# Elastic Indexes: Dynamic Space vs. Query Efficiency Tuning for In-Memory Database Indexing

Moshik Hershcovitch
Tel Aviv University & IBM Research
Israel

Artem Khyzha
Tel Aviv University
Israel

Daniel Waddington
IBM Research
USA

Adam Morrison
Tel Aviv University
Israel

## ABSTRACT

In-memory database management systems (DBMSs) are important elements of data pipelines, wherein they store data produced over a sliding (or periodic) time window by real-time data sources for analytics and monitoring purposes. These pipelines produce vastly different amounts of data across time windows, dictating that some memory over-provisioning is required for the DBMS. A major source of DBMS memory overhead—which stresses memory provisioning demands—is storage of indexed keys by the DBMS index data structures. However, such *internal-key storage* is crucial for fast scans, which are important in these workloads.

This paper proposes an **elastic index** design framework, which transforms an index with internal-key storage into an elastic index that adjusts its memory overhead to the data size. Under typical conditions, the elastic index offers the same performance as the original index. When large amounts of data are ingested, the elastic index temporarily shrinks so that the DBMS does not exceed its memory budget. Shrinking is performed by dynamically converting index nodes to a compact representation with indirect key storage.

We demonstrate our design with an **elastic B$^+$-tree**, whose compact nodes use a novel blind trie representation. We show that the elastic B$^+$-tree can store 2×–5× the number of keys (depending on key size) than a B$^+$-tree with only moderate (< 25%) degradation of index throughput. We also integrate the elastic B$^+$-tree into the MCAS in-memory storage system. Compared to MCAS' default B$^+$-tree index, the elastic B$^+$-tree can consume 3× less space with 1.8%–2.6% throughput degradation on a workload modeling analysis and monitoring of cloud log data.

## 1 INTRODUCTION

In-memory database management systems (DBMSs) [8, 9, 11, 26] store the entire database in main memory and thereby avoid the overhead and complexities of working with slow, block-based storage media. An emerging use case for in-memory DBMSs is as elements of a *data pipeline* [5, 27]. Here, the in-memory DBMS stores data produced *over a sliding (or periodic) time window* by real-time data sources (e.g., transactions, sensors, logs, etc.) During this window, the DBMS is used for hybrid transactional/analytic processing (HTAP) to drive analytics and decisioning tasks, such as personalization, fraud detection, monitoring and detection, etc. [5, 24, 28]. As data ages, becoming less relevant for these tasks, it is moved from the in-memory DBMS into longer-term slower storage.



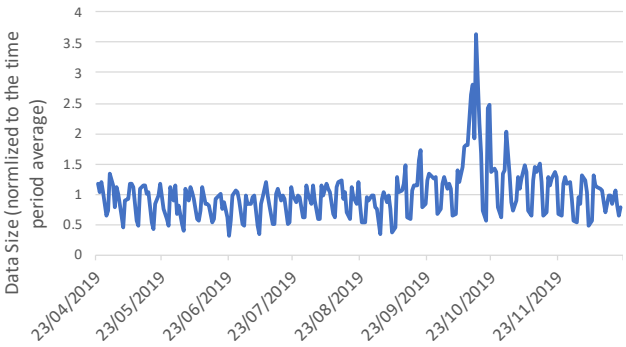



**Figure 1: Data pipeline size variability: Size of daily data extracted from a cloud provider's object storage system logs, over a period of 8 months, normalized to the average size over that period. (Absolute numbers omitted for confidentiality.)**

The memory requirements of an in-memory DBMS element of a data pipeline can vary greatly across time windows. For instance, Figure 1 shows the size of the data extracted each day (for monitoring and analytics) from the logs of a commercial cloud provider's object storage system, which is used by both customers and internal clients. There are many days in which the size of the data is 1.5× that of the average data size over the reported period, and in some days the data size exceeds the average by 2×–3.5×.

This data size variability dictates that in-memory DBMS pipeline elements need to be over-provisioned with memory (with respect to their average utilization). It is desirable that such over-provisioning be minimal, for cost and efficiency reasons. However, DBMS index data structures, particularly secondary indexes, impose significant memory overhead. For instance, the above workload contains many high-cardinality columns that require indexing, resulting in index sizes that are roughly the same size as the data set—i.e., indexes take up ≈ 50% of DBMS memory. Prior work and industry report similar index overhead numbers [23, 33].

A major source of index memory overhead is from storing copies of the indexed keys, as occurs in B$^+$-trees [6], skip lists [25, 32], and BwTrees [18, 31]. However, this design choice allows the index to store keys in-order and thereby provide extremely fast, cache-efficient scan operations, which are important for data pipeline analytics workloads [15, 24]. Therefore, the choices available today for addressing index memory overhead are to drop indexes under memory pressure, which destroys query performance, or to use specialized compact indexes [3, 33], which are suboptimal for our target workloads (§ 2).

**Elastic indexes** This paper proposes to rein in index memory overhead by making it **elastic**. We propose an **elastic index** design framework, which transforms an index with internal-key storage into an elastic version.

Under normal conditions, the elastic index is identical to the standard index, leveraging the extra space provided by limited memory over-provisioning for its key storage and thereby offering optimal query performance. However, when a large amount of data is ingested, the elastic index temporarily shrinks itself until memory pressure subsides. To dynamically shrink itself, the elastic index switches a portion of the nodes to a compact representation with indirect key storage. Once memory pressure subsides, the converted nodes gradually revert back to their original form, storing keys internally.

The elastic index framework can be applied to any index with internal key storage, such as a B$^+$-tree, skip list, or Bw-Tree. The design is parameterized by (1) the *elasticity* algorithm, which adjusts the index's space utilization by deciding when and which nodes should start/stop using indirect key storage; and (2) the compact node representation, which determines the space vs. query efficiency tradeoff.

To demonstrate the design framework, we devise an elastic B$^+$-tree. For the elasticity algorithm, we propose a method that piggybacks on node splits and merges as the point in time to convert regular nodes to/from blind tries [13]. For the compact node representation, we propose a novel blind trie representation. Our blind trie obtains space efficiency comparable to the most compact blind trie in the literature [12] but with query complexity comparable to that of faster, but less compact, representations [4, 13].

We evaluate the elastic B$^+$-tree using uniform and YCSB workloads, as well as by integrating it into the MCAS in-memory data store. We find that an elastic version of the STX B$^+$-tree [2] can store 2×/5× the number of 8-byte/30-byte keys with only a 25% throughput degradation. To evaluate the space/performance trade-off of the elastic B$^+$-tree in a full system, we evaluate the MCAS [29] in-memory data store on a workload modeling ingestion and querying of cloud log data. Depending on the elasticity parameters, the elastic B$^+$-tree consumes from 0.76× to 0.3× the space of the default B$^+$-tree index, with only 0.5% to 2.6% degradation in query throughput.

**Contributions** We make the following contributions:

(1) The elastic index design framework, which enables stretching a fixed memory budget while gracefully degrading when large amounts of data are consumed.
(2) Applying the framework to construct an elastic B$^+$-tree.
(3) A novel blind trie representation, which serves as the main building block of the elastic B$^+$-tree.
(4) Empirically evaluating our algorithm against the state of the art indexes in the literature.

## 2 THE CASE FOR ELASTIC INDEXES

We make the case for elastic indexes by explaining why existing approaches for addressing DBMS index size overheads are suboptimal for our setting. We consider only *ordered* indexes—such as B$^+$-trees or tries—that support iteration, range selection, etc. in addition to point queries. The reason is that in-memory DBMSs serve as data pipeline elements to support analytics, monitoring, and decisioning queries, which require multiple ordered secondary indexes.

**Giving up internal-key storage?** Internal-key storage is fundamental to the performance of index operations. In principle, comparison-based in-memory indexes such as B$^+$-trees could avoid internal-key storage by using indirection, i.e., by storing pointers to keys' tuples and loading a key from the database when an index operation needs to read it for a comparison. However, such a design would eliminate the spatial locality of having keys co-located in index nodes, which means that any key access by the index would incur CPU cache misses, severely slowing down index operations.

A similar problem exists with index designs based on Patricia tries [21], such as HOT [3]. These indexes do not rely on key comparisons and so are able to store keys indirectly without compromising index point query performance. Unfortunately, large scans—which are common in analytics workloads [15, 24]—slow down significantly if each scanned key has to be loaded from the database (see § 6). Included-column queries [20]—i.e., whose result is computed solely from the values composing the key and the included columns—are similarly slowed down.

**Using prior index compression schemes?** The elastic index approach provides better trade-offs than prior index node compress schemes. The overhead of general-purpose compression [22]—i.e., decompressing and re-compressing each accessed index node—is significantly higher than the indirect key accesses performed on nodes compacted by an elastic index. While prefix compression [14, 23]—which compresses a B$^+$-tree leaf by storing redundant prefixes only once—is cheap, its compression ratio depends on the key distribution and it might even increase space usage if keys do not share common prefixes. In contrast, switching a node to a compact representation always saves space.

**Using hybrid indexes?** Hybrid indexes [33] improve index space efficiency using a two-stage architecture, in which recently added data is held in a small dynamic index while a compact, read-only index holds the bulk of the index entries. A periodic merge process migrates aged entries from the small dynamic index to the compact index. Because the compact index does not support individual-key updates, the merge process must rebuild it entirely. In contrast, the elastic index design is fine-grained, converting individual nodes to/from a compact representation, and supports updates of the compact nodes. Moreover, the efficiency of the hybrid index approach is predicated on a skewness assumption—that updated index entries are those recently inserted. The elastic index design does not require this assumption.

## 3 ELASTIC INDEX FRAMEWORK

The elastic index framework takes a *baseline* index whose nodes internally store indexed keys (e.g., a B$^+$-tree or skip list) and transforms it into an *elastic* index. Under memory pressure, an elastic index can trade off some query efficiency for better space efficiency by dynamically converting its data structure nodes into a compact representation, which stores keys indirectly instead of explicitly.

Our framework targets the baseline index's *leaf* nodes, which are where index searches terminate, because these nodes occupy most of the space in the index. The observation underlying the elastic index transformation is that such leaves can be viewed as "mini indexes"—as they map from keys to database tuples—with their own abstract data type (ADT) interface. For example, the B$^+$-tree leaf ADT supports six familiar operations [6]: insert, remove, find, predecessor/successor, split, and merge. Therefore, a compact node representation that implements the leaf ADT can

*co-exist* in the index with the standard leaf nodes, without any changes to the baseline index algorithm. In turn, the co-existence of regular and compact leaves allows devising an *elasticity* algorithm that responds to changing memory pressure by switching leaves to/from their compact representation.

The elastic index framework is parameterized by the following:

- The **compact node representation**. The representation must support the original index's leaf operations. Designing a good representation requires carefully balancing compactness and the efficiency of leaf operations. While the elastic index trades off some query efficiency for space efficiency, the goal is to minimize impact on query efficiency.
- The **elasticity algorithm**. This algorithm monitors index space consumption. In response to an increase in the database size, it begins converting leaves to the compact representation. When the database size reduces, it converts compact leaves back to the normal representation. The algorithm must respond to change in memory consumption as quickly as possible while minimizing interference on index operation under normal conditions. The main challenge in designing an elasticity algorithm is how to pick the leaves that will be compacted.

Choosing these "parameters" creates a large design space. Here, we initiate exploration of the design space by presenting an elastic B$^+$-tree based on two main ideas: (1) basing the compact representation on a novel *blind trie* [13] representation (§ 5) and (2) having the elasticity algorithm piggyback on B$^+$-tree split/merge operations as the point to efficiently convert leaves to/from the compact representation (§ 4).

## 4 B$^+$-TREE ELASTICITY ALGORITHM

Our B$^+$-tree's elasticity algorithm operates by striving to keep the size of the index *constant* once it reaches a certain threshold, and leveraging the conversion of leaves to their compact representation to index more keys with a fixed memory budget. The idea is to detect a window of time in which significantly more data than usual is ingested. In such a case, the index is temporarily shrunk to make room for the extra data.

The algorithm is configured with a soft *size bound* that indicates the maximum size the index should be allowed to grow to. The algorithm attempts to keep index size below this bound. When the index size grows close to the bound (e.g., reaches 90% of it), the algorithm enters a *shrinking* state and starts converting leaves to the compact representation, which can index more items without increasing the index's overall memory consumption. When the index size decreases because the data set size decreases, the algorithm enters an *expansion* state and starts converting leaves back to the standard representation. To prevent scenarios of oscillation between shrinking and expansion, the algorithm uses different thresholds for triggering shrinking and expansion. That is, the algorithm switches from shrinking to expansion only when the index size decreases far enough from the size bound.

The elasticity algorithm works by compacting the index incrementally as the dataset grows, resulting in compact and standard nodes co-existing in the same tree. This approach minimizes overhead on index operations, in contrast to existing approaches that compact the *entire* index in bulk [33], which can take significant time. Our incremental approach also results in a smoother space/efficiency trade-off, as only queries touching compact nodes become slower.

The elasticity algorithm relies on a *grow/shrink policy* to select which leaves to compact/decompact when the index grows/shrinks. In the following, we describe the algorithm with a policy that chooses leaves which experience B$^+$-tree overflow/underflow events due to insertions/removals. There is, however, a design space of possible policies. For example, the policy could pick infrequently accessed nodes for compaction, to minimize the impact on query speed. We leave exploration of different policies to future work.

The elasticity algorithm assumes that the compact leaf representation provides leaves of varying *capacity*, i.e., maximum number of keys that can be stored. The capacity of a leaf determines its size; the more compact the representation, the greater capacity that can be offered by a (compact) leaf of size $S$. The elasticity algorithm requires that a compact leaf with capacity of $2n$ keys be smaller than a standard B$^+$-tree node with capacity of $n$ keys. (Blind trie representations, including ours, typically satisfy this property.) Other than these requirements, the elasticity algorithm is agnostic to the compact representation.

**Shrinking** When index size grows close to the size bound, the elasticity algorithm enters a *shrinking* state. In the shrinking state, the algorithm's goal is to absorb insertions while (1) reducing the memory consumed by leaf nodes and (2) not increasing memory consumption by inner nodes. We achieve this by modifying the way in which a leaf overflow is handled. Normally, an insertion about to overflow a leaf will split the leaf into two new leaves, which requires further insertions of separators in the leaf's ancestor inner nodes. Our algorithm piggybacks on this operation, which is already costly and infrequent[1] and replaces the leaf through a compacting operation.

In the shrinking state, upon an insertion attempt into a full $n$-key standard leaf, the elastic B$^+$-tree replaces the leaf with a compact node with a capacity of **2n keys** (instead of splitting). The compact node is initialized to contain the original leaf's $n$ keys and tuple identifiers. The insertion is then performed into the new compact leaf. As a result, we save space in two ways: the new compact leaf is smaller than the original leaf, and no insertions—possibly causing more splits—need to be performed at the upper layers of the tree.

Subsequent insertions into the compact node might drive its occupancy to the capacity, creating an overflow. Typically, we handle the overflow as before, by replacing the overflowing compact leaf with a compact leaf with double the capacity. While this increases the size of the leaf, it still saves changes in the inner nodes. However, as a compact leaf grows larger, queries on it become slower (§ 5). Therefore, the elastic B$^+$-tree caps the maximum capacity that a compact leaf may reach due to overflow handling. An overflow of a compact leaf with maximum capacity results in a split of that leaf. In practice, we have found that starting from a capacity of 16 keys and capping it at 128 keys works well (§ 6).

The elasticity algorithm maintains an invariant for compact leaves that is similar to the invariant of B$^+$-tree leaves, namely, that a compact leaf with a capacity of $2k$ keys must contain at least $k + 1$ keys. A remove operation that would bring a compact leaf beyond its lower threshold—cause an underflow—either replaces the compact leaf with a compact leaf with capacity $k$, if $k$ is greater than the capacity of a B$^+$-tree leaf, or simply replaces the compact leaf with a standard B$^+$-tree leaf.

[1] The amortized cost of a split in a B$^+$-tree is $O(1)$.

**Expansion** When index size decreases far enough from the size bound, the elasticity algorithm enters an *expansion* state, in which it strives to regain query performance by switching compact leaves back to the faster standard leaf representation. The expansion process undoes the steps taken during the shrinking state. Expansion occurs in two ways, as follows.

One part of the expansion process occurs naturally as a consequence of removes, which are frequent operations once the dataset size starts decreasing. As explained above, a remove that underflows a compact leaf replaces it with a smaller compact leaf, and eventually with a standard leaf.

The second aspect of the expansion process is aimed at cases in which a compact leaf becomes popular but does not experience removals. Such a leaf could potentially offer queries lower performance indefinitely. To avoid this problem, the elasticity algorithm modifies the behavior of searches as follows. In the expansion state, a search that ends at a compact leaf with a capacity of $2k$ keys may randomly decide to split the leaf into leaves of capacity $k$. If $k$ is equal to the $B^+$-tree leaf capacity, we split into standard leaves; otherwise, we split into compact leaves.

## 5 COMPACT NODE REPRESENTATION

Here, we describe SeqTree, our blind trie representation, which serves as the compact representation of our elastic $B^+$-tree nodes. In § 5.4 we describe an optimization to reduce the space occupied by the tuple identifiers stored in the node.

Recall that we target a *database indexing* setting. In this setting, the index is indexing rows of a DBMS table, so the "values" stored in the index are tuple identifiers (pointers to rows of the table). In particular, the key can be extracted from the row it indexes (e.g., the key is the value of a specific column). This property is exploited by indexes such as HOT [3] to avoid storing explicit copies of the keys. Instead, the search path in these indexes is determined by the searched key's bits and they load a stored key from the database table at the last step of the search, to check whether it matches the searched key.

### 5.1 Background: Blind tries

A *trie* is a search tree in which the position of a node in the tree defines the key associated with it: the root is associated with the empty string, and all descendants of the node associated with key $k$ share the common prefix $k$. A *compressed trie* is a path-compressed representation of a trie in which the edges in every non-branching path are merged into a single edge. Figure 2b depicts a compressed trie indexing the keys in Figure 2a. Because the edges on a search path contain all key bits, a compressed trie stores keys directly, just without duplicating common prefixes.

A *blind trie* (also known as Patricia trie [21]) is a compressed trie that does not label the edges but instead records in each node the position of the bit in which the node's children differ. The blind trie does not explicitly store all key bits, and so is more compact than a compressed trie. Figure 2c shows a blind trie for the same set of keys. A blind trie search proceeds by examining the *discriminating* bit indicated in the current node and continuing to the appropriate child. However, to complete a search, the algorithm must look up the indexed tuple (pointed to by the leaf) to confirm that the indexed key matches the searched key.

A blind trie requires only one node to be inserted for every unique key stored. Its memory consumption is thus $O(np + (n-1)\log k)$ bits, where $n$ is the number of keys, $k$ is the size of a key, and $p$ is the tuple identifier size. The constant factor behind the $O(\cdot)$ notation is determined by the blind trie's representation. For example, we would like to avoid storing pointers to children in each node.

**Blind trie representations of B-tree nodes** Work on external-memory B-trees has explored replacing the B-tree's standard node structure of a sorted key sequence with some blind trie representation. Let $n$ be the capacity of the B-tree node. The blind trie representations leverage the fact that $n$ is small (say, $n \leq 256$).

The String B-Tree [13] uses a straightforward representation, in which each trie node contains two pointers to its children. Suppose that the index of a discriminating bit can be represented with 1 byte (i.e., the key size $k \leq 32$ bytes) and that a pointer to a trie node requires 1 byte (recall $n \leq 256$). Then this representation requires 3 B (bytes) per key (ignoring tuple identifiers). Bumbulis and Bowman [4] proposed a more compact representation that we call *SubTrie*. The SubTrie stores the nodes in an array sorted in pre-order (depth-first) traversal order of the trie. In this order, a node's left child (if it exists) is simply the adjacent node in the array. To facilitate finding right children and determining if a left child exists, the SubTrie also maintains an array in which the $i$-th entry stores the size of the $i$-th node's left sub-tree (inclusive of the node). The SubTrie thus requires 2 B per key, under the same assumptions on key size and number of keys. The densest representation, which we call *SeqTrie*, is due to Ferguson [12]. It requires only 1 B per key, but its search complexity is linear in the number of indexed keys (see § 5.2 for details).

### 5.2 The SeqTree representation

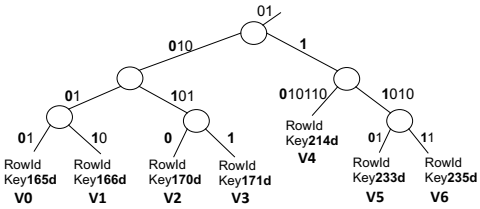
We present SeqTree, a blind trie representation whose space efficiency is comparable to the SeqTrie but with improved search time. The SeqTrie obtains optimal space efficiency by storing only an array of size $n-1$ whose entries are the discriminating bits—without any auxiliary information. However, a SeqTrie search must sequentially scan this entire array. Our observation in SeqTree is that this search can be sped up considerably by maintaining a *small* auxiliary tree that helps a search quickly identify a small range in the array in which its result is found, and thereby reduces the length of the sequential scan it must perform. The storage cost of the auxiliary tree is negligible, because the blind trie is only meant to replace a $B^+$-tree leaf, and so it stores a small number of keys. As a result, the SeqTree's space efficiency in practice is comparable to the SeqTrie's but its search performance is comparable to the SubTrie.

Next, we explain the SeqTrie representation and how it is searched. We then explain how the SeqTree algorithm is obtained by adding a small auxiliary tree to the SeqTrie, which is used to locate the range within a target key must reside without performing a full sequential scan of the array.

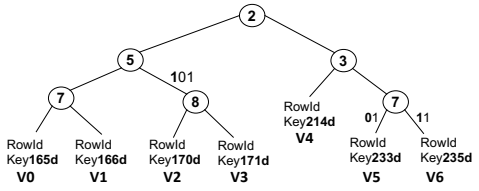
**SeqTrie** The SeqTrie maintains an array in which entry $i$ contains the first bit discriminating between the $i$-th and $(i+1)$-th keys, where the keys are sorted in lexicographic order and bits are numbered from zero, starting from the most significant bit (see the rectangles in Figure 2a). The SeqTrie representation of our example key set is shown in Figure 3a as the array named *BlindiBits*. We use this name as the array contains the discriminating bits of the blind index (Blindi). The SeqTrie also stores an array with the values associated with the stored keys, denoted $V0, \ldots, V6$ (as in Figure 2c).

| | msb | | | | | | | Lsb |
|---|---|---|---|---|---|---|---|---|
| #dec | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 165d | 0 | 1 | 0 | 1 | 0 | 0 | 1 | **0** | 1 |
| 166d | 0 | 1 | 0 | 1 | 0 | **0** | 1 | **1** | 0 |
| 170d | 0 | 1 | 0 | 1 | 0 | **1** | 0 | 1 | **0** |
| 171d | 0 | 1 | **0** | 1 | 0 | 1 | 0 | 1 | **1** |
| 214d | 0 | 1 | **1** | **0** | 1 | 0 | 1 | 1 | 0 |
| 233d | 0 | 1 | 1 | **1** | 0 | 1 | 0 | **0** | 1 |
| 235d | 0 | 1 | 1 | 1 | 0 | 1 | 0 | **1** | 1 |

(a) Indexed keys and their binary representation.

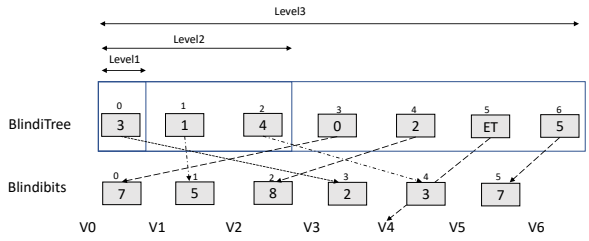



(b) Compact trie.

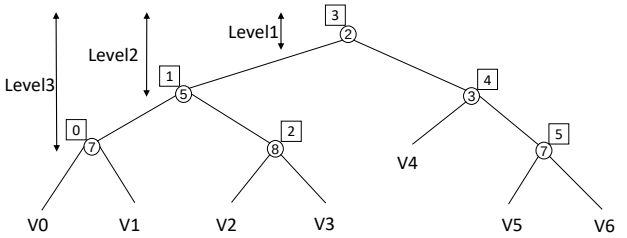



(c) Blind trie.

**Figure 2: Different tries indexing a 7-row table whose rows are indexes by keys shown in (a). Rectangles mark the first bit in which two consecutive keys differ. The notation $Vx$ refers to the value associated with the $x$-th key, e.g., a tuple identifer.**

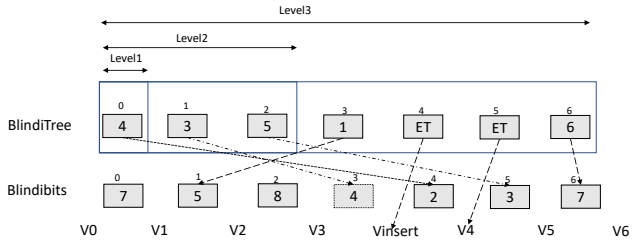



(a) SeqTree arrays. *BlindiTree* is the physical layout of the tree shown in (b). *BlindiBits* is the trie's SeqTrie representation. Numbers above array entries indicate their index.

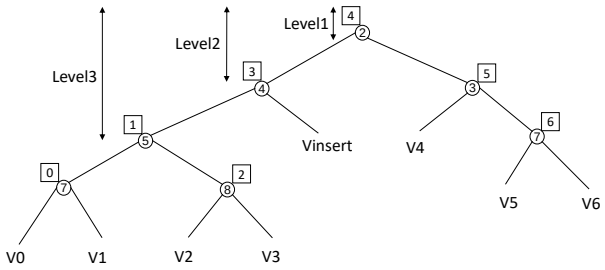



(b) SeqTree logical representation. The boxed number next to a node shows its index in the *BlindiBits* array.

**Figure 3: SeqTree representation for the key set of Figure 2a with tree of height 3.**




(a) SeqTree arrays




(b) Insert logic representation

**Figure 4: SeqTree representation for the key set of Figure 2a, after insertion of the mapping $186 \mapsto V_{insert}$.**

**SeqTrie search** The search has predecessor semantics, i.e., it returns the position of the searched key or, if the searched key is not indexed, the position of the largest key smaller than it. (This enables insertions and range scans to use the search procedure to locate the position in which the key should be inserted or the scan should start, respectively.) Let $SKey$ denote the searched key. The search proceeds under the assumption that $SKey$ is one of the indexed keys. It scans the array sequentially, attempting to find $SKey$'s position. Initially, we assume $SKey$ is the first key (indexed 0). Suppose we assume $SKey$ is the $j$-th key and we scan entry $i$. Let $b_i = BlindiBits[i]$. Then bit $b_i$ is 0 in the $i$-th key but 1 in the $(i+1)$-th key. Thus, if $SKey[b_i] = 1$, it cannot be that $SKey$ is the $i$-th key (or any smaller key). We call this case a *hit*, and we now assume tht $SKey$'s position is $i+1$. If $SKey[b_i] = 0$, then $SKey$ cannot be the $(i+1)$-th key, but our assumption that $SKey$ is the $j$-th key may still be true. We call this case a *miss*. Following a miss, we need not check any discriminating bit $b > b_i$, because for any such entry, the relevant key has bit $b_i$ set and therefore cannot be $SKey$. We thus keep scanning the array but ignore any such bit.

At the end of the sequential scan, the "assumed" position $j$ will be $SKey$'s position if $SKey$ is one of the indexed keys. We verify this by loading the $j$-th key from the database and comparing it to $SKey$. If they do not match, then we now know the bit $b_d$ discriminating $SKey$ and the $j$-th key, as well as whether $SKey$ is greater or smaller than the $j$-th key. Suppose $SKey$ is greater than the $j$-th key. To find $SKey$'s successor, we scan the array to the right of the $j$-th entry, looking for an index $i$ such that $b_i = BlindiBits[i] < b_d$. This means that the $(i+1)$ key has bit $b_i$ set and is therefore larger than $SKey$, but the $i$-th key does not have it set. Therefore $SKey$'s predecessor is the $i$-th key. The case where $SKey$ is smaller than the $j$-th key is similar, except we scan to the left of the $j$-th entry.

**SeqTree representation** The SeqTree augments the SeqTrie with an explicit tree representation of the top levels of the blind trie. Figure 3b shows the logical representation of a tree of height 3 for our example key set (Figure 2). The tree is physically laid out as an array in a compact representation, where the left and right children of the node indexed $i$ are indexed $2i+1$ and $2i+2$, respectively. Each tree node points to the SeqTrie array entry that it represents. The array *BlindiTree* in Figure 3a shows the tree's physical layout. Given a node $u$ in the tree, we denote by $ind(u)$ the index of $u$'s entry in the SubTrie array. For example,

suppose $u$ is the root of the tree in Figure 3b. Then $u$ corresponds to the discriminating bit 2, whose index in *BlindiBits* is 3. Thus, $ind(u) = 3$, i.e., the value stored in index 0 of *BlindiTree* is 3 (Figure 3a). We use a special ENDTREE marker, denoted $ET$ in the figure, to indicate when a trie node does not have a child.[2] Notice that, for instance, because the node of discriminating bit 3 (whose *BlindiTree* index is 2) does not have a left child, entry 5 in *BlindiTree* contains $ET$.

Crucially, in practice using a small tree size imposes almost no space overhead, because the tree can occupy space that was previously unused due alignment considerations. (For example, a 3-level tree requires only 7 B.)

**SeqTree search** The *BlindiTree* enables the search to begin as a standard trie search, until it reaches the end of the tree and must continue by sequentially scanning the array. Our key insight is that the node in which a search "falls off" the tree defines the range in the SeqTrie array that must contain the searched key, if it exists. To see this, notice that the SeqTrie array can also be viewed as containing the discriminating bits (i.e., trie nodes) sorted in *in-order* traversal order of the trie. (In-order traversal visits a node's left subtree, then the node, and then its right subtree.) This allows us to define the *range* associated with a tree node recursively as follows. The range of the tree's root is the entire SeqTrie array. Assume the range of tree node $u$ is $[i, j]$. Then the range of $u$'s left child is $[i, ind(u) - 1]$ and the range of $u$'s right child is $[ind(u) + 1, j]$.

Searching the SeqTree is thus simply a matter of maintaining the range of the current tree node on the search path, and then performing a sequential search on that range after "falling off" the tree. One subtle point is that if the search reaches tree node index $i$ where $BlindiTree[i] = ET$, then its result is simply the $i$-th key. (For example, as Figure 3b shows, the index of tree node 3 in $BlindiTree$ is 5, which corresponds to the 5th key.)

**Identifying a predecessor** As with the SeqTrie, once the SeqTree search terminates, the found key is loaded from the database and compared to the search key. If they do not match, the search key's predecessor needs to be located. The predecessor search can occur sequentially, in the SeqTrie, or by traversing the tree. Which case occurs depends on $b$, the index of the bit discriminating the search key from the key loaded from the database.

We refer to the tree node in which the search fell off the tree as the *tree position*, or *TreeP*. If $b$ is greater than the discriminating bit that *TreeP* represents, then a search for the predecessor would arrive at *TreeP* and similarly fall off. Therefore, the predecessor search needs to be performed in *TreeP*'s range, as in the SeqTrie algorithm. On the other hand, if $b$ is smaller than the discriminating bit that *TreeP* represents, we need to find the predecessor by ascending up the tree until reaching an ancestor $u$ of *TreeP* that represents a discriminating bit which is smaller than $b$. This is the point where we would have to insert a new trie node, if we were inserting the searched key into the trie. Therefore, the predecessor is either the largest node in $u$'s left subtree or the smallest node in $u$'s right subtree, depending on whether the search for our key traverses right or left at $u$.

**Putting it together** Overall, the flow of the SeqTree search is as follows. The search first traverses the SeqTree until finding the searched key's assumed position, as described above. The indexed key is then looked up and compared to the searched key. If they do not match, a second-stage procedure is used to find the searched key's predecessor, either by traversing the tree or by performing a sequential search as in the SeqTrie.

[2]Technically, ET is the maximum number of keys plus one, i.e., an invalid index.

## 5.3 SeqTree maintenance

Here we describe how a SeqTree is initialized and maintained.

**Compacting a B$^+$-tree node into a SeqTree** To do this, we need to convert the sorted key sequence stored in the B$^+$-tree node into a SeqTree. We first construct the SeqTrie, i.e., the *BlindiBits* array, and initialize all *BlindTree* entries to $ET$. To construct the *BlindiTree*, we start from the root, $BlindTree[0]$. We scan *BlindiBits* to find the minimum entry (discriminator bit), and point the root at this entry. Next, the left (respectively, right) child of the root is set to point to the minimum entry among all BlindiBits entries to the left (respectively, right) of the root. To initialize deeper levels, we look from the minimum entry in the range starting with the lowest common ancestor (LCA) of the node and the node to its left on the same level (or 0 if the node is the leftmost in the level) and ending with the LCA of the node and the node to its right (or the end of the array if the node is the rightmost in the level). We use a lookup table to compute the LCAs. Whenever the sequential range to scan is empty, the child is left marked as $ET$. Initializing each level in the tree thus requires scanning all the *BlindiBits* array.

**Insertions & removals** The search locates the position of the new key in the *BlindiBits* SeqTrie array. To insert the key, the array entries of the successor keys need to be shifted one position to the right, providing room for the new key. Subsequently, the *BlindiTree* SeqTree subtrees whose range the shifted positions fall in are updated to point to the new location. In some cases, the discriminating bit of the new key is such that a new node must be inserted into the SeqTree tree. Suppose the new node $x$ needs to be inserted between node $u$ and its child $v$. We splice $x$ as a child of $u$, making $v$ its child, and the newly inserted key its other child. Figure 4 shows an example. The figure shows the SeqTree obtained after inserting the key 186 (mapped to value $V_{insert}$) into the SeqTree containing the key set of Figure 2a. SeqTree removals proceed analogously to insertions; if a node of the tree gets removed, its subtree is rebuilt.

**Splits & merges** A split takes a SeqTree $B$ and splits it into a left SeqTree $B_L$ and a right SeqTree $B_R$, each with half of the keys. A split eliminates a discriminating bit, the one that discriminates the last key in $B_L$ from the first key in $B_R$. We use the position of this bit to split the SeqTree tree. Nodes to its left are moved to $B_L$ and nodes to its right are moved to $B_R$. Since the new trees are not fully populated, we initialize the empty portions of the trees as explained above for creating a SeqTree from a B$^+$-tree node. A merge is the reverse operation, which combines two SeqTrees into one. A merge thus introduces a new discriminating bit, whose position we use to merge the trees of the merged nodes.

## 5.4 Breathing

In a standard B$^+$-tree node, the space occupied by keys typically dominates the space consumed by tuple identifiers. For example, assuming tuple identifiers are 8 B in size (i.e., pointers), then they consume only 20% of the B$^+$-tree leaf for a key size of 32 B. With SeqTree, however, the situation is reversed. Because keys are stored indirectly and very compactly, space consumption of tuple identifiers becomes a dominating factor. Since the SeqTree

requires ≈ 1 B/key for 32 B keys, tuple identifiers will account for almost 90% of the compact node's space. Even with larger keys, where SeqTree requires ≈ 2 B/key, tuple identifiers will still account for ≈ 80% of the compact node's space.

To address this problem, we introduce a *breathing node* optimization to the SeqTree. The idea is to allocate roughly as much space for tuple identifiers as needed for the items currently *stored* in the node, rather than for the node's full capacity. The motivation for this approach is the observation that in many workloads, the occupancy of leaves is far from their capacity. For example, when keys are distributed uniformly at random, the average occupancy of a B$^+$-tree leaf is 70%. In such a case, breathing can theoretically reduce a SeqTree's memory consumption by 27% (30% of the 90% taken by tuple identifiers, described above).

To provide "breathing room" for the node to grow, we allocate additional slack space of $s$ tuple ids. That is, a SeqTree node with capacity $n$ that holds $k < n$ keys will have allocated space for $k + s$ tuple ids. Once insertions fill up the slack space of the node, we replace to node's tuple id array with an array larger by $s$ slots. When splitting a node with $k$ keys, each new node allocates space for $k/2 + s$ tuple ids. The breathing parameter $s$ controls the trade-off between space efficiency and overhead on insertions. For example, $s = 1$ implies almost no slack, so high space efficiency, but every second insertion to the node will need to reallocate the tuple id array. Importantly, however, breathing does not impose overhead on searches.

## 6 EXPERIMENTAL EVALUATION

Our evaluation consists of the following:

- Evaluating the space vs. efficiency tradeoffs that the elastic B$^+$-tree creates on each type of index operation (§ 6.1).
- Evaluating the elastic B$^+$-tree's tradeoff on the YCSB workloads (§ 6.2).
- Evaluating the elastic B$^+$-tree in a full-blown in-memory data store (§ 6.3).
- Analyzing the factors affecting the SeqTree's performance and memory consumption (§ 6.4).

**Setup** In §§ 6.1 and 6.3, we use a server with 2 Intel Xeon E5-2630 v4 (Broadwell) CPUs, each of which has 20 2.2 GHz cores. The server is equipped with 64 GB DDR4 DRAM. In §§ 6.2 and 6.4, we use a server with 4 Intel Xeon E5-4669 v4 (Broadwell) CPUs, each of which has 22 2.2 GHz cores, and 512 GB DDR4 DRAM. Both systems run Linux and all the code is compiled with GCC. We use `jemalloc` for single-threaded experiments and `tcmalloc` for multi-threaded ones.

### 6.1 Elastic index operation tradeoffs

In this section, we evaluate the space vs. operation efficiency tradeoff provided by the elastic B$^+$-tree, for various types of index operations. We compare the following indexes:

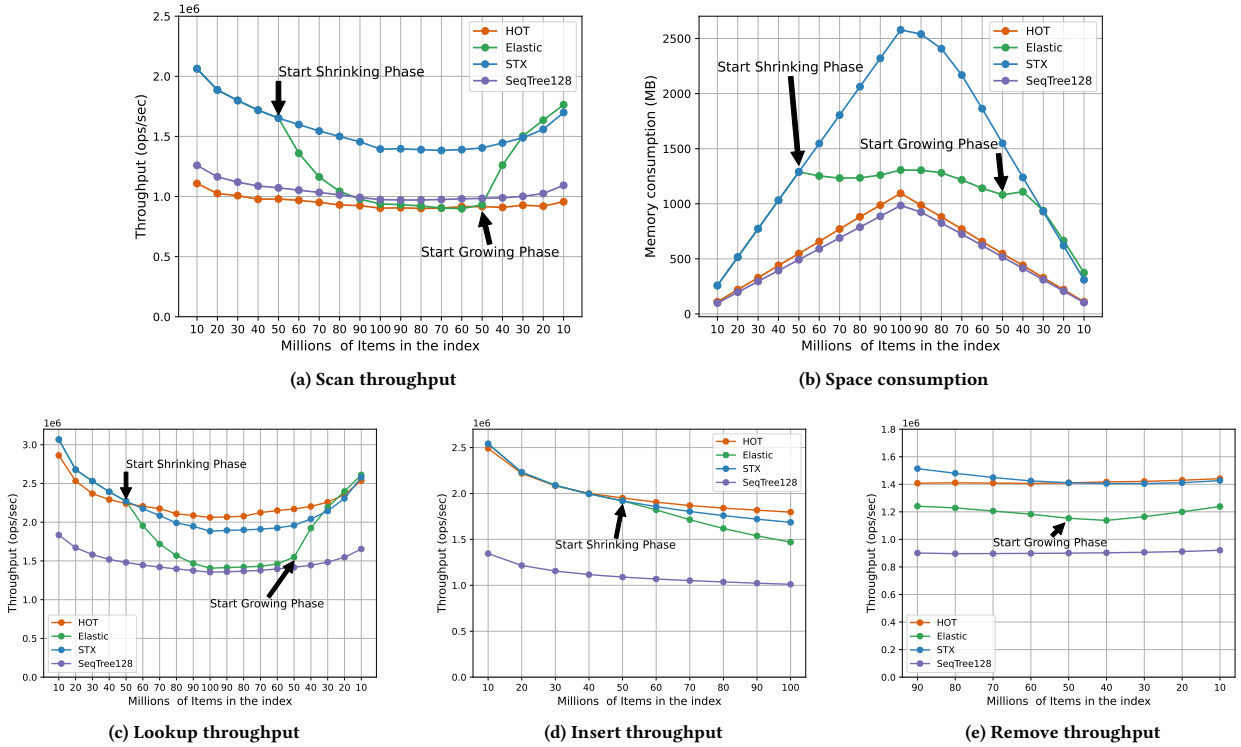
- The elastic B$^+$-tree, based on the STX B$^+$-tree. We use the SeqTree with tree level 2 for the compact leaf representation with at most 128 keys per leaf (the baseline STX B$^+$-tree leaf capacity is 16). We set the breathing parameter to 4. We choose these parameters based on the results of § 6.4.
- STX B$^+$-tree, which serves as the upper bound for space usage of the elastic B$^+$-tree. We use 16-key leaves.
- SeqTree128, which is the STX B$^+$-tree in which all leaves are represented using the SeqTree with tree level 2, the breathing parameter set to 4, and 128 keys per leaf. SeqTree128 thus represents the elastic B$^+$-tree's maximum space savings, but also its maximum overhead on index operations.
- HOT: Height-Optimized Trie [3], which has been shown to be more space efficient and to generally outperform other state-of-the-art indexes, such as Masstree and ART.
- We also compare against Masstree [19], skip list, Bw-tree [31], and ART [16]. We omit the results of these indexes from the plots, because these algorithms are always dominated by some other algorithm: (1) Masstree and skip lists consume more memory than STX; (2) Bw-tree's space consumption is only slightly smaller than that of STX, but it performs worse; and (3) ART is outperformed by HOT, which is also more space efficient.

We evaluate the efficiency of index insertions, lookups (point queries), and scans (range queries) in a workload that exercises the elastic B$^+$-tree's growth and shrink capabilities. Each experiment consists of a single thread inserting 100 M items into the index and subsequently deleting them. Each item is a key/value mapping that indexes a unique row in a 100 M-row table. Insertions and deletions are performed in chunks of 10 M keys. After each such chunk, the thread performs 3 M lookups of random keys and 1 M scans (iterating over 15 keys starting from a random key). We measure the throughput of lookups and scans after every chunk, and the insertion and deletion throughput of each insertion or deletion chunk, respectively. We report average throughputs over 10 runs of the test.

When testing the elastic B$^+$-tree, we configure it to start shrinking when the index sizes grows beyond 50 M items (1289 MB), and to start expanding when the index size decreases beyond 1081 MB (see § 4). Because the latter threshold applies to the elastic B$^+$-tree in its shrinking state, it also corresponds to 50 M items—the index is simply more compact. During the initial insertion phase, the index initially uses B$^+$-tree leaf nodes holding up to 16 keys, and then (on demand and gradually) replaces some of them with SeqTree nodes holding up to 32, 64, and 128 keys, at which point it stops increasing the capacity of a node. During the deletion phase, the elastic B$^+$-tree gradually undoes these leaf compactions. The query phase after each chunk can thus execute on a partially compacted B$^+$-tree, allowing us to study the elastic B$^+$-tree's query overhead.

We run the benchmark with key sizes of 64 bits, 128 bits, and 30 bytes. In a nutshell, larger keys favor the elastic index. Compared to with 64-bit keys, it achieves a better compression rate and its performance degradation is smaller. Due to space constraints, we detail only 64-bit the key results (Figure 5).

**Elasticity facilitates efficient scans** Figure 5a shows that designs such as HOT, which obtain memory efficiency using unconditional indirect key storage, impose a heavy cost on scan queries. With HOT's indirect key storage, each scanned key has to be loaded from the table, whereas the B$^+$-tree holds the keys in its leaves, and so can avoid these extra memory references. HOT thus *always* pays the cost of indirect keys storage, making the throughput of its scan queries 1.5–2× worse than of STX. In contrast, the elastic B$^+$-tree achieves B$^+$-tree performance under typical memory demands and only starts paying under memory pressure, at which point its scan query throughput gracefully degrades—depending on the severity of the memory pressure—until it converges with HOT's scan performance. Once the index sizes decreases, the elastic B$^+$-tree's scan throughput gracefully improves until converging with the B$^+$-tree's scan




(a) Scan throughput

(b) Space consumption

(c) Lookup throughput

(d) Insert throughput

(e) Remove throughput

**Figure 5: Space & throughput comparison of elastic B$^+$-tree vs. other indexes (64-bit keys).**

throughput. Under maximal memory pressure, the elastic B$^+$-tree's scan throughput is a bit lower than that of SeqTree128. This happens because SeqTree128 has *only* compact leaves and a scan there typically traverses fewer leaves than in the elastic B$^+$-tree.

**Memory consumption** Figure 5b shows the memory efficiency side of the tradeoff. While STX maintains the best scan query throughput, its memory consumption grows linearly with the number of inserted keys. The STX index size reaches 1281 MB once 50 M items get inserted, which corresponds to the moment the elastic index starts shrinking. In contrast to STX, the elastic B$^+$-tree's size remains relatively flat from 50 M to 100 M items. This is because every time a leaf node splits, it uses a more compact representation for new nodes. At peak index size, the elastic B$^+$-tree's space consumption is about 25% more than HOT. Both HOT and SeqTree128, which always use indirect key storage, are about 2.5× as memory efficient as STX at peak index size (with SeqTrie being 10% smaller compared to HOT).

**Lookup and insertion throughput (Figures 5c–5d)** The elastic B$^+$-tree exhibits the same trend for both operation types: it performs identically to STX until it starts shrinking, at which point its throughput gracefully degrades. Lookups are also measured when the elastic B$^+$-tree expands, and their throughput eventually converges back to STX's throughput. For lookups, the elastic B$^+$-tree's throughput trends towards SeqTree128, which is 30%-35% lower than that of HOT. The reason for this trend is that HOT's lookups are faster than SeqTree lookups, due to HOT's use of SIMD instructions and other optimizations. HOT is about 10% faster than STX beyond 50 M items, so the elastic B$^+$-tree cannot outperform HOT on lookups. The elastic B$^+$-tree's insertion throughput similarly degrades as it starts shrinking, which involves compacting leaves during insertions that cause splits. At 100 M items, the elastic B$^+$-tree's throughput is 25% lower than STX's.

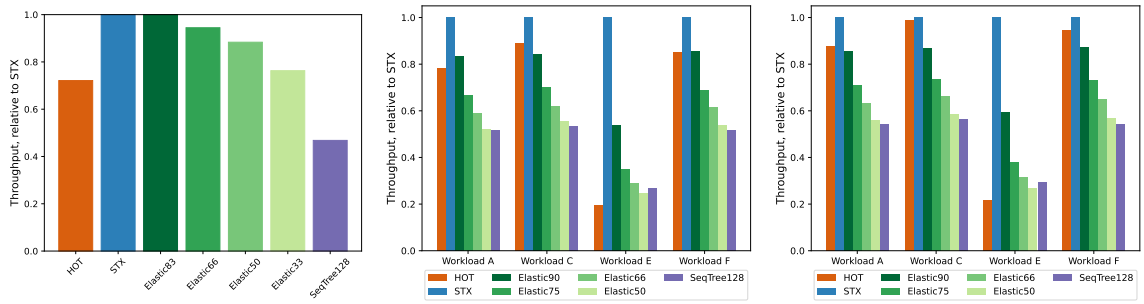
Overall, we observe the same trend as with scans: under typical memory demands, an elastic B$^+$-tree performs identically to a B$^+$-tree. Once it starts shrinking due to memory pressure, its lookups and insertions operations become slower, but in exchange, the index can still fit into the configured memory budget.

**Remove throughput** Figure 5e shows the throughput of remove operations. STX removes are faster than HOT's when the index is at peak size, but STX's remove throughput degrades to HOT's below 60 M items. The elastic B$^+$-tree remove throughput is about 10% lower than STX's, whereas SeqTree128 remove throughput is 40%–45% lower than STX's.

**Operation cost** To break down the cost of the elastic B$^+$-tree operations, we profile the execution of the entire insertion phase of 100 M items (including the last 50 M items, during which the elastic B$^+$-tree enters the shrinking state). We find that 18.3% of the execution time consists of work related to elasticity, broken down as follows: 8.6% is spent on searching compact leaves (not including loading keys from the table); 5% is spent on key comparisons in compact leaves; and 4.7% is spent on converting B$^+$-tree leaves to the compact SeqTrie representation and overflowed compact leaves to higher capacity compact leaves (§ 4).

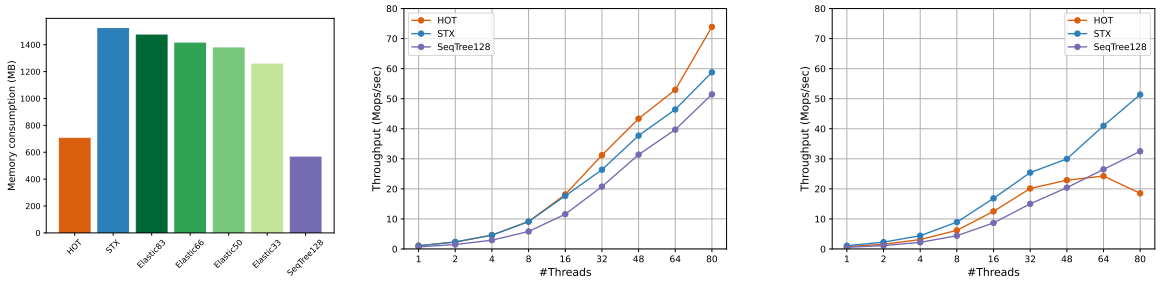
## 6.2 Elasticity in YCSB workloads

We use the framework of Wang et al. [31] to evaluate the indexes on YCSB workloads [7]. We evaluate the core YCSB workloads: A (50% read, 50% update), B (95% read, 5% update), C (read-only), D (latest-read, 95% read, 5% insert), E (95% range scan of size




(a) Load throughput (b) Transaction throughput (uniform) (c) Transaction throughput (zipf)

**Figure 6: Throughput of load and transaction phases of single-threaded YCSB runs.**




(a) Space consumption (b) Workload C throughput (c) Workload C load throughput

**Figure 7: Index space consumption (a) and throughput of multi-threaded YCSB workload C runs (b)–(c).**

1–100 (randomly chosen), 5% insert) and F (50% read, 50% read-modify-write). Each workload is separated into two phases: a load phase inserting 50 million uniformly distributed 64-bit keys, and a transaction phase performing 100 million operations specific to the workload (for workload E we reduced the number of transactions to 25 million) and with zipfian distribution of keys to manipulate. Workloads B, C and D yield similar results and hence are not shown in the plots.

**Single-threaded experiments** Here, we evaluate our elastic B$^+$-tree with different thresholds for shrinking the index. The Elastic$X$ version starts shrinking after $X$% of the 50 million loaded items are inserted. We compare these variants to HOT, STX, and STX-SeqTree with leaf node capacity of 128.

Figure 7a shows memory consumption. HOT consumes 50% less memory than STX, and STX-SeqTree consumes even 10% less than HOT. Memory consumption of the elastic index is almost proportional to the shrinking threshold: Elastic90, Elastic75 and Elastic66 consume 97%, 93%, 91% of STX's memory measurements, respectively.

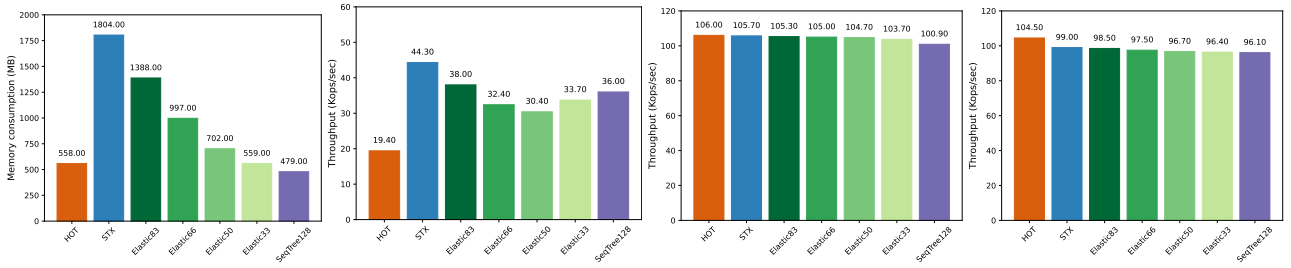
Figure 6a shows the throughput of the load phase. Performance of all elastic indexes is better than HOT's. STX-SeqTree is more than twice as slow as STX.

Figures 6b and 6c show throughput of the transaction phase for each workload, with uniform and zipfian distribution of keys for transaction operations, respectively. Except for the scan workload (E), STX has slightly better performance than HOT, with the gap being narrower with zipfian keys, while with uniform keys HOT is closer to Elastic90. The elastic indexes cause a drop in performance the lower the shrinking threshold is: starting the shrinking phase earlier in the run decreases the throughput.

**Scan performance** The E workload, which consists predominantly of scan operation, is where STX significantly outperforms all the blind trie implementations. The STX B$^+$-tree stores full keys in its leaf nodes, whereas blind tries require bringing the keys from the database table. As a result, in Figure 6b, HOT's throughput is more than 3× lower than STX's. The elastic B$^+$-trees, however, perform better than HOT, because they only compact some of the leaf nodes into a blind trie representation. The flip side of the elastic approach is that the elastic B$^+$-tree is less compact than SeqTrie128. For example, Elastic66 and Elastic50 use 30% more space than SeqTrie128. Their scan throughput is comparable, however, because the fraction of compact leaves is much higher than 70% and compact leaves are larger (up to 128 keys). The result is that scans have a high probability of completing from a single compact leaf and therefore behaving as a SeqTree128 scan.

**Multi-threaded experiments** Here, we compare three indexes: (1) BTreeOLC, which is a B$^+$-tree with multi-threading support via Optimistic Lock Coupling [17, 31], (2) BTreeOLC-SeqTree, which integrates the SeqTree leaves with capacity of 128 items into BTreeOLC, and (3) HOT. Although we have not implemented an elastic BTreeOLC, we use BTreeOLC-SeqTree and BTreeOLC as the lower and the upper bounds for memory consumption of the elastic BTreeOLC, and, similarly, the upper and the lower bounds for performance. In other words, we evaluate the expected performance and memory consumption under normal conditions (BTreeOLC) as well as in the worst-case performance-wise, obtained under maximal leaf compaction (BTreeOLC-SeqTree).

In the transaction phase of all the workloads (with uniform and zipfian distribution of keys), the indexes exhibit similar behaviors with scaling close to linear. HOT scales the best, followed by BTreeOLC and BTreeOLC-SeqTree. Figure 7b shows representative results from workload C. (The results of other workloads are similar, and omitted due to space constraints.) For insertions (Figure 7c), BTreeOLC-SeqTree scales up to 80 threads, but not




(a) Memory consumption (b) Scan throughput (c) Insert throughput (d) Lookup throughput

**Figure 8: Impact of using the elastic B$^+$-tree in the MCAS key-value store: index space consumption (a) and single-threaded throughput (b)–(c) of HOT and the STX B$^+$-tree vs. the elastic B$^+$-tree with different size bounds (*ElasticXX*). Size bounds are specified as percentages of the dataset size. (Absolute values are shown above the bars (MBs (a) and Kops/sec (b)–(c).)**

linearly. HOT scalability is linear up to 16 threads, then slows down. BTreeOLC scales the best, resulting in 2.5× the throughput of HOT and 1.66× the performance of BTreeOLC-SeqTree with 80 threads.

## 6.3 In-memory data store

Here, we show that in the context of a full system, the elastic B$^+$-tree can provide significant space savings with comparable (if not equal) transactional throughput to other indexes. Our study vehicle is the Memory Centric Active Storage (MCAS) system [29]. MCAS is a network-attached, in-memory storage system, built from the ground up to support advanced storage technologies, such as persistent memory. MCAS uses a partitioned architecture, in which operations in each partition are handled by a single-threaded execution engine.

Our experiments use MCAS to model a monitoring/analytics component in a commercial cloud provider's object storage system. This component ingests data about object accesses extracted from the logs of the object store system (as in Figure 1) and answers queries for monitoring, anomaly detection, and so on.

We implement support for in-memory indexed data tables in MCAS using its active data object (ADO) [30] functionality. (An ADO plugin provides custom functionality to the MCAS store; in our case, this is the implementation of an indexed multi-column table and a domain-specific API for loading and querying its data.)

**Dataset & methodology** We load MCAS with data from the Storage Networking Industry Association's (SNIA) I/O Traces, Tools and Analysis (IOTTA) repository [1]. This repository contains a set of anonymized logs of REST operations issued on a single bucket in IBM's object storage system [10]. For our evaluation, we use a 12-hour log trace containing 48 M rows. Each row has 4 8-byte columns: the request's timestamp, type, target object ID, and size. (The original logs contain many additional columns that were removed from the public data in the repository.) The table in MCAS is indexed by 16-byte tuples of timestamp and object ID.

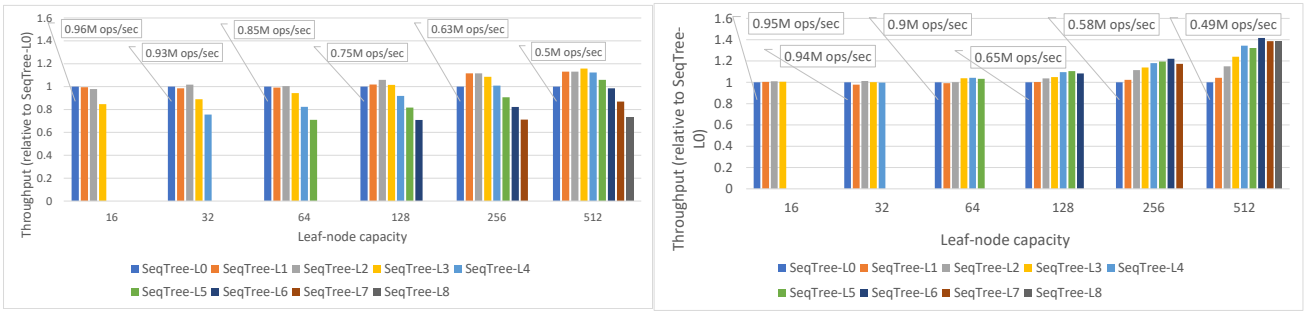
We evaluate MCAS with different indexes: (1) HOT; (2) the STX B$^+$-tree; (3) elastic B$^+$-trees (based on STX) with varying shrinking size bounds, where we denote ElasticXX for an elastic B$^+$-tree that starts shrinking when the index size reaches $XX$% of the dataset size (i.e., of the space consumed by 48 M fixed-size rows in our table); and (4) STX-SeqTree128, which is an STX B$^+$-tree whose leaves are compacted using SeqTree with leaf node capacity of 128, which represents the maximum space savings and overhead of our approach. Apart from the differing size bounds, the elastic B$^+$-trees use the parameters detailed in § 6.1.

**Benchmark description** In our experiments, we configure our MCAS ADO to use the evaluated index type. We use the same machine for the client (performing the operations) and server. We first load MCAS with the trace's log data by performing insert operations (one for each row in the log). Thus, the dataset size is every experiment is identical and only the index size varies. After data ingestion, we measure two types of workloads: lookups of the indexed keys and scans of 1000 keys starting from a random key. We report the index memory consumption and the throughput obtained during the load phase (insertions) and the query phase (lookups and scans). We report single-threaded results averaged over four benchmark runs, each starting with an empty database.

**Memory efficiency** Figure 8a shows the effectiveness of index memory elasticity given a fixed memory budget. With STX, the index size is 1.2× the dataset's size (so overall, the index consumes 54% of the MCAS memory). With the elastic B$^+$-tree, as we tighten the elastic B$^+$-tree's size bound and cause shrinking to start once the index reaches 83%, 66%, 50%, or 33% of the dataset's size, index memory consumption decreases accordingly to 76%, 55%, 39%, and 30% of the B$^+$-tree's index size. For comparison, SeqTrie128—in which all B$^+$-tree leaves are compacted using SeqTree and thus represents the maximum space savings and overhead of our approach—reduces index memory consumption to 26% of the B$^+$-tree's index size. Because this workload has 16-byte keys, which are larger than the key size in § 6.1, both the elastic B$^+$-tree and SeqTrie128 achieve better compression rates than in those experiments. HOT is significantly more memory efficient than STX. It uses 30% of the space of STX, which is comparable to Elastic33.

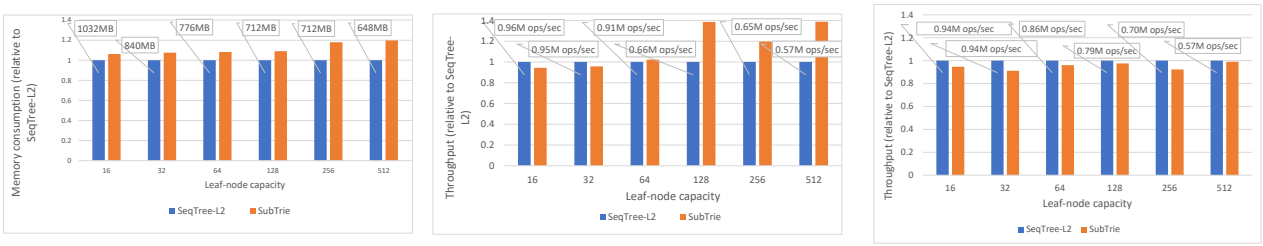
**Throughput** Figures 8b–8d show the throughput trade-offs caused by the more compact representations. For scans, STX has the best throughput—2.3× that of HOT. But a major result is that *Elastic33's scan throughput is* 1.73× *that of HOT, despite both having essentially identical space use.* In fact, Elastic33's scan throughput is higher than that of Elastic50 and Elastic66. The reason is that scans in Elastic33 traverse fewer leaves, as it has fewer B$^+$-tree leaves, so typical leaves are compact and hold many keys.

For insertions and lookups, due to the fact that index operations are only part of end-to-end performance, the elastic B$^+$-tree's significant space savings do not come at a matching performance loss. Compared to STX, insert throughput degrades by 0.37%–1.8% and lookup throughput degrades by 0.5%–2.6%. HOT is slightly faster than the B$^+$-tree variants: its insertion and lookup throughput is 0.28% and 5.5% higher, respectively, than



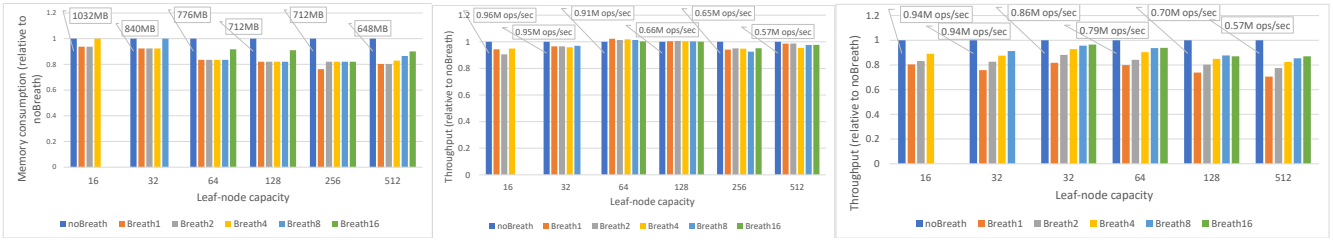

**(a) Insertion performance** **(b) Search performance**

**Figure 9: Throughput of STX B$^+$-tree variants with SeqTree leaf representation (64-bit keys).**




**(a) Space consumption** **(b) Search throughput** **(c) Insert throughput**

**Figure 10: Space & throughput comparison of STX-SeqTree vs. STX-SubTrie, normalized to STX-SeqTree (64-bit keys).**




**(a) Space consumption** **(b) Search throughput** **(c) Insert throughput**

**Figure 11: Space & throughput comparison of STX-SeqTree for different breathing parameters, normalized to without breathing (64-bit keys).**

that of STX. Compared to Elastic33, which uses the same space as HOT, HOT's insertion and lookup throughput is 2.2% and 8.4% higher, respectively.

**Takeaway** Our results illustrate the type of workload that the elastic B$^+$-tree targets. This is a workload where on one hand, scan performance is important, making a B$^+$-tree the index of choice and an index such as HOT a suboptimal choice. But on the other hand, a non-elastic B$^+$-tree would not be able to absorb dataset size spikes (as in Figure 1) due to its high memory usage, requiring either dropping the index, paging data to secondary storage, or failing to ingest new data. In such a setting, an elastic B$^+$-tree offers B$^+$-tree-like scans under typical conditions with *graceful* degradation in throughput when the dataset size spikes.

## 6.4 SeqTree analysis

To evaluate various aspects of the SeqTree's performance, we evaluate STX-SeqTree and STX-SubTrie, which are variants of the STX B$^+$-tree whose leaf representation is a SeqTree or SubTrie, respectively.

We evaluate the B$^+$-tree variants using the index microbenchmarks framework of Levandoski et al. [18]. Our experiment consists of inserting 50 million of uniformly distributed 64-bit and 128-bit keys, and afterwards performing 50 million uniformly distributed searches. Results are averages of 5 runs on an otherwise idle machine; variance is not shown, as it is negligible.

**Tree Levels** Figure 9 shows how the tree level parameter affects the throughput of the STX-SeqTree for various leaf capacities, with breathing disabled. We show results for 64-bit keys; 128-bit key results are similar. For a given leaf-node capacity leafSlots, up to ($\log_2$ leafSlots − 1) tree levels are available in the SeqTree.

For insertions, a small tree level enables better throughput over the baseline of the level 0 as leafSlots grows. The performance boost peaks with medium values of tree levels: although adding more of them enables locating the place for the insertion faster, the insert needs to update the growing number of levels of the tree in SeqTree. Performance peaks at level 3 with leafSlots set to 512, and at level 2 otherwise.

For searches, as leafSlots increases, larger values of tree levels result in higher throughput. Peak performance is not at the maximum tree level: it is at level 5 for leafSlots set to 128, and at level 6 with leafSlots set to 256 or 512.

Theoretical memory consumption of a SeqTree node grows with levels, as the amount of tree meta-data increases. However, in practice, levels 1–3 do not introduce any overhead, as the SeqTree implementation leverages alignment of leaf nodes in

memory. Levels 4 and 5 introduce minor overheads, and levels 6 and 7 use around 10% of extra space.

**Comparison with SubTrie** In Figures 10a, 10b and 10c, we compare the performance of STX-SubTrie and STX-SeqTree the tree levels parameter set to 2 and with breathing disabled, and normalize the results w.r.t. the latter. We omit graphs for insertions, as their performance trend are analogous to the searches; the graph of space efficiency for 128-bit keys is analogous to 64-bit too.

With the increase of the number leafSlots of leaf node entries, the SubTrie consumes more space peaking at 20% of space overhead for 512 leaf slots. The overhead comes from the SubTrie maintaining an additional array of the size (leafSlots − 1), with each cell being an integer holding values up to leafSlots and thus requiring 1 byte when leafSlots is under 256 and 2 bytes afterwards.

STX-SeqTree is almost always slightly faster than SubTrie, but it has a drop in performance with 64-bit keys and over 128 leaf slots: in that case the throughput of insertions and searches in the SubTrie is 20% and 40% higher, respectively.

Nevertheless, we use the SeqTree as the blind trie for the elastic $B^+$-tree, because of SeqTree's space efficiency and performance with leafSlots $\leq$ 128. We empirically find that leaf nodes of capacity as high as 128 become frequent in the elastic $B^+$-tree only once it holds three times as many items as a $B^+$-tree would for a fixed size bound. Specifically, if X is the amount of items a $B^+$-tree can hold without overflowing the size bound, then at 4X items 10% of the leaves in the elastic index are SeqTree nodes with capacity of 128, and that number reaches 37% at 5X items.

**Breathing** Figure 11 shows the space usage and throughput of STX-SeqTree with different values of the breathing parameter and leaf capacity, using 64-bit keys. Figure 11a shows the overall space consumed by all leaf nodes normalized to STX-SeqTree with breathing disabled. Empirically (though not shown), leaf nodes are filled on average at 70% of their capacity, so ideally, the remaining 30% of the space could be saved by breathing. In practice, breathing saves 20% when leaf capacity is $\geq$ 64. This is due to our implementation, e.g., we only condense the tuple id array of SeqTree, leaving its SeqTrie array intact.

Measurements for breathing parameters 1, 2 and 4 often coincide. This happens because the associated space overheads are small, and overheads need to be at least 64 MB to be visible, as `jemalloc` allocates memory from the OS with 64 MB granularity.

Breathing causes a small degradation in search performance (Figure 11b), since there is one more pointer to dereference before reaching the data pointer. The search performance degradation is in small leafSlots and with leafSlots $\geq$ 64 there is almost no performance degradation. Inserts (Figure 11c) suffer a notable performance degradation with breathing due to the extra work of allocating a new node and copying the old content into it.

Since setting the breathing parameter to 4 enables almost all the compression possible at 20% without almost no performance penalty in search and with 10% performance degradation in insert, we choose it for the elastic $B^+$-tree. Our experiments with 128-bit keys showed similar results.

## 7 RELATED WORK

There is a vast literature on DBMS indexes. In § 2, we discuss the most relevant efforts, and why they are not a good fit for our context. Here, we discuss additional related work.

Several works explore representing a $B^+$-tree node with different blind trie representation instead of using the standard sorted key sequence representation [4, 12, 13]. (We discuss these representations in detail in § 5.1.) These works do not target in-memory indexes. Further, they do not selectively and dynamically switch between different node representations.

HOT [3] is a generalized Patricia (blind) trie that adapts the number of bits considered in each node (the span of the node) to the key distribution, so that every node has a high fan-out independent of the data set. SeqTree—our blind trie representation—is more space efficient (§ 6), but its queries are slower. In principle, HOT can be used as a compact node representation within the elastic index framework. However, the HOT implementation is limited to holding at most 32 keys/node, whereas the SeqTree has no such limitation.

## 8 CONCLUSION

This paper proposes an elastic index framework, a novel approach to address the index memory overhead problem in in-memory DBMS data pipeline elements. The framework transforms a space-inefficient index that directly stores keys into an elastic version that can temporarily shrink itself in response to memory pressure.

To demonstrate our framework, we design an elastic $B^+$-tree, whose compact nodes use a novel blind trie representation. We evaluate the elastic $B^+$-tree using uniform and YCSB workloads, as well as by integrating it into the MCAS in-memory data store. We find that the elastic $B^+$-tree can be 2×–5× more space efficient than a $B^+$-tree with only moderate impact on query throughput.

## ACKNOWLEDGMENTS

We thank Oz Anani and Gal Lushi for their work on an earlier stage of this project. We thank the anonymous reviewers for their feedback, which helped improve the paper.

This research was funded in part by the Israel Science Foundation (grant 2005/17) and the Blavatnik Family Foundation.

## REFERENCES

[1] 2021. SNIA: IOTTA repository home. http://iotta.snia.org/. (2021).
[2] Timo Bingmann. 2013. STX B+ Tree C++ Template Classes v0.9. https://github.com/bingmann/stx-btree. (2013).
[3] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *SIGMOD 2018*.
[4] Peter Bumbulis and Ivan T. Bowman. 2002. A Compact B-tree. In *SIGMOD 2002*.
[5] Steven Camina, Kevin White, Conor Doherty, and Gary Orenstein. 2015. *Building Real-Time Data Pipelines*. O'Reilly Media, Inc.
[6] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM CSUR* 11, 2 (June 1979).
[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC 2010*.
[8] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *SIGMOD 1984*.
[9] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD 2013*.
[10] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen I. Kat. 2020. It's Time to Revisit LRU vs. FIFO. In *HotStorage 2020*.
[11] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA Database: Data Management for Modern Business Applications. *ACM SIGMOD Record* 40, 4 (Jan. 2012).
[12] David E. Ferguson. 1992. Bit-Tree: A Data Structure for Fast File Processing. *CACM* 35, 6 (June 1992).
[13] Paolo Ferragina and Roberto Grossi. 1999. The String B-tree: A New Data Structure for String Search in External Memory and Its Applications. *JACM* 46, 2 (March 1999).
[14] Goetz Graefe. 2011. Modern B-Tree Techniques. *Foundations and Trends in Databases* 3, 4 (April 2011).

[15] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time Analytical Processing with SQL Server. *VLDB* 8, 12 (Aug. 2015).
[16] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In *ICDE 2013*.
[17] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *DaMoN 2016*.
[18] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE 2013*.
[19] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *EuroSys 2012*.
[20] Microsoft. 2017. Create Indexes with Included Columns (SQL Server 2017 Documentation). https://docs.microsoft.com/en-us/sql/relational-databases/indexes/create-indexes-with-included-columns?view=sql-server-2017. (2017).
[21] Donald R. Morrison. 1968. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *JACM* 15, 4 (Oct. 1968).
[22] MySQL. [n. d.]. How Compression Works for InnoDB Tables. http://dev.mysql.com/doc/refman/5.5/en/innodb-compression-internals.html. ([n. d.]).
[23] Oracle. [n. d.]. Compressing your Indexes: Index Key Compression. https://blogs.oracle.com/dbstorage/compressing-your-indexes:-index-key-compression-part-1. ([n. d.]).
[24] Pedro Eugenio Rocha Pedreira. 2018. In-Memory Analytic DBMSs: Design and Lessons Learned. In *SBBD 2018*.
[25] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *CACM* 33, 6 (June 1990).
[26] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *VLDB 2007*.
[27] VoltDB. 2018. *Stream Processing*. Technical Report.
[28] VoltDB. 2018. *The 5G Revolution*. Technical Report.
[29] Daniel Waddington, Clem Dickey, Moshik Hershcovitch, and Sangeetha Seshadri. 2021. An Architecture for Memory Centric Active Storage (MCAS). *arXiv e-prints* arXiv:2103.00007 (2021). arXiv:cs.AR/2103.00007
[30] Daniel Waddington, Clem Dickey, Luna Xu, Moshik Hershcovitch, and Sangeetha Seshadri. 2021. A High-Performance Persistent Memory Key-Value Store with Near-Memory Compute. *arXiv e-prints* arXiv:2104.06225 (2021). arXiv:CoRR/2104.06225
[31] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD 2018*.
[32] Z. Xie, Q. Cai, H. V. Jagadish, B. C. Ooi, and W. Wong. 2017. Parallelizing Skip Lists for In-Memory Multi-Core Database Systems. In *ICDE 2017*.
[33] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *SIGMOD 2016*.