IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 6, 2020, accepted March 6, 2020, date of publication April 3, 2020, date of current version June 9, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2985407

TLSM: Tiered Log-Structured Merge-Tree Utilizing Non-Volatile Memory

JIHWAN LEE, WON GI CHOI, DOYOUNG KIM, HANSEUNG SUNG, AND SANGHYUN PARK, (Member, IEEE)
Department of Computer Science, Yonsei University, Seoul 03722, South Korea

Corresponding author: Sanghyun Park (sanghyun@yonsei.ac.kr)

This work was supported by the Institute for Information and Communications Technology Promotion (IITP) funded by the Korea Government, Ministry of Science and ICT (MSIT), through the SW Starlab Research and Development of the High Performance In-Memory Distributed Database Management System Based on Flash Memory Storage in the IoT Environment, under Grant IITP-2017-0-00477.

ABSTRACT Log-Structured Merge-tree (LSM-tree) organizes write-friendly and hierarchical structure, which leads to inevitable disk I/O from data compaction occuring between layers. Previous research tried to reduce write bottlenecks of LSM-tree by using non-volatile memory (NVM) to LSM-tree, but write amplification on compaction does not resolved. In this paper, we present tiered LSM (TLSM), a persistent LSM-based key-value database tiering NVM, which is regarded as fast storage device. Our design aims to exploit multi-layered LSM system utilizing NVM and reduce redundant cost on compaction. We address 4 key points: (1) analysis of disk I/O overhead on current LSM-tree, (2) novel design of TLSM preserving the concept of LSM-tree and utilizing NVM, (3) persistent and byte-addressable data compaction minimizing disk write, and (4) tiering policy for increasing usability to TLSM. To implement TLSM-tree, we extended LevelDB, which is a simple LSM-based key-value store. On micro-benchmark DB Bench, the write and read latencies of TLSM are improved up to 34% and 57%, respectively, compared to those of LevelDB-NVM. In evaluation of YCSB, TLSM also reduces the write and read latencies of existing LevelDB up to 25%, 33%, respectively. Furthermore, frequency of write stall and total amount of data write are reduced significantly in TLSM.

INDEX TERMS Database recovery, key-value, store, log-structured merge-tree, non-volatile, memory.

I. INTRODUCTION

With the emergence of various and enormous data from the web, Internet-of-Things (IoT), and mobile systems, key-value stores play a major role in database systems today. Key-value stores have unique features such as fast write, flexibility for data size, and storing various data types. To provide these characteristics, log-structured merge-tree (LSM-tree) can be used for the general data structure of key-value stores.

LSM-tree [1] is a write-friendly data structure that organizes hierarchy using log-structured write, write buffering, and merge operation. The well-known LSM-based databases are Bigtable [2], Apache HBase [3], Cassandra [4], LevelDB [5], and RocksDB [6]. LSM-tree consists of components in both memory and disk. These components construct a hierarchical storing structure, requiring data compaction to reduce the cost of read, write, and space. However, it is inevitable in LSM for read-write-space amplifications to occur, which is well-known to be a trade-off. Therefore, many methodologies to alleviate the cost of these amplifications on LSM-tree have been especially researched.

Non-volatile memory (NVM) technology such as phase-change memory (PCM) is being considered as fast storage device having features from both main memory such as DRAM and secondary storage such as solid state drive (SSD) and hard disk drive (HDD). Because database systems with NVM have opportunities to significantly improve overall throughput, many novel designs applying NVM have been proposed. With its characteristics such as byte-addressability, lower latency, higher density, and durability, NVM is expected to be the next-evolution secondary storage device.

LSM-tree currently has many chances of exploiting NVM, therefore, LSM-tree with NVM has been researched in many ways. First, Hybrid LSM-based memory system that use both DRAM and NVM has been investigated [7]. Reducing

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

logging overhead was attempted by implementing persistent skip list. Second, LSM-tree caching using NVM, specifically its capability to decrease the cost of cache miss, has been studied [8]. To apply NVM caching method, file format is constructed in the persistent memory pool, called *pmemenv*. Third, single-level architecture of LSM-tree in disk has been proposed [9]. B+-tree index was constructed in NVM, which can provide fast access to on-disk data. Fourth, *NVM-Rocks* [10] implemented Rocksdb based on persistent structure, introducing some features benefiting from NVM-aware system. Unlike these researches, our design is the first approach that does not replace DRAM with NVM, but instead constructs tiered system using NVM as a middle flexible layer between main memory and block-based device. Furthermore, we focus on preserving the concept of LSM-tree and showing appropriate use of NVM on LSM-based system.

We analyze LSM-tree for appropriate utilization of NVM in experiments. LSM-based systems currently have two issues. (1) Write stall: There are slow-down conditions when write buffer is full and data should be flushed into disk. To maintain system conditions, an LSM-based system intentionally delays flushing data, called write stall. The write stall accumulates exponentially as the amount of data increases, affecting overall system performance. Thus, it is required to minimize write stall in flush operation. (2) Inevitable write amplification: By hierarchical storing structure, the data files at the upper level on disk are rewritten to new files at the next level during data compaction, and then the old data files at the upper level are deleted within a few compaction times after they are created. Because it is essential that data files are compacted to lower level through upper level, this process occurs frequently and causes repetitive but inevitable disk I/O. Based on two analyses, we check feasibility of NVM as a middle layer and design novel LSM structure to mitigate these issues.

In this paper, we propose tiered LSM-tree (TLSM), a persistent LSM-based key-value system that preserves the features of LSM and utilizes NVM as additional middle tier. Our design aims to achieve practical LSM-tree structure, exploit the byte-addressability of NVM, and guarantee persistence of the system. This proposal has four main contributions.

1) Analysis of inevitable repetitive disk I/O patterns: We aggregate amount of write stall and average lifetime of data files. Write stall accumulates exponentially according to the database size. Furthermore, the data files at the upper level are quickly deleted during data compaction, which increases I/O overhead. In our design, the amount of write from stall and compaction drastically decreases as data size increases, compared to current LSM-tree.
2) Architecture of TLSM preserving LSM-base and using NVM efficiently: We present novel LSM-based architecture conducting sequential-write, failure-atomic operations, and minimal data copy. Our design uses persistent buffer for sequential-write, copies pointer for guaranteeing data persistence, and uses byte-addressable data structure. It not only preserves the concept of LSM, but also is also optimal use of NVM to LSM.
3) Minimization of write amplification through persistent, byte-addressable, and lightweight data compaction: We implement compaction method to eliminate redundant data copy. The method also ensures persistence and implements recovery.
4) Tiering policy for increasing usability of TLSM: Practical use of TLSM is considered by adjusting tiering options according to the available resources on the system. Because the capacity of NVM for the price is far lower than those of block-based storage, TLSM with tiering policy is flexible for covering various environments.

TLSM-tree is extended from LevelDB and we emulate NVM using Persistent Memory Development Toolkit (PMDK), provided by Intel. We evaluate TLSM-tree using popular benchmarks DB Bench and Yahoo! Cloud Serving Benchmark (YCSB). In evaluations of DB Bench, TLSM-tree achieves higher throughput than LevelDB-NVM by reducing write latency up to 34% and read latency up to 57%. In evaluations of YCSB, TLSM also shows about 18% to 33% lower latencies than that of LevelDB-NVM in all workloads. In addition, the total amount of data write from compaction and the accumulation of write stall are reduced significantly. Thus, our proposal can enhance the read throughput as well as write efficiency.

The rest of the paper is organized as follows: In Section II, we describe the required background and cover related studies. In Section III, we analyze issues with current LSM-based systems using experimental statistics. In Section IV, we present the design of TLSM-tree with tiering policy and recovery method. In Section V, we evaluate TLSM-tree for latency on methodologies, tiering options, novel compaction, write efficiency, and level threshold. Finally, we conclude the paper in Section VI.

II. BACKGROUND AND RELATED WORKS

In this section, we provide a background on LSM-tree and LSM-based stores LevelDB and RocksDB. We then present characteristics of NVM and NVM emulation tool, PMDK. Furthermore, we review some recent studies about LSM-tree and NVM.

A. LSM-TREE

LSM-tree is derived from log-structured file-system [11], which stores all data sequentially by version. LSM-tree focuses to improve the performance of data insertion in database systems where fast levels like main memory and slow levels like block-based storage coexist. The benefit of LSM-tree is the write-friendly structure that inserts data into write buffer in memory and batch writes to disk. Thus, the data insertion has low write latency and if the in-memory write buffer is full, data persistence is guaranteed by flushing

data to storage, such as a disk. LSM-tree keeps the inserted data in memory buffer as much as possible and accesses from the latest data when seeking. Merge processes are also required to reorder key-value entries and update overlapped keys between adjacent components called layers. The LSM-tree can be designed differently depending on the characteristics of the system and can also be designed with various data structures and file formats.

B. LSM-BASED KEY-VALUE STORE

Two examples of LSM-based key-value stores are LevelDB and RocksDB. We describe some features of each store.

1) LEVEL DB

LevelDB is an open-source key-value database project developed by Google. Implementing minimum functionalities of LSM-tree, LevelDB is a simple, fast, and lighweight database. LevelDB is also an embedded database that does not use a server process. It operates on a single main thread and schedules background thread only when data flush and compaction are required. With these features, various LSM-based key-value databases such as HyperLevelDB and RocksDB were derived from LevelDB.

The LSM-tree in LevelDB is a persistent structure in which the memory area and the disk area form a hierarchy. In memory, a memtable implemented by skip list stores arbitrary key-value entries sorted in key order. On the other hand, the data are stored as sorted-string table (SST) file format in disk. The SST file contains several blocks, consisting of a data block for storing data, an index block for indexing the location of data block, and a footer block for handling the location of index blocks. Optionally, if a filter is used to improve read performance, a meta block and a meta-index block may be included in the SST file. The disk area is divided into abstract levels such as $L_0, L_1, L_2, \ldots, L_k$, and the SST file is included in a specific level. The maximum capacity of level increases as the level becomes lower. Fig. 1 illustrates an overview of LSM-tree in LevelDB and some operations. Data insertion is performed by inserting a key-value entry into memtable. Memtable is converted to immutable memtable when it exceeds the size threshold, then the immutable memtable is flushed to disk as the SST file at the top level (L_0) by background thread. In order to guarantee data persistence of memtable, LevelDB use write-ahead logging (WAL), leading to additional disk I/O. Fig. 2 shows on-disk data compaction process. If the maximum capacity of L_i is exceeded, (1) select one of the SST files and seek the SST files in L_{i+1} having the key that overlaps with the key range of this file, (2) load the files into memory, (3) then perform the *merge sort* operation. (4) As a result, newly created SST files are involved in L_{i+1}, and (5) garbage collection is finally conducted for the old SST files. Thus, current on-disk compaction requires many inevitable disk I/Os, except flushing memtable, in Steps (2) and (4). This phenomenon is called write amplification, which is a common issue with LSM-tree.

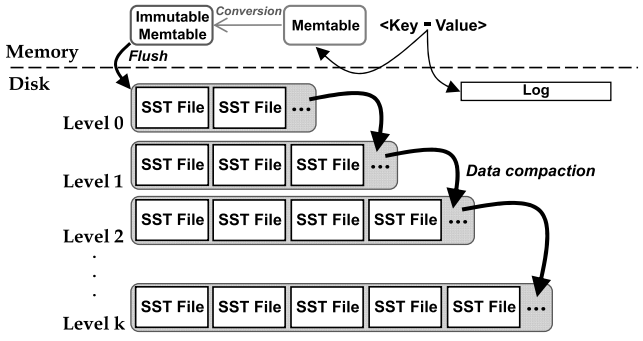


FIGURE 1. An overview of LSM-tree in LevelDB.

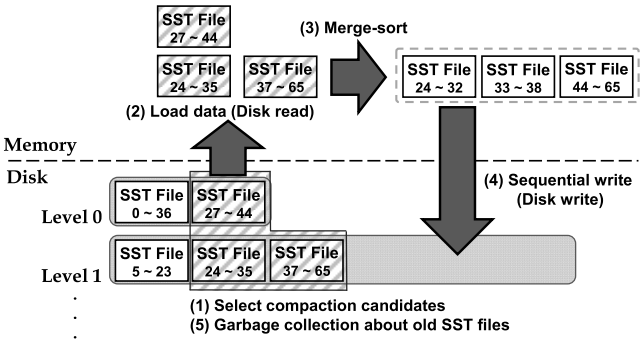


FIGURE 2. Process for data compaction (1) – (5).

Because LSM-tree organizes hierarchical storing structure, the lookup is first conducted in the memtable, and then in immutable memtable, and if memtable and immutable memtable do not have the desired data, the SST files are sequentially searched from the upper level to the lower level on the disk. At each level, the SST file containing the required key range is searched first, then the location of data block containing the key range is searched in the index block. If the corresponding data block is read, it is checked for whether the lookup key exists through the iteration. Therefore, many block reads are required for the lookup operation.

2) ROCKS DB

RocksDB is a key-value store based on LevelDB. RocksDB has targeted SSD as storage device and applies several techniques to exploit the internal parallelism of SSD. So, RocksDB implement high-performance techniques for practical use as well as multi-threading during data compaction [12]. For example, RocksDB provides diverse memtables, which include not only skip list but also vector, hash linked list, hash skip list, and cuckoo, configures compaction algorithm to choose candidate SST files, and implements L_0-compaction between the SST files at the top level L_0. In addition, detailed configuration is required according to many options, such as multiplier for determining the capacity of each level and individual compression for applying each level.

In this study, we developed a TLSM-tree by extending LevelDB. This is because the simplicity and lightness of LevelDB

are likely to allow the development of new LSM-based store in the future. Furthermore, LevelDB is used in commercial deployments. With regard to RocksDB, a recent study [13] has revealed that some options affect system performance considerably. We considered that the throughput of RocksDB can vary greatly depending on the configuration.

New designs and methodologies to decrease overhead on LSM-tree have been researched. We classify these designes and methodologies under three groups: LSM-based architecture, data compaction mechanism on LSM-tree, and practical techniques on LSM-tree.

In terms of LSM-based architecture, Lu *et al.* [14] presented *WiscKey*, which is a new separating structure of LSM-tree for exploiting the parallelism of SSD. They separated the value from the key-value entry. While the value is inserted into a value log, the key with value offset pair is inserted into the existing LSM-tree. Only the key that is much smaller than value size is used during data compaction; hence, this mechanism alleviates write amplification. Furthermore, they maintained minimized value-log by implementing garbage-collection method. Wu *et al.* [15] designed *LSM-trie*, which is a trie structure-based LSM-tree with hash. They targeted the cases that generated small but several key-value data such as SNS data. For the read throughput, they applied clustering method to bloom filter for more efficient data accessing.

In terms of data compaction mechanism, Pan *et al.* [16] proposed the lazy compaction methodology, *dCompaction*, to reduce compaction overhead. *dCompaction* means delayed compaction about physical write. They categorized real compaction, which is the said existing compaction and virtual compaction, which is the new method to reduce amplification. During virtual compaction, they delayed write to new block-based file and stored parent file numbers with some metadata to seek real data from parent file numbers. They, moreover, defined trigger conditions of real or virtual compaction to adjust the ratio of compaction. Similarly, Yao *et al.* [17] presented *Light-weight compaction*, which merges and sorts only metadata among compaction candidate files. Zhang *et al.* [18] dealt with the computation overhead of data compaction. They analyzed the compaction procedure one by one, then applied the parallelism of CPU and I/O devices to *pipelined compaction*.

In terms of practical techniques on LSM-tree, Dong *et al.* [19] optimized the use of storage space by implementing methods such as effective compression and, dynamic level size setting. In addition, they analyzed the patterns of each level, which include the amount of disk write, the number of data compaction, and the location of desired data during read operation in RocksDB. Ouaknine *et al.* [13] presented RocksDB combined with Redis, which is a popular in-memory database, in order to cope with growing data size in Redis. Focusing especially, on RocksDB, because there are so many techniques and configurations, they showed major parameters affecting performance on RocksDB. Sears and Ramakrishnan [20] improved the merging process by implementing the merge scheduler called *spring and gear scheduler* and applied proper bloom filter based on double hashing.

C. NON-VOLATILE MEMORY

Non-Volatile Memory (NVM) is considered as next-generation storage. Non-volatile, byte-addressability, low-latency, high-capacity, and high-density NVM plays an important role in guaranteeing durability of data. Unlike SSD and HDD, which are block-based storage devices, NVM is byte-addressable. In other words, data serialization–deserialization cost between blocks in file and bytes in memory is not required. In addition, because block-based access is not used, unnecessary load doesn't be operated. Although the write and read latencies of NVM are slightly higher than that of DRAM, NVM shows faster read-write than that of SSD. Through the development of NVM such as PCM, existing database systems optimized for block-based storage can be implemented as NVM-aware systems.

In order for data on NVM to be consistent, system should use additional operations such as *clflush* and *mfence*. *clflush* is an instruction for flushing cache line to memory. It ensures that data are stored in NVM completely. *mfence* is the memory barrier instruction for preventing CPU from reordering instructions. In NVM-aware systems, these instructions should be added between operations despite the additional cost because it is important to execute instructions sequentially.

Because NVM has a chance to be applied to several domains, there are so many researches related to database systems with NVM. Kim *et al.* [21] introduced WAL on NVRAM, *NVWAL*, guaranteeing failure atomicity and durability. *NVWAL* can reduce overhead of log-management through byte-granularity differential logging, user-level heap manager, and transaction-aware lazy synchronization. Izraelevitz *et al.* [22] presented a new failure-atomic process, *JUSTDO* logging, for simplicity of logging and fast recovery. They utilized transactional memory to support program-defined failure-atomic sections (FASEs). The Logging mechanism within FASE can improve performance and preserve data integrity. Oh *et al.* [23] proposed SQLite with PCM for optimizing mobile environment that executes small transaction but causes many random writes. To reduce write amplification, they applied per-page logging and physiological loging to PCM. Yang *et al.* [24] presented persistent B+-tree structure, called *NV-tree*. They constructed internal nodes in DRAM and leaf nodes in NVM. Leaf nodes are unsorted and conduct append-only-update. When *NV-tree* recovers from failure, the tree should reconstruct internal nodes. Oukid *et al.* [25] presented similar persistent B+-tree design, called *FP-tree*. However, to achieve better performance, *FP-tree* uses fingerprinting techniques, which use 1-byte hashes of leaf keys. Lee *et al.* [26] proposed persistent write optimal radix tree, *WORT*. For data consistency, they constructed path compression header that has depth information, which ensures failure atomicity when

node split occurs. Furthermore, an adaptive radix tree that uses pre-defined node size can guarantee consistency from the *copy-on-write* method.

D. PMDK

Persistent Memory Development Kit (PMDK) [27] is a persistent memory-usage library developed by Intel. It is implemented to allocate the persistent memory area using the file system at the kernel level. To emulate NVM, PMDK makes a part of the DRAM area into a file system and maps the corresponding area to virtual memory using Direct Access (DAX). Three key concepts are needed for PMDK to access persistent memory areas: (1) persistent memory pool, which is an area allocated by PMDK, (2) root structure, which is created in the persistent memory pool and represents the persistent area to be used, and (3) persistent pointer, which acts as a pointer to find allocated area including the root structure in a specific persistent memory pool. In the host program, the root structure is designed by the user, who gets a persistent pointer to access the root structure in the pool. When the user allocates or de-allocates some NVM area, PMDK manages the persistent area automatically. Furthermore, PMDK ensures data persistence by implementing persist function, *persist*, substituting for *clflush*, and *mfence* operations. From these features, some past studies already emulated NVM using PMDK [8], [9].

E. LSM-TREE USING NVM

LSM-tree is currently optimized for using memory and block-based storage. However, LSM-tree is not implemented to exploit NVM for methods such as WAL logging, caching, in-memory buffer, and on-disk format. Thus, there are good chances of improving the performance by utilizing NVM. Kannan *et al.* [7] designed *NoveLSM* system based on persistent skip list and parallel read operation. They showed that current LSM-based systems do not fully exploit the characteristics of NVM in random write–read operations. They also showed the cost breakup of write and read operations, then demonstrated that the compaction time and data serialization–deserialization cost, respectively account for large portion of these operation costs. To exploit the byte-addressability of NVM, they constructed a hybrid structure that creates memtable and immutable memtable on both DRAM and NVM. *NoveLSM* also reduced recovery cost by ensuring in-place updates. Lersch *et al.* [8] proposed an efficient cache method for LSM-tree with NVM. In order to reduce the cost of cache miss, they applied the 2Q cache policy, which requires a little memory usage, to NVM. In uniform and skew experiments, they showed that the LSM caching requires only some blocks repeatedly. Kaiyrakhmet *et al.* [9] proposed *SLM-DB*, which is the single-level LSM-tree with NVM. They stored data into a single layer in disk, then constructed B+-tree index in NVM. Thus, the model can provide quick access to on-disk data. Furthermore, they implemented selective compaction method in order to collect obsolete key-value pairs. *NVM-Rocks* [10] was developed to provide NVM-aware functionality to RocksDB. *NVM-Rocks* has some persistent methods such as allocator, memtable, and multi-tiered cache using NVM. The developers of *NVM-Rocks* mentioned the need for redesigning on-disk data structure with NVM to optimize NVM-aware LSM system. Our design can thus be a contributive part to current design principles of LSM-based systems. We have devised TLSM by using NVM as middle layer between main memory and block-based storage.

III. MOTIVATION

LSM-tree performs two operations: (1) flush across memory and disk, and (2) compaction across levels on disk. In this section, we analyze two practical issues through some experiments. As a result, we try to check the validity of utilizing NVM in the LSM-tree.

A. ACCUMULATION OF WRITE STALL

When flush occurs in the system across memory and disk, the immutable memtable is converted to a single SST file in L_0. However, when a large amount of data is quickly inserted, the system intentionally delays data flush to the disk in order to balance the speed of flush and maintain L_0 capacity threshold. This is called write stall. We experimented with SSD via DB Bench to see how much the write delay is accumulated. As shown in Table 1, the write stall increases with the amount of data. Only 8 GB of data is inserted but about 15 minutes of write delays occur in the system. If more data are inserted continuously, write stall may accumulate exponentially. The result shows that the system suffered from too much balancing overhead. Therefore, it is required to reduce flush delay using any technique. In this respect, our proposal is expected to serve as a buffer between the main memory and the storage by placing NVM as the middle layer. NVM can improve flush performance, because NVM latency is greatly lower than that of other storage. Furthermore, byte-addressability of NVM may accelerate on-disk compaction. Thus, an appropriate format of on-disk data to increase the speed of flush, and byte-addressable, persistent method to accelerate compaction at L_0 are required.

TABLE 1. Accumulation of write stall.

Data size	Accumulated write stalls on SSD(us)
2 GB	61,902,809
4 GB	310,872,755
6 GB	569,200,925
8 GB	901,254,333

B. AVERAGE LIFETIME OF SST FILES

The characteristics of each level of the LSM-tree are distinctly different [19]. Compaction cost and the amounts of disk write are concentrated in a specific level. Hierarchical

TABLE 2. The average lifetime of SST files.

Level	Average lifetime of SST files	The number of compacted files	The ratio of compacted files	The amount of write during compaction (MB)
L0	4.498	35434	99.989%	113230
L1	10.679	118038	99.986%	196361
L2	51.163	379120	99.961%	511559
L3	403.795	481843	99.728%	650031
L4	4340.665	339725	96.629%	599837

storing structure also causes the LSM-tree to gradually accumulate data from higher level to lower level. About these features, we measured the lifetime of the SST file as the value of when it is deleted with respect to when it was created. The lifetime of SST file is that *how long the file exists based on the number of compaction times at the corresponding level*. If the SST file at a specific level is not deleted after much compaction, the lifetime is high. We experimented with inserting 100 GB of data into the lower level (assuming L_6 or lower) and carried out random-write and random-read benchmarks sequentially.

Table 2 shows the results by level, which implies three information: (1) for the average lifetime of SST files, SST files at upper level (assuming L_0 to L_2 or L_3) are generated then deleted in a short time by data compaction, (2) for the number of compacted files and the ratio of compacted files, the proportion of compacted files at the upper level is not small due to hierarchical structure of LSM-tree, and (3) the amount of write during compaction at upper level is also not insignificant, which involves redundant, but unavoidable, disk I/O. Although we insert enough data into LevelDB, the number of compacted files and the amount of write in L_4 are lower than in L_3. If we insert more data, average lifetime, the number of compacted files, and the amount of write during compaction at L_4 become far higher than those at the other levels. However, we focused on the lifetime of the SST files, which is so high at L_4. Thus, we consider that utilizing NVM can reduce redundant disk I/O at upper levels, while has less effect on compaction cost at L_4. NVM may resolve write amplification of existing system through byte-addressable but persistent compaction method.

IV. THE DESIGN OF TLSM

In this section, we explain in detail the implementation of TLSM-tree step by step for adequate structure. Furthermore, we define some tiering policies for efficient usage of multiple devices and describe the recovery logic for unexpected failures.

A. PERSISTENT DATA STRUCTURE

In order to exploit byte-addressability of NVM, block-based format like SST file should be replaced with data structure. According to [7], the cost of SST serialization– deserialization between memory and block device increases as the value size grows in LSM. Thus, we implement persistent skip list, which replace each SST file. Note that our design aims to substitute for the SST file, while previous researches focused on persistent write buffer in memory. Persistent data structure is implemented to skip list but can be constructed as an ordered data structure because read, write, and compaction operations are performed based on key-value entry iterators. Each persistent skip list has its own number, called a skip list ID.

Persistent skip list includes key length, key, value length, and value for each node. When looking up the data, the algorithm starts searching from the highest level of the persistent skip list, then compares the key to find the desired data. On flush operation, an iterator, which is sorted data by key, is used, so there is no need for additional reordering cost when copying data in memory buffer into the persistent skip list. During data compaction, we allocate a new persistent skip list and generate an iterator from the lowest list of an old skip list, then insert the data from iterator up to the capacity of the skip list.

B. BYTE-ADDRESSABLE COMPACTION

We devised a byte-addressable compaction process that can maintain persistence of data and reduce high cost of existing compaction. When data compaction occurs at disk level, byte-addressable compaction is conducted only with the pointing operations for the fields of each key and value allocated in the node. It does not require additional data writes about the key, length of key, value, and length of value. The procedure also does not require allocation of new area. Byte-addressable compaction just points to the corresponding area on sequential nodes by the new skip list until the size threshold of the skip list, then finally adds the null node for the recovery. Persistent memory devices are expected to guarantee failure-atomic writes in 8-byte units [25], [28], [29], [30]. Because the size of pointer variables is 4 bytes in 32-bit system and 8 bytes in 64-bit system, failure-atomic operation on compaction is valid. Data copy is only required when data in write buffer is flushed into skip list at L_0. Thus, byte-addressable compaction reuses already written but valid data while current compaction rewrites same key-value data on newly created SST file. This mechanism can significantly alleviate redundant writes for on-disk data, compared to existing compaction. Detailed procedure of byte-addressable compaction combined with additional method is described in Section IV-C.

The key-value entries for the old version or to be deleted are sequentially freed after finishing the compaction process.

Algorithm 1: Flush

Input:
memtableIter: key-value iterator from immutable memtable
Output:
fileMetadata: metadata including new output number, smallest/largest key, and filesize

```
    /* Step 1. Iteration about all data in memtable */
1:  buf ← [] // append all KV-pair, then memcpy at a time
2:  pBuf ← create new persistent buffer
3:  startOffset ← get new start offset from persistent
    buffer
4:  pos ← 0 // relative position from start_offset
5:  pSkiplist ← create new persistent skip list with new ID
    /* Step 2. Iteration about all data in memtable */
6:  for KV − pair in memtableIter do
7:      (key, value) ← Encode(KV-pair)
8:      entryLength ← key.length + value.length
9:      Append(buf ← key, value)
         /* Insertion into skip list node through the relative
         position of pBuf */
10:     pSkiplist.current ← Node(startOffset + pos,
         pSkiplist.prev)
11:     Persist(pSkiplist.current)
12:     pSkiplist.current.next ← pSkiplist.prev.next
13:     Persist(pSkiplist.current.next)
14:     pSkiplist.prev.next ← pSkiplist.current
15:     Persist(pSkiplist.prev.next)
16:     pSkiplist.current ← pSkiplist.current.next
17:     Persist(pSkiplist.current)
18:     Update(pos ← pos + entryLength)
19: end
    /* Step 3. Finally, add null node for recovery */
20: pSkiplist.current ← NullNode()
21: Persist(pSkiplist.current)
    /* Step 4. For persistent buffer, bulk copy */
22: Copy(pBuf ← buf until buf.size)
23: Persist(pBuf)
24: Update(fileMetadata)
```

The last null node for each skip list indicates the complete data structure from compaction. Thus, the last null node is used when recovering the persistent skip list in compaction failure, which is discussed in detail in Section IV-E.

C. SEQUENTIAL COPY WITH LIGHTWEIGHT DATA STRUCTURE

However, the proposed compaction method in Section IV-B does not take advantage of the batch write of LSM-tree. Sequential access in memory becomes faster than random access through CPU caching and prefetching. In the byte-addressable compaction scheme, data copy occurs only for L_0 skip list. However, random copies to NVM are required for the key, key length, value, and value length in the areas allocated for each node. These steps require numerous random accesses to each area in the node, so, we needed to modify the method to preserve the basic concept of the LSM-tree.

Fig. 3 illustrates an overview of TLSM-tree with sequential copy. TLSM uses persistent buffer to perform sequential copy from batch and lightens the persistent skip list by storing only the buffer pointer to the offset of persistent buffer. The content of persistent buffer is similar to the SST file, except for index block and footer block. Persistent buffer is appended with the key and value encoded for each length, sequentially. We use the relative position of allocated area and copy data in write buffer into skip list at one time.Algorithm 1 explains how to flush write buffer into skip list with persistent buffer. First, TLSM prepares some objects such as buffer in host, new persistent buffer, start offset from persistent buffer, relative position and new persistent skip list (lines 1–5). Then, for the iterator of immutable memtable, each key and value with encoding length is appended to the buffer (lines 7–9). Using the length of key-value pair, TLSM can get the relative position of key-value data in the buffer. TLSM inserts the offset into the current node and conducts list pointing with *persist* functions (lines 10–17). Note that these *persist* functions guarantee data persistence, like with using a combination of *clflush* and *mfence* operations. Then, TLSM updates relative position in the buffer to prepare the next insertion of key-value pair (line 18). After iteration, TLSM inserts the last null node (lines 20–21). Finally, TLSM operates memory copy from the buffer in host program to persistent buffer in NVM and updates metadata about the new output (lines 22–24). Note that because TLSM uses WAL logging by default, system failure during flush can be recovered based on on-disk log.

The lookup operation is fulfilled as an existing procedure. Following the hierarchical order, it searches memtable, immutable memtable, and skip list at each level, sequentially. Although unlike current SST file format, the persistent buffer does not contain index block and footer, persistent skip list serves for indexing using skip-based searching. Thus, we expected to enhance the performance of lookup by skip searching and removing the cost of data serialization. The lookup of raw data in skip list is performed by obtaining the persistent buffer offset of the key-value entry stored in the node of the skip list, then getting raw key-value data from that offset and comparing the keys. The algorithm may require additional read operation compared to the method in Section IV-B.

Byte-addressable compaction is performed through the pointing operation on the newly created skip list. For the management of NVM area, TLSM allocates the area of persistent buffer dynamically depending on batch size and copies the batch of data into the area at one time. Note that LSM-tree has the version, which is used to capture data at a specific moment. TLSM-tree conducts de-allocation operations lazily. Because the reference count of each version is checked internally, if the reference count is 0, the system frees the

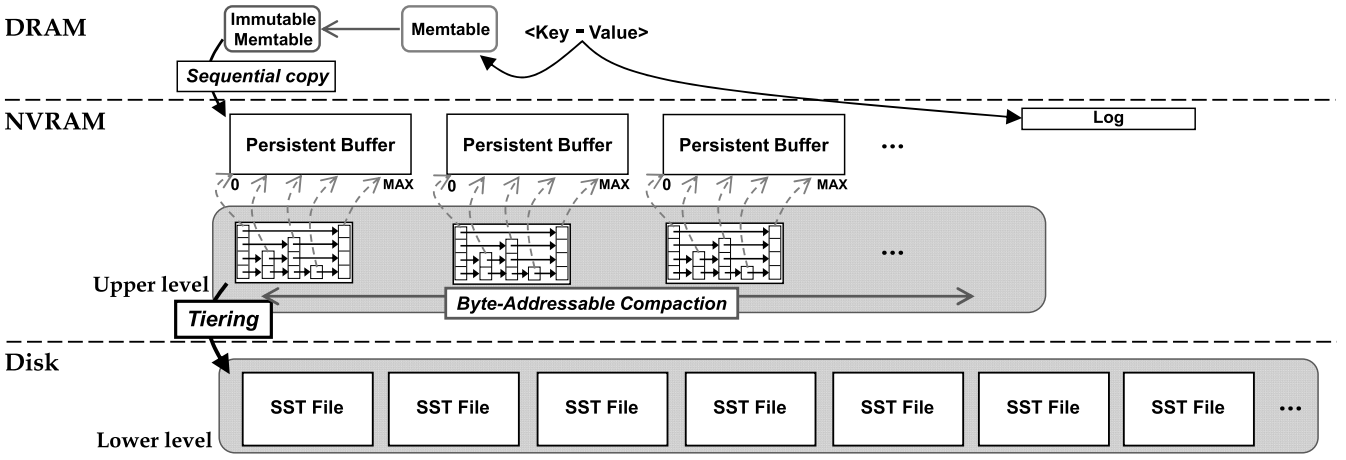


FIGURE 3. An overview of TLSM.

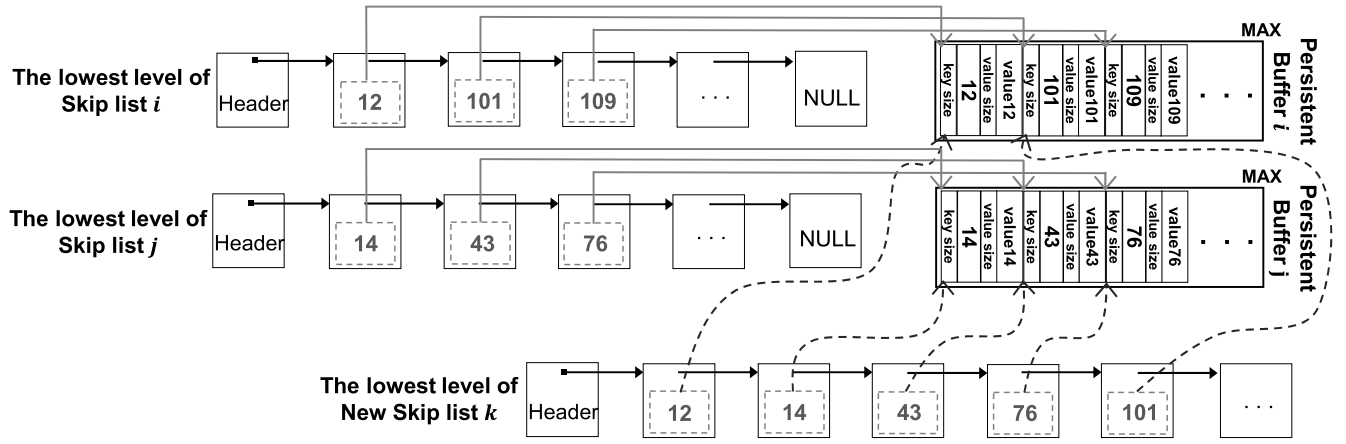


FIGURE 4. An example of byte-addressable compaction with persistent buffer between L_0 and L_1.

corresponding skip list and the obsolete buffer area together. Fig. 4 illustrates an example of byte-addressable compaction with persistent buffer between L_0 and L_1. Before performing compaction, TLSM creates iterators from the lowest level of persistent skip lists. In this example, each persistent skip list to be compacted has persistent buffer because it is sequentially copied from in-memory write buffer. By *merge sort* operation, each node of newly created skip list points to the corresponding buffer offset, which leads to removing redundant writes. After finishing compaction, obsolete skip lists at L_0 are freed. Thus, byte-addressable compaction can resolve fundamental issue on current compaction mechanism. Algorithm 2 explains how to implement in detail byte-addressable compaction. TLSM uses a merge iterator from iterators of skip lists to be compacted. Each iterator is created from the lowest list of persistent skip list. With regard to merge iterator, each key-value pair has a buffer pointer that points to the corresponding entry in persistent buffer. During iteration, TLSM first checks whether the buffer pointer is valid (lines 2–5). If the buffer pointer is not valid, the algorithm suggests that compaction is finished, or that input iterator on compaction is in invalid status. This is handled after the end of iteration. Next, TLSM checks whether output skip list is null (lines 6–8). If so, TLSM allocates new skip list and inserts metadata about input IDs of compaction into header of the skip list. This metadata is used for resuming compaction when the system recovers from failure. Then, TLSM inserts only the buffer offset into current node and conducts list pointing with *persist* functions (lines 9–16). In the insertion of new node, this method, compared to flush operation, is different only in the use of the buffer offset. Finally, TLSM checks whether the current output is full (lines 17–21). If so, TLSM inserts null node into the last location, inserts metadata about newly created output into the list of metadata, and resets current output to null. As a result, byte-addressable compaction can reduce unnecessary writes and also guarantee data consistency by pointing operations.

D. TIERING POLICY

We define tiering policies of TLSM-tree. Because the capacity of NVM for the price is far lower than block-based storage such as SSD, we consider the systems that have various resource usages from utilizing multiple devices such as DRAM, SSD, and NVM. As shown in Fig. 3, tiering means that the data in the persistent buffer is flushed to the block device as an SST file when TLSM meets specific conditions. Of course, tiering may require additional disk writes according to tiering policy. However, tiering is the best method that

Algorithm 2: Byte-Addressable Compaction

Input:
mergeIter: merge and sorted iterater from skip list IDs to be compacted
Output:
fileMetadataList: list of metadata including new output number, smallest/largest key, and filesize

```
for KV − pair in mergeIter do
    /* Step 1. Get key-value entry by buffer pointer*/
    if not KV − pair.isBufferPointerValid() then
        break // end of iteration or invalid status
    else
        bufPtr ← KV-pair.GetBufferPointer()
    /* Step 2. Check whether the persistent skip list is
    valid */
    if pSkiplist is Null then
        pSkiplist ← CreatePersistentSkiplist(new ID)
        Insert(pSkiplist.metadata ← inputIDs) // for
        recovery
    /* Step 3. Failure-atomic pointing operation with
    persistent buffer */
    pSkiplist.current ← Node(bufPtr, pSkiplist.prev)
    Persist(pSkiplist.current)
    pSkiplist.current.next ← pSkiplist.prev.next
    Persist(pSkiplist.current.next)
    pSkiplist.prev.next ← pSkiplist.current
    Persist(pSkiplist.prev.next)
    pSkiplist.current ← pSkiplist.current.next
    Persist(pSkiplist.current)
    /* Step 4. Check whether persistent skiplist is
    full */
    if pSkiplist.isFull() then
        pSkiplist.current ← NullNode()
        Persist(pSkiplist.current)
        Insert(fileMetadataList ← new metadata)
        pSkiplist ← Null
end
```

Algorithm 3: Recover database

Input: None
Output:
fileMetadataList: list of metadata including new output number, smallest/largest key, filesize

```
/* Step 1. Get persistent pointer from NVM pool*/
pools ← GetPoolsInDirectory()
persistentPtrs ← GetPersistentPointers(pools)
logFiles ← GetLogFilesInDirectory()
/* Step 2. Restore metadata of persistent skip list from
persistent pointer */
for pptr in persistentPtrs do
    restoredSkiplist ← pptr.GetSkiplist()
    last ← restoredSkiplist.SeekToLastNode()
    /* Detect system failure occurs */
    if last.isNullNode() then
        /* Failure when flush data into skip list at L0
        */
        if restoredSkiplist.ID = logFiles.ID then
            Reset(restoredSkiplist)
            memtableIter ←
            logFiles.createIteratorAboutID()
            Flush(memtableIter)
        /* Failure when byte-addressable compaction
        at lower levels */
        else
            IDs ← restoredSkiplist.metadata.IDs //
            Get IDs to be compacted
            mergeIter ← Iterator(IDs)
            while
            mergeIter.key<=restoredSkiplist.current.key
            do
                mergeIter ← mergeIter.next // pass
                already inserted data
            end
            ByteAddressableCompaction(mergeIter) //
            resume for remaining mergeIter
    metadata ← restoredSkiplist.generateMetadata()
    Insert(fileMetadata ← metadata)
end
```

considers a practical system for using storage devices more efficiently in terms of price and capacity.

We present three tiering policies. (1) Leveled tiering: is a method to store the data entries from the upper levels to NVM as the persistent skip list and the remaining from the lower levels to the block device as SST files, i.e., new SST files at lower level are created when byte-addressable compaction among persistent skip list or existing compaction among SST files occurs. For analysis that measured lifetime of SST files at each level, it is the simplest and most intuitive method. (2) Least recently used (LRU): is a method of flushing the persistent skip list that has been staying in NVM for the longest time, if the capacity of the NVM is full. In LRU policy, tiering occurs according to the capacity of NVM. Thus, new output from flush and compaction is created only as a persistent skip list. In general, the probability of eviction for SST files at the upper level is low, while the persistent skip list at the lower level is more likely to be tiered. (3) No tiering (Full-NVM): is a method to save all data to NVM in the form of skip list without using block device. This policy can be used when the capacity of NVM is enough to replace disk and flash memory.

E. RECOVERY

When a TLSM-based system recovers the database, the system should restore all metadata about persistent skip list. Thus, we obtain persistent pointer from persistent pool, then deal with two failure cases in order to recover both metadata in host and skip list in NVM. Algorithm 3 shows how to

recover database from program halt and unexpected failure. First, we obtain persistent pool in database directory, then get persistent pointers about persistent skip list object (lines 1–2). Next, we obtain log files in database directory if they exist (line 3). For each persistent pointer, TLSM checks whether persistent skip list is valid by checking the last node (lines 5–7). If the last node is not a null node, TLSM detects that system failure occurred from flush operation on L_0 or byte-addressable compaction (line 7–8). If persistent skip list ID is in the log file list, then TLSM clears it and re-runs flush operation based on the contents of the log file (lines 8–11). However, if not, TLSM suggests that the system failure occurred during byte-addressable compaction. In this case, TLSM copes with the crash by resuming compaction. When TLSM conducts byte-addressable compaction, it stores IDs to be compacted to header. Using this metadata, TLSM creates an old merge iterator that will participate in this compaction, then resumes byte-addressable compaction after passing already processed key-value pairs (lines 13–18). Finally, TLSM generates metadata information from current persistent skip list, then insert metadata into metadata list (lines 19–20). Thus, TLSM can recover all metadata for persistent skip lists from log file, persistent pointer and resumption of compaction.

V. EXPERIMENTS

In this section, we evaluate TLSM-based system according to some topics. We first describe experimental setups and workloads. Then, we evaluate and analyze throughput of TLSM-tree.

A. EXPERIMENTAL SETUP

The system is equipped with Intel I7-6700K processor that has 4 cores, running at 4.00GHz. The system has 64GB of DRAM, among which 16GB is emulated for NVM. The equipped SSD is 256GB of Samsung 850 PRO.

We ran the system in Linux CentOS7 with kernel version 4.4.166. We implemented our TLSM-based system in LevelDB 1.20. Because the persistent memory was not yet commercially available, we emulated NVM using PMDK 1.6, which is the Persistent Memory Development Kit on kernel. PMDK helps in organizing a persistent environment that creates and mounts ext4 file system with DAX, and then uses *mmap* function to access files. Kernel 4.0 and above support this environment, which is called NVDIMM - PMEM. We set the read and write latencies of NVM to 100ns and 450ns (i.e., about 4-5 times higher write latency compared to DRAM), respectively, similar to [7], [8], [9], [28]. In LevelDB settings, we turned off the compression and filter method to avoid influence on our evaluation.

B. WORKLOADS

We evaluated TLSM-based system using DB Bench and Yahoo! Cloud Serving Benchmark (YCSB) [31]. DB Bench is a popular micro-benchmark that is used by LevelDB as the default benchmark. DB Bench supports various evaluation mode such as *fillrandom* (random write), *readrandom*, *overwrite*, *fillseq* (sequential write), and *readseq*. In this evaluation, we ran *fillrandom*, then *readrandom* benchmarks, because current LSM-tree does not exploit the benefits of NVM and suffers from overheads in random write–read operations [7]. We set the key size to 16-bytes that is the default value on DB Bench.

In addition, we used macro-benchmark YCSB, which provides 6 real-world workloads including mixed operation with insert, read, scan, and update. YCSB can adjust the ratio of these operations, but many database systems evaluate those 6 workloads in general. We implement YCSB with TLSM derived from YCSB-LevelDB.

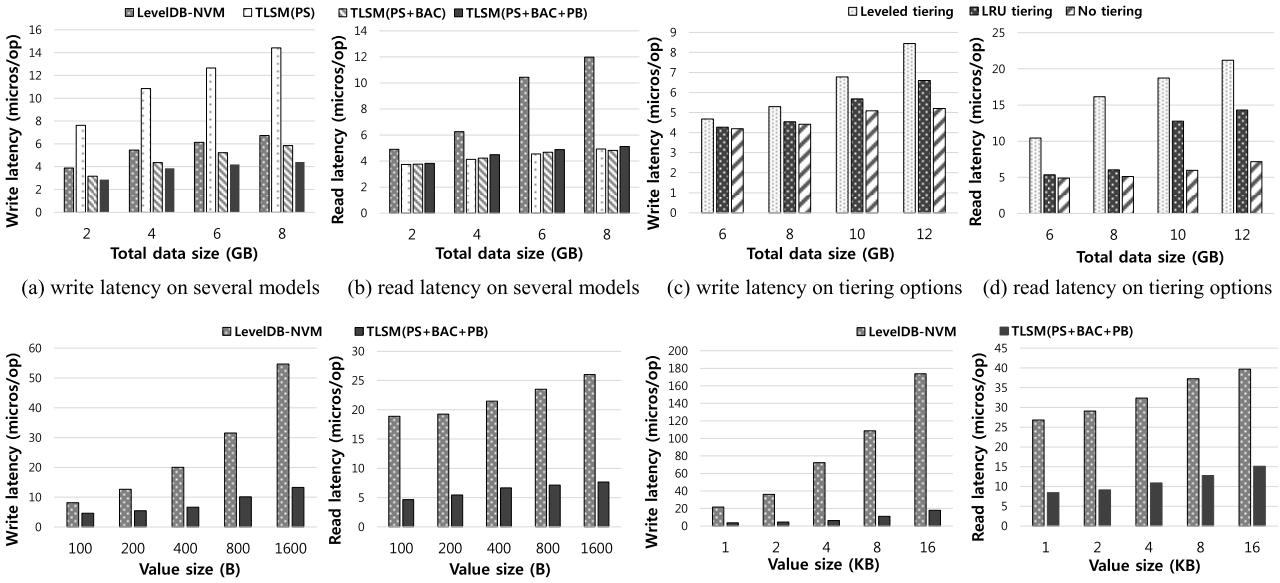
C. RESULTS ON MICROBENCHMARKS

In this experiment, we evaluated via DB Bench the write and read latencies compared to LevelDB-NVM and TLSM models. LevelDB-NVM uses fast and persistent memory area based on ext4 file system with *mmap* function. LevelDB-NVM is as fast as PMDK emulation because its environment creates and accesses SST files in file system mapped virtual memory. However, LevelDB-NVM cannot exploit the byte-addressability of NVM. We show the performance according to these categories: (1) impact on TLSM methods, (2) comparison with tiering policies, and (3) impact of byte-addressable compaction.

1) IMPACT ON TLSM METHODS

In this experiment, we compared LevelDB-NVM and TLSM models that add the proposed methods one by one. There are three version of TLSM: TLSM(PS), which uses persistent skip list only; TLSM(PS+BAC), which uses persistent skip list with byte-addressable compaction; TLSM(PS+BAC+PB), which is the full design that implements persistent skip list, byte-addressable compaction, and sequential copy with persistent buffer. PS, BAC, and PB denote persistent skip list, byte-addressable compaction, and persistent buffer, respectively. The write-buffer size and value size are 4 MB and 100 B, respectively, by default. We experiment with growing data size up to 8 GB.

Fig. 5(a) shows the write latency in microseconds/Op of each model. Unlike the TLSM(PS) model, the performances of the other two TLSM models outperform LevelDB-NVM because TLSM(PS) implements existing compaction method with persistent data structure, which leads to write amplification. On the contrary, in TLSM(PS+BAC) and TLSM (PS+BAC+PB) models, data writes with byte-addressable compaction are quite reduced. TLSM (PS+BAC+PB) also conducts the optimal copy operations preserving the concept of current LSM-tree. Thus, compared to LevelDB-NVM, TLSM with full methods improves write latency up to about 34%. Furthermore, compared to TLSM(PS+BAC) model, TLSM with the full design reduces write latency up to about 20%. It shows that batch write with persistent buffer enhances the write throughput.



(a) write latency on several models (b) read latency on several models (c) write latency on tiering options (d) read latency on tiering options

(e) write latency on small value-size (f) read latency on small value-size (g) write latency on large value-size (h) read latency on large value-size

FIGURE 5. Throughput of TLSM according to measurements in micro-benchmarks.

In Fig. 5(b), for the read latency, TLSM with full methods also outperforms LevelDB-NVM up to about 57%. The result suggests that skip-based searching on persistent skip list serves as indexing even if the persistent skip list on TLSM does not store index information like index block in SST file. Generally, TLSM models significantly enhance read performance by eliminating data serialization–deserialization between byte-format memory and block-based storage although LevelDB-NVM also conducts fast access from virtual memory. However, TLSM(PS+BAC+PB) model is slightly slower than the other two models. The reason is that the fully combined model stores the buffer offset in the persistent skip list, which requires additional data read in order to access raw key-value pair in the persistent buffer. For TLSM(PS) and TLSM(PS+BAC), each node of skip list has an area for storing key and value, which therefore makes it possible to directly access the key-value data. Nonetheless, we consider this to be an acceptable performance degradation because the model improves write throughput by a far greater degree.

Consequently, despite using the same persistent area, NVM-aware data structures and efficient methods remove redundant data transformation, load, and write. As data size grows, the gap of write and read latency between models increases. This phenomenon suggests that the more data are inserted, the more the gap between LevelDB-NVM and TLSM grows. Because TLSM with full methodologies is the optimal model for random write–read and because we also pay more attention to this model, we measured the performance only of this model for upcoming evaluations.

2) COMPARISON WITH TIERING POLICIES

We defined tiering options for the practical use of TLSM. Although tiering operations cause more data writes on the system, tiering can cope with various environments that have different resources. Thus, to analyze the effect of each policy, we compared the latencies between three tiering policies. For leveled tiering, we set the level threshold to L_3 after considering the result shown in Table 2, i.e, if new output should be inserted at L_4 or less, then it is created as SST file in disk. For LRU tiering, we set the capacity of NVM to 8 GB. Thus, if NVM is full and there is the new output, least-recently-used skip list is flushed into disk as SST file format, then new output is generated as persistent skip list. For no tiering, all output is created as persistent skip list. We measured the latency up to 12 GB of data. TLSMs with the tiering options, except for no tiering, use SSD device in order to store flushed SST file.

Fig. 5(c) and Fig. 5(d) illustrate the write and read latencies according to data size. TLSM with no tiering shows the best performance in both write and read, as expected. The result demonstrates the influence of using SST file format when tiering occurs. TLSM with LRU tiering shows performance similar to that of no tiering for up to 8 GB of data, while the gap between no and LRU tiering gets bigger when data size is greater than 8 GB. This result happens because old skip list is flushed as SST file when data size is greater than the capacity of NVM, then all new output in LRU tiering is generated as persistent skip listm i.e., existing compaction with SST file may degrade system performance if NVM is full. On the other hand, TLSM with leveled tiering performs poorly than no tiering and even LRU tiering. The result implies that LRU tiering is the extension of leveled tiering because most eviction of old skip list occurs at lower levels in LRU tiering. Moreover, current LSM-tree can trigger compaction during read operation, which leads to generating new outputs. In this condition, LRU tiering can retain new skip list, whereas leveled tiering results in more SST files to be generated despite

the space of NVM available. Thus, LRU is a more flexible option than leveled tiering. Even if TLSM with LRU tiering is slower than TLSM with no tiering, tiering can cover various resource environments.

3) IMPACT OF BYTE-ADDRESSABLE COMPACTION

In order to show how byte-addressable compaction affects the performance of system, we adjust value size in this experiment. As value size increases, compaction may occur more frequently because persistent skip list or SST file becomes full easily. Thus, we divide the experiments into two cases: one is small value-size from 100 bytes to 1600 bytes with write-buffer size at 4 MB, while the other is large value-size from 1 KB to 16 KB with write-buffer size at 64 MB. The results of the former are shown in Fig. 5(e) and Fig. 5(f), while Fig. 5(g) and Fig. 5(h) show the results for the latter. In evaluations, we set the total size of database to 12 GB and use no-tiering policy.

In Fig. 5(e), as value size increases, LSM-tree brings about more compaction, which degrades the write performance. However, the performance decline of TLSM is very small. This observation shows that byte-addressable compaction eliminates redundant writes by reusing valid key-value pairs. As a result, for small value sizes, the write latency of TLSM is about 75% lower than that of LevelDB-NVM.

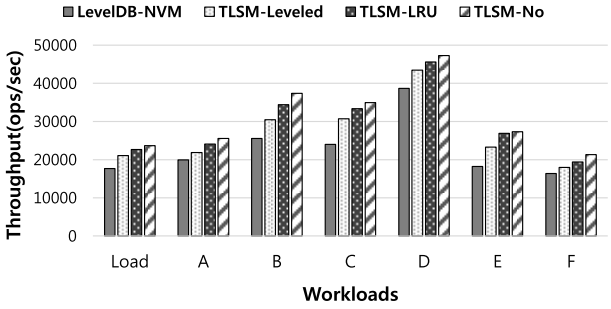
In Fig. 5(g), when value with size greater than 1 KB is inserted, the write latency increases considerably in the existing system. However, TLSM prevents the write performance from degrading dramatically. Compared to Fig. 5(e), the gap between the write performance of LevelDB-NVM and TLSM gets bigger in Fig. 5(g). The write latency of TLSM with large value outperforms that of LevelDB-NVM up to about 90%. In summary, TLSM with byte-addressable method seems to be almost unaffected bywrite amplification on compaction.

As shown in Fig. 5(f), TLSM also enhances the read throughput. Fig. 5(f) shows that TLSM can seek and get raw data efficiently, compared to current LSM-tree. The result demonstrates that TLSM can exploit byte-addressability of NVM from persistent data structure and expedite lookup with skip-based searching and persistent buffer. This tendency demonstrated by the results according to the size of value is similar to that shown for large value in Fig. 5(i). TLSM yields higher read latency about 76% and 72% for both small and large size of value, respectively. As a result, TLSM obtains significantly better read performance regardless of value size.

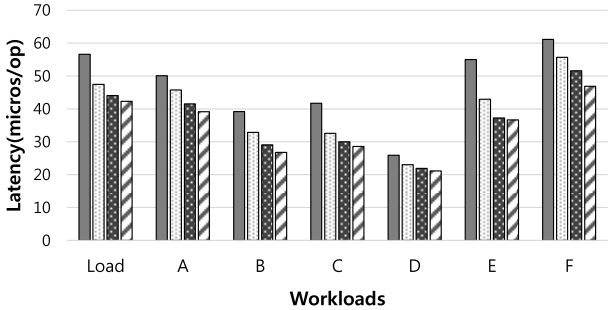
D. RESULTS ON MACROBENCHMARKS

In this experiment, we evaluated via YCSB real-world workloads compared between LevelDB-NVM and TLSM with full design model. YCSB provides 6 workloads, from workload-A to F. Each workload executes two phases: (1) load phase as a warm-up time, and (2) run phase as actual operations for benchmarking database. We measured load throughput separately. For each of the run phases, we destroyed the whole database, then re-loaded data, followed by executing the run phase of each workload. Regarding each of the workloads, workload A performs 50% read and 50% update operations, workload B performs 95% read and 5% update operations, workload C performs only read operations, workload D performs 95% read for the latest data and 5% insert operations, workload E performs 95% scan and 5% insert operations in order to evaluate short ranges, and workload F performs 50% read and 50% read-modify-write operations. All workloads use Zipfian distribution. The value size in YCSB is around 1 KB, consisting of each 100-byte value in 10 fields. We set the total size of data to 12 GB.

Fig. 6(a) and Fig. 6(b) illustrate the overall throuputs in Ops/second and the latency in microsecond/op. All workloads including load phase show performance for TLSM with no tiering, TLSM with LRU tiering, TLSM with leveled tiering, and LevelDB-NVM, in decreasing order. Generally, TLSM improves read and write throughputs and TLSM with leveled tiering performs more poorly than the other two models. For load phase, the latency of TLSM is about 16% to 25% lower than that of existing LevelDB-NVM. The result shows that TLSM can accelerate write operations through byte-addressablility and overcome typical shortcomings on current compaction. For workload A, TLSM-based systems have performance gains of about 9% to 22% higher than that of LevelDB-NVM. Workload A is update-intensive, which means that the update operates in seek, followed by insertion. Hence, TLSM with leveled tiering seems to have degraded performance from data insertion, while the performance of LRU tiering is slightly lower than that of no tiering. For read-intensive benchmark in both workloads B and C, the TLSM models outperform the latencies of LevelDB-NVM by about 18% to 31% and 22% to 31%, respectively.

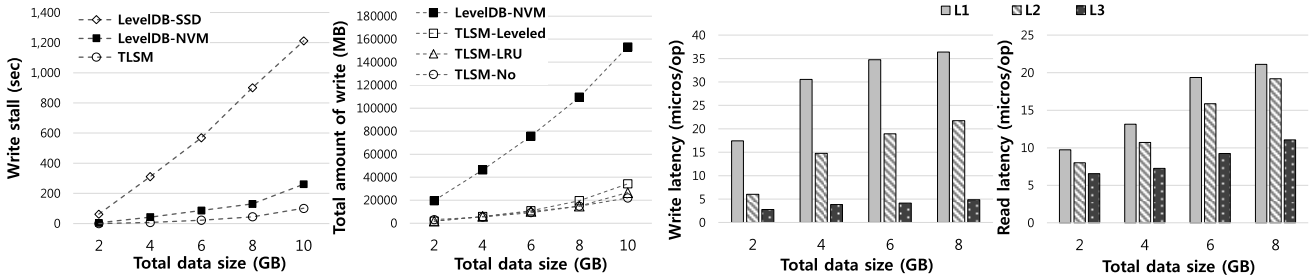


(a) Throughput on macro-benchmark



(b) Latency on macro-benchmark

FIGURE 6. Results of TLSM in macro-benchmarks.



(a) Accumulation of write stall (b) Amount of write from compaction (c) Write latency on level threshold (d) Read latency on level threshold

FIGURE 7. Evaluations for write efficiency and level threshold on TLSM.

The result demonstrates that TLSM-based models improve the performance of read and insert operations. For workload D, TLSM models also improve the latency of reading latest key by about 11% to 18%. Although current LevelDB already shows good performance, TLSM can achieve better read performance by taking advantage of byte-addressable data structure. For short-scan benchmark in workload E, the greatest gap in performance between LevelDB-NVM and TLSM, at about 22% to 33%, is demonstrated. Compared to workloads B and C, current LSM-tree was most affected by the performance degradation due to scan operations, implying that TLSM can also carry out fast scan through iteration and skip-based indexing. For read-modify-write workload in F, the latency of TLSM is around 9% to 23% lower than that of LevelDB-NVM. As a result, in macro-benchmark, TLSM decreases write and read bottlenecks using efficient read operation based on byte-addressable data structure and getting rid of redundant rewrites from novel compaction method. Additionally, we found that LRU tiering policy is optimally designed for maximizing NVM as middle layer and for considering practical use, while leveled tiering is a simple but vague structure.

E. WRITE EFFICIENCY

Next, we evaluate two issues referred to in Section III in order to verify the effect of TLSM as middle layer resolving latency gap between DRAM and storage. First, we measured the amount of write delay when write buffer in memory should be flushed. Second, we measured total amount of data write induced by on-disk compaction. We ran *fillrandom* workload via DB Bench.

1) ACCUMULATION OF WRITE STALL

Current LSM-based system deliberately causes write delay by balancing the speed between flush of write buffer and generation of file at L_0. If L_0 contains many SST files, LSM-based system postpones making the new SST file at L_0. In this condition, if a new key-value pair is inserted into memtable, and memtable is full despite holding immutable memtable, data flush from memory to disk is delayed by the system. Because the amount of write stall increases exponentially as more data are inserted, existing system often suffered from that behavior.

Fig. 7(a) shows how much the write stall is accumulated according to increasing data size. We compare the latencies between LevelDB-SSD, LevelDB-NVM, and TLSM with full methods. LevelDB-SSD is a typical environment that uses SSD as storage device. Meanwhile, TLSM use the no tiering option. In LevelDB-SSD, as data size grows, write stall reaches about 20 minutes, i.e., to insert 10 GB of data, the system would spend more than 20 minutes to complete the process. From the results, we can infer that the amount of accumulation increases greatly as data size increases, whereas write stall drastically decreases in both LevelDB-NVM and TLSM model. LevelDB-NVM may treat many SST files as if they are in memory, and that it is able to access the files quickly. Furthermore, TLSM can benefit from byte-addressability without unnecessary writes, which results in surpassing the overall speed in LevelDB-NVM. Thus, TLSM alleviate system bottlenecks between main memory and storage.

2) TOTAL AMOUNT OF WRITE ON COMPACTION

To maintain system architecture and perform garbage collection, current LSM-tree triggers data compaction. However, write and read amplification occurs during compaction when the system reads SST files in disk, performs *merge sort* through the key comparison, then writes new SST files with valid key-value pairs. We especially focused on the write amplification because current compaction method performs redundant data writes. We evaluated the mitigation of amplification compared to that of existing system.

Fig. 7(b) illustrates the total amount of data write from compaction in LevelDB-NVM and TLSM with tiering policies. Typically, TLSM reduces write amplification at a greater degree than does LevelDB-NVM. Furthermore, as data size grows, the gap of total writes between LevelDB-NVM and TLSM models widens. From this result, our byte-addressable and write-optimal methodologies by pointer operations on NVM eliminate rewriting for valid key-value data. In TLSM models, the tiering options, except for no tiering, require more data writes to perform compaction among SST files and generate new SST files. TLSM with leveled tiering divides

storage space into two devices by the level. It also exploits the features of NVM and reduces much more write than does the current LSM-tree, yet seems to have available space in NVM. On the other hand, TLSM with LRU tiering keeps persistent skip list according to the capacity of NVM. Thus, it can make good use of NVM to capacity, and write operations are performed less than in leveled tiering. Finally, TLSM with no tiering, which is the ideal design, decreases the total amount of data write up to around 18% relative to LRU tiering, 36% relative to leveled Tiering. We expect that TLSM may show a similar trend even if more key-value data are inserted.

F. IMPACT OF LEVEL THRESHOLD ON LEVELED TIERING

From the result presented in Section III-B, SST files at upper levels have shorter lifetimes than those of SST files at lower levels. Because LSM-tree organizes hierarchical structure and requires on-disk compaction, the SST files at upper levels are deleted quickly. Thus, in this section, we investigate the impact of leveled tiering depending on level threshold. We set the level threshold to level 1, level 2, and level 3, because NVM cannot keep all SST files at level 4 or less. We used no tiering policy in this experiment.

Fig. 7(c) and Fig. 7(d) show the write and read latencies of TLSM according to the level threshold. L1 means that TLSM uses persistent skip list as data format until level 1, and SST files are generated in L_2 or less. We excluded L0 threshold because TLSM cannot apply the proposed compaction method in this environment. The performance gap in both Fig. 7(c) and Fig. 7(d) is proportional to the number of compacted files from the result shown in Table 2. For both write and read, the more TLSM embraces on-disk levels, the more it exhibits performance gains as expected. In other words, if TLSM creates more persistent skip lists and performs byte-addressable compaction compared to the other models, then it can lead to the reduction of data writes. Thus, the result shows that L3 threshold on leveled tiering is the most effective in TLSM design.

VI. CONCLUSION

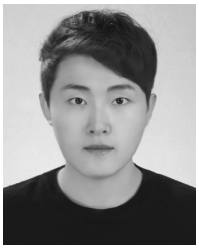
Utilizing persistent memory provides a more efficient method to current block-based database system. In this study, we presented tiered LSM-based system that is multi-layered with NVM and improves write and read performance. Before that, we analyzed two issues that occurred in LSM-based system, then verified the feasibility of usage of NVM to decrease unavoidable, repeated I/O costs. We applied persistent data structure, byte-addressable compaction, and sequential copy with light-weight data structure step by step to TLSM-based system by failure-atomic manner. In addition, we tried to cover various environments by defining tiering policy.

In the evaluation of micro-benchmarks, TLSM improves the write and read latencies up to 34% and 57%, respectively, compared to those of LevelDB-NVM. According to value size, write and read latencies of TLSM outperform those of current system up to about 90% and 62%, respectively. On macro-benchmarks, TLSM also enhance the load and read-intensive throughputs by about 25% and 31%. Furthermore, TLSM achieves performance gains with all workloads. The results with respect to write efficiency show that TLSM significantly reduces write delay on LSM-based system and the total amount of write on compaction. Finally, we showed the correlation between compacted output at upper levels and the performance of TLSM by evaluation according to level threshold. As a result, we considered LRU tiering to the optimal policy for making the good use of NVM to capacity and TLSM can be applied to various system resources, which multiple devices.

REFERENCES

[1] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, Jun. 1996.
[2] 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.
[3] *Apache HBase*. Accessed: Feb. 2019. [Online]. Available: https://hbase.apache.org/
[4] *Cassandra*. Accessed: Feb. 2019. [Online]. Available: http://cassandra.apache.org/
[5] *LevelDB*. Accessed: Feb. 2019. [Online]. Available: http://leveldb.org/
[6] *RocksDB*. Accessed: Feb. 2019. [Online]. Available: https://rocksdb.org/
[7] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning LSMs for nonvolatile memory with NoveLSM," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2018, pp. 993–1005.
[8] L. Lersch and I. S. I. L. W. Oukid, "Rethinking DRAM caching for LSMs in an NVRAM environment," in *Proc. Eur. Conf. Adv. Databases Inf. Syst.*, Switzerland: Springer, Cham, Sep. 2017, pp. 326–340.
[9] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. Choi, "SLM-DB: Single-level key-value store with persistent memory," in *Proc. 17th USENIX Conf. File Storage Technol. (USENIX FAST)*. Feb. 2019, pp. 191–205.
[10] *NVMRocks: RocksDB on Non-Volatile Memory Systems*. Accessed: Feb. 2019. [Online]. Available: http://istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems/
[11] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM SIGOPS Operating Syst. Rev.*, vol. 25, no. 5, pp. 1–15, Oct. 1991.
[12] *RocksDB Wiki*. Accessed: Feb. 2019. [Online]. Available: https://github.com/facebook/rocksdb/wiki/
[13] K. Ouaknine, O. Agra, and Z. Guz, "Optimization of RocksDB for redis on flash," in *Proc. Int. Conf. Compute Data Anal. ICCDA*, May 2017, pp. 155–161.
[14] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," *ACM Trans. Storage*, vol. 13, no. 1, pp. 1–28, Mar. 2017.
[15] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "LSM-trie: An LSM-tree-based ultra-large key-value store for small data items," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2015, pp. 71–82.
[16] F. Pan, Y. Yue, and J. Xiong, "DCompaction: Delayed compaction for the LSM-tree," *Int. J. Parallel Program.*, vol. 45, no. 6, pp. 1310–1325, Dec. 2017.
[17] T. Yao, J. Wan, P. Huang, X. He, Q. Gui, F. Wu, and C. Xie, "A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores," in *Proc. 33rd Int. Conf. Massive Storage Syst. Technol. (MSST)*, 2017, pp. 1–13.
[18] Z. Zhang, Y. Yue, B. He, J. Xiong, M. Chen, L. Zhang, and N. Sun, "Pipelined compaction for the LSM-tree," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, May 2014, pp. 777–786.
[19] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in RocksDB," in *Proc. CIDR*, vol. 3, Jan. 2017, p. 3.
[20] R. Sears and R. Ramakrishnan, "BLSM: A general purpose log structured merge tree," in *Proc. Int. Conf. Manage. Data SIGMOD*, 2012, pp. 217–228.

[21] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "NVWAL: Exploiting NVRAM in write-ahead logging," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 385–398, Jun. 2016.
[22] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via JUSTDO logging," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 2, pp. 427–442, 2016.
[23] G. Oh, S. Kim, S.-W. Lee, and B. Moon, "SQLite optimization with phase change memory for mobile applications," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1454–1465, Aug. 2015.
[24] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-Tree: Reducing consistency cost for NVM-based single level systems," in *Proc. 13th USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 167–181.
[25] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A hybrid SCM-DRAM persistent and concurrent B-Tree for storage class memory," in *Proc. Int. Conf. Manage. Data SIGMOD*, Jun. 2016, pp. 371–386.
[26] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: Write optimal radix tree for persistent memory storage systems," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, 2017, pp. 257–270.
[27] *PMDK: Persistent Memory Programming*. Accessed: Feb. 2019. [Online]. Available: https://pmem.io/pmdk/
[28] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ. SOSP*, Oct. 2009, pp. 133–146.
[29] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 1, pp. 91–104, Mar. 2011.
[30] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-atomic slotted paging for persistent memory," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 91–104, 2017.
[31] 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.



JIHWAN LEE received the B.S. and M.S. degrees in computer science from Yonsei University, Seoul, South Korea, in 2017 and 2019, respectively. His current research interests include database systems, distributed processing systems, NoSQL, and non-volatile memory.

WON GI CHOI received the B.S. degree from Yonsei University, Seoul, South Korea, in 2014, where he is currently pursuing the Ph.D. degree. His current research interests include database systems, flash memory, and non-volatile memory.

DOYOUNG KIM received the B.S. and M.S. degrees in computer science from Yonsei University, Seoul, South Korea, in 2018 and 2020, respectively. His current research interest includes key-value stores with utilizing non-volatile memory or graphic processing unit (GPU).

HANSEUNG SUNG received the B.S. degree in software and computer engineering from Korea Aerospace University, in 2017. He is currently pursuing the M.S. degree in computer science with Yonsei University, Seoul, South Korea. His current research interests include database systems, in-memory, key-value stores, and logging and recovery.

SANGHYUN PARK (Member, IEEE) received the B.S. and M.S. degrees in computer engineering from Seoul National University, in 1989 and 1991, respectively, and the Ph.D. degree from the Department of Computer Science, University of California at Los Angeles (UCLA), in 2001. He is currently a Professor with the Department of Computer Science, Yonsei University, Seoul, South Korea. His current research interests include databases, data mining, bioinformatics, and flash memory.