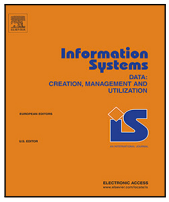
Information Systems 119 (2023) 102287

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is



DB$^+$-tree: A new variant of B$^+$-tree for main-memory database systems

Yongsik Kwon [a,b], Seonho Lee [b], Yehyun Nam [b], Joong Chae Na [c], Kunsoo Park [b], Sang K. Cha [b], Bongki Moon [b,*]

[a] SAP Labs Korea, 235 Banpo-daero, Seocho-gu, Seoul, 06578, South Korea
[b] Seoul National University, 1 Gwanak-ro, Gwanak-gu, Seoul, 08826, South Korea
[c] Sejong University, 209 Neungdong-ro, Gwangjin-gu, Seoul, 05006, South Korea

ARTICLE INFO

Keywords:
B$^+$-tree
Branching algorithm
Point search
Range search
Update operations
Distinction bit slice
Trie

ABSTRACT

The B-tree and its variants are an indispensable tool for database systems and applications. Hence the efficiency of the B-tree is one of the few critical factors that determine the performance of a database system. In main-memory database systems, the computational overhead intrinsic in the B-tree algorithms for branching becomes the dominant factor in performance. In this paper, we propose yet another but disruptive variant of the B$^+$-tree called the DB$^+$-tree that redesigns the node structure for faster branching operations. The novel branching algorithm of the DB$^+$-tree can be implemented in an $O(1)$ number of SIMD and other sequential instructions, which supports fast branching, and this leads to efficient point search, range search, and update operations.

1. Introduction

Since it was invented by Bayer and McCreight [1] a half century ago, the B-tree and its variants have been an indispensable tool for database systems and applications. Hence the efficiency of the B-tree is one of the few critical factors that determine the performance of a database system that relies on it for indexing and data access methods. The traditional optimization methods are centered around maintaining the highest possible fanout of the B-tree so that the number of I/O operations required per database operation can be minimized. Examples include finding the optional node size, compressing index entries by adopting prefix or partial keys, and reducing the latency of I/O operations by bulky read and write operations. For a database system that manages all or most of the data objects in main memory, however, we have come to realize that the computational overhead intrinsic in the B-tree algorithms becomes the dominant factor in performance. To demonstrate the point, we have profiled the runtime of the B$^+$-tree (the most popular variant of the B-tree) for point search operations. The result of profiling is shown in Fig. 1, where *branching* means the cumulative cost of finding the correct child of an internal node, and *node access* means the cumulative cost of fetching the correct child node. When the index keys are short (8B), about 40% of search time is spent on branching. The cost of branching grows even larger for long keys (128B). The trend is still the same for the partial key B-tree (pkB-tree) [2], which is known for its efficient branching for long keys.

In this paper, we propose yet another but disruptive variant of the B$^+$-tree that redesigns the node structure for faster branching operations. In our index called the DB$^+$-tree (which is short for D-bit B$^+$-tree), we store partial information of the keys in a node, which is called the *distinction bit slice* (*D-bit slice* for short), which leads to efficient search and update operations. More specifically, our contributions are as follows.

- Unlike the B$^+$-tree (or its variants), the branching algorithm of the DB$^+$-tree based on D-bit slices does not have loops, but it consists of an $O(1)$ number of SIMD and other sequential instructions, irrespective of the key length. Also, our branching algorithm uses key pointer dereferencing less frequently, especially when compared to the pkB-tree. As shown in Fig. 1, the branching time of the DB$^+$-tree is significantly smaller than those of the other B$^+$-tree variants, which leads to fast point search. Specifically, the DB$^+$-tree was faster for point searches than the B$^+$-tree up to 350% (on average 230%), and faster than the pkB-tree up to 280% (on average 170%) in our experiments.
- In range search, the D-bit information enables us to determine whether all the keys in a leaf node are within the search range by a simple test. For range searches, the DB$^+$-tree outperformed the B$^+$-tree by up to 670% (on average 290%) and the pkB-tree by up to 170% (on average 130%).
- In the D-bit slices, some bits of the keys can remain *unspecified*, which allows us to perform local changes in the D-bit information,

* Corresponding author.
E-mail addresses: yong.sik.kwon@sap.com (Y. Kwon), shlee2@theory.snu.ac.kr (S. Lee), yhnam@theory.snu.ac.kr (Y. Nam), jcna@sejong.ac.kr (J.C. Na), kpark@theory.snu.ac.kr (K. Park), chask@snu.ac.kr (S.K. Cha), bkmoon@snu.ac.kr (B. Moon).

https://doi.org/10.1016/j.is.2023.102287
Received 17 March 2022; Received in revised form 17 August 2023; Accepted 18 September 2023
Available online 21 September 2023

Frequently used notations

M	Maximum number of keys in a node
K_i	Key stored in a node
D-bit(α, β)	Distinction bit position of α and β
D_i	Distinction bit position of K_{i-1} and K_i
D_{min}	Minimum value of D_i's
$\mathcal{D}$	Position set including all D_i's
DS_i, $p\text{DS}_i$	D-bit slice and partial D-bit slice of K_i for $\mathcal{D}$
DS(Q)	D-bit slice of Q for $\mathcal{D}$

when there are insert or delete operations of keys. For insertions and deletions, the DB$^+$-tree was faster than the B$^+$-tree up to 350% (on average 250%), and faster than the pkB-tree up to 160% (on average 140%).

Organization. The rest of the paper is organized as follows. Section 2 gives basic definitions and related work. Section 3 presents an overview of the DB$^+$-tree. Section 4 describes a novel branching algorithm for the DB$^+$-tree. Section 5 presents search and update operations based on the branching algorithm. Section 6 presents the results of performance evaluation, and we conclude in Section 7.

2. Preliminaries

The properties of the B$^+$-tree that are needed to describe our index are as follows.

1. Each node x has $x.n$ sorted keys $x.K_1, x.K_2, \dots, x.K_{x.n}$, where $x.n \le M$, the maximum number of keys in a node. For simplicity of description, we denote by $x.K_0$ the largest key in the left sibling of x.
2. Each internal node x has $x.n$ pointers $x.C_1, x.C_2, \dots, x.C_{x.n}$ to its children. Any key k in the subtree rooted at C_i satisfies $x.K_{i-1} < k \le x.K_i$ $(1 \le i \le x.n)$, and the largest key $C_i.K_{C_i.n}$ in C_i is equal to $x.K_i$.

We omit node x in notations if it is not confusing.

We consider key values as binary strings throughout this paper. The *distinction bit position*[1] of two binary strings S_1 and S_2, which is denoted by D-bit(S_1, S_2), is defined as the most significant bit position where the two strings differ [3]. Bit positions start with 0, and the bit of a binary string S in bit position i, denoted by $S[i]$, will be called the $(i+1)$st bit of the key (i.e., the bit in bit position 0 is the first bit, the bit in bit position 1 is the second bit, etc.). The first bit (i.e., in bit position 0) is the most significant bit in the keys.

2.1. Related work

The length of keys has a direct impact on the performance of the B-tree, because the fanout is inversely proportional to that. In a main-memory database system, the index keys and their corresponding data records are kept together in the main memory. This presents new opportunities for addressing the problem of indexing long keys. The indirect-key approach [7] eliminates the duplicate keys by storing a pointer to the data key instead of the index key in the B-tree. Although this approach makes the index entries fixed-length, it increases the ratio of cache misses. The prefix-key approach [8] compresses the long keys by retaining the minimal prefixes required for comparison. This approach can make the keys shorter but the prefixes will be variable-length and are subject to frequent recomputation by insertions and deletions. In the partial-key approach [2], an index entry stores the offset of the first distinctive bit and a fixed number of *consecutive* bits from the offset as well as the pointer to the data record. Thus, the index entries can be fixed-length and the B-tree can be more memory efficient. However, if a comparison cannot be resolved by the partial keys, dereferencing the pointer is required.

The B-tree has also been studied heavily for its architectural adaptation. Rao and Ross's CSS-tree [9] and CSB$^+$-tree [10] are cache-conscious variants of the B-tree, and Chen et al.'s pB$^+$-tree [11] and fpB$^+$-tree [12] optimize the performance of the B-tree by prefetching. Zhang et al. parallelize the B-tree operations with SIMD and GPU [13, 14]. Levandoski et al.'s Bw-tree [15] and Na et al.'s IPL B-tree [16] are proposed as flash-aware B-tree variants. Kim et al.'s FAST [17] and Yamamuro et al.'s VAST-tree [18] are a binary tree optimized for the features of the modern architecture such as page size, cache line size, and SIMD width.

The trie and its variants [19,20] are traditionally considered in-memory search structures, but in main-memory database systems they are viable options for indexes. Recently, many trie-variant indexes have been proposed such as Boehm et al.'s Generalized Prefix Tree [21], Kissinger et al.'s KISS-tree [22], Leis et al.'s Adaptive Radix Tree (ART) [23,24], Zhang et al.'s SuRF [25], and Binna et al.'s Height Optimized Trie (HOT) [6]. ART is a trie where the size of each node is optimized adaptively, and HOT combines multiple nodes of a binary trie into compound nodes such that the height of the index is optimized. In particular, HOT is a seminal work in main-memory indexes which not only introduces the layout of partial keys in a node but also outperforms other state-of-the-art indexes.

There has been research in the theory community [4,5,26,27] to improve $O(\log N)$ time bound for point search in an index, where N is the total number of keys in the index. The van Emde Boas tree [26,27] takes $O(\log\log|U|)$ time for point search, where U is the universe of all possible keys, and it requires $O(|U|)$ space. The fusion tree [4,5] is a B$^+$-tree in which branching in a node takes $O(1)$ time. Since the fusion tree is a B$^+$-tree, its height is $O(\log_M N)$. By choosing $M = \log_2 N$, the fusion tree obtains $O(\log N / \log\log N)$ time for point search. However, each node of the fusion tree requires a lookup table of size M^2 to get $O(1)$ branching time, and updating the lookup table in a node takes $O(M^4)$ time.

3. Overview of new B$^+$-tree

The DB$^+$-tree is a variant of the B$^+$-tree. The tree structure of the DB$^+$-tree is the same as that of the B$^+$-tree; only the information about keys inside a node is different for fast branching. A novel branching algorithm using this information, which can be implemented in an $O(1)$ number of SIMD and other sequential instructions, is the key ingredient of the DB$^+$-tree.

Suppose that n sorted keys $K_1, \dots, K_n$ are stored in an internal node. Let K_0 be the largest key in the left sibling of the internal node. If a query key Q arrives at the internal node during a search operation, it means that Q satisfies $K_0 < Q \le K_n$.

D-bit slice. For fast branching, each node x of the DB$^+$-tree contains auxiliary information called the D-bit slices. Let

$$D_i = \text{D-bit}(K_{i-1}, K_i) \text{ for } 1 \le i \le n,$$

i.e., the distinction bit position of two adjacent keys in sorted order. We call D_i's the *D-bit positions* of node x. Let D be an integer set including all the D-bit positions of x. (Note that D may include some *dummy* positions which are not the D-bit positions of x.) Let DS_i be the concatenation of the bits of K_i at the positions in D, which is called the *distinction bit slice* (or *D-bit slice*) of K_i for D.

[1] Ferguson [3] used the term *distinction bit*; Fredman and Willard [4,5] *distinguishing bit*; Binna et al. [6] *discriminative bit*.

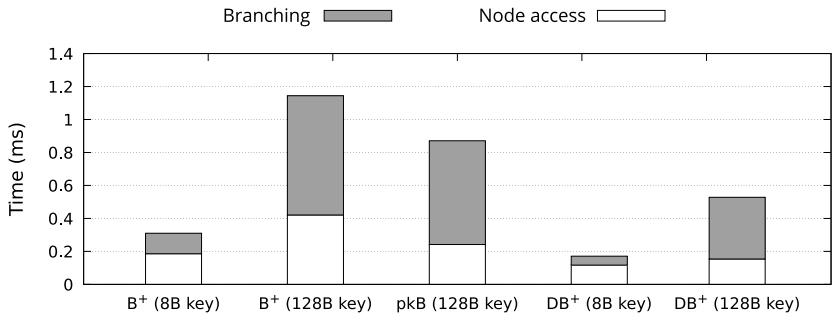


Fig. 1. Branching and node access in point search operations.

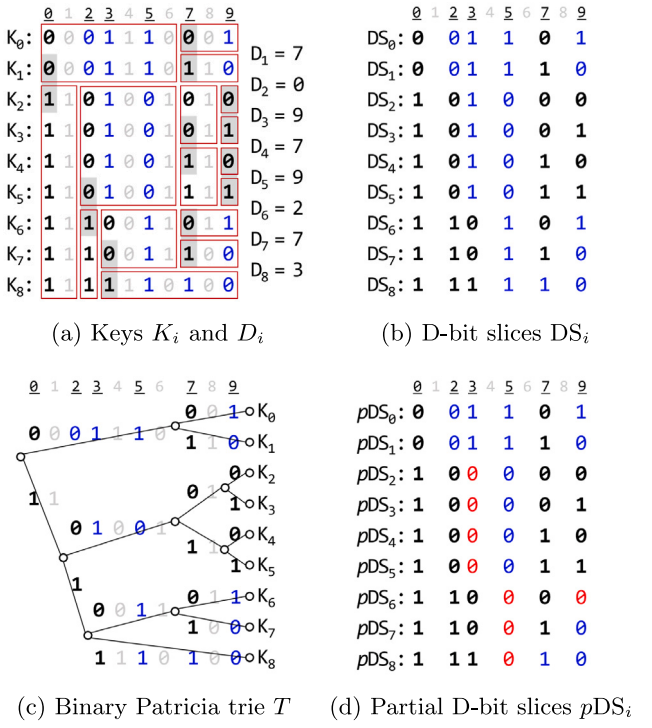


Fig. 2. Information on keys stored in a node when $D = \{0, 2, 3, 5, 7, 9\}$. Bold black bits represent bits at branching positions of each key, and blue bits or red bits represent bits at non-branching positions of D. Red bits in (d) represent unknown bits. In (a), a red rectangle indicates substrings of keys representing the label of an identical edge in the binary Patricia trie T of (c), and a gray box indicates the distinction bit position of two adjacent keys. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Example 3.1. In Fig. 2(a), the D-bit positions are 0, 2, 3, 7, and 9. When $D = \{0, 2, 3, 5, 7, 9\}$, therefore, position 5 is a dummy position. In Fig. 2(b), DS_3 is 101 001, which is the concatenation of the bits of K_3 at the positions in D.

Dummy positions make it possible to update D and D-bit slices lazily. Suppose that K_8 is deleted in Fig. 2(a). Then position 3 is not a D-bit position any more but a dummy position. By allowing dummy positions, we can keep D as it is and thus we do not need to modify D-bit slices other than DS_8, which is deleted. When D has too many dummy positions, dummy positions are eliminated from D, and D-bit slices are updated accordingly. We will describe the details in Section 5.

In addition to the properties of the B+-tree in Section 2, the DB+-tree has the following property:

- Each node x has a position set $x.D$ including all the D-bit positions of x, and the D-bit slices for $x.D$ (i.e., $x.DS_1, x.DS_2, \ldots, x.DS_n$).

The D-bit slices are closely related to the binary Patricia trie T representing the keys [20]. Fig. 2(c) shows the binary Patricia trie of the keys in Fig. 2(a). A rectangle box in Fig. 2(a) indicates substrings of keys representing the label of an identical edge in T. For each key K_i, a position p is called a *branching position* if there exists a key K_j $(i \neq j)$ such that D-bit$(K_i, K_j) = p$, and it is called a *non-branching position* otherwise. The bits in the branching positions of K_i are used for branching when traversing down T with K_i. In Fig. 2, the branching positions of K_3 are $\{0, 2, 7, 9\}$ and the other positions are the non-branching positions of K_3. Note that all the branching positions of a key K_i are included in D and thus the D-bit slices contain sufficient information necessary for branching. We will describe a branching algorithm using the D-bit slices in Section 4.

Search and update operations (i.e., point search, range search, insertion, and deletion) in the DB+-tree which are based on the branching algorithm will be presented in Section 5.

4. Branching algorithm

In this section, we show how to solve the branching problem in a node x.

Branching problem. Given the sorted keys $K_0, K_1, \ldots, K_n$ in node x and a query key Q $(K_0 < Q \leq K_n)$, find b such that $K_{b-1} < Q \leq K_b$.

4.1. Basic algorithm

The branching problem can be solved by finding the longest prefix Q' of query key Q that matches a path from the root in T, the binary Patricia trie of the keys in node x. Q' can be found by computing $\text{lcp}(Q, K_i)$ $(0 \leq i \leq n)$ and taking the maximum of them, where $\text{lcp}(\alpha, \beta)$ denotes the length of the longest common prefix of α and β. When keys are long, however, it is expensive to compute $\text{lcp}(Q, K_i)$ for all keys.

Our branching algorithm uses the D-bit information to find Q' and then find keys whose prefixes are Q'. The following is our basic branching algorithm.

1. Find q such that $\text{lcp}(\text{DS}(Q), \text{DS}_q)$ is maximum for $1 \leq q \leq n$, where $\text{DS}(Q)$ is the D-bit slice of Q for D. (If there are many such q's, the algorithm works with any q.)
2. Find $D \leftarrow \text{D-bit}(Q, K_q)$ by comparing Q and K_q.
3. Find b such that $K_{b-1} < Q \leq K_b$ as follows. If $Q = K_q$, b is q; If $Q > K_q$, b is the smallest integer $> q$ such that $D_b \leq D$; If $Q < K_q$, b is the largest integer $\leq q$ such that $D_b \leq D$.

Since the D-bit slices contain bits at all the branching positions, Step 1 finds a key K_i such that $\text{lcp}(Q, K_i)$ is maximum, which is proven in Theorem 4.1.

Example 4.1. Given the keys and the D-bit slices in Fig. 2, for each query Q below, the basic algorithm works as follows.

(a) Q = 11011 00010 (i.e., DS(Q) = 101000). Then $q = 2$ in Step 1, $D = 4$ in Step 2 (i.e., $Q' = 1101$), and $b = 6$ in Step 3. Note that $K_2, \ldots, K_5$ have $Q' = 1101$ as their prefixes, and $K_5 < Q \le K_6$ (i.e., $b = 6$ means the end of the keys having $Q' = 1101$ as their prefixes).
(b) Q = 10100 10111 (i.e., DS(Q) = 110111). Then $q = 7$, $D = 1$, $b = 2$, and thus $K_1 < Q \le K_2$.
(c) Q = 00011 10011 (i.e., DS(Q) = 001101). Then $q = 1$ (due to the condition $q \ge 1$), $D = 7$, $b = 1$, and thus $K_0 < Q \le K_1$.

The following lemmas and theorems show that the basic algorithm is correct.

Lemma 4.1. *D-bit*$(K_i, K_j) = \min_{i<k\le j} D_k$ *for all* $0 \le i < j \le n$.

Proof. It follows from the definition of D-bit. □

Lemma 4.2. *Let* x_1 *and* x_2 *be bit strings such that* $x_1 < x_2$ *and D-bit*$(x_1, x_2) = c$.

(1) If D-bit$(y, x_1) > c$, *then* $y < x_2$ *and D-bit*$(y, x_2) = c$.
(2) If D-bit$(y, x_2) > c$, *then* $y > x_1$ *and D-bit*$(y, x_1) = c$.

Proof. Since $x_1 < x_2$ and D-bit$(x_1, x_2) = c$, the $(c+1)$st bit of x_1 is 0, and that of x_2 is 1.

(1) If D-bit$(y, x_1) > c$, the $(c+1)$st bit of y is 0, and thus $y < x_2$ and D-bit$(y, x_2) = c$.
(2) If D-bit$(y, x_2) > c$, the $(c+1)$st bit of y is 1, and thus $y > x_1$ and D-bit$(y, x_1) = c$. □

Theorem 4.1. *The integer* q *found in Step 1 of the basic algorithm satisfies that D-bit*(Q, K_q) *is maximum for* $1 \le q \le n$.

Proof. Let d be the first bit position where Q and K_q differ, i.e., D-bit$(Q, K_q) = d$. We show that it is maximum as follows. Consider any key K_i $(1 \le i \le n, i \ne q)$.

- If D-bit$(K_i, K_q) < d$, then D-bit$(Q, K_i) < d$ because the first d bits of Q are the same as those of K_q.
- If D-bit$(K_i, K_q) > d$, then D-bit$(Q, K_i) = d$ because the $(d+1)$st bit of K_i is the same as that of K_q, which is different from that of Q.
- D-bit(K_i, K_q) cannot equal d, which can be proved by contradiction. Suppose that D-bit$(K_i, K_q) = d$. Then, d is a distinction bit position by Lemma 4.1 and thus it is part of the distinction bit slices. Let l be the position in DS(Q) corresponding to the position d in Q. Since D-bit$(Q, K_q) = d$ and D-bit$(K_i, K_q) = d$, the first l bits of DS(Q), DS$_q$, and DS$_i$ are the same, and we have DS(Q)$[l] \ne$ DS$_q[l]$ and DS$_q[l] \ne$ DS$_i[l]$. Therefore DS(Q)$[l] =$ DS$_i[l]$. That is, lcp(DS(Q), DS$_i$) > lcp(DS(Q), DS$_q$), which contradicts the fact that lcp(DS(Q), DS$_q$) is maximum.

Hence, D-bit$(Q, K_i) \le d$ for all $1 \le i \le n$. □

Theorem 4.2. *Step 3 of the basic algorithm finds* b *such that* $K_{b-1} < Q \le K_b$. *In addition, (1) when* $Q > K_q$, *D-bit*$(K_{b-1}, Q) = D$ *and D-bit*$(Q, K_b) = D_b$; *(2) when* $Q < K_q$ *and* $b > 1$, *D-bit*$(K_{b-1}, Q) = D_b$ *and D-bit*$(Q, K_b) = D$.

Proof. There are three cases in Step 3. If $Q = K_q$, the theorem holds trivially.

When $Q > K_q$, we show that the integer b in Step 3 satisfies $K_{b-1} < Q \le K_b$, and D-bit$(K_{b-1}, Q) = D$ and D-bit$(Q, K_b) = D_b$.

- For $b-1$, D-bit$(K_q, K_{b-1}) > D =$ D-bit(Q, K_q) because D-bit$(K_q, K_{b-1}) = \min_{q<k<b} D_k$ by Lemma 4.1 and $D_k > D$ $(q < k < b)$ by definition of b. Hence $K_{b-1} < Q$ and D-bit$(K_{b-1}, Q) = D$ by Lemma 4.2(1) because K_q, Q, K_{b-1} correspond to x_1, x_2, y, respectively (e.g., when Q = 11011 00010, $q = 2$, $b - 1 = 5$ in Example 4.1a).
- For b, D_b cannot equal D. (Suppose that $D_b = D$. The $(D+1)$st bit of K_q is 0, that of Q is 1, and that of K_b must be 1. Thus D-bit$(Q, K_b) >$ D-bit(Q, K_q), a contradiction to Theorem 4.1.) Since $D_b < D$, D-bit$(K_q, K_b) = \min_{q<k\le b} D_k = D_b <$ D-bit$(Q, K_q) = D$. Hence $Q < K_b$ and D-bit$(Q, K_b) = D_b$ by Lemma 4.2(1) because K_q, K_b, Q correspond to x_1, x_2, y, respectively (e.g., when Q = 11011 00010, $q = 2$, $b = 6$ in Example 4.1a).

When $Q < K_q$, the integer b in Step 3 satisfies $K_{b-1} < Q \le K_b$, and D-bit$(K_{b-1}, Q) = D_b$ (if $b - 1 > 0$) and D-bit$(Q, K_b) = D$ as follows.

- For b, D-bit$(K_b, K_q) = \min_{b<k\le q} D_k > D =$ D-bit(Q, K_q). Therefore $K_b > Q$ and D-bit$(K_b, Q) = D$ by Lemma 4.2(2) because Q, K_q, K_b correspond to x_1, x_2, y, respectively (e.g., when Q = 10100 10111, $q = 7$, $b = 2$ in Example 4.1b).
- For $b-1$, we show that $Q > K_{b-1}$. If $b - 1 = 0$, it holds trivially since $Q > K_0$ by the precondition of the branching algorithm. Otherwise, D_b cannot equal D. (Suppose that $D_b = D$. The $(D+1)$st bit of Q is 0, that of K_q is 1, and that of K_{b-1} must be 0. Thus D-bit$(Q, K_{b-1}) >$ D-bit(Q, K_q), a contradiction to Theorem 4.1.) Since $D_b < D$, D-bit$(K_{b-1}, K_q) = \min_{b\le k\le q} D_k = D_b <$ D-bit$(Q, K_q) = D$. Hence $Q > K_{b-1}$ and D-bit$(Q, K_{b-1}) = D_b$ by Lemma 4.2(2) because K_{b-1}, K_q, Q correspond to x_1, x_2, y, respectively (e.g., when Q = 10100 10111, $q = 7$, $b - 1 = 1$ in Example 4.1b). □

Algorithm 1: BRANCH(x, Q)

Input: a node x and a query key Q
Output: largest integer b such that $x.K_{b-1} < Q$, and $D = \max_{1\le i\le n}$(D-bit$(Q, x.K_i)$)

▷ `Step 1: Find` q `such that lcp(DS(`Q`), DS`$_q$`) is maximum.`
1 Make n copies of DS(Q);
2 **for** $1 \le i \le n$ **do** // SIMD
3 $\quad$ XOR$_i \leftarrow$ XOR(DS(Q), DS$_i$);
4 Find q such that XOR$_q$ is minimum among XOR$_i$'s;
▷ `Step 2:`
5 Find $D \leftarrow$ D-bit(Q, K_q) by comparing Q and K_q;
▷ `Step 3: Find largest` b `such that` $K_{b-1} < Q$.
6 **if** $Q = K_q$ **then** $b \leftarrow q$;
7 **else**
8 $\quad$ Make n copies of D;
9 $\quad$ **for** $1 \le i \le n$ **do** // SIMD
10 $\quad\quad$ **if** $D_i \le D$ **then** $C[i] \leftarrow 1$;
11 $\quad\quad$ **else** $C[i] \leftarrow 0$;
12 $\quad$ **if** $Q > K_q$ **then**
13 $\quad\quad$ Find the smallest $b > q$ such that $C[b] = 1$ by shifting left by q and finding the first 1-bit position (if such integer b does not exist, b is set to $n + 1$);
14 $\quad$ **else** // if $Q < K_q$
15 $\quad\quad$ Find the largest $b \le q$ such that $C[b] = 1$ by shifting right and finding the last 1-bit position;
16 **return** (b, D);

4.2. Optimizations

We refine the basic algorithm for SIMD implementation. Algorithm 1 shows the detailed code of the basic algorithm using SIMD instructions. In Step 1, we use bitwise-XOR operations for computing the lcp's.

In Step 2, we compare Q and K_q and compute $D = \text{D-bit}(Q, K_q)$, which is the maximum among D-bit(Q, K_i)'s. In Step 3, we use a bit array $C[1..n]$ where $C[i]$ is 1 if $D_i \leq D$, and 0 otherwise. When $Q > K_q$, we compute the smallest $b > q$ such that $C[b] = 1$ by shifting left C by q to remove the elements whose indices are less than or equal to q and finding the first 1-bit in the shifted C. If such b does not exist, then $Q > K_n$ and thus we set $b = n + 1$, which occurs in range search presented in Section 5. When $Q < K_q$, we similarly find the largest $b \leq q$ such that $C[b] = 1$. In this case, such b must exist since $Q > K_0$.

Partial D-bit slice. When keys in a node are updated by insertion or deletion, maintaining D-bit slices exactly may require key accesses, which cause cache misses. For example, inserting a new key 11010 01100 in Fig. 2(a) makes position 8 a new D-bit position and thus the bit values at position 8 of all the keys must be added into the D-bit slices. To reduce key accesses, we use a partial version of D-bit slice, where some bits are allowed to be unspecified. More precisely, the *partial D-bit slice* of K_i, denoted by $p\text{DS}_i$, is defined as follows:

1. For a branching position of K_i, $p\text{DS}_i$ has an exact value.
2. For a non-branching position of K_i, $p\text{DS}_i$ has an exact value or is expressed as an *unknown bit*, which is represented as 0. Thus, for a non-branching position, bit 0 of $p\text{DS}_i$ means that its real value can be 0 or 1, while bit 1 means that its real value is 1.
3. For any $j \neq i$, if a substring α of $p\text{DS}_i$ and a substring β of $p\text{DS}_j$ are derived from the label of an identical edge in trie T, then α and β are the same.

Example 4.2. Fig. 2(d) shows partial D-bit slices of the keys in Fig. 2(a), where unknown bits are indicated by red 0. For K_i's ($i = 2, \ldots, 5$), bits in position 3 are all 1 but they are expressed as (unknown bit) 0 in $p\text{DS}_i$'s (the third bits). Note that it is possible that the third bits of $p\text{DS}_i$'s ($i = 2, \ldots, 5$) are all 1, but it is not allowed that some bits are 0 and the others are 1 by Condition (3) of pDS.

The notion of the partial D-bit slice was inspired by the sparse partial key of HOT [6], but it is different from the sparse partial key as follows. The bits at non-branching positions, marked in blue or red in Fig. 2(d), are all 0 in the sparse partial keys, while these bits can be 0 or 1 in our partial D-bit slices. The advantages of using value 1 in partial D-bit slices are twofold.

- When partial D-bit slices are newly computed in a node (e.g., for bulk loading), we can do it by simply setting $p\text{DS}_i$ as DS_i (D-bit slice) because the exact D-bit slice is a special case of the partial D-bit slice.
- The value 1 in partial D-bit slices leads to performance improvement in unsuccessful searches, as described in Section 5.

Step 1 of Algorithm 1 works correctly with the partial D-bit slices, which we explain by the following example.

Example 4.3. Given the keys and the partial D-bit slices in Fig. 2, consider a query Q = 11010 00101 (DS(Q) = 101011). When using the D-bit slices DS_i's, XOR_5 = 000000 is the minimum among XOR_i's (i.e., $q = 5$). We can get the same result using $p\text{DS}_i$'s. Let XOR'_i denote $\text{XOR}(\text{DS}(Q), p\text{DS}_i)$ to distinguish it from $\text{XOR}_i = \text{XOR}(\text{DS}(Q), \text{DS}_i)$. Since the third bit of $p\text{DS}_5$ = 10$\underline{0}$011 is different from real value 1, XOR'_5 = 00$\underline{1}$000. However, the third bit '1' of XOR'_5 does not affect the result that XOR'_5 is the minimum among XOR'_i's because the third bits of $p\text{DS}_i$'s ($i = 2, \ldots, 5$) are all 0 by Condition (3) of pDS.

Contiguous version. If the positions in D for a node x are contiguous bit positions, the branching problem can be solved by the following simpler algorithm because the D-bit slice DS_i of K_i is a substring of K_i. (Notice that DS_i is used in this algorithm but not $p\text{DS}_i$.)

Table 1
Data structure of an internal node.

Member	Description (size when $M = 16$)
n	The number of keys stored in the node (1B)
$K_1, \ldots, K_{16}$	K_i is the ith key itself or its embedded substring (8B × 16)
$C_1, \ldots, C_{16}$	C_i is a pointer to the ith child (8B × 16)
$hkeyptr$	Pointer to the largest key K_n (8B)
$next$	Pointer to the next sibling (8B)
$D_1, \ldots, D_{16}$	D_i is the D-bit position of K_{i-1} and K_i (2B × 16)
D_{min}	The minimum value of $\{D_1, \ldots, D_n\}$ (2B)
$\text{DS}_1, \ldots, \text{DS}_{16}$	DS_i is the partial or contiguous D-bit slice of K_i (2B × 16)
D-positions and D-masks	Byte positions (1B × 16) and 8-bit masks (1B × 16) for D

1. We find b such that $\text{DS}_{b-1} < \text{DS}(Q) \leq \text{DS}_b$ using SIMD instructions as follows. First, make n copies of DS(Q) and compute $C[i]$ for $1 \leq i \leq n$, where $C[i] = 0$ if $\text{DS}_i < \text{DS}(Q)$; $C[i] = 1$ otherwise. Then, find the first 1-bit position in C.
2. If $\text{DS}(Q) = \text{DS}_b$, we make a comparison of Q and K_b, and if $Q > K_b$, increase b by one.

The contiguous version is more efficient than the basic algorithm as follows. Step 1 of the contiguous version is simpler than Step 1 of the basic algorithm. The basic algorithm always makes a comparison of two full keys, while the contiguous version makes such a comparison only when $\text{DS}(Q) = \text{DS}_b$. Step 3 of the basic algorithm using D_i's is not necessary in the contiguous version.

However, the contiguous version cannot solve the branching problem when the positions in D are not contiguous, as shown in Example 4.4. When the positions in D are not contiguous, therefore, we must find q such that D-bit(Q, K_q) is maximum as in Step 1 of the basic algorithm.

Example 4.4. Given four keys $(K_0, \ldots, K_3) = (0001, 1100, 1101, 1111)$ and $D = \{0, 2, 3\}$, $(\text{DS}_0, \ldots, \text{DS}_3) = (001, 100, 101, 111)$. If $Q = 1010$, then DS(Q) = 110 and the contiguous version finds $b = 3$. However, the correct answer is not 3, but 1.

To take advantage of the contiguous version of the branching algorithm, we store D-bit slices as substrings of the keys if it is possible. Note that the number of the D-bit positions at a node is at most $n \leq M$. Let D_{min} be the minimum D-bit position and D_{max} be the maximum D-bit position. When the difference of D_{min} and D_{max} is less than or equal to M, we use the substring of K_i of length M starting at D_{min} as the D-bit slice DS_i, that is, $D = \{D_{min}, D_{min} + 1, \ldots, D_{min} + M - 1\}$. We call it *a contiguous D-bit slice*. Note that no unknown bit is used in a contiguous D-bit slice. Then, we can use the contiguous version of the branching algorithm for a node storing the contiguous D-bit slices. We call a node a *contiguous node* if the node stores a contiguous D-bit slice, and a *non-contiguous node* otherwise.

4.3. Implementation details

Table 1 and Fig. 3 show the data structure of an internal node, where M is assumed to be 16. When keys are too long to store inside nodes, we embed a part of each key (8B from the byte containing D_{min}) in a node to reduce cache misses due to the full key access. Thus, in Step 2 of Algorithm 1, we first compare the embedded substring of K_q and the corresponding substring of Q, and if they are the same, then we compare the full K_q and Q. We can get the pointer to the full K_q from the child C_q. Note that K_q is the largest key in C_q and the pointer to the key is stored in C_q ($C_q.hkeyptr$).

Since D is defined per node, extracting DS(Q) from query key Q should be performed at each node. To extract DS(Q) fast, we maintain byte positions (the D-positions) to which bit positions in D belong, and

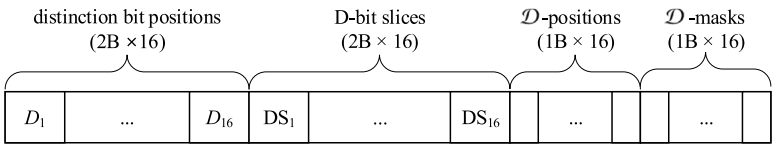


Fig. 3. Data structure for the D-bit information.

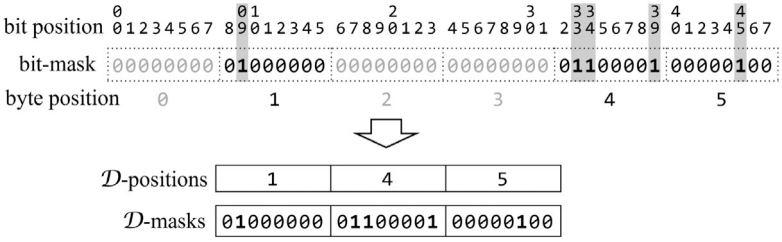


Fig. 4. $\mathcal{D}$-positions and $\mathcal{D}$-masks for $\mathcal{D} = \{9, 33, 34, 39, 45\}$.

8-bit masks for the bytes (the $\mathcal{D}$-masks) at each node (see Fig. 4). The structure of a leaf is also similar. Instead of children $C_1, \ldots, C_{16}$, each leaf has pointers $Rid_1, \ldots, Rid_{16}$ to entries whose keys are $K_1, \ldots, K_{16}$, respectively.

We use four types of nodes: a contiguous internal node, a non-contiguous internal node, a contiguous leaf, a non-contiguous leaf. In our implementation, we set $M = 16$ and allocate 384 bytes to each type of node except for a contiguous leaf. For a contiguous leaf, we exclude keys $K_1, \ldots, K_{16}$ to reduce the space and allocate 256 bytes.

Since 16 DS_i's (or D_i's), each of which is 2 bytes long, correspond to SIMD width 256 (32 bytes), lines 2–3 (or lines 9–11) of Algorithm 1 can be performed by one (or three) SIMD instruction from AVX2 instruction set. Hence, unlike the B$^+$-tree (or its variants), Algorithm 1 and the contiguous version do not have loops for branching, but they consist of an $O(1)$ number of SIMD and other sequential instructions, irrespective of the key length.

Comparison with the fusion tree. Both DB$^+$-tree and fusion tree are B$^+$-trees with $O(1)$ time branching and $\log_M N$ height, where N is the total number of keys. Whereas each node of the fusion tree requires a lookup table of M^2 words to get $O(1)$ branching time, the DB$^+$-tree does not need a lookup table and requires $O(M)$ words per node. By choosing $M = \log_2 N$, the fusion tree obtains $O(\log N / \log\log N)$ time for point search (theoretically beating $O(\log N)$ bound).

We choose the value of M as follows, so that the DB$^+$-tree can be a practical index for main-memory database systems. (In what follows, "SIMD width" means the width of SIMD register in bits, and "SIMD parallelism" means the number of D-bit slices processed by a single SIMD instruction). As an example, consider lines 2–3 of Algorithm 1.

- When $M <$ key length (in bits): The number of SIMD instructions to process all D-bit slices is M/SIMD parallelism. To include all D-bit positions, the length of a D-bit slice should be M bits. Hence, SIMD parallelism is SIMD width/M. Combining the two formulas above, the number of SIMD instructions to process all D-bit slices is M^2/SIMD width.
- When $M \geq$ key length (in bits): Again the number of SIMD instructions to process all D-bit slices is M/SIMD parallelism. Since the length of a D-bit slice is the key length in this case, SIMD parallelism = SIMD width/key length. Hence, the number of SIMD instructions to process all D-bit slices is $M\times$ key length/SIMD width.

Therefore, the best choice in practice is to choose M such that $M^2 =$ SIMD width. In our experiments, $M = 16$ and SIMD width = 256. So all D-bit slices are processed by a single SIMD instruction.

5. Search and update

For search and update operations, inter-node algorithms (i.e., algorithms between nodes) in the DB$^+$-tree are the same as those in the B$^+$-tree. However, intra-node algorithms (i.e., algorithms inside a node) are quite different due to the D-bit information (D_i, DS_i, and $\mathcal{D}$).

Point search. Given a query key Q, the point search is to find the entry whose key is Q in the DB$^+$-tree. It can be done by traversing down the tree from the root to a leaf x which can contain Q using branching at each node. Searching leaf x is simpler than branching at internal nodes since we only need equality check to Q.

- Contiguous leaf: We first find b such that $DS_b = DS(Q)$ (Step 1) and then compare K_b and Q (Step 2). If such b does not exist or K_b is not equal to Q, there is no entry in the tree whose key is Q.
- Non-contiguous leaf (Algorithm 2): Although a key K_i is equal to Q, the partial D-bit slice pDS_i may not be equal to $DS(Q)$ due to unknown bits. Thus, we first find b such that $XOR_b = XOR(DS(Q), pDS_b)$ is minimum as in the branch algorithm (Step 1), and compare K_b and Q (Step 2). However, we can skip the full key comparison in Step 2 if there is a bit position where the bit of $DS(Q)$ is 0 and the bit of pDS_b is 1 since it means that a mismatch occurs at the position. (Note that bit 1 in pDS means that its real value is 1.) This case can be determined by the expression $AND(XOR_b, pDS_b)$ where AND is a bitwise-AND operation. A bit value of the expression is 1 if and only if the bits of $DS(Q)$ and pDS_b are 0 and 1, respectively. Thus, we perform a comparison of K_b and Q if all the bit values of the expression are 0. Step 3 of Algorithm 1 is not necessary.

Range search. We consider two kinds of range search operations:

- Given two keys Q_1 and Q_2 ($Q_1 < Q_2$), RangeSearch1(Q_1, Q_2) is to find all keys k such that $Q_1 \leq k < Q_2$ in the index.
- Given a key Q_1 and a positive integer R, RangeSearch2(Q_1, R) is to find the R smallest keys larger than or equal to Q_1.

Range search can be performed by first searching for Q_1 and simply scanning rightward the leaves until a key larger than or equal to Q_2 is found (RangeSearch1) or R keys are reported (RangeSearch2).

Algorithm 3 shows the pseudocode of our algorithm for RangeSearch1. After searching for Q_1 (line 1), for every leaf x in scanning, we find the largest integer b such that $x.K_{b-1} < Q_2$, which can be done basically using the branching algorithm (lines 2 and 7). If $b \leq x.n$, scanning ends at the leaf x, and otherwise it continues at the next leaf (lines 4–8). Note that Q_2 may be greater than the largest key $x.K_n$ in node x.

Algorithm 2: SEARCHNONCONTILEAF(x, Q)

Input: a leaf x and a query key Q ($x.K_0 < Q \le x.K_n$)
Output: integer b such that $x.K_b = Q$ (-1 if such b does not exist)

▷ Step 1:
1 Find b such that XOR_b is minimum among $\text{XOR}_i = \text{XOR}(\text{DS}(Q), p\text{DS}_i)$ $(1 \le i \le n)$;
▷ Step 2:
2 **if** $\text{AND}(\text{XOR}_b, p\text{DS}_b) = 0$ **then**
3 Compare Q and K_b;
4 **if** $Q = K_b$ **then return** b;
5 **return** -1;

Algorithm 3: RANGESEARCH1(Q_1, Q_2)

Input: two keys Q_1 and Q_2 ($Q_1 < Q_2$)
Output: report all keys k such that $Q_1 \le k < Q_2$

▷ Scanning the first leaf.
1 Find the leaf x and the integer a such that $x.K_{a-1} < Q_1 \le x.K_a$ (point search);
2 $(b, D) \leftarrow$ BRANCH(x, Q_2);
3 Report $x.K_a, \ldots, x.K_{b-1}$ as outputs;
▷ Scanning the next leaves.
4 **while** $b > x.n$ ***and*** $x.next \neq nil$ **do**
5 $x \leftarrow x.next$;
6 **if** $D < x.D_{min}$ **then** $b \leftarrow x.n + 1$;
7 **else** $(b, D) \leftarrow$ BRANCH(x, Q_2);
8 Report $x.K_1, \ldots, x.K_{b-1}$ as outputs;

To improve the performance, before executing the branching algorithm, we first check if $D < x.D_{min}$ in line 6. By Lemma 5.1, D = D-bit($x.K_0, Q_2$) at the time of executing line 6. That is, the $(D+1)$st bit of $x.K_0$ is 0, and that of Q_2 is 1, because $x.K_0 < Q_2$. If $D < x.D_{min}$, the first $D+1$ bits of $x.K_n$ are the same as those of $x.K_0$, and thus $x.K_n < Q_2$ and D = D-bit($x.K_n, Q_2$). Therefore, we can conclude $b = x.n+1$, and reports all keys in x as outputs. Otherwise (i.e., $D \ge x.D_{min}$), we execute the branching algorithm in line 7. This test ($D < x.D_{min}$) makes our range search particularly fast, as shown in Section 6.

Lemma 5.1. *At the time of executing line 6 of Algorithm 3, D is equal to D-bit($x.K_0, Q_2$).*

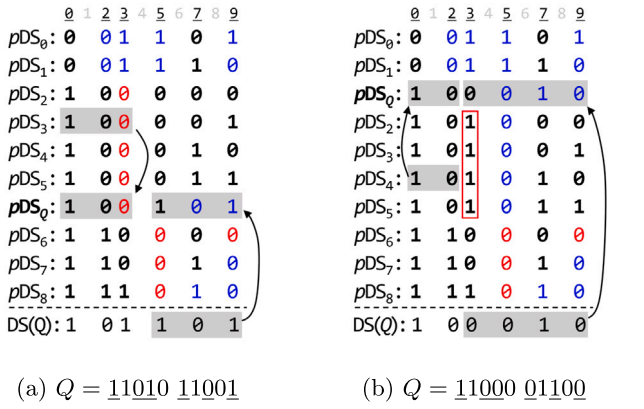
Proof. Let x' be the previous leaf of x. When processing the leaf x', we have two cases.

(i) Procedure BRANCH was executed (line 2 or 7): In x', b was $x'.n + 1$. Hence, the value D returned by BRANCH is equal to D-bit($x'.K_n, Q_2$), because the equality D-bit(K_{b-1}, Q) $= D$ in Theorem 4.2(1) holds even for $b = n + 1$. Since $x.K_0 = x'.K_n$, we have D = D-bit($x.K_0, Q_2$).
(ii) Line 6 was executed: Note that D did not change in line 6. Assume inductively (initially by Case (i) and then by repetitions of Case (ii)) that D = D-bit($x'.K_0, Q_2$). By the explanation above Lemma 5.1, D = D-bit($x'.K_n, Q_2$). Since $x.K_0 = x'.K_n$, D = D-bit($x.K_0, Q_2$). □

Insertion. The insertion operation is to insert a new key Q to a tree. We first find the leaf x such that $x.K_0 < Q \le x.K_n$ and store Q into leaf x. If the leaf x is full, x is split into two nodes before Q is stored. The node split causes an insertion in the parent of x and it is propagated toward the root until it reaches a node that is not full.

We also update the D-bit information (D_i's, DS_i's, and D). We only show how to update the information in a non-contiguous leaf x. (The update in a contiguous leaf is obvious and the update in an internal node is similar to that in a leaf.) First, we consider the case of storing a new key Q in node x without splitting x. Let (b, D) be the return value of BRANCH(x, Q) and q be the integer found in Step 1 of BRANCH(x, Q). We assume the general case in which $b > 1$. (We omit the details on the case that $b = 1$, which is handled exceptionally since the information on K_0 is not stored in node x.) To update D_i's, we need D-bit(K_{b-1}, Q) and D-bit(Q, K_b), which can be determined without key comparisons by Theorem 4.2. D-bit slices are updated as follows.

- Computing the partial D-bit slice $p\text{DS}_Q$ of Q for D: For a position r in $p\text{DS}_Q$, let $\hat{r}$ be the position in Q corresponding to the position r. If $\hat{r} < D$, $p\text{DS}_Q[r]$ is set equal to $p\text{DS}_q[r]$. Otherwise, $p\text{DS}_Q[r]$ is set equal to $\text{DS}(Q)[r]$. See Example 5.1.
- Updating $p\text{DS}_i$'s for the existing keys: Since a new branch is created in the trie T due to Q, some unknown bits in $p\text{DS}_i$'s must be changed to 1 by Condition (1) of the partial D-bit slice. If $Q[D] = 0$, for every K_i such that D-bit(Q, K_i) $= D$, the bit of $p\text{DS}_i$ corresponding to position D must be set to 1. If $Q[D] = 1$, we do nothing. See Example 5.2.



(a) $Q = \underline{11010}\ \underline{11001}$ (b) $Q = \underline{11000}\ \underline{01100}$

Fig. 5. Updating partial D-bit slices when inserting Q, where gray boxes indicate where bits of $p\text{DS}_Q$ are from. Note that bits in the red rectangle of (b) were unknown bits 0 before inserting Q. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Example 5.1. Suppose that $Q = \underline{11010}\ \underline{11001}$ (DS(Q) = 101101) is inserted in the node of Fig. 2. Then, $q = 3$, $D = 5$, and $b = 6$. That is, Q is stored between K_5 and K_6. Then, $p\text{DS}_Q$ is $\underline{100}101$ (bits at positions $\hat{r} < D$ are underlined), because 100 is from $p\text{DS}_3$ and 101 is from DS(Q) (see Fig. 5(a)). Note that the third bit of $p\text{DS}_Q$ is 0 (unknown bit) like those of $p\text{DS}_2, \ldots, p\text{DS}_5$, which satisfies Condition (3) of the partial D-bit slice.

Example 5.2. Suppose that $Q = \underline{11000}\ \underline{01100}$ (DS(Q) = 100010) is inserted in the node of Fig. 2. Then, $q = 4$, $D = 3$, $b = 2$, and Q is stored between K_1 and K_2. Then, $p\text{DS}_Q$ is $\underline{100}010$ and the third bits (unknown bits) of $p\text{DS}_2, \ldots, p\text{DS}_5$ are changed to 1 (see Fig. 5(b)).

If D is a new position not in $\mathcal{D}$, D is added into $\mathcal{D}$ and one bit corresponding to position D is inserted in every $p\text{DS}_i$ as follows. We first set the bit as 0 (unknown bit) without accessing key K_i and then perform computing $p\text{DS}_Q$ and updating $p\text{DS}_i$'s described above. Note that although the node x has less than M keys, the size of $\mathcal{D}$ may be M because of dummy positions in $\mathcal{D}$. Thus, before adding the new position in $\mathcal{D}$, if $\mathcal{D}$ is full (i.e., it has M positions), we eliminate dummy positions from $\mathcal{D}$ so that it has only the D-bit positions of x. For this, we compute new $\mathcal{D}$ from the D-bit positions of x, and eliminate bits at dummy positions from the partial D-bit slices.

Next, consider the case of splitting a node x. In this case, we also split D-bit positions and D-bit slices in the node. However, we do not split the position set D.

Example 5.3. Suppose that we split the node in Fig. 2 into two nodes x_1 and x_2 so that x_1 has $K_1, \ldots, K_4$ and x_2 has $K_5, \ldots, K_8$. Then, the D-bit positions of x_1 are $\{0, 7, 9\}$ but we set $x_1.D$ to be $\{0, 2, 3, 5, 7, 9\}$, which is equal to $x.D$. Note that positions 2, 3 and 5 are not necessary for branching at the node x_1. However, we do not eliminate these dummy positions immediately since eliminating the positions causes updating D-bit slices. As described above, we eliminate them later when $x_1.D$ contains too many dummy positions.

The insertion may cause to convert the type of nodes. A contiguous node x may be converted to a non-contiguous node when storing Q in x. Moreover, a non-contiguous node x may be converted to a contiguous node when splitting x.

Overall, our insertion operation requires $O(M)$ time per node, and thus it takes $O(M \log_M N)$ time if node split is propagated toward the root.

Deletion. The deletion operation is to delete an existing key Q from the DB$^+$-tree. We first find the leaf x storing the key K_b such that $K_b = Q$, and delete K_b from x. The node x is merged with its adjacent sibling when x becomes empty, and deletion is propagated toward the root.

Updating the D-bit information in x is quite simple. Assume the general case in which $n > 1$ and $b < n$. We delete DS_b and the larger of D_b and D_{b+1} because D-bit$(K_{b-1}, K_{b+1}) = \min(D_b, D_{b+1})$ by Lemma 4.1. But, we do not change the position set D even though there is a new dummy position.

Comparison with HOT. Although our partial D-bit slice is similar to the sparse partial key of HOT [6], our point search/update/range search algorithms are quite different from those of HOT because HOT and DB$^+$-tree are structurally different (i.e., HOT is a trie and DB$^+$-tree is a B$^+$-tree).

- The point search of HOT (with query Q) traverses from the root to a leaf node. If Q exists in the index, it must exist in the arrived leaf. Thus, one full key comparison confirms the existence of Q in the arrived leaf. The insertion of HOT (with query Q) first traverses from the root to a leaf node just like the point search. However, the arrived node may not be the correct branching point for inserting Q. Since HOT is a trie-based structure, the branching point for insertion is on the path from the root to the arrived leaf. Thus, the insertion algorithm of HOT goes up the tree to the correct branching point and then inserts Q.
- The DB$^+$-tree is a B$^+$-tree-based structure and thus it is important to find the correct child to follow in an internal node. The algorithms of HOT cannot be directly applied to a B$^+$-tree. The point search algorithm of HOT, if it is applied to a B$^+$-tree, may not find the correct child in each node and consequently it may not arrive at the leaf node storing query key Q even if Q exists in the tree. Also for insertion, the arrived leaf using the point search algorithm of HOT may not be the leaf node where Q should be inserted. The branching algorithm of the DB$^+$-tree always finds the correct child to follow in an internal node. Hence, the point search and insertion algorithms of the DB$^+$-tree work just like those of the original B$^+$-tree.
- The range search algorithm of the DB$^+$-tree using our branching algorithm is significantly faster than those of other indexes, as shown in the experiments.

Therefore, efficient point search/update/range search algorithms in a "B$^+$-tree" is the main contribution of our paper.

6. Performance evaluation

Since B-tree-based indexes and trie-based indexes have different characteristics and applicabilities, we conduct two baseline evaluations: one for comparing the DB$^+$-tree with other B$^+$-tree variants, and another for comparing the DB$^+$-tree with trie variants.

6.1. Experimental settings

We have implemented the DB$^+$-tree in C++ to analyze the performance experimentally. All the other indexes to be compared are also implemented in C++. All the experiments were carried out on an Intel Xeon-based stand-alone computer with 4 E7-8890 v3 2.5 GHz CPUs, one TB RAM, running SUSE Linux. The main metric for evaluation was the throughput of queries processed (in million operations per second) with a chosen index, averaged over five runs.

6.2. Synthetic and real-world datasets

We used four real-world datasets and four synthetic datasets to evaluate the DB$^+$-tree and other existing indexes. The datasets used in the experiments are briefly described below.

6.2.1. Synthetic datasets

Alphanumeric : Each key is a 32 byte string, each byte of which is selected from {'0'–'9', 'A'–'Z', 'a'–'z'} following the Zipf distribution.

Random220 : Each key is a 32 byte string, each byte of which is selected from {1–220} following the Zipf distribution.

Customer Name : The keys are obtained from the Name column of Customer table in the TPC-H (version 2.18.0) [28]. The length of a key is 18 bytes.

8B Integer : Each key is a 63-bit random integer as in [6].

6.2.2. Real datasets

ERP Data : Each key is a concatenation of three columns whose domain cardinalities are 30, 100 000, and 5. The length of a key is 19 bytes.

Wiki Title : Each key is a Wikipedia title.[2]

Wiki URL : Each key is a Wikipedia link of DBPedia [29].

YAGO : The keys are the yagoFacts from YAGO (version 3.1) [30]. In Wiki Title, Wiki URL, and YAGO datasets, keys longer than 128 bytes are excluded so that a key is not longer than 128 bytes.

6.2.3. Workload generation

We generated a set of benchmark workloads with Zhang et al.'s index micro-benchmark [31], which uses the YCSB benchmark [32] to generate workloads and measures the index performance. We included a few additional workloads to evaluate the DB$^+$-tree and other indexes with respect to index construction, point search, range search, and insert/delete operations.

- The workload for point search is composed of 50% successful searches and 50% unsuccessful searches. For successful searches, query keys were generated by YCSB core workload C. For unsuccessful searches, query keys were generated by YCSB Insert operations, and they are all distinct. The workload includes one million point search queries.

[2] http://dumps.wikimedia.org/enwiki/.

- Six workloads are used for range search: the selectivity set to 0.0001%, 0.001%, 0.01%, 0.1%, 1%, or 10%. Each workload of small ranges (less than or equal to 0.1%) includes 1 million range queries, and the workloads of larger ranges include 100 thousand and 10 thousand range queries for selectivity 1% and 10%, respectively.
- The workload for insert/delete is composed of 50% insertions and 50% deletions. The workload includes one million insert/delete queries.

A range query is defined by two search keys that specify the beginning and the end of a key range (RangeSearch1(Q_1, Q_2)). However, trie-like indexes do not support such a range query directly. So, for trie-like indexes, a range query is redefined as a pair of the smallest key and the number of keys to be retrieved (RangeSearch2(Q_1, R)).

6.2.4. Index construction

A B$^+$-tree index and its variants can be built by inserting individual keys or by bulk loading. Since the performance of B$^+$-tree variants is sensitive to how it has been constructed, we consider two scenarios of index construction in the experiments. In the first scenario, a B$^+$-tree index is built by bulk-loading 10 million keys with the fill factor set to 100%. In the second scenario, a B$^+$-tree index is built by bulk-loading 10 million keys with the fill factor set to 75% followed by a mix of 0.5 million random insertions and 0.5 million random deletions. These two B$^+$-tree indexes are targeted for read-only workload for a static database and update-heavy workload for a dynamic database, respectively. In both the scenarios, the resulting B$^+$-tree stores the same set of keys.

Unlike the B$^+$-tree variants, the trie variants, ART and HOT, have no bulk-loading code available for them. Thus, they are built by inserting keys individually in random order.

6.3. Baseline evaluation one - with B$^+$-tree variants

In this section, we compare the DB$^+$-tree with a few B$^+$-tree variants, namely, the STX B$^+$-tree,[3] and the pkB-tree [2]. The B$^+$-tree stores either the keys or the pointers to the keys in the tree nodes. The details of the B$^+$-tree variants are described below.

- The B$^+$-tree (key) is an STX B$^+$-tree that stores the keys in the tree nodes. The size of a tree node is fixed to 256 bytes for short keys or is increased to guarantee fanout 8 for longer keys.
- The B$^+$-tree (key pointer) is an STX B$^+$-tree that stores key pointers. The size of a tree node is fixed to 256 bytes and the fanout is 16.
- The pkB-tree stores partial keys of two bytes. The size of an internal node is 384 bytes and the size of a leaf node is 256 bytes. We implemented the pkB-tree ourselves.

6.3.1. Construction of B$^+$-tree variants

Fig. 6(a) compares the DB$^+$-tree and the B$^+$-tree variants with respect to the time for building an index by bulk loading with 100% and 75% fill factors for the eight datasets, in which the 8B Integer dataset is not applicable to the B$^+$-tree (key pointer). Since the DB$^+$-tree and the pkB-tree require additional data structures and computation for index construction, their construction time was a little higher than that of the B$^+$-tree (key pointer). For long keys (Wiki Title, Wiki URL, and YAGO), however, the DB$^+$-tree was faster than the B$^+$-tree (key).

Fig. 6(b) compares the DB$^+$-tree and the other B$^+$-tree variants with respect to index sizes for the static and dynamic database scenarios. Note that the DB$^+$-tree and the pkB-tree store partial keys in their indexes. Due to additional data structures, the index sizes of the DB$^+$-tree and the pkB-tree are larger than that of the B$^+$-tree (key pointer).

[3] https://github.com/bingmann/stx-btree.

Table 2
Ratio (%) of contiguous nodes in point search (successful search 50%, unsuccessful search 50%).

Dataset	Internal	Leaf
Alphanumeric	82.15	39.94
Random220	93.93	68.31
Customer name	57.58	78.89
8B Integer	99.99	98.33
ERP	44.95	0.00
Wiki Title	34.61	1.41
Wiki URL	13.44	1.47
YAGO	23.10	0.07

6.3.2. Point search

Fig. 7 shows the throughput of the DB$^+$-tree and the other B$^+$-tree variants for point search when the ratio of successful searches was 50%, where node prefetching was applied to all indexes. Overall, the DB$^+$-tree was 1.3x–2.8x (on average 1.7x) faster than the pkB-tree, 1.9x–3.5x (on average 2.7x) faster than the B$^+$-tree (key pointer) and 1.5x–2.2x (on average 1.9x) faster than the B$^+$-tree (key) for point search queries.

The DB$^+$-tree built for the static database scenario widened the throughput gap with the B$^+$-tree variants than the DB$^+$-tree built for the dynamic database scenario. The DB$^+$-tree outperformed the B$^+$-tree variants because its branching algorithm was superior running in the SIMD operational mode and required key pointer dereferencing less frequently.

We have looked into the effects of key pointer dereferencing and the ratio of contiguous nodes found on the search paths in point searches for the static database scenario. Fig. 8 shows that the DB$^+$-tree required dereferencing key pointers much less frequently than the pkB-tree, which is one of the main reasons for the superior search performance of the DB$^+$-tree.

Table 2 reports the ratios of contiguous internal and leaf nodes the DB$^+$-tree encountered on the search paths. It shows that the performance of the DB$^+$-tree tended to be higher when the ratio of contiguous nodes was higher among the index nodes encountered during searches.

6.3.3. Range search

To evaluate the range search performance of the DB$^+$-tree and the B$^+$-tree variants, we carried out two sets of experiments (one for the query selectivity and the other for the data sets) with RangeSearch1(Q_1, Q_2). The results are shown in Figs. 9(a) and 9(b), respectively.

Fig. 9(a) shows the relative throughput of the DB$^+$-tree and the B$^+$-tree variants for range queries with increasing selectivity (i.e., with an increasing number of keys retrieved). The Alphanumeric dataset was used for this experiment. The DB$^+$-tree was the best across all the selectivities and the performance gap was widened as the selectivity increased. This is because the DB$^+$-tree was able to determine whether all the keys in a leaf node were within the search range by a simple test examining D_{min}. The blue line in Fig. 9(a) shows the ratio of leaf nodes that were skipped without any key comparison by the simple test. Fig. 9(b) shows the relative throughput of the DB$^+$-tree and the B$^+$-tree variants for range queries for the four real datasets and the four synthetic datasets. The selectivity was fixed to 1% for all the datasets. The DB$^+$-tree performed better than the B$^+$-tree variants in all datasets except the 8B integer dataset. Specifically, the DB$^+$-tree was 1.0x–1.7x (on average 1.3x) times faster than the pkB-tree, 1.5x–3.6x (on average 2.2x) faster than the B$^+$-tree (key pointer), and 0.7x–6.7x (on average 3.5x) faster than the B$^+$-tree (key). As the length of keys increased, the DB$^+$-tree outperformed the B$^+$-tree (key) with wider margins.

6.3.4. Insert/Delete

Figs. 10(a), 10(b), and 10(c) show the throughput of the DB$^+$-tree and the other B$^+$-tree variants for insertions and deletions, when the ratio of insertion was 100%, 50%, and 0%, respectively. For each

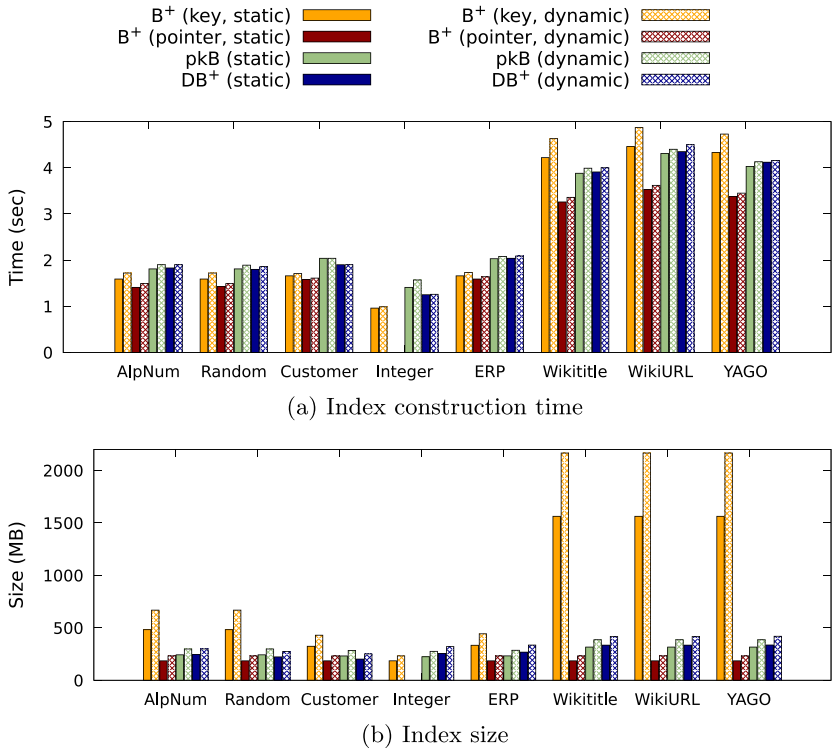


Fig. 6. Index construction performance.

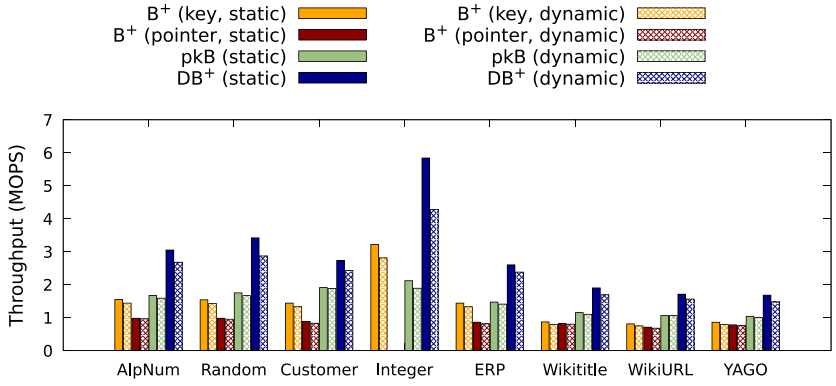


Fig. 7. Point search (successful search 50%, unsuccessful search 50%).

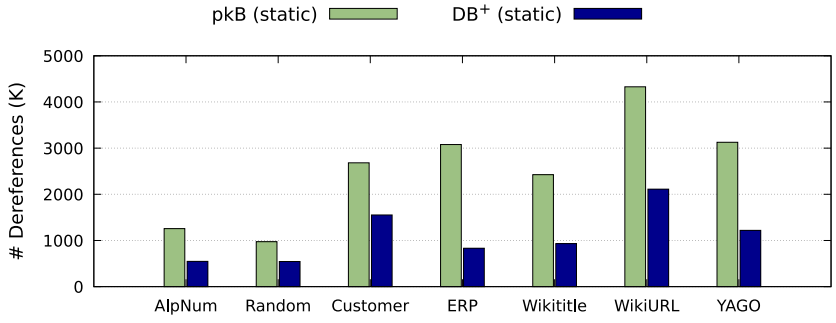
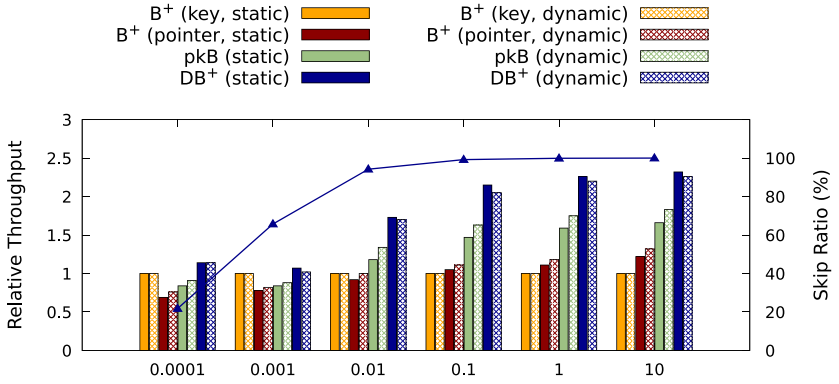


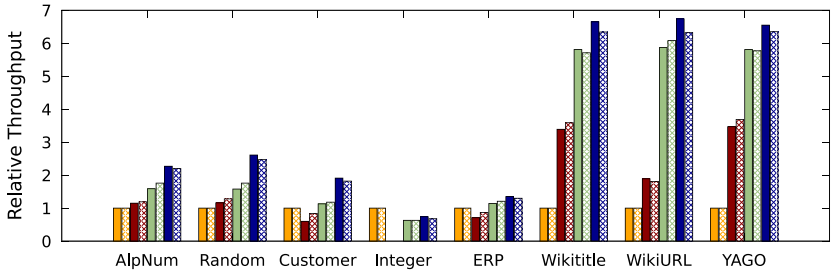
Fig. 8. Dereferencing key pointers in point search (successful search 50%, unsuccessful search 50%).

dataset shown in the figures, we measured the throughput of one million operations achieved by the indexes in comparison.

Across all the eight datasets, the DB$^+$-tree achieved the best throughput. Specifically, the DB$^+$-tree was 1.1x–1.6x (on average 1.4x) faster than the pkB-tree, 1.9x–3.5x(on average 2.7x) faster than the B$^+$-tree (key pointer), and 1.3x–2.8x (on average 2.3x) faster than the B$^+$-tree (key). This superior performance of the DB$^+$-tree for insertions and deletions was possible due to the partial D-bit slice, which allows us to perform local changes on the D-bit information in the processing of insertions and deletions.



(a) Relative throughput and leaf skip ratio of the DB$^+$-tree with different selectivities (%) for the Alphanumeric dataset



(b) Relative throughput with different datasets when the selectivity was fixed to 1%

Fig. 9. Range search (RangeSearch1(Q_1, Q_2)).

6.4. Baseline evaluation two - with trie variants

The trie and its variants are traditionally considered in-memory search structures but they are now viable options for main-memory database systems. In this section, we compare the DB$^+$-tree with ART and HOT, which are both trie-variant indexes. The source codes of ART and HOT are obtained from the authors [6,23], and we added an additional procedure for range search in ART.

6.4.1. Index construction

Fig. 11(a) compares the DB$^+$-tree with the trie variants with respect to the time for building an index. The DB$^+$-tree was built by bulk loading with 100% and 75% fill factors. The construction time of the DB$^+$-tree was approximately 1.8x and 2.7x faster than that of ART and HOT, respectively. Both ART and HOT were built by inserting individual keys because there is no bulk-loading code available for them. Fig. 11(b) shows the index sizes of the DB$^+$-tree and the trie variants. The index size of HOT was significantly smaller than those of the other indexes.

6.4.2. Point search

Fig. 12 shows the throughput of the DB$^+$-tree and the trie variants for point search when the ratio of successful searches was 50%. Overall, the DB$^+$-tree performed comparably with ART and HOT. The relative throughput of the DB$^+$-tree for the static database scenario was in the range from 0.9x to 1.6x in comparison with ART and in the range from 1.0x to 1.6x in comparison with HOT. The DB$^+$-tree for the dynamic database scenario underperformed slightly but the throughput reduction was on average 13 percent of that of the DB$^+$-tree for the static database scenario.

6.4.3. Range search

To evaluate the range search performance of the DB$^+$-tree and the trie variants, we carried out two sets of experiments (one for the query selectivity and the other for the data sets) with RangeSearch2(Q_1, R). The results are shown in Figs. 13(a) and 13(b), respectively.

Fig. 13(a) shows the relative throughput of the DB$^+$-tree and the trie variants for range queries with increasing selectivity (i.e., with an increasing number of keys retrieved). The Alphanumeric dataset was used for this experiment. When the selectivity was 0.01% or greater, the DB$^+$-tree produced significantly higher throughput than ART and HOT. Specifically, the throughput of the DB$^+$-tree for the static database scenario was 14x–19x higher than that of ART, and 6x–8x higher than that of HOT. This result should not be surprising given the inherent difference in the way a range query is processed by a B$^+$-tree variant and by a trie variant. Fig. 13(b) shows the relative throughput of the DB$^+$-tree and the trie variants for range queries for the four real datasets and the four synthetic datasets. The selectivity was fixed to 1% for all the datasets. The same trend was observed in this experiment too.

6.4.4. Insert/Delete

Figs. 14(a), 14(b), and 14(c) show the throughput of the DB$^+$-tree and the trie variants for insertions and deletions, when the ratio of insertion was 100%, 50%, and 0%, respectively. For each dataset shown in the figures, we measured the throughput of one million operations achieved by the indexes in comparison.

ART is structurally closest to the conventional trie and is able to process insertions quickly. The performance of ART varies greatly depending on the dataset since its depth varies from dataset to dataset. For datasets where ART has a small depth, ART is the best and the DB$^+$-tree is the runner-up. For others, the DB$^+$-tree outperformed the trie variants with non-negligible margins in most cases.

6.5. Sensitivity analysis

We have conducted sensitivity analyses of the DB$^+$-tree with respect to four parameters: key length, number of keys, entropy of key values, and ratio of successful searches. We measured the throughput of point

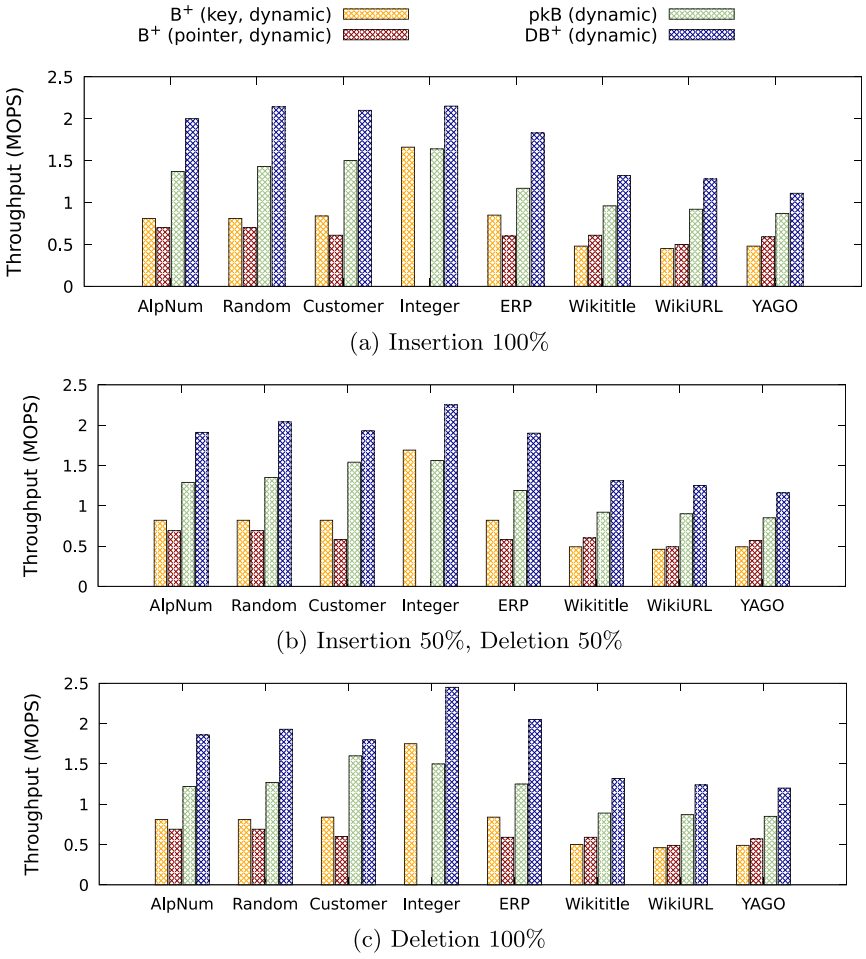


Fig. 10. Insertion and deletion.

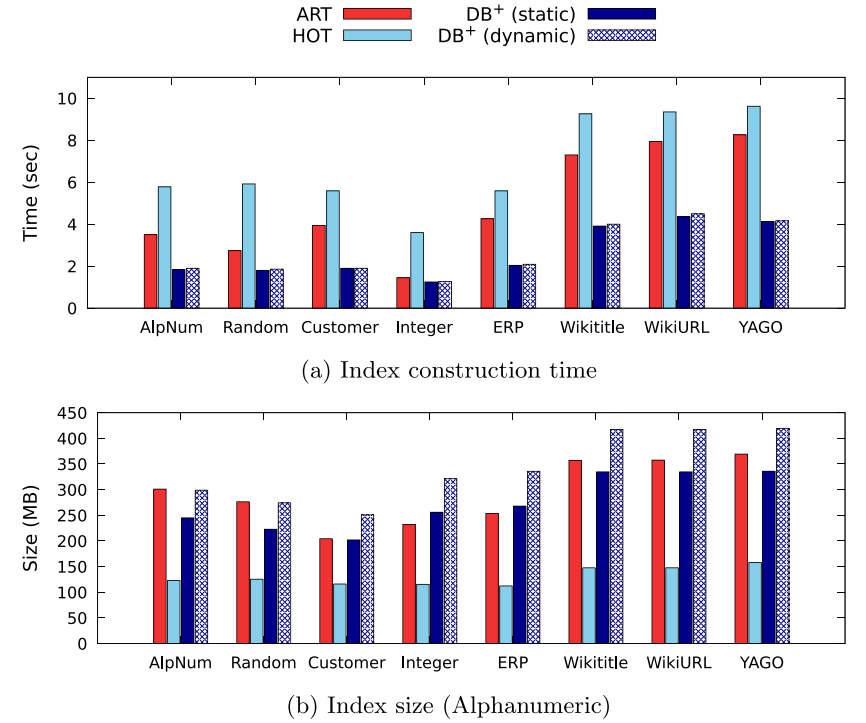


Fig. 11. Index construction performance.

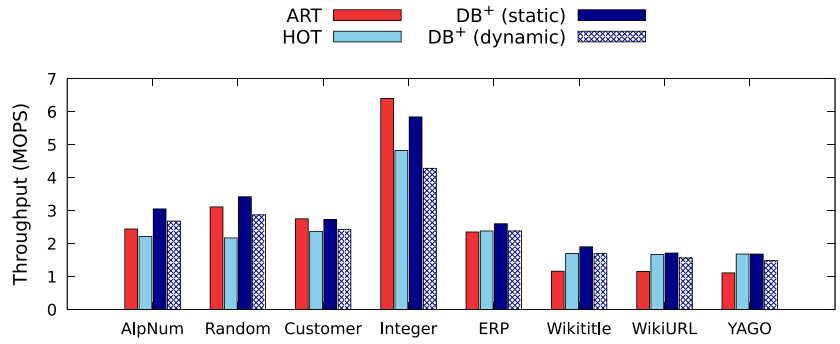
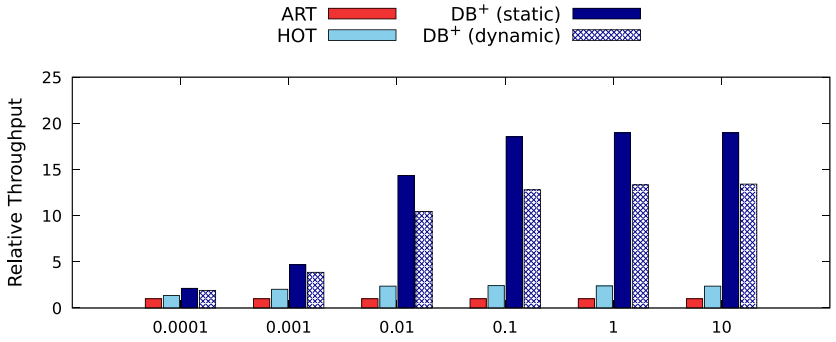
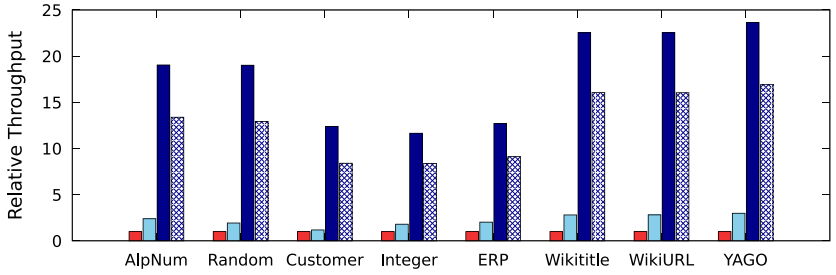


Fig. 12. Point search (successful search 50%, unsuccessful search 50%).



(a) Relative throughput with different selectivities (%) for the Alphanumeric dataset



(b) Relative throughput with different datasets when the selectivity was fixed to 1%

Fig. 13. Range search (RangeSearch2(Q_1, R)).

searches in the static database scenario. For key length, number of keys, and entropy of key values, the ratio of successful searches was 50%. For key length, number of keys, and ratio of successful searches, the Alphanumeric dataset was used.

6.5.1. Key length

Fig. 15(a) shows the throughput of point searches measured with different key lengths varied from 16 bytes to 128 bytes. The general trend was that the throughput declined consistently in all indexes compared, as the key length increased. The DB$^+$-tree was the best performer in all key lengths.

6.5.2. Number of keys

Fig. 15(b) shows the throughput of point searches measured with the different numbers of keys indexed varied from 2.5 million keys to 40 million keys. As was expected, the throughput declined consistently in all indexes compared, as the number of keys indexed increased. However, the DB$^+$-tree was always the best and more scalable than any other index compared with.

6.5.3. Entropy of key values

We conduct a sensitivity analysis on the entropy of key values, as in [2]. When the key values in a byte are in the range $[1, 12]$ (resp., $[1, 62]$, $[1, 100]$, $[1, 220]$) in the Random220 dataset, the byte entropy for the generated keys is 3.09 (resp., 4.85, 5.34, 6.11). Fig. 15(c) shows the throughput of point searches measured with different entropy of the key values. Among the six indexing methods, the three B$^+$-tree variants as well as HOT were insensitive to the entropy of key values. On the other hand, the throughput of the DB$^+$-tree and ART improved as the entropy increased. The DB$^+$-tree achieved the highest throughput among all the six indexing methods.

Table 3
Ratio (%) of contiguous nodes in the sensitivity analysis on the entropy of key values.

Dataset	Internal	Leaf
$[1, 12]$	40.81	2.68
$[1, 62]$	85.53	37.96
$[1, 100]$	90.62	49.55
$[1, 220]$	93.93	68.31

The DB$^+$-tree has two types of nodes in both internal nodes and leaves: contiguous and non-contiguous. Since the branching algorithm of contiguous nodes is simpler and faster than that of non-contiguous nodes, the performance of the DB$^+$-tree increases as the ratio of contiguous nodes gets higher in point searches. Table 3 shows the ratio of contiguous nodes when the entropy of key values varies. When the

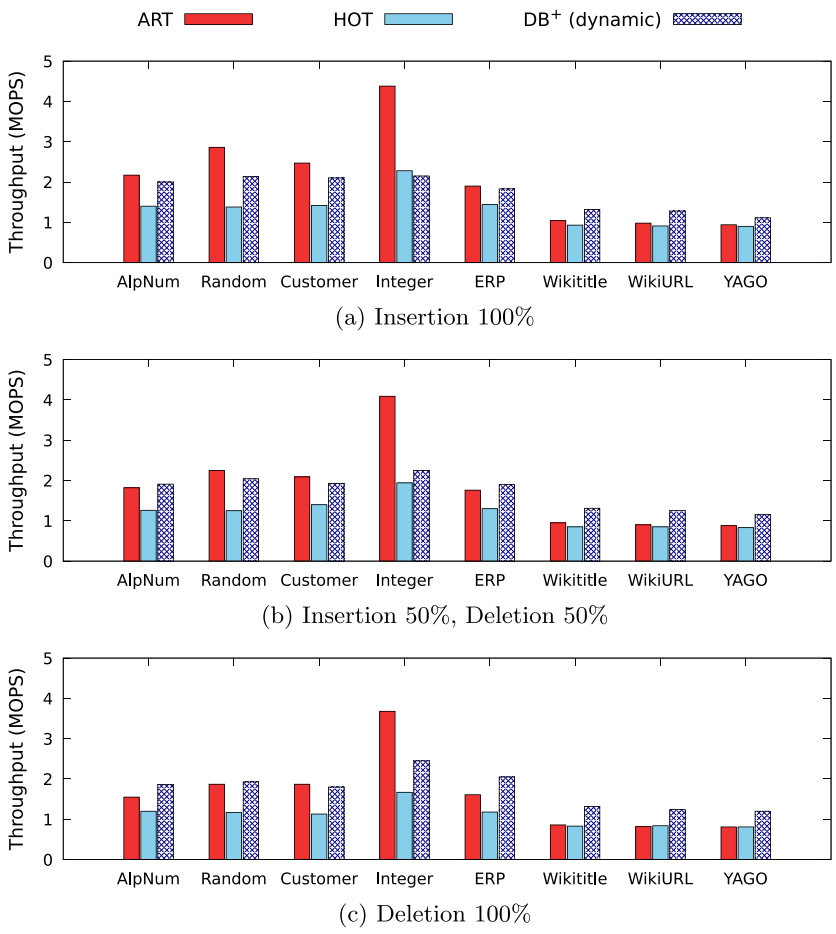


Fig. 14. Insertion and deletion.

entropy gets larger, the possibility that distinction bit positions appear in consecutive M (16 in our implementation) bit positions (i.e., the possibility that a node becomes a contiguous node) increases. That is, as the entropy increases, so does the ratio of contiguous nodes (Table 3), which in turn improves the performance of the DB$^+$-tree (Fig. 15(c)).

6.5.4. Ratio of successful searches

We also conduct a sensitivity analysis on the ratio of successful searches. Fig. 15(d) shows the throughput of point searches when the ratio of successful searches varies from 100% to 0%. Again, the three B$^+$-tree variants and HOT were insensitive to the ratio of successful searches, the throughput of the DB$^+$-tree and ART improved as the ratio decreased, and the DB$^+$-tree got the highest throughput among the six indexes.

Since the DB$^+$-tree contains the partial information of keys (i.e., partial D-bit slices and embedded substrings of keys) in a node, it scarcely needs key pointer dereferencing when it performs branching in a node. For a successful search (i.e., the query key exists in the index), one full key comparison (thus one key pointer dereferencing) is necessary at the end of the point search to confirm the existence of the query key in the index. When the ratio of successful searches is 100% in Fig. 16, the number of key pointer dereferencing in the DB$^+$-tree is slightly over 1 million (which is the number of point searches in the experiment). That is, the number of key pointer dereferencing other than full key comparisons is very small in the DB$^+$-tree, when compared to the pkB-tree. As the ratio of successful searches decreases, so does the number of key pointer dereferencing in the DB$^+$-tree (Fig. 16), which increases the performance of the DB$^+$-tree (Fig. 15(d)). Especially when the ratio of successful searches is 0% (i.e., insertions and many cases of range searches), the performance of the DB$^+$-tree is much better than the pkB-tree.

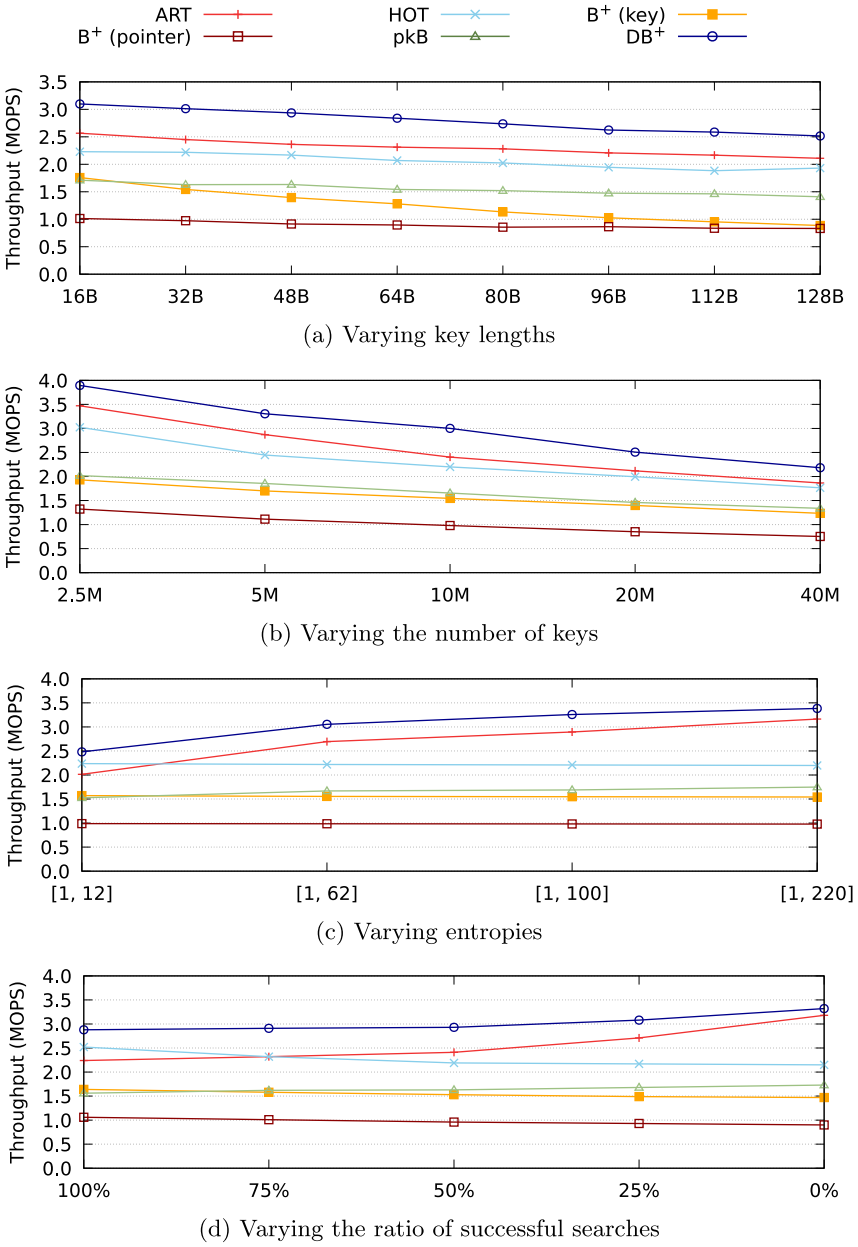
6.6. DB$^+$-tree as in-memory index

B$^+$-trees have been widely used in database systems due to its proven performance with disk storage. Since we propose the DB$^+$-tree as an in-memory index, it has different parameters than the B$^+$-tree (see [10] for an initial work of in-memory index structures). The size of a DB$^+$-tree node is either 256 or 384 bytes, which are small multiples (4 or 6) of the cache line size (64 bytes in our implementation). With the use of data prefetching instructions, 4 or 6 consecutive cache lines can be accessed without much additional latency compared with a random single cache line access. Thus, the DB$^+$-tree can achieve fast branching with a limited memory access cost, which is a dominant cost factor of in-memory index structures. Such data prefetching is also used for other in-memory indexes (e.g., HOT).

When compared to trie-based indexes such as HOT, B$^+$-trees have nice properties which the DB$^+$-tree can benefit from in an in-memory setting. For example, (1) fixed node sizes make it easy to handle storage space management, and (2) every leaf node has the same depth, which provides stable search performance as well as effective random sampling for approximate query processing [33,34]. Since the DB$^+$-tree has the same tree structure as the B$^+$-tree, it shares these properties.

7. Conclusion

In this paper, we have proposed a novel branching algorithm in the B$^+$-tree which can be implemented in an $O(1)$ number of SIMD and other sequential instructions, irrespective of the key length. The resulting variant of the B$^+$-tree, called the DB$^+$-tree, has fast branching, which leads to fast point search, range search, and update operations. Our experiments show that the DB$^+$-tree is the best performer among



(a) Varying key lengths

(b) Varying the number of keys

(c) Varying entropies

(d) Varying the ratio of successful searches

Fig. 15. Sensitivity analysis.

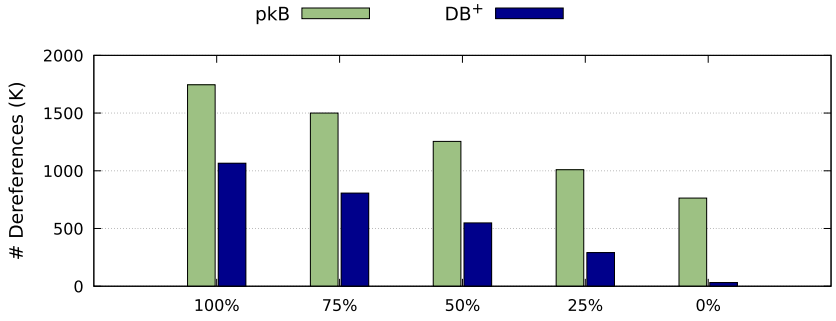


Fig. 16. Key pointer dereferencing in point search (varying the ratio of successful searches).

B-tree-based indexes, and it is comparable to trie-based indexes HOT and ART.

Extending the DB$^+$-tree to a disk-based B$^+$-tree to handle datasets larger than main memory [35] is a future work. That is, if a disk page of the B$^+$-tree is accessed, the disk page is copied to an in-memory buffer. From the page in the buffer, we can build a DB$^+$-subtree. As more and more disk pages are accessed, many DB$^+$-subtrees are built in memory, which are part of one big DB$^+$-tree. The DB$^+$-tree built as above can have better performances than B$^+$-trees.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors do not have permission to share data.

Acknowledgments

Lee, Nam and Park were supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2018-0-00551, Framework of Practical Algorithms for NP-hard Graph Problems). Na was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2020R1F1A1068873). Moon was supported by National Research Foundation (NRF) of Korea (Grant No. NRF-2020R1A2C1010358).

References

[1] R. Bayer, E.M. Mccreight, Organization and maintenance of large ordered indexes, Acta Inform. 1 (3) (1972) 173–189.
[2] P. Bohannon, P. Mcllroy, R. Rastogi, Main-memory index structures with fixed-size partial keys, in: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, 2001, pp. 163–174.
[3] D.E. Ferguson, Bit-tree: A data structure for fast file processing, Commun. ACM 35 (6) (1992) 114–120.
[4] M.L. Fredman, D.E. Willard, Blasting through the information theoretic barrier with fusion trees, in: Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, 1990, pp. 1–7.
[5] M.L. Fredman, D.E. Willard, Surpassing the information theoretic bound with fusion trees, J. Comput. Syst. Sci. 47 (3) (1993) 424–436.
[6] R. Binna, E. Zangerle, M. Pichl, G. Specht, V. Leis, HOT: A height optimized trie index for main-memory database systems, in: Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data, 2018, pp. 521–534.
[7] T.J. Lehman, M.J. Carey, A study of index structures for main memory database management systems, in: Proceedings of the 12th International Conference on Very Large Data Bases, 1986, pp. 294–303.
[8] R. Bayer, K. Unterauer, Prefix B-trees, ACM Trans. Database Syst. 2 (1) (1977) 11–26.
[9] J. Rao, K.A. Ross, Cache conscious indexing for decision-support in main memory, in: Proceedings of the 25th International Conference on Very Large Data Bases, 1999, pp. 78–89.
[10] J. Rao, K.A. Ross, Making B+-trees cache conscious in main memory, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, 2000, pp. 475–486.
[11] S. Chen, P.B. Gibbons, T.C. Mowry, Improving index performance through prefetching, in: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, 2001, pp. 235–246.
[12] S. Chen, P.B. Gibbons, T.C. Mowry, G. Valentin, Fractal prefetching B+-trees: Optimizing both cache and disk performance, in: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, 2002, pp. 157–168.
[13] W. Zhang, Z. Yan, Y. Lin, C. Zhao, L. Peng, A high throughput B+tree for SIMD architectures, IEEE Trans. Parallel Distrib. Syst. 31 (3) (2020) 707–720.
[14] Z. Yan, Y. Lin, L. Peng, W. Zhang, Harmonia: A high throughput B+tree for GPUs, in: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, 2019, pp. 133–144.
[15] J.J. Levandoski, D.B. Lomet, S. Sengupta, The Bw-tree: A B-tree for new hardware platforms, in: Proceedings of the 29th International Conference on Data Engineering, 2013, pp. 302–313.
[16] G.-J. Na, S.-W. Lee, B. Moon, Dynamic in-page logging for B-tree index, IEEE Trans. Knowl. Data Eng. 24 (7) (2012) 1231–1243.
[17] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A.D. Nguyen, T. Kaldewey, V.W. Lee, S.A. Brandt, P. Dubey, FAST: Fast architecture sensitive tree search on modern CPUs and GPUs, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, 2010, pp. 339–350.
[18] T. Yamamuro, M. Onizuka, T. Hitaka, M. Yamamuro, VAST-tree: A vector-advanced and compressed structure for massive data tree traversal, in: Proceedings of the 15th International Conference on Extending Database Technology, 2012, pp. 396–407.
[19] R. De La Briandais, File searching using variable length keys, in: IRE-AIEE-ACM Western Joint Computer Conference, 1959, pp. 295–298.
[20] D.R. Morrison, PATRICIA—Practical algorithm to retrieve information coded in alphanumeric, J. ACM 15 (4) (1968) 514–534.
[21] M. Boehm, B. Schlegel, P.B. Volk, U. Fischer, D. Habich, W. Lehner, Efficient in-memory indexing with generalized prefix trees, in: Proceedings of the 14th BTW Conference on Database Systems for Business, Technology, and Web, 2011, pp. 227–246.
[22] T. Kissinger, B. Schlegel, D. Habich, W. Lehner, KISS-tree: Smart latch-free in-memory indexing on modern architectures, in: Proceedings of the 18th International Workshop on Data Management on New Hardware, 2012, pp. 16–23.
[23] V. Leis, A. Kemper, T. Neumann, The adaptive radix tree: ARTful indexing for main-memory databases, in: Proceedings of the 29th International Conference on Data Engineering, 2013, pp. 38–49.
[24] V. Leis, F. Scheibner, A. Kemper, T. Neumann, The ART of practical synchronization, in: Proceedings of the 12th International Workshop on Data Management on New Hardware, 2016, pp. 1–8.
[25] H. Zhang, H. Lim, V. Leis, D. Andersen, M. Kaminsky, K. Keeton, A. Pavlo, SuRF: practical range query filtering with fast succinct tries, in: Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data, 2018, pp. 323–336.
[26] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, third ed., MIT Press, 2016.
[27] P. van Emde Boas, Preserving order in a forest in less than logarithmic time, in: Proceedings of the 16th Annual Symposium on Foundations of Computer Science, 1975, pp. 75–84.
[28] M. Poess, C. Floyd, New TPC benchmarks for decision support and web commerce, ACM SIGMOD Rec. 29 (4) (2000) 64–71.
[29] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, Z. Ives, Dbpedia: A nucleus for a web of open data, in: Proceedings of the 6th International Semantic Web Conference, 2007, pp. 722–735.
[30] F.M. Suchanek, G. Kasneci, G. Weikum, Yago: A Core of Semantic Knowledge, in: 16th International Conference on the World Wide Web, 2007, pp. 697–706.
[31] H. Zhang, D.G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, R. Shen, Reducing the storage overhead of main-memory OLTP databases with hybrid indexes, in: Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, 2016, pp. 1567–1581.
[32] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with YCSB, in: Proceedings of the 1st ACM Symposium on Cloud Computing, 2010, pp. 143–154.
[33] F. Olken, D. Rotem, Random sampling from B+ trees, in: Proceedings of the 15th International Conference on Very Large Data Bases, 1989, pp. 269–277.
[34] Z. Zhao, D. Xie, F. Li, AB-Tree: Index for concurrent random sampling and updates, in: Proceedings of the 48th International Conference on Very Large Data Bases, 2022, pp. 1835–1847.
[35] V. Leis, M. Haubenschild, A. Kemper, T. Neumann, LeanStore: In-memory data management beyond main memory, in: Proceedings of the 34th International Conference on Data Engineering, 2018, pp. 185–196.