FlashTKV: A High-Throughput Transactional Key-Value Store on Flash Solid State Drives

Robin Jun Yang
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Hong Kong, China
yjrobin@cse.ust.hk

Qiong Luo
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Hong Kong, China
luo@cse.ust.hk

***Abstract*—We propose FlashTKV, a high-performance transactional key-value store optimized for flash-based solid state drives. Transactional key-value stores process large numbers of concurrent reads and writes of key-value pairs, and maintain transactional consistency. As such systems are I/O dominant, flash SSDs are a promising storage alternative to improve the system performance. Catering the asymmetry in the read and write performance of flash SSDs, FlashTKV uses a purely sequential storage format where all data and transactional information are log records. Furthermore, this sequential storage format supports multi-version concurrency control (MVCC) efficiently. We evaluate FlashTKV on enterprise SSDs as well as on magnetic disks. While on magnetic disks FlashTKV performs similarly to systems with MVCC on page-based storage or locking on sequential storage under TPC-C workloads, it improves the transaction throughput by 70% over the competitors on flashSSDs.**

Keywords-KV-store; Flash SSD; Log-structured; MVCC .

I. INTRODUCTION

Flash Solid State Drives (SSDs) are emerging as a competitive storage alternative for laptops, desktops, as well as servers, due to their outstanding I/O performance, shock resistance, and energy efficiency. Table I shows the performance comparison between a representative enterprise flash SSD and a high-end magnetic disk. While both disks achieve an almost identical throughput on sequential writes, the sequential read throughput of the flash SSD is 1.5 times of that on the hard disk. A striking difference between the two disks across access patterns, is that, the read and write performance is symmetric on the magnetic disk but is not on the flash SSD. In particular, on the SSD the sequential read throughput is 1.5 times of the sequential write, and the random read throughput is over 10 times of the random write. Finally, the performance gap between random and sequential patterns is reduced from a factor of 200 on the hard disk to around 2 for reads and 15 for writes on the flash SSD. While these numbers confirm the superb performance of flash SSDs, they also suggest that performance optimization strategies for the flash may be different from those for the hard disk due to the read-write asymmetry.

Acknowledgement: This work was supported in part by grant HUAW28-15L05211/12PN from Huawei Technologies.

Table I: Performance Comparison between An Intel X25-E Flash SSD and A SAS 15kRPM Magnetic Disk

Device	Flash SSD	Magnetic disk
Seq. Read Throughput	248MB/s	164MB/s
Seq. Write Throughput	167MB/s	166MB/s
Ran. 4KB Read IOPS (Calculated Throughput)	33,569 (127.2MB/s)	192 (0.75MB/s)
Ran. 4KB Write IOPS (Calculated Throughput)	2,940 (11.5MB/s)	192 (0.75MB/s)
Read Latency	75μs	5200μs
Write Latency	85μs	5200μs

Recently there have been studies on optimizing the I/O performance of a database management system component, such as query processing [1], buffer management [2], [3], indexing [4], [5], [6], [7] and storage management [8] for flash SSDs. There has also been work on using flash SSDs for key-value stores (KV-stores), such as FlashStore [9] and SkimpyStash [10]. In comparison, we focus on transactional key-value stores, which is an important type of workload in practice yet is challenging for flash SSDs due to the large number of random writes.

A transactional KV-store, such as the Oracle BerkeleyDB [11], supports read and write operations on key-value pairs, and guarantees transactional consistency of these read and write operations. As a result, there are large numbers of random I/O for key-value pair reads and writes as well as a large amount of transaction log writes. Considering the characteristics of flashSSDs, we propose FlashTKV, a transactional KV-store for flash SSDs. FlashTKV has the following three distinguishing features:

- It has a purely sequential storage format where all the data are stored as log records (log as data).
- All transactional information are also written as log records into the sequential storage.
- The sequential storage supports the multiversion concurrency control protocol (MVCC) for transactional consistency.

 ISBN: 978-1-61208-226-4

The main technical challenges in FlashTKV are how to support (1) reads and (2) MVCC efficiently on the sequential storage. Specifically, log-structured approaches [12] optimize writes by converting random data writes into sequential log writes, but slow down reads as up-to-date data pages must be constructed by applying change logs to the original data pages. Also, MVCC has two main drawbacks: (1) the overhead of writing multiple versions of each data item; and (2) wasted processing due to transaction rollbacks. The first drawback is less costly on flash SSDs than on hard disks as writes to flash memory will be to new pages anyway and random reads are fast on flash. The second drawback remains on flash SSDs; nevertheless it is outweighed by the fast reads on flash SSDs, as we will see in the experiments. Furthermore, existing MVCC algorithms and implementations all assume a page-based data storage format and a separate, write-ahead logging (WAL) transaction log. It is unclear how a sequential storage format without a page-based data storage or a separate transaction log can support MVCC correctly and efficiently.

To support reads efficiently, our sequential storage with a uniform set of logs replaces separated sets of data pages and change logs. Consequently, there is no merging operation between data pages and change logs. Instead, we only need to retrieve a suitable log record for a given key on a read request. To speed up exact-match as well as range searches, we further maintain a B^+-tree to index the KV-pairs and use an in-memory node buffer pool to keep recently accessed B^+-tree nodes. To support MVCC on our sequential storage format, we keep necessary transactional information in log records and retrieve a suitable log record for each transactional read based on timestamp information.

We have implemented FlashTKV and evaluated it in comparison with the Oracle Berkeley DB (BDB), a leading industrial-strength transactional key-value store on an enterprise-grade flash SSD. Our results show that (1) the estimated read I/O time in FlashKTV was almost identical to that in BDB and the estimated write time in FlashKTV was only 30% of that in BDB; (2) the measured performance of FlashKTV under different degrees of read-write contention was up to 40% faster than that of BDB; (3) under TPC-C workloads, FlashKTV improves the throughput by up to 70% over BDB. This paper is organised as follows: Section II discusses the background and related work of our paper, Section III describes the detailed design and implementation of FlashTKV, Section IV compares the I/O operations in the traditional page storage and the sequential storage used in FlashTKV, Section V shows the experimental setup and results and Section VI concludes the paper.

II. Background and Related Work

In this section, we first discuss the read-write asymmetry of flash SSDs. Then we review related work on optimization techniques that addressed this issue, especially log-structured approaches. Finally we compare FlashKTV with other key-value stores, especially the Oracle Berkeley DB Java Edition (BDBJE), which also adopts a sequential storage format.

Flash SSDs use the NAND flash memory as the storage media which does not support in-place update, but instead requires an erase operation before a write. The erase operation can only be performed at the granularity of an erase block (typically 64 flash pages). The FTL (Flash Translation Layer) inside an SSD alleviates this problem by directing writes to clean pages; however, it also causes garbage collection to run more frequently. As a result, random writes continue to be the worst-performing access pattern on flash SSDs.

To address the random write problem on flash SSDs, a few new file systems [13], [14] have been proposed. They are similar to the log-structured file system [12], which maintains a mapping between logical and physical addresses of pages and transforms write requests to sequential append operations to the storage device. This log-structured approach avoids random writes to the storage device, but slows down read operations due to the use of the mapping table to locate the current page. Furthermore, garbage collection in these file systems needs to run frequently and degrades the performance severely, especially on flash SSDs of a large capacity.

There has been a flurry of work on optimizing DBMS components such as query processing [1], buffer management [2], [3], indexing [4], [5], [6], [7] and storage management [8] for flash SSDs. As transactional workloads such as OLTP (Online Transactional Processing) generate a large number of random writes on traditional database systems, they are the most challenging to optimize on flash SSDs. There has also been work on using flash SSDs for lightweight database systems such as the Key-Value Stores (KV-stores), e.g., FlashStore [9] and SkimpyStash [10]. These systems focus on minimizing the metadata size per key-value pair (KV-pair) in the RAM so that they can provide fast access and insertion to large datasets without introducing a significant maintenance cost for the metadata (index) of the KV-pairs. Nevertheless, these systems do not support user-defined database transactions and thus are unsuitable for OLTP applications.

Traditional two-phase locking has been the protocol of choice for concurrency control. With the read-write performance asymmetry of flash SSDs, there has been initial work exploring alternative concurrency control protocols on flash disks. In particular, Lee et al. [15] experimented with storing the MVCC rollback segments in a commercial database server on flash SSDs. Nevertheless, there has not been work on studying a full MVCC transactional system with sequential storage on flash SSDs. Our work is most related to the Berkeley DB Java Edition [16] (BDBJE), a well-known sequential storage engine. As it is Java-based,

BDBJE relies on JAVA NIO and has no explicit control on the underlying storage device. Furthermore, only locking, not MVCC, is supported for concurrency control in BDBJE.

A drawback of log-structured approaches, which are often adopted for flash-optimized techniques, is an essential and expensive operation, known as *merge*. The merge operation is necessary because the original data and the logged changes are separate entities, and these two need to be integrated from time to time to bring the data up-to-date. In comparison, our FlashTKV adopts a sequential storage format where all data and their changes are recorded uniformly as logs in time order. As a result, there is no merge operation needed. To speed up the random reads, we maintain an in-memory buffer pool for log nodes and organize these nodes into an in-memory B^+-tree for exact match as well as range searches on the keys.

III. Design and Implementation

In this section, we present the design and implementation of FlashTKV.

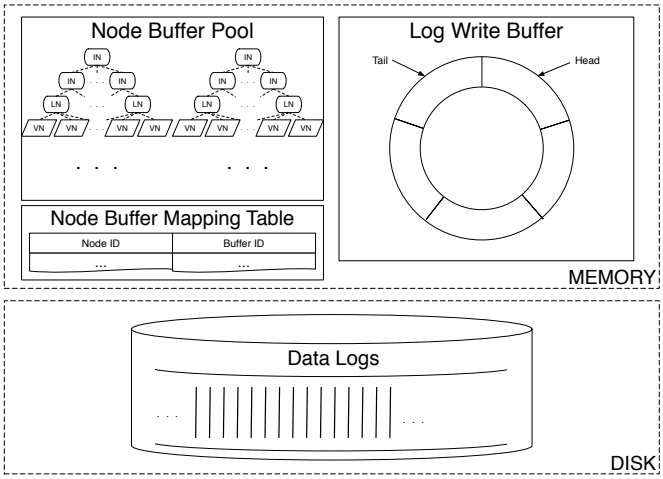


Figure 1: FlashTKV Storage

A. System Overview

Figure 1 illustrates the storage design in FlashTKV. On disk, we store all KV-pairs in *data logs* in the order of time when an insertion/deletion/update happens. For efficiency, we use an in-memory ring buffer as the *log write buffer* to batch up the tail of the data logs and write them to disk when the buffer is full or when transactions commit.

Since data logs are written in time order to the disk whereas KV-pair operations are based on keys, we use an in-memory buffer to cache frequently accessed KV-pairs. Furthermore, to support lookups and range searches efficiently, we maintain a B^+-tree index for each set of KV-pairs. Specifically, we store the keys in LNs (Leaf Nodes) and the values in VNs (Value Nodes), and create INs (Internal Nodes) to form a tree. We separate keys and values in memory because the sizes of values may vary greatly. Since the nodes are variable-sized, multiple nodes may reside on a single buffer page, and large nodes may span over multiple pages. A retrieval on a B^+-tree in this *node buffer pool* will start from the root, find the node ID of the child by the search key, use the node buffer mapping table to find the buffer page that contains the child node, and go down the tree iteratively until it finds the value node in the buffer or on disk or reports the non-existence of such a key in the database.

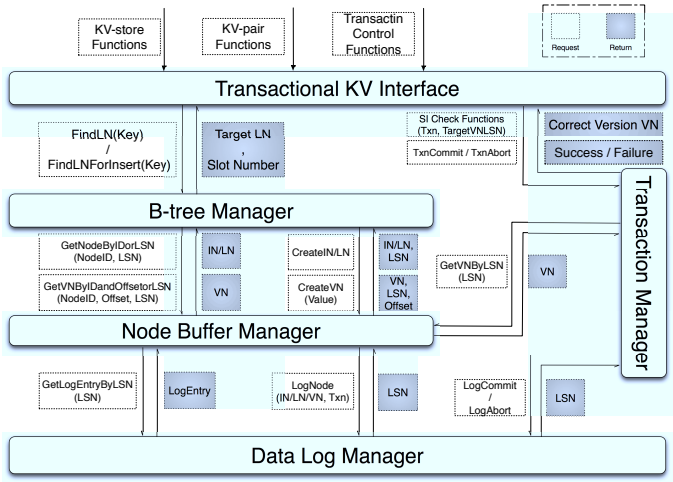


Figure 2: FlashTKV System Architecture

FlashTKV consists of five main components as shown in Figure 2. All the changes of the KV-pairs are stored in *data logs* and appended to the disk through the **Data Log Manager**. To support search on the KV-pairs, the **B^+-tree Manager** builds B^+-trees for all the KV-pairs using their keys. Frequently visited KV-pairs are kept in the RAM in the form of *nodes* in the *node buffer pool* maintained by the **Node Buffer Manager**. The **Transaction Manager** manages transactions of KV-pair operations. It utilizes MVCC to provide SI (Snapshot Isolation) for all transactions. By introducing a few more types of data logs in the **Data Log Manager** for storing all the information of transactions, *data logs* can be used to not only store KV-pairs but also provide transaction support. All the available functions in FlashTKV are provided by **Transactional KV Interface**. We discuss the five components in the following sections.

B. Transactional KV Interface

The *transactional KV interface* provides the interface of KV-store functions (create, open or close the KV-stores), KV-pair functions (transactional retrieval, insertion, update and deletion of KV-pairs), transaction control functions (start, abort and commit a transaction). It calls the B^+-tree

manager and the transaction manager to implement all the functions. The *Database* in FlashTKV is a directory in the file system. Each set of KV-pairs of the same schema are stored in a *TupleStore*. A database may contain multiple TupleStores.

C. B^+-tree Manager

The KV-pairs in one TupleStore are stored in one B^+-tree. The B^+-tree in FlashTKV has three types of nodes, **IN**, **LN** and **VN**. As shown in Figure 3, all keys are stored in the LNs, all values are stored in the VNs. One LN contains multiple keys whereas one VN contains only one value. The B^+-tree manager relies on the node buffer manager to maintain the memory space used by all nodes.

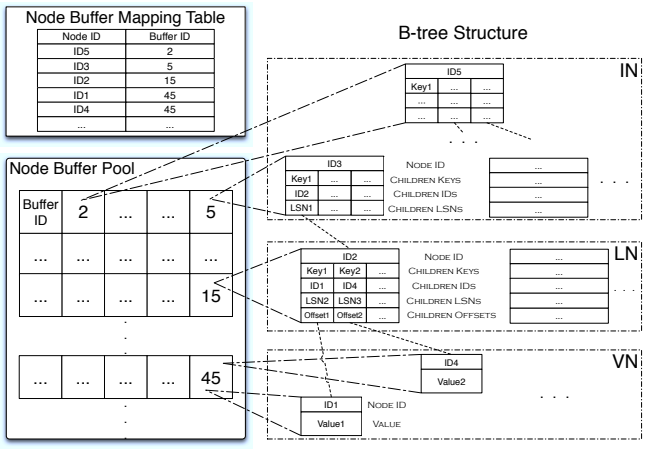


Figure 3: The B^+-tree Structure and Node Buffer Pool Layout for Sequential Storage in FlashTKV

To support the sequential storage and MVCC, the B^+-tree in FlashTKV has a few unique features compared to the standard B^+-tree. The standard B^+-tree uses one identifier, the node ID, which is the same as the page ID in the page-based storage, as the *persistent pointer* to locate a node in memory and on disk. However, such identifier is not enough to locate the node on disk in the sequential storage because writing an updated node to the disk is to append a new log to the disk, which means the physical position of the node on disk is changed. Therefore, the B^+-tree for the sequential storage uses the LSN of the node on disk as the persistent pointer. Moreover, the node size is flexible because all the data of the nodes are stored in data logs. Considering the maintenance cost and the efficiency of the memory access, we set the size of the IN/LN/VN to the size of a node buffer in the node buffer pool. We discuss the details of the memory allocation in Section III-D. Lock coupling [17] is used to provide high concurrency in INs while LNs and VNs can be accessed by multiple transactions. Furthermore, we perform opportunistic split: we split all full nodes on the search path for the insertion. Thus, latches can be obtained strictly from top down so that deadlocks can be avoided.

The biggest drawback of such design is the update efficiency. More specifically, when a VN is inserted or updated, its parent LN also needs to be updated because one of the LN's children LSNs changes and such updates will propagate up all the way to the root. We call this the *update propagation problem*. To overcome this problem, we write the log for the new VN and update the corresponding child LSN in the LN but only mark the status of the slot for the new VN in the LN *dirty* without writing logs for the updated LN immediately so that the update propagation is prevented. The logs for the updated IN/LN are written only when the IN/LN is evicted from memory. This treatment does not lose any change in data because the logs for the VNs already contain the whole KV-pair.

D. Node Buffer Manager

The node buffer manager is responsible for maintaining the memory space used by the nodes in the B^+-tree and returning the memory address of the requested IN/LN or VN. As shown in Figure 3, the *node buffer pool* is a large chunk of memory, with each unit called *a node buffer*. Compared to a buffer manager for the page-based storage, it has some unique features.

The node buffer manager allocates exactly one node buffer for each IN/LN but multiple VNs can be stored in a single node buffer. This different treatment is because (1) the numbers of INs/LNs are much fewer than VNs in the memory because of the tree structure; (2) the size of each VN varies. The maximum size of the VN is limited to the size of a node buffer, and we put multiple VNs into a single node buffer to save memory space.

When the node buffer pool is full, we use an LRU algorithm to choose a node buffer for eviction. Such LRU may choose a node buffer for IN to swap out while some of the node's children are still in the buffer pool and marked dirty. Since we must guarantee its latest version is written on disk when a node is evicted from the node buffer pool, we must write all its dirty children to disk to get their latest LSNs before writing the parent node. This process happens recursively until all the dirty nodes in the subtree rooted at the victim IN are written to disk. As a result, we can free all the node buffers for all the INs/LNs in this subtree. Note that the node buffers for the VNs in this subtree are not evicted because they may contain frequently visited VNs from other LNs. The node buffers for VNs are evicted only when the replacement algorithm chooses them as victims, which indicates all of VNs in this buffer are not recently used.

E. Data Log Manager

The data log manager is responsible for (1) transforming B^+-tree nodes into data logs and writing them onto the disk,

 ISBN: 978-1-61208-226-4

and (2) reading data logs from the disk and transforming them to B$^+$-tree nodes. Figure 4 shows all types of data logs in FlashTKV. The data log manager maintains a global log write buffer. The data logs to be written to disk are appended in the buffer and the buffer is flushed to disk when (1) the size of the existing data logs that have not been flushed exceeds the size of a flash erase block or (2) a transaction commits. The size of the buffer is an integral multiple of the flash erase block size. We organize the buffer as a ring buffer which further saves read I/O cost in the retrieval of the last committed version of a KV-pair in SI transactions. In addition, we implement the *group commit* algorithm to further increase the write I/O efficiency.

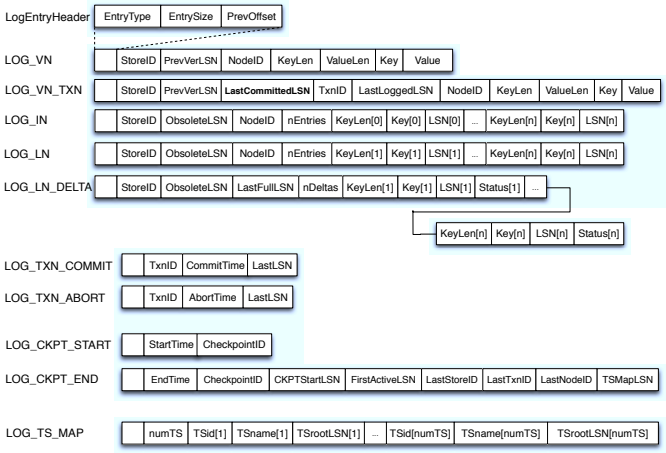


Figure 4: The Format of All Types of Data Logs

1) Data Logs for B$^+$-tree Nodes: We have five types of data logs to store three kinds of B$^+$-tree nodes. Both LOG_VN and LOG_VN_TXN are for the VNs. The difference between them is that LOG_VN_TXN is for the VNs inserted or updated by user-defined transactions. The *PrevVerLSN* in LOG_VN and LOG_VN_TXN is the LSN of the data log for the previous version of the VN. LOG_LN, and LOG_IN are for LNs and INs, respectively. The *ObsoleteLSN* is the LSN of the previous version of the corresponding LN or IN. The difference between LOG_IN and LOG_LN is that the IN has *nEntries + 1* children (IN or LN) but the LN has *nEntries* children VNs.

We further optimize the data logs for LNs because we found it is inefficient to log the entire dirty LN every time it is evicted from the buffer pool because only a few slots of the LSN array or the children ID array are dirty. We introduce a LOG_LN_DELTA log which contains only the updated part of the LN since its last LOG_LN log on disk. We use a simple I/O cost estimation to decide which type of log for LNs to use. Table II shows the total I/O time of writing either type of the logs and reading the LN back based on the number of the dirty entries to be written in a delta log. If $W_{delta} + R_{delta} < W_{LN} + R_{LN}$, LOG_LN_DELTA is chosen; otherwise, LOG_LN is chosen.

Table II: Total I/O Time Estimation for Deciding Which Log to Use

	Write I/O Time	Read I/O Time
DELTA	$W_{delta} = \left(\frac{N_{dirty} S_{LN}}{N_{LN} S_{fp}}\right) T_{SW}$	$R_{delta} = 2T_{RR}$
FULL	$W_{LN} = \left(\frac{S_{LN}}{S_{fp}}\right) T_{SW}$	$R_{LN} = T_{RR}$

S_{delta}: the size of the delta log
S_{LN} : the size of the log of the full version LN
S_{fp} : the flash page size of the flash SSD in use
N_{dirty} : the number of the dirty entries since last full version of the LN
N_{LN} : the total number of entries of LN
T_{SW} : I/O time of write a flash page sequentially
T_{RR} : I/O time of read a flash page randomly

When the LN is later reconstructed from a LOG_LN_DELTA, the dirty entries in it are still marked *dirty* in the LN. This marking is necessary because later if we decide to log the LOG_LN_DELTA again, we still need to log the previous dirty slots. An LN can be reconstructed either directly from (1) an LOG_LN or (2) an LOG_LN_DELTA and the LOG_LN. We need at most two random read I/Os to reconstruct an LN. Considering disk space cost, we set the maximum number of consecutive LOG_LN_DELTA logs for each LN as a configurable parameter.

2) Snapshot Isolation (SI) Support: To support SI transactions, we log the KV-pairs updated or created by SI transactions as LOG_VN_TXN. It has a similar format to LOG_VN except it contains some transactional information of the VN. More specifically, *LastLoggedLSN* is the LSN of the previous data log that belongs to the same transaction as the current log. *LastCommittedLSN* in LOG_VN_TXN is the key field to provide MVCC support in the sequential storage: it is the LSN of the last committed version of the VN before the transaction which generates this log starts. We discuss how this field helps implement SI in transaction processing in Section III-F.

F. Transaction Manager

One of the most important features of FlashTKV is its efficient support of Snapshot Isolation (SI).The SI for a KV-store means a transaction T never sees the modifications of KV-pairs done by other transactions that start later than T. Since the LSN used by the entire system is monotonically increasing, we use it as the timestamp to decide the order of transactions. More specifically, when a transaction starts, we use its first LSN as the start timestamp of the transaction.

1) KV-pair Operations in SI Transactions: Because all the KV-pairs in FlashTKV are contained in LOG_VN/LOG_VN_TXN logs, the transactional KV-pair operations only affect the access of LNs and VNs. For

 ISBN: 978-1-61208-226-4

the KV-pair retrieval, the correct version of the KV-pair is located by following the *PrevVerLSN* in the logs. For the KV-pair insertion/update/deletion, the VN with new value is logged using LOG_VN_TXN. It contains *LastCommittedLSN*, the LSN of the last committed version of this VN to be seen by this SI transaction, *TxnID*, and *LastLoggedLSN*. These fields are used later to (1) check whether the transaction can commit or not and (2) undo the aborted transaction.

2) Transaction Commit: We adopt the *First-Committer-Wins* rule[18] to decide whether a transaction can be committed or not. In FlashTKV, the rule requires that an SI transaction T can commit only if all the KV-pairs it wrote are not written by any other committed transactions that started later than T. More specifically, when an SI transaction wants to commit, for each LOG_VN_TXN it generates, we check the corresponding VN to see if the *LastCommittedLSN* is the same as that in the data log. If all of them are the same, the SI transaction can commit, otherwise, FlashTKV automatically aborts it. If a transaction commits, we write a LOG_TXN_COMMIT log to the log write buffer and flush it.

3) Transaction Abort: To abort an SI transaction, we must undo all the changes the transaction made. Before the undo, we add a LOG_TXN_ABORT log and flush it to make sure the transaction will be aborted even if a crash happens during the undo. More specifically, for each LOG_VN_TXN it generates, if the current LSN of the corresponding VN is the same as the LSN of the data log, we set its LSN to the *PrevVerLSN* in the data log.

G. Checkpoint and Recovery

The recovery of FlashTKV is quite different from those storage systems that contain data pages: It involves rebuilding the B^+-tree with all KV-pairs updated or inserted by committed transactions. Similar to the traditional DBMSs, we do checkpointing to help reduce the recovery time.

The checkpointing in FlashTKV flushes the following data logs to disk: (1) the LOG_CKPT_START log, (2) the LOG_LN log for a dirty LN, and the LOG_IN logs for the INs that are ancestors of a dirty LN, (3) the LOG_TS_MAP log, and (4) the LOG_CKPT_END log.

The recovery in FlashTKV starts by a backward scan of the data logs. The scan stops immediately after find the most recent checkpoint. Then, starting from the end of the checkpoint, we scan the data logs forward to replay all the data logs for INs/LNs to reconstruct them. Finally, we start from the *FirstActiveLSN* in the LOG_CKPT_END log to undo (redo) all the VN logs from uncommitted (committed) transactions.

H. Garbage Collection

In FlashTKV, the data logs for INs/LNs/VNs may become obsolete when the corresponding nodes are updated. To recycle the disk space used by those obsolete data logs, we perform garbage collection (GC) on those log files in which most of the data logs are obsolete (default is 70% in FlashTKV). BDBJE proposed a solution to recycle a data log file in the sequential storage scheme: the system copies the non-obsolete data logs in the file to a new place before erasing the entire file. However, this requires the exclusive locks on those data logs which violates the design principle of FlashTKV that reads are never blocked. In addition, we cannot block all the SI transactions during the GC because FlashTKV is designed for OLTP workloads that usually have response time requirement (such as TPC-C). Therefore, there are two main challenges for doing GC in FlashTKV: (1) how to determine whether a data log is recyclable when there are some active SI transactions and (2) how to recycle a file without exclusive locks.

We propose a novel approach to do GC in FlashTKV. For the first challenge, we observe that if the up-to-date version of the data is already visible to the oldest active transaction, all the previous versions of the data are safe to be recycled. Therefore, we use an array, called *GC-array*, to keep track of all committed updates (old and new LSNs). The old versions of the data are marked obsolete only when the up-to-date versions of the data are visible to the current oldest active transaction. For the second challenge, we treat the copying of unrecyclable data logs as the update of the corresponding INs/LNs/VNs with the some content, and group those updates into a normal SI transaction, called *GC-transaction*. As long as *GC-transaction* commits, the log file can be erased. *GC-transaction* always restarts automatically when it aborts because of other SI transactions. Note that the abortion may not only happen to the *GC-transaction*, but also the user transactions due to the commit of the *GC-transaction* under First-Committer-Win rule. However, our experiments show that the number of transaction abortion caused by *GC-transaction* (<0.04%) is neglectable compared to the normal abortion rate for TPC-C (≈0.5%). Details can be found in Section V-E.

IV. I/O Cost Comparison

We compare the time cost of all the I/O operations in traditional page storage scheme and sequential storage scheme in Table III. We only discuss the I/O operations during the normal execution. In other words, the I/O during the recovery, checkpoint, and garbage collection is not included because these operations are not frequently executed. The comparison is based on the workloads that do not involve any scan and all queries can be processed through indices, e.g. TPC-C.

In a traditional storage scheme, data pages contain all the data and transaction logs are stored separately. Under a transactional read-write workload, the database system using the traditional storage scheme may produce physical I/Os in three ways during the normal execution: (1) Page read due to

 ISBN: 978-1-61208-226-4

Table III: Comparison of I/O Operations in Page Storage and Sequential Storage

Storage Scheme	I/O Operation	I/O Time
Page Storage	Dirty Page Flush	T_{RW}
	Txn Log Flush	T_{SW}
	Page Read	T_{RR}
Sequential Storage	Data Log Flush	T_{SW}
	Data Log Read	T_{RR}

T_{SW} : I/O time of writing a flash page sequentially
T_{RR} : I/O time of reading a flash page randomly
T_{RW} : I/O time of writing a flash page randomly

page buffer miss (random read), (2) dirty page flush (random write) and (3) transaction logs flush (sequential write). In our sequential storage scheme, however, there are no data pages, instead, data is encapsulated in the *data logs*. As a result, there are only two ways to produce physical I/Os: (1) Data logs read due to node buffer miss (random read) and (2) data logs flush (sequential write). Note that the random write I/Os generated by flushing dirty pages in the page storage scheme no longer exist in the sequential storage. This is because all the updates are transformed into data logs which are appended to the log write buffer sequentially.

V. EXPERIMENT

In this section, we first compare the performance of the sequential storage scheme and the traditional page-based storage scheme on synthetic workloads with different read-write ratios. Then we quantify the performance impact of the locking-based concurrency control on the flash SSDs under workloads with different degrees of read-write lock contentions. Finally, we compare the overall performance of our FlashTKV with two well-known KV-stores, Berkeley DB (which uses the page-based storage) and Berkeley DB Java Edition (which uses the sequential storage).

A. Experimental Settings

1) Hardware: All of our experiments run on a Dell R410 server with a 2.7GHz Intel Xeon E5520 CPU and 8GB RAM. In the server, we have a 150GB 15000RPM magnetic disk connected with the SAS interface, and a 64GB Intel X25-E flash SSD [19] connected through the SATA interface. The hardware specification and detailed I/O performance of the storage devices are listed in Table I.

2) Software: The operating system is CentOS 5.2 Final (kernel version 2.6.18-92.el5). We use Ext3 of Linux as the default file system and the file system cache is disabled to stress the I/O performance. GLib 2.30 is used in the FlashTKV library. For comparison, we use BerkeleyDB (BDB) 5.0.32 and BerkeleyDB Java Edition (BDBJE) 4.0.71 as the representatives of the page storage and sequential storage accordingly. We modify an existing TPC-C [20] implementation [21] to work for FlashTKV, BDB and BDBJE.

Table IV: Workloads for I/O Time Comparison of The Page-based Storage and The Sequential Storage

Workload	No. of Retrievals	No. of Updates
A1	10,000	0
A2	7,500	2,500
A3	5,000	5,000
A4	2,500	7,500
A5	0	10,000

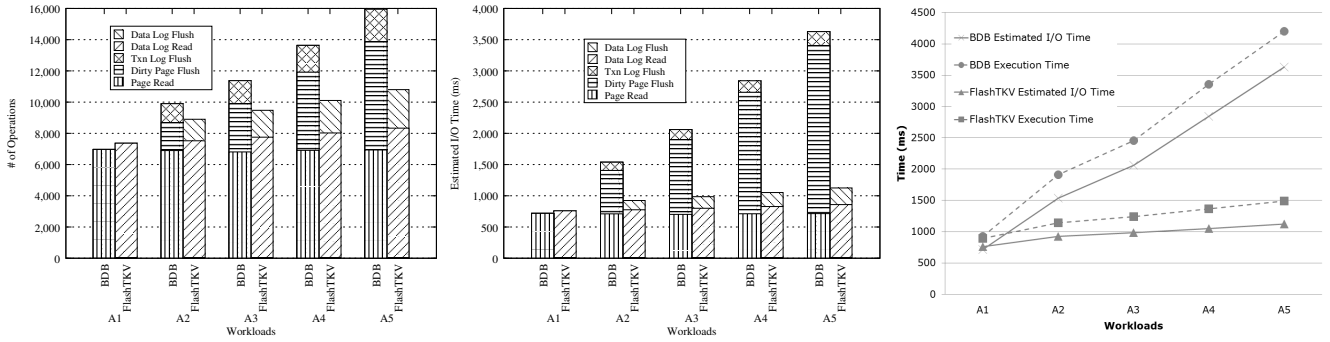
Table V: Workloads for Comparing MVCC and Locking On Flash SSDs

Workload	Key Range For Retrieval
B1	1 - 8000
B2	1 - 4000
B3	1 - 1000

3) Workloads: The workloads used for comparing the page storage and sequential storage are listed in Table IV. The synthetic workloads are generated in a database with 100 million key-value pairs (around 10GB in size). The benchmark is a single-threaded program that generates a sequence of KV-pair retrieval/update (read/write) operations with random keys in BDB or FlashTKV with every 10 operations forming a transaction. We modify the source code of BDB and FlashTKV so that it can count the total number of each kind of operations listed in Table III and we observe almost no performance degradation caused by the modification compared with the original system. Both BDB and FlashTKV have a 2GB memory buffer (page buffer/node buffer) and a 30 minutes warm-up time before counting the operations. The counting lasts for 10,000 KV-pair operations for all the workloads.

The workloads used for comparing MVCC and locking on flash SSDs are listed in Table V. The workload is generated in a database with 10,000 key-value pairs (around 1MB in size). The workload consists of the writer threads and the reader threads. Each writer thread is responsible for repeatedly updating a subset of KV-pairs. The key range of the KV-pairs each writer thread updates is equal to the number of KV-pairs divided by the number of the writer threads so that we can guarantee there are no write-write conflicts. In our case, every writer thread updates 100 KV-pairs. Each reader thread continuously picks one KV-pair with a random key to retrieve and each retrieval operation forms a read-only transaction. In our experiments, we change the key range of the retrieval operation to get different degrees of read-write conflicts. Note that we use a 100MB buffer which is much larger than the data size (1MB) so that there are no other write I/Os than the transaction logs flush. The big buffer also guarantees that there is always enough space in the buffer to hold the old versions for MVCC. Furthermore, we bring all the KV-pairs into the buffer before measuring the total time of the workloads so that we can eliminate the impact of the read I/O caused by

 ISBN: 978-1-61208-226-4



(a) Total Number of Different I/O Operations in Berkeley DB and FlashTKV

(b) Estimated I/O Time of Berkeley DB and FlashTKV on The Flash SSD

(c) Comparison of Estimated I/O Time and Execution Time on The Flash SSD

Figure 5: Page-based Storage v.s Sequential Storage

the buffer miss in the locking-based system and focus on the lock waiting time and the I/O time of the transaction log flushes.

Table VI: TPC-C Workload Settings

Workload	Scale	Database Size	Buffer Size
C1	100W	12GB	512MB
C2	100W	12GB	2GB
C3	100W	12GB	4GB
D1	100W	12GB	4GB
D2	200W	24GB	4GB
D3	300W	37GB	4GB

The workloads used to measure the overall performance are described in Table VI. TPC-C workloads have a large number of concurrent random read/write operations. There are three types (C1, C2, C3) of TPC-C workloads with a fixed database size and different buffer sizes. There are another three types (D1, D2, D3) of TPC-C workloads with a fixed buffer size and different database sizes. Note that by default, FlashTKV uses MVCC for transaction processing, therefore we show the performance of the locking-based FlashTKV only to quantify the impact of the programming language of the storage systems when comparing with BDBJE.

B. Comparison of The Page-based Storage and The Sequential Storage

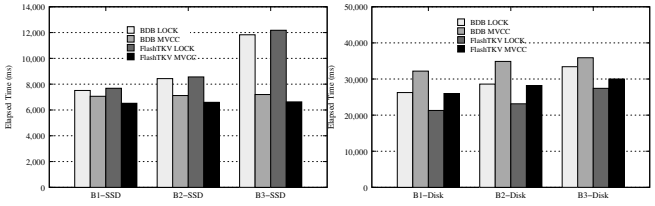
To quantify the benefit of using the sequential storage instead of the page-based storage on flash SSDs, we count the total number of each operation listed in Table III under synthetic workloads with different read-write ratios. Figure 5a shows the total number of each kind of operations listed in Table III under the synthetic workloads described in Table IV. Under the synthetic workloads, both BDB and FlashTKV have a similar buffer miss rate since they use the same buffer replacement policy, LRU. Because the read-only workload does not generate any LN delta logs, the numbers of the page read I/O and the node read I/O number are almost the same. This indicates even FlashTKV uses an entirely different storage scheme from BDB, the node-based buffer strategy can achieve a similar performance to the traditional page-based buffer strategy. As the workload becomes more write-intensive, the dirty page flush in the BDB increases but the transaction log flush remains the same because we only flush the transaction logs when the transaction commits and the number of transactions for each workload is the same. In FlashTKV, the number of data log reads for buffer miss also increases when the workload becomes write-intensive. This increase is because LN delta logs may incur one to two data log reads for each LN retrieval. However, this increase is moderate because LNs are likely to be hold in the memory. Different from BDB where the number of transaction log flushes remains almost the same, the number of data log flushes in FlashTKV increases slightly because LNs may also be flushed to the data log.

Based on the I/O performance of the flash SSD we use, we can derive the three hardware-related parameters in Table III by taking the read/write latency into account, in the worst case, $T_{RR} = 103\mu s$, $T_{RW} = 388\mu s$, $T_{SW} = 108\mu s$. We then calculate the total I/O time of the workloads shown in Figure 5a according to Table III. As shown in Figure 5b, by avoiding the random writes, the most expensive operations in the flash SSD, FlashTKV can achieve a higher performance than BDB under read-write workloads on the flash SSD. More specifically, the more write-intensive the workload is, the more performance speedup FlashTKV gains over BDB, e.g., about 3x speedup for the write-only workload. We compare the estimated I/O time and the execution time of BDB and FlashTKV on all the workloads in Figure 5c. The difference between the estimated I/O time and the execution time for each storage engine can be accounted to the CPU and memory access time. The difference is small, which indicates that in both FlashTKV and BDB, the total execution time is dominated by I/O time. In both engines, our estimated I/O time follows the trend of the total

execution time.

C. MVCC Versus Locking

To quantify the negative impact of the read-write lock contention on the flash SSD, we implement a small benchmark to compare the performance drop when increasing the degree of the read-write conflict.



(a) Comparison of MVCC and Locking on SSD with Degree of The Read-write Conflict Varied

(b) Comparison of MVCC and Locking on Disk with Degree of The Read-write Conflict Varied

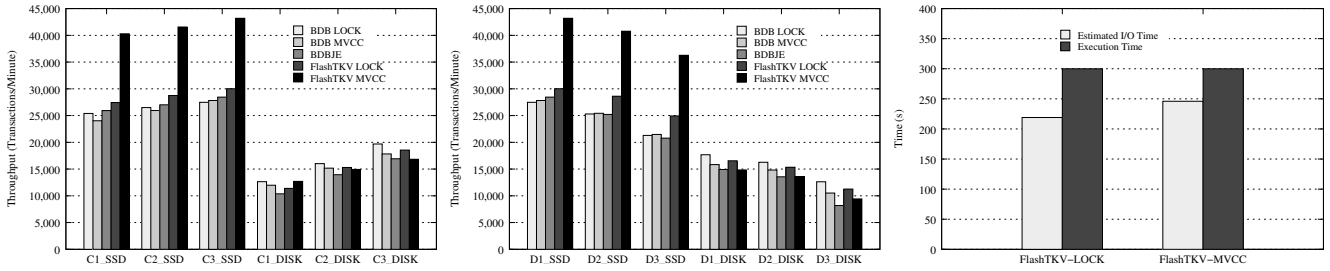
Figure 7: MVCC v.s Locking

Figures 7a and 7b show the total elapse time of BDB and FlashTKV running the workloads in Table V on the SSD and the magnetic disk. Note that in the order of workload B1, B2, and B3, the degree of the read-write conflict increases. Under all of these three workloads, with the same storage engine, locking always outperforms MVCC on the magnetic disk. This is because the random read performance is much worse than the sequential write performance on the disk. As a result, the I/O time spent on the random reads for multiple versions of data in MVCC is more than the time of waiting for the log flushes (sequential writes). In contrast, on the flash SSD, the MVCC version always wins. This is because on the SSD, the random reads for multiple versions of data cost much less than waiting for the sequential writes. This result suggests that on the flash SSD, MVCC is better than the locking-based concurrency control under workloads with read-write conflicts.

D. Overall Performance

We compare the overall performance of our FlashTKV with a sequential storage engine (BDBJE) and a page-based storage engine (BDB with MVCC) by measuring the throughput under TPC-C workloads with different database sizes and buffer sizes. As shown in Figure 6a, on the flash SSD, MVCC-based FlashTKV always outperforms other storage engines. However on the disk, MVCC-based FlashTKV has a similar performance to others, and is even worse when the buffer gets larger. This is because the extra read I/Os used to retrieve old versions of KV-pairs cannot be saved by increasing the buffer size.

Figure 6b compares the performance among the storage engines with different numbers of warehouses. BDBJE and locking-based FlashTKV are very similar in both the storage scheme and concurrency control approach, but there is about 20% performance difference in D3 workload. This performance difference is mainly due to the platform (Java versus C) and implementation. Furthermore, as shown both in Figure 6a and 6b, the flash SSD substantially helps increase the overall throughput of the storage engines. Due to the poor performance of the random read on the magnetic disk, the performance of the sequential storage engines, including both BDBJE and FlashTKV, is even worse than the page-based storage engine BDB on the magnetic disk. On the SSD, however, FlashTKV outperforms the other two storage engines, achieving a speedup of 1.68x over BDB, and 1.54x over BDBJE. We count the numbers of the I/O operations in FlashTKV under Workload D1 for five minutes and compare the estimated I/O time with the execution time in Figure 6c. As one can see, the estimated I/O time is 73% and 82% of the execution time in locking-based FlashTKV and MVCC-based one, respectively. This indicates that the TPC-C workload in FlashTKV is still I/O dominant. Therefore, the performance improvement is mainly because the I/O time is reduced in the sequential storage.



(a) Throughput Comparison among Different Storage Engines with Varied Buffer Size

(b) Throughput Comparison among Different Storage Engines with Varied Database Size

(c) Comparison of Estimated I/O Time and Execution Time of FlashTKV on SSD under Workload D1

Figure 6: Overall Performance Comparison

 ISBN: 978-1-61208-226-4

E. Garbage Collection

Since GC in FlashTKV may introduce more transaction abortion, we quantify the impact of GC on the overall performance by counting the total number of transaction abortion because of the *GC-transaction*. We run the TPC-C workload C3 with and without GC for 2 hours.

Table VII: Comparison of The Transaction Abortion under Workload C3 with And without GC

	Without GC	With GC
Total # of Transactions	5,181,842	5,166,385
New-Order Abortion	24,974	25,753
GC-transaction Abortion	/	504
Other transactions Abortion	/	608
Overall Abortion Rate	**0.48%**	**0.52%**

As shown in Table VII, without GC, there is a 0.48% abortion rate for New-Order transactions in TPC-C and no abortion of other transactions. With GC, the number of New-Order transaction abortion slightly increased. In addition, the *GC-transaction* and some other transactions (such as Payment or Delivery) also have abortion. However, the total abortion rate of all the transactions only increased 0.04% which is neglectable compared to that without GC.

VI. Conclusion and Future Work

In conclusion, we have presented the design and implementation of FlashTKV, a transactional KV-store optimized for flash-based solid state drives. The two main features of FlashTKV are (i) a sequential storage format that stores logs as data and also incorporates transactional information; (ii) Snapshot Isolation transaction support through MVCC on the sequential storage. We have evaluated FlashTKV in comparison with both BerkeleyDB C version (BDB), which has a page-based storage layout, and Java version (BDBJE), which has a sequential storage layout with locking based concurrency control. Our results show that, under TPC-C workloads, while FlashTKV is slightly worse than BDB with locking on magnetic disks, it outperforms its competitors by 70% in throughput on flash SSDs. Based on these results, we believe that our sequential storage format with MVCC is a promising approach for transactional key-value stores on flash disks.

References

[1] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe, "Query Processing Techniques for Solid State Drives," in *SIGMOD Conference*, 2009, pp. 59–72.

[2] Y. Ou, T. Härder, and P. Jin, "CFDC: A Flash-aware Replacement Policy for Database Buffer Management," in *DaMoN*, 2009, pp. 15–20.

[3] Y. Lv, B. Cui, B. He, and X. Chen, "Operation-aware Buffer Management in Flash-based Systems," in *SIGMOD Conference*, 2011, pp. 13–24.

[4] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh, "Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices," *PVLDB*, vol. 2, no. 1, pp. 361–372, 2009.

[5] Y. Li, B. He, Q. Luo, and K. Yi, "Tree Indexing on Flash Disks," in *ICDE*, 2009.

[6] C.-H. Wu, L.-P. Chang, and T.-W. Kuo, "An Efficient R-tree Implementation Over Flash-memory Storage Systems," in *GIS*, 2003, pp. 17–24.

[7] C.-H. Wu, T.-W. Kuo, and L. P. Chang, "An Efficient B-tree Layer Implementation for Flash-memory Storage Systems," in *RTCSA*, 2003, pp. 409–430.

[8] S.-W. Lee and B. Moon, "Design of Flash-based DBMS: An In-page Logging Approach," in *SIGMOD Conference*, 2007, pp. 55–66.

[9] B. Debnath, S. Sengupta, and J. Li, "FlashStore: High Throughput Persistent Key-Value Store," *PVLDB*, vol. 3, no. 2, pp. 1414–1425, 2010.

[10] B. Debnath, S. Sengupta, and J. Li, "SkimpyStash: RAM Space Skimpy Key-value Store on Flash-based Storage," in *SIGMOD Conference*, 2011, pp. 25–36.

[11] Oracle, "Berkeley DB Products," 2010.

[12] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, 1992.

[13] C. Manning, "YAFFS: The NAND-specific Flash File System," 2002.

[14] D. Woodhouse, "JFFS: The Journalling Flash File System," in *Ottawa Linux Symposium*, 2001.

[15] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A Case for Flash Memory SSD in Enterprise Database Applications," in *SIGMOD Conference*, 2008, pp. 1075–1086.

[16] Oracle, "White Paper: Berkeley DB Java Edition Architecture," 2006.

[17] R. Bayer and M. Schkolnick, "Concurrency of Operations on B-Trees," *Acta Inf.*, vol. 9, pp. 1–21, 1977.

[18] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil, "A Critique of ANSI SQL Isolation Levels," in *SIGMOD Conference*, 1995, pp. 1–10.

[19] Intel Corp., *Intel X25-E SATA Solid State Drive Datasheet*, 2008.

[20] TPC, *TPC Benchmark C Standard Specification Revision 5.9*. Transaction Processing Performance Council, 2007.

[21] SYNAR, Simon Fraser University, "Tpc-c benchmark on bdb," <http://synar.cs.sfu.ca/systems/code/tpcc-bdb.tar>, 08.19.2012.

 ISBN: 978-1-61208-226-4