JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 35, 447-469 (2019)
DOI: 10.6688/JISE.201903_35(2).0012

DSM: A Low-Overhead, High-Performance, Dynamic Stream Mapping Approach for MongoDB

TRONG-DAT NGUYEN AND SANG-WON LEE+
College of Information and Communication Engineering
Sungkyunkwan University
Suwon, 16419 Korea
E-mail: {datnguyen; swlee}@skku.edu

For write-intensive workloads, reclaiming free blocks in flash SSDs is expensive due to data fragmentation problem that leads to performance degradation. This paper addresses that problem in MongoDB, a popular document store in the current market, by introducing a novel stream mapping scheme that exploits unique characteristics of MongoDB and multi-streamed technology. It dynamically assigns streams for corresponding writes according to their hotness values and distinguishes writes on primary index files from writes on secondary index files. The proposed method is high-performance, low-overhead, and independent of data models or workloads. Empirical results in Linkbench benchmark show that compared to the original WiredTiger our approach improves the throughput and reduces the 99th-percentile latency by up to 65% and 46.2% respectively. Compared to the best-performance in the prior research, our approach improves the throughput and reduces the 99th-percentile latency by up to 23% and 28.5% respectively. Distinguishing writes on primary index files from writes on secondary index files enhances the throughput and the 99th-percentile latency by up to 11.7% and 15.7% respectively. Moreover, by tuning the leaf page size in B+Tree of MongoDB, we can significantly improve the throughput by 1.6×−2.1× in Linkbench.

Keywords: data fragmentation, hot/cold data identification, multi-streamed SSD, NoSQL database, MongoDB

1. INTRODUCTION

NAND flash Solid state drives (SSDs) perform *erase-before-write* such that they erase a non-empty data block before writing new pages on that block [1, 2]. Because erase operations are orders of magnitude slower than read operations and write operations [3], flash SSDs write updated data in empty blocks and mark the old data pages as invalid instead of erasing the current data blocks. *Garbage Collection* (*GC*), a component of *Flash Translation Layer* (*FTL*) inside flash SSD, is responsible for reclaiming free blocks when the number of empty blocks is lower than a threshold. During this process, if a non-empty data block is selected as a victim for reclaiming, valid pages from that block need to be copied back to another empty block before the actual erase operation is done. That leads to increasing the overhead of the reclaiming process. Moreover, NAND flash blocks have a limited number of erase cycles; Flash FTLs use *wear-leveling*, a technique that ensures writes are distributed evenly among flash blocks, to enhance the lifespan of flash SSD.

The locality of data access has a significant impact on the performance of flash

Received October 10, 2017; revised March 26, 2018; accepted July 10, 2018.
Communicated by Chang-Tien Lu.
+ Corresponding author.

memory and its lifetime due to the high-overhead of reclaiming free blocks and wear-leveling. In practice, IO workloads from clients exist skewness, *i.e.*, a small proportion of data has frequently accessed [4-6]. That forms the hot *logical block addresses* (*LBAs*) (LBAs have frequently accessed) and the cold LBAs (remain LBAs have less frequently accessed) which are called in short as *hot data* and *cold data* respectively. *Data fragmentation* in flash SSD happens when one physical block includes hot data and cold data which in turn increase the overhead of reclaiming blocks significantly.

Data fragmentation problem can be solved by identifying hot/cold data either based on update frequency [7-9] or based on history address information [10]. However, those approaches need to keep track of metadata in DRAM and high-cost of CPU for identifying hot/cold blocks thus increase the overhead of the system. Min *et al.* [11] design a Flash-oriented file system that groups hot and cold segments according to write frequencies. Park *et al.* [12] use a write buffer in the SSD's controller to improve the performance of the system by separating sequential writes from random writes.

With the high volumes and varieties of generated data, *not-only SQL* (*NoSQL*) solutions have become the alternatives for traditional RDBMSs [13-17]. Data fragmentation in DBMSs is a common problem not only in RDBMS but also in NoSQL solutions. For instance, Cassandra and RocksDB take the *log-structured merge* (*LSM*) tree [18] approach that has different update lifetime for files in each level of the LSM tree. *Multi-streamed SSD* (*MSSD*) technique [4, 5] assign different *streams* to different file types in Cassandra, RocksDB, so that writes on the similar update lifetimes can issue on the same physical data blocks.

In the virtualization environment, data fragmentation exists in flash SSDs when virtual machines have different IO workloads but share the same physical storage devices. One solution to that problem is that writes from one virtual machine are mapped to the same stream so that the scheduler can provide the corresponding resource [19].

To the best of our knowledge, there are few works address the data fragmentation problem in MongoDB [20] – one of the common document stores with WiredTiger [21] as the default storage engine. Most of the researchers compare RDBMSs with NoSQLs [22-25], address data modeling transformation [15, 26-29] or improve load-balanced sharding [30, 31]. Murugesan *et al.* [32] argue different logging techniques in MongoDB and propose a simple log management model that is useful for profiling the system. Based on the unique characteristics of space management in WiredTiger, TRIM commands are used to reduce the overhead of MongoDB [33]. However, TRIM commands do not entirely solve the data fragmentation [4]. In another research, file-based approach and boundary-based approach are proposed to address the data fragmentation in MongoDB [34]. However, those methods still have a level of data fragmentation such that writes from regions with different lifetimes are mapped to the same stream. Also, writes on primary indexes have different patterns and lifetimes compare to writes on secondary indexes. The boundary-based method is inadequate to distinguish writes on those two index types.

To solve those problems, we propose a novel *dynamic stream mapping* (*DSM*) that is an online high-effective stream mapping based on hot/cold values of each data block. We summarize our contributions as below:

- First, by investigating WiredTiger's block management in detail and revisiting bound-

ary-based approach with a complex data model, *i.e.*, Linkbench benchmark [35], we point out two flaws existing in the approach: (1) writes from regions with different lifetime update are mapped to the same stream, and (2) writes from primary indexes mixed with writes from secondary indexes.
- Based on this observation, we propose a novel mapping scheme named Dynamic-Stream Mapping (DSM) that groups writes on corresponding streams based on their hotness values and separates writes on primary indexes from writes on secondary indexes. In Linkbench, compared to the original WiredTiger, DSM improves the throughput and the 99th-percentile latency by up to 65% and 46.2% respectively. Compared to the best-performance method in the prior work, our proposed method enhances the throughput and the 99th-percentile latency by up to 23% and 28.5% respectively. Moreover, index filtering gains additional 11.7% and 15.7% improvement in terms of throughput and 99th-percentile latency respectively.
- Lastly, we combine the leaf page size tuning with DSM. The final results improve the throughput by 1.6×–2.1× for Linkbench benchmark.

The rest of this paper is organized as follow. Section 2 explains the background of multi-streamed SSD and MongoDB in detail. We revisit the prior works in Section 3. Proposed methods and related algorithms are described in Section 4. Section 5 discusses the evaluation results and analysis. Lastly, the conclusion is given in Section 6.

2. BACKGROUND

This section provides a background of Multi-streamed SSD technique that originally used in Cassandra and exploited in our proposed method. Also, we briefly introduce the relationship between the data model in RDBMSs and MongoDB and discuss the block management mechanism of WiredTiger in detail.

2.1 Multi-Streamed SSD

Multi-streamed SSD is a technique allows applications from the *user space* or the *kernel space* explicitly assign a *stream* along with a *write* or *pwrite* system call. In the device layer, the FTL writes pages that have the same stream on the same physical SSD block [5]. Multi-streamed technique requires modifications of both OS's kernel and the SSD's firmware that are available in commercial products [36].

Fig. 1 compares the differences between the regular SSD and the multi-streamed SSD. Suppose that LBA2, LBA4, LBA6, and LBA8 are hot data; LBA1, LBA3, LBA5, and LBA7 are cold data. There are two write sequences occur on both SSDs. The first one continuously writes from LBA1 to LBA8 and the second one writes only hot data in the following order: LBA6, LBA2, LBA4, and LBA8. In the normal SSD, after the first sequence, the LBAs are appended in an empty block in order regardless of hot/cold data as shown in Fig. 1 (b). After the second write sequence, new LBAs are appended in block 2 according to their order. The old LBAs are marked as invalid in block 0 and block 1 as illustrated in Fig. 1 (c). In such case, if block 1 is a candidate for discarding in the GC processing, there are overheads of FTL for searching a new empty block and copying two valid LBAs (*i.e.*, LBA5 and LBA7) back to the empty block before erasing

the block 1.

Conversely, in multi-streamed SSD, writes are assigned to corresponding streams based on their frequency, *i.e.*, hotness value. After the first write sequence, all hot LBAs are located in block 1, and all cold LBAs are located in block 0 as shown in Fig. 1 (e). The second sequence appends data in block 2 and marks invalid LBAs as in regular SSD, but all invalid pages locate in block 1 as illustrated in Fig. 1 (f). Erasing block 1 is fast because the FTL updates the mapping table without extra cost for searching new empty block and copying back valid LBAs.

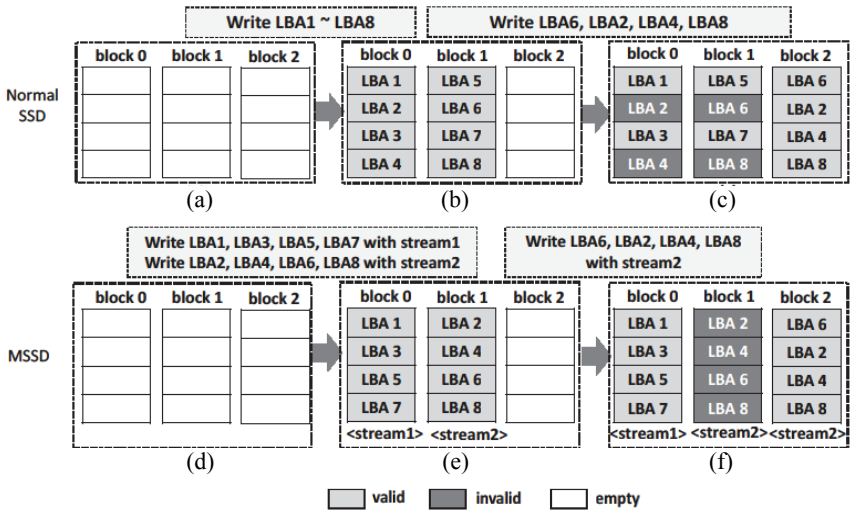


Fig. 1. Comparison between normal SSD and multi-streamed SSD.

2.2 MongoDB and WiredTiger

MongoDB is a popular document store in NoSQL solutions that shares many characteristics with RDBMS such as secondary index support, transaction processing, and concurrency control. The data model in MongoDB can be mapped to the one in RDBMS. Database concepts of both models are similar. *Collections*, *documents*, and *key-value pairs* in MongoDB are mapped to *tables*, *rows*, and *columns* in RDBMS respectively.

We research the internal block management of WiredTiger in detail to identify the causes of data fragmentation. WiredTiger keeps the metadata of free pages, allocated pages, and invalid pages in a special page called *checkpoint page*. There is only one checkpoint page is maintained in DRAM (*i.e.*, live checkpoint), the other checkpoint pages located in non-volatile devices such as disks. WiredTiger flushes dirty pages from DRAM to disks through either eviction processes of the buffer pool or checkpoint processes. WiredTiger neither adopts *in-place updates* as in traditional RDBMS nor *append-only* approach as in LSM-based DBMS. Upon a write request, the space management first searches for a free page to write on. Then after the data page is successfully written, WiredTiger marks the written page and the old page as valid and invalid respectively. During the checkpoint time, invalid pages are reclaimed and reused in the next checkpoint.

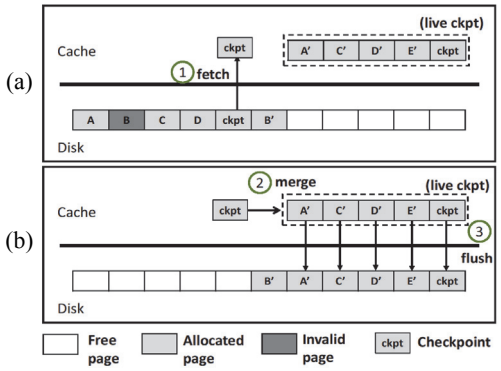


Fig. 2. Reusing data at checkpoint time in WiredTiger.

Fig. 2 illustrates how WiredTiger reuses invalid data at the checkpoint time. When an updated version of page B (*i.e.*, B′) is flushed to disk by an eviction process, the corresponding old version of page B is marked as invalid. Other updated pages (*i.e.*, A′, C′, D′, and E′) are kept in the live checkpoint in DRAM. Before flushing those pages to disk, WiredTiger fetches the previous checkpoint page from disk to DRAM (step 1) and merges its metadata with the metadata of the live checkpoint (step 2) as shown in Fig. 2 (a). Consequently, invalid data of the same pages exist in the current version are reclaimed and can be reused again after the checkpoint process finished. After the merging phase, all dirty pages along with the live checkpoint are flushed to disk (step 3) as illustrated in Fig. 2 (b).

This approach has two advantages: (1) Avoiding expensive compaction operations that are popular in LSM-based approaches, and (2) old versions can serve as the backups used in the recovery process. However, the approach has one drawback such that the valid data and invalid data of the same page are switched after each checkpoint, which forms an *internal fragmentation* in the SSD.

3. FILE-BASED STREAM MAPPING AND BOUNDARY-BASED STREAM MAPPING

We revisit the prior MSSD-based techniques in this section and argue the flaws in those methods by researching the IO patterns achieved from *blktrace*[1]. Moreover, we define the requirements that an optimized MSSD method should satisfy.

3.1 File-based Stream Mapping

Typically, different file types in DBMS have different data accessed frequencies and write patterns. As shown in Table 1, we use workloads with different operations such as create, read, update, delete (CRUD) from YCSB [37] and Linkbench [35] as in the previous research [34]. For simple data model in YCSB, workload A (Y-Update-Heavy), and only update workload (Y-Update-Only) are carried out. For complex data model in Linkbench, we use original Linkbench workload (LB-Original), mixed operations work-

[1] Blktrace is a block layer IO tracing tool in Linux that generates traces of the IO traffic on block devices.

Table 1. The proportions of data written to file types with several of workloads [34].

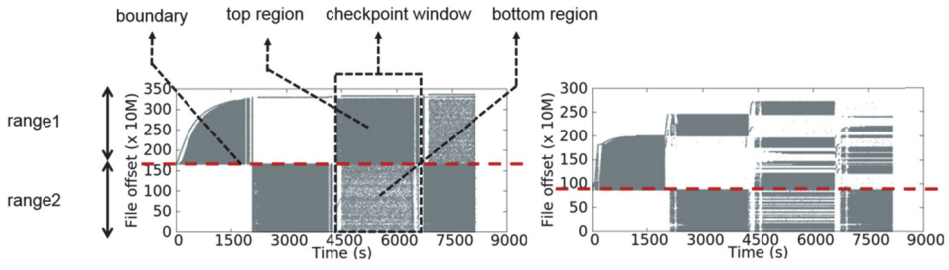
Benchmark	Operation ratio C:R:U:D	Collections	Primary Indexes	2nd Indexes	Journal
Y-Update-Heavy	0:50:50:0	93.60	n/a	n/a	6.40
Y-Update-Only	0:0:100:0	89.60	n/a	n/a	10.40
LB-Original	12:69:15:4	58.60	3.10	37.22	1.08
LB-Mixed	12:0:84:4	66.10	0.50	31.13	2.27
LB-Update-Only	0:0:100:0	67.60	0.02	30.20	2.18

load (LB-Mixed), and only update Linkbench workload (LB-Update-Only). Note that the table only includes information of most updated file types, the other minor updated file types, *e.g.*, metadata files, system files, lock files are excluded.

Almost updates from workloads are carried on the collection files and secondary index files. Because the data model in YCSB has only one collection and one primary index Y-Update-Heavy and Y-Update-Only do not write on the primary index. For MongoDB systems use SSDs as the storage devices, this asymmetric write in file types forms the data fragmentation such that frequently written file types (*i.e.*, collection and secondary index) are considered as hot data, and the others file types are considered as cold data. *File-based* MSSD approach solves this problem by mapping each file type to a distinguish stream. This simple optimization gains moderate improvement of performance; however, there is another kind of data fragmentation called *internal data fragmentation* that is discussed in the next subsection.

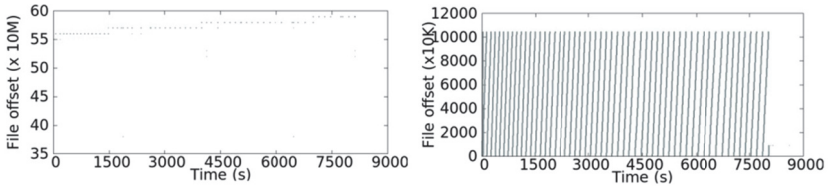
3.2 Boundary-based Stream Mapping

To illustrate the internal data fragmentation, we use *blktrace* while running the LB-Update-Only workload in two hours to keep track of information of each write command on the underlying storage device. Fig. 3 shows written patterns of different file types in the system. The x-axis is the elapsed time in seconds, and the y-axis is the file offset. To avoid the effect of Operating System's cache, we enable *DirectIO*. There are two types of written patterns: (1) heavy randomly writes that occur on the collection file and secondary index file as shown in Figs. 3 (a) and (b) respectively, and (2) sequential writes that occur in the primary index file and journal file as illustrated in Figs. 3 (c) and (d) respectively. *Checkpoint window* is the period between a checkpoint and its very next checkpoint. *Write region* (*region* in short) is the area limited by two logical file offsets (on the y-axis) and two time-points (on the x-axis), usually is a checkpoint window.



(a) Collection (b) Secondary index

Fig. 3. Write patterns of various file types in WiredTiger with Linkbench benchmark.



(c) Primary index (d) Journal

Fig. 3. (Cont'd) Write patterns of various file types in WiredTiger with Linkbench benchmark.

At a given checkpoint window, the amount of data written to the *top* region and the *bottom* region are asymmetric and switches in the next checkpoint window. For instance, in Fig. 3 (a), after the first checkpoint, the bottom region has high frequency written than the top region. Then after the second checkpoint, the trend is reversed such that the top region receives more writes compared to the bottom.

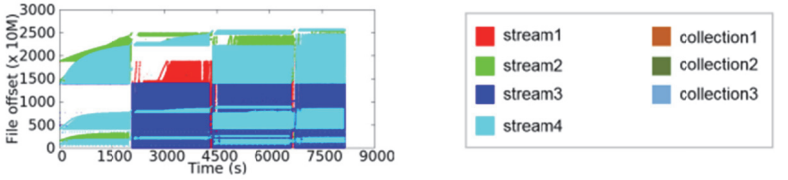
```
1: Require: boundary of each collection file and index file has computed
2: Input: file, and offset to write on
3: Output: sid – stream assign for this write
4: boundary ← getboundary(file)
5: if file is collection then
6:       if offset < boundary then
7:             sid ← COLL_SID1
8:       else
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
```

Fig. 4. Boundary-based stream mapping algorithm.

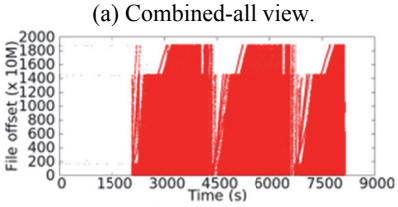
We revisit the *boundary-based* stream mapping approach in Fig. 4 [34]. The input is a *file* and file *offset* to write on. The output is the mapped stream *sid*. The first step is to retrieve the *boundary* of each collection file or index file which is the last file offset at the end of loading phase (line 4). Then from the rest of the algorithm, the corresponding stream is assigned to *sid* based on the input file type. The input file offset is compared with the boundary to determine whether the write is on the top region or the bottom region. 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.

Unfortunately, empirical results show that boundary-based approach remains some flaws in a complicated data model, *i.e.*, Linkbench rather than the simple one as in YCSB. Fig. 5 illustrates IO patterns of different file types using LB-Update-Only workload with y-axis is the file offset and x-axis is the elapsed time by second. We map writes on col-

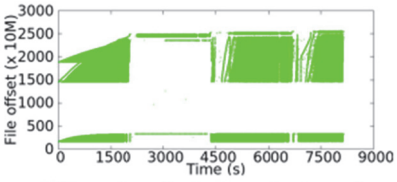
lections to stream1 and stream2, writes on indexes to stream3 and stream4 then represent four streams by four different colors as shown in Fig. 5 (a). The single-stream view of stream1 (*i.e.*, the bottom regions of collections) and the single-stream view of stream2 (*i.e.*, the top regions of collections) are shown in Figs. 5 (b) and (c) respectively. Inside one stream, however, regions of particular files have different written-lifetimes that lead to another unexpected overlapped writes as shown in Figs. 5 (d) and (e). We named that phenomenon as *cross-region fragmentation*.



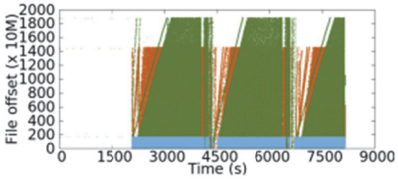
(a) Combined-all view.



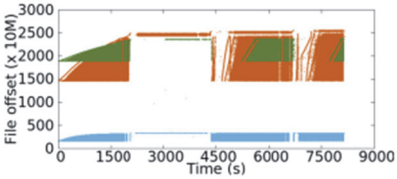
(b) Bottom regions of collections – by stream view.



(c) Top regions of collections by – stream view.



(d) Bottom regions of collections – by collection view.



(e) Bottom regions of collections – by collection view.

Fig. 5. WiredTiger's write patterns with different views of file types in Linkbench.

Also, as shown in Fig. 3 and Table 1, collection files and secondary index files have heavy random write patterns while primary indexes have scattered sequential write patterns. Therefore, the sequential writes should map to different streams from randomly writes. In summary, an optimal stream mapping scheme should satisfy below requirements:

1. Solve the internal fragmentation of collection files and index files.
2. Map writes on primary indexes to different streams from those of secondary indexes, and collections.
3. Solve the cross-region fragmentation.
4. Work independently of various data models (number of collections, primary indexes, and secondary indexes) and the limitation of the number of streams supported by SSDs.

Boundary-based approach satisfies the first and the last requirement, however, suffers from the requirements (2) and (3). The file-based approach maps each file types to

different streams that solve only the second requirement. There is an extended solution from the file-based method that maps each physical collection file or index file to a stream. This approach solves requirements (2) and (3) but becomes impossible when the number of files is larger than the maximum number of streams that the underlying SSDs support.

Due to those limitations of current approaches, we introduce a novel stream mapping scheme that satisfies all the requirements that discussed in detail in the next section.

4. DYNAMIC STREAM MAPPING

To solve the cross-region fragmentation problem in boundary-based approach, we propose a novel online stream mapping scheme that classifies all regions of collection files into *K* groups based on their hotness values, then assign each group to a distinguish stream. We adopt the similar stream mapping scheme for all regions of index files. Moreover, we separate writes on primary indexes from writes on secondary indexes by using statistical information. Note that it requires the underlying SSDs support at least 2K streams. The hotness value of one region changed after each checkpoint based on many aspects, *e.g.*, current workload, data model, and cache size. Regardless of those aspects, our proposed approach dynamically classifies writes on regions into groups according to their hotness values and maps groups to corresponding streams. Therefore we named our proposed method as *Dynamic Stream Mapping* (*DSM*). Note that, the boundary-based approach is a particular case of DSM where *K* is equal to two.

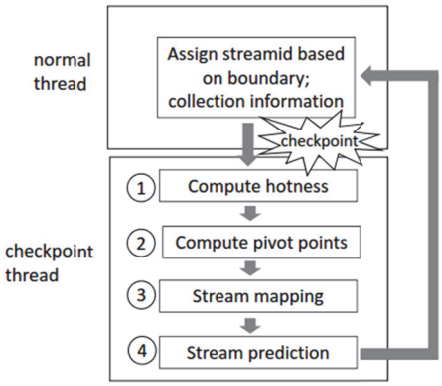


Fig. 6. Overall process of DSM approach.

Fig. 6 illustrates the proposed idea in sequential steps. During normal operations of an eviction thread (*i.e.*, normal thread), before writing data from the buffer pool to the storage system, we assign predicted streams to regions based on the boundary of each file. We then collect statistical information for each collection or index file. When issuing checkpoints, the system blocks normal threads until the checkpoint process finished. At the checkpoint time, our proposed method follows the below steps:

Step 1: Computes hotness values for two regions of each file based on statistical infor-

mation collected previously in the normal thread.

Step 2: For each collection file or index file, computes *pivot points* that aid to grouping regions according to their hotness values computed in step 1.

Step 3: Maps streams to regions using hotness values and pivot points computed in steps 1 and 2.

Step 4: For each region, predicts corresponding stream for the next checkpoint based on current mapped stream in step 3 and predicted-stream in the previous checkpoint.

After remain works of checkpoint process finished, the system resumes the normal threads. We discuss the detail of each step in the below subsections.

4.1 Stream Assignment at the Normal Thread

We assign streams during the normal thread as in boundary-based method. The algorithm in Fig. 4 is rewritten in Fig. 7. The main differences are in lines 7 and 12 where the streams *file.sid*1 and *file.sid*2 assigned to the top region and the bottom region of each file are no longer constants but recomputed after each checkpoint. For the first checkpoint, the assigned streams are set by initial values. From the second checkpoint, assignment for *file.sid*1 and *file.sid*2 are done at the checkpoint time that discussed in subsection 4.4. Statistical information per each region (the number of writes, the lower-bound offset, and the upper-bound offset of written files) are saved as illustrated at lines 7-9 and 13-15 in DSM algorithm in Fig. 7.

```
1: Require:
   + boundary of each collection file and index file has computed
   + Predicted-streams for two regions of each file i.e., file.sid1 and file.sid2 are 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 OR file is index then
6:       if offset < boundary then
7:             sid ← file.sid1
8:             file.numw1 ← file.numw1 + 1
9:             file.off_min1 ← min(file.off_min1, offset)
10:            file.off_max1 ← max(file.off_max1, offset)
11:      else
12:            sid ← file.sid2
13:            file.numw2 ← file.numw2 + 1
14:            file.off_min2 ← min(file.off_min2, offset)
15:            file.off_max2 ← max(file.off_max2, offset)
16: else ►Other files i.e., metadata
17:      sid ← OTHER_SID
```

Fig. 7. Dynamic stream mapping algorithm (at a normal thread).

4.2 Hotness Computing

In the DSM approach, hotness value of a region is defined as the average number of writes occur per 4KB data page on that region. Notice that hotness value of writing data

in one region changed after each checkpoint, so the stream assignment should base on relative values rather than absolute values. To illustrate the idea, we use Fig. 3 (a) which is the IO pattern of a collection file as discussed from the previous experiment. For a given collection file or index file, there is a *boundary* partition the file into two regions (*i.e.*, top and bottom) that located by file offsets named *range*1 and *range*2 respectively. For a given region, the writing density is computed as in Eq. (1). $numw_i$ is the number of writes on that region within a checkpoint window, *PAGESIZE* is set as 4KB.

$$density_i = (numw_i / range_i) \times PAGESIZE \tag{1}$$

$$hotness_i = lg(density_i / t_i - t_1)) \tag{2}$$

where $i = 1, 2$ if writing on top region or bottom region respectively.

Then, we compute the hotness value as in Eq. (2). Note that the checkpoint window is depended on: (1) type of workload, (2) ratio between cache size and database size, and (3) data model. Therefore, to eliminate those constraints, we divide the density by the interval time to get the writing density in a unit of time. Moreover, we adopt logarithmic scaling to solve the common sub-optimize problem that inspired from the proportional selection phase in Genetic Algorithm [38]. Fig. A.2 in the Appendix describes the detail implementation of hotness computing.

4.3 Compute Pivot Points and Stream Mapping

After hotness values of all regions are computed, the next step is to classify all regions of one file type, *e.g.*, collection into K groups. We partition a min-max range into $K - 1$ evenly disjoint sections using $K - 1$ pivot points p_i such that: $min \le p_i \le max$, where $1 \le i \le K - 1$, *min* and *max* are minimum hotness value and maximum hotness value of all regions respectively.

The pivot points are computed using Eq. (3). To increase the flexibility of the approach for various workloads, we use positive-integer weight parameter α by treating the first group and the last group (the "coldest" group and the "hottest" group) separately with the remains groups (the "warm" groups). Note that in a particular case when α equals to K, all groups have equal weight.

$$\begin{cases} p_1 = \min + (\max - \min) \times (1/\alpha) \\ p_{K-1} = \min + (\max - \min) \times ((\alpha - 1)/\alpha) \\ p_j = p_1 + ((p_{K-1} - p_1)/(k-2)) \times (j-1), \quad \text{where } 2 \le j \le K \end{cases} \tag{3}$$

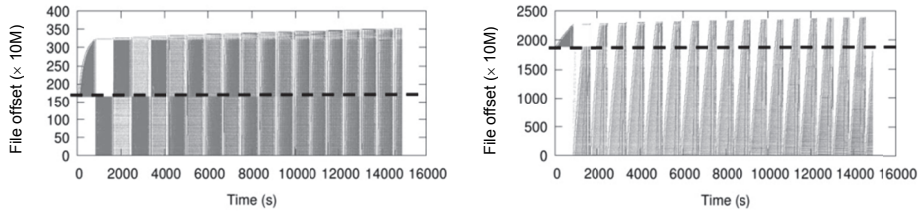
Next, for a given region, we assign it to a corresponding group according to its *hotness* value as described in Eq. (4). Each group, in turn, mapped to a stream id.

$$group = \begin{cases} group_1, & \min \le hotness \le p_1 \\ group_K, & p_{K-1} \le hotness \le \max \\ group_{g+1}, & p_g \le hotness \le p_{g+1}, 1 \le g \le K-2 \end{cases}. \tag{4}$$

The detail implementation for pivot points computing and stream mapping are described in Figs. A.3 and A.4 in the Appendix section respectively.

4.4 Stream Prediction

Before writing to a region, we assign a stream id (sid) using *posix_fadvise*(*fid*, *offset*, *sid*, *advice*) command. In order words, we must predict the hot-cold trends for each region before the writes come. One region is hot in the current checkpoint may remain hot or reversely become cold in the next checkpoint. For example, Fig. 8 plots write patterns of link collection and count collection for four hours with LB-Update-Only workload. The boundaries between two regions are marked as dash lines. In Fig. 8 (a), the hot-cold trend switches between regions in some first checkpoints then keep on the same trend after around two hours of executing time. On the other hand, in Fig. 8 (b), the hotness levels between two regions are almost similar during the execution time except for the first checkpoint.



(a) *link* collection file (b) *count* collection file

Fig. 8. The difference between hot-cold trends.

We propose a low overhead hot-cold data prediction as described in Eq. (5). *cursid*1, *cursid*2 are current streams assigned to corresponding groups in the previous step. sid_i is the stream finally maps to each region that used for the next checkpoint. The detail of stream prediction is described in Fig. A.5 in the Appendix section.

$$sid_i = \begin{cases} cursid1, & sid_i == cursid1 \\ cursid2, & \text{otherwise} \end{cases} \quad (5)$$

where $i = 1, 2$

Finally, we summarize all steps of DSM algorithm discussed so far in Fig. 9. Firstly, we use function *ProcessStat*(*F*) (see Fig. A.1 in the Appendix for detail) to extract supported information from the number of writes on each region and other statistical data (computed in the normal thread as described in the previous section). Then we compute hotness values and pivot points in lines 8, 9, and 10 respectively. The *for* loop from line 11 to line 20 classifies two regions in each collection file or index file into corresponding groups. Due to primary indexes have different write patterns compare with the secondary indexes, we handle them separately by using the *globalpct*1 and *globalpct*2 of each file that are the percentage of writes on the bottom regions and the top regions respectively over total writes on all files during a checkpoint. Function *ProcessStat*(*F*) describes the detail of how to compute *globalpct*1 and *globalpct*2. For a given index file, if there is a

low-frequency write region (percentage of writes less than the THRESHOLD2) or low-frequency write on the whole file (total number of writes on two regions less than the THRESHOLD1) then that index file is considered as a primary index file. The last step predicts stream for each region of the underlying file at line 21.

```
1: Require: number of writes on bottom region numw1 and bottom region numw2 are computed for each file within the checkpoint
2: Input: F – List of collection files and index files, K – number of groups
3: Output: Stream ids of each collection or index file are mapped
4: CP ← ∅                                                    ► Collection pivot points
5: IP ← ∅                                                    ► Index pivot points
6: cursid1 ← cursid2 ← nul                                   ► streams in current checkpoint
7: ProcessStat(F)
8: ComputeHotness(F, coll_min, coll_max, idx_min, idx_max)
9: ComputePivots(CP, K − 1, coll_min, coll_max)
10: ComputePivots(IP, K − 1, idx_min, idx_max)
11: for each file f in F do                                  ► phase1: Compute the total writes in a stream id
12:     if f is collection then
13:         cursid1 ← MapSIDByHotness(hotness1_f, CP, COLL_INIT_SID)
14:         cursid2 ← MapSIDByHotness(hotness2_f, CP, COLL_INIT_SID)
15:     else                                                 ► Index files
16:         if (globalpct1_f + globalpct2_f ≤ THRESHOLD1) OR
               (globalpct1_f ≤ THRESHOLD2) OR
                 (globalpct2_f ≤ THRESHOLD2) then            ► Primary index
17:             cursid2 ← cursid1 ← PRIMARY_IDX_SID
18:         else
19:             cursid1 ← MapSIDByHotness(hotness1_f, IP, IDX_INIT_SID)
20:             cursid2 ← MapSIDByHotness(hotness2_f, IP, IDX_INIT_SID)
21:     PredictStream(f, cursid1, cursid2)
```

Fig. 9. Dynamic stream mapping algorithm (at a checkpoint thread).

Our proposed DSM approach is low overhead and dynamically adapts to any workload, any data model. Firstly, for each collection file or index file, we use a data structure named *mssd-object* to capture the statistical information (*i.e.*, number of writes, minimum and maximum write offset) for each region. Because the data model in NoSQL DBMS is flexibly changed rather than fixed as in RDBMS, we allocate those *mssd-objects* dynamically based on the current number of files (*i.e.*, number of collections or number of indexes). Moreover, during the normal thread, instead of updating statistical information for each data block as in previous studies, we update on region-based *mssd-objects* that have small memory footprint (*i.e.*, lower than 100B). Also, hotness values and stream mapping are based on relative values (*i.e.*, proportions) rather than absolute values. Thus our proposed method works independently of the workloads.

5. EVALUATION AND ANALYSIS

This section describes experiment settings, evaluation results of our proposed method

compared to the original WiredTiger (as the base line) and prior methods. We also analyze the effectiveness of distinguishing writes on primary index files from writes on secondary index files as well as the effectiveness of reducing the maximum leaf page size of collection file as done in the prior research.

5.1 Experimental Settings

To fairly compare our proposed method with file-based approach and boundary-based approach, we adopt the same experimental setup with the previous research [34]. To enable multi-streamed SSD technique, we use both modified Linux kernel 3.13.11 and customized-firmware Samsung 840 Pro SSD as in [4]. For eliminating network latency, we set up both the client and the server in the same commodity computer with 48 cores Intel Xeon 2.2GHz processor, 32GB DRAM. In the client layer, we use YCSB 0.5.0[2] and LinbenchX 0.1[3] (an extended version of Linkbench that support MongoDB) with diversity workloads as shown in Table 1. The number of documents in YCSB is set to 23 million, and *maxid*1 in Linkbench is set to 80 million. All benchmarks are executed during two hours with 40 client threads. In the server layer, we use a stand-alone MongoDB 3.2.1[4] server with DirectIO and various cache sizes from 5GB to 30GB. WiredTiger is used as the storage engine with all default settings.

5.2 Multi-Streamed SSD Optimization Evaluation

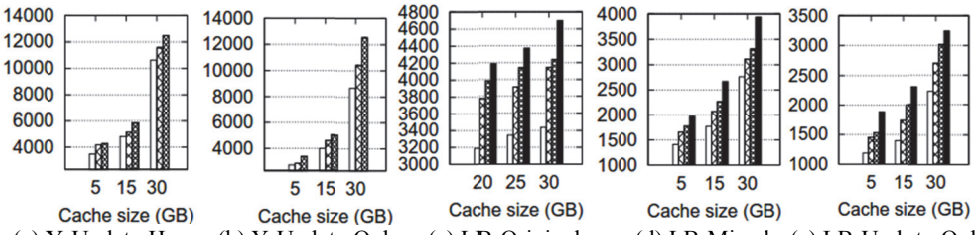
Table 2 summarizes mapping schemes of all methods. There is no stream mapping in the original WiredTiger, so all writes are mapped to the default stream 0 (reserved for files in the Linux kernel). In the file-based approach, each file type is mapped to a distinguish stream. In the case of boundary-based approach, all collection files are mapped to two streams: one for top regions and another for bottom regions. The same mapping scheme is adopted for index files. In the DSM approach, we use there-group mapping DSM (*i.e.*, set K equal to 3) with three streams for collection files and three streams for secondary index files. Writes on primary index files are mapped to a distinguish stream to writes on secondary index files. There are some important notes:

- Except for the DSM method, the remains map writes on primary index files and writes on secondary index files to the same stream.
- Writes on metadata files and writes on journal files are mapped to the same stream for all methods.
- The Samsung 840 Pro SSD support maximum only eight streams from 0 to 7. In the DSM approach, six streams are used for collection files and secondary index files. Therefore, writes on journal files and writes on primary index files share the same stream (*i.e.*, stream1). It is adequate because writes on those files follow lightly sequential patterns, hence can be considered as cold data and can be mapped in the same stream.

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

Table 2. Stream mapping schemes in multi-streamed approaches included the original WiredTiger.

Method	Kernel	Metadata	Journal	Primary Index	Collection	2nd Index
Original	0	0	0	0	0	0
File-based	0	0	1	3	2	3
Boundary-based	0	0	1	4,5	2,3	4,5
DSM	0	0	1	1	2,3,4	5,6,7



(a) Y-Update-Heavy (b) Y-Update-Only (c) LB-Original (d) LB-Mixed (e) LB-Update-Only

Fig. 10. Throughput of optimized methods compare with the original.

Figs. 10 and 11 show the performance results in terms of throughput and the 99th-percentile latency improvement respectively. In LB-Original workload, because the ratio of read operations are high (*i.e.*, 69%) and the data size is approximate 32GB, pages are frequently fetched in (flush out) to (from) the buffer pool. We set the buffer pool sizes large enough (*i.e.*, 20GB, 25GB, and 30GB) to keep secondary indexes in DRAM as long as possible and avoid the performance degradation due to the excessively overhead of reclaiming free space in the buffer pool. Note that the DSM approach is not carried out in YCSB benchmark due to its simple data model such that there are a collection file and a primary index file. DSM in this benchmark is almost similar to the boundary-based method.

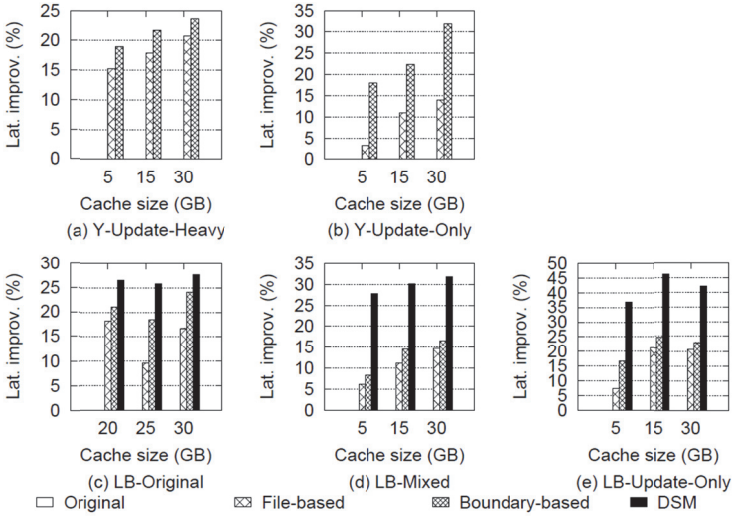


Fig. 11. Latency of optimized methods compare with the original.

Empirical results have shown boundary-based approach lost its effectiveness in solving internal fragmentation problem when the data model becomes complex. In YCSB, the boundary-based has additional 2.8%–13.1% and 9.73%–20.4% of throughput improved rate compared to the file-based method. However, those gaps become smaller in Linkbench that are 2.2%–5.8%, 6.5%–9.4% and 5.6%–15% for LB-Original, LB-Mixed, and LB-Update-Only respectively. There is the same trend with 99th-percentile latency. Compared to the file-based approach, the boundary-based has additional 3.6%–4.6% and 12.7%–20.9% improvement rate for Y-Update-Heavy and Y-Update-Only respectively. With heavily-read workload (*i.e.*, LB-Original), the additional improved rate is similar or even better than the Y-Update-Heavy with 3.5%–9.8%. However, with heavier write workloads (*i.e.*, LB-Mixed and LB-Update-Only), the additional rates are 2.1%–3.9%, and 2.5%–10% respectively, that are significantly lower than the Y-Update-Only. The reason is with the complex data model in Linkbench, cross-region phenomenon is dominant thus the boundary-based approach becomes less effective.

Conversely, the DSM approach effectively solves the internal fragmentation and cross-region fragmentation. Compared to the best-performing method in prior work (*i.e.*, boundary-based), DSM improves the throughput by up to 10.8%, 19.2%, and 23%; enhances the 99th-percentile latency by up to 8.9%, 21.2%, and 28.5% for LB-Original, LB-Mixed, and LB-Update-Only respectively. Overall, DSM is the best method, compared to the original WiredTiger, DSM improves the throughput by up to 36.5%, 50.7%, and 65%; enhances the 99th-percentile latency by up to 27.6%, 31.8%, and 46.2% for LB-Original, LB-Mixed, and LB-Update-Only respectively.

5.3 Effects of Primary Index Filtering

DSM not only solves cross-region fragmentation but also distinguish writes on primary index files from secondary index files. Fig. 12 shows the effects of primary index filtering by using typical DSM and the DSM without primary index filtering, *i.e.*, *DSM-SkipPriIdx*. The y-axis shows the degradation rate of throughput and latency of a method compared with the base line method (*i.e.*, DSM). The boundary-based results are include-

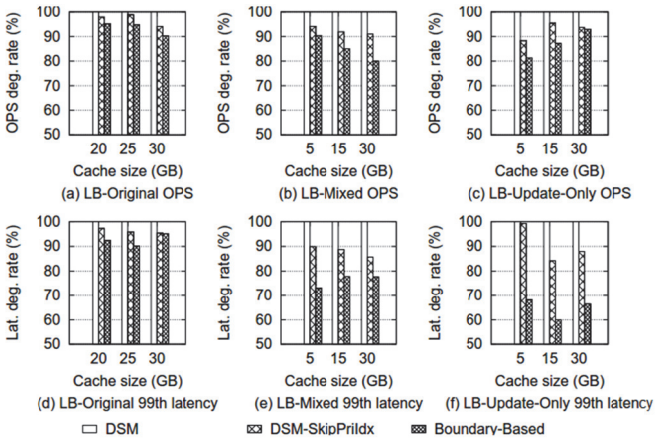


Fig. 12. Effects of primary index filtering in term of throughput and 99th latency.

ed as the reference. Typically, heavier write workloads have more effective in primary index filtering. Without primary indexing filtering, the throughput benefit decreases by up to 5.9%, 9%, 11.7%, and the 99th-percentile latency benefit lost by up to 4.6%, 14.2%, 15.7% for LB-Original, LB-Mixed, and LB-Update-Only respectively. Even though without primary index filtering, DSM still shows better performance than boundary-based both on throughput and 99th-percentile latency.

5.4 Leaf Page Size Optimization Evaluation

In this subsection, we further optimize MongoDB by reducing the maximum leaf page size of collection files (the leaf page sizes of index files are unchanged). To evaluate the impact of leaf page size on the performance of our proposed method, we adopt the same experiment with the prior research. We setup YCSB benchmark and Linkbench benchmark using various workloads and cache sizes for 32KB leaf page size (default) and 4KB leaf page size. For the space limitation in the paper, we only show the throughput results of the most write-intensive workloads, *i.e.*, Y-Update-Only and LB-Update-Only as in Fig. 13. Overall, DSM-4KB still is the best method that improves throughput 1.6×–2.1×, and 1.8%–48.6% compared to the Original-32KB, and DSM-32KB respectively.

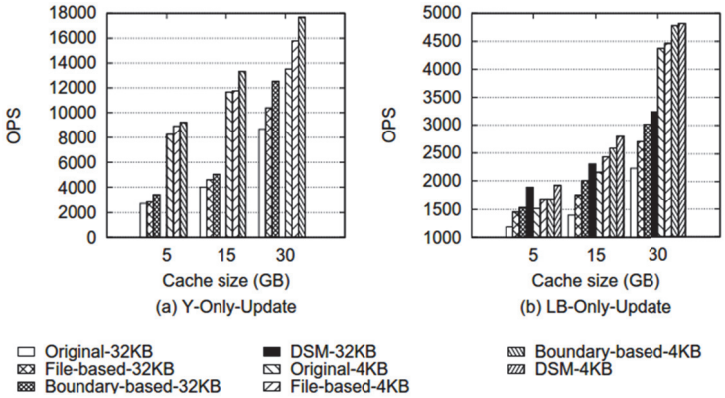


Fig. 13. Leaf page size optimization evaluation of methods with LB-Update-Only.

Small leaf page size optimization is effective in the simple data model in YCSB; however, lost its effectiveness in the complex data model in Linkbench. For instance, in YOnly-Update, the boundary-based method shows significantly improving the throughput performance by up to 2×–3.3×, but only improves 1.4×–2.1× in LB-Only-Update.

5.5 Discussion

The proposed method is based on the statistic information (*i.e.*, write regions, and write frequencies) collected during a checkpoint period. Besides, distinguishing data types (primary index and secondary indexes) from one large data file also aids to improve the performance. Thus, any storage engine which has various data objects contain inside one large data file and those data objects have different access lifetimes can gain

benefits from DSM. For instance, DSM can be adopted in InnoDB storage engine in MySQL that uses one large per-table user tablespace for both primary index and secondary indexes and one large system tablespace file for metadata, double write buffer, and rollback segments.

There is a trade-off between the performance improvement and storage footprint when reducing the leaf page size of collection files. The database system must fetch a whole page (usually some KBs) from the storage device to the buffer pool to read a somebytes record. With the same buffer pool size, the 4KB page system could keep more cache pages than the 32KB page system, thus reducing more IO accesses and achieving better performance. However, with the same number of records (*i.e.*, documents), reducing the leaf page size from 32KB to 4KB lead to increasing number of leaf pages and internal pages in the B+Tree. Consequently, collection files and index files become larger. For instance, reducing the leaf page size of collection files from 32KB to 4KB leads to the database size increases from 51.7GB to 58.4GB (+12.8%) in YCSB, and increases from 55GB to 108.5GB (approximate double) in Linkbench.

6. CONCLUSION

In this paper, we have discussed data fragmentation in MongoDB as well as the proposed methods 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 the simple data model in YCSB, the boundary-based approach is adequate to solve the internal fragmentation that shows good performance improvement. However, it retains cross-region fragmentation with complex data model in Linkbench. To address that challenge, we extended the boundary-based method by introducing DSM, a novel low-overhead stream mapping scheme that dynamically grouping writes on corresponding streams based on hotness values in each checkpoint period. DSM works independently of data models, workloads, and the limitation of the number of streams that the physical SSD supported. The number of groups and other parameters are configurable to gain the best performance. Stream mapping using primary index filtering in DSM has considerable performance improvement. Moreover, simple data model in YCSB gains more benefits from decreasing B+ tree leaf page size than complex data model in Linkbench. In practical applications, the data models are complex with many collection files, index files rather than simply as in YCSB. Our proposed method is adequate for such applications thus it works effectively regardless of data models or workloads.

In the next research, we plan to evaluate the performance of our proposed method with emerging multi-streamed SSD devices (*i.e.*, NVMe SSD). We also optimize the algorithms for distributed environment regard to replica sets and shards.

ACKNOWLEDGMENT

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).

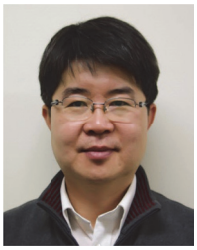
REFERENCES

1. Y. Deng and J. Zhou, "Architectures and optimization methods of flash memory based storage systems," *Journal of Systems Architecture*, Vol. 57, 2011, pp. 214-227.
2. J.-U. Kang, J.-S. Kim, C. Park, H. Park, and J. Lee, "A multi-channel architecture for high-performance NAND flash-based storage system," *Journal of Systems Architecture*, Vol. 53, 2007, pp. 644-658.
3. S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A case for flash memory ssd in enterprise database applications," in *Proceedings of ACM International Conference on Management of Data*, 2008, pp. 1075-1086.
4. F. Yang, K. Dou, S. Chen, M. Hou, J.-U. Kang, and S. Cho, "Optimizing nosql db on flash: A case study of rocksdb," in *Proceedings of IEEE 12th International Conference on Ubiquitous Intelligence and Computing*, 2015, pp. 1062-1069.
5. J. Bhimani, J. Yang, Z. Yang, N. Mi, N. K. Giri, R. Pandurangan, C. Choi, and V. Balakrishnan, "Enhancing ssds with multi-stream: What? why? how?" in *Proceedings of IEEE 36th International Conference on Performance Computing and Communications Conference*, 2017, pp. 1-2.
6. S.-W. Lee and B. Moon, "Design of flash-based dbms: an in-page logging approach," in *Proceedings of ACM International Conference on Management of Data*, 2007, pp. 55-66.
7. J. Kim, D. H. Kang, B. Ha, H. Cho, and Y. I. Eom, "Mast: Multi-level associated sector translation for NAND flash memory-based storage system," *Computer Science and its Applications*, LNEE Vol. 330, 2015, pp. 817-822.
8. T. Jung, Y. Lee, J. Woo, and I. Shin, "Double hot/cold clustering for solid state drives," *Advances in Computer Science and its Applications*, LNEE Vol. 279, 2014, pp. 141-146.
9. D. Park and D. H. Du, "Hot data identification for flash-based storage systems using multiple bloom filters," in *Proceedings of IEEE 27th Symposium on Mass Storage Systems and Technologies*, 2011, pp. 1-11.
10. J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient identification of hot data for flash memory storage systems," *ACM Transactions on Storage*, Vol. 2, 2006, pp. 22-40.
11. C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "Sfs: random write considered harmful in solid state drives," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012, p. 12.
12. S.-H. Park, J.-W. Park, S.-D. Kim, and C. C. Weems, "A pattern adaptive NAND flash memory storage structure," *IEEE Transactions on Computers*, Vol. 61, 2012, pp. 134-138.
13. T. I. Damaiyanti, A. Imawan, F. I. Indikawati, Y.-H. Choi, and J. Kwon, "A similarity query system for road traffic data based on a nosql document store," *Journal of Systems and Software*, Vol. 127, 2017, pp. 28-51.
14. H. Shim, "Phash: A memory-efficient, high-performance key-value store for large-scale data-intensive applications," *Journal of Systems and Software*, Vol. 123, 2017, pp. 33-44.
15. Y.-T. Liao, J. Zhou, C.-H. Lu, S.-C. Chen, C.-H. Hsu, W. Chen, M.-F. Jiang, and Y.-C. Chung, "Data adapter for querying and transformation between sql and nosql

database," *Future Generation Computer Systems*, Vol. 65, 2016, pp. 111-121.
16. D. G. Chandra, "Base analysis of nosql database," *Future Generation Computer Systems*, Vol. 52, 2015, pp. 13-21.
17. R. Dharavath and C. Kumar, "A scalable generic transaction model scenario for distributed nosql databases," *Journal of Systems and Software*, Vol. 101, 2015, pp. 43-58.
18. P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsmtree)," *Acta Informatica*, Vol. 33, 1996, pp. 351-385.
19. B. Jun and D. Shin, "Workload-aware budget compensation scheduling for nvme solid state drives," in *Proceedings of IEEE Non-Volatile Memory System and Applications Symposium*, 2015, pp. 1-6.
20. MongoDB, "Mongodb architecture," https://www.mongodb.com/mongodbarchitecture, 2017.
21. WiredTiger, "Wiredtiger, making big data roar," http://www.wiredtiger.com/, 2017.
22. C. Győrödi, R. Győrödi, G. Pecherle, and A. Olah, "A comparative study: Mongodb¨ vs. mysql," in *Proceedings of the 13th IEEE International Conference on Engineering of Modern Electric Systems*, 2015, pp. 1-6.
23. S. H. Aboutorabi, M. Rezapour, M. Moradi, and N. Ghadiri, "Performance evaluation of sql and mongodb databases for big e-commerce data," in *Proceedings of IEEE International Symposium on Computer Science and Software Engineering*, 2015, pp. 1-7.
24. C. Băzăr, C. S. Iosif, *et al.*, "The transition from RDBMS to NOSQL. a comparative analysis of three popular non-relational solutions: Cassandra, mongodb and couchbase," *Database Systems Journal*, Vol. 5, 2014, pp. 49-59.
25. B. Alexandru, R. Florin, and I. A. Laura, "Mongodb vs. oracle-database comparison," in *Proceedings of the 3rd IEEE International Conference on Emerging Intelligent Data and Web Technologies*, 2012, pp. 330-335.
26. C.-H. Lee and Y.-L. Zheng, "SQL-to-NOSQL schema denormalization and migration: A study on content management systems," in *Proceedings of IEEE International Conference on Systems*, *Man*, *and Cybernetics*, 2015, pp. 2022-2026.
27. G. Zhao, Q. Lin, L. Li, and Z. Li, "Schema conversion model of SQL database to NOSQL," in *Proceedings of the 9th IEEE International Conference on P2P*, *Parallel*, *Grid*, *Cloud and Internet Computing*, 2014, pp. 355-362.
28. A. Kanade, A. Gopal, and S. Kanade, "A study of normalization and embedding in mongodb," in *Proceedings of IEEE International Advance Computing Conference*, 2014, pp. 416-421.
29. G. Zhao, W. Huang, S. Liang, and Y. Tang, "Modeling mongodb with relational model," in *Proceedings of the 4th IEEE International Conference on Emerging Intelligent Data and Web Technologies*, 2013, pp. 115-121.
30. X. Wang, H. Chen, and Z. Wang, "Research on improvement of dynamic load balancing in mongodb," in *Proceedings of the 11th IEEE International Conference on Dependable*, *Autonomic and Secure Computing*, 2013, pp. 124-130.
31. Y. Liu, Y. Wang, and Y. Jin, "Research on the improvement of mongodb autosharding in cloud environment," in *Proceedings of the 7th IEEE International Conference on Computer Science and Education*, 2012, pp. 851-854.
32. P. Murugesan and I. Ray, "Audit log management in mongodb," in *Proceedings of IEEE World Congress on Services*, 2014, pp. 53-57.

33. T.-D. Nguyen and S.-W. Lee, "I/o characteristics of mongodb and trim-based optimization in flash ssds," in *Proceedings of the 6th ACM International Conference on Emerging Databases: Technologies, Applications, and Theory*, 2016, pp. 139-144.
34. T.-D. Nguyen and S.-W. Lee, "Optimizing mongodb using multi-streamed ssd," in *Proceedings of the 7th International Conference on Emerging Databases: Technologies, Applications, and Theory*, 2017, pp. 1-13.
35. T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "Linkbench: a database benchmark based on the facebook social graph," in *Proceedings of ACM International Conference on Management of Data*, 2013, pp. 1185-1196.
36. Samsung, "Multi-stream technology," http://www.samsung.com/semiconductor/insights/article/25465/multistream, 2016.
37. B. F. 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*, 2010, pp. 143-154.
38. D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," *Foundations of Genetic Algorithms*, Vol. 1, 1991, pp. 69-93.

Trong-Dat Nguyen received the M.S. degree from the School of Computer Science and Engineering, Kyungpook National University, Korea, in 2014. He is currently working toward the Ph.D. degree at Sungkyunkwan University, Suwon, Korea. His research interests include NoSQL DBMSs, and flashbased database technology.



Sang-Won Lee received the Ph.D. degree from the Computer Science Department, Seoul National University, Korea, in 1999. He is a Professor with the College of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea. He was a Research Professor at Ewha Womans University and a technical staff at Oracle, Korea. His research interest includes flash-based database technology.

APPENDIX

Fig. A.1 describes the algorithm of processing the statistical information. The total number of writes for each region is computed as in the first *for* loop in line 4 – 10. For a given file, $numw1_f$ and $numw2_f$ are numbers of writes in *region*1 and *region*2 of that file respectively. In the second *for* loop, the global percentage of each region ($globalpct1_f$ or $globalpct2_f$) is computed which is used for distinguishing primary index from secondary index.

Fig. A.2 is the detailed implementation of computing density value and hotness value as described in Eqs. (1) and (2) respectively.

```
1: function PROCESSSTAT(F)   ► Processing statistical data for each file i.e., hotness
   value, global percentage of write
2:  coll_count2 ← coll_count1 ← 0
3:  idx_count2 ← idx_count1 ← 0
4: for each file f in F do                                  ► phase1: Compute the total writes
5:      if f is collection then
6:          coll_count1 ← coll_count1 + numw1_f
7:          coll_count2 ← coll_count2 + numw2_f
8:      else
9:          idx_count1 ← idx_count1+numw1_f
10:         idx_count2 ← idx_count2+numw2_f
11: for each file f in F do                                 ► phase2: Compute the hotness
12:     if f is collection then
13:         globalpct1_f ← numw1_f/coll_count1 ∗ 100
14:         globalpct2_f ← numw2_f/coll_count2 ∗ 100
15:     else
16:         globalpct1_f ← numw1_f/idx_count1 ∗ 100
17:         globalpct2_f ← numw2_f/idx_count2 ∗ 100
```

Fig. A.1. Algorithm of processing statistical information.

```
1: function COMPUTEHOTNESS (F, coll_min, coll_max, idx_min, idx_max)
   ►Processing statistical data for each file i.e., hotness value, global percentage of write
2: for each file f in F do                                  ► phase2: Compute the hotness
3:     //Number of 4KB page writes on each range in a unit of time
4:     density1 ← (numw1_f ∗ 4096)/range1_f/(t2−t1)
5:     density2 ← (numw2_f ∗ 4096)/range2_f/(t2−t1)
6:     hotness1_f ← lg(density1/t)
7:     hotness2_f ← lg(density2/t)
8:     if f is collection then
9:         coll_min ← min(coll_min, hotness1_f, hotness2_f)
10:        coll_max ← max(coll_max, hotness1_f, hotness2_f)
11:    else
12:        idx_min ← min(idx_min, hotness1_f, hotness2_f)
13:        idx_max ← max(idx_max, hotness1_f, hotness2_f)
```

Fig. A.2. Algorithm of computing hotness.

Fig. A.3 is the detailed implementation of Eq. (3). Remind that alpha is the positive integer. The first and the last pivot point are computed as in line 2 and line 3 respectively. Then other pivot points are computed in the *for* loop in lines 5-6.

We describe the detailed implementation of Eq. (4) in Fig. A.4. For a given region, we find the pair of pivot points such that $pivots_j \leq hotness \leq pivots_{j+1}$, then assign the corresponding stream *sid* to that region in line 3, 5, and 9.

Fig. A.5 describes the detailed implementation of Eq. (5). For a given file, if the current computed stream *e.g.*, *cursid*1 is same with the predicted stream in the previous checkpoint ($sid1_{file}$), it means the hot-cold trends is unchanged then the predicted stream for next checkpoint is kept same as before. Otherwise, we map that region to the other stream (*cursid*2).

```
1: function COMPUTEPIVOTS (pivots, n, min, max)
2: pivots[0]   ←   min + (max−min)/α
3: pivots[n−1] ←   min+(max−min)∗(α−1)/α
4: step ← (pivots[n−1]− pivots[0])/(n−1)
5: for i ← 1 to (n−2) do
6:       pivots[i] ← pivots[0]+ step ∗ i
```

Fig. A.3. Algorithm of computing pivot points.

```
1: function MAPSIDBYHOTNESS(hotness, pivots, n, initsid)
2: if hotness ≤ pivots[0] then                          ► the most left
3:       sid ← initsid
4: else if pivots[n−1] ≤ hotness then                   ► the most right
5:       sid ← initsid + n
6: else
7:       Find the pivot point j in array pivots such that:
8:       pivots[j] ≤ hotness ≤ pivots[j + 1]
9:       sid ← initsid + (j + 1)
10: return sid
```

Fig. A.4. Algorithm of mapping SID by hotness.

```
1: function PREDICTSTREAM(file, cursid1, cursid2)
2: if sid1_file == cursid1 then                         ► the hot-cold trend is same, do not swap
3:       sid1_file ← cursid1
4: else
5:       sid1_file ← cursid2
6: if sid2_file == cursid2 then                         ► the hot-cold trend is same, do not swap
7:       sid2_file ← cursid2
8: else
9:       sid2_file ← cursid1
```

Fig. A.5. Algorithm of stream prediction.