Optimizing MongoDB Using Multi-streamed SSD

Trong-Dat Nguyen(✉) and Sang-Won Lee

College of Information and Communication Engineering,
Sungkyunkwan University, Suwon 16419, Korea
{datnguyen,swlee}@skku.edu

Abstract. Data fragmentation in flash SSDs is a common problem that leads to performance degradation, especially when the underlying storage devices become aged by heavily updating workloads. This paper addresses that problem in MongoDB, a popular document storage in the current market, by introducing a novel stream mapping scheme that based on unique characteristics of MongoDB. The proposal method has low overhead and independent with data models and workloads. We use YCSB and Linkbench with various cache sizes and workloads to evaluate our proposal approaches. Empirical results shown that in YCSB and Linkbench, our methods improved the throughput by more than 44% and 43.73% respectively; reduced 99th-percentile latency by up to 29% and 24.67% in YCSB and Linkbench respectively. In addition, by tuning the leaf page size in B+Tree of MongoDB, we can significantly improve the throughput by 3.37x and 2.14x in YCSB and Linkbench respectively.

Keywords: Data fragmentation · Multi-streamed SSD · Document store · Optimization · MongoDB · WiredTiger · Flash SSD · NoSQL · YCSB · Linkbench

1 Introduction

Flash solid state drives (SSDs) have several advantages over hard drives e.g. fast IO speed, low power consumption, and shock resistance. One unique characteristic of NAND flash SSDs is "erase-before-update" i.e. one data block should be erased before writing on new data pages. *Garbage collection (GC)* in flash SSD is responsible for maintaining free blocks. Reclaiming a non-empty data block is expensive because: (1) erase operation itself takes orders of magnitude slower than read and write operations [1], and (2) if the block has some valid pages, GC first copy back those pages to another empty block before erasing the block.

Typically, locality of data access has a substantial impact on the performance of flash memory and its lifetime due to wear-leveling. IO workload from client queries has skewness i.e. small proportion of data that has frequently accessed [10,11,15]. In flash-based storage systems, hot data identification is the process of

W. Lee et al. (eds.), *Proceedings of the 7th International Conference on Emerging Databases*, Lecture Notes in Electrical Engineering 461,
https://doi.org/10.1007/978-981-10-6520-0_1

distinguishing *logical block addresses (LBAs)* that have frequently accessed data (*hot data*) with the others less frequently accessed data (*cold data*). Informally, *data fragmentation* in flash SSD happens when writing data pages with different lifetimes to a block in an interleaved way. In that case, one physical block includes hot data and cold data which in turn increases the overhead of reclaiming blocks significantly. Prior researchers solved the problem by identifying hot/cold data either based on history address information [12] or based on update frequency [13,14]. However, those approaches had a degree of overhead for keeping track of metadata in DRAM as well as CPU cost for identifying hot/cold blocks. Min et al. [16] designed a Flash-oriented file system that groups hot and cold segments according to write frequency. In another approach, TRIM command is introduced to aid upper layers in user space and kernel space notifying flash FTL which data pages are invalid and no longer needed, thus reducing the GC overhead by avoiding unnecessary copy back of those pages when reclaiming new data blocks [18].

Recently, NoSQL solutions have become popular and been alternatives to traditional relational database management systems (RDBMSs). Among many NoSQL solutions, MongoDB[1] is one of the representative document stores with WiredTiger[2] as the default storage engine that shares many common characteristics with traditional RDBMS such as transaction processing, multi-version concurrency control (MVCC), and secondary index supporting. Moreover, there is a conceptual mapping between MongDB's data model and traditional table-based data model in RDBMS [7]. Therefore, MongoDB is interested not only by developers from industrial but also from researchers in academia. Most of the researchers compared between RDBMSs and NoSQLs [3,4], addressing data modeling transformation [8,9] or load-balanced sharding [5,6].

Performance degradation due to data fragmentation also exists in NoSQL solutions with SSDs as the underlying storage devices. For example, NoSQL DBMSs such as Cassandra and RocksDB take the log-structured merge (LSM) tree [21] approach have different update lifetime for files in each level of LSM tree. Kang et al. [10] proposed a *Multi-streamed SSD (MSSD)* technique to solve data fragmentation in Cassandra. The key idea is assigning different *streams* to different file types then groups data pages with similar update lifetimes into same physical data blocks. Extended from the previous research, Yang et al. Adopting file-based mapping scheme from Cassandra to RocksDB is inadequate because in RocksDB, there are concurrency compaction threads that compact files into several files. Therefore writes on files with different lifetime are located in the same stream. To address that problem, Yang et al. [11] extended the previous mapping scheme with a novel stream mapping with locking scheme for RocksDB.

To the best of our knowledge, no study has investigated on data fragmentation problem in MongoDB using multi-streamed SSD technique. Nguyen et al. [17] exploited TRIM command to reduce overhead in MongoDB. However, TRIM command does not entirely solve data fragmentation [11]. WiredTiger uses

[1] https://www.mongodb.com/mongodb-architecture.
[2] http://source.wiredtiger.com/2.7.0/index.html.

B+Tree implementation for its collection files as well as index files. However, the page sizes of internal pages and leaf pages in collection files are not equal i.e. 4 KB and 32 KB respectively. Meanwhile, the smaller page size is known to work better for flash SSD because it can help reducing *write amplification* ratio [2], so we can further improve throughput in WiredTiger by tuning smaller leaf page size.

In this paper, we propose a novel boundary-based stream mapping to exploit the unique characteristic of WiredTiger. We further extend the boundary-based stream mapping by introducing an on-line high efficient stream mapping based on data locality. We summarize our contributions as below:

- We investigated WiredTiger's block management in detail and pointed out two causes for data fragmentation: (1) writing on files have different lifetimes, and (2) there is internal fragmentation in collection files and index files. We adopt a simple stream mapping scheme based on those observations that map each file types with different streams. Further, we proposal a novel stream mapping scheme for WiredTiger based on the boundaries on collection files or index files. This approach improves the throughput in YCSB [19] and Linkbench [20] up to 44% and 43.73% respectively, improved the 99th-percentile latency in YCSB and Linkbench up to 29% and 24.67% respectively.
- We suggested a simple optimization of changing the leaf page size in B+tree from its default value 32 KB to 4 KB. In combination with the multi-streamed optimization, this simple tuning technique improved the throughput by three-fold and 2.16x for YCSB and Linkbench respectively.

The rest of this paper is organized as follow. Section 2 explains the background of multi-streamed SSD and MongoDB in detail. Proposal methods are described in Sect. 3. We explain the leaf page size optimization in Sect. 4. Section 5 discusses evaluation results and analysis. Lastly, the conclusion is given in Sect. 6.

2 Background

2.1 Multi-streamed SSD

Kang et al. [10] originally proposed the idea of mapping streams to different files so that data pages with similar update lifetime are grouped in the same physical block. Figure 1 illustrates how different between regular SSD and multi-streamed SSD (MSSD) work. Suppose that the device had eight logical block addresses (LBAs) and divided into two groups: hot data (LBA2, LBA4, LBA6, LBA8), and cold data are remains. There are two write sequences for both regular SSD and MSSD. The first sequence is written continuously from LBA1 to LBA8, and then the second write sequence only includes hot LBAs i.e. LBA6, LBA2, LBA4, and LBA8.

In regular SSD, after the first write sequence, LBAs are mapped to block 0 and block 1 according to write order regardless of hot or cold data.

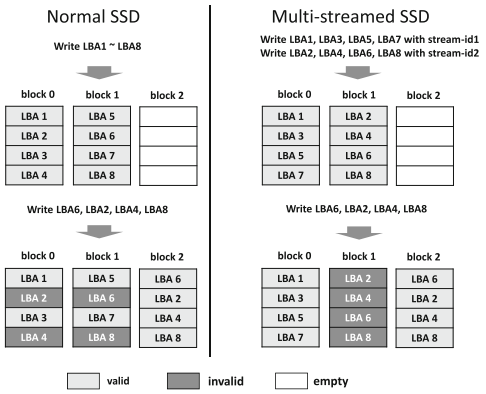

Fig. 1. Comparison between normal SSD and multi-streamed SSD

When the second write sequence occurs, new coming writes are done in empty block 2, corresponding old LBAs become invalid in block 0 and block 1. If the GC process reclaims block 1, there is an overhead for copying back LBA5 and LBA7 to another free block before erasing block 1.

The write sequences are similar in MSSD; however, in the first write sequence, LBAs assigned to a corresponding stream according to their hotness values. Consequently, all hot data grouped into block 1. After the second write sequence finished, all LBAs in block 1 become invalid and erasing block 1 in such case is quite fast, due to the copying back overhead is eliminated.

2.2 MongoDB and WiredTiger

MongoDB and RDBMS. Document store shares many similar characteristics to traditional RDBMS such as transaction processing, secondary indexing, concurrency controlling. MongoDB has emerged as standard document stores in NoSQL solutions. There is a conceptually mapping between the data model in RDBMS and the one in MongoDB. While *database* concept is same for both models; *tables*, *rows*, and *columns* in RDMBS can be seen as *collections*, *documents*, and *document fields* in MongoDB, respectively. Typically, MongoDB encodes documents as BSON[3] format and uses WiredTiger as the default storage engine since the version 3.0. WiredTiger uses B+Tree implementation for collection files as well as index files. In collection file, maximum page sizes are 4 KB and 32 KB for internal pages and leaf pages respectively. From now on, we use WiredTiger and MongoDB interchangeably unless there is some specific distinguishes.

Block Management. Understanding internal block management of WiredTiger is the key to optimizing the system using MSSD approach.

[3] http://bsonspec.org/spec.html.

WiredTiger uses *extents* to represent location information of data blocks in memory i.e. offsets and sizes.

Each checkpoint keeps track of three linked lists of extents for managing allocated space, discard space, and free space, respectively. WiredTiger keeps only one special checkpoint called *live checkpoint* in DRAM that includes block management information for the current working system. When a checkpoint is called, before writing the current live checkpoint to disk, WireTiger fetches the previous checkpoint from the storage device to DRAM; then merges its extent lists with the live checkpoint. Consequently, reused allocated space from the previous checkpoint after the merging phase finished. During the checkpoint time, WiredTiger discards unnecessary log files and the reset the log write offset to zero.

An important observation is that, once a particular region of the storage device is allocated in a checkpoint, it is reused again in the next checkpoint. That forms an internal fragmentation in the storage device that leads to the high overhead of GC process if the underlying storage is SSD. Next section discusses this problem in detail.

3 Boundary-Based Stream Mapping

3.1 Asymmetric Amount of Data Written to File Types

The amount of data written to files is a reliable criterion to identify the bottleneck of the storage engine and the root cause of data fragmentation that leads to high overhead in GC process.

Table 1. The proportions of data written to file types with various of workloads

Benchmark	Operation ratio C:R:U:D	Colls	Pri. Indexes	2nd indexes	Journal
Y-Update-Heavy	0:50:50:0	93.6	n/a	n/a	6.4
Y-Update-Only	0:0:100:0	89.6	n/a	n/a	10.4
LB-Original	12:69:15:4	58.6	3.1	37.22	1.08
LB-Mixed	12:0:84:4	66.1	0.5	31.13	2.27
LB-Update-Only	0:0:100:0	67.6	0.02	30.2	2.18

Table 1 shows the proportions of data written to collection files, index files and journal files under various workloads with different operations i.e. create, read, update, delete (CRUD). For simple data model, YCSB workload A (Y-Update-Heavy), and YCSB only update workload (Y-Update-Only) are carried out. To experiment more complex data model, we use original Linkbench workload (LB-Original), mixed operations workload (LB-Mixed), and only update Linkbench workload (LB-Update-Only). Write ratios to other files

e.g. metadata, system data, are too small i.e. less than 0.1%, that can be excluded from the table.

As observed from the table, write distributions to file types are different depending on the CRUD ratios of the workload. In YCSB benchmark, since the data model is simple key-value with only one collection file and one primary index file, almost writes are on collection file, and there is no update on primary index file. In Linkbench, however, collection files and secondary index files are hot data which have frequency accessed, primary index files and journal files are cold data that receive the low proportion of writes i.e. less than 5% in total. This observation implicates that difference written ratios in file types lead to hot data and cold data locate in the same physical data block in SSD that result in high overhead in GC process as explained in the previous section.

To solve this problem, we use a simple file-based optimization that assigns different streams for different file types. To minimize the overhead of the system, we assign a file to a corresponding stream only when open that file. Table 3 in Sect. 5 describes the detail of stream mapping in file-based method.

3.2 Multi-streamed SSD Boundary Approach

We further analyze the write patterns of WiredTiger to improve the optimization. We define write region (*region* in short) is the area between two logical file offsets that data is written on in a duration of time. Figure 2 illustrates the written patterns of different file types in the system under Linkbench benchmark with LB-Update-Only workload in two hours using *blktrace*. The x-axis is the elapsed time in seconds, the y-axis is the file offset. *DirectIO* mode is used to eliminate the effect of Operating System cache. Collection file and secondary index file have heavily random write pattern on two regions i.e. *top* and *bottom* that separated by a boundary in dashed line as illustrated in Fig. 2(a), and (c). In the other hand, the primary index file and journal file follow sequence write patterns as illustrated in Fig. 2(b), and (d) respectively.

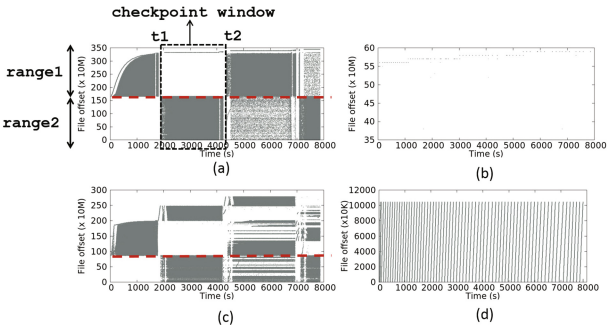

Fig. 2. Write patterns of various file types in WiredTiger with Linkbench benchmark, (a) Collection file, (b) primary index file, (c) secondary index, and (d) journal file

Algorithm 1. Boundary-based stream mapping

```
1: Require: boundary of each collection file and index file has computed
2: Input: file, and offset to write on
3: Output: sid - stream assign for this write
4: boundary ← getboundary(file)
5: if file is collection then
6:     if offset < boundary then
7:         sid ← COLL_SID1
8:     else
9:         sid ← COLL_SID2
10: else if file is index then
11:     if offset < boundary then
12:         sid ← IDX_SID1
13:     else
14:         sid ← IDX_SID2
15: else                                   ▷ Other files i.e. metadata
16:     sid ← OTHER_SID
```

One important observation is that, at a given point of time, the amount of data written to two regions i.e. *top* and *bottom* is asymmetric and switches after each checkpoint. In this paper, we call that phenomenon is *asymmetric regions writing*. Due to the asymmetric regions writing phenomenon, for a given file, there is an *internal fragmentation* that dramatically affects to the overhead of GC in SSDs. Obviously, file-based optimization is inadequate to solve the problem. In this approach, writes on one file are mapped with one stream, thus inside that stream, internal fragmentation still occurs. Therefore, we proposal a novel stream assignment named *boundary-based* stream mapping. The key idea is using the file boundary that separates the logical address of a given file to the top region and the bottom region. As described in Algorithm 1, firstly, the boundary of each collection and index file is computed as the last file offset after the load phase finished. Then in query phase, before writing a block data on a given file, the boundary is retrieved again as in line 4, based on the boundary values and the file types, the stream mapping is carried out as in line 7, 9, 12, 14, and 16. After stream id is mapped, the write command to the underlying file is given as *posix_fadvise(fid, offset, sid, advice)*, where *fid* is file identify, *offset* is the offset to write on, *sid* is stream id mapped and *advice* is passed as a predefined constant.

4 B+Tree Leaf Page Size Tuning

WiredTiger uses B+Tree to implement collection files and index files. Accesses to internal pages are usually more frequent than leaf pages. Due to page replacement policy in the buffer pool, internal pages are kept in DRAM longer than leaf pages. So there is an asymmetric amount of data written to components of B+Tree as presented in Table 2. We keep track of the number of writes on each component of B+Tree by modifying the original source code of WiredTiger. Exclude from

typical components i.e. root page, internal page, and leaf page, *extent page* is a special type that only keeps metadata for extent lists in a checkpoint. For only update workload, while writes only occur on collection file in YCSB, both collection files and index files in Linkbench are updated. Because root pages and internal pages are accessed more frequent than leaf pages, WiredTiger keeps them in DRAM as long as possible and mostly writes them to disk at the checkpoint time belong with extent pages. In the other hand, leaf pages are flushed out not only at the checkpoint time but also at the normal thread through evicting dirty pages from buffer pool in the reconciliation process. Therefore, more than 99% of the total writes occur on leaf pages.

Table 2. Percentage of writes on page types in collection files and index files in YCSB and Linkbench

Benchmark	Collection (%)				Index (%)			
	Root page	Int. page	Leaf page	Ext. page	Root page	Int. page	Leaf page	Ext. page
YCSB	$5e^{-5}$	0.27	99.72	$9.3e^{-5}$	0	0	0	0
Linkbench	$7e^{-5}$	0.61	39.86	$14e^{-5}$	$13e^{-5}$	0.28	59.23	$26e^{-5}$

Note that the default sizes for internal pages and leaf pages are 4 KB and 32 KB respectively. Large leaf page size leads to high write amplification such that some bytes update from workload lead to whole 32 KB data page written out to disk. It becomes worse with heavy random update workload such that almost 99 percent of writes occur on leaf pages. We suggest a simple but effective tuning that decreases the leaf page size from its default 32 KB to 4 KB. This changing improves the throughput significantly as discussed in the next section.

5 Evaluation and Analysis

5.1 Experimental Settings

We conducted the experiments with YCSB 0.5.0[4] and LinbenchX 0.1[5] (an extended version of Linkbench that supports MongoDB) as the top client layer and used various of workloads as illustrated in Table 1. We use 23 million 1-KB documents in YCSB and *maxid1* equal to 80 million in Linkbench respectively. In the server-side, we adopt a stand-alone MongoDB 3.2.1[6] server with WiredTiger as storage engine. Cache sizes vary from 5 GB to 30 GB, other settings in WiredTiger are kept as default. To enable multi-streamed technique, we use a modified Linux kernel 3.13.11 along with customized Samsung 840 Pro as in [11].

[4] https://github.com/brianfrankcooper/YCSB/releases/tag/0.5.0.
[5] https://github.com/Percona-Lab/linkbenchX.
[6] https://github.com/mongodb/mongo/archive/r3.2.1.tar.gz.

To exclude the network latency, we setup the client layer and the server layer on the same commodity server with 48 cores Intel Xeon 2.2 GHz processor, 32 GB DRAM. We execute all benchmarks during 2 hours with 40 client threads.

5.2 Multi-streamed SSD Optimization Evaluation

To evaluate the effect of our proposal multi-streamed SSD based methods we conducted experiments with different stream mapping schemes as shown in Table 3. In the original WiredTiger, there is no stream mapping; thus all file types use stream 0 that reserve for files in Linux Kernel as default. In file-based stream mapping, we used total four streams that map each stream to a file type. In boundary-based stream mapping, metadata files and journal files are mapped in the same way with the file-based approach. The different is that there is two streams map with collection files, one for all top regions and another for the bottom regions. We map streams for index files in the same manner without considering primary index files or secondary index files.

Figure 3 illustrates the throughput results for various benchmarks and workloads. Note that in Linkbench benchmark with maxid1 equals to 80 million, the total index size is quite large i.e. 33 GB that requires the buffer pool size large enough to keep almost index files in DRAM. In addition, in LB-Original workload that exist read operations, pages tend to fetched in and flush out buffer

Table 3. Stream mapping schemes

Method	Kernel	Metadata	Journal	Collections	Indexes
Original	0	0	0	0	0
File-based	0	1	2	3	4
Boundary-based	0	1	2	3,4	5,6

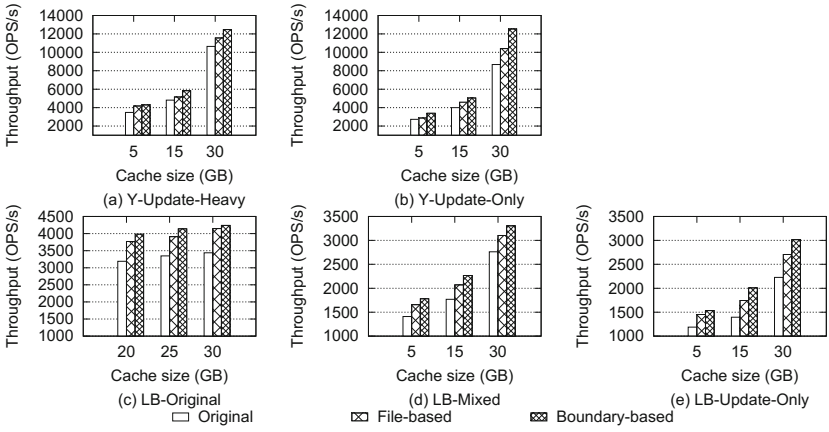

Fig. 3. Throughput of optimized methods compared with the original

pool more frequent hence we use large cache sizes i.e. 20 GB, 25 GB, and 30 GB in LB-Original workload. In general, multi-streamed based methods have greater throughput than the original. The more frequent writing the workload has, the more benefit multi-streamed based approaches gain.

In YCSB benchmark, boundary-based shows the throughput improve up to 23% at 5 GB cache size and 44% at the cache size is 30 GB in Y-Update-Heavy and Y-Update-Only workload respectively. For Linkbench benchmark, the boundary-based method has throughput improve up to 23.26%, 28.12%, and 43.73% for LB-Original, LB-Mixed, and LB-Update-Only respectively. In the YCSB benchmark, the percentage of throughput improvement of the boundary-based method has remarkable gaps compared with file-based that up to approximate 14% and 24.4% for Y-Update-Heavy and Y-Update-Only respectively. However, those differences become smaller in Linkbench that just 6.84%, 11% and 18.8% for LB-Original, LB-Mixed, and LB-Update-Only respectively. For NoSQL applications in distributed environment, it is also important to consider the 99th-percentile latency of the system to ensure clients have acceptable response times. Figure 4 shows the 99th-percentile latency improvements of multi-streamed based methods compared with the original. Overall, similar with throughput improvement, 99th-percentile latency correlates with the overhead of the GC hence the better one method solve data fragmentation, the lower 99th-percentile latency it reduces.

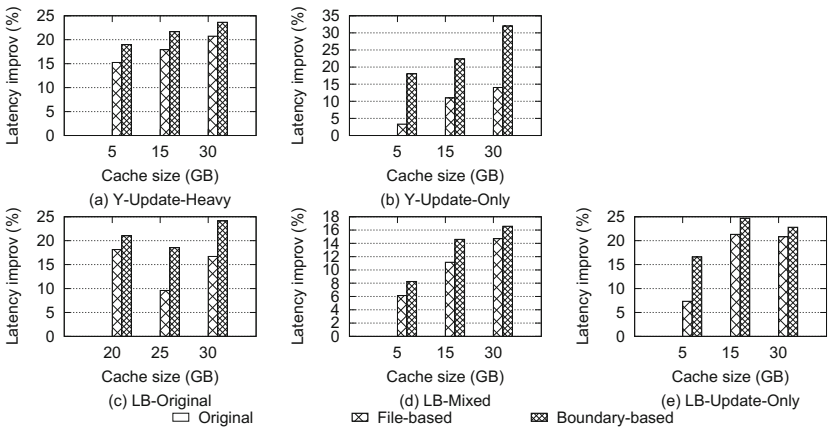

Fig. 4. Latency of optimized methods compared with the original

Boundary-based is better than file-based method. In YCSB, compared with the original WiredTiger, the boundary-based method reduces the 99th-percentile latency 29.3%, 29% for Y-Update-Heavy, Y-Update-Only, respectively. In Linkbench, it reduces up to 24.13%, 16.56%, and 24.67% for LB-Original, LB-Mixed, and LB-Update-Only respectively. Once again, boundary-based benefits

in reducing the latency in simple data model i.e. YCSB decrease a little in complex data model i.e. Linkbench.

5.3 Leaf Page Size Optimization Evaluation

To evaluate the impact of leaf page size we conducted the experiment on original WiredTiger as well as our proposal method with YCSB benchmark and Linkbench using various workloads and cache sizes for 32 KB leaf page size (default) and 4 KB leaf page size. For the space limitation in the paper, we only show the throughput results of the heaviest write workloads i.e. Y-Update-Only and LB-Update-Only as in Fig. 5. Overall, compared with the original WiredTiger 32-KB as the based line, Boundary-based-4 KB leaf page shows dramatically improvement of throughput that up to 3.37x and 2.14x for Y-Update-Only and LB-Update-Only respectively. In YCSB, with the same method, changing leaf page size from 32 KB to 4 KB increase the throughput sharply triple or double. In Linkbench, however, reducing leaf page size from 32 KB to 4 KB has the maximum throughput improvement are 1.96x, 1.4x, and 1.5x for the original method, the file-based method, and the boundary-based method respectively. Note that in Linkbench, small leaf page size optimization lost its effect with small cache size i.e. 5 GB. The reason is with the same maxid1 value, reducing the leaf page size from 32 KB to 4 KB increases the number of leaf pages and the number of internal pages in the B+Tree that lead to collection files and index files become larger and require more space from the buffer pool to keep the hot index files in DRAM.

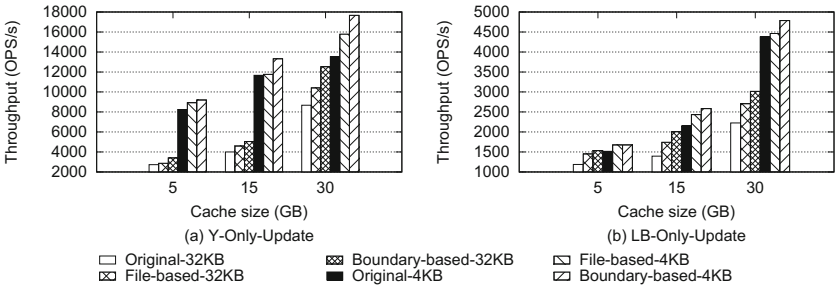

Fig. 5. Throughput of optimized methods compared with the original

6 Conclusion

In this paper, we discussed data fragmentation in MongoDB in detail. The file-based method is the simplest one that solves the data fragmentation due to the different lifetime of writes on file types but remains internal fragmentation caused by asymmetric regions writing. For simple data model in YCSB, the boundary-based approach is adequate to solve the internal fragmentation that

shows good performance improvement but lost its benefits with the complex data model in Linkbench. In addition, reducing the maximum leaf page size in collection files or index files from 32 KB to 4 KB can gain significant improvement in throughput in both YCSB and Linkbench. In general, our proposal approaches can adopt to any storage engine that has similar characteristics with WiredTiger i.e. asymmetric files writing and asymmetric region writing. Moreover, we expect to further optimize the WiredTiger storage engine by solving the problem of boundary-based with complex data model i.e. Linkbench in the next research.

Acknowledgments. This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the "SW Starlab" (IITP-2015-0-00314) supervised by the IITP (Institute for Information & communications Technology Promotion).

References

1. Lee, S.W., Moon, B., Park, C., Kim, J.M., Kim, S.W.: 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 (2008). doi:10.1145/1376616.1376723
2. Lee, S.W., Moon, B., Park, C.: Advances in flash memory SSD technology for enterprise database applications. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pp. 863–870 (2009)
3. Aboutorabi, S.H., Rezapour, M., Moradi, M., Ghadiri, N.: Performance evaluation of SQL and MongoDB databases for big e-commerce data. In: International Symposium on Computer Science and Software Engineering (CSSE), pp. 1–7. IEEE, August 2015. doi:10.1109/CSICSSE.2015.7369245
4. Boicea, A., Radulescu, F., Agapin, L.I.: MongoDB vs oracle-database comparison. In: EIDWT, pp. 330–335, September 2012
5. Liu, Y., Wang, Y., Jin, Y.: Research on the improvement of MongoDB auto-sharding in cloud environment. In: 7th International Conference on Computer Science and Education (ICCSE), pp. 851–854. IEEE (2012). doi:10.1109/iccse.2012.6295203
6. Wang, X., Chen, H., Wang, Z.: Research on improvement of dynamic load balancing in MongoDB. In: 2013 IEEE 11th International Conference on Dependable, Autonomic and Secure Computing (DASC), pp. 124–130. IEEE, December 2013. doi:10.1109/DASC.2013.49
7. Zhao, G., Huang, W., Liang, S., Tang, Y.: Modeling MongoDB with relational model. In: 2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies (EIDWT), pp. 115–121. IEEE (2013). doi:10.1109/EIDWT.2013.25
8. Lee, C.H., Zheng, Y.L.: SQL-to-NoSQL schema denormalization and migration: a study on content management systems. In: 2015 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 2022–2026. IEEE, October 2015. doi:10.1109/SMC.2015.353
9. Zhao, G., Lin, Q., Li, L., Li, Z.: Schema conversion model of SQL database to NOSQL. In: 2014 Ninth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), pp. 355–362. IEEE, November 2014. doi:10.1109/3PGCIC.2014.137

10. Kang, J.U., Hyun, J., Maeng, H., Cho, S.: The multi-streamed solid-state drive. In: 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14) (2014)
11. Yang, F., Dou, K., Chen, S., Hou, M., Kang, J.U., Cho, S.: Optimizing NoSQL DB on flash: a case study of RocksDB. In: Ubiquitous Intelligence and Computing and 2015 IEEE 12th International Conference on Autonomic and Trusted Computing and 2015 IEEE 15th International Conference on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), pp. 1062–1069 (2015). doi:10.1109/uic-atc-scalcom-cbdcom-iop.2015.197
12. Hsieh, J.W., Kuo, T.W., Chang, L.P.: Efficient identification of hot data for flash memory storage systems. ACM Trans. Storage (TOS) **2**(1), 22–40 (2006). doi:10.1145/1138041.1138043
13. Jung, T., Lee, Y., Woo, J., Shin, I.: Double hot/cold clustering for solid state drives. In: Advances in Computer Science and Its Applications, pp. 141–146. Springer, Heidelberg (2014). doi:10.1007/978-3-642-41674-3_21
14. Kim, J., Kang, D.H., Ha, B., Cho, H., Eom, Y.I.: MAST: multi-level associated sector translation for NAND flash memory-based storage system. In: Computer Science and its Applications, pp. 817–822. Springer, Heidelberg (2015). doi:10.1007/978-3-662-45402-2_116
15. Lee, S.W., Moon, B.: Design of flash-based DBMS: an in-page logging approach. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 55–66. ACM, June 2007. doi:10.1145/1247480.1247488
16. Min, C., Kim, K., Cho, H., Lee, S.W., Eom, Y.I.: SFS: random write considered harmful in solid state drives. In: FAST, p. 12, February 2012
17. Nguyen, T.D., Lee, S.W.: I/O characteristics of MongoDB and trim-based optimization in flash SSDs. In: Proceedings of the Sixth International Conference on Emerging Databases: Technologies, Applications, and Theory, pp. 139–144. ACM, October 2016. doi:10.1145/3007818.3007844
18. Kim, S.H., Kim, J.S., Maeng, S.: Using solid-state drives (SSDs) for virtual block devices. In: Proceedings Workshop on Runtime Environments, Systems, Layering and Virtualized Environments, March 2012
19. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154 (2010). doi:10.1145/1807128.1807152
20. Armstrong, T.G., Ponnekanti, V., Borthakur, D., Callaghan, M.: LinkBench: a database benchmark based on the Facebook social graph. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 1185–1196. ACM, June 2013. doi:10.1145/2463676.2465296
21. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (LSM-tree). Acta Informatica **33**(4), 351–385 (1996). doi:10.1007/s002360050048