Cluster Comput (2017) 20:3551–3563
DOI 10.1007/s10586-017-1101-3

Design and implementation of an efficient flushing scheme for cloud key-value storage

Yongseok Son[1] · **Heon Young Yeom**[1] · **Hyuck Han**[2]

Received: 24 February 2017 / Revised: 16 June 2017 / Accepted: 3 August 2017 / Published online: 24 August 2017

Abstract A key-value store is an essential component that is increasingly demanded in many scale-out environments, including social networks, online retail environments, and cloud services. Modern key-value storage engines provide many features, including transaction, versioning, and replication. In storage engines, transaction processing provides atomicity and durability by using write-ahead logging (WAL), which flushes log data before the data page is written to persistent storage in synchronous commit. However, according to our observation, WAL is a performance bottleneck in key-value storage engines since the flushing of log data to persistent storage incurs a significant overhead of lock contention and `fsync()` calls, even with the various optimizations in the existing scheme. In this article, we propose an approach to improve the performance of key-value storage by optimizing the existing flushing scheme combined with group commit and consolidate array. Our scheme aggregates the multiple flushing of log data into a large request on the fly and completes the request early. This scheme is an efficient group commit that reduces the number of frequent lock acquisitions and `fsync()` calls in the synchronous commit while supporting the same transaction level that the existing scheme provides. Furthermore, we integrate our flushing scheme into the replication system and evaluate it by using multiple nodes. We implement our scheme on the WiredTiger storage engine. The experimental results show that our scheme improves the performance of the key-value workload compared to the existing scheme.

Keywords Key-value storage · Flash-based SSD · Write-ahead logging

1 Introduction

A key-value store plays an important role in many scale-out environments, including social networks, online retail environments, and cloud services [3]. Many enterprise cloud services adopt a key-value store (e.g., DynamoDB [33] and MonogoDB [4] at Amazon [8], Redis [6] at GitHub, Memcached [10] at Facebook, and Voldemort [38] at Linkedin). Because of the significance of key-value store as mentioned above, much effort has been made to provide high performance, scalability, and reliability. This has resulted in various functionalities in the latest key-value systems, including replication, versioning, locking, and transaction [1,9].

Modern key-value storage engines [15,22,40] support transaction properties (e.g., atomicity and durability) by using write-ahead logging (WAL) [25], which writes a log data to the log area before the data page is written to the original data location. In WAL, transactions synchronously flush their own log data to persistent storage during the commit procedure by executing an `fsync()` system call. Unfortunately, flushing log data to persistent storage imposes an overhead since `fsync()` calls flush the dirty pages (i.e., data

A preliminary version [35] of this article was presented at the 1st IEEE Internaltional Workshops on Foundations and Applications of Self* Systems, Augsburg, Germany, September 2016.

✉ Hyuck Han
hhyuck96@dongduk.ac.kr

Yongseok Son
ysson@dcslab.snu.ac.kr

Heon Young Yeom
yeom@dcslab.snu.ac.kr

[1] Department of Computer Science and Engineering, Seoul National University, Seoul, South Korea

[2] Department of Computer Science, Dongduk Women's University, Seoul, South Korea

and metadata for the log file) in file systems for serialized and ordered updates.

Some studies have addressed the overhead issue associated with commit and flushing operations. Group commit [16] groups multiple log flush requests during a specific time into a single I/O operation. This scheme is widely used in modern database systems [11,40] to significantly improve logging performance. Asynchronous commit [30] allows transactions to be completed without waiting for the completion of their log flush requests. Unfortunately, the committed work can be lost if a crash occurs. Aether's [18] flush pipelining, which combines asynchronous and group commit for better performance, delays the return to the client until the transaction is committed by a daemon; meanwhile, it continuously processes other transactions. However, it could lose more data modified by transactions if a crash occurs.

This article investigates an efficient approach to improve the performance of key-value storage by optimizing the existing flushing scheme in transaction processing. Our analysis shows that transactions flush log data and acquire locks unnecessarily on the basis only of their own log sequence number (LSN) in the current strategy, thereby incurring frequent lock acquisition and `fsync()` calls even if the strategy is already improved by group commit [16] and consolidate array [18]. Our key idea is to aggregate flushing requests from other transactions more efficiently and complete grouped transactions faster without additional lock acquisition and `fsync()` calls.

Furthermore, we integrate our scheme into the replication system and evaluate it by using multiple nodes to provide high reliability and availability. The system performs a synchronous replication, and issues and completes the key-value request in parallel. In our previous work [35], we focused on the study of a single node configuration. This article extends and applies our scheme to a replication system.

We analyze the WiredTiger storage engine [40], which is a high performance, scalable, and open source extensible platform for data management that includes ACID [14] transactions with standard isolation levels and durability.WiredTiger is gaining popularity due to its high performance and scalability and it was selected as a default storage engine for MongoDB, starting with version 3.0 [26]. Recently, cloud providers, such as Amazon, have begun to provide MongoDB as a platform in their cloud services, as a larger demand is recognized to deploy NoSQL databases in cluster settings for scalability.

WiredTiger adopts group commit and consolidation array, which improves the traditional logging by allowing the transactions to acquire a log buffer in a more fine-grained way instead of acquiring the coarse-grained lock for a large log buffer. We implement our scheme on top of its consolidation array implementation [18] with synchronous group commit and evaluate its performance on solid-state drives (SSDs). The experimental results demonstrate that our scheme improves the performance in different configurations compared to the existing scheme.

Our contributions can be summarized as follows:

- We analyze and explore the existing flushing scheme with combined consolidate array and group commit from a real implementation.
- We propose an efficient optimization technique to improve transaction performance.
- We integrate our scheme into a replication system to provide reliability and availability for a key-value storage engine.
- The experimental results demonstrate that the proposed scheme improves performance compared to the existing scheme on different configurations.

The rest of this article is organized as follows: Sect. 2 discusses the background and motivation. Section 3 describes the flushing optimization. Section 4 presents the experimental results. Section 5 describes related work. Finally, Sect. 6 concludes this article.

2 Background and motivation

2.1 Transactional log in key-value storage engine

In WAL [25], every log record in a transaction log is uniquely identified by its LSN for ordering updates. LSNs are ordered such that if LSN_2 is greater than LSN_1 and the log record of LSN_2 is written to persistent storage, the change in LSN_1 is also written. In synchronous commit, a transaction gets its own LSN and flushes its own log record individually by calling `fsync()` after acquiring the lock, which protects the log data structure and ensures a sequential order of flushing.

WiredTiger chose the consolidation array instead of the traditional log buffer insertion [18] because the latter incurs high contention when acquiring a coarse-grained lock before inserting each log record into a large log buffer. The consolidation array divides a log buffer into slot arrays, where the slots consolidate transaction threads without a large lock.

As shown in Fig. 1, a leader (T_1)[1] allocates a slot ($slot_1$), and the following threads join the slot by getting their own offsets[2] within the slot buffer using *Compare-and-Swap* (CAS) operations. The leader prepares another slot ($slot_2$) from the slot pool, and then closes the joined slot to block threads from joining, at which point a new slot can be allocated by another leader (T_4).

[1] A leader is the first thread to acquire a log buffer.

[2] The offset is its own LSN.

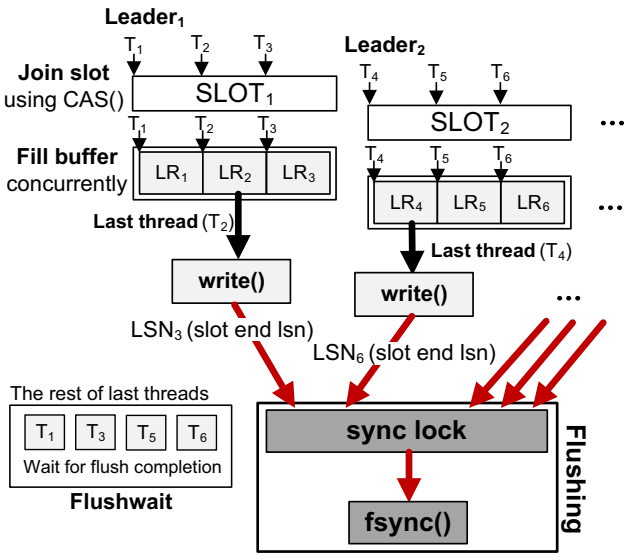


Fig. 1 Overview of consolidation array with synchronous commit. *T* thread, *LR* log record

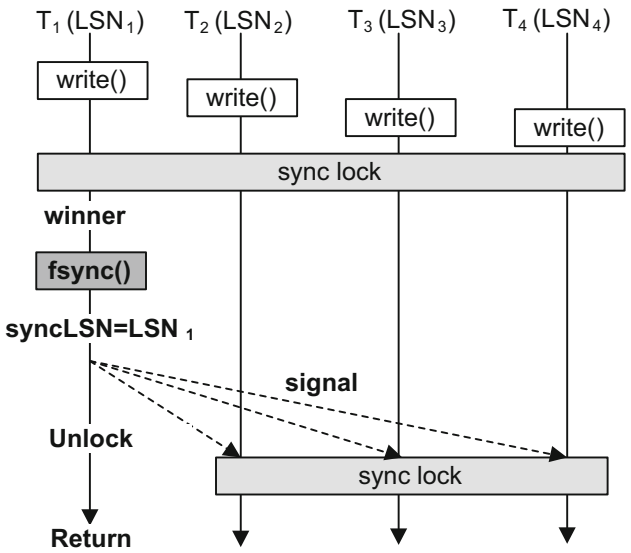


Fig. 2 An example of existing flushing

T_1 acquires the log buffer based on the group size of the slot on behalf of the joined threads.[3] After this, the joined threads concurrently fill the slot buffer with their log records at their own offsets. Next, the last thread to fill the slot buffer, T_2, which is the flushing thread in this example, writes the grouped slot buffer using a `write()` system call and performs an `fsync()` call based on the end LSN of the slot (LSN_3). At this time, T_2 contends with another last thread (T_4) to acquire the sync lock.

Before any flushing thread acquires the lock, it is guaranteed that any slot buffer with a smaller LSN than the LSN for the flushing thread is written to the file system. The remaining threads (T_1, T_3, T_5, and T_6) wait for the log records of their own LSN (flushwait) to be flushed. We note that if there is low contention of acquiring the log buffer, the consolidation array is not activated in their algorithm [18]. The impact of this performance is mentioned in the evaluation section.

2.2 Existing flushing scheme

The existing flushing scheme is based on a group commit method. After the flushing threads contend for the sync lock, a winner thread flushes the log records of the other threads. Consequently, the flushing scheme writes log requests to storage through a large request. However, the existing flushing scheme does not consider the unnecessary lock acquisitions and `fsync` calls since it cannot identify whether the individual request is completed or not. Therefore, each thread still performs individual flushing operations instead of performing them in a co-operated manner albeit group commit.

[3] The group size is the size of the total joined log records in the slot.

Figure 2 shows an example of the existing flushing scheme. There are four flushing threads $T_1 - 4$ arriving concurrently with unflushed $LSN_1 - 4$ after writing their own log records through `write()` system calls. They try to acquire a sync lock and process flushing operations one by one. In our example, T_1 becomes the first winner by acquiring the lock, and it compares LSN_1 with syncLSN, which is the latest flushed LSN written to persistent storage. If LSN_1 is smaller than syncLSN, T_1 just releases the lock and finishes the commit work since the slot buffer of the LSN_1 is already written to storage. Otherwise, T_1 starts to flush by calling `fsync()` and then updates syncLSN to LSN_1. Subsequently, T_1 signals waiters and releases the sync lock so that the waiters ($T_2 - 4$) wake up and try again to acquire the sync lock in order to start flushing.

The existing scheme cannot identify whether the log records are actually written to storage by other threads or not since the threads perform flushing on the basis only of their LSNs. In this example, the `fsync()` called by T_1 flushes the log records of $T_1 - 4$ to persistent storage since the `fsync()` call flushes all dirty data and metadata in a log file. Although the committed work is already applied to persistent storage, they ($T_2 - 4$) try to acquire the sync lock again, which increases lock contention.

If the next winner is T_2, it starts to flush the log records written by other threads (e.g., $T_5 - x$) even though its log records have already been flushed by T_1 since its own LSN (LSN_2) is larger than syncLSN (LSN_1). In the worst case, each thread $T_1 - 4$ flushes individually, increasing the `fsync()` frequency fourfold. This aggravates transaction latency because T_2 calls an additional `fsync()` even if it suffices to return. Moreover, the resultant higher frequency

of `fsync()` decreases I/O efficiency by reducing the aggregated I/O size per `fsync()` call.

In short, the existing flushing scheme is a form of group commit but it is not an efficient one. Even though the actual I/O of a request occurs, the threads issue and complete their requests on the basis only of their own LSNs. Therefore, they encounter increased contention to the sync lock, higher transaction latency, and inferior I/O efficiency due to the smaller aggregated I/O size. This scheme negatively affects the performance of applications even if fast storage devices [23,27,34,36,37] are adopted.

3 Design and implementation

3.1 Flushing optimization

Our scheme to optimize flushing in synchronous commit is to fill the semantic gap between logging in the storage engine and the `fsync()` primitive in the file system by flushing log records while also considering LSNs from other flushing threads. The proposed scheme is an efficient group commit that aggregates the flushing requests of other LSNs into a single request on the fly and completes the request early to reduce the number of flushing and lock acquisitions.

In the existing scheme, there is no global information that can be shared among the flushing threads for more efficient flushing. Therefore, we catch the start and end points of the flushing of each request by identifying each request from transaction threads. Each thread assembles its own request for global information so that the winner thread that acquires the sync lock can consider the information.

To provide the global information for all requests, we maintain a primary queue and a secondary queue to reduce queue contention by pipelining enqueue and dequeue operations; flushing threads enqueue their requests in the primary queue while the winner thread dequeues requests from the secondary queue after swapping.

As shown in Fig. 3, four threads ($T_1 - 4$) arrive concurrently to flush log records with unflushed $LSN_1 - 4$. Each thread makes its own request ($R_1 - 4$), which includes its LSN and the *completion flag*[4] for the LSN. Each thread then enqueues its request in a primary request queue and tries to acquire the sync lock after writing the log records to the file system. In this example, T_1 acquires the sync lock (winner) and aggregates the requests while other threads ($T_2 - 4$) wait for the completion flag of their own requests by waiting for the signal from T_1.

After this step, T_1 first closes the primary request queue (RQ) to block threads from entering, initializes the secondary queue, and swaps two queues. The requests from concurrent

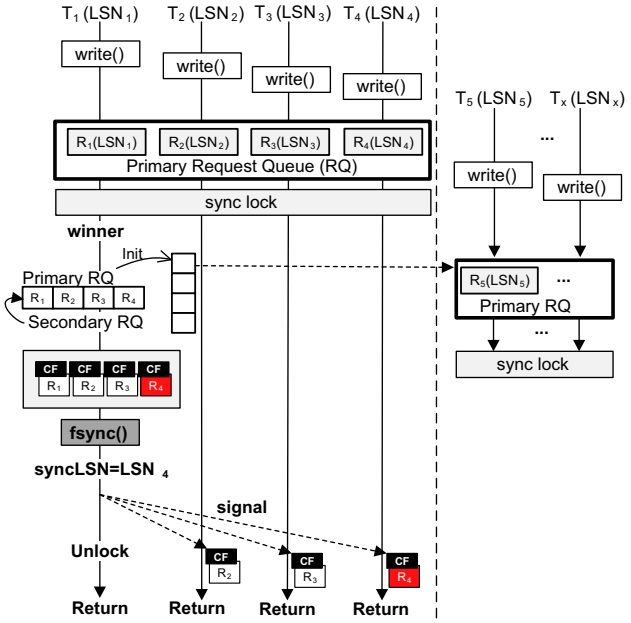


Fig. 3 Proposed flushing

transactions are aggregated just until the primary queue is closed by the winner thread.

Next, T_1 dequeues the requests from the secondary queue while giving the empty primary queue to upcoming new requests. At this time, the flushing threads (T_5 and T_x) can enqueue their own requests to the primary queue until the next winner is decided. After dequeuing the requests from the secondary queue, T_1 sets the completion flags of the requests and finds the lastLSN (LSN_4) by comparing all LSNs of the requests from the queue. Finally, it starts to flush, updates the syncLSN to lastLSN, and releases the sync lock after signaling the waiters. The waiting threads return after checking the completion flags without retrying to acquire the sync lock.

We note that the winner issues and completes the flushing requests on the basis of the lastLSN, regardless of which thread acquired the sync lock. By referencing other LSN information, this scheme maximally reduces frequent flushing so that the reduced number of `fsync()` calls and lock acquisitions improves the transaction processing performance.

Consequently, our scheme decreases the frequency of `fsync` by flushing a greater amount of log data at once and reduces the number of sync lock acquisitions by using early completion. This allows threads to return without retrying the sync lock when flushing is unnecessary since a large request is more efficient than multiple small requests.

3.2 Replication in key-value storage

Replication is a strategy in which multiple copies of some data are stored in multiple nodes [5]. Replication improves

[4] The completion flag denotes whether the LSN is completed or not.

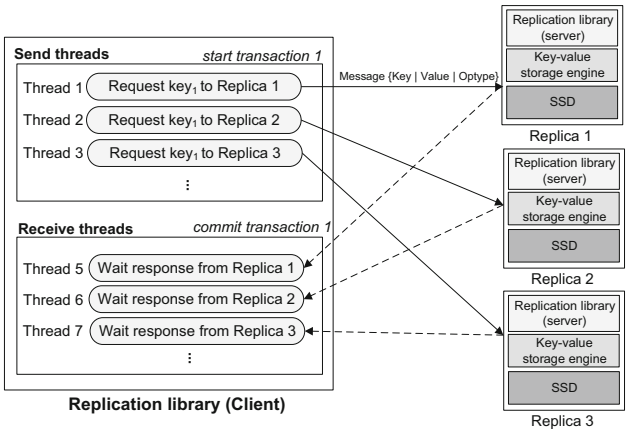


Fig. 4 Replication on key-value storage

reliability and availability by storing the data in more than one node. If some data in a node fail, a system can operate by using replicated data, thus increasing reliability, availability, and fault tolerance [13]. At the same time, as the data is stored in multiple nodes, the request can find the data close to the site where the request originated, thereby increasing the performance of the system.

The flushing can more affect the performance in a replication configuration than that in a single configuration. It is because the flushing operations occur in all replication nodes and a transaction is committed only if all the flushing operations are completed in all nodes. Thus, we integrate our flushing scheme into a replication system to improve the flushing performance in the replication configurations. The replication system provides synchronous replication[5] so that the transactions in all nodes are consistent. The transaction can be committed only if all nodes are willing to commit it. This can result in an all-or-nothing atomicity property in a replication system.

Figure 4 shows our replication system, which consists of three replicas. A client includes a replication library on the client side. A replica includes a replication library on the server side, a key-value storage engine, and an SSD. When a client requests a write operation such as INSERT, UPDATE, or DELETE for a key-value pair, the replication library on the client side issues the transaction request to replicas at once. The request includes key, value, and operation type.

[5] Replication can be broadly classified into two categories, such as synchronous and asynchronous replication. In synchronous replication, the transactions are committed simultaneously in all nodes. The master and the slave always remain synchronized. Thus, the data is guaranteed to be consistent in all nodes when the transaction is committed. Meanwhile, in asynchronous replication, the transactions are committed at the master server first and then they are replicated to the slave. This means that the master and slave may not be consistent. The advantage of using asynchronous replication is that it is faster and scales better than synchronous replication. However, the data is not guaranteed to be consistent in all nodes.

When the request arrives in the server node, the replication library on the server side parses the request and takes action according to the specified request. Next, the replication library starts the transaction by using the key-value storage engine. The key-value storage performs the write operation and then commits the transaction. This procedure is executed in all nodes.

When the transaction is committed, the replication library on the server side returns the committed transaction ID to the client. The client library records the transaction ID until all server nodes return the IDs. When the client library receives the response for the completion from all nodes, it finally commits the transaction. If the client library cannot receive the completion, the client library aborts the transaction. This replication system guarantees that transactions are consistent in all nodes.

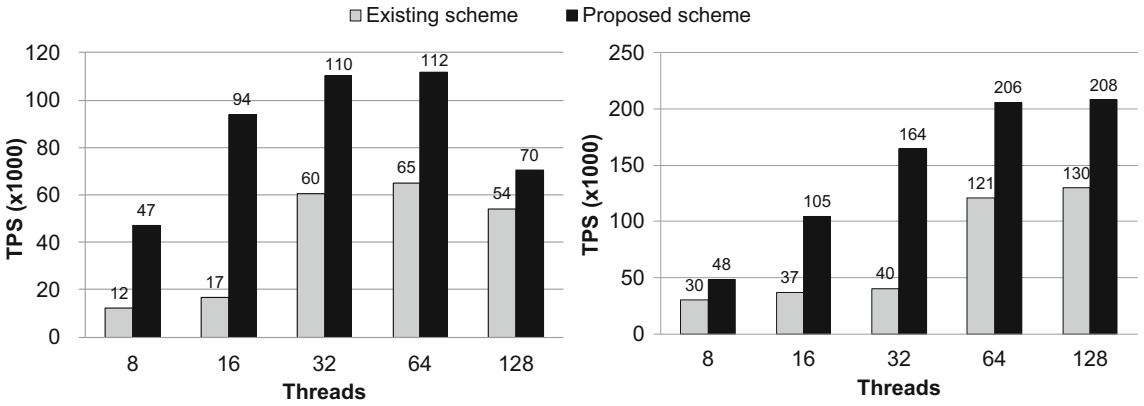
3.3 Implementation

We implemented our flushing mechanism on top of the consolidation array and group commit implementation on the 2.6.1 version of WiredTiger. We modified 110 LoC in total, which are rather small modifications to the storage engine. The modifications are mainly done on `log_release()`, in which transaction threads flush log records to persistent storage.

We make two queues by using circular doubly linked lists for pipelining enqueue and dequeue operations for the requests of log records. When each thread starts to flush, it makes its own request structure, which contains the following three fields: a completion flag, an LSN, and a linked list pointer. We use a spin lock to protect the queue operations in a fine-grained way with little overhead.

4 Evaluation

In this section, we show the experimental results for the performance of the existing and proposed flushing schemes. Our machine has two Intel Xeon CPU E5-2670 (2.6 GHz) with eight physical cores each, 8 GiB of memory, and PCIe 3.0, and it runs Linux with kernel 3.14.3. We use a Samsung NVMe SSD (XS1715) [32] with a capacity of 400 GiB and a Samsung 840Pro with a capacity of 256 GiB. In the replication configurations, we run experiments on a cluster system consisting of four identical machines connected by a 10 GbE network. Each machine is equipped with an Intel Core CPU i7-4790 (3.60 GHz) with four physical cores, which totals eight cores with hyperthreading, 32 GiB DRAM, and SATA3 interface. We use 240 GiB Samsung SM843Tn [31] on all storage servers. We use EXT4 [24] as an underlying file system for WiredTiger.



(a) INSERT **(b)** UPDATE (50%) / SEARCH (50%)

Fig. 5 TPS results under different threads (micro-benchmark)

All experimental results report the average value of five runs. We enable log and synchronous options, use btree for indexing, and use a slot pool size of 16 in the WiredTiger configuration.

4.1 A single node configuration

4.1.1 Micro-benchmark results

We performed a key-value workload provided by WiredTiger as follows: 50,000 single-operation transactions per thread with keys and values of size 20 and 100 bytes [1], respectively. Figure 5a shows the transaction per second (TPS) results for the INSERT workload. Our scheme outperforms the existing scheme, including consolidate array and group commit by 3.9×, 5.5×, 1.83×, 1.72×, and 1.3× at 8, 16, 32, 64, and 128 threads, respectively.

The cases of 8 and 16 threads show a much larger performance gap between the two schemes since there are almost no consolidation operations in the existing scheme at this number of threads. This can be attributed to the fact that the consolidation array is only activated if there is lock contention to the log buffer, as presented in a previous study [18]. If the threads succeed in acquiring a lock to secure the log buffer, they perform a log insert directly as in traditional logging since the overhead of the consolidation operation can be larger than that of the lock contention.

However, our scheme aggregates individual flushing requests more efficiently than the existing group commit does in either consolidation or traditional logging. The two schemes present the highest TPS at 64 threads, but the performance begins to decrease afterward because a large number of threads on a limited number of cores incur frequent context switches and increase the number of threads waiting for flushing completion.

Figure 5b shows the TPS results for the UPDATE (50%) and SEARCH (50%) workloads. Our scheme outperforms the existing scheme by 1.6×, 2.8×, 4.1×, 1.7×, and 1.6× at 8, 16, 32, 64, and 128 threads, respectively. The overall performance gap between the two schemes in this workload is less than that in the INSERT workload due to mixed read and write. Unlike the INSERT workload, this workload shows the largest gap at 32 threads since the consolidation is not activated at this number of threads due to the reduced lock contention to the log buffer; this can be attributed to the increased portion of SEARCH threads. In conclusion, our scheme shows better performance on top of consolidation array and group commit.

4.1.2 Macro-benchmark results

We performed a sysbench OLTP workload [21] in a single configuration. The sysbench OLTP workload consists of a mixture of UPDATE, DELETE, and INSERT in a transaction. Figure 6 shows the TPS results under different threads in

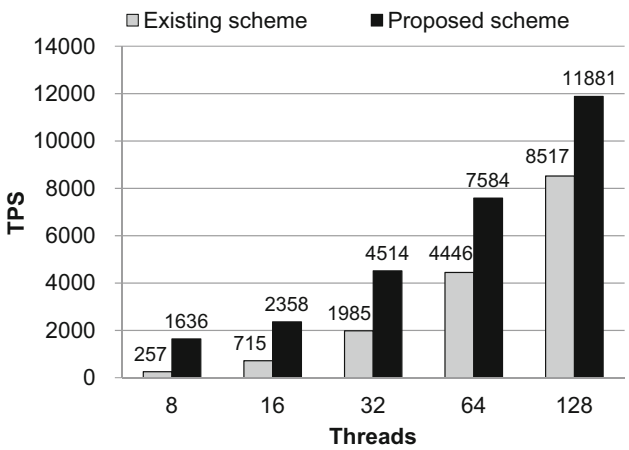


Fig. 6 TPS results under different threads (macro-benchmark)

the existing and proposed schemes. Our scheme improves performance by 6.4×, 3.3×, 2.3×, 1.7×, and 1.4× in the case of 8, 16, 32, 64, and 128 threads, respectively, compared to the existing scheme.

Especially in the case of 8 and 16 threads, the TPS is lower than that in other cases because of the relatively low number of threads. As the number of threads is increased, the TPS of both existing and proposed schemes is increased. Consequently, in a single configuration, our scheme is beneficial and it outperforms the existing scheme even in the case of the macro-benchmark as well as the micro-benchmark.

4.1.3 Experimental analysis

Table 1 shows the performance analysis results for the two schemes in terms of average latency per transaction and the number of operations. The proposed scheme improves the average transaction latency by 3.8×, 5.5×, 1.79×, 1.71×, and 1.3× compared to the existing scheme, which shows a similar gap to that of TPS.

In the existing scheme, the latency of the sync lock waiting for the `fsync()` call accounts for 82.4, 82.7, and 41.6% of the transaction latency at 8, 16, and 32 threads, respectively. As the number of threads is increased, the latency of waiting for flush completion by the rest of the flushing threads is increased at 64 and 128 threads (flushwait).

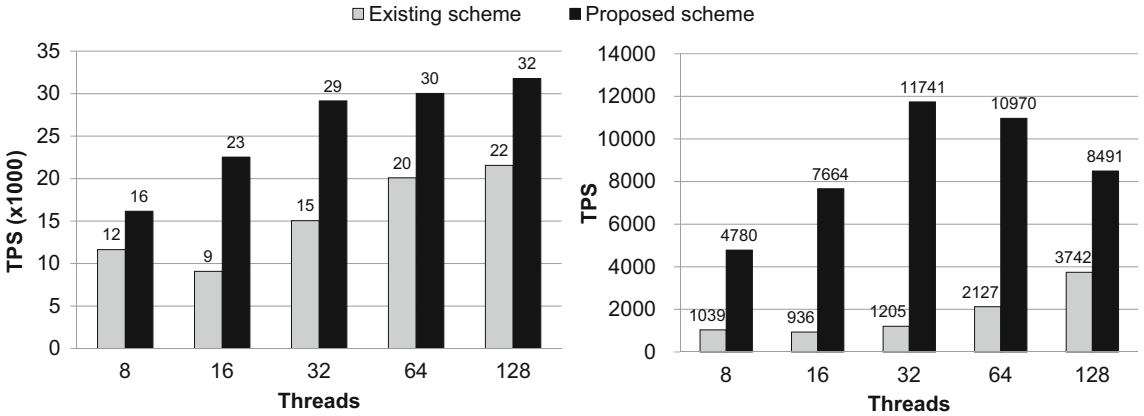
Our scheme improves the latency of the sync lock and `fsync` by 4.4×/3.7×, 8.5×/5.1×, 2.1×/1.9×, 1.6×/1.6×, and 1.3×/1.6**x** at 8, 16, 32, 64, and 128 threads, respectively, by reducing the total number of sync lock and `fsync`. The latency of sync lock is larger than the `fsync` portion since the transactions are waiting for the `fsync`, which is issued by only one transaction thread among flushing threads. The average latency of `fsync()` calls is approximately 100–130 μs, which means that the flushing threads wait for at least this amount of time in both schemes.

At eight threads, the average number of consolidated threads per slot (consol.) is zero in both schemes, which means that all transaction threads perform flushing with only the existing group commit. In the existing scheme, this lack of consolidation incurs a higher latency of sync lock and `fsync` calls per transaction at eight threads. However, the latency is improved in our scheme since the flushing threads are aggregated as much as 7.7 per queue (aggre.) on average. The performance gap is largest at 16 threads since the number of consolidation is few and the number of aggregations is the highest.

The number of consolidations in the existing scheme is larger than that in our scheme because slower flushing consolidates more threads to a slot. In both schemes, as the number of threads is decreased, the number of consolidated threads per slot is increased due to the limited slot pool size.

Table 1 Experimental analysis (INSERT) for average latency (μs) per transaction and the number of operations in a single configuration

Schemes	Threads	Average latency per transaction (μs)				The number of operations (#)			
		Transaction	Synclock	fsync	Flushwait	Synclock (× 10,000)	fsync (× 10,000)	Consol. (avg.)	Aggre. (avg.)
Existing	8	643	530	70	2.6	156.9	23	0	0
	16	934	773	48	70	434.5	33.3	1.2	0
	32	518	216	12	193	235.5	19.3	4.5	0
	64	977	176	8.3	551	130.6	21.1	6.2	0
	128	2360	230	7.8	1641	111.9	31.8	9.4	0
Proposed	8	168	119	19	2.8	39.4	5.9	0	7.7
	16	169	90	9.3	37	66.8	6.4	1.2	11.4
	32	289	101	6.3	135	81.3	9.6	1.8	9.4
	64	570	107	5	375	78.7	14.4	3.5	6.4
	128	1818	173	4.8	1342	80.9	24.1	6.2	4.4



(a) Replication configuration (factor: 1)

(b) Replication configuration (factor: 3)

Fig. 7 TPS results under different configurations (micro-benchmark)

The number of aggregated threads means that a flushing thread issues and completes the flushing requests with a single `fsync()` and lock acquisition. The number of aggregations in our scheme is decreased as the number of threads is increased from 16 to 128 since greater thread contention by a larger number of threads reduces the number of concurrent flushing threads.

4.2 Replication configuration

4.2.1 Micro-benchmark results

We evaluate the existing and proposed schemes in replication configurations, as shown in Fig. 7a, b, respectively. The workload is the same as the INSERT workload used for a single node configuration. All the results are measured in the client node. Figure 7a shows that our scheme improves performance by 1.3×, 2.5×, 1.93×, 1.5×, and 1.45× in the case of 8, 16, 32, 64, and 128 threads, respectively, compared to the existing scheme with a replication factor of 1. The performance gains are lower than those in the single node configuration due to the network overhead.

In the replication factor of 3 shown in Fig. 7b, our scheme improves performance by 4.6×, 8.1×, 9.7×, 5.15×, and 2.26× in the case of 8, 16, 32, 64, and 128 threads, respectively, compared to the existing scheme. This result shows that the performance gains are higher than those in the single node and the replication factor of 1. This is because all nodes benefit from the flushing optimization. As the synchronous replication system performs a commit operation when all requests are accepted, the existing flushing in each node delays the commit operation.

A comparison of the replication factors of 1 and 3 at 32 threads shows that the TPS of the replication factor of 1 is 12.4× higher than that of the replication factor of 3 in the existing scheme. Meanwhile, the TPS of the replication factor of 1 is 2.4× higher than that of the replication factor of 3 in the proposed scheme. This result demonstrates that a high replication factor significantly affects the existing scheme compared to our proposed scheme. Consequently, our proposed scheme improves the performance in the replication configurations as well as the single node configuration. Especially, the performance gains in the replication factor of 3 are the highest among other configurations.

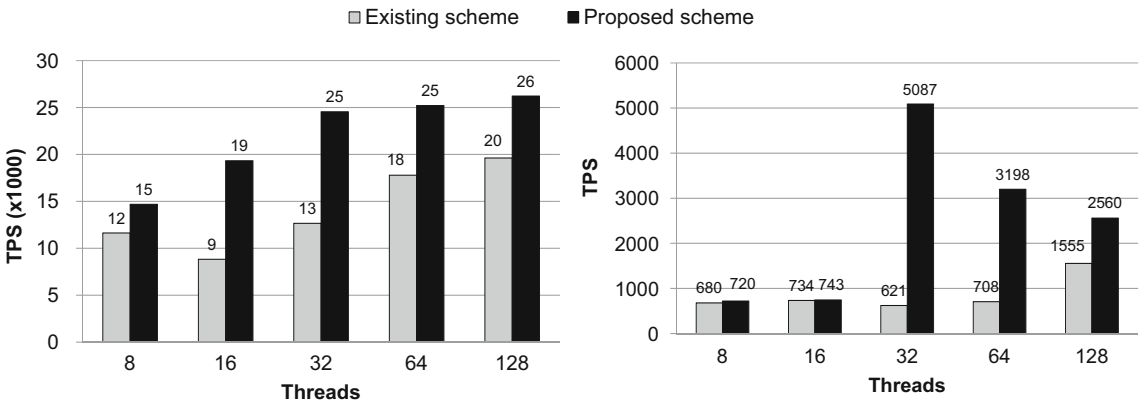
4.2.2 Macro-benchmark results

We run the sysbench OLTP workload used in a single configuration to evaluate the existing and proposed schemes in replication configurations as shown in Fig. 8. Figure 8a shows that our scheme improves performance by 1.25×, 2.1×, 1.92×, 1.38×, and 1.3× in the case of 8, 16, 32, 64, and 128 threads, respectively, compared to the existing scheme with the replication factor of 1. This result shows that the flushing optimization is beneficial to the replication configuration even in the macro-benchmark.

Figure 8b shows that our scheme improves performance by 1.06×, 1.01×, 8.19×, 4.52×, and 1.65× in the case of 8, 16, 32, 64, and 128 threads, respectively, compared to the existing scheme with a replication factor of 3. Similar to the results of micro-benchmark, the highest performance gain of the replication factor of 3 is much higher than that of the replication factor of 1 since all nodes benefit from the flushing optimization. Consequently, our scheme provides a higher TPS than the existing scheme does in the replication factors of 1 and 3 in the macro-benchmark as well as the micro-benchmark.

4.2.3 Experimental analysis

Table 2 shows the performance analysis results in the replication factor of 1. The proposed scheme improves the average



(a) Replication configuration (factor: 1)

(b) Replication configuration (factor: 3)

Fig. 8 TPS results under different configurations (macro-benchmark)

transaction latency in the client node by 1.48×, 1.85×, 1.87×, 1.75×, and 1.62× compared to the existing scheme. This result demonstrates that the proposed scheme is effective in reducing the transaction latency in the replication factor of 1. In terms of the transaction latency of the server node, the proposed scheme improves the average transaction latency by 1.8×, 2.7×, 3.6×, 3.7×, and 3.6× compared to the existing scheme. The performance gains in the server node are higher than those in the client node due to the network overhead.

Similar to the results of a single node configuration, the latency of sync lock accounts for a large portion of the transaction latency. The portion of the flushwait latency of the transaction latency is increased as the number of threads is increased. Overall, the average number of consolidated threads in the replication factor of 1 is less than that in the single node configuration since the log contention is lower due to the network overhead. In our scheme, the number of aggregated threads increases as the number of threads increases. This result demonstrates that the proposed scheme reduces the flushing overhead.

Table 3 shows the results of performance analysis in the replication factor of 3. We report the transaction latency in the client node and the average results in the server nodes. As shown in the table, using the replication factor of 3, the proposed scheme improves the average transaction latency in the client node by 6.8×, 13.8×, 12.5×, 10.6×, and 2.72× compared to the existing scheme. This result demonstrates that the existing flushing scheme affects the transaction latency more using the replication factor of 3 since a transaction can be committed if all nodes finish the flushing operation.

The proposed scheme improves the average transaction latency in the server node by 2.2×, 3×, 3.8×, 3.2×, and 4.9× compared to the existing scheme. The sync lock and flush wait latencies are reduced by 3.5×/11×, 4×/15.7×, 3.76×/9.6×, 4.8×/33.5×, and 3×/29.5× in the case of 8, 16, 32, 64, and 128 threads, respectively. This reduced latency of sync lock and flush wait demonstrates that the transaction latency is improved in each server node. The number of consolidated threads is low and similar to those in the replication factor of 1. Meanwhile, the number of aggregated threads is 3.4, 4.3, 5.5, 6.8, and 7.6 in the case of 8, 16, 32, 64, and 128, respectively. This result demonstrates that the aggregation among threads reduces the flushing overheads in the replication factor of 3.

5 Related work

There have been several research studies to improve performance by reducing the logging and flushing overheads.

Group commit is a widely used technique to reduce logging overhead [16]. This strategy aggregates multiple log flush requests into a single I/O operation. Our scheme is based on group commit, but we propose a more efficient way to do group commit.

Asynchronous commit [30] extends group commit not only by aggregating I/O requests but also by allowing transactions to complete without waiting for the completion of those requests. This scheme provides a significant performance improvement and is used widely in commercial and open source database systems. However, the database system can lose the committed work if a crash occurs, whereas our proposed optimization achieves performance improvement without sacrificing durability.

To reduce the number of context switches due to logging, Aether [18] proposed early lock release that allows transactions to commit without scheduler activity. Aether's flush pipelining delays the return to the client until the transaction is committed by a daemon and continuously processes the transactions to reduce the commit overhead. However, it could lose more data modified by transactions if a crash occurs. Aether also provides a consolidate array to reduce the log contention instead of a transactional log buffer insertion.

Table 2 Experimental analysis (INSERT) for average latency (μs) per transaction and the number of operations using a replication factor of 1

Schemes	Threads	Average latency per transaction (μs)				The number of operations (#)				
		trx (c)	trx (s)	Synclock	fsync	Flushwait	Synclock (× 10,000)	fsync (× 10,000)	Consol. (avg.)	Aggre. (avg.)
Existing	8	450	346	273	55	2.6	453	125	1	0
	16	690	518	454	36.2	6.4	344	37.2	1	0
	32	1005	706	394	34	193.3	326	35.6	1.5	0
	64	1556	863	373	30	298.8	351	36.3	1.9	0
	128	2815	1093	378	29	405.9	311	31.5	2.1	0
Proposed	8	302.3	185	126	43.8	2.2	158	40	1	4.9
	16	372.9	191	130	35	3.8	217	46	1	5.6
	32	535	192	115	25	19.7	229	38	1	6.9
	64	888	233	136	28	26.5	278	40	1	7.8
	128	1742	301	150	26	55.1	285	38	1	8.4

trx (c) and trx (s) denote the transaction in client and server nodes, respectively

Table 3 Experimental analysis (INSERT) for average latency (μs) per transaction and the number of operations using a replication factor of 3

Schemes	Threads	Average latency per transaction (μs)				The number of operations (#)				
		trx (c)	trx (s)	Synclock	fsync	Flushwait	Synclock (× 10,000)	fsync (× 10,000)	Consol. (avg.)	Aggre. (avg.)
Existing	8	4458	464	350	83.4	11	26.7	23	0	0
	16	10,029	520	410	55.7	33	28.3	33.3	1	0
	32	14,723	681	418	42.5	50	28.7	19.3	1.4	0
	64	18,571	794	380	34	262	39.4	21.1	1.7	0
	128	19,005	1033	392	31	358	59.8	31.8	1.9	0
Proposed	8	652	204	99	72	1	18	5.9	0	3.4
	16	722	173	102	52	2.1	22	6.4	1	4.3
	32	1171	178	111	38	5.2	24.7	9.6	1	5.5
	64	1742	248	79	21	7.8	18	14.4	1	6.8
	128	6979	208	127	27	12.1	12.2	24.1	1	7.6

trx (c) and trx (s) denote the transaction in client and server nodes, respectively

We also implemented our scheme on the consolidate array while showing better performance.

There are a number of studies to reduce the overhead of logging by using flash and non-volatile memory (NVM), such as phase-change memory (PCM) and spin-transfer torque RAM (STT-RAM) [2,7,12,17,39]. FlashLogging [7] proposed a logging system design that exploits multiple USB flash drives for synchronous logging. This shows that USB flash drives are a good match for synchronous logging because of their low price and high efficiency.

PCMLogging [12] presents a logging scheme that exploits PCM devices for both data buffering and transaction logging in disk-based databases. Wang et al. [39] proposed a new NVM-based scalable distributed logging approach. They showed that a distributed log could provide considerable scalability.

NV-Logging [17] investigated the cost-effective use of NVM in transaction systems. It showed that using NVRAM only for the logging subsystem provides much higher transactions per dollar than replacing all disk storage with it. It also presents per-transaction logging to enable concurrent logging for multiple transactions and to reduce the performance bottleneck stemming from centralized log buffers.

Write-behind logging (WBL) [2] is a new protocol that is designed for a hybrid storage hierarchy with NVM and DRAM. We demonstrate that tailoring these algorithms for NVM not only improves the runtime performance of the DBMS, but also enables it to recover from failures nearly instantaneously. Using this logging method, the DBMS can flush the changes to the database before recording them in the log. By ordering writes to NVM correctly, the DBMS can guarantee that all transactions are durable and atomic. Our study is in line with these approaches [2,7,12,17,39] in terms of improving the logging performance. In contrast, they require additional devices that add to the total system cost.

There have been studies on improving the database system on flash-based SSDs. Lee et al. [23] present a quantitative and comparative analysis of magnetic disk and flash-based SSD in terms of performance impacts on database workloads. Their empirical study demonstrates that the low latency of flash-based SSD can drastically alleviate the log bottleneck at commit time.

Ouyang et al. [29] propose a storage primitive beyond a simple block I/O. The primitive, atomic-write, batches multiple I/O operations into a single logical group. The operations will persist as a whole or will be rolled back upon failure. The authors present an example of how a database system can benefit from atomic-write by modifying MySQL InnoDB.

DuraSSD [19] removes the double write buffer in MySQL InnoDB by exploiting the internal durable cache and modifying the firmware inside the SSD. X-FTL [20] proposes a transactional flash translation layer (FTL) for SQLite databases. X-FTL offloads the burden of guaranteeing the transactional atomicity from a host system to flash storage. To do this, X-FTL uses the copy-on-write strategy used in FTLs.

SHARE [28] is a flash storage interface that remaps the address mapping inside the flash storage. SHARE allows host-side database storage engines to achieve write atomicity without causing write amplification. Our study is in line with these approaches [19,20,28,29] in terms of reducing the overhead of logging on SSDs. In contrast, we focus on improving the performance on the commodity SSDs without modifying the FTL layer.

6 Conclusion

In this article, we have investigated transaction processing in key-value storage and we have shown that the existing flushing scheme is an obstacle to the performance of key-value storage. To address this issue, we have proposed an optimized flushing scheme that aggregates flushing data among multiple transactions into a single request on the fly and completes the request earlier to eliminate unnecessary `fsync()` calls and lock acquisition. Furthermore, we have integrated our scheme into a synchronous replication system for high reliability and availability. We have implemented the efficient flushing scheme in the WiredTiger storage engine with small modifications.

The experimental results show that the proposed scheme improves performance by up to 6.3× compared to the existing scheme in a single node configuration. In the replication configurations, the proposed scheme improves performance by up to 9.7× compared to the existing scheme. We anticipate that the implications and results of our study will provide directions for flushing on key-value storage systems. In future work, we will apply our optimization on other key-value storage systems and evaluate them by using different workloads and environments.

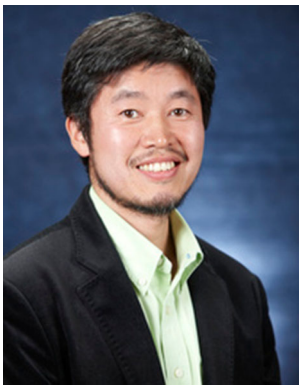
Acknowledgements This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2015M3C4A7065581, 2015M3C4A7065645). Prof. Han's work was partly supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2014R1A1A2055032).

References

1. Aguilera, M.K., Leners, J.B., Walfish, M.: Yesquel: scalable sql storage for web applications. In: Proceedings of the 25th Symposium on Operating Systems Principles, ACM, pp. 245–262 (2015)
2. Arulraj, J., Perron, M., Pavlo, A.: Write-behind logging. Proc. VLDB Endow. **10**(4), 337–348 (2016)

3. Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., Paleczny, M.: Workload analysis of a large-scale key-value store. ACM SIGMETRICS Perform. Eval. Rev. **40**, 53–64 (2012)
4. Banker, K.: MongoDB in Action. Manning Publications Co., Greenwich (2011)
5. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Currency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
6. Carlson, J.L.: Redis in Action. Manning Publications Co., Greenwich (2013)
7. Chen, S.: Flashlogging: exploiting flash devices for synchronous logging performance. In: SIGMOD, New York, NY, USA, SIGMOD'09, ACM, pp. 73–86 (2009)
8. Cloud, A.E.C.: Amazon web services (2011). Accessed on 9 November 2011
9. Felber, P., Pasin, M., Rivière, É., Schiavoni, V., Sutra, P., Coelho, F., Oliveira, R., Matos, M., Vilaça, R.: On the support of versioning in distributed key-value stores. In: 2014 IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS), IEEE, pp. 95–104 (2014)
10. Fitzpatrick, B.: Distributed caching with memcached. Linux J. **2004**(124), 5 (2004)
11. Fruhwirt, P., Kieseberg, P., Schrittwieser, S., Huber, M., Weippl, E.: Innodb database forensics: reconstructing data manipulation queries from redo logs. In: 2012 Seventh International Conference on Availability, Reliability and Security (ARES) (2012)
12. Gao, S., Xu, J., He, B., Choi, B., Hu, H.: Pcmlogging: reducing transaction logging overhead with pcm. In: 20th ACM International Conference on Information and Knowledge Management, New York, NY, USA, CIKM'11, ACM, pp. 2401–2404 (2011)
13. Goel, S., Buyya, R.: Data replication strategies in wide-area distributed systems. In: Enterprise Service Computing: From Concept to Deployment. IGI Global, pp. 211–241 (2007)
14. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Elsevier, San Francisco (1992)
15. Han, J., Haihong, E., Le, G., Du, J.: Survey on NoSQL database. In: 2011 6th International Conference on Pervasive Computing and Applications (ICPCA), IEEE, pp. 363–366 (2011)
16. Helland, P., et al. Group commit timers and high volume transaction systems. In: High Performance Transaction Systems. Springer, New York, pp. 301–329 (1989)
17. Huang, J., Schwan, K., Qureshi, M.K.: Nvram-aware logging in transaction systems. Proc. VLDB Endow. **8**(4), 389–400 (2014)
18. Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M., Ailamaki, A.: Aether: a scalable approach to logging. Proc. VLDB Endow. **3**(1–2), 681–692 (2010)
19. Kang, W.-H., Lee, S.-W., Moon, B., Kee, Y.-S., Oh, M.: Durable write cache in flash memory ssd for relational and nosql databases. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14 (2014)
20. Kang, W.-H., Lee, S.-W., Moon, B., Oh, G.-H., Min, C.: X-FTL: transactional FTL for SQLite databases. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, SIGMOD'13, ACM, pp. 97–108 (2013)
21. Kopytov, A.: Sysbench: a system performance benchmark. http://sysbench.sourceforge.net (2004)
22. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
23. Lee, S.-W., Moon, B., Park, C., Kim, J.-M., Kim, S.-W.: A case for flash memory ssd in enterprise database applications. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, ACM, pp. 1075–1086 (2008)
24. Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., Vivier, L., Bull S.A.S: A and viver, l. the new ext4 filesystem: current status and future plans. In: Ottawa Linux Symposium. http://ols.108.redhat.com/2007/Reprints/mathur-Reprint.pdf (2007)
25. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P.: Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. Database Syst. (TODS) **17**(1), 94–162 (1992)
26. MongoDB: https://www.mongodb.com/press/wired-tiger (2014)
27. NVM express: http://www.nvmexpress.org (2012)
28. Oh, G., Seo, C., Mayuram, R., Kee, Y.-S., Lee, S.-W.: SHARE interface in flash storage for relational and NoSQL databases. In: Proceedings of the 2016 International Conference on Management of Data, New York, NY, USA, SIGMOD'16, ACM, pp. 343–354 (2016)
29. Ouyang, X., Nellans, D., Wipfel, R., Flynn, D., Panda, D.K.: Beyond block I/O: rethinking traditional storage primitives. In: 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA), IEEE, pp. 301–311 (2011)
30. Ramakrishnan, R., Gehrke, J.: Database Management Systems. Osborne/McGraw-Hill, Berkeley (2000)
31. SAMSUNG 843Tn Data Center Series: http://www.samsung.com/semiconductor/global/file/insight/2015/08/PSG2014_2H_FINAL-1.pdf
32. Samsung: XS1715 Ultra-fast Enterprise Class. http://www.samsung.com/global/business/semiconductor/file/product/XS1715_ProdOverview_2014_1.pdf (2014)
33. Sivasubramanian, S.: Amazon dynamodb: a seamlessly scalable non-relational database service. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, ACM, pp. 729–730 (2012)
34. Son, Y., Kang, H., Han, H., Yeom, H.Y.: An empirical evaluation and analysis of the performance of nvm express solid state drive. Cluster Comput. **19**, 1–13 (2016)
35. Son, Y., Kang, H., Han, H., and Yeom, H.Y.: Improving performance of cloud key-value storage using flushing optimization. In: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W), pp. 42–47 (2016)
36. Son, Y., Yeom, H., Han, H.: Optimizing i/o operations in file systems for fast storage devices. IEEE Trans. Comput. **66**, 1071–1084 (2016)
37. Song, N.Y., Son, Y., Han, H., Yeom, H.Y.: Efficient memory-mapped i/o on fast storage device. ACM Trans. Storage **19:1**(19:27), 12–4 (2016)
38. Sumbaly, R., Kreps, J., Gao, L., Feinberg, A., Soman, C., Shah, S.: Serving large-scale batch computed data with project voldemort. In: Proceedings of the 10th USENIX Conference on File and Storage Technologies, USENIX Association, p. 18 (2012)
39. Wang, T., Johnson, R.: Scalable logging through emerging non-volatile memory. Proc. VLDB Endow. **7**, 10 (2014)
40. WiredTiger: http://www.wiredtiger.com (2014)

Yongseok Son received his B.S. degree in Information and Computer Engineering from Ajou University in 2010, and the M.S. degree in Department of Intelligent Convergence Systems at Seoul National University in 2012. Currently, he is a Ph.D. candidate in Computer Science and Engineering at Seoul National University. His research interests are distributed systems, operating systems, and database systems.



Heon Young Yeom is a Professor with the School of Computer Science and Engineering, Seoul National University. He received B.S. degree in Computer Science from Seoul National University in 1984 and his M.S. and Ph.D. degrees in Computer Science from Texas A&M University in 1986 and 1992 respectively. From 1986 to 1990, he worked with Texas Transportation Institute as a Systems Analyst, and from 1992 to 1993, he was with Samsung Data Systems as a Research Scientist. He joined the Department of Computer Science, Seoul National University in 1993, where he currently teaches and researches on distributed systems and transaction processing.

Hyuck Han received his B.S., M.S., and Ph.D. degrees in Computer Science and Engineering from Seoul National University, Seoul, Korea, in 2003, 2006, and 2011, respectively. Currently, he is an assistant professor with the School of Dongduk Women's University. His research interests are distributed computing systems and algorithms.