FB$^+$-tree: A Memory-Optimized B$^+$-tree with Latch-Free Update

Yuan Chen
School of Computer Science, Wuhan University
yuan.chen@whu.edu.cn

Ao Li
School of Computer Science, Wuhan University
leo210@whu.edu.cn

Wenhai Li*
School of Computer Science, Wuhan University
lwh@whu.edu.cn

Lingfeng Deng
School of Computer Science, Wuhan University
lingfengdeng@whu.edu.cn

ABSTRACT

B$^+$-trees are prevalent in traditional database systems due to their versatility and balanced structure. While binary search is typically utilized for branch operations, it may lead to inefficient cache utilization in main-memory scenarios. In contrast, trie-based index structures drive branch operations through prefix matching. While these structures generally produce fewer cache misses and are thus increasingly popular, they may underperform in range scans because of frequent pointer chasing.

This paper proposes a new high-performance B$^+$-tree variant called **Feature B$^+$-tree (FB$^+$-tree)**. Similar to employing bit or byte for branch operation in tries, FB$^+$-tree progressively considers several bytes following the common prefix on each level of its inner nodes—referred to as features, which allows FB$^+$-tree to benefit from prefix skewness. FB$^+$-tree blurs the lines between B$^+$-trees and tries, while still retaining balance. In the best case, FB$^+$-tree almost becomes a trie, whereas in the worst case, it continues to function as a B$^+$-tree. Meanwhile, a crafted synchronization protocol that combines the link technique and optimistic lock is designed to support efficient concurrent index access. Distinctively, FB$^+$-tree leverages subtle atomic operations seamlessly coordinated with optimistic lock to facilitate latch-free updates, which can be easily extended to other structures. Intensive experiments on multiple workload-dataset combinations demonstrate that FB$^+$-tree shows comparable lookup performance to state-of-the-art trie-based indexes and outperforms popular B$^+$-trees by 2.3x ~ 3.7x under 96 threads. FB$^+$-tree also exhibits significant potential on other workloads, especially update workloads under contention and scan workloads.

PVLDB Reference Format:
Yuan Chen, Ao Li, Wenhai Li, and Lingfeng Deng. FB$^+$-tree: A Memory-Optimized B$^+$-tree with Latch-Free Update. PVLDB, 18(6): 1579 - 1592, 2025.
doi:10.14778/3725688.3725691

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Spear-Neil/IndexResearch.git.

*Wenhai Li is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.
doi:10.14778/3725688.3725691

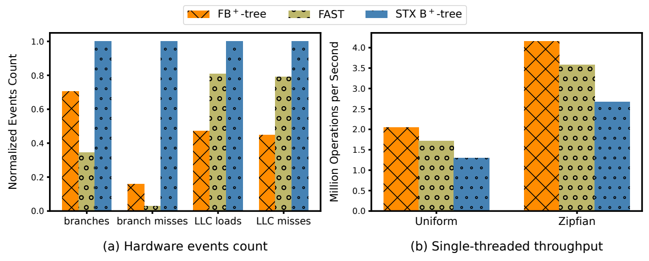

Figure 1: (a) Hardware statistics under uniform distribution, (b) Single-threaded throughput under different distributions.

1 INTRODUCTION

The storage engine significantly influences the overall performance of a database system, especially the index data structure. Even in main-memory systems, index query operations account for 14% ~ 94% of the overhead [35]. B-trees (B$^+$-tree, B*-tree, etc.) are ubiquitous in disk-based database systems because of their prominent IO efficiency [19], yet scarcely exist in main-memory database systems due to their poor utilization of hierarchical caches. ART [43], Masstree [49], and other trie-based structures are generally more efficient than main-memory B$^+$-tree [1, 10, 43, 49, 72], which makes them more prevailing in modern main-memory systems (e.g., Silo [65] and HyPer [31]). What makes main-memory B$^+$-tree untenable is the orders of magnitude difference in access latency between disk and memory. Generally, main memory provides an access latency of 80 ~ 100 ns, whereas the disk and flash access latency are about 10 ms and 50 us, respectively [57]. When IO dominates the overhead of a B$^+$-tree, the impact of CPU caches is nearly negligible. However, effective cache utilization becomes prominent in enhancing index performance when IO is eliminated [27, 43, 49, 56, 60].

Cache and Branch Optimization. A B$^+$-tree consists of inner nodes and leaf nodes. Inner nodes serve as index nodes, containing anchor keys[1] and pointers to child nodes. Key-value pairs are stored in leaf nodes sequentially or semi-ordered [47, 53]. All leaf nodes are linked together in a linked list, enabling efficient range queries. A query starts from the root node and navigates to a leaf node using binary search for branch operations, i.e., index traversal. Thanks to such design, B$^+$-tree shows efficient disk access. But when it comes to main-memory scenarios, B$^+$-tree suffers from several interconnected and complex problems.

First, B$^+$-tree's structure and binary search render it inefficient on cache utilization, as detailed in Section 3. Second, each comparison

[1]Also known as separator keys, we follow the term anchor keys used in previous work.

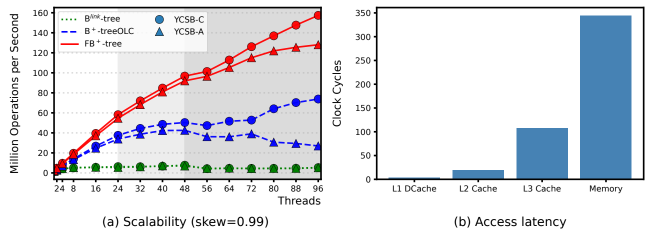

Figure 2: (a) Multi-core scalability of B^{link}-tree, B^{+}-treeOLC, and FB^{+}-tree, (b) Access latency of memory hierarchies.

in binary search depends on the outcome of the previous comparison, and the result of each comparison is hard to predict [43]. This intrinsic characteristic of binary search hinders the full exploitation of modern CPUs' computational and memory-level parallelism potential. Hard-to-predict comparisons pose challenges for dynamic branch prediction, causing modern CPUs' long pipelines to stall. Dependences between instructions render superscalar, speculation, and out-of-order execution powerless [21, 22].

In this paper, we propose a new optimistic main-memory B^{+}-tree variant named **Feature B^{+}-tree (FB^{+}-tree)** to tackle these issues. FB^{+}-tree effectively mitigates the mismatch between B^{+}-tree's memory access patterns and the intrinsic characteristics of hardware architecture through progressively byte-wise parallel processing. In addition, FB^{+}-tree employs a simple for loop for branch operations in most cases. This enables FB^{+}-tree to fully leverage modern CPUs' parallelism potential and to facilitate sequential memory access. Figure 1 illustrates a comparison of throughput and hardware event statistics of FB^{+}-tree, FAST [32] and STX B^{+}-tree [9] during 100 million random 64-bit integer lookups following uniform and zipfian distributions on a dual-socket, 48-core (96-hyperthread) server. Hardware events during lookups under uniform distribution are measured with *perf*. Compared to STX B^{+}-tree, FB^{+}-tree shows fewer branch instructions, branch misses, LLC loads and LLC misses. The FAST tree is a binary tree that collapses multiple nodes into one large node to facilitate SIMD instructions and cache consciousness. Although FAST has fewer branch instructions and branch misses, FB^{+}-tree has better cache utilization due to its discriminative byte processing. This makes FB^{+}-tree outperform STX B^{+}-tree and FAST in single-threaded execution.

Synchronization Protocol. Another crucial factor that impacts the overall performance of a database system is the synchronization protocol of its index. A fine-grained locking protocol, such as lock coupling (hand-over-hand lock) [3] and B^{link}-tree [38, 40], has poor scalability in main-memory database systems. Optimistic lock utilizes a version combined with a lock to detect changes within a node [14, 28, 42, 49]. These protocols enable latch-free[2] index traversal except during node modification, thereby ensuring high scalability. FB^{+}-tree employs a similar yet highly optimized protocol for index traversal and incorporates the link technique [40] for concurrent structure modification (i.e., node split and merge).

The salient innovation of FB^{+}-tree's synchronization protocol is latch-free update. Previous work acquires a lock on a node or locks on more nodes when performing an update. They protect overmuch auxiliary computations in critical sections and frequently retry if they cannot hold the lock, leading to coherence cache invalidations. Such locks thus prevent other threads from performing even unrelated operations and make updates poorly scalable under high-contention workloads. As shown in Figure 2(a), B^{+}-treeOLC synchronized by optimistic lock coupling [42] is more scalable than B^{link}-tree on YCSB-C workload (read-only). However, it suffers from performance collapse on YCSB-A workload (50%-read, 50%-update). FB^{+}-tree mitigates such collapse via latch-free update.

In short, FB^{+}-tree employs a more fine-grained synchronization protocol which enables an update to be performed without acquiring any locks. FB^{+}-tree executes an update using the compare-and-swap (CAS) instruction. This way, a lookup can be executed simultaneously with updates, and updates only contend on the same key-value pairs. Our protocol offers a general technique for performing updates without acquiring any locks. Index structures synchronized by optimistic lock can implement latch-free updates with a few changes to existing code. Benchmarks on YCSB workload A (update heavy) demonstrate that FB^{+}-tree outperforms existing indexes with optimistic lock under contentions.

Contributions and Paper Organization. This paper makes the following contributions:

- A technique supporting arbitrary key types named feature comparison is introduced, which allows FB^{+}-tree to work like a trie and leverage modern CPUs' computational and memory-level parallelism potential. This technique combined with hashtags in leaf nodes enables FB^{+}-tree to outperform typical B^{+}-trees in both performance and scalability on index traversal.
- Thanks to feature comparison, FB^{+}-tree does not need to frequently access anchor keys like typical B^{+}-trees. FB^{+}-tree thus only stores pointers to anchor keys for space efficiency.
- A highly optimized optimistic synchronization protocol is devised to facilitate multi-core scalable concurrent index accesses. In particular, the protocol introduces a general latch-free update technique that utilizes CAS primitive for update operations without holding any locks.
- Comprehensive experiments are conducted on multiple workload-dataset combinations for existing index structures and FB^{+}-tree. The results demonstrate that FB^{+}-tree outperforms popular B^{+}-trees by 2.3x ~ 3.7x on read-only workloads under 96 threads. On update-heavy workloads, FB^{+}-tree shows the best scalability thanks to latch-free update technique.

The rest of this paper is organized as follows. Section 2 introduces background and related work. Section 3 presents the motivation, data structure, and algorithm of FB^{+}-tree. Section 4 describes the synchronization protocol enabling FB^{+}-tree multi-core scalable. Section 5 shows the experiment setup, workloads, and evaluation results. The conclusion is given in Section 6.

2 RELATED WORK

Querying an index for a key (i.e., index traversal) involves the process of narrowing the search key space until confirming whether the key exists. In B-trees, the process relies on comparisons between the target key and the anchors. Meanwhile, in tries, it depends on prefix matching. This section provides some background and related work

[2]We follow the term "latch-free" used in OLFIT [14] which means without acquiring locks to distinguish it from the term "lock-free" which means non-blocking. We follow the synchronization literature to use "lock" except "latch-free".

on ordered index. In addition, previous work on synchronization protocols is presented at the end.

Memory Access. From a more detailed perspective, index traversal can be divided into memory access and actual computation. Compared to memory access, the computational overhead is almost negligible. On modern machines, several to hundreds of instructions can be evaluated and retired in parallel, and most SIMD instructions have 1-cycle throughput [21, 22, 25]. However, memory access typically takes several hundred cycles to complete, as shown in Figure 2(b). Various main-memory ordered indexes have been proposed to optimize cache utilization. We categorize existing ordered indexes into three groups: B-trees, tries, and learned indexes. Next, we introduce their related work respectively.

B-trees. B-trees are a class of balanced, comparison-based indexes[3], encompassing many variants such as B-tree, B*-tree, B$^+$-tree, binary B-tree, AVL tree, and others [19]. As memory capacity increased, the T-tree placed multiple records in one binary tree's node for main-memory database systems [41]. The CSS-trees [55] and CSB$^+$-tree [56] store each node's children in contiguous memory to mitigate pointer chasing. Bender et al. conducted an intensive theoretical analysis and experimental evaluation on cache-oblivious B-trees [5–8]. Chen et al. proposed to optimize B$^+$-trees' performance through prefetching for both cache and disk [17, 18].

Currently, software prefetching is extensively used in main-memory indexes [10, 43, 49], as it targets short array streams and irregular memory address patterns [24, 39, 66]. The prefix B-trees explored not directly storing keys but constructing prefixes in inner nodes [4, 26]. The pkT-trees and pkB-trees store fixed-size parts of keys directly in the tree nodes [12, 52]. The B^2-tree organizes inner nodes as embedded trie-like structures for indexing string keys [59]. The DB$^+$-tree [37] incorporates the discriminative bits from HOT [10] into B-trees. From another perspective, k-Ary, FAST , P-ary, etc. exploit binary search with SIMD instructions to leverage parallelism of both CPUs and GPUs [30, 32, 58, 61, 75]. PALM performs multiple concurrent queries in batches [60].

Tries. Tries, also known as radix trees, prefix trees, or digital search trees, drive branch operations through prefix matching, as illustrated in Figure 3. These data structures directly use the digital representation of keys and may have excessive worst-case space consumption [43]. The Patricia trie introduces path compression and only stores prefixes of keys in trie nodes [51]. The burst trie substitutes leaf nodes with containers maintaining a small set of keys [29]. The HAT-trie maintains hash tables as containers for better performance [1]. The generalized prefix tree is a trie with variable prefix length for indexing arbitrary data types [13].

The Adaptive Radix Tree (ART) adaptively uses four different node layouts depending on the number of child nodes [43]. Path compression and lazy expansion allow ART to efficiently index string keys by collapsing nodes. The Masstree is a trie-like concatenation of B$^+$-trees, where each trie node is a B$^+$-tree indexing different 8-byte slices of keys [49]. The Height Optimized Trie (HOT) dynamically varies the number of discriminative bits considered at each node [10, 11]. Similar to B$^+$-trees, another interesting design Wormhole [68] maintains a double-linked list of leaf nodes. It then constructs a trie with all prefixes of the lower bound of leaf nodes and represents the trie with a hash table. The Cuckoo Trie shares a similar idea and exploits memory-level parallelism to alleviate pointer chasing [71].

Learned Indexes. Indexes can be viewed as models to map a key to the position of a record. Kraska et al. proposed learned indexes (machine learning models) as replacements for index structures. They have demonstrated that learned indexes have the potential to offer benefits over state-of-the-art indexes [36]. The ALEX exploits linear regression combined with exponential search to guarantee accuracy and enable adaptive update [23, 48]. Some previous work has explored indexing string keys using learned indexes [33, 34, 64, 70]. Learned indexes and database systems with AI have become a prominent research topic, and extensive work has been dedicated to making them practical [16, 33, 46, 73, 74, 76].

In summary, previous research has demonstrated that tries outperform B-trees in terms of point lookup. For range iteration, B-trees, especially B$^+$-trees, demonstrate better performance because of their balanced structure. The Wormhole combines the strengths of B$^+$-tree and trie. Unfortunately, its hashed representation of meta-trie and indirect ordered leaf node may incur significant overhead for insert and range scan operations, respectively. Learned indexes may offer advantages; however, challenges still exist in achieving accuracy, adaptive update capabilities, and indexing string keys.

Synchronization on Indexes. Designing an efficient synchronization protocol is critical and challenging. Lock coupling [3] and B^{link}-tree [40] lock only one node simultaneously, thus demonstrating good scalability in traditional database systems. Optimistic lock coupling, OLFIT, Masstree, and many other indexes combine version with lock to avoid coherence cache miss [10, 14, 28, 42, 43, 49, 71]. In these protocols, readers verify the version to detect changes within a node without acquiring the lock. Meanwhile, writers acquire a write lock and update the version when modifying a node. The Read-Optimized Write Exclusion (ROWEX) protocol used by ART and HOT goes one step further by only providing exclusion relative between writers while allowing readers to always succeed without block nor restart [10, 42, 44].

OLFIT and several indexes employ a bottom-up strategy for structure modification, e.g. split and merge, and adopt the link technique and high key from B^{link}-tree for concurrency support [14, 49]. Optimistic lock coupling provides a more general approach to synchronize index structures [42, 44], utilizing a top-down strategy for concurrent structure modification. If a change occurs during index traversal, it restarts from the root. ROWEX utilizes subtle atomic operations to ensure that reads are always consistent and never restart [44]. OptiQL incorporates queue-based locking and opportunistic read techniques into optimistic lock coupling to mitigate performance collapse under high contention [63]. The Bw-tree implements lock-free operations through delta records and mapping table [45, 67], but it may incur heavy overhead. Some work has proposed universal construction strategies to transform trees from sequential code to linearizable concurrent versions [62].

We have learned lessons from previous work. FB$^+$-tree considers a B$^+$-tree from trie's prefix matching perspective. Common prefixes and parts of anchors are directly stored in inner nodes but with a quite different arrangement. Unlike k-ary, FAST, etc., which

[3]Skip list [54] and its variants are also comparison-based.

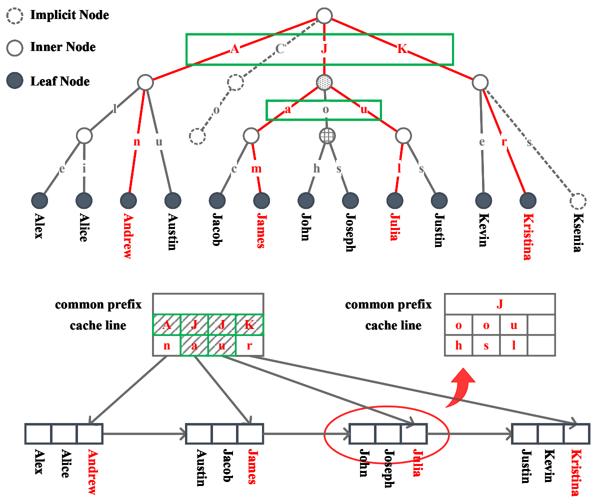

Figure 3: Illustration of a trie and the main idea of FB$^+$-tree

only consider data types supported by SIMD instructions, FB$^+$-tree facilitates byte-wise parallel processing for branch operations, allowing arbitrary data types. For concurrency support, FB$^+$-tree uses a highly optimized optimistic synchronization protocol and employs the link technique for concurrent structure modification. Furthermore, FB$^+$-tree allows multiple writers to perform updates on one leaf node without blocking other updates nor reads.

3 THE FB$^+$-TREE DATA STRUCTURE

In this section, we start by providing a comprehensive analysis of the overhead associated with comparison operations in B$^+$-trees. Next, we introduce our feature comparison technique, FB$^+$-tree's data structure, and the branch algorithm. The synchronization protocol is discussed in Section 4.

3.1 Dissonance between B$^+$-tree and Hardware

As previously mentioned, binary search is employed to retrieve child nodes in index traversal of B$^+$-trees. In binary search, the anchor key for comparison is the median of the current search key space. This leads to a complex memory access pattern that underutilizes effective hardware and software prefetching. Moreover, hard-to-predict comparisons and dependences between comparisons make memory-level parallelism unattainable [71].

Consider a B$^+$-tree indexing string keys, where only pointers to anchor keys are stored in inner nodes and the contents of anchors are stored in memory allocated via malloc. Assuming a fanout factor 256, binary search in an inner node would typically involve eight comparisons. On modern 64-bit machines, storing 256 pointers to anchors would require 256 * 8 bytes of space, equivalent to 32 cache lines. Consequently, each binary search would access six cache lines for pointers alone, whereas only eight pointers are effectively used. For B$^+$-trees indexing integer keys, similar issues arise. Several indexes therefore configure their nodes as 256 bytes with a proper fanout to alleviate this cache inefficiency. [10, 43, 49, 67].

After obtaining the pointer to an anchor, the content of the anchor is compared with that of the target key following dereference. No matter how many bytes the anchor has and how many bytes are used for comparison, at least one complete cache line is loaded from memory. However, in many cases, only a few bytes are necessary to establish the relative order between string keys (similarly for integer keys) [4, 12, 52]. Consequently, this results in wasted memory bandwidth and the eviction of several hot cache lines.

This problem is exacerbated on modern CPUs, because continuous several cache lines will be automatically prefetched but remain unused, leading to bandwidth waste [21, 24]. In concurrent environments, these random small memory accesses further impede full utilization of memory bandwidth and Ultra Path interconnect (UPI) bandwidth [21, 71]. These problems result in B$^+$-trees' sub-optimal scalability even on read-only workloads.

3.2 Feature Comparison

3.2.1 Motivation. Tries have advantages over B$^+$-trees in cache utilization, as they drive branch operations through prefix matching. The key idea of FB$^+$-tree is to integrate such mechanism into B$^+$-trees to benefit from architecture while preserving B$^+$-trees' properties, particularly balance. We start with the intrinsic similarities between B$^+$-trees and tries, as illustrated in Figure 3.

In B$^+$-trees, the entire key space is divided into intervals defined by anchors in inner nodes, which serve as the upper and lower bounds of these intervals. In contrast, tries partition the entire key space into sub-trees using prefixes. B$^+$-trees and tries are functionally equivalent in this partition sense. In other words, trie nodes inherently imply the relative order between keys through their digital representation. Tries accomplish key space partitioning with byte-wise prefix matching (or using smaller radix), whereas comparisons between strings are also performed in a byte-wise manner. Comparison between binary keys can be conducted similarly after code transition, as detailed in Subsection 3.6. These similarities suggest that branch operation in B$^+$-trees could potentially be implemented similarly to prefix matching in tries.

As shown in Figure 3, we build a trie and a B$^+$-tree for a collection of keys, where the anchor keys of the B$^+$-tree are highlighted in both structures. Consider a use case of querying the trie for the key "John". The most significant byte "J" is used to retrieve the dotted node following the root node. The search key space is then reduced to the sub-tree with the dotted node as root. Next, the latticed node is retrieved via the second byte "o", followed by reaching the leaf node "John". Revisiting this byte-wise prefix matching from a B$^+$-tree's comparison perspective, it implies the first byte "J" is compared with the first byte of the four anchor keys "Andrew", "James", "Julia", and "Kristina" simultaneously. After this byte-wise parallel comparison, the search key space is narrowed down to the keys greater than "Andrew" and less than "Kristina".

3.2.2 Algorithm. Obviously, this byte-wise parallel comparison can be implemented with SIMD instructions. B$^+$-tree's branch operation can be implemented with progressively byte-wise processing, as shown in Figure 3. As anchor keys may share a common prefix, byte-wise processing on the common prefix is unproductive and space-inefficient. FB$^+$-tree constructs the common prefix in inner nodes and performs byte-wise parallel processing on the following bytes. *An interesting property similar to prefixes in tries is that the common prefix length of a child node is no smaller than that of its parent node.* Branch operation could skip the common prefix and

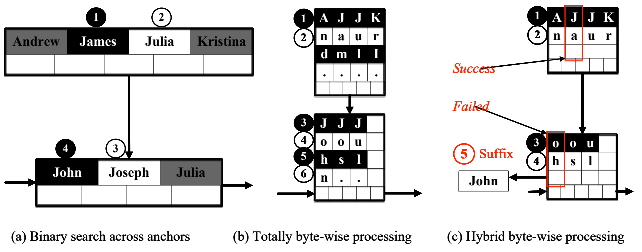

Figure 4: Binary search vs. FB$^+$-tree's branch algorithm.

index traversal could gradually handle different slices of a target key on each level of inner node. FB$^+$-tree thus almost becomes a trie in terms of computational complexity.

One reason why tries are generally more cache-conscious is their capability to fit complex data distribution or prefix skewness. Trie nodes shared by more keys would reside in a cache level closer to CPUs. By contrast, B$^+$-trees mechanically select the middle key as the anchor key during node split, which is unconscious of prefix skewness. In other words, several anchors may share a much longer common prefix (sparse keys). Performing byte-wise parallel processing over all the remaining bytes following the common prefix thus may be inefficient. For space efficiency, FB$^+$-tree only stores a few fixed-size discriminative bytes of anchors directly in inner node. A binary search over suffixes will be performed when byte-wise parallel comparison cannot achieve branch operation.

For example, consider the B$^+$-tree's third leaf node shown in Figure 3 as an inner node. We show the process of querying the key "John" with binary search, totally byte-wise parallel processing, and FB$^+$-tee's hybrid branch algorithm respectively in Figure 4, where the numbers highlight the order of memory access along hierarchical inner nodes. Binary search generates irregular and small memory access, as shown in Figure 4(a). Totally byte-wise parallel processing does not take into account common prefixes and sparse keys, as shown in Figure 4(b). Supposing only two bytes following common prefix are stored directly in inner nodes in Figure 4(c). In the root node, branch operation succeeds after two byte-wise comparisons. In the child node, the common prefix "J" can be skipped. In this special case, byte-wise parallel processing fails in branch and a binary search over suffixes has to be performed.

3.2.3 Discussion. With this byte-wise parallel processing on discriminative bytes, FB$^+$-tree works like a trie and alleviates direct access to anchors. As index traversal descends to a leaf node, common prefix length grows and branch operation gradually processes different slices of a target key. Therefore, binary search over suffixes would occur with very low probability. FB$^+$-tree thus evolves into a trie in the best case. In the worst case, branch operations on all inner nodes depend on binary search over suffixes, which is almost impossible. FB$^+$-tree thus degenerates into a B$^+$-tree.

In most cases, FB$^+$-tree forms a hybrid structure that combines the characteristics of both B$^+$-trees and tries. Prefix matching is significantly efficient for dense keys but sub-optimal for sparse keys[4] [10, 43, 49]. Binary search generally incurs expensive cache misses, while not affected by the sparseness of keys. FB$^+$-tree employs byte-wise parallel processing for dense keys to enhance cache consciousness. For sparse keys, this byte-wise processing on discriminative bytes acts as a filter to narrow the search space for binary search. This hybrid structure allows FB$^+$-tree to efficiently index both sparse and dense keys while maintaining a balanced structure that facilitates efficient range iteration.

To some extent, these discriminative bytes imply some data distribution characteristics of the keys in node intervals (sparse or dense). Therefore, we refer to these bytes as features and name this byte-wise parallel processing as feature comparison. Even though FB$^+$-tree stores feature bytes, FB$^+$-tree only stores pointers to anchor keys in inner nodes, making it even more space-efficient than typical B$^+$-trees. Next, we introduce the node implementation, algorithm, and some optimizations.

```
// for binary keys          // for string keys
struct InnerNode:           struct InnerNode:
    Control control;            Control control;
    int knum;                   int knum;
    int plen;                   int plen;
    char prefix[8];             String* huge;  // huge prefix
    node* next;                 node* next;
    char features[fs][ns];      char features[fs][ns];
    node* children[ns];         char tiny[224]; // embedded prefix
                                String* anchors[ns];
                                node* children[ns];

struct LeafNode:            struct String:
    Control control;            int len;
    uint64_t bitmap;            char str[];
    KeyType high_key;
    LeafNode* sibling;      struct KVPair:
    char tags[ns];              KeyType key;
    KVPair* kvs[ns];            ValueType value;
```

Figure 5: Node structures of FB$^+$-tree.

3.3 Node Implementation

Figure 5 illustrates the node structures of FB$^+$-tree. The *control* is an 8-byte atomic variable that governs the synchronization behaviors. The *knum* and *plen* indicate the number of anchor keys and the length of the common prefix in an inner node. The *next* specifies an inner node's sibling or last child. The *fs* and *ns* which can be manually configured represent the feature and node size, respectively. In leaf nodes, the *bitmap* indicates whether the corresponding slot in *kvs* is occupied and the *tags* field contains the corresponding hashtags. The *high_key* is the upper bound of a leaf node.

Essentially, both structures of FB$^+$-tree's inner nodes and leaf nodes are identical to that of a typical B$^+$-tree, except for the features in inner nodes and the hashtags in leaf nodes. To facilitate concurrent structure modification, all nodes on the same level are linked in a single-linked list. Each node starts with an 8-byte *control* field, indicating the node type. Binary keys and string keys share similar inner node and leaf node structures.

Leaf Node. Typically, key-value pairs are ordered in leaf nodes, which could lead to time-consuming rearrangement and binary search during insertion. These complex operations within a critical section may limit multi-core scalability. To address these issues, we store the unsorted key-value pairs in a pointer array *kvs* and

[4]For tries, node collapse or path compression can mitigate this problem but not completely. HOT dynamically changes the number of discriminative bits used in a node, and meanwhile almost stays a constant fanout. HOT thus achieves almost extreme cache utilization for lookup. Unfortunately, the common problems of trie structures still exist, which is inefficient and complicated concurrent range iteration.

```
1   void* branch(String& key) { // inner node
2     void* node = nullptr;
3     int pcmp = prefix_compare(key);
4     if (pcmp == 0) { // prefix matches
5       int idx, fid, cmps = min(fs, key.len - plen);
6       uint64_t mask, eqmask = (0x01ul << knum) - 1;
7       for (fid = 0; fid < cmps; fid++) {
8         char byte = key.str[plen + fid] + 128;
9         mask = compare_equal(features[fid], byte);
10        mask = mask & eqmask;
11        if (mask == 0) break;
12        eqmask = mask;
13      } // feature comparison
14      if (fid < cmps) {
15        char byte = key.str[plen + fid] + 128;
16        mask = compare_less(features[fid], byte);
17        mask = mask & eqmask;
18        if (mask == 0) idx = index_least1(eqmask);
19        else idx = 64 - countl_zero(mask);
20      } else { // binary search on suffixes
21        int hid = 64 - countl_zero(eqmask);
22        int lid = index_least1(eqmask);
23        idx = suffix_bs(key, plen + cmps, lid, hid);
24      }
25      node = children[idx];
26    } else { node = children[0]; }
27    return node;
28  }

30  KVPair* lookup(String& key) { // leaf node
31    char tag = hash(key.str, key.len);
32    uint64_t mask = compare_equal(tags, tag);
33    mask = mask & bitmap; // candidates
34    while (mask) {
35      int idx = index_least1(mask);
36      KVPair* kv = kvs[idx].load();
37      if (kv != nullptr && key == kv->key)
38        return kv; // key found
39      mask &= ~(0x01ul << idx);
40    }
41    return nullptr; // key not found
42  }

44  // compare 64 bytes to a char 'c', AVX512
45  uint64_t compare_equal(void* p, char c) {
46    __m512i v1 = _mm512_loadu_si512(p);
47    __m512i v2 = _mm512_set1_epi8(c);
48    return _mm512_cmpeq_epi8_mask(v1, v2);
49  }
```

Figure 6: Lookup Related Algorithms

utilize hashtags for efficient lookup, as in previous work [47, 53, 68]. Additionally, each leaf node includes a *high_key* to support concurrent structure modification, which indicates the upper bound. For binary keys, the *high_key* is directly stored in the node; for string keys, each leaf node maintains a pointer to *high_key*.

Inner Node. The anchor keys in an inner node are ordered. For binary keys, all bytes of an anchor are directly stored in *features*. The common prefix is stored in *prefix*. For better performance, the prefixes of anchors are truncated and the remaining bytes in *features* are shifted adjacent to the *next* field.

For string keys, anchor keys are stored in a pointer array—*anchors*. Thanks to feature comparison, branch operation would rarely access anchors. Therefore, unlike typical B$^+$-trees that copy anchor keys into inner nodes, FB$^+$-tree stores the actual contents of anchor keys in leaf nodes (i.e., *high_key*), while inner nodes only maintain pointers to *high_key*, which makes FB$^+$-tree more space-efficient. Whenever possible, the entire common prefix is embedded in the *tiny* field[5]. Slab memory allocators, such as jemalloc and tcmalloc, always allocate a memory block not smaller than the required size. Therefore, we configure the size of *tiny* to fully utilize any excess memory available. The *huge* field points to the first anchor as the common prefix in case it is too long to reside in *tiny*.

[5]For the sake of concurrency support, the entire common prefix is embedded in an inner node, even if the parent and child nodes have exactly the same common prefix.

3.4 Lookup and Update

Figure 6 presents the slightly simplified code for *branch* in inner nodes and *lookup* in leaf nodes, without considering concurrency. The lookup process in an FB$^+$-tree is identical to that in a typical B$^+$-tree. Index traversal utilizes the *branch* algorithm (lines 1 ~ 28) to retrieve child nodes until a leaf node. Subsequently, the hash-based *lookup* algorithm (lines 30 ~ 42) is executed to retrieve the key-value pair in the leaf node. Similar processes are used for binary keys. The update process of FB$^+$-tree has a minor difference from the lookup process (lines 36 ~ 38). An atomic update operation based on CAS is employed without holding any locks.

The *branch* algorithm begins by comparing the target key with the common prefix (line 3). The most significant difference between FB$^+$-tree's branch algorithm and typical binary search lies in the feature comparison (lines 7 ~ 13). Each byte of the target key following the prefix is compared with the corresponding byte in features (line 9), until either the last feature is reached or there is no more matching byte. No matching bytes in features means that the child node could be determined immediately using the *compare_less* function (line 16). In cases where a binary search on suffixes is necessary, the bytes including both the common prefix and features are truncated to improve performance (line 23). The reason why 128 is added to each byte (lines 8 and 15) will be presented in Subsection 3.6.

The *lookup* algorithm employs a hash fingerprint in leaf nodes. Candidates are first filtered with hashtags (line 32), followed by a verification comparison using the real content to prevent false positives (line 37). Lines 45 ~ 49 show the implementation of *compare_equal* using AVX512. BMI and SIMD instructions, such as LZCNT, are utilized for efficient bit manipulation, for instance, *index_least1* (line 18) and *countl_zero* (line 19).

3.5 Insert and Remove

A key-value insertion into a leaf node simply requires locating an empty slot using hashtags and then installing the key-value into *kvs*. FB$^+$-tree adopts a bottom-up strategy for insertion that involves structure modification. The algorithm for finding the position to insert an anchor key into an inner node also relies on feature comparison. The primary disparity lies in the recomputation of the common prefix and features. The common prefix of an inner node is recomputed only when the new anchor is less than the minimum anchor key. In most cases, an anchor key insertion could be easily conducted by inserting the pointer and features. Meanwhile, inner nodes are modified only during structure modifications. Remove follows a similar process.

3.6 Optimization and Tricks

We summarize some optimizations and technique tricks in FB$^+$-tree:

Optimization. As discussed in previous work, tree nodes of four cache lines exhibit the highest overall performance [49, 67]. Modern machines have nearly identical latency for 64-byte and 256-byte memory accesses. For better single-threaded performance and space

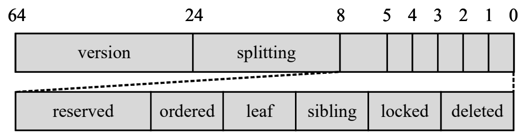

Figure 7: The layout of optimistic lock variable (*control*).

efficiency, the *ns* and *fs* are configured to 64 and 4, respectively. A smaller fanout increases the tree depth, while a larger feature size entails more bandwidth requirements and incurs higher access latency. A feature size of four ensures that feature comparison would not lead to excessive overhead when it fails to determine the branch. To minimize space consumption, the inner nodes only store pointers to anchor keys (i.e., *high_key* in leaf nodes). The *high_key* serves as the upper bound of a leaf node and can be constructed using discriminative prefixes to improve performance and space consumption. Additionally, anchor keys in an inner node can also be copied to contiguous memory to enhance locality.

Tricks. The relative magnitude between two unsigned integers can be determined by byte-wise comparison. For signed integers, however, such a pattern doesn't work because of its complement representation. In essence, complement representation is designed to utilize the overflow bit to operate positive and negative integers uniformly. The relative magnitude among positive or negative integers depends on the remaining bits except for the signed bit. For example, in 8-bit complement representation, -2, -1, 0, 1, and 2 are represented as 0xFE, 0xFF, 0x00, 0x01, and 0x02, respectively. An unsigned comparison on the integers whose sign bit is flipped can be treated equivalently to a signed comparison on the original integers, and vice versa. Except for AVX512, the unsigned byte comparison instructions are not supported in AVX2 and SSE2. Therefore, we add a magic number 128 before feature comparison in Figure 6, making FB$^+$-tree widely applicable.

4 SYNCHRONIZATION PROTOCOL

Besides cache utilization, the performance of a main-memory index significantly depends on its synchronization protocol. In this section, we begin by introducing lock-based synchronization protocols and their optimistic variants. Next, we present how FB$^+$-tree is synchronized using an optimistic protocol and an optimization that mitigates the overhead caused by synchronization protocol during index traversal. In particular, we propose a general latch-free update technique that allows updates without holding any locks.

4.1 Preliminaries to Index Synchronization

Index synchronization consists of two parts: node protection and concurrent structure modification. Traditional database systems typically use pessimistic lock for node protection, along with lock coupling [3] or the link technique from B^{link}-tree [38, 40] for concurrent structure modification. In main-memory scenarios, optimistic lock combines version with lock to replace pessimistic lock, allowing index traversal without holding locks [14, 42, 44, 49]. For structure modification, optimistic lock coupling (OLC) keeps track of versions across multiple nodes and restarts if a change occurs [42, 44]. OLFIT proceeds to sibling node when detecting a structure modification through comparison with high key [14, 40].

Optimistic lock offers a general approach to accessing a node without holding the lock, making it particularly beneficial for read-heavy workloads. However, update-heavy workloads under contention suffer from performance collapse in two aspects. First, existing optimistic locks are typically implemented as spin locks with optimistic reads [28, 42, 44, 63]. Writers must acquire an exclusive write lock via CAS instruction before modifying a node. This often leads to a scenario where many writers frequently retry CAS in a single node until they hold the lock. A backoff algorithm may somewhat mitigate this collapse, but not completely. Second, readers usually have to wait for the writer's modification and may have to restart if the version validation fails.

OptiQL extended the classic MCS lock with optimistic reads to alleviate the former problem [50, 63]. ROWEX mitigates the latter problem by only providing exclusion relative among writers while allowing readers to always succeed without block nor restart [42, 44]. The Bw-tree, incorporating lock-free semantics through chaining delta records and mapping table, seems to be an ideal solution. Unfortunately, delta records require expensive merge operations and the mapping table incurs additional overhead for other operations. FB$^+$-tree alleviates the former problem by minimizing the critical section to allow latch-free update operation. Read operations thus can always succeed when concurrent with updates, which eliminates the latter problem. We show their comparisons when reads are concurrent with updates in Table 1. Next, we introduce FB$^+$-tree's synchronization protocol.

Table 1: Comparisons of Synchronization Protocols

	OLC	ROWEX	OptiQL	Bw-tree	FB$^+$-tree
latch-free update				✓	✓
no merge overhead	✓	✓	✓		✓
non-blocking read		✓		✓	✓
no auxiliary struct	✓	✓	✓		✓

4.2 FB$^+$-tree's Synchronization Protocol

FB$^+$-tree utilizes a similar optimistic lock for node protection and latch-free index traversal. Figure 7 illustrates the layout of per-node optimistic lock variable, denoted as *control* in Figure 5. Insert and remove operations increment the *version*, while update operations do not, which differs from previous work. The *splitting* is only used in leaf nodes, indicating whether the node is undergoing a split and if the new node has not been inserted into its parent node. The *ordered* field indicates whether the key-value pairs in leaf nodes are ordered, which is lazily set only when either split and merge of leaf nodes or range iteration. The *leaf* specifies the type of node. The *sibling* indicates whether the node has a sibling node. The *locked* is set when acquiring an exclusive write lock. The *deleted* is set if the node's contents have been merged into its left-sibling node, indicating that the node can be safely reclaimed later.

Node Protection. Index traversal loads the *version* before accessing a node and validates that it has not changed after the access. If validation fails, node access needs to restart. Both insert and remove operations acquire an exclusive write lock to prevent concurrent modification within the same node. Since read-only node

```
1   KVPair* lookup(String& key) {
2     void* node = root_;
3     while (!is_leaf(node)) {
4       node = InnerNode(node)->branch(key);
5     }
6     uint64_t version;
7     do { // descending to leaf node
8       version = Control(node)->begin_read();
9       while (LeafNode(node)->to_sibling(key, node)) {
10        version = Control(node)->begin_read();
11      } // check whether to proceed to sibling node
12      KVPair* kv = LeafNode(node)->lookup(key);
13      if (kv != nullptr) return kv; // key found
14    } while (!Control(node)->end_read(version));
15    return nullptr; // key not found
16  }
```

Figure 8: Concurrent Lookup Algorithm

access may occur concurrently with node modification, atomic operations are employed for insert and remove operations. Lookup operation atomically loads the pointer, thereby preventing readers from accessing a partially modified key-value.

Concurrent Structure Modification. FB$^+$-tree adopts the link technique from B^{link}-tree.[6] The top half of Figure 10 illustrates the process of concurrent node split.[7] A leaf node split involves two steps: (1) transfer the greater half key-value pairs into the newly created node, denoted as n'; (2) insert the pointer to node n' into its parent node p. A leaf node n is said to be undergoing a split until the pointer to the new node n' is inserted into its parent node.

After step (2), an index traversal could correctly descend to node n'. An incorrect leaf node occurs only when an index traversal descends to a leaf node that is undergoing a split. Given that the key-values in node n' are greater than those in node n, it can be addressed through an alternative bypass by linking node n to n'. Therefore, upon descending to a leaf node, a comparison is performed with the upper bound to determine whether it is necessary to proceed to the right sibling node. Split operations that propagate to higher-level nodes can be performed iteratively.

4.3 Concurrent Lookup

Figure 8 presents the concurrent lookup algorithm. The *branch* algorithm (line 4) has been slightly modified to start by loading the *version* using *begin_read* and to restart if validation fails using *end_read*. Unlike its sequential version in Figure 6, the *branch* may return a sibling node if a node split occurs. Similarly, the *lookup* in leaf node (line 12) is protected by *begin_read* and *end_read*. To avoid retrieving an incorrect leaf node, the *to_sibling* function (line 9) performs a comparison with *high_key* upon descending to a leaf node. If the key-value pair is found, the result is returned directly to eliminate unnecessary restarts (line 13).

Cross-Node Tracking. The overhead of comparison with *high_key* is negligible for traditional systems, but it may be expensive in main-memory scenarios. Since structure modifications occur infrequently, this comparison is unnecessary in most cases. We eliminate this comparison overhead through a technique named cross-node tracking. Index traversal descends to an incorrect leaf node only

[6]FB$^+$-tree can also utilize lock coupling for concurrent structure modification. We employ the link technique because it facilitates optimizations to latch-free update.
[7]Concurrent node merge can be implemented similarly to node split [38].

```
node = index_traversal()
node->lock_exclusive()
... querying / validation
... install kv into kvs
node->unlock_exclusive()
```

```
node = index_traversal()
ver = node->begin_read()
... querying / validation
... install kv into kvs
node->end_read(ver)
```

Figure 9: Lock-based (left) and latch-free update (right).

when the node is undergoing a split. Additionally, a leaf node split ends up after inserting the new anchor key into its parent node.

We thereby embed a *splitting* field in *control* indicating the node is undergoing a split, which is set when step (1) begins. Once the new anchor key is inserted into the parent node in step (2), the *splitting* field is unset and the *version* of the parent node is incremented. Through validating the *splitting* and the *version* of the parent node, it can be shown that index traversal descends to a correct leaf node. Consequently, the comparison is performed only when the *splitting* is set or the *version* of the parent node has changed.

4.4 Latch-free Update

We propose a general latch-free update technique to achieve good scalability in heavy-contention scenarios. Figure 9 with the critical sections highlighted illustrates the key difference between the traditional lock-based approach and latch-free update. One primary reason for their sub-optimal scalability is that too many ancillary operations are protected inside the critical section. In previous optimistic protocols, a write lock is held to prevent other concurrent modifications when reaching the leaf node. Unfortunately, querying the node for the existing key-value pairs or validation process is also protected in the critical section. Our latch-free update allows this process to be executed in parallel. Only the process of installing kv into *kvs* is protected in the critical section.

When only considering update and lookup, tree structure and key-value residences in leaf nodes never change. Update operation thus can be easily implemented using CAS primitive. The challenging problem arises when another thread concurrently performs a split operation on the leaf node. As a result, the key-value to be replaced might have been moved to another node. For example, consider an update operation that tries to replace the key-value 5-5 with 5-6 as shown in the bottom half of Figure 10. In this case, the update descends to the leaf node and then stalls, while another thread performs a node split on this node. When the update resumes, it definitely fails because the key-value with key 5 cannot be found. Similar issues exist in node merge and node rearrangement.

Our latch-free update solves this false negative problem by checking the version. If the version has not changed, the update fails because the key-value to be updated has not been inserted yet. Otherwise, if the version changes, it implies that the key-value may have been moved to another node. In such cases, a comparison with *high_key* is then performed to determine whether the key-value is beyond the current node. If so, it proceeds to the sibling node to perform the update again, as illustrated in the bottom half of Figure 10. Otherwise, it indicates that either the node may have been rearranged or the key-value may have been removed. Restart in the leaf node could deal with these situations.

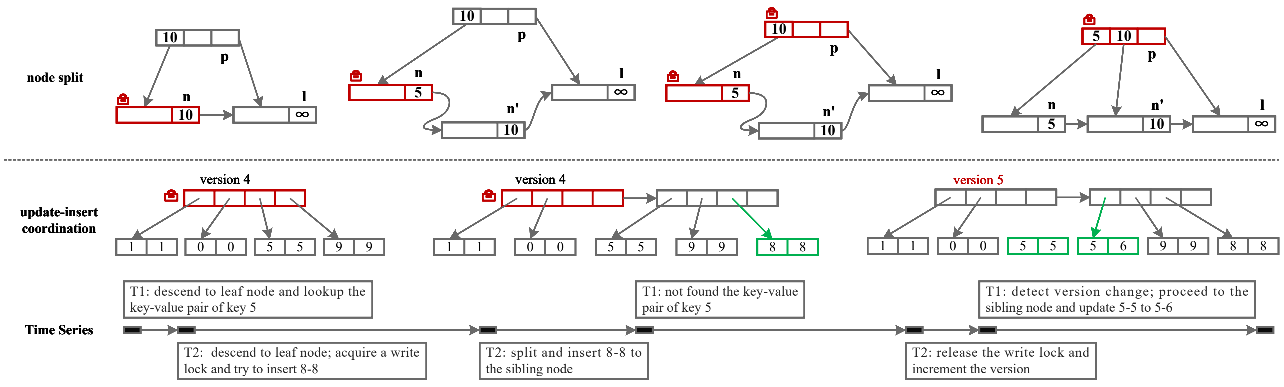
node split
p
n
l
n'
update-insert coordination
version 4
version 5
Time Series
T1: descend to leaf node and lookup the key-value pair of key 5
T2: descend to leaf node; acquire a write lock and try to insert 8-8
T2: split and insert 8-8 to the sibling node
T1: not found the key-value pair of key 5
T2: release the write lock and increment the version
T1: detect version change; proceed to the sibling node and update 5-5 to 5-6

Figure 10: Illustration of structure modification (top half) and update-insert coordination (bottom half, T1-update, T2-insert).

To coordinate with latch-free update when moving key-value pairs to another node, we utilize atomic exchange instruction to replace the pointer with NULLPTR and obtain the latest pointer to a key-value. Concurrent updates would thus fail because they load a NULLPTR or fail to perform CAS, and then proceed to the sibling node accordingly. Node merge and rearrangement coordinate with updates similarly. Consequently, updates are performed in an almost non-blocking way, and lookups can be executed concurrently.

4.5 Range Iteration

In FB$^+$-tree, all the leaf nodes are linked in a totally ordered list. A concurrent scan operation can be performed in two steps: (1) find the starting point on the leaf node list; (2) sequentially iterate on the list. The former is achieved using the *upper_bound* and *lower_bound* functions, and the latter is implemented with a concurrent iterator.

Lazy Rearrangement. Step (1) is akin to a lookup operation except lazy rearrangement. Since maintaining key-values in order is expensive for both lookup and insert, FB$^+$-tree stores unsorted key-values in leaf nodes. The *ordered* bit is embedded in *control* to indicate whether the key-values are in order. When descending to a leaf node, it checks the *ordered* bit. If unordered, it acquires a write lock and then rearranges the key-value pointers, allowing range iteration to benefit from a sequential memory access pattern. Otherwise, it finds the start point without holding the lock. It should be noted that lazy rearrangement incurs a small overhead, because over half of key-values are sorted during node split or merge.

FB$^+$-tree's concurrent iterator in step (2) is almost identical to an STL iterator. In concurrent environments, however, an iterator has to coordinate with insert and remove operations. FB$^+$-tree's iterator thus contains a version to detect whether any modifications occur in a leaf node. If the version changes during iteration, the *ordered* would be checked and the node could be rearranged when necessary. The iterator can thus access the newly inserted key-values.

Cross-Node Tracking. If a node split occurs when the iteration crosses leaf nodes, the successor key-value is determined using binary search. The cross-node tracking technique is employed to detect if any structure modifications occur when crossing nodes during range iteration. If no changes have occurred after proceeding to the sibling node, the iteration could optimistically access the minimum key-value for better performance.

5 EXPERIMENT EVALUATION

In this section, we experimentally evaluate FB$^+$-tree and compare it with other state-of-the-art main-memory index structures.

5.1 Experiment Setup

Platform. We use a Dell PowerEdge R740 Server with two NUMA nodes. Each node contains an Intel Xeon(R) Gold 6248R processor with 24 3.0 GHz cores (with up to 48 hyperthreads). Each processor has 35.75 MB L3 cache and is equipped with 64 GB DDR4-2133 memory. We run Ubuntu 20.04 with kernel version 5.4.0. All our code is implemented with C++ 17 and compiled using GCC/G++ 11.4.0 with O3 optimization level. We use jemalloc to reduce dynamic memory allocation overhead at runtime. Threads are pinned to hardware threads to avoid migrations by the OS scheduler.

Indexes. We compare FB$^+$-tree with seven popular main-memory index structures, including the variants of both B$^+$-tree and trie:

- STX B$^+$-tree[8]: A highly optimized B$^+$-tree container with improved memory fragmentation and cache efficiency [9].
- FAST[9]: A read-only binary search tree that collapses multiple nodes into one large node to facilitate SIMD instructions [32].
- B$^+$-treeOLC[10]: A thread-safe B$^+$-tree implementation synchronized by optimistic lock coupling [67].
- ART: The default index of HyPer [31]. We use two thread-safe implementations ARTOLC[10] and ARTOptiQL[11], synchronized by optimistic lock coupling and OptiQL, respectively [63, 67].
- HOT[12]: The Height Optimized Trie dynamically varies the number of discriminative bits considered in each node[10]. HOT utilizes the ROWEX protocol for synchronization [42, 44].
- Masstree[13]: The Masstree is a trie-like concatenation of B$^+$-tree used by silo [49, 65]. It employs a customized optimistic lock protocol along with the link technique for synchronization.
- Wormhole[14]: The Wormhole substitutes inner nodes of B$^+$-tree with a trie structure and represents it as a hash table. It takes $O(log\ L)$ worst-case time for querying a key of length L [68].

[8]https://github.com/tlx/tlx.git
[9]https://github.com/RyanMarcus/fast64.git
[10]https://github.com/wangziqi2016/index-microbench.git
[11]https://github.com/sfu-dis/optiql
[12]https://github.com/speedskater/hot.git
[13]https://github.com/kohler/masstree-beta.git
[14]https://github.com/wuxb45/wormhole.git

In STX B$^+$-tree, B$^+$-treeOLC, and FB$^+$-tree, integer keys are stored directly in inner nodes, whereas string keys are stored via pointers. In our experiments, all indexes maintain a pointer to each key-value. To avoid excessive compiler optimization, such as unused result optimization, we compile the code of index structures into a shared library. For FB$^+$-tree, the *ns* and *fs* are configured to 64 and 4 respectively, and AVX2 instructions are configured as default. Other indexes are configured to their default configurations. We do not compare against learned indexes, as these hardly support insert operation. We also do not compare against hash tables, because these do not support range iteration.

Workloads. Our experiments are based on the standard workloads from the Yahoo! Cloud Serving Benchmark[15] (YCSB) [20]. We evaluate four core workloads with the default YCSB parameters (requests follow the Zipfian distribution, skew=0.99): LOAD (100% insert), A (50% read, 50% update), C (100% read), and E (95% range-scan, 5% insert). Our main concern is the index structure itself, so range scan only reads pointers without actually accessing the records. Each workload consists of two phases: the load phase inserts 100 million keys in random order into indexes (one percent of keys are inserted for warmup); the run phase executes 100 million operations specified by the workload multiple times. We evaluate each workload on five datasets with different key lengths (Table 2). The *rand-int* consists of random 64-bit integers. The *3-gram*[16] [15] contains unique sequences of triple words often used in language models. We also use the default keys generated by YCSB. The *twitter*[17] (cluster-27) is anonymized data by collecting real-world production traces from in-memory cache clusters at Twitter [69]. The *url* consists of distinct URLs in DBPedia dataset [2].

Table 2: Datasets used in the experiments.

Name	Description	Avg. key size (bytes)
rand-int	64-bit random integers	8.0
3-gram	unique sequences of triple words	15.8
ycsb	keys generated by YCSB	22.9
twitter	anonymized keys in cluster-27	52.7
url	urls in DBPedia dataset	76.8

5.2 Comparison against B$^+$-tree Variants

We first measure the single-threaded performance of B$^+$-tree variants to evaluate the structural optimization of FB$^+$-tree. Figure 11 shows the throughput of FB$^+$-tree and two competitors, STX B$^+$-tree and B$^+$-treeOLC. On all workload-dataset combinations, FB$^+$-tree outperforms the other two competitors—by up to 2.5x (LOAD), 2.9x (YCSB-A), 2.9x (YCSB-C), and 2.2x (YCSB-E). As presented in Section 3.2, the feature comparison technique significantly reduces cache misses and enables memory-level parallelism, leading to superior single-threaded lookup, update, and scan performance. Besides feature comparison, the unordered arrangement of key-values in FB$^+$-tree contributes to efficient insert performance. Since FAST is a read-only structure and only supports integer keys, we compare it to FB$^+$-tree in a separate evaluation, as shown in Figure 1.

[15]https://github.com/brianfrankcooper/YCSB/
[16]https://www.statmt.org/lm-benchmark/
[17]https://github.com/twitter/cache-trace

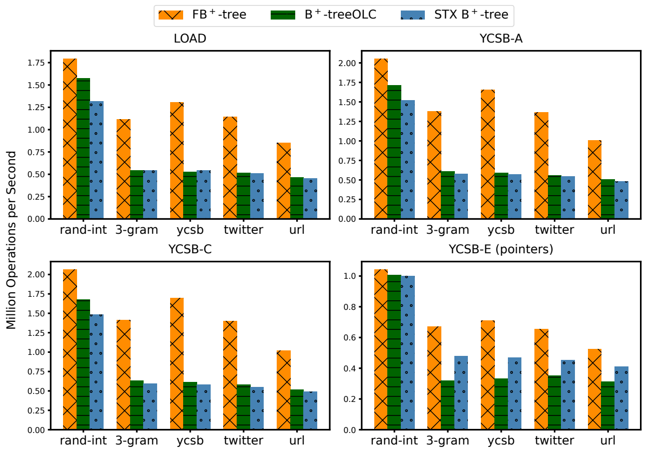

Figure 11: Single-threaded throughput of B$^+$-tree variants.

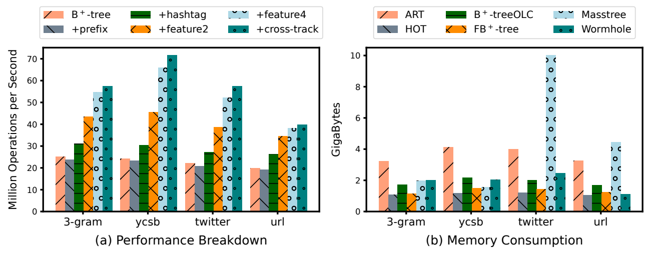

Figure 12: (a) FB$^+$-tree's multi-threaded (48 cores) throughput on workload YCSB-C by gradually enabling the optimizations, and (b) Memory consumption of different indexes.

5.3 Evaluation on Our Optimizations

Factor Analysis on Structural Optimizations. We gradually enable the optimizations to evaluate the multi-threaded throughput of FB$^+$-tree, as shown in Figure 12(a). We initially consider the B$^+$-tree without any optimization, which employs binary search in inner and leaf nodes. Next, we enable prefix (denoted as +prefix) and hashtag (in leaf nodes, called +hashtag) in sequence to evaluate the structural optimizations described in Section 3.3. The +prefix only stores the common prefixes of anchors directly in inner nodes, in which branch operation compares the target key with the common prefix and then performs a binary search on the suffix.

It even decreases the performance, since more cache lines need to be loaded, as discussed in Section 3.1. We then evaluate the effects of the feature comparison technique (i.e., +feature2 and +feature4, by configuring two and four features, respectively), which improves cache utilization and exploits memory-level parallelism. Lastly, we enable the cross-node tracking optimization (denoted as +cross-track), which eliminates the overhead of accessing the *high_key* in leaf nodes, as detailed in Section 4.3. Since some optimizations in Figure 12(a) are not employed for binary keys, we do not evaluate these on rand-int dataset. In conclusion, the multi-threaded (48 cores) throughput of FB$^+$-tree (+cross-track) on YCSB-C is higher than that of typical B$^+$-tree—by 2.3x (*3-gram*), 2.9x (*ycsb*), 2.6x (*twitter*), 2.0x (*url*), and 2.1x (*rand-int*, as shown in Figure 17).

To deeply evaluate the impact of feature size, we also evaluate the multi-threaded (48 cores) performance, average suffix comparison

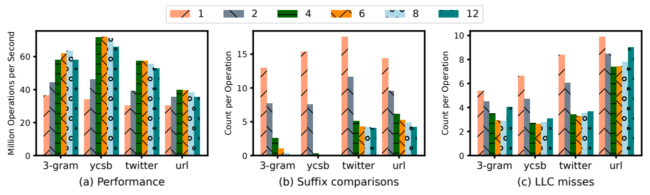
Figure 13: Evaluation with different feature size (YCSB-C).

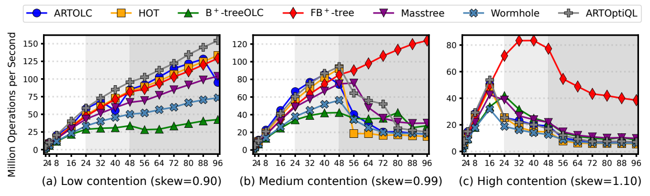
Figure 14: Scalability under different skews (YCSB-A).

count per operation, and average LLC-misses per operation with different feature sizes, as shown in Figure 13. Since the feature size of binary keys is fixed, we do not evaluate these on rand-int dataset. Although the average suffix comparison count per operation gradually declines as feature size increases, the average LLC-misses per operation first decreases and then gradually increases, as shown in Figure 13(b)(c). This occurs because the mechanical selection of anchor keys is unconscious of prefix skewness, as discussed in Section 3.2. As a result, the performance of FB$^+$-tree first increases with feature size, then gradually decreases, as shown in Figure 13(a). In addition, Figure 13 also illustrates the reason why FB$^+$-tree's performance varies across different datasets. The twitter and url datasets have more complicated prefix patterns leading to more suffix comparisons and LLC-misses.

Scalability of Latch-free Update. First, we compare FB$^+$-tree with other state-of-the-art index structures to evaluate their scalability on rand-int dataset using YCSB-A (update-heavy) workload with different skews. The results are shown in Figure 14. All index structures are multi-core scalable under low contention. Thanks to its latch-free update and non-blocking read described in Section 4.1 and Section 4.4, FB$^+$-tree is almost linearly scalable under median contention, while other index structures suffer from performance collapse. Due to retrying CAS on the pointers, FB$^+$-tree also experiences performance collapse under high contention. However, it still maintains the best performance thanks to its minimal hardware-level critical section, as illustrated in Figure 14(c).

Next, we horizontally compare the latch-free update with two other competitors, optimistic lock using CAS primitive (denoted as OptLock) and optimistic lock with backoff (called OptLock-Backoff). The scalability of the three techniques on FB$^+$-tree is illustrated in Figure 15. Due to space limitation, we only evaluate the *rand-int* and *url* datasets (the highest and lowest performance on YCSB-C) with the standard workload YCSB-A (skew=0.99). The OptLock suffers from scalability collapse over 48 threads on *rand-int* dataset and over 64 threads on *url* dataset, respectively. The OptLock with backoff algorithm could mitigate such performance collapse, however, leading to performance degradation with fewer threads. Our latch-free update exhibits the best scalability and outperforms OptLock by 6.6x (*rand-int*) and 2.8x (*url*) under 96 threads.

5.4 Comparison against State-of-the-art

We compare FB$^+$-tree with five state-of-the-art main-memory indexes to evaluate their scalability and performance in a concurrent environment. Meanwhile, we use the statistic interface of jemalloc to evaluate their space efficiency.

Memory Consumption. Since these indexes utilize different key-value storage formats, we only report the index memory consumption (the memory required by the index, including the pointers to key-values but excluding the key-values). The Masstree employs a complicated design, in which key-values are not stored together and key slices are stored in its inner nodes. We count the whole memory footprint and then subtract the memory footprint for storing key-values as Masstree's memory consumption. The results on *3-gram*, *ycsb*, *twitter*, and *url* datasets are presented in Figure 12(b). The results on *rand-int* dataset is consistent with results on *3-gram* dataset. HOT only considers discriminative bits in inner nodes and thus is highly space-efficient. Except for HOT, FB$^+$-tree is more space-efficient than other indexes on almost all datasets.

Performance and Scalability. The throughput and scalability of these indexes on all workload-dataset combinations are illustrated in Figure 17. On workload LOAD, FB$^+$-tree outperforms all other indexes on all datasets. ARTOptiQL and HOT exhibit comparable scalability and performance on *ycsb*, *twitter*, and *url* dataets. Thanks to the latch-free update technique, FB$^+$-tree shows the best scalability on workload YCSB-A. All other indexes suffer from performance collapse as threads increase. On the read-only workload YCSB-C, HOT gives the best performance and scalability on all datasets except rand-int. FB$^+$-tree performs almost as fast as other trie-based structures. As shown in Figure 16, the average LLC-misses and branch misses per operation of all index structures are counted under a multi-threaded (48-core) environment as a shred of evidence. Except for these two metrics, parallelism between instructions, memory access patterns, and memory bandwidth utilization are also significant, which leads to FB$^+$-tree's better performance than

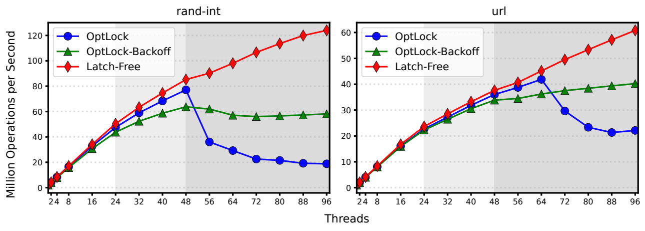
Figure 15: Scalability on rand-int and url datasets (YCSB-A).

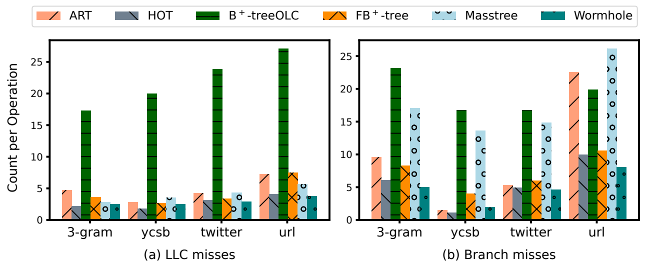
Figure 16: Hardware events count on workload YCSB-C.

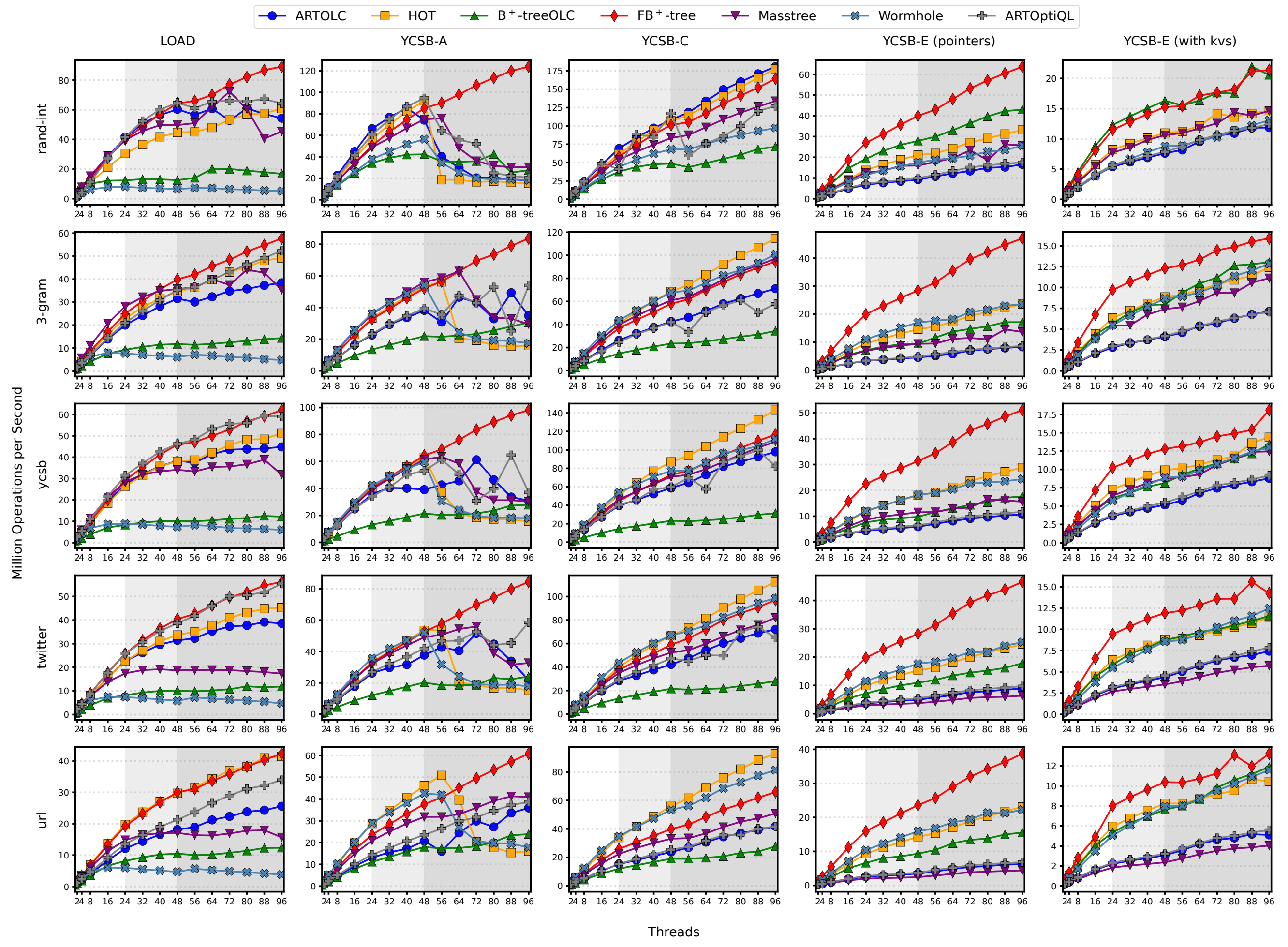

Figure 17: Index throughput and scalability on different workload-dataset combinations.

ART and Mastree. It should be mentioned that all these structures perform lookup without holding any locks.

The performance of range scan may be vitally important in many database applications, especially when secondary indexes are extensively utilized to retrieve valuable information and translate database operators into computations based on primary keys. In these scenarios, a balanced structure may be extremely effective. FB$^+$-tree exhibits superior performance than other structures on workload YCSB-E thanks to its balanced structure and sequential pointer arrangement as described in Section 4.5. Although Wormhole has a similar leaf node structure, its indirect ordered key-value arrangement hinders efficient range scan. Frequent pointer chasing in trie-based index structures leads to inferior range scan performance. For completeness, we also illustrate the range scan performance with the actual key-value records access. In summary, the experiments demonstrate that FB$^+$-tree dominates existing B$^+$-tree variants on all the workload-dataset combinations. Compared with trie-based structures, FB$^+$-tree may have lower performance on point lookup, while it has a big advantage on range scan. On update-heavy workloads, FB$^+$-tree also demonstrates significant potential and better scalability over other structures.

6 CONCLUSION

In this paper, we present the FB$^+$-tree, a fast, cache-efficient, and balanced B$^+$-tree variant taking memory-level and computational parallelism into consideration. We show how to reduce cache misses in binary search and exploit prefetching to leverage memory-level parallelism. We highlight our feature comparison technique enabling vectorization of binary search from a different perspective than previous work. The evaluation results demonstrate FB$^+$-tree exhibits comparable point lookup performance to state-of-the-art indexes, while exhibiting superior range scan performance. We sincerely believe that mitigating dependences and exploring the possibility of parallelization and vectorization would be increasingly important to improve the performance of existing algorithms.

ACKNOWLEDGMENTS

This research was sponsored by the National Key Research and Development Program of China (No. 2023YFC3321304), the National Science Foundation of China (No. 61572373), and CCF-Huawei Populus Grove Fund (No. CCF-HuaweiDB202410). We thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] Nikolas Askitis and Ranjan Sinha. 2007. HAT-Trie: A Cache-Conscious Trie-Based Data Structure For Strings. In *ACSC (CRPIT)*, Vol. 62. Australian Computer Society, 97–105.

[2] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. 2007. DBpedia: A Nucleus for a Web of Open Data. In *ISWC/ASWC (Lecture Notes in Computer Science)*, Vol. 4825. Springer, 722–735.

[3] Rudolf Bayer and Mario Schkolnick. 1977. Concurrency of Operations on B-Trees. *Acta Informatica* 9 (1977), 1–21.

[4] Rudolf Bayer and Karl Unterauer. 1977. Prefix B-Trees. *ACM Trans. Database Syst.* 2, 1 (1977), 11–26.

[5] Michael A. Bender, Roozbeh Ebrahimi, Haodong Hu, and Bradley C. Kuszmaul. 2016. B-Trees and Cache-Oblivious B-Trees with Different-Sized Atomic Keys. *ACM Trans. Database Syst.* 41, 3 (2016), 19:1–19:33.

[6] Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. 2006. Cache-oblivious string B-trees. In *PODS*. ACM, 233–242.

[7] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. 2005. Concurrent cache-oblivious b-trees. In *SPAA*. ACM, 228–237.

[8] Michael A. Bender, Haodong Hu, and Bradley C. Kuszmaul. 2010. Performance guarantees for B-trees with different-sized atomic keys. In *PODS*. ACM, 305–316.

[9] Timo Bingmann. 2018. TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers. https://panthema.net/tlx, retrieved Oct. 7, 2020.

[10] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *SIGMOD Conference*. ACM, 521–534.

[11] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2022. Height Optimized Tries. *ACM Trans. Database Syst.* 47, 1 (2022), 3:1–3:46.

[12] Philip Bohannon, Peter McIlroy, and Rajeev Rastogi. 2001. Main-Memory Index Structures with Fixed-Size Partial Keys. In *SIGMOD Conference*. ACM, 163–174.

[13] Matthias Böhm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. 2011. Efficient In-Memory Indexing with Generalized Prefix Trees. In *BTW (LNI)*, Vol. P-180. GI, 227–246.

[14] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *VLDB*. Morgan Kaufmann, 181–190.

[15] Ciprian Chelba, Tomás Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. 2014. One billion word benchmark for measuring progress in statistical language modeling. In *INTERSPEECH*. ISCA, 2635–2639.

[16] Guanduo Chen, Meng Li, Siqiang Luo, and Zhenying He. 2024. Oasis: An Optimal Disjoint Segmented Learned Range Filter. *Proc. VLDB Endow.* 17, 8 (2024), 1911–1924.

[17] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. 2001. Improving Index Performance through Prefetching. In *SIGMOD Conference*. ACM, 235–246.

[18] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. 2002. Fractal prefetching B±Trees: optimizing both cache and disk performance. In *SIGMOD Conference*. ACM, 157–168.

[19] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137.

[20] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. ACM, 143–154.

[21] Intel Corporation. 2023. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* https://cdrdv2.intel.com/v1/dl/getContent/779559?fileName=355308-Software-Optimization-Manual-048-Changes-Doc-2.pdf

[22] Intel Corporation. 2023. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* https://cdrdv2.intel.com/v1/dl/getContent/789583?fileName=325462-sdm-vol-1-2abcd-3abcd-4.pdf

[23] 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 *SIGMOD Conference*. ACM, 969–984.

[24] Ronald G. Dreslinski, Ali G. Saidi, Trevor N. Mudge, and Steven K. Reinhardt. 2007. Analysis of hardware prefetching across virtual page boundaries. In *Conf. Computing Frontiers*. ACM, 13–22.

[25] Babak Falsafi and Thomas F. Wenisch. 2014. *A Primer on Hardware Prefetching.* Morgan & Claypool Publishers.

[26] Goetz Graefe, Hideaki Kimura, and Harumi A. Kuno. 2012. Foster b-trees. *ACM Trans. Database Syst.* 37, 3 (2012), 17:1–17:29.

[27] Goetz Graefe and Per-Åke Larson. 2001. B-Tree Indexes and CPU Caches. In *ICDE*. IEEE Computer Society, 349–358.

[28] Rachid Guerraoui and Vasileios Trigonakis. 2016. Optimistic concurrency with OPTIK. In *PPoPP*. ACM, 18:1–18:12.

[29] Steffen Heinz, Justin Zobel, and Hugh E. Williams. 2002. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.* 20, 2 (2002), 192–223.

[30] Tim Kaldewey, Jeff Hagen, Andrea Di Blas, and Eric Sedlar. 2009. Parallel search on video cards. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism (HotPar'09)*. USENIX Association, 9.

[31] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. IEEE Computer Society, 195–206.

[32] 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 *SIGMOD Conference*. ACM, 339–350.

[33] Minsu Kim, Jinwoo Hwang, Guseul Heo, Seiyeon Cho, Divya Mahajan, and Jongse Park. 2024. Accelerating String-key Learned Index Structures via Memoization-based Incremental Training. *Proc. VLDB Endow.* 17, 8 (2024), 1802–1815.

[34] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *aiDM@SIGMOD*. ACM, 5:1–5:5.

[35] Yusuf Onur Koçberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin T. Lim, and Parthasarathy Ranganathan. 2013. Meet the walkers: accelerating index traversals for in-memory databases. In *MICRO*. ACM, 468–479.

[36] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD Conference*. ACM, 489–504.

[37] 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.

[38] Vladimir Lanin and Dennis E. Shasha. 1986. A Symmetric Concurrent B-Tree Algorithm. In *FJCC*. IEEE Computer Society, 380–389.

[39] Jaekyu Lee, Hyesoon Kim, and Richard W. Vuduc. 2012. When Prefetching Works, When It Doesn't, and Why. *ACM Trans. Archit. Code Optim.* 9, 1 (2012), 2:1–2:29.

[40] 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.

[41] Tobin J. Lehman and Michael J. Carey. 1986. A Study of Index Structures for Main Memory Database Management Systems. In *VLDB*. Morgan Kaufmann, 294–303.

[42] 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.

[43] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. IEEE Computer Society, 38–49.

[44] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *DaMoN*. ACM, 3:1–3:8.

[45] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*. IEEE Computer Society, 302–313.

[46] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. AI Meets Database: AI4DB and DB4AI. In *SIGMOD Conference*. ACM, 2859–2866.

[47] 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.

[48] 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.

[49] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *EuroSys*. ACM, 183–196.

[50] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (1991), 21–65.

[51] Donald R. Morrison. 1968. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (1968), 514–534.

[52] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. 1995. AlphaSort: A Cache-Sensitive Parallel External Sort. *VLDB J.* 4, 4 (1995), 603–627.

[53] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD Conference*. ACM, 371–386.

[54] William W. Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676.

[55] Jun Rao and Kenneth A. Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB*. Morgan Kaufmann, 78–89.

[56] Jun Rao and Kenneth A. Ross. 2000. Making B^+-Trees Cache Conscious in Main Memory. In *SIGMOD Conference*. ACM, 475–486.

[57] Steve Scargall. 2020. *Programming Persistent Memory: A Comprehensive Guide for Developers.* Springer Nature.

[58] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. 2009. k-ary search on modern processors. In *DaMoN*. ACM, 52–60.

[59] Josef Schmeißer, Maximilian E. Schüle, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2021. B^2-Tree: Cache-Friendly String Indexing within B-Trees. In *BTW (LNI)*, Vol. P-311. Gesellschaft für Informatik, Bonn, 39–58.

[60] 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.

[61] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2016. A Hybrid B+-tree as Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. In *SIGMOD Conference*. ACM, 1523–1538.

[62] Tomer Shanny and Adam Morrison. 2022. Occualizer: Optimistic Concurrent Search Trees From Sequential Code. In *OSDI*. USENIX Association, 321–337.

[63] Ge Shi, Ziyi Yan, and Tianzheng Wang. 2023. OptiQL: Robust Optimistic Locking for Memory-Optimized Indexes. *Proc. ACM Manag. Data* 1, 3 (2023), 216:1–216:26.

[64] Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. 2021. Bounding the Last Mile: Efficient Learned String Indexing. *CoRR* abs/2111.14905 (2021).

[65] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*. ACM, 18–32.

[66] Zhenlin Wang, Doug Burger, Steven K. Reinhardt, Kathryn S. McKinley, and Charles C. Weems. 2003. Guided Region Prefetching: A Cooperative Hardware/-Software Approach. In *ISCA*. IEEE Computer Society, 388–398.

[67] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD Conference*. ACM, 473–488.

[68] Xingbo Wu, Fan Ni, and Song Jiang. 2019. Wormhole: A Fast Ordered Index for In-memory Data Management. In *EuroSys*. ACM, 18:1–18:16.

[69] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *OSDI*. USENIX Association, 191–208.

[70] Yifan Yang and Shimin Chen. 2024. LITS: An Optimized Learned Index for Strings. *Proc. VLDB Endow.* 17, 11 (2024), 3415–3427.

[71] Adar Zeitak and Adam Morrison. 2021. Cuckoo Trie: Exploiting Memory-Level Parallelism for Efficient DRAM Indexing. In *SOSP*. ACM, 147–162.

[72] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *SIGMOD Conference*. ACM, 1567–1581.

[73] Jiaoyi Zhang, Kai Su, and Huanchen Zhang. 2024. Making In-Memory Learned Indexes Efficient on Disk. *Proc. ACM Manag. Data* 2, 3 (2024), 151.

[74] 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.

[75] Weihua Zhang, Chuanlei Zhao, Lu Peng, Yuzhe Lin, Fengzhe Zhang, and Yunping Lu. 2023. Boosting Performance and QoS for Concurrent GPU B+trees by Combining-Based Synchronization. In *PPoPP*. ACM, 1–13.

[76] Rong Zhu, Lianggui Weng, Wenqing Wei, Di Wu, Jiazhen Peng, Yifan Wang, Bolin Ding, Defu Lian, Bolong Zheng, and Jingren Zhou. 2024. PilotScope: Steering Databases with Machine Learning Drivers. *Proc. VLDB Endow.* 17, 5 (2024), 980–993.