# Extending Storage Capacity in Redis by adding a Memory-Mapped File Storage Layer

Yi Wang(s1128361)
LIACS
Leiden University
The Netherlands

Dr. Erwin Bakker
LIACS
Leiden University
The Netherlands

## Abstract

*Redis is an open-source, networked, in-memory, key-value data store with optional durability. In order to achieve its outstanding performance, Redis works with an in-memory dataset. One recurring problem in practice for users and developers is how to overcome the physical memory size limitations without causing a notable performance degradation. We present a new effective solution to this problem by adding a memory-mapped-file storage layer internally for Redis and the benchmark on Yahoo Cloud System Benchmark(YCSB) proves its feasibility.*

## 1. Introduction

Redis [4] is a high-performance key/value database if sufficient physical memory is available. However it is difficult for Redis to deal with datasets that are larger than RAM efficiently. When the memory limit is reached Redis will try to remove keys accordingly to the eviction policy selected. This is even dangerous to delete data for some application scenarios.

Therefore it is challenging and important to provide users with a safe, high-capacity data storage solution based on Redis for certain applications. In our work, the key components include (i) memory-mapped-file storage layer is developed so Redis can 'see' a broader memory address and contain more data (ii) certain memory-mapping granularity control rules are also applied to improve overall performance.

Besides considering already existing approaches in Section 2, this article is organized as follows: Section 3 describes in detail the idea of design. The technical details on implementation are explained in Section 4. In Section 5, we deal with the performance evaluation with different experimental settings on YCSB[3]. Section 6 contains conclusion and extensive discussion about the results obtained. Additionally, future work is presented in Section 7.

## 2. Related Work

Actually there has been a discussion about extending Redis to deal with the datasets bigger than RAM. Some models and existing implementations are summarized in this section.

Redis Virtual Memory is a feature that will appear for the first time in a stable Redis distribution in Redis 2.0. Since Redis follows a Key-Value model, all the keys are associated with some values. Usually Redis takes both Keys and associated Values in memory. Sometimes this is not the best option, and while Keys must be taken in memory by design (and in order to ensure fast lookups), Values can be swapped out to disk when they are rarely used.In practical terms this means that if you have a dataset of 100,000 keys in memory, but only 10% of this keys are often used, Redis with Virtual Memory enabled will try to transfer the values associated to the rarely used keys on disk.When these values are requested, as a result of a command issued by a client, the values are loaded back from the swap file to the main memory. In Redis's Virtual Memory implementation, fread(), fwrite() and fflush() functions are invoked every time loading, swapping occurs, which might cause a potential disk IO bottleneck. Redis VM mechanism is now deprecated because it has several disadvantages and problems.

Pluggable back-end storage engine model is also mentioned. The basic idea is that using a disk-based data storage engine which is responsible for managing data at low level. Disk-based engine can take the advantage of optimal index structures such as B-tree or B+tree to retrieve records on disk in a efficient way. MySQL, PostSql and level-db are all discussed while some problems have to be taken into account: For MySQL those relational database, how to convert the schema-less Key/Value model into a rigid relational model is a difficult task; for those disk-based Key/Value store library(e.g. leveldb), interaction, compatibility and configuration these issues require great effort.

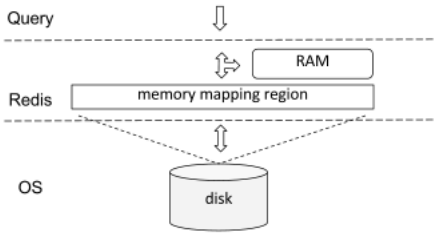



**Figure 1. Basic architecture of Redis memory-mapping-file solution.**

## 3. Solution Overview

Our solution is inspired by MongoDB [2] in which the memory-mapped files are the critical piece of the storage engine. A memory-mapped file is a segment of virtual memory which has been assigned a direct byte-for-byte correlation with some portion of a file or file-like resource. This resource is typically a file that is physically present on-disk, but can also be a device, shared memory object, or other resource that the operating system can reference through a file descriptor. Once present, this correlation between the file and the memory space permits applications to treat the mapped portion as if it were primary memory. The basic idea in our solution is that by using memory-mapped files, Redis can treat the content of its data files as if they were all in memory even though data size is sometimes much larger than the size of the available physical memory. Because Redis supports a wide range of data structures storage, it is obviously unwise to utilize memory-mapped files for all types of data structures without taking application domain information into account. Certain rules should be employed to control which kinds of data structures can be suitable for memory mapping fashion.

Specifically, once a data structure is determined to be stored in the memory mapping fashion, one file is created and mapped into a region to hold the Values while the Keys are all still in memory in order to achieve a fast lookup. Redis manipulates data objects in this region as they were in primary memory. Note that the size of memory mapped file can be larger than that of the available physical memory, therefore part of data objects may be swapped out once there is no more space in physical memory to contain all of them.

The main advantage of using memory-mapped files is that Least Recently Used(LRU) cache algorithm in modern Operating System(OS) can automatically calculate and intelligently choose which elements to be swapped out and in. The OS cache acts as a large cache layer. For read operations, the OS just delivers the data to Redis if the data are already in memory. Otherwise, a read() system call is invoked internally by the OS to fetch data from disk. Compared with Redis's Virtual Memory mechanism aforementioned in Section 2, the OS does not need to copy data from its private kernel buffer to the user space buffer by using memory mapped file. For write operations, data are flushed to disk only when needed and random writes are converted to sequential writes in order to improve IO performance. This novel design pushes down the responsibility of managing, caching, swapping out-of-date Value objects to the OS level without completely modifying the low level code in Redis. An basic architecture for our solution is illustrated in Figure 1.

## 4. Design and Implementation

The implementation in our solution resembles a combination of MongoDB's memory-mapped file storage engine and Redis's Virtual Memory mechanism. All the Key objects must stay in memory so as to retrieve corresponding value objects promptly. The Value objects might be stored in memory or in a memory-mapped file which depends on how memory mapping controlling rules are performed(namely, depending on which kind of data structure is selected to be memory-mapped).

In our implementation, an extra memory-mapped file storage layer is introduced for *Hash* type data structure in Redis. This new layer is also written in ASCII C which seamlessly integrated with original source code of Redis.

### 4.1. memory-mapping pool

When initializing Redis, a mapped file is created; then a large character array *memory-mapping-pool* is mapped to this file through mmap() system call. All the Value objects associated with *Hash* type data structure are located resident in *memory-mapping-pool*. A series of functions for manipulating objects in *memory-mapping-pool* are provided. From a point of view of Redis, there is no difference between data in memory and data in memory-mapped file so common C functions referencing to memory operation, including memset(), memcpy() etc., can be also used conveniently. This feature make it possible to reuse existing code as well.

**page number and page size** In order to use the space of *memory-mapping-pool* efficiently, we draw lessons from the implementation of Virtual Memory mechanism [5] in Redis, original code is reused and refactored. Specifically, The whole pool is virtually divided into *pages*. Note that the term *page* is distinct from the concept of *page* in the Operating System. The term *page* is defined in *memory-mapping-pool* for the following purposes: (i) to record how much free space remains in the pool; (ii) to plot a free space distribution for better retrieving new free pages and reallocating used pages. A bitmap is reserved to remember page usage statistics information. Virtually, one object can be saved on multiple pages, but one page can not be shared between objects. As a result, the total size of *memory-mapping-pool* is :

$$total\_size = page\_number \times page\_size$$

If the page size is too big, obviously storing small value objects will waste a lot of space which might lead to 'holes' in one page; if the page size is too small, finding many continuous pages is a time-consuming task, sometimes it even might fail to locate a proper continuous free area. The page number and the page size for *memory-mapping-pool* can be configured manually according to application scenarios.

According to the Redis VM mechanism code and explanation, the page allocation algorithm is modified based on Redis VM code. Basically a pointer indicating current position in *memory-mapping-pool* is reserved globally. Firstly, Redis tries to allocate pages sequentially up to a given limit. When this limit is reached, Redis restarts at the page 0 which tries to improve locality. If Redis can not find continuous free pages for a while, it jumps forward random steps. A pseudo code is included in algorithm 1.

**Algorithm 1** Simple Page Allocation Algorithm based on original code in Redis VM.

```
function FIND(pages)
    base = current_position;
    offset = 0, numfree = 0, jump = 0;

    while offset not exceeds mapping region do
        pos = base + offset;
        if pos reaches end then
            pos = pos − total_pages;
        end if

        if find one free page then
            numfree + +;
            if numfree = pages then return
                pos − (n − 1)
            end if
        else
            numfree = 0;
        end if

        jump + +;
        if numfree = 0 then
            if jump too many times then
                offset+ = random_steps;
                jump = 0;
            end if
        else
            offset + +;
        end if
    end while
return error
end function
```

**mapping granularity** Another significant setting in the implementation called *mapping granularity* is introduced in our solution. An important question is how to define *mapping granularity*? Currently, in out implementation, mapping granularity refers to the rule of determining which kind of data structures in Redis are selected to store in a memory-mapped file.

Usually, application scenarios information is helpful.In this following paragraphs, the experiment on Yahoo Cloud System Benchmark(YCSB) is selected as an example for further explanation about *mapping granularity*.

First of all, the background knowledge of YCSB is of necessity to understand. In YCSB, two data structures are involved: *Hash* and *sorted set*. In order to build a standard and generic benchmark for both NoSQL and SQL databases, YCSB decides to use the Hash and the Sorted Set to record data. A Hash in Redis refers to a map between string fields and string values which can be represented as :

$$key \Rightarrow \langle field1 : value1, field2 : value2, \ldots \rangle$$

The Sorted Set in Redis is the non repeating collection of string elements. The difference is that every member of a Sorted Set is associated with a score, that is used in order to take the sorted set ordered, from the smallest to the greatest score. While members are unique, scores may be repeated. A Redis Sorted Set can be presented as:

$$key \Rightarrow \langle score1 : value1, score2 : value2, \ldots \rangle$$

Each record is stored in one Hash of which an unique key and associated fields-value pairs generated by YCSB consists. The generated unique keys are all stored in a Sorted Set called '_indices' as well.

Moreover, the '_indices' Sorted Set is used internally for SCAN operation in YCSB. A SCAN operation is performed in this way: YCSB firstly chooses a starting position 'fromIndex' and scan operation count 'scanCnt' (usually 50 100) randomly; then requests a ordered list of keys containing the specified 'scanCnt' elements that staring from the specified 'fromIndex' in '_indices' based on corresponding score. Furthermore, the '_indices' Sorted Set is also used as a lookup table to examine whether a key exists or not.

Therefore it is apparent that '_indices' should be kept in memory all the time, accelerating access to keys/values pairs and other special operations. Some Hashes might be frequently asked(hot data) while other Hashes may be rarely used by clients. It is reasonable to store Redis Hashes in a memory-mapped file way.

As concluded, the mapping granularity controlling rule for experiments on YCSB is that storing the Sorted Set in physical memory and the Hashes are accessed through the memory-mapped file layer.

## 5. Benchmark and Results

### 5.1. Experimental environment

For benchmark tool, Yahoo Cloud System Benchmark(YCSB) is used to examine our solution's feasibility and performance. The goal of the YCSB project is to develop a framework and common set of workloads for evaluating the performance of different key-value and cloud serving stores. Many NoSQL database client drivers are supported, including Redis. Internally, different core workloads can be defined to plot the performance of system under a simulating application scenario. By default, six core workloads are defined in YCSB:

1 Workload A: Update heavy workload. This workload has a mix of 50/50 reads and writes. An application example is a session store recording recent actions.

2 Workload B: Read mostly workload. This workload has a 95/5 reads/write mix. Application example: photo tagging; add a tag is an update, but most operations are to read tags.

3 Workload C: Read only. This workload is 100% read. Application example: user profile cache, where profiles are constructed elsewhere (e.g., Hadoop).

4 Workload D: Read latest workload. In this workload, new records are inserted, and the most recently inserted records are the most popular. Application example: user status updates; people want to read the latest.

5 Workload E: Short ranges. In this workload, short ranges of records are queried, instead of individual records. Application example: threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id).

6 Workload F: Read-modify-write. In this workload, the client will read a record, modify it, and write back the changes. Application example: user database, where user records are read and modified by the user or to record user activity.

After running a specified workload, valuable statistic information: thoughput, running time and average latency are automatically produced as output for users. Apart from defined workloads, several runtime parameters in YCSB used with the core workload generator are:

1. fieldcount: Number of fields in a record.

2. fieldlength: Size of each field(in byte).

3. operationcount: Number of operations to perform.

4. recordcount: Number of records totally generated.

5. threads: Number of client threads.

Through all the experiments, several runtime parameters are constant: fieldcount is 10, fieldlength is 100 bytes, operationcount is 100,000 and the number of client threads is 100. The recordcount varies in experiments for different data size.

For experiment hardwarem, all the experiments are conducted on DAS-4 machine in Leiden site[1]. More information of DAS-4 machine is available on [1]. Here essential parameters on DAS-4 machine is listed:

1. RAM Size: 50GB.

2. CPU: Intel E5620.

3. Operating System: CentOS Linux.

For parameters in memory-mapping storage layer: the page size in *memory-mapping-pool* is 32 bytes and the page number is 8073741824. As a result, the size of the memory-mapped file is 240GB. Persistence option(both RDB and AOF) is also disabled.

### 5.2. Experiments on hard disk

The memory mapped file is created on hard disk and all the six core workloads in YCSB are tested. Three experiments are conducted with different dataset size – 20 GB(10,000,000 records), 40 GB(15,000,000 records) and 60 GB(20,000,000 records), respectively.

Throughput under different dataset size in each workload , as a measurement factor, is plotted in Figure 2(a). When the dataset size is 20 GB, physical memory is sufficient to contain the whole dataset. Less swapping between disk and memory occurs. The Operating System tends to keep all of data in memory rather than swap data out and the throughput looks very well. One notable workload is the workloadE, average throughput is 484 ops/sec. As explained in Section 4.1, SCAN operation usually requires random reads. If records are swapped out and located in disk file, a number of swap-in operations have to be performed which might lead to an overall throughput degradation. It becomes worse when scanning a larger dataset. Other workloads give a good impression – average throughputs are all above 10,000 ops/sec.

After dataset is increased to 40 GB, the physical memory is nearly exhausted since the RAM size is 50 GB on testing machine. 40 GB can be considered as a critical turning point on testing machine. Frequent swap-out and swap-in happens, causing the throughputs of all the workloads except for workloadD dramatically decrease below 700 ops/sec. The workloadE, for scan operation, performs worst – the average throughput is only 6 ops/sec. Only workloadD occupies a 1532 ops/sec throughput.

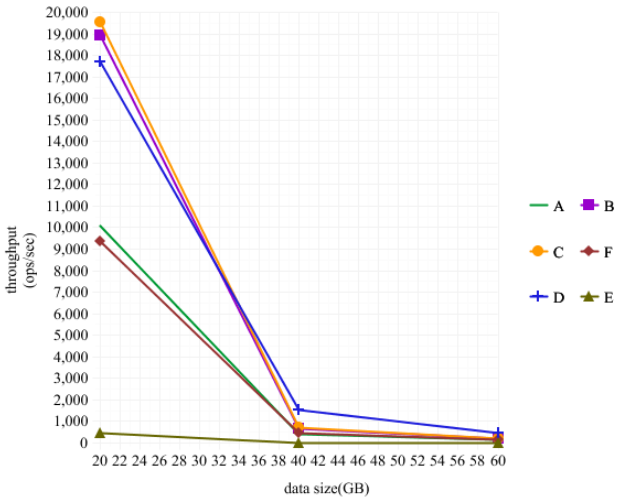
Further enlarging the dataset to 60 GB, the overall throughput declines. Even though the workloadD has a relative highest score – 483 ops/sec throughput, others are all below 200 ops/sec. The workloadE is the worst one, only 2 ops/sec throughput is achieved.

Additionally, when trying to test 100 GB dataset on hard disk, the time-out exception always take place to interrupt benchmark. This means request is blocked in Redis and the latency has been reached to 30s due to swapping, which is not acceptable in most of applications. For this reason, the throughput variation is not plotted in Figure 2(a). More detailed results data are recorded in Appendix 8.

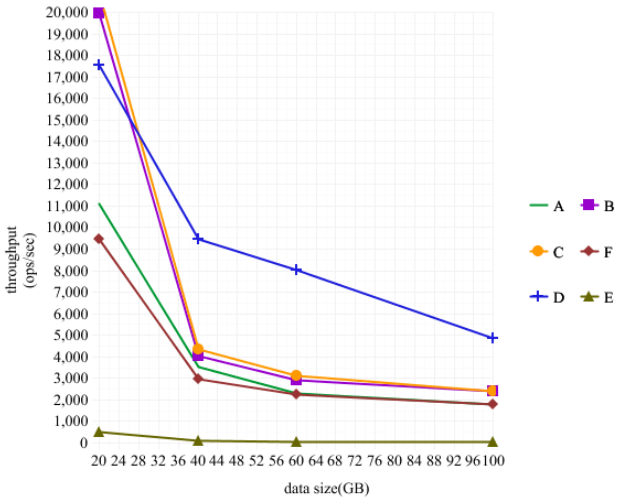
### 5.3. Experiments on SSD

The memory mapped file is created on SSD and all the six core workloads in YCSB are tested. Three experiments are conducted with different dataset size – 20 GB(10,000,000 records), 40 GB(15,000,000 records), 60 GB(20,000,000 records) and 100 GB(30,000,000 records), respectively.

Throughput under different dataset size in each workload , as a measurement factor, is plotted in Figure 2(b). Compared with the results on hard disk, the performance on SSD is really impressive and satisfying. To be more specific, the initial size of dataset is also 20 GB. For each workload, there is no significant contrast between hard disk and SDD. As aforementioned, 40 GB is the critical point on testing machine. When the dataset size is added to 40 GB, each workload starts to decrease but not vastly. Except for the workloadE, the average throughput of others still remains above 3500 ops/sec. The workloadD has the highest throughput – 9474 ops/sec. The workloadE obtains a 85 ops/sec throughput that is 10 times of that in harddisk.




(a) experiment on hard disk




(b) experiment on SSD

**Figure 2. Experiments conducted on hard disk and on SSD. Y-axis means throughput(ops/sec) and X-axis means dataset size(GB). A, B, C, D, E and F represents throughput variation under different dataset size in workloadA, workloadB, workloadC, workloadD, workloadE and workloadF mentioned in Section 5.1.**

Larger datasets are examined as well. Particularly, for 60 GB dataset, the performance of each workloads is much better those of on hard disk. The workloadE declines slightly to 58 ops/sec and others still keep above 2000 ops/sec. No doubt that workloadD outperforms others with a 8041 ops/sec throughput. For 100 GB dataset, there is no time-out exception, in contrast to test on hard disk. The scan operation, namely the workloadE gains a 44 ops/sec throughput. Others perform distinctly varying from 1777 ops/sec(in workloadF) to 4880 ops/sec(workloadD). Within more and more value objects being swapped out, throughout tends to slope down gradually. More detailed results data are recorded in Appendix 8.

### 5.4. Result Analysis

From the two sets of experiments, interesting results are exposed: Firstly, comparing with hard disk, SSD can manifestly improve throughout for memory mapping layer. Hard disk can not handle dataset that is larger than 60 GB due to recurring time-out exceptions.

Secondly, the throughput in workloadD outperforms others, especially in large dataset. As mentioned in Section 5.1, the workloadD is a read latest workload in which new records are inserted, and the most recently inserted records are to be queried. This scenario benefits from the LRU algorithm in the OS since those newly inserted records have a higher possibility to be cached, instead of being swapped out when physical memory is exhausted. The throughout in the workloadE is not desirable no matter launching the memory mapping layer on hard disk or on SSD. This is because SCAN operations in the workloadE often involves a large portion of random access in which situation the LRU algorithm in the OS could not work as expected.

Furthermore, when dealing with massive dataset, it is observed that the throughput tends to slope down as adding more records. Under the circumstance that dataset has already exceeds the physical memory capacity, there must be value objects that are swapped out and flushed into disk. However, this is inevitable due to the fact that mapping an extreme large file usually causes a higher page fault probability. Consequently, repeatedly loading data from file in a random order reduces the throughput.

## 6. Conclusions

It is feasible to launch a memory mapping file storage layer in Redis to cope with datasets exceeding the physical memory capacity. As shown in tests, SSD is a preferable option to achieve high performance. It is also indicated that a READ-LATEST application scenario can benefit from our solution. The throughput tends to decline continuously with dataset increasing which means storing an extraordinarily large set of data(for example, 10 times large than RAM) by using memory-mapped file should be careful.

## 7. Future works

There is room to promote the efficiency of memory mapping layer solution. Several perspectives are listed for readers who are interested in it.

**finer mapping granularity control** In our implementation, only Redis Hash is supported in memory mapping storage layer because it is not a wise option to swap out an extensive Sorted Set. So all the elements in the Sorted Set are forced to be kept in physical memory. A finer mapping granularity control rule is required: in a Sorted Set, partial elements are regularly visited(hot data) while the rest might be occasionally asked(cold data). keep hot data in physical memory and storing cold data in the memory mapped file way is an optimized implementation to save memory space and reduce page faults in the OS cache.

**organize memory mapping region** As mentioned above, current implementation use a character array to represent the *memory-mapping-pool*. Using a linked list to organize it might be a more effective way. For instance, the linked list consists of many nodes; each node represents a fixed size space in the pool and basic statistic information like free space, used space and lru time is contained to improve efficiency.

**synchronize memory mapped file** Normally, it is the Operating System that is responsible for determining which and when to flush cached data into disk. An interface for manually synchronizing data might be useful for special situations.

## 8. Acknowledgements

I would like to acknowledge and extend my heartfelt gratitude to the following persons who have made the completion of this article. I would like to thank My supervisor, Dr. Erwin M. Bakker, who guides me a clear research direction through weekly discussion and feedback. I have learned a lot from this project. I would like to thank Salvatore Sanfilippo, creator of Redis, who enthusiastically reply my questions about Redis internals and other members in Google Redis Group.

## References

[1] Das-4 clusters. `http://www.cs.vu.nl/das4/clusters.shtml`[Online, accessed 16-Feb-2013], Feb. 2013.

[2] K. Chodorow and M. Dirolf. *MongoDB: the definitive guide*. O'Reilly Media, 2010.

[3] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[4] T. Macedo and F. Oliveira. *Redis Cookbook*. O'Reilly Media, 2011.

[5] S. Sanfilippo. Redis virtual memory: the story and the code. `http://oldblog.antirez.com/post/redis-virtual-memory-story.html`, Feb. 2013.

## A. Results Data on YCSB

**Table 1. Results of experiments on 20 GB dataset.**

(a) memory mapped file on hard disk

| workload | time(sec) | throughput(ops/sec) |
|---|---|---|
| A | 9.936 | 10064 |
| B | 5.290 | 18903 |
| C | 5.119 | 19535 |
| F | 10.708 | 9338 |
| D | 5.653 | 17689 |
| E | 206.565 | 484 |

(b) memory mapped file on SSD

| workload | time(sec) | throughput(ops/sec) |
|---|---|---|
| A | 8.991 | 11122 |
| B | 5.008 | 19968 |
| C | 4.781 | 20916 |
| F | 10.549 | 9479 |
| D | 5.692 | 17568 |
| E | 203.429 | 491 |

**Table 2. Results of experiments on 40 GB dataset.**

(a) memory mapped file on hard disk

| workload | time(sec) | throughput(ops/sec) |
|---|---|---|
| A | 256.905 | 389 |
| B | 155.804 | 641 |
| C | 143.289 | 697 |
| F | 223.661 | 447 |
| D | 65.243 | 1532 |
| E | >1610 | 6.5 |

(b) memory mapped file on SSD

| workload | time(sec) | throughput(ops/sec) |
|---|---|---|
| A | 28.154 | 3551 |
| B | 24.617 | 4062 |
| C | 23.031 | 4341 |
| F | 33.662 | 2970 |
| D | 10.555 | 9474 |
| E | 1172.551 | 85 |

**Table 3. Results of experiments on 60 GB dataset.**

(a) memory mapped file on hard disk

| workload | time(sec) | throughput(ops/sec) |
|---|---|---|
| A | 608.553 | 164 |
| B | 521.755 | 191 |
| C | 516.773 | 193 |
| F | 625.813 | 159 |
| D | 206.692 | 483 |
| E | >3432 | 2.5 |

(b) memory mapped file on SSD

| workload | time(sec) | throughput(ops/sec) |
|---|---|---|
| A | 43.673 | 2289 |
| B | 34.104 | 2932 |
| C | 32.143 | 3111 |
| F | 44.922 | 2226 |
| D | 12.435 | 8041 |
| E | 1708.112 | 58 |

**Table 4. Results of experiments on 100 GB dataset.**

(a) memory mapped file on SSD

| workload | time(sec) | throughput(ops/sec) |
|---|---|---|
| A | 55.805 | 1791 |
| B | 41.310 | 2420 |
| C | 41.570 | 2405 |
| F | 56.270 | 1777 |
| D | 20.489 | 4880 |
| E | 2226.132 | 44 |