I/O Characteristics of MongoDB and Trim-based Optimization in Flash SSDs

Trong-Dat Nguyen
College of Information and Communication Engr.
Sungkyunkwan University
Suwon 440-746, Korea
(82+)031-290-7988
datnguyen@skku.edu

Sang-Won Lee
College of Information and Communication Engr.
Sungkyunkwan University
Suwon 440-746, Korea
(82+)031-290-7988
swlee@skku.edu

ABSTRACT
NoSQL solutions become emerging for large scaled, high performance, schema-flexible applications. WiredTiger is cost effective, non-locking, no-overwrite storage used as default storage engine in MongoDB. Understanding I/O characteristics of storage engine is important not only for choosing suitable solution with an application but also opening opportunities for researchers optimizing current working system, especially building more flash-awareness NoSQL DBMS. This paper explores background of MongoDB internals then analyze I/O characteristics of WiredTiger storage engine in detail. We also exploit space management mechanism in WiredTiger by using TRIM command.

Categories and Subject Descriptors
H.2.4 [**DATABASE MANAGEMENT**]: Systems – *Distributed Databases, Transaction processing*.

General Terms
Measurement, Performance, Experimentation,

Keywords
I/O characteristics, I/O pattern, MongoDB, WiredTiger, TRIM command, SSD, YCSB, NoSQL.

1. INTRODUCTION
NoSQL solutions become emerging for large scaled, high performance, schema-flexible applications not only in industrial area but also in academia. Typically, a NoSQL system requires BASE (Basic availability, Soft-state, and Eventually consistent) properties that is more relax than ACID (Atomic, Consistency, Isolation, and Durability) properties in traditional Relational Database Management System (RDBMS) [1]. While original NoSQL solutions are used for some specific purpose as Dynamo [3] from Amazon supports simple key-value data model, or BigTable [4] column store from Google that supports very large and various tables. However, the majority applications that require transaction processing supported and consistency in specific level. Moreover, when customers consider to replace current RDBMS solutions with new NoSQL approaches, shifting process should be simple, inexpensive and effective. Thus, all-purpose, hybrid DBMS is necessary in those cases. MongoDB is a common choice that shares some core features with traditional RDBMSs e.g. journaling, transaction processing, multi-version concurrency control, secondary index support and friendly query language. With WiredTiger as storage engine, MongoDB become a high performance, scalable NoSQL that support both row-oriented storage and column-oriented storage.

WiredTiger is designed with default implicit assumption that hard disks are used as the underlying storage device. However, NAND-flash based solid state disk (SSD) is considered as standard replacement for hard disk with multiple orders of magnitude faster access rate, lower power consumption, light weight, and shock resistance due to eliminating movement parts [5]. One major flaw in SSD is non-empty blocks need to be erased before they are programed again. Erase operation takes an order of magnitude longer than write operation [5]. In additional, if that block contains some valid pages, those pages need to be copy back to an empty block before the old block is erased. That phenomenon make Garbage Collection (GC) overhead increase significantly. Typically, underlying flash SSD has no knowledge about which pages are invalid from user space and kernel space until those pages are written out. Thus, during GC working time, coping valid pages to new block which will be invalid soon is waste. With the introduction of TRIM command, higher layers i.e. user space and kernel space can notify which pages are unused any more so that GC can treat them as other invalid pages, thus reduce overhead significantly [7].

Understanding in detail I/O characteristics of a storage engine and working mechanism of underlying storage device is not only important for choosing suitable solutions for business, but also opens opportunities for researchers optimize current storage engine [6]. Official documentation from MongoDB [9] and WiredTiger [10] is inadequate for the sake of understanding WiredTiger I/O in depth. This paper using YCSB benchmark in the stand-alone MongoDB server to evaluate and analyze in detail major aspects of WiredTiger storage engine as well as optimize it by exploit TRIM command.

EDB, October 17-19, 2016, Jeju Island, Republic of Korea

ACM 978-1-4503-4754-9/16/10.
http://dx.doi.org/10.1145/3007818.3007844

2. BACKGROUND

2.1 Directory Structure

MongoDB uses separate directory for data files, journal files and metadata files. With properly setting, data files is grouped in per-database directories and each database has one data file per collection and one index file per either primary index or secondary index. MongoDB stores system database in local directory for tracking metadata and management information. Figure 1 shows an example of directory structure with one database named YCSB.

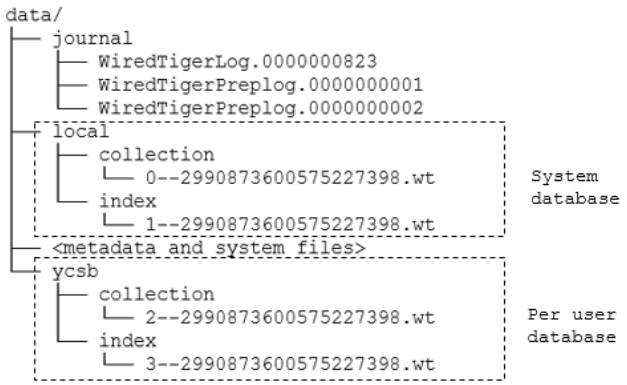

Figure 1. Directory structure with one YCSB database along with journal files and system files and metadata files

2.2 Mapping between MongoDB and RDBMS

There is a conceptual mapping between traditional RDBMS and MongoDB data modeling where collection in MongoDB is mapped with table in RDBMS, a document in BSON format is considered as row and key-value pairs in a document are mapped with fields in a row. When a new document is generated, MongoDB automatically assign "_id" as primary key and its value is monotonically increase, a primary index is also built automatically based on "_id" key. Secondary index is supported in MongoDB, whenever a new secondary index is created, a new index file is generated in associate directory. However, when one decide to add a secondary index, a trade-off between read and write performance should be carefully considered. For heavily update workload, more secondary indices lead to slow down whole system performance, for each update on collection pages, the according secondary indices are also updated. WiredTiger use B+Tree data structure to store both collection and index files. Unlike RDBMS, schemas are flexible within semi-structure data BSON and JOIN operations are not necessary in MongoDB.

For consistency and durability reasons, MongoDB writes updates on journaling fashion and periodically calls checkpoint in the interval time. Each update operation, first is captured in log records that located in log buffer in D-RAM, log server thread periodically check the log buffer and write out group of log records to log files persistently before the update is synchronized on data file. When a log file is accumulated fill in up to a threshold e.g. 100 MB, a new log file is created to receive next log record. MongoDB just kept adequate number of log files for the last checkpoint, the older log files are deleted. In additional to log files, checkpoint is called periodically for each interval time e.g. 60 seconds or the amount of log data written reach a threshold e.g. 2 GB. In traditional RDBMS checkpoint process is expensive due to a lot of works should be done at this moment. WiredTiger is not exception, at checkpoint time all update changes in versions of a data object should be merged to the original on-disk image, remained log records firstly are written to persistent log file, then all dirty pages are sync on persistent store. In additional, old checkpoints are read from disk to D-RAM in order to merge with the current checkpoint before discarded. Shared resources are also lock/unlock during checkpoint time make overhead even higher.

2.3 MVCC and Space Management in WiredTiger

WiredTiger shares similar multiple version concurrency control (MVCC) mechanism as RDBMS with some customizations. When a update request occurs from client, the underlying page contains associate records is firstly fetched from storage to D-RAM, WiredTiger uses copy-on-write approach that avoid locking by keep the original on-disk image and write changes in multiple version for every update occurs. As default, WiredTiger adopts read committed isolation level such that at the same time, if multiple users desire to read on the same page, only the recent committed visible version are viewed. In case no updates occur on that page, the original on-disk version will be read. Dirty pages are kept in buffer pool until it becomes a victim of replacement policy.

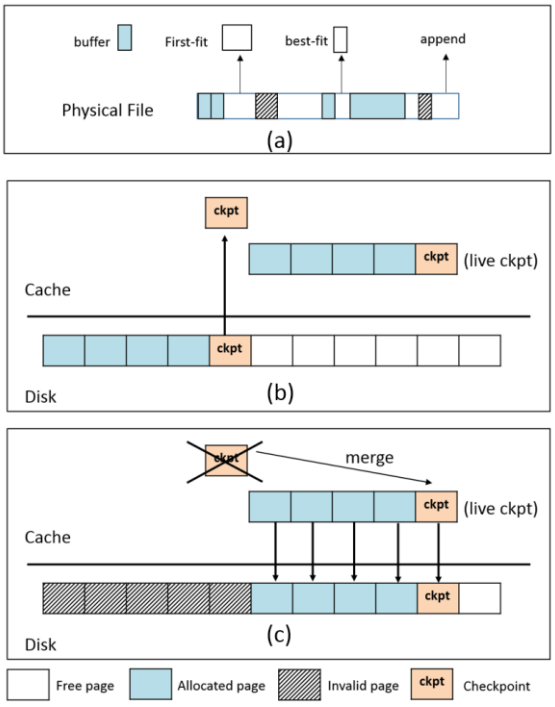

Figure 2. (a) Space allocation methods in WiredTiger, (b) and (c) Checkpoint merging and invalid old version page during checkpoint time

As shows in Figure 2(a), WiredTiger manage space by extend data structure that each includes logical disk offset and size. There are three extend list for each file that keep track of allocated space, available space, and discard space. Before an data buffer is actually write out, lasted update version is apply to original on-disk image, then space management allocate the logical disk address for the upcoming write based on three approaches: (1) first-fit that selects the first extend in available extend list that fit

the data buffer, (2) best-fit that select the first smallest extend that fit the data buffer, and (3) append at the end of file.

Figure 2(b) illustrates how WiredTiger manages extend list and reuse previous allocated space by using checkpoints. Extend list information is kept in checkpoint structure and write out on persistent storage at the checkpoint time. WiredTiger uses a special unique checkpoint named live checkpoint that only exist when the system is running and located in D-RAM. During properly working time, whenever update requests come from client, live checkpoint keeps track of both data changes and extend list grow/shrink information; then at the time checkpoint server is signaled, the previous checkpoint is fetched from persistent storage to D-RAM and merged with live checkpoint before write out to storage system. After the merging process finish, previous disk space occupied by the same data page is available and can be reused for next writes.

3. Optimizing WiredTiger using TRIM Command

Typically, SSDs do not understand file structure of an OS, they only track valid/invalid data blocks that reported by OS. When a data block is no longer necessary by either delete operation by OS or logically invalid by an application, the data block is marked as "invalid" without informing to SSD drive, so the corresponding physical page is seen as "valid" by SSD. Therefore, there are non-necessary garbage collection overheads of copying back valid pages to empty blocks. To solve this problem, TRIM command is introduced to allow an OS to inform SSD which data blocks are no longer used, so that, those blocks can be skipped in garbage collection operating, that lead to reduce overhead, lower write amplification and longer lifespan of SSD.

We found out TRIM command is useful in the case of WiredTiger. When a logical data page becomes dirty and written out in new address provided by the space management, the old address become invalid logically, we call that phenomenon is address replacement. We describe how TRIM command is used to optimize WiredTiger by an example in Figure 3. Suppose each extend e.g. Ext A in WiredTiger map with two logical blocks in SSD e.g. A1, and A2. As showed in Figure 3(a), in the original state (1), WiredTiger views file offsets as extends while SSD has logical view on file as logical block addresses (LBA), Flash Translation Layer (FTL) inside SSD translates logical LBA to physical block address (PBA) before write on flash memory chip. There is an unused space named over provisioning that invisible with higher layers and reserved for GC processing. At state (2), when client e.g. YCSB workload issues an update request for a record again page P1, WiredTiger's space management use either first-fit or best-fit approach to provide an available address e.g. Ext C for P1, an address replacement from Ext A to Ext C makes Ext A become invalid logically in WiredTiger. However, in the point of view of SSD, its associates LBA A1 and A2 still valid until it reach the state (3) where space management reuse the Ext A again for a write request. If GC occurs between state (2) and state (3), A1 and A2 are copied back unnecessarily.

In WiredTiger, a data page is written in non-in-place-up-date fashion, hence there are exist multiple versions of that page scatter on disk. Thing becomes worse when the client workload e.g. YCSB includes huge number of small random updates, that not only lead to increase the majority high overhead of GC process but also effect the lifespan of SSD. For that reason, as illustrated in figure 3(b) that similar with previous use case except we adopt TRIM command at application layer i.e. WiredTiger, such that at state (2) whenever address replacement taken, we actively call TRIM command to notify invalid pages to FTL. Logical view of SSD to A1, A2 as free space while physical view to those as invalid, so that in case the GC is called between state (2) and state (3), it will not copy PBA A1, A2 to new block, that reduce the overhead of GC significantly.

In practical, there is a side effect for TRIM command used. The naïve approach such that call TRIM command for every address replacement lead to high overhead, especially in the case of write intensive workloads. We solved that problem by delay calling TRIM commands up to a threshold, which is a specific number of address replacements occurred, then apply TRIM commands in batch for those saved addresses. The threshold should large enough to reduce the overall overhead of TRIM command calls, but should small enough to avoid consuming user space occupation, as well as, overhead of arranging addresses when batch calling TRIM commands occurred.

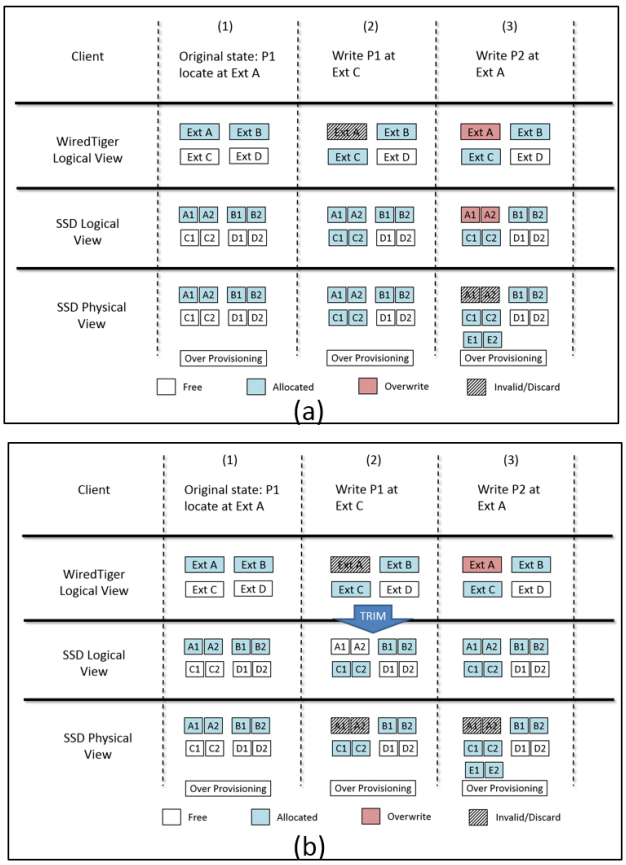

Figure 3. Effect of TRIM command on WiredTiger and SSD, (a) without TRIM command and (b) with TRIM command

4. EVALUATION AND ANALYSIS

4.1 Experiment Setup

We use MongoDB server 3.2 with WiredTiger as storage engine. For concurrency processing, we use 40 threads from YCSB client and run on a commodity server with 48 cores Intel Xeon 2.2 GHz processer, 32 GB D-RAM with Samsung SSD 840 Pro as the storage device. To stress out the storage device performance, we

use 100 percentage update workload that modified from original workloada with 30 million records and 30 million operations. The workload is write intensive with randomly request follow zipfian distribution rule [11]. Blktrace is used to keep track of I/O patterns. Moreover, we embed trace code in original WiredTiger to track write pattern in filesystem level.

4.2 Asymmetric amount of written data

In order to find the bottle neck in WiredTiger, we measure the amount of data written for various dataset size from 10 millions of records to 150 millions of records as show in Table 1. The amount of data written in MB of each file type is showed associate with their fraction. Almost data written are occurred from collection file with more than 90 percentage except for very small dataset i.e. 10 millions. Index file is excluded from the table because there is no index update for only update workload.

Collection file is stored as B+Tree in WiredTiger that include a root page, internal node pages, and leaf node pages; and as described from previous sections, checkpoint pages are used to manage extend list during the system is working. To further analysis, we embedded traces in original WiredTiger source code to examine workload on each page types inside a collection file. The results from table 2 clearly shows that more than 99 percentage of page written from leaf page, it because not only the amount of leaf pages is out numbers to internal pages but also leaf page size 32KB that is eight folds of internal page size i.e. 4KB. In additional, due to characteristics of B+Tree data structure, internal pages have more frequency accesses than leaf pages, therefore are kept in buffer pool longer.

Table 1. Asymmetric amount of written data of file types

YCSB number of records (millions)	Amount of written data (MB)		
	Collection	Journal	Others
10	14,815 (70.71%)	6,135 (29.28%)	1 (0.01%)
50	360,282 (92%)	31,176 (7.96%)	27 (0.04%)
100	811,145 (92.85%)	62,335 (7.13%)	63 (0.02%)
150	1,263,428 (93%)	93,957 (6.92%)	92 (0.08%)

Table 2. Asymmetric of page types in collection file

Page type	# of writes	Fraction (%)
Root page	8	0.000049
Internal page	43,730	0.27
Leaf page	16,085,738	99.7
Checkpoint page	15	0.00015

4.3 Collection file I/O Patterns

Figure 4 shows the I/O patterns of Collection in WiredTiger tracked by blktrace, the x axis is runtime by second, the y axis is logical block address (LBA) that the I/O occurs. Collection is fully random read with more compact than write pattern, it can be explain by the natural characteristics of YCSB workload that occur heavy small random requests. Whenever read request occur and if the desired page is not in D-RAM it need to be fetched from persistent device, however when write request come, updated page is not write immediately to disk, instead kept in buffer pool until is evicted.

Figure 4(b) illustrates the write pattern of Wiredtiger's collection file such that there is a different in density of writes occur between areas of file i.e. the bottom and the top. The reason is some areas are accessed more frequency than others due to the zipfian distribution from YCSB workload [11]. The write pattern is mixed between sequential write and random write as showed with zoom in view from the point 200 seconds to 400 seconds in Figure 4(c). Typically, write pattern is expected as randomly due to the characteristics of YCSB workload, however, as WiredTiger disable direct IO as default, the dirty page write out is actually kept in OS cached before flushed to disk. Hence pages are sorted depended on IO scheduling algorithm in kernel, which make the write pattern become sequential.

The effect of checkpoint is showed near the end of benchmark in figure 4(b); there is a spare space in write pattern and write occurs in whole file area. When checkpoint is trigger, the overhead of whole system is high due to a lot of locked content occurs, all updates on recent versions are merged with the original on-disk image, there is a transformation between in-memory page and on-disk page occurred internally include encoding/decoding, compressions are carried out at this time, finally WiredTiger write dirty collection pages of whole area to files on disk.

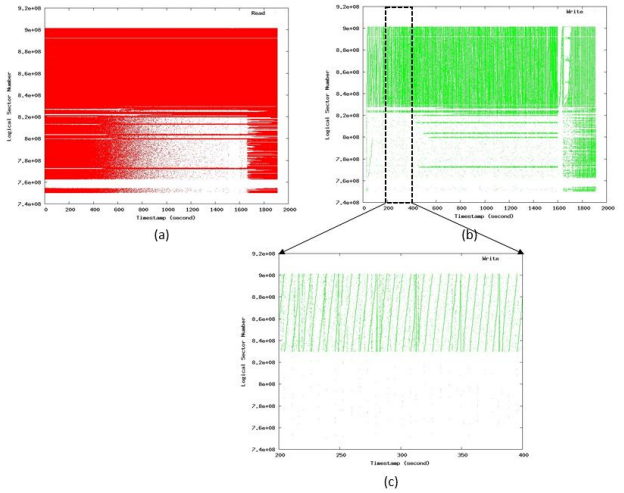

Figure 4. I/O patterns of (a) Read collection file, (b) Write collection file, (c) Zoom in view of write collection pattern

4.4 Index files and Journal files I/O Patterns

WiredTiger use B+Tree data structure to store both collection data and index data. For the primary index B+Tree, index leaf node entry stores the logical record id of collection that unchanged when the value of record is updated, so there is no index write occur during the benchmark is running. Index read pattern are randomly as showed in figure 5(a), it is heavy read at the begin of benchmark as all index pages need to fetch from disk to D-RAM before collection pages are read. Another highly index read is at checkpoint time i.e. the period time between point 1500 and 1600 in figure 5(a).

Figure 5(b) shows write pattern of journal file as all sequential writes. During benchmark time, before update requests are applied

on new version of collection data page, changes will be captured in log records and kept in log buffer in D-RAM; for each pre-defined interval time e.g. 100 milliseconds, log server flush log buffer to persistent log file. Before a log file reach its limited size e.g. 100MB, a new log file is created and ready for the next commit log fill, logical offset in log file is kept increase monotonically even though new file is created. There is an exception that in checkpoint time, the logical offset is reset to zero as show in figure 5(b).

We also observe I/O patterns of secondary index in WiredTiger by creating a new one in the same dataset. As showed in figure 5(d), when secondary index is created, there is writes occur on index file with the same fashion with writing in collection file. There is a tradeoff between read and write performance need to be carefully considered when using secondary index. Firstly, even though secondary index help speed up read performance, the write overhead for updating secondary index pages are high. Second, secondary indices consume more space on both D-RAM as well as on disk as showed in the figure. Lastly, as the checkpoint mechanism in WiredTiger, more data write means log files are consumed quicker and checkpoint occur earlier. In figure 5(c), the dashed line rectangle at the bottom is sectors consumed by primary index in figure 5(a), the two figure clearly show that there is majority space in index file occupied by secondary index. For all those reasons, secondary index is not recommended for high write intensive workload.

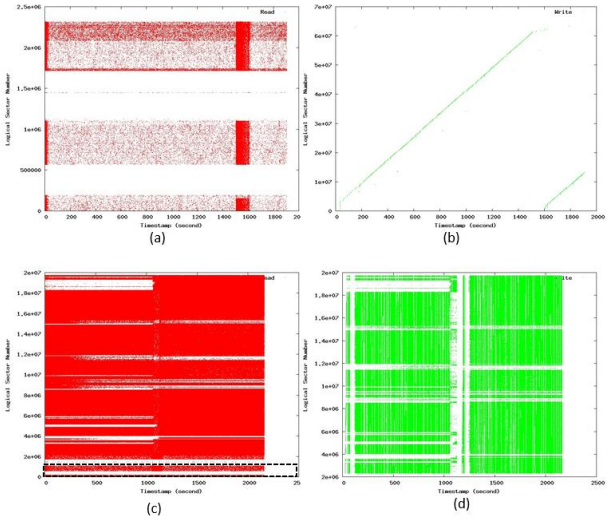

Figure 5. (a) Primary index file read pattern, (b) journal file write pattern, (c) primary and secondary index file read pattern, and (d) primary and secondary index file write pattern

4.5 The effect of direct IO

As default, WiredTiger disables direct IO for the sack of performance such that writes are cached in OS and can quickly return without waiting for the write finish on persistent disk. However, OS cache may hint the real I/O pattern of the DBMS system i.e. change the desired order of I/Os due to IO scheduling mechanism e.g. LRU. Moreover, OS cache overhead probably effect the performance unexpectedly, especially for fast storage device e.g. SSD. For those reasons, almost DBMSs have a feature that disable OS cached by using direct IO. To evaluate the effect of direct IO on WiredTiger, we enable that feature and run the same YCSB benchmark with ten million records on a typical hard disk and a SSD.

Figure 6 show the I/O patterns and performance of benchmark on both HDD and SSD. As expected, the I/O patterns changed when enable direct IO in both Figure 6(b) and Figure 6(d). First, when direct IO enable, the write patterns become denser because page writes are not kept in OS cached anymore, instead come direct to block layer of device. Second, when direct IO enable, the IO scheduling from OS is eliminated so the patterns remain random fashion from the workload. Finally and surprisingly, direct IO effects differently on HDD and SSD. With HDD, random writes are slow due to the high seek time of disk head, OS cached not only reduces latency time by return write request immediately but also orders write in sequential fashion when it flush to disk. Hence direct IO makes the performance in HDD reduce significantly as showed in figure 6(a) and figure 6(b). In other hand, SSD has fast random write, especially with modern SSD e.g. Samsung SSD Pro 840, the overhead of OS cached is greater than the latency of random write that lead to the performance turn out improves when direct IO is enable.

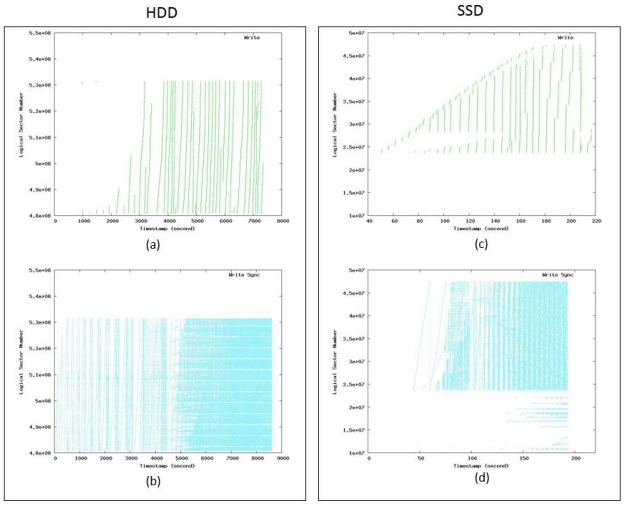

Figure 6. Effect of direct IO on HDD (a) direct IO off (b) direct IO on; and on SSD: (c) direct IO off, and (d) direct IO on.

4.6 TRIM command optimization evaluation

In order exploit space management of WiredTiger using TRIM command, we modified original source code such that whenever address replacement occurs, the logical invalid offset is kept track of. Because there is an overhead of using TRIM command, when address replacement occurs, the eviction server is very busy on checking page on LRU queue, as well as reconciling candidate pages, we delay calling commands at that time, instead accumulate until a threshold is reached, we call this threshold is trim frequency such that for a trim frequency value k, we save k old addresses then call TRIM commands for those at once. Table 3 shows the performance of TRIM command optimization in WiredTiger with the last column shows OPS/s improved percentage of optimized methods with different trim frequency, compare to the original WiredTiger as the baseline. When the trim frequency is small i.e. 10,000, there is a little improvement in OPS/s that just more than five percentages, due to the overhead of TRIM still high. The performance improved to more than ten percentages when the frequency high enough e.g. 15,000 and

20,000 as showed in the table accordingly. However, when we continue increased the frequency, the performance went down for case of 30,000 and even lower than the baseline. The reason is in such situation, the overhead of merging address space is considered, and sending a huge number of TRIM commands at once stress out the whole system.

Table 3. TRIM command performance with YCSB benchmark

Method	Run time (seconds)	OPS/s	OPS/s improved (%)
Original	7593	3950	0
TRIM-10000	7211	4159	+5.29
TRIM-15000	6682	4489	+13.6
TRIM-20000	6620	4531	+14.7
TRIM-30000	7188	4173	+5.64
TRIM-40000	8076	3714	-5.9

5. CONCLUSION AND FUTURE WORK

In this paper we brought the background internals of WiredTiger storage engine in detail as well as examined the I/O characteristics of typical files. Collection file is most bottleneck accessed that show randomly read and mixed random-sequential write patterns when direct IO is disable. For update workloads, primary index only has random read pattern while secondary index show both random read and mixed random-sequential write patterns as collection file. Journal files always have sequential write pattern and the offset is reset whenever the checkpoint occurs. One should consider the effect of checkpoint due to it has high overhead and mainly contribute to the overall performance. Direct IO changes the I/O pattern due to eliminating the role of OS cache. Moreover, it show that HDD get benefit from disable direct IO. This paper just focus on stand-alone MongoDB server.

We exploited WiredTiger's space management by delay calling TRIM commands for address replacements on flash-based SSD. The performance is improved more than ten percentages with carefully tuning value of trim frequency. Next work will examine cluster setting in distributed environment with sharding and replicas in MongoDB. Due to the whole dataset need to be horizontal split during sharding, the I/O patterns may interested. Other flash-based optimization techniques e.g. Multi-streamed SSD will be examined.

6. ACKNOWLEDGMENTS

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP). (R0126-16-1108, NVRam Based High Performance Open Source DBMS development), and Samsung Electronics.

7. REFERENCES

[1] Sadalage, Pramod J., and Martin Fowler. NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Pearson Education, 2012.

[2] Stonebraker, M. (2010). SQL databases v. NoSQL databases. *Communications of the ACM*, 53(4), 10-11. DOI = DOI:10.1145/1721654.1721659.

[3] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ... & Vogels, W. (2007). Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6), 205-220.

[4] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., ... & Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 26(2), 4.

[5] Lee, S. W., Moon, B., Park, C., Kim, J. M., & Kim, S. W. (2008, June). A case for flash memory SSD in enterprise database applications. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (pp. 1075-1086). ACM.

[6] Schindler, J. (2012). I/O characteristics of NoSQL databases. Proceedings of the VLDB Endowment, 5(12), 2020-2021.

[7] Kim, S. H., Kim, J. S., & Maeng, S. (2012). Using solid-state drives (SSDs) for virtual block devices. Proceedings of the runtime environments, systems, layering and virtualized environments (RESoLVE'12).

[8] Oh, G., Seo, C., Mayuram, R., Kee, Y. S., & Lee, S. W. (2016, June). SHARE Interface in Flash Storage for Relational and NoSQL Databases. *In Proceedings of the 2016 International Conference on Management of Data* (pp. 343-354). ACM.

[9] WiredTiger, http://source.wiredtiger.com/2.7.0/index.html.

[10] The MongoDB 3.2 Manual https://docs.mongodb.com/manual/

[11] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010, June). Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM symposium on Cloud computing (pp. 143-154). ACM.