%iowait* increases to about 23.12%. In Figure 7(b), the slope of mem-



(a) CPU Utilization (b) Memory consumption (c) swapping

Figure 5: The overhead of Redis *dictionary expansion* with swapping.



(a) CPU Utilization (b) Memory consumption (c) swapping

Figure 6: The overhead of Memcached *dictionary expansion* with swapping.

ory changing with swapping is 37.48% of non-swapping case. This is partly caused by the slow swap allocation and the huge swap-out operations, as shown in Figure 7(c).

For ***AOF-No*** and ***AOF-Everysec***, similar throughput can be observed. By Table 1, the throughput overhead of *AOF-No*, is 12.99%, compared to *nosave* with swapping. Figure 8 provides the measurements of CPU utilizations, memory usage and swapping events for *AOF-No*. It is observed that the physical memory is full at the 80$^{\text{th}}$ second and the swapping starts. After 119$^{\text{th}}$ seconds, *AOF rewrite* starts running, the memory is totally full at the 119^{th} second. This further illustrates sharp decrease of *AOF-No* throughput at 119^{th} second in Figure 1(b). Moreover, as *AOF rewrite* job includes large amount of read and write operations, operating system has to swap-in and swap-out constantly as shown in Figure 8(c), leading to memory thrashing, which seriously worsens the system performance.

The insight we gained from this set of experiments is four folds. First, adding one additional thread dedicated to dictionary expansion could improve throughput for in-memory key-value systems like Redis. Second, *AOF-Rewrite* is a CPU and memory intensive task, which results in obvious degradation of system performance. If swapping also occurs, it can generate severely worse throughput. Third, swapping seriously impacts on the performance of Memcached. Finally, *snapshot* based persistency model in comparison shows the best throughput performance compared with other persistent models for both non-swapping and swapping workloads.

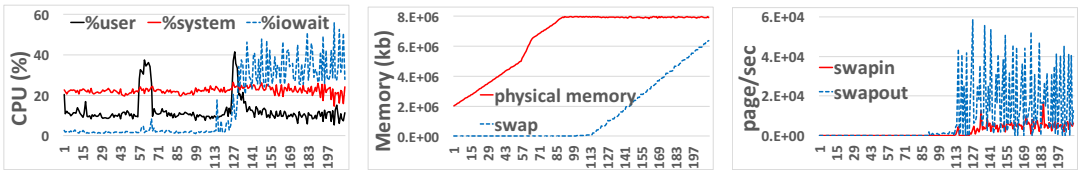
4.4. Overhead of Different Data Structures

We examine four types of overheads for different data structures: memory space overhead, cache overhead, read/write operation overhead and fragmentation.

4.4.1. Space Overhead. For the same dataset, the memory space allocated by *linkedlist* is 3.02 times of the size of *ziplist*. By using the compact internal structure, *ziplist* saves lots of physical memory. Although *linkedlist* is more flexible for insertion and deletion operations, it uses several additional data structures. Figure 9(a) shows that the size of *linkedlist RDB* file is only 25.87% of its memory allocation. These design principles also apply to *set*, *sortedset* and *hash*. Figure 9(b) shows that *Intset* memory allocation is only 11.73% of *hashtable*. *Intset AOF* file is about 2.4 times bigger than its memory allocation (Figure 9(b)). Data Structure of *skiplist* can be viewed as a combination of *hashtable* and *skiplist* to enhance both search and sort performance. However, the drawback is obvious on its high memory consumption. It occupies 7.50GB for 1GB dataset (see Figure 9(c)), which is highest memory allocation among all data structures supported by Redis. In comparison, *ziplist* is a good choice, since for 1GB dataset, it only occupies 2.08GB memory. For *hashtable*, although it has great search performance, it consumes too much space. For 1GB dataset, it occupies about 6.10GB memory (Figure 9(d)), 6 times of the original dataset size. Figure 10(a) shows that Memcached uses 1.26% and 1.30% more memory than Redis for loading 1,000,000 and 2,000,000 dataset records generated by *memtier_benchmark*, each of 1KB.

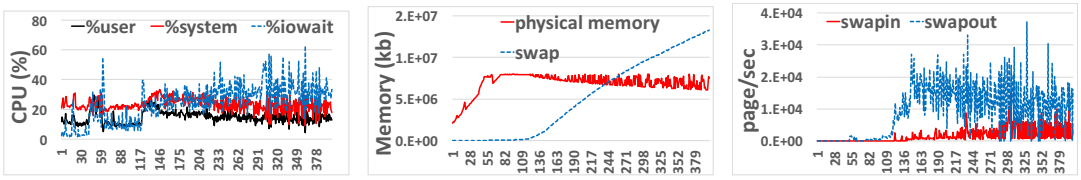
4.4.2. Caching Overhead. Caching locality is one of key factors when designing system data structure since main memory reference is 200× lower than L1 cache reference. Array like data structures, *ziplist* and *intset*, are contiguous memory blocks, so large chunks of them will be loaded into the cache upon first access. *linkedlist*, *hashtable*, and *skiplist* on the other hand are not in contiguous blocks of memory, and could lead to more cache misses.

4.4.3. Read/Write Operation Performance. Different data structures have their own pros and cons. Table 3 compares various data structures with read/write operations. For list data structures, we mesured the performance of push/pop operation on both left-side and right-side, since *ziplist* shows maximum 3 times performance variation on different side operation, which is caused by the internal design of *ziplist*. For set data structures, we found that performance of *intset* operations is sensitive to the size of dataset, unlike *hashtalbe*. For sortedset data structures, we tested the add, remove, check, and iterate operations, showing that *skiplist* outperfoms *ziplist* for all operations except iterate operation, since the continuous memory structure makes *ziplist*



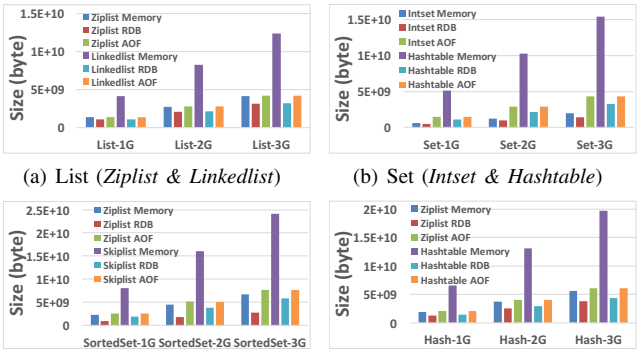
(a) CPU Utilization (b) Memory consumption (c) swapping

Figure 7: The overhead of *snapshot* with swapping.



(a) CPU Utilization (b) Memory consumption (c) swapping

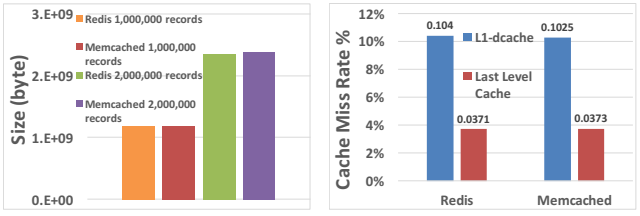
Figure 8: The overhead of *AOF-No* with swapping.



(a) List (*Ziplist & Linkedlist*) (b) Set (*Intset & Hashtable*)

(c) Sortedset (*Ziplist & Skiplist*) (d) Hash (*Ziplist & Hashtable*)

Figure 9: Memory and storage consumption of different data structures.



(a) Space overhead (b) Caching overhead

Figure 10: Space and Cache comparison between Redis and Memcached.

better suited for scanning records. For hash data structures, we benchmarked set, get, remove, and iterate operations, showing that *hashtalbe* always keeps excellent performance for expanding dictionary. In summary, *linkedlist* performs well for both experiments, the performance has nearly no connection with dataset size except *lrange* operation. Since *ziplist* needs additional time to move memory, it shows worse result as the increasing number of entries, especially for *lpush* and *lpop* operations.

Finally, we compare Redis and Memcached by using *set* and *get* operation. Figure 11 shows that the performance difference between Redis and Memcached is only 0.0059%. The *set* throughput of Redis is unstable, which is caused by *dictionary expansion* and its single thread design. Redis has better performance than Memcached without *dictionary expansion*. But when dictionary expansion occurs, Memcached outperforms Redis on *set* operation by 1.08%. This further proves that optimization for dictionary expansion, such as using a dedicated thread, is critical for improving throughput performance of Redis.

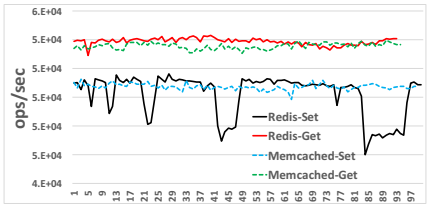


Figure 11: *set/get* performance of Redis and Memcached

4.4.4. Fragmentation. We measure the heap allocation for comparing the fragmentation of Redis and Memcached as shown in Table 4. The total allocation for Redis is slightly larger than that of Memcached, since Redis uses lots of structs representing different intermediate objects, which is relatively more complex than Memcached. However, Redis releases 7.19% (1,000,000 records test) and 7.49% (2,000,000 records test) memory during its runtime and Memcached only releases 0.58% (1,000,000 and 2,000,000 records test), thus the final memory usages of both are almost the same. As a result, Memcached uses 1.27% (1,000,000 records test) and 1.32% (2,000,000 records test) more memory than Redis. Moreover, since Redis and Memcached both have similar memory usage and Redis allocates 5449 (1,000,000 records test) and 5795 (2,000,000 records test) times more memory blocks than Memcached, it shows that Redis allocates lots of small memory blocks and however Memcached allocates only huge blocks. The average block size of Redis is 181 (both tests) bytes and that of Memcached is 933320 (1,000,000 records test) and 993398 (2,000,000 records test) bytes, showing a totally different memory management strategy between Redis and Memcached. From this set of experiment, we make the following important observations: (1) Array-like structure makes *ziplist* and *intset* good for memory consumption and

TABLE 3: Operation Performance (all results are average CPU time (usec) consumed by commands)

	Dataset-1 (each record = 100 × 10byte)					Dataset-2 (each record = 1000 × 10byte)				
List Data Structure	rpush	rpop	lpush	lpop	lrange	rpush	rpop	lpush	lpop	lrange
ziplist	0.8	0.84	1.01	1.07	5.04	0.93	1.01	2.91	2.92	45.5
linkedlist	0.63	1	0.65	0.98	4.84	0.69	1.21	0.69	1.12	38.44
Set Data Structure	sadd	scard	smembers	sismember	srem	sadd	scard	smembers	sismember	srem
intset	1.01	0.52	13.98	0.73	0.85	1.82	0.66	114.61	1.02	1.59
hashtable	1.28	0.56	28.33	1.23	1.11	1.12	0.69	238.21	1.1	0.97
SortedSet Data Structure	zadd	zrem	zrange	zrank	zscore	zadd	zrem	zrange	zrank	zscore
ziplist	3.11	2.63	5.91	1.91	3.74	14.36	15.67	55.08	10.92	12.34
skiplist	1.73	1.64	9.92	1.95	2.71	2.02	2.41	84.98	2.88	2.9
Hash Data Structure	hset	hget	hgetall	hdel	hexists	hset	hget	hgetall	hdel	hexists
ziplist	2.33	1.49	10.29	1.94	1.99	15.33	7.13	152.89	11.19	14.95
hashtable	1.67	1.32	15.25	1.42	1.16	1.19	1.01	204.36	1.32	1.18

TABLE 4: Heap Measurement

key-value system	Memory Usage (byte)	Memory Wasted (byte)	Heap						
			Total Allocation (byte)	Total Block Allocated	Max Live Allocation (byte)	Max Live Block	Num of Malloc()	Num of Free()	Block Size (byte)
Redis1,000,000 records	1,180,042,265	na	1,271,538,266	7,013,016	1,112,739,857	4,011,659	7,001,265	3,000,691	181
Memcached 1,000,000 records	1,195,074,650	102,131,302	1,201,184,070	1,287	1,193,318,638	1,276	1,136	4	933320
Redis 2,000,000 records	2,356,863,304	na	2,547,742,990	14,013,327	2,226,148,187	8,011,918	14,002,762	6,001,225	181
Memcached 2,000,000 records	2,388,001,817	203,214,028	2,402,036,870	2,418	2,385,782,830	2,406	2,266	5	993398

cache locality at the expense of low performance for big dataset. (2) The operation performance is really related to specific data structure and workload. (3) The performance of right-side operations (*rpush, rpop*) of *ziplist* are always better than that of left-side operations (*lpush, lpop*). (4) Thanks to *dictionary expansion*, *hashtable* might even show better performance with bigger dataset. (5) *skiplist* is a lot faster than *ziplist* for large datasets, such gap cannot be compensated by the use of continuous memory space. (6) By using different memory management policy, jemalloc vs. slab allocator and arbitrary allocation vs. pre-allocation, both Redis and Memcached achieve good performance while reducing the fragmentation.

5. Conclusion

We have presented a performance evaluation and analysis of in-memory key-value systems. To the best of our knowledge, this is the first in-depth measurement study on critical performance and design properties of in-memory key-value systems, such as the use of different internal data structures, different persistency models and different policies for memory allocators. We measure a number of typical overheads such as memory space, caching, read/write operation performance, fragmentation, and workload throughput performance, to illustrate the impact of different data structures and persistency models on the throughput performance of in-memory key-value systems. We conjecture that the multiple factors on memory efficiency will provide system designers and big data users with a better understanding on how to configure and tune the in-memory key-value systems for high throughput performances under different workloads and internal data structures.

Acknowledgement: This research has been partially support by the National Science Foundation under Grants IIS-0905493, CNS-1115375, NSF 1547102, SaTC 1564097, and Intel ISTC on Cloud Computing, and an RCN BD Fellowship, provided by the Research Coordination Network (RCN) on Big Data and Smart Cities. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the RCN or National Science Foundation. The last author is a visiting PhD student while collaborating on this work, funded by China Scholarship Council.

References

[1] Redis [Online]. Available: http://redis.io
[2] Memcached [Online]. Available: http://memcached.org/
[3] Cooper, Brian F., Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking cloud serving systems with YCSB." In Proceedings of the 1st ACM symposium on Cloud computing, pp. 143-154. ACM, 2010.
[4] Godard, Sbastien. "Sysstat: system performance tools for the Linux OS, 2004."
[5] Nethercote, Nicholas, and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." In ACM Sigplan notices, vol. 42, no. 6, pp. 89-100. ACM, 2007.
[6] Berezecki, Mateusz, et al. "Power and performance evaluation of memcached on the tilepro64 architecture." Sustainable Computing: Informatics and systems 2.2 (2012): 81-90.
[7] Evans, Jason. "A scalable concurrent malloc (3) implementation for FreeBSD." In Proc. of the BSDCan Conference, Ottawa, Canada. 2006.
[8] Zhang, Hao, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. "in-memory big data management and processing: A survey." (2015).
[9] Bonwick, Jeff. "The Slab Allocator: An Object-Caching Kernel Memory Allocator." In USENIX summer, vol. 16. 1994.
[10] Evans, Jason. "Scalable memory allocation using jemalloc." (2011).
[11] Branco, Rodrigo Rubira. "Ltrace internals." In Linux symposium, p. 41. 2007.
[12] de Melo, Arnaldo Carvalho. "The new linux'perf'tools." In Slides from Linux Kongress. 2010.
[13] Redis Labs. Memtier Benchmark. https://github.com/RedisLabs/memtier benchmark.
[14] Atikoglu, Berk, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. "Workload analysis of a large-scale key-value store." In ACM SIGMETRICS Performance Evaluation Review, vol. 40, no. 1, pp. 53-64. ACM, 2012.
[15] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in Proc. 11th USENIX Conf. Netw. Syst. Des. Implementation, 2014, pp. 429444.
[16] Drepper, Ulrich. "What every programmer should know about memory." Red Hat, Inc 11 (2007).
[17] glibc [Online]. Available: https://www.gnu.org/software/libc/