Persistence and Recovery for In-Memory NoSQL Services: A Measurement Study

Xianqiang Bao*†
*HPCL & School of Computer,
National University of Defense Technology,
baoxianqiang@nudt.edu.cn

Ling Liu† and Wenqi Cao†
†School of Computer Science,
Georgia Institute of Technology,
{lingliu,wcao39}@cc.gatech.edu

Nong Xiao*and Yutong Lu*
*HPCL & School of Computer,
National University of Defense Technology,
{nongxiao,ytlu}@nudt.edu.cn

***Abstract*—NoSQL systems are deployed as the core components for delivering big data Web services today. With growing main memory capacity, we witness the growing interest and deployment of in-memory NoSQL services (*IM-NoSQL*), which are designed to maximize the utilization of DRAM for ultra-low latency services. To address the volatility of DRAM for in-memory computing services, persistence and failure recovery are important functionality for IM-NoSQL. In this paper we report an extensive measurement study on the performance of persistence and recovery for IM-NoSQL. We evaluate the performance and effectiveness of several common mechanisms used for persistence and recovery in the presence of server crashes, such as snapshot and logging based approaches. Through this study, we are able to answer some of the most frequently asked questions in provisioning of IM-NoSQL services: (i) Can an IM-NoSQL system work effectively when the available memory is insufficient to load the whole dataset? (ii) What is the overhead of maintaining snapshot compared to logging? (iii) How fast an IM-NoSQL system can recover in the presence of failure? And (iv) how does an IM-NoSQL system respond to the different persistence models? We report our comprehensive measurement results on execution, persistence and recovery performance of Redis, a representative implementation of IM-NoSQL services.**

1. Introduction

NoSQL (Not only SQL) systems are attractive and widely deployed for big data driven Web services. Successful examples include Bigtable [9] at Google; Dynamo [10] at Amazon; HBase [11] at Facebook and Yahoo!; Voldemort [12] at Linkedin and so forth. As the memory capacity increases and the unit price of DRAM decreases in recent years, in-memory NoSQL systems (*IM-NoSQL*) become vital components for many Internet-scale Web services [5, 9, 10] offering ultra-low latency [1, 2, 14]. Examples include Memcached [2], Redis [1], MongoDB [19] (partially in-memory) from industry; RAMCloud [15], MemC3 [20], FaRM [22], MICA [23] from academia. Although Non-Volatile Memory (NVM) [25] is gaining increasing attention, most IM-NoSQL systems are built on volatile DRAM. Thus, the persistence and recovery become critical components for IM-NoSQL to provide continued data services in the presence of server crashes or system failures.

There are two types of persistence and recovery models commonly used for in-memory data management systems [1, 14–18]. They are *Snapshot* and *Log* based approaches. In comparison, log-based systems are slightly more popular. For example, RAMCloud [15] uses a logging approach similar to log-structured file systems while Redis [1] supports both snapshot and logging and uses *Snapshot* way by default. [16] and [18] focus on logging scalability. Memcached [2] is designed as an in-memory cache system without persistence and MongoDB [19] is partially in-memory system using memory-mapped file. Moreover, many existing research efforts on persistence and recovery of data stores [16–18] have primarily been focused on SQL-based transaction systems. As IM-NoSQL becomes widely deployed in real world applications, both NoSQL developers and users often need to make a choice on specific persistence and recovery mechanism when configuring their NoSQL systems for ensuring runtime reliability of their applications [13, 21].

In this paper we conduct an extensive measurement study on the performance of persistence and recovery for IM-NoSQL under different workloads. We evaluate the performance and effectiveness of several common mechanisms used for persistence and recovery in the presence of NoSQL server crashes, such as snapshot and logging based methods. We report our comprehensive measurement results on execution, persistence and recovery performance of NoSQL workloads. We choose Redis [1] as our target system in this study because Redis is widely recognized as the most efficient implementation of IM-NoSQL systems. It is worth mentioning that existing research efforts on persistence and recovery of database services have been primarily focused on relational SQL-based transactional systems [16–18]. To the best of our knowledge, this work is the first extensive comparative study on different persistence models, their performance impact on typical NoSQL workloads, and their performance impact on recovery efficiency for NoSQL-based in-memory systems. Moreover, as emerging byte-addressable, non-volatile memory (NVM) [25] changes the design principle of main memory systems due to new memory hierarchy, researchers argue that one should rethink the design of persistence and recovery methods [6, 7, 16, 24]. Thus this in-depth measurement study will offer insights which are instrumental to the design of efficient persistence

and recovery mechanisms for NVM based NoSQL systems.

2. Overview

Here we describe five types of persistence models and two types of recovery models that are most popularly used in IM-NoSQL.

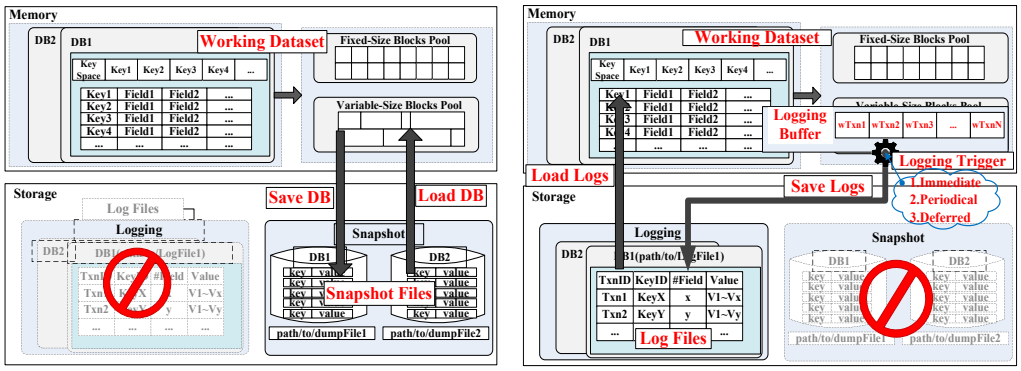
2.1. The Persistence Models

The persistence requirements for current IM-NoSQL typically depend on the data persistence requirements of hosted applications. Some systems have no persistence support. For example, Memcached [2] is a popular distributed in-memory cache system that it treats the memory as data cache and thus its design and implementation do not incorporate persistence support at all. On the other hand, some other in-memory systems, such as Redis [1] has strong and customizable support for persistence. The common persistence approach is to employ snapshot or logging to flush every update of the in memory data into persistent storage.

2.1.1. *Snapshot* Persistence. When an IM-NoSQL system uses the *Snapshot* model to achieve persistence, the system periodically takes a snapshot of all the working dataset hosted in memory and then dumps the snapshot into the persistence storage as the latest snapshot file. Typical time intervals are defined by a given number of seconds such as 900 seconds, 300 seconds, and so forth. Some NoSQL systems also support aperiodic snapshot triggering conditions based on other parameters such as the number of write operations that have occurred such as every 10 writes or every 100 writes. It is also possible to combine periodic time-based triggers with aperiodic update condition based triggers, such as triggering snapshot event every 300 seconds or every 10 write operations, whichever occurs the first. Figure 1(a) shows a sketch of the *Snapshot* persistence model. The snapshot process consists of the following two steps to generate persistent snapshots: (1) *Snapshot trigger*: This step determines when to take a snapshot. If the trigger condition is aperiodic with a given number of writes as the threshold, then the trigger manager continuously or periodically checks with the update profiling manager, which records the number of write requests and maintains a time interval for all writes from the previous snapshot or the system start. Whenever it detects that there are enough writes occurred during a certain time interval, the snapshot process will be triggered and the snapshot save (flush) begins. The read/write APIs are typically implemented as *get/set* for NoSQL. So the statistics for write requests used by the snapshot trigger can be directly collected from the number of *set* that is invoked. (2) *Snapshot save*: Upon triggering the snapshot action, the target NoSQL system dumps the snapshot to the persistence storage and stores it as the most recent snapshot file. More specifically, the target system forks to generate both child and parent processes. The parent process will continue the routine operations of the NoSQL system whereas the child process is responsible of performing snapshot and it starts to write the working dataset to a temporary snapshot file in the persistent storage. Upon completing the writing of the last data entry to the temporary snapshot file, the new version of the snapshot file is generated and it replaces the old version. The child process uses the copy-on-write mechanism to dump in-memory structured dataset into the temporary snapshot file, so the parent process can still handle write requests without halt. And the main instance only needs to stop when performing the old snapshot file replacement. Also, the in-memory structured dataset is typically compressed before dumping to achieve storage I/O efficiency and space saving.

2.1.2. *Log* Persistence. When the *Log* model is used to achieve persistence, the system first records every write request received by the server as a *log* record entry and writes it to the log buffer before it exercises the update operation. When the log buffer reaches a pre-defined threshold, the logging process is triggered to flush the log buffer to a log file stored in the persistence storage. The logging trigger threshold determines when to flush the log buffer and the setting of this threshold also has significant impact on the performance of write-intensive workloads and the persistence guarantee of the dataset currently being updated. The larger this threshold is, the less frequently the logging will be flushed to the persistent storage, the less performance impact by the logging based persistence on the performance of routine NoSQL workloads, and the weaker persistence guarantee for the dataset currently being updated.

The *Log* model can be implemented differently and at different levels of granularity for persistence support. We below describe three different modes of implementations. (1) Immediate flushing (*Log-Immediate*): When a log record is created after the NoSQL server receives a write request, the logging process immediately flushes the log record to the log file. In this *Log-Immediate* persistence model, the logging trigger is fired upon each write request. This may lead to a longer delay for each write request under *Log-Immediate*, because the server has to be blocked and wait for the corresponding log record to be appended to the log file first by the write ahead logging (WAL) principle. This performance degradation is the worst when the log file is stored on the slow persistent storage media such as HDD, because the disk I/O can easily become the bottleneck. One advantage of immediate flushing is that the logging process only needs to allocate a small size log buffer. (2) Periodical flushing (*Log-Periodical*): The log buffer is periodically checked by the logging process at the pre-defined interval set at the system configuration time (e.g., one second for Redis [1]). If there are newly inserted log records in the log buffer, the logging process will flush the log buffer to the log file by log-append operations. Thus, the logging trigger is responsible for checking the time interval and the status of log buffer. Under *Log-Periodical*, a write request is completed after the corresponding log record has been successfully appended to the log buffer. Also, the log buffer uses copy-on-write mechanism to dump log records into the log file. The drawback of this model is that it does not provide the write ahead logging (WAL) guarantee. (3) Deferred flushing (*Log-Deferred*): It typically has two steps, first the logging process continuously creates log records to log incoming updates in the log buffer hosted in memory, and



(a) Snapshot persistence model (b) Log persistence model

Figure 1: Typical persistence models for IM-NoSQL.

then relies on the operating system to manage the log buffer flushing. Thus, it is assumed that the operating system will be responsible for flushing its write buffer and its log buffer to the persistence storage. Similar to the *Log-Periodical*, a write request under *Log-Deferred* is completed after its log record has successfully appended into the log buffer. This model shares the same drawback as *Log-Periodical*. If the operating system fails to flush the log buffer before a server crash, then the persistence guarantee is up to the previous flushing.

In comparison, *Log-Immediate* with immediate log flushing requires the shortest log buffer flushing interval and achieves highest working dataset persistence, but the cost is also the highest, because its write performance is significantly decreased due to immediate logging of every write request to the slow storage I/O stack. Although *Log-Periodical* and *Log-Deferred* both batch the log records to be flushed, the interval of *Log-Deferred* relies on the operating system setting to flush the write buffer. It can be much longer than the periodical-interval under *Log-Periodical*. Thus, the deferred log flushing may have the longest log buffer flushing interval and achieves lowest level of persistence, but *Log-Deferred* offers high write performance. *Log-Periodical* can be seen as a tradeoff method between high persistence and high write performance compared to *Log-Immediate* for high persistence with low performance and *Log-Deferred* for high performance with low persistence. We argue that *Log-Periodical* may be employed by IM-NoSQL to achieve high write performance with acceptable data loss risk.

2.1.3. *NoSave* without Persistence. To gain in-depth understanding of different implementations of snapshot and logging based persistence, we also include the *NoSave* model as a naive baseline model with no persistence support. For *NoSave*, all the working dataset (i.e., database data and the index data) is hosted in memory. The operating system may utilize its memory swap area when the dataset is bigger than the physical RAM size and it can fit into OS swap area. The drawback of this *NoSave* model is that when the NoSQL system is shutdown due to crashes or other reasons, all the working dataset hosted in the physical RAM as well as the swap area will be lost because of no persistence support. Obviously, *NoSave* offers better throughput performance than any of the snapshot or logging based persistence models, especially when the runtime environment has fierce resource competitions, which may lead to performance bottlenecks such as the working dataset is bigger than physical RAM and the OS swapping (page fault) is involved.

2.2. The Recovery Models

2.2.1. Snapshot-based Recovery. When the target NoSQL system needs to restart from a server instance crash or shutdown, under the *snapshot* recovery model, the server will perform two steps to recover the dataset from the most recent snapshot file: (i) loading the complete snapshot file into memory, and (ii) reconstructing the working datasets: (1) *Snapshot file loading*: When the server instance restarts, by detecting that if the *snapshot* persistence model is configured, the server starts the snapshot-based recovery process. First, the recovery process locates the snapshot file and checks the file status, including whether the file is completed, whether it is the right file or right version for this instance, and so on. Then the recovery process starts loading the snapshot file by streaming reads (snapshot file is usually stored as binary format), and each time single or a batch of key-value formatted records are loaded into memory. (2) *Working dataset reconstruction*: After the records have been loaded into memory, if the records are compressed, they will be uncompressed first. Then the recovery process directly inserts these records into the new created in-memory data structure, such as hash table. Also the index for these records will be created at the same time. If the snapshot file is big but still can be handled in memory through OS swapping, then the OS memory swap area will be used upon page faults. The frequent memory page swapping may lead to a performance degradation of the reconstruction process. However, when the snapshot file is too big to be handled in memory, the NoSQL system will experience the out of memory error because the recovery process runs out of memory space, including the OS swap area. Thus, the recovery process will halt the working dataset reconstruction task. Hence, it is very important to estimate the available memory capacity based on the snapshot file size. Only when the whole dataset has been reconstructed successfully, the NoSQL server instance can resume its handling of newly arrived workload requests.

TABLE 1: Typical persistence & recovery implementation in Redis system.

Generic Models		Redis Implementation			
		Persistence Implementation		Recovery Implementation	
		Model	Parameter Config	Model	Parameter Config
NoSave		*NoSave*	[save " "]; [appendonly no]	*N/A*	*N/A*
Snapshot		*Snapshot(DRB)*	[save 900 1]; [save 300 10]; [save 60 1000]; [appendonly no] (*Default*)	*Snapshot(RDB)*	*Default*
Log	*Log-Deferred*	*AOF-No*	[save " "]; [appendonly yes]; [appendfsync no]	*AOF-Rewrite/ AOF-NoRewrite*	*auto-aof-rewrite-percentage 100/0*
	Log-Periodical	*AOF-Everysec*	[save " "]; [appendonly yes]; [appendfsync everysec]		
	Log-Immediate	*AOF-Always*	[save " "]; [appendonly yes]; [appendfsync always]		

TABLE 2: Setup of testbed.

CPU	Intel Core i3, 2.6 GHz
CPU cores	2 (4 with Hyper-threading)
Processor cache	L1–128 KB, L2–512 KB, L3–3 MB
DRAM–8GB	8GB (4GB×2, 1600 MHz DDR3)
DRAM–16GB	16GB (8GB×2, 1600 MHz DDR3)
Storage: SSD	SATA-3 with 3Gb/s (Intel 320 series)
OS	Ubuntu 14.04 with kernel version 3.11.0+

2.2.2. Log-based Recovery. When a NoSQL server restarts the system under a log-based persistence model including *Log-Immediate/Periodical/Deferred*, the log-based recovery process needs to reconstruct the whole working dataset by replaying all the log records in two steps: (1) *Log file loading*: Similar to the snapshot file loading, the recovery process needs to first locate the log file and check the file status. Normally, the log file is stored in text format and each log record contains the write command as well as the operands. Each time, single or a batch of log records will be loaded into the memory. (2) *Log records replay*: Reconstructing the working dataset from the text formatted log file is very different from the binary formatted snapshot file. During log records replaying, the NoSQL server will rely on the recovery client maintained by the server under the log-based persistence configuration. After the log records are loaded into memory, the recovery client will parse out the write commands, including the operands from the corresponding log record entry, and redo the write command to reconstruct the matching key-value record. The whole working dataset is successfully reconstructed after all the log records are replayed by the recovery client. Also, when the log file is big and exceeds the memory capacity of the NoSQL server, the recovery performance will decrease sharply, and the recovery may fail to complete.

3. Measurement Study: Design Guidelines

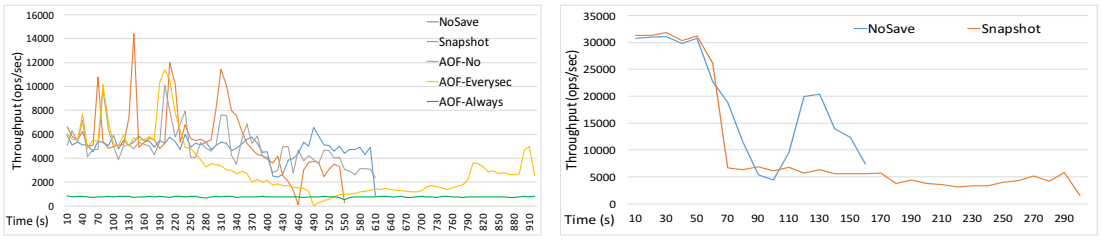
The main objectives of this measurement study are to gain in-depth understanding of the performance impact of different persistence models and the effectiveness of snapshot based recovery model and logging based recovery model. Through this study, we want to answer some of the most frequently asked questions with respect to the research and development of IM-NoSQL systems: How does an in-memory NoSQL system respond to the different persistence models? Can an in-memory NoSQL system work effectively when the available memory is insufficient to load the whole dataset? What is the overhead of maintaining snapshot compared to logging? And how fast an in-memory system can recover in the presence of failure?

We set up the testbed using two servers of two different DRAM sizes: 8GB and 16GB respectively. The choice of the DRAM sizes allows us to study how in-memory NoSQL performs under sufficient DRAM memory v.s. under insufficient memory. Table 2 shows the details of the measurement setup. We use Yahoo! Cloud Serving Benchmark (YCSB) [3, 8] to generate target NoSQL workloads.

Measurement Metrics. We want to measure the performance of an in-memory NoSQL write workloads in the presence of sufficient DRAM and insufficient DRAM with respect to the size of the dataset to be loaded into the NoSQL store. We use the open source tool *SYSSTAT* [4] to measure and analyze when and how the CPU or memory resource bottleneck occurs. The CPU bottleneck can be detected from the CPU utilization of user level activities (*%user*) because all the NoSQL workloads are running as the user level processes. We can detect the memory bottleneck by measuring the DRAM utilization (*%memused*), and spot the storage I/O bottleneck by measuring the CPU time used for I/O requests (*%iowait*), measuring the major page faults (*majflt/s*), and whether swapping starts (*%swpused*).

Persistence and recovery performance measurements. Redis [1] is chosen as our target IM-NoSQL system for two reasons. First, Redis is widely recognized as the fastest and most representative IM-NoSQL system. Second, Redis provides support for both snapshot based persistence model and logging based persistence model. Table 1 shows a summary of the persistence and recovery implementation in Redis. Specifically, we can disable the persistence module of Redis to implement the *NoSave* scenario. In the *NoSave* case, the whole dataset is stored in memory and the OS-managed swap storage partition when the main memory is insufficient to host the whole dataset. Data is not persistent when Redis is shutdown or crashes.

When the *Snapshot* model is configured as the persistence model for Redis, the configuration should also specify the time interval for periodic snapshot triggering the snapshot and the number of writes for threshold based snapshot triggering. For example, the condition of 900sec or 1 write defines that the snapshot should be triggered every 900 seconds or upon receiving one write request. Also, the working dataset is flushed to the persistent storage as *RDB* file by the *rdbSave* process with default configuration. When the persistence model is snapshot based, the snapshot based recovery model will be used upon failure for recovery from the *RDB* file. For Redis, the *rdbLoad* process is used to load *RDB* file and reconstruct the working dataset. Redis also implements the logging based persistence models represented by *AOF-No/Everysec/Always* as shown in Table 1, corresponding to *Log-Deferred/Periodical/Immediate* models. Redis stores the log file as the append only file (AOF) with or without rewriting. Thus, the log based recovery model is implemented as *AOF-Rewrite* or *AOF-NoRewrite*.



(a) Light-weight load (b) Heavy-weight load

Figure 2: The realtime throughput with different persistence.

4. Measurement and Evaluation Results

4.1. Persistence

Here we first run different persistence models with both light-weight workloads and heavy-weight workloads, generated by 1/16 YCSB threads. We vary the raw dataset size from 1GB (1 million records, and 1KB size for each record), 4GB (4 million records) and 8GB (8 million records). Figure 2(a) and Figure 2(b) measure the throughput performance results (ops/sec) by running all five persistence models using the light-weight workloads and the heavy-weight workloads respectively. For light-weight workloads, we mean that the physical memory (DRAM) is sufficient to host the whole dataset of 4GB. In contrast, the heavy-weight workloads refer to the cases in which we observe that the physical memory (DRAM) is insufficient to host the whole dataset of 8GB using the server setup with 8GB DRAM.

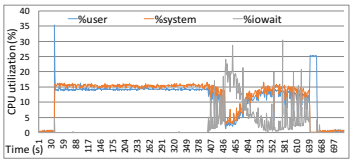
We make a number of interesting observations: First, under the light-weight workloads, data can be loaded successfully under all five persistence models. However, the throughput of *AOF-Always* is the lowest though relatively stable. This is because each write operation will bypass the write buffer and sync (by invoking *fsync()*) to the persistent storage, so the low speed storage media such as HDD or SSD always becomes the bottleneck compared with DRAM. This can be further observed by the CPU (*%iowait*) trace results in Figure 9(a). Second, the throughput of *AOF-Everysec* is the next worst compared to *AOF-No*, *Snapshot* and *NoSave* configuration. This is because log flushing is triggered every second, which is costly compared to (i) *AOF-No*, which let the OS determine when to flush the data to the persistent storage, or (ii) *Snapshot*, which periodically triggers the snapshot of the whole working dataset and flushes it to the persistent storage. As a result, the throughput of *AOF-Everysec* is seriously affected by insufficient DRAM and can decrease drastically when memory swapping starts as shown in Figure 2(a). Third and more interestingly, we observe that the throughput performance of *Snapshot* and *AOF-No* is unstable with frequently ups and downs. This real-time throughput volatility is primarily caused by the sudden increase and release of the CPU, DRAM and I/O resources, which results in making the data loading performance for *Snapshot* and *AOF-Everysec* faster than *NoSave*. Finally, under the heavy-weight workload, the DRAM is insufficient to host the whole dataset of 8GB. All *AOF-No/Everysec/Always* logging models cannot finish the dataset loading compared to *NoSave*, the *Snapshot* persistence model shows unstable throughput performance due to the real-time throughput volatility (Figure 2(b)). Concretely, the throughput decreasing is due to the fact that the operating system (OS) has to swap some of the dataset from memory to disk swap partition, and handle partial working dataset in the swap space. We can see from Figure 5(a) and Figure 6(a) that the CPU time (*%iowait*) used for I/O processing increases sharply when the memory swapping starts. From Figure 5(a), even DRAM is enough, the storage I/O is consumed by flushing the working dataset into a snapshot file. The CPU utilization (*%iowait* and *%user*) may suddenly increase and drops when flushing starts and finishes respectively. Similar situation happens for *AOF-Everysec* as the log buffer is periodically flushed when DRAM is enough.

Moreover, from Figure 2(b), we can see the realtime throughput of *Snapshot* decreases sharply when DRAM is not enough and maintains the low throughput steadily until the end. While the realtime throughput of *NoSave* model can rise again after swapping becomes stable (see major page faults patterns in Figure 4(c)), and I/O resource competition under *NoSave* is not as high as under the *Snapshot* model, as shown in Figure 4(c) and Figure 6(c).

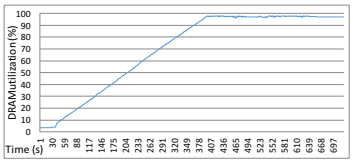
4.2. Recovery

The recovery measurement mainly focuses on recovery time and collects the trace results during recovery. Then we further analysis the recovery bottlenecks based on the trace results. As the logging files of *AOF-No/Everysec/Always* are the same, so the recovery processing from each logging model is also the same. While as the logging file rewrites can impact the log file size as well as logging records organisation much, we dig into recovery processing from logging file with/without rewrite (*AOF-Rewrite*/*AOF-NoRewrite*).

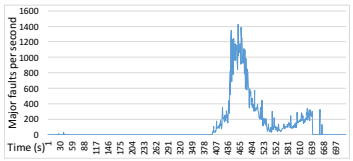
As shown in Figure 10, we can see different DRAM size has significant impact on recovery time when doing recovery from persistence files with different dataset size. For 1Million case with 1GB raw dataset size in our setup, both *DRAM-8GB* and *DRAM-16GB* can handle all the working dataset in DRAM, so the recovery time is similar and the bottleneck is from CPU (similar situation to 4Million case of *DRAM-16GB*, we will explain the details in the following 4Million case with trace results). For 4Million case with 4GB raw dataset size, the recovery speed of *DRAM-16GB* is much faster than *DRAM-8GB*. And the specific recovery speedup for *Snapshot/AOF-Rewrite/AOF-NoRewrite* model is 1.82x/2.09x/2.15x. Then we dig into the 4Million case with trace results, as shown in Figure 12 for



(a) *NoSave*-CPU(Light)

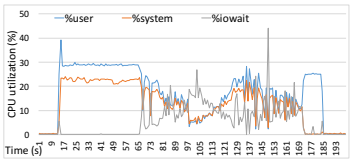


(b) *NoSave*-DRAM(Light)

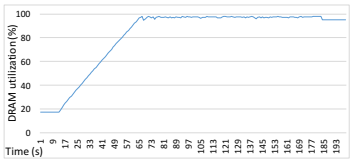


(c) *NoSave*-MajorFault(Light)

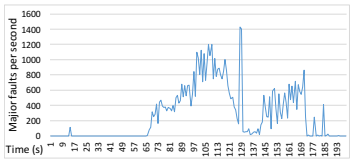
Figure 3: Trace results of *NoSave* (Light).



(a) *NoSave*-CPU(Heavy)

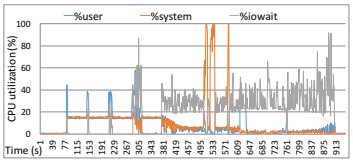


(b) *NoSave*-DRAM(Heavy)

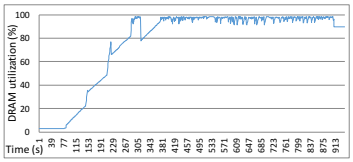


(c) *NoSave*-MajorFault(Heavy)

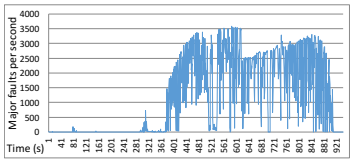
Figure 4: Trace results of *NoSave* (Heavy).



(a) *Snapshot*-CPU(Light)

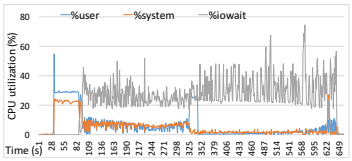


(b) *Snapshot*-DRAM(Light)

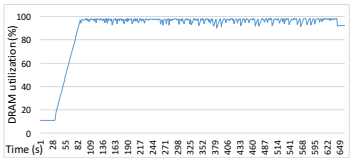


(c) *Snapshot*-MajorFault(Light)

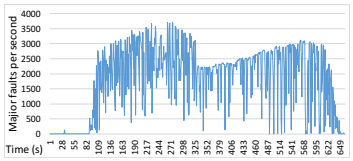
Figure 5: Trace results of *Snapshot* (Light).



(a) *Snapshot*-CPU(Heavy)

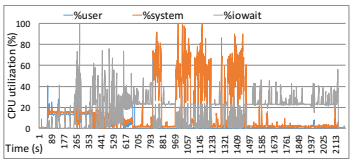


(b) *Snapshot*-DRAM(Heavy)

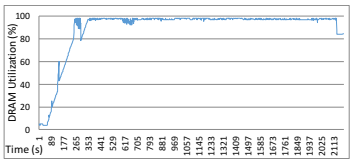


(c) *Snapshot*-MajorFault(Heavy)

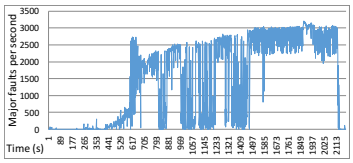
Figure 6: Trace results of *Snapshot* (Heavy).



(a) *AOF-No*-CPU(Light)

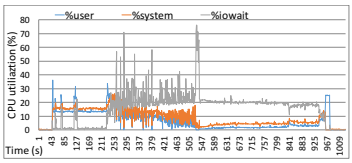


(b) *AOF-No*-DRAM(Light)

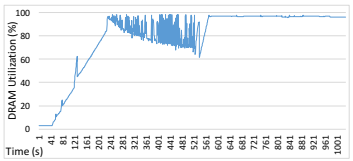


(c) *AOF-No*-MajorFault(Light)

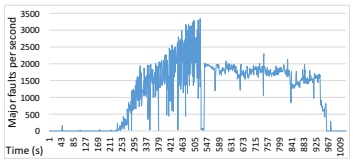
Figure 7: Trace results of *AOF-No* (Light).



(a) *AOF-Everysec*-CPU(Light)

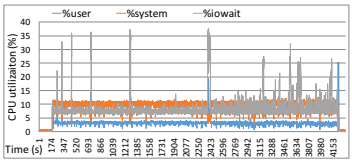


(b) *AOF-Everysec*-DRAM(Light)

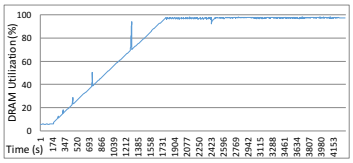


(c) *AOF-Everysec*-MajorFault(Light)

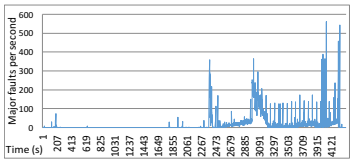
Figure 8: Trace results of *AOF-Everysec* (Light).



(a) *AOF-Always*-CPU(Light)



(b) *AOF-Always*-DRAM(Light)



(c) *AOF-Always*-MajorFault(Light)

Figure 9: Trace results of *AOF-Always* (Light).

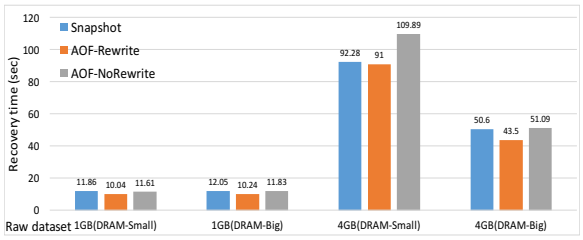
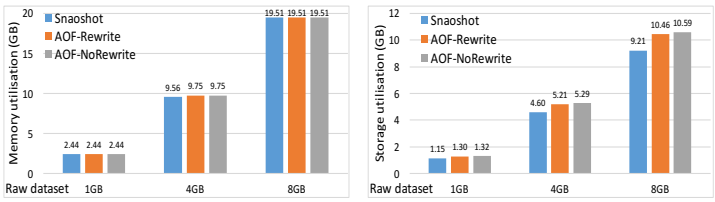
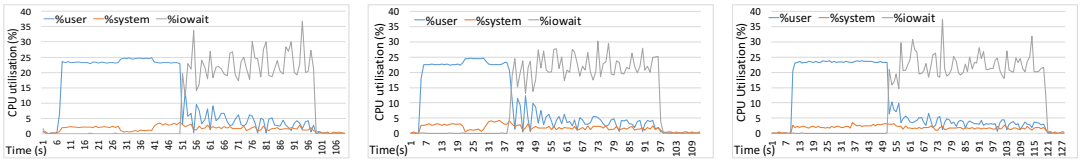


Figure 10: Recovery time with different DRAM size.



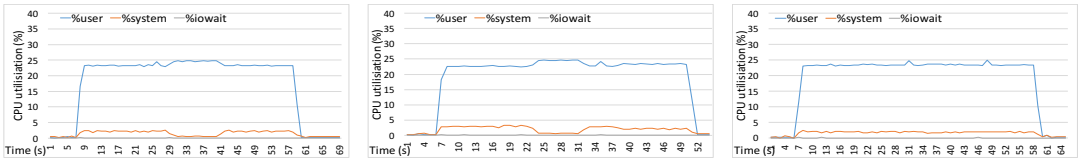
(a) Memory usage (b) Storage usage

Figure 11: Memory and storage usage of different dataset.



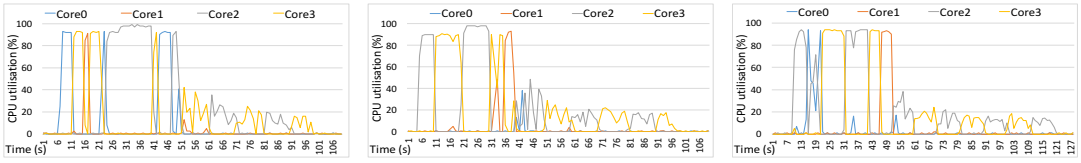
(a) *Snapshot*-CPU (b) *AOF-Rewrite*-CPU (c) *AOF-NoRewrite*-CPU

Figure 12: Recovery trace from different persistence files with *DRAM-8GB*.



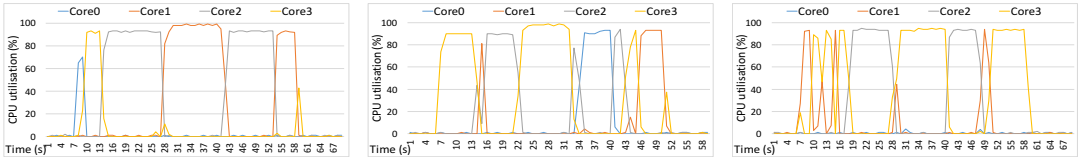
(a) *Snapshot*-CPU (b) *AOF-Rewrite*-CPU (c) *AOF-NoRewrite*-CPU

Figure 13: Recovery trace from different persistence files with *DRAM-16GB*.



(a) *Snapshot*-CPU (Core level) (b) *AOF-Rewrite*-CPU (Core level) (c) *AOF-NoRewrite*-CPU (Core level)

Figure 14: Recovery trace of CPU core level from different persistence files with *DRAM-8GB*.



(a) *Snapshot*-CPU (Core level) (b) *AOF-Rewrite*-CPU (Core level) (c) *AOF-NoRewrite*-CPU (Core level)

Figure 15: Recovery trace of CPU core level from different persistence files with *DRAM-16GB*.

DRAM-8GB, we can see much higher CPU utilisation for handling outstanding storage I/O requests (*%iowait*) begins when DRAM is not enough to host the new constructed working dataset and swapping is involved to use virtual memory (see Figure 12(a) for *Snapshot*, Figure 12(b) for *AOF-Rewrite* and Figure 12(c) for *AOF-NoRewrite*). While compared with trace results for 4Million case of *DRAM-16GB* in Figure 13, we can see no CPU time is idle for storage I/O requests. Moreover, from Figure 10, we can see logging file rewriting helps to speedup the recovery for both DRAM scales as expected. And here we reconstruct the dataset from the logging file generated only by handling insert workload, if update workload is also added then more speedup from rewriting can be achieved. Another interesting result is that although persistence file of *Snashot* is always smaller than *AOF-Rewrite* (see Figure 11(b)), but the recovery time of *Snapshot* is longer than *AOF-Rewrite*.

Moreover, as shown in Figure 14 and Figure 15, we verified that Redis is the single thread system as only one CPU core is used for each moment. And for each running Redis instance, when DRAM is enough the capability of a single CPU core is critical for recovery performance, and can easily become the bottleneck if the single core capability is not enough powerful, such as our setup here that each core of Intel i3 from hyper-threading is not that powerful one. So although the CPU utilisation (*%user*) is just using 25% of the whole CUP source as shown in the Figure 13, but one core is already fully used (almost 95∼100%) as shown in Figure 15 (one out of the four cores). When DRAM is not enough and swapping starts, then storage I/O can easily become the bottleneck and the CPU bottleneck per core is not existing anymore. So as shown in Figure 12, also

though the CPU utilisation for storage I/O request (*%iowait*) is approximate 25% (one out of four cores) but there has been I/O bottleneck during the recovery processing for *DRAM-8GB*. What's more, the network performance is also critical for recovery processing and can easily become the bottleneck especially when DRAM and CPU become very powerful. Because network latency has significant impact of the round trip time for Internet service based IM-NoSQL, such as Redis to redo the logging records, especially when doing recovery under distributed runtime environment.

4.3. Resource Usage

As we have already been detailed discussed the CPU utilisation, here we focus on the memory and storage space usage. We use *DRAM-16GB* environment to finish these experiments, and use single YCSB thread to load and generate 1GB, 4GB and 8GB raw datasets. Then we use statistic tool of Redis client (*"Redis-cli> info memory"*) to collect memory information from running instance, and use linux shell to get persistence file size. From Figure 11, memory usage is about 2.2x of the raw dataset size as shown in Figure 11(a). While the storage usage is about 1.1x∼1.2x of the raw dataset. More specifically, the snapshot file size (*Snapshot*) is 1.1x and logging file size (*AOF*) is around 1.2x of raw dataset size, so the *Snapshot* persistence model saves 10% space) compared with *AOF* model. Here we want to mention that although logging file rewriting for *AOF-Rewrite* only gets about 2% storage space saving compared with *AOF-NoRewrite*, but as we only use insert workload in this measurement and if update workload is added then rewriting mechanism can save much more space. Another point we want to mention is that, when the whole working dataset has been persisted in storage, the compression functionality provided by Redis can get rid of almost 100% space for both *Snapshot* and *AOF* models. However, the cost is that a lot of CPU resource is consumed by uncompressing the persistence files when need to do recovery.

5. Conclusion

We have conducted an extensive measurement study on the performance of different persistence and recovery models for IM-NoSQL services. We evaluate the performance and effectiveness of several common mechanisms used for persistence and recovery upon server crashes, such as snapshot and logging. We report our comprehensive measurement results on execution, persistence and recovery performance of Redis [1]. A number of interesting observations are made. First, NoSQL services with persistence support does not necessarily have low throughput performance compared to the *NoSave* case. Second, IM-NoSQL can leverage OS swapping to deal with the problem of insufficient DRAM. Third, the internal representation of the raw dataset in a NoSQL system is typically two times bigger than the raw dataset size, thus, even the raw dataset can fit into the DRAM, the memory swapping may still occur. Finally, the recovery efficiency of an IM-NoSQL service is seriously constrained by the DRAM capacity. We conjecture that the results of this measurement study provide some deeper insights on effectively configuring and tuning IM-NoSQL in the presence of server failure.

Acknowledgments

This work is carried out when Xianqiang Bao is a joint PhD student in Georgia Institute of Technology supported by the scholarship from CSC. Authors from National University of Defense Technology (NUDT) are partially supported by 863 Program of China under Grant No.2015AA015305 and the NSF of China under Grant Nos. 61433019 and 61232003. The authors from Georgia Tech are partially support by the National Science Foundation under Grants IIS-0905493, CNS-1115375, NSF 1547102, SaTC 1564097, and Intel ISTC on Cloud Computing.

References

[1] Redis. http://redis.io/
[2] Memcached. http://memcached.org/
[3] YCSB. https://github.com/brianfrankcooper/YCSB
[4] SYSSTAT. http://sebastien.godard.pagesperso-orange.fr/
[5] NoSQL databases. http://www.nosql-database.org/
[6] J. Huang, K. Schwan, M. K. Qureshi. NVRAM-aware Logging in Transaction Systems. Proc. of the VLDB Endowment, Vol. 8, No. 4, 2014.
[7] J. Arulraj, A. Pavlo, S. R. Dulloor. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. ACM SIGMOD 2015, pp.707–722.
[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, et al. Benchmarking Cloud Serving Systems with YCSB. ACM SoCC 2010, pp.143–154.
[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, et al. Bigtable: A Distributed Storage System for Structured Data. USENIX OSDI 2006, pp.205–218.
[10] G. DeCandia, D. Hastorun, M. Jampani, et al. Dynamo: Amazons Highly Available Key-Value Store. ACM SOSP 2007, pp.205–220.
[11] HBase. http://hbase.apache.org/
[12] Voldemort. http://project-voldemort.com/
[13] X. Bao, L. Liu, N. Xiao, et al. Policy-driven Configuration Management for NoSQL. IEEE CLOUD 2015.
[14] H. Zhang, G. Chen, B. C. Ooi, et al. In-Memory Big Data Management and Processing: A Survey. IEEE TKDE 2015.
[15] D. Ongaro, S. M. Rumble, R. Stutsman, et al. Fast Crash Recovery in RAMCloud. ACM SOSP 2011.
[16] T. Wang and R. Johnson. Scalable Logging through Emerging Non-Volatile Memory. VLDB 2014, vol.7.
[17] N. Malviya, A. Weisberg, S. Madden, et al. Rethinking Main Memory OLTP Recovery. ICDE 2014.
[18] C. Yao, D. Agrawal, G. Chen, et al. Adaptive Logging for Distributed In-Memory Databases. ArXiv e-prints.
[19] MongoDB. https://www.mongodb.org/
[20] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. NSDI 2013, pp. 371–384.
[21] X. Bao, L. Liu, N. Xiao, et al. HConfig: Resource Adaptive Fast Bulk Loading in HBase. CollaborateCom 2014.
[22] A. Dragojevic, D. Narayanan, O. Hodson, et al. FaRM: Fast Remote Memory. NSDI 2014, pp. 401–414.
[23] H. Lim, D. Han, D. G. Andersen, et al. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. NSDI 2014.
[24] R. Fang, H. Hsiao, B. He, et al. High Performance Database Logging using Storage Class Memory. ICDE 2011.
[25] A. Caulfield, J. Coburn, T. I. Mollov, et al. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. SC 2010.