Design of a Write-Optimized Data Store

Hrishikesh Amur
Georgia Tech.

David G. Andersen
Carnegie Mellon University

Michael Kaminsky
Intel Labs. Pittsburgh

Karsten Schwan
Georgia Tech.

ABSTRACT

The WriteBuffer (WB) Tree is a new write-optimized data structure that can be used to implement per-node storage in unordered key-value stores. The WB Tree provides faster writes than the Log-Structured Merge (LSM) Tree that is used in many current high-performance key-value stores. It achieves this by replacing compactions in LSM Trees, which are I/O-intensive, with light-weight *spills* and *splits*, along with other techniques. By providing nearly 30× higher write performance compared to current high-performance key-value stores, while providing comparable read performance (1-2 I/Os per read using 1-2B per key of memory), the WB Tree addresses the needs of a class of increasingly popular write-intensive workloads.

1. INTRODUCTION

Handling write-heavy interactive workloads is becoming increasingly important for key-value stores. For example, at Yahoo!, typical key-value store workloads have transitioned from being 80-90% reads in 2010 to only 50% reads in 2012 [28]. In this paper, we present a new write-optimized data structure, the WriteBuffer (WB) Tree, that provides more than an order higher write performance than other state-of-the-art write-optimized stores while also supporting random access queries. For comparison, to write 64B records, the WB-Tree provides nearly 7× the write throughput of LevelDB [17] along with equal or better read performance.

Earlier key-value stores targeted two categories of workloads: low-latency workloads that were typically read-intensive, such as those enabled by memcached, and latency-insensitive workloads that allowed efficient batch insertion, such as non-realtime analytics or the earlier versions of Google's MapReduce-based indexing. However, the increasing demand for real-time results breaks these models. Twitter's real-time search makes a tweet searchable 10s after creation [7]. Google's Percolator [26] and Continuous Bulk Processing (CBP) [23] also seek to perform incremental updates to large datasets (e.g. web search indexes) efficiently (i.e., without having to run a large MapReduce job). To support such functionality, key-value storage systems must ingest incoming data at a high rate as well as allow analysis codes and/or front-ends to query this data. For this purpose, Percolator uses BigTable [8].

In previous work, two data structures are popular choices for implementing single-node data stores and databases: B+ Trees are used in systems more similar to conventional databases (e.g. BerkeleyDB [24]). The Log-Structured Merge (LSM) Tree [25], however, has emerged as the data structure of choice in single-node storage for many "NoSQL" systems. Systems in which the per-node storage is provided by an LSM variant include HBase [1], Hyperdex [13], PNUTS [10, 28], BigTable [8], and Cassandra [20].

The B+ Tree is used in when low-latency reads are required. Reads from a B+ Tree typically require a single I/O from disk (by caching the frequently-accessed higher levels in memory). Its drawback is that updates to existing keys are performed by writing the new data *in place* which leads to poor performance due to many small, random writes to disk. The LSM Tree avoids this by performing disk I/O in bulk. It organizes data into multiple, successively larger components (or levels); the first component is in-memory and the rest are on disk. When a component becomes full, data is moved to the succeeding component by performing a *compaction*. Compactions ensure that each component contains at most one copy of any key[1]. Unlike B+ Tree reads, which only check one location on disk, an LSM Tree read might check all components. By protecting components with in-memory filters [17, 28], LSM Trees can provide reads that mostly require only one disk I/O per read.

The drawback of an LSM tree is that its compactions are very I/O-intensive (§2.3). Briefly, if $M = \frac{\text{size of component } i+1}{\text{size of component } i}$, a compaction performs $2 \cdot M \cdot B$ bytes of I/O to compact B bytes of data from component i to $i+1$. Since ongoing compactions can stall writes, this can lead to low and bursty write throughput.

WB Trees are a new *unordered* data structure that make two main improvements over LSM Trees. First, WB Trees replace compactions with cheaper primitives called *spills* and *splits*. This relaxes the constraint in LSM Trees that a component can

[1] Some LSM Tree implementations (e.g. LevelDB) relax this constraint for the first disk-based component for faster inserts.

contain at most one copy of each key. This relaxation provides a significant increase in write throughput, as explained in §3.2.

The second improvement is a technique called *fast-splitting*. Compactions have two objectives: (a) they ensure that no future compaction becomes *very* expensive (because compactions can block inserts); and (b), they reclaim disk space by deleting outdated records. The nature of the compaction is such that it cannot separate the two objectives. Instead, fast-splitting allows separate mechanisms to be used for (a) and (b). Fast-splitting allows the more expensive but less critical garbage collection to run less frequently. This provides higher and less-bursty write throughput, in exchange for using additional disk space, as explained in §3.3.

At a high level, the B+ Tree, LSM Tree and WB Tree approaches can be viewed as occupying a spectrum of increasing write performance as well as increasing degrees of freedom in the location of a given record on disk. A consequence of this is that a naive implementation of random reads in WB Trees can result in very poor performance (owing to more locations to search).

As a solution, after considering various alternatives (§3.4), we adopt a technique also used by some LSM Tree implementations [17, 28] and protect each possible location using in-memory Bloom filters. During a read, the filter for each possible location is tested first, and only positive tests result in I/Os. This allows the WB Tree to use just 1-2 I/Os per read, providing similar throughput and latency as LSM-Tree reads, with a memory overhead of about 1-2B/key. Thus, when ordered access to keys is not required, the WB Tree can replace the LSM Tree because it achieves similar read latencies and throughput while offering significantly higher write throughput.

The WB Tree is *unordered*, in that it does not store keys in lexicographical order. Instead, keys are stored in order of hashes of keys. While this decision improves write performance (by avoiding expensive string comparisons) and simplifies index design, the WB Tree only supports random read queries.

More concretely, our contributions are as follows:

- We introduce a new write-optimized data structure called the WB Tree which provides up to 30× and 160× higher write performance than two popular LSM Tree implementations: LevelDB and bLSM, respectively.
- We introduce new primitives called *spills* and *splits* to replace compactions, which bias the read-write performance tradeoff towards writes compared to compactions.
- We introduce a new technique called *fast-splitting* that further improves write performance.
- We show that, as with LSM Trees, Bloom filters effectively augment the WB Tree to reduce the number of I/Os required per read to 1-2, and provide read performance equal to that of LSM Trees.

In the remainder of the paper, the WB Tree is often interchangeably used to refer to both the data structure as well as the key-value store build around the data structure. We provide background including a description of the LSM Tree in Section 2. The WB Tree design, along with write and read optimizations, is discussed in Section 3. We provide an comparisons with other single-node key-value stores along with a deep-dive into WB Tree performance in Section 4.

2. BACKGROUND

2.1 Terminology

The tradeoff between read and write performance is a recurring theme in this paper. The read and write performance of key-value stores depends, to a great extent, on read and write amplification respectively. We define write amplification as,

$$\textit{write amplification} = \frac{\textit{Total I/O performed to write record}}{\textit{size of record}}$$

B+ Tree variants and other in-place update schemes such as external hashing have a write amplification ≥ 1. However, write-optimized stores batch multiple records into a single write leading to an amortized write amplification $\ll 1$.

We use two metrics for read performance: the *worst case number of seeks* and *read amplification*. Read amplification is defined as

$$\textit{read amplification} = \frac{\textit{Total I/O performed to read record}}{\textit{size of record}}$$

Data structures such as buffer trees [3] also buffer reads allowing read amplification to be $\ll 1$. However, we are only concerned with low-latency reads for which the preferred value for the number of seeks and read amplification is 1.

2.2 Write-Optimized Key-Value Stores

In-place update data structures such as variants of the B+ Tree provide low worst-case read latency. By using high fanouts, B+ Trees require less memory as up to 99% of the data can reside in the leaves [18]. By caching the upper levels of the tree in the page cache, B+ Trees typically require a single I/O for reads. However B+ Trees, like other in-place update structures, suffer from poor write performance especially for small records.

Write-optimized data stores are gaining prominence because, as noted earlier, workloads are increasingly write-heavy, and the relatively high capacity of DRAM in modern clusters allows a greater proportion of reads to be satisfied from memory (e.g. using memcached) while writes must be persisted to disk.

Historically, log-structured systems have been used for write-heavy workloads; among these, *insert-ordered* and *key-ordered* log-structured stores may be distinguished. Insert-ordered stores, such as the Log-Structured File System (LFS) [27], FawnDS [2] etc., write data to disk immediately. These have excellent write throughput, but suffer from latency spikes due to garbage collection, have poor scan performance, and require a large amount of memory to support low-latency reads. Key-ordered log-structured stores buffer updates in memory and *sort* them before writing to disk; key-ordered log-structured stores have lower write throughput compared to insert-ordered stores [29]. Examples of key-ordered stores include Buffer Trees [3] and Log-Structured Merge (LSM) Trees [25].

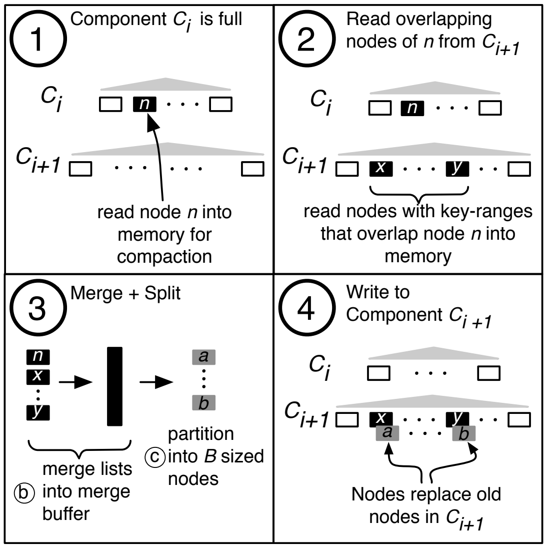
Among these, LSM Trees have typically provided a practical tradeoff between read and write performance. They provide significantly better write performance than in-place update stores, and provide random read performance comparable to B+ Trees with a modest cost in memory. Many state-of-the-art systems including BigTable [8], PNUTS [10, 28], and HBase [1] use variants of the LSM Tree, which we detail next.

Data Structure	Worst-case I/Os required (no caching) INSERT	 GET
B+ Tree	h	h
LSM Tree [25]	$\frac{2(M+1)eh}{p}$	$h \lg \frac{B}{p}$
WB Tree	$\frac{2eh}{p}$ (§3.2)	$Mh \lg \frac{B}{Mp}$ (§3.4)
Append-to-file	$\frac{e}{p}$	$\frac{Ne}{p}$

Increasing INSERT Performance

Table 1: Comparison of I/Os (assuming no caching) performed by various data structures to GET the value associated with a key or INSERT a new key-value pair: h is the height of the tree, e is the length of the record, p is the unit size of data movement between disk and memory, B is node size for WB Trees and partition size for LSM Trees, $M = \frac{\text{size of } C_{i+1}}{\text{size of } C_i}$ for LSM Trees or the fan-out for WB Trees, and N is the total number of unique keys.

2.3 LSM Trees



1
Component C_i is full
C_i
n
C_{i+1}
read node n into memory for compaction
2
Read overlapping nodes of n from C_{i+1}
C_i
n
C_{i+1}
x
y
read nodes with key-ranges that overlap node n into memory
3
Merge + Split
n
x
y
a
b
merge lists (b) into merge buffer
partition (c) into B sized nodes
4
Write to Component C_{i+1}
C_i
C_{i+1}
x
y
a
b
Nodes replace old nodes in C_{i+1}

Figure 1: Compaction in LSM Tree at component C_i: Triggered when total size of nodes in C_i exceeds a threshold.

An LSM Tree consists of multiple tree-like components C_i. The C_0 component is memory-resident and allows in-place updates whereas the remaining components reside on disk and are append-only. The components are ordered by freshness; newest data is present in C_0 and age increases with i. For specificity, we consider an LSM variant that stores each component as a B+ Tree, with the size of each tree node B bytes, and the maximum sizes of the components fixed such that $\frac{\text{size of } C_{i+1}}{\text{size of } C_i} = M$.

For writes into an LSM Tree, records are inserted into C_0 until it fills to capacity. When any component C_i fills to capacity, merges or compactions are performed between components C_i and C_{i+1} as shown in Figure 1. During the compaction, records from some range of keys, represented by node n in the figure, from C_i are read into memory along with records in the same key range from C_{i+1} and merged (steps ①-③). The newly-merged records replace the records for the key range in component C_{i+1} (step ④). Except C_0, the nodes of all other components are not modified in-place. The records are written back as new nodes to disk and the replaced nodes are garbage-collected. By avoiding in-place updates for on-disk components and only performing I/O in large, sequential chunks, LSM Trees achieve significantly better write performance than B+ Trees.

Unfortunately, the compaction operation is extremely I/O-intensive. Recall that each node is sized B; because C_{i+1} has around M times the number of nodes as C_i, the number of nodes in C_{i+1} that contain keys overlapping with keys in node n is also close to M. This means moving B bytes from C_i to C_{i+1} during a compaction requires $(M+1) \times B$ bytes to be read and the same number written back to disk after merging into C_{i+1}, resulting in high write amplification as shown next.

An inserted record moves progressively from $C_0, C_1, \ldots$ to C_{h-1}, where h is the number of components or "height" of the tree; therefore, the total I/O performed for the record is the sum of the I/O performed at each component. During a compaction from C_{i-1} to C_i, the total I/O performed is $(M+1) \times B$ bytes read plus an equal amount written. This is amortized across the B/e records in the node being compacted where e is the size of each record. This yields a per-record I/O cost of $(M+1) \times e$ bytes read plus written. From §2.1,

$$\textit{write amplification} \leq \frac{1}{e} \sum_{i=1}^{h} \frac{2(M+1)B}{B/e} = 2h(M+1)$$

During an LSM Tree GET, the in-memory component C_0 is first searched (recall that the age of records in component C_i increases with i). If the queried key is not found in C_0, then the disk-based components $C_1, C_2, \ldots$ are progressively searched. As a result of the compaction process, the LSM Tree maintains the invariant that there is at most one record for any key in each component. This leads to a worst-case read cost of h seeks. Nodes can be protected with Bloom filters to avoid wasteful I/Os for a modest memory cost [17, 28]. Table 1 shows the I/O requirements (without the page cache) of some data structures.

In this paper, for discussion, we consider an LSM Tree with partitioned, exponentially-sized levels such as used in LevelDB [17]. We also evaluate a second variant without partitioning in Section 4.

3. WRITE BUFFER TREE

This section introduces the WB Tree, an *unordered* key-value store optimized for high insert performance while maintaining fast random read access. The tree's key novel elements are its replacement of performing compactions with cheaper spills and splits realized via mechanisms described in §3.2. Further, fast-splitting substantially improves write performance, as explained in §3.3. The WB Tree includes indexes that limit read amplification to 1-2 I/Os per read (§3.4). Finally, garbage collection (§3.5), logging and recovery are discussed (§3.6).

3.1 Overview of the basic WB Tree

The WB Tree exports a simple API: INSERT(key, value), for both insertion and update, GET(key), and DELETE(key). It also supports bulk insertion and deletion for high throughput.

The WB Tree maintains a single root node. The root node is unique in that it consists simply of a single in-memory buffer of size B. Non-root nodes are divided into *leaf* nodes and *internal* nodes. Each non-root node contains one or more *lists* of sorted records on disk; a *list* is similar to an SSTable [8], but does not contain any indexing information. The total size of the lists in a node must be less than or equal to its capacity, B.

The WB Tree uses high fan-out, which means that a large proportion of nodes are leaf nodes. For example, for a fan-out of 256, around 99.2% of the nodes are leaf nodes[2]. A high fan-out helps reduce write amplification by reducing the height of the tree. The write amplification depends on the height of the tree, because as records progress down the tree, they are read and written at each level.

As mentioned before, the WB Tree uses hashes of the actual keys to route records within the tree, i.e. the hash of the key in each record decides the location of the record in the tree. Therefore, the sub-tree rooted at each node is responsible for a subset of the hash-space. Each node's hash-space is partitioned between the children of the node (i.e. hash-spaces for sibling nodes *do not* overlap).

An empty WB Tree consists of only the root node. To `INSERT` a key-value pair (record) into the WB Tree, the tuple ⟨`hash, size, record`⟩ is appended to the memory buffer of the root node; *hash* is a hash of the key, and *size* is the size of the record. To `DELETE` a key is to actually `INSERT` the key with a special tombstone value τ, i.e., `DELETE(k) = INSERT(k,`τ`)`. Discussion of the `GET` operation is deferred until §3.4.

The two primitives in the WB Tree are *spills* and *splits*. Recall that, in an LSM Tree, when the total size of a component exceeds a threshold, data is *compacted* from that component to the next. In the WB Tree, when the total size of any non-leaf node reaches its capacity of B, it undergoes a *spill*. This operation performs a similar function as the compaction, in that it moves inserted data progressively down the tree. However, it differs in that it simply appends the spilled data to the nodes in the next level of the tree and performs no reading or merging of data that a compaction does. In this manner, a spill performs substantially less I/O than a compaction.

Leaf nodes, instead, *split* when full. Conceptually, a split converts a full leaf node into two half-full leaf nodes, making room to receive further spills from the parent node.

The strict enforcement of a spill or a split when a node reaches its capacity ensures non-bursty insert performance. A long-running spill or split can block insertions, similar to how compactions can block insertions in an LSM Tree. Before we discuss mechanisms for spills and splits (§3.2), we describe collapsing, which is used in both operations.

Collapsing Given a sorted list with buffered operations, the list can be collapsed by replacing multiple operations on the same key with a single operation. For example, a `DELETE` appearing after an `INSERT` can be replaced by a `DELETE`, or two `INSERT`s for the same key can be replaced by the later one. Collapsing is important as it allows disk space to be reclaimed from outdated records. Collapsing is intra-list garbage collection; we introduce a new term to differentiate it from the garbage collection of entire lists (§3.5).

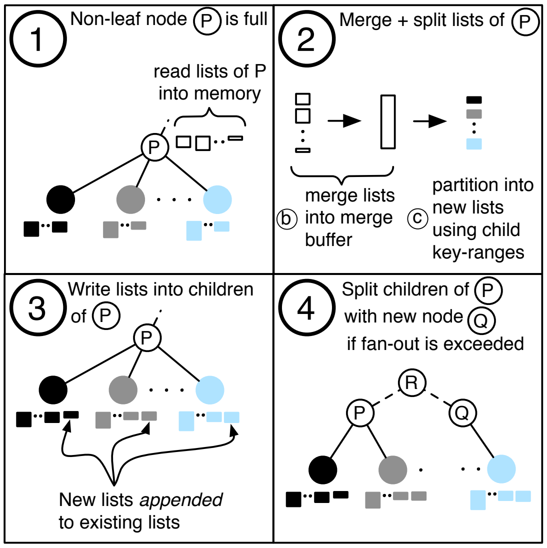

1
Non-leaf node P is full
read lists of P into memory
P
2
Merge + split lists of P
b merge lists into merge buffer
c partition into new lists using child key-ranges
3
Write lists into children of P
P
New lists *appended* to existing lists
4
Split children of P with new node Q if fan-out is exceeded
R
P
Q

Figure 2: Spilling in a WB Tree at Internal Node P: Triggered when total size of lists in Node P exceeds node size.

Collapsing can be performed in a single pass over the list because the list is sorted by hash. However, due to hash collisions, it is possible for colliding keys to be interleaved in the list (e.g. for different keys a and b, the following order might occur: `{`h_a`,INSERT(a,1)}, {`h_b`,INSERT(b,3)}, {`h_a`,DELETE(a)}` where $h_a = h_b$). These are handled using a separate hashtable for colliding keys whenever a collision is detected. A stable sorting algorithm (we use a fast radix sort because integer hashes are being sorted) ensures that insertion order is preserved for records with the same key.

3.2 Spills and Splits

3.2.1 *On Spilling*

The spill procedure works differently for root nodes and internal nodes. For the root node, the root buffer is, first, sorted and collapsed. Then, the root node spills into its children by partitioning the in-memory buffer according to the hash-spaces handled by each of its children, and writes each partition as a new list to a child node. If the root is the only node in the tree, it is a leaf and would be split not spilled.

An internal node may contain many lists – one from each spill of its parent. A spill of node P is depicted in Figure 2. The lists of node P (already sorted) are read into memory (step ①). In step ②, the lists are merged into a merge buffer, collapsing the list during the merge. After the merge, the contents of the merge buffer are partitioned according to the key-ranges of P's children and written as new lists in the children in step ③.

[2] Let there be l leaf nodes and let fan-out be $f = 256$, then there are at least $l/128$ nodes in the level above the leaf nodes, $l/128^2$ in the level above that and so on until a single root node. The total number of nodes is therefore $N = l + \lceil l/128 \rceil + \lceil l/128^2 \rceil + \cdots + 1 \leq l \times \frac{1}{1-1/128}$. Therefore, $l/N \geq 0.9921$, i.e. more than 99.2% of the nodes are leaves

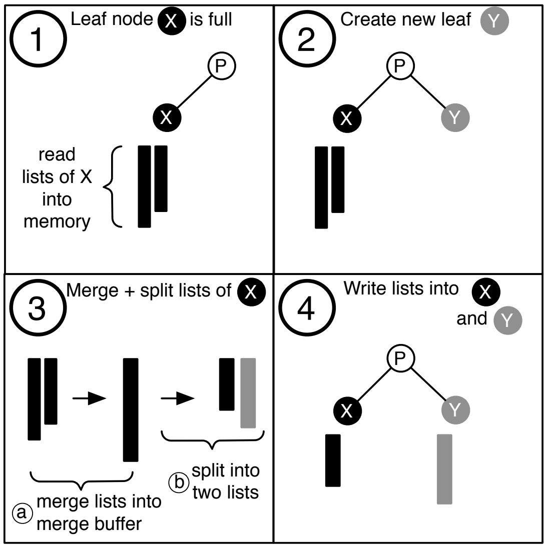


Figure 3: Slow-split operation for leaf node X.

After each spill, if the number of children of P is greater (see §3.2.2 for when the number of children of a node might increase) than the maximum fan-out allowed, then a new node Q is created and half of P's children are transferred to Q. Node Q is then added as a child to P's parent. If P is the root, then a new root is created, increasing the height of the tree by one (step ④). Node P is empty after a spill, so there is no data buffered in P that requires splitting.

3.2.2 On Splitting

For a leaf node, when the total size of all lists reaches the capacity, the node undergoes a split to form a new leaf node, just like a B+ Tree. The new leaf is added as a child to the parent of the split leaf.

One way to perform the split is to read the on-disk lists of the leaf into memory, merge them into a merge buffer, and split the merge buffer into two lists. One list replaces the list being split in the current leaf and the second is added to the new leaf. Both lists are written to disk. This approach, termed a *slow-split*, is shown in Figure 3.

3.2.3 Spills and Splits Replace Compactions

Compactions in an LSM Tree and spills in the WB Tree perform the same function, i.e., moving B bytes of data from one level to the next. Compaction, however, requires significantly more I/O. As explained in §2.3, during each compaction, to move B bytes from component C_i to C_{i+1}, requires reading $(M+1) \times B$ bytes and writing $(M+1) \times B$ bytes, where $M = \frac{\text{size of } C_{i+1}}{\text{size of } C_i}$. In contrast, both a spill and a split read only B bytes to fetch all the lists from a node into memory and write back B bytes for the new lists in the split leaves (slow-split). Thus, for the WB Tree,

$$\textit{write amplification} \leq \frac{1}{e} \sum_{i=1}^{h} \frac{2B}{B/e} = 2h$$

The WB Tree performs a factor of $(M+1)$ less I/O per write. In practice, this replacement yields a 6× improvement in `INSERT` throughput over LevelDB (§4.3).

3.3 Fast-Splits

As noted earlier, the WB Tree uses a large fan-out to reduce write amplification, leading to a large proportion of nodes being leaves. Because leaf nodes undergo splits and not spills, improving the performance of splits is crucial.

Slow-splits, while cheaper than compactions, still incur significant I/O (B bytes to read the lists and B bytes to write back the split halves of the merged list). We provide a simple solution to speed up splitting – *fast-splits*. The basic idea of a fast-split is to avoid reading the leaf from disk; instead, the splitting offset in each list is marked and stored in the new leaf.

Mechanism Figure 4 illustrates the operation of a fast-split on Node X, which is a full leaf node. The median hash in a random list from X is selected as the separator, which is used to split each list in X. Because each list is sorted, finding the median is fast (logarithmic number of seeks) using binary search. Each list in the node is partitioned using the separator value, and one partition assigned to the new leaf (step ③). Unlike slow-split, this process avoids bulk data movement and is fast. When X's parent node P spills for the next time, the new lists are written into the correct nodes (step ④).

Further splits of X are handled similarly. Steps ⑥-⑧ show the creation of new leaf Z. Various heuristics can be used to decide how many fast-splits to perform on a node before a slow-split is required. Next, we provide some intuition behind why fast-splitting works.

Intuition Consider what a slow-split accomplishes: (1) when a leaf reaches size B, it converts the multiple lists in the leaf to a single list and collapses the list; and (2) it splits the list into two parts of equal size and assigns one part each to the split leaves. The latter effect crucially allows the parent node to continue spilling into the newly-created leaves. The former effect frees disk space by deleting outdated versions of records during collapsing.

Fast-splitting splits a leaf into two without merging its lists, i.e., it splits the leaf and ensures that insertions do not block, but avoids the more expensive task of reclaiming disk space occupied by outdated records. Instead of the $2B$ bytes of I/O for a slow-split, a fast-split requires only $O(\log B)$ random I/Os. As explained briefly in Section 1, fast-splits provide improved write performance by trading off extra disk space. The outcome is superior, consistent and non-bursty write performance.

The advantages of fast-splitting are subtle. It may seem that slow-splitting less frequently might yield the same benefits, but that is not the case. For example, suppose that leaf nodes were 4× the size of other nodes, i.e., $4B$; this would result in 4× fewer slow-splits of leaves. However, each split would now cost $8B$ bytes of I/O ($4B$ to read and $4B$ to write), which is 4× the cost of slow-splitting a B-sized leaf. Table 2 shows the I/O performed by slow-splits, slow-splits with larger leaves, and fast-splits (4 fast-splits per slow-split). Starting with just 1 leaf, the table shows how each scheme causes leaves to split as data is spilled into the leaves. Using fast-splits performs the least

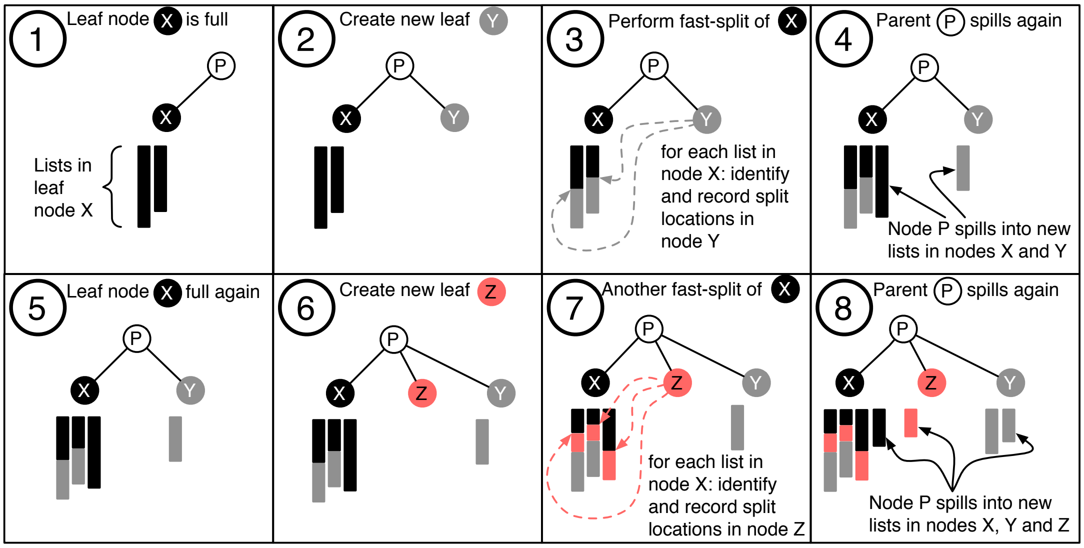

1
Leaf node X is full
P
X
Lists in leaf node X
2
Create new leaf Y
P
X
Y
3
Perform fast-split of X
P
X
Y
for each list in node X: identify and record split locations in node Y
4
Parent P spills again
P
X
Y
Node P spills into new lists in nodes X and Y
5
Leaf node X full again
P
X
Y
6
Create new leaf Z
P
X
Z
Y
7
Another fast-split of X
P
X
Z
Y
for each list in node X: identify and record split locations in node Z
8
Parent P spills again
P
X
Z
Y
Node P spills into new lists in nodes X, Y and Z

Figure 4: Fast-split operation: The figure shows leaf node X splitting two times (steps 3 and 7). There is no data copied during fast-splits, and newly-created leaf nodes only point to the offsets of the partitions in the original files.

Data	slow-split		slow-split (4)		fast-split (4:1)	
spilled	Leaves	I/O	Leaves	I/O	Leaves	I/O
B	$1 \rightarrow 2$	$2B$	1	–	$1 \rightarrow 2$	$\log B$
$+B$	$2 \rightarrow 4$	$4B$	1	–	$2 \rightarrow 4$	$2 \log B$
$+2B$	$4 \rightarrow 8$	$8B$	$1 \rightarrow 2$	$8B$	$4 \rightarrow 8$	$4 \log B$
$+4B$	$8 \rightarrow 16$	$16B$	$2 \rightarrow 4$	$16B$	$8 \rightarrow 16$	$8 \log B$
$+8B$	$16 \rightarrow 32$	$32B$	$4 \rightarrow 8$	$32B$	$16 \rightarrow 32$	$32B$
Total						
$16B$	$63B$		$56B$		$32B$	

Table 2: Comparison of splitting schemes: Fast-splits perform the least I/O while splitting.

amount of I/O. Using larger leaves for slow-splits requires less I/O, but this causes bursty write performance as INSERTs can block, waiting for a large leaf to slow-split.

Along with a favorable choice of system parameters, these factors contribute to a 30× improvement in write performance over LevelDB (§4.3). These performance improvements are not free. Compactions perform worse because they enforce the constraint that each component in the LSM Tree, analogous to a level in the WB Tree, can have only one record per key. This bounds the number of possible locations for the key when performing a GET. In the WB Tree, this constraint is relaxed, and can lead to significantly higher I/O during GETs. Next, we explain how we limit read amplification to achieve read performance in WB Trees that is comparable to that of other key-value stores.

3.4 Higher Read Performance with Indexes

We measure read performance using two metrics: (1) worst case number of seeks; and (2) read amplification, which is the total amount of I/O performed to read a record divided by the size of the record. Ideally we should perform reads with a single seek and read amplification of 1 (if record size is less than page size).

Recall from §2.3 that in an LSM Tree, a GET is performed by successively checking components $C_0, C_1, \ldots, C_{h-1}$, where h is the number of components. Because each component contains at most one version of a key, a GET requires at most h seeks in an LSM Tree with no additional indexes.

By contrast, in the WB Tree, each node can contain more than one record per key (a key can occur potentially in each list in a node). In a WB Tree with no indexes, a GET is, therefore, performed by starting from the root and proceeding to search along some root-leaf path in the tree (the specific path is dependent on the key). Then, in each node, starting from the last-added list, all the lists have to be searched. Figure 5 shows an unindexed GET: In step ①, the root node is searched and if the record is found, it is returned. If not, step ② is invoked recursively until a leaf is reached or the queried key is found. Because lists of all nodes (except the root) are maintained on disk, each GET can result in an unacceptably large number of I/Os. Clearly, to achieve our goal of a single I/O per GET, an in-memory index is required that maps each key to the list in the tree that contains the most recent version of the key.

3.4.1 *Index for List Selection*

The desired properties of such an index are: (a) it must map a key to the list in the tree that contains the latest record for the key; (b) it must be fast to construct and update, because each spill or split causes multiple updates to the index, and slowing down spills or splits can block insertions; (c) it must be fast to query, because each GET potentially queries the index multiple times; and (d) it must be memory-efficient, because a greater number of keys can be indexed in memory, boosting GET performance. Design alternatives for such an index are described next and summarized in Table 3.

Property	Full-Tree	Per-Node		Per-List
	Hashtable		Perfect Hashing	Bloom filters
Per-key Memory	O(size of key)	O(size of key)	$O(1)$	$O(1)$
Dynamic	Yes	Yes	Requires keys to recompute hash function	Yes
Frequency of updates	For each INSERT, spill and split	For each spill and split		
Absent keys return "not found"	Yes	Yes	No	Likely

Table 3: Design alternatives for List-Selection Index: a Bloom filter satisfies all requirements; it needs 10 bits/key for a 1% false-positive rate, it is dynamic and fast to build.

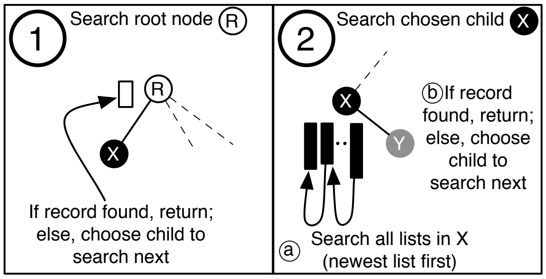

1
Search root node R
R
X
If record found, return; else, choose child to search next
2
Search chosen child X
X
Y
b If record found, return; else, choose child to search next
a Search all lists in X (newest list first)

Figure 5: GET **operation in an *unindexed* WB Tree.**

Full-tree index A straightforward design maintains a dictionary data structure (e.g., hashtable) that maps each key in the tree to the list that contains the most recent record for the key. Unfortunately, this index would require updating not only for each spill and split, but also for each INSERT and DELETE. GETs would have to synchronize with updates to access the index which results in high synchronization overhead.

Per-node indexes reduce synchronization overhead. An index can be maintained by each node that stores the list in the node that contains the latest record for a key for every key in the node. This index only requires updating when a new list is spilled to the node or during a split or spill of the node. Because a node's (hashed) key-space is partitioned among its children, the records for any key can only be contained by nodes on some particular root-leaf path in the tree. In this scheme, a GET request would, therefore, have to check the per-node index of each node that occurs on this path.

Hashtables can be used to implement the index, but hashtables are space-inefficient. First, in order to check for collisions, hashtables typically maintain the entire key or a digest (e.g. SHA-1 hash) in memory. Second, closed hashtables (e.g. probing for collision resolution) typically avoid filling up the buckets to capacity to maintain performance, whereas open hashtables (e.g. chaining-based) use extra space for chaining-related data structures. This problem of having to store the key in memory also affects other dictionary data structures such as red-black trees, skip-lists, etc.

Perfect hashing offers a potential solution to having to store keys in memory. Perfect hashing maps the elements from a set S of size n elements to a set of integers with no collisions. Minimal perfect hashing further constrains the size of the set of integers to n. The advantage of this idea is that, because there are no collisions, keys can be stored on disk (instead of memory). However, in order to make a (minimal) perfect hashing scheme dynamic, i.e., supporting insertions and deletions, parts of the hashtable may require rehashing in case of collisions due to newly inserted keys [12]. Rehashing requires keys, which have to be read back into memory using multiple random I/Os. This proves unsustainable for insert-heavy workloads.

Per-list index Another alternative is to extend the idea of a per-node index to a per-list index. Instead of using an index that maps a key to a list, a per-list index stores membership information only. A GET would check *all* lists along some root-leaf path in order of age. Recall that each key (more precisely, its hash) maps to some specific root-leaf path in the tree.

Owing to the deficiencies of the first two alternatives, the WB Tree opts for a per-list index. For membership, a per-list index must implement a set data structure for the keys in the list. The data structure must be memory-efficient, without requiring entire keys to be stored in memory. Additionally, it must be fast to construct and query.

Bitmaps using hashes can potentially be used, but become inefficient for lists that sparsely populate the hash-space. Compressed bitmaps are efficient for sparse lists and provide fast queries, but are slow to build. The WB Tree, instead, opts for Bloom filters. Bloom filters are compact (10 bits/key), and fast to build and query. The tradeoff is that a Bloom filter can yield a small percentage of false positives (but never false negatives).

3.4.2 Index for List Offsets

Having explained the design of a List-Selection index to select the list that contains the most recent record for a key, we now explain how to find the record within the list itself.

Given that each list in the WB Tree can be large, to maintain low read amplification, for each list, a *List-Offset* index determines the offset within a list at which the record is located. Recall that lists in the WB Tree are sorted by hashes of the keys. This makes it possible to maintain a simple index, called the *first-hash-index* which stores in memory the hash of the key of the first record in each page (e.g., 4kB) of the list.

Searching for a key in a list then proceeds by: Binary search the in-memory first-hash-index to find the page that contains the queried key. Read this page into memory and search sequentially until the target key is found or the end of the page is reached. Binary-searching within the page is not possible because the size of the record is stored with the record on disk.

3.5 On Garbage collection

Each list in the WB Tree is backed by a separate file on disk. For an internal node, the files that back lists in that node can be deleted after the node has completed spilling. In the case of leaves that undergo slow-split, the files that back the original lists of the leaf can safely be deleted after the new lists, created by the slow-split, have been written to disk. Fast-splitting complicates garbage collection: the file containing the original list is now pointed to by multiple leaf nodes. We solve this problem by maintaining ref-counts for each file and deleting the file only after no lists reference it any longer.

3.6 Logging and Recovery

The WB Tree uses a write-ahead log to write all `INSERT` and `DELETE` operations to disk before successfully returning to the client. The system also supports synchronous operation, in which `fsync()` is invoked on the log before returning from an `INSERT` or `DELETE` operation. When the root node spills to its children (or splits, forming a new root), its contents are written as lists in the children nodes; the log can be cleared after this. Recovery consists of replaying the contents of the log.

4. EVALUATION

This section compares the performance of the WB Tree with two LSM Tree variants: LevelDB and bLSM. bLSM differs from LevelDB in that it uses only three components in its LSM Tree and allows the overlap factor, M, to vary. Further, while LevelDB seeks to use small partitions to reduce the worst-case compaction time, bLSM opts not to use partitioning and instead relies on a more sophisticated compaction scheduler.

The experiments use a 12-core server (two 2.66GHz six-core Intel X5650 processors) with 12GB of DDR3 RAM. The disk used is an Intel 520 SSD. We report the median of three runs of each experiment. We use the `jemalloc` [14] memory allocator for all experiments. We use bulk interfaces for insertion. Write-ahead logging is enabled for all systems (`fsync()` is invoked on the log file before returning to the client). The Yahoo! Cloud Serving Benchmark (YCSB) [11] tool is used to generate workload traces, which are replayed in a light-weight workload generator. The dataset used is denoted as (U,R,e,S) where U is the number of unique keys, R is the average number of repeats for each key, e is the record size and S is the total size of the dataset. We use uniformly distributed data for experiments.

4.1 Tuning

WB Tree We use a fan-out of 256 and node size of 600MB. As explained in §4.3, using relatively high fan-outs and node sizes favors `INSERT` performance.

LevelDB We found that allocating memory to the page cache (instead of a special LevelDB write buffer) improves insert performance. Compression is turned off. The creation of Bloom filters is enabled for fast reads. We use the default values for the overlap factor $M = 10$ and partition size (2MB); the heat-map in Figure 9b shows that, unlike the WB Tree, relatively small values for these parameters provide better insert performance.

bLSM For bLSM, insert performance improves if a large in-memory component C_0 is used [28]. We allocate 6GB of memory to C_0 and the remaining to the buffer cache.

4.2 Full System Benchmarks

Figure 6 shows the throughput of each key-value store. The datasets used are: D_1: $(2\times10^9$, 2, 16B, 42GB), D_2: $(5\times10^8$, 8, 64B, 42GB), and D_3: $(10^8$, 2, 256B, 48GB) for 16B, 64B and 256B records respectively. Figure 7 compares the key-value stores in terms of memory use and performance of negative `GET`s. Four important observations stand out:

- `INSERT` throughput in the WB tree is nearly 30× and 160× faster than LevelDB and bLSM for small (16B) records. For 64B records, the improvements are 6.6× and 14× respectively. For 256B records, the improvement are 1.5× and 3.3× respectively.
- As the record size increases, the number of `INSERT`s per second achieved by the WB Tree drops as expected. The net amount of data written (= `INSERT`s/sec. × record size) actually increases from 45MB/s to 65MB/s. The `INSERT`s/sec. remains almost constant for both LevelDB and bLSM, which seems to indicate that some other process (e.g., compactions or locking), rather than disk bandwidth, is the bottleneck.
- The WB Tree offers equal or slightly higher `GET` throughput than LevelDB and bLSM, with similar `GET` latencies.
- For the 50%-`INSERT` workload, the WB Tree and LevelDB perform similarly. The runtime for this workload is dominated by the `GET` operations.

Figure 7 shows the memory use of the different systems along with negative `GET` throughput and latency on dataset D_2. LevelDB and WB Tree require about 1.5B per key; bLSM requires about 3B. The figure also shows throughput and latency for `GET` requests for absent keys. LevelDB's latency is lower because it checks fewer Bloom filters. bLSM uses *unpartitioned* components, so for positive Bloom filter tests, it must sequentially search a part of the component for the key. For negative `GET`s, this can be expensive. While the mean latency is just 0.2ms (not shown), 95th percentile latency is high.

4.3 Write Performance

In this section of the evaluation, we demonstrate the effects of various optimizations on write performance. Figure 8 compares the WB Tree and LevelDB write performance. LevelDB uses default settings for file size (2MB) and overlap factor (10). The Baseline WB Tree uses a node size of 2MB and a fanout of 10. Using spills and splits instead of compactions allows a 6× performance improvement.

Write performance depends on keeping write amplification low. For the WB Tree, reducing the height of the tree reduces write amplification. Figure 8 shows that a careful choice of WB Tree parameters significantly increases write performance.

Figure 9a shows that, generally, increasing the fan-out increases throughput. This is because a high fan-out decreases

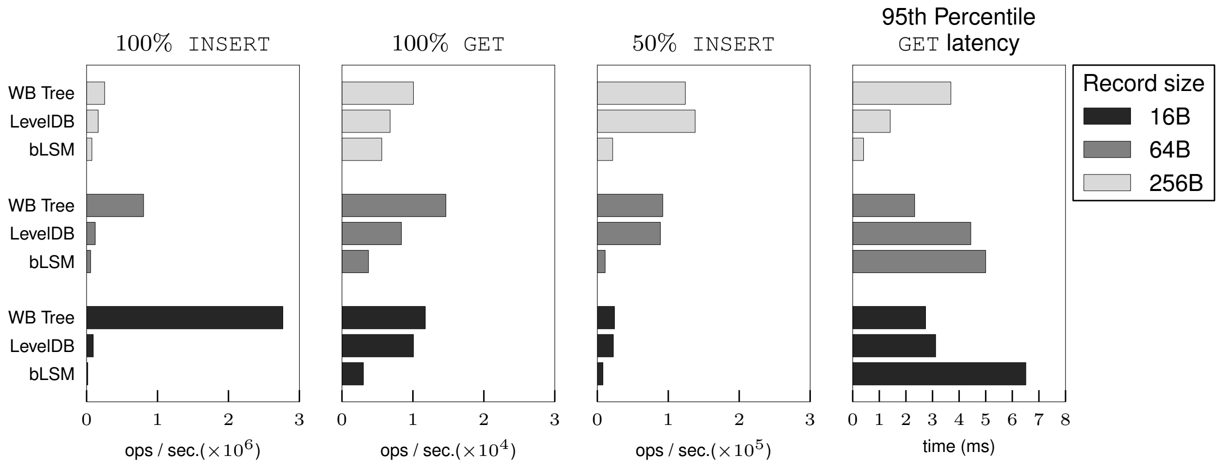


Figure 6: Comparison of the WB Tree with various other key-value stores: The WB Tree has nearly 30× and 160× higher write throughput than single instances of LevelDB and bLSM respectively; for large records, the WB Tree is 2× faster. The read throughput and latency of the WB Tree are similar to other systems.

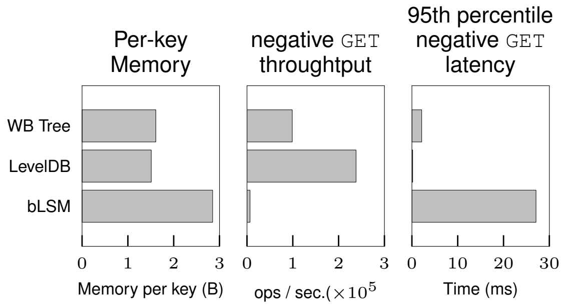


Figure 7: Memory use and negative `GET` performance

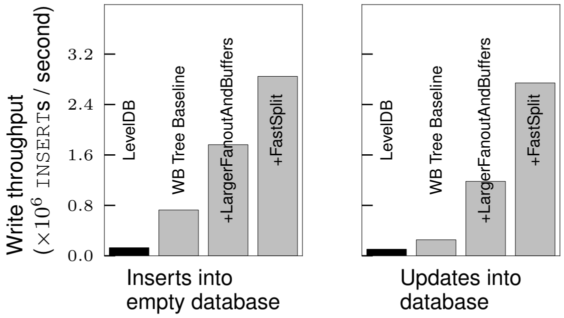


Figure 8: Write performance contributions: Large fan-out/node sizes and fast splitting are both significant.

the height of the tree which leads to lower write amplification. An exception is small node sizes, where high fan-outs lead to mostly small, random writes. Increasing the node size also decreases the height of the tree and generally improves write performance. However, extremely large nodes lead to bursty write performance (not shown).

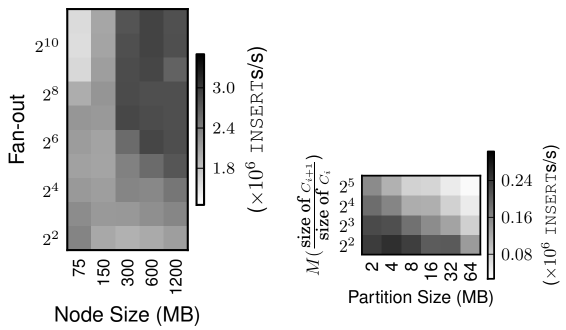
Crucially, the same improvements cannot be applied to LSM Trees. Recall from §2.3 that the upper bound on write amplification for LSM Trees is $2h(M+1)$ where h is the height of the tree and $M = \frac{\text{size of } C_{i+1}}{\text{size of } C_i}$, is the overlap factor. Increasing the overlap factor reduces the height of the tree, but leads to poorer write performance due to more expensive compactions as shown in Figure 9b. Therefore, using relatively small M and partition sizes works best for LSM Trees.

Figure 8 shows that fast-splitting yields a further improvement of nearly 2× in both cases. One possible heuristic to decide when to fast-split is to use a fixed fast-split / slow-split ratio. Figure 10 shows that the write throughput of the WB Tree increases with increasing values of this ratio.

4.4 Read Performance

To understand `GET` performance, Figure 11 shows heat-maps for `GET` throughput, 95th percentile `GET` latency and I/Os per `GET`. The following observations can be made:

- Figure 11a and 11b show that the fan-out-node-size combination that maximizes `GET` throughput and minimizes latency is the same, viz. small values of fan-out and node-sizes. Also, the combination that works best for reads, unfortunately, minimizes write throughput (Figure 9a).
- The trend is partially explained by the heat-map in Figure 11c which shows the number of I/Os per `GET`. Large fan-outs and small node sizes increase the number of lists per node, which, in turn, incurs more false-positives from the Bloom filters protecting the lists. False positives cause wasted I/Os leading to lower `GET` throughput and higher latencies. An exception is low fan-out with large node sizes: this incurs few I/Os per `GET`, but the large node sizes increase latency from cache misses while searching the List-Offset index.



(a) WB Tree: The upper bound on write amplification is $O(h)$; h is decreased by large fan-outs and node-sizes.

(b) LSM Tree: The upper bound on write amplification is $O(hM)$; h decreases logarithmically with M, favoring small M.

Figure 9: Effect on INSERT performance by parameters.

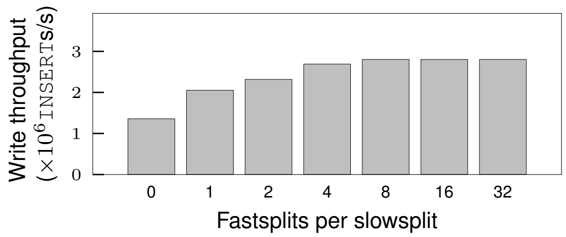


Figure 10: Write throughput increases with an increase in the fastsplit / slowsplit ratio.

GET performance depends primarily on the number of I/Os performed per GET operation. The number of I/Os performed depends on the total number of lists that have to be checked.

4.5 Summary

When to use WB Trees The WB Tree's INSERT throughput for small records is $5-30\times$ higher than LevelDB's, and it has slightly better GET throughput with similar latency. The memory cost is a modest 1-2B per key. For large records, the WB Tree provides a more modest improvement of $1.5-2\times$ higher INSERT throughput.

When to use LSM Trees If a key-ordered store is needed. Also, if memory is insufficient for List-Selection and List-Offset indexes, an LSM Tree will provide higher GET throughput.

How to use the WB Tree Figure 9 and Figure 11 show that system parameters trade between INSERT and GET performance. For high INSERT performance, high fan-out and large node sizes must be used; For high GET performance, small fan-out and small node sizes must be used. A fast-split / slow-split ratio of 8 or 16 can be used, as higher values can lead to excessive lists in each node which degrades GET performance.

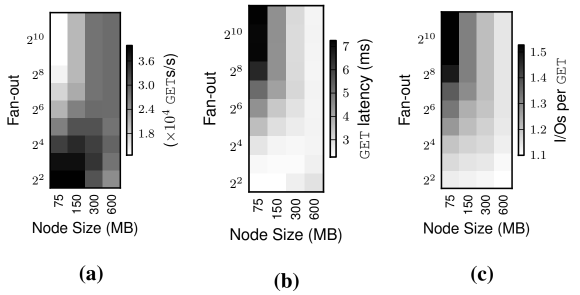


Figure 11: GET performance is helped by low fan-outs and relatively small node sizes.

5. RELATED WORK

5.1 Write-Optimized Stores

There is substantial previous work that has sought to improve upon the write throughput of systems that use in-place update indexes such as the B+tree [24], and Extendible hashing [15].

The Log-Structured Merge (LSM) Tree [25] is a data structure that uses multiple disk-based components of increasing size to buffer updates and progressively move data down the tree using efficient bulk I/O. It incorporates the general technique of using exponentially-sized components proposed by Bentley [6] to make static data structures dynamic with only logarithmic increases in query and insertion time. Many real-world systems including BigTable [8], bLSM [28] are variants of the LSM tree. FD-Trees are LSM Trees that optimize for SSDs [21]. Many implementations of LSM Trees (e.g., LevelDB) also use partitioning [19] as a means to limit compaction activity to heavily-written key-ranges for inputs non-uniformly distributed over the key-space.

The WB Tree replaces I/O-intensive compaction operations in LSM Trees with cheaper spills and splits that allow significantly faster inserts, and extends the idea of relaxing the number of possible locations for a record with fast-splitting.

The Sorted Array Merge Tree (SAMT) used in Cassandra and GTSSL [30] is the closest to the WB Tree. The SAMT uses exponentially-sized levels and, similar to WB Trees, writes *multiple* possibly-overlapping ranges from one component to the next before having to perform a compaction. GTSSL develops techniques to adapt to changing read-write ratios and adapting to hybrid disk-flash systems.

These improvements are orthogonal to the ones discussed in this paper and the WB Tree focuses on offering better write performance through fast-splitting.

There are many other write-optimized schemes we are unable to cover in detail. The log-structured file system (LFS) introduced many of the ideas used in write-optimized systems. The Buffer Tree [3] offers excellent write and read throughput if good amortized read performance is sufficient.

5.2 Read Performance

With respect to read performance, LSM Trees require each component to be checked for a read. To improve performance, datastores typically (a) cache frequently accessed data in memory, (b) protect components with Bloom filter to prevent wasteful accesses (e.g. LevelDB, Cassandra, bLSM), and (c) use fractional cascading [9], where partial results from searching one component are used to speed up searching following components (e.g. Cache-Oblivious Lookahead Arrays (COLA) [5]).

While WB Trees benefit from caching of frequently-accessed data in memory, caching is not the focus of this paper. WB Trees adopt the use of Bloom filters to protect each list in every node. In addition, WB Trees use additional indexes to store the offset of a record within a list, since lists can be quite large.

The List-Selection index (§3.4.1) maps each key in the WB Tree to the level in the tree that stores the key. This index can be implemented using hashtables, but even a memory-efficient hashtable like Sparsehash [16] is space-inefficient for this function as it has to store entire keys in memory (for collision resolution). SILT [22] includes an immutable index that uses minimal perfect hashing that maps n keys to the consecutive integers $0 \dots n-1$ with no collisions; this does not require the keys to be present in memory for non-mutating accesses. However, for dynamic perfect hashing, keys are required to be present in memory to allow rehashing parts of the hashtable in case of inserts that may cause collisions [12]. The WB Tree, instead, constructs the List-Selection index by using Bloom filters to protect each list.

6. DISCUSSION

Providing ordered access. While many systems require only per-object retrieval, many also benefit from the range query support provided by an ordered store. While the fundamental notion of spills and splits applies naturally to both ordered and unordered stores, extending the design of the WB tree to support ordered access is important future work that will require non-trivial engineering to do well while preserving the structure's high performance.

Bounding worst-case memory per key. Being parsimonious with memory is particularly important when dealing with many small key/value pairs. Here we consider two possible scenarios where the memory used per key can become amplified. We show that the problem is non-existent in the first scenario and provide a solution for the second.

For records smaller than the page size the memory used per key is due, predominantly, to the Bloom filters used in the List-Selection index (the List-Offset index uses only 8 bytes per (4kB) page of records). For each key in a list, the Bloom filter protecting the list requires about 10 bits for a 1% false positive ratio. If a key appears in multiple lists in the WB Tree, then the memory used for that key would be 10 bits *per list*. This *amplification* of memory can occur in two scenarios: (a) repeats in different nodes in the tree, and (b) repeats in different lists within a single node. We consider each case separately.

In the former case, the problem of memory amplification does not arise. For ease of analysis, suppose that copies of all keys in the WB Tree are present in the leaf level (i.e., level 0). As shown by the analysis in Section 3.1, with large fan-outs (e.g., 256), the proportion of leaf nodes in the tree is close to 1. This means that the memory used by the non-leaf nodes is small (less than 1%) of that of the leaf nodes.

In the latter case, if the number of lists in a node is l, a key could occur in each of l lists amplifying the memory used by l. To solve this problem, we need an estimate of $\alpha = \frac{\text{number of unique keys in node}}{\text{total number of keys in node}}$. The ratio α provides a measure of memory amplification in the node. If α is 1, then no keys repeat; if $\alpha = \frac{1}{l}$, then all keys repeat l times. If an estimate of α can be maintained, then memory amplification can be bounded by forcing a spill or slow-split on the node when α reaches the desired threshold. To bound the memory amplification to 2, for example, a node is spilled or slow-split whenever α becomes less than 1/2. Next, we provide a method for estimating α.

Suppose that $L_1, L_2, \dots, L_l$ are sets containing the keys in each list; the number of unique keys in the node is $|L_1 \cup L_2 \cup \dots \cup L_l|$. Computing the union of the lists is difficult, because when the list corresponding to set L_i is spilled from the parent, all older lists in the node have already been written to disk. To solve this, we propose the use of *cardinality estimators*. K-Minimum Values (KMV) [4] is a cardinality estimator that can estimate the cardinality of a list of elements by inspecting a small fraction of the list. Assuming that a hash function exists that uniformly distributes the elements of the list, intuitively, if there are n elements in the list, the average spacing between the hash values would be $1/n$-th of the range of hash values. KMV uses this idea to maintain a digest of the k-smallest hash values seen in the list; the average distance between these k-successive hashes yields an estimate of the cardinality of the list. For a union of m lists, the digest of each list can simply be merged and truncated to k to obtain a digest for the union. This solution works well for WB Tree lists because the sorted hashes of all keys in a list are already available.

7. CONCLUSION

This paper presents the WriteBuffer (WB) Tree, a new data structure that forms the basis of a write-optimized, single-node key-value store. State-of-the-art write-optimized key-value stores are typically based on variants of the popular Log-Structured Merge (LSM) Tree. The WB Tree replaces the I/O-heavy primitive in the LSM Tree, the compaction, with new light-weight primitives called spills and splits. Further, a novel technique called fast-splitting is proposed to improve the performance of splits. Using these techniques, the WB Tree's insert throughput is up to 7× and 14× faster than LevelDB and bLSM, two LSM Tree implementations, for 64B records. The tradeoff is that unindexed read performance in a WB Tree is worse than unindexed LSM Tree performance. A solution to restore read performance is then proposed: a new set of indexes for the WB Tree allow reads to be performed with 1-2 seeks using less than 2B/key for the index.

8. REFERENCES

[1] hbase.apache.org.

[2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a fast array of wimpy nodes. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating systems principles*, pages 1–14, New York, NY, USA, 2009. ACM.

[3] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[4] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques*, RANDOM '02, pages 1–10, London, UK, UK, 2002. Springer-Verlag.

[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 *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 81–92, New York, NY, USA, 2007. ACM.

[6] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[7] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-Time Search at Twitter. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 1360–1369, Washington, DC, USA, 2012. IEEE Computer Society.

[8] 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. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[9] B. Chazelle and L. J. Guibas. Fractional Cascading: A Data Structuring Technique with Geometric Applications. In *Automata, Languages and Programming*, volume 194 of *Lecture Notes in Computer Science*, pages 90–100. Springer Berlin Heidelberg, 1985.

[10] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

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

[12] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. *SIAM J. Comput.*, 23(4):738–761, Aug. 1994.

[13] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A Distributed, Searchable Key-Value Store. *SIGCOMM Comput. Commun. Rev.*, 42(4):25–36, Aug. 2012.

[14] J. Evans. A Scalable Concurrent `malloc(3)` Implementation for FreeBSD. *BSDcan*, 2012.

[15] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing - a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, Sept. 1979.

[16] Google. Sparsehash. `http://code.google.com/p/sparsehash/`.

[17] Google. Leveldb. `code.google.com/p/leveldb`, 2012.

[18] G. Graefe. Write-Optimized B-trees. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 672–683. VLDB Endowment, 2004.

[19] C. Jermaine, E. Omiecinski, and W. G. Yee. The Partitioned Exponential File for Database Storage Management. *The VLDB Journal*, 16(4):417–437, Oct. 2007.

[20] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[21] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree Indexing on Solid State Drives. *Proc. VLDB Endow.*, 3(1-2):1195–1206, Sept. 2010.

[22] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, New York, NY, USA, 2011. ACM.

[23] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful Bulk Processing for Incremental Analytics. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 51–62, New York, NY, USA, 2010. ACM.

[24] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '99, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[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, June 1996.

[26] D. Peng and F. Dabek. Large-scale Incremental Processing using Distributed Transactions and Notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–15, Berkeley, CA, USA, 2010. USENIX Association.

[27] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[28] R. Sears and R. Ramakrishnan. bLSM: A General Purpose Log-Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228, New York, NY, USA, 2012. ACM.

[29] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File System Logging Versus Clustering: A Performance Comparison. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 21–21, Berkeley, CA, USA, 1995. USENIX Association.

[30] R. P. Spillane, P. J. Shetty, E. Zadok, S. Dixit, and S. Archak. An Efficient Multi-tier Tablet Server Storage Architecture. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 1:1–1:14, New York, NY, USA, 2011. ACM.