Chair for Database Systems
School of Computation, Informatics and Technology
Technical University of Munich

Optimizing B$^+$-tree Node Layout for Improved Search Performance Using SIMD Instructions

Guided research

Uroš Stojković ✉

Advisor: Altan Birler
Supervisor: Prof. Thomas Neumann

✉ stoj@in.tum.de

April 18, 2024

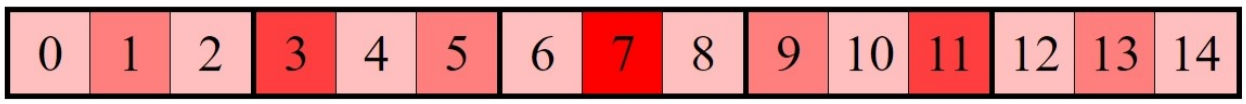
Abstract — B$^+$-tree is a ubiquotous data structure in database systems where it is used to index large datasets which do not fit into main memory. Its initial design was based on two premises: first, external memory has several orders of magnitude higher latency than main memory; second, unit of transfer between external memory and main memory is often much bigger than a single data record. In the following work, we aim to further improve the performance of B$^+$-trees also by drawing inspiration from hardware characteristics – specifically the latency discrepancy between CPU caches and main memory, as well as availability of Single-Instruction-Multiple-Data (SIMD) instructions on modern processors. We experiment with different ways of organizing data within a single B$^+$-tree node, conceived to more effectively leverage the aforementioned capabilities, with primary goal of optimizing lookups without worsening other operations like inserts, erasures and range-scans. Our benchmarks show that lookups become 75% faster on average when we use Eytzinger layout combined with SIMD instructions, whereas inserts and erasures are at most 15% slower and often faster than with standard layout.

Contents

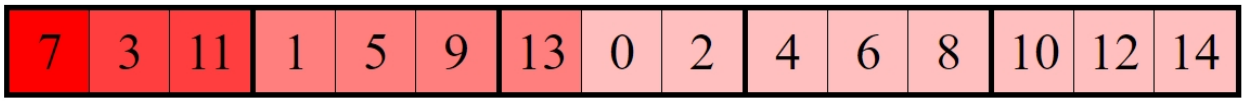
1 Introduction

All of the most important, user-facing, B-tree operations, including lookups, inserts and deletes, involve several node-local lookups – one per level. By lookup, we mean retrieval of a value associated with a specific key. Within a single node, keys and values are usually sorted, enabling use of binary search.

Unfortunately, binary search is not very well suited for modern CPUs which use multi-level memory hierarchy and favor sequential access. To understand why this is the case, we start out by taking a closer look at the memory access pattern of binary search over a sorted array of elements completely resident in memory. As a reminder, binary search works by maintaining a range within array which may contain the requested element, and then, on each step, compares it with the element in the middle and halves the range, until it collapses to a single element, leading to logarithmic time complexity. If we examine the positions of elements that were loaded for comparison, we observe that they are located far apart from each other at first, with distance exponentially decreasing over time, ultimately ending on the same cache line in the final iterations. In the first approximation, this means that first few loads will be cache misses, while last few will be cache hits. The unfortunate thing is that the cache line containing a last couple of elements will almost certainly be irrelevant for the next search targeting a different element if we assume that the size of the cache line is negligible to the size of the whole array and that each element is equally likely to be looked up. On the other hand, each query is determined to load a cache line which contains a middle element of the array (i.e. a starting point of binary search). This cache line is thus likely to stay in cache, although most of the time we will use only a single element from it, wasting precious cache space.



(a) Standard (sorted) layout



(b) Eytzinger layout

Figure 1 Expected frequency of access during binary search (lighter means less frequent). Cache line boundaries are bolded; for simplicity, we assume a single cache line contains 3 elements.

The problem with storing elements in sorted order (in the context of binary search) is that we mix together commonly and rarely retrieved data which then leads to suboptimal use of cache (see Figure 1a). Instead, we can try to sort elements based on their expected frequency of access during binary search i.e. with median element coming first, followed by first and third quartile, and so on, as depicted in Figure 1b. This effectively corresponds to the layout of statically constructed binary search tree[1] , also known as Eytzinger layout[2]. For each node stored at index k, its children are stored at indices $2k + 1$ and $2k + 2$ and represent two next possible steps in binary search (e.g. children of the root, which is a median element, are first and third quartile, respectively). With such a layout, spatial locality would be good in the beginning as first few accesses touch the same cache line, but poor in the end, in contrast to the previous case. However, it is enhanced temporal locality that gives the Eytzinger layout advantage over regular one. To understand why this is the case, assume that all 3 quartiles reside in cache due to frequent access to them. They will probably occupy 3 cache lines when elements are sorted, but only one with Eytzinger layout. The net result is that cache appears larger because we are using the cache space smarter, consequently leading to less cache misses and faster execution.

[1] This is not exactly correct. Root of such a tree is not necessarily median.

[2] Named after an Austrian nobleman called Eytzinger who invented a way to concisely record family trees.

Another feature of modern processors that encourages us to rethink binary search are vector instructions which allow us to perform a single operation on multiple pairs of operands in parallel. For instance, the latest SIMD extension to x86 instruction set – AVX512 – lets one operate on entire cache line in parallel. In our case, we can leverage these instructions to compare many keys simultaneously while paying roughly the same price as for a single-element comparison. Thus, we are incentivized to consider an adaptation of Eytzinger layout that represents statically stored n-ary tree, where $n - 1$ is the number of elements that fits a cacheline.

In this work, we apply above ideas to B-tree nodes and develop accompanying algorithms. We evaluate our implementation and show that we can significantly enhance lookup speed without compromising performance of other B-tree operations. The report is structured as follows: Section 3 provides a brief overview of essential concepts required for understanding the subsequent sections. Section 4 presents our implementation of the standard B-tree node layout. In Section 5, we delve into our implementation of the Eytzinger layout, emphasizing the algorithms used to maintain this layout during updates. Section 6 explores the generalization of the Eytzinger layout to accommodate implicit representations of n-ary trees, enabling the utilization of SIMD instructions. Finally, in Section 7, we present and analyze the results obtained from benchmarking B-tree operations across different node layouts.

2 Related work

B-trees were introduced by Rudolf Bayer and Edward M. McCreight [1]. A comprehensive treatment of the them and their use in database systems is given by Goetz Greafe [2]. Furthermore, multiple articles and blog posts were written about optimizing binary search and use of Eytzinger layout [3, 7, 5]. However, most of them focus only on lookup performance, and don't tackle the problem of maintaining the layout in presence of updates. We, on the other hand, not only allow, but expect nodes to be dynamic i.e. to change over time. Furthermore, they assume their dataset fits entirely in memory and that no paging is needed, so their data structures are not written atop of a buffer manager. Also, while they run their experiments on a whole spectrum of array sizes, we are only interested in smaller lengths, up to several thousand, which is approximately the maximum number of records you can fit on a node as big as 64 KB.

3 Preliminaries

Before delving into the intricacies of our work, it is beneficial to review some fundamental concepts and data structures that underpin our research and are therefore essential to grasp.

3.1 B-tree

One of the important and recurring problems in computer science is efficient search in a corpus of data. Without loss of generality, we can assume data consists of records which are just a key-value pairs and that search is based on keys. The critical thing is devising a data structure which allows fast point and range queries while sacrificing update performance as little as possible.

B-tree is one of the data structures invented to tackle this problem. It is a self-balancing n-ary tree with two distinctive properties:

- all leaf nodes[3] are on the same level
- every node except the root is at least half-full

A variation of B-tree – B^+-tree – impose an additional constraint that all records are stored in leaf nodes. Inner nodes only contain keys and pointers to child nodes - which can be either leaves or inner nodes again. When we talk about B-trees in the rest of the text, we really mean B^+-trees.

[3] leaf nodes are nodes that have no children

B-trees are often used when the storage system has large granularity of reads and writes. To see why this is the case, we can briefly consider why storing ordered array of records on disk is not optimal. To find a record associated with a particular key, we would naturally use binary search, starting by fetching the middle record's key. It's important to note here that we can load from disk only in multiples of disk page size (e.g. 4KB), so we will necessarily fetch neighboring keys as well. However, the next key to be accessed – either first or third quartile – is, assuming records span potentially hundreds of pages, bound to be on a different page. In fact, only a last few steps in binary search are likely to be on the same page. Thus, for most disk loads, which are expensive operations compared to memory access, we only utilize a tiny fraction of data, which is very inefficient. It would be much better to use all the information provided on a given page to further direct our search.

This is where B-trees come in. Size of the B-tree node is typically chosen such that it is a multiple of the unit of transfer of the underlying storage system. Data within a single node is usually sorted, allowing one to perform binary search in main memory. Then, if the node is inner node, we find a pointer to the node one level below where we ought to continue our search. Once we reach a leaf, we return the record associated with a given key, if it happens to exist. Since inner nodes contain significantly more than one key, the depth of the tree (and thus the number of required nodes to perform a lookup) is smaller. Also, the fact that all leaf nodes are on the same level makes search time predictable and uniform over all keys.

On the other hand, updates to B-tree affect only a single leaf node most of the time, whereas in the worst case they require modifications to the number of nodes proportional to the height of the tree. Although such excellent characteristics are paid for by space amplification, it is a worthy trade-off.

For these reasons, B-trees are ubiquitous in database systems where they are used to implement indices or relations themselves. Our work focuses on B-trees in that setting. This makes a difference and should be highlighted because B-tree has to be buffer-managed and implemented with concurrency in mind.

The fact that multiple threads might access parts of the tree in parallel affect the implementation of lookups, inserts and deletes. First of all, during lookups we have to perform lock coupling – we must maintain shared access to the parent page until we acquire the lock of the child page – because it might just happen that after we release the lock of the parent page, another thread, running insert operation, splits the child page such that the key we're looking for gets moved to another page. Furthermore, since a thread must attempt to lock a page while still holding another one, we have to make sure to avoid deadlocks. The most elegant way to do this is to establish an order in which locks have to be acquired. Thus, it is not allowed to hold a child page and request an exclusive access to parent page. This rule in turn affects the way inserts and deletes are implemented. In a single-threaded B-tree, these might trigger split and merge/rebalance of leaf nodes which in turn cause insert and delete/update in their parent node, respectively, and this can repeat recursively up to the root. Hence, these operations require lock-coupling in bottom-up direction, which we saw earlier has to be forbidden. To overcome this issue, splits and merges are instead performed eagerly, during top-down traversal, as described in [6]. For inserts, this means that we guarantee that interior node can always accommodate a new key, while for deletes this means that we tolerate underfull pages until another erase thread doesn't access it and perform a merge/rebalance with a neighboring one. However, note that page can never have less then $N/2 - 1$ keys, where N stands for node's capacity, so we still enjoy the same performance guarantees.

3.2 Buffer manager

A B-tree can become fairly large and its size can easily exceed the capacity of main memory or the portion of memory designated for the running database instance. Moreover, it has to compete with other data structures of database system for this limited space.

Buffer manager is a software component that decides which part of the dataset will be resident in RAM. Since loading from disk is performed in pages (blocks of bytes like 4KB), buffer manager often also works on that level of granularity [4]. That is, when you want to load some data, you also get all the data on the same page.

One of the most important goals of buffer manager is to maximize the number of requests for pages that can be served from main memory and thus minimize expensive reading and writing to disk. The optimal strategy for

[4] There are buffer managers which support variable-size pages. While this design choice results in a more complex implementation, it provides many benefits such as easier handling of large data structures (think large tuples or dictionaries).

this task can be inferred only if we know the page access pattern in advance. However, since this is impossible, buffer manager has to try to predict which pages are likely to be requested again and try to keep them in memory. Whenever a page which is not in RAM is requested, the buffer manager has to decide which page is to be sacrificed. The algorithm which buffer manager uses to make that decision is called eviction policy.

Buffer managers typically expose two methods: `fix` and `unfix` to request access to a page and to release a page, respectively. On `fix`, buffer manager first checks if the page associated with a provided page identifier is already in memory. If not, it decides to evict another page based on certain replacement policy and loads requested page in its place. Finally, buffer manager locks the page in either shared/exclusive mode (this is another parameter to `fix`) and returns an address to a location in memory where page resides. On `unfix`, buffer manager unlocks the page and if no other thread holds it, makes it eligible for eviction. Between calls to `fix` and `unfix`, buffer manager has to make sure that a reference to a page remains valid.

One of the bottlenecks of traditional buffer managers is the fact that each access requires an expensive hash table lookup to translate page identifier to virtual memory address. To solve this problem, some modern buffer managers such as LeanStore [4] rely on a technique called pointer-swizzling. The idea is that a reference to a page can store a disk address (page identifier) or a virtual memory address inside a single 64 bit field. In the latter case, we say reference is swizzled, while in the former it is unswizzled. The most significant bit is used to determine the state, exploiting the the fact that virtual addresses on 64-bit processors use less than 64 bits. Such references are called swips. Whenever a buffer manager loads a page from disk, it swizzles the passed swip (ie. it overwrites it with an address of the allocated memory frame), so later requests to that page can amount only to a bit check. However, as a consequence of using this technique, if there are more than one reference to some page, it becomes very hard to efficiently maintain consistency between them. That's why in LeanStore each page has a single reference making each data structure built on top of it tree-like. Applied to B-trees, this constraint means that leaf nodes cannot hold a reference to neighboring leaf nodes, something which is usually done to speed up range queries.

3.3 Memory hierarchy

The various B-tree node layouts which we describe in later sections are primarily influenced by an approach that explicitly models and acknowledges the influence of memory hierarchy on program performance. By memory hierarchy, we understand a multitude of storage devices used to implement a memory system which differ in their speed and storage capacity. Due to hardware design constraints, these two characteristics are inversely proportional. The highest level of memory hierarchy is typically a register file, while the lowest one can be anything from local SSD or disk, archival magnetic tape or remote storage accessed over the network, but for our purposes it suffices to assume it is main memory. In between, there are several layers of caches. Each level in the hierarchy serves to store and enable a quicker access to a subset of data stored in a level below. The important thing to underline is that data is transferred between caches in cache lines, which are 64 bytes long on most modern systems.

4 Standard layout

Although B-tree implementation was a necessary part of our project, our focus primarily lied in implementing and evaluating non-standard node layouts. Before we discuss those, it will be useful to quickly recapitulate on standard layout which we use as a reference in measurements.

First of all, we differentiate between leaf nodes and inner nodes. While their layout and operations are similar, they still contain differences that justify a separation of their implementations. However, in the following subsections all code snippets correspond to leaf nodes.

In addition, our implementation assumes that both keys and values have fixed, predetermined size. Generalizing it to variable-size should not be very difficult, but would include additional bookkeeping. However, since main ideas we wanted to test out are independent of this choice, we decided to go down the simpler path.

In standard implementations, as taught in algorithms and database courses, B-tree nodes contain array of keys and values[5] which are sorted by keys in the ascending order. In our implementation, a node starts with 4 byte long header which contains information about the level of the node (leaf nodes have level 0) and the current number of values (number of keys in interior nodes is one less). This is followed by `keys` and `values` array. All the operations - lookup, insert, split, merge and rebalance - have canonical implementations and we won't explain them here. However, we are going to spend some time discussing lookup because this is the operation we try to optimize and will be our baseline. Lookups are supported by `lower_bound` function which takes a target key and returns an index of the smallest key which is not smaller than the target key, along with a boolean indicating if equality holds ie. if the key was actually found. Naturally, `lower_bound` relies on binary search.

4.1 Binary search revisited

Binary search algorithm we use somewhat differs from the textbook version. First, instead of indices of the first and the last key within currently "active" window, we maintain only the index of the first along with the size of the window. On each iteration, we split the window into 2 parts and compare the target with the last element of the first half. If the target is greater, we know that it can only be found in the second half, so we increase the base index to point to its first element. In any case, we update the length of the window. Usually, body of the loop contains if-else block that handles both outcomes (lines 9-15 in Listing 1). To reduce the cost of conditional jumps, modern CPUs start speculatively executing one of the branches. However, since the sequence of comparison outcomes during binary search is essentially random, CPU has a hard time predicting the branch which should be taken resulting in lots of branch misses, causing long pipeline stalls and hindering performance.

```
1  std::pair<uint32_t, bool> lower_bound(const KeyT &target) {
2      if (count==0) return {0, false}; // no keys
3
4      ComparatorT less;
5      uint32_t i=0, n = count;
6      // branchy binary search
7      while (n > 1) {
8          auto half = n/2;
9          if (less(keys[i+half-1], target)) {
10             i += half;
11             n = n - half;
12         }
13         else {
14             n = half;
15         }
16     }
17
18     return (i==count-1 && less(keys[i], target)) ?
19     // check if last key is less than target
20         std::make_pair(static_cast<uint32_t>(count), false) :
21         std::make_pair(i, keys[i] == target);
22 }
```

Listing 1 Branchy binary search

Thus, the main idea is to eliminate branching altogether by transforming control dependency to data dependency, as shown in Listing 2. While the value of `i` still depends on the result of the comparison, there exist only a single stream of instructions. Moreover, notice that, despite the general case where the two halves are not of equal length, we always round up to the larger one to avoid branching, even though this sometimes results in an

[5] We can consider references to other nodes (swips) to be values in interior nodes.

extra iteration[6].

```
1  std::pair<uint32_t, bool> lower_bound(const KeyT &target) {
2      if (count==0) return {0, false}; // no keys
3
4      ComparatorT less;
5      uint32_t i=0, n = count;
6      // branchless binary search
7      while (n>1) {
8          auto half = n/2;
9          n -= half; // ceil(n/2)
10         i += less(keys[i+half-1], target) * half;
11         // gcc 13.2 translates this to cmov
12     }
13
14     return (i==count-1 && less(keys[i], target)) ?
15     // check if last key is less than target
16         std::make_pair(static_cast<uint32_t>(count), false) :
17         std::make_pair(i, keys[i] == target);
18 }
```

Listing 2 Branch-free binary search

As a side note, it is interesting that measurements by Khuong and Morin show that branch-free binary search performs worse for arrays larger than L3 cache [3]. The reason is that modern CPUs speculatively execute one branch, so that they start loading next key for comparison earlier than branch-free code. Thus, 50% of the time branchy binary search saves one load-time, and in the other 50% it wastes some time due to pipeline-flushing. It just happens that this is a good trade-off when data is not cached, but bad otherwise.

4.2 Fast updates

During inserts and erasures, on average, we need to shift half of keys and values one position to the right and left, respectively. However, when size of value elements becomes fairly big, the above procedure becomes rather costly. Upon closer consideration, one realizes that it's only keys that need to be sorted, while values don't have to be as long as we maintain a mapping between key slots and value slots. It is only convenient to maintain values sorted because mapping is then a simple identity mapping - value associated with a key at index i is also stored in values array at index i. However, as value size increases, that benefit starts to become outweighed by costs of moving data around. At some point, it makes more sense to add a level of indirection – an array `pointers` of pointers which explicitly stores the mapping between key indices and value indices such that `keys[i]` is associated with `values[pointers[i]]`. During inserts, we can write a value to any free value slot, as long as we update the mapping accordingly. We still have to shift keys and pointers, but that is cheaper now because pointers can be represented as small integers (in our implementation they are represented as 16 bit unsigned integers which is enough since nodes never accommodate more than 2^{16} elements). Regarding managment of free value slots, we have two options:

1. Keep occupied value slots compact[7] and insert new values to the next free slot (leftmost free slot). On erasures, we would have to copy the value from the last occupied slot to the freshly emptied slot and

[6] One may argue that we can again try to transform this into data dependency and rely on `cmov`. The problem is that `while` condition depends on it, so we want to update it as fast as possible. Code where we always round up the size of the window allows CPU to fill the pipeline with instructions of multiple iterations in advance. In addition, when querying immutable array many times, it allows CPU branch predictor to learn the pattern since the number of iterations is independent of the target key.

[7] If there are n values, they occupy first n value slots

update the corresponding pointer. However, the problem is that there's no efficient way to find an index of this pointer i.e. index `i` such that `pointers[i]` points to the last occupied slot.

2. Allow for holes in value slots and maintain a linked list of free value slots. During inserts and erasures, we remove and insert a value slot to the beginning of the list, respectively.

We adopt a second approach and keep the linked list head in the node's header. Empty value slots store the index of the next empty value slot in the list. Furthermore, we place pointer array between keys and values. This is not a random, but calculated choice. First of all, let us stress that there are two versions of `lower_bound` in this setting: one which returns the index in the key array and other which returns the index in the value array. In the latter case, we use pointer array to translate key index found by binary search to value index. Thus, during lookup we touch header (to load `count`), key array (to perform binary search) and potentially pointer array. Profiling using `perf` has shown that `lower_bound` is memory bound[8] ie. most of the time is spent waiting data to be brought to CPU. It turns out that when we place pointers next to the header and key array, fetching the entry in pointer array is slightly faster than when we place them after value array. This is probably because last few keys and a first few pointers share the same cache line, but it also might be due to hardware prefetching of neighboring cache lines.

5 Eytzinger layout

As we described earlier, using Eytzinger layout means storing keys in a succinct static binary search tree in breadth-first order, illustrated in Figure 2. Consequently, the tree always remains balanced.
One trick we employ is to use 1-based indexing with left and right child of a node at index k being at indices $2k$ and $2k+1$, respectively. If the size of the key type in bytes is a power of 2 (as is the case for integer types) and we align `&keys[0]` to a cacheline, this choice means children, grand-children, etc. of a certain node reside on the same cache line. For instance, if keys are 64-bit integers, meaning 8 of them fit into a cacheline, grand-children of `key[k]` are located at indices $4k + j$ $0 \leq j < 4$ (half of a cache line) and grand-grand-children are located at indices $8k + j$, $0 \leq j < 8$ (another cache line). Thus, we can explicitly ask CPU to prefetch a cache line several steps in advance knowing for sure it will be used [3, 7]. In theory, this can improve performance when we have excess memory bandwidth, although we haven't measured any significant speedup.[9]

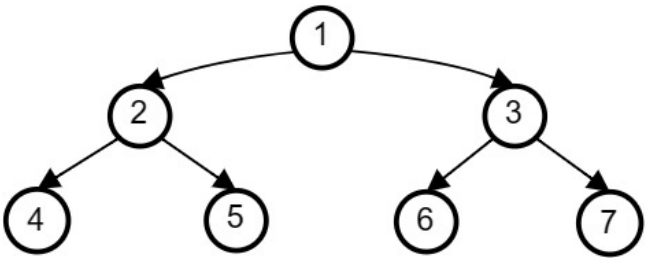


Figure 2 Array as static binary tree: numbers inside nodes stand for their indices

5.1 Lookups

During lookups, we start from the root key and compare it against the target key. If it's less, we descend to the right child; otherwise we proceed to the left child. We continue so until we get past leaf nodes which happens when running index exceeds `count`. We have to remember the last node n where we went left because it contains the smallest key which is not less than the target key. Indeed, if a lesser key meeting this criterion existed, we would have either visited its node after n and continued left from there, or not visited n at all. Fortunately, we can recover this information from the final value of the index since each bit represents a turn we took at the

[8] This applies primarily for leaf nodes
[9] This is in agreement with results obtained by others [7] for similar array sizes.

corresponding level: 0 stands for going left, 1 for going right. We can cheaply calculate how many consecutive right turns starting from the 0^{th} level we made using `std::countr_one` which counts the number of trailing 1s in bit representation of a number, and translates to `not` followed by `tzcnt` assembly instructions. Then we undo these right turns by right-shifting exactly that many positions and then once more to cancel the last left turn. It's important to note that if the target is bigger than all the keys, we always go right, and the above algorithm yields 0 as a resulting index. The implementation of the algorithm described above can be found in Listing 3.

```
1  std::pair<uint32_t, bool> lower_bound(const KeyT &target) {
2      ComparatorT less;
3      uint32_t i=1;
4      while (i <= count) {
5          __builtin_prefetch(&keys[(CACHELINE/sizeof(KeyT))*i]); // optional
6          i = 2*i + less(keys[i], target);
7      }
8
9      // recover index
10     i >>= std::countr_one(i)+1;
11
12     return std::make_pair(i, keys[i] == target);
13 }
```

Listing 3 Code for lookup within a leaf node

5.2 Update operations

A more difficult problem is maintaining Eytzinger layout on updates. In case of B-tree nodes these include not only inserts and erasures, but also operations like split, merge and rebalance. As we'll show later, in our implementation, all of them rely on iterating over indices in in-order traversal (in both directions).

5.2.1 Iteration in in-order traversal

Logic for iteration is encapsulated in Iterator class which implements increment and decrement operators. Inside, we maintain the current index, as well as the first out-of-bounds index, updating the former on each call of increment/decrement. For example, when the latter is 8, Iterator should generate a following sequence of indices:

$$4, 2, 5, 1, 6, 3, 7$$

which corresponds to in-order traversal of tree depicted in Figure 2.

Increment operator, which makes a step forward in in-order traversal, works as follows. If the right child exists, we descend to it and then go left as much as possible. If it doesn't exist, we climb up the tree until we are at some left child, and then we climb once again (see Listing 4). Decrement operator, which makes a step backward in in-order traversal, works similarly.

Iterator class also provide two static functions `begin` and `rbegin` which take first out-of-bounds index as a parameter and return Iterator instance pointing to the first and the last index in in-order traversal, respectively.

```
1  EytzingerIterator& operator++() {
2      if (2*k+1<n) {
3          // go right
4          k = 2*k+1;
5          // then left as much as possible
6          while (2*k<n) k = 2*k;
7      }
```

```
8       else {
9           // climb up until you're the left child and then climb up one more time
10          k = k >> (std::countr_one(k)+1);
11      }
12      return *this;
13  }
```

Listing 4 Implementation of the increment operator

There is a simple lemma concerning the order of indices during in-order traversal which will be important for proving the correctness of some update procedures:

Let $a(n)$ be a sequence of indices of nodes visited in in-order traversal where $n-1$ is the greatest valid index (within bounds) and $a_i(n)$, $1 \leq i < n$ the i-th value of that sequence. Then, if $a_i(n) = a_j(m)$, then $n \leq m \Leftrightarrow i \leq j$.[10]

Simply put, given two sequences, an index cannot appear earlier in the longer one than in the shorter one.

5.2.2 Insert

Finally, we are ready to start with inserts. These are easy when keys are sorted. One first finds an index where new key K should be inserted and checks if it's already there using `lower_bound`. If yes, the corresponding value is just updated; otherwise all the keys beginning with the smallest which greater than K are shifted while taking care nothing is overwritten (ie. iteration goes from right to left). The situation is illustrated in Figure 3. Key array has capacity 12 and new `count` is 9; currently occupied slots are colored in green, a slot to be filled upon insertion is colored in yellow, while vacant slots are colored in red. The key slot at index returned by `lower_bound` is colored in dark green. Numbers inside slots stand for their indices.

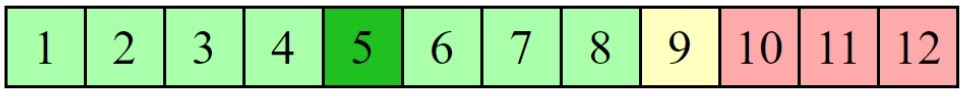


Figure 3 Sorted key slots with standard layout (as earlier, numbers inside nodes stand for their indices). To carry out insertion, we need to move keys starting from slot 5 one position to the left and write the new key to slot 5.

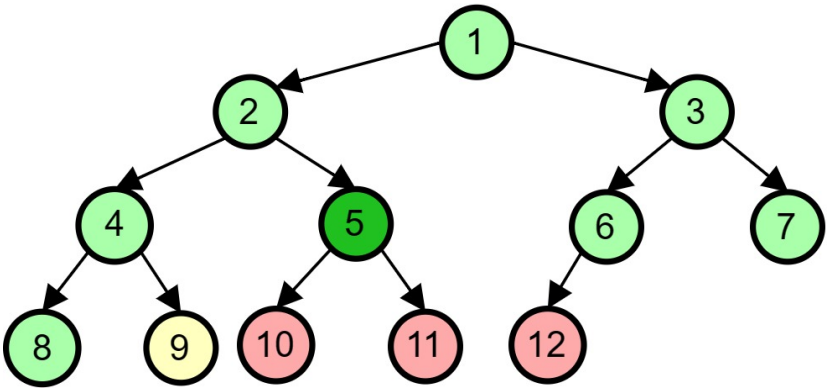
Unfortunately, previously described procedure doesn't work anymore with Eytzinger layout. The reason is that the logical and physical ordering no longer coincide: the first empty slot can appear at arbitrary position in in-order traversal, both before and after the index returned by `lower_bound`.

Bearing that fact in mind, we can still generalize the solution above. We start from the first empty slot (at index `count`) and then move either forward or backward in in-order traversal, shifting keys, until we reach a key that is bigger or smaller than K, respectively (see lines 41-54 in Listing 5). To determine the relative position of "yellow slot" with respect to "dark green slot" i.e to decide if we should go forward or backward, we:

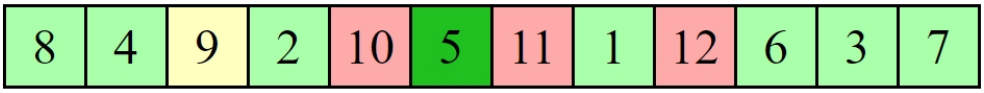
1. Check if `lower_bound` returned 0 (which means that K is greater than all the keys) or if the first empty slot is the new first slot in in-order traversal. If any of these conditions is fulfilled, we know we have to move forward (line 26).

2. Check if the first empty slot is the new last slot in in-order traversal. If yes, we know we have to move backward (line 27).

3. Check if the index i, which comes right after `count` in in-order traversal, is equal to the index returned by `lower_bound`. If yes, we can just insert K and the corresponding value at index `count` and be done. Otherwise, we compare the key at i with K. If it's less, we move forward; otherwise we move backward (lines 28-39).



(a) Eytzinger binary tree



(b) In-order projection of the tree above

Figure 4 Sorted keys with Eytzinger layout. Unlike the Standard layout, the first free slot (light yellow) can appear before the lower bound of the to-be-inserted key (dark green). To carry out insertion, we need to move the key from slot 2 to slot 9 and then write the new key to slot 2.

Figure 4b depicts the same scenario as in the Figure 3, only with Eytzinger layout. As we can see, first empty slot (at index 9) appears before the one returned by `lower_bound` in sorted order. In this case we only have to move the key in slot 2 to slot 9 and write K to slot 2. Since slot 9 is neither the first nor the last in sorted-order (these are 8 and 7, respectively), we use the third rule above to determine that we should go forward (to the right) to arrive at slot 5 by comparing K with a key in slot 2, which logically follows slot 9.

```
1  void insert(const KeyT &key, const ValueT &value) {
2      assert(count < kCapacity); // more place available
3
4      // trivial case with no keys
5      if (count == 0u) {
6          // no keys
7          keys[1] = key;
8          values[1] = value;
9          ++count;
10         return;
11     }
12
13     auto [index,found] = lower_bound(key);
14     if (found) {
15         values[index] = value;
16         return;
17     }
18
19     ++count;
20     Iterator lead(count, count+1);
21     Iterator lag = lead;
22     ComparatorT less;
23     bool forward;
```

[10] In mathematical parlance, $a(n)$ is a subsequence of $a(m)$ when $n \leq m$

```
24      // determine if we should go forward or backward in in-order traversal
25      if (!index || lead == Iterator::begin(count+1)) forward = true;
26      else if (lead == Iterator::rbegin(count+1)) forward = false;
27      else {
28          ++lead;
29          if (*lead == index) goto end; // we insert key into new slot
30          if (less(keys[*lead], key)) {
31              forward = true;
32              keys[*lag] = keys[*lead];
33              values[*lag] = values[*lead];
34              lag = lead;
35          }
36          else {
37              forward = false;
38              lead = lag;
39          }
40      }
41      if (forward) {
42          ++lead;
43          for (; *lead != index; lag=lead, ++lead) {
44              keys[*lag] = keys[*lead];
45              values[*lag] = values[*lead];
46          }
47      }
48      else {
49          --lead;
50          for (; *lag != index; lag=lead, --lead) {
51              keys[*lag] = keys[*lead];
52              values[*lag] = values[*lead];
53          }
54      }
55      end:
56      keys[*lag] = key;
57      values[*lag] = value;
58  }
```

Listing 5 Insertion code

5.2.3 Erase

Deletion is very similar, as can be seen in Listing 6, only now the slot of the to-be-deleted key K acts as a free slot and should be "propagated" to the last position in the array, which then goes out-of-bounds[11]. Here as well, before actual shifting, we have to determine if the last slot comes before or after the slot where K is. Yet, this is much simpler than for `insert` – it just amounts to comparing K with a key in the last slot.

```
1  bool erase(const KeyT &key) {
2
3      auto [index, found] = lower_bound(key);
4      if (!found) return false; // key not found
```

[11] It is also overwritten to minimum possible value for the corresponding key type. We will explain the reason for this when we discuss SIMD adaptation.

```
5
6       Iterator lag(index, count+1);
7       Iterator lead = lag;
8       ComparatorT less;
9
10      if (less(key, keys[count])) {
11          // go right
12          ++lead;
13          for (; *lag != count; lag=lead, ++lead) {
14              keys[*lag] = keys[*lead];
15              values[*lag] = values[*lead];
16          }
17      }
18      else {
19          // go left
20          --lead;
21          for (; *lag != count; lag=lead, --lead) {
22              keys[*lag] = keys[*lead];
23              values[*lag] = values[*lead];
24          }
25      }
26      keys[count] = kNegInf;
27      --count;
28      return true;
29  }
```

Listing 6 Erase code

5.2.4 Split, merge, rebalance

Splits occur when node (B-tree node, not our single-key Eytzinger node) becomes full and has to be divided to accept new keys. We first shift the greater half of keys to the new node (lines 11-14 of Listing 7), and then reorganize and compactify the first half (lines 18-21). During the latter stage, we both read from and write to the same array, so the question arises if we can accidentally overwrite a slot before we moved its contents elsewhere. Fortunately, we can guarantee that never happens when we go backwards because, as we showed, same index can only appear earlier in "shorter" iterators. Thus, by the time we write to some slot, we have already read from it (in the "worst" case, we will issue assignment where source and destination slot are the same).

```
1   KeyT split(char* buffer) {
2       LeafNode* new_node = new (buffer) LeafNode();
3       // old_node retains ceil(count/2) keys
4       // new_node gets floor(count/2) keys
5       uint16_t left_node_count = (count+1u)/2, right_node_count = count/2;
6       assert(left_node_count+right_node_count==count);
7
8       Iterator old_it = Iterator::rbegin(count+1u),
9           left_it = Iterator::rbegin(left_node_count+1u),
10          right_it = Iterator::rbegin(right_node_count+1u);
11      for (auto i=0; i<right_node_count; ++i, --old_it, --right_it) {
12          new_node->keys[*right_it] = keys[*old_it];
13          new_node->values[*right_it] = values[*old_it];
14      }
15      assert(right_it == Iterator::rend(right_node_count+1));
```

```
16
17      auto separator = keys[*old_it];
18      for (auto i=0; i<left_node_count; ++i, --old_it, --left_it) {
19          keys[*left_it] = keys[*old_it];
20          values[*left_it] = values[*old_it];
21      }
22      assert(left_it == Iterator::rend(left_node_count+1u));
23      assert(old_it == Iterator::rend(count+1u));
24      for (auto i=left_node_count+1; i<=count; ++i) {
25          keys[i] = kNegInf;
26      }
27      count = left_node_count;
28      new_node->count = right_node_count;
29      return separator;
30  }
```

Listing 7 Split code

Merging of two nodes happens when both of them become underfull, and is done analogously to splits, only in reverse order. The left node, which is the resulting one, is first reorganized such that it makes space for keys from the right node (lines 6-9 of Listing 7), which are then added (lines 12-15). During reorganization, we use "longer" iterator to generate write indices and "shorter" one to generate read indices, in contrast to compactification procedure in `split`. Hence, the correct way is to go forward, starting from `begin()` and calling increment operator.

```
1  void merge(LeafNode &other) {
2      Iterator left_it = Iterator::begin(count+1u),
3      right_it = Iterator::begin(other.count+1u),
4      merge_it = Iterator::begin(count+other.count+1u);
5      // rearrange values in left node
6      for (auto i=0; i<count; ++i, ++left_it, ++merge_it) {
7          keys[*merge_it] = keys[*left_it];
8          values[*merge_it] = values[*left_it];
9      }
10     assert(left_it == Iterator::end(count+1u));
11     // insert values from right node
12     for (auto i=0; i<other.count; ++i, ++right_it, ++merge_it) {
13         keys[*merge_it] = other.keys[*right_it];
14         values[*merge_it] = other.values[*right_it];
15     }
16     assert(right_it == Iterator::end(other.count+1u));
17     // update counts
18     count += other.count;
19     other.count = 0;
20 }
```

Listing 8 Merge code

Lastly, rebalancing, which occurs when one of the nodes becomes underfull, starts with determining which node has a surplus and will be the "source" and which one has a deficit and will be the "destination". Destination node then moves its keys to make space for keys from the source, after which a part of keys from the source node are copied into those vacant places. Finally, the source node performs compactification. We omit to show code because it's a bit longer and doesn't introduce any new concepts.

It is obvious that all the update operations have the same time complexity as when keys are stored sorted – it remains $O(n)$, where n stands for number of keys. In fact, it can be shown that the expected number of shifts during inserts and erasures is still $n/2$, just as earlier.

However, there are two main problems with operations above. First, because of the convoluted layout, shifting the keys no longer follows nice sequential access pattern. Second, the iterator code which generates the indices is not very efficient as it contains several conditional jumps. It's also important to say that inserts and erasures are more critical to performance, since other functions are only rarely invoked.

As an alternative to the iterator logic as implemented above, one can try generate arrays of indices of various sizes during compile time and then use the one that fits the best (with smallest first out-of-bounds index not less than the exact one). During traversal, we would go through this array, skipping index values which are out-of-bounds. The greater number of these arrays we generate, the more precise fit we get and we have to skip less elements, but the more memory we use. Unfortunately, our experiments have showed that this approach doesn't beat the iterator logic even when the precision is high.

Finally, we should mention that we also implement update-optimized nodes which use Eytzinger layout in the manner described in subsection 4.2. However, this has no influence on algorithms except that everywhere we shift pointers instead of values.

6 Exploiting SIMD

So far, our endeavours have been focused on optimizing temporal locality and trying to minimize cache misses during lookups. Now we want to redesign our layout such that we can leverage the ability to perform multiple comparisons at once. The idea is to make nodes be the size of a cache line and then fit as many keys as possible inside, making it n-ary tree. Within the node, we use SIMD instructions to find a smallest key which is not less than the target key, and then continue search at the corresponding child node. Here again we will have to backtrack to the last node where we turned left, but unfortunately we can no longer use the neat bit-manipulation trick, but have to explicitly maintain this information. Although we don't need to, we again choose to start indexing at 1, so the root node actually has one key less than others. As a result, we can completely reuse the code for insert, erase, split, merge and rebalance. In fact, we only have to reimplement `lower_bound` and provide an iterator which yields indices for the new layout. [12] Each key slot can be identified either by its offset within key array k or by pair (i, j) denoting node index and offset within the node, two representations being connected by equation $k = i \times B + j$ where B is the number of keys that fit a single cache line and $0 \leq j < B$. Furthermore, j-th child of i-th node is the node with index $i \times (B + 1) + j$, which can be easily proved using mathematical induction. We will also refer to this node as a child of a key slot with coordinates (i, j) (when $j = B$ the parent is a fictitious key slot which contains ∞). Finally, Eytzinger layout can be seen as a special case when $B = 1$.

6.1 Lookups

We now turn to briefly discussing `lower_bound` implementation, shown in the Listing 9 below. We assume that keys are 64 bit integers [13]. We start by broadcasting the target key to a vector register, essentially filling it out with 8 copies. We also calculate the index of the last non-empty cache line. Starting from the root node, we load keys into another vector register and compare them against the target to obtain an integer mask with 1s on positions where they are lesser and 0s where they are smaller (line 8). We then count the number of trailing 1s to find the index j of the smallest key not less than target. If such a key exists, we get a number smaller than B, and we save its index (because it might be the actual lower bound). In any case, we use the result to calculate the index of the next node where we continue our search.

```
1  std::pair<uint32_t, bool> lower_bound(const KeyT &target) {
2      constexpr auto B = CACHELINE/sizeof(KeyT); // block size
```

[12] As long as we adhere to one-based indexing, this is all we have to do to add an arbitrary layout. It's worth noting that this choice also simplifies iterator logic.

[13] Unfortunately, there's no elegant way to cover arbitrary keys because of specific SIMD intrinsics

```
3       __m512i target_vec = _mm512_set1_epi64(target);
4       uint32_t N = count/B;
5       auto k=0u; // save last not-less-than target
6       for (uint32_t i=0, j, mask; i <= N; i += i*B+j) {
7           __m512i key_vec = _mm512_load_si512(&keys[i*B]);
8           mask = _mm512_cmplt_epi64_mask(key_vec, target_vec); // comparison
9           j = std::countr_one(mask);
10          k = j < B ? i*B+j : k; // save descent to the left
11      }
12      return std::make_pair(k, keys[k] == target);
13  }
```

Listing 9 SIMD version using AVX512 intrinsics

There are a couple of nuances worth emphasizing. First of all, we should not compare against the first key in the root node since our keys start from index 1. This can be solved by storing $-\infty$ (ie. smallest representable value for a given key type) and assuming such a key will never be a target. Second, in most of the cases last node is only partially occupied. This means that we either have to recognize when we compare against it and handle this case differently (for example, by pulling it out of the for loop), or somehow fill empty key slots such that we can apply a uniform procedure for all nodes. As can be seen from the code snippet, we choose the second option. We pad the keys with $-\infty$ so that in cases when target is bigger than all the integers in the last node, we can guarantee that we don't overwrite k[14].

6.2 Updates

As we said, using one-based indexing allows us to completely reuse the code of update operations for Eytzinger layout and it's only necessary to provide an iterator which implements in-order traversal of the tree. We show the code for increment operator in the Listing 10. Method `child()` returns an offset of the child node of the current key slot. Method `parent()` takes an index of the node and returns a pair (i, j) representing its parent key slot. To move forward, we first try to go to the child node of the next key slot (line 4) and then descend to the leftmost child as long as we don't go out-of-bounds. If that is not possible, we check if the next key slot is within bounds (of both whole array and current node) and if yes, we move to it. This is the most probable case as the majority of slots are leaves (approximately $B/(B+1)$ of them) and we use C++ attributes to instruct the compiler to generate assembly code which favours this case (so that there are no jumps on this path). Lastly, we climb up until we reach a non-fictitious parent slot.

```
1   EytzingerIterator& operator++() {
2       if (B+child() < n) [[unlikely]] {
3           // go to right child
4           k = B+child();
5           // then left as much as possible
6           while (child() < n) k=child();
7           return *this;
8       }
9       uint32_t j = k % B;
10      if (k+1u<n && j<B-1u) [[likely]] {
11          ++k;
12          return *this;
13      }
14      uint32_t i = k / B;
15      // climb up while rightmost child
```

[14] If we however insist to allow users to negative infinity, we can initialize `mask` to 0, change = to |= in line 8, and reset it to 0 on each subsequent iteration.

```
16      do {
17          std::tie(i,j) = parent(i);
18      } while (j==B);
19      k = i*B+j;
20      return *this;
21  }
```

Listing 10 Eytzinger iterator for non-unit node size

7 Evaluation

While our design may seem appealing in theory, it must demonstrate superior performance in practice compared to the standard solution to be deemed truly useful. In this section, we outline the benchmarking methodology we used to evaluate our work, present the obtained results and offer their interpretation.

7.1 Experimental setup

Our concurrent B^+ tree implementation, including both internal and leaf node implementation for all layouts, amounts to around 2500 lines of C++ 20 code. It works atop of a very simplified buffer manager which implements LeanStore API. We never do any I/O or evict pages; we support the notion of swips, but once they are swizzled, they are never unswizzled. Thus, all page accesses are very fast except for the first one.

The benchmark program itself is single-threaded and synchronization code (locking) is turned off in an attempt to get a clearer view about the influence of layout changes. The experiments were performed on a Linux system with an Intel Core i7-1065G7 CPU (1.30GHz, 4 cores, 8 hardware threads, AVX-512) and 16GB of main memory. Additionally, we developed a suite of 36 tests during the development phase to validate the correctness of our implementation.

7.2 Benchmark description

The program starts by inserting $N = 2^{22}$ key-value pairs into an empty tree in a random order, where keys are 64-bit integers and a single value is a block of 32 bytes (array of 4 64-bit integers). This is followed by $10 \times N$ lookups with target keys sampled from uniform or Zipf distribution ($\alpha = 1$). Furthermore, to evaluate range scan performance, we measured time needed to calculate a sum of values (more precisely, first 8 bytes interpreted as 64-bit integer) corresponding to a range of keys, averaged over different range lengths. Finally, benchmark completes with erasing all the keys from the B-tree in the same order they were inserted.

We ran these experiments for three layouts – standard, binary Eytzinger and n-ary Eytzinger – and their update-optimized variants using different page sizes[15]. The average page fill rate remains approximately 65% regardless of the page size or layout. Additionally, the total number of pages occupied by a tree decreases as the page size increases (which is expected since we insert constant number of keys).

7.3 Lookup performance

Bar plots in Figure 5 show both binary and n-ary Eytzinger layout improve lookup performance across all sizes, with the latter combined with vector instructions coming out as a clear winner. Regarding n-ary Eytzinger layout, we experimented with disabling SIMD to ascertain which factor – cache utilization or vector instructions – exerts a more significant influence on performance. We relied on linear search within a single Eytzinger node, which consists of 8 keys in our benchmarks[16]. Our findings revealed that the n-ary Eytzinger layout without SIMD instructions still performs consistently better than standard layout, while it is comparable to binary Eytzinger on average, being slower for smaller, but faster for larger page sizes. We attribute the latter

[15] Page size is directly proportional to the node capacity, which represents the maximum number of key-value pairs it can store.

[16] We cannot employ binary search without modifying our approach to pad keys with $-\infty$. However, given the relatively small size of the array in our case, this does not impose a significant penalty.

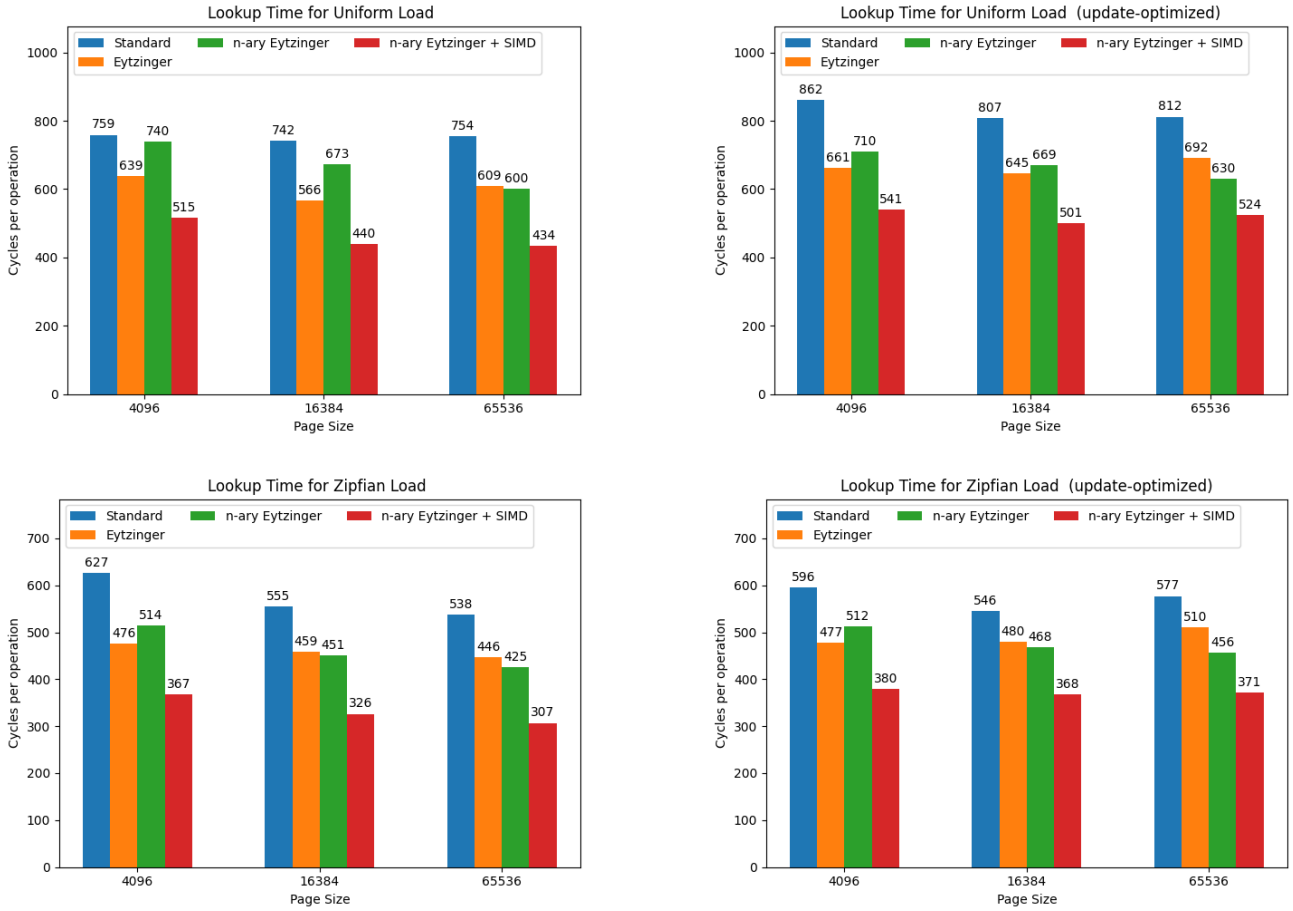


Figure 5 Comparison of lookup performance. The upper row of bar plots displays the average number of cycles per operation when target keys are sampled from a uniform distribution, while the lower row shows the performance when keys are sampled from a Zipf distribution ($\alpha = 1$).

observation to the fact that the difference between the number of cache lines needed to be fetched to facilitate lookups for binary and n-ary Eytzinger increases as leaf node capacity increases. Given that leaf nodes are not cached most of the time, this advantage supersedes the drawback of additional comparisons for large pages. We used `perf` profiling tool to count cache-misses and confirmed that Eytzinger layout really improves cache utilization. As Table 1 shows, n-ary Eytzinger outperforms the others by a significant margin – including binary Eytzinger, which comes in second, has 50% more cache misses.

Nevertheless, plots in Figure 5 unambiguously show that superiority of n-ary Eytzinger layout should be primarily ascribed to its ability to carry out parallel comparisons using SIMD instruction, which decreases the number of iterations by a factor of $\log_2 B$ compared to binary search, where B represents the number of keys per cache line (8 for our setup).

One intriguing observation from Figure 5 is that lookups are on average 8% slower in the update-optimized version. This is due to the fact that `lower_bound` must return the index of the value associated with a target key, which it needs to look up in the `pointers` array. Profiling reveals that this load often results in an expensive cache miss, as leaf nodes are rarely cached.

Layout	Cache misses
Standard	75851
Eytzinger	62102
n-ary Eytzinger	41272

Table 1 Number of cache mises triggered by `lower_bound`

7.4 Insert, erase and scan performance

Performance differences between layouts for inserts, erasures and range scans are depicted in Figure 6. As expected, update-optimized versions outperform regular ones in inserts and deletions, with the margin becoming more pronounced as the page size increases since they save more time by avoiding the movement of values. It's interesting to note that inserts and erasures remain competitive and sometimes even faster, especially with n-ary Eytzinger layout. This doesn't seem quite right at first because iteration with Eytzinger layout is more expensive than with standard one. However, memory/cache latency probably hides part of that cost because CPU can start incrementing/decrementing `lead` before it moved keys and values/pointers as these operations are independent. Furthermore, iterating through the n-ary Eytzinger layout is not significantly more expensive, as most of the time it involves moving to the next key within the node, which amounts to a simple increment/decrement operation. On the other hand, profiling with `perf` shows that iterating through binary Eytzinger layout incurs a lot of branch-misses hindering performance, as can be clearly seen for $64K$ pages. This behavior is anticipated because CPU has to predict which of the two equally probable branches will be taken during increment/decrement operation (recall code in Listing 4). In addition, each insert operation of a B-tree consists of multiple interior node lookups and one leaf node lookup, aside from actual insertion into a leaf node[17], enabling Eytzinger layouts to save some time due to their superior lookup performance. Again, profiling demonstrates that lookups amount to $20 - 30\%$ of the total execution time of insert operation, depending on the page size. Similar arguments apply for deletions as well.

While it may seem counterintuitive at first, range scans with the Eytzinger layout generally exhibit comparable performance to the standard layout. This observation may be surprising given that traversing the Eytzinger layout in-order lacks a nice sequential pattern. However, when the specified range R completely encompasses the key range S of a certain leaf node, that is $R \cap S = S$, we can process key-value pairs in arbitrary order, at least as long as the operation we apply is commutative. In such cases, we can traverse key-value pairs in the order they are "physically" stored, resembling the sequential memory access of the standard layout. Only in "boundary" nodes, where ranges partially overlap, do we resort to more expensive in-order traversal.

[17] Here we neglect splits and recursive insertions in interior nodes because they happen rarely.

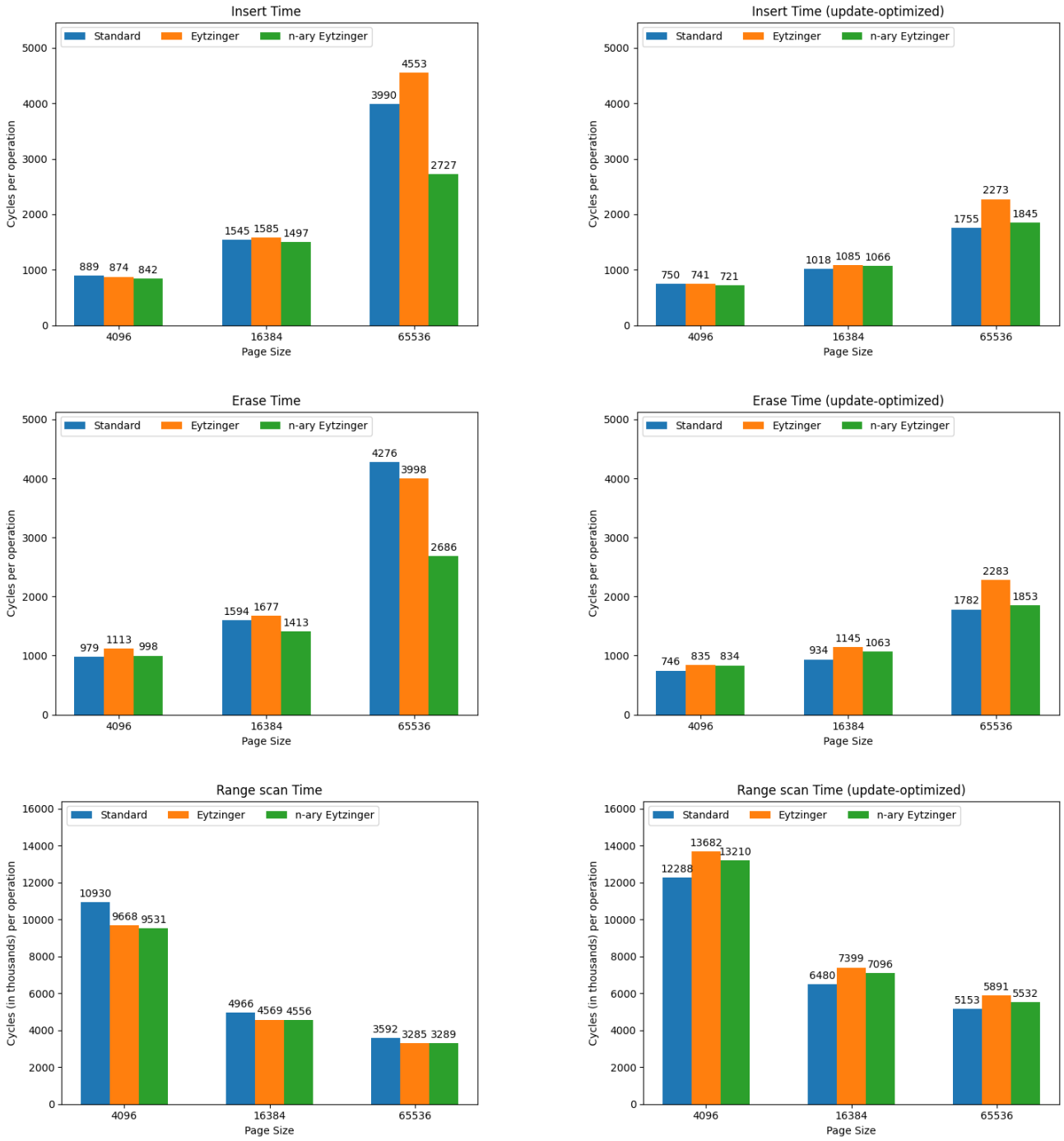


Figure 6 Insert, erase and range scan comparison. *n*-ary Eytzinger layout is able to match the performance of the Standard layout.

8 Summary

We have introduced innovative layouts for B-tree nodes along with associated algorithms aimed at more efficiently utilizing the memory subsystem and vector instructions available on modern CPUs. Our experiments demonstrate that while the binary Eytzinger layout offers faster lookup times compared to the standard layout, it compromises insert and delete performance due to slow in-order traversal and non-sequential memory access. However, the *n*-ary Eytzinger layout combined with SIMD instructions outperforms both the binary Eytzinger and Standard layouts. It achieves up to a 75% speedup over the Standard layout in lookups by optimizing cache utilization and leveraging SIMD instructions, while maintaining comparable insert and delete times.

References

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, page 107–141, New York, NY, USA, 1970. Association for Computing Machinery.

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

[3] Paul-Virak Khuong and Pat Morin. Array layouts for comparison-based searching, 2017.

[4] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. Leanstore: In-memory data management beyond main memory. *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 185–196, 2018.

[5] Joaquín M López Muñoz. Cache-friendly binary search, 2015.

[6] Thomas Neumann. Making unwinding through jit-ed code scalable - b-tree operations, 2022.

[7] Sergey Slotin. *Algorithms for Modern Hardware*, chapter Binary Search. 2023.