B^S-tree: A gapped data-parallel B-tree

Dimitrios Tsitsigkos
Archimedes, Athena RC
Athens, Greece
dtsitsigkos@athenarc.gr

Achilleas Michalopoulos
CSE Dept, U. of Ioannina
Ioannina, Greece
amichalopoulos@cse.uoi.gr

Nikos Mamoulis
U. of Ioannina & Archimedes, Athena RC
Ioannina, Greece
nikos@cs.uoi.gr

Manolis Terrovitis
IMSI, Athena RC
Athens, Greece
mter@athenarc.gr

***Abstract*—We propose B^S-tree, an in-memory implementation of the B^+-tree that adopts the structure of the disk-based index (i.e., a balanced, multiway tree), setting the node size to a memory block that can be processed fast and in parallel using SIMD instructions. A novel feature of the B^S-tree is that it enables gaps (unused positions) within nodes by duplicating key values. This allows (i) branchless SIMD search within each node, and (ii) branchless update operations in nodes without key shifting. We implement a frame of reference (FOR) compression mechanism, which allows nodes to have varying capacities, and can greatly decrease the memory footprint of B^S-tree. We compare our approach to existing main-memory indices and learned indices under different workloads of queries and updates and demonstrate its robustness and superiority compared to previous work in single- and multi-threaded processing.**

***Index Terms*—indexing, main memory, data parallelism**

I. Introduction

The B^+-tree is the dominant indexing method for DBMSs, due to its low and guaranteed cost for (equality and range) query processing and updates. It was originally proposed as a disk-based index, where the objective is to minimize the I/O cost of operations. As memories become larger and cheaper, main-memory and hardware-specific implementations of the B^+-tree [14], [16], [25], [32], [33], [38], [40], [44], [52], [53], [58], [66], as well as alternative access methods for in-memory data [10], [13], [15], [36], [42], [48], [50], [67], [69] have been proposed. The optimization objective in all these methods is minimizing the computational cost and cache misses during search. More recently, learned indices [21], [24], [26], [35], [37], [45], [64], [65], [70], [71], which replace the inner nodes of the B^+-tree by ML models have been suggested as a way for reducing the memory footprint of indexing and accelerating search at the same time.

In this paper, we propose B^S-tree, a B^+-tree for main memory data, which is optimized for modern commodity hardware and data parallelism. B^S-tree adopts the structure of the disk-based B^+-tree (i.e., a balanced, multiway tree), setting the node size to a memory block that can be processed in parallel. At the heart of our proposal lies a data-parallel *successor* operator ($succ$), implemented using SIMD, which is applied at each tree level for branching during search and updates and for locating the search key position at the leaf level. To facilitate fast updates, without affecting SIMD-based search, we propose a novel implementation for gaps (unused positions) by duplicating keys. The main idea is that we write in each unused slot the next used key value in the node or a global MAXKEY value if all subsequent slots are unused. Our B^S-tree construction algorithm initializes sparse leaf nodes with intentional gaps in them, in order to (i) delay possible splits and (ii) reduce data shifting at insertions. Splitting also adds gaps proactively. Finally, we apply a frame-of-reference (FOR) based compression method that allows nodes that use fixed-size memory blocks to have *varying capacities*, which saves space and increases data parallelism.

Novelty and contributions. There already exist several SIMD-based implementations of B-trees and k-ary search [28], [33], [38], [56], [58], [66]. In addition, updatable learned indices [21] use gaps to facilitate fast updates. Finally, key compression in B-trees has also been studied in previous work [14]. To our knowledge, our proposed B^S-tree is the first B^+-tree implementation that gracefully combines all these features, achieving at the same time minimal storage and high throughput. In particular, the use of duplicate key vales in unused gaps/slots allows (i) branchless, data-parallel SIMD search at each node, and (ii) efficient key insertions and deletions with limited shifting of keys within each node. In addition, our compression scheme allows for direct application of data-parallel search on compressed nodes. To our knowledge, using duplicate keys in gaps within each node for efficient data-parallel search and updates at the same time is novel and has not been supported by previous B-tree implementations [28]. We implement a version of optimistic lock coupling in B^S-tree for concurrency control and extensively compare B^S-tree with open-source single- and multi-threaded implementations of state-of-the-art non-learned and learned indices on widely used real datasets, to find that B^S-tree and its compressed version consistently prevail in different workloads of reads and updates, typically achieving 1.5x-2x higher throughput than the best competitor.

Outline Section II presents related work. The B^S-tree is described in Section III and its updates and construction in Section IV. Section V describes B^S-tree compression. Implementation details and concurrency control are discussed in Section VI and VII, respectively. Section VIII includes our experimental evaluation. We conclude in Section IX.

II. Related Work

A. B-tree

The B^+-tree is considered the de-facto access method for relational data, having substantial advantages over hash-based

indexing with respect to construction cost, support of range queries, sorted data access, and concurrency control [23], [27], [28]. As memory sizes grow, the interest has shifted to in-memory access methods [68]. Rao and Ross [52] were the first to consider the impact of cache misses in memory-based data structures; they proposed *Cache-Sensitive Search Trees* (CSS-trees), in which every node has the same size as the cache-line of the machine and does not need to keep pointers for the links between nodes, but offsets that can be calculated by arithmentic operations. Rao and Ross [53] also proposed the Cache Sensitive B^+-tree (CSB+tree), which achieves cache performance close to CSS-Trees, while having the advantages of a B^+-tree. Chen et al. [18], [19] showed how prefetching can significantly improve the performance of index structures by reducing memory access latency. The pkB-tree [14] is an in-memory variant of the B-tree that uses partial-keys (fixed-size parts of keys), which reduce cache misses and improve search performance. Zhou and Ross [73] investigated buffering techniques, based on fixed-size or variable-sized buffers, for memory index structures, aiming to avoid cache thrashing and to improve the performance of bulk lookup in relation to a sequence of single lookups. Graefe and Larson [29] surveyed techniques that improve the perfomance of B^+-tree by exploting CPU caches.

B. (Data) parallelism in B-trees

The advent of SIMD instructions and GPUs opened new perspectives for in-memory index structures. In an early work, Zhou and Ross [72] explored the use of SIMD to parallelize key database operations (such as scans, joins, and filtering), minimizing branch mispredictions. Schlegel et al. [56] present methods for SIMD-based k-ary search (find which out of k partitions contains a search key). FAST [33] optimizes k-way tree search by leveraging architecture-specific features such as cache locality, SIMD parallelism on CPUs, and massive parallelism on GPUs. [25] introduced a "braided" B^+-tree structure optimized for parallel searches on GPUs, enabling lock-free traversal using additional pointers. Kaczmarski [32] proposed a bottom-up B^+-tree construction and maintenance technique using CPU and GPU for bulk-loading and updates. Bw-Tree [44] is a highly scalable and latch-free B^+-tree variant optimized for modern hardware platforms, including multi-core processors and flash storage. Hybrid B^+-tree [58] leverages both CPU and GPU resources to optimize in-memory indexing, by dynamically balancing the workload between the CPU and the GPU. Yan et al. [66] proposed a B^+-tree tailored for GPU and SIMD architectures. This structure decouples the "key region", which contains keys of the B^+-tree with the "child region", which is organized as a prefix-sum array and stores only each node's first child index in the key region. Kwon et al. [38] proposed DB+-tree, a B^+-tree with partial keys, that utilizes SIMD and other sequential instructions for fast branching. PALM [57] is a parallel latch-free variant of the B^+-tree, which is optimized for multi-core processors, enabling concurrent search and update operations. Other works explore the implementation of B-trees on flash memory [8], [11], [30], [31], [46], [51], [60], [63], non-volatile memory [20], [47] and hardware transactional memory [59].

C. Other in-memory access methods

Besides B-trees, other data structures have also been used for in-memory indexing, especially trie-based ones, such as the HAT-trie [9], [10], the generalized prefix tree (trie) [15], KISS-TREE [36], and Masstree [48]. Leis et al. [42] proposed a fast and space-efficient in-memory trie called ART, which dynamically adjusts its node sizes providing a compact and cache-efficient representation. ART uses lazy expansion and path compression to improve space utilization and search performance. Leis et al. proposed two synchronization protocols for ART in [43], which have good scalability despite relying on locks: optimistic lock coupling and the read-optimized write exclusion (ROWEX) protocol. Height Optimized Trie (HOT) [13] is an in-memory trie-based index that reduces tree height through path compression and node merging. SuRF (Succinct Range Filter) [69] leverages succinct tries to provide a space-efficient solution for range query filtering.

D. Learned Indexing

The advent of fast and accurate machine learning techniques inspired the design of a new type of index structure, called *learned index* [37]. The main idea is to learn a cumulative distribution function (CDF) of the keys and define a Recursive Model Index (RMI) that replaces the inner nodes of a B^+-tree by a hierarchy of models that can predict very fast the position of the search key. FITing tree [26] and PGM-index [24] build upon RMI with a focus on improving model performance. with provable worst-case bounds on query time and space usage. The RadixSpline (RS) [35] learned index can be constructed in a single traversal of sorted data. ALEX [21] is an updatable learned index structure, based on RMI. ALEX utilizes a *gapped array* layout that gracefully distributes extra space between elements based on the model's predictions, enabling faster updates and lookups by exponential search. Other updatable learned indices include CARMI [70], NFL [65], LIPP [64], and DILI [45], and Hyper [71]. Refs. [34], [49], [62] provide comprehensive evaluations on updatable learned indices and traditional indices including many important findings, based on tests on several real-world datasets.

III. The B^S-tree

The B^S-tree follows the structure of the B^+-tree. Each internal node of the tree fits up to N references to nodes at the lower level and up to $N-1$ keys. Leaf nodes contain rid-key pairs, where a record-id (rid) is the address (potentially on the disk) of the record that has the corresponding key value. We assume that keys are unique; in case of duplicate keys, one key is kept in the index with a reference to a block with the rid's of all records having that key. The storage of the rid's is decoupled from the storage of the keys, i.e., they are stored in two different (aligned) arrays, such that the rid array is accessed only if necessary (i.e., only if the key is found and we need access to the corresponding record). Similarly,

the storage of keys in an internal node is decoupled from the storage of children addresses, to facilitate fast search, as we explain later. Each leaf node hosts the address of the next leaf in the total key order to facilitate range queries. For the B^S-tree nodes we use a value of N that facilitates fast and parallel search, as we will explain in Sections V and VI. For the efficient handling of updates, we allow gaps (i.e., unused slots) in nodes, similarly to previous work [21], [45], [64], as will be discussed in Section IV.

Figure 1 shows an example of a B^S-tree, where each node holds up to $N-1=4$ keys. Each non-leaf node is shown as an array of N node pointers (bottom) and $N-1$ keys (top) that work as separators. All keys in the subtree pointed by the i-th pointer are strictly smaller than the i-th key and greater than or equal to the $(i-1)$-th key (if $i>0$). Any unused key slots at the end of each node carry a special MAXKEY value (denoted by ∞ in the figure), which is larger than the maximum possible value in the key domain. For example, if keys are unsigned 64-bit integers, MAXKEY $=2^{64}-1$ and key values range in $[0, 2^{64}-2]$.

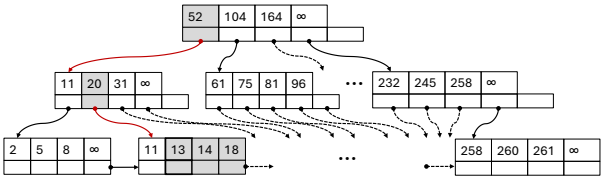
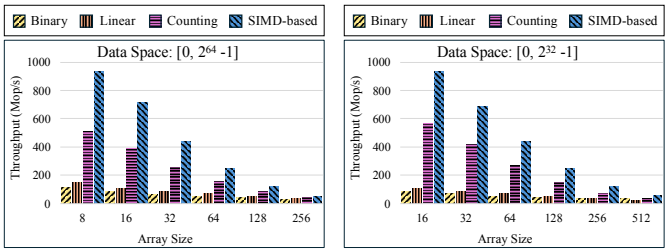


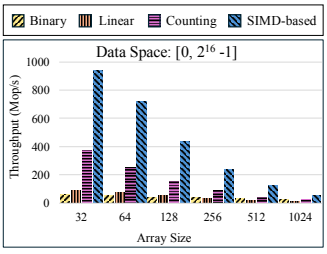
Fig. 1: Example of B^S-tree

A. Search within a B^S-tree node

We now elaborate on the implementation of *branching* at each node of the B^S-tree, i.e., selecting the next node to visit. Traditionally, at each visited node, starting from the root, finding the smallest key which is strictly greater that the query key k is done either by binary search or by linearly scanning the entries until we find the first key greater than k. Both these approaches have high cost due to branch mispredictions.



(a) `uint64` (b) `uint32`



(c) `uint16`

Fig. 2: Successor search techniques in small `uint` arrays

We denote by $succ_>$ the operator that finds the *smallest key position* which is *strictly greater than* the query key k (used in non-leaf nodes) and by $succ_\geq$ the finding of the *smallest key position* which is *greater than or equal to* k (used in leaf nodes). For example, in Figure 1, $succ_{>124}(root)=2$, as the smallest key which is greater than 124 is at position 2. $succ_>$ is applied at each non-leaf node along the path from the root to the leaf that includes the (first) search result.

We use an efficient implementation of $succ_>$, which exploits data parallelism (i.e., SIMD instructions) and does not include if statements or while statements with an uncertain number of loops; hence, it does not involve search decisions. Specifically, let v be a node and k be the search key. Then, $succ_{>k}(v)=|\{x: x\in v.keys \wedge k\geq x\}|$, where $|S|$ denotes cardinality of S. Based on this, $succ_{>k}(v)$, for `uint64` keys can be implemented by code Snippet 1. The corresponding SIMD-fied code (AVX 512) is Snippet 2, where Line 6 loads the node keys vector, Line 7 creates a comparison mask which has 1 at key positions where the search key is greater than the node key, and Line 8 counts the 1's in the mask. As the code snippets do not include if-statements, they do not incur branch mispredictions. CAPACITY is the (fixed) maximum capacity of the node, so the number of iterations of the for-loop is hardwired; all these favoring data parallelism.

Snippet 1: Counting search

```
1  int succG(Node *v, uint64 skey) {
2    int count = 0;
3    for(int i=0; i<CAPACITY; i++)
4      count += (skey >= v->keys[i]);
5    return count;
6  }
```

Snippet 2: SIMD-based counting search (AVX 512)

```
1  int succG_SIMD(Node *v, uint64 skey) {
2    int count = 0;
3    __mmask8 cmp_mask = 0;
4    __512i vec,Vskey = _mm512_set1_epi64(skey);
5    for(int i = 0; i < CAPACITY; i += 8) {
6      vec = _mm512_loadu_epi64((__512i*)(v->keys+i));
7      cmp_mask = _mm512_cmpge_epu64_mask(Vskey,vec);
8      count += _mm_popcnt_u32((uint32_t)cmp_mask);
9    }
10   return count;
11 }
```

Similarly, $succ_{\geq k}(v)=|\{x: x\in v.keys \wedge k > x\}|$ and the same code snippets can be used, by replacing comparison >= by > and `_mm512_cmpge_epu64_mask` by `_mm512_cmpgt_epu64_mask`. This approach (with slightly different implementation) has also been suggested for SIMD-based k-way search in [56], [58], [72].

The experiments of Figure 2 illustrate the efficiency of data-parallel $succ_>$ compared to traditional implementations of branching in multiway trees, on sorted arrays of 64-bit, 32-bit, and 16-bit unsigned integers. The arrays simulate key arrays in a full B^S-tree node, with values drawn randomly

from the corresponding `uint` domain. We performed random successor (i.e., branching) operations to the array and measured the throughput (in millions operations per second) of four implementations of $succ_{>}$:

- **Binary:** use of (non-recursive) binary search
- **Linear:** scan from the beginning until successor is found
- **Counting:** count-based in a for-loop (Snippet 1)
- **SIMD-based:** count-based using SIMD (Snippet 2)

We tested various sizes of the array, modeling different key-array sizes in a B^S-tree node. We used array sizes that are multiples of 8, which allows us to take full advantage of SIMD-parallelism. Counting (Snippet 1) is autovectorized by compilation flags `-O3` and `-march=native`.

Binary search and linear scan perform similarly, with linear search being superior on small arrays and binary search prevailing on larger arrays, as expected. Counting search (Snippet 1) is much faster than binary/linear scan, due to the absence of branch instructions and due to autovectorization optimizations at the assembly level, by the `-O3 -march=native` compilation flag. Observe the excellent performance of SIMD-based search (Snippet 2) for all array and key sizes. Compared to black-box `-O3` compilation, custom vectorization offers significant advantages and achieves the theoretically optimal performance. For example, for 64-bit keys and key-array capacity 16, it achieves 7x performance improvement over binary search, which is even higher than the theoretically expected 4x ($\log_{16} 16$ vs. $\log_2 16$). For `uint16` keys, Snippet 2 is more than 2x faster than autovectorized Snippet 1.

B. B^S-tree search

Algorithms 3 and 4 show how the B^S-tree is searched for (i) equality queries and (ii) range queries. For equality, we traverse the tree by applying a $succ_{>k}(v)$ operation at each non-leaf node v. At the reached leaf v we apply a $succ_{\geq k}(v)$ operation to find the first position r in the leaf having a key greater than or equal to k. If $v.keys[r]$ equals k, then the record at position r is returned; otherwise, k does not exist. Equality search requires one $succ_{>}$ or $succ_{\geq}$ operation per node along the search path. As an example, consider searching for key 13 in the tree of Figure 1. A $succ_{>13}$ operation on the root will give position 0, as there are 0 keys smaller than or equal to 13, so the first pointer of the root will be followed. Then, the $succ_{>13}$ operation on the visited node will return 1, which means that we then visit the 2nd leaf, where $succ_{\geq 13}$ is applied that returns 1, i.e., the position of 13 in the leaf.

For range queries, assume that we are looking for all keys x, such that $k_1 \leq x \leq k_2$. We traverse the tree using $succ_{>k_1}$ operations to find the first leaf that may contain a query result. In that leaf, we apply a $succ_{\geq k_1}$ operation to locate the position r_1 of the first qualifying key. We repeat the same search to find the position r_2 of the first key greater than k_2. The results are the keys in positions from r_1 to r_2 (exclusive).

ALGORITHM 3: Equality Search

Input : search key k, B^S-tree root node v
Output : record-id corresponding to key k

```
while v is non leaf do
    v ← node pointed by entry at position v[succ_{>k}(v)]
r ← succ_{≥k}(v)                    ▷ leaf node;
if v.keys[r] == k then
    return record-id in v at positon r
else ▷ k does not exist
    return NULL
```

ALGORITHM 4: Range Search

Input : search keys k_1, k_2, B^S-tree root node v
Output : record-ids of keys x, where $k_1 \leq x \leq k_2$

```
n1, n2 ← v;
while n1 is non leaf do
    n1 ← node pointed by entry at position
      n1[succ_{>k1}(n1)]
r1 ← succ_{≥k1}(n1);
while n2 is non leaf do
    n2 ← node pointed by entry at position
      n2[succ_{>k2}(n2)]
r2 ← succ_{>k2}(n2);
return record-ids of keys from position r1 to position r2
```

IV. Gaps and Updates

The main novelty of B^S-tree is the implementation of gaps in nodes using *duplicated dummy keys*, which facilitates efficient updates and allows the use of Snippet 2 for search at the same time. Specifically, while SIMD-based k-way search on *packed* arrays has also been suggested before [56], [58], [72], to our knowledge it has never been applied on B$^+$-tree nodes with unused slots. As discussed in Sec. III, unused key slots at the end of each node are filled with MAXKEY values; hence, uniqueness is not a requirement for unused key positions. In addition, B^S-tree does not require the used key slots to be continuous at the beginning of the node. This means that 'gaps' with unused key slots are allowed in a node.

In B^S-tree, we enforce that the key value in a gap (unused slot) is the same as the *first subsequent non-gap key*. By managing auxiliary information at each node, i.e., (i) the *slot use* (number of used slots) and (ii) a *bitmap* indicating used slots, we can efficiently track unused slots; see Figure 3 for an example. In the rest of this section, we will show how deletions and insertions are handled in the B^S-tree. We will explain how our approaches minimize the overhead of node modifications while maintaining high search performance.

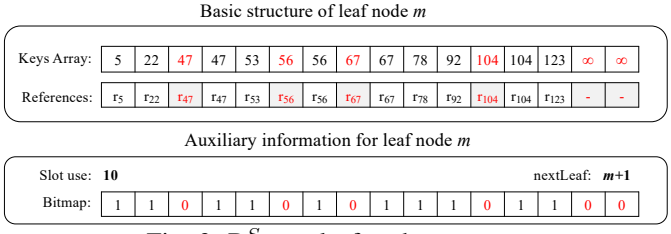


Fig. 3: B^S-tree leaf node structure

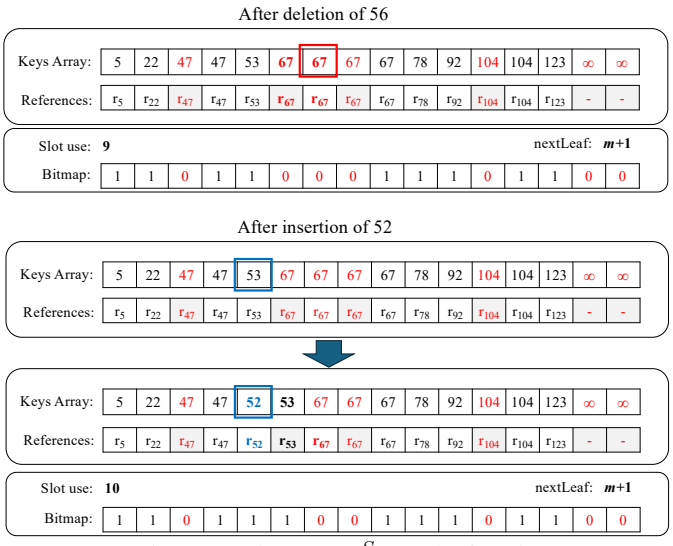


Fig. 4: Updates to B^S-tree leaf node

ALGORITHM 5: Deletion in B^S-tree

Input : key k, B^S-tree root node v

1 find leaf v and position r by running lines 1-3 of Alg 3
2 **if** $v.keys[r] \neq k$ **then**
3 **return FAIL**
4 $bitmap \leftarrow v.bitmap$
5 **if** $r == N-1$ **then** ▷ last key in node
6 $bitmap \leftarrow bitmap \oplus$ 0x0001
7 $replicasOfKey \leftarrow \_tzcnt(bitmap)$
8 **for** $i \leftarrow 0$ ***to*** $replicasOfKey - 1$ **do** ▷ copy backwards
9 $v.keys[r-i] \leftarrow MAXKEY$
10 **else** ▷ r is not the last position in the leaf
11 $bitmap \leftarrow bitmap \oplus (0x8000 \ll r)$
12 $replicasOfKey \leftarrow \_lzcnt(bitmap)$
13 $nextValidKey \leftarrow v.keys[r + replicasOfKey + 1]$
14 **for** $i \leftarrow 0$ ***to*** $replicasOfKey$ **do** ▷ copy backwards
15 $v.keys[r+i] \leftarrow nextValidKey$
16 $bitmap \leftarrow bitmap \oplus (0x8000 \gg (r + replicasOfKey))$
17 $v.slotuse \leftarrow v.slotuse - 1$
18 $v.bitmap \leftarrow bitmap$
19 **return SUCCESS**

A. Deletions

To delete a key, we first locate its position i in a leaf node, using the equality search algorithm (discussed in Section III-B). Then, we copy the key value from position $i+1$ to position i and propagate it backwards to previous gap positions in the node. If i is the last position in the leaf, we set $v.keys[i]$=MAXKEY. One subtle point to note is that $succ_{\geq}$ may not give us the real position of the key k to be deleted but may give us the first of a sequence of gaps that have key value equal to k. For example, for deleting $k = 56$ in Figure 3, we apply $succ_{\geq}(56)$ which gives us position 5. Then, we find the range of all positions having 56 (i.e., [5,6]) and copy into them the next value (i.e., 67). Finding all the positions can be done very fast using bitwise operations. Figure 4 (top) shows the leaf node of Figure 3 after deleting key 56. Algorithm 5 is a pseudocode for deletions to the B^S-tree.

As in previous work [53], [64], we do not take action for nodes with fewer than 50% occupied slots, as we anticipate insertions to be more frequent than deletions, so node merges or key redistributions are not expected to pay-off. If the last entry is deleted from a node, the node is marked as empty and the corresponding separator entry at its parent is 'deleted' by copying the next key into it.

B. Insertions

Inserting a new key k to the B^S-tree entails searching for the leaf node and the position in it to place it. Search is conducted by applying $succ_{>k}$ operations starting from the root and following the corresponding pointers. When we reach the leaf v where in k should be inserted, we apply a $succ_{\geq k}$ operation which finds the proper slot in v to insert k. Then, we verify whether the slot i returned by $succ_{\geq k}$ is occupied by another key. This is done by a simple test. If $v.keys[i] = v.keys[i+1]$, then we are sure that position i is free, so we place k there and finish. For example, assume that we want to insert key 55 to the leaf node of Figure 3. We apply a $succ_{\geq k}$ operation to the leaf, which will give us position 5. Since the next position (6) has the same key, position 5 corresponds to a free slot (gap), hence, we have directly put the inserted key 55 there. On the other hand, if the key at position i is different compared to the key at position $i+1$, this means that the position is occupied. In this case, we first find the first position j after i, which is unused (i.e., a gap), and right-shift all keys (and the corresponding record pointers) from position i to position $j-1$, to make space, so that key k can be inserted at position i. If there is no free position after i, then we move one position to the left (left-shift) all keys and record ids from position i until the first free position to the left of i. Figure 4 (bottom) shows an example of inserting key 52. As the slot where 52 should go is occupied by 53 and it is not a gap (the key following 53 is not equal to 53), we search for the next gap, which is the position next to 53, right-shift 53 there and make room for the new key 52. Algorithm 6 describes B^S-tree insertion.

In case leaf v is full, then we conduct a *split* of v and introduce a new leaf node. The existing keys in v together with k are split in half and distributed between the two leaves. Instead of placing the distributed keys to the first half of each of the two leaves, we interleave each key with a gap to facilitate fast insertion of future keys.

C. Tree building (bulk loading)

Like typical B$^+$-tree bulk loading algorithms, we first sort the keys to construct the leaf level of the index. To facilitate fast future insertions, we do not pack the nodes with keys, that is we leave free space to accomodate future insertions. Specifically, if N is the capacity of a leaf node, we construct all leaf nodes by adding to them the keys in sorted order; each leaf node takes $\alpha \cdot N$ (key, record-id) pairs, where α ranges from 0.5 (half-full nodes) to 1 (full nodes). We set $\alpha = 0.75$ by default to achieve a good tradeoff between the costs of

ALGORITHM 6: Insertion in B^S-tree

```
Input : key k, B^S-tree root node v
compute leaf v and position r by running lines 1-3 of Alg 3
if v.slotuse < N then
    bitmap ← ¬v.bitmap
    if v.keys[r] == v.keys[r+1] then ▷ slot r is empty
        v.keys[r] ← k
        bitmap ← bitmap ⊕ (0x8000 ≫ r)
    else
        keysForShift ← _lzcnt(bitmap ≪ r)
        if keysForShift < N then ▷ empty to the right
            shift right keysForShift keys of node v
            v.keys[r] ← k
            bitmap ← bitmap ⊕ (0x8000 ≫
              (r + keysForShift))
        else                          ▷ empty slot to the left
            keysForShift ← _tzcnt(bitmap ≫ (N − r − 1)) − 1
            shift left keysForShift keys of node v
            v.keys[r − 1] ← k
            bitmap ← bitmap ⊕ (0x8000 ≫
              (r − (keysForShift + 1)))
        v.slotuse ← slotuse + 1
        v.bitmap ← ¬bitmap
else
    split leaf node v
return
```

queries and updates, after a sensitivity analysis (omitted from the paper, for the interest of space). For each leaf, instead of placing all keys at the beginning of the leaf and leaving $(1-\alpha)N$ consecutive empty slots at the end of the node, we *spread* the entries in the leaf by placing one gap (empty slot) after every $\frac{1}{1-a}-1$ entries.[1] For each leaf node (except the first one) a *separator key*, equal to the first key of the leaf, is added to an array. For each separator key, a node pointer to the previous leaf is associated to the separator. Finally, a node pointer to the last leaf is introduced at the end of the array (without a key value). After constructing the leaves, the (already sorted) array of separator keys is used to construct the next level of B^S-tree (above the leaves), recursively. We increase α as we go up, since we anticipate much fewer insertions (and node overflows) at higher levels.

Complexity. Let f be the fanout of the B^S-tree nodes and assume that each node can be processed by a (small) constant number of SIMD instructions (in our implementation, at most 2 SIMD instructions are executed for each node during searches and updates). Then, the B^S-tree needs $O(n)$ space to accomodate n keys (as any B$^+$-tree does) and the computational cost of search and update operations in B^S-tree is $O(\log_f n)$, because one path is traversed per search or update operation (range queries access two paths) and each node requires a constant number of SIMD instructions.

V. Key Compression

B^S-tree, as it has been discussed so far, stores the exact keys in its nodes. Previous work on key compression for B$^+$-tree [14] uses fixed-size partial keys. One issue with partial keys is the overhead of decompression which may compromise performance. For B^S-tree, we opt for the frame-of-reference (FOR) compression approach, which has minimal decompression overhead.

Specifically, for each node v, we store in the node's auxiliary information (see Figure 3) the first key $v.k_0$ of the node and replace the v's key array (of size N) by an array where each original key k is replaced by the difference $k - k_0$. This allows us to potentially double or quadruple the size of the array if the differences occupy much less space than the original keys. If $N = 16$ and the original array stores 64 bits, it may potentially be replaced by an array of $N = 32$ 32-bit differences or $N = 64$ 16-bit differences. Since the keys in a node are ordered, we expect the differences to be small, especially in leaf nodes, so the space savings due to the reduction in the number of nodes are expected to be significant. To achieve optimal performance of our data-parallel $succ_>$ implementation, we set the key array size to 1024 bits, so N can be 16, 32, or 64. As we have seen in Fig. 2, the cost of our SIMD-based $succ_>$ (Snippet 2) on $N = 64$ `uint16` keys is the same as that on $N = 16$ `uint64` keys. This means that, after compression, the height of the tree can decrease and search can be accelerated.

Tree construction Our goal is to construct the tree in one pass over the sorted keys and to result in leaf nodes having 75% occupancy (except when we are dealing with regions of sequential key values), while achieving the best possible compression. For this, we begin by checking whether the leaf can be filled with 16-bit differences for the keys. If this is not feasible, we reattempt the process by checking if half of the keys can be stored as 32-bit differences. If this attempt also fails, we conclude by storing the exact 64-bit keys.

Search To apply $succ_{>k}$ at a node v we first compute $k' = k - v.k_0$, where $v.k_0$ is the first key value of v, stored explicitly in v's meta-data (this is the only decompression overhead). Then, we apply $succ_{>k'}$ to the node to find the position of the node pointer to follow. The same procedure is applied at the leaf nodes for $succ_{\geq k}$; the position of $k' = k - v.k_0$ corresponds to the position of k, or if $succ_{\geq k'}$ returns NULL, k does not exist.

Insert To insert a new key k, we first run the search algorithm discussed above to find the leaf v and the position in v whereto insert k and then store the difference $k - v.k_0$ there. The new nodes after a splitting a node v can be of the same type as v, or they can be further compressed as they include fewer entries than v with the first and the last one having smaller differences.

Delete Deletion is not affected by key compression. When the key to be deleted is found (represented exactly or by its difference to k_0) at the corresponding leaf node, we simply copy into it the value of the next key, or MAXKEY if the deleted key is the last one in the leaf node. In compressed nodes, MAXKEY is the maximum value that can be represented using all available bits. If the first key of a node is

[1] Gaps between consecutive key values (for integer keys) are not introduced.

deleted, we do not change k_0 (as it is not stored in a slot of the array) and keep in the slots the differences to k_0.

VI. Implementation Details

This section presents some important tuning and implementation details of the B^S-tree.

Node size and structure. As in previous work [13], [21], [38], [42], [44], [56], [58], [64], [66], [67], [69], [71], we aim at indexing large keys, each being a 64-bit unsigned integer. To take full advantage of our SIMD $succ_>$ implementation, each node stores a maximum of $N = 16$ entries; based on this, we allocate $16 \times 64 = 1024$ bits for the keys of each node. Hence, the keys of each node (internal or leaf) fill two cache lines (each cache line can store 64 bytes). This means that for $succ_>$, we perform 2 SIMD instructions per node by loading the keys at 2 registers of 512 bits (8 keys at each register). 1024 bits are also allocated for keys in the compressed CB^S-tree nodes; a compressed key array may have 32 32-bit or 64 16-bit entries. This decision is based on the experiment of Figure 2, which indicates that 1024 bits achieves the best speedup of Snippet 2 over alternative tree branching methods.

Memory management. To store the B^S-tree in memory, we utilize two main structures: one to store the inner nodes and another for the leaf nodes. Each inner node consists of two arrays with 16 entries. The first array holds 64-bit keys, while the second contains 32-bit references to nodes. 32 bits are sufficient for the references because they are in fact offsets to fixed-length slots in memory arrays allocated for nodes (one for inner nodes and one for leaf nodes).[2] The auxiliary data for each node are put in a separate dedicated array aligned with the node arrays. Hence, each inner node has a size of 192 bytes, which fits into 3 cache lines. Our tested B^S-tree implementation only has keys and no record-ids in its leaves, so a leaf node contains a single array of 16 64-bit keys, with each leaf node occupying 128 bytes, fitting into 2 cache lines. The inner and leaf nodes are stored in a contiguous array, aligned to Transparent Huge Pages (2 MB) for efficiency. Alignment plays a crucial role in optimizing both cache efficiency and the use of SIMD operations, making it a key factor in the performance of B^S-tree. By aligning data to cache lines, we minimize cache misses and ensure that the CPU can retrieve entire nodes in a single memory access, significantly speeding up operations. By aligning nodes to huge pages (2 MB), we reduce translation lookaside buffer (TLB) misses. We also make use of __builtin_prefetch, a compiler intrinsic that allows us to pre-load data into the cache before it is needed, reducing latency. By combining cache-line and SIMD-friendly alignment, along with appropriate use of __builtin_prefetch, B^S-tree allows for SIMD acceleration, and reduces memory access latency, leading to significantly better overall performance.

Compress or not? The compressed version of B^S-tree with variable-capacity nodes (Section V) may reduce the memory footprint of the index and improve its performance, but also comes with the overhead of explicitly keeping the first key of a node, which does not pay off for nodes having 64-bit differences that cannot be compressed.[3] Hence, we employ a *decision mechanism* for choosing between the construction of a B^S-tree or a compressed B^S-tree, based on the input data. The first key is stored in the auxiliary structure. Before bulk-loading the tree, we virtually split the sorted keys input into segments of 13 keys each, subtract the smallest key from the largest key in each bucket, and calculate the number of leading zeros. After performing these calculations for all segments, we take the average number of leading zeros. If this average is greater or equal to 32 bits, we conclude that the dataset can benefit from a compact B^S-tree compression, and we go ahead with its construction. Otherwise, we create a standard (uncompressed) B^S-tree. The selection of 13 keys is not arbitrary, as we put 25% gaps at each leaf, and the 13th key serves as the separator for the node. We found out that compression is not effective for inner nodes, so our final compressed B^S-tree implementation has uncompressed inner nodes and compressed leaves.

VII. Concurrency control

A number of concurrency control techniques has been proposed for the B$^+$-tree and other indices, including lock coupling [12], [27], right-sibling pointers [39], fine-grained locking with lock coupling and logical removals [17], Bw-tree's lock-free mechanism [44], [61] and Read-Optimized Write Exclusion (ROWEX) [43]. For B-trees, Leis et al. [41], [43] proposed an Optimistic Lock Coupling (OLC) technique, which is easy to implement and highly efficient. In OLC, when a thread wants to read or modify a node, it first acquires an optimistic read lock, allowing it to traverse the tree while maintaining a local copy of the node's state. If the thread intends to perform an update, it checks whether the node has been modified by another thread since it was read. If not, it commits the changes atomically. In the event of a conflict (i.e., if another thread has modified the node), the thread rolls back its changes and retries the operation from the root.

Our current implementation of B^S-tree employs the OLC mechanism [41], [43], because it preserves the original tree structure, adding only a lightweight version counter per node to ensure atomicity and isolation. We introduced a slight modification to OLC in how node splits are handled. In the original OLC, when a thread takes on a write task, it splits the first full node (inner or leaf) it encounters during traversal. After completing the split, the thread restarts its traversal until it finds a path where every node has at least one empty slot. This efficiently reorganizes node contents, ensuring that insertions triggering splits at multiple tree levels are managed without requiring a global lock. For example, when a split occurs at a leaf node, the restart process is only necessary if its parent node has no free slot to fit the separator of the two

[2] If we had to use 64-bit references, the search performance of the B^S-tree only drops by up to 5%, according to our tests.

[3] As we want all leaf nodes to have the same fixed size (for alignment purposes), we do not allow the same B^S-tree to have both uncompressed and compressed leaves.

new leaves. On the other hand, if the parent node has a free slot, restarting the process is not necessary.

VIII. Experiments

We experimentally compare B^S-tree to alternative main-memory indices (learned and non-learned). As in previous work [13], [42], [62], we have built and compared indices for key data only; record ids or values are not stored or accessed in any competitor index, but the objective of each index is to locate the position(s) of the searched key(s). The implementation of all methods is in C++ and compiled with `gcc` (v13) using the flags `-O3` and `-march=native`. The experiments were conducted on a system with an 11th Gen Intel® Core™ i7-11700K processor with 8 cores (at 3.60 GHz), 128 GB of RAM, having AVX 512 support. The operating system used was Ubuntu 22.04. We extended the codebase of GRE [62] to include B^S-tree.

A. Setup

Datasets. We ran our tests on standard benchmarking real datasets of varying sparsity, used in previous work [34], [49], [62]; each one consists of unsigned 64-bit integer keys. In Amazon BOOKS [34], [49], each key represents the popularity of a specific book. In FB [34], [49], [55], each key is a Facebook user-id. OSM [34], [49] contains unique integer-encoded locations from OpenStreetMap. GENOME [54], [62] includes loci pairs from human chromosomes. PLANET [2], [62], a planet-wide collection of integer-encoded geographic locations compiled by OpenStreetMap. According to [62], [71], OSM, FB, GENOME, and PLANET are complex real-world datasets that can pose challenges for learned indices. In contrast, the key distribution of BOOKS is easy to learn. We did not conduct experiments using synthetic datasets with common distributions, as, according to [49], it would be trivial for a learned index to model such distributions. Although we do not include experiments with non-integer keys, B^S-tree can be used for floats, by changing the SIMD intrinsics in Snippet 2, and for strings, after being dictionary encoded (see [22]).

Competitors. We compare our proposed **B^S-tree** and its compressed version, denoted by **CB^S-tree**, with five updatable learned and non-learned indices, for which the code was publicly available. We did not compare to methods found inferior in previous comparative studies [13], [62] and to those with proprietary or unavailable code (e.g., [38], [71]).

Non-learned Indices. STX library [1] is a fully optimized C++ implementation of a main-memory B$^+$-tree. We use the set-based implementation from STX, which *does not store values in the leaf nodes*. For its construction, we used its fast bulk-loading method. We used the default block size of STX (256 bytes), so each leaf node holds 32 keys ($32 \times 8 = 256$ bytes). We used two versions of the STX tree: the first is the original code, referred to as **B$^+$-tree**, while the second version creates 25% empty space at the end of each leaf node, denoted by **Sparse B$^+$-tree** (for fairness, as our B^S-tree also proactively introduces 25% of gaps in leaves).

TABLE I: Construction time (for 150 million keys)

Construction Time (sec)					
Indices / Datasets	**BOOKS**	**OSM**	**FB**	**GENOME**	**PLANET**
B^S-tree	**0.33**	0.33	0.33	0.33	0.33
CB^S-tree	0.35	**0.32**	**0.18**	**0.20**	**0.18**
B$^+$-tree	0.39	0.39	0.39	0.39	0.39
Sparse B$^+$-tree	0.50	0.50	0.50	0.50	0.50
HOT	15.61	16.65	16.56	16.23	15.33
ART	5.65	6.11	6.62	6.42	6.19
ALEX	25.43	41.60	45.46	30.74	30.06
LIPP	9.58	9.31	6.98	7.01	7.05

We also compare to SIMD-based implementations of **HOT** [3], [13] and **ART** [7], [42]. These tries do not support bulk-loading. However, we found that pre-sorting the data improves the construction time for both and results in more efficient structures. The HOT code release does not support range queries, so we implemented them ourselves. HOT cannot handle keys greater than $2^{63} - 1$, so we removed values exceeding this limit from certain datasets.

Learned Indices. **ALEX** [5], [21] and **LIPP** [6], [64] are the state-of-the-art updatable learned indices for key-value pairs [62]. To use them, for each dataset we used as value of each key the key itself. ALEX and LIPP both support bulk-loading. Neither uses SIMD intrinsics during search. The reason is their large leaves (in the order of KB), where SIMD-based search is not effective (see Fig. 2). ALEX uses exponential search in leaves, while LIPP eliminates the last-mile search by ensuring that predicted positions are exact for each key in the index.

Note that all B^S-tree competitors do not utilize huge pages. In their implementations, each node occupies a random location in memory and connects to its child nodes via pointers. This design results in the retrieval of a significantly larger number of memory pages into the TLB cache, thereby diminishing the potential advantage of using huge pages.

Workloads. We set up different workloads to measure performance. First, we randomly selected 150 million entries from each dataset and bulk-loaded them to an index (HOT and ART do not support bulk-loading, however, they both benefit from sorting). For our workloads, we used 50 million keys, that are selected randomly (i.e., queries and updates hit a random region of the space). Our workloads are:

- **Workload A** (Read-Only): 100% equality searches.
- **Workload B** (Write-Only): 100% writes (insertions).
- **Workload C** (Read-W): 50% reads, 50% writes.
- **Workload D** (Range-W): 95% range searches, 5% writes.
- **Workload E** (Mixed): 60%/35%/5% reads/writes/deletes.
- **Workload F** (Update-Intens.): 50% writes, 50% deletes.

B. Construction Cost and Memory Footprint

Table I presents the construction times of all tested methods, while Table II shows their memory footprints, for 150M keys. We exclude sorting from the construction cost. B^S-tree construction time also includes the decision-making mechanism (roughly takes 0.03 sec) on whether we will construct a B^S-tree or a CB^S-tree (see Section VI). To calculate the memory

TABLE II: Memory footprint (for 150 million keys)

Memory Footprint (GB)					
Indices / Datasets	**BOOKS**	**OSM**	**FB**	**GENOME**	**PLANET**
B^S-tree	1.84	1.84	1.84	1.84	1.84
CB^S-tree	2.03	1.75	**0.55**	**0.80**	**0.51**
B^+-tree	**1.41**	**1.41**	1.41	1.41	1.41
Sparse B^+-tree	1.88	1.88	1.88	1.88	1.88
HOT	1.78	1.79	1.83	1.92	1.71
ART	3.86	4.12	3.88	3.78	3.66
ALEX	2.73	2.77	2.77	2.73	2.73
LIPP	13.51	14.69	10.89	11.66	11.62

usage of each method, we utilize the C function `getrusage`[4]. Since all learned indices essentially store 64-bit values together with the keys, we report half of their measured memory requirements, to approximate the memory required just for the keys and their inner structure.

As expected, non-learned indices (except from the CB^S-tree) have a stable construction time and memory footprint. On the other hand, learned indices involve model training and take longer to build. The Sparse B^+-tree is larger than B^+-tree, so it is costlier to build and has a larger memory footprint.

CB^S-tree has the smallest memory footprint than all methods for FB, GENOME, and PLANET because of its high compression effectiveness, which also has a positive impact to the construction time. On the other hand, CB^S-tree occupies more space than B^S-tree on BOOKS because the distribution of keys there does not provide many compression opportunities. Note that for BOOKS and OSM, our decision mechanism (see Section VI) chooses to construct a B^S-tree while for FB, GENOME, and PLANET it decides to construct a CB^S-tree.

HOT is very expensive to build compared to B^S-tree and B^+-tree, because HOT requires keys to be inserted one at a time, lacking a bulk-loading mechanism. HOT uses slightly less memory than B^S-tree; however, our CB^S-tree has a significantly smaller memory footprint than HOT in three out of the five datasets. ART is faster to build compared to HOT (but occupies more space), due to its simpler insertion process.

In conclusion, the B^S-tree has low construction cost, with the B^+-tree exhibiting comparable performance alongside a very small memory footprint. Additionally, the CB^S-tree achieves the fastest construction time and consumes from 56% to 94% less memory than all methods in FB, GENOME, and PLANET. Our results align with the findings of Wongkham et al. [62], which conclude that memory efficiency is not a distinct advantage of updatable learned indices.

C. Single-Threaded Throughput

Next, we evaluate the throughput of all methods (single-threaded) on the five workloads described in Section VIII-A. Figure 5 presents the throughput (millions of operations per second) of all methods for Workload A (read-only). B^S-tree and CB^S-tree outperform all competitors across the board except for BOOKS, where ALEX is marginally faster than CB^S-tree and has the same throughput as B^S-tree. The excellent performance of ALEX on BOOKS is due to the smooth key distribution there, which is easy for ALEX to learn. On average, our methods have a significant performance gap compared to the nearest competitor. Specifically, B^S-tree is roughly 2.5x faster than HOT on OSM, 1.5x faster than LIPP on FB and GENOME and 2.2x faster than ART on FB and GENOME. CB^S-tree is about 7% slower than ALEX on BOOKS, but much faster than previous work on all other datasets and even faster than B^S-tree on FB, GENOME, and PLANET, while having a much smaller memory footprint. CB^S-tree exploits the highly compressible keys of FB, GENOME, and PLANET to drastically reduce the capacity of leaf nodes and the overall space required for the index. This increases the likelihood that multiple searches hit the same leaves, exploiting the memory cache, as we will also show in the next set of experiments.

For Workload B (write-only), as Figure 6 shows, B^S-tree outperforms all methods, while CB^S-tree loses to ALEX only on BOOKS. ART achieves relatively good performance in this workload because of its lazy expansion and path compression. LIPP is more robust than ALEX due to its accurate prediction mechanism, which reduces the need for frequent node splits or rebalancing due to the more appropriate placement of keys during its construction. Observe that the Sparse B^+-tree is faster than the B^+-tree because it requires much fewer splits. CB^S-tree has competitive write performance to B^S-tree.

On Workload C (Figure 7), the performance of all methods stands between that of Workload A (read-only) and Workload B (write-only), which is expected. For the results of Workload D (range-write), see Figure 8. The results are similar compared to Workload A (read-only) since Workload D is read-heavy. Range queries retrieve 153 keys on average. LIPP performs much worse than ALEX, as LIPP's structure is not optimized for range queries. On the other hand, ALEX has large nodes and facilitates jumps to sibling nodes. HOT and ART are not optimized for range queries, so they performs poorly. On the compressible datasets (FB, GENOME, PLANET), CB^S-tree outperforms B^S-tree as its compressed leaves have larger capacities and fewer leaves need to be scanned per query. Figure 9 shows the results using Workload E (read-write-delete). The results are similar to those for Workloads A and B. B^S-tree and CB^S-tree outperform all competitors across all datasets except for BOOKS, where CB^S-tree is slightly inferior to ALEX. Deletions do not impose an overhead to all methods. Finally, Figure 10 shows performance on the update-heavy workload F with a mix of writes and deletions. The relative performance of the methods is the similar as for other workloads that include many writes (e.g., Workloads B and C). Once again, ALEX is favored by the easy-to-learn distribution of BOOKS, but it does not perform well on other datasets.

D. Performance Counters

Besides throughput, we also compared all methods with respect to various performance counters, including instructions executed, cycles, mispredicted branches, and misses in L1 and TLB (Translation Lookaside Buffer) misses. As representative

[4]https://man7.org/linux/man-pages/man2/getrusage.2.html

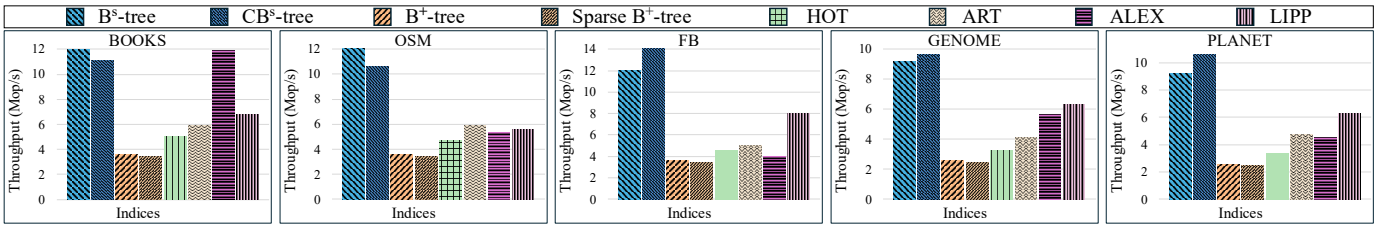


Fig. 5: Workload A: Read Only (100%), single-threaded throughput

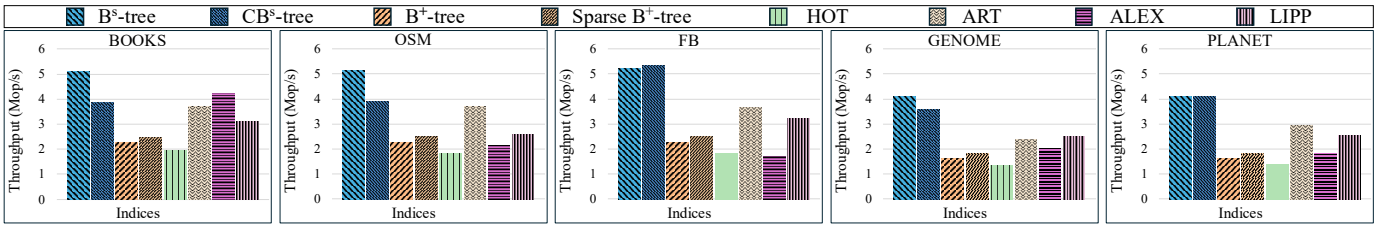


Fig. 6: Workload B: Write Only (100%), single-threaded throughput

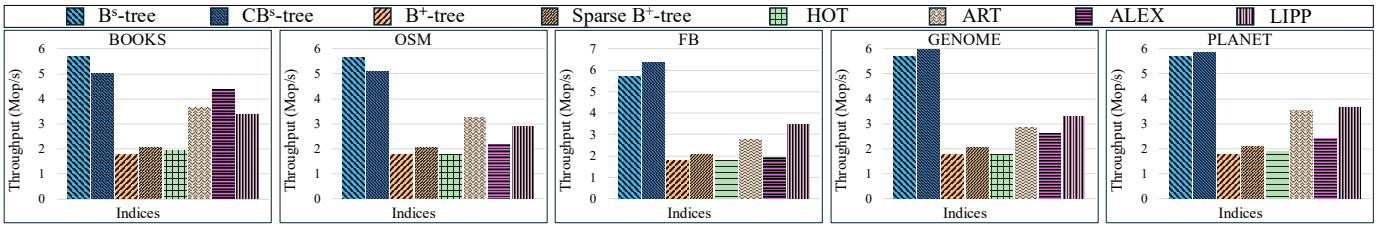


Fig. 7: Workload C: Read (50%) - Write (50%), single-threaded throughput

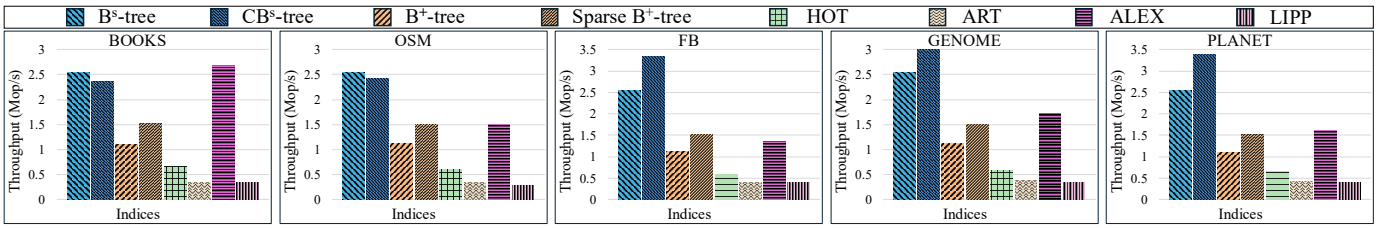


Fig. 8: Workload D: Range (95%) - Write (5%), single-threaded throughput

datasets, we selected BOOKS and FB.[5] We chose to measure Workload C (reads - writes), as it is update-heavy and the unpredictable nature of writes can lead to numerous splits, stressing the indices. To calculate these metrics, we utilized Leis's perf_event code [4]. The average performance measures per operation for all methods are presented in Table III and Table IV for BOOKS and FB, respectively.

B^S-tree needs the smallest number of instructions and cycles on BOOKS, with CB^S-tree being a close runner up. For FB, CB^S-tree needs less cycles, which explains its superiority to B^S-tree. Our methods are simple and efficient, benefiting from the use of SIMD instructions and alignment, which reduce the number of cycles and instructions required for each task. They have the fewest mispredicted branches, which is expected due to their branchless search; mispredicted branches in B^S-tree arise from the insertions. Regarding L1, on average, our algorithms incur 16 cache misses, which is consistent with our expectation. Specifically, the height of our trees is 6 and their fanout is 16; we encounter 2 cache misses per tree level and 2 cache misses at the leaf level ($2 \times 6 + 2 = 14$ misses), with the remaining 2 misses caused by insertions. For BOOKS, ART achieves the best performance in terms of L1 cache misses, though our algorithms are close in comparison. Lastly, in terms of TLB misses, our algorithms exhibit outstanding performance relative to our competitors. This can be attributed to our use of huge pages (see Sec. VI). Overall, the performance counters show that our B^S-tree and CB^S-tree are fully optimized and cache-efficient via the use of SIMD instructions, huge pages, and branchless code.

[5]Recall that ALEX presents competitive behavior on BOOKS, whereas on FB B^S-tree and CB^S-tree have a large performance gap to other methods.

E. Multi-Threaded Workloads

Next, we present multi-threaded experiments, where workloads are executed by multiple threads simultaneously. For concurrency control, our B^S-tree and CB^S-tree use OLC as described in Section VII, ART and B$^+$-tree use OLC, HOT uses ROWEX, and ALEX and LIPP use optimistic locking. We applied Workload A, Workload B, and Workload C, which are the most representative ones. Figure 11, presents

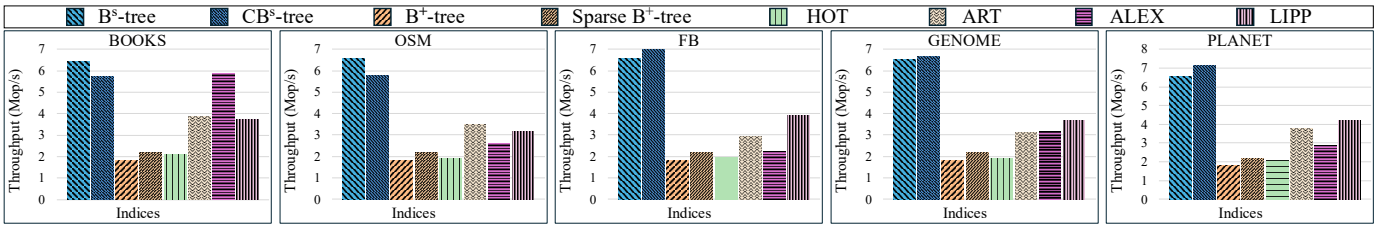


Fig. 9: Workload E: Read (60%) - Write (35%) - Deletions (5%), single-threaded throughput

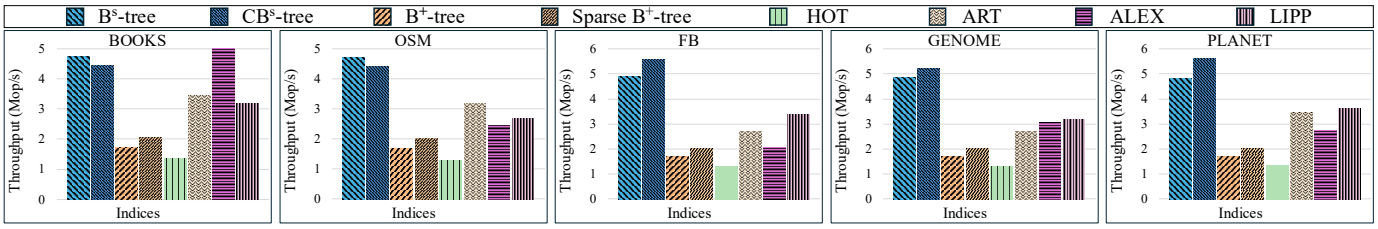


Fig. 10: Workload F: Write (50%) - Deletions (50%), single-threaded throughput

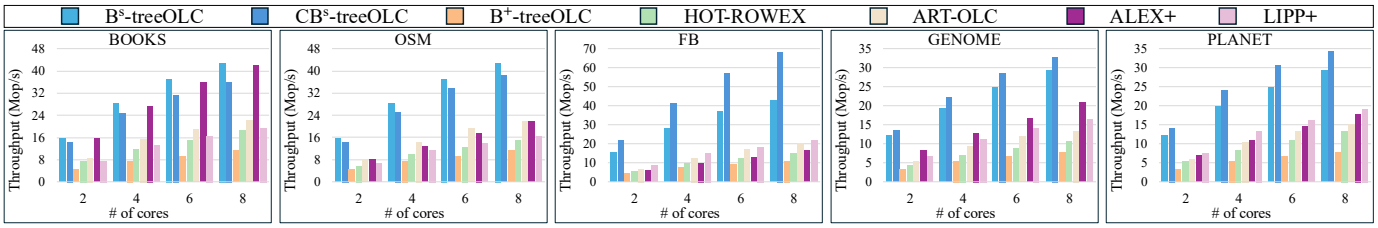


Fig. 11: Workload A: Read Only (100%), multi-threaded throughput

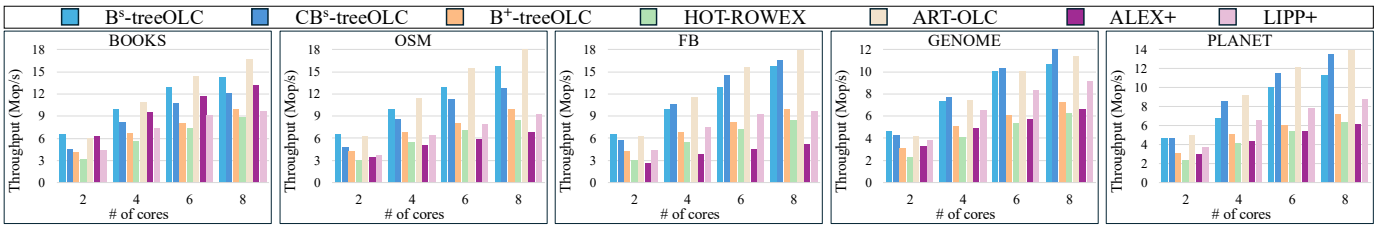


Fig. 12: Workload B: Write Only (100%), multi-threaded throughput

TABLE III: Performance counters for BOOKS, Workload C

Workload C - Dataset: BOOKS					
Indices / Events	**Instr.**	**Cycles**	**Misp. Branches**	**L1 Misses**	**TLB Misses**
B^S-tree	**220.18**	**884.16**	**1.03**	16.41	0.61
CB^S-tree	277.87	997.57	1.03	16.33	**0.05**
B$^+$-tree	656.84	2806.93	11.49	33.49	4.44
Sparse B$^+$-tree	565.33	2408.02	10.61	28.29	3.84
HOT	898.99	2585.71	3.44	31.78	4.75
ART	435.56	1443.29	2.12	**14.12**	3.06
ALEX	612.57	1165.58	5.23	21.03	2.22
LIPP	300.47	1379.23	1.95	18.66	5.02

TABLE IV: Performance counters for FB, Workload C

Workload C - Dataset: FB					
Indices / Events	**Instr.**	**Cycles**	**Misp. Branches**	**L1 Misses**	**TLB Misses**
B^S-tree	**220.51**	877.78	**1.03**	16.43	0.59
CB^S-tree	269.33	**784.75**	1.09	**14.29**	**0.00**
B$^+$-tree	655.42	2807.87	11.54	33.45	4.41
Sparse B$^+$-tree	566.01	2395.46	10.62	28.37	3.85
HOT	962.58	2816.29	4.08	33.47	5.06
ART	603.69	1804.94	2.33	14.47	3.49
ALEX	1049.11	2634.38	6.97	40.31	4.72
LIPP	276.74	1350.58	1.49	18.62	5.11

the throughput (millions of operations per second) of all methods for Workload A (read-only). Observe that the access methods have the same relative performance as in single-threaded processing. Our trees outperform all competitors, with ALEX being competitive only on BOOKS. For the write-only Workload B (Figure 12), ART achieves the best performance, while our trees are competitive in some cases. ART's strong performance is largely due to OLC, which is well-suited to its structure. For the mixed read-write Workload C (Figure 13), either B^S-tree or CB^S-tree outperforms all competitors across all datasets. All methods scale well with the number of cores.

F. Scalability Tests

We also evaluated the performance of all tested methods for different data and workload scales. Specifically, we used the BOOKS dataset with 150, 300, 450, and 600 million keys under a 50%-50% read–write workload (Workload C), with 50 million operations. As Figure 14(a) shows, B^S-tree and CB^S-tree consistently achieve the best performance as the dataset size increases. We observe a small decrease in the throughput of all indices with the number of initial keys, which

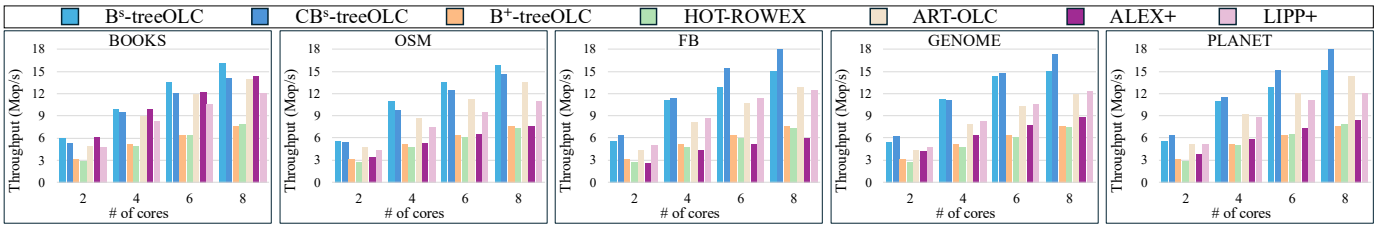
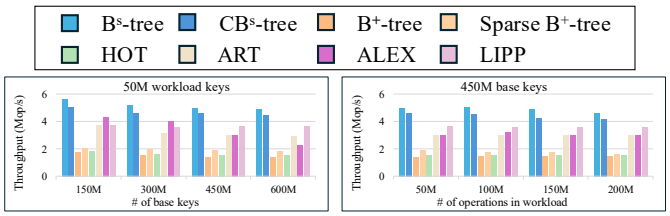


Fig. 13: Workload C: Read (50%) - Write (50%), multi-threaded throughput

is expected. Figure 14(b) shows how the average throughput is affected if the number of operations on indices, with 450M keys initially, grows from 50M to 200M (Workload C on BOOKS). Note that the relative performance of all methods is not affected by the scale of the workload.



(a) Data scalability (b) Workload scalability

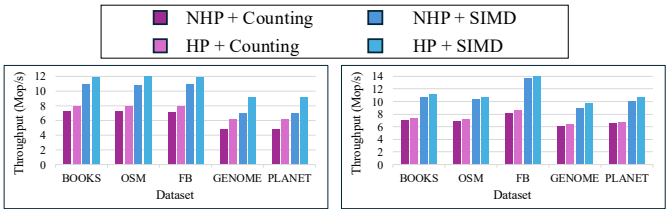
Fig. 14: Scalability experiments (BOOKS, Workload C)

G. Impact of B^S-tree design

In the final set of experiments, we assess the impact of the design choices in B^S-tree. Specifically, we analyze the effectiveness of the B^S-tree features, with the most powerful ones being SIMD-based branching (Snippet 2) and transparent huge pages. We implemented four versions of our structure: one auto-vectorized version without huge pages (NHP + Counting), one with only huge pages enabled (HP + Counting), another using only our Snippet 2 (NHP + SIMD), and finally, a fully optimized version that combines both huge pages and AVX instructions (HP + SIMD). Figure 15 shows the throughput of the four versions of B^S-tree for Workload A (reads). We observe that both optimizations (HP and SIMD) boost B^S-tree. Additionally, we find that even the basic version of B^S-tree (NHP + Counting) remains competitive with all other methods shown in Figure 15. Huge pages do not improve the performance of CB^S-tree as much as they do for B^S-tree. For datasets that can be compressed, CB^S-tree produces larger leaf nodes (storing up to 64 keys of 16 bits), which increases the likelihood that the corresponding leaf node is already in the cache. On the other hand, for datasets that cannot be compressed, there is an overhead for decoding the node type (16 keys of 64 bits, 32 keys of 32 bits, or 64 keys of 16 bits).

H. Summary of Experimental Findings

In summary, B^S-tree and CB^S-tree exhibit excellent and robust performance for different workloads and different datasets of varying distribution, being superior than all competitors in most cases, in single- and multi-threaded processing. They also have the lowest construction cost and memory footprint.



(a) B^S-tree (b) CB^S-tree

Fig. 15: Effect of implementation design under Workload A

Note that our decision mechanism, which imposes a small overhead in the construction (up to 10% of the construction cost) decides automatically and correctly which of B^S-tree or CB^S-tree to build for a given dataset. B^S-tree outperforms trie-based indices like HOT and ART, except for multithreaded write-heavy workloads, where ART is superior. Still, we have not implemented yet B^S-tree for string keys, where trie-based structures are known to perform best. Regarding updatable learned indices, our study shows that they are typically outperformed by optimized non-learned indices like our B^S-tree.

IX. Conclusions

We proposed B^S-tree, a main-memory B^+-tree with a data-parallel implementation for branching at each level during search and updates. B^S-tree is based on a novel representation of gaps in nodes by duplicating existing keys, which does not affect SIMD-based branchless search in nodes. B^S-tree uses FOR compression for nodes that include keys with small differences. Our experimental evaluation demonstrates the superiority of B^S-tree compared to open-source state-of-the-art non-learned and learned indices, with respect to construction time, memory footprint, and throughput for various workloads that include queries and updates in single- and multi-threaded processing. Inspired by [58], in the future, we plan to implement B^S-tree in a hybrid setup, where the top (and infrequently updated levels) are handled by the GPU that allows much higher data parallelism and the lower (frequently updated) levels are handled by the CPU. In addition, we will study the support of string keys in B^S-tree (one solution is to use Binary, ASCII or Base64 encoding [21]).

Acknowledgements

Work supported by project MIS 5154714 of the National Recovery and Resilience Plan Greece 2.0 funded by the European Union under the NextGenerationEU Program.

REFERENCES

[1] Stx b+tree code, 2013. https://github.com/bingmann/stx-btree/tree/master.

[2] Google cloud. openstreetmap., 2017. https://console.cloud.google.com/marketplace/details/openstreetmap/geo-openstreetmap?project=practice-bigtable.

[3] Hot code, 2018. https://github.com/speedskater/hot.

[4] Perfevent code, 2018. https://github.com/viktorleis/perfevent?tab=readme-ov-file.

[5] Alex code, 2020. https://github.com/microsoft/ALEX.

[6] Lipp code, 2021. https://github.com/Jiacheng-WU/lipp.

[7] Art code, 2022. https://github.com/pohchaichon/ARTSynchronized/tree/808372ded6b8c5a6d3a1741090510b79042f2aa7.

[8] D. Agrawal, D. Ganesan, R. K. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: An optimized index structure for flash devices. *Proc. VLDB Endow.*, 2(1):361–372, 2009.

[9] N. Askitis and R. Sinha. Hat-trie: A cache-conscious trie-based data structure for strings. In G. Dobbie, editor, *Computer Science 2007. Proceedings of the Thirtieth Australasian Computer Science Conference (ACSC2007). Ballarat, Victoria, Australia, January 30 - February 2, 2007. Proceedings*, volume 62 of *CRPIT*, pages 97–105. Australian Computer Society, 2007.

[10] N. Askitis and R. Sinha. Engineering scalable, cache and space efficient tries for strings. *VLDB J.*, 19(5):633–660, 2010.

[11] M. Athanassoulis and A. Ailamaki. Bf-tree: Approximate tree indexing. *Proc. VLDB Endow.*, 7(14):1881–1892, 2014.

[12] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9:1–21, 1977.

[13] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis. HOT: A height optimized trie index for main-memory database systems. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 521–534. ACM, 2018.

[14] P. Bohannon, P. McIlroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. In S. Mehrotra and T. K. Sellis, editors, *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 163–174. ACM, 2001.

[15] M. Böhm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner. Efficient in-memory indexing with generalized prefix trees. In T. Härder, W. Lehner, B. Mitschang, H. Schöning, and H. Schwarz, editors, *Datenbanksysteme für Business, Technologie und Web (BTW), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 2.-4.3.2011 in Kaiserslautern, Germany*, volume P-180 of *LNI*, pages 227–246. GI, 2011.

[16] J. Boyar and K. S. Larsen. Efficient rebalancing of chromatic search trees. In O. Nurmi and E. Ukkonen, editors, *Algorithm Theory - SWAT '92, Third Scandinavian Workshop on Algorithm Theory, Helsinki, Finland, July 8-10, 1992, Proceedings*, volume 621 of *Lecture Notes in Computer Science*, pages 151–164. Springer, 1992.

[17] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In R. Govindarajan, D. A. Padua, and M. W. Hall, editors, *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pages 257–268. ACM, 2010.

[18] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In S. Mehrotra and T. K. Sellis, editors, *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 235–246. ACM, 2001.

[19] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching b$\pm$trees: optimizing both cache and disk performance. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, pages 157–168. ACM, 2002.

[20] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, 2015.

[21] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. B. Lomet, and T. Kraska. ALEX: an updatable adaptive learned index. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 969–984. ACM, 2020.

[22] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *Proc. VLDB Endow.*, 14(2):74–86, 2020.

[23] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 1989.

[24] P. Ferragina and G. Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.*, 13(8):1162–1175, 2020.

[25] J. Fix, A. Wilkes, and K. Skadron. Accelerating braided b+ tree searches on a gpu with cuda. In *2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC), in conjunction with ISCA*. Citeseer, 2011.

[26] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. Fiting-tree: A data-aware index structure. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1189–1206. ACM, 2019.

[27] G. Graefe. Modern b-tree techniques. *Found. Trends Databases*, 3(4):203–402, 2011.

[28] G. Graefe. More modern b-tree techniques. *Found. Trends Databases*, 13(3):169–249, 2024.

[29] G. Graefe and P. Larson. B-tree indexes and CPU caches. In D. Georgakopoulos and A. Buchmann, editors, *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 349–358. IEEE Computer Society, 2001.

[30] P. Jin, C. Yang, C. S. Jensen, P. Yang, and L. Yue. Read/write-optimized tree indexing for solid-state drives. *VLDB J.*, 25(5):695–717, 2016.

[31] M. V. Jørgensen, R. B. Rasmussen, S. Saltenis, and C. Schjønning. Fb-tree: a b$^+$-tree for flash-based ssds. In B. C. Desai, I. F. Cruz, and J. Bernardino, editors, *15th International Database Engineering and Applications Symposium (IDEAS 2011), September 21 - 27, 2011, Lisbon, Portugal*, pages 34–42. ACM, 2011.

[32] K. Kaczmarski. B $^+$ -tree optimized for GPGPU. In R. Meersman, H. Panetto, T. S. Dillon, S. Rinderle-Ma, P. Dadam, X. Zhou, S. Pearson, A. Ferscha, S. Bergamaschi, and I. F. Cruz, editors, *On the Move to Meaningful Internet Systems: OTM 2012, Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2012, Rome, Italy, September 10-14, 2012. Proceedings, Part II*, volume 7566 of *Lecture Notes in Computer Science*, pages 843–854. Springer, 2012.

[33] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: fast architecture sensitive tree search on modern cpus and gpus. In A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 339–350. ACM, 2010.

[34] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. Sosd: A benchmark for learned indexes. *NeurIPS Workshop on Machine Learning for Systems*, 2019.

[35] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. Radixspline: a single-pass learned index. In R. Bordawekar, O. Shmueli, N. Tatbul, and T. K. Ho, editors, *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*, pages 5:1–5:5. ACM, 2020.

[36] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. *KISS-Tree*: smart latch-free in-memory indexing on modern architectures. In S. Chen and S. Harizopoulos, editors, *Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN 2012, Scottsdale, AZ, USA, May 21, 2012*, pages 16–23. ACM, 2012.

[37] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 489–504. ACM, 2018.

[38] Y. Kwon, S. Lee, Y. Nam, J. C. Na, K. Park, S. K. Cha, and B. Moon. Db+-tree: A new variant of b+-tree for main-memory database systems. *Inf. Syst.*, 119:102287, 2023.

[39] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[40] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In W. W. Chu, G. Gardarin,

S. Ohsuga, and Y. Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 294–303. Morgan Kaufmann, 1986.

[41] V. Leis, M. Haubenschild, and T. Neumann. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. *IEEE Data Eng. Bull.*, 42(1):73–84, 2019.

[42] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In C. S. Jensen, C. M. Jermaine, and X. Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 38–49. IEEE Computer Society, 2013.

[43] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*, pages 3:1–3:8. ACM, 2016.

[44] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In C. S. Jensen, C. M. Jermaine, and X. Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 302–313. IEEE Computer Society, 2013.

[45] P. Li, H. Lu, R. Zhu, B. Ding, L. Yang, and G. Pan. DILI: A distribution-driven learned index. *Proc. VLDB Endow.*, 16(9):2212–2224, 2023.

[46] Y. Li, B. He, J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 3(1):1195–1206, 2010.

[47] J. Liu, S. Chen, and L. Wang. Lb+-trees: Optimizing persistent index performance on 3dxpoint memory. *Proc. VLDB Endow.*, 13(7):1078–1090, 2020.

[48] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In P. Felber, F. Bellosa, and H. Bos, editors, *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 183–196. ACM, 2012.

[49] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska. Benchmarking learned indexes. *Proc. VLDB Endow.*, 14(1):1–13, 2020.

[50] D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[51] G.-J. Na, S.-W. Lee, and B. Moon. Dynamic in-page logging for b^+-tree index. *IEEE Transactions on Knowledge and Data Engineering*, 24(7):1231–1243, 2012.

[52] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 78–89. Morgan Kaufmann, 1999.

[53] J. Rao and K. A. Ross. Making b^+-trees cache conscious in main memory. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 475–486. ACM, 2000.

[54] S. S. Rao, M. H. Huntley, N. C. Durand, E. K. Stamenova, I. D. Bochkov, J. T. Robinson, A. L. Sanborn, I. Machol, A. D. Omer, E. S. Lander, et al. A 3d map of the human genome at kilobase resolution reveals principles of chromatin looping. *Cell*, 159(7):1665–1680, 2014.

[55] P. V. Sandt, Y. Chronis, and J. M. Patel. Efficiently searching in-memory sorted arrays: Revenge of the interpolation search? In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 36–53. ACM, 2019.

[56] B. Schlegel, R. Gemulla, and W. Lehner. k-ary search on modern processors. In P. A. Boncz and K. A. Ross, editors, *Proceedings of the Fifth International Workshop on Data Management on New Hardware, DaMoN 2009, Providence, Rhode Island, USA, June 28, 2009*, pages 52–60. ACM, 2009.

[57] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proc. VLDB Endow.*, 4(11):795–806, 2011.

[58] A. Shahvarani and H. Jacobsen. A hybrid b+-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms. In F. Özcan, G. Koutrika, and S. Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1523–1538. ACM, 2016.

[59] D. Siakavaras, P. Billis, K. Nikas, G. I. Goumas, and N. Koziris. Efficient concurrent range queries in b+-trees using RCU-HTM. In C. Scheideler and M. Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 571–573. ACM, 2020.

[60] L. Wang, Z. Zhang, B. He, and Z. Zhang. Pa-tree: Polled-mode asynchronous B+ tree for nvme. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 553–564. IEEE, 2020.

[61] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. Building a bw-tree takes more than just buzz words. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 473–488. ACM, 2018.

[62] C. Wongkham, B. Lu, C. Liu, Z. Zhong, E. Lo, and T. Wang. Are updatable learned indexes ready? *Proc. VLDB Endow.*, 15(11):3004–3017, 2022.

[63] C. Wu, T. Kuo, and L. Chang. An efficient b-tree layer implementation for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.*, 6(3):19, 2007.

[64] J. Wu, Y. Zhang, S. Chen, Y. Chen, J. Wang, and C. Xing. Updatable learned index with precise positions. *Proc. VLDB Endow.*, 14(8):1276–1288, 2021.

[65] S. Wu, Y. Cui, J. Yu, X. Sun, T. Kuo, and C. J. Xue. NFL: robust learned index via distribution transformation. *Proc. VLDB Endow.*, 15(10):2188–2200, 2022.

[66] Z. Yan, Y. Lin, L. Peng, and W. Zhang. Harmonia: a high throughput b+tree for gpus. In J. K. Hollingsworth and I. Keidar, editors, *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, pages 133–144. ACM, 2019.

[67] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In F. Özcan, G. Koutrika, and S. Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1567–1581. ACM, 2016.

[68] H. Zhang, G. Chen, B. C. Ooi, K. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Trans. Knowl. Data Eng.*, 27(7):1920–1948, 2015.

[69] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. Surf: Practical range query filtering with fast succinct tries. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 323–336. ACM, 2018.

[70] J. Zhang and Y. Gao. CARMI: A cache-aware learned index with a cost-based construction algorithm. *Proc. VLDB Endow.*, 15(11):2679–2691, 2022.

[71] S. Zhang, J. Qi, X. Yao, and A. Brinkmann. Hyper: A high-performance and memory-efficient learned index via hybrid construction. *Proc. ACM Manag. Data*, 2(3):145, 2024.

[72] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, pages 145–156. ACM, 2002.

[73] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, editors, *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, pages 405–416. Morgan Kaufmann, 2003.