IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 5, 2021, accepted March 23, 2021, date of publication April 12, 2021, date of current version April 19, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3072378

Boosting Compaction in B-Tree Based Key-Value Store by Exploiting Parallel Reads in Flash SSDs

JONGBAEG LEE[1], GIHWAN OH[1], AND SANG-WON LEE[2]
[1]Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, South Korea
[2]College of Computing, Sungkyunkwan University, Suwon 16419, South Korea

Corresponding author: Sang-Won Lee (swlee@skku.edu)

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(Ministry of Science and ICT) (No.2015-0-00314, NVRam Based High Performance Open Source DBMS Development)

ABSTRACT Append-only B-tree based key-value stores provide superior search and update performance based on their structural characteristics; however, they periodically require the compaction task that incurs significant I/O overhead. In this paper, we present that the compaction's degraded read performance deteriorates the overall performance in ForestDB, a representative append-only B-tree engine. We demonstrate that despite the exceptional performance of the SSD, the cause of the slow read performance is the underutilization of the SSD's internal parallelism due to the read operations using synchronous I/O. Furthermore, this paper proposes a novel compaction method that improves the compaction's read performance by exploiting SSD's internal parallelism by requesting multiple read operations in a batch using the asynchronous I/O technique. We implemented our proposed methods on ForestDB using two Linux asynchronous I/O interfaces, AIO and io_uring. The evaluation results confirm that our method drastically improves the compaction's read performance up to ten times compared to the conventional compaction method. In particular, we confirmed that the proposed method using io_uring, the latest asynchronous I/O interface, is effective regardless of the file I/O mode and outperforms the others in all cases.

INDEX TERMS Asynchronous I/O, compaction, flash memory SSD, ForestDB, io_uring, libaio.

I. INTRODUCTION

With the active use of mobile devices and the recent spread of Internet services such as social media, search engines, and e-commerce, there has been increasing interest in deploying storage techniques that effectively store and retrieve massive amounts of unstructured data. Traditional relational database management systems have difficulties processing these types of data since they use a strict data model and are designed to run on a single machine. On the other hand, NoSQL databases, especially key-value store techniques, use flexible data structures without schemas and are designed to run on a cluster consists of plural machines [1], [2]. Thus, they effectively handle large amounts of unstructured data and are widely used in data-intensive applications [3], [4].

Since key-value stores directly interact with storage devices, they are designed to access storage sequentially to optimize the data transfer rate. Some of them append the database mutations at the end of the database file to generate sequential writes. For example, key-value stores, such as LevelDB [5] and RocksDB [6], apply a log-structured merge tree (LSM tree), an append-only data structure, to increase the write performance while sacrificing the read performance. On the other hand, key-value stores such as ForestDB [7] and CouchDB [8] achieve both fast read performance and fast write performance using the B-tree index structure in an append-only manner.

An append-only key-value store periodically carries out a process called ''compaction'' to remove accumulated obsolete data. Because the compaction process copies all of the valid data from an old database file to a new database file, it generates many I/O operations, and its overhead is significant. ForestDB [7], a representative append-only B-tree based key-value store, alternately access data that are not physically adjacent during compaction, resulting in the slow read performance of compaction. Our evaluation results in section III also verify that tree search and read operations are the main culprits of degrading the compaction's overall performance.

The associate editor coordinating the review of this manuscript and approving it for publication was Cristian Zambelli.

Meanwhile, in the storage landscape, flash memory-based solid-state drives (SSDs) are replacing traditional hard disk drives (HDDs) as main storage. Based on their high bandwidth and extremely fast random access speed, applying SSDs to the append-only B-tree based key-value stores is expected to accelerate the tree search and read performance of compaction. However, merely replacing the storage devices with SSDs improves the tree search performance, but not the compaction's read performance. This is because the compaction task reads data through the synchronous I/O method, so it cannot take advantage of the internal parallelism inherent in SSDs and underutilizes the SSD's bandwidth.

This paper proposes a new compaction method, parallel fetch (p-fetch), which improves the read performance of the ForestDB's compaction. Our method requests multiple I/O operations in a batch and fetches the results in parallel through asynchronous I/O, thereby utilizing the SSD's internal parallelism and dramatically improving compaction's read performance. The technical contributions of this paper are as follows.

- We found that the overhead of tree search and read operation during the ForestDB's compaction is considerable (Section III). Even if an SSD is used as a storage device, it failed to improve the performance of the operations because the SSD's internal parallelism was not fully utilized.
- We designed a novel compaction method, p-fetch, for append-only B-tree based key-value store utilizing the SSD's internal parallelism through asynchronous I/O. We implemented two versions of the p-fetch in ForestDB using Linux asynchronous I/O interfaces. The evaluation results show that our implementations remarkably improves the compaction's read performance by up to 10 times compared to the conventional compaction. In particular, p-fetch with io_uring, a new asynchronous I/O interface in Linux, effectively works in all use cases regardless of the file I/O mode.

II. BACKGROUND

A. ForestDB

ForestDB [7] is a document-oriented back-end key-value store engine of Couchbase server [9] that provides efficient indexing for variable-length key-value pairs through an append-only B-tree based index structure. Specifically, ForestDB applied an on-disk hierarchical B+-tree based trie (HB+-trie) in which each node of the trie is a B+-tree. In HB+-trie, each B+-tree's leaf node points to the disk location of another B+-tree's root node or actual data. Through this index structure, ForestDB minimizes the number of disk blocks accessed for each key-value operation and quickly retrieves desired data from persistent storage.

Although B-tree-based index structures provide superior search performance, they have a limitation in excessively generating random writes that run slowly not only on HDDs but also on SSDs when using the basic in-place update method. Previous studies have proposed methods that convert random writes of B-tree index to sequential writes to make full use of storage bandwidth [10]–[12].

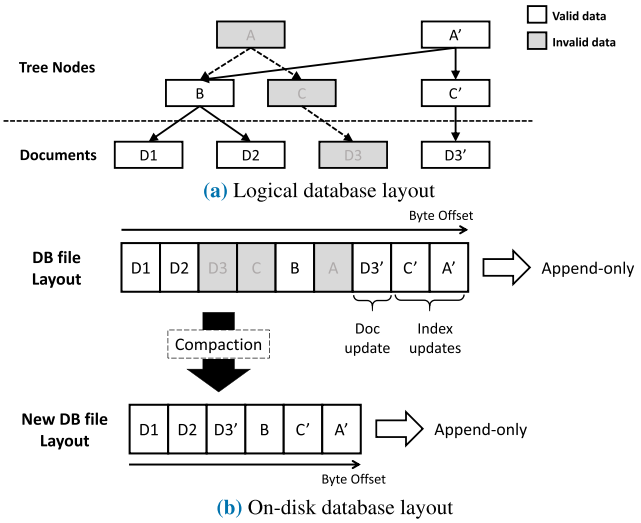


FIGURE 1. Example of ForestDB update.

ForestDB also applied two optimization techniques to achieve high disk write throughput. First, ForestDB stores all mutations in the database in an append-only manner to perform sequential disk write. That is, ForestDB appends a new key-value pair at the end of the database file for each update and insert rather than overwriting the original key-value pair in place. ForestDB does not remove the original key-value pair from the disk for each delete but marks it as deleted. Second, ForestDB exploits an in-memory index called write buffer index (WB index). When a leaf node is changed in an append-only B-tree, all tree nodes from that node to the root must be updated in a cascaded way. In ForestDB, updates of HB+-trie's internal nodes are not written immediately. Instead, the WB index points to the disk locations of data stored in the database file but not yet applied to HB+-trie. When the number of WB index entries exceeds a certain threshold, The WB index entries are flushed and reflected in HB+-trie, reducing the amount of disk I/O per update.

Figure 1 shows an example of changes in the B+-tree structure and database file when an update occurs in ForestDB. In ForestDB, when a document D3 is updated, it marks that D3 is invalid instead of updating it in place and appends a modified new document D3' to the end of the database file. Since the disk location of the data is changed, the tree should update all tree nodes on a path from the leaf to the root in a cascaded way. As a result, it marks the tree nodes C and A as invalid and appends new modified nodes C' and A' to the end of the database file.

Due to the append-only strategy's characteristic, ForestDB accumulates the invalid data with more updates, and the space occupied by the database file monotonically increases with more updates. To prevent invalid data from running out of disk space, ForestDB periodically triggers compaction task

to remove invalid data and reclaim the disk space, as shown in Figure 1b. The compaction task searches valid key-value pairs by iterating the tree, copies them to a new database file, constructs a new tree, and replaces the old file with the new one. After compaction, the documents are stored in a new database file in the order of corresponding keys. Because compaction requires significant I/O overhead, append-only B-tree based key-value stores need to optimize compaction task for better performance.

B. SSD INTERNAL PARALLELISM

One single NAND flash memory package can provide limited data transfer bandwidth (e.g., 40MB/s), and write operations in flash memory are much slower than read operations. Also, unlike HDD, flash memory does not allow in-place updates, and it must precede writing new content with a high-cost erase operation. SSDs inherent multiple levels of internal parallelism can process concurrent I/O operations simultaneously to address these hardware constraints [13], [14].

Most SSDs keep data in an array of NAND flash memory packages for higher bandwidth. SSD issues I/O commands to NAND flash memory through an internal component called a flash memory controller. In general, the flash memory controller and flash memory packages are connected through multiple channels, and multiple flash memory packages share a single channel for data transmission. Since each channel operates independently, the flash memory controller can transfer data to/from multiple flash memory packages connected to different channels in parallel, thereby providing high bandwidth in aggregate. This parallelism is called channel-level parallelism, and applications generating multiple outstanding I/Os can utilize it.

Also, multiple flash memory packages sharing a channel can operate independently. The flash memory packages on the same channel can be used in parallel through interleaving. This parallelism is called package-level parallelism. In addition, multiple chips and planes within a single flash package can execute operations simultaneously, and this parallelism is called chip-level parallelism and plane-level parallelism.

Among the four levels of parallelism, utilizing channel-level parallelism impacts most significantly on performance. The reason is that channel-level parallelism provides a higher bandwidth as the number of overlapped channels increases. On the other hand, when commands are executed simultaneously in multiple packages, chips, or planes, data transmission for each command shares one channel. Therefore, the performance achieved from package-level, chip-level, and plane-level parallelism is limited by the transmission bandwidth that one channel can provide [15].

Besides, current data transmission protocols between the host and storage devices provide command queueing technologies. The command queueing technologies queue multiple I/O commands and enable SSDs to process the commands in parallel internally. For example, the Advanced Host Controller Interface (AHCI) [16] protocol for SATA-based devices supports one command queue with a depth of 32 commands. The Non-Volatile Memory Express (NVMe) [17] protocol for high-performance storage media supports up to 65,535 command queues with as many as 65,536 commands per queue.

C. SYNCHRONOUS AND ASYNCHRONOUS I/O

Most applications store and retrieve data through the operating system's file-based I/O. The operating system provides two different I/O method types depending on whether the I/O operation blocks the process or thread that requested the I/O: synchronous I/O (sync I/O) and asynchronous I/O (async I/O).

In sync I/O, a process or thread starts an I/O operation and then wait for it to complete. Since sync I/O uses intuitive function calls such as *read()* and *write()* in Linux, programming and maintenance complexity is low. However, this approach blocks the program progress while the I/O operation is in progress and makes the processor idle. Thus, when a program requires many I/O operations, the overall performance can be degraded.

Alternatively, in async I/O, a process or thread sends one or more I/O requests to the kernel in parallel, then it continues processing the next commands alongside the I/O operations without blocking. Because async I/O requires additional routines for requesting I/O operations to the kernel and handling completed I/Os, it has higher programming and maintenance complexity than sync I/O. However, applications using async I/O can efficiently utilize the system resources by overlapping I/O operations and CPU works. Moreover, according to the design principles for utilizing SSD's parallelism suggested by Roh *et al.* (2011) [18], async I/O enables channel-level parallelism of SSD by generating multiple outstanding I/Os and provides an opportunity to apply optimizations in file systems and storage devices such as grouping and reordering.

Linux has included a Linux-native asynchronous I/O (AIO) facility in the 2.6 Linux kernel. Specifically, applications use the Linux AIO model as follows:

1) Create an I/O context object for submitting/reaping I/O requests to/from the kernel and initialize it.
2) Create one or more I/O request objects, and fill them with information representing desired operations.
3) Submit I/O request objects to the I/O context object.
4) Reap the completed I/O results from the I/O context object.
5) Return to step 2 as needed.

However, because of its limitations, AIO is not frequently used. First, AIO supports async I/O only if the file is opened in direct I/O mode; otherwise, it behaves like sync I/O. When using direct I/O, all file reads and writes bypass the operating system cache, and I/O size and alignment restraints occur. Second, handling I/O submission and completion via AIO generates additional memory copies between user space and kernel space. Depending on the I/O size, it can be noticeable.

From kernel version 5.1, Linux introduced a new asynchronous I/O interface, io_uring [19]. The basic operation

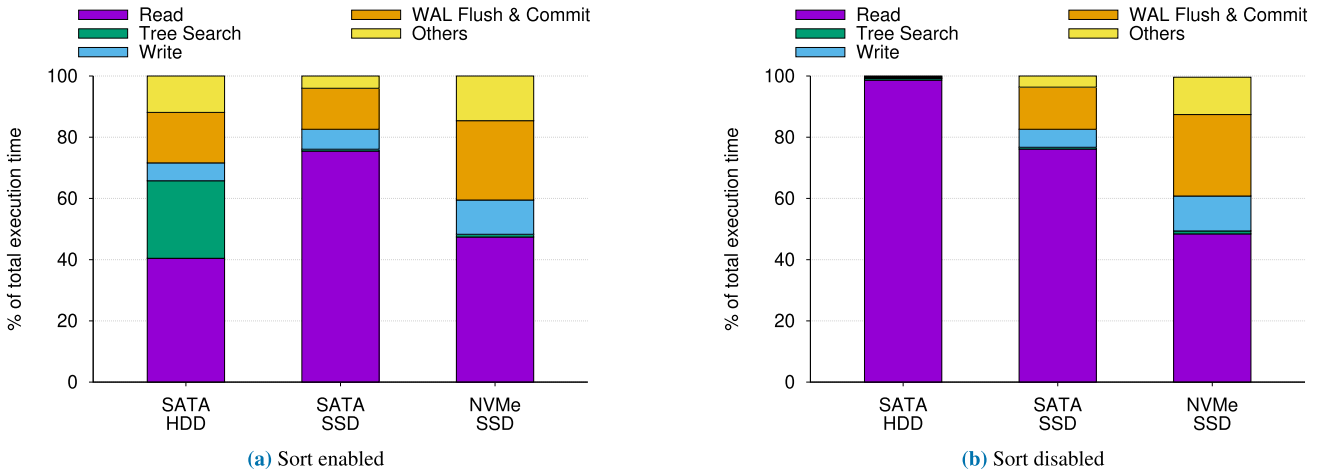


FIGURE 2. The percentage of execution time for each operation composing the compaction to the total execution time.

theory of io_uring is similar to the AIO, but some crucial differences eliminate AIO's limitations. Unlike AIO, io_uring supports async I/O with any I/O mode such as direct I/O, buffered I/O that passes through operating system cache, and socket I/O. Besides, since io_uring uses the memory space shared between user and kernel for handling I/O submission and completion, it does not require an additional memory copy. This approach needs to manage data access synchronization between the application and the kernel. io_uring enables efficient synchronization through a single producer and a single consumer ring buffer consisting of a submission queue (SQ) and a completion queue (CQ). The application submits I/O requests by pushing them to SQ, and the kernel executes the requests. Conversely, the kernel pushes completed I/Os to CQ, and the application reaps them.

III. LIMITATION OF APPEND-ONLY B-TREE BASED KEY-VALUE STORE COMPACTION

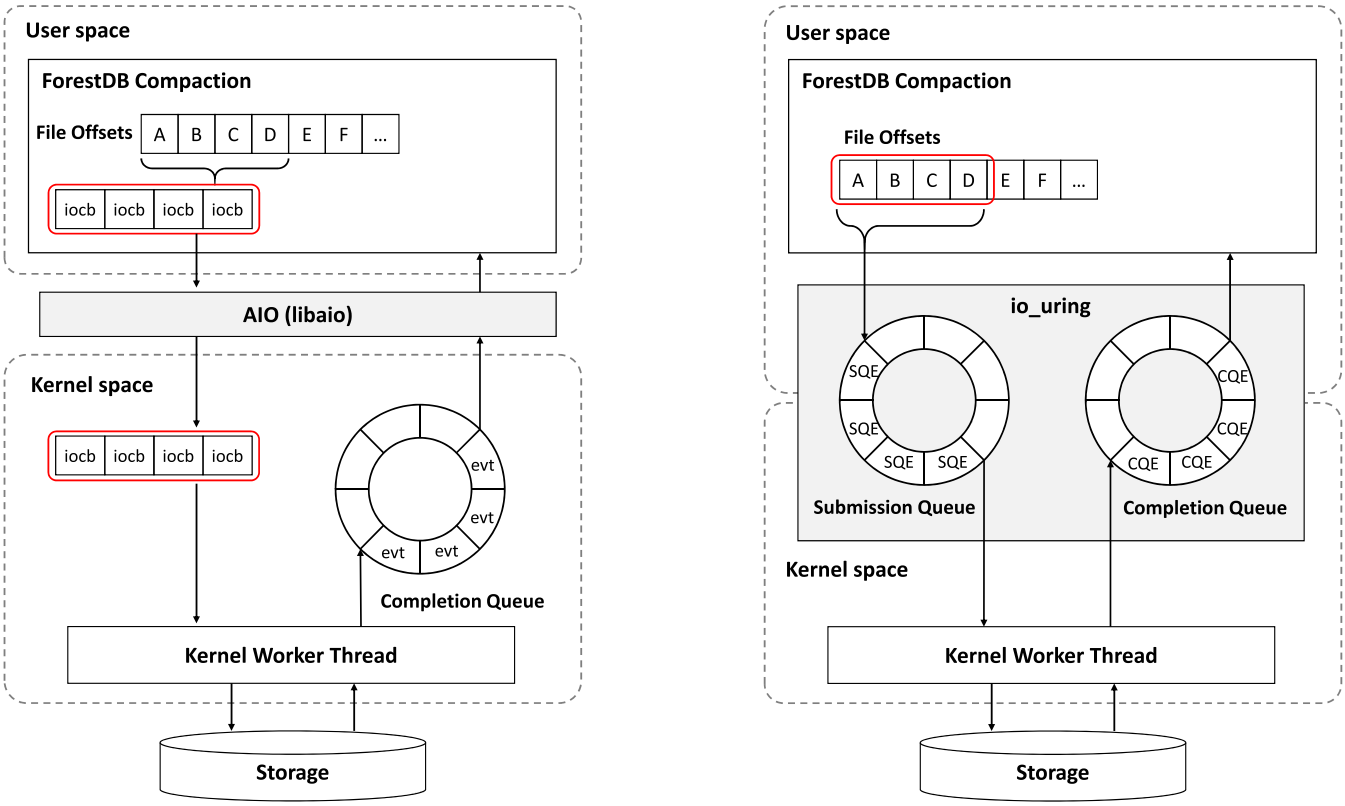
In this section, we analyze the factors affecting the ForestDB's compaction performance. As described in section II-A, the compaction task generates many I/O operations; therefore, the overhead on the overall system performance is considerable. In particular, Ahn *et al.* (2015) [7] pointed out that reading a wide range of data from the uncompacted database file, such as the read operation during compaction task, shows slow performance since invalid documents in the data pages generates more I/Os.

To analyze the performance overhead caused by tree search and read operations during compaction, we measured the performance of the operations comprising the compaction task and the proportions of each operation's performance to the overall compaction. We conducted all experiments in the environment described in section V-A and used a database consisting of 1.5 million key-value pairs with 4KB size.

ForestDB's compaction task works as follows. First, the compaction task creates a new database file and a new HB+-trie structure for storing valid data from the old database file. Next, it iterates over the old HB+-trie nodes and retrieves the disk location where valid data is stored as the offset values of the old database file (Tree search). After tree search, it sorts the offset values to read data from disk sequentially (Sort). Then, it reads the data located at the offset in the sorted order from the old database file (Read) and writes the data to the new HB+-trie and the new database file (Write). During the compaction task, flushing the write-ahead log data and committing the transaction can occur (WAL flush & Commit). Finally, it removes the old database file after writing all valid data to the new database file.

Figure 2 shows the proportions of each operation's execution time that composes ForestDB compaction to the total execution time. Figure 2a shows the result of typical compaction for each storage device, and Figure 2b shows the result of compaction configured not to sort the offsets. Through the results obtained, we found that the overhead from read operation is the largest among the operations comprising the compaction regardless of the storage type. In particular, we found two crucial points. First, the overhead of operations randomly accessing the HDD is large. In the case of HDD, when the file offsets are not sorted, the read operation's performance deteriorates by two orders of magnitude, and even when the file offsets are sorted, tree search that accesses disk randomly takes up 20% of the total execution time. In the case of SSDs, thanks to their fast random access performance, the tree search overhead is reduced to less than 1%, and the effect of sorting file offsets is negligible.

We also found that even though SSD's maximum throughput is much higher than that of HDD, using SSD as a storage device does not sufficiently improve the read operation's performance. Even in compaction using SATA SSD, reading the same amount of data spent more time than that of HDD. In both SATA SSD and NVMe SSD, the read operations occupy 45% of the total execution time. Compared to the performance improvement of tree search, the performance



(a) Overview of the compaction using AIO

(b) Overview of the compaction using io_uring

FIGURE 3. Overview of the compaction methods using asynchronous I/O interfaces.

improvement of the read operation is insignificant. Hence the overhead became prominent.

Despite using SSD as the storage device, the read operations' performance improvement is insignificant because the compaction task used sync I/O to read the database file. Therefore, it cannot efficiently utilize system resources. Therefore, the compaction task cannot generate enough outstanding I/Os to utilize the SSD's channel-level parallelism, resulting in performance degradation.

IV. PROPOSED METHOD

A. DESIGN

To overcome the ForestDB compaction's slow read performance, we propose a new compaction method, parallel fetch (p-fetch), that submits multiple read requests in a single call and fetches the I/O results in parallel through async I/O, thereby improving the read performance of the compaction by exploiting the SSD's internal parallelism. The overall process of the p-fetch is similar to conventional compaction. However, the following four steps are additionally performed for async I/O: initialization for async I/O, preparing the read requests, submitting read requests for multiple pages, and handling the completed read operations.

In the initialization step, p-fetch initializes objects used for async I/O. This step is processed before the tree search operation. In this process, p-fetch initializes the objects related to async I/O, such as the I/O request objects, objects representing completed I/Os, the number of requests submitted in batch, and objects for I/O submission and completion. It also initializes the objects related to file I/O, such as buffer space to store read data and the file descriptor.

Next, p-fetch prepares the read requests while the conventional compaction task requests one read operation for each offset using sync I/O. This step follows searching the tree and sorting the file offsets. In this step, it fills the I/O request objects with the information representing desired operations, including the file offset, the type of operation, and the memory address to store the operation result.

After preparing the I/O request objects as many as the submission batch size, which means the number of requests to submit in batch, p-fetch submits the read request objects to the kernel. Lastly, p-fetch handles the completed read requests after submitting the requests. When the read operations are completed, the kernel pushes I/O results representing the completed read operations to the completion queue. When one or more completed I/O results are pushed to the queue, p-fetch reaps the queue results, extracts appropriate information from the results, and continues the compaction.

B. IMPLEMENTATION

Based on the above design, we implemented two versions of the p-fetch on ForestDB using Linux's async I/O interfaces, AIO and io_uring. Figure 3 illustrates the overviews of the p-fetch implementations. First, to implement p-fetch using

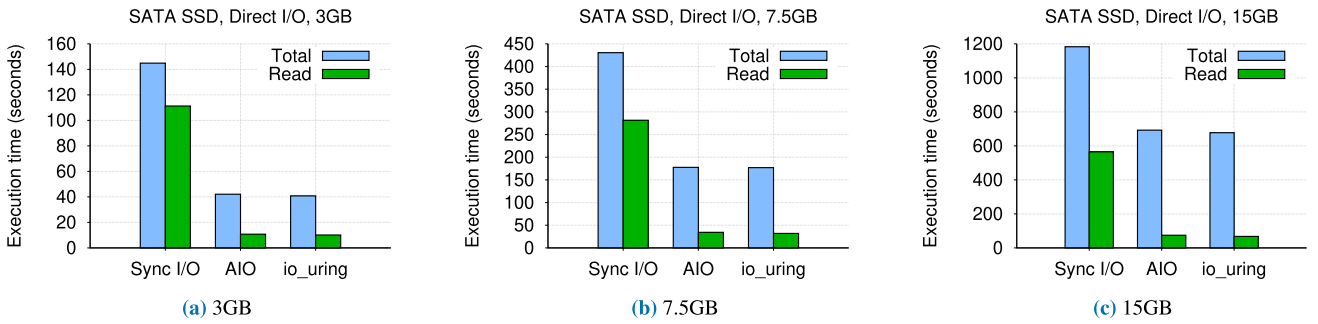


FIGURE 4. ForestDB-Benchmark compaction time (SATA SSD, Direct I/O, Submission batch size 128).

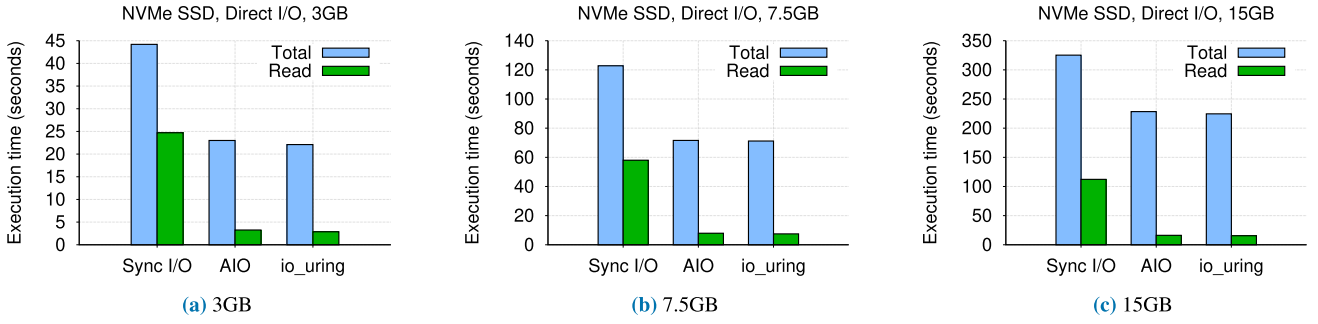


FIGURE 5. ForestDB-Benchmark compaction time (NVMe SSD, Direct I/O, Submission batch size 128).

AIO, we used the libaio library that supports the AIO interface. Figure 3a shows the overview of p-fetch using AIO, and the details of implementation are as follows. In the initialization step, the p-fetch using AIO initializes the I/O context object that supports AIO's I/O submission/completion by calling the *io_queue_init()* function. It also initializes the I/O control block (iocb) array representing the I/O requests and the io_event array representing the completed I/O operations. In the preparation step, the method calls the *io_prep_pread()* function to fill an iocb object, the I/O request object used in AIO, with information representing desired I/O operation such as operation type, file offset, and buffer space to store data. It repeats the preparation until as many iocb objects as the submission batch size are prepared. After preparing the I/O requests, the method submits the I/O requests to the kernel through the I/O context object by calling the *io_submit()* function. Finally, it reaps the completed I/O requests in the io_event objects by calling the *io_getevents()* function.

Next, to implement p-fetch using io_uring, we used the liburing library supporting the io_uring interface. Figure 3b shows the overview of p-fetch using io_uring, and the details of implementation are as follows. The io_uring interface uses one SQ and one CQ for I/O submission and completion. Thus, in the initialization step, the proposed method using io_uring initializes async I/O objects, including SQ, CQ, and each queue's entries by calling *io_uring_queue_init()*. In the preparation step, the method gets one submission queue entry (SQE) from SQ and fills it with information representing the desired I/O operation (operation type, file offset, and buffer space to store data) by calling the *io_uring_prep_read()* function. After preparing a sufficient number of SQEs in the SQ, the method requests I/Os to the kernel by submitting the SQEs through the *io_uring_submit()* function call. The kernel pushes the completed I/O operations to the CQ as a completion queue entry (CQE); therefore, the method reaps the completed I/O requests from the CQEs through the *io_uring_peek|wait()_cqe()* function call. Unlike the AIO method, this method does not require memory copies between user space and kernel space since SQ and CQ are shared between them. Besides, it supports asynchronous reads in both direct I/O and buffered I/O.

V. EVALUATION

This section describes the experimental setup for evaluating the performance of p-fetch and shows the experimental results and analysis.

A. EXPERIMENTAL SETUP

We conducted experiments on a Linux platform with the 5.3 kernel running on a machine equipped with two 24-core CPUs (Intel Xeon CPU E5-2670 2.30GHz) and 64GB DRAM. We used Samsung SSD 850 Pro 256GB with SATA 3.0 interface (SATA SSD) and Intel Optane SSD 900P Series 480GB with PCIe interface (NVMe SSD) as storage devices. Each storage device is mounted with the Linux ext4 file system, with default options. We set the ForestDB data block size to 4KB and the buffer cache size to 5GB.

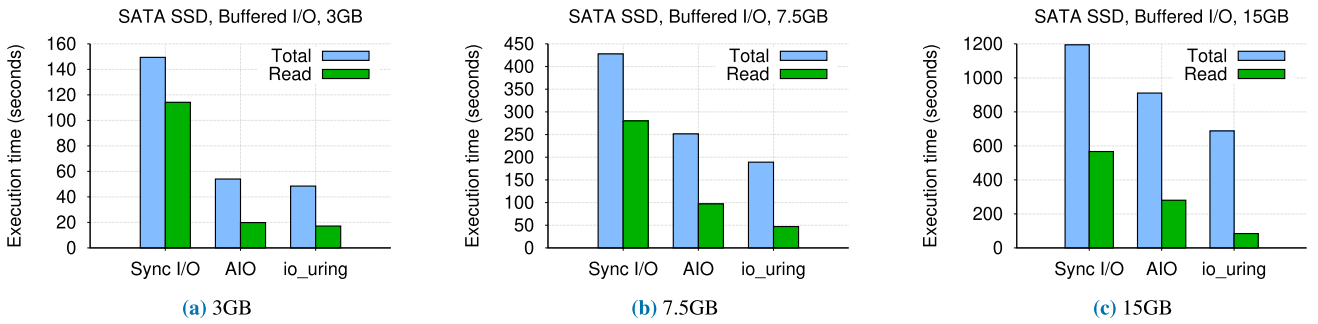


FIGURE 6. ForestDB-Benchmark compaction time (SATA SSD, Buffered I/O, Submission batch size 128).

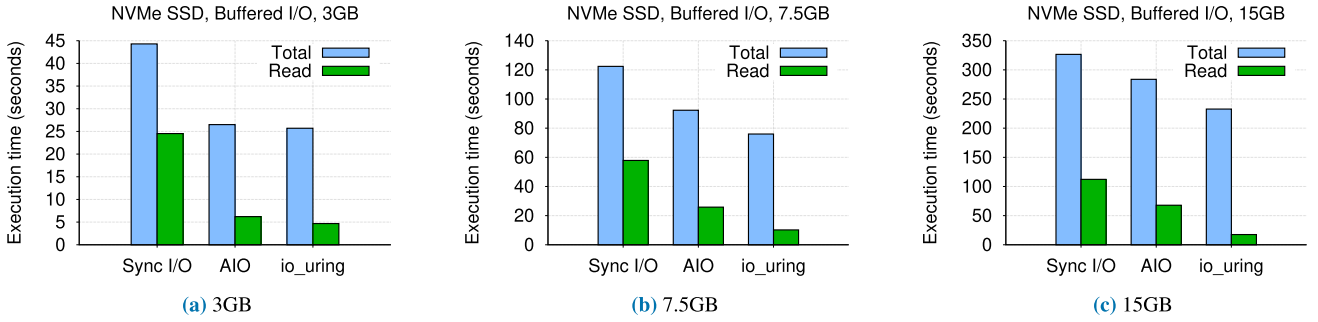


FIGURE 7. ForestDB-Benchmark compaction time (NVMe SSD, Buffered I/O, Submission batch size 128).

We cleaned the operating system's page cache and ForestDB buffer cache before all experiments.

We ran the ForestDB-Benchmark [20], a benchmark program for key-value storage engines, to evaluate the performance of p-fetch using AIO and io_uring compared to the conventional compaction using sync I/O. We carried out all experiments with databases composed of 4KB key-value pairs. We modified the benchmark source code to measure the time spent on read operations in addition to the compaction execution time.

B. EXPERIMENTAL RESULTS

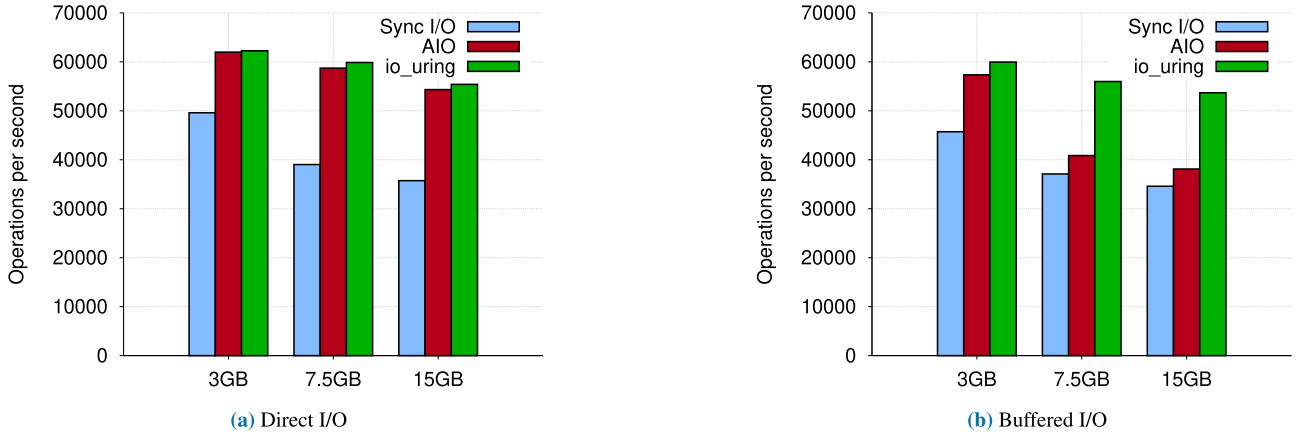
1) OVERALL PERFORMANCE

We compared the overall compaction performance of p-fetch using AIO and p-fetch using io_uring with the conventional compaction method using sync I/O. For three compaction methods, we measured the total compaction execution time and the time spent reading during compaction with 3GB, 7.5GB, and 15GB data sets opened in direct I/O and buffered I/O, respectively. In this test, we set the benchmark to call the compaction task once manually, and other tasks except for compaction not to work. Additionally, we set the submission batch size for AIO and io_uring to 128, which means the number of I/O operations requested in a batch.

Figure 4 and Figure 5 show the evaluation results when the database file is opened with direct I/O (Figure 4 shows the results of using SATA SSD, and Figure 5 shows the results of using NVMe SSD). The results show that when using AIO and io_uring, the operations' performance except read is the same as sync I/O in both storage devices, while the read operation's performance has improved about 8 to 10 times compared to sync I/O. As a result of improved read performance, the overall compaction performance improved 1.4 to 2 times and 2 to 3.5 times when using NVMe SSD and SATA SSD, respectively.

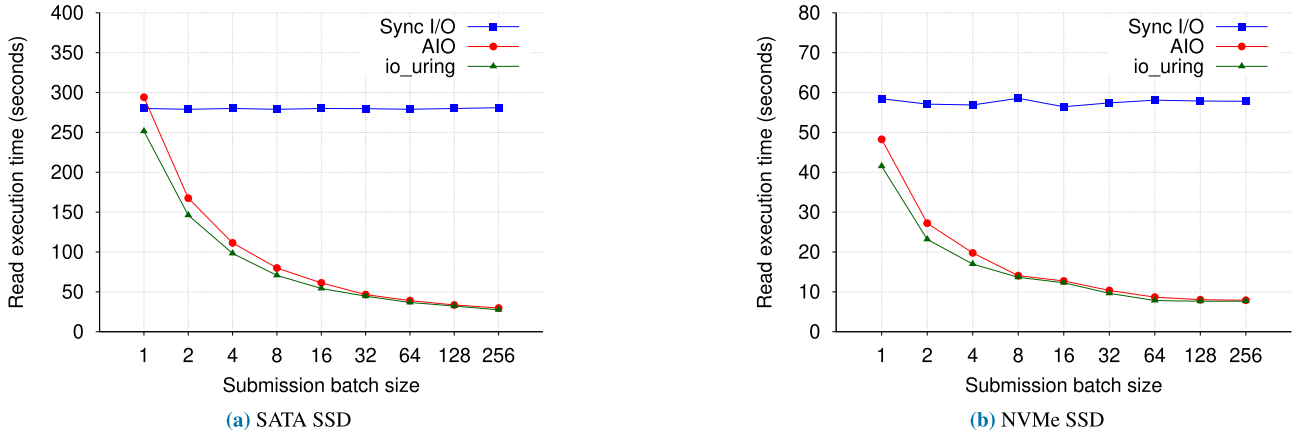
Next, Figure 6 and Figure 7 show the evaluation results when the database file is opened with buffered I/O (Figure 6 shows the results of using SATA SSD, and Figure 7 shows the results of using NVMe SSD). Since io_uring supports asynchronous I/O for the file opened in buffered I/O, p-fetch using io_uring significantly improved the compaction's read performance than sync I/O and p-fetch using AIO. When using NVMe SSD as a storage device, read operations' performance using io_uring was 5.2 to 6.3 times faster than sync I/O, resulting in the overall compaction performance improvement of about 1.3 to 1.7 times. When using SATA SSD as a storage device, read operations' performance using io_uring was more than six times faster than sync I/O, resulting in the overall compaction performance improvement of about 1.7 to 3.1 times.

To verify the impact of compaction schemes on the overall throughput of the ForestDB, we performed the update-heavy workload of the YCSB benchmark [21] with 50 percent reads, and 50 percent writes. We requested 2M, 5M, and 10M operations on the 3GB, 7.5GB, and 15GB datasets, respectively, and measured the average number of operations processed per second(in each case, compaction was called



(a) Direct I/O (b) Buffered I/O

FIGURE 8. Overall throughput of the ForestDB with update-heavy YCSB workload.



(a) SATA SSD (b) NVMe SSD

FIGURE 9. Execution time of read during ForestDB compaction according to the submission batch size.

once.). As shown in Figure 8, we demonstrated that the overall throughput of the ForestDB was increased as p-fetch significantly improved the compaction performance. In particular, p-fetch using io_uring was effective in both Direct I/O and Buffered I/O.

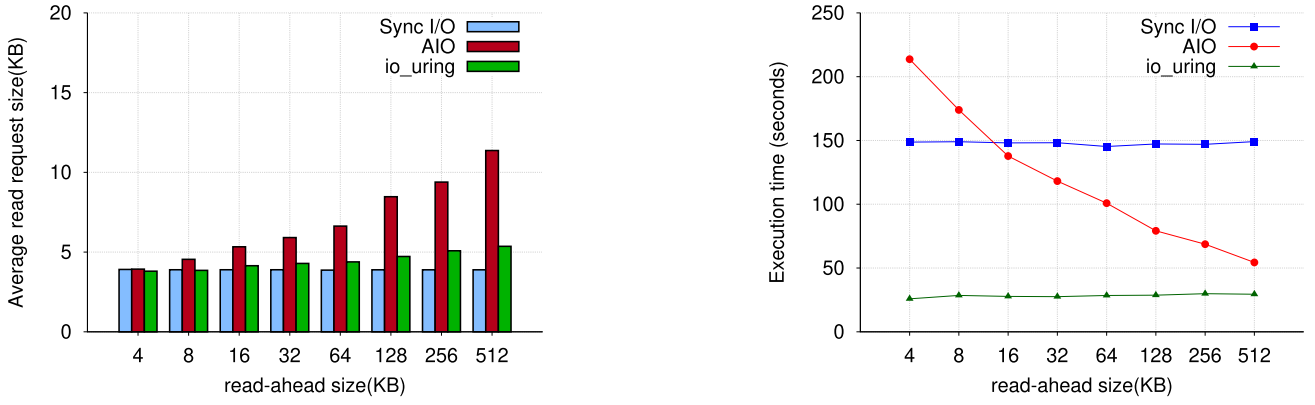
2) IMPACT OF THE SUBMISSION BATCH SIZE

To examine the effect of requesting I/O operations in a batch through async I/O on the ForestDB's compaction performance, we measured the compaction's read performance by changing the submission batch size, as seen in Figure 9. In this experiment, we used a 7.5GB database file opened in direct I/O mode and changed the submission batch size from 1 to 256.

Figure 9a and Figure 9b illustrate the results when using a SATA SSD and an NVMe SSD, respectively. The results show that p-fetch methods improve the read operations' performance as the submission batch size increases in both SATA SSD and NVMe SSD cases, while the read performance of the conventional compaction method using sync I/O is not affected by the submission batch size. In the case of AIO and io_uring, submitting 256 I/O operations in a batch is up to ten times faster than submitting one I/O operation. Since the physical location of data within the SSD affects the parallelism and performance, and the number of SSD channels is limited, the submission batch size and the read performance are not proportional.

3) IMPACT OF THE PREFETCHING

On the other hand, we observed that when the file is opened with buffered I/O, p-fetch using AIO shows better performance than sync I/O even though AIO does not support asynchronous I/O. We found that the reason is that AIO exploits the SSD's package-level parallelism by requesting a larger size of I/O operation than the I/O operation requested through the I/O prefetching technique in Linux called readahead [22]. To examine the readahead technique's effect on the read performance using AIO, we changed the readahead size, which means the maximum size of prefetched data, from 4KB to 512KB, and measured the average read operation size requested to the storage device and the performance, as shown in Figure 10a. When the readahead size is set to 4KB, as same as the ForestDB block size, the AIO's average read request size is about 4KB, similar to sync I/O and



(a) Average read request size

(b) Execution time of read operation

FIGURE 10. Average read request size and execution time of read according to the readahead size.

io_uring. As the readahead size increases, the AIO's average read request size tends to increase. When the readahead size is set to 512K, the average read request size of AIO is three times larger than that of sync I/O. We also found that the read operation performance increases as the average read request size increases, as presented in Figure 10b. The result shows that when the readahead size is 4KB, the read performance of AIO is slower than that of sync I/O. As the readahead size increases, AIO's read performance is 2.7 times faster than sync I/O when setting the AIO's readahead size to 512KB.

VI. CONCLUSION

In this paper, we demonstrated that the slow read performance of the append-only B-tree based ForestDB's compaction deteriorates the overall performance. We demonstrated that the slow read performance was the underutilization of the SSD's internal parallelism, and the reason that the SSD was not fully utilized was the compaction's read operation using synchronous I/O.

We proposed p-fetch, a new compaction technique for append-only B-tree-based key-value store to overcome the problem. P-fetch utilizes the SSD's internal channel-level parallelism by requesting multiple I/O operations in batch through async I/O interfaces, thereby significantly improving compaction's read performance. We designed p-fetch and implemented it in ForestDB using Linux async I/O interfaces, AIO and io_uring. Our evaluation results show that p-fetch remarkably improves compaction's read performance, result in the overall compaction performance enhancement. In particular, p-fetch using io_uring is prominent both in terms of performance and use cases.

REFERENCES

[1] R. Cattell, "Scalable SQL and NoSQL data stores," *ACM SIGMOD Rec.*, vol. 39, no. 4, pp. 12–27, May 2011.

[2] J. Han, H. E, G. Le, and J. Du, "Survey on NoSQL database," in *Proc. 6th Int. Conf. Pervas. Comput. Appl.*, Oct. 2011, pp. 363–366.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, Jun. 2008.

[5] S. Ghemawat and J. Dean. *Leveldb*. Accessed: Mar. 2, 2021. [Online]. Available: http://code.google.com/p/leveldb

[6] Facebook. *Rocksdb*. Accessed: Mar. 2, 2021. [Online]. Available: https://github.com/facebook/rocksdb

[7] J.-S. Ahn, C. Seo, R. Mayuram, R. Yaseen, J.-S. Kim, and S. Maeng, "ForestDB: A fast key-value storage system for variable-length string keys," *IEEE Trans. Comput.*, vol. 65, no. 3, pp. 902–915, Mar. 2016.

[8] CouchDB. *Apache Couchdb*. Accessed: Mar. 2, 2021. [Online]. Available: https://couchdb.apache.org

[9] Couchbase. *Couchbase Server*. Accessed: Mar. 2, 2021. [Online]. Available: https://www.couchbase.com/products/server

[10] G. Graefe, "Write-optimized b-trees," in *Proc. 13th Int. Conf. Very Large Data Bases-*, vol. 30, 2004, pp. 672–683.

[11] G.-J. Na, B. Moon, and S.-W. Lee, "In-page logging b-tree for flash memory," in *Proc. Int. Conf. Database Syst. Adv. Appl.* Brisbane, QLD, Australia: Springer, 2009, pp. 755–758.

[12] H. Roh, S. Kim, D. Lee, and S. Park, "As b-tree: A study of an efficient b+-tree for ssds," *J. Inf. Sci. Eng.*, vol. 30, no. 1, pp. 85–106, 2014.

[13] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. USENIX Annu. Tech. Conf.*, vol. 57. Boston, MA, USA, 2008, pp. 1–14.

[14] F. Chen, B. Hou, and R. Lee, "Internal parallelism of flash memory-based solid-state drives," *ACM Trans. Storage*, vol. 12, no. 3, pp. 1–39, Jun. 2016.

[15] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1141–1155, Jun. 2013.

[16] J. A. Boyd, "Serial ATA advanced host controller interface (AHCI) 1.3," Intel, Serial ATA, Jun. 2008.

[17] NVM Express Workgroup. *NVM Express Base Specication Revision 1.4 b*. Accessed: Mar. 2, 2021. [Online]. Available: https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4b-2020.09.21-Ratified.pdf

[18] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee, "B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives," *Proc. VLDB Endowment*, vol. 5, no. 4, pp. 286–297, Dec. 2011.

[19] A. Jens. (2019). *Efficient IO With io_uring*. Accessed: Mar. 2, 2021. [Online]. Available: https://kernel.dk/io_uring.pdf

[20] Couchbaselabs. (2020). *Forestdb-Benchmark*. [Online]. Available: https://github.com/couchbaselabs/ForestDB-Benchmark/

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput. (SoCC)*, 2010, pp. 143–154.

[22] F. Wu, "Sequential file prefetching in Linux," in *Advanced Operating Systems and Kernel Applications: Techniques and Technologies*. Hershey, PA, USA: IGI Global, 2010, pp. 218–261.

JONGBAEG LEE received the B.S. degree in computer science from Sungkyunkwan University, Suwon, South Korea, in 2013, where he is currently pursuing the Ph.D. degree in electrical and computer engineering. His research interests include flash-based database technology and distributed database systems.

GIHWAN OH received the B.S. degree in computer science from Sungkyunkwan University, Suwon, South Korea, in 2013, where he is currently pursuing the Ph.D. degree in electrical and computer engineering. He has been experienced research interships at Altibase, South Korea, OCZ Technology, Samsung Electronics, Couchbase, and Circuit Blvd, USA. His research interests include optimizing database and storage management system based on flash memory and next-generation memory storage devices exploiting intelligent SSD technologies.

SANG-WON LEE received the Ph.D. degree from the Department of Computer Science, Seoul National University, in 1999. He was a Research Professor with Ewha Womans University and a Technical Staff with Oracle, South Korea. Since 2002, he has been a Professor with the College of Computing, Sungkyunkwan University, Suwon, South Korea. His research interest includes flash-based database technology.