SHARE Interface in Flash Storage for Relational and NoSQL Databases

Gihwan Oh
Sungkyunkwan University
Suwon, Korea
wurikiji@skku.edu

Chiyoung Seo
Couchbase, Inc.
Santa Clara, USA
chiyoung@couchbase.com

Ravi Mayuram
Couchbase, Inc.
Santa Clara, USA
ravi@couchbase.com

Yang-Suk Kee
Samsung Semiconductor, Inc.
San Jose, USA
yangseok.ki@samsung.com

Sang-Won Lee
Sungkyunkwan University
Suwon, Korea
swlee@skku.edu

ABSTRACT

Database consistency and recoverability require guaranteeing write atomicity for one or more pages. However, contemporary database systems consider write operations non-atomic. Thus, many database storage engines have traditionally relied on either journaling or copy-on-write approaches for atomic propagation of updated pages to the storage. This reliance achieves write atomicity at the cost of various *write amplifications* such as redundant writes, tree-wandering, and compaction. This write amplification results in reduced performance and, for flash storage, accelerates device wear-out. In this paper, we propose a flash storage interface, SHARE. Being able to explicitly remap the address mapping inside flash storage using *SHARE* interface enables host-side database storage engines to achieve write atomicity without causing write amplification. We have implemented *SHARE* on a real SSD board, OpenSSD, and modified MySQL/InnoDB and Couchbase NoSQL storage engines to make them compatible with the extended *SHARE* interface. Our experimental results show that this *SHARE*-based MySQL/InnoDB and Couchbase configurations can significantly boost database performance. In particular, the inevitable and costly Couchbase *compaction* process can complete without copying any data pages.

1. INTRODUCTION

In the era of *all-flash* cloud computing data centers, it is urgent to flash-optimize open source database engines, including traditional relational DBMSs (*e.g.*, MySQL/InnoDB) and emerging NoSQL storage engines (*e.g.*, Couchbase). Considering the intrinsically asymmetric performance between flash storage read and write operations, optimizing flash-based database write efficiency is a practical and critical problem that should be immediately addressed.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2882910

When considering that data consistency is a uncompromisable database requirement, optimizing reliable writes can be an intrinsic problem. With Hard Disk Drives (HDDs), a database update action may involve multiple pages, each in turn usually consisting of multiple disk sectors. A sector write requires a slow mechanical process and is susceptible to power failures. If a power failure occurs in the middle of a sector write, the sector might be only partially updated. So, a sector write operation is considered non-atomic by most contemporary database systems [3, 24].

To guarantee write atomicity, which is critical for database consistency, many database storage engines have assumed HDDs as their main storage media and resorted to either *copy-on-write* or *journaling*. With a journaling approach, each updated page's *before* or *after* image is redundantly stored in a dedicated journal area. Many database engines such as MySQL/InnoDB, PostgreSQL, SQLite, and Sybase SQL Anywhere rely on a journaling approach [3, 4, 6, 31]. In contrast, a copy-on-write approach, inspired by the shadow paging technique [21], does not update data pages in place, but instead writes all updated pages to other persistent storage locations, while the original pages remain immutable and become stale. Consequently, copy-on-write needs to reclaim stale pages periodically. It should be noted that these two techniques for guaranteeing atomic write are heavily used - not only in enterprise database applications but also in modern file systems [13, 22] and mobile databases [17].

These out-of-place write schemes for achieving atomicity, however, suffer from write amplification. The journaling approach requires writing one logical page *redundantly* to the storage. With the copy-on-write approach, the costly *compaction* process (a form of garbage collection) should be carried out [7]. The *wandering-tree* problem is additionally encountered in a tree-structured NoSQL storage engine and file system metadata [32]. This type of logical write amplification at the host layers is often cited as a primary cause of tardy performance and also aggravates flash storage wear-out [16, 17].

Meanwhile, because flash memory does not allow overwriting any page in place, a flash storage page update is commonly carried out by writing the new content into a clean flash page at another location, leaving the original page untouched [9, 20]. Due to this *out-of-place* strategy, flash storage devices are equipped with a firmware module called the FTL (flash translation layer), whose main role is to maintain the mapping between kernel block layer logical addresses and flash memory chip physical addresses. And, mainly for performance reasons, most contemporary flash storage devices have adopted fine-grained page unit mapping [23]. Fortunately, this indirection of *logical to physical* address mapping, which has been necessarily adopted to overcome flash memory's *no-overwrite* limitation provides an excellent opportunity in obviating the write amplification overhead data write reliability requires.

In this paper, we present the *SHARE* interface. This interface exposes an abstraction that allows host applications to explicitly change the *address mapping* inside flash storage. With *SHARE*, two different logical pages can share one physical page. This simple interface allows the host applications using either a journal or copy-on-write approach to achieve write atomicity without incurring write amplification. It also allows NoSQL storage engines, including Couchbase, to carry out a compaction process without copying data. The key contributions of this work are summarized as follows.

- *SHARE* provides an abstraction which allows host applications to change the address mapping table which has traditionally been managed internally only by FTL. Many database engines and file systems can easily exploit *SHARE* to achieve write atomicity without causing write amplification. In addition to atomic writes, *SHARE* can also make other numerous write-heavy cases almost cost free. This includes the NoSQL compaction process and file copy operations that can occur almost without copying data.

- We have implemented *SHARE* on an open SSD development hardware platform called *OpenSSD* by enhancing its FTL code with the *SHARE* features this paper presents. We have demonstrated both relational and NoSQL storage engines can easily exploit *SHARE* with only minimal code changes.

SHARE resembles the well-known *TRIM* command [30]. Both *SHARE* and TRIM allow upper-layer applications to provide useful information to flash storage devices, thus enabling them to achieve better performance. The TRIM command is successfully established as a new standard flash storage interface because it effectively improves the flash storage garbage collection process by informing flash storage devices which data blocks are no longer valid. We believe that the *SHARE* command could be incorporated into operating system kernel as easily as the TRIM command, and its performance benefit would be no less than the TRIM command's. Also, as is contrasted in Section 6, to our best knowledge, *SHARE* is the first work on ameliorating write amplifications for database engines utilizing the out-of-place update strategy.

The rest of the paper is organized as follows. Section 2 describes the I/O inefficiency for guaranteeing data write reliability in database engines and presents the motivation for our work. Section 3 presents the architecture of *SHARE* and its abstraction, and discusses its advantages for database storage engines. Section 4 gives the implementation details of *SHARE* architecture. In Section 5, we evaluate the performance impact of *SHARE* on MySQL/InnoDB and Couchbase storage engines using LinkBench benchmark and YCSB benchmark, respectively. Section 6 compares our work with the existing flash storage write optimization techniques. Lastly, Section 7 summarizes this paper's contributions and suggests future work.

2. MOTIVATION

A crucial assumption for database consistency and recovery is that each individual page is atomically written to storage. However, secondary storage devices such as spinning disks and flash memory SSDs do not generally guarantee page write atomicity. Hence, when a system crashes when a page write is in progress, the on-disk copy of the page may contain a mix of old and new data after system reboots. Such a *torn* page, however, cannot be completely restored - even with *Aries-style* write ahead logging (WAL) schemes [6, 24]. Therefore, for better recoverability, database systems should be armed with a mechanism to guarantee the atomic propagation of updated pages, which is orthogonal to the Aries-like recovery protocol. And, as solutions to achieve the write atomicity in databases and file systems, there have been two popular approaches: journaling and copy-on-write. However, the high write amplification these two approaches cause becomes a major obstacle to achieving scalable, consistent performance for normal read and write operations.

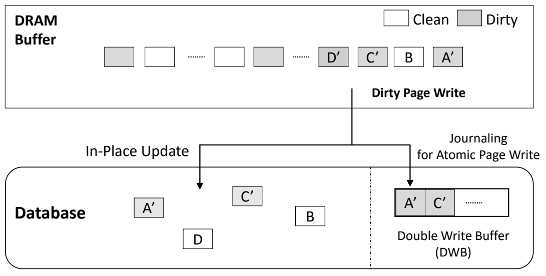
In this section, we explain how these two techniques are used in real database storage engines such as the MySQL/InnoDB server and the Couchbase storage engine. Then, we describe the definition of the problem this paper addresses and the opportunity flash storages provide.

2.1 Redundant Writes in MySQL/InnoDB

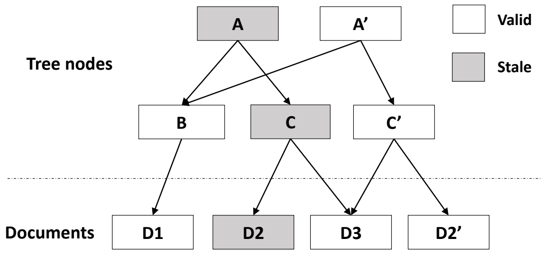
The MySQL/InnoDB storage engine takes a variant of journaling, called *double-write* [4], to deal with the partial page write problem. As shown in Figure 1(a), when a updated page is evicted from the buffer cache, prior to overwriting the old copy in its original database location, the InnoDB engine first appends its new copy (*i.e.*, *after-image*) to a separate journal area, *double-write-buffer*. And when the write to the journal area is forcefully completed (using `fsync` call), InnoDB writes the page to its original location. When the system recovers from a crash, InnoDB can always find a consistent page copy either in the database or in *DWB*.

The PostgreSQL server is also taking a redundant journaling approach to guarantee write atomicity. Specifically, when the server runs by default with the `full_page_write` option on, whenever a page is updated first after the last checkpoint, the *before-image* of the page is saved in the WAL log. It is also well known that SQLite, a popular embedded database system, provides two journaling modes, *rollback* and *write-ahead log*, to guarantee atomic page write, and the overhead of journaling either before-image or after-image of every updated page is very expensive [17].

This journaling-based redundant write paradigm has also been widely adopted in modern file systems such as `ext4` and `XFS` so as to guarantee the consistency of data and metadata despite the torn page problem [22]. However, because journaling both data and metadata pages (*i.e.*, `full` journaling mode) is too expensive, those file systems are by default configured to journal only metadata (*i.e.*, `ordered` journaling mode) so that the consistency of file system structure is at least preserved.



(a) Double Write Buffer in MySQL/InnoDB



(b) Copy-On-Write in Couchbase

Figure 1: MySQL/InnoDB and Couchbase Journaling and Copy-on-Write

2.2 Copy-on-Write in Couchbase

Couchbase is a document-oriented NoSQL database system. Its storage engine is based on the append-only B^+-tree that takes a copy-on-write strategy for updating document(s) atomically: rather than overwriting the existing old document copy in place, it appends its new copy at the end of the database file. As is illustrated in Figure 1(b), when a document is updated, its new copy is written to the end of a database file while its old copy is left intact, but marked as *stale*. One undesirable consequence of this append-only update policy is that all tree nodes on a path from a leaf pointing to the document to the root should be updated and written in a cascaded way, where each node update also takes a copy-on-write strategy. The Couchbase storage engine relies on the so-called *wandering-tree* [32] scheme. A tree may be called a wandering tree if an tree node update requires updating its parent nodes up to the root due to inability to perform in-place updates. Consequently, the wandering-tree scheme amplifies the data to be written with N index node pages, where N is the height of B^+-tree.

Despite the high *write amplification*, Couchbase deliberately adopted this copy-on-write and wandering-tree combination mainly for two purposes: 1. as modern storage devices do not support an atomic write feature, the copy-on-write strategy has been adopted as an implementation method to support atomic writes [21]. 2. Couchbase has opted for the sequential write pattern of the copy-on-write strategy over the random write pattern of the update-in-place strategy. In particular, because spinning disk write latency is mainly dominated by the mechanical arm movement(*e.g.*, disk seek time), the write strategy Couchbase adopted can provide better write throughput than the traditional update-in-place policy.

As more documents are updated or newly-inserted under the copy-on-write and wandering-tree scheme, the percentage of stale database file pages increases. When the ratio of stale data reaches a configured database file threshold, the costly *compaction* operation is necessarily invoked to reclaim the unused space the stale data occupies. For this, the compaction task typically reads all non-stale documents and index nodes from the current database file and copies them to a new file. This incurs significant I/O overhead and write amplification. The current database file is later deleted when no longer accessed by any reader. Other NoSQL database systems, such as BigTable [11], Cassandra [19], and MongoDB [1] that adopt a Log-Structured Merge (LSM) tree [25] as their underlying storage engines have the similar issue.

2.3 Problem Statement

As aforementioned, journaling and copy-on-write commonly implement a two-phase write scheme to avoid the torn-write problem: perform a first write for recording updates (*e.g.*, *double-write-buffer* in the case of InnoDB and copy-on-write in the case of Couchbase) and perform a second write for applying the recorded writes to live data (*e.g.*, in-place update in the case of MySQL/InnoDB and compaction in the case of Couchbase). As a result, these systems double storage update write and thus the effective available user bandwidth is halved.

In particular, when database systems run on flash storage devices rather than on hard disks, the performance effect of this write amplification becomes more serious. A NAND flash page can be written only when it is in clean, empty state. Once it is written, it must be erased before it is rewritten. However, online erasure is impractical due to its several millisecond long erase time. Instead, SSD employs FTL that allows logical address to map to different physical locations, deferring page erasures. However, the stale pages eventually require a later clean up process. This garbage collection operation generates additional internal traffic (*i.e.*, read and write amplification) to relocate valid pages, causing a significant IO operations jitter. This paper refers to this activity as *copyback*. The more update data is written, the more garbage collection is required. That is, user writes are further amplified when the storage system uses SSD devices. Further, this undesirable write amplification shortens flash storage device lifespan.

Whether intended or not, the net effect of the copy-on-write operation inside flash storages remarkably resembles what database storage engine's out-of-place write schemes achieve. This provides an excellent opportunity to achieve the write atomicity almost at no cost of redundant writes, the tree wandering, and the compaction process. Based on this observation, we aim to develop a simple, but powerful abstraction at the flash storage level that allows database storage engines to avoid write amplification across storage systems and storage devices. Specifically, this interface allows host-side applications to explicitly change the address mapping inside flash storage.

3. SHARE INTERFACE

In this section, we present a new solution called *SHARE* that supports the atomic page writes at the flash storage layer, so that upper layers such as database and NoSQL engines can be freed from the burden of write amplifications.

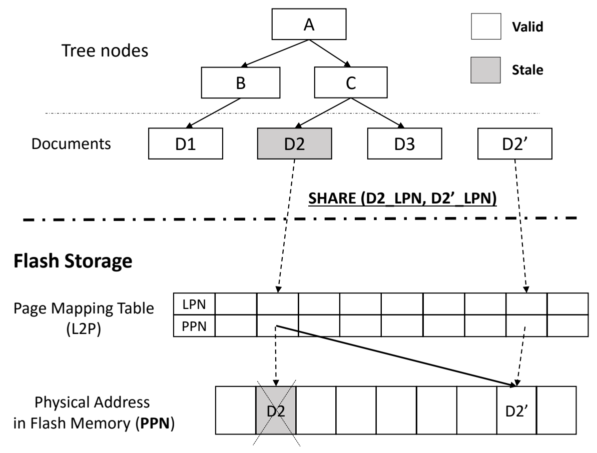


Figure 2: *SHARE* Architecture

The architecture, the interface, and the benefits of *SHARE* will be explained in turn.

3.1 SHARE Architecture

The solution discussed in this paper is based on the holistic understanding of problems across the layers of application, OS, and storage device. The FTL's *page-grained indirection* of logical-to-physical addressing mapping provides an excellent opportunity for solving the write amplification problem occurring in both the journaling and copy-on-write approaches. Let us first illustrate how we can, by exploiting the unique FTL page-level address mapping feature, avoid the Couchbase write amplification without compromising atomic write semantics. As is shown in Figure 2, when a new copy of document D2, denoted as D2', is written to flash storage media, each of two document copies, D2 and D2', has its own logical and physical address. Now, *what if the logical address of D2 in FTL can be remapped to point to the physical address of D2'?* The new D2' copy can be reached through D2's LPN. Hence, unlike the original Couchbase, all index nodes along the path from the corresponding leaf node to the root node need not to be copied-on-write. That is, the write amplification due to Couchbase's wandering-tree scheme can be totally avoided while still preserving the atomic write semantics required to update document D2. Similarly, with MySQL/InnoDB *double-write-buffer*, the redundant write of an updated page to the original location in the primary database can be avoided simply by FTL changing the logical address of its original location to point to the physical addresses of new copy written in the *double-write-buffer*, without losing the atomic write property.

From the examples described above, we know that write amplification can be completely avoided by providing upper-layer storage engines with an adequate abstraction (or API) which allows changing the logical-to-physical page mapping table flash storage devices internally manage.

3.2 SHARE Interface

The remaining issue is, then, how upper-layer applications can inform FTLs to enable two LPNs to share one physical address. For this purpose, we propose a new interface called `SHARE(LPN1, LPN2)` for flash storage. Upon receiving a *SHARE* command with a first LPN (LPN1) and a second LPN (LPN2) for an update, wherein the first LPN maps to a first PPN and the second LPN maps to a second PPN, FTL atomically remaps the first LPN so that the first LPN maps to the second PPN, trimming a mapping of the first LPN to the first PPN [30]. Since there is no matching command in current storage interface such as SATA, the `share` command has been added as a SATA vendor unique command as summarized below.

share(LPN1, LPN2, length) This is a new, added SATA command. When host issues a *SHARE* command, FTL changes the physical address mapped to `LPN1` to the physical address currently mapped to `LPN2` so both logical addresses share one physical data address. The third argument, `length`, is optionally used when the length of data to be shared is longer than the FTL mapping granularity. The length must be a multiple of mapping unit size. When the length is larger than 1, the range between LPN1 and LPN1+length cannot be overlapped with the range between LPN2 and LPN2+length.

Since the SATA command set is not always available to database storage engines and other applications that access files through a file system, we exploited the `ioctl` infrastructure so that the *SHARE* command can pass through the file system to the storage device, instead of invoking the SATA command directly from applications.

Even though our description so far assumes a SHARE command associated with a single pair of LPNs, this concept can be naturally extended to multiple LPN pairs in a batch. In this case, FTL should be able to support the atomic address remapping for the given set of LPN pairs upon a system crash or power-off failure. In this regard, the idea of *SHARE* resembles the shadow page technique [21], but it offloads the shadow paging overhead to the flash storage firmware, where the transactional atomicity can be implemented much more efficiently. Section 4 will describe one way to implement the *SHARE* command atomically. This batch SHARE operation can reduce the non-negligible round-trip overhead in the IO stack of issuing the command via ioctl [10]. In addition, this batch can reduce the number of potential flash writes to persist the updated mapping information.

3.3 SHARE Advantages

To the best of our knowledge, *SHARE* is the first work which exposes the existence of address mapping table inside flash storage to host applications, allowing them to explicitly change the mapping for their own purposes. The advantages of *SHARE* are threefold. First, *SHARE* exploits the address mapping mechanism every flash-based storage device uses [9], so that upper layer applications can complete the costly operations such as transactional atomicity and compaction at low cost with minimal writes. Second, since *SHARE* command semantics are simple and clear, it is easy to incorporate them into the existing storage interface frameworks (*e.g.*, using the vendor specific command) and does not require involving intermediate layers such as the kernel block layer. So standards such as SATA, SAS, NVMe, and FC (Fiber Channel) can incorporate the *SHARE* abstraction with minimal changes. Third, upper layer applications including MySQL/InnoDB and Couchbase storage engines can use the *SHARE* service with marginal code changes, and further those changes are limited to modules interacting flash storage via the *SHARE* command.

Our *SHARE* scheme is novel in that it attempts to convert a flash memory weakness (*i.e.*, being unable to update in place and thus maintaining another level of address mapping) into a strength (*i.e.*, atomic write without write amplification and zero-copy compaction). This enables applications with strict atomic write requirements to achieve low-cost transactional support as well as minimal write amplification. Moreover, reducing flash storage write prolongs a flash storage device's life span.

Compaction by SHARE

Let us explain how the *SHARE* interface can be effectively used in mitigating the costly compaction overhead in Couchbase. First, the compaction operation is less frequently triggered in Couchbase. Because the *SHARE* interface replaces the cascaded updates of index nodes with a simple host interface command, write amplification caused by the wandering tree in the original Couchbase can be avoided. Hence, the database file is filled with invalid pages at slower pace by the *SHARE*-assisted Couchbase than by the original Couchbase. Second, and more importantly, the *SHARE* interface can boost the compaction process itself. Given that all write transactions in most key-value stores slow down during database compaction, it is crucial to complete compaction as fast as possible.

When a compaction operation is triggered in the original Couchbase, it creates a new database file and copies all valid documents from the old database file to the new file. In contrast, with the *SHARE* interface, as depicted in Figure 3, compaction can proceed without copying any valid documents. That is, after creating a new file, it first allocates new address space for the new database file (*i.e.*, using `fallocate()` system call) and then invokes the *SHARE* command to make the valid document pages in the old file shared by the new LPN addresses in the new file. After the *SHARE* command is successfully completed, the only remaining thing is to build up tree index nodes for new database file.

Other Applications of SHARE

Though the benefit of *SHARE* will be, throughout this paper, demonstrated with two database systems, MySQL/InnoDB and Couchbase, we believe that the idea of *SHARE* is easily and generically applicable to other database systems such as PostgreSQL and SQLite. For example, if *SQLite* is minimally modified to run on *SHARE*, it can achieve the atomic propagation of updated pages in a transaction without using the costly journaling of either *rollback* or *write-ahead log* mode. Instead, it can simply turn them off, because *SHARE* supports transactional atomicity and durability at the storage level. Also, *SHARE* can be leveraged in journaling or copy-on-write file systems [13, 22] suffering from redundant write and tree-wandering problems.

4. IMPLEMENTATION

This section details the *SHARE* architecture and its prototype implementation. Most of the *SHARE* architecture is implemented as an OpenSSD development board firmware extension. New commands for the *SHARE* features are prototyped via vendor unique commands. A user-level library that implements a protocol for the new commands via the

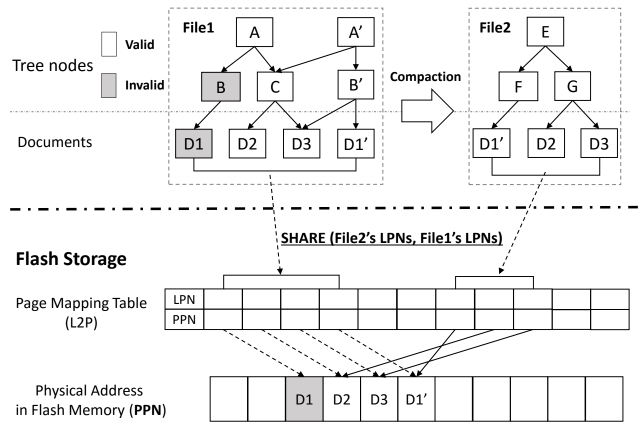


Figure 3: Couchbase Compaction with *SHARE*

ioctl system call supports applications and SSDs. This approach not only allows quickly prototyping the concept in development environments, i.e., Linux with ext4 file system, but also to make the prototype portable to most file systems. Please note that application changes to leverage the *SHARE* features are minimal - a few tens of lines of codes in Couchbase and MySQL/InnoDB total.

4.1 OpenSSD

OpenSSD is the first open development platform for SSD hardware/software. Its primary intent is to promote academia research [2, 17]. The first generation was introduced in 2011, and its third generation has been available since late 2014. This paper relies on the first generation, and the discussion in this section is limited to the first generation. Newer generation details are available at OpenSSD's project web site [2]. Since the WAF problem this paper discusses is independent of SSD performance, even though the actual performance observed by the end user can be influenced, this paper's claims are valid regardless of storage devices.

The first OpenSSD generation employs the *barefoot* controller and Samsung K9LCG08U1M NAND flash chips. The *barefoot* controller is a commercial product developed by Indilinx which was acquired by OCZ which was acquired by Toshiba Corporation. The controller is based on a 87.5MHz ARM processor and has a 96KB SRAM for the firmware and a 64MB Mobile SDRAM for metadata such as mapping tables. The type of NAND Flash is MLC (Multiple Level Cell) which contains two bits per cell. For flash memory management, a simple page mapping scheme is adopted, which is the most common management scheme used in contemporary SSD products and eMMC flash memory cards. We discuss more about FTL (Flash Translation Layer) and page mapping in the next section. The OpenSSD board is connected to the host system through a SATA interface.

4.2 Firmware Extension

4.2.1 SHARE *command*

A SSD FTL implements an indirection layer between a logical address space the host system sees and physical address space the FTL uses to store data on flash media. The most popular scheme is a page mapping which implements a mapping table that maps a logical NAND page (LPN) to a physical NAND page (PPN). When NAND Flash media page data is updated, the new content is written to a new physical NAND page and the mapping entry for the logical NAND page is updated to point to the new physical NAND page. We call this *forward mapping* or *L2P (logical page to physical page) mapping* in this paper.

The *SHARE* leverages this existing indirection. Instead of writing identical content twice for data consistency - as implemented in Couchbase compaction, InnoDB double write, and ext4 journaling - *SHARE* makes two logical NAND pages linked to the same physical NAND page, avoiding the second write.

This simple high-level description, however, imposes several technical challenges on firmware design and development. First, to preserve data consistency in the presence of failure, the sharing operation must be atomic. That is, the logical page must point to the old physical page if the operation fails, or to the new physical page if it succeeds. Second, a physical page can be accessed by two logical pages, complicating the SSD garbage collection process. A physical page can be recycled only when it is not referenced by any logical page. A simple solution to this problem is to maintain reverse mapping information for each physical page. If a physical page has a valid P2L (physical page to logical page) mapping and the logical page referenced by the mapping has a valid L2P mapping to the physical page, the physical page is valid. Therefore, each physical page needs to maintain multiple reverse mappings when using *SHARE*. Third, these extra data structures may compete with a normal mapping table for SSD memory space and this can have performance implications. The internal DRAM capacity of modern SSDs is proportional to the NAND capacity: roughly 1MB DRAM per 1GB NAND is available. Most of the DRAM space is used by the forward mapping table and the remaining space is used for I/O buffers and cache. To minimize the performance impact, we trade a portion of cache space for the reverse mapping and the size of reverse mapping table is empirically determined for the OpenSSD platform while the entire forward mapping table is kept in DRAM. The size of the P2L reverse mapping table is determined by several factors: the amount of available memory, the frequency of *SHARE* operations, and the lifespan of shared pages. The current implementation maintains only a small number of entries: 250 entries (4KB) or 500 entries (8KB).

4.2.2 Atomicity

As discussed in previous sections, double writes or journaling implement atomic data updates. Therefore, *SHARE* command must atomically update both the normal L2P mapping and the reverse P2L mapping. Furthermore, when such an update spans over multiple NAND pages, a *SHARE* operation for the update also applies to multiple pages. As such, a *SHARE* operation must be transactional and prevent any partial mapping updates in order to emulate the database semantics.

The FTL in modern SSDs maintains the consistency of the mapping table by logging mapping table state changes - called a Delta in this paper, since a reliably persistent version, i.e., a base mapping table, was created [18]. When a page is updated, the page is first written to a physical page and the mapping table entry in memory is updated. Then, a mapping table entry for the L2P is written to a log. Once a log entry is created, the update is deemed persisted. If an SSD has an emergency power capacitor, the creation of an in-memory log entry guarantees persistency. Otherwise, the log entry must be flushed to flash. The modified portion of mapping table is flushed to flash and the log is truncated regularly to balance the update performance and the recovery overhead.

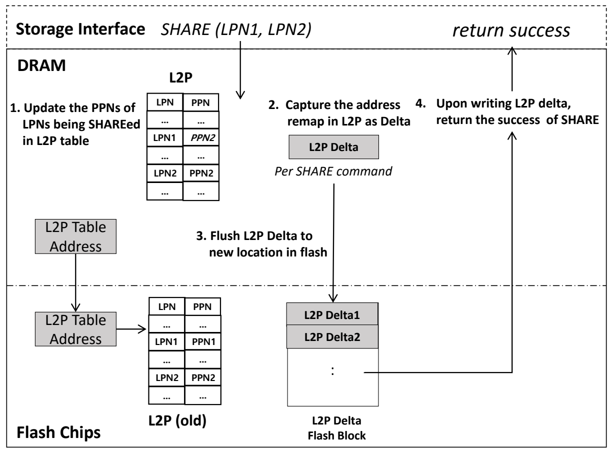


Figure 4: Atomicity in SHARE

For *SHARE*, the first step of writing to the physical page is skipped. Instead, the delta that includes the changes of reverse mapping is written to the log in a form of (LPN, old PPN, new PPN). Figure 4 summarizes the procedure of *SHARE* operation. First, when a `SHARE (LPN1, LPN2)` command is received, the FTL updates the PPN entry for LPN2 in the L2P mapping table with the PPN1 for the LPN1. Then it generates a Delta, (LPN2, PPN2, PPN1), and writes it to a log. If a *SHARE* command is associated with multiple pages, a series of deltas are logged after updating all the mapping table entries. Thereafter, the `share` command is considered completed atomically and persistently. The maximum size of Deltas cannot exceed the mapping page size because only a page is written atomically to flash for most SSDs [16]. The *SHARE* command returns after logging finishes.

4.3 Application Extension

This section describes the changes we made in MySQL/InnoDB and Couchbase to enable them to run on the *SHARE* interface. As will be described below, the changes in both databases are minimal.

Changes made in MySQL/InnoDB

In the original MySQL/InnoDB, as explained in Section 2, when evicting a set of dirty victim pages from the buffer cache, each page is written to its original location after forcefully writing another copy in the *double-write-buffer* area.

In contrast, when MySQL/InnoDB runs on *SHARE*, it simply calls the `share` command with the LPN pair(s) (the database file LPN and the double write buffer LPN) after the victim page set is successfully written in the *double-write-buffer* area, instead of writing each page redundantly at its original location. When the `share` call terminates normally, it is guaranteed that each page is propagated to its original location in its entirety. On a system reboot after system crashes, because the address remapping between a dirty page in the *double-write-buffer* area and its original location is atomically guaranteed at the storage level, the modified MySQL/InnoDB server can find a consistent copy of the page in either the database or *the double-write buffer* area. The code changes made in InnoDB storage engine were minimal - less than 200 lines of new code were added to just two modules of `buffer` and `file`.

Changes made in Couchbase

Even though the Couchbase was designed to minimize unnecessary storage writes, write amplification caused by its tree wandering problem is inevitable. When a document is updated in the Couchbase storage engine, its new version is appended and a leaf node pointing to the old version is updated to point to the new version. This leaf node update triggers the cascading updates of parents nodes that incurs high write amplification. We have adapted the Couchbase storage's `commit` API to use *SHARE* interface to avoid a leaf node update for each document, which consequently prevents the cascading parent node updates.

For the database recovery, if Couchbase terminates abnormally without failing the entire system, its original recovery process is applicable since the Couchbase storage engine only appends a new document at the end of the database file. In case of the entire system failure (*e.g.*, by a power outage), the original recovery process can be still performed as the SSD page mapping changes are persisted in an atomic way (Section 4.2.2).

In addition, Couchbase performs the background compaction to reclaim stale pages by reading all active documents from the old file and writing them to the new file. To avoid this heavy read/write I/O overhead, we have adapted the Couchbase storage's `compact` API, so that the LPNs of the documents in the old file and the new LPNs of the documents in the new file can be passed to the SSD FTL layer via the *SHARE* interface. Upon crashing during this compaction, the partially compacted new file is deleted and the whole compaction process restarts at system reboot. Couchbase's *SHARE* integration required quite small code changes - less than 500 line of new codes were added to `commit` and `compact` APIs.

5. PERFORMANCE EVALUATION

In this section, we present performance evaluation results to analyze the impact of the *SHARE* interface on relational and NoSQL storage engines.

5.1 Experimental Setup

The experiments were performed on a Linux platform with the 3.13.0 kernel running on an Intel Core i3-3220 3.3GHz processor equipped with 8GB DRAM. The host machine had one 4GB OpenSSD drive and one Samsung PM853T 240GB SSD. The OpenSSD drive was used as the main storage device implementing the *SHARE* interface. In the case of MySQL/InnoDB benchmark, the Samsung PM853T SSD was used as a database log device. Both the OpenSSD drive and Samsung PM853T SSD connected to the host using SATA interfaces.

We used the *ext4* file system in the `ordered` mode for metadata journaling. The direct I/O option(O_DIRECT) was enabled to avoid the effect of file system page caching. The benchmarking clients were run along with the database processes on the same hardware to exclude network latency in evaluation results. For more realistic experimental environment, an aging operation of the OpenSSD drive was pre-run so that the overhead of garbage collections was reflected in the result.

5.2 Workloads

For relational database systems, we used LinkBench workload [8] to conduct the performance evaluations on MySQL/InnoDB under the two configurations (i.e., double write buffer enabled, *SHARE* with double write buffer disabled). We included a NoSQL benchmark (YCSB benchmark [14]) to evaluate the impact of *SHARE* on the performance of a Couchbase NoSQL system in normal writes as well as compaction operations. As benchmark workloads, transactions of LinkBench and YCSB are much smaller than a traditional TPC-C workload. On the other hand, they are quite similar to a TPC-C workload in that both benchmark tools generate a large number of small random reads and writes. Below are described the characteristics of two benchmarks, LinkBench and YCSB.

LinkBench LinkBench is a configurable open-source database benchmark for a large-scale social graph [8]. This benchmark tool reflects the characteristics of Facebook social graph data where the majority of read requests are served by a caching layer. This reduces the temporal and spatial locality of read requests reaching the underlying database servers [19]. However, the overall workload reaching the database servers is still read-intensive with approximately 30% writes. The force index option was turned on to avoid additional I/O operations required for query optimization. Also, the buffer flush neighbors option, which flushes any neighbor pages together for a dirty victim page, was turned off to reduce unnecessary write overhead.

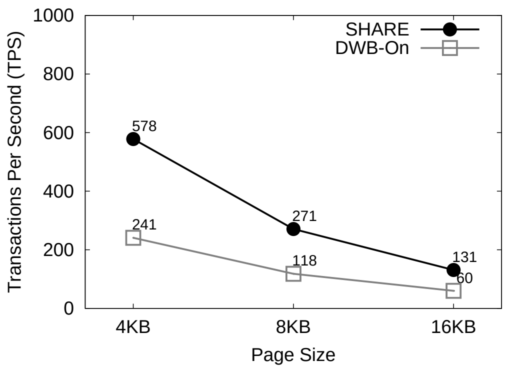
YCSB Yahoo! Cloud Serving Benchmark (YCSB) is a benchmark framework created for evaluating cloud systems [14]. YCSB consists of six workload types and mimics web applications running a huge number of simple queries, each of which touches a single record. Since all the workloads except for workload-A and workload-F are read-intensive, workload-A and workload-F were used to evaluate the performance impact of *SHARE*.

5.3 Run-Time Performance

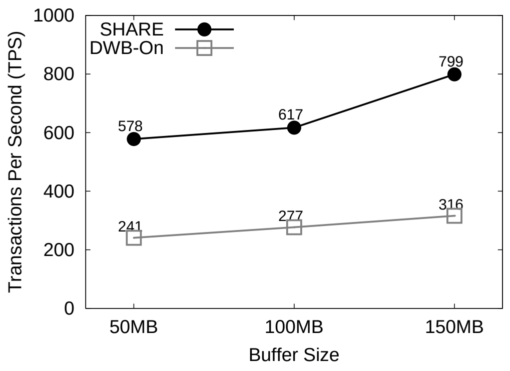
This section demonstrates the effectiveness of *SHARE* by comparing the performance of MySQL/InnoDB using LinkBench with and without *SHARE*. Similarly, the performance of Couchbase NoSQL system was tested using two YCSB workloads with and without *SHARE*. Each performance measurement in this section was an average of three runs.

5.3.1 MySQL/InnoDB for LinkBench

The version of MySQL/InnoDB used in the experiments was the most recent 5.7.5-m15 development release. We created three LinkBench databases of 1.5GB with page sizes of 4KB, 8KB, and 16KB to compare the effect of different page sizes. Throughout all the LinkBench experiments, 16 client threads were concurrently run. For steady performance measurement, the LinkBench was pre-run for a 300 second warm-up time to fill the InnoDB buffer cache. A total of 160,000 transactions (10,000 per client) were run in

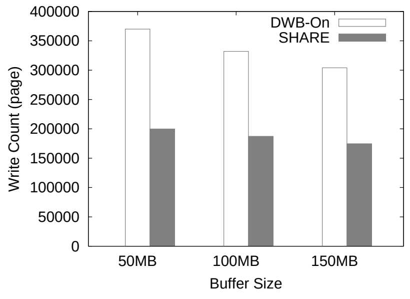


(a) Varying page size

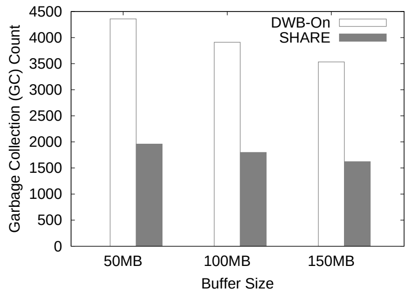


(b) Varying buffer size

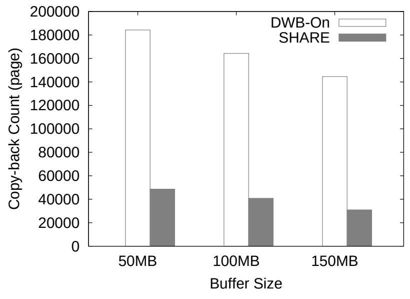
Figure 5: LinkBench throughput on MySQL/InnoDB



(a) OS page write count



(b) Garbage collection count



(c) Copyback page count

Figure 6: IO activities inside OpenSSD (50MB buffer cache, 4KB page)

each experiment. We use the `DWB-On` and `SHARE` symbol to denote MySQL/InnoDB execution in its default mode with the double-write-buffer on and in the *SHARE* mode with the double write buffer off, respectively.

The Effect of SHARE *on Throughput*

Two sets of experiments were conducted to evaluate the effect of *SHARE* on database throughput. In the first set, we ran the MySQL server with three LinkBench databases created with different page sizes when the buffer pool size was fixed to 50MB. For the second set, we ran the MySQL server by varying the buffer pool size from 50MB to 150MB when the LinkBench database page size was fixed to 4KB.

Figure 5 presents the results of measured transaction throughput. As clearly shown in Figure 5(a) and Figure 5(b), the *SHARE* consistently helped MySQL process LinkBench transactions much faster than the `DWB-On` mode by more than two times over all tested configurations regardless of buffer size and page size.

For all the experiments, we also measured the transaction throughput while running the MySQL server in the `DWB-Off` mode to compare the performance differences between three different modes. Interestingly, for all six configurations, although omitted from Figure 5 for clarity, the performance gaps between `DWB-Off` and `SHARE` modes were less than one percent. Recalling that the `DWB-Off` mode also avoids redundant writes, it is obvious that two modes generate the same IO traffic except the `share` commands issued in `SHARE` mode. This comparison derives two interesting points. One is that *SHARE* can achieve the write atomicity at almost no performance cost. The other is that the run-time overhead of our `share` implementation is very marginal.

The Effect of SHARE *on IO Activities*

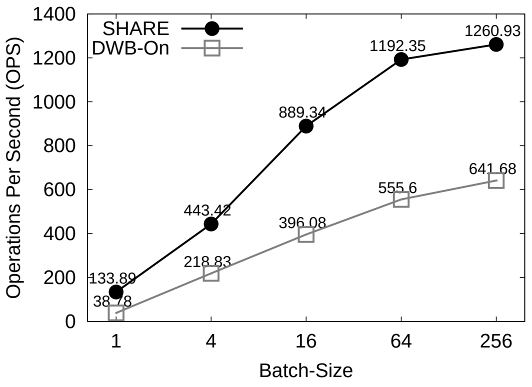
This considerable performance gain of *SHARE*, which is shown in Figure 5, was a direct reflection of reducing the number of written pages. This is confirmed by the number of page writes requested by the host when running the same set of experiments for Figure 5(b). As is shown in Figure 6(a), the *SHARE*-based implementation reduced the number of requested page writes to the SSD by 45% compared to the original MySQL/InnoDB. The reduction ratio is less than 50% because the traffic includes file system metadata page write requests as well as MySQL data page write requests.

To understand how the input traffic reduction influences internal SSD behavior, we measured the number of garbage collection events and the number of valid pages copied-back due to garbage collection. As shown in Figure 6(b) and Figure 6(c), the `SHARE`-based implementation exacts much less SSD burden than the original MySQL/InnoDB. Regardless of buffer size, the garbage collection operations decreased by 55% while the number of copyback pages decreased by 75%.

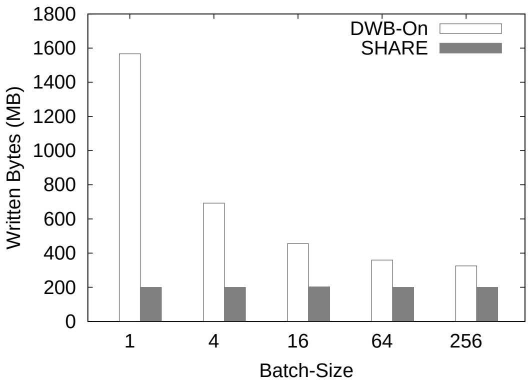
It is interesting to observe the trend between the number of OS page writes, the number of SSD garbage collections, and the number of SSD copyback pages. *SHARE* contributed to 45% reduction of OS page writes, which led to 55% reduction of garbage collection events, and then ended up with 75% reduction of copybacks. By avoiding the redundant writes of pages, *SHARE* halves the number of page

Transactions		DWB-On						SHARE					
I/O Type	Name	Mean	P25	P50	P75	P99	Max	Mean	P25	P50	P75	P99	Max
Read	Get_Node	51.4	4	12	28	600	1363.3	23.9	5	10	23	300	901.1
	Count_Link	32.8	2	5	15	600	1244.4	14.4	2	5	17	200	747.4
	Multiget_Link	40.7	0.3	5	19	500	1573.5	15.2	0.8	6	17	200	313.3
	Get_Link_List	39.4	0.3	5	18	500	17467.4	17.2	0.3	5	17	200	6140.7
Write	ADD_Node	6.3	0.3	0.4	0.4	200	1521.0	1.5	0.3	0.3	0.4	24	554.6
	Update_Node	64.0	5	15	42	700	2071.4	28.3	5	14	27	300	823.8
	Delete_Node	62.6	5	13	40	600	1104.6	26.3	5	12	25	300	596.7
	Add_Link	119.3	17	40	200	1000	2248.2	49.7	14	27	57	400	730.4
	Delete_Link	70.5	4	16	55	800	1417.6	30.3	4	14	30	300	1132.8
	Update_Link	114.9	16	38	200	1000	2270.5	49.4	13	26	54	400	1102.5

Table 1: Distribution of LinkBench transaction latency (in millisec)



(a) Throughput



(b) Total amount of written data

Figure 7: YCSB throughput on Couchbase: Workload-F

writes to the storage and thus triggers less garbage collections (*i.e.*, 10% less GCs). Further, this let a single block survive probabilistically 2x longer before it becomes a GC victim. As a result, data pages in each data block will have more chances to be invalidated before the data blocks are reclaimed for garbage collection. Therefore, in *SHARE*, a victim block likely has less valid data pages at GC time (*i.e.*, roughly a half copybacks per each victim block and quater copybacks in total). This implies that the *SHARE* interface can provide longer device lifespan while less garbage collection events provide more consistent IO performance with less performance jitter.

As is mentioned in Section 2, to guarantee write atomicity, PostgreSQL takes a similar approach to MySQL. In order to evaluate the performance impact of the `full_page_write` feature in the PostgreSQL server, we carried out a separate experiment using the `pgbench` benchmark [5]. In the experiment, when the `full_page_write` option was turned off, the transaction throughput approximately doubled and the amount of WAL log data reduction was roughly same as the total amount of data pages written to the database tables. This indicates that a PostgreSQL server also can expect significant performance gains by leveraging the *SHARE* interface as a mechanism to guarantee the atomicity of page writes.

The Effect of SHARE *on Tail Tolerance*

High tail latency poses serious challenges for online service providers since even small increase may result in reduced traffic and revenue [15]. In order to evaluate the impact of *SHARE* on read and write transactions latencies, we also measured the latencies while running LinkBench with a 50MB buffer cache and 4KB page size. The latency statistics were reported by the LinkBench script at the end of each benchmark run, and are summarized in Table 1.

Table 1 compares two modes of MySQL/InnoDB, `DWB ON` and `SHARE`, in terms of latencies at 25, 50, 75 and 99 percentiles as well as the average and maximum latencies for ten different types of read and write transactions. It is clear from this table that the *SHARE* interface significantly reduced the latency in all measurements. Specifically, the average latency was reduced by a factor of 2.1 to 4.2, while the maximum latency was reduced by a factor of 1.2 to 3.4. More importantly, the 99 percentile latency was also reduced by a factor of 2.0 to 8.3. Once again, this is clear evidence that *SHARE* reduces the tail latencies, contributing greatly to tail tolerance.

Another observation made from Table 1 is that *SHARE* could considerably lower the latency of read transactions as well as that of write transactions. One main reason is that faster write request completion by avoiding redundant writes can shorten the response time of read requests blocked by preceding writes on a page miss [16].

5.3.2 Couchbase for YCSB

To analyze the effect of *SHARE* on NoSQL workloads, we ran a Couchbase system against `workload-F` and `workload-A` of YCSB benchmark with a 1GB database consisting of 250,000 key-value records. The average size of key-value records in YCSB was 4KB. The workload-A consists of 50% read and 50% update operations by default. The workload-F consists of 100% read-modify-write operations by default.

The Couchbase throughput was measured in operations per second (OPS), where an operation is like a RDBMS system transaction. All the experiments were done in a single thread mode. The average index tree depth was three, and the size of each tree node was 4KB. Thus, the average amount of each update operation was about 16KB including the tree nodes and a document update. Couchbase can adjust the `fsync` frequency in order to trade durability for performance by changing the value of a `batch-size` parameter. If the `batch-size` is set to a positive integer, say k, a `fsync` call is executed every k updates.

Effect of SHARE *on Workload-F*

Figure 7 shows the experiment results in workload-F measured under different configurations created by changing the batch size from one to 256 and using *SHARE*. As Figure 7(a) clearly shows, the *SHARE* interface helped Couchbase process operations much faster than the original Couchbase by a factor of 1.96 (batch size 256) to 3.45 (batch size 1).

And Figure 7(b) demonstrates that the considerable performance gain reflected the reduction of the amount of the written pages. Recall that with *SHARE* interface, Couchbase does not update all tree nodes on a path from the root to a leaf pointing to an updated document. For this reason, the amount of written data with the *SHARE* interface is almost constant regardless of batch-size. On the other hand, the original Couchbase experiences less write amplification due to tree-wandering as the batch-size increases. Thus, the gap in the written data amount between the original and the *SHARE*-based Couchbase became much narrower from a factor of 7.86 to 1.64 by changing the batch-size from 1 to 256.

Reducing Couchbase's write amplification is crucial because NoSQL systems usually manage huge data volumes of mobile and web applications. In addition, less write amplification gives Couchbase various performance benefits, such as reduced compaction frequency, providing a consistent performance for normal operations, and prolonging the SSD lifespan.

Effect of SHARE *on Workload-A*

Similarly, in order to also evaluate the effect of *SHARE* on a mixed workload, we ran YCSB's workload-A. The experiments were conducted with the same configurations for the workload-F. Figure 8 shows that *SHARE*-based Couchbase outperformed its original version by a factor of 1.61 (batch-size 256) to 2.23 (batch-size 1).

Effect of SHARE *on Compaction*

In order to evaluate the effect of *SHARE* on compactions, we measured the elapsed times of compaction with the original and the *SHARE*-based Couchbase. As shown in Table 2, compared to the original Couchbase, the *SHARE*-based compaction completed faster by a factor of 3.1 and also reduced the amount of written data by a factor of 7.5. Though quite impressive, the improvement ratio in total elapsed time is less than the reduction ratio in total amount of written data. And the reason is that even with the *SHARE*-based compaction, the header page of each valid document in the old file should be read from the old database file so as to check the length of the document, which is required in the `share` command.

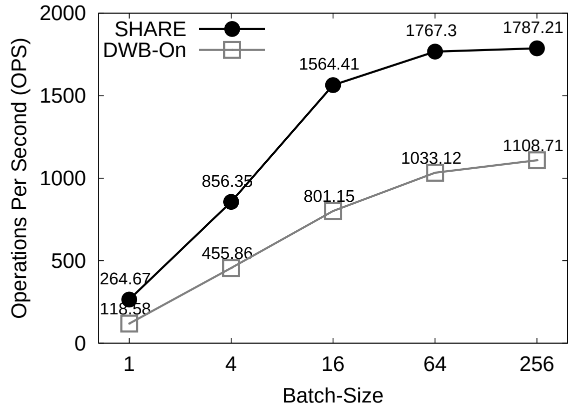


Figure 8: YCSB throughput on Couchbase: Workload-A

	Elapsed Time (sec)	Written Bytes (MB)
Original	277.52	1126.4
SHARE	88.38	150.6

Table 2: Effect of *SHARE* on compaction

6. RELATED WORK

Three types of existing work are closely related to *SHARE*: atomic write in flash storage [26, 27, 29], transactional FTL [17], and address remapping technique [12]. All these techniques exploit the copy-on-write approach used in FTL in order to provide host-side applications in place, page write atomicity. In contrast, with *SHARE*, host-side applications employ an out-of-place update approach and call the *SHARE* command to avoid the redundant write for the atomic writes. Now, let us briefly explain each work and compare it with *SHARE*.

6.1 Atomic Write in Flash Storage

Recently, several interesting suggestions have been made to efficiently support atomic updates on flash-based storage devices, mainly from the storage and file system communities [26, 27, 29]. These FTL approaches have focused more on ensuring the atomicity of a fixed set of pages, predetermined at write time.

To the best of our knowledge, the atomic write FTL by Park *et al.* [27] is the first study for supporting multiple flash storage page atomic writes by exploiting flash storage out-of-place update characteristics. This work only focuses on supporting multiple page atomic writes to be written in a single write request like `write(p1,..,pn)`. Upon receiving the write command, the FTL first writes all the pages into the flash chip media in a synchronous way, then leaves the commit record. If a crash occurs during the data page write or before writing the commit record, the FTL, upon rebooting, undoes all the page writes when the commit record is not found for the transaction. Recently, this approach was applied to FusionIO Atomic Write Extension, (the only device available in the market that supports page write atomicity at the storage level) [26]. And, Ouyan *et al.* [26] showed that it can be used to replace the double buffer area in MySQL/InnoDB, whose goal is to support the atomic data page writes. Similarly, Prabhakaran *et al.* [29] proposed an FTL called *txFlash*, which supports atomic writes semantics tailored for file system journaling.

While these existing approaches focus on supporting the atomic propagation of one or more pages written in an update-in-place manner inside flash storage, the *SHARE* scheme can efficiently support the atomic write of data pages in host applications without write amplification using the out-of-place update approach. And, unlike the existing approaches, our *SHARE* scheme does not require flushing the whole page set at once. Instead, with *SHARE*, applications can write any data page at any time and call the *SHARE* command against a set of pages at an appropriate time.

In this respect, *SHARE* more flexibly supports atomicity semantics. Moreover, *SHARE* can support the Couchbase compaction process, which is not possible with existing atomic write approaches.

6.2 Transactional FTLs

To support the transactional atomicity semantics database applications require, a transactional FTL called *X-FTL* [17] recently was proposed. It is more flexible than the FTLs described above in that *X-FTL* allows writing any data page at any time while still supporting atomicity of any group of pages belonging to a transaction. By also exploiting the flash storage *copy-on-write* mechanism, *X-FTL* can efficiently guarantee that, without resorting to amplified writes, all pages any transaction updates are successfully propagated or no pages are written upon a failure. In other words, it implements flash storage transactional atomicity, which is similar to the shadow paging technique [21]. With *X-FTL*, database applications such as SQLite are relieved from the burden of implementing their own proprietary transactional semantics and also benefit from better performance [17].

A common *X-FTL* and *SHARE* benefit is that they can guarantee transactional atomicity. However, *SHARE* is developed to support applications taking the out-of-place update approach for atomic write while *X-FTL* supports applications using update-in-place. Of course, *X-FTL*, like other related works, cannot be used to support the Couchbase compaction process either.

6.3 Address Remapping in JFTL

Regarding *SHARE*, other interesting flash storage work on and FTL community is JFTL [12]. Under the full journaling mode of a journaling file system [28], for higher file system consistency, the same data page in a file is redundantly written to both the journal area and the original file. To avoid redundant writes, JFTL (*i.e.*, FTL for journal mode) updates the mapping table instead of copying pages from the journal area to the original file. After writing a set of pages such as data pages and the file system's metadata (*e.g.*, inodes and superblocks) in the journal area, the journaling file system informs the flash storage of the list of the journaled data pages using a new interface. Upon receiving the list, the JFTL simply remaps the logical-to-physical mapping information of each data page in the list to the corresponding pre-written page in the journal area, thus avoiding writing the data pages redundantly. In this respect, JFTL is the closest approach to *SHARE*. However, unlike JFTL which is tailored to journaling file system and thus is based on a proprietary interface between OS kernel thread and FTL, *SHARE* is an explicit interface which any application can use. Also, *SHARE* is more flexible in that it supports the Couchbase compaction process.

7. CONCLUSION

Many database storage engines, as well as file systems, utilize a redundant write or copy-on-write approach for atomic writes. Consequent write amplification at the software level adversely impacts both performance and flash storage device lifespans. This paper proposes a novel interface for flash storage called *SHARE*, which allows the applications to explicitly change the FTL indirection address mapping, thus relieving costly write amplification from those applications without compromising atomic write properties.

We have implemented *SHARE* on an SSD development platform called OpenSSD by enhancing its FTL code with *SHARE* features. We have modified MySQL/InnoDB and Couchbase storage engine to exploit *SHARE* with only minimal code changes. Using a set of popular workloads for relational and NoSQL databases, we have demonstrated that *SHARE* achieves a significant performance improvement.

There are numerous applications that can benefit from the *SHARE* interface, including database systems such as PostgreSQL and SQLite and file systems such as Ext4 and BTRFS. As a future work,we will modify those systems to exploit the *SHARE* interface and evaluate its impact using various synthetic and real workloads.

Acknowledgments

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP). (R0126-15-1108, Flash-SQL: NVRam Based High Performance Open Source DBMS development and 1041244, SmartTV 2.0 Software Platform) And, the authors would like to express our sincere appreciation to Mark Callaghan of Facebook for his feedback on early draft of this paper.

8. REFERENCES

[1] mongoDB. https://www.mongodb.org/, 2007.
[2] OpenSSD Project. http://goo.gl/J0Ts5, 2011.
[3] Atomic Commit In SQLite. http://www.sqlite.org/atomiccommit.html, 2014.
[4] MySQL 5.7 Reference Manual. http://dev.mysql.com/doc/refman/5.7/en/, 2015.
[5] pgbench. http://www.postgresql.org/docs/devel/static/pgbench.html, 2015.
[6] PostgreSQL 9.4.4 Documentation. http://www.postgresql.org/docs/9.4/, 2015.
[7] Ahn, Jung-Sang and Seo, C. and Mayuram, R. and Yaseen, R. and Kim, Jin-Soo and Maeng, S. Forestdb: A fast key-value storage system for variable-length

string keys. *IEEE Transactions on Computers*, PP(99):1–14, May 2015.

[8] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: a Database Benchmark based on the Facebook Social Graph. In *Proceedings of the 39th SIGMOD International Conference on Management of Data (SIGMOD '13)*, pages 1185–1196, 2013.

[9] A. Ban. Flash file system, Apr 1995. US Patent 5,404,485.

[10] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*, pages 385–395, 2010.

[11] 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. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[12] H. J. Choi, S.-H. Lim, and K. H. Park. JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory. *ACM Transactions on Storage*, 4(4):14:1–14:22, Feb 2009.

[13] 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, 2009.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, pages 143–154, 2010.

[15] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, Feb. 2013.

[16] W.-H. Kang, S.-W. Lee, B. Moon, Y.-S. Kee, and M. Oh. Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases. In *Proceedings of the 40th SIGMOD International Conference on Management of Data (SIGMOD '14)*, pages 529–540, 2014.

[17] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C.-W. Min. X-FTL: Transactional FTL for SQLite Databases. In *Proceedings of the 39th SIGMOD international conference on Management of data (SIGMOD '13)*, pages 97–108, 2013.

[18] S.-Y. Kim and S.-I. Jung. A log-based flash translation layer for large nand flash memory. In *Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference*, volume 3, pages 1641–1644, Feb 2006.

[19] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[20] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A Log Buffer-based Flash Translation Layer using Fully-associative Sector Translation. *ACM Transactions on Embedded Computing Systems*, 6(3):18, 2007.

[21] R. A. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Database Systems*, 2(1):91–104, Mar 1977.

[22] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013*, pages 31–44, 2013.

[23] D. Ma, J. Feng, and G. Li. A Survey of Address Translation Technologies for Flash Memories. *ACM Computing Survey*, 46(3):36:1–36:39, Jan. 2014.

[24] C. Mohan. Disk Read-Write Optimizations and Data Integrity in Transaction Systems Using Write-Ahead Logging. In *Proceedings of ICDE*, pages 324–331, 1995.

[25] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[26] X. Ouyang, D. W. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. Beyond Block I/O: Rethinking Traditional Storage Primitives. In *Proceedings of International Conference on High-Performance Computer Architecture (HPCA '11)*, pages 301–311, 2011.

[27] S. Park, J. H. Yu, and S. Y. Ohm. Atomic Write FTL for Robust Flash File System. In *Proceedings of the Ninth International Symposium on Consumer Electronics (ISCE 2005)*, pages 155 – 160, Jun 2005.

[28] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of USENIX Annual Technical Conference*, 2005.

[29] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional Flash. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[30] F. Shu. Notification of Deleted Data Proposal for ATA-ACS2. http://t13.org, April 2007.

[31] Sybase. SQL Anywhere I/O Requirements for Windows and Linux. A Whitepaper from Sybase, an SAP Company, March 2011.

[32] D. Woodhouse. JFFS: The Journaling Flash File System. In *Proceedings of the Ottawa Linux Symposium*, 2001.