An Efficient Multi-Tier Tablet Server Storage Architecture

Appears in the proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC 2011)

Richard P. Spillane
rick@fsl.cs.sunysb.edu
Stony Brook University

Pradeep J. Shetty
pshetty@fsl.cs.sunysb.edu
Stony Brook University

Erez Zadok
ezk@cs.sunysb.edu
Stony Brook University

Shrikar Archak
sarchak@cs.sunysb.edu
Stony Brook University

Sagar Dixit
ssdixit@cs.sunysb.edu
Stony Brook University

ABSTRACT

Distributed, structured data stores such as Big Table, HBase, and Cassandra use a cluster of machines, each running a database-like software system called the Tablet Server Storage Layer or *TSSL*. A TSSL's performance on each node directly impacts the performance of the entire cluster. In this paper we introduce an efficient, scalable, multi-tier storage architecture for tablet servers. Our system can use any layered mix of storage devices such as Flash SSDs and magnetic disks. Our experiments show that by using a mix of technologies, performance for certain workloads can be improved beyond configurations using strictly two-tier approaches with one type of storage technology. We utilized, adapted, and integrated cache-oblivious algorithms and data structures, as well as Bloom filters, to improve scalability significantly. We also support versatile, efficient transactional semantics. We analyzed and evaluated our system against the storage layers of Cassandra and Hadoop HBase. We used wide range of workloads and configurations from read- to write-optimized, as well as different input sizes. We found that our system is 3–10× faster than existing systems; that using proper data structures, algorithms, and techniques is critical for scalability, especially on modern Flash SSDs; and that one can fully support versatile transactions without sacrificing performance.

1. INTRODUCTION

In recent years, many scientific communities are finding that they are not limited by CPU or processing power, but instead they are being drowned by a new abundance of data, and are searching for ways to efficiently structure and analyze it. For example, Schatz argues that with the exponentially decreasing cost of genome sequencing [54], data to analyze is increasingly more abundant. The National Radio Astronomy Observatory is hosting a workshop to focus on ways of "extracting the science from the data" [43]. Scientific researchers in fields ranging from archaeobiology [32] to atmospheric science [49] are searching for ways to store, and then *analyze* big data. To solve peta-scale problems, researchers have turned to database clustering software such as the Big Table inspired Hadoop HBase [5], or the Dynamo-inspired Cassandra [33].

How these systems interact with underlying storage devices is a critical part of their overall architecture. Cluster systems comprise many individual machines that manage storage devices. These machines are managed by a networking layer that distributes to them queries, insertions, and updates. Each machine runs database or database-like software that is responsible for reading and writing to the machine's directly attached storage. Figure 1 shows these individual machines, called *tablet servers*, that are members of the larger cluster. We call the portion of this database software that communicates with storage, the *Tablet Server Storage Layer* (TSSL). For example, Hadoop HBase [5], a popular cluster technology, includes a database, networking, and an easier-to-program abstraction above their TSSL (*logical layer*).

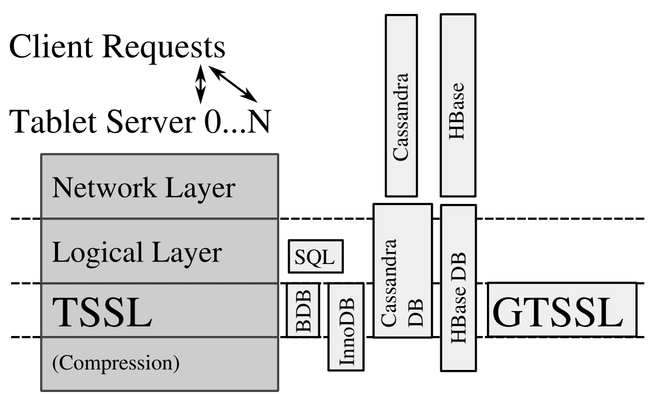

Figure 1: Different storage and communication technologies are used at different layers in a typical database cluster architecture.

The performance and feature set of the TSSL running on each node affects the entire cluster significantly. If performance characteristics of the TSSL are not well understood, it is difficult to profile and optimize performance. If the TSSL does not support a critical feature (e.g., transactions), then some programming paradigms can be difficult to implement efficiently across the entire cluster (e.g., distributed and consistent transactions).

It takes time to develop the software researchers use to analyze their data. The programming model and abstractions they have available to them directly affect how much time development takes. This is why many supercomputing/HPC researchers have come to rely upon structured data clusters that provide a database interface, and therefore an efficient TSSL [1, 49].

One of the most important components to optimize in the cluster is the TSSL. This is because affordable storage continues to be orders of magnitude slower than any other component, such as the CPU, RAM, or even a fast local network. Any software that uses storage heavily, such as major scientific applications, transactional systems [40, 64], databases [45], file systems [31, 57, 58, 61, 65], and more, are designed around the large difference in performance between RAM and storage devices. Using a flexible cluster architecture that can scale well with an increasing number of nodes is an excellent way to prorate these storage and computing costs [14]. Complimentary to this effort is increasing the *efficiency* of each of these nodes' TSSLs to decrease overall computing costs.

We expect that the architecture we outline here, along with our extensions to the compaction algorithms typically used, and our software will have a high level of transferability to a larger cluster installation. Others have transferred a separately designed TSSL over to a cluster framework. Abouzeid et al. splice a Hadoop distribution framework on top of PostgreSQL [50], and Vertica [66] also supports similar integration of its column-store with Hadoop. Determining the performance improvement for a Hadoop cluster using a more efficient TSSL is a subject of future work.

In this work we present a new, highly scalable, and efficient TSSL architecture called the *General Tablet Server Storage Layer* or GTSSL. GTSSL employs significantly improved compaction algorithms that we adapted to *multi-tier storage architectures*, architectures that mix together multiple storage devices (e.g., magnetic disk and Flash SSD) to improve performance to cost ratio for important workloads. GTSSL aggressively uses advanced data structures and algorithms to improve efficiency while fully integrating versatile and efficient transactions into its total architecture. By focusing on a single node we were better able to understand the performance and design implications of these existing TSSLs, and how they interact with Flash SSD devices when using small as well as large data items.

We designed and developed GTSSL using the insights we acquired from our analysis of the Cassandra and HBase TSSLs. In this paper we detail both our theoretical and benchmarked evaluation of existing TSSLs, and we introduce our GTSSL solution. Specifically, our contributions are:

- We improved data compaction algorithms significantly, and adapted them to multi-tier storage architectures. We discuss multi-tier experiments for TSSL architectures in Section 5.4. According to our survey of related work (Section 6), this is the first paper to discuss a multi-tier design and experiments.
- We aggressively use advanced algorithms, data structures, and Bloom filters to achieve 3–10× faster lookups (reads), and 5× faster insertions (writes) over Cassandra and HBase.
- We integrated versatile and efficient transactions without compromising performance.
- We include an empirical and theoretical evaluation of GTSSL, the Cassandra TSSL, and the HBase TSSL. We especially evaluated a wide range of configurations from read-optimized to write-optimized.
- We evaluated a wide range of input sizes and found that performance can become CPU-bound on small input sizes with Flash SSD devices.
- We demonstrate that a write-optimized TSSL architecture can remain efficient for transactional workloads in comparison to Berkeley DB and MySQL's InnoDB.

Supercomputing research must solve large data problems quickly. Data sets are now measured in petabytes, soon in exabytes [52]. Understanding and improving the performance and architecture of the cluster software that interacts most directly with storage is critical for integrating the next generation of storage technology.

We introduce the standard TSSL architecture and terminology in Section 2. We theoretically analyze existing TSSL compaction techniques in Section 3. In Section 4 we introduce GTSSL's design and compare it to existing TSSLs. Our evaluation, in Section 5, compares the performance of GTSSL to Cassandra and HBase, and GTSSL's transactional performance to Berkeley DB and MySQL's InnoDB. We discuss related work in Section 6. We conclude and discuss future work in Section 7.

2. BACKGROUND

As shown in Figure 1, the data stored in the TSSL is accessed through a high-level logical interface. In the past, that interface has been SQL. However, HBase and Cassandra utilize a logical interface other than SQL. As outlined by Chang et al. [10], HBase, Cassandra, and Big Table organize structured data as several large tables. These tables are accessed using a protocol that groups columns in the table into *column families*. Columns with similar compression properties, and columns which are frequently accessed together, are typically placed in the same column family. Figure 1 shows that the logical layer is responsible for implementing the column-based protocol used by clients, using an underlying TSSL.

The TSSL provides an API that allows for atomic writes, as well as lookups and range queries across multiple trees which efficiently store variable length key-value pairs or *pairs* on storage. These trees are the on-storage equivalent of column families. Although TSSLs transactionally manage a set of tuple trees to be operated on by the logical layer, TSSL design is typically different from traditional database design. Both the TSSL and a traditional database perform transactional reads, updates, and insertions into multiple trees, where each tree is typically optimized for storage access (e.g., a B+-tree [12]). In this sense, and as shown in Figure 1, the TSSL is very similar to an embedded database API such as Berkeley DB (BDB) [60], or a DBMS storage engine like MySQL's InnoDB. However, unlike traditional database storage engines, the majority of insertions and lookups handled by the TSSL are *decoupled*. A decoupled workload is one where transactions either perform only lookups, or only inserts, updates, or deletes. This workload is important because in a clustered system, one process may be inserting a large amount of data gathered from sensors, a large corpus, or the web, while many other processes perform lookups on what data is currently available (e.g., search). Furthermore, most of these insertions are simple updates, and do not involve large numbers of dependencies across multiple tables. This leads to two important differences from traditional database storage engine requirements: (1) most insertions do not depend on a lookup, not even to check for duplicates, and (2) their transactions typically need only provide atomic insertions, rather than support multiple read and write operations in full isolation. We call the relaxation of condition (1) *decoupling*, and it permits the use of efficient *write-optimized* tree data-structures that realize vastly higher insertion throughputs, and are very different from traditional B+-trees. We call the relaxation of condition (2) *micro-transactions*; it enables using a simple, non-indexed, redo-only journal. Thus, the TSSL need not support a mix of asynchronous and durable transactions.

Compaction.

Fast TSSL insertions performance is key to realizing cheaper large structured data clusters. If each node inserts faster, then fewer nodes are required to meet a target insertion throughput.

To achieve faster TSSL insertion performance, HBase, Cassandra, Big Table, and other systems such as Hypertable [27] do not use a traditional tree structure, but a tree-like structure that exploits decoupling. This tree-like structure writes sorted buffers to storage, which are then asynchronously merged into larger buffers by a process called *compaction*. Although the compaction algorithms of these designs differ, the overall goal is the same.

All these TSSL designs maintain separate caches for each tree in RAM. These caches are sorted arrays called *memtables*. Every write operation inserts a value into this cache. Once the cache exceeds a pre-configured size, it and its associated metadata structures are serialized to disk in one sequential write, and the memtable is marked clean. The serialized memtable is called an *SSTable*.

SSTables are divided into a data portion, and a metadata portion. The data portion includes all the pairs stored within the SSTable, and these pairs are interspersed with clusters of offsets that point to individual pairs. A cluster and its associated pairs can be read in a single IO. The smallest amount of pairs in bytes between two clusters of offsets is called the *block size*, and the block size of Cassandra (default 256KB), HBase (default 64KB), and GTSSL (default 4KB) are all configurable. We discuss the effects of these block size choices both on magnetic and Flash storage medium in Section 5.2. The metadata portion consists of a secondary index and Bloom filter. The SSTable's secondary index associates the first key of each cluster to the offset of each cluster within the data portion, facilitating single IO lookups into an SSTable, as the secondary index is always resident in RAM. The SSTable's Bloom filter was populated when the SSTable was a memtable, is serialized after the secondary index when the SSTable is created, and is used during lookup to avoid IO when possible.

To perform a read on a tree, each SSTable on storage and the memtable are all queried: the closest, most recent value is returned. Each of these SSTable-queries requires only a single IO. As more values are inserted, the number of SSTables can grow unbounded. To limit the number of these tables, and the cost of lookup and scan, therefore, an asynchronous *merging compaction* process often merges together smaller SSTables into larger SSTables. These merges can be performed efficiently as the SSTables are sorted.

Periodically, a *major compaction* is performed. This major compaction merges all SSTables belonging to a tree into one. At this time we process deletes. During normal operations, to delete a tuple, a new tuple with a matching key but an additional DELETE flag set is inserted into the tree. These tuples are ignored during merging compactions, but during major compactions, they effectively cancel out the matching tuples: the major merging process simply omits the delete and matching tuples from the output list.

3. TSSL COMPACTION ANALYSIS

GTSSL was designed to scale to a multi-tier storage hierarchy, and much of how compaction works must be re-thought. We analyze and compare the existing compaction methods employed by HBase and Cassandra, and then in Section 4, we introduce what extensions are necessary in a multi-tier regime.

We analyzed the compaction performance of Cassandra, HBase, and our GTSSL using the Disk-Access Model (DAM) for cost, and using similar techniques as those found in Bender et al.'s work on the cache-oblivious look-ahead array [7]. The DAM is a simple yet sufficiently accurate cost model. DAM divides the system into a memory M and storage S. The unit of transfer from S to M is a block of b bytes. Operations and manipulations of data in M are free, but every block transferred either from M to S or from S to M costs 1. For the remainder of this analysis, we will use $B = \frac{b}{\text{pair size}}$ instead of b. Approximately, this means each data structure is penalized 1 unit if either one random pair, or B serial pairs were transferred from M to S, or from S to M.

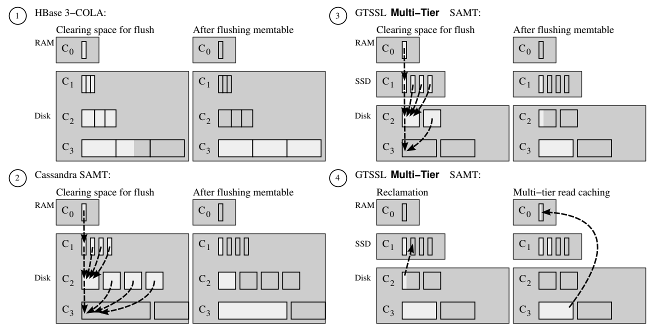

Figure 2: In panel ①, HBase merges C_0, C_1, C_2, and half of C_3 back into C_3, like a 3-COLA would. In panel ②, Cassandra merges buffers in quartets to create space for a flushing memtable. In panels ③ and ④, GTSSL merges, and then promotes the resulting SSTable up into a higher tier. Subsequent reads are also cached into the higher tier via re-insertion.

HBase Analysis.

HBase [18] is a variation of the Cache-Oblivious Lookahead Array (R-COLA) [7]. The R-COLA supports increasingly more read-optimized configurations as its R parameter is increased. HBase sets $R = 3$, which is optimal in practice for the R-COLA [7], so we call HBase's configuration a *3-COLA*. Figure 2, panel ①, shows the R-COLA consists of $\lceil \log_R N \rceil$ arrays of exponentially increasing size, stored contiguously (C_0 through C_3), for N the number of elements inserted into the data structure. In this example, $R = 3$. C_1 through C_3 on storage can be thought of as three SSTables, and C_0 in RAM can be thought of as the memtable. When the memtable is serialized to disk (and turned into an SSTable), the R-COLA checks to see if level 0 is full. If not, it performs a merging compaction on level 0, on all adjacent subsequent arrays that are also full, and on the first non-full level, into that same level. In Figure 2's example, C_0 through C_3 are merged into C_3; after the merge, the original contents of C_3 have been written twice to C_3. Each level can tolerate $R - 1$ merges before it too must be included in the merge into the level beneath it. This means that every pair is written $R - 1$ times to each level.

Bender et al. provide a full analysis of the R-COLA, but in sum, the amortized cost of insertion is $\frac{(R-1)\log_R N}{B}$, and the cost of lookup is $\log_R N$. This is because every pair is eventually merged into each level of the R-COLA; however, it is repeatedly merged into the same level $R-1$ times due to subsequent merges. So for N total pairs inserted, each pair would have been written $R - 1$ times to $\log_R N$ levels. As all pairs are written serially, we pay 1 in the DAM for every B pair written, and so we get $\frac{(R-1)\log_R N}{B}$ amortized insertion cost. A lookup operation must perform 1 random read transfer in each of $\log_R N$ levels for a total cost of $\log_R N$. Bender et al. use fractional cascading [11] to ensure only 1 read per level. Practical implementations and all TSSL architectures, however, simply use small secondary indexes in RAM.

By increasing R, one can decrease lookup costs in exchange for more frequent merging during insertion. HBase sets $R = 3$ by default, and uses the R-COLA compaction method. HBase adds additional thresholds that can be configured. For example, HBase performs major compactions when the number of levels exceeds 7.

SAMT Analysis.

The R-COLA used by HBase has faster lookups and slower insertions by increasing R. GTSSL and Cassandra, however, can both be configured to provide faster *insertions* and slower lookups by organizing compactions differently. We call the structure adopted by Cassandra's TSSL and GTSSL, the *Sorted Array Merge Tree* (SAMT). As shown in Figure 2, panel ②, rather than storing one list per level, the SAMT stores K lists, or *slots* on each level. The memtable can be flushed K times before a compaction must be performed. At this time, only the slots in C_1 are merged into a slot in C_2. In the example depicted, we must perform a cascade of compactions: the slots in C_2 are merged into a slot in C_3, so that the slots in C_1 can be merged into a slot in C_2, so that the memtable in C_0 can be serialized to a slot in C_1. As every element visits each level once, and merges are done serially, we perform $\frac{\log_K N}{B}$ disk transfers per insertion. Because there are K slots per level, and $\log_K N$ levels, we perform $K \log_K N$ disk transfers per lookup. The cost of lookup with the SAMT is the same for $K = 2$ and $K = 4$, but $K = 4$ provides faster insertions. So $K = 4$ is a good default, and is used by both GTSSL and Cassandra.

Comparison.

We now compare the SAMT compaction algorithm to the COLA compaction algorithm, and show how utilization of Bloom filters permits the SAMT algorithm to out-perform the COLA algorithm in most cases, especially on Flash SSD. Although the HBase 3-COLA method permits more aggressive merging during insertion to decrease lookup latency by increasing R, it is unable to favor insertions beyond its default configuration. This permits faster scan performance on disk, but for 64B or larger keys, random lookup performance is already optimal for the default configuration. This is because for the vast majority of lookups, Bloom filters [8] on each SSTable avoid all $\log_R N$ SSTables except the one which contains the sought after pair. Furthermore, on Flash SSD the 3-COLA is less optimal, as the seeking incurred from scanning is mitigated by the Flash SSD's obliviousness toward random and serial reads.

Conversely, the SAMT can be configured to further favor insertions by increasing K, while maintaining lookup performance on Flash SSD and disk by using Bloom filters, and maintaining scan performance on Flash SSD. Although Bloom filters defray the cost of unnecessary lookups in SSTables, as the number of filters increases, the total effectiveness of the approach decreases. When performing a lookup in the SAMT with a Bloom filter on each SSTable, the probability of having to perform an unnecessary lookup in some SSTable is $1-(1-f)^{N_B}$ where N_B is the number of Bloom filters, and f is the false positive rate of each filter. This probability is roughly equal to $f * N_B$ for reasonably small values of f. In our evaluation, Bloom filters remain effective as long as the number of SSTables for each tree/column-family is less than 40.

4. DESIGN AND IMPLEMENTATION

We studied existing TSSLs (Cassandra and HBase) as well as existing DBMS storage engines (Berkeley DB and InnoDB). This guided GTSSL's design. GTSSL utilizes several novel extensions to the SAMT (discussed in Section 3). As shown in Figure 2 panels ③ and ④, GTSSL supports storage device specific optimizations at each tier. GTSSL intelligently migrates recently written *and read* data between tiers to improve both insertion and lookup throughput and permit effective caching in storage tiers larger than RAM.

TSSL efficiency is critical to overall cluster efficiency. GTSSL extends the scan cache (described in Section 2) and buffer cache architecture used by existing TSSLs. GTSSL completely avoids the need to maintain a buffer cache while avoiding common MMAP overheads; GTSSL further aggressively exploits Bloom filters so they have equal or more space in RAM than the scan cache.

Although Web-service MapReduce workloads do not typically require more than atomic insertions [10], parallel DBMS architectures and many scientific workloads require more substantial transactional semantics. GTSSL introduces a light-weight transactional architecture that allows clients to commit transactions as either durable or non-durable. Durable transactions fully exploit group-commit as in other TSSL architectures. However, GTSSL also allows non-durable transactions, and these can avoid writing to the journal completely for heavy insertion workloads without compromising recoverability. In addition, GTSSL provides the necessary infrastructure to support transactions that can perform multiple reads and writes atomically and with full isolation.

We discuss how we improved the SAMT structure so that it could operate in a multi-tier way that best exploits the capabilities of different storage devices in Section 4.1. We detail our caching architecture and design decisions in Section 4.2. We discuss GTSSL's transactional extensions to the typical TSSL in Section 4.3.

4.1 SAMT Multi-Tier Extensions

GTSSL extends the SAMT merging method in three ways. (1) Client reads can be optionally re-inserted to keep recently read (hot) data in faster tiers (e.g., a Flash SSD). (2) Lists of recently inserted data are automatically promoted into faster tiers if they fit. (3) Different tiers can have different values of K (the number of slots in each level; see Section 3). We call our improved SAMT the Multi-Tier SAMT or *MTSAMT*. In addition, our implementation also includes support for full deletion, variable-length keys and values, and allows the logical layer to specify whatever format, bits, or timestamps deemed necessary by the logical layer, as other TSSLs do (see Section 2).

Re-Insertion Caching.

Whenever a pair is inserted, updated, deleted, or read, the C_0 (fastest) cache is updated. The cache is configured to hold a preset number of pairs. When a pair is inserted or updated, it is marked DIRTY, and the number of pairs in the cache is increased. Similarly, after a key is read into the C_0 cache, it is marked as RD_CACHED, and the number of pairs is increased. Once a pre-set limit is met, the cache *evicts* into the MTSAMT structure using the merging process depicted in Figure 2 panel ③. By including RD_CACHED pairs in this eviction as regular updates, we can answer future reads from C_1 rather than a slower lower level. However, if the key-value pairs are large, this can consume additional write bandwidth. This feature is desirable when the working-set is too large for C_0 (RAM) but small enough to fit in a fast-enough device residing at one of the next several levels (e.g., C_1 and C_2 on Flash SSD). Alternatively, this feature can be disabled for workloads where saving the cost of reading an average pair is not worth the additional insertion overhead, such as when we are not in a multi-tier scenario. All RD_CACHED values are omitted during a compaction whose merge includes the slots of the lowest level, and for which we are trying to relieve space pressure in the tier (i.e., a major compaction), and RD_CACHED values are omitted during a merging compaction if another pair with the same key can be emitted instead. Therefore, no additional space is used by inserting RD_CACHED pairs. Read caching across multiple tiers (i.e., outside of RAM) is a new topic, and discussion of it is not found in our survey of related work. We present some initial experiments related to multi-tier performance in Section 5.4. One subject of future work is more carefully exploring cache policies that work well for SAMTs or MT-SAMTs when caching outside of RAM where random access is costly, even on Flash SSD.

When scanning through trees (MTSAMTs), if read caching is enabled, the scanner inserts scanned values into the cache, and marks them as RD_CACHED. We have found that randomly reading larger tuples (>4096KB) can make effective use of a Flash SSD tier, however for smaller tuples (<64B) the time taken to warm the Flash SSD tier with reads is dominated by the slower random read throughput of the magnetic disk in the tier below. By allowing scans to cache read tuples, applications can exploit application-specific locality to pre-fetch pairs within the same or adjacent rows whose contents are likely to be later read.

Evictions of read-cached pairs can clear out a Flash SSD cache if those same pairs are not intelligently brought back into the higher tier they were evicted from after a cross-tier merging compaction. In Figure 2 panel ④, we see evicted pairs being copied back into the tier they were evicted from. This is called *reclamation*, and it allows SSTables, including read-cached pairs, that were evicted to magnetic disks (or other lower-tier devices) to be automatically copied back into the Flash SSD tier if they can fit.

Space Management and Reclamation.

We designed the MTSAMT so that more frequently accessed lists would be located at higher levels, or at C_i for the smallest i possible. After a merge, the resulting list may be smaller than the slot it was merged into because of resolved deletes and updates. If the resultant list can fit into one of the higher (and faster) slots from which it was merged (which are now clear), then it is moved upward, along with any other slots at the same level that can also fit. This process is called *reclamation* and requires that the total amount of pairs in bytes that can be reclaimed must fit into half the size of the level they were evicted from. By only reclaiming into half the level, a sufficient amount of space is reserved for merging compactions at that level to retain the same asymptotic insertion throughput. In the example in Figure 2, the result of the merging compaction in panel ③ is small enough to fit into the two (half of four) available slots in C_1, and specifically in this example requires only one slot. If multiple slots were required, the SSTable would be broken up into several smaller SSTables. This is possible because unlike Cassandra and HBase, GTSSL manages blocks in the underlying storage device directly, rather than treating SSTables as entire files on the file system, which allows for this kind of optimization. Reclamation across levels within the same tier is very inexpensive, as this requires merely *moving* SSTable blocks by adjusting pointers to the block, rather than *copying* them across devices. If these rules are obeyed, then partially filled slots are guaranteed to always move upward, eliminating the possibility that small lists of pairs remain stuck in lower and slower levels. As long as all lists are in the smallest levels in which they can fit, we retain the optimal asymptotic performance outlined in Section 3. By performing reclamation after every merge, we ensure this is always true, because reclamation effectively searches for the smallest level in which to fit a list produced by a merge.

We optimized our MTSAMT implementation for throughput. Our design considers space on storage with high latency and high read-write throughput characteristics (e.g., disk) to be cheaper than other hardware (e.g., RAM or Flash SSD). GTSSL can operate optimally until $1/2$ of total storage is consumed; after that, performance degrades gradually until the entire volume is full, save a small amount of reserve space (usually 5% of the storage device). (Such space-time trade-offs are common in storage systems [39], such as HBase [18], Cassandra [33], and even Flash SSD devices [29], as we elaborate further below.) At this point, only deletes and updates are accepted. These operations are processed by performing the equivalent of a major compaction: if there is not enough space to perform a merging compaction into the first free slot, then an in-place compaction of all levels in the MTSAMT is performed using the GTSSL's reserve space. As tuples are deleted, space is reclaimed, freeing it for more merging compactions that intersperse major compactions until $1/2$ of total storage is again free; at that point, only merging compactions need be performed, regaining the original optimal insertion throughput.

Chang et al. do not discuss out of space management in Big Table [10] except to say that a major compaction is performed in those situations; they also do not indicate the amount of overhead required to perform a major compaction. Cassandra simply requires that half of the device remain free at all times [33], arguing that disk storage is cheap. It is not uncommon for write-optimized systems, such as modern Flash SSD firmware, to require a large amount of storage to remain free for compaction. High performance Flash SSD devices build these space overheads (among other factors) into their total cost [21]. Even commodity Flash SSD performs far better when the partition actually uses no more than 60% of the total storage capacity [29]. To exploit decoupling, compaction-based systems such as GTSSL have some overhead to maintain optimal insertion throughput in the steady state, without this space their throughput degrades. Alternative systems such as Cassandra simply cease to operate when exceeding $1/2$ of the storage space. We believe that GTSSL's gradual degradation of performance beyond 50% space utilization is a sufficient compromise.

4.2 Committing and Stacked Caching

We showed how the MTSAMT extends the typical SAMT to operate efficiently in a multi-tier environment. In addition to efficient compaction, reclamation, and caching as discussed above, the efficiency of the memtable or C_0 (Section 2) as well as how efficiently it can be serialized to storage as an SSTable is also extremely important. As we evaluate in Section 5, the architecture of the transaction manager and caching infrastructure is the most important determiner of insertion throughput for small key-value pairs (< 1KB). GTSSL's architecture is mindful of cache efficiency, while supporting new transactional features (asynchronous commits) and complex multi-operation transactions.

Cache Stacking.

The transactional design of GTSSL is implemented in terms of GTSSL's concise cache-stacking feature. Like other TSSLs, GTSSL maintains a memtable to store key-value pairs. GTSSL uses a red-black tree with an LRU implementation, and DIRTY flags for each pair. An instance of this cache for caching pairs in a particular column family or tree is called a *scan cache*. Unlike other TSSL architectures, this scan cache can be stacked on top of another cache holding pairs from the same tree or MTSAMT. In this scenario the cache on top or the *upper cache* evicts into the *lower cache* when it becomes full by locking the lower cache and moving its pairs down into the lower cache. This feature simplifies much of GTSSL's transactional design, which we explore further in Section 4.3. In addition to the memtable cache, like other TSSLs, GTSSL requires a buffer cache, but as we discuss in the next paragraph, we do not need to fully implement a user-level buffer cache as traditional DBMSes typically do.

Buffer Caching.

We offload to the Linux kernel all caching of pages read from 128MB blocks, by MMAPing all storage in 1GB *slabs*. This simplifies our design as we avoid implementing a buffer cache. 64-bit machines' address spaces are sufficient and the cost of a random read I/O far exceeds the time spent on a TLB miss. Cassandra's default mode is to use MMAP within the Java API to also perform buffer caching. However, serial writes to a mapping incur reads as the underlying Linux kernel always reads the page into the cache, even on a write fault. This can cause overheads on serial writes of up to 40% in our experiments. Other TSSL architectures such as Cassandra do not address this issue. To avoid this problem, we PWRITE during merges, compactions, and serializations, and then we invalidate only the affected mapping using MSYNC with MS_INVALIDATE. As the original slots are in place during the merge, reads can continue while a merge takes place, until the original list must be deallocated. Once deallocated, reads can now be directed to the newly created slot. The result is that the only cache which must be manually maintained for write-ordering purposes is the journal cache, which is an append-only cache similar to that implemented by the POSIX FILE C API, which is light-weight, and simple.

All TSSLs that employ MMAP, even without additionally optimizing for serial writes like GTSSL, typically avoid read overheads incurred by a user-space buffer cache. On the other hand, traditional DBMSes can not use MMAP as provided by commodity OSes. This is because standard kernels (e.g., Linux) have no portable method of pinning dirty pages in the system page cache. Without this, or some other write-ordering mechanism, traditional DBMSes that require overwrites (e.g., due to using B+-trees), can violate write-ordering and break their recoverability. Therefore they are forced to rely on complex page cache implementations based on MALLOC [24,55,64] or use complex kernel-communication mechanisms [62–64]. TSSLs utilized in cloud based data stores such as Cassandra, HBase, or GTSSL never overwrite data during the serialization of a memtable to storage, and therefore need not pin buffer-cache pages, greatly simplifying these designs.

4.3 Transactional Support

Pavlo et. al [47] and Abouzeid et. al [1] use traditional parallel DBMS architectures for clustered structured data workloads, but these still rely on distributed transaction support. GTSSL's transactional architecture permits for atomic durable insertions, batched insertions for higher insertion-throughput, and larger transactions that can be either asynchronous or durable. This lets the same TSSL architecture to be used in a cluster operating under either consistency model.

We described MTSAMT's design and operation and its associated cache or memtable (C_0). As mentioned before, each MTSAMT corresponds to a tree or column family in a cloud storage center. GTSSL operates on multiple MTSAMTs to support row insertions across multiple column families, and more complex multi-operation transactions as required by stronger consistency models. Applications interact with the MTSAMTs through a transactional API: BEGIN, COMMIT_DURABLE, and COMMIT_ASYNC.

GTSSL's transaction manager (TM) manages all transactions for all threads. As shown in Figure 3, the TM maintains a stacked scan cache (Section 4.2) called the *staged cache* on top of each tree's C_0 (also a scan cache). When an application begins a transaction with BEGIN, the TM creates a handler for that transaction, and gives the application a reference to it. At any time, when a thread modifies a tree, a new scan cache is created if one does not already exist, and is stacked on top of that tree's staged cache. The new scan cache is placed in that transaction's handler. This new scan cache is called a *private cache*. In Figure 3 we see three handlers, each in use by three separate threads P_0 through P_2. Each thread has modified each of the three trees (MTSAMT$_0$ through MTSAMT$_2$).

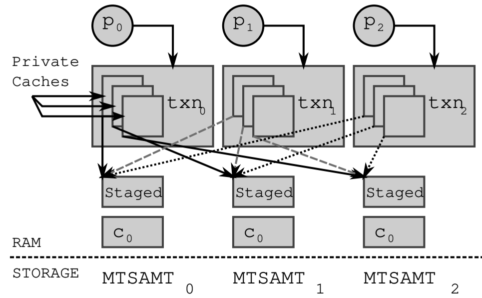

Figure 3: Three processes, $p_0 ... p_2$, each maintain an ongoing transaction that has modified all 3 MTSAMTs so far.

Transactions managed by GTSSL's TM are in one of three states: (1) they are uncommitted and still exist only with the handler's private caches; (2) they are committed either durably or asynchronously and are in either the staged cache or C_0 of the trees they effect; or (3) they are entirely written to disk. Transactions begin in state (1), move to state (2) when committed by a thread, and when GTSSL performs a snapshot of the system, they move to state (3) and are atomically written to storage as part of taking the snapshot.

Durable and asynchronous transactions can both be committed. We commit transactions durably by moving their transaction to state (2), and then *scheduling and waiting* for the system to perform a snapshot. While the system is writing a snapshot to storage, the staged cache is left unlocked so other threads can commit (similar to EXT3 [9]). A group commit of durable transactions occurs when multiple threads commit to the staged cache while the current snapshot is being written, and subsequently wait on the next snapshot together as a group before returning from COMMIT. Asynchronous transactions can safely commit to the staged cache and return immediately from COMMIT. After a snapshot the staged cache and the C_0 cache swap roles: the staged cache becomes the C_0 cache.

Next we discuss how we efficiently record snapshots in the journal, and how we eventually remove or garbage-collect snapshots by truncating the journal.

Snapshot, Truncate, and Recovery.

Unlike other BigTable based cluster TSSL architectures, GTSSL manages blocks directly, not using separate files for each SSTable. A block allocator manages each storage device. Every block allocator uses a bitmap to track which blocks are in use. The block size used is 128MB to prevent excessive fragmentation, but the OS page cache still uses 4KB pages for reads into the buffer cache.

Each tree (column family) maintains a cluster of offsets and metadata information that points to the location of all SSTable block offsets, secondary index block offsets, and Bloom filter block offsets. This cluster is called the *header*. When a snapshot is performed, all data referred to by all headers, including blocks containing SSTable information, and the bitmaps, are flushed to storage using MSYNC. Afterward, the append-only cache of the journal is flushed, recording all headers to the journal within a single atomic transaction. During recovery, the most recent set of headers are read back into RAM, and we recover the state of the system at the time that header was committed to the journal.

Traditional TSSLs implement a limited transaction feature-set that only allows for atomic insertion. Chang et. al [10] outline a basic architecture that implements this. Their architecture always appends insertions to the journal durably before adding them to the memtable. Cassandra and HBase implement this transactional architecture as well. By contrast Pavlo et. al [47] and Abouzeid et. al [1] make the case for distributed transactions in database clusters. GTSSL's architecture does not exclude distributed transactions, and is as fast as traditional TSSLs like Cassandra or HBase, or a factor of 2 faster when all three systems use asynchronous commits. One important feature of GTSSL is that high-insertion throughput workloads that can tolerate partial durability (e.g., snapshotting every 3–5 seconds) need not write the majority of data into the journal. Although Cassandra and HBase support this feature for many of their use cases as well, they only delay writing to the journal, rather than avoid it. GTSSL can avoid this write because if the C_0 cache evicts its memtable as an SSTable between snapshots, the cache is marked clean, and only the header need be serialized to the journal, avoiding double writing. This design improves GTSSL's performance over other TSSLs.

5. EVALUATION

We evaluated GTSSL, Cassandra, and HBase along with some traditional DBMSes for various workloads. However, we focus here on their four most important properties relevant to this work: (1) the flexibility and efficiency of their compaction methods, (2) the efficiency of their serialization and caching designs for smaller key-value pairs, (3) the multi-tier capabilities of GTSSL, and (4) the transactional performance of GTSSL and potentially other TSSLs with respect to traditional DBMSes for processing distributed transactions in a cluster. As laid out in Sections 3 and 4, we believe these are key areas where GTSSL improves on the performance of existing TSSL architectures.

5.1 Experimental Setup

Our evaluation ran on three identically configured machines running Linux CentOS 5.4. The client machines each have a quad-core Xeon CPU running at 2.4GHz with 8MB of cache, and 24GB of RAM; the machines were booted with kernel parameters to limit the amount of RAM used to either 4.84GB, or 0.95GB of RAM to test out-of-RAM performance, and we noted with each test how much RAM was used. Each machine has two 146.1GB 15KRPM SAS disks (one used as system disk), a 159.4GB Intel X-25M Flash SSD (2^{nd} generation), and two 249.5GB 10KRPM SATA disks. Our tests used pre-allocated and zeroed out files for all configurations. We cleared all caches on each machine before running any benchmark. To minimize internal Flash SSD firmware interference due to physical media degradation and caching, we focus on long-running throughput benchmarks in this evaluation. Therefore, we reset all Flash SSD wear-leveling tables prior to evaluation (using the TRIM command), and we also confined all tests utilizing Flash SSD to a 90GB partition of the 159.4GB disk, or 58% of the disk. To control for variance, all benchmarks are run over long periods of time (e.g., one half to two hours or longer) until throughput converges.

In tests involving HBase and Cassandra, we configured both systems to run directly on top of the file system. That is, HBase did not use HDFS, but ran directly on top of Ext3, the same file system used by all the other systems. This was to isolate performance to just the TSSL layer of HBase, and not penalize HBase for HDFS-related activities. This is the default behavior for Cassandra, but HBase had to be specially configured. Both systems were configured as efficient single-node systems according to their documentation to avoid network layer overheads [15,16]. We gave both systems 3GB of JVM heap, and we used the remaining 1.84GB as a file cache.

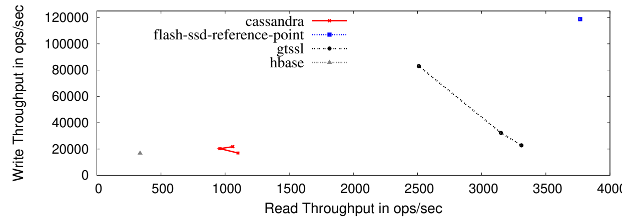

Figure 4: Cassandra has comparable insertion performance to GTSSL when both systems retain as much lookup throughput as possible. GTSSL reaches much further into the trade-off space. HBase is already optimally configured, and cannot further specialize for insertions.

We configured GTSSL to use upwards of 3GB for non-file cache information, including secondary indexes and Bloom filters for each slot in each tree, and the tuple cache (C_0) for each tree. GTSSL often used much less than 3GB, depending on the size of the pairs, but never more. We disabled compression for all systems because measurements of its effectiveness and for which data-sets are orthogonal to efficient TSSL operation. To prevent swapping heap contents when the file cache was under memory pressure due to MMAP faults, we set the SWAPPINESS parameter to 0 for all systems and monitored swap-ins and swap-outs to ensure no swapping took place. All tests, except the multi-tier storage tests in Section 5.4 were run on the Intel X25-M Flash SSD described above.

5.2 Read-Write Trade-off

We evaluated the performance of Cassandra, HBase, and GTSSL when inserting 1KB pairs into 4 trees, to exercise multi-tree transactions. 1KB is the pair size used by YCSB [13]. In this particular experiment, keys and values were generated with uniform distribution across the key space. Lookups are randomly uniform. For each system we varied its configuration to either favor reads or writes. HBase supported only one optimal configuration, so it was not varied. Cassandra and GTSSL can trade off lookup for insertion performance by increasing K (see Section 3). Our configuration named BALANCED sets $K = 4$, the default; configuration MEDIUM sets $K = 8$; configuration FAST sets $K = 80$. We measured insertion throughput and lookup separately to minimize interference, but both tests utilized 10 writers or readers.

Configuration.

In addition to the configuration parameters listed in Section 5.1, to utilize 4 trees in Cassandra and HBase, we configured 4 column families. We computed the on-disk footprint of one of Cassandra's pairs based on its SIZE routine in its TSSL sources (which we analyzed manually), and we reduced the size of the 1KB key accordingly so that each on-disk tuple would actually be 1KB large. We did this to eliminate any overhead from tracking column membership in each pair. We did the same for HBase, and used tuples with no column membership fields for GTSSL, while also accounting for the 4 byte size field used for variable length values. This minimized differences in performance across implementations due to different feature sets that require more or less metadata to be stored with the tuple on disk. Overall, we aimed to configure all systems as uniformly as possible, to isolate only the TSSL layer, and to configure Cassandra and HBase in the best possible light.

Results.

Figure 4 is a parametric function, where each point represents

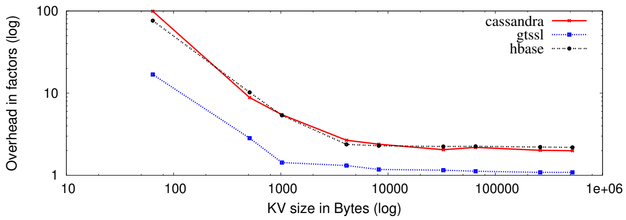

Figure 5: Neither Cassandra nor HBase improve beyond an overhead of 2.0× for large pairs, or 76.3× for small pairs.

a run, and the parameter varied is the system configuration. The x-axis measures that configuration's insertion (write) throughput, and the y-axis measures its random lookup (read) throughput. The maximum lookup throughput of each structure can not exceed the random read performance of the drive; similarly, the maximum insertion throughput can not exceed the serial write throughput of the drive. These two numbers are shown as one point at FLASH SSD REFERENCE POINT. For the BALANCED configuration, Cassandra and GTSSL have similar insertion throughputs of 16,970 ops/s and 22,780 ops/s, respectively. However, GTSSL has a 3× higher lookup throughput than Cassandra, and a 10× higher than HBase. GTSSL utilizes aggressive Bloom filtering to reduce the number of lookups to effectively 0 for any slot that does not contain the sought-after key. The random read throughput of the Flash SSD drive tested here is 3,768 reads/s, closely matching the performance of GTSSL. Cassandra uses 256KB blocks instead of 4KB blocks, but uses the metadata to read in only the page within the 256KB block containing the key. We observed that block read rates were at the maximum bandwidth of the disk, but Cassandra requires 3 IOs per lookup [20] when memory is limited, resulting in a lookup throughput that is only $1/3$ the random read throughput of the Flash SSD. Both HBase and Cassandra utilize Bloom filtering, but Bloom filtering is a new feature for HBase that was recently added. HBase caches these Bloom filters in an LRU cache. So although HBase can swap in different Bloom filters, for uniform or Zipfian lookup distributions, HBase has to page in Bloom filter data pages to perform lookups, causing a 10× slowdown compared to Cassandra and GTSSL. However if we perform a major compaction (which can take upwards of an hour) we notice that with 4KB blocks, HBase lookups can be as high as 910 lookups/s, but for the same block size before major compaction, lookup throughput is 200 lookups/s, lower than with the default 64KB blocksize. Performing major compactions with high frequency is not possible as it starves clients. For the more write-optimized configurations, GTSSL increased its available bandwidth for insertions considerably: for MIDDLE, GTSSL achieved 32,240 ops/s and 3,150 ops/s, whereas Cassandra reached only 20,306 ops/s and 960 ops/s, respectively. We expected a considerable increase in insertion throughput and sustained lookup performance for both Cassandra and GTSSL as they both use variants of the SAMT. However, Cassandra's performance could not be improved beyond 21,780 ops/s for the FAST configuration, whereas GTSSL achieved 83,050 ops/s. GTSSL's insertion throughput was higher thanks to its more efficient serialization of memtables to SSTables on storage. To focus on the exact cause of these performance differences, we configured all three systems (HBase, Cassandra, and GTSSL) to perform insertions but no compaction of any sort. We explore those results next.

Cassandra and HBase limiting factors.

To identify the key performance bottlenecks for a TSSL, we ran an insertion throughput test, where each system was configured to insert sizes of pairs varying from 64B to 512KB as rapidly as possible, using 10 parallel threads. In this particular experiment, keys were generated with uniform distribution across the key space. Lookups are randomly uniform. Cassandra, HBase, and GTSSL were all configured to commit asynchronously, but still maintain atomicity and consistency (the FAST configuration). Furthermore, Cassandra's compaction thresholds were both set to 80 (larger than the number of SSTables created by the test); HBase's compaction (and compaction time-outs) were simply disabled, leaving both systems to insert freely with *no compactions* during this test. The ideal throughput for this workload is the serial append bandwidth of the Flash SSD (110MB/s), divided by the size of the pair used in that run. Figure 5 shows these results. Each point represents an entire run of a system. The y-axis represents how many times slower a system is compared to the ideal, and the x-axis represents the size of the pair used for that run. All three systems have the same curve shape: a steep CPU-bound portion ranging from 64B to 1KB, and a shallower IO-bound portion from 1KB to 512KB.

For the IO-bound portion, HBase and Cassandra both perform at best 2.0× worse than the ideal, whereas GTSSL performs 1.1× worse than the ideal, so GTSSL is 2× faster than Cassandra and HBase in the IO-bound portion. Cassandra and HBase both log writes into their log on commit, even if the commit is asynchronous, whereas GTSSL behaves more like a file-system and avoids writing into the log if the memtable can be populated and flushed to disk before the next flush to the journal. This allows GTSSL to avoid the double-write to disk that Cassandra and HBase perform, a significant savings for IO-bound insertion-heavy workloads that can tolerate a 5-second asynchronous commit. For configurations not able to tolerate this delay, GTSSL still outperforms in CPU-bound workloads (additionally we perform durable commit experiments in Section 5.5).

For the CPU-bound portion, we see that GTSSL is a constant factor of 4× faster than both HBase and Cassandra, and additionally that HBase and Cassandra have very similar performance: the ratio of Cassandra's overhead to HBase's is always within a factor of 0.86 and 1.3 for all runs. When running Cassandra and its journal entirely in RAM, their insertion throughput of the 64B pair improved by only 50%, dropping from 99.2× to 66.1×, which is still 4× slower than GTSSL which was *not* running in RAM. The meager change in performance for running entirely in RAM further confirms that these workloads were CPU-bound for smaller pairs (< 1KB), and that the typically acceptable overheads introduced by the JVM—such as garbage collection, bounds-checking, copying of file caches across the JVM boundary—are not acceptable for these CPU-bound pair sizes. GTSSL's design minimizes memory copies by addressing directly through MMAP during serialization, and only copying once into its scan cache for lookups and updates. Objects are never copied but always moved between stacked caches. These results corroborate the reported inefficient use of CPU and RAM in HBase by others as well [4].

Future TSSL architectures must seriously consider CPU efficiency as the cost of a random write drops significantly for 1KB block sizes on Flash SSD.

5.3 Deduplication

To evaluate the performance of Cassandra, HBase, and GTSSL when processing a real-world workload, we built a deduplication index. We checksummed every 4KB block of every file in a research lab network of 82 clients of home directory files and directories, with a total of 1.6TB hashes for each chunk. Chunking [17,68] was done on a 4KB boundary, with no variable chunking. This gen-

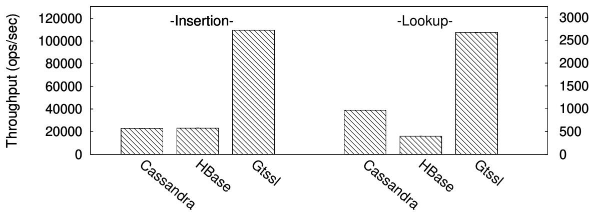

Figure 6: Deduplication insertion and lookup performance of the Cassandra and HBase TSSLs, and GTSSL.

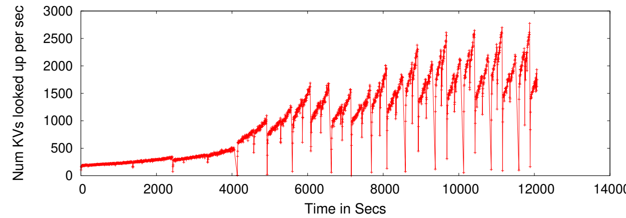

Figure 7: Multi-tier results: initially throughput is disk-bound, but as the hot-set is populated, it becomes Flash SSD-bound, and is periodically evicted and reclaimed.

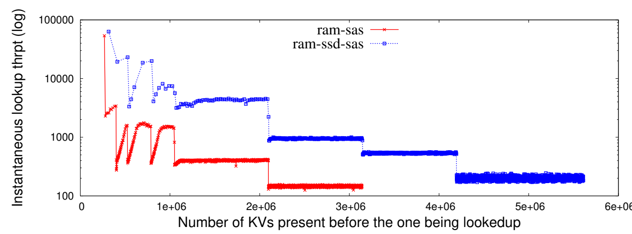

Figure 8: Multi-tier insertion caching allows for lookups of recently inserted data to happen rapidly, while still allowing for very large data-sets that can be cheaply stored mainly on disk.

erated over 1 billion hashes. In our analysis of the hashes, we found a typical Zipfian shape [13] where after the first 100 unique hashes, there was effectively a uniform distribution. We measured the time taken to insert these hashes with 10 parallel insertion threads for all systems. We then measured the time to perform random lookups on these hashes for a uniformly randomly selected subset.

Configuration.

We generated the deduplication hashes by chunking all files in our corpus into 4KB chunks, which were hashed with SHA256. We appended these 32B hashes to a file in the order they were chunked (depth-first traversal of the corpus file systems). To control the source of random read IO, we did not want to randomly select hashes to query from the corpus during lookup. Instead we wanted to serially traverse a pre-built list of lookups to perform. Therefore, to test lookups, we shuffled the hashes in advance into a separate lookup list. During insertion and lookup, we traversed the hashes serially, introducing little overhead during evaluation of each system.

Results.

As seen in Figure 6 we found that performance is analogous to the 64B case in Section 5.2, which used randomly generated 64B numbers instead of a stream of 32B hashes. Cassandra, HBase, and GTSSL were able to perform 22,920 ops/s, 23,140 ops/s, and 109,511 ops/s, respectively. For lookup performance they scored 967 ops/s, 398 ops/s, and 2,673 ops/s, respectively. As we have seen earlier, the performance gap between Cassandra and HBase compared to GTSSL are due to CPU and I/O inefficiency, as the workload is comparable to a small-pair workload, as discussed above. Real-world workloads can often have pairs of 1KB or smaller in size, such as this deduplication workload. An efficient TSSL can provide up to 5× performance improvement without any changes to other layers in the cluster architecture.

5.4 Multi-Tier Storage

We modified the SAMT compaction method so that multiple tiers in a multi-tier storage hierarchy would be naturally used for faster insertion throughputs and better caching behaviors. Here we explore the effectiveness of caching a working set that is too large to fit in RAM, but small enough to fit in our Flash SSD. We analyze two caching policies, namely: (1) LRU caching in a Flash SSD with a hot-set, and (2) Recent-insert caching.

Configuration.

As mentioned above, in the previous tests we used only the Flash SSD. In this test we also use the SAS disk. We configured GTSSL with the first tier as RAM, the second on Flash SSD, and the third on the SAS disk. The Flash SSD tier holds two levels, each with a maximum of 4 SSTables (slots): the maximum SSTable size on the first level is 256MB, and on the second level is 1GB. The SAS tier holds one level, with a maximum of 4 SSTables, each no larger than 4GB. For LRU caching, the size of the hot-set is 1GB, the size of available cache is 256MB. The size of the pairs was 4KB.

Results.

LRU-Caching: As shown in Figure 7, initial lookup throughput was 243 lookups/s, which corresponds to the random read throughput of the disk, 251 reads/s. Pairs are read into the scan cache (C_0), and once 256MB have been read, as described in Section 4.1, data in C_0 is flushed into the Flash SSD to facilitate multi-tier read caching. This corresponds to the 20 sudden drops in lookup throughput. Once the entire 1GB of hot-set has been evicted into the Flash SSD tier, subsequent reads, even from the Flash SSD, are re-inserted. These reads cause the contents of the Flash SSD to flush into the SAS tier, where they are dropped while merging as the SAS tier is the lowest level. However, as the hot-set is small enough to fit into a single slot, it is reclaimed back into the Flash SSD tier via a copy. The mean lookup throughput is 1,815 lookups/s, an 7.4× speedup over the disk read throughput, and 48% the Flash SSD random read throughput. The sudden drops in lookup throughput are due to evictions, now being caused by reads which actually result in writes. Latency spikes are a common problem with compaction based TSSLs. HBase and Cassandra use load-balancing at higher layers and schedule daily compactions to reduce their effect. For these tests their effects on performance were minimized through configuration, and these techniques occur at higher layers, or are easily adaptable to GTSSL.

Recent-insert Caching: For hotsets that are queried over a long period of time, read-caching for random reads from lower storage tiers can be beneficial, as we have shown above. Additionally, caching of recently inserted values in higher tiers is an automatic effect of the MTSAMT. We re-run the above experiment with a data-set of 16GB of randomly inserted 1KB keys with read caching disabled. After all values are inserted, we search for each pair from most recently inserted to least. Figure 8 shows our results. Insertion of the pairs was 33% faster for the RAM-SSD-SAS configuration as the more frequent merging compactions of the higher tiers took place on a Flash SSD device, and merges across tiers did not have to read and write to the same device at once. The pairs are inserted randomly, but the SSTables on storage are sorted, so we see a series of random reads within each SSTable. After insertions, the scan cache of 256MB was full, there were 3 256MB SSTables, and 3 1GB SSTables in the first tier, and 3 4GB SSTables in the second tier. For RAM-SSD-SAS only the 3 4GB SSTables were on SAS, for RAM-SAS they all were. Although each SSTable is guarded by an in-RAM bloom filter, false positives can cause lookups to check these tables regardless. Furthermore the ratio of buffer cache to SSTable size shrinks exponentially as the test performs lookups on lower levels. This causes the stair-step pattern seen in Figure 8. Initial spikes in lookup throughput occur as the buffer cache is extremely effective for the 256MB SSTables, but mixing cache hits with the faster cache-populating Flash SSD (14,118 lookups/s) provides higher lookup throughput than with the SAS (1,385 lookups/s). Total lookup throughput of the first 3,000,000 pairs, or the first 27% of the data-set was 2,338 lookups/s for RAM-SSD-SAS, and 316 lookups/s for RAM-SAS, a 7.4× performance improvement.

5.5 Cross-Tree Transactions

We designed GTSSL to efficiently process small transactions, as well as high-throughput insertion workloads. We evaluated GTSSL's transaction throughput when processing many small and large transactions. We ran two tests: (1) TXN-SIZE, and (2) GROUP-COMMIT. In TXN-SIZE, the number of executing threads is fixed at one, but each commit is asynchronous, so this thread need not wait for the commit to hit storage. Each run of the benchmark performs a transaction that inserts four pairs, each into a separate tree. Each run uses a different size for the four pairs, which is either 32B, 64B, 256B, or 4096B. In GROUP-COMMIT, each transaction inserts a random 1KB pair into 4 separate trees, and then commits durably. We ran the benchmark with 512 threads executing in parallel to test scalability on our 4-core machine.

Configuration.

We configured 3 systems for comparison in this test: GTSSL, MySQL (using InnoDB), and Berkeley DB (BDB). We configured each system identically to have 1GB of cache. We did not include HBase or Cassandra in these results as they do not implement asynchronous transactions. We configured BDB as favorably as possible through a process of reconfiguration and testing: 1GB of cache and 32MB of log buffer. We verified that BDB never swapped or thrashed during these tests. We configured BDB with a leaf-node size of 4096B. We configured InnoDB favorably with a 1GB of cache and 32MB of log buffer. We configured GTSSL with 1GB of cache (four 256MB caches).

Results.

GTSSL outperformed MySQL and BDB on the whole by a factor of about 6–8×. We inserted 1,220MB of transactions (9,994,240 transactions of four 32-byte insertions). For 32-byte insertions, overall insertion performance for BDB, MySQL, and GTSSL is 683, 732, and 8,203 commits/s, respectively. For 256 byte insertions it is 294, 375, and 3,140 commits/s, respectively. At 4KB insertions, MySQL does not permit 4K columns, and so we omit this result. However, GTSSL and BDB each have throughputs of 804 and 129, respectively. GTSSL is 6.23× faster than BDB. We found that the difference in performance was because synchronous appends are much faster on our Flash SSD drive than random writes, so MySQL and BDB begin converging on their B-Tree insertion throughput as they write-back their updates. GTSSL, on the other hand, avoids random writes entirely, and pays only merging overheads periodically due to merging compactions.

Despite the stark difference in throughput for the whole workload, we also found that BDB, GTSSL, and MySQL had equivalent insertion throughput. As transactions are submitted serially, the current transaction must wait for the disk to sync its write before the next transaction can proceed. On magnetic disks, we found that synchronous appends and random writes were both ideally 300 commits/s for direct updates to a file on Ext3. However, on Flash SSD, due to a sophisticated FTL, with *durable* transactions, our Intel X-25M Flash SSD was able to sustain 15,000 synchronous serial appends of 32 bytes/s. Consequently, MySQL, BDB, and GTSSL each attained *initial* insertion throughputs of 2,281, 5,474, and 9,908 transactions/s, respectively, when just updating their own journals. GTSSL is able to keep the total amount written per-commit small—as it must only flush the dirty pairs in its C_0 cache plus book-keeping data for the flush (111 bytes). This additional amount written per transaction gives direct synchronous append an advantage of 66% over GTSSL; however, as GTSSL logs only redo information, its records require no reads to be performed to log undo information, and its records are smaller. This means that as BDB and MySQL must routinely perform random IOs as they interact with a larger-than-RAM B+-tree, GTSSL need only perform mostly serial IOs, which is why GTSSL and other TSSL architectures are better suited for high insertion-throughput workloads.

When testing peak Flash SSD-bandwidth GROUP-COMMIT throughput, we found that GTSSL could perform 26,368 commits/s for transactions, updating 4 trees with 1KB values, at a bandwidth of 103MB/s. The high commit throughput was due to the Flash SSD being able to perform serial durable writes much more quickly than random durable writes.

Evaluation summary.

TSSL architectures have traditionally optimized for IO-bound workloads for pairs 1KB or larger on traditional magnetic disks. For 1KB pairs, GTSSL has a demonstrably more flexible compaction method. For the read-optimized configuration, GTSSL lookups are near optimal: 88% the maximum random-read throughput of the Flash SSD, yet our insertions are still 34% faster than Cassandra and 14% faster than HBase. For the write-optimized configuration, GTSSL achieves 76% of the maximum write throughput of the Flash SSD, yet our lookups are 2.3× and 7.2× faster than Cassandra and HBase, respectively. This performance difference was due to Cassandra and HBase being CPU-bound for pairs 1KB or smaller. When we varied the pair size, we discovered that for smaller pairs, even when performing no compaction and no operations other than flushing pairs to storage, all TSSLs became CPU-bound, but GTSSL was still 5× faster than the others.

For larger pairs, all TSSLs eventually became IO-bound. GTSSL achieved 91% of the maximum serial write throughput of the Flash SSD. Cassandra and HBase achieved only 50% of the maximum Flash SSD throughput, due to double-writing insertions even when transactions were asynchronous. Cassandra's and HBase's designs were geared for traditional hard-disks whose latencies are much slower than RAM; but as modern Flash SSD's get faster, the bottleneck in such designs shifts from I/O to CPU. By contrast, GTSSL's design explicitly incorporates Flash SSD into a multi-tier hierarchy. When we insert a Flash SSD into a traditional RAM+HDD storage stack, GTSSL's insertion throughput increased by 33%, and our lookup throughput increased by 7.4×. This allows the bulk of colder data to reside on inexpensive media, while most hot data automatically benefits from faster devices.

Based on our current experiments, we predict that caching at higher layers will not significantly alter our current performance results because the task of sorting and compaction still lies on the shoulders of the TSSL layer. Constraints on throughput for existing TSSL designs are not due to latency in servicing read or write requests into the cache, but due to the overall inefficiency of performing serialization of the cache, compaction, and lack of careful integration with the operating system caches. Further exploration of this question is a subject of future work.

Lastly, supporting distributed transactions in clusters does not necessarily require a different TSSL layer as suggested by related research [47] (i.e., a read-optimized approach). GTSSL's transactions are light-weight yet versatile, and achieve 10.7× and 8.3× faster insertion throughputs than BDB and MySQL InnoDB, respectively.

6. RELATED WORK

We discuss cluster evaluation (1), multi-tier and hierarchical systems (2–4), followed by alternative datastructures for managing trees or column families in a TSSL architecture (5–7).

(1) Cluster Evaluation.

Super computing researchers recognize the need to alter out-of-the-box cluster systems, but there is little research on the performance of individual layers in these cluster systems, and how they interact with the underlying hardware. Pavlo et al. have measured the performance of a Hadoop HBase system against widely used parallel DBMSes [47]. Cooper et al. have compared Hadoop HBase to Cassandra [33] and a cluster of MySQL servers (similar to HadoopDB and Perlman and Burns' Turbulence Database Cluster). The authors of HadoopDB include a similar whole-system evaluation in their paper [1]. We evaluate the performance bottlenecks of a single node's storage interaction, and provide a prototype architecture that alleviates those bottlenecks.

Some supercomputing researchers develop custom cluster designs for a particular application that avoids logical layer overheads (such as SQL) when necessary [14, 37, 41]. These researchers still want to understand the performance characteristics of the components they alter or replace, and *especially* if those components are storage-performance bottlenecks. In addition to its multi-tier contributions, this paper outlines many of the critical aspects of architecting an efficient TSSL layer for these researchers.

(2) Multi-tier storage.

Flash SSD is becoming popular [25]. Solaris ZFS can use intermediate SSDs to improve performance [34]. ZFS uses an Flash SSD as a DBMS log to speed transaction performance, or as a cache to decrease read latency. But this provides only temporary relief: when the DBMS ultimately writes to its on-disk tree, it bottlenecks on B-tree throughput. ZFS has no explicit support for very large indexes or trees, nor does it utilize its three-tier architecture to improve indexing performance. GTSSL, conversely, uses a compaction method whose performance is bound by disk bandwidth, and can sustain high-throughput insertions across Flash SSD flushes to lower tiers with lower latencies. Others used Flash SSD's to replace swap devices. FlashVM uses an in-RAM log-structured index for large pages [53]. FASS implements this in Linux [30]. Payer [48] describes using a Flash SSD hybrid disk. Conquest [67] uses persistent RAM to hold all small file system structures; larger ones go to disk. These systems use key-value pairs with small keys that can fit entirely in RAM. GTSSL is more general and can store large amounts of highly granular structured data on any combination of RAM and storage devices.

(3) Hierarchical Storage Management.

HSM systems provide disk backup and save disk space by moving old files to slower disks or tapes. Migrated files are accessible via search software or by replacing migrated files with links to their new location [28, 46]. HSMs use multilevel storage hierarchies to reduce overall costs, but pay a large performance penalty to retrieve migrated files. GTSSL, however, was designed for always-online access as it must operate as a TSSL within a cluster, and focuses on maximum performance across all storage tiers.

(4) Multi-level caching.

These systems address out-of-sync multiple RAM caches that are often of the same speed and are all volatile: L2 vs. RAM [19], database cache vs. file system page cache [22], or located on different networked machines [36, 59]. These are not easily applicable to general-purpose multi-tier structure data storage due to large performance disparities among the storage devices at the top and bottom of the hierarchy.

(5) Write-optimized trees.

The *COLA* maintains $\mathbf{O}(\log(N))$ cache lines for N key-value pairs. The amortized asymptotic cost of insertion, deletion, or updates into a COLA is $\mathbf{O}(\log(N)/B)$ for N inserted elements [7]. With fractional cascading, queries require $\mathbf{O}(\log(N))$ random reads [11]. GTSSL's SAMT has identical asymptotic insertion, deletion, and update performance; however, lookup with SAMT is $\mathbf{O}\left(\log^2(N)\right)$. In practice GTSSL's secondary indexes easily fit in RAM though, and so lookup is actually equivalent for trees several TBs large. Furthermore, as we show in our evaluation, GTSSL's Bloom filters permit 10–30× faster lookups for datasets on Flash SSD than what the COLA (used by HBase) can afford. *Log-Structured Merge* (LSM) trees [44] use an in-RAM cache and two on-disk B-Trees that are R and R^2 times larger than cache, where $\frac{1}{R} + R + R^2$ is the size of the tree. LSM tree insertions are asymptotically faster than B-Trees: $\mathbf{O}\left(\frac{\sqrt{N}\log N}{B}\right)$ [56] compared to $\mathbf{O}\left(\log_{B+1} N\right)$, but asymptotically slower than GTSSL's SAMT. LSM tree query times are more comparable to B-Tree's times. Rose is a variant of an LSM tree that compresses columns to improve disk write throughput [56]. Anvil [38] is a library of storage components for assembling custom 2-tier systems and focuses on development time and modularity. Anvil describes a 2-COLA based structure and compares performance with traditional DBMSes in TPC-C performance. GTSSL's uses the multi-tier MTSAMT structure, and is designed for high-throughput insertion and lookups as a component of a cluster node. We evaluate against existing industry standard write-optimized systems and not random-write-bound MySQL InnoDB. Data Domain's deduplicating SegStore uses Bloom filters [8] to avoid lookups to its on-disk hash table, boosting throughput to 12,000 inserts/s. GTSSL solves a different problem: the base insertion throughput to an on-disk structured data store (e.g., Data Domain's Segment Index, for which insertion is a bottleneck). GTSSL is complimentary to, and could significantly improve the performance of similar deduplication technology.

(6) Log-structured data storage.

Log-structured file systems [51] append dirtied blocks to a growing log that must be compacted when full. Graefe's log-structured B-trees [23] and FlashDB [42] operate similarly to WAFL [26] by rippling up changes to leaf pointers. Goetz uses fence-keys to avoid expensive rippling, and uses tree-walks during scans to eliminate leaf pointers. FAWN [3] is a distributed 2-tier key-value store designed for energy savings. It uses a secondary index in RAM and hash tables in Flash SSD. FAWN claims that compression (orthogonal to this work) allows large indexes to fit into 2GB of RAM. By contrast, GTSSL has been tested with 1–2TB size indexes on a single node. Log-structured systems assume that the entire index fits in RAM, and must read in out-of-RAM portions before updating them. This assumption breaks down for smaller (64B) pairs where the size of the index is fairly large; then, compaction methods employed by modern TSSLs become vital.

(7) Flash SSD-optimized trees.

Flash SSD has high throughput writes and low latency reads, ideal for write-optimized structured data storage. FD-Trees's authors admit similarity to LSM trees [35]. Their writes are worse than an LSM-tree for 8GB workloads; their read performance, however, matches a B-tree. GTSSL's insertions are asymptotically faster than LSM trees. LA-Tree [2] is another Flash SSD-optimized tree, similar to a Buffer Tree [6]. LA-Trees and FlashDB can adaptively reorganize the tree to improve read performance. Buffer Trees have asymptotic bound equal to COLA. However, it is not clear or discussed how to efficiently extend Buffer Trees, LA-Trees, or FD-Trees for multiple storage tiers or transactions as GTSSL does.

7. CONCLUSIONS

We introduced GTSSL, an efficient tablet server storage architecture that is capable of exploiting Flash SSD and other storage devices using a novel multi-tier compaction algorithm. Our multi-tier extensions have 33% faster insertions and a 7.4× faster lookup throughput than traditional RAM+HDD tiers—while storing 75% of the data (i.e., colder data) on cheaper magnetic disks. By minimizing the number of IOs per-lookup, and aggressively using of Bloom filters, GTSSL achieves 2.3× and 7.2× faster lookups than Cassandra and HBase, respectively—while maintaining 3.8× faster insertions for standard sized inputs. We demonstrated GTSSL's much wider range of support for either lookup-heavy or insert-heavy workloads, compared to Cassandra and HBase. We have shown how the existing TSSL layer can be extended to support more versatile transactions capable of performing multiple reads and writes in full isolation without compromising performance. GTSSL achieved 10.7× and 8.3× faster insertion throughputs than BDB and MySQL's InnoDB, respectively.

Our analysis shows that the SAMT structure is better suited for newer storage technologies such as Flash SSD that naturally have a smaller DAM block size, however we have shown in Section 5.4 that significant performance gains can be made while still keeping the majority of data stored on a magnetic disk by extending the SAMT to support multi-tier workloads. These extensions are outlined in Section 4.1. Furthermore, our performance evaluation of existing TSSL architectures show that, faced with increasingly faster random I/O from Flash SSD's, CPU and memory efficiency are paramount for increasingly more complex and granular data.

From our evaluation we learned several key lessons: (1) multi-tier support is critical for better performance to cost ratio in TSSLs, (2) more complex transaction support requires a more traditional DBMS/file system cache architecture tuned for fast insertion and exploiting the log-structured nature of the TSSL compaction algorithms, and (3) efficiency of key-value pairs below 1KB in size depends on the cache implementation and general CPU-boundedness of the design more than the specific compaction algorithms used.

Most importantly, integrating modern storage devices into a TSSL requires a more general approach to storage than currently available, and one that operates in a generic fashion across multiple tiers. GTSSL offers just that.

Future Work.

We will explore measuring the performance effect of multi-tier TSSLs on a larger cluster to demonstrate design transferability and improved performance. Currently our Bloom filters lie completely in RAM, however we are actively exploring ways of efficiently inserting and searching within Bloom filter-like structures across multiple tiers.

We are planning to release the GTSSL system, sources, and all benchmarks and data-sets later this year.

8. REFERENCES

[1] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2:922–933, August 2009.

[2] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: an optimized index structure for flash devices. *Proc. VLDB Endow.*, 2(1):361–372, 2009.

[3] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '2009)*, pages 1–14. ACM SIGOPS, October 2009.

[4] Eric Anderson and Joseph Tucek. Efficiency matters! *SIGOPS Oper. Syst. Rev.*, 44:40–45, March 2010.

[5] The Apache Foundation. Hadoop, January 2010. `http://hadoop.apache.org`.

[6] L. Arge. The buffer tree: A new technique for optimal i/o-algorithms (extended abstract). In *University of Aarhus*, pages 334–345. Springer-Verlag, 1995.

[7] 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.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[9] M. Cao, T. Y. TsâĂŽo, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas. State of the Art: Where we are with the Ext3 filesystem. In *Proceedings of the Linux Symposium*, Ottawa, ON, Canada, July 2005.

[10] 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.

[11] B. Chazelle and L. J. Guibas. Fractional cascading: A data structuring technique with geometric applications. In *Proceedings of the 12th Colloquium on Automata, Languages and Programming*, pages 90–100, London, UK, 1985. Springer-Verlag.

[12] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[14] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. The cost of doing science on the cloud: the montage example. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 50:1–50:12, Piscataway, NJ, USA, 2008. IEEE Press.

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

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

[17] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, San Jose, California, USA, 2011.

[18] Bruno Dumon. Visualizing hbase flushes and compaction. http://outerthought.org/blog/465-ot.html, February 2011.

[19] E. D. Demaine. Cache-Oblivious Algorithms and Data Structures, 1999.

[20] J. Ellis. Re: Worst case iops to read a row, April 2010. `http://cassandra-user-incubator-apache-org.3065146.n2.nabble.com/Worst-case-iops-to-read-a-row-td4874216.html`.

[21] FusionIO. IODrive octal datasheet. `http://www.fusionio.com/data-sheets/iodrive-octal/`.

[22] B. S. Gill. On Multi-level Exclusive Caching: Offline Optimality and Why promotions are better than demotions. In *FAST '08: Proccedings of the 6th conference on File and storage technologies*, Berkeley, CA, USA, 2008. USENIX Association.

[23] G. Graefe. Write-optimized b-trees. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 672–683. VLDB Endowment, 2004.

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

[25] F. Hady. Integrating NAND Flash into the Storage Hierarchy ... Research or Product Design?, 2009. `http://csl.cse.psu.edu/wish2009_invitetalk1.html`.

[26] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, pages 235–245, San Francisco, CA, January 1994. USENIX Association.

[27] Hypertable. Hypertable. `http://www.hypertable.org`, 2011.

[28] IBM. Hierarchical Storage Management. `www.ibm.com/servers/eserver/iseries/hsmcomp/`, 2004.

[29] 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`.

[30] D. Jung, J. Kim, S. Park, J. Kang, and J. Lee. FASS: A Flash-Aware Swap System. In *Proc. of International Workshop on Software Support for Portable Storage (IWSSPS)*, 2005.

[31] R. Kothiyal, V. Tarasov, P. Sehgal, and E. Zadok. Energy and Performance Evaluation of Lossless File Data Compression on Server Systems. In *Proceedings of the 2nd Israeli Experimental Systems Conference (ACM SYSTOR '09)*, Haifa, Israel, May 2009. ACM.

[32] Hans-Peter Kriegel, Peer Kröger, Christiaan Hendrikus Van Der Meijden, Henriette Obermaier, Joris Peters, and Matthias Renz. Towards archaeo-informatics: scientific data management for archaeobiology. In *Proceedings of the 22nd international conference on Scientific and statistical database management*, SSDBM'10, pages 169–177, Berlin, Heidelberg, 2010. Springer-Verlag.

[33] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 5–5, New York, NY, USA, 2009. ACM.

[34] A. Leventhal. Flash storage memory. *Communications of the ACM*, 51(7):47–51, 2008.

[35] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1303–1306, Washington, DC, USA, 2009. IEEE Computer Society.

[36] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in thor. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 318–329, New York, NY, USA, 1996. ACM.

[37] Wei Lu, Jared Jackson, and Roger Barga. Azureblast: a case study of developing science applications on the cloud. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 413–420, New York, NY, USA, 2010. ACM.

[38] 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.

[39] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[40] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[41] Beomseok Nam, Henrique Andrade, and Alan Sussman. Multiple range query optimization with distributed cache indexing. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[42] S. Nath and A. Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. Technical Report MSR-TR-2006-168, Microsoft Research, November 2006.

[43] National Radio Astronomy Observatory. Innovations in data-intensive astronomy. http://www.nrao.edu/meetings/bigdata/, April 2011.

[44] 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.

[45] Oracle. Database administrator's reference. `http://download.oracle.com/docs/cd/B19306_01/server.102/b15658/tuning.htm`, March 2009.

[46] R. Orlandic. Effective management of hierarchical storage using two levels of data clustering. *Mass Storage Systems, IEEE Symposium on*, 0:270, 2003.

[47] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[48] H. Payer, M. A. A. Sanvido, Z. Z. Bandic, and C. M. Kirsch. Combo Drive: Optimizing Cost and Performance in a Heterogeneous Storage Device. In *Proc. Workshop on Integrating Solid-state Memory into the Storage Hierarchy (WISH)*, 2009. `http://csl.cse.psu.edu/wish2009_papers/Payer.pdf`.

[49] Eric Perlman, Randal Burns, Yi Li, and Charles Meneveau. Data exploration of turbulence simulations using a database cluster. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 23:1–23:11, New York, NY, USA, 2007. ACM.

[50] PostgreSQL Global Development Team. PostgreSQL. `http://www.postgresql.org`, 2011.

[51] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 1–15, Asilomar Conference Center, Pacific Grove, CA, October 1991. Association for Computing Machinery SIGOPS.

[52] J. Rydningcom and M. Shirer. Worldwide hard disk drive 2010-2014 forecast: Sowing the seeds of change for enterprise applications. IDC Study 222797, `www.idc.com`, May 2010.

[53] M. Saxena and M. M. Swift. Flashvm: Revisiting the virtual memory hierarchy, 2009.

[54] Michael C. Schatz. Cloud computing and the dna data race. *Nature Biotechnology*, 28:691–693, 2010.

[55] R. Sears and E. Brewer. Stasis: Flexible Transactional Storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, Seattle, WA, November 2006. ACM SIGOPS.

[56] R. Sears, M. Callaghan, and E. Brewer. Rose: Compressed, log-structured replication. In *Proceedings of the VLDB Endowment*, volume 1, Auckland, New Zealand, 2008.

[57] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating Performance and Energy in File System Server Workloads extensions. In *Proceedings of the Eighth USENIX Conference on File and Storage Technologies (FAST '10)*, pages 253–266, San Jose, CA, February 2010. USENIX Association.

[58] P. Sehgal, V. Tarasov, and E. Zadok. Optimizing Energy and Performance for Server-Class File System Workloads. *ACM Transactions on Storage (TOS)*, 6(3), September 2010.

[59] L. Shrira, B. Liskov, M. Castro, and A. Adya. How to scale transactional storage systems. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 121–127, New York, NY, USA, 1996. ACM.

[60] Sleepycat Software, Inc. *Berkeley DB Reference Guide*, 4.3.27

edition, December 2004. www.oracle.com/technology/documentation/berkeley-db/db/api_c/frame.html.
[61] Keith A. Smith. File system benchmarks. http://www.eecs.harvard.edu/~keith/usenix96/, 1996.
[62] 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.
[63] R. Spillane, R. Sears, C. Yalamanchili, S. Gaikwad, M. Chinni, and E. Zadok. Story Book: An Efficient Extensible Provenance Framework. In *Proceedings of the first USENIX workshop on the Theory and Practice of Provenance (TAPP '09)*, San Francisco, CA, February 2009. USENIX Association.
[64] 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.
[65] A. Traeger, N. Joukov, C. P. Wright, and E. Zadok. A Nine Year Study of File System and Storage Benchmarking. *ACM Transactions on Storage (TOS)*, 4(2):25–80, May 2008.
[66] Vertica. The Vertica Analytic Database. http://vertica.com, March 2010.
[67] A. Wang, G. Kuenning, P. Reiher, and G. Popek. The conquest file system: Better performance through a disk/persistent-ram hybrid design. *Trans. Storage*, 2(3):309–348, 2006.
[68] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, San Jose, California, USA, 2008.