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

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

Achilleas Michalopoulos
University of Ioannina & Archimedes, Athena RC
amichalopoulos@cse.uoi.gr

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

Manolis Terrovitis
Athena RC, Athens, Greece
mter@imis.athena-innovation.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, paired with bit shifting, which can greatly decrease its size in memory with potential performance gains. Finally, we propose an effective and efficient bulk loading algorithm which combines bottom-up construction of leaves with top-down FOR-compressed non-leaf nodes. 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.

CCS CONCEPTS

• **Information systems** → **Query operators**; **Main memory engines**; **Data access methods**.

KEYWORDS

main memory indices, access methods, data parallelism

1 INTRODUCTION

The B$^+$-tree has been the dominant indexing method for DBMSs, due to its low and guaranteed cost for query processing and updates and due to its support of range queries (in addition to equality searches, which are also well-supported by hash indices). It has been designed for disk-based storage, 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 [16, 18, 28, 39, 40, 46, 48, 52, 63, 64, 69, 79], as well as alternative access methods for in-memory data [9, 15, 17, 43, 50, 58, 61, 80, 82] have been proposed. The optimization objective in all these methods is minimizing the computational cost and cache misses during search. More recently, learned indices [24, 27, 30, 42, 45, 54, 77, 78, 83, 84], 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 fast. 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 the cost of 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. This way, (i) branchless, data-parallel SIMD search can still be applied, and (ii) key insertions and deletions do not require shifting of keys within each node. To our knowledge, our implementation and use of 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 [33].

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. The computational cost of search and update operations in B^S-tree is $O(\log_f n)$, where f is the capacity of the nodes, assuming that f is selected such that each node can be processed by a (small) constant number of SIMD instructions.

Novelty and contributions. There already exist several SIMD-based implementations of B-trees and k-ary search [33, 40, 46, 67, 69, 79]. In addition, several indices (especially learned ones [24]) use gaps to facilitate fast updates. Finally, key compression in B-trees has also been studied in previous work [16]. 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. All these thanks to (i) our simple but efficient data-parallel implementation of branching; (ii) its integration with a novel implementation of gaps using duplicate keys that does not affect correctness and performance; and (iii) our compression scheme that allows for direct application of operations on compressed nodes. We extensively compare our B^S-tree implementation with open-source 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 prevails in different workloads of reads and updates, typically achieving 1.5x-2x higher throughput than the best competitor from previous work.

Outline Section 2 presents related work. The B^S-tree is described in Section 3 and its updates and construction in Section 4. Section 5 describes B^S-tree compression. Implementation details and concurrency control are discussed in Sec. 6 and 7, respectively. Sec. 8 includes our experimental evaluation and we conclude in Sec. 9.

2 RELATED WORK

2.1 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, concurrency control, etc. [26, 32]. It was firstly introduced as B-tree [13, 23], a multiway extension of self-organizing red black trees [12, 35], where each key appears in exactly one node. To support range queries efficiently, the B-tree evolved to a B^+-tree, where all keys appear sorted in the (linked) leaf nodes and some keys are replicated in the non-leaf nodes acting as domain separators.

As memory sizes grow, the interest has shifted to in-memory access methods [81]. A set of rebalancing operations leading to significantly more efficient updates for red black trees was proposed in [18]. One of the first in-memory indices was the T-tree [48], which combines the intrinsics of binary search trees with the storage characteristics of B-trees. Rao and Ross [63] 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 [64] 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. [20] highlighted how prefetching can significantly improve the performance of index structures by reducing memory access latency. The same authors later proposed a fractal prefetching technique that bulk-reads B+-tree nodes in a hierarchical manner, minimizing both cache and disk accesses [21]. Hankins et al. [36] proved that the optimal index performance in a CSB+tree, can be achieved by balancing the cache misses, instruction count and TLB misses; they include an extended analysis of how the size of the node affects performance. PkT-trees and pkB-trees [16] are in-memory variants of the T-tree and the B-tree, respectively, that use partial-keys (fixed-size parts of keys), which reduce cache misses and improve search performance. Zhou and Ross [86] 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 [34] provided a survey of all the available techniques that can improve the perfomance of B^+-tree by exploting CPU caches.

2.2 (Data) parallelism in B-trees

Modern CPUs, where multiple comparisons can be performed by a single SIMD instruction and the evolution of GPUs opened new perspectives for in-memory index structures. In an early work, Zhou and Ross [85] explored how SIMD instructions can be used to optimize key database operations, like scans, joins, and filtering. The inherent parallelism of SIMD and the avoidance of branch misprediction can greatly improve the performance of an index. Schlegel et al. [67] present two k-ary search algorithms (find which partition of sorted data out of k contains a search key), one for sorted arrays and one using linearized k-ary search trees, using SIMD instructions. FAST [40], designed for modern CPUs and GPUs, optimizes k-ary tree search by leveraging architecture-specific features like cache locality, SIMD parallelism on CPUs, and massive parallelism on GPUs. [28] introduced a "braided" B^+-tree structure optimized for parallel searches on GPUs, enabling lock-free traversal using additional pointers. By leveraging CUDA for parallelism and optimizing memory access, the approach significantly improves search performance over traditional CPU-based methods, especially for large datasets. Kaczmarski [39] proposed a bottom-up B^+-tree construction and maintenance technique using CPU and GPU for bulk-loading and updates. Bw-Tree [52] is a highly scalable and latch-free B^+-tree variant optimized for modern hardware platforms, including multi-core processors and flash storage. Bw-Tree uses a mapping table for indirection and delta updates for efficient modifications. It also does not use locks, so it can achieve high throughput in workloads with concurrent operations. Hybrid B^+-tree [69] leverages both CPU and GPU resources to optimize in-memory indexing. It dynamically balances the workload between the CPU and GPU and exploits high GPU parallelization. Yan et al. [79] 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. They also provided two optimizations: first, they partially sort queries to enable coalesced memory access, and second, they group queries to decrease unnecessary comparisons within a warp, thereby reducing warp execution time. Kwon et al. [46] recently proposed DB+-tree, a B^+-tree with partial keys, that utilizes SIMD and other sequential instructions for fast branching. PALM [68] is a parallel latch-free variant of the B^+-tree, that is optimized for multi-core processors, enabling concurrent search and update operations. Several other papers explore the implementation of B-trees on newer hardware platforms, including flash memory, [7, 11, 37, 38, 55, 62, 72, 76], Non-Volatile Memory [22, 56] and Hardware Transactional Memory [70].

2.3 Other in-memory access methods

Besides B-trees, other data structures have also been used for in-memory indexing, especially trie-based ones. The trie [29, 44, 61] was originally used for storing and searching strings, aiming at path compression. HAT-trie [8, 9] is an in-memory, cache-conscious data structure that combines aspects of both tries and hashing to optimize performance, particularly in terms of memory access patterns. Aktipis and Zobel [10] proposed new update techniques for disk-based tries. The generalized prefix tree (trie) [17] is an in-memory index structure for arbitrary data types, which uses a variable prefix length. This method attempts to overcome the common

problems of tries, such as large trie height and memory requirements. Kissinger at al. [43] proposed KISS-TREE, a latch-free in-memory index, based on the generalized prefix tree [17], which uses memory management functionalities (like MMU) provided by the operating system and compression mechanisms to minimize the number of memory accesses. The techniques that they introduced to achieve that are direct addressing, on-demand allocation, compact pointers, and compression. Masstree [58] is a persistent data structure that combines aspects of B^+-tree and trie. It keeps all the data in memory and shares them with all cores to preserve load balance, maintains high concurrency using optimistic concurrency control, and can support keys with shared prefixes efficiently.

Leis et al. [50] proposed a fast and space-efficient in-memory trie called ART. ART dynamically adjusts its node sizes (4, 16, 48, or 256 bytes), providing a compact and cache-efficient representation. It uses lazy expansion (inner nodes are only created if they are required to distinguish at least two leaf nodes) and path compression (removes all inner nodes that have only a single child) to improve space utilization and search performance. Leis et al. proposed two synchronization protocols for ART in [51], which have good scalability despite relying on locks: optimistic lock coupling and the read-optimized write exclusion (ROWEX) protocol. Height Optimized Trie (HOT) [15] is an in-memory trie-based index that reduces tree height through path compression and node merging. It combines several nodes from a binary Patricia trie into compound nodes with a maximum fanout, to reduce the height of the structure. Additionally, it uses partial keys for each original key, determined by the bit divisions at each node, to conserve memory. Each node's layout is designed for efficiency, ensuring compactness and enabling fast searches with the use of SIMD instructions. Zhang et al. [80] presented a hybrid index, that uses two different data structures, aiming to achieve memory efficiency and high-performance. Their key idea is that certain data items are accessed more often than others. The first structure is dynamic and provides fast insertions and accesses and the second structure is more compact and read-optimized to serve reads for colder data. SuRF (Succinct Range Filter) [82] uses ideas behind the bloom filter to support range queries efficiently. It leverages succinct tries to provide a space-efficient solution for range query filtering.

2.4 Learned Indexing

The advent of fast and accurate machine learning techniques inspired the design of a new type of index structure, called *learned index* [45]. 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. Local search is used to identify the true position of a key if the prediction of RMI has an error. Galakatos et al [30] propoosed FITing tree, a data-aware index structure, based on RMI. FITing tree maintains a hierarchical structure similar to that of B^+-tree, but instead of rigidly dividing data into blocks, it uses the learned interpolation function to predict a key's location within each block. FITing uses linear interpolation models that learn a piecewise linear approximation of the cumulative distribution of the keys, which is more accurate than alternative, rule-based methods. PGM-index [27] is a fully-dynamic, compressed learned index that uses a piecewise geometric model to efficiently index and query large datasets, with provable worst-case bounds on query time and space usage. The RadixSpline (RS) [42] learned index can be constructed in a single traversal of sorted data.

ALEX [24] is a learned index structure, based on RMI, designed to efficiently handle updates. It retains a hierarchical structure, like a B^+-tree, where inner nodes use linear regression models. ALEX utilizes a *gapped array* layout that gracefully distributes extra space between elements based on the model's predictions, enabling faster insertions and lookups. CARMI [83] incorporates data partitioning into the construction of RMI and allows for data updates. NFL [78] is a two-stage Normalizing-Flow-Learned index framework that, instead of directly segmenting the CDF curve, initially utilizes the Numerical Normalizing Flow to convert the original keys into nearly uniformly distributed keys, resulting in a CDF curve that is approximately linear. Subsequently, the transformed keys effectively approximate the transformed CDF. LIPP [77] is an updatable learned index that also uses gaps to facilitate updates. One of the major challenges of learned indices is the approximation error when predicting the position of a key. LIPP proposed methods to improve the precision of key predictions, reducing the need for local search after model predictions, leading to faster lookups. The authors introduce a new metric for determining the layout of tree nodes and propose a dynamic adjustment strategy to keep the tree height tightly constrained. DILI [54], is a yet another distribution-driven learned tree for main memory that uses linear regression models for each node to map keys to corresponding children or records. Learning is done during DILI's bulk loading construction, which is performed in two phases. In the first phase, a balanced bottom-up tree is created using linear regression models that account for both global and local key distributions. In the second phase, DILI is constructed in a top-down approach based on previous constructed bottom-up tree, customizing the fanout of internal nodes based on the local key distributions. Zhang et al. [84] proposed Hyper, an in-memory and multi-threaded learned index, that uses hybrid construction and runtime adjustment techniques to improve query performance and memory footprint. Their core idea, is that leaf nodes need more memory space than inner nodes, but non-leaf nodes are very important, because they should be more accurate. So, they create a hybrid construction, where the leaf nodes are created bottom-up to reduce the overall memory overhead and the inner nodes are conctructed top-down, allowing more memory consumption to achieve better and more accurate predictions. Other learned indices have also been proposed to support concurrency [31, 53, 71, 73], for bloom filters [60], and for persistent memory [57]. [41, 59] and [75] provide comprehensive evaluations on updatable learned indices and traditional indices including many important findings, based on tests on several real-world datasets.

3 THE B^S-TREE

We propose B^S-tree, an in-memory implementation of the B^+-tree, which supports fast searches and updates by exploiting data parallelism (i.e., SIMD instructions). We first present the data structure in Section 3.1. Then, Section 3.2 describes the implementation of

the successor operator applied to each node for branching and key location during search and updates. Finally, Section 3.3 presents the algorithms for equality and range search.

3.1 The structure

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 (if not, rid is replaced by a pointer to a block that keeps the rid's of all records having the corresponding search key). The storage of the rid's is decoupled from the storage of the keys, i.e., they are stored in two different 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 node (memory) pointers, to facilitate fast search, as we explain later. Each leaf node hosts a node pointer to the next leaf; i.e., the leaves of the tree are chained based on the total order of the keys. Chaining is used for the efficient support of range queries as the results of such queries should be in consecutive tree leaves. For the B^S-tree nodes we use a value of N that facilitates fast and parallel search, as we will explain later. For the efficient handling of updates, we allow gaps (i.e., unused slots) in nodes, similarly to previous work [24, 54, 77].

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 ∞ value, which is a MAXKEY value, larger than the maximum possible value in the key domain.

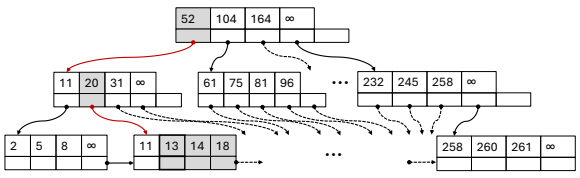

Figure 1: Example of B^S-tree

3.2 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 incur significant CPU cost due to branch mispredictions.

We use an efficient implementation of the *successor* operator applied to each node along the search path, which does not involve search decisions. We denote by $succ_>$ the operator that finds the smallest key position which is strictly greater that 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. This means that if we are looking for key 124, we should follow the node pointer at position 2 of the root. In general, $succ_>$ is the most frequently applied operation during search, as we use it on each non-leaf node along the path from the root to the (first) leaf node that includes the search result.

Our approach exploits data parallelism (i.e., SIMD instructions) and does not include if-statements or while statements with an uncertain number of loops. 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, $succ_{>k}(v)$ 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.

Snippet 1: Counting search

```
1 int succG(Node *v, int 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, int 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 }
```

The code snippets do not include if-statements and do not incur branch mispredictions. CAPACITY is the (fixed) capacity of the node, so the number of iterations of the for-loop is hardwired; all these favoring data parallelism.

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 [67, 69, 85].

The experiments of Figures 2, 3, and 4 illustrate the efficiency of data-parallel $succ_>$ compared to traditional ways for branching in multiway trees, on sorted arrays of 64-bit, 32-bit, and 16-bit unsigned integers, respectively. 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[1] of the operation:

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

Binary search and linear scan perform similarly, with binary search improving 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 optimizations at the assembly level, such as autovectorization, caused 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 compilation, custom vectorization offers significant advantages and achieves the theoretically optimal performance improvement. 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. $\log_2 16$). As a final note, for small array sizes or small key sizes Snippet 2 has about double throughput compared to the code produced by compiling Snippet 1.

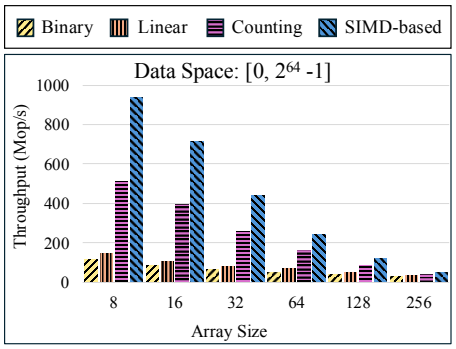

Figure 2: Successor search techniques uint64

3.3 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.

[1]See Section 8 for our experimental setup.

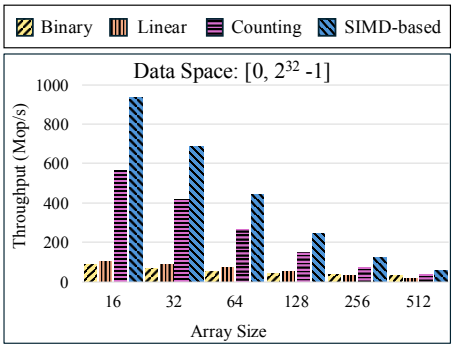

Figure 3: Successor search techniques uint32

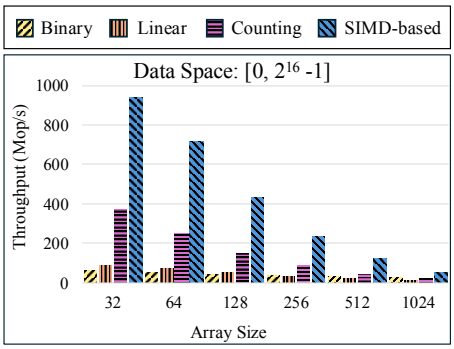

Figure 4: Successor search techniques uint16

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. To find the end position r_2 of the query results, starting from the current leaf v, we search for the leaf which includes the first key greater than k_2, by performing one $succ_{>k_2}$ operation per leaf. Hence, range searches require one $succ_>$ operation per node along the search path in search for k_1 plus one $succ_>$ operation for each leaf that includes range query results. For large query ranges whose results appear in numerous leaves, we apply an alternative implementation of range queries, where one equality search is used to locate r_1 and another equality search is then applied to locate r_2. This is expected to be faster than Algorithm 4 if the height of the tree is smaller compared to the number of leaves that include the query results.

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. A range search for keys in [13, 17] first locates 13 and then finds the upper bound 18 in the same leaf after applying $succ_{>17}$.

ALGORITHM 3: Equality Search

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

```
while n 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 $k_1 \leq x \leq k_2$

```
while n is non leaf do
    v ← node pointed by entry at position v[succ_{>k1}(v)]
r1 ← succ_{≥k1}(v)];
while (r2 ← v.keys[succ_{>k2}(v)]) == N do
    v ← nextleaf(v)             ▷ next to v leaf
    if v == NULL then
        break                   ▷ last leaf reached
return record-ids of keys from position r1 to position r2 (excl.)
```

4 GAPS AND UPDATES

The main novelty of B^S-tree is the implementation of gaps in nodes using duplicated keys, which facilitates efficient updates. Specifically, while SIMD-based k-way search on *packed* arrays has also been suggested before [67, 69, 85], to our knowledge it has never been applied on arrays with gaps, e.g., on B$^+$-tree nodes which are not full. As discussed in Sec. 3.1, 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 from 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 is the same as the *first subsequent non-gap key*. By managing auxiliary information at each node (i.e., number of used slots, bitmap indicating used slots), we can efficiently track unused slots (see Figure 5 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.

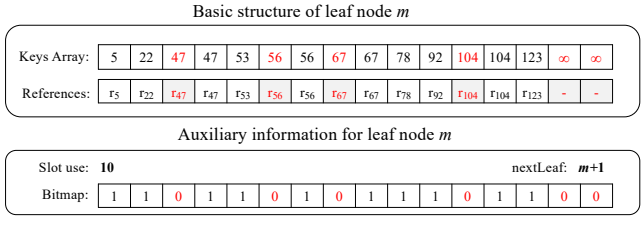

Figure 5: B^S-tree leaf node structure

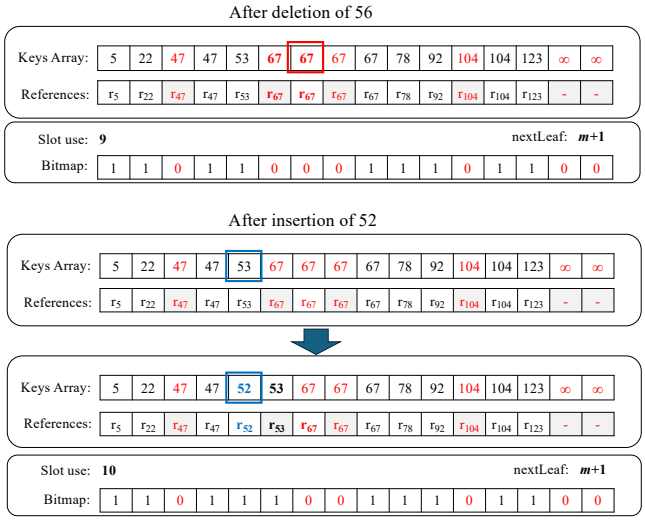

Figure 6: Updates to B^S-tree leaf node

4.1 Deletions

To delete a key, we first locate its position i in a leaf node, using the equality search algorithm (discussed in Section 3.3). To conduct the deletion, 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 $v \rightarrow keys[i]$=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 5, 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 6 (top) shows the leaf node of Figure 5 after deleting key 56. Algorithm 5 is a pseudocode for deletions to the B^S-tree.

As in previous work [64, 77], we do not handle underflows and allow nodes with fewer than 50% occupied slots. The reason is that insertions are anticipated to be more frequent than deletions in workloads, so node merges or key redistributions are not expected to pay-off in the long run. If a node becomes empty after a deletion, the node is removed and the corresponding separator entry at its parent is also 'deleted' by copying the next key into it.

4.2 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 $v \rightarrow keys[i] = v \rightarrow keys[i+1]$, then we are sure that position i is free, so we can place k there and finish. For example, assume that we want to insert key 55 to the leaf node of Figure 5. We apply a $succ_{\geq k}$ operation to the leaf, which will give us position 5. Since

ALGORITHM 5: Deletion in B^S-tree

```
Input  : key k, B^S-tree root node v
find leaf v and position r by running lines 1-3 of Alg 3
if v.keys[r] ≠ k then
    return FAIL
bitmap ← v.bitmap
if r == N − 1 then                                   ▷ last key in node
    bitmap ← bitmap ⊕ 0x0001
    replicasOfKey ← _tzcnt(bitmap)
    for i ← 0 to replicasOfKey − 1 do                ▷ propagate backwards
        v.keys[r − i] ← MAXKEY
else                                                 ▷ r is not the last position in the leaf
    bitmap ← bitmap ⊕ (0x8000 ≪ r)
    replicasOfKey ← _lzcnt(bitmap)
    nextValidKey ← v.keys[r + replicasOfKey + 1]
    for i ← 0 to replicasOfKey do                    ▷ propagate backwards
        v.keys[r + i] ← nextValidKey
    bitmap ← bitmap ⊕ (0x8000 ≫ (r + replicasOfKey))
v.slotuse ← v.slotuse − 1
v.bitmap ← bitmap
return SUCCESS
```

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               ▷ r is an empty slot
        v.keys[r] ← k
        bitmap ← bitmap ⊕ (0x8000 ≫ r)
    else
        keysForShift ← _lzcnt(bitmap ≪ r)
        if keysForShift < N then                     ▷ empty slot 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
```

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 6 (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 space for the new key 52. Algorithm 6 describes the insertion procedure to a B^S-tree.

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. Proactive gapping is described in the next subsection.

4.3 Tree building

We now describe the algorithm for building a B^S-tree from a set of keys (bulk loading). Like typical B$^+$-tree construction 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 typically set $\alpha = 0.75$. 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.[2] 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.

5 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 [16] 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 incurs minimal search overhead.

For each node v, we store in the node's auxiliary information (see Figure 5) 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. Another benefit of compression is that a potentially radical decrease in the number

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

of (leaf) nodes may reduce the number of tree levels, rendering the tree faster.

The variability in the capacity of nodes necessitates some modifications to the auxiliary information at each node. Previously, 8 bytes were sufficient to store all relevant details about slot use, the bitmap of a node, and the next leaf reference. While the number of bits for slot use and the next leaf reference can still be represented as before, this is not the case for the bitmap. Since the bitmap corresponds to the total number of entries in a node, we have opted to allocate 64 bits for this purpose, which is sufficient for any type of leaf. We now discuss the impact of this scheme in the tree operations.

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 metadata. 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 We can directly use the B^S-tree insertion algorithm to insert a new key k, by first running 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. Keys can be stored exactly only as first keys in new nodes that are constructed after a split. The new nodes after a node v is split 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 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.

6 IMPLEMENTATION DETAILS

This section presents some implementation details of the B^S-tree that have a significant impact in its performance in practice.

Node size and structure. As in previous work [15, 24, 46, 50, 52, 67, 69, 77, 79, 80, 82, 84], 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 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, so a compressed key array may have 32 32-bit entries (key differences) or 64 16-bit entries. For gap management, we use a 32-bit variable which keeps the slot use and bitmap of each node. For the leaves, we have an additional 32-bit field for the next leaf address. As discussed in Section 4.3, gaps are added to the tree nodes proactively at construction time, to allow efficient ingestion of insertions. Since we anticipate insertions to affect mainly the leaves, in practice, we use a much smaller percentage of gaps at inner nodes (one), to keep the height of the tree small.

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). 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 not values (i.e., 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 nodes are stored in a contiguous array, aligned to Transparent Huge Pages (2 MB) for efficiency. The leaf nodes are also stored in a contiguous array, but their alignment depends on the array size. If the array is smaller than 3 GB, we align it using huge pages. Otherwise, it is aligned per cache line (64 bytes). 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 inner 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 5) 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. 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

[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.

for all segments, we take the average number of leading zeroes. 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^+-tree implementation has uncompressed inner nodes and compressed leaves.

7 CONCURRENCY CONTROL

A wide range of concurrency control techniques has been proposed for the B^+-tree and other indices, including lock coupling [14, 32], right-sibling pointers [47], fine-grained locking with lock coupling and logical removals [19], Bw-tree's lock-free mechanism [52, 74] and Read-Optimized Write Exclusion (ROWEX) [51]. For B-trees and B+ trees, Leis et al. [49, 51] 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 applies the changes and commits them 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 of the tree.

Our current implementation of B^S-tree employs the OLC mechanism [49, 51] with a slight modification 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 approach efficiently reorganizes node contents, ensuring that insertions triggering splits at multiple tree levels are managed without requiring a global lock on the involved. The use of sparse nodes in B^S-tree allows for a more flexible procedure. Specifically, when a split occurs at a leaf node, the restart process is only necessary if the separator must be stored in the first node of the couple, which has no available free slot. This relaxation reduces unnecessary retraversals while preserving the tree's initial balance as much as possible.

8 EXPERIMENTS

In this section, we experimentally compare B^S-tree and its compressed version to alternative main-memory indices (learned and non-learned). As in previous work [15, 50], we have built and compared indices for key data only; record ids or references are not stored in each index, but the objective of each method 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 threads, running at 3.60 GHz, 128 GB of RAM, and AVX 512 support. The operating system used was Ubuntu 22.04. We extended the codebase of GRE [75] to include B^S-tree.

8.1 Setup

Datasets. We ran our tests on standard benchmarking real datasets used in previous work [41, 59, 75]; each on consists of unsigned 64-bit integer keys (potentially reduced to 63-bit, as HOT cannot handle 64-bit keys). In Amazon BOOKS [41, 59], each key represents the popularity of a specific book. In FB [41, 59, 66], each key is a Facebook user-id. OSM [41, 59] contains unique integer-encoded locations from OpenStreetMap. GENOME [65, 75] includes loci pairs from human chromosomes. PLANET [2, 75], a planet-wide collection of integer-encoded geographic locations compiled by OpenStreetMap. We preprocessed the datasets to eliminate any duplicates. According to [75, 84], 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 [59], it would be trivial for a learned index to model such distributions. B^S-tree (and its competitors) can also be used for floats, after discretization, and for strings, after being dictionary encoded, as discussed in [25].

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 single-threaded code was publicly available by the authors (we thank them!). For the parallel versions of the competitors, we use the code from [75]. The selection was done based on the superiority of these indices compared to alternative methods.

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), and each inner node holds 16 keys and 16 pointers ($16 \times 8 + 16 \times 8 = 256$ 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 gaps at its construction to support fast insertions). STX supports $2^{64} - 1$ values in keys.

The implementations of HOT [3, 15] and ART [6, 50] that we use store only keys and 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 include a built-in range query implementation, so we created one based on B^S-tree's logic. The ART code also lacks range query support, but we were unable to implement it due to its structure. HOT cannot handle keys greater than $2^{63} - 1$, so we removed values exceeding this limit from certain datasets.

Learned Indices. **ALEX** [4, 24] and **LIPP** [5, 77] are all learned indices for key-values. To use them, for each dataset we used as value of each key the key itself, during construction and insertions (writes). They all support bulk-loading. ALEX and LIPP can handle 64-bit keys, i.e., (unsigned) integers up to $2^{64} - 1$.

Workloads. We used several different workloads used to measure throughput. First, we randomly selected 150 million entries from

each dataset for the construction phase. where we sorted the data and applied bulk-loading (except for HOT and ART, which, however, 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:

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

8.2 Construction Time and Memory Footprint

In this section, we compare all tested methods with respect to their construction cost and memory footprint. From each dataset, we used 150 million keys to construct each index. Since all indices require the data to be sorted (or benefit from sorting), we exclude sorting from the construction cost. Table 1 presents the construction times, while Table 2 shows the memory footprints. The construction time of our B^S-tree 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 6). To calculate the memory 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 the models (and inner structure) that they use.

As expected, non-learned indices (except from the CB^S-tree) have a stable construction time and memory footprint. On the other hand, the construction time of learned indices can vary significantly, as different data distributions can greatly affect them. As expected, the construction cost of the Sparse B^+-tree is higher compared to that of the B^+-tree and the two versions of B^+-tree there is also a similar-scale difference in their memory footprints. As we will show later, Sparse B^+-tree has a performance advantage over B^+-tree for any workload that includes write operations. Our B^S-tree (and its compressed CB^S-tree version) is faster to build compared to B^+-tree and Sparse B^+-tree mainly due to our better memory management (static pre-allocation vs. dynamic allocation) and because we use offset addressing instead of memory pointers (see Section 6).

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; in this case, most leaf nodes store 64-bit differences and also need to explicitly store the 64-bit key of the first key, which renders the size of the index even larger than that of B^S-tree. Note that for BOOKS and OSM, our decision mechanism (see Section 6) chooses to construct a B^S-tree while for FB, GENOME, and PLANET it decides to construct a CB^S-tree.

HOT, requires significantly more time to build compared to our methods and B^+-tree. The main issue with HOT is that it is a top-down trie and lacks a bulk-loading mechanism, requiring keys to be inserted one at a time. 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 faces the same construction time issue as HOT but is significantly faster due to its simpler insertion process. ART has low memory consumption when the key space is dense enough, achieving a similar memory footprint as our uncompressed B^S-tree.

Regarding learned indices, ALEX and LIPP also have high construction time compared to the versions of B^+-tree and B^S-tree, due to the overhead of training models that predict key positions. Note that all versions of B^+-tree and B^S-tree have smaller memory footprints compared to the learned indices. Among learned indices, LIPP is the fastest one to construct, because it does not readjust the models when conflicts occur (two or more keys are mapped to the same position). To address conflicts, LIPP creates new nodes, speeding up bulk-loading. ALEX has higher construction time because it frequently adjusts its learned models to preserve key prediction accuracy. This results in computationally expensive node splits and reorganizations. LIPP's downside is its large memory footprint, caused by creating new nodes during conflicts. Node reorganization in ALEX requires less memory. The significant memory overhead of LIPP has also been coined in other experimental studies [75, 84].

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. [75], which conclude that

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

Table 1: 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

Table 2: 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	7.23	7.36	7.42	7.38	7.85
ALEX	2.73	2.77	2.77	2.73	2.73
LIPP	13.51	14.69	10.89	11.66	11.62

memory efficiency is not a distinct advantage of updatable learned indices.

8.3 Single-Threaded Throughput

Next, we evaluate the throughput of all methods (single-threaded) on the five workloads described in Section 8.1. Figure 7 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 with B^S-tree. The excellent performance of ALEX on BOOKS is due to its smooth distribution which is easy for ALEX to learn. In general, learned indices are known to perform well when applied on easy-to-learn CDFs. 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. Among the compressed datasets, GENOME is the most challenging because most keys are 32-bit differences. LIPP outperforms ALEX on datasets other than BOOKS because their models are precise (i.e., ALEX applies exponential search at the last mile, when its prediction is imprecise).

For Workload B (write-only), as Figure 8 shows, B^S-tree outperforms all methods, while CB^S-tree loses to ALEX only on BOOKS. ART achieves good performance in this workload across all datasets, except for BOOKS, because of its lazy expansion and path compression. Compared to LIPP, ALEX performs worse on the other four datasets because It frequently needs to readjust its learned models to sustain key prediction accuracy, resulting in computationally expensive node splits and reorganizations. LIPP is more robust than other learned indices 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 compared to B^S-tree, except on the GENOME dataset, which is the most challenging from the compressed datasets.

The results on Workload C (read-write) are presented in Figure 9. In this workload, our algorithms outperformed all competitors across all datasets. In general, 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 10. The results are similar compared to Workload A (read-only) since Workload D is read-heavy. Range queries retrieve 153 keys on average. ALEX has a better performance compared to other learned indices for range queries, as the structures of LIPP is not optimized for range scans. ALEX can have large nodes with more sequential keys, allowing it to avoid jumping to sibling nodes. HOT is also not optimized for range scans, therefore its low compared to other workloads. On the compressible datasets (FB, GENOME, PLANET), CB^S-tree outperforms B^S-tree due to the smallest number of memory accesses it requires to obtain the results of range queries, as it reads numerous consecutive compressed leaf nodes.

Figure 11 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 B^S-tree and all other methods.

8.4 Multi-Threaded Workloads

Next, we present multi-threaded experiments to evaluate the throughput of all methods. ART and B^+-tree use OLC concurrency control, HOT uses ROWEX, and ALEX and LIPP use Optimistic Locking. We use Workload A, Workload B, and Workload C, which are the most representative. We present the B^S-tree parallel implementation as a competitor to previous work.

Figure 12, presents the throughput (millions of operations per second) of all methods for Workload A (read-only). We observe that the experiments exhibit the same behavior as the single-threaded ones. Our B^S-tree outperforms all competitors, with ALEX being the only one competitive on the BOOKS dataset.

The results on Workload B (write) are presented in Figure 13. In this workload, ART achieves the best performance, while our B^S-tree remains competitive in some cases. ART's strong performance is largely due to OLC, which is well-suited to its structure (see Section 7).

Figure 14 shows the throughput of Workload C (read-write). Our B^S-tree outperforms all competitors across all datasets.

Overall, we demonstrate that our structures can scale with different numbers of cores and achieve the best performance in most cases.

8.5 Impact of B^S-tree design

In the final set of experiments, we assess the impact of the design choices in B^S-tree. Figure 15 shows the impact of our data-parallel implementation using gaps that copy the key values from the next used slot, on the cost of writes. We compare our B^S-tree implementation to an implementation of B^S-tree, which supports gapped arrays, as in ALEX [24], where SIMD is not applicable, because gaps are implemented simply with the help of a bitmap, as in [24]. In such an implementation, branching in each node to find the slot where to insert a new key is done by linear scan, using the bitmap to ignore unused slots. The figure shows that our SIMD implementation with gaps reduces the cost of insertions (and node searches, which are part of the insertion procedure) by 30%-35%.

We also analyze the effectiveness of the optimizations used in B^S-tree, with the most important ones being SIMD instructions and Transparent Huge Pages. We implemented four versions of our structure: one without any optimizations (NHP + Counting), one with only huge pages enabled (HP + Counting), another with only SIMD instructions enabled (NHP + SIMD), and finally, a fully optimized version that combines both huge pages and SIMD instructions (HP + SIMD). Figure 16 shows the throughput of the four

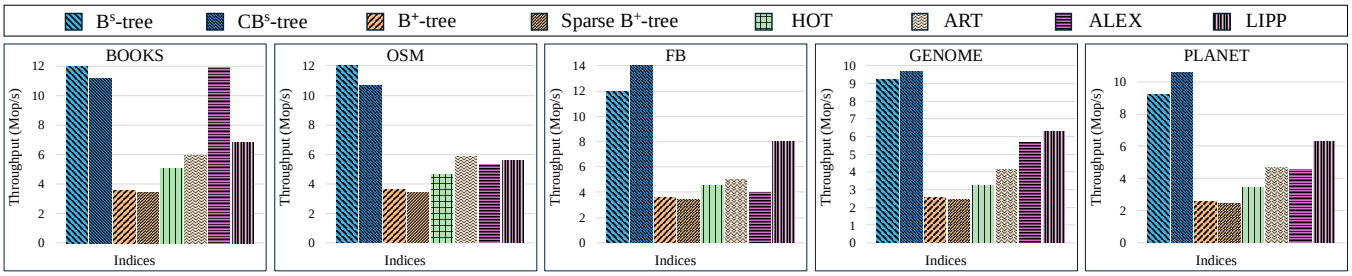

Figure 7: Workload A : Read Only (100%)

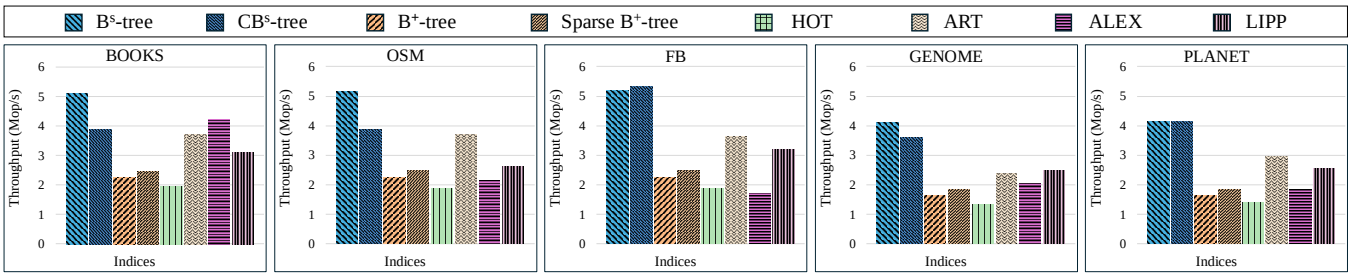

Figure 8: Workload B : Write Only (100%)

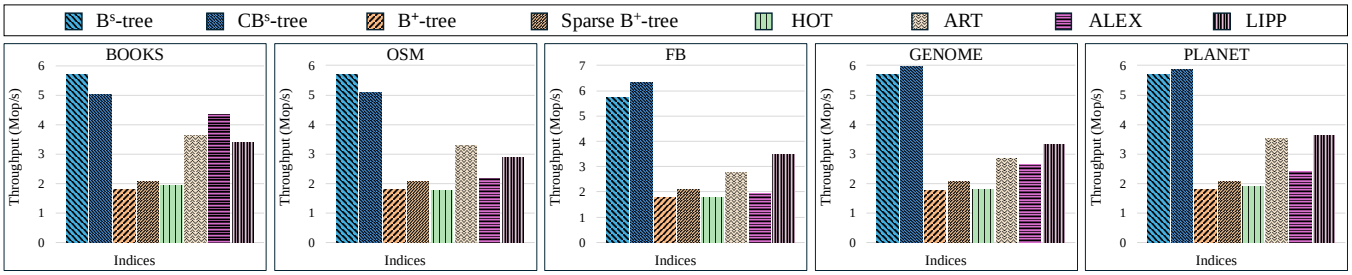

Figure 9: Workload C : Read (50%) - Write (50%)

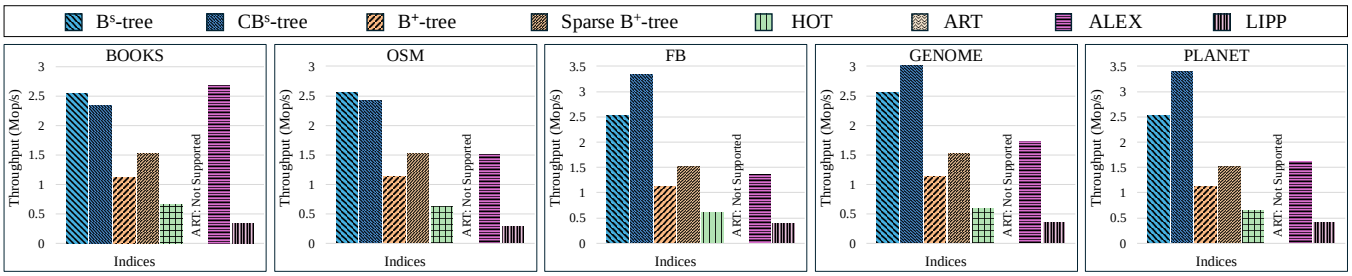

Figure 10: Workload D : Range (95%) - Write (5%)

versions of B^S-tree for Workload A (reads). We observe that both optimizations (HP and SIMD) boost B^S-tree, which is designed to take advantage of them. Additionally, we find that even the basic version of B^S-tree (NHP + Counting) remains competitive with all other methods shown in Figure 7.

8.6 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. We also show, that our algorithm scales efficiently with the number of cores. Note that our decision mechanism, which imposes a small overhead in the construction (up to 10% of the construction cost) decides automatically which of B^S-tree or CB^S-tree to build for a given dataset. The decision is B^S-tree for BOOKS and OSM and CB^S-tree for FB, GENOME, and PLANET. Although CB^S-tree is not superior to B^S-tree on all workloads over FB, GENOME, and PLANET, their difference of the two is not high on these datasets, while CB^S-tree is much faster to build and has a much lower memory footprint.

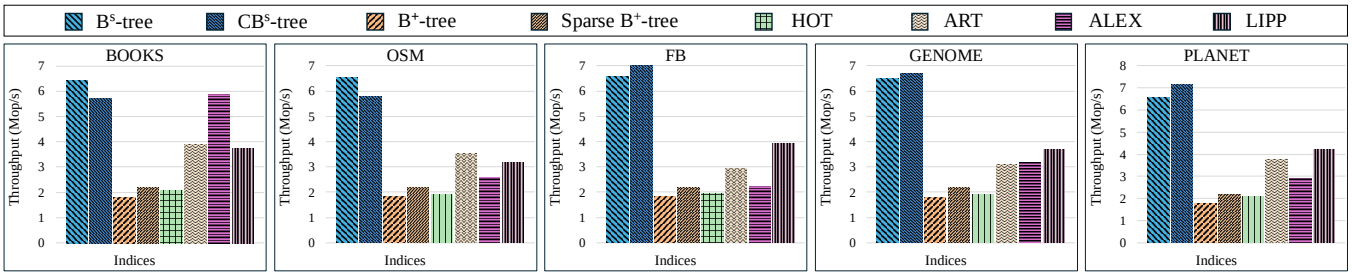

Figure 11: Workload E : Read (60%) - Write (35%) - Deletions (5%)

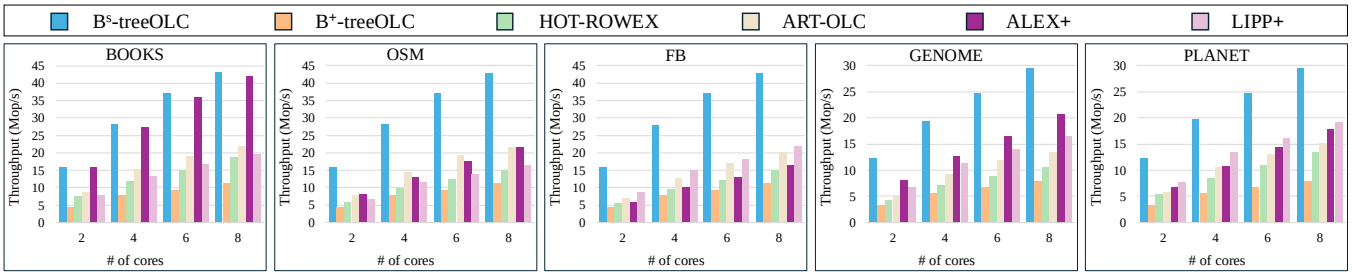

Figure 12: Workload A : Read Only (100%) - OLC

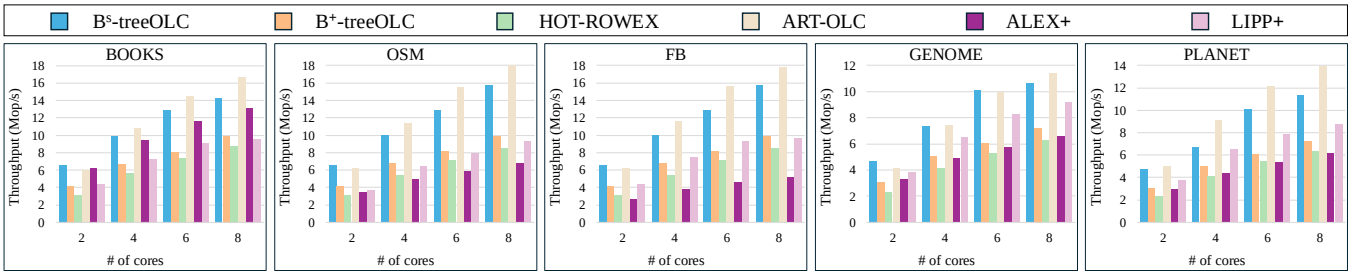

Figure 13: Workload B : Write Only (100%) - OLC

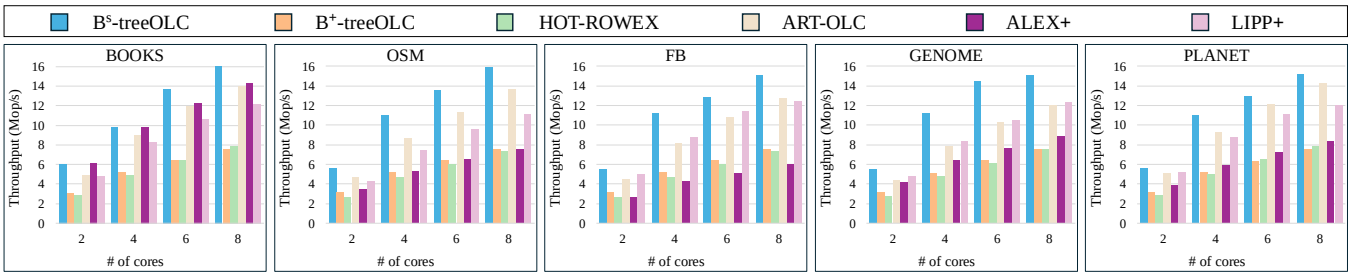

Figure 14: Workload C : Read (50%) - Write (50%) - OLC

Regarding updatable learned indices, our study shows that they are typically outperformed by data-parallel non-learned indices optimized for main-memory, such as our B^S-tree. From the tested methods, LIPP appears to be superior and more robust than ALEX, except for range workloads, where ALEX performs best. In addition, their construction cost is high and they have large memory footprint (except for ALEX) compared to non-learned indices. Even on the BOOKS, which has a easy-to-learn key distribution, our B^S-tree demonstrates competitive performance in searches and range scans compared to ALEX and outperforms it in all other workloads.

9 CONCLUSIONS

We proposed B^S-tree, an efficient main-memory implementation of the B$^+$-tree that uses a simple and intuitive data-parallel implementation for branching at each level during search and update operations. We propose a novel way for implementing gaps (unused slots) at nodes by copying into them the next used key, which does not affect the functionality and efficiency of our branching operator. Finally, we choose a compression mechanism for B^S-tree

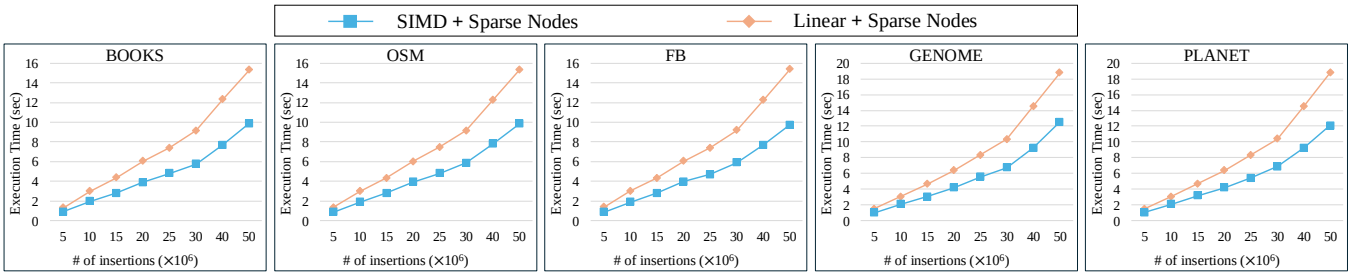

Figure 15: Impact of data parallelism to the Insertion cost of B^S-tree

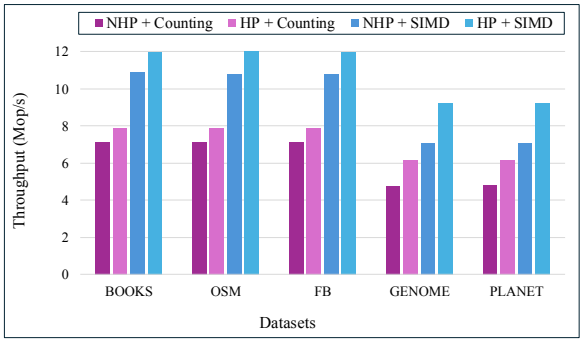

Figure 16: Effect of B^S-tree design (Workload A)

nodes that allows them to have variable capacity based on the allowed room for compression. 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. Inspired by [69], 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 additional data types, such as strings (one solution is to use Binary, ASCII or Base64 encoding [24]).

REFERENCES

[1] 2013. *STX B+Tree code.* https://github.com/bingmann/stx-btree/tree/master

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

[3] 2018. *HOT code.* https://github.com/speedskater/hot

[4] 2020. *ALEX code.* https://github.com/microsoft/ALEX

[5] 2021. *LIPP code.* https://github.com/Jiacheng-WU/lipp

[6] 2022. *ART code.* https://github.com/pohchaichon/ARTSynchronized/tree/808372ded6b8c5a6d3a1741090510b79042f2aa7

[7] Devesh Agrawal, Deepak Ganesan, Ramesh K. Sitaraman, Yanlei Diao, and Shashi Singh. 2009. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. *Proc. VLDB Endow.* 2, 1 (2009), 361–372. https://doi.org/10.14778/1687627.1687669

[8] Nikolas Askitis and Ranjan Sinha. 2007. HAT-Trie: A Cache-Conscious Trie-Based Data Structure For Strings. In *Computer Science 2007. Proceedings of the Thirtieth Australasian Computer Science Conference (ACSC2007). Ballarat, Victoria, Australia, January 30 - February 2, 2007. Proceedings (CRPIT)*, Gillian Dobbie (Ed.), Vol. 62. Australian Computer Society, 97–105. http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV62Askitis.html

[9] Nikolas Askitis and Ranjan Sinha. 2010. Engineering scalable, cache and space efficient tries for strings. *VLDB J.* 19, 5 (2010), 633–660. https://doi.org/10.1007/S00778-010-0183-9

[10] Nikolas Askitis and Justin Zobel. 2009. B-tries for disk-based string management. *VLDB J.* 18, 1 (2009), 157–179. https://doi.org/10.1007/S00778-008-0094-1

[11] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-Tree: Approximate Tree Indexing. *Proc. VLDB Endow.* 7, 14 (2014), 1881–1892. https://doi.org/10.14778/2733085.2733094

[12] Rudolf Bayer. 1972. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica* 1 (1972), 290–306. https://doi.org/10.1007/BF00289509

[13] Rudolf Bayer and Edward M. McCreight. 1970. Organization and Maintenance of Large Ordered Indexes. In *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access.* ACM, 107–141.

[14] Rudolf Bayer and Mario Schkolnick. 1977. Concurrency of Operations on B-Trees. *Acta Informatica* 9 (1977), 1–21. https://doi.org/10.1007/BF00263762

[15] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 521–534. https://doi.org/10.1145/3183713.3196896

[16] Philip Bohannon, Peter McIlroy, and Rajeev Rastogi. 2001. Main-Memory Index Structures with Fixed-Size Partial Keys. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, Sharad Mehrotra and Timos K. Sellis (Eds.). ACM, 163–174.

[17] Matthias Böhm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. 2011. Efficient In-Memory Indexing with Generalized Prefix Trees. In *Datenbanksysteme für Business, Technologie und Web (BTW), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 2.-4.3.2011 in Kaiserslautern, Germany (LNI)*, Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, and Holger Schwarz (Eds.), Vol. P-180. GI, 227–246. https://dl.gi.de/handle/20.500.12116/19581

[18] Joan Boyar and Kim S. Larsen. 1992. Efficient Rebalancing of Chromatic Search Trees. In *Algorithm Theory - SWAT '92, Third Scandinavian Workshop on Algorithm Theory, Helsinki, Finland, July 8-10, 1992, Proceedings (Lecture Notes in Computer Science)*, Otto Nurmi and Esko Ukkonen (Eds.), Vol. 621. Springer, 151–164. https://doi.org/10.1007/3-540-55706-7_14

[19] Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, R. Govindarajan, David A. Padua, and Mary W. Hall (Eds.). ACM, 257–268. https://doi.org/10.1145/1693453.1693488

[20] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. 2001. Improving Index Performance through Prefetching. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, Sharad Mehrotra and Timos K. Sellis (Eds.). ACM, 235–246. https://doi.org/10.1145/375663.375688

[21] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. 2002. Fractal prefetching B±Trees: optimizing both cache and disk performance. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki (Eds.). ACM, 157–168. https://doi.org/10.1145/564691.564710

[22] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (2015), 786–797. https://doi.org/10.14778/2752939.2752947

[23] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137. https://doi.org/10.1145/356770.356776

[24] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 International Conference on Management of*

Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 969–984. https://doi.org/10.1145/3318464.3389711

[25] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.* 14, 2 (2020), 74–86.

[26] Ramez Elmasri and Shamkant B. Navathe. 1989. *Fundamentals of Database Systems.* Benjamin/Cummings.

[27] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* 13, 8 (2020), 1162–1175. https://doi.org/10.14778/3389133.3389135

[28] Jordan Fix, Andrew Wilkes, and Kevin Skadron. 2011. 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.

[29] Edward Fredkin. 1960. Trie memory. *Commun. ACM* 3, 9 (1960), 490–499. https://doi.org/10.1145/367390.367400

[30] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1189–1206. https://doi.org/10.1145/3299869.3319860

[31] Jiake Ge, Huanchen Zhang, Boyu Shi, Yuanhui Luo, Yunda Guo, Yunpeng Chai, Yuxing Chen, and Anqun Pan. 2023. SALI: A Scalable Adaptive Learned Index Framework based on Probability Models. *Proc. ACM Manag. Data* 1, 4 (2023), 258:1–258:25. https://doi.org/10.1145/3626752

[32] Goetz Graefe. 2011. Modern B-Tree Techniques. *Found. Trends Databases* 3, 4 (2011), 203–402.

[33] Goetz Graefe. 2024. More Modern B-Tree Techniques. *Found. Trends Databases* 13, 3 (2024), 169–249.

[34] Goetz Graefe and Per-Åke Larson. 2001. B-Tree Indexes and CPU Caches. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, Dimitrios Georgakopoulos and Alexander Buchmann (Eds.). IEEE Computer Society, 349–358. https://doi.org/10.1109/ICDE.2001.914847

[35] Leonidas J. Guibas and Robert Sedgewick. 1978. A Dichromatic Framework for Balanced Trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*. IEEE Computer Society, 8–21. https://doi.org/10.1109/SFCS.1978.3

[36] Richard A. Hankins and Jignesh M. Patel. 2003. Effect of node size on the performance of cache-conscious B^+-trees. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2003, June 9-14, 2003, San Diego, CA, USA*, Bill Cheng, Satish K. Tripathi, Jennifer Rexford, and William H. Sanders (Eds.). ACM, 283–294. https://doi.org/10.1145/781027.781063

[37] Peiquan Jin, Chengcheng Yang, Christian S. Jensen, Puyuan Yang, and Lihua Yue. 2016. Read/write-optimized tree indexing for solid-state drives. *VLDB J.* 25, 5 (2016), 695–717. https://doi.org/10.1007/S00778-015-0406-1

[38] Martin V. Jørgensen, René Bech Rasmussen, Simonas Saltenis, and Carsten Schjønning. 2011. FB-tree: a B^+-tree for flash-based SSDs. In *15th International Database Engineering and Applications Symposium (IDEAS 2011), September 21 - 27, 2011, Lisbon, Portugal*, Bipin C. Desai, Isabel F. Cruz, and Jorge Bernardino (Eds.). ACM, 34–42. https://doi.org/10.1145/2076623.2076629

[39] Krzysztof Kaczmarski. 2012. B^+-Tree Optimized for GPGPU. In *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 (Lecture Notes in Computer Science)*, Robert Meersman, Hervé Panetto, Tharam S. Dillon, Stefanie Rinderle-Ma, Peter Dadam, Xiaofang Zhou, Siani Pearson, Alois Ferscha, Sonia Bergamaschi, and Isabel F. Cruz (Eds.), Vol. 7566. Springer, 843–854. https://doi.org/10.1007/978-3-642-33615-7_27

[40] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, 339–350. https://doi.org/10.1145/1807167.1807206

[41] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems* (2019).

[42] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*, Rajesh Bordawekar, Oded Shmueli, Nesime Tatbul, and Tin Kam Ho (Eds.). ACM, 5:1–5:5. https://doi.org/10.1145/3401071.3401659

[43] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. 2012. *KISS-Tree*: smart latch-free in-memory indexing on modern architectures. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN 2012, Scottsdale, AZ, USA, May 21, 2012*, Shimin Chen and Stavros Harizopoulos (Eds.). ACM, 16–23. https://doi.org/10.1145/2236584.2236587

[44] Donald Ervin Knuth. 1998. *The art of computer programming, , Volume III, 2nd Edition.* Addison-Wesley. https://www.worldcat.org/oclc/312994415

[45] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 489–504. https://doi.org/10.1145/3183713.3196909

[46] Yongsik Kwon, Seonho Lee, Yehyun Nam, Joong Chae Na, Kunsoo Park, Sang K. Cha, and Bongki Moon. 2023. DB+-tree: A new variant of B+-tree for main-memory database systems. *Inf. Syst.* 119 (2023), 102287.

[47] Philip L. Lehman and S. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Trans. Database Syst.* 6, 4 (1981), 650–670. https://doi.org/10.1145/319628.319663

[48] Tobin J. Lehman and Michael J. Carey. 1986. A Study of Index Structures for Main Memory Database Management Systems. In *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi (Eds.). Morgan Kaufmann, 294–303. http://www.vldb.org/conf/1986/P294.PDF

[49] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019), 73–84. http://sites.computer.org/debull/A19mar/p73.pdf

[50] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[51] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. 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*. ACM, 3:1–3:8. https://doi.org/10.1145/2933349.2933352

[52] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 302–313. https://doi.org/10.1109/ICDE.2013.6544834

[53] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: A Fine-grained Learned Index Scheme for Scalable and Concurrent Memory Systems. *Proc. VLDB Endow.* 15, 2 (2021), 321–334. https://doi.org/10.14778/3489496.3489512

[54] Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. 2023. DILI: A Distribution-Driven Learned Index. *Proc. VLDB Endow.* 16, 9 (2023), 2212–2224. https://doi.org/10.14778/3598581.3598593

[55] Yinan Li, Bingsheng He, Jun Yang, Qiong Luo, and Ke Yi. 2010. Tree Indexing on Solid State Drives. *Proc. VLDB Endow.* 3, 1 (2010), 1195–1206. https://doi.org/10.14778/1920841.1920990

[56] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.* 13, 7 (2020), 1078–1090. https://doi.org/10.14778/3384345.3384355

[57] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A High-Performance Learned Index on Persistent Memory. *Proc. VLDB Endow.* 15, 3 (2021), 597–610. https://doi.org/10.14778/3494124.3494141

[58] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, Pascal Felber, Frank Bellosa, and Herbert Bos (Eds.). ACM, 183–196. https://doi.org/10.1145/2168836.2168855

[59] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13. https://doi.org/10.14778/3421424.3421425

[60] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Optimizing by Sandwiching. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 462–471. https://proceedings.neurips.cc/paper/2018/hash/0f49c89d1e7298bb9930789c8ed59d48-Abstract.html

[61] Donald R. Morrison. 1968. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (1968), 514–534. https://doi.org/10.1145/321479.321481

[62] Gap-Joo Na, Sang-Won Lee, and Bongki Moon. 2012. Dynamic In-Page Logging for B$^+$-tree Index. *IEEE Transactions on Knowledge and Data Engineering* 24, 7 (2012), 1231–1243. https://doi.org/10.1109/TKDE.2011.32

[63] Jun Rao and Kenneth A. Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie (Eds.). Morgan Kaufmann, 78–89.

[64] Jun Rao and Kenneth A. Ross. 2000. Making B$^+$-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein (Eds.). ACM, 475–486.

[65] Suhas SP Rao, Miriam H Huntley, Neva C Durand, Elena K Stamenova, Ivan D Bochkov, James T Robinson, Adrian L Sanborn, Ido Machol, Arina D Omer, Eric S Lander, et al. 2014. A 3D map of the human genome at kilobase resolution reveals principles of chromatin looping. *Cell* 159, 7 (2014), 1665–1680.

[66] Peter Van Sandt, Yannis Chronis, and Jignesh M. Patel. 2019. Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 36–53. https://doi.org/10.1145/3299869.3300075

[67] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. 2009. k-ary search on modern processors. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware, DaMoN 2009, Providence, Rhode Island, USA, June 28, 2009*, Peter A. Boncz and Kenneth A. Ross (Eds.). ACM, 52–60. https://doi.org/10.1145/1565694.1565705

[68] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. 2011. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *Proc. VLDB Endow.* 4, 11 (2011), 795–806. http://www.vldb.org/pvldb/vol4/p795-sewall.pdf

[69] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2016. A Hybrid B+-tree as Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1523–1538. https://doi.org/10.1145/2882903.2882918

[70] Dimitrios Siakavaras, Panagiotis Billis, Konstantinos Nikas, Georgios I. Goumas, and Nectarios Koziris. 2020. Efficient Concurrent Range Queries in B+-trees using RCU-HTM. In *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, Christian Scheideler and Michael Spear (Eds.). ACM, 571–573. https://doi.org/10.1145/3350755.3400237

[71] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: a scalable learned index for multicore data storage. In *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, Rajiv Gupta and Xipeng Shen (Eds.). ACM, 308–320. https://doi.org/10.1145/3332466.3374547

[72] Li Wang, Zining Zhang, Bingsheng He, and Zhenjie Zhang. 2020. 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*. IEEE, 553–564. https://doi.org/10.1109/ICDE48307.2020.00054

[73] Youyun Wang, Chuzhe Tang, Zhaoguo Wang, and Haibo Chen. 2020. SIndex: a scalable learned index for string keys. In *APSys '20: 11th ACM SIGOPS Asia-Pacific Workshop on Systems, Tsukuba, Japan, August 24-25, 2020*, Taesoo Kim and Patrick P. C. Lee (Eds.). ACM, 17–24. https://doi.org/10.1145/3409963.3410496

[74] 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 *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 473–488. https://doi.org/10.1145/3183713.3196895

[75] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are Updatable Learned Indexes Ready? *Proc. VLDB Endow.* 15, 11 (2022), 3004–3017. https://doi.org/10.14778/3551793.3551848

[76] Chin-Hsien Wu, Tei-Wei Kuo, and Li-Ping Chang. 2007. An efficient B-tree layer implementation for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.* 6, 3 (2007), 19. https://doi.org/10.1145/1275986.1275991

[77] Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *Proc. VLDB Endow.* 14, 8 (2021), 1276–1288. https://doi.org/10.14778/3457390.3457393

[78] Shangyu Wu, Yufei Cui, Jinghuan Yu, Xuan Sun, Tei-Wei Kuo, and Chun Jason Xue. 2022. NFL: Robust Learned Index via Distribution Transformation. *Proc. VLDB Endow.* 15, 10 (2022), 2188–2200. https://doi.org/10.14778/3547305.3547322

[79] Zhaofeng Yan, Yuzhe Lin, Lu Peng, and Weihua Zhang. 2019. Harmonia: a high throughput B+tree for GPUs. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 133–144. https://doi.org/10.1145/3293883.3295704

[80] 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 *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1567–1581. https://doi.org/10.1145/2882903.2915222

[81] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *IEEE Trans. Knowl. Data Eng.* 27, 7 (2015), 1920–1948. https://doi.org/10.1109/TKDE.2015.2427795

[82] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 323–336. https://doi.org/10.1145/3183713.3196931

[83] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: A Cache-Aware Learned Index with a Cost-based Construction Algorithm. *Proc. VLDB Endow.* 15, 11 (2022), 2679–2691. https://doi.org/10.14778/3551793.3551823

[84] Shunkang Zhang, Ji Qi, Xin Yao, and André Brinkmann. 2024. Hyper: A High-Performance and Memory-Efficient Learned Index via Hybrid Construction. *Proc. ACM Manag. Data* 2, 3 (2024), 145. https://doi.org/10.1145/3654948

[85] Jingren Zhou and Kenneth A. Ross. 2002. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki (Eds.). ACM, 145–156. https://doi.org/10.1145/564691.564709

[86] Jingren Zhou and Kenneth A. Ross. 2003. Buffering Accesses to Memory-Resident Index Structures. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer (Eds.). Morgan Kaufmann, 405–416. https://doi.org/10.1016/B978-012722442-8/50043-4