bLSM:* A General Purpose Log Structured Merge Tree

Russell Sears
Yahoo! Research
Santa Clara, CA

Raghu Ramakrishnan
Yahoo! Research
Santa Clara, CA

ABSTRACT

Data management workloads are increasingly write-intensive and subject to strict latency SLAs. This presents a dilemma: Update in place systems have unmatched latency but poor write throughput. In contrast, existing log structured techniques improve write throughput but sacrifice read performance and exhibit unacceptable latency spikes.

We begin by presenting a new performance metric: *read fanout*, and argue that, with *read* and *write amplification*, it better characterizes real-world indexes than approaches such as asymptotic analysis and price/performance.

We then present *bLSM*, a Log Structured Merge (LSM) tree with the advantages of B-Trees and log structured approaches: (1) Unlike existing log structured trees, bLSM has near-optimal read and scan performance, and (2) its new "spring and gear" merge scheduler bounds write latency without impacting throughput or allowing merges to block writes for extended periods of time. It does this by ensuring merges at each level of the tree make steady progress without resorting to techniques that degrade read performance.

We use Bloom filters to improve index performance, and find a number of subtleties arise. First, we ensure reads can stop after finding one version of a record. Otherwise, frequently written items would incur multiple B-Tree lookups. Second, many applications check for existing values at insert. Avoiding the seek performed by the check is crucial.

Categories and Subject Descriptors

H.3.2 [**Information Storage**]: File organization

General Terms

Algorithms, Performance

Keywords

Log Structured Merge tree, merge scheduling, read fanout, read amplification, write amplification

*Pronounced "Blossom"

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

1. INTRODUCTION

Modern web services rely upon two types of storage for small objects. The first, update-in-place, optimizes for random reads and worst case write latencies. Such stores are used by interactive, user facing portions of applications. The second type is used for analytical workloads and emphasizes write throughput and sequential reads over latency or random access. This forces applications to be broken into "fast-path" processing, and asynchronous analytical tasks.

This impacts end-users (e.g., it may take hours for machine learning models to react to users' behavior), and forces operators to manage redundant storage infrastructures.

Such limitations are increasingly unacceptable. Cloud computing, mobile devices and social networking write data at unprecedented rates, and demand that updates be synchronously exposed to devices, users and other services. Unlike traditional write-heavy workloads, these applications have stringent latency SLAs.

These trends have two immediate implications at Yahoo!. First, in 2010, typical low latency workloads were 80-90% reads. Today the ratio is approaching 50%, and the shift from reads to writes is expected to accelerate in 2012. These trends are driven by applications that ingest event logs (such as user clicks and mobile device sensor readings), and later mine the data by issuing long scans, or targeted point queries.

Second, the need for performant index probes in write optimized systems is increasing. bLSM is designed to be used as backing storage for PNUTS, our geographically-distributed key-value storage system [10], and Walnut, our next-generation elastic cloud storage system [9].

Historically, limitations of log structured indexes have presented a trade off. Update-in-place storage provided superior read performance and predictability for writes; log-structured trees traded this for improved write throughput. This is reflected in the infrastructures of companies such as Yahoo!, Facebook [6] and Google, each of which employs InnoDB and either HBase [1] or BigTable [8] in production.

This paper argues that, with appropriate tuning and our merge scheduler improvements, LSM-trees are ready to supplant B-Trees in essentially all interesting application scenarios. The two workloads we describe above, interactive and analytical, are prime examples: they cover most applications, and, as importantly, are the workloads that most-frequently push the performance envelope of existing systems. Inevitably, switching from B-Trees to LSM-Trees entails a number of tradeoffs, and B-Trees still outperform log structured approaches in a number of corner cases. These are summarized in Table 1.

Operation	Our bLSM-Tree		B-Tree		LevelDB
Point lookup (seeks)	**1**	§3.1.1	**1**	§2.2	$O(log(n))$
Read modify write (seeks)	**1**	§2.3	2	§2.2	$O(log(n))$
Apply delta to record (seeks)	**0**	§2.3	2	§2.2	**0**
Insert or overwrite (seeks)	**0**	§3.1.2	2	§2.2	**0**
Short ($\leq$ 1 page) scans (seeks)	2	§3.3	**1**	§2.2	$O(log(n))$
Long (N page) scans (seeks)	**2**	§3.3	Up to N	§2.2	$O(log(n))$
Uniform random insert latency	**Bounded**	§4.1-§4.3	**Bounded**	§2.2	Unbounded
Worst-case insert latency	Unbounded	See §4.2.2 for a fix	**Bounded**	§2.2	Unbounded

Table 1: Summary of results. bLSM-Trees outperform B-Trees and LSM-Tees in most interesting scenarios.

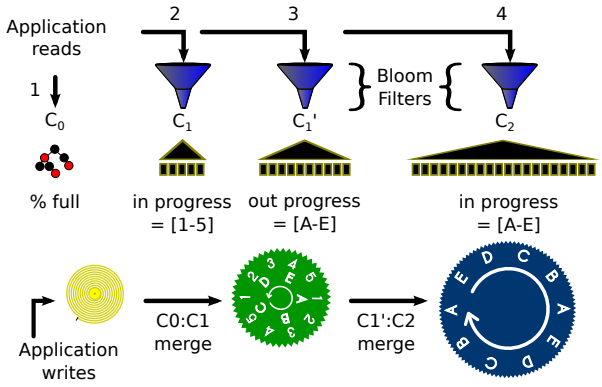

Figure 1: bLSM-Tree architecture

Concretely, we target "workhorse" data management systems that are provisioned to minimize the price/performance of storing and processing large sets of small records. Serving applications written atop such systems are dominated by point queries, updates and occasional scans, while analytical processing consists of bulk writes and scans. Unlike B-Trees and existing log structured systems, our approach is appropriate for both classes of applications.

Section 2 provides an overview of log structured index variants. We explain why partitioned, three level trees with Bloom filters (Figure 1) are particularly compelling.

Section 3 discusses subtleties that arise with the base approach, and presents algorithmic tweaks that improve read, scan and insert performance.

Section 4 describes our design in detail and presents the missing piece: with careful scheduling of background merge tasks we are able to automatically bound write latencies without sacrificing throughput.

Beyond recommending and justifying a combination of LSM-Tree related optimizations, the primary contribution of this paper is a new class of merge schedulers called *level schedulers*. We distinguish level schedulers from existing *partition schedulers* and present a level scheduler we call the *spring and gear scheduler*.

In Section 5 we confirm our LSM-Tree design matches or outperforms B-Trees on a range of workloads. We compare against LevelDB, a state-of-the-art LSM-Tree variant that has been highly optimized for desktop workloads and makes different tradeoffs than our approach. It is a multi-level tree that does not make use of Bloom filters and uses a partition scheduler to schedule merges. These differences allow us to isolate and experimentally validate the effects of each of the major decisions that went into our design.

2. BACKGROUND

Storage systems can be characterized in terms of whether they use random I/O to update data, or write sequentially and then use additional sequential I/O to asynchronously maintain the resulting structure. Over time, the ratio of the cost of hard disk random I/O to sequential I/O has increased, decreasing the relative cost of the additional sequential I/O and widening the range of workloads that can benefit from log-structured writes.

As object sizes increase, update-in-place techniques begin to outperform log structured techniques. Increasing the relative cost of random I/O increases the object size that determines the "cross over" point where update-in-place techniques outperform log structured ones. These trends make log structured techniques more attractive over time.

Our discussion focuses on three classes of disk layouts: update-in-place B-Trees, ordered log structured stores, and unordered log structured stores. B-Tree read performance is essentially optimal. For most applications they perform at most one seek per read. Unfragmented B-Trees perform one seek per scan. However, they use random writes to achieve these properties. The goal of our work is to improve upon B-Tree writes without sacrificing read or scan performance.

Ordered log structured indexes buffer updates in RAM, sort them, and then write sorted runs to disk. Over time, the runs are merged, bounding overheads incurred by reads and scans. The cost of the merges depends on the indexed data's size and the amount of RAM used to buffer writes.

Unordered log structured indexes write data to disk immediately, eliminating the need for a separate log. The cost of compacting these stores is a function of the amount of free space reserved on the underlying device, and is independent of the amount of memory used as cache. Unordered stores typically have higher sustained write throughput than ordered stores (order of magnitude differences are not uncommon [22, 27, 28, 32]). These benefits come at a price: unordered stores do not provide efficient scan operations. Scans are required by a wide range of applications (and exported by PNUTS and Walnut), and are essential for efficient relational query processing. Since we target such use cases, we are unable to make use of unordered techniques. However, such techniques complement ours, and a number of implementations are available (Section 6). We now turn to a discussion of the tradeoffs made by ordered approaches.

2.1 Terminology

A common theme of this work is the tradeoff between asymptotic and constant factor performance improvements. We use the following concepts to reason about such tradeoffs.

Read amplification [7] and *write amplification* [22] charac-

terize the cost of reads and writes versus optimal schemes. We measure read amplification in terms of seeks, since at least one random read is required to access an uncached piece of data, and the seek cost generally dwarfs the transfer cost. In contrast, writes can be performed using sequential I/O, so we express write amplification in terms of bandwidth. By convention, our computations assume worst-case access patterns and optimal caching policies.

$$read\ amplification = worst\ case\ seeks\ per\ index\ probe$$

$$write\ amplification = \frac{total\ seq.\ I/O\ for\ object}{object\ size}$$

Write amplification includes both the synchronous cost of the write, and the cost of deferred merges or compactions.

Given a desired read amplification, we can compute the *read fanout* (our term) of an index. The read fanout is the ratio of the data size to the amount of RAM used by the index. To simplify our calculations, we linearly approximate read fanout by only counting the cost of storing the bottom-most layer of the index pages in RAM. Main memory has grown to the point where read amplifications of one (or even zero) are common (Appendix A). Here, we focus on read fanouts with a read amplification of one.

2.2 B-Trees

Assuming that keys fit in memory, B-Trees provide optimal random reads; they perform a single disk seek each time an uncached piece of data is requested. In order to perform an update, B-Trees read the old version of the page, modify it, and asynchronously write the modification to disk. This works well if data fits in memory, but performs two disk seeks when the data to be modified resides on disk. Update-in-place hashtables behave similarly, except that they give up the ability to perform scans in order to make more efficient use of RAM.

Update in place techniques' *effective* write amplifications depend on the underlying disk. Modern hard disks transfer 100-200MB/sec, and have mean access times over 5ms. One thousand byte key value pairs are fairly common; it takes the disk $\frac{10^3}{10^8}$ seconds = 10us to write such a tuple sequentially. Performing two seeks takes a total of 10ms, giving us a write amplification of approximately 1000.

Appendix A applies these calculations and a variant of the five minute rule [15] to estimate the memory required for a read amplification of one with various disk technologies.

2.3 LSM-Trees

This section begins by describing the base LSM-Tree algorithm, which has unacceptably high read amplification, cannot take advantage of write locality, and can stall application writes for arbitrary periods of time. However, LSM-Tree write amplification is much lower than that of a B-Tree, and the disadvantages are avoidable.

Write skew can be addressed with tree partitioning, which is compatible with the other optimizations. However, we find that long write pauses are not adequately addressed by existing proposals or by state-of-the-art implementations.

2.3.1 Base algorithm

LSM-Trees consist of a number of append-only B-Trees and a smaller update-in-place tree that fits in memory. We call the in-memory tree C_0. Repeated scans and merges of these trees are used to spill memory to disk, and to bound the number of trees consulted by each read.

The trees are stored in key order on disk and the in-memory tree supports efficient ordered scans. Therefore, each merge can be performed in a single pass. In the version of the algorithm we implement (Figure 1), the number trees is constant. The on-disk trees are ordered by freshness; the newest data is in C_0. Newly merged trees replace the higher-numbered of the two input trees. Tree merges are always performed between C_i and C_{i+1}.

The merge threads attempt to ensure that the trees increase in size exponentially, as this minimizes the amortized cost of garbage collection. This can be proven in many different ways, and stems from a number of observations [25]:

1. Each update moves from tree C_i to C_{i+1} at most once. All merge costs are due to such movements.

2. The cost of moving an update is proportional to the cost of scanning the overlapping data range in the large tree. On average, this cost is $R_i = \frac{|C_{i+1}|}{|C_i|}$ per byte of data moved.

3. The size of the indexed data is roughly $|C_0| * \prod_{i=0}^{N-1} R_i$.

This defines a simple optimization problem in which we vary the R_i in order to minimize the sum of costs (bullet 2) while holding the index size constant (bullet 3). This optimization tells us to set each $R_i = \sqrt[N-1]{\frac{|data|}{|C_0|}}$. It immediately follows that the amortized cost of insertion is $O(\sqrt[N-1]{|data|})$.

Note that the analysis holds for best, average and worst-case workloads. Although LSM-Tree write amplification is much lower than B-Trees' worst case, B-Trees naturally leverage skewed writes. Skew allows B-Trees to provide much lower write amplifications than the base LSM-Tree algorithm. Furthermore, LSM-Tree lookups and scans perform $N-1$ times as many seeks as a B-Tree lookup, since (in the worst case), they examine each tree component.

2.3.2 Leveraging write skew

Although we describe a special case solution in Section 4.2, partitioning is the best way to allow LSM-Trees to leverage write skew [16]. Breaking the LSM-Tree into smaller trees and merging the trees according to their update rates concentrates merge activity on frequently updated key ranges.

It also addresses one source of write pauses. If the distribution of the keys of incoming writes varies significantly from the existing distribution, then large ranges of the larger tree component may be disjoint from the smaller tree. Without partitioning, merge threads needlessly copy the disjoint data, wasting I/O bandwidth and stalling merges of smaller trees. During these stalls, the application cannot make forward progress.

Partitioning can intersperse merges that will quickly consume data from the small tree with merges that will slowly consume the data, "spreading out" the pause over time. Sections 4.1 and 5 argue and show experimentally that, although necessary in some scenarios, such techniques are inadequate protection against long merge pauses. This conclusion is in line with the reported behavior of systems with partition-based merge schedulers, [1, 12, 19] and with previous work, [16] which finds that partitioned and baseline LSM-Tree merges have similar latency properties.

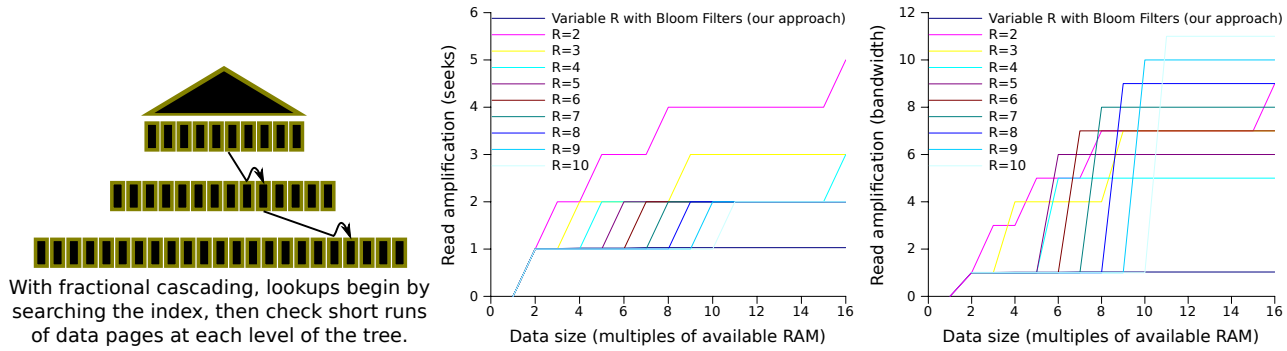

Figure 2: Fractional cascading reduces the cost of lookups from $O(log(n)^2)$ to $O(log(n))$, but mixes keys and data, increasing read amplification. For our scenarios, Bloom filters' maximum amplification is 1.03.

3. ALGORITHMIC IMPROVEMENTS

We now present bLSM, our new LSM-Tree variant, which addresses the LSM-Tree limitations we describe above. The first limitation of LSM-Trees, *excessive read amplification*, is only partially addressed by Bloom filters, and is closely related to two other issues: *exhaustive lookups* which needlessly retrieve multiple versions of a record, and *seeks during insert*. The second issue, *write pauses*, requires scheduling infrastructure that is missing from current implementations.

3.1 Reducing read amplification

Fractal cascading and Bloom filters both reduce read amplification. Fractal cascading reduces asymptotic costs; Bloom filters instead improve performance by a constant factor.

The Bloom filter approach protects the $C_1 ... C_N$ tree components with Bloom filters. The amount of memory it requires is a function of the number of items to be inserted, not the items' sizes. Allocating 10 bits per item leads to a 1% false positive rate,[1] and is a reasonable tradeoff in practice. Such Bloom filters reduce the read amplification of LSM-Tree point lookups from N to $1 + \frac{N}{100}$. Unfortunately, Bloom Filters do not improve scan performance. Appendix A runs through a "typical" application scenario; Bloom filters would increase memory utilization by about 5% in that setting.

Unlike Bloom filters, fractional cascading [18] reduces the asymptotic complexity of write-optimized LSM-Trees. Instead of varying R, these trees hold R constant and add additional levels as needed, leading to a logarithmic number of levels and logarithmic write amplification. Lookups and scans access a logarithmic (instead of constant) number of tree components. Such techniques are used in systems that must maintain large number of materialized views, such as the TokuDB MySQL storage engine [18].

Fractional cascading includes pointers in tree component leaf pages that point into the leaves of the next largest tree. Since R is constant, the cost of traversing one of these pointers is also constant. This eliminates the logarithmic factor associated with performing multiple B-Tree traversals.

The problem with this scheme is that the cascade steps of the search examine pages that likely reside on disk. In effect, it eliminates a logarithmic in-memory overhead by increasing read amplification by a logarithmic factor.

Figure 2 provides an overview of the lookup process, and plots read amplification vs. fanout for fractional cascading and for three-level LSM-Trees with Bloom filters. No setting of R allows fractional cascading to provide reads competitive with Bloom filters—reducing read amplification to 1 requires an R large enough to ensure that there is a single on-disk tree component. Doing so leads to $O(n)$ write amplifications. Given this, we opt for Bloom filters.

3.1.1 Limiting read amplification for frequently updated data

On their own, Bloom filters cannot ensure that read amplifications are close to 1, since copies of a record (or its deltas) may exist in multiple trees. To get maximum read performance, applications should avoid writing deltas, and instead write base records for each update.

Our reads begin with the lowest numbered tree component, continue with larger components in order and stop at the first base record. Our reads are able to terminate early because they distinguish between base records and deltas, and because updates to the same tuple are placed in tree levels consistent with their ordering. This guarantees that reads encounter the most recent version first, and has no negative impact on write throughput. Other systems nondeterministically assign reads to on-disk components, and use timestamps to infer write ordering. This breaks early termination, and can lead to update anomalies [1].

3.1.2 Zero-seek "insert if not exists"

One might think that maintaining a Bloom filter on the largest tree component is a waste; this Bloom filter is by far the largest in the system, and (since C_2 is the last tree to be searched) it only accelerates lookups of non-existent data. It turns out that such lookups are extremely common; they are performed by operations such as "insert if not exists."

In Section 5.2, we present performance results for bulk loads of bLSM, InnoDB and LevelDB. Of the three, only bLSM could efficiently load and check our modest 50GB unordered data set for duplicates. "Insert if not exists" is a widely used primitive; lack of efficient support for it renders high-throughput writes useless in many environments.

3.2 Dealing with write pauses

Regardless of optimizations that improve read amplification and leverage write skew, index implementations that impose long, sporadic write outages on end users are not particularly practical. Despite the lack of good solutions,

[1]Bloom filters can claim to contain an item that has not been inserted, causing us to unnecessarily examine the disk.

LSM-Trees are regularly put into production. We describe workarounds that are used in practice here.

At the index level, the most obvious solution (other than unplanned downtime) is to introduce extra C_1 components whenever C_0 is full and the C_1 merge has not yet completed [13]. Bloom filters reduce the impact of extra trees, but this approach still severely impacts scan performance. Systems such as HBase allow administrators to temporarily disable compaction, effectively implementing this policy [1]. As we mentioned above, applications that do not require performant scans would be better off with an unordered log structured index.

Passing the problem off to the end user increases operations costs, and can lead to unbounded space amplification. However, merges can be run during off-peak periods, increasing throughput during peak hours. Similarly, applications that index data according to insertion time end up writing data in "almost sorted" order, and are easily handled by existing merge strategies, providing a stop-gap solution until more general purposes systems become available.

Another solution takes partitioning to its logical extreme, creating partitions so small that even worst case merges introduce short pauses. This is the technique taken by Partitioned Exponential Files [16] and LevelDB [12]. Our experimental results show that this, on its own, is inadequate. In particular, with uniform inserts and a "fair" partition scheduler, each partition would simultaneously evolve into the same bad state described in Figure 4. At best, this would lead to a throughput collapse (instead of a complete cessation of application writes).

Obviously, we find each of these approaches to be unacceptable. Section 4.1 presents our approach to merge scheduling. After presenting a simple merge scheduler, we describe an optimization and extensions designed to allow our techniques to coexist with partitioning.

3.3 Two-seek scans

Scan operations do not benefit from Bloom filters and must examine each tree component. This, and the importance of delta-based updates led us to bound the number of on-disk tree components. bLSM currently has three components and performs three seeks. Indeed, Section 5.6 presents the sole experiment in which InnoDB outperforms bLSM: a scan-heavy workload.

We can further improve short-scan performance in conjunction with partitioning. One of the three on-disk components only exists to support the ongoing merge. In a system that made use of partitioning, only a small fraction of the tree would be subject to merging at any given time. The remainder of the tree would require two seeks per scan.

4. BLSM

As the previous sections explained, scans and reads are crucial to real-world application performance. As we decided which LSM-Tree variants and optimizations to use in our design, our first priority was to outperform B-Trees in practice. Our second concern was to do so while providing asymptotically optimal LSM-Tree write throughput. The previous section outlined the changes we made to the base LSM-Tree algorithm in order to meet these requirements.

Figure 1 presents the architecture of our system. We use a three-level LSM-Tree and protect the two on-disk levels with Bloom filters. We have not yet implemented partitioning, and instead focused on providing predictable, low latency writes. We avoid long write pauses by introducing a new class of merge scheduler that we call a *level scheduler* (Figure 4). Level schedulers are designed to complement existing partition schedulers (Figure 3).

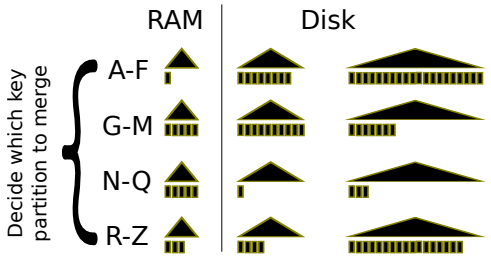

Figure 3: *Partition schedulers* leverage skew to improve throughput. A greedy policy would merge N − Q; which uses little I/O to free a lot of RAM.

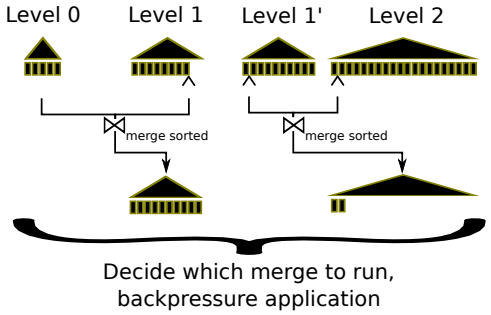

Figure 4: *Level schedulers* (such as spring and gear) decide which level to merge next. This tree is in danger of unplanned downtime: Level 0 (RAM) and 1 are almost full, and the merge between level 1 and 2 has fallen behind.

4.1 Gear scheduler

In this section we describe a *gear scheduler* that ensures merge processes complete at the same time (Figure 5). This scheduler is a subcomponent of the *spring and gear scheduler* (Section 4.3). We begin with the gear scheduler because it is conceptually simpler, and to make it clear how to generalize spring and gear to multiple levels of tree components.

As Section 3.2 explained, we are unwilling to let extra tree components accumulate, as doing so compromises scan performance. Instead, once a tree component is full, we must block upstream writes while downstream merges complete.

Downstream merges can take indefinitely long, and we are unwilling to give up write availability, so our only option is to synchronize merge completions with the processes that fill each tree component. Each process computes two progress indicators: $in_{progress}$ and $out_{progress}$. In Figure 5, numbers represent the $out_{progress}$ of C_0 and the $in_{progress}$ of C_1. Letters represent the $out_{progress}$ of C_1 and the $in_{progress}$ of C_2.

We use a clock analogy to reason about the merge processes in our system. Clock gears ensure each hand moves at a consistent rate (so the minute and hour hand reach 12 at the same time, for example). Our merge processes ensure that trees fill at the same time as more space becomes available downstream. As in a clock, multiple short

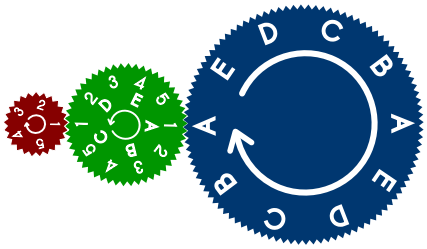

Figure 5: The gear scheduler tracks the progress of the merges and ensures they complete at the same time, preventing them from stalling the application.

upstream merges (sweeps of the minute hand) may occur per downstream merge (sweep of the hour hand). Unlike with a clock, the only important synchronization point is the hand-off from smaller component to larger (the meeting of the hands at 12).

We define $in_{progress}$ as follows:

$$in_{progress_i} = \frac{bytes\ read\ by\ merge_i}{|C'_{i-1}| + |C_i|}$$

Note that we ignore the cost of writing data to disk. An important, but subtle property of $in_{progress}$ is that any merge activity increases it, and that, within a single merge, the cost (in bytes transferred) of increasing $in_{progress}$ by a fixed amount will never vary by more than a small constant factor. We say that estimators with this property are *smooth*.

Runs of deletions or non-overlapping ranges of the input trees can cause more intuitive estimates to become "stuck" at the same value for long periods of time. For example, early versions of our scheduler used estimates that focused on I/O associated with the larger tree component (which consists of the bulk of the work). This led to routine stalls.

The definition of $out_{progress}$ is more complex:

$$out_{progress_i} = \frac{in_{progress_i} + floor(|C_i|/|RAM|^i)}{ceil(R)}$$

Returning to our clock analogy, the $floor$ term is the computation one uses to determine what hour is being displayed by an analog clock.

Assuming an insert-only workload, each tree needs to be merged R times before it fills up and becomes ready to be merged with the downstream tree. The $floor$ term uses current tree sizes to estimate the number of such merges this tree component has undergone. $\frac{|C_i|}{|RAM|^i}$ must be less than R, or the previous merge completion would have triggered a merge between C_i and C_{i+1}. This, combined with the fact that $in_{progress_i}$ ranges from zero to one ensures that $out_{progress}$ ranges from zero to one, and that it is set to one immediately before a new merge is triggered. Finally, within a given merge with the upstream tree, $out_{progress}$ inherits its smoothness property from $in_{progress}$.

Across merges, workload shifts could cause the current tree to shrink (for example, the application could delete data), which could cause $out_{progress}$ to decrease substantially. This will cause the downstream mergers to shut down until the current tree increases in size. If we had been able to predict the workload shift ahead of time, we could have throttled the application less aggressively. Since we cannot predict future workload shifts, pausing downstream mergers seems to be the best approach.

Section 5.2 presents a timeseries plot of the write throughput of our index. The throughput varies by a bit under a factor of two due to errors in our $in_{progress}$ and $out_{progress}$ estimates. In results not presented here, we experimented with better estimators of $in_{progress}$ and $out_{progress}$ but found that external factors such as I/O interference lead to more variance than inaccuracies in our model.

4.2 Snowshoveling

In this section, we describe *snowshoveling*, also called *tournament sort* or *replacement-selection sort*, which significantly improves throughput for sequential and some skewed workloads. For random workloads, snowshoveling increases the effective size of C_0 by a factor of four, doubling write throughput. For adversarial workloads, it increases the size by a factor of two. In the best case, updates arrive in sorted order, and it streams them directly to disk.

Unfortunately, snowshoveling can block application writes for arbitrary periods of time. At the end of this section, we explain how partitioning can be used to "fix up" the problems snowshoveling creates.

The basic idea stems from the sorting algorithms used by tape and punch card based database systems. It is important to minimize the number of runs performed by merge sort on such machines, as merging N runs requires N physical drives or multiple passes. Naive implementations fill RAM, sort its contents, and write a RAM-sized run. Snowshoveling fills RAM, writes back the lowest valued item, and then reads a value from the input. It proceeds by writing out the lowest key that comes after the last value written.

If the data arrives in random order, this doubles the length of runs, since, on average, each item has a 50% chance of coming after the cursor. When the run starts, most incoming data lands after the cursor; at the end, almost all lands before it. Alternatively, if the input is already sorted, all writes arrive after the cursor, and snowshoveling produces a run containing the entire input. In the worst case, updates are in reverse sorted order, and the run is the size of RAM.

4.2.1 Implications for LSM-Tree merges

Snowshoveling increases the amount of data that each C_0 : C_1 merge consumes. Recall from Section 2.3 that LSM-Tree write amplification is $O(\sqrt{\frac{|data|}{|RAM|}})$; snowshoveling effectively increases $|RAM|$. In addition to generating longer sorted runs, it eliminates the partitioning of C_0 and C'_0.

In the gear scheduler, each merge reads from a special tree component C'_{i-1} instead of directly from C_{i-1}. Incoming writes are delivered to C_{i-1}, which atomically replaces C'_{i-1} when it is ready to be merged with the downstream tree. This is particularly inconvenient for C_0, since it halves the pool of RAM that can be used for writes; after all, at the end of a C_0 to C_1 merge, the data stored in C'_0 is useless and discarded. Snowshoveling ensures that the entire pool contains useful data at all times. This, combined with the factor of two improvement it provides for random writes, is the basis of our claim that snowshoveling increases the effective size of C_0 by a factor of four.

However, breaking the partitioning between old and new updates risks stalling application writes. Partitioning the write pool decoupled application writes from the rate at which data from C'_0 is consumed. Thus, as long as $in_{progress}$

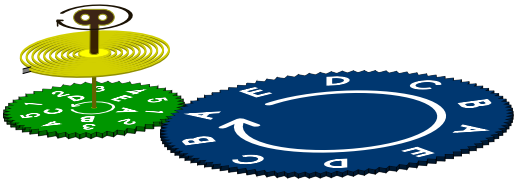

Figure 6: The spring and gear scheduler reduces the coupling between application write rates and merge rates. In our clock analogy, turning the key engages a ratchet that winds a coil spring. Our implementation applies proportional backpressure to application writes once memory is nearly full. As memory begins to empty, it slows down merge processing.

was smooth, we could use it to determine when to apply backpressure (or not).

With snowshoveling, we lose this luxury. Instead, application writes must wait for the $C_0 : C_1$ merger to consume data. In the worst case, the merge consumes the entirety of C_0 at the beginning (or end) of the merge, blocking writes while the entirety of C_1 is scanned and copied into C_1'.

4.2.2 The need for partitioning

The stalls introduced by snowshoveling are due to mismatched distributions of keys in C_0 and C_1. In the worst case, C_0 and C_1 do not overlap at all, and there is no need to scan or copy C_1. This is not a new observation; partitioned exponential files were designed to leverage such skew, as were BigTable [8], Cassandra [19], HBase [1], and LevelDB [12]. With skewed inserts, such systems have asymptotically better write throughput than unpartitioned LSM-Trees.

Furthermore, they can reorder the work performed by the C_0 and C_1 merger so that it consumes data from C_0 at a uniform rate. It is easy to see (and confirmed by the experiments in Section 5) that no reordering of work within the $C_0 : C_1$ merge will prevent that merge from blocking on the $C_1 : C_2$ merge; the two problems are orthogonal.

4.3 Spring and gear scheduler

The gear scheduler suffers from a number of limitations. First, we found that its behavior was quite brittle, as it tightly couples the timing of insertions into C_0 with the timing of the merge threads' progress.

Second, as Section 4.2.2 explained, the gear scheduler interacts poorly with snowshoveling; it requires a "percent complete" estimate for merges between C_0 and C_1, which forces us to partition RAM, halving the size of C_0.

We modify the gear scheduler based on the observation that snowshoveling (and more sophisticated partition-based schemes) expose a more natural progress indicator: the fraction of C_0 currently in use. The spring and gear scheduler attempts to keep the size of C_0 between a low and high water mark. It pauses downstream merges if C_0 begins to empty and applies backpressure to the application as C_0 fills. This allows it to "absorb" load spikes while ensuring that merges have enough data in C_0 to perform optimizations such as snowshoveling and intelligent partition selection. The downstream merge processes behave as they did in the gear scheduler, maintaining the decoupling of their consumption rates from the application.

We believe this type of merge scheduler is a natural fit for existing partitioned log structured trees. The main difficulty in applying this approach is in tracking sizes of tree components and estimating the costs of future merges. A simple approach would take the sum of the sizes of each level of C_i and apportion resources to each level accordingly. More complex schemes would extend this by taking write skew into account at each level of the merge.

4.4 Implementation details

This section describes our bLSM-Tree implementation. We leave out details that apply to conventional indexes, and instead focus on components specific to bLSM-Trees.

4.4.1 Merge threads

Merge is surprisingly difficult to implement. We document some of the issues we encountered here.

We needed to batch tree iterator operations to amortize page pins and mutex acquisitions in the merge threads. Upon implementing batching, we applied it to application-facing scans as well. This introduced the possibility that a tree component would be deleted in the middle of an application scan. To work around this problem, we add a logical timestamp to the root of each tree component, and increment it each time a merge completes and zeros out the old tree.

Next, we implemented merge scheduling, which introduced a new set of concurrency problems. It is prohibitively expensive to acquire a coarse-grained mutex for each merged tuple or page. Therefore, each merge thread must take action based upon stale statistics about the other threads.

It is possible for an upstream merger to throttle a downstream merger and vice versa. They must use the stale statistics to do so without introducing deadlocks or idling the disk. Also, the stale statistics must be "fresh enough" to ensure that the progress estimates seen by the mergers are smooth. Otherwise, application writes may stall.

More recently, we rewrote a number of buffer manager subcomponents to avoid synchronization bottlenecks, allowing the buffer manager to saturate disk when run in isolation.

Each of these changes significantly improved performance. However, our experiments in Section 5.2 suggest that additional concurrency bottlenecks exist.

4.4.2 Buffer management and Recovery

The bLSM-Tree implementation is based upon Stasis, a general-purpose transactional storage system [29]. We use Stasis for two reasons. First, its *region allocator* allows us to allocate chunks of disk that are guaranteed contiguous, eliminating the possibility of disk fragmentation and other overheads inherent in general-purpose filesystems.

Second, when we began this project, Stasis' buffer manager had already been carefully tuned for highly concurrent workloads and multi-socket machines. Furthermore, we have a good understanding of its code base, and added a number of features, such as support for a CLOCK eviction policy (LRU was a concurrency bottleneck), and an improved writeback policy that provides predictable latencies and high-bandwidth sequential writes.

Stasis uses a write ahead log to manage bLSM's metadata and space allocation; this log ensures a physically consistent [30] version of the tree is available at crash. We do not write the contents of our index and data pages to the log; instead we simply force-write them to disk via the buffer

manager. Similarly, merge threads avoid reading pre-images of pages they are about to overwrite.

We use a second, *logical*, log to provide durability for individual writes. Below, when we say that we disable logging, it is in reference to the logical log. The use of a logical log for LSM-Tree recovery is fairly common [1, 12, 19, 31, 34], and can be used to support ACID transactions [16], database replication [31] and so on. bLSM also provides a degraded durability mode that does not log updates at all. After a crash, older (up to a well-defined point in time) updates are available, but recent updates may be lost. These semantics are useful for high-throughput replication [31], and are increasingly supported by LSM-Tree implementations, including LevelDB [12]. However, replaying the log at startup is extremely expensive. Snowshoveling delays log truncation, increasing the amount of data that must be replayed.

4.4.3 Bloom filters

We were concerned that we would incur significant Bloom filter overheads. In practice, the computational, synchronization and memory overheads of the Bloom filters are insignificant, but recovery is non-trivial.

Our Bloom filter is based upon double hashing [17]. We create one Bloom filter each time a merge process creates a new tree component, and we delete the Bloom filter when we delete the corresponding on-disk tree component. Since the on-disk trees are append-only, there is never a need to delete from the Bloom filter. Furthermore, we track the number of keys in each tree component, and size the Bloom filter for a false positive rate below 1%.

Bloom filter updates are *monotonic*; bits always change from zero to one, and there is no need to atomically update more than one bit at a time. Therefore, there is no reason to attempt to insulate readers from concurrent updates.[2]

However, Bloom filters complicate recovery, and we currently do not persist them to disk. The Bloom filters are small compared to the other data written by merges, so we do not expect them to significantly impact throughput. However, they are too large to allow us to block writers as they are synchronously written to disk. We considered a number of techniques, such as Bloom filter checkpoints and logging, or existing techniques [4]. Ultimately, we decided to overlap Bloom filter writeback with the next merge.

Stasis ensures each tree merge runs in its own atomic and durable transaction. bLSM ensures the transactions are isolated and consistent. Therefore, we can defer the commit of the merge transaction until the Bloom filter is written back, but run the next merge immediately and in parallel. Since Stasis does not support concurrent, conflicting transactions, we must ensure that the next merge modifies data that is disjoint from the previous one. Running merges that touch disjoint data within a single level requires partitioning. Therefore, we deferred implementation of this feature.

5. EXPERIMENTS

bLSM targets a broad class of applications. In this section, we compare its performance characteristics with the full range of primitives exported by B-Trees (we compare to InnoDB) and another LSM-Tree variant (LevelDB).

[2]Though, when moving data from C_0 to C_1, we issue a memory barrier after updating the Bloom filter and before deleting from C_0. The memory barrier is "free," as it is implemented by releasing a mutex associated with C_0.

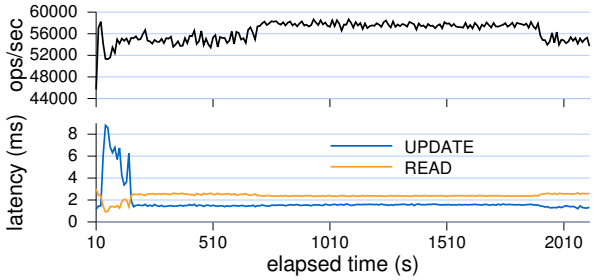

Figure 9: bLSM shifting from 100% uniform writes to Zipfian accesses (80% read, 20% blind write).

We begin with insert-heavy scenarios. This gives us an idea of the bulk load performance of the systems, which is crucial to analytical processing.

Second, we compare random lookup performance. As we argued above, best-of-class random reads are a prerequisite for most interactive applications.

Third, we evaluate throughput as we vary read:write ratios. We distinguish between read-modify-write, where B-Trees are within a small constant factor of optimal, and blind writes, where LSM-Trees have an opportunity to significantly improve performance. Read-modify-write is particularly important in practice, as it is used by a wide range of applications and is therefore a crucial component of any general purpose index implementation.

Finally, we turn our attention to scans. Short scans are crucial for delta-based update schemes and a wide range of applications rely upon range scan APIs.

We argue that performing well on these operations is sufficient for present-day data management systems.

5.1 Experimental setup

Our experimental setup consists of a single socket machine with 16GB of RAM and an Intel "Sandy Bridge" i7 processor with 8 threads. We have set up two software RAID 0 arrays with 512KB stripes. The first consists of two 10K RPM SATA enterprise drives. The second consists of two OCZ Vertex 2 SSDs. The filesystems are formatted using ext4.

We use YCSB, the *Yahoo! Cloud Serving Benchmark* tool, to generate load [11]. YCSB generates synthetic workloads with varying degrees of concurrency and statistical distributions. We configured it to generate databases that consist of 50 gigabytes worth of 1000 byte values. We make use of two request distributions: uniform and Zipfian. The Zipfian distribution is run using YCSB's default parameters. All systems run in a different process than the YCSB workers.

We dedicate 10GB of RAM to buffer cache for InnoDB and LevelDB. For bLSM, we divide memory into 8 GB for C_0 and 2 GB for the buffer cache. LevelDB makes use of extremely small C_0 components, and is geared toward merging these components as they reside in OS cache, or its own memory pool. We disable LevelDB compression to make it easier to reason about memory requirements. We use YCSB's default key length which is variable, and on the order of tens of bytes. This yields a fanout of $> \frac{4096}{50}$, and motivated our choice to dedicate most of RAM to C_0.

We load the systems with 256 unthrottled YCSB threads, and run the remainder of the experiments with 128 unthrottled threads. With hard disks, this setup leads to latencies in

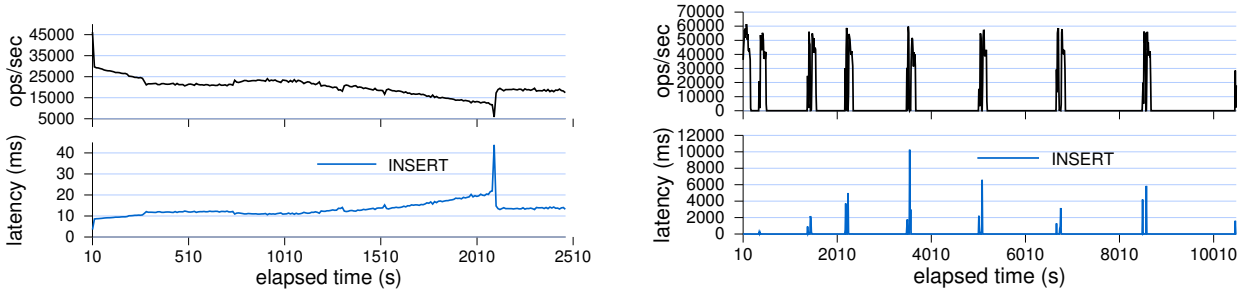

Figure 7: bLSM (left) and LevelDB (right) performing random-order inserts. The systems load the same data; bLSM's throughput is more predictable and it finishes earlier.

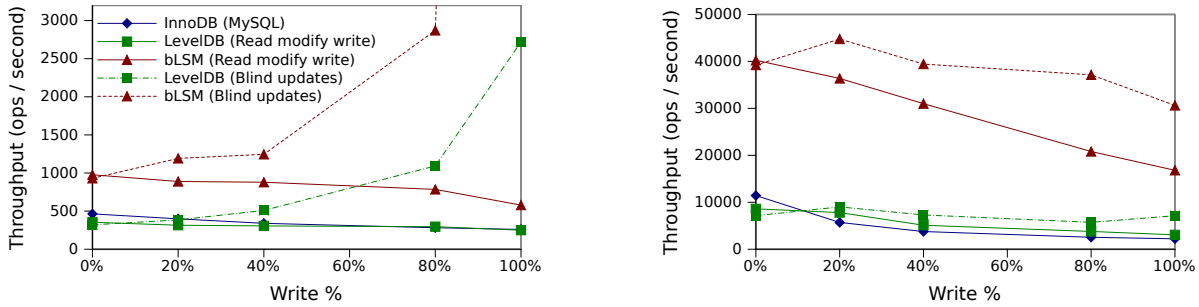

Figure 8: Throughput vs. write/read ratio (uniform random access) for hard disks (left) and SSD (right).

the 100's of milliseconds across all three systems. Throttling the threads, as would be done in production, would reduce the latencies. However, running the systems under continuous overload reliably reproduces throughput collapses that might otherwise be non-deterministic and difficult to characterize. We disable write-ahead logging to the greatest extent that is compatible with running the systems in production; none of the systems sync their logs at commit.

LevelDB and bLSM are designed with compatibility with external write ahead logs in mind. We expect future logs to be stored on dedicated hardware that supports high-throughput commits, such as filers with NVRAM, RAID controllers with battery backups, enterprise SSDs with supercapacitors, and logging services.

5.2 Raw insert performance

The goal of this experiment is twofold. First, we determined the strongest set of semantics each system could provide without resorting to random reads. InnoDB provides the weakest fast insert primitive: we had to pre-sort the data to get reasonable throughput. LevelDB came in second: random inserts have high throughput, but only if we use blind-writes. Also, LevelDB introduced long pauses as the load commenced. bLSM fared the best on this test. It provided steady high-throughput inserts, and tested for the pre-existence of each tuple as it was inserted.

We conclude that, although InnoDB and LevelDB are capable of servicing high throughput batch or asynchronous writes, neither is appropriate for high-throughput, insert-heavy, low-latency workloads.

Furthermore, bLSM matches or outperforms both systems for the workloads they specialize in: it provides insert semantics that are equivalent to the B-Tree ("insert if not exists"), and significantly higher throughput than LevelDB.

None of the systems we tested were able to run the disks at full sequential bandwidth. bLSM came the closest to doing so, and had higher throughput than InnoDB and LevelDB. In principle, InnoDB's load of pre-sorted data should have let it run at near disk speeds; with write ahead logging overheads, we expect B-Trees to have write amplifications of 2-4 for sequential writes. It loaded the data at approximately 7,000 operations per second (7MB/sec) The underlying disks are capable of 110-130 MB/second each.

We were surprised our approach outperformed InnoDB, and suspect that tuning problems are to blame. Similarly, we expected LevelDB's multi-level trees to provide higher write throughput than our two-level approach. We have been told that significant improvements have been made to LevelDB's merge scheduler; we leave more detailed performance comparisons between two-level and multi-level trees to future work.

It is notoriously difficult to tune storage systems, and we read little into the constant factor differences we see between InnoDB's sequential bandwidth (and random reads) and our own. As we discuss our experiments, we focus on differences that are fundamental to the underlying algorithms.

5.3 Random read performance

Historically, read amplification has been a major drawback of LSM-trees [14, 16]. Figure 8 shows that this is no longer the case for random index probes. Again, we see unexpected constant factor differences between InnoDB and bLSM; bLSM is about 2x faster on hard disk, and 4x faster on SSD. We confirmed that InnoDB and bLSM perform about one disk seek per read[3], which is the underlying metric we are interested in.

Upon further examination, we found that the drive write queues are shorter under InnoDB than they are under bLSM.

[3]The actual number is a bit lower due to cache hits.

Furthermore, InnoDB uses 16KB pages[4], while we opt for 4KB, and the version of MySQL we used hard codes a number of optimizations, such as prefetching, that are counterproductive for this workload. These factors reduce the number of I/O operations per second (IOPS) the drives deliver.

We also confirmed that LevelDB performs multiple disk seeks per read, as expected. This is reflected by its read throughput numbers. We believe the random read performance gap between bLSM and LevelDB is due to the underlying algorithm, though modified versions of LevelDB with support for Bloom filters are now available.

5.4 Update costs

Figure 8 also allows us to compare update strategies. As expected, with hard disks and SSDs, read-modify-writes are strictly more expensive than reads. In contrast, on hard disks, blind write operations are significantly faster than reads in both systems. This reiterates the importance of eliminating hard disk seeks whenever possible.

The SSD trends are perhaps the most interesting results of this batch of experiments. SSDs provide many more IOPS per MB/sec of sequential bandwidth, but they severely penalize random writes. Therefore, it is unclear whether LSM-Tree writes will continue to outperform B-Tree writes as SSD technology matures. At 100% writes, InnoDB's throughput was 20% its starting throughput. In contrast, bLSM maintained 41% of its original throughput for read-modify-write, and 78% for blind writes. The hard drive trends are similar to the SSD, except that the relative cost of writes is much lower. This reduces the slopes of the read-modify-write trendlines, and causes write throughputs to increase rapidly as we approach 100% blind writes.

Again, we note that none of the systems are making particularly effective use of the disks. Here, we focus on problems with bLSM.

At 100% blind writes, bLSM hits a concurrency bottleneck. Hard disk throughput is 33,000 ops/second; 10% higher than the throughput we achieve with the SSDs. Each hard drive provides 110-130 MB/sec. Each SSD provides 285 (275) MB/sec sequential reads (writes). Therefore, we expect bLSM to provide at least 2-3x higher write throughput on SSD, assuming we overcome the current implementation bottlenecks.

5.5 Shifting workload distributions

Section 4.2.2 explains why shifts in workload distributions can lead our merge scheduler to introduce arbitrarily long pauses. We are confident that we can reproduce such issues with adversarial workloads (such as reverse-order bulk inserts). In this section, we measure the impact in a more realistic scenario. Figure 9 is a timeseries plot of bLSM's throughput as it shifts from a 100% uniform write workload to an 80% read Zipfian workload. At the beginning of the experiment, bLSM has been saturated with uniform writes for an extended period of time; we switch to Zipfian at $t = 0$. This corresponds to scenarios that switch from bulk or analytical processing to serving workloads.

Initially, performance ramps up as internal index nodes are brought into RAM.[5] At the end of this phase, it settles into high-throughput writes with occasional drops due to merge hiccups, and quickly levels off. Such performance characteristics are likely acceptable in practice. Also, even with 128 unthrottled workers, bLSM maintains stable latencies around 2ms. (This test was run atop the SSDs.)

5.6 Scans

Our final set of experiments compares the scan performance of InnoDB and bLSM. To simulate B-Tree fragmentation, we ran the scan experiment last, after the trees were fragmented by the read-write tests. First, we measured short scan performance. YCSB generates scans of length 1-4 uniformly at random in this test. We expect most scans to read one page in InnoDB, and to touch all three tree components with bLSM. The results bear this out. MySQL performs 608 short range scans per second, while bLSM only performs 385.

We note that this has important implications for delta-based schemes: scans are approximately three times more expensive than point lookups. Therefore, applications that generate fewer than two deltas per read would be better off using read-modify-write. If, as in Section 3.3, we can reduce this to two seeks, then, even for read heavy workloads, the overhead of using deltas instead of read-modify-write can be reduced to the cost of inserting the delta. In such scenarios, reads that encounter a delta could immediately insert the merged tuple into C_0.

Finally, we repeated the scan test using ranges that contain 1 to 100 tuples. As expected, B-Tree fragmentation erases InnoDB's performance advantage; bLSM provides 165 scans per second vs. InnoDB's 86.

6. RELATED WORK

A wide range of log structured indexes have been proposed, implemented and deployed in commercial settings. Algorithmically, our bLSM-tree is closest to Log Structured Merge Trees [25], which make use of exponentially sized tree components, do not leverage Bloom filters, and make use of a different approach to merges than we describe above. Partitioned exponential files introduce the idea of partitioning [16]. That work also includes an exhaustive survey and performance study of related techniques. The primary limitation of partitioned exponential files is that they rely upon specifics of hard disk geometry for good random read performance. When combined with partitioning, we believe that the bLSM-Tree outperforms or matches their approach on all workloads except for short range scans atop hard disks.

FD-Trees [20] are LSM-Trees that include a deamortization scheme that is similar to our gear scheduler and is incompatible with snowshoveling. It backpressures based on the number of pages left to merge; we note that this estimate is smooth. FD-Trees optimize for SSDs, based on the observation that writes with good locality have performance similar to sequential writes.

BigTable is an LSM-Tree variant based upon major and minor compactions. The idea is to write sorted runs out to disk, and then to attempt to merge runs of similar size using *minor compactions*. Occasionally, when there are too many such runs, a *major compaction* scans and rewrites all of the delta blocks associated with a given base block. This technique is also used by HBase and Cassandra (pre 1.0). To the best of our knowledge, this technique has two fundamental drawbacks. First, merging based on block size (and not chronology) breaks one-seek lookups. The problem is that

[4]See Appendix A for more information about page sizes.

[5]The cache warming period could be reduced by telling the buffer manager to prefer index pages over pages that were pinned by merge threads.

some unsearched block could have a newer delta, so extra base records and deltas must be fetched from disk. Furthermore, without a more specific merge policy, it is very difficult to reason about the amortized cost of inserts. To the best of our knowledge, all current implementations of this approach suffer from write pauses or degraded reads.

Cassandra version 1.0 adds a merger based upon exponential sizing and addresses the problem of write pauses [13]. If merges fall behind it writes additional (perhaps overlapping) range partitions to C_1, degrading scans but allowing writes to continue to be serviced.

TokuDB [18] and LevelDB [12] both make use of partitioned, exponentially sized levels and fractional cascading. As we argued above, this improves write throughput at the expense of reads and scans. Riak modified LevelDB to support Bloom filters and improved the merge scheduler [3]. These structures are related to cache oblivious lookup arrays (COLA) and streaming B-trees [5], which include deamortization algorithms with goals similar to our merge schedulers, except that they introduce extra work to each operation, while we focus on rate-limiting asynchronous background processes to keep the system in a favorable steady state.

The bLSM implementation is based upon Rose, a column-compressed LSM-Tree for database replication tasks [31]. Rose does not employ Bloom filters and has a naive merge scheduler with unbounded write latency. The compression techniques lead to constant factor decreases in write amplification and do not impact reads. LevelDB makes use of a general-purpose compression algorithm.

A number of flash-optimized indexes have been proposed for use in embedded systems, including FlashDB [24], which varies its layout based upon read-write ratios and hardware parameters (as do TokuDB and many major-minor compaction systems), and MicroHash [35], which targets embedded devices and has extremely high read fanout. In contrast to these systems, we achieve lower read and write amplifications, but assume ample main memory is available.

We do not have space to cover unordered approaches in sufficient detail; the LFS (Log File System) introduced many of the techniques used by such systems. WAFL [27] is a commercially-available hybrid filesystem that stores small objects in an LFS-like format. BitCask [33], BDB-JE [26] and Primebase [2] are contemporary examples in the key value and database space. BDB-JE uses unordered storage for data, but a B-Tree for the index, allowing it to perform (potentially seek-intensive) scans. Primebase is a log structured MySQL storage engine. SILT is an unordered store that uses 0.7 bytes per key to achieve a read amplification of 1 [21].

A number of indexes target hybrid storage systems. LHAM indexes provide historical queries over write-once-read-many (WORM) storage [23], and multi-tier *GTSSL* log structured indexes target hybrid disk-flash systems and adapt to varying read-write ratios [34].

7. CONCLUSION

bLSM-Trees provide a superset of standard B-Tree functionality; they introduce a zero-seek "blind write" primitive. Blind writes enable high-throughput writes and inserts against existing data structures, which are crucial for analytical processing and emerging classes of applications.

Our experiments show that, with the exception of short range scans (a primitive which is primarily useful for leveraging blind writes), bLSM-Trees provide superior performance across the entire B-Tree API. These results rely heavily upon our use of a three-level bLSM-tree and Bloom filters.

However, our spring and gear scheduler is what ultimately allows bLSM-Trees to act as drop-in B-Tree replacements. Eliminating write throughput collapses greatly simplifies operations tasks and allows bLSM-Trees to be deployed as the primary backing store for interactive workloads.

These results come with two caveats. First, the relative performance of update-in-place and log-structured approaches is subject to changes in the underlying devices. Second, we target applications that manage small pieces of data. Fundamentally, log structured approaches work by replacing random I/O with sequential I/O. As the size of objects increase, the sequential costs dominate and update-in-place techniques provide superior performance.

Furthermore, a number of technical issues remain. Our system remains a partial solution; it cannot yet leverage (or gracefully cope with) write skew. Also, transitioning to logical logging has wide ranging impacts on recovery, lock management, and other higher-level database mechanisms.

Well-known techniques address each of these remaining concerns; we look forward to applying these techniques to build unified transaction and analytical processing systems.

8. ACKNOWLEDGMENTS

We would like to thank Mark Callaghan, Brian Cooper, the members of the PNUTS team, and our shepherd, Ryan Johnson for their invaluable feedback.

bLSM is open source and available for download:

`http://www.github.com/sears/bLSM/`

9. REFERENCES

[1] http://hbase.apache.org/.
[2] https://launchpad.net/pbxt.
[3] http://wiki.basho.com/.
[4] M. Bender, M. Farach-Colton, R. Johnson, B. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. In *HotStorage*, 2011.
[5] 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*, 2007.
[6] D. Borthakur, J. Gray, J. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, et al. Apache Hadoop goes realtime at FaceBook. In *Sigmod*, 2011.
[7] M. Callaghan. Read amplification factor. *High Availability MySQL*, August 2011.
[8] F. Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.
[9] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears. Walnut: A unified cloud object store. In *Sigmod*, 2012.
[10] B. F. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2), 2008.
[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. SoCC '10, 2010.
[12] J. Dean and S. Ghemawat. *LevelDB*. Google, http://leveldb.googlecode.com.
[13] J. Ellis. The present and future of Apache Cassandra. In *HPTS*, 2011.
[14] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google file system. In *SOSP*, 2003.

	SSD		Hard disk	
	SATA	PCI-E	Server	Media
Capacity (GB)	512	5000	300	2000
Reads / second	50K	1M	500	250
Access Frequency	GB of B-Tree index cache per drive			
Minute	0.302	6.03	0.003	0.002
Five minute	1.51	30.2	0.015	0.008
Half hour	9.05	-	0.091	0.045
Hour	-	-	0.181	0.091
Day	-	-	4.35	2.17
Week	-	-	-	15.2
Month	-	-	-	-
Full disk	12.5	122	7.32	48.8

Table 2: RAM required to cache B-Tree nodes to address various storage devices. Hot data causes devices to be seek bound, leading to unused capacity and less cache. We assume 100 byte keys, 1000 byte values and 4096 byte pages.

[15] J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Record*, 26(4), 1997.

[16] C. Jermaine, E. Omiecinski, and W. G. Yee. The partitioned exponential file for database storage management. *The VLDB Journal*, 16(4), 2007.

[17] Kirsch and Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In *ESA*, 2006.

[18] B. C. Kuszmaul. How TokuDB fractal trees indexes work. Technical report, TokuTek, 2010.

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

[20] Y. Li, B. He, J. Y. 0001, Q. Luo, and K. Yi. Tree indexing on solid state drives. *PVLDB*, 3(1):1195–1206, 2010.

[21] H. Lim, B. Fan, D. Andersen, and M. Kaminsky. Silt: a memory-efficient, high-performance key-value store. In *SOSP*, 2011.

[22] M. Moshayedi and P. Wilkison. Enterprise SSDs. *ACM Queue*, 6, July 2008.

[23] P. Muth, P. O'Neil, A. Pick, and G. Weikum. The LHAM Log-structured history data access method. In *VLDB Journal*, 2000.

[24] S. Nath and A. Kansal. FlashDB: Dynamic self-tuning database for NAND flash. In *IPSN*, 2007.

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

[26] Oracle. *Berkeley DB Java Edition.*

[27] C. Rev, D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance, 1995.

[28] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *SOSP*, 1992.

[29] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *OSDI*, 2006.

[30] R. Sears and E. Brewer. Segment-based recovery: Write-ahead logging revisited. In *VLDB*, 2009.

[31] R. Sears, M. Callaghan, and E. Brewer. Rose: Compressed, log-structured replication. *VLDB*, 2008.

[32] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *Usenix Annual Technical Conference*, 1995.

[33] J. Sheehy and D. Smith. Bitcask, a log-structured hash table for fast key/value data. Technical report, Basho, 2010.

[34] R. Spillane, P. Shetty, E. Zadok, S. Archak, and S. Dixit. An efficient multi-tier tablet server storage architecture. In *SoCC*, 2011.

[35] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. Najjar. MicroHash: An efficient index structure for fash-based sensor devices. In *FAST*, 2005.

APPENDIX

A. PAGE SIZES ON MODERN HARDWARE

In this section we argue that modern database systems make use of excessively large page sizes. Most systems make use of a computation that balances the cost of reading a page against the search progress made by each page traversal [15].

This approach is flawed for two reasons: (1) Index pages generally fit in RAM and should be sized to reduce the size of the tree's upper levels. This is a function of key size, not the hardware. (2) Systems (but not the original analysis [15]) often conflate optimal index page sizes with optimal data page sizes. This simplifies implementations, but can severely impact performance if it leads to oversized data pages.

A.1 Index nodes generally fit in RAM

Assuming index nodes are large enough to hold many keys, we can ignore the upper levels of the tree and space wasted due to index page boundaries. Therefore, we can fit approximately $\frac{memory\ size}{key\ size + pointer\ size}$ pointers to leaf pages in memory. Each such pointer addresses approximately

$$max(page\ size, key\ size + value\ size)$$

bytes of data, giving us a read fanout of:

$$\frac{max(page\ size, key\ size + value\ size)}{key\ size + pointer\ size} \approx \frac{page\ size}{key\ size}$$

"Typical" keys are under 100 bytes, and we use 4KB pages. This yields a read fanout of 40. Prefix compression and increased data page sizes significantly improve matters.

Table 2 applies these calculations to various hardware devices, and reports the amount of RAM needed for a read amplification of one. As data becomes hotter, the devices become seek-bound instead of capacity-bound. This *decreases* memory requirements until data becomes so hot that it is more economical to store it in memory than on disk [15].

Our Bloom filters consume 1.25 bytes per key and store entries for all keys, not just those in index nodes. Four entries fit on a leaf, leading to a $4 * 1.25 = 5\%$ overhead.

A.2 Choosing page sizes when RAM is large

Since index nodes generally fit in RAM, we choose their size independently of the underlying disk. Index pages 10x the average size of keys ensure that upper levels of the tree use a small (and ignorable) fraction of RAM.

Pages that store data are subject to different page size tradeoffs than pages that store index nodes. bLSM uses a simple append-only data page format that efficiently stores records that span multiple pages and bounds the fraction of space wasted by inconveniently sized records.

We set these pages to 4KB, which is the minimum SSD transfer size. This minimizes transfer times and also improves cache behavior for workloads with poor locality: It reduces the number of cold records that are read and cached alongside hot records, increasing the average heat of the pages cached in RAM.