An Introduction to B$^\varepsilon$-trees and Write-Optimization

Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson,
Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan

A B$^\varepsilon$-tree is an example of a *write-optimized* data structure and can be used to organize on-disk storage for an application such as a database or file system. A B$^\varepsilon$-tree provides a key-value API, similar to a B-tree, but with better performance, particularly for inserts, range queries, and key-value updates. This article describes the B$^\varepsilon$-tree, compares its asymptotic performance to B-trees and Log-Structured Merge trees (LSM-trees), and presents real-world performance measurements. After finishing this article, a reader should have a basic understanding of how a B$^\varepsilon$-tree works, its performance characteristics, how it compares to other key-value stores, and how to design applications to gain the most performance from a B$^\varepsilon$-tree.

1 B$^\varepsilon$-trees

B$^\varepsilon$-trees were proposed by Brodal and Fagerberg [1] as a way to demonstrate an asymptotic performance tradeoff curve between B-trees [2] and buffered repository trees [3]. Both data structures support the same operations, but a B-tree favors queries whereas a buffered repository tree favors inserts.

Researchers, including the authors of this article, have recognized the practical utility of a B$^\varepsilon$-tree when configured to occupy the "middle ground" of this curve—realizing query performance comparable to a B-tree but insert performance orders of magnitude faster than a B-tree. The B$^\varepsilon$-tree has since been used by both the high-performance, commercial TokuDB database [4] and the BetrFS research file system [5]. For the interested reader, we have created a simple, reference implementation of a B$^\varepsilon$-tree, available at `https://github.com/oscarlab/Be-Tree`.

We first explain how the basic operations on a B$^\varepsilon$-tree work. We then give the motivation behind these design choices and illustrate how these choices affect the asymptotic analysis.

API and basic structure. A B$^\varepsilon$-tree is a B-tree-like search tree for organizing on-disk data, as illustrated in Figure 1. Both B-trees and B$^\varepsilon$-trees export a key-value store API:

- `insert`(k, v)
- `delete`(k)
- v `= query`(k)
- `[`v_1`,` v_2`,...] = range-query`(k_1, k_2)

Like a B-tree, the node size in a B$^\varepsilon$-tree is chosen to be a multiple of the underlying storage device's block size. Typical B$^\varepsilon$-tree node sizes range from a few hundred kilobytes to a few megabytes. In both B-trees and B$^\varepsilon$-trees, internal nodes store pivot keys and child pointers, and leaves store key-value pairs, sorted by key. For simplicity, one can think of each key-value or pivot-pointer pair as being unit size; both B-trees and B$^\varepsilon$-trees can store keys and values of different sizes in practice. Thus, a leaf of size B holds B key-value pairs, which we call **items** below.

The distinguishing feature of a B$^\varepsilon$-tree is that internal nodes also allocate some space for a buffer, as shown in Figure 1. The buffer in each internal node is used to store **messages**, which encode updates that will eventually be applied to leaves under this node. This buffer is not an in-memory data structure; it is part of the node and is written to disk, evicted from memory, etc., whenever the node is. The value of ε, which must be between 0 and 1, is a tuning parameter that selects how much space internal nodes use for pivots ($\approx B^\varepsilon$) and how much space is used as a buffer ($\approx B - B^\varepsilon$).

Inserts and deletes. Insertions are encoded as "insert messages", addressed to a particular key, and added to the buffer of the root node of the tree. When enough messages have been added to a node to fill the node's buffer, a batch of messages are flushed to one of the node's children. Generally, the child with the most pending messages is selected. Over the course of flushing, each message is ultimately delivered to the appropriate leaf node, and the new key and value are added to the leaf. When a leaf node becomes too full, it splits, just as in a B-tree. Similar to a B-tree, when an interior node gets too many children, it splits and the messages in its buffer are distributed between the two new nodes.

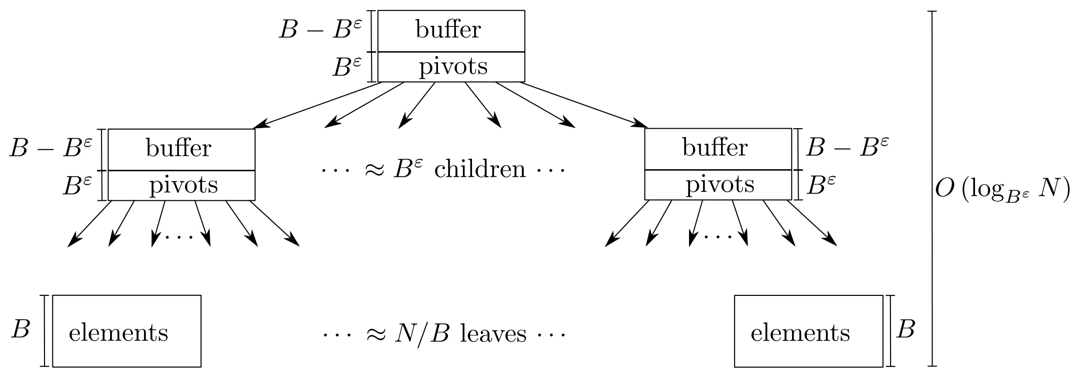

Figure 1: A B$^\varepsilon$-tree. Each node is roughly of size B, and ε controls how much of an internal node's space is used for pivots (B^ε) and how much is used for buffering pending updates ($B - B^\varepsilon$). As in a B-tree, items are stored in leaves, and the height of the tree is logarithmic in the total number of items (N), based on the branching factor (here B^ε).

Moving messages down the tree in batches is the key to the B$^\varepsilon$-tree's insert performance. By storing newly-inserted messages in a buffer near the root, a B$^\varepsilon$-tree can avoid seeking all over the disk to put elements in their target locations. The B$^\varepsilon$-tree only moves messages to a subtree when enough messages have accumulated for that subtree to amortize the I/O cost. Although this involves rewriting the same data multiple times, this can *improve* performance for smaller, random inserts, as our analysis in the next section shows.

B$^\varepsilon$-trees delete items by inserting "tombstone messages" into the tree. These tombstone messages are flushed down the tree until they reach a leaf. When a tombstone message is flushed to a leaf, the B$^\varepsilon$-tree discards both the deleted item and the tombstone message. Thus, a deleted item, or even entire leaf node, can continue to exist until a tombstone message reaches the leaf. Because deletes are encoded as messages, deletions are algorithmically very similar to insertions.

A high-performance B$^\varepsilon$-tree should detect and optimize the case where a a large series of messages all go to one leaf. Suppose a series of keys are inserted that will completely fill one leaf. Rather than write these messages to an internal node, only to immediately rewrite them to each node on the path from root to leaf, these messages should flush directly to the leaf, along with any other pending messages for that leaf. The B$^\varepsilon$-tree implementation in TokuDB and BetrFS includes some heuristics to avoid writing in intermediate nodes when a batch of messages are all going to a single child.

Point and range queries. Messages addressed to a key k are guaranteed to be applied to k's leaf or in some buffer along the root-to-leaf path towards key k. This invariant ensures that point and range queries in a B$^\varepsilon$-tree have a similar I/O cost to a B-tree.

In both a B-tree and a B$^\varepsilon$-tree, a point query visits each node from the root to the correct leaf. However, in a B$^\varepsilon$-tree, answering a query also means checking the buffers in nodes on this path for messages, and applying relevant messages before returning the results of the query. For example, if a query for key k finds an entry (k, v) in a leaf and a tombstone message for k in the buffer of an internal node, then the query will return "NOT FOUND", since the entry for key k has been logically deleted from the tree. Note that the query need not update the leaf in this case—it will eventually be updated when the tombstone message is flushed to the leaf. A range query is similar to a point query, except that messages for the entire range of keys must be checked and applied as the appropriate subtree is traversed.

In order to make searching and inserting into buffers efficient, the message buffers within each node are typically organized into a balanced binary search tree, such as a red-black tree. Messages in the buffer are sorted by their target key, followed by timestamp. The timestamp ensures that messages are applied in the correct order. Thus, inserting a message into a buffer, searching within a buffer, and flushing from one buffer to another are all fast.

1.1 Performance analysis

We analyze the behavior of B-trees, B$^\varepsilon$-trees, and LSM-trees in this article in terms of I/Os. Our primary interest is in data sets too large to fit into RAM. Thus, the first-order performance impact is how many I/O requests must be issued to complete each operation. In the algorithms literature, this is known as the disk-access-machine (DAM) model, the external-memory model, or the I/O model [6].

Performance model. In order to compare B-trees and B$^\varepsilon$-trees in a single framework, we make a few simplifying assumptions. We assume that all key-value pairs are the same size and that each node in the tree can hold B key-value pairs. The entire tree stores N key-value pairs. We also assume that each node can be accessed with a single I/O transaction—i.e., on a rotating disk, the node is stored contiguously and requires only one random seek.

This model focuses on the principal performance characteristics of a block storage device, such as a hard drive or SSD. For instance, on a hard drive, this model captures the latency of a random seek to read a node. In the case of an SSD, the model captures the I/O bandwidth costs, i.e., the number of blocks that must be read or written from the device per operation. Regardless of whether the device is bandwidth or latency bound, for a given node size B, minimizing the number of nodes accessed minimizes both bandwidth and latency costs.

B$^\varepsilon$-tree I/O performance. Table 1 lists the asymptotic complexities of each operation in a B-tree and B$^\varepsilon$-tree. We will explain upserts and epsilon (ε), as well as how they affect performance, later in the article. For this discussion, note that ε is a tuning parameter between 0 and 1; ε is generally set at design time and becomes a constant in the analysis.

The point-query complexities of a B-tree and a B$^\varepsilon$-tree are both logarithmic in the number of items ($O(\log_B N)$); a B$^\varepsilon$-tree adds a constant overhead of $1/\varepsilon$. Compared to a B-tree with the same node size, a B$^\varepsilon$-tree reduces the fanout from B to B^ε, making the tree taller by a factor of $1/\varepsilon$. Thus, for example, querying a B$^\varepsilon$-tree, where $\varepsilon = 1/2$, will require at most twice as many I/Os.

Range queries incur a logarithmic search cost for the first key, as well as a cost that is proportional to the size of the range and how many disk blocks the range is distributed across. The scan cost is roughly the number of keys read (k) divided by the block size (B). The total cost of a range query is $O(k/B+\log_B N)$ I/Os. Compared to a B-tree, batching messages *divides the insertion cost by the batch size* ($B^{1-\varepsilon}$). For example, if $B = 1024$ and $\varepsilon = 1/2$, a B$^\varepsilon$-tree can perform inserts $\approx \varepsilon B^{1-\varepsilon} = \frac{1}{2}\sqrt{1024} = 16$ times faster than a B-tree.

Write optimization. Batching small, random inserts is an essential feature of write-optimized data structures (WODS), such as a B$^\varepsilon$-tree or LSM-tree. Although a WODS may issue a small write multiple times as a message moves down the tree, once the I/O cost is divided among a large batch, the cost per insert or delete is much smaller than one I/O per operation. In contrast, a workload of random inserts to a B-tree requires a *minimum* of one I/O per insert—to write the new element to its target leaf.

The B$^\varepsilon$-tree flushing strategy is designed to ensure that it can always move elements in large batches. Messages are only flushed to a child when the buffer of a node is full, containing $B - B^\varepsilon \approx B$ messages. When a buffer is flushed, not all messages are necessarily flushed— messages are only flushed to children with enough pending messages to offset the cost of rewriting the parent and child nodes. Specifically, at least $(B - B^\varepsilon)/B^\varepsilon \approx B^{1-\varepsilon}$ messages are moved from the parent's buffer to the child's on each flush. Consequently, any node in a B$^\varepsilon$-tree is only rewritten if a sufficiently large portion of the node will change.

Caching. Most systems cache a subset of the tree in RAM. With an LRU replacement policy, accesses to the top of the tree are likely to hit in the cache, whereas accesses to leaves and "lower nodes" will more commonly miss. Thus, when the cache is warm, the actual cost of a search may be much less than $O(\log_B N)$ I/Os. For both B-trees and B$^\varepsilon$-trees, if only the leaves are out-of-cache, point queries and updates require a single I/O, whereas a range query has an I/O cost that is linear in the number of leaves read.

1.2 The impact of node size (B) on performance

B-trees have small nodes to balance the cost of insertions and range queries. B-tree implementations face a trade-off between update and range-query performance. A larger node size B favors range queries and a smaller node size favors inserts and deletes. Larger nodes help range query performance because the I/O costs, such as seeks, can be amortized over more data. However, larger nodes make updates more expensive because a leaf and possibly internal nodes must be completely re-written each time a new item is added to the index, and larger nodes mean more to rewrite.

Thus, many B-tree implementations use small nodes (tens to hundreds of KB), resulting in sub-optimal range-query performance. As free space on

Table 1: Asymptotic I/O costs of important operations. B$^\varepsilon$-trees simultaneously support efficient inserts, point queries (even in the presence of upserts), and range queries. These complexities apply for $0 < \varepsilon \leq 1$. Note that ε is a design-time constant. We show the complexity for general ε and evaluate the complexity when ε is set to a typical value of 1/2. The $1/\varepsilon$ factor evaluates to a constant that disappears in the asymptotic analysis.

Data Structure	Insert	Point Query no Upserts	Point Query w/ Upserts	Range Query
B$^\varepsilon$-tree	$\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}$	$\frac{\log_B N}{\varepsilon}$	$\frac{\log_B N}{\varepsilon}$	$\frac{\log_B N}{\varepsilon} + \frac{k}{B}$
B$^\varepsilon$-tree ($\varepsilon = 1/2$)	$\frac{\log_B N}{\sqrt{B}}$	$\log_B N$	$\log_B N$	$\log_B N + \frac{k}{B}$
B-tree	$\log_B N$	$\log_B N$	$\log_B N$	$\log_B N + \frac{k}{B}$
LSM	$\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}$	$\frac{\log_B^2 N}{\varepsilon}$	$\frac{\log_B^2 N}{\varepsilon}$	$\frac{\log_B^2 N}{\varepsilon} + \frac{k}{B}$
LSM+BF	$\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}$	$\log_B N$	$\frac{\log_B^2 N}{\varepsilon}$	$\frac{\log_B^2 N}{\varepsilon} + \frac{k}{B}$

disk becomes fragmented, B-tree nodes may also become scattered on disk; this is sometimes called *aging*. Now a range query must seek for each leaf in the scan, resulting in poor bandwidth utilization.

For example, with 4KB nodes stored on a disk with a 5ms seek time and 100MB/s bandwidth, updating a single key only rewrites 4KB. Range queries, however, must perform a seek for each 4KB leaf node, resulting in a net bandwidth of 800KB/s, less than 1% of the disk's potential bandwidth.

B$^\varepsilon$-trees have efficient updates and range queries even when nodes are large. In contrast, batching in a B$^\varepsilon$-tree allows B to be much larger in a B$^\varepsilon$-tree than in a B-tree. In a B$^\varepsilon$-tree the bandwidth cost per insert is $O(\frac{B^\varepsilon \log_B N}{\varepsilon})$, which grows much more slowly as B increases. As a result, B$^\varepsilon$-trees use node sizes of a few hundred kilobytes to a few megabytes.

By using large B, B$^\varepsilon$-trees can perform range queries at near disk bandwidth. For example, a B$^\varepsilon$-tree using 4MB nodes need perform only one seek for every 4MB of data it returns, yielding a net bandwidth of over 88MB/s on the same disk as above.

In the comparison of insert complexities above, we stated that a B$^\varepsilon$-tree with $\varepsilon = 1/2$ would be twice as deep as a B-tree, as some fanout is sacrificed for buffer space. This is only true when the node size is the same. Because a B$^\varepsilon$-tree can use larger nodes in practice, a B$^\varepsilon$-tree can still have close to the same fanout and height as a B-tree.

1.3 The role of ε

The parameter ε in a B$^\varepsilon$-tree was originally designed to show that there is an optimal trade-off curve between insert and point query performance. Parameter ε ranges between 0 and 1. As we explain in the rest of this section, making ε an exponent simplifies the asymptotic analysis and creates an interesting trade-off curve.

Intuitively, the trade-off with parameter ε is how much space in the node is used for storing pivots and child pointers ($\approx B^\varepsilon$) and how much space is used for message buffers ($\approx B - B^\varepsilon$). As ε increases, so does the branching factor (B^ε), causing the depth of the tree to decrease and searches to run faster. As ε decreases, the buffers get larger, batching more inserts for every flush and improving overall insert performance.

At one extreme, when $\varepsilon = 1$, a B$^\varepsilon$-tree is just a B-tree, since interior nodes contain only pivot keys and child pointers. On the other, when $\varepsilon = 0$, a B$^\varepsilon$-tree is a binary search tree with a large buffer at each node, called a buffered repository tree [3].

The most interesting configurations place ε strictly between 0 and 1, such as $\varepsilon = 1/2$. For such configurations, a B$^\varepsilon$-tree has the same asymptotic point query performance as a B-tree, but asymptotically better insert performance.

For inserts, setting $\varepsilon = 1/2$ *divides* the cost by the square root of node size. Formally, the cost then becomes: $O(\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}) = O(\frac{\log_B N}{\sqrt{B}})$. Since the insert cost is divided by $\sqrt{B}$, selecting larger nodes (increasing B) can dramatically improve insert performance.

Assuming all other parameters are the same, decreasing ε slows down point queries by a constant

$1/\varepsilon$. To see the query performance for $\varepsilon = 1/2$, evaluate the point query cost in Table 1: $O(\frac{\log_B N}{\varepsilon}) = O(\frac{\log_B N}{1/2}) = O(2 \log_B N)$—doubling the number of I/Os. Changing ε from 1/2 to 1/4 would make this a factor of 4. This cost can be offset by increasing B, which, as pointed out above, also improves insert performance.

The above analysis assumes all keys have unit size and that nodes can hold B keys; real systems must deal with variable-sized keys, so B, and hence ε, are not fixed or known a priori. Nonetheless, the main insight of B^ε-trees—that we can speed up insertions by buffering items in internal nodes and flushing them down the tree in batches—still applies in this setting.

In practice, B^ε-tree implementations select a fixed physical node size and fanout (B^ε). For the implementation in TokuDB and BetrFS, nodes are approximately 4MB and the branching factor ranges from 4 to 16. As a result, the fractal tree can always flush data in batches of at least 256KB.

1.4 How to speed up an applications by using a B^ε-tree

A practical consequence of the analysis above is that a B^ε-tree can perform updates orders of magnitude faster than point queries. This *search-insert asymmetry* has two implications for designing applications on B^ε-trees.

Performance rule. *Avoid query-before-update whenever possible.*

Because of the search-insert asymmetry, the common read-modify-write (or query-modify-insert) pattern will be bound to the speed of a query, which is no faster in a B^ε-tree than in a B-tree.

Upserts. B^ε-trees support a new *upsert* operation, to help applications bridge this performance asymmetry. An upsert is a type of message that encodes an update with a callback function that can be issued without first knowing the value of the key.

Upserts can encode any modification that is asynchronous and depends only on the key, the old value, and some auxiliary data that can be stored with the upsert message. Tombstones are a special case of upserts. Upserts can also be used to increment a counter, update the access time on a file, update a user's account balance after a withdrawal, and many other operations.

With upserts, an application can update the value associated with key k in the B^ε-tree by inserting an "upsert message" $(k, (f, \Delta))$ into the tree, where f is a call-back function and Δ is auxiliary data specifying the update to be performed. This upsert message is semantically equivalent to performing a query followed by an insert:

$$v \leftarrow \texttt{query}(k); \quad \texttt{insert}(k, f(v, \Delta)).$$

However, the upsert does not perform these operations. Rather, the message $(k, (f, \Delta))$ is inserted into the tree like an insert or tombstone message.

When an upsert message $(k, (f, \Delta))$ is flushed to a leaf, the value v associated with k in the leaf is replaced by $f(v, \Delta)$ and the upsert message is discarded. If the application queries k before the upsert message reaches a leaf, then the upsert message is applied to v before the query returns.

As with any an insert or delete message, multiple upserts can be buffered for the same key between the root and leaf. If a key is queried with multiple upserts pending, each upsert must be collected on the path from root to leaf, and applied to the key in the order they were inserted into the tree.

The upsert mechanism does not interfere with I/O performance of searches, because the upsert messages for a key k always lie on the search path from the root of the B^ε-tree to the leaf containing k. Thus, the upsert mechanism can speed up updates by one to two orders of magnitude without slowing down queries.

Performance rule. *Use insert performance to improve query performance by maintaining appropriate indices.*

Secondary indices. In a database, secondary indices can greatly speed up queries. For example, consider a database table with three columns, k_1, k_2, and k_3, and an application that sometimes performs queries using k_1 and sometimes using k_2. If the table is implemented as a B-tree sorted on k_1, then queries using k_1 are fast, but queries using k_2 are extremely slow—they may have to scan essentially the entire database. To solve this problem, the table can be configured to maintain two indices—one sorted by k_1 and one sorted by k_2. Queries can then use the appropriate index based on the type of the query.

When a multiple indices are maintained with B-trees, each index update requires an additional insert. Because inserts are as expensive as a point query, maintaining an index on each column is often impractical. Thus, the table designer must carefully analyze factors such as the expected type of queries and distribution of keys in deciding which columns to index, in order to ensure good overall performance.

B^ε-trees turn these issues upside down. Indices are cheap to maintain. Point queries are fundamentally expensive—B$^\varepsilon$-tree point queries are no faster than in a B-tree. Thus, B$^\varepsilon$-tree applications should maintain whatever indices are needed to perform queries efficiently.

There are three rules for designing good B$^\varepsilon$-tree indices.

First, maintain indices sorted by the keys used to query the database. For example, in the above example, the database should maintain two B$^\varepsilon$-trees—one sorted by k_1 and one sorted by k_2.

Second, ensure that each index has all the information required to answer the intended queries. For example, if the application looks up the k_3 value using key k_2, then the index sorted by k_2 should store the corresponding k_3 value for each entry. In many databases, the secondary index contains only keys into the primary index. Thus, for example, a query on k_2 would return the primary key value, k_1. To obtain k_3, the application has to perform another query in the primary index using the k_1 value obtained from the secondary index. An index that contains all the information relevant to a query is called a *covering* index for that query.

Finally, design indices to enable applications to perform range queries whenever possible. For example, if the application wants to lookup all entries (k_1, k_2, k_3) for which $a \leq k_1 \leq b$ and k_2 satisfies some predicate, then the application should maintain a secondary index sorted by k_1 that only contains entries for which k_2 matches the predicate.

1.5 Log-structured merge-trees

Log-structured merge trees (LSM-trees) [7] are a WODS with many variants [8, 9]. An LSM-tree typically consists of a logarithmic number of B-trees of exponentially increasing size. Once an index at one level fills up, it is emptied by merging it into the index at the next level. The factor by which each level grows is a tunable parameter comparable to the branching factor (B^ε) in a B$^\varepsilon$-tree. For ease of comparison, Table 1 gives the I/O complexities of operations in an LSM-tree with growth factor B^ε.

LSM-trees can be tuned to have the same insertion complexity as a B$^\varepsilon$-tree, but queries in a naïvely implemented LSM-tree can require $O(\frac{\log_B^2 N}{\varepsilon})$ I/Os because the query must be repeated in $O(\log_B N)$ B-trees. Most LSM-tree implementations use Bloom filters to avoid queries in all but one of the B-trees, improving point query performance to $O(\frac{\log_B N}{\varepsilon})$ I/Os.

One problem for LSM-trees is that the benefits of Bloom filters do not extend to range queries. Bloom filters are only designed to improve point queries and do not support range queries. Thus, a range query must be done on every level of the LSM-tree—squaring the search overhead in Table 1, and yielding strictly worse asymptotic performance than a B$^\varepsilon$-tree or a B-tree.

A second advantage of a B$^\varepsilon$-tree over an LSM-tree is that B$^\varepsilon$-trees can effectively use upserts, whereas upserts in an LSM-tree will ruin the performance advantage of adding Bloom filters. As discussed above, upserts address a search-insert asymmetry common to any WODS, including LSM-trees. When an application uses upserts, it is possible for a message for that key to be present in *every* level of the tree containing a pending message for the key. Thus, a subsequent point query will still have to query every level of the tree, defeating the purpose of adding Bloom filters. Note that querying every level of an LSM-tree also squares the overhead compared to a B$^\varepsilon$-tree or B-tree, and is more expensive than walking the path from root-to-leaf in a B$^\varepsilon$-tree.

In summary, Bloom-filter-enhanced LSM-trees can match the performance of B$^\varepsilon$-trees for some but not all workloads. B$^\varepsilon$-trees asymptotically dominate LSM-tree performance. In particular, B$^\varepsilon$-trees are asymptotically faster than LSM-trees for small range queries and point queries in upsert-intensive workloads.

2 Performance comparison

To give a sense of how B$^\varepsilon$-trees performs in practice, we present some data from BetrFS, an in-kernel, research file system based on B$^\varepsilon$-trees. We compare BetrFS to other file systems, including `btrfs`, which is built with B-trees. A more thorough evaluation appears in our recent FAST paper [5].

All experimental results were collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU, 4 GB RAM, and a 250 GB, 7200 RPM ATA disk. Each file system used a 4096-byte block size. The system ran Ubuntu 13.10, 64-bit, with Linux kernel version 3.11.10. Each experiment compared with several general purpose file systems, including `BTRFS`, `ext4`, `XFS`, and `ZFS`. Error bars and ± ranges denote 95% confidence intervals. Unless otherwise noted, benchmarks are cold-cache tests.

Small Writes. We used the TokuBench benchmark [10] to create 3 million 200-byte files in a balanced directory tree with fanout of 128, using 4 threads (one per CPU). In BetrFS, file creations are implemented as B$^\varepsilon$-tree inserts and small file writes are implemented using upserts, so this benchmark demonstrates the B$^\varepsilon$-tree's performance on these two

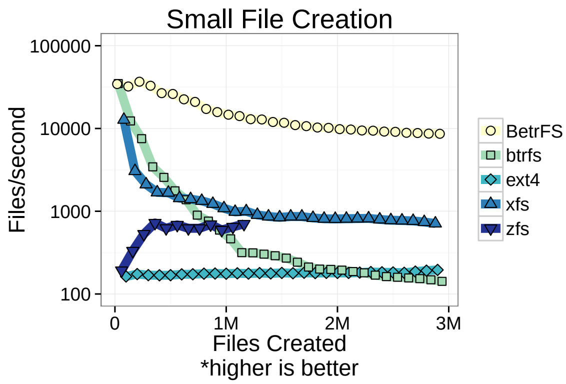

Figure 2: Sustained rate of file creation for 3 million 200-byte files, using 4 threads. Higher is better.

FS	**find**	**grep**
B*etr*FS	0.36 ± 0.06	3.95 ± 0.28
BTRFS	3.87 ± 0.94	14.91 ± 1.18
ext4	2.47 ± 0.07	46.73 ± 3.86
XFS	19.07 ± 3.38	66.20 ± 15.99
ZFS	11.60 ± 0.81	41.74 ± 0.64

Table 2: Directory operation benchmarks, measured in seconds. Lower is better.

operations. Figure 2 shows the sustained rate of file creation in each file system (note the log scale). In the case of ZFS, the file system crashed before completing the benchmark, so we reran the experiment five times and used data from the longest-running iteration. B*etr*FS is initially among the fastest file systems, and continues to perform well for the duration of the experiment. The steady-state performance of B*etr*FS is an order of magnitude faster than the other file systems.

This performance distinction is attributable to both fewer total writes and fewer seeks per byte written—i.e., better aggregation of small writes. Based on profiling from `blktrace`, one major distinction is total bytes written: B*etr*FS writes 4–10× fewer total MB to disk, with an order of magnitude fewer total write requests. Among the other file systems, ext4, XFS, and ZFS wrote roughly the same amount of data, but realized widely varying underlying write throughput.

Locality and Directory Operations. In BetrFS, fast range queries translate to fast large directory scans. Table 2 reports the time taken to run `find` and `grep -r` on the Linux 3.11.10 source tree, starting from a cold cache. The `grep` test recursively searches the file contents for the string "cpu_to_be64", and the `find` test searches for files named "wait.c".

Both the `find` and `grep` benchmarks do well because file system data and metadata are stored in large nodes, and sorted lexicographically by full path. Thus, related files are stored near each other on disk. BetrFS also maintains a second index that contains only metadata, so that metadata scans can be implemented as range queries. As a result, B*etr*FS can search directory metadata and file data one or two orders of magnitude faster than the other file systems.

Limitations. It is important to note that BetrFS is a still a research prototype and currently has three primary cases where it performs considerably worse than other file systems: large directory renames, large deletes, and large sequential writes (more details in [5]). Renames and deletes are slow because BetrFS does not map them directly onto B^ε-tree operations. Sequential writes are slow largely because the underlying fractal tree appends all data to a log before inserting it into the tree, so everything is written to disk at least twice. We believe these issues can be addressed in ongoing research and development efforts; our goal, supported by the asymptotic analysis, is for BetrFS to match or exceed the performance of other file systems on *all* workloads.

3 Conclusion

B^ε-tree implementations can match the search performance of B-trees, perform inserts and deletes orders-of-magnitude faster, and execute range queries at near disk bandwidth. The design and implementation of B^ε-trees converts a trade-off between update and range query costs into a mutually-beneficial synergy between batching small updates and large nodes. Our results with BetrFS demonstrate that the asymptotic improvements of B^ε-trees can yield practical performance improvements for applications that are designed for B^ε-tree's performance characteristics.

Acknowledgments

This work was supported in part by NSF grants CNS-1409238, CNS-1408782, CNS-1408695, CNS-1405641, CNS-1149229, CNS-1161541, CNS-1228839, CNS-1408782, IIS-1247750, CCF-1314547, Sandia National Laboratories, and the Office of the Vice President for Research at Stony Brook University.

References

[1] G. S. Brodal and R. Fagerberg, "Lower bounds for external memory dictionaries," in *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 546–554, 2003.

[2] D. Comer, "The ubiquitous B-tree," *ACM Computing Surveys*, vol. 11, pp. 121–137, June 1979.

[3] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook, "On external memory graph traversal," in *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 859–860, 2000.

[4] Tokutek, Inc., "TokuDB: MySQL Performance, MariaDB Performance." `http://www.tokutek.com/products/tokudb-for-mysql/`, 2013.

[5] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter, "BetrFS: A right-optimized write-optimized file system," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pp. 301–315, 2015.

[6] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, pp. 1116–1127, Sept. 1988.

[7] P. O'Neil, E. Cheng, D. Gawlic, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[8] 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*, pp. 217–228, 2012.

[9] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with VT-trees," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pp. 17–30, 2013.

[10] J. Esmet, M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul, "The TokuFS streaming file system," in *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage (HotStorage)*, June 2012.