Investigation of Adaptive Hotspot-Aware Indexes for Oscillating Write-Heavy and Read-Heavy Workloads - An Experimental Study

Lu Xing
Purdue University, West Lafayette, IN
xingl@purdue.edu

Walid G. Aref
Purdue University, West Lafayette, IN
aref@purdue.edu

ABSTRACT

HTAP systems are designed to handle transactional and analytical workloads. Besides a mixed workload at any given time, the workload can also change over time. A popular kind of continuously changing workload is one that oscillates between being write-heavy and being read-heavy. These oscillating workloads can be observed in many applications. Indexes, e.g., the B$^+$-tree and the LSM-Tree cannot perform equally well all the time. Conventional adaptive indexing does not solve this issue either as it focuses on adapting in one direction. This paper investigates how to support oscillating workloads with adaptive indexes that adapt the underlying index structures in both directions. With the observation that real-world datasets are skewed, we focus on optimizing the indexes within the hotspot regions. We encapsulate the adaptation techniques into the Adaptive Hotspot-Aware Tree adaptive index. We compare the indexes and discuss the insights of each adaptation technique. Our investigation highlights the trade-offs of AHA-tree as well as the pros and cons of each design choice. AHA-tree can behave competitively as compared to an LSM-tree for write-heavy transactional workloads. Upon switching to a read-heavy analytical workload, and after some transient adaptation period, AHA-tree can behave as a B$^+$-tree and can match the B$^+$-tree's read performance.

1 INTRODUCTION

Nowadays, database management systems are not just built for a single purpose, rather they face various requirements from users. Hybrid Transactional and Analytical Processing (HTAP) systems are becoming more popular as they address a hybrid of requirements. The hybrid requirements include transactional processing as well as analytical queries. However, HTAP systems usually face a situation where workload changes over time. The HTAP system benchmark [28] includes workloads that feature transactions first, then analytical next. The change in workload is also observed as a diurnal pattern in one of the Rocksdb use cases at Meta [8], and is also observed in [11], where the number of tweets fluctuate across the day. In C-store [25], Stonebraker et al. mention that data warehouses periodically perform a bulk load of new data followed by a relatively long period of ad-hoc analytical queries.

One category of changing workloads is the oscillating workload that is write-heavy at times and is read-heavy at other times. In social media applications, users post and comment actively during the day, and browse content in the night or early in the morning. This corresponds to the write-heavy (post and comment) and read-heavy (browse) workloads that keep repeating every day. Similar examples can be found in the context of traffic incidents management. When incidents happen, there can be a surge in the write operations while at other times a read-heavy workload is the dominant one.

We focus on range search query as a representative analytical query with read operations. In contrast to a point query, one can control selectivity by changing the size of the range. Traditional non-adaptive indexes, e.g., the B$^+$-tree [5, 10] and the Log-Structured Merge Tree (LSM-Tree) [21] that are optimized for only one operation cannot do well in the oscillating write-heavy and read-heavy workloads all the time. Existing adaptive indexes [3, 9, 17, 19, 23, 29] are not designed for this oscillating workload.

A natural thought to deal with the oscillating workload is to make the index write like an LSM-Tree in the write-heavy phase, while read like a B$^+$-tree in the read-heavy workload. We are inspired by the structure of a buffer tree [4], where it can be viewed as a tree part (a B$^+$-tree) plus the buffer part (an LSM-Tree). However, a buffer tree may not solve the problem of oscillating workloads as it is still a non-adaptive structure.

To make the buffer tree adaptive, we let the index adapt itself in either workload. During the read-heavy workload, the buffered data is sent to the leaf nodes s.t. range searches probe the index like a B$^+$-tree. During the write-heavy workload, data is buffered in batches to avoid I/Os caused by individual insertions. However, making the buffer tree adaptive introduces several challenges, and this adaptive index can still be optimized for the oscillating workloads. First, the original buffer tree buffers the writes and the range searches [4]. This degrades the latency of individual range searches. As we are dealing with oscillating workloads, batching range search queries during the read-heavy phase may not be favored from latency perspective. Second, the leaf level of the buffer tree is the same as the B$^+$-tree, where data is stored in leaf pages. However, merging data from the buffers of the tree with leaf pages is expensive and can be blocking. Third, the buffer is composed of several blocks of fixed size. When emptying the buffer, this requires expensive merge-sort and disk I/Os. Lastly, straightforward adaptation of the buffer tree overlooks the fact that most real-world data is skewed, meaning that the data sets have hotspots.

We enclose the above mentioned adapting techniques into a new adaptive index termed Adaptive Hotspot-Aware Tree. We test its adaptation under the oscillating write- and read-heavy workloads. We use an LSM-Tree as the structure for the buffer associated with each node. To further improve the write throughput, we treat the buffer of the root node differently. Besides, to further improve the write throughput, leaf nodes also contain LSM-Trees as buffers instead of storing data items in pages. This greatly improves the write throughput. And AHA-tree is hotspot-aware. This makes the adaptation process confined to the hotspot region. Thus, the adaptation process can get completed within a reasonable amount of time.

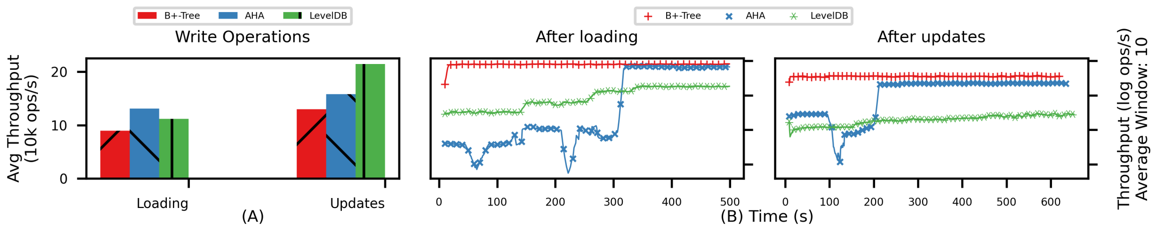


Figure 1: Index performance under oscillating write- and read-heavy workloads

We evaluate baseline indexes against AHA-tree for oscillating workloads. AHA-tree adapts to have competitive performance similar to that of the corresponding static (non-adaptive) index that is optimal for that workload. Being adaptive, AHA-tree takes time to adapt. The throughput during the adaptation phase can be low. Thus, AHA-tree demonstrates the trade-off between having a relatively low performance during the adaptation phase at the benefit of having competitive performance once the adaptation is completed. This trade-off between current and future performance is exemplified in AHA-tree.

The contributions of this paper can be summarized as follows:

- We introduce the adaptive index, AHA-tree, that is devised for workloads that oscillate between being read-heavy at times and being write-heavy at other times. AHA-tree handles concurrent read and write operations as well as adapts to workload changes using concurrent background adaptation threads.
- We conduct a thorough investigation of the performance of the index under various scenarios of oscillating workloads, hotspot awareness, and potential index optimizations. We provide insights into the trade-offs and effectiveness of these optimizations and techniques that are introduced in the context of adaptive hotspot-aware indexes.
- Based on the findings of this investigation, we provide a section on the lessons learned and recommendations on when to apply and when not to apply certain optimizations in relation to adaptive hotspot-aware tree indexes.

The rest of this paper proceeds as follows. Section 2 introduces the motivation and background for adaptive indexing. We present the design of AHA-tree and its operations in Section 3. Section 4 describes the optimizations we propose for AHA-tree. Sections 5 and 6 discuss the adaptation process and the techniques used during the adaptation phase. Section 7 presents the experimental results and investigates the performance of AHA-tree in contrast to existing read- and write-optimized indexes. Section 8 lists the recommendations on applying the certain optimizations to the adaptive hotspot-aware tree indexes. Section 9 discusses the related work, and Section 10 concludes the paper.

2 MOTIVATION AND BACKGROUND

Oscillating read- and write-heavy workload is a typical workload for HTAP systems. Write-heavy operations involve fast ingestion of data while read-heavy operations involve analysis of the newly ingested data. This workload pattern can be challenging for the non-adaptive indexes including the B$^+$-tree and the LSM-Tree. We load all the indexes with the same amount of data, then issue range search queries over the hotspot area. The B$^+$-tree shows the best range search throughput overtime while the LSM-Tree performs the worst in Figure 1(B). After this phase, each index is updated with the same number of updates in a hotspot region. This time, the average throughput of the LSM-Tree is the best (Figure 1(A)). We again issue range search queries in the hotspot, the B$^+$-tree performs better than the LSM-Tree. It is almost impossible for an index that is optimized for write (range search) to perform well for range search (read).

In [15], Idreos et al. have proposed a design continuum among indexes that indexes can be generalized under the same set of parameters. The transition between B-tree and LSM-Tree can be bridged by Bϵ-tree [6] and bLSM [24] as displayed in [15]'s Figure 5. Besides choosing an appropriate intermediate structure, we also observe that real-world workloads usually include one or more hotspots. These hotspot can be visited or inserted often thus it is worthwhile to treat hotspots differently from cold spot.

AHA-tree is a tree-like structure similar to either buffer tree [4] or Bϵ-tree [6]. While being adaptive, AHA-tree focuses on the hotspot. After initial index loading, AHA-tree adapts itself from time 0 to 300 sec. During this period, its throughput is relatively low (Figure 1(B)). After the adaptation finishes, AHA-tree catches up with B$^+$-tree in the range search performance (Figure 1 middle time 300 sec and above). While in the followed updates, AHA-tree updates faster than B$^+$-tree but slower than LSM-Tree (Figure 1(A)). In the subsequent range search phase, as more data are added to the hotspot, AHA-tree takes time to adapt and finally reaches B$^+$-tree (Figure 1(B)).

3 BASIC DESIGN OF AHA-TREE

AHA-tree combines ideas from the LSM-Tree [21], the B$^+$-tree [5, 10] and the buffer tree [4]. Under a write-heavy workload, all write operations are performed as if AHA-tree is an LSM-Tree. Under a read-heavy workload within the hotspot, all range queries search as if AHA-tree is a B$^+$-tree. A key challenge is how to maintain a valid index structure. We introduce a buffer tree-like structure [4] as the intermediate structure. Every node in AHA-tree has an associated buffer that is an LSM-tree with a fixed size. Let `rootLSM-tree` be the buffer LSM-tree of the root node and `nodeLSM-tree` be a buffer tree that is associated with each of the other nodes in the tree. In AHA-tree, the `rootLSM-tree` has both a memory component and a disk component. In contrast, the `nodeLSM-trees` for all the other nodes have only disk-components and are also of fixed sizes. The `rootLSM-tree` accepts all the incoming writes. When

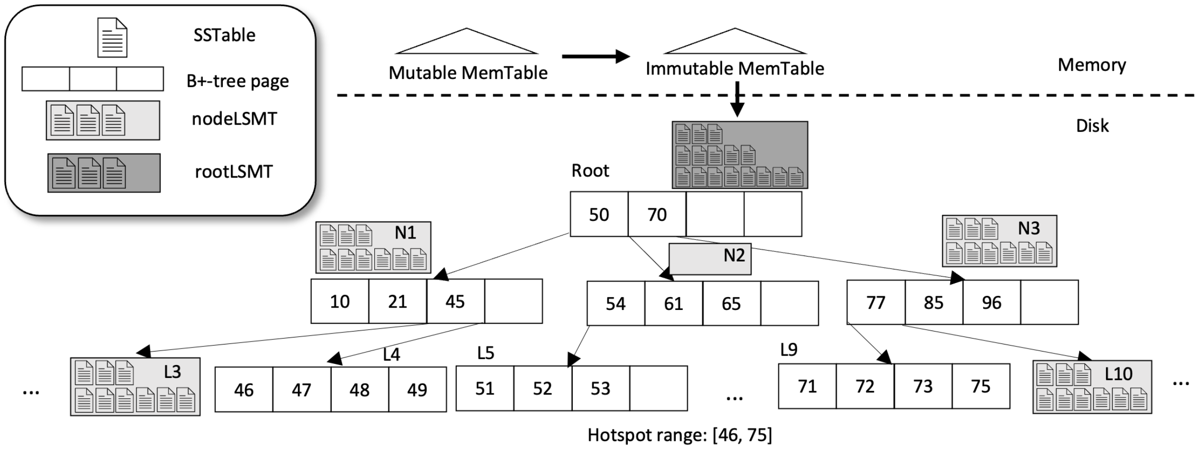


Figure 2: Structure of AHA-tree

the rootLSM-tree overflows, a level-emptying process is triggered to dispatch data items to the proper nodeLSM-trees of the children. When the workload becomes read-heavy, two major processes get started: A hotspot emptying process and a leaf node transformation process. The hotspot emptying process is similar to the level emptying process except that it gathers files that overlap with the hotspot range for flushing. In contrast, the leaf node transformation process transforms the leaf nodes with buffers into the B^+-tree leaf pages to speed up hotspot searches.

The rest of this section proceeds as follows. We present the index structure in Section 3.1, the insertion and range search operations in Sections 3.2 and 3.3, respectively. We describe the optimizations for the AHA-tree index in Section 4, the adaptation process in Section 5, and the adaptation techniques in Section 6.

3.1 Index Structure

Write operations are first batched in rootLSM-tree's memory component rootLSM-tree. MemTable. When rootLSM-tree.MemTable becomes full, it is written into rootLSM-tree's disk component and becomes immutable, and a new empty rootLSM-tree.MemTable is created. The memory component works in the same way as that for an ordinary LSM-tree, e.g., LevelDB [1], or RocksDB [2].

Figure 2 shows the structure of AHA-tree. Each node is associated with a nodeLSM-tree buffer. This differs from buffer tree where only non-leaf nodes have an associated buffer [4]. To facilitate reads inside the hotspot region, in the AHA-tree, the nodeLSM-tree buffers of the leaf pages that are inside the hotspot region gradually evolve into regular B^+-tree leaf nodes with pages and not LSM levels. AHA-tree maintains the following invariant:

Data Freshness Invariant:

(1) *Data in* rootLSM-tree *is fresher than data in any of the* nodeLSM-tree*s below it*, and
(2) *The closer an* nodeLSM-tree *to the root, the fresher the data.*

This invariant is enforced at all times for correctness of execution

The structure in Figure 2 is hybrid at the leaf level. Within a hotspot region starting from L4 to L9, the leaf nodes do not have nodeLSM-trees, but rather all data is stored in regular B^+-tree leaf pages. In the non-leaf nodes above the leaf nodes in the hotspot region, N2 has an empty nodeLSM-tree; N1 and N3 still have non-empty nodeLSM-trees but these nodeLSM-trees do not have hotspot data as this data is pushed all the way to the leaf node pages. This helps in speeding up range queries and analytics over the hotspot region. From the hotspot's perspective, the nodes above L4 to L9 do not have nodeLSM-trees to expedite the search, and this portion of AHA-tree is completely a B^+-tree.

Initially, during index construction, AHA-tree only has rootLSM-tree without any tree structure. When rootLSM-tree reaches a certain size (that is empirically-set size), a tree structure starts to form. Leaf nodes are created with files dispatched from rootLSM-tree and the root node page is populated with new routing keys. The initial tree construction can be expensive if the files in rootLSM-tree are re-compacted and are written to disks. We use a technique for bottom-up bulk-loading of the LSM-tree (to be explained in Section 4.1) that speeds up this process.

3.2 Insertion

First, we discuss how *batch insert* works given a write-heavy workload, especially how the level-emptying process proceeds. Then, we discuss *tree insert* in the context of a completely adapted AHA-tree.

3.2.1 Batch Insert. Under a write-heavy workload, AHA-tree behaves as if it is an LSM-tree. Updates are first buffered into the rootLSM-tree.MemTable. Once full, data items are written to the disk component of rtl, e.g., as in the SSTables that are used in LevelDB [1] and RocksDB [2]. SSTable is added to Level-0 of rootLSM-tree. The levels of rootLSM-tree compact and flush the same way as in the ordinary LSM-tree. Since blocking of rootLSM-tree may slow down data ingestion during write-heavy workloads, rootLSM-tree is designed differently in contrast to the other nodeLSM-trees. Section 4.2 discusses optimizations for rootLSM-tree to address this issue.

We enforce a limit in size for all node-associated buffers. When rootLSM-tree or nodeLSM-tree of non-leaf nodes fail to write files to a new level due to exceeding the size limit, the *level-emptying* *process* is triggered. If the `nodeLSM-tree` of a leaf node reaches its size limit, it is split into new leaf nodes; each with one new `nodeLSM-tree`. New routing keys are added to the parent of this leaf node, and this may be propagated up until the root node.

The Level-Emptying Process. This process is triggered by an overflowing `nodeLSM-tree` of a non-leaf node. A new level in `nodeLSM-tree` needs to be created to hold files, but this is disallowed when `nodeLSM-tree`'s size limit is reached. In Figure 2, the root node dispatches files in the bottom level of `rootLSM-tree` to its children nodes N1, N2 and N3. Notice that the bottom levels of `rootLSM-tree` are the ones migrating to the children nodes to maintain the freshness invariant of AHA-tree stated in Section 3.1. During compaction, the routing keys in the root are used as one more input s.t. the boundary of the resulting files align with the routing keys. This compaction is termed *guarded compaction* [22]. This idea was first discussed by PebblesDB in the context of the LSM-Tree with randomly picked guards [22].

Split. Splits in AHA-tree differ in the cases of leaf vs. non-leaf node splits. Initially, leaf nodes in AHA-tree hold data in their corresponding `nodeLSM-trees`. When a leaf node's `nodeLSM-tree` overflows, the leaf node splits. The entire `nodeLSM-tree` is read and is compacted, then new files are assigned to the new leaf nodes with new routing keys added to the parent node. One leaf node can be split to more than two nodes. We allow this so that each resulting leaf node may have smaller sized files. Also, in this case, leaf node splits can happen less frequently to reduce write amplification.

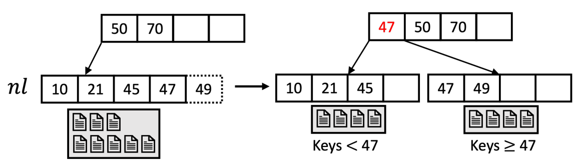


Figure 3: Internal node split

For or non-leaf nodes, two situations may occur. Either that the node page itself overflows or that a node's associated `nodeLSM-tree` overflows. The latter case is handled via the Level-Emptying Process discussed above. In the former case, a non-leaf page overflows. We need to articulate what happens to its corresponding `nodeLSM-tree`. A non-leaf node page, say *nl*, holds the underlying tree's routing keys. *nl* may overflow when new routing keys need to be inserted into *nl*, e.g., when one of *nl*'s child nodes splits and a new routing key needs to be inserted into *nl* that is already full. In this case, *nl* is split into two new nodes. Unlike in the case of the buffer tree [4], where it guarantees an empty buffer for the splitting internal nodes, AHA-tree does not hold this guarantee as only parts of a node's `nodeLSM-tree` is sent to children nodes during the Level-Emptying Process. In contrast, when the overflowed node page *nl* splits, pivots are assigned evenly to two new pages, e.g., in Figure 3, a page with 10, 21 and 45, and another page with the remaining routing keys. *nl*'s original `nodeLSM-tree` is split in the same way, i.e., two resulting `nodeLSM-trees` with one containing only the data items smaller than 47 and the other containing the remaining data items (that are greater than or equal to 47). A new routing key 47 is added to *nl*'s parent node page as shown in Figure 3.

Since node split requires compaction of `nodeLSM-tree`, which is expensive and blocking, we use *double buffering* (Section 4.4) and apply guarded compaction (Section 4.3) to reduce the overhead.

3.2.2 Tree Insert. In contrast to bulk inserting into the LSM-tree buffers of AHA-tree during the write-heavy workload, we may elect to perform a full tree insert for items, e.g., in the hotspot region. Refer to Figure 2. Tree insert can only happen in the hotspot region in Figure 2. When the hotspot region is completely adapted (adaptation to hotspot regions will be explained in Sections 5 and 6), no hotspot data items exists in any of the `nodeLSM-trees` of nodes above L4 to L9. Now, a new data item, e.g., Key 55, can be inserted into L6 (not shown in the figure) bypassing all the `nodeLSM-trees` in the path. Since all `nodeLSM-trees` are already free of data items belonging to the hotspot region, AHA-tree's freshness invariant still holds because, for the hotspot, the freshest data items are in the leaf pages, and there are none of them in the `nodeLSM-trees`.

3.3 Range Search

Recall that, in this investigation paper that tests the adaptation of indexes in oscillating workloads, for simplicity of presentation, we simulate the analytics read-heavy workload phase using simple range searches, and we control the amount being retrieved and used in the analytics operation by adjusting the search ranges.

To improve the latency of the analytics operations, range search queries are not batched as is the case in the original buffer tree [4] because we want to minimize the latency of the individual analytics queries. Initially when AHA-tree has not been adapted, the requested data items may reside in all `nodeLSM-trees` so we rely on a *merged iterator* to produce the sorted results. After AHA-tree has been completely adapted, only the leaf pages that have an overlapping range with the query need to be searched and we do not need to search any of the `nodeLSM-trees` in this case.

To protect reads from reading inconsistent results, we use read and wrte locks to protect each node and its `nodeLSM-tree`. During reading, the entire subtree that overlaps the queried range is read-locked. This forbids the `BGFlushThread` from modifying any `nodeLSM-trees` in that subtree. In the meantime, other readers can proceed as usual. This lock prevents all the readers from observing an inconsistent index until the read is completed.

4 OPTIMIZATIONS

In this section, we explain the optimizations for AHA-tree that address:

(1) How the initial tree structure in constructed (Section 4.1),
(2) The way to make write operations non-blocking (Section 4.2),
(3) Reducing the number of unnecessary and repeated file compaction operations (Section 4.3),
(4) Allowing for higher concurrency (Section 4.4), and
(5) Reducing the adaptation time (Section 4.5).

These optimizations are highlighted below.

4.1 Bottom-up Bulk-loading

Recall from Section 3.1 that we need to gradually push data in batches from the initial write-optimized `rootLSM-tree` at the root of AHA-tree down the tree. We rely on the idea that data in the levels of the LSM-tree is sorted except for data in Level-0 (unless a memory-based skiplist is used for the MemTable portion of the LSM-tree). Since we empirically set rootLSM-tree level limit to be more than one level, we use the sorted bottom levels of rootLSM-tree to push down and build the tree portion of AHA-tree. Data pushdown is triggered when the size limit of rootLSM-tree is reached. During the pushdown process, each file in the bottom level is assigned to one new leaf node. The boundaries of the files are used as routing keys for the new nodes. The idea is similar to that in [18] but the technical details are tailored for the AHA-tree as we explain below.

4.2 Non-blocking Writes in AHA-tree

In contrast to the other nodeLSM-trees in AHA-tree, rootLSM-tree has both a memory component and a disk component. Incoming writes are first added to rootLSM-tree so it is crucial that rootLSM-tree can finish its file compaction fast. A background thread, termed BGCompactThread, is responsible for compacting files in rootLSM-tree. When the size limit of rootLSM-tree is reached, the second background thread BGFlushThread is signaled to flush files downwards the tree. In order to further reduce the possible blocking of rootLSM-tree, a soft size limit is enforced to allow rootLSM-tree to exceed the limit temporarily so that the incoming write operations are not blocked in case the size is exceeded.

4.3 Guarded Compaction

Compaction can happen frequently both within nodeLSM-trees and in between nodeLSM-trees of parents and children. To reduce compaction costs, we apply guarded compaction to all inner-nodeLSM-tree compactions of non-leaf nodes. When nodeLSM-tree needs to compact its levels, the node page is used as one more input, and the resulting files can align with the routing keys. Later, during a Level-Emptying Process, if the node page has not been modified since, the bottom level of this nodeLSM-tree can be dispatched to children nodes without re-compaction. Dispatching files from a parent nodeLSM-tree to a child nodeLSM-tree is facilitated as we restrict that all the nodeLSM-trees except for rootLSM-tree do not have memory-based MemTable components. This way, in most cases, files can be migrated and rerouted directly from the bottom level of a parent node's nodeLSM-tree to the top level of a child node's nodeLSM-tree by pointer shuffling without reading the file into memory.

4.4 Double Buffering in AHA-tree

One main bottleneck for AHA-tree is the compaction operation as it requires sorting and rewriting files, and it blocks all concurrent reads on the same files. To resolve this issue, nodeLSM-tree compaction is delegated to a concurrent BGFlushThread thread. However, performing in-place compaction locks the node exclusively. We use double buffering to let BGFlushThread work in the background without write-locking any node. In double buffering, new writes are directed to a new nodeLSM-tree while reads take place in the original nodeLSM-tree. Only when compaction finishes does the BGFlushThread check the locking status, and atomically apply the compacted results. If there is any reader reading the nodeLSM-tree, BGFlushThread waits until the read is completed. BGFlushThread is first signaled by BGCompactThread when the size limit of rootLSM-tree is reached. Then, BGFlushThread compacts the overflowing nodeLSM-trees level by level adding the to-be-compacted nodes in a queue. During the Level-Emptying Process, files are first added to children nodes without any compaction. If some children nodes require an inner-nodeLSM-tree compaction, BGFlushThread compacts one of them in the background at one time. If later their nodeLSM-tree overflows, the Level-Emptying Process is invoked recursively and a minimal number of nodes are write-locked at any given time.

4.5 Hotspot-Emptying Process

Refer to Figure 2. In the figure, N1's nodeLSM-tree does not have hotspot data items. This is achieved by a hotspot-emptying process during the adaptation process (To be explained in the next section - Section 5). Automatic identification of hotspots is orthogonal to our study. In this paper, we assume that hotspots are detected online and are known in advance. There are researches that identify hot data from cold data, including [3, 12, 19, 29]. Within this hotspot range, only data that belongs to the hotspot can reach the leaf pages eventually to address read performance during the read-heavy phase of the workload. This alleviates the overhead of compacting the entire level of an nodeLSM-tree.

5 THE ADAPTATION PROCESS

As the workload shifts from being write-heavy to being read-heavy, AHA-tree needs to adapt accordingly to mostly behave as a B$^+$-tree. Automatic detection of when the workload changes is out of the scope of this work. However, once detected, we describe how the adaptation process takes place. This section addresses this issue.

The adaptation process is conducted by an independent background thread, termed BGFlushThread. For read-heavy workloads, the goal is to have all hotspot data items be stored in the leaf pages of AHA-tree. This helps avoid searching for data inside rootLSM-tree or inside any of the nodes' nodeLSM-trees, which is very time consuming. The whole process involves a *Hotspot-Emptying Process* and a *Leaf-Nodes Transformation Process.*

To migrate hotspot data from rootLSM-tree and the nodes' nodeLSM-trees, all the AHA-tree nodes from root to leaf having overlapping ranges with the hotspot range need to be checked. In order to avoid unnecessary adaptation, we only empty the nodeLSM-trees if these nodes are queried by some range search. During range search, if the tree traversal discovers a non-hotspot-free nodeLSM-tree, the node is recorded in a queue. BGFlushThread iterates over this queue, and finds that node in the tree. All the data items that belong to the hotspot are compacted, and then are sent down the tree to the children level. In Figure 4, files that are selected are compacted, and are sent to the destination children nodes. We do not force the data items to go directly into the leaf levels as there may be more levels in-between and until the leaf level, and this would invalidate the AHA-tree freshness invariant, and hence is avoided. Instead, hotspot data items are pushed one level at a time, and this can also help remove obsolete data items in the child level. This process is almost identical to the Buffer-Emptying Process except that the way to gather the files is different.

Leaf nodes with nodeLSM-trees are transformed differently. The goal is to rewrite this nodeLSM-tree into leaf pages. We have two

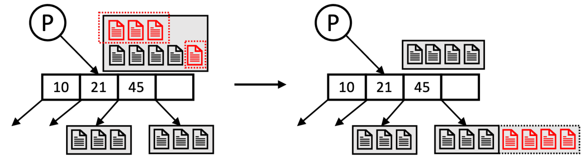


Figure 4: The adapting process

techniques to achieve that: *down-split* and *side-split*. Both techniques will be explained in Section 6.1. Once the leaf nodes are transformed into leaf pages, the new routing keys are added into the parent node, which may cause further split of the parent node.

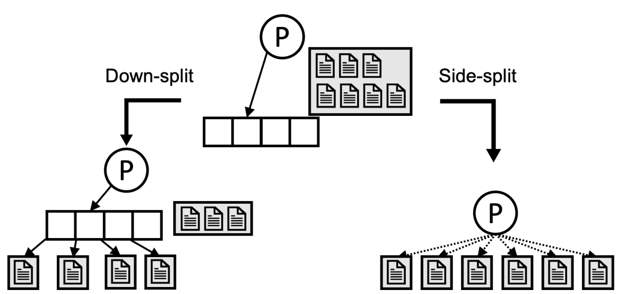
In the experimental study, we start the adaptation process when there is a range search query. Nodes that need to be processed are recorded in a shared work queue that can be visited by `BGFlushThread`.

6 ADAPTATION-RELATED OPTIMIZATIONS

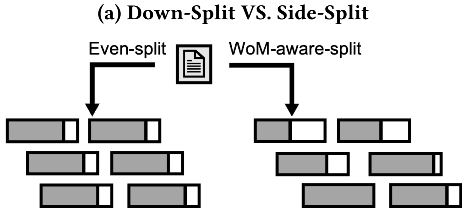
During the adaptation process, we can choose to adapt the leaf nodes in one of two ways. We compare them in Section 6.1. Also, we observe a phenomenon called *waves of misery* [13, 26] when merging the files with the tree pages. This is explained in Section 6.2.

6.1 Down-Split VS. Side-Split

We devise two ways to split a leaf node with `nodeLSM-tree`. One is termrd *down-split*, where the bottom level of `nodeLSM-tree` is extracted, and each file is assigned to a new node. The original leaf node becomes a non-leaf node with the remaining files in its `nodeLSM-tree`. This makes the tree unbalanced (See Figure 5a) as it adds one more level in the middle but does not require any compaction during split.



(a) Down-Split VS. Side-Split



(b) Even-split VS. Sound remedy-split

Side-split keeps AHA-tree in balance. It recompacts its `nodeLSM-tree` and obtains sorted files. Each one file is assigned to a leaf node and the routing keys are added in the parent. In Figure 5a, the connections between the parent node and the new nodes are in dashed lines because the parent node may need to split.

In both strategies, we take one more step to transform a leaf node with `nodeLSM-tree` to a leaf page. First, we transform it to a leaf node with only one file in its `nodeLSM-tree`. The reason for breaking down the steps is that a file size is typically larger than a tree page. Rewriting many files into tree pages may cause a bloating of routing keys. This causes AHA-tree to undergo severe structural modification that may cause unnecessary recompaction and delays.

6.2 Waves of Misery (WoM)

After a leaf node is transformed into a smaller leaf node with only one file in its `nodeLSM-tree`, this single file needs to be written into tree pages. The most straightforward way is to distribute the data items evenly into multiple pages. We refer to this strategy by *even-split*. However, this strategy may face an issue of waves-of-misery [13, 26] with future write operations. We can view data in uniform distribution if they are inside a small range. And the evenly distributed pages are of the same page utilization. When data are added in batches, these pages may overflow at the same time which doubles or triples the number of pages. These resulting pages may be of a low page utilization depending on the amount of added data. In turn, this degrades the range search performance. Thus, we follow the sound remedy introduced in [13, 26] to allocate data items into each page as shown in Figure 5b. Each page is not of a fixed utilization, and the goal is to split pages sequentially rather than all at the same time.

7 EXPERIMENTS AND EVALUATION

In this section, we evaluate AHA-tree against other indexes under oscillating write-heavy and read-heavy workloads and analyze the performance.

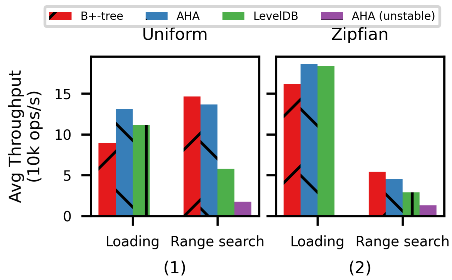
7.1 Experiment Setup

We use a 152-core machine having Intel(R) Xeon(R) Platinum 8368 CPU @ 2.40GHz with 197 GB installed Ubuntu 22.04.2 LTS of two NUMA nodes. We pin all our experiments in one NUMA node to eliminate NUMA-related performance issues.

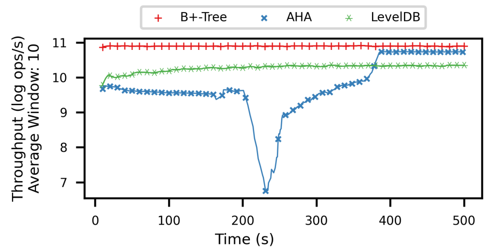
We use synthetic data in our experiments. Two data distributions are used: Uniform and Zipfian distributions. The key for one data item is a 20-byte string, and the value is a 128-byte string. We load the indexes with 500 million key-value pairs and the compared indexes are executed for the same amount of time or the same amount of operations. This is described in detail in each experiment.

7.2 Indexes Under Comparison

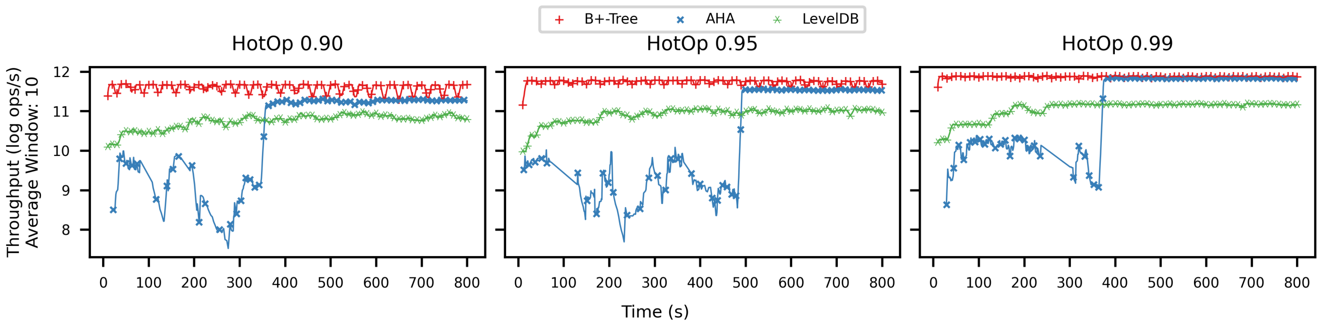
AHA-tree uses LevelDB [1] as its `rootLSM-tree` and `nodeLSM-tree`. Thus, we compare AHA-tree with LevelDB in the experiments. However, since the implementation of the buffer LSM-Trees is a pluggable component for AHA-tree, other LSM-Trees can fit as well. We choose LevelDB for its conciseness. We implement an in-house disk-based B$^+$-tree for the experiments. During the experiments, user threads send requests, including write operations and range search queries, to the index. AHA-tree uses two background threads during index modification and LevelDB uses one thread. For a fair



(a) Average throughput under Uniform and Zipfian data distributions



(b) Range search throughput after index loading overtime



(c) Change of ratios of hotspot operations

Figure 6: The performance after index loading

comparison with the B$^+$-tree, we allow the same number of threads running for each index at any given time. Only after the background thread(s) finish their work can the user thread(s) be awakened to send requests.

7.3 Range Search After Index Construction

First, we evaluate the range search performance after the indexes are constructed. We load each index with 500 million key-value pairs in the Uniform and Zipfian distribution cases. For the Uniform distribution, we choose a hotspot range of size 1% of the entire key-space. The hotspot start point is located randomly. For the case of the Zipfian distribution, we choose a smaller hotspot range of size 0.01% and anchor the hotspot at the beginning of the key-space that coincides with the highly ranked data items. The length of the range search query is 1000 for the Uniform distribution and 100 for the Zipfian distribution. A smaller length in the case of the Zipfian distribution is to have a similar size of the returned results in contrast with the Uniform distribution case.

Hotspot-100% Range Search. We show the results of index construction as well as the range search after index construction in Figure 6. All three indexes (AHA-tree, the B$^+$-tree, and the LSM-tree) are first loaded with the same amount of data. Their average throughput during loading is displayed in Figure 6a-(1) with Uniform data and the Zipfian data in Figure 6a-(2). AHA-tree has the highest average throughput as it has two background threads that can compact files concurrently. LevelDB is slower but is still faster than the B$^+$-tree. This is consistent for both the Uniform and the Zipfian data.

Starting at Time 0, the workload transitions to range search, we show the throughput overtime in Figure 6b and Figure 1 middle panel. The points in the figure are the results of a running average of Window Size 10 over a logarithmic scale of Base 10 of the recorded throughput. Figure 1 middle gives the results for Uniform data while Figure 6b gives the results for the Zipfian data. In both figures, AHA-tree can adapt to the hotspot. It takes about 300 to 400 seconds to adapt. During adaptation, AHA-tree shows a relatively low throughput. However, by the end of the adaptation process, the throughput of AHA-tree catches with that of the B$^+$-tree.

We compute the average throughput of the fully adapted AHA-tree and AHA-tree undergoing adaptation and plot them in Figure 6a. AHA-tree (unstable) denotes when adaptation is in progress and the throughput is low as the background thread is occupied to adapt the hotspot region into a B$^+$-tree. In both the Uniform and Zipfian data distributions, a fully adapted AHA-tree shows a throughput close to B$^+$-tree.

In either data distribution, AHA-tree is the first to finish loading the same amount of data with the same number of threads. In the following range query phase, the stable AHA-tree is slightly lower than the B$^+$-tree in throughput but is better than LevelDB.

Since we do not observe a different behavior between Uniform and Zipfian data except for the duration of the adapting process, we use Uniform data in the experiments to follow.

Hotspot-x% Range Search. Figure 6c compares the indexes when some range searches query cold spots while the majority of the queries query the hotspot. We show the comparison for 90%, 95% and 99% range search queries on hotspot. In the Figure 6c, when

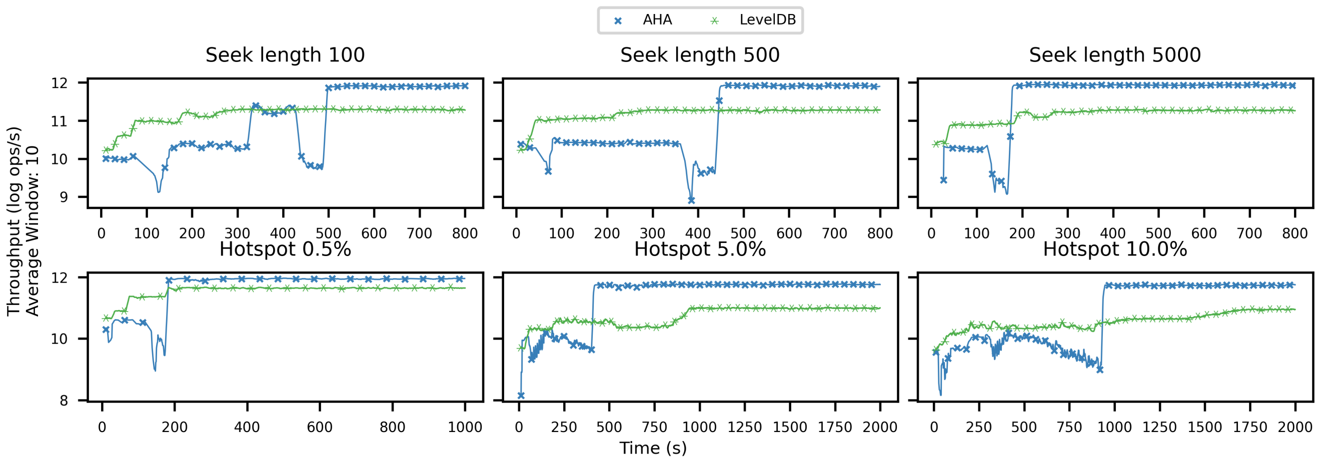
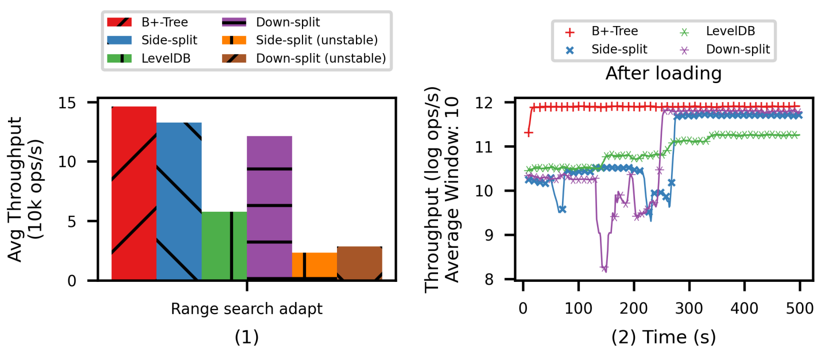
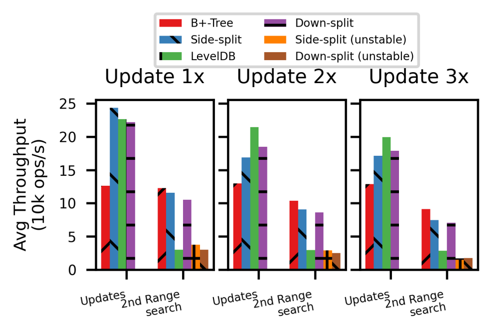


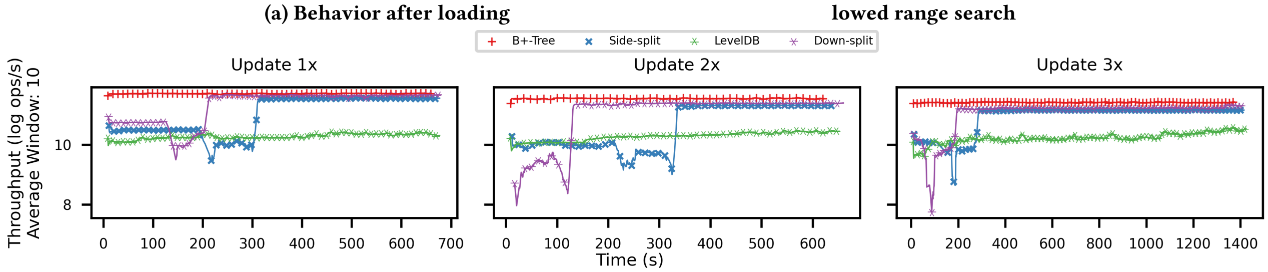
Figure 7: Comparison of seek-triggered compaction with adaptation



(a) Behavior after loading



(b) Average throughput of updates and the followed range search



(c) Range search throughput after updates overtime

Figure 8: Comparison between down-split and side-split

there are fewer operations on the hotspot, e.g., 90% and 95%, the discrepancy between AHA-tree and B$^+$-tree is larger. That is because cold spot range search need to search through the node`LSM-trees` for the cold data that qualify the search. In terms of the time taken to adapt the hotspot, all ratios do not exhibit significant differences.

7.4 AHA-tree VS. LSM-Tree

In LevelDB [1], the file compaction optimization can be triggered by excessive file reads. It helps compact frequently-read files to reduce I/O. A file is allowed to be seeked 100 times by default. If this number is exceeded, compaction is triggered to compact all the files of the overlapping range, and produce new filex. We observe the effect of this optimization in Figure 6b as the range search throughput gradually improves overtime, and stays stable afterwards. This suggests that the number of I/Os has decreased as there are fewer files to read. We compare this behavior with the adaptivity of AHA-tree. In Figure 7, we vary the length of the range search query in the first row. LevelDB shows an increase from 30k ops/sec to 78k-80k ops/sec, respectively, and stays stable. In the second row of Figure 7, we change the size of the hotspot from 0.5% of the entire key-space to 10%. AHA-tree takes longer time to adapt with a larger hotspot and LevelDB also reaches the stable point slower. If there are many queries confined in a range, seek-triggered compaction shows more obvious effect. However, AHA-tree still performs better than LevelDB after LevelDB's seek-triggered compaction has finished.

7.5 Performance Comparison of Post-Adaptation Indexes For Write-Heavy Workloads

Next, we show the performance of write operations on a fully adapted AHA-tree. After AHA-tree has been range queried for some time, the hotspot region has been completely transformed into B^+-tree. Then, the workload is oscillated back to be write-heavy by issuing only write operations on the hotspot only. Here, we focus on the hotspot writes only to stress AHA-tree. If writes are distributed in the cold and hot spots, it is expected that the average throughput would be close to the loading phase. All indexes are given the same number of write operations, followed by range search operations in the hotspot region. Figure 1(A) gives the average throughput during the updates-only phase. AHA-tree writes faster than the B^+-tree but slower than LevelDB. This is expected as all buffered writes undergo a level-emptying process. The reason is that the hotspot data needs to be merged with the existing hotspot pages. This merge is more expensive than appending buffers in the cold spot. During the adaptation phase, AHA-tree has a short period of low-throughput then the throughput recovers and catches up with the performance of the B^+-tree (Figure 1(B)).

Observe that all indexes exhibit lower range search throughput after the write operations compared to before the updates. This is reasonable as more data has been added into the hotspot region, and thus more data is to be reported, and hence the more time. Also, observe that AHA-tree shows a 36% degradation while the B^+-tree is only 30% worse. This is a phenomenon due to the Waves of Misery (WoM) described in Section 6.2 that we address in the next section.

7.6 Effect of Adaptation Optimizations

We study the effect on performance of the two adaptation optimizations presented in Section 6.

Effect of Leaf-Node Splitting Strategies. We compare the effect of the leaf-node down-split and side-split strategies on performance. The results of this comparison are given in Figure 8. In Figure 8a-(1), we show the effect of adapting AHA-tree after index construction. The average throughput of both stable down-split and stable side-split do not show much difference in the range search adaptation, and neither are their unstable throughputs (Figure 8a-(1)). Also, the time needed to adapt does not differ much as shown in Figure 8a-(2).

Next, we compare how future write operations over the adapted index perform under either split strategy. We use three different update sizes: 1×, 2× and 3× of the hotspot size. With the increase in the update size, the average throughput of AHA-tree drops especially when the leaf node is side-split. AHA-tree with down-split updates faster than the side-split one (Figure 8b). As the updated data are later merged with the adapted leaf pages, this is more costly than the Level-Emptying Process between the internal nodes. With down-split, a new layer of nodes is created that allows for more room to buffer the incoming data. Thus, it is faster and quicker to absorb new updates in the case of a down-split than in the case of a side-split. Figure 8c compares the range search throughput after updates over time. Down-split is faster in adapting than side-split as it does not involve compaction. The eventual throughput does not show much difference between side- and down-splits.

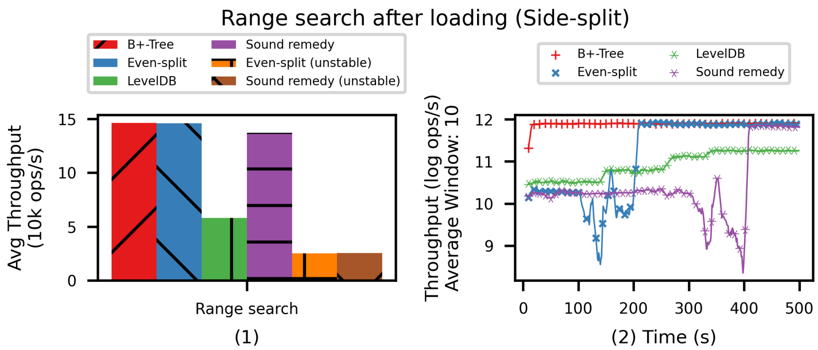
Effect of Leaf Allocation. We compare the performance of two different leaf allocation strategies: even-allocation and sound-remedy-allocation. First, we compare their average throughput for range search when leaf nodes are side-split (See Figure 9a). There is no obvious difference in the average throughput between even-allocation and sound-remedy-allocation in Figure 9a-(1). However, sound-remedy-allocation takes more time to fully adapt as shown in Figure 9a-(2). The reason is that sound-remedy-allocation needs to compute the capacity of each page, and may need to allocate more pages than even-allocation.

After the indexes are completely adapted in the range search, update operations are applied to the indexes. We can insert data in two ways: batch-insert or tree-insert. We compare both in Figure 9b. Tree-insert is the slowest as data is inserted one item at a time. Upon page overflow, internal nodes with buffers are split, which makes this technique even slower than the B^+-tree updates. Buffer-insert can insert faster but is still slower than LevelDB.

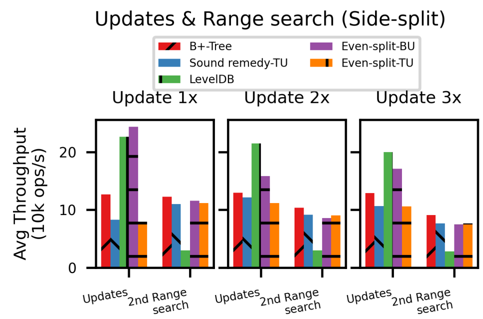
Then, we compare the two allocation techniques when leaf nodes are down-split (Figure 9c and Figure 9d). The overall trend is similar to what is observed for the side-split case.

The range search throughput over time shows that tree-update does not require an adapting period and the performance is nearly as good as the B^+-tree (Figures 9e and 9f). There is a trade-off between current write throughput and future read throughput, i.e., we can sacrifice current write performance in return for no needed adaptation for the future read-heavy workload.

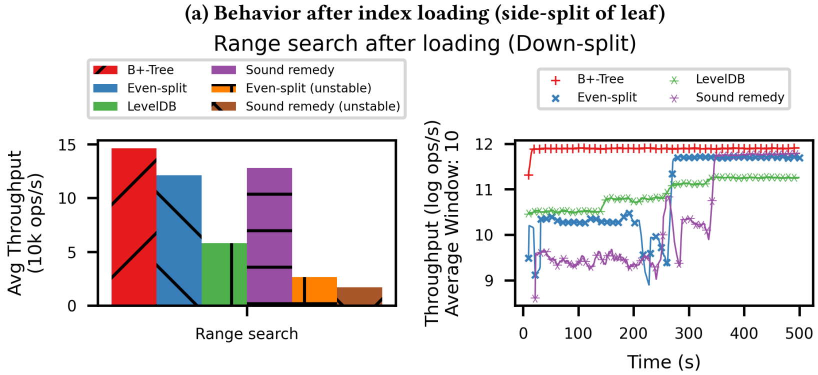
Observe that the throughput of the different allocations varies. The variation is due to differences in the leaf page utilization. If all leaf pages are packed full, all relevant range queries touch the smallest number of pages. We then plot the leaf page utilization distribution in Figure 10. The page distribution of the B^+-tree is displayed in Figure 10g. This distribution matches the ideal result of sound remedy in [13, 26] which guarantees page distributions are spread and are not clustered. Thus, the B^+-tree will not suffer from Waves of Misery (WoM). If AHA-tree leaf pages are allocated by sound remedy and later inserted by tree-insert, the resulting distribution of leaf page utilization is in Figure 10e and 10f demonstrating that WoM does not occur. Since they do not suffer from WoM, their performance is almost as good as the B^+-tree. If the pages are allocated by even distribution (in contrast to sound remedy), there will be a utilization ratio that the majority of the pages belong to. This majority utilization ratio depends on the number of writes applied on AHA-tree. For example, when there are 1× updates, this peak utilization ratio is 100% suggesting that the majority of the pages are full. Thus, the range search time is low. For 2× updates, the page utilization distribution is more uniform within 78% to 100%. Thus, the range search may touch more pages, and the needed search time is higher (Figures 10a and 10b).



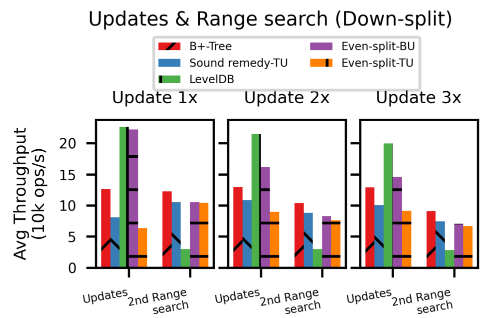
(a) Behavior after index loading (side-split of leaf)



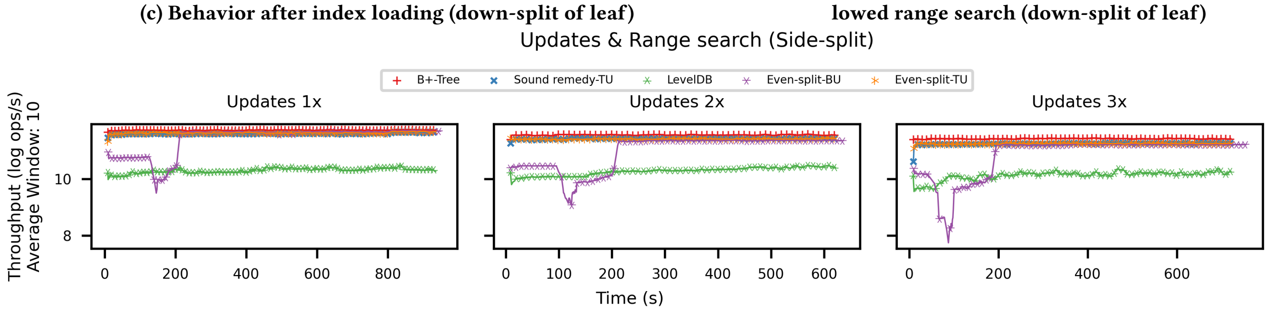
(b) Average throughput of updates and the followed range search (side-split of leaf)



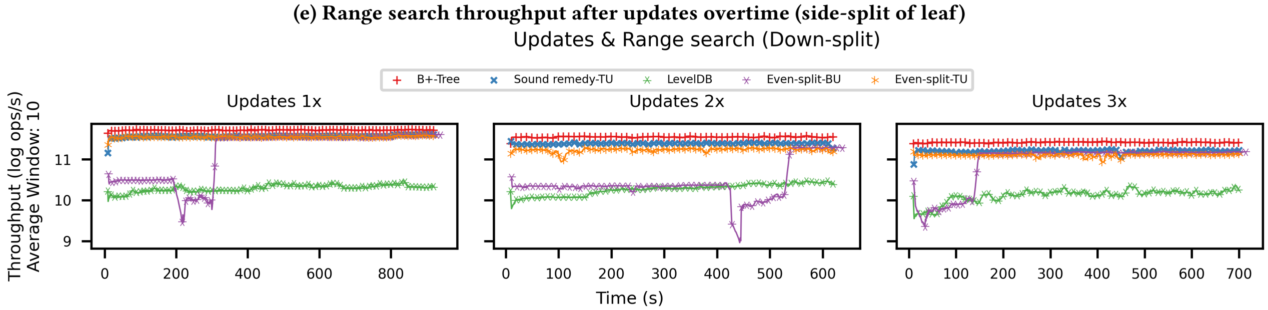
(c) Behavior after index loading (down-split of leaf)



(d) Average throughput of updates and the followed range search (down-split of leaf)



(e) Range search throughput after updates overtime (side-split of leaf)



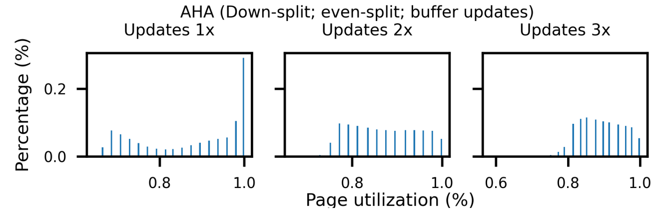
(f) Range search throughput after updates overtime (side-split of leaf)

Figure 9: Comparison of leaf allocation techniques

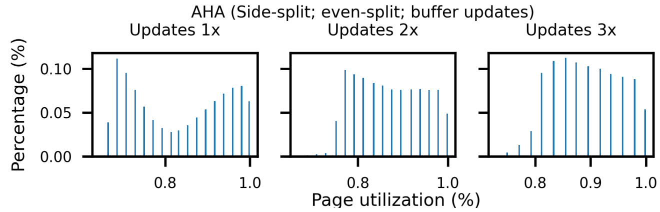
7.7 Mixed Read-Write Operations

Since write-heavy (read-heavy) workloads may still include read (write) operations, we evaluate the indexes under mixed workloads. All mixed workloads start after the indexes have been constructed.

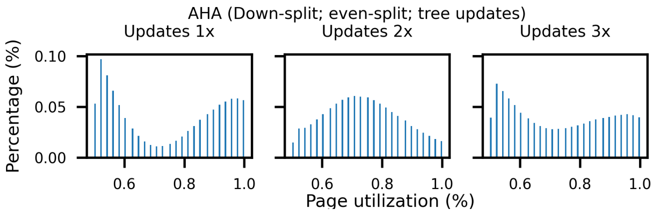
Workload A starts with 90% range search plus 10% updates, then 10% range search plus 90% updates, followed by 90% range search plus 10% updates, then 10% range search plus 90% updates. Each phase takes 300 seconds. AHA-tree adapts itself only under 90%



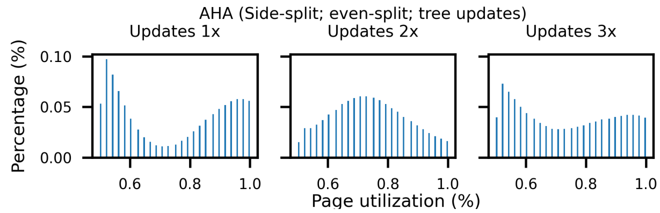
(a) Down-split with even-allocation then buffer-updates



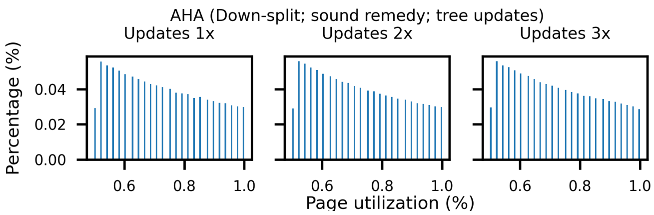
(b) Side-split with even-allocation then buffer-updates



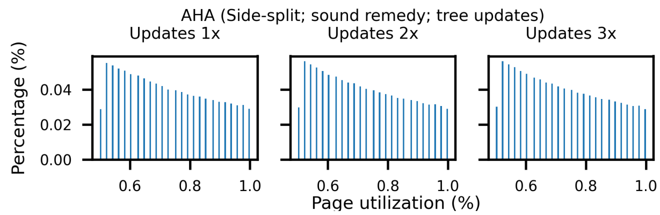
(c) Down-split with even-allocation then tree-updates



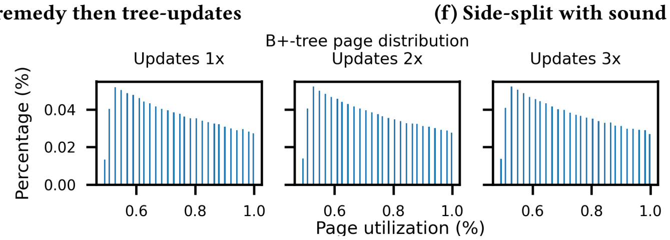
(d) Side-split with even-allocation then tree-updates



(e) Down-split with sound remedy then tree-updates

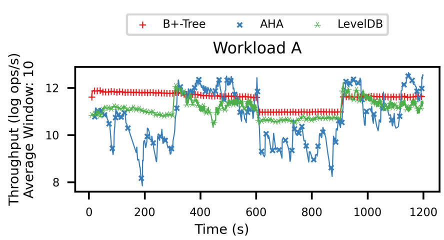


(f) Side-split with sound remedy then tree-updates

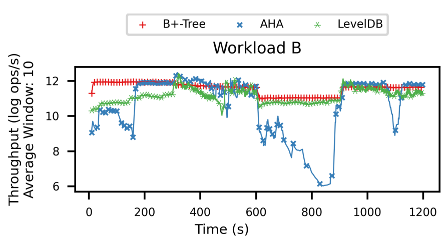


(g) Leaf page distribution of B$^+$-tree

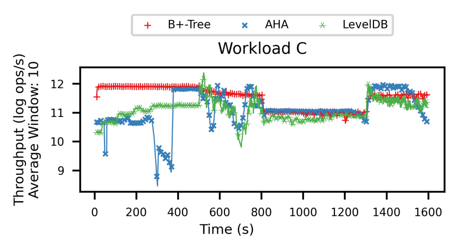
Figure 10: Leaf page distribution of AHA-tree



(a) Throughput of workload A



(b) Throughput of workload B



(c) Throughput of workload B if updates are inserted by tree-update

Figure 11: Performance under a mixed oscillating read-write workloads

range search. The result in Figure 11a shows that AHA-tree does not perform well if adapting and writing happen at the same time (0-300 sec and 600-900 sec). AHA-tree can have a relatively good throughput under 90% updates (300-600 sec and 900-1200 sec).

Since adaptation and writes compete against each other, we use a fully adapted AHA-tree. Workload B includes 300 sec of 100% range search s.t. AHA-tree can be adapted completely. This is followed by 300 sec of 10% range search plus 90% updates. In Figure 11b, AHA-tree adapts itself during the 100% range search phase and its overall throughput remains high until the workload becomes 90% range search plus 10% updates at Time 600 sec. AHA-tree writes and adapts during this phase, and its throughput becomes lowest among all. Since tree-insert can avoid the problem of writing and adapting at the same time, we conduct a similar experiment in Figure 11c using tree-insert. From time 500 sec to 800 sec where the workload is 10% range search plus 90% updates, AHA-tree is unstable during this phase. In the next 90% range search plus 10% updates phase, AHA-tree has the highest throughput. When the ratio of updates increases, the throughputs of both AHA-tree and LevelDB fluctuate.

8 LESSONS LEARNED

We summarize the lessons learned and recommendations from the experiments as follows:

- If the workload oscillates between read-heavy and write-heavy rapidly, non-adaptive indexes like the B$^+$-tree and LSM-tree will be a good choice as AHA-tree takes time to adapt.
- If the relatively low throughput is not desired during the adapting period, non-adaptive index is a better choice.
- If the read-heavy workload requires a relatively high workload and the adapting period can be tolerated, AHA-tree is a good choice as its eventual throughput can be as high as the B$^+$-tree but AHA-tree's load and update throughput are higher.
- If the duration of the write-heavy workload is short, using tree-insert is favored in AHA-tree as it maintains the hotspot structure, and does not require further adaptation.
- If the duration of the write-heavy workload is long, using batched-insert in AHA-tree can improve write throughput.

9 RELATED WORK

Numerous studies have been conducted for improving the write performance of tree-like structures. In order to minimize individual I/O, the buffer tree [4] batches multiple operations (insertion, deletions and range searches) into one segment in the memory. Later this in-memory block is added to the buffer of the root and may trigger buffer emptying process with new data finally lands into the leaf page. Graefe has proposed write-optimized B-tree on top of traditional B-trees in [14]. This design makes page migration inexpensive in log-structured systems, and supports both in-place updates and large append-only write operations at the same time. This does not need an indirection layer to locate a B-tree on disk. Bϵ-tree [6] is a tree structure where some space of the tree node is allocated to a buffer. The buffer stores updates that are eventually applied to the leaf nodes. ϵ decides the amount of space for pivots and the remaining amount for buffer. [27] presents a Nested B-tree where each B-tree node contains a B$^+$-tree.

Since the LSM-tree sacrifices read performance in exchange of better write performance, researches have been conducted research to improve the LSM-tree's read performance. The Bloom filter [7] can greatly improve point reads of LSM-tree by filtering and reducing unnecessary I/Os. However, the Bloom filter cannot filter range searches. Rosetta [20] uses a hierarchically-stacked Bloom filters, and each range query is converted into multiple probes into the Bloom filters. REMIX [30] improves range searches by adding a sorted view across multiple files in the LSM-tree s.t. range search can find the target key using binary search and retrieve following keys in order without comparison.

Database Cracking [17]reorganizes data entries with incoming queries. This technique is based on an observation that only when data is queried should it be necessary to sort (reorganize) data. The initial database cracking is introduced in a column-store but can also be applied to row-stores [17]. In order to support dynamic databases in database cracking, [16] proposes several algorithms to update a cracked database.

There are other adaptive indexes that adapt to the changes in workload [9, 23], or in the data distribution [3, 19, 29]. Adaptive Hybrid Indexes [3] are proposed to address data hotspots s.t. cold data can be more compressed to make room for the less compressed hot data. This technique has been shown effective both in the B$^+$-tree and in the trie [3]. VIP-hashing [19] deals with the data hotness issue but in the context of the hash table. Data distribution is learned and compared as more queries come. Then, the access path of hot data can be reduced on the fly. SA-LSM [29] can adapt itself to long-tailed data, that data popularity decreases overtime. SA-LSM uses survival analysis to train a model to predict the next access time of the data item, and moves cold data into slow storage during LSM-tree compaction. A real-time LSM-tree termed LASER is developed [23]. The observation is that recent data is stored in row-store for OLTP while older data is stored in a column-store for OLAP. LASER allows different data layouts in different levels of the LSM-tree. B^{link}-hash [9] solves the issue of inserting monotonically increasing keys by using a hash table as a leaf node in the tree. Later, the hash table is adapted to a leaf page upon queries.

10 CONCLUSION

In this paper, we present a kind of workload that oscillates between write-heavy and read-heavy. This workload is observed in many real-world applications. Traditional non-adaptive indexing and adaptive indexing cannot fit into this workload well as they are either non-adaptive or adapt in only one direction. With the observation that real data sets are skewed, we focus only on the hotspot. We encapsulate the adapting techniques in the index AHA-tree to make it adaptive in both directions. In our investigation, we evaluate AHA-tree against traditional indexes, and present the pros and cons for each adaptation strategy that is helpful to deal with the oscillating workload.

REFERENCES

[1] [n. d.]. *LevelDB*. https://github.com/google/leveldb

[2] [n. d.]. *RocksDB*. https://github.com/facebook/rocksdb

[3] Christoph Anneser, Andreas Kipf, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. 2022. Adaptive hybrid indexes. In *Proceedings of the 2022 International Conference on Management of Data*. 1626–1639.

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

[5] Rudolf Bayer and Edward McCreight. 1970. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. 107–141.

[6] Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. 2015. An introduction to B-trees and write-optimization. *login; magazine* 40, 5 (2015).

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

[8] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[9] Hokeun Cha, Xiangpeng Hao, Tianzheng Wang, Huanchen Zhang, Aditya Akella, and Xiangyao Yu. 2023. Blink-hash: An adaptive hybrid index for in-memory time-series databases. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1235–1248.

[10] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[11] Anas Daghistani, Walid G Aref, Arif Ghafoor, and Ahmed R Mahmood. 2021. Swarm: Adaptive load balancing in distributed streaming systems for big spatial data. *ACM Transactions on Spatial Algorithms and Systems* 7, 3 (2021), 1–43.

[12] Jiake Ge, Huanchen Zhang, Boyu Shi, Yuanhui Luo, Yunda Guo, Yunpeng Chai, Yuxing Chen, and Anqun Pan. 2023. SALI: A Scalable Adaptive Learned Index Framework based on Probability Models. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–25.

[13] Nikolaus Glombiewski, Bernhard Seeger, and Goetz Graefe. 2019. Waves of misery after index creation. (2019).

[14] Goetz Graefe. 2004. Write-optimized B-trees. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 672–683.

[15] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn.. In *CIDR*.

[16] Stratos Idreos, Martin L Kersten, and Stefan Manegold. 2007. Updating a cracked database. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 413–424.

[17] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. 2007. Database Cracking.. In *CIDR*, Vol. 7. 68–78.

[18] Varun Jain, James Lennon, and Harshita Gupta. 2019. Lsm-trees and b-trees: The best of both worlds. In *Proceedings of the 2019 International Conference on Management of Data*. 1829–1831.

[19] Aarati Kakaraparthy, Jignesh M Patel, Brian P Kroth, and Kwanghyun Park. 2022. VIP hashing: adapting to skew in popularity of data on the fly. *Proceedings of the VLDB Endowment* 15, 10 (2022), 1978–1990.

[20] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2071–2086.

[21] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[22] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.

[23] Hemant Saxena, Lukasz Golab, Stratos Idreos, and Ihab F Ilyas. 2023. Real-time LSM-trees for HTAP workloads. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1208–1220.

[24] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 217–228.

[25] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-store: a column-oriented DBMS. *Proceedings of the VLDB Endowment* (2005), 553–564.

[26] Lu Xing, Eric Lee, Tong An, Bo-Cheng Chu, Ahmed Mahmood, Ahmed M Aly, Jianguo Wang, and Walid G Aref. 2021. An experimental evaluation and investigation of waves of misery in r-trees. *arXiv preprint arXiv:2112.13174* (2021).

[27] Sepanta Zeighami. 2019. *Nested b-tree: efficient indexing method for fast insertions with asymptotically optimal query time.* Ph. D. Dissertation.

[28] Chao Zhang, Guoliang Li, and Tao Lv. [n. d.]. HyBench: A New Benchmark for HTAP Databases. ([n. d.]).

[29] Teng Zhang, Jian Tan, Xin Cai, Jianying Wang, Feifei Li, and Jianling Sun. 2022. SA-LSM: optimize data layout for LSM-tree based storage using survival analysis. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2161–2174.

[30] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. {REMIX}: Efficient Range Query for {LSM-trees}. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 51–64.