Building Workload-Independent Storage with VT-Trees

Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok
Stony Brook University

Abstract

As the Internet and the amount of data grows, the variability of data sizes grows too—from small MP3 tags to large VM images. With applications using increasingly more complex queries and larger data-sets, data access patterns have become more complex and randomized. Current storage systems focus on optimizing for one band of workloads at the expense of other workloads due to limitations in existing storage system data structures. We designed a novel workload-independent data structure called the *VT-tree* which extends the LSM-tree to efficiently handle sequential and file-system workloads. We designed a system based solely on VT-trees which offers concurrent access to data via file system and database APIs, transactional guarantees, and consequently provides efficient and scalable access to both large and small data items regardless of the access pattern. Our evaluation shows that our user-level system has 2–6.6× better performance for random-write workloads and only a small average overhead for other workloads.

1 Introduction

Users with ever-increasing data storage requirements use two solutions: file systems and databases. File systems are designed to store large files such as movies and VM images. Users expect to access files sequentially at close to disk throughput, requiring file data to be stored on disk with minimal fragmentation. Databases, however, store large numbers of comparatively small tuples, all indexed so that lookups and scans are efficient. Additionally, transactional semantics are a must for applications to maintain consistent relationships between the large numbers of tuples that databases support.

Today, users must choose between file systems and databases. If program data does not precisely fit one of these two options—or requires *both*—then developers have to carefully consider the workload to decide which trade-off will hurt performance the least. Applications that manage both kinds of data [3, 38] can do no better than to manage data in two separate stores. For example, media players store an index of the title and artist information alongside a directory of media files and Web browsers index browsing history along with a directory of cached Web pages. If such applications also require transactional semantics, then they must rely on complicated procedures to treat separate file system and database transactions as a single atomic commit [19, 37].

Supporting both file system and database workloads is difficult because there is no data structure that can efficiently handle both. File systems typically use an extent-based data structure that is optimized for accessing large tuples (i.e., files) but is not equipped to store or search large numbers of small tuples [10, 29]. For these small-tuple workloads, database engines rely on either of two other data structures, B+ trees or Log-Structured Merge (LSM) trees [21], to efficiently search and read massive indexes [33]. These three structures suffer under workloads that do not match their strengths. Previous attempts split data between (1) a file system store and (2) a B+ tree index [12, 19, 20, 39] but did not solve the problem completely. The resulting hybrid is complex [18] and there is no general solution yet to the question of how to determine which data should use which structure. Worse, neither of these two structures is as efficient as an LSM-tree. A data structure which is preferred for random updates, inserts, and deletes [4, 33].

We present the *VT-tree*, our novel extension to the LSM-tree, designed to combine the performance strengths of file systems, B+ trees, and LSM-trees. The VT-tree's novel *stitching* optimization improves on the LSM-tree's performance by avoiding unnecessary data copies for sequential data. The VT-tree keeps data sorted enough so that it does not sacrifice query performance, but it does not get bogged down in sorting sequential data that is already as sorted as it needs to be. As a result, our VT-tree is positioned to perform well for datasets of any tuple size, whether accessed sequentially or randomly.

On top of the VT-tree, we implemented a prototype key-value storage system called *KVDB* that is versatile and serves as a general-purpose database storage engine. We used KVDB to develop *KVFS*, our FUSE-based [35] POSIX-compliant file system. We conducted extensive evaluations. For most workloads, our KVDB is anywhere from 23% faster to 6.6× faster than LevelDB [16]. Compared to the in-kernel Ext4, KVFS is no slower than 9% and as much as 2× faster.

2 Background

LSM-trees offer 1–2 orders of magnitude faster insertions in exchange for 1–10× slower point queries and scans [4]. They are useful when large data sets need to be indexed for querying, and these queries can be parallelized (e.g., Web searches in Bigtable [7]). LSM-trees support inserts, updates, deletes, point queries, and scans. Figure 1 illustrates the operation of a basic LSM-tree. An in-RAM buffer called a *memtable* (e.g., a red-black tree or skip list) holds recently inserted items.

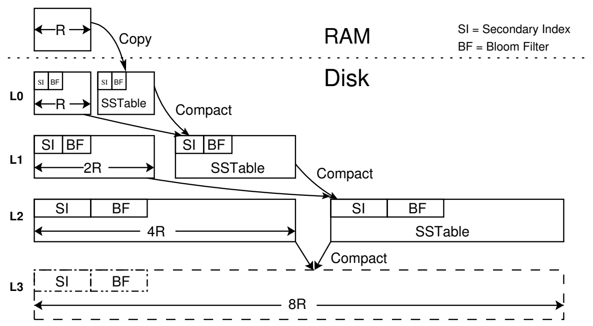


Figure 1: An LSM-Tree performing a compaction

When this buffer is sufficiently full, it is flushed to disk as a single list of sorted tuples. A secondary index and a Bloom filter are generated and stored alongside the sorted list to accelerate future scans and point queries. The list plus its secondary index and Bloom filter is called an *SSTable* [7], as shown in Figure 1.

Point queries search for the value belonging to a key in all SSTables starting with the most recently created ones. Searches in SSTables not containing the key can usually be avoided by using Bloom filters. All remaining searches typically require only one I/O by using the secondary index to avoid a full binary search on disk. *Scan* operations in an LSM-tree are used to find successors, predecessors, or to view all tuples within a range. A scan operation first searches each SSTable to place a cursor; it then increments and merges the cursors so as to visit keys across the multiple SSTables in sorted order. Deletions are as fast as inserts. Deletes are performed by inserting a *deletion marker*. If a search through an SSTable encounters a deletion marker matching the sought-after key, the system returns a “does not exist” indicator.

As more tuples are inserted, the memtable is repeatedly filled and serialized into an SSTable on disk. As the number of tuples inserted increases, the number of on-disk SSTables increases too. Since queries must typically search through most or all of the SSTables, queries slow down as more tuples are inserted. Therefore we limit the number of SSTables to a function of the number of tuples inserted to bound lookup and scan latency.

Figure 1 shows a simplified variant of the algorithm [33] used by our implementation. The number of SSTables in Figure 1 is bounded to $2 \times \lceil log_2 N \rceil$. We will use this bound as a running example, but LSM-trees can be configured with different bounds to achieve different read-write trade-offs. The number of SSTables in Figure 1 is bounded because each SSTable must reside in a level, and each level can only hold 2 SSTables and each SSTable can only hold $R * 2^i$ tuples where i is the index of the level of the SSTable. In Figure 1 we show three levels, $i = 0$ through $i = 2$, with the two SSTables in each level holding R, $2R$, and $4R$ tuples, respectively. After the memtable is full, we must compact to make room in the first level for a new SSTable. We compact the LSM-tree by merging together the two SSTables at level $i = 0$ into a new SSTable of size $2R$ and placing it in level $i = 1$. In Figure 1 there are already two SSTables in level $i = 1$ so we reapply our rule to level $i = 1$ and for the same reason to level $i = 2$. After compaction, there will be an SSTable of size $8R$ in level $i = 3$, and SSTables of sizes $2R$ and $4R$ in levels $i = 1$ and $i = 2$, respectively. Level $i = 0$ will be empty, allowing us to serialize the memtable in RAM into level $i = 0$ as a new SSTable of size R. As each tuple visits each level once and is always sequentially copied when moving between levels, a single random insertion costs $log_2 N$ sequential tuple copies. Therefore, if we can sequentially copy B tuples in the time it takes to perform one random I/O to the device, then we can perform $\frac{B}{log_2 N}$ random insertions in the same time it takes a hashtable or B-tree to perform one random insertion. This is why LSM-trees provide such speedy random updates compared to traditional in-place data structures.

Superfluous Sorting. The LSM-tree’s primary weakness is that every tuple written is eventually copied $log_2 N$ times. These copies are by far the fastest way to organize incoming random data for efficient lookups, as evidenced by the LSM-tree’s excellent performance for random insert workloads; but when inputs are already sorted, they serve no purpose. As an example, to store sorted data $16\times$ the size of the memtable, LSM-trees write each byte up to eight times. A traditional file system writes a file only once and leaves it in place until the next defragmentation. LSM-trees could be configured to break SSTables into 2MB chunks as LevelDB [16] does, but this is ineffective for smaller sequential runs; moreover, configuring LevelDB to use smaller chunks incurs fragmentation that punishes scan performance. Alternatively, we could skip compacting SSTables with sufficient sequentiality; this determination, however, is even more coarse then LevelDB’s 2MB chunk approach and quickly leaks space, leading to a costly compaction of all SSTables. Our VT-tree closes the performance gap for sequential workloads while preserving the LSM-tree’s superior random insert performance.

3 Design and Implementation

For KVFS, our design goal is to provide a single storage implementation that handles mixed workloads and supports full system transactions [32] with minimal overhead. First, we present a storage system capable of supporting both file system and database workloads efficiently using a single, versatile data structure: the *VT-tree*. Second, we present an efficient transactional architecture that avoids most double writes, is highly parallelizable for partitionable workloads, and supports larger-than-RAM transactions. Applications

VT-tree	Format
nmap	$\langle\{$*parent-inode#*, *path-component*$\}$, *inode#*$\rangle$
imap	$\langle$*inode#*, *inode*$\rangle$
dmap	$\langle\{$*inode#*, *offset*$\}$, *data-block*$\rangle$

Table 1: Three dictionary formats used within fs-schema

using KVFS can group together a sequence of POSIX and key-value storage operations into a single atomic transaction. Next we introduce KVFS's architecture. We then discuss VT-trees and our sequential optimization in Section 3.1. We detail our transactional architecture and extensions of FUSE in Section 3.2.

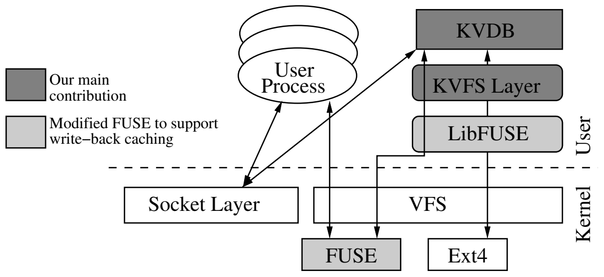


Figure 2: KVFS Architecture

KVFS Architectural Overview. Figure 2 shows KVFS's architecture. KVFS uses FUSE [35] to provide a POSIX-compliant file system API for traditional applications. A user read or write request is passed on to FUSE by the VFS. The request goes to the KVFS layer running as a user-space daemon. KVFS translates the request into one or more key-value operations sent to KVDB, which implements a transactional database storage engine based on the VT-tree. KVDB performs all necessary I/Os using a series of `mmaps` of a disk file stored on a back-end Ext4 file system. The response follows the same request path but in reverse. We modified FUSE to cache dirty writes in the kernel to avoid and defer expensive upcalls.

When initializing KVDB, we create a schema with one or more dictionaries, each backed by a VT-tree. KVFS defines three VT-trees in a single KVDB schema, called *fs-schema*, which supports file system operations. The three dictionary formats, also shown in Table 1, are: (1) *nmap* for namespace entries, similar to `dentries`; here, the *path-component* of a file and its parent directory's inode number form the key, and the value is the file's inode number. (2) *imap* for storing inode attributes; and (3) *dmap* for the files' data blocks.

The two-table design separates meta-data (pathnames) from the data (objects). Our current implementation can be adapted to other indexing forms (photo cache, music cache, mail tags, etc.) [31, 39]. KVFS can store any external meta-data associated with a file by creating a new dictionary format within the same KVDB schema. KVFS also provides a key-value interface directly to KVDB using sockets; this is useful for databases and applications that process random-access workloads. KVFS supports access to both the POSIX and key-value interface in the *same* system transaction. For example, an application like iTunes [2] could use a transaction to atomically save an MP3 file, add its title and artist information to a music library, and index it to be searchable by a Spotlight-like indexing system [3].

3.1 VT-Tree Sequential Optimizations

An LSM-tree with our extensions for efficient sequential insertions and large tuples is called a *Variable-Transmission tree* (*VT-tree*). A VT-tree can be tuned to behave like a log-structured file system (LFS) and avoid writing more than once—or to behave like a LSM tree, and guarantee bounds on the number of seeks required to perform a scan. Intermediate configurations are possible, to trade off seek and scan performance for repeated write performance. Thus, a VT-tree can span the entire continuum from an LSM-tree to an LFS.

Stitching: pay-as-you-go merging for LSM-trees. Section 2 explained how LSM-trees copy already sorted data multiple times to sort it anyway. Figure 3 shows how VT-trees use *stitching* to avoid these superfluous copies when performing a merge. VT-trees are stored on a log-structured block device ①. We merge SSTables ② and ③ into ④. For each SSTable, Figure 3 shows the secondary indexes in RAM as well as the list of tuples on disk divided into blocks, pointed at by the secondary index entries. This example's secondary index contains both the lowest and highest key in each block; our actual implementation stores only the lowest key.

The first step in stitching is to identify blocks in SSTables ② and ③ that do *not* overlap with any other block. The secondary index entries pointing at these blocks are marked with *C* as they are merely (c)opied over (in RAM) to construct the secondary index for ④. Such blocks that are included in the resulting SSTable are called *stitched*. The two blocks that do overlap (dotted line) are read in from disk, merged, and then written out in sorted order to the *fill*. New secondary index entries (marked with *N*) are made to point to the merged blocks in the fill. Such blocks that are included in the resulting SSTable are called *merged*. Thus we finish constructing the secondary index for ④. Figure 3 shows how stitching allows the VT-tree to merge only blocks that overlap and stitch non-overlapping blocks, thus avoiding the superfluous copying induced by traditional LSM-tree merging for already sorted runs of tuples.

The example in Figure 3 stitches all non-overlapping blocks. However, this can induce fragmentation. Fragmentation can waste space and harm scan performance. Fragmentation wastes space because every time we merge an overlapping block, the original blocks in the source list (e.g., the blocks pointed at by $(4, 6)$ and $(5, 7)$

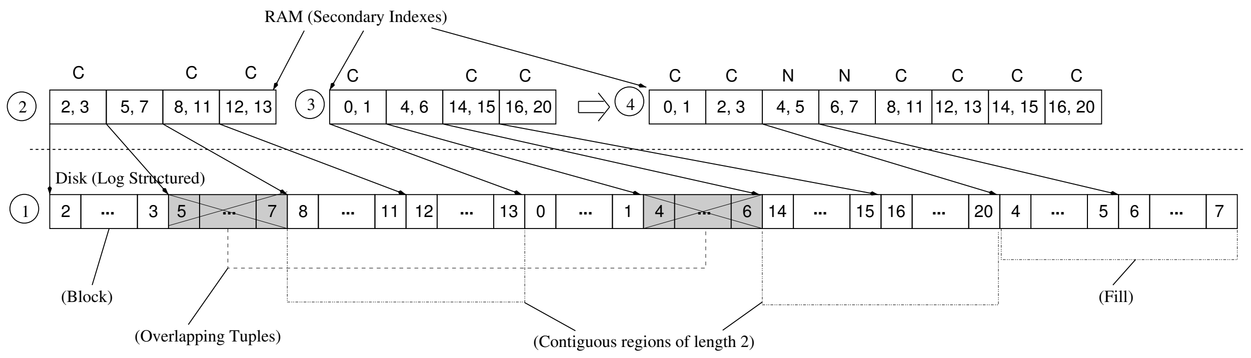


Figure 3: Stitching Example

in ② and ③, respectively) are no longer pointed at by any secondary index after ④ is constructed. As indicated by being *X*-ed out in Figure 3, they become *holes*: lost space. To reclaim this lost space, VT-trees are stored on a log-structured block device so that an LFS-style defragmenter can be run to reclaim lost space. We discuss our defragmentation policy further below.

Fragmentation can also harm scan performance. In Figure 3 every non-overlapping block was stitched. Scanning SSTable ④ would require 5 seeks. In the worst case, a sequential read could require one seek for each block pointed at by the secondary index. We can minimize fragmentation due to stitching by requiring on-disk regions being stitched to be larger than N blocks, be contiguous, and not overlap. We call these regions *contiguous spans* and their minimum length in blocks is the *stitching threshold*. With a stitching threshold of 2, the merge depicted in Figure 3 would have also copied blocks $(2, 3)$ and $(0, 1)$ into the fill—even though they are non-overlapping—because they are not part of a contiguous span of length two or more. This reduces the number of seeks to scan ④ to 4 at the cost of merging two more blocks. The stitching threshold limits stitching to preserve a minimum level of data contiguity. It can be dynamically changed at any time, even while merging, based on the values of tuples being merged.

Stitching threshold. The stitching threshold represents a trade-off between maximizing the sequentiality of tuples on disk and minimizing the number of writes. Smaller thresholds perform fewer copies but at the cost of requiring additional random I/O for scans.

Figure 4 compares four types of trees: read-on-append trees (e.g., log-structured B-trees), LSM-trees, in-place trees (e.g., B-trees), and the VT-tree. Each tree is depicted showing the path a tuple took to its final destination. Horizontal lines represent an opportunity to sequentially copy a tuple into a larger list during compaction or a similar process. Only LSM-trees take every opportunity, copying every tuple $\log_2 N$ times until all tuples are physically contiguous on disk (bottom level). None of the other trees take every opportunity

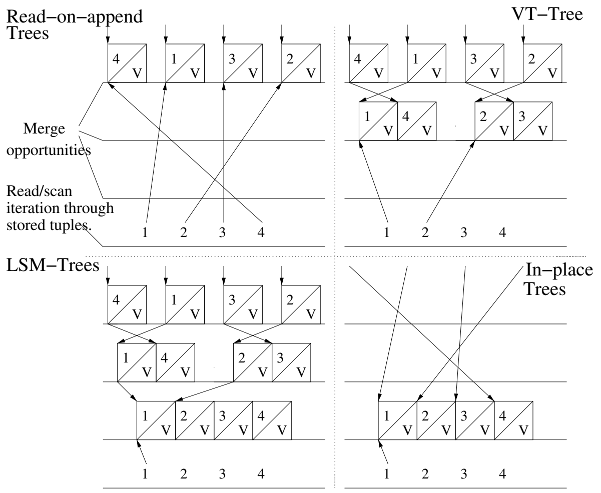


Figure 4: LSM vs. LFS. 'V': storage for tuple value.

to compact tuples; this allows them to potentially write more efficiently. In-place trees directly flush each tuple to its final location, but perform random writes. Read-on-append trees sequentially write every tuple to disk initially, but require random reads to scan in sorted order. LSM-trees sequentially read as efficiently as in-place trees; and, while they perform $\log_2 N$ writes for each tuple, these can still be faster overall than in-place writes that incur disk seeks. The VT-tree is thus a compromise between a straight LFS-based approach and a straight LSM-tree approach: contiguous tuple sequences smaller than the stitching threshold are compacted until they form spans large enough to be left in place.

Quotient filters. Typically, LSM-trees use Bloom filters (BFs) to avoid IO when performing point queries for tuples that do not exist. The VT-tree instead must use Quotient Filters (QFs) [5]. In a typical LSM-tree, when merging two SSTables together, all tuples are typically rehashed into a new Bloom filter. Since VT-trees use stitching to avoid reading many of the tuples, rehashing is not possible without sacrificing performance. VT-trees consequently use QFs. Unlike Bloom filters, the entries in a QF can be rehashed into a larger QF without requiring the original key value. Therefore we can merge the QFs of two merging SSTables entirely in RAM, preserving the performance gains of stitching while also allowing for high-throughput point queries.

Defragmentation. Section 3.1 describes how VT-trees are stored on log-structured storage. When tuples are merged while their neighbors are stitched, the merged tuples will leave holes, and these holes will waste space as shown in Figure 4. However, the minimum unit of allocation and deallocation or segment size in our log-structured store is 8MB, and we refer to these segments as *zones*. Since zones are larger than blocks, we cannot deallocate the block without deallocating its containing zone. Therefore to deallocate a block we must first move other blocks in the same zone elsewhere. The process of moving live or non-hole blocks to other zones so a zone can be deallocated is called *evacuating* the zone. The process of evacuating the under-utilized zones in our log-structured storage to remove holes and create free space is called *defragmentation*.

Evacuation works by moving tuples in an underutilized zone A into other zones and then freeing the now empty zone A. (1) We select an underutilized zone; next (2) we determine the length of the first contiguous span of tuples in this zone; then (3) we use *first-fit* allocation to find a suitable new location for this span in another zone; and finally (4) we move the span to its new location. We repeat these steps until all spans in the underutilized zone are moved and then free the zone. When choosing a zone to evacuate, we prioritize first based on the amount of unused space but also on the number of contiguous spans. It is easier to relocate smaller spans as it is more likely for first-fit to find them suitable new locations. We weigh each zone by computing the ratio of the number of tuples it contains to the number of contiguous spans it contains. We compute these weights in RAM using secondary index.

Our defragmentation method is similar to that used by Rosenblum et. al. [26] and was designed to preserve the on-disk sequentiality of data created by VT-tree compaction and to avoid overwrites which would in turn require undo-logging. Defragmentation only occurs when the device is almost out of space. On the other hand, compaction happens regularly as part of the LSM-tree algorithm inherited by the VT-tree. Therefore, requiring defragmentation only adds one infrequent extra copy in exchange for avoiding $log_2 N$ additional copies for sequential data. Our current implementation operates offline, defragmenting the data store while it is not in use. Potential extensions to our defragmentor and avenues for future work are discussed in Section 6.

3.2 Snapshots and Transactional Support

The VT-tree's log structure allows KVDB to naturally support snapshots [30]. A new KVDB database starts with a single schema, containing a VT-tree for each table. This initial schema serves as the *mainline*, to which all updates are written. To create a snapshot, the mainline schema is marked as read only, and a new schema is created with empty VT-trees. This empty schema becomes the new mainline and has a dependency on the previous mainline. When querying a table (from either the mainline or a snapshot), the search includes SSTables in the queried schema and all dependent schemas, in order to include the table's complete history. Snapshots allow for incremental backups without taking the database offline. Additionally, KVDB uses snapshots to provide support for transactions.

Transaction support in KVDB. KVDB supports ACID transactions with two modes of isolation: (1) snapshot isolation, which guarantees that reads within the transaction are consistent with a snapshot created when the transaction began, and (2) serializability, which provides the highest level of isolation [13].

Snapshot isolation (SI) is weaker than serializability but adds less overhead and affords better concurrency. Users can further boost concurrency by disabling conflict detection, if parallel transactions do not operate on overlapping data. When a process begins a transaction, KVDB creates a new schema for the transaction to write to. In SI mode, KVDB also creates a snapshot. KVDB then marks the new schema dependent on this snapshot, to ensure isolation. Updates happening outside of the transaction go to the new mainline schema; transaction do not see these external updates. In SI mode, conflicts are resolved at commit time. In case of a conflict, KVDB sequentially scans through the conflicting data and calls a schema-specific conflict-resolution routine.

In *serializability mode* (SER), the new schema behaves differently from a snapshot: reads within the transaction see recent updates to the mainline and recent updates from other committed transactions. Isolation is instead provided with locking (described below). Starting a new transaction in this mode does not freeze and replace the current mainline; the new schema is always dependent on the most recent mainline.

Writing a transaction's updates to a separate schema ensures that they are not visible to other readers before the transaction commits. Aborting a transaction involves only freeing its schema, so KVDB does not need to write undo or redo records, even for transactions that are larger than RAM. Additionally, because each transaction writes to private VT-trees, our architecture provides good performance for parallel threads writing to separate transactions, especially in snapshot-isolation mode.

To commit, KVDB merges the transaction schema into the mainline by moving SSTables from the transaction schema's VT-trees to the mainline ones. The transaction's SSTables are placed before the SSTables already in the mainline VT-tree, so updates in the transaction effectively overwrite the original mainline values. Note that VT-trees in the transaction schemas have their own memtables, which are flushed to the on-disk structure before the merge. Writing to a separate VT-tree as part of transaction potentially limits the stitching opportunity. However, a compaction followed by a transaction commit allows the new data from the transaction to be stitched with the data in the mainline schema.

Locking policy. In serializability mode, KVDB uses strong, strict two-phase locking (SS2PL) for isolation. Before accessing any key-value pair, a transaction takes a shared lock for reads or an exclusive lock for writes. The lock is held until the transaction commits or aborts. When there is contention for a lock, the highest priority transaction proceeds first, and other transactions may wait or abort. In case of a deadlock, the lowest priority thread aborts. Transactions can optionally lock a range of keys within a table ahead of time, to avoid the need to lock a large number of keys individually. For example, KVFS transactions acquire a range lock to gain exclusive access to an entire directory. By range-locking directories, KVFS can avoid having multiple transactions simultaneously accessing the same set of files, allowing for better partitioned workloads and improved concurrency. KVDB stores all range-lock reservations in a tree, so it can efficiently check to see whether a transaction holds a reservation for any object in the database.

Transaction support in KVFS. KVFS stores the entire file system representation in a KVDB database. Therefore, KVFS can provide system transactions using either of KVDB's isolation modes. When an application opens a transaction, KVFS exposes a directory link to the root directory that leads to the same file system but that embeds the new transaction ID. All file operations through the link execute in the scope of the transaction. The transaction-begin function in KVFS's user library opens a new transaction and then uses `chroot` to transparently redirect the application's view of the file system to the open transaction's link. Child processes inherit the `chroot` view, so they also participate in the transaction (and share locks held by the parent process). Multi-process transactions make it possible to write shell scripts that leverage file system transactions.

When a file is accessed from two or more transactions, the kernel page cache must not share cached data between transactions, to ensure isolation. In KVFS, transactions see different inode numbers for the same file, ensuring that the kernel page cache maintains separate caches for each transaction. A file accessed through a transaction has an apparent inode number composed of the transaction ID and the actual on-disk inode number. When the file's cache is flushed, though, KVFS uses the on-disk inode number to send the write to KVDB. Separate caches also ensure that a transaction's first access to a file is never served from the kernel's page cache, but must go through KVFS's file access routines, to ensure locking using a reader-writer lock. To end a transaction, the application need only leave the `chroot` and then commit or abort. KVFS's user library transparently accomplishes these steps; KVFS removes the transaction's directory link, invalidates its caches, and forwards the commit or abort command to the underlying database.

KVFS's architecture requires no modifications to the kernel, unlike other system transaction extensions [23, 32]. FUSE [35] provides interfaces to flush and invalidate kernel page caches. Enabling transactions in a user application also requires few application modifications to insert calls to KVFS's begin-, commit-, and abort-transaction functions (supported by our library).

3.3 FUSE Write-Back Cache Support

FUSE [35] helped us implement KVFS quickly without complex kernel coding, but it comes at a performance cost. The kernel VFS makes an upcall to user space to service file operations, adding two extra context switches and as much as $2\times$ overhead [24]. FUSE avoids some of this overhead by servicing file reads directly from the kernel page cache. File writes, however, always go directly to the user-space server—a write-through cache. FUSE's *big-write* optimization sends 128KB worth of data in one upcall, but only when applications write 128KB in a single operation to the same file. We modified FUSE to implement a straightforward write-back cache. The user-space upcall is deferred until the kernel dirty-page writeback thread is called. Our enhancement can aggregate up to 128KB of write requests to *any* number of files, instead of requiring one full upcall for each request. To maintain POSIX compliance, we flush a file's dirty pages upon `close`. As transactions in KVFS have private writable snapshots and use different inode numbers to refer to the same file, the write-back cache does not threaten our isolation guarantees.

4 Evaluation

We mainly focus on evaluating the two design goals mentioned in Section 3. First, using micro-benchmarks we evaluate the effectiveness of stitching, our extension to the LSM, which is our key for achieving a unified storage architecture. We continue to evaluate stitching optimization by comparing KVDB against LevelDB [16], another LSM-based key-value store. Then, in Section 4.2 we compare KVFS's performance against Ext4, concluding that it is practical to use KVFS. Second, with KVDB's simple and efficient transactional architecture, we show in Section 4.3 that transactions in KVFS come with minimal overhead and are highly concurrent. We measure the effectiveness of our defragmentation algorithm, devised for VT-trees in KVFS, in Section 4.4.

Experimental Setup. We conducted experiments on three identically configured machines running Linux Ubuntu 10.04.3 LTS. Each machine includes a single-socket Intel Xeon X5650 (4 cores with one hyperthread on each core, 2.66GHz, 12MB L2 cache). The machines have 64GB of RAM; to test the out-of-RAM performance, we booted them with 4GB each. Each machine has a 146.2GB 15KRPM SAS disk used as the system disk and a 160GB SATA II 2.5in Intel X25-M Solid State Drive (SSD) used to store the out-of-RAM part of the data. We use only a 95GB partition of the SSD to minimize the SSD FTL firmware interference [14]. We measured the 10-thread random-read throughput on the Intel X-25M at 15,000 IOps, a random-write throughput of 3,500 IOps, a sequential read throughput of 245MB/sec, and a sequential write throughput of 107MB/sec. We dropped all the caches before running any benchmark.

For all KVDB and KVFS runs we set the *stitching-threshold* to 64KB. The average file size in a general-purpose file system is around 32–64KB [1]. To avoid lots of copies for average sized files, we set the stitching threshold to 64KB. All benchmarks in Sections 4.1–4.3 ran on KVDB with no defragmentation. The reported results and comparison with other systems are not affected by this. At the end of these benchmarks, around 50% of usable file system space was still available. Systems with online defragmentation rarely run it when there is plenty of space available and less fragmentation on disk.

4.1 Stitching Microbenchmark

To evaluate the performance impact of stitching, we benchmarked the VT-tree with two synthetic workloads: SEQUENTIAL-INSERTION and RANDOM-APPEND. Both workloads insert 10GB of data into an empty VT-tree. After populating a tree, we test read throughput when scanning the inserted tuples sequentially and using random point queries of known keys. The SEQUENTIAL-INSERTION workload simulates an application that writes large files sequentially. We simulate writing a file by inserting 16,384 4KB tuples with sequentially increasing keys, starting from a random key in the range [1, $2^{64}-1$]. The workload repeats the simulated file write until it has inserted 10GB of tuples. The RANDOM-APPEND workload models an application that randomly selects a file and then appends a block to it until a specific amount of data has been written, as in Filebench's `/var/mail` workload [11]. We begin with 2,040 predefined ranges in the key space. We randomly select one of the ranges, insert the first tuple in the range that has not yet been added, and repeat until all ranges have been completely written. Each range gets 1,280

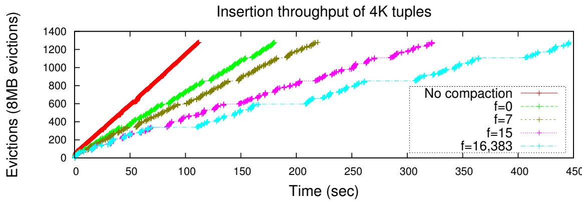


Figure 5: Stitching: Insertion results

4KB tuples, for a total of 10GB of data.

Figure 5 shows insert results for RANDOM-APPEND with four different stitching thresholds. The stitching threshold f is given in units of 4KB (the block size). For $f = 0$, tuples are written once and then stitched in every compaction. With an $f = 16,383$ threshold, tuples are copied from level to level as they are not part of large enough ranges to be stitched during a minor compaction. Thus, tuples are appended to the fill. Eventually, they are part of a large enough range to be stitched and only their secondary index entries are copied in subsequent minor compactions (See Section 3.1). The *no compaction* run shows results for a simple data structure that writes SSTables but never compacts them, resulting in 1,280 SSTables after inserting all the input data. The plot shows on the y-axis how many memtable evictions were completed at each point in time (x-axis), with each eviction flushing the entire 8MB memtable to disk. Periods of time during which no evictions complete indicate that the VT-tree was busy performing a compaction. Lower stitching thresholds spend less time compacting and have significantly faster insertion rates (indicated by a steeper slope in the plot). Although the $f = 0$ configuration never copies data, time spent compacting is visible in the plot. Processing time is required to compute the new secondary index and quotient filter, which adds some overhead over the no compaction profile.

Figure 6 shows the trade-off for insertion performance in the RANDOM-APPEND workload. Though lower values of f have faster insertion throughput, higher values of f result in higher read throughput when scanning the inserted data. The extra copying that occurs when the stitching threshold is high serves to group tuples sequentially, allowing for faster scans. For SEQUENTIAL-INSERTION (not shown), stitching performed well for thresholds less than the size of a fully inserted range (16,384 tuples). For $f = 0$ and $f = 15$, we measured a insertion throughput of 85MB/s and 87MB/s, respectively. We found that these tests were CPU-bound, which explains why they were slightly slower than inserting with no compaction, which had a throughput of 91MB/s (compared to the SSD's maximum serial write throughput of 110MB/s). For $f = 16,383$, the workload was unable to take full advantage of stitching, and throughput was only 28MB/s, less than a third of the performance we obtained with lower stitching thresh-

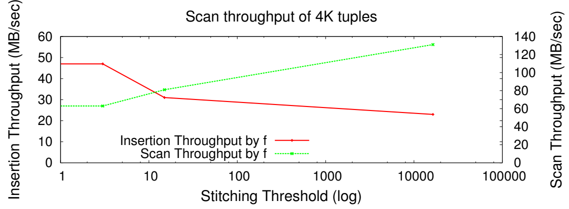


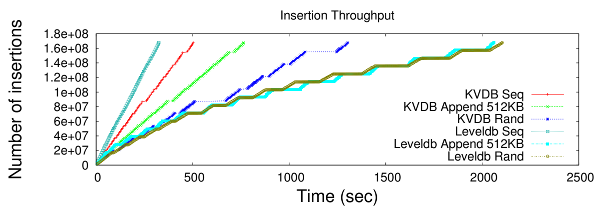
Figure 6: Stitching: Scan and Point query results

olds. Moreover, we found that, for sequentially inserted data, stitching does not affect scan performance. Read throughput was 152MB/s, 158MB/s, and 149MB/s for the $f = 0$, $f = 15$, and $f = 16,383$ profiles, respectively: between 60–64% of the SSD's maximum read throughput. Without compaction, though, read throughput was only 25MB/s. This is because a scan requires placing cursors in 1,280 SSTables and doing random I/O across these SSTables to read tuples in serial order. With compaction there are only 40 SSTables to deal with.

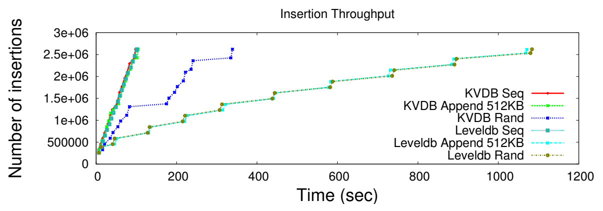
We found that stitching does not affect point query performance. For both workloads, point query performance stayed within a range for all the stitching configurations we tested: 14,600–14,800 IOps. Even with no compaction, throughput was 14,100 IOps. With Quotient Filters (QFs) [5] and secondary indexes, most point queries search only a single region on disk.

We also found that stitching gives the VT-tree significantly higher insertion throughput. For RANDOM-APPEND workloads, the VT-tree performed at least as well as the LSM-tree, and if the user is willing to sacrifice scan performance, up to 2.5× faster for insertions. For SEQUENTIAL-INSERTION workloads, insertions were 3.1× faster with no loss in scan performance.

SATA disk results. We ran the same RANDOM-APPEND benchmark on a magnetic disk to show that VT-tree and its stitching performs equally well on a magnetic disk. We used a 249GB 7.2KRPM SATA disk for this purpose. The results show the same behavior seen for the SSD in Figures 5 and 6. As expected, *no compaction* and runs with a lower stitching threshold had a higher insertion throughput but lower scan throughput when compared to the runs with higher stitching thresholds. Insertion throughput for *no compaction*, $f = 0$, $f = 7$, $f = 15$, and $f = 16383$ were 42MB/s, 34MB/s, 24MB/s, 16MB/s, and 10MB/s, respectively. The sequential write throughput of the disk is only 51MB/s. Scan throughputs are 5MB/s, 5MB/s, 17MB/s, 31 MB/s, and 56MB/s, respectively. The $f = 16383$ run's scan performance (56MB/s) was closer to that of the sequential read throughput of the disk (59MB/s), as expected. Point query performance was the same for the all runs and is close to the random-read throughput of the disk (0.7MB/s). This shows that VT-tree can be used with the same efficiency on both SSDs and magnetic disks.



(a)Inserting 64B key-value results



(b)Inserting 4KB key-value results

Figure 7: KVDB vs. LevelDB results

Database performance. To compare KVDB and LevelDB, we tested both with three workloads: SEQUENTIAL, RANDOM, and RANDOM-APPEND. All three insert 10GB of tuples. The SEQUENTIAL workload inserts the tuples in sorted order; the RANDOM workload inserts keys randomly using a uniform distribution. The RANDOM-APPEND workload randomly chooses from a set of ranges to append to, as in Section 4.1. In this RANDOM-APPEND workload, we selected the range size such that the average amount of data written to each range on evicting the memtable is 512KB. We ran all these workloads with two different tuples sizes: 64B and 4KB. For the most direct comparison of the data structures used in each database, we disabled compression and reliability features (e.g., write logging) in the version LevelDB we used (version 1.5). We used a 256MB memtable for both KVDB and LevelDB; and we configured quotient filters in KVDB and Bloom filters in LevelDB to have the same false positive rate of 0.008.

Figure 7(a) shows that with small tuples, KVDB is faster than LevelDB for all workloads except SEQUENTIAL. LevelDB has optimizations for sequential insertion: during a compaction, it can avoid copying a 2MB zone if there are no keys from other zones that interleave it. However, it is not practical to decrease zone sizes because LevelDB stores a separate Bloom filter for each zone; also, these zones are stored as separate files in the underlying file system. As a result, LevelDB cannot avoid copying the smaller contiguous spans that appear in the RANDOM and RANDOM-APPEND workloads: thus KVDB is 38% faster for RANDOM and 23–53% faster for RANDOM-APPEND. LevelDB sees a big performance drop in the RANDOM-APPEND 512KB workload, because the average contiguous span of tuples in an evicted memtable is less than its 2MB threshold. Random point queries had similar performance in both systems: 13K ops/s in KVDB and 12.7K ops/s in LevelDB.

Figure 7(b) shows that KVDB outperforms LevelDB for larger tuples. Insertion throughput is similar for SEQUENTIAL (KVDB is 4% faster), but KVDB is 3.2× faster for RANDOM and 5.1–6.6× faster for RANDOM-APPEND. With larger tuples, secondary indexes have fewer entries, making stitching operations more efficient, thus amplifying their performance advantage. As with small tuples, we found similar point query speeds: 14.2K ops/s for KVDB and 13.6K ops/s for LevelDB.

We conclude from these results that KVDB compares favorably with LevelDB for database workloads and is better for file system workloads. In our small-tuple tests, designed to show database performance, KVDB was faster for all workloads except pure SEQUENTIAL; for the large-tuple RANDOM-APPEND workloads, testing file system performance, KVDB was up to 6.6× faster.

4.2 File System Performance

KVFS uses FUSE to support POSIX operations. FUSE syscalls require two additional context switches and buffer copies, resulting in about 2× overhead over native file systems [24]. However, serial reads in FUSE are comparable and at times better than native file systems. This is due to caching and read-ahead performed at both the FUSE kernel and the lower native file system [24]. The vanilla FUSE kernel module caches read pages, but writes are immediately sent to the FUSE server running in user space. Our simple write-back caching minimizes these context switches and buffer copies on writes. We compare KVFS's performance with plain Ext4, FUSE-Ext4, and FUSE-Ext4-no-wc. FUSE-Ext4 is a pass-through FUSE mounted on Ext4 that uses our write-back cache. FUSE-Ext4-no-wc does not use our write-back caching. KVFS by default uses the write-back cache. We can measure FUSE overhead by comparing FUSE-Ext4 and FUSE-Ext4-no-wc runs against Ext4. We use Filebench [11] for our evaluation. We compare results for serial-read, serial-write, and random-write micro-benchmarks in Section 4.2.1, and we benchmark with Filebench's real-world workloads in Section 4.2.2.

4.2.1 File System Micro-Benchmarking

KVFS creates an *fs-schema* consisting of three VT-trees in KVDB as described in Section 3: *nmap*, *imap*, and *dmap*. We configured *nmap*, *imap*, and *dmap* to have RAM buffers of sizes 6MB, 12MB, and 256MB, respectively. Larger RAM buffers can improve performance, but we must also leave enough space to accommodate secondary indexes and QFs. We used Filebench's randomread, randomwrite, singlestreamread, and singlestreamwrite micro workloads. All the I/Os are done at 4KB size unless otherwise mentioned. Filebench's randomread workload reads a 30GB single file randomly. The randomwrite workload performs random writes on a pre-allocated 30GB file. The singlestreamread workload serially reads a pre-created 30GB file with I/O size of 16KB. The singlestreamwrite workload sequentially starts writing to a file at offset zero.

	SR	SW	RR	RW
Ext4	26,069	27,293	14,617	4,165
FUSE-Ext4-no-wc	26,256	14,006	13,625	3,889
FUSE-Ext4	25,785	24,739	13,494	4,078
KVFS	25,513	24,191	13,396	8,638

Table 2: Filebench micro-benchmark results (ops/s). SR: Seq. Read; SW: Seq. Write; RR: Rand. Read; RW: Rand. write.

Results. Table 2 shows that all FUSE variants, including KVFS, have similar serial-read performance as Ext4. Since in the serial-read workload, the file is first written sequentially, the VT-tree does not have to sort sort data: stitching preserves the block order, allowing fast subsequent serial reads. The VT-tree's stitching optimization avoids copying during compaction for serial writes, allowing its performance to be similar to FUSE-Ext4 even though Ext4 includes extents and delayed-allocation optimizations. Our write-back cache in FUSE-Ext4 provides a 43% improvement over FUSE-Ext4-no-wc, allowing it to bring down the FUSE overhead from 2× to only 9% for sequential writes. In KVFS, a random 4KB write becomes a key-value pair insertion into KVDB. In KVDB, these random insertions hit the disk as serially written SSTables, so KVFS's random write performance is better than even native Ext4. Random writes in KVFS are at least 2× faster than Ext4 and FUSE-Ext4. Inserting lots of pairs results in frequent merges; and since the data is random, stitching does not help, making KVFS's random-write performance worse than KVFS's serial-write performance, but still much better than other file systems. Using QFs and secondary indexes, KVDB achieves the SSD's random-read throughput for random point queries, making random reads in KVFS comparable to FUSE-Ext4 and Ext4.

4.2.2 File System Macro-Benchmarking

The FileServer workload performs a sequence of creates, deletes, appends, reads, writes, and stat operations on a directory tree. We configured the mean file size to 32KB, mean append size to 8KB, directory width to 20, and number of files to 400,000. Filebench pre-allocates 80% of the files and randomly selects a file for each of the above operations. We ran the benchmark on an SSD for ten minutes, using ten threads, with 4KB I/O size. The WebServer workload produces a sequence of open-read-close calls with 100 threads on 1,000 files in a directory tree, plus a single file-append thread with a mean append size of 16KB for every ten read threads. For the read cycle, the benchmark randomly selects and reads an entire file. We omit macro-benchmarks results for FUSE-

	FileServer	WebServer
Ext4	7,371	20,626
FUSE-Ext4	7,272	19,215
KVFS	9,015	19,191

Table 3: Filebench macro-benchmark results (ops/s)

Ext4-no-wc, having shown that it has slower write performance and similar read performance to FUSE-Ext4.

Results. As seen in Table 3, for both macro-benchmarks, KVFS performs similar to or better than FUSE-Ext4 and Ext4. In each iteration, the FileServer workload includes three meta-data file operations—open or create, delete, and stat. Examining the ops/sec for each operation, we observed that KVFS outperforms FUSE-Ext4 and Ext4 for these meta-data operations. Meta-data operations are similar to random database workloads, consisting of small tuples, and are ideal for the VT-tree. FileServer also performs 8KB appends to randomly chosen files, which also behave like random writes when there are 400,000 files. For randomly reading the whole file, all file systems have similar performance. In KVFS, compactions of on-disk lists in the VT-tree bring randomly appended data together over the lifetime of the file. Frequent insertions of data in the file-server workload ensure that the compactions are triggered often, improving performance of subsequent sequential reads. The frequency of compaction is determined by the insertion rate and can also be triggered periodically, to further improve KVFS's performance for whole-file reads. On average, KVFS performs 18% better than Ext4 and FUSE-Ext4. For the WebServer workload, KVFS is 7% slower than Ext4 and comparable to FUSE-Ext4. The WebServer workload is mainly read oriented, and files are not large enough to benefit from FUSE's readaheads. Ext4 performs better for the read cycle of the workload and hence better overall.

4.3 KVFS's Transactional Performance

We used a Linux kernel compilation and related operations to compare KVFS's performance to Ext4 and FUSE-Ext4. For KVFS, we ran the same set of operations with and without transactions to measure transactional overheads. We ran transaction benchmarks in both snapshot-isolation (KVFS-TXN-SI) and serializability (KVFS-TXN-SER) modes. In this benchmark we include the following operations in this order:

1. txn-start: begin a transaction; used only for KVFS when it is running in a transaction.
2. untar: Untar the downloaded source tarball.
3. removetar: Remove the downloaded tarball.
4. make: Run `make` inside the source directory.
5. findall: Run `/bin/find` on the source directory.
6. readall: recursively read all files in the source directory reading each file and writing it to `/dev/null`.

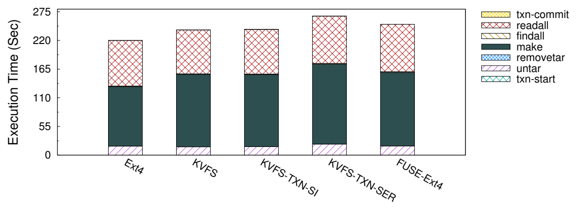


Figure 8: Linux kernel compilation results

7. txn-commit: commit a transaction; used only for KVFS when it is running in a transaction.

We ran single- and multi-threaded versions of this test, to show that transactions are highly concurrent. The single-threaded version performs these operations on a Linux kernel version 3.0.1 source tarball. The multi-threaded version operates on four source tarballs—for versions 3.0.1, 3.1.1, 3.2.1, and 3.3.1—in four concurrent transactions.

Results. Figure 8 shows the result for all file systems for operations on the single Linux kernel source. Ext4 completed the benchmark in 220s, FUSE-ext4 in 250s, and KVFS in 240s. Here, compilation took around 52% of the total time and `readall` took 40% of the time. After compilation, there were 45,737 files left with a total size of 739MB. KVFS performs comparable or superior to Ext4 on all operations except the compilation; KVFS is 17% slower than Ext4. FUSE-Ext4 is also 17% slower than Ext4 for compilation. The workload in this run fits entirely in RAM, and FUSE has a higher overhead for CPU-bound operations. Also, for the compilation workload, our FUSE write-back cache did not help. Compilation workload has lot of meta-data operations that need to be synchronous; also after writing to the object files, `make` closes them immediately. As we start `make` on a cold cache, the write-back cache is empty initially. Our write-back cache requires writing back the dirty blocks of a file to its user-level file system when that file is closed. This is because the FUSE kernel module frees the internal file handle on a close request. So, for small object files getting created during compilation, the FUSE write-back cache saves little. KVFS-TXN-SI completed in 241s, almost no overhead compared to the KVFS run with no transaction. This is because the time spent in `txn-start` is negligible and `txn-commit` took only 0.28s to complete. `txn-start` involves taking a snapshot and creating new writable schemas. `txn-commit` requires moving the SSTables from the transaction's private snapshot to the mainline schema and removing the entry for the snapshot directory from the kernel cache. As described in Section 3.2, these are quick operations in KVFS. The transaction in serializability mode shows an overhead of 10% when compared to KVFS running without transactions. As serializability needs to maintain strong isola-

	Ext4	KVFS	KVFS-TXN-SI	FUSE-Ext4
Time (sec)	453	514	491	571

Table 4: Parallel Linux kernel compilation results

tion guarantees, these overheads mainly come from the range-lock tree and deadlock-avoidance checks.

Table 4 shows the results for the second benchmark. In each run, the benchmark picks a file system and runs all operations (untar, removetar, make, ..., readall) in parallel on four Linux kernels. KVFS performed 11% better than FUSE-Ext4 and 13.4% slower than Ext4. Having four Linux kernel sources forces the workload out of RAM. KVFS achieved better throughput as four parallel compilations produced more randomness, suitable for KVFS's faster random writes (see Section 4.2.1). For running transactions in KVFS, the SI mode is best suited as these operations do not have conflicting requests. The KVFS-TXN-SI run completed in 491s, 4.5% faster than without transactions. We ran each of these four benchmarks in separate transactions in SI mode. All the transactions worked in their own snapshot directory without stepping over each other and they commit their changes at the end. These snapshots had their own VT-tree with a smaller private cache. Thus, concurrently running transactions can perform better than the run without transactions. Our evaluation of SI-mode transactions in KVDB showed that running a multithreaded partitionable workload in separate transactions can increase throughput by 67%. This micro-benchmark uses only writes compared to all the operations performed in KVFS-TXN-SI. Due to lock-contention, writes benefit more from KVFS's transactional architecture. Also, as the benchmark ran at the KVDB layer, it had no FUSE overhead. We omit these results for brevity.

4.4 Defragmentation in KVFS

Log-structured systems face the issue of fragmentation; KVFS, which uses VT-trees, is no exception. We have devised a defragmentation algorithm described in Section 3.1. We measured the effectiveness of our defragmentation algorithm using Filebench's fileserver write-only workload. We run this benchmark twice: once with a stitching threshold of 64KB and once without stitching. From our analysis, omitted here, having smaller threshold like 64KB results in higher fragmentation. Filebench supports configuring the number of iterations it performs. We configured it the same between the two runs to ensure that the amount of data inserted in these two runs is the same. This benchmark inserts into KVFS around 38GB of *dmap* pairs representing the file data blocks. There will be no fragmentation if the workload is either completely random or sequential. Fileserver's random and large, random-append writes ensures that we stitch some small amount of data and copy the rest, resulting in a worst-case-like fragmentation.

Results. The run with stitching finished inserting 38GB of data in 855s. After inserting all the data, the total number of used zones was 9,477. But the number of required zones was only 4,864—a 47% space waste due to fragmentation. After scanning through all the lists, our algorithm determined that it can reclaim 4,483 zones. The remaining 130 zones were either used by the lists at the top or by the disk-backed secondary indexes and QFs. Before running our defragmentation (DF) algorithm, we ran a range query in our KVDB, to read all the *dmap* pairs in sorted order. As the data is sorted only within each list and not across lists, this operation needs to perform more work than sequentially reading all the lists. This can be worse if there is any fragmentation due to stitching. This operation took around 331s, reading at 117MB/s; we compare figure after DF. As our DF considers the sequentiality of existing data before moving it around, it should improve the performance of the range operation. DF required moving around 4.8GB of data from the candidate zones to the rest of the zones. Our DF algorithm reclaimed all 4,483 zones in only 96s with a rate of 51.4MB/s. We sequentiality read the data from one zone and writes it to the new sequentially. Because we use first fit, these writes may not be contiguous. We achieve an acceptable throughput of 51.4MB/s considering that the data needs to be read from and be written to different locations. The range query operation after the DF took only 238s, an improvement of 29%. The run without stitching inserted 38GB of data in 1,982s, which is 2.3× slower; however, this run resulted in no fragmentation. The same range query took 228s to read the entire data back. Considering the DF time taken for the run with stitching, the run without stitching is still 2× slower. Also, the degradation in range-query performance is only 4%. We concluded that in KVDB, stitching plus an occasional DF gives a better insertion throughput with negligible loss in read performance.

5 Related Work

Write-optimized databases. GTSSL [33] uses an LSM-tree variant [21] with a multi-tier extension. GTSSL supports ACID transactions, but does not support snapshots and transactions larger than RAM, useful for a transactional file system [32]. GTSSL also lacks the VT-tree's sequential optimizations. VT-trees can be easily extended to support multi-tier storage. Anvil [17] is a modular framework for applications to use different data layouts. Many of its disk formats resemble LSM-trees and could benefit from the VT-tree design. Cassandra [8] and Hbase [9] also lack these sequential optimization and their Java-based implementation hinders performance by 2–5× [33].

Log-structured file systems. Log-structured file systems (LFS) [26] write the changes to a circular log and do not overwrite the file data on disk. LFSs require that all keys in their data structure reside in RAM to avoid disk I/O boundedness [36]. This works well for file systems that deal with large 4KB pages, but becomes a performance bottleneck for smaller tuples. KVFS, however, was designed to support smaller and larger tuples efficiently. KVFS behaves like an LFS for file-system workloads and like an LSM for database workloads, performing equally well on both types of workloads.

Transactional file systems and OSes. Amino [39] uses `ptrace` to interpose file system operations, to enforce transactions; `ptrace` introduces high overheads. Valor [32] adds in-kernel logging for write-ordering and locking to provide a transactional file interface. Valor supports long-lived transactions and transactions larger than RAM. QuickSilver [28] is one of the first to incorporate transactions into the OS, but each component had to support two-phase commit and rollback. TxOS [23] detects conflicts and uses data versioning at the virtual file system layer to achieve transaction-like behavior. TxOS does not support transactions larger than RAM. TxF [37] uses the existing transactional manager in NTFS. TxF does not operate on a log-structured file system and instead uses a traditional transactional architecture. Therefore, when data is overwritten, TxF writes it twice and must gather undo images using reads. All these transactional file system interfaces either add high overhead, require complex kernel changes, or require a complete redesign. KVFS, conversely, supports long-lived and larger-than-RAM transactions, requires no kernel modifications, introduces negligible overhead, and uses a simpler design.

File system meta-data indexing. FFS's optimizations for directory lookups and insertions are simple but do not scale as well as B-trees. XFS [34], ReiserFS [25], Ext3, and Btrfs [22] use the better B+ trees to index path names. Perspective [27] offers a user-friendly distributed file system based on MySQL. BeFS [12] and HFaD [29] focus on multiple indexes for the same file, using traditional B+ trees. Randomly written B+ trees can be fairly inefficient, even for SSDs [31, 33]. TokuFS [10] is a prototype file system based on fractal-trees. It is designed to support many, small, partial or unaligned writes, by converting a read-modify-write operation to a single write or an insert. In VT-tree, updates are already treated as inserts and it is easy to incorporate support partial writes as TokuFS. TokuFS, however, does not perform well for large sequential writes, and does not provide a transactional file system interface [10]. Spyglass [15] partitions namespace meta-data into smaller KD-trees [6] that fit in RAM; it writes updated KD trees in new locations and links them to previous versions to increase update throughput. The authors admittedly avoid tree compaction [15], which is often required [7]; they also do not specify asymptotic performance or compare against other write-optimized indexes. The VT-tree, however, scales well when indexes exceed RAM and works well for any type of hard-to-partition indexed data (e.g., dedup, Web search index).

6 Conclusions

Our main goal was to build a transactional storage system that supports both file system and database workloads efficiently. Our VT-tree performs well for both highly sequential and highly random workloads, as well as any mixes thereof. Our stitching feature improves sequential and file-system workloads' performance, but increases fragmentation; our defragmentation technique allows us to balance these two conflicting needs. We designed an efficient transactional architecture with negligible overhead for single asynchronous transactions—that can improve highly concurrent performance by up to 67%. Transactions requiring strong isolation incur only a 10% overhead. We designed a simple write-back caching for FUSE that allows KVFS to better compete with in-kernel file systems. Our KVFS performance is comparable and sometimes superior to Ext4, showing KVFS's wide practicality. Finally, KVFS provides snapshots and low-overhead concurrent transactions, and it leverages FUSE's caches without violating consistency.

Future work. Stitching can cause file system fragmentation over time. We currently have an offline defragmentation tool. We plan to extend it to run online and also explore other DF techniques such as using LFS-style cleaning when no zone can be evacuated to free space. Most of the framework for an online DF is already in KVDB. Since our algorithm does not overwrite any existing data, reads can happen in parallel. KVDB's transactional nature with internal journaling mechanism allows easy handling of crash recovery while running defragmentation. Unless KVDB runs out of space, new writes can continue to come in as they land on new SSTables. Additional work KVDB would need is a throttling mechanism—to throttle incoming writes when their rate is higher than the DF throughput and the system is out of usable space.

Acknowledgements. We thank the anonymous USENIX reviewers for their valuable feedback. This work was supported in part by NSF awards CCF-0937833 and CCF-0937854.

References

[1] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, pages 3–3, San Jose, CA, February 2007. USENIX Association.

[2] Apple, Inc. iTunes. `www.apple.com/itunes/`.

[3] Apple, Inc. Working with Spotlight. http://developer.apple.com/macosx/spotlight.html.

[4] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J.Nelson. Cache-oblivious streaming b-trees. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 81–92, New York, NY, USA, 2007. ACM.

[5] M. A. Bender, M. Farach-Colton, R. Johnson, , R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. In *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB '12)*, Istanbul, Turkey, August 2012. Morgan Kaufmann.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

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

[8] Cassandra Documentation. Getting started. http://wiki.apache.org/cassandra/GettingStarteda, 2011.

[9] HBase Documentation. Hbase: Bigtable-like structured storage for hadoop hdfs. http://wiki.apache.org/hadoop/Hbasea, 2011.

[10] John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. The tokufs streaming file system. In *Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems*, HotStorage'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.

[11] Filebench. http://filebench.sourceforge.net.

[12] D. Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[13] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[14] Intel Inc. Over-provisioning an intel ssd. Technical Report 324441-001, Intel Inc., October 2010. cache-www.intel.com/cd/00/00/45/95/459555_459555.pdf.

[15] A. W. Leung, M. Shawo, T. Bisson, S. Pasupathy, and E. L. Miller. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *FAST '09: Proceedings of the 7th USENIX conference on File and Storage Technologies*, Berkeley, CA, USA, 2009. USENIX Association.

[16] LevelDB, January 2012. http://code.google.com/p/leveldb.

[17] M. Mammarella, S. Hovsepian, and E. Kohler. Modular data storage with anvil. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 147–160, New York, NY, USA, 2009. ACM.

[18] Microsoft Corporation. Microsoft MSDN WinFS Documentation. http://msdn.microsoft.com/data/winfs/, October 2004.

[19] Neeraj Mittal and Hui-I Hsiao. Database managed external file update. In *ICDE'01*, pages 557–564, 2001.

[20] N. Murphy, M. Tonkelowitz, and M. Vernal. The Design and Implementation of the Database File System. www.eecs.harvard.edu/~vernal/learn/cs261r/index.shtml, January 2002.

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

[22] Oracle. Btrfs, 2008. http://oss.oracle.com/projects/btrfs/.

[23] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 161–176. ACM, 2009.

[24] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *25th Symposium On Applied Computing*. ACM, March 2010.

[25] H. Reiser. ReiserFS v.3 Whitepaper. http://web.archive.org/web/20031015041320/http://namesys.com/.

[26] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-structured File System. In *ACM Transactions on Computer Systems (TOCS)*, pages 26–52. ACM, 1992.

[27] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger. Perspective: Semantic data management for the home. In *FAST '09: Proceedings of the 7th USENIX conference on File and Storage Technologies*, Berkeley, CA, USA, 2009. USENIX Association.

[28] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 239–253, Pacific Grove, CA, October 1991. ACM Press.

[29] M. Seltzer and N. Murphy. Hierarchical file systems are dead. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, 2009.

[30] M. I. Seltzer. Transaction Support in a Log-Structured File System. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 503–510, Vienna, Austria, April 1993.

[31] R. Spillane, S. Dixit, S. Archak, S. Bhanage, and E. Zadok. Exporting kernel page caching for efficient user-level I/O. In *Proceedings of the 26th International IEEE Symposium on Mass Storage Systems and Technologies*, Incline Village, Nevada, May 2010. IEEE.

[32] R. P. Spillane, S. Gaikwad, E. Zadok, C. P. Wright, and M. Chinni. Enabling transactional file access via lightweight kernel extensions. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, pages 29–42, San Francisco, CA, February 2009. USENIX Association.

[33] R. P. Spillane, P. J. Shetty, E. Zadok, S. Archak, and S. Dixit. An efficient multi-tier tablet server storage architecture. In *2nd ACM Symposium on Cloud Computing*. ACM, Oct 2011.

[34] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 1–14, San Diego, CA, January 1996.

[35] M. Szeredi. Filesystem in Userspace. `http://fuse.sourceforge.net`, February 2005.

[36] Andy Twigg, Andrew Byde, Grzegorz Milos, Tim Moreton, John Wilkes, and Tom Wilkie. Stratified b-trees and versioned dictionaries. In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, HotStorage'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[37] S. Verma. Transactional NTFS (TxF). `http://msdn2.microsoft.com/en-us/library/aa365456.aspx`, 2006.

[38] Microsoft's Windows Search. `http://http://msdn.microsoft.com/en-us/library/windows/desktop/ff628790%28v=vs.85%29.aspx`.

[39] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID Semantics to the File System. *ACM Transactions on Storage (TOS)*, 3(2):1–42, June 2007.