B^2-Tree: Cache-Friendly String Indexing within B-Trees.

Josef Schmeißer[1] Maximilian E. Schüle[2] Viktor Leis[3] Thomas Neumann[4] Alfons Kemper[5]

Abstract: Recently proposed index structures, that combine trie-based and comparison-based search mechanisms, considerably improve retrieval throughput for in-memory database systems. However, most of these index structures allocate small memory chunks when required. This stands in contrast to block-based index structures, that are necessary for disk-accesses of beyond main-memory database systems such as Umbra. We therefore present the B^2-tree. The outer structure is identical to that of an ordinary B+-tree. It still stores elements in a dense array in sorted order, enabling efficient range scan operations. However, B^2-tree is composed of multiple trees, each page integrates another trie-based search tree, which is used to determine a small memory region where a sought entry may be found. An embedded tree thereby consists of decision nodes, which operate on a single byte at a time, and span nodes, which are used to store common prefixes. This architecture usually accesses fewer cache lines than a vanilla B+-tree as shown in our performance evaluation. As a result, the B^2-tree answers point queries considerably faster.

Keywords: Indexing; B-tree; String

1 Introduction

Low overhead buffer managers are a fairly recent development which provide in-memory performance in case the data does fit into RAM [Le18; NF20]. However, database systems based on such a low overhead buffer manager still require efficient index structures which harness this new architecture. While systems like HyPer [KN11] could use pure in-memory based index structures, like the Adaptive Radix Tree (ART) [LKN13] or the more recent Height Optimized Trie (HOT) [Bi18], these are no longer an option for Umbra [NF20]. Pure in-memory index structures usually offer better performance than various B-tree flavors, yet their tendency to allocate small varying sized memory chunks limits their range of applicability.

With the presentation of LeanStore [Le18], Leis et al. revisited the role of buffer managers. LeanStore is a storage engine designed to resolve the overhead issues of traditional buffer management architectures [Ha08]. Its main feature is to abandon a hash table based pinning

[1] TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching josef.schmeisser@tum.de
[2] TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching m.schuele@tum.de
[3] Friedrich Schiller University Jena, Ernst-Abbe-Platz 2, 07743 Jena viktor.leis@uni-jena.de
[4] TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching neumann@in.tum.de
[5] TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching kemper@in.tum.de

doi:10.18420/btw2021-02

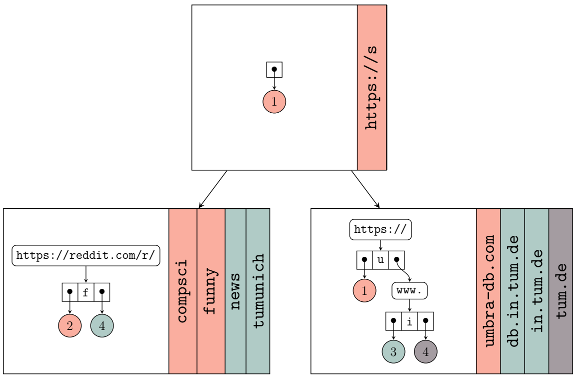


Fig. 1: The B^2-tree consists of decision nodes, similar to B+-tree nodes that contain separators and pointers to sub-nodes, and span nodes for common prefixes.

architecture, which buffer managers usually use, in favor of a technique called *pointer swizzling* [GUW09]. Umbra's buffer manager extends this concept by the ability to serve variable-sized pages with a minimum page size of 64 KiB [NF20]. This obviously affects the architectural requirements imposed on index structures. The imposed constraint precludes the use of most state-of-art pure in-memory based index structures. B-trees and their variations, on the other hand, fit well into Umbra's architecture. However, we found that even a highly optimized B+-tree implementation is no longer competitive, with regard to string indexing, in comparison to index structures like ART and HOT. Our B^2-tree operates on top of Umbra's buffer manager and provides significant throughput improvements over the original Umbra B+-tree.

Fig. 1 shows a small B^2-tree. It hosts an embedded tree per B-tree page. This embedded tree serves the purpose of directing incoming searches into narrowed down search spaces. A search on the embedded tree yields a pair of slot indices which define a span wherein a sought key may be found. Circular nodes point to the beginning of a search range, the upper bound. Each search space is also highlighted by the distinct coloring of its records and the corresponding node within the embedded tree.

Modern CPU architectures usually provide three layers of cache between their registers and main memory in order to mitigate the imbalance between CPU performance and main memory latency [SPB05]. Performing a naïve binary search over all the entries stored on a reasonably large B-tree page usually results in high lookup costs. This is especially the case when the B-tree page is used to store variable-sized records. One of the main reasons is the binary search's tendency to produce cache-unfriendly memory access patterns and its relatively high amount of branch mispredictions during the search [LKN13]. Some approaches try to mitigate these effects by using smaller nodes, often as small as a single cache line, which are optimized for cache hierarchies of modern processors [JC10; RR99; SPB05]. However, decreasing the page size down to the size of a single cache line may be infeasible or at least undesirable. Letting the storage backend handle such small pages would also lead to a considerable overhead. In the end, the choice of a certain page size will always be a trade-off.

We argue, that traditional index-structures for disk-based database systems can be adapted for beyond main-memory database systems. This work focuses on the development of an index structure based on the versatile B-tree layout. Thereby, we try to resolve the previously stated cache-unfriendliness of most B-trees variants. The presented approach hence aims to increase the number of successful cache accesses by applying data access patterns with higher locality. Our approach utilizes a secondary embedded index contained within each page. This secondary index is used to direct incoming searches to narrow down the search space within a given page. Consequently, fewer cache lines will be accessed during the search. We have chosen to retain the B+-tree [Co79] leaf layout, where keys are stored sequentially in accordance to their ordering. This allows us also to maintain the usual strength of B+-trees—their high range scan throughput.

This work's main contributions are:

- the B^2-tree, a disk-based index structure tuned for cache-friendly, page-local lookups,
- the adaption of radix trees to disk-based index structures,
- and a comparison to the already optimized Umbra B+-tree.

The focus of this work lies on the development of an index structure operating on given pages administered by Umbra's buffer manager. Concurrency is another aspect, our proposal utilizes an optimistic synchronization technique [Ch01], namely Optimistic Lock Coupling (OLC) [LHN19].

This work is structured as follows: Sect. 2 gives a summary of related work on modern index structures. Sect. 3 introduces the B^2-tree, which consists of the description of span and decision nodes as well as insertion and retrieval algorithms. Finally, Sect. 4 compares our proposed index structures to Umbra's B+-tree.

2 Related Work

While there has been constant development and research in the area of index structures, recent approaches mainly focus on main-memory database systems. Many of those index structures are therefore not designed to be used in conjunction with a paging based storage engine, however, their general design may still provide valuable insight.

There are a couple of proposals which aim to improve the cache-friendliness of B-trees. One of which is the Cache Sensitive B+-Tree (CSB+-Tree) [JC10]. Completely different approaches are the so-called Cache-Oblivious B-tree and the Cache-Oblivious string B-tree

[BDF05; BFK06]. Both proposals are based on an important building block, the packed-memory array (PMA). The PMA maintains its elements in physically sorted order, however, elements are not organized in a dense manner, instead, empty spaces will be deployed as necessary [BH07].

The String B-Tree is a B-tree specifically optimized to manage unbounded-length strings [FG99] while minimizing disk I/O. It is composed of Patricia tries [Mo68] as internal nodes where each Patricia trie node stores only the branching character. This architecture enables the use of a constant fanout independent of the lengths of the referenced strings since the Patricia trie leafs only store logical pointers. For this reason, searches within the String B-Tree have to progress optimistically. A search may thereby initially yield a result which does not match the queried key. By comparing the resulting string with the actual query, the length of the longest common prefix will be determined. This information is then used to find the corresponding node within the Patricia trie in question. From there on the correct path based on the actual difference between the resulting string and the queried key will be taken.

Additionally, the choice of a concrete binary search implementation also plays an important role. Index structures which depend heavily on binary search, like B-trees, require an efficient implementation thereof to achieve the best possible performance. Khuong and Morin suggest the use of their branch-free binary search implementation for arrays smaller than the size of the L2 cache [KM17].

Masstree is another key-value store that has mainly been designed to provide fast operations on symmetric multiprocessing (SMP) architectures [MKM12]. It stores all data in main memory, hence it is constructed to be used within the context of main-memory database system. Masstree's design resembles a trie [Br59; Fr60] data structure with embedded B+-trees as trie nodes.

3 The B^2-tree

The B^2-tree is a variation of the classic B-tree, its core structure is based on the B+-tree layout. We extend the existing layout by embedding another tree into each page, as emphasized by the name B^2-tree. The term embedded tree refers to this tree structure, it serves the purpose of improving the lookup performance while maintaining minimal impact on the size consumption as well as on the throughput of insert and delete operations. Our implementation also features some commonly known optimization techniques like the derivation of a shortened separator during a split [Ga18; GL01; Gr11].

Other approaches that combine or nest different index structures have already proved their potential. Masstree for instance showed considerable performance improvements [MKM12]. However, Masstree is not designed to be used in conjunction with paging based storage engines. Another point of concern is the direct correlation between the outer trie height and the indexed data. The inflexible maximum span length of eight bytes may lead to a relatively low utilization and fanout of the lower tree levels when indexing strings, this is usually caused by the sparse distribution of characters found in string keys. This is not unique to Masstree: ART's fanout on lower tree levels also decreases in such usage scenarios [Bi18]. B+-trees on the other hand feature a uniform tree height by design, since the tree height does not depend on the data distribution. Comparison-based index structures such as the B+-tree on the other hand are often outperformed by trie-based indexes in point accesses [Bi18].

Our approach intends to combine the benefits of both worlds, the uniform tree height of B+-trees with the trie-based lookup mechanics, while still featuring a page based architecture. Our trie-based embedded tree on each page serves the purpose of determining a limited search space where the corresponding queried key may reside. However, we still utilize a comparison-based search on these limited subranges. This design aims to improve the general cache-friendliness of the search procedure on each page.

3.1 The Embedded Tree

In the following we will present the inner page layout of our B^2-tree, the general outline can be observed in Fig. 2. As already mentioned, the general page organization follows the common B+-tree architecture, hence, payloads are only stored in leaf nodes. Leaf nodes are also interlinked, like it is originally the case in a B+-tree, in order to maintain the high range scan throughput usually achieved by B+-trees.

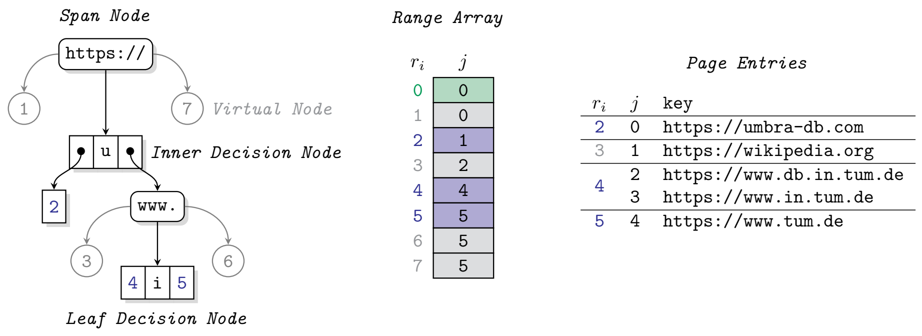


Fig. 2: The embedded tree structure together with an array responsible for translating the values stored in the embedded tree (the r_i) into search ranges where sought key-value pairs may reside. Its values are the exclusive upper bounds of offsets j for the rightmost table (page entries). The grayish virtual nodes are not part of the physically stored tree structure. Empty search ranges are omitted in the rightmost table. This table shows the complete form of the stored keys, without their associated payload.

The embedded tree itself is composed of a couple of different node types. First, we define the *decision node*, it acts like a B-tree node by directing incoming queries onto the corresponding

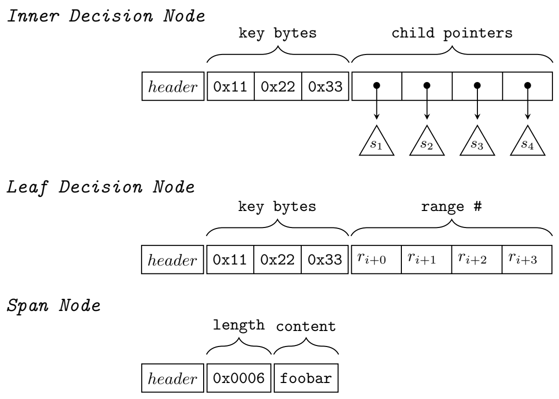


Fig. 3: Memory layout of all the embedded nodes deployed by B^2-tree. Each node contains a one byte large header. A flag inside the header determines whether a node contains pointers to subtrees or references to search ranges.

child. Probably the main difference to a B-tree node, is the fact that these nodes operate on a fixed size decision boundary represented by a single byte in our implementation, in contrast to B+-tree nodes, which usually operate on multiple bytes at once. We hence decompose keys into smaller units of information similar to how the trie data structure operates [Br59; Fr60]. Nodes of this decision type direct the search to the first child where the currently investigated byte of the search key is less or equal to the separator. The fanout of this type of node is also limited in order to improve data locality. Another similarity to B-tree nodes is the fact that they can be hierarchically arranged just like B-tree nodes. This node type bears some similarity to the *branch node* found in Patricia tries [Mo68]. However, Patricia's branch nodes only compare for equality, our decision nodes use the range of bytes to determine the position of a corresponding child. In Fig. 2 this type is illustrated as divided rectangular shape. Fig. 3 illustrates the memory layout for this node type. Note that, inner decision nodes and their leaf counterparts share the same layout, they just differ in the interpretation of their two byte large payloads. Leaf nodes terminate the search for a queried key even if it is not fully processed, the remainder of a queried key will then be further processed by the subsequent comparison based search.

The second node type we define are *span nodes*. These store the byte sequence which forms the longest common prefix found in the subtree rooted at the current node. Their memory layout is shown in Fig. 3. This node type can be compared to the *extension* concept of the Patricia trie [Mo68], however, span nodes have two additional outgoing edges to handle non-equality. Note that, by using an order preserving storage layout for the nodes, there is no necessity to store any next pointer within the span node, since the child node will directly succeed the span node. In Fig. 2 span nodes are illustrated as rounded rectangles. The deployment of span nodes is necessary to advance the queried key past the length of a span if the current subtree has a common prefix. At the following key depth, decisions, whether a queried key is part of a certain range, can be made once again by the deployment of decision nodes.

Obviously, the content of a span node does not have to match the corresponding excerpt of the queried key exactly. In case the stored span does not match, three scenarios can occur. Firstly, the size of the span may actually exceed the queried key. In that case the input will be logically padded with zero bytes. This may lead to the second case where the input is shorter. Any further comparisons with subsequent nodes are therefore meaningless. Hence, we introduce the concept of a virtual edge pointing from each span to its leftmost child, a so-called *virtual node*. To the edge itself we will refer as *minimum edge*. In Fig. 2 such an edge and its corresponding node is always colored gray to emphasize the aspect that it is not part of the physical tree structure. We follow this edge every time the input is less than the content of the span node. Note that encountering a fully processed input key implies that the minimum edge of a span node has to be taken. Fig. 2 illustrates the usage of this concept with the insertion of the Wikipedia URL after the construction of the embedded tree. This URL does not match the second span node, hence, it is delegated to the virtual node labeled “3”.

The last case where the input is greater than the span node’s content is completely symmetrical to the minimum edge situation. Therefore, a second virtual edge and node pair exists for every span node to handle the *greater than* case. We will cover the algorithmic details more elaborately in Sect. 3.2.

Fig. 2 also illustrates the *range array*, which stores the positions of key-value pairs. These define limited search spaces on the page. This array serves two purposes. First, it eliminates the need to alter the actual contents of the embedded tree during insert and removal operations, this simplifies modification operations significantly. Second, it enables the use of the aforementioned minimum and maximum edges.

During a lookup on a page this array is used to translate the output r_i of a query on the embedded tree into a position j on the actual page. Each lookup on the embedded tree itself yields an index into this array. This array, on the other hand, contains indices into the page indirection vector [Gr06], whereas the indirection vector itself points to any data that does not fit into a slot within the indirection vector [Gr06]. A resulting index thereby specifies an upper limit for the search of a queried key, whereas the directly preceding element specifies the lower limit. In Fig. 2 the annotated positions are colored differently in accordance to their origin. The very first position is colored green, this special element ensures that the lower limit for a search can always be determined. Indices originating from virtual edges are colored gray, whereas blue is used for regular positions. We denote these indices as r_i where i represents the corresponding position within the array of prefix sums. Each r_i occupies two bytes within each leaf node, the memory layout is illustrated in Fig. 3.

Insertion and removal operations, which are to be performed on the overlying page, also affect the embedded tree. More precisely, this affects the search range given by the embedded tree where the actual operation took place and all subsequent search ranges, since adjusting an upper boundary of one particular search range also implies that subsequent search ranges have to be shifted in order to retain their original limits. This is achieved by simply adjusting the values within the range array for the directly affected search range and every following search range.

3.1.1 Construction

One aspect we have not covered so far is the construction of the embedded tree structure. The construction routine is triggered each time a page is split or merged and also periodically depending on the number of changes since the last invocation.

The construction routine always starts by determining the longest common prefix of the given range of entries beginning at the very first byte of each entry. We will refer to the position of the currently investigated byte as *key depth*, which is zero within the context of the first invocation. On the first invocation, this spans the entire range entries on the current page. Based on the length of the longest common prefix a root node will be created. If the length of the longest common prefix is zero, a decision node will be created, else a span node. In the latter case, the newly created node contains the string forming the longest common prefix. Afterwards, the construction routine recurses by increasing the key depth to shift the observation point past the length of the longest common prefix.

The creation of a decision node is more involved, here we investigate the byte at the current key depth of the key in the middle of the given range. Subsequently, with the concrete value of this byte, a search on the entries right to that key is performed. This search determines the lower bound key index with regard to that value at the current key depth. In some cases, the resulting index may lie right at the upper limit of the given key range. For this reason, we also search in the opposite direction and take the index which divides the provided range of keys more evenly. This procedure is repeated on both resulting subranges until either the size of a subranges falls below a certain threshold or until the physical node structure of the current decision node does not contain enough space to accommodate another entry. Once a decision node is constructed, the construction routine recurses on each subrange, however, this time the key depth remains unchanged. This process is repeated until each final subrange is at most as large as our threshold value.

3.2 Key Lookup

On the page level, the general lookup principle is performed as in a regular B+-tree. The only difference is the applied search procedure. We start by querying the embedded tree which yields an upper limit for the search on the page records within the indirection vector. With the upper limit known, the lower limit can be obtained by fetching the previous entry from the range array. Afterwards, a regular binary search on the limited range of entries will be performed.

Querying the embedded tree not only yields the search range but also further information about the queried key's relationship to the largest common prefix prevailing in the resulting search range. The concrete relationship is encoded in `skip`, the stored value corresponds to the length of the largest common prefix within the returned search range. It also indicates that the key's prefix is equivalent to this largest common prefix. This information can be exhibited to optimize the subsequent search procedure by only comparing the suffixes.

Algorithm 1 Traversal of the embedded tree structure.

```
 1: function TRAVERSE(node, key, length, skip)
 2:     if ISSPAN(node) then
 3:         (exh, diff) ← CMPSPAN(node, key, length, skip)
 4:         if diff > 0 then
 5:             return MAXIMUMLEAF(node)
 6:         else if diff < 0  or exh then
 7:             return MINIMUMLEAF(node)
 8:         else
 9:             spanLength ← GETSPANLENGTH(node)
10:             key ← key + spanLength
11:             length ← length − spanLength
12:             skip ← skip + spanLength
13:             TRAVERSE(child, key, length, skip)
14:         end if
15:     else
16:         child ← GETCHILD(node, key, length)
17:         if ISLEAF(node) then
18:             return (child, skip)
19:         else
20:             TRAVERSE(child, key, length, skip)
21:         end if
22:     end if
23: end function
```

Algorithm 1 depicts a recursive formulation of the embedded tree traversal algorithm. It inspects each incoming node whether it is a span node or not. We compare the stored span with the corresponding key excerpt at the position defined by `skip`, in case a span node is encountered. The difference between the stored span and the key excerpt will be the result of this comparison. We also determine whether the key is fully processed in this step, meaning that the byte sequence stored within the span node exceeds the remaining input key. Three cases have to be differentiated at this point.

Firstly, the obtained difference stored in `diff` may be greater than zero, hence, the span did not match. However, this also implies that the remaining subtree cannot be evaluated for this particular input key. One of the outgoing virtual edges must therefore be taken. Implementation-wise, this edge is realized by a call to `MaximumLeaf`. It traverses the remaining subtree by choosing the edges corresponding to the largest values. The final result is thus the rightmost node of the remaining subtree.

The second case, where the excerpt of the input key is smaller, is mostly analog. However, the condition must now not only include the result, whether `diff` is smaller than zero, but also the result, whether the input key has been fully processed during the span comparison or not. An input key that is shorter than the sum of all span nodes, which led to the key's destination search range, will be logically padded with zeros. This leads to another interesting observation. Consider two keys with different lengths and their largest common prefix being the complete first key, all remaining bytes of the second key are set to zero. The index structure has to be able to handle both keys. However, from the point of view of the embedded tree, both keys will be considered as equal. This also implies that the embedded structure has to ensure that both keys will be mapped into the same search range. It is therefore up to the construction procedure to handle such situations accordingly. The subsequent binary search has to handle everything from there on.

The third and last case, where the key excerpt matches the span node, should be the usual outcome for most input keys. We obviously have to account for the actual length of the span to advance the queried key beyond this byte sequence. Hence, the point of observation on the key has to be shifted accordingly. This is also the case where `skip` is adjusted accordingly. It holds the accumulated length of all span nodes which were encountered during the lookup, or an invalid value if one of the span nodes did not match or more precisely if `diff` evaluated to a non-zero value. The subsequent call to either `MaximumLeaf` or `MinimumLeaf` thereupon returns an invalid value for the `skip` entry in the result tuple.

3.3 Key Insertion

We have already briefly discussed, how the insertion of new entries, affects the embedded tree, and its yielded results. Two cases have to be addressed. Either there is enough free space on the affected page to accommodate the insertion of a new entry, or the space does not suffice. A new entry can be inserted as usual if the page has enough free space left. However, this will also require some value adjustments within the range array in order to reflect the change. The latter case, where the page does not hold enough free space for the new entry, will lead to a page split. Splitting a page additionally results in roughly half of the embedded tree being obsolete.

For a simple insertion that does not lead to a page split, updating the embedded tree is trivial. We first determine the affected r_i in the range array where the insertion takes place. The updated search range is then defined by the preceding value and the value at r_i, which has to be incremented, since the search range grew by exactly one entry. In Fig. 2 these index values are denoted as j, and they are stored within the range array. However, this change must also be reflected in all subsequent search ranges. Therefore, all the following entries within the range array have to be incremented as well, in order to point to their original elements. By conducting this change, subsequent index values will then span all the original search spaces, which were valid up to the point where the insertion occurred.

The case where an insertion triggers a page split has to be handled differently. A split usually implies that approximately half of the embedded tree represents the entries on the original page whereas the other half would represent the entries on the newly created page. Consequently, the index values defining the search ranges of one page are now obsolete. Although, the structure could be updated to correctly represent the new state of both pages, we instead opted to reconstruct the embedded trees. This allows us to utilize the embedded structure to a higher degree, since the current prevailing state of both pages can be captured more accurately. Having a newly split page also ensures that roughly half of the available space is used. We can thus construct a more efficient embedded tree, which specifies smaller search ranges. In turn, smaller ranges can be used to direct incoming searches more efficiently.

3.4 Key Deletion

Deletion is handled mostly analogously. However, the repeated deletion of entries, which define the border between two ranges, may lead to empty ranges. This is no issue per se: The subsequently executed search routine just has to handle such a scenario accordingly. As it is the case with insertions, the deletion of entries also requires further actions. Directly affected search ranges have to be resized accordingly. Hence, the corresponding j values within the range array have to be decremented in order to reflect those changes. All subsequent values also have to be decremented in order to point to their original elements on the page.

3.5 Space Requirements

Another interesting aspect is the space requirement of the embedded tree structure. In the following we will analyze the worst-case space consumption in that regard. We start by determining an upper bound for the space consumption of a path through the embedded tree to its corresponding section of the page which defines a search range.

For now, we only consider the space required by the structure itself, not the contents of span nodes. The complete length of all the contents of span nodes forms the longest common prefix of a certain page section, which our second part of this analysis takes into account. Furthermore, a node in the context of the following first part refers to a compound construction of a decision node and a zero-length span node, this represents the worst-case space consumption scenario, where each decision node is followed by a span. Similar to the analysis of ART's worst-case space consumption per key [LKN13], a space budget $b(n)$ in byte for each node n is defined. This budget has to accommodate the size required by the embedded tree to encode the path to that section. x denotes the worst-case space consumption for a path through the embedded tree in byte. The total budget for a tree is recursively given by the sum of the budgets of its children minus the fixed space $s(n)$ required for the node itself. Formally, the budget for a tree rooted in n, can be defined as

$$b(n) = \begin{cases} x & \text{isTerminal}(n) \\ \sum_{c \in \text{children}(n)} b(c) - s(n) & \text{else.} \end{cases}$$

Hypothesis: $\forall n : b(n) \geq x$.

Proof. Let $b(n) \geq x$. We give a proof by induction over the length of a path through the tree.

Base case: The base case for the terminal node n, i. e. a page section, is trivially fulfilled since $b(n) = x$.

Inductive step:

$$\begin{aligned} b(n) &= \sum_{c \in \text{children}(n)} b(c) - s(n) \\ &\geq b(c_1) + b(c_2) - x && \text{(a node has at least two children)} \\ &\geq 2x - x = x && \text{(induction hypothesis).} \end{aligned}$$

Conclusion: Since both cases have been proved as true, by mathematical induction the statement $b(n) \geq x$ holds for every node n. □

An upper bound for the payload of the span nodes is obtained by assigning the complete size of the prefix of each section to the section itself. Assigning the complete prefix directly to a section implies that the embedded tree does not use snippets of the complete prefix for multiple sections, therefore, each span node has a direct correlation with a search range defined by the embedded tree. The absence of shared span nodes, thus, maximizes the space consumption for the embedded tree. An upper bound for the space consumption of the embedded tree is given by

$$\sum_{r \in \text{searchRanges}(p)} (l(r) + x)$$

where $l(r)$ yields the size of the longest common prefix of the search range r within page p.

We can therefore conclude that the additional space required by the embedded tree mostly depends on the choice of how many search ranges are created and the size of common prefixes within them. Our choice of roughly 32 elements per search range yielded the optimal result on all tested datasets, however, this is a parameter which may require further tuning in different scenarios. In our setting, the space consumption of the embedded structure never exceed 0.5 percent of the page. Note that, the prefix of each key within the same search range does not have to be stored, the B^2-tree may therefore also be used to compress the stored keys.

In the following we will analyze how modern CPUs may benefit from B^2-tree's architecture. Both AMD's and Intel's current x86 lineup feature L1 data caches with a size of 32 KiB, 8-way associativity, and 64-byte cache-lines. Our previous worst-case space consumption showed that the size of the embedded tree is mostly influenced by the size of common prefixes. The constant parameter x, on the other hand, can be set to 15, which is the size of a decision node and an empty span node. With the aforementioned setup of 32 elements per search range and a page size of 64 KiB, we can assume that the embedded structure, excluding span nodes, fits into a couple of cache lines, our evaluation also supports this assumption.

Efficient lookups within the limited search ranges are the second important objective of our approach. With the indirection vector being the entry point for the subsequent binary search, it is beneficial to prefetch most of the accessed slots. In our implementation, each slot within the indirection vector occupies exactly 12 bytes. Therefore, with 32 elements per search range, only six cache-lines are required to accommodate the entire section of the indirection vector. Recall that it is a common optimization strategy to store the prefix of a key within the indirection vector as unsigned integer variable. The B^2-tree, however, utilizes this space to store a substring of each key since the prefixes are already part of the embedded tree. We will refer to this substring as *infix*. It can also be observed that the stored infix values within the indirection vector are usually more decisive, since the embedded tree already confirmed the equality for all the prefix bytes. Overall, this implies that fewer indirection steps, to fetch the remainder of a key, have to be taken.

3.6 Concurrency

B^2-tree was designed with concurrent access via optimistic latching approaches taken into consideration. While this approach adapts well to most vanilla B-tree implementations, other architectures may require additional logic. This section covers all necessary adaptions and changes required by the B^2-tree in order to ensure correctness in the presence of concurrent accesses.

Optimistic latching approaches often require additional checks in order to guarantee thread safety. Leis et al. [LHN19] list two issues that may arise through the use of speculatively locking techniques such as OLC. The first aspect concerns the validity of memory accesses. Any pointer obtained during a speculative read may point to an invalid address due to concurrent write operations to the pointer's source. Readers have hence to ensure that the read pointer value was obtained through a safe state. This issue can be prevented by the introduction of additional validation checks. Before accessing the address of a speculatively obtained pointer, the reader has to compare its stored lock version with the version currently stored within the node. Any information obtained before the validation has to be considered as invalid if those versions differ. Usually, an operation will be restarted upon encountering such a situation.

Secondly, algorithms have to be designed in a manner that their termination is guaranteed under the presence of write operations performed by interleaving threads. Leis et al. discuss one potential issue concerning the intra-node binary search implementation as such. They note that its design has to ensure that the search loop can terminate under the presence of concurrent writes [LHN19]. Optimistically operating algorithms, therefore, have to ensure that no accesses without any validation to speculatively obtained pointers are performed and that termination under the presence of concurrent writes is guaranteed.

However, the presented traversal algorithm does not guarantee termination without the introduction of further logic. One main aspect concerns the observation that span nodes can contain arbitrary byte sequences. It is hence possible to construct a key containing a byte sequence that resembles a valid node. Such a node may also contain links pointing to itself. An incoming searcher may then end up in a cycle due to previous modifications performed by an interleaving writer which had conducted modifications to the embedded structure in said manner.

To prevent issues such as the one described, certain countermeasures have to be taken. We have to ensure that the traversal progresses with every new node. Furthermore, node pointers must not exceed the boundary of their containing page. We could have used the validation scheme presented by Leis et al. [LHN19]. This would require a validation on the optimistic lock's version after each node fetch. However, we can also use the fact, that in our implementation each parent node has a smaller address than any of its children. We furthermore have to ensure that each obtained node pointer lies within the boundary of the current page. Note that any search range obtained through the embedded tree is also a possible candidate leading to invalid reads. We hence have to ensure that each obtained boundary value also lies within the boundary of the currently processed page. Our binary search implementation, which will be performed directly afterwards, trivially fulfills the previously described termination requirement.

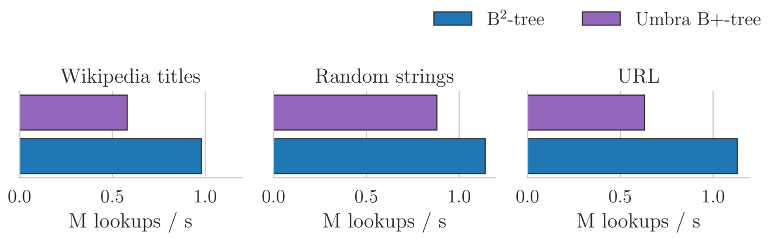
Insert and delete operations do not require any further validation steps, since they do not depend on any unvalidated speculative reads and exclusive locks will be held during such operations anyway.

4 Evaluation

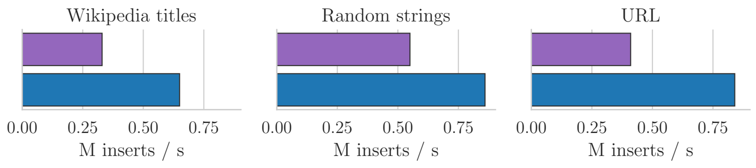
In the following we evaluate various aspects of our B^2-tree and compare them to the Umbra B+-tree. Note that the Umbra B+-tree is our only reference due to the lack of any other page based index structure capable of running on top of Umbra's buffer manager. In the following we analyze B^2-tree's performance as well as its scalability, the space requirements for the embedded tree, and the time required to construct the embedded tree.

4.1 Experimental Setup

All the following experiments were conducted on an Intel Core i9 7900X CPU at stock frequency paired with 128 GB of DDR4 RAM. Furthermore, index structures do not have



(a) Lookup throughput in million operations per second.



(b) Insert throughput in million operations per second.

Fig. 4: Single-threaded throughput comparison of the B^2-tree and the Umbra B+-tree grouped by the used dataset and imposed workload.

to access background memory, everything will be kept in main memory, unless otherwise stated. B^2-tree as well as the standard B+-tree have been compiled to use 64 KiB pages which is the smallest page size Umbra's buffer manager provides. The evaluation system runs on Linux with GCC 9.3, which has been used to compile all index structures.

Our reference will be the Umbra B+-tree as already stated. This particular B+-tree implementation uses some commonly known optimizations like the choice of the smallest possible separator within the neighborhood of separators around the middle of each page, and a data locality optimization where the first bytes of each key are stored within its corresponding entry in the indirection vector [GL01; Gr06; Gr11].

4.2 Datasets

We have used a couple of different datasets in our evaluation. Those datasets were chosen to resemble real-world workloads to a certain degree. Indexing of URLs and English Wikipedia titles[6] should resemble real-world scenarios. We also included a completely synthetic dataset consisting of randomly drawn strings, this dataset will be denoted as *Random* dataset.

[6] `https://dumps.wikimedia.org/enwiki/20190901/enwiki-20190901-all-titles.gz`

	distinct count	average length	median length	min length	max length
Wikipedia	48 454 094	21.82	17	1	257
Random	30 000 000	68.00	68	8	128
URL	6 393 703	62.14	59	14	343

Tab. 1: Parameters of the used datasets.

Tab. 1 summarizes some of the most important characteristics of the used datasets. The Random dataset is generated by a procedure which generates each string by drawing values from two random distributions. Thereby, the first distribution determines the length of the string which is about to be generated. Subsequently, the second distribution is used to draw every single character in sequence until the final destination length is reached.

4.3 Lookup Performance

In the following we will compare the point lookup throughput of our B^2-tree against our reference. The lookup benchmark queries each key from the randomly shuffled dataset which has been used for the construction of the index itself. Fig. 4a summarizes the results of our string lookup benchmark whereas Fig. 4b shows the influence of B^2-tree's more efficient lookup approach onto the insert throughput. B^2-tree's lookup throughput is roughly twice as high as that of its direct competitor. Keys in the URL and Wikipedia datasets often share large common prefixes, discriminative bits are therefore often not part of the integer field within the indirection vector of Umbra's B+-tree. In these situations, the B^2-tree has an advantage since the entries within the indirection vector are more likely to contain discriminative bits. The Random dataset, on the other hand, features very short common prefixes and a larger amount of discriminative bits between the bit string representation of keys. It is therefore not surprising that the performance gap between the Umbra B+-tree and our B^2-tree is smaller on this dataset.

	Approach	Inst.	IPC	L1D-Miss	LLC-Miss	BR-Miss
Random	B^2-tree	**1402**	0.39	**38.32**	**10.17**	**15.9**
	Umbra B+-tree	2519	**0.51**	44.63	20.02	19.84
URL	B^2-tree	**1839**	0.49	**45.69**	**11.35**	22.74
	Umbra B+-tree	3382	**0.51**	79.58	28.88	**16.15**
Wikipedia	B^2-tree	**1593**	0.38	**46.02**	**13.84**	**22.76**
	Umbra B+-tree	3147	**0.43**	61.7	30.82	28.22

Tab. 2: Performance counters per lookup operation. The best entry in each case is highlighted in bold type. B^2-tree mostly dominates the Umbra B+-tree which is in accordance with the previously discussed throughput numbers.

Recording performance counters during experiments usually facilitates further insights, Tab. 2 therefore contains an exhaustive summary. Comparing the averaged amount of instructions required per lookup between the B+-tree and our B^2-tree already reveals a considerable advantage in favor of the latter approach. This advantage also exists between the observed amount of L1 data cache misses (L1D-Miss) and last level cache misses (LLC-Miss), where the latter metric reveals that the standard B+-tree produces roughly twice as many misses. This is most likely related to the redesigned search procedure. Thereby, binary search is performed on a smaller search range. Furthermore, the contents of the infix fields within the indirection vector are usually more decisive than the contents wherein stored by the Umbra B+-tree. As a result, the comparison procedure, which will be invoked by the binary search procedure, can often refrain from performing any comparisons on the suffixes stored within the area where the remainder of the records are stored. This also reduces the total amount of cache accesses. For the B^2-tree one might expect fewer branch mispredictions, since the infix values are usually more decisive, however, the metric for the amount of mispredicted branches (BR-Miss) per lookup reveals no significant differences between both approaches. This is most likely the result of the additional logic performed during the lookups on the embedded tree.

4.4 Scalability

Additionally, to evaluating B^2-tree's single-threaded point lookup and range scan throughput, we also analyzed its scalability. We ran the same workload as in the single-threaded point lookup experiment. The results of this experiment are shown exemplarily for the URL dataset in Fig. 5. Note that we omitted the results for the remaining datasets due to them being very similar.

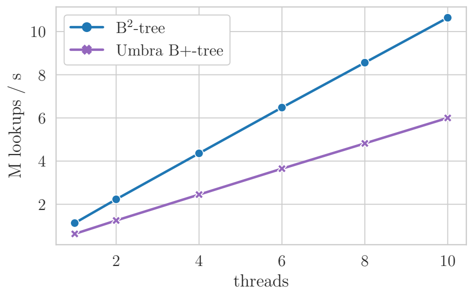


Fig. 5: Scalability on the URL dataset.

Also, the performance difference between our standard B+-tree and B^2-tree remains as the number of threads increases. Overall, the B^2-tree scales well for still being a B+-tree from an implementation point of view. This also correlates with previous work which did analyze the lookup throughput of B+-trees in combination with OLC [Le18; Wa18].

4.5 Throughput With Page Swapping

The experiment was set up as follows: in the first phase, all the keys of the Wikipedia titles were inserted into an empty index structure. If necessary, pages were swapped out into a temporary in-memory file by the buffer manager. In the second phase, the retrieval time for each key of the randomly shuffled input was measured.

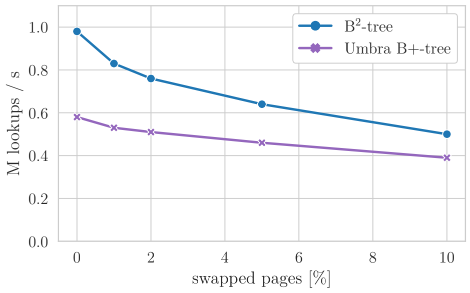


Fig. 6: Lookup throughput on the URL dataset with index structures utilizing Umbra's buffer manager.

Fig. 6 shows the results of the comparison between these two index structures in dependence of the percentage of swapped out pages. The B^2-tree outperforms the Umbra B+-tree for every tested percentage of swapped out pages. However, note that both curves eventually converge as the workloads become increasingly I/O bound.

4.6 Space Consumption

Another important aspect of the presented approach is the total amount of additionally required space on each page. Recall that we use 64 KiB large pages. We were able to fit the complete embedded tree structure in just a couple of hundred bytes as Tab. 3 affirms.

dataset	size [%]
Wikipedia titles	0.48
Random strings	0.49
URLs	0.52

Tab. 3: Averaged space consumption for the complete embedded tree in percent of the page size.

The space utilization of the embedded tree has therefore never been a source of concern in our point of view. However, it should be noted that the size of the embedded tree is variable, and that it will be influenced by the structure of the input data. Especially long shared prefixes have an impact on the overall space consumption of the embedded tree.

5 Conclusion

We presented the B^2-tree which speeds up lookup operations by embedding an additional tree into each tree node. The B^2-tree showed considerable performance improvements in comparison to an optimized B+-tree. This is related to the total number of instructions required per lookup, which in this case is lower than the number required by the Umbra B+-tree. Our B^2-tree, therefore, provides considerable improvements regarding the point lookup throughput. The overhead inflicted by the construction of an embedded tree during each page split is no point of concern as our experimental analysis showed. Furthermore, the additional space required for the embedded structure is mostly negligible, as our evaluation confirmed.

References

[BDF05] Bender, M. A.; Demaine, E. D.; Farach-Colton, M.: Cache-Oblivious B-Trees. SIAM J. Comput. 35/2, pp. 341–358, 2005.

[BFK06] Bender, M. A.; Farach-Colton, M.; Kuszmaul, B. C.: Cache-oblivious string B-trees. In: Proceedings of SIGMOD 2006. Pp. 233–242, 2006.

[BH07] Bender, M. A.; Hu, H.: An adaptive packed-memory array. ACM Trans. Database Syst. 32/4, p. 26, 2007.

[Bi18] Binna, R.; Zangerle, E.; Pichl, M.; Specht, G.; Leis, V.: HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In: Proceedings of SIGMOD 2018. Pp. 521–534, 2018.

[Br59] Briandais, R. D. L.: File searching using variable length keys. In: Papers presented at the the March 3-5, 1959, western joint computer conference. Pp. 295–298, 1959.

[Ch01] Cha, S. K.; Hwang, S.; Kim, K.; Kwon, K.: Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In: Proceedings of VLDB 2001. Pp. 181–190, 2001.

[Co79] Comer, D.: Ubiquitous B-Tree. ACM Comput. Surv. 11/2, pp. 121–137, June 1979.

[FG99] Ferragina, P.; Grossi, R.: The String B-tree: A New Data Structure for String Search in External Memory and Its Applications. J. ACM 46/2, pp. 236–280, 1999.

[Fr60] Fredkin, E.: Trie memory. In: CACM. 1960.

[Ga18] Galakatos, A.; Markovitch, M.; Binnig, C.; Fonseca, R.; Kraska, T.: A-Tree: A Bounded Approximate Index Structure. CoRR abs/1801.10207/, 2018.

[GL01] Graefe, G.; Larson, P.-Å.: B-Tree Indexes and CPU Caches. In (Georgakopoulos, D.; Buchmann, A., eds.): IEEE Data Eng. 2001. Pp. 349–358, 2001.

[Gr06] Graefe, G.: B-tree indexes, interpolation search, and skew. In: Workshop on Data Management on New Hardware, DaMoN 2006. P. 5, 2006.

[Gr11] Graefe, G.: Modern B-Tree Techniques. Found. Trends Databases 3/4, pp. 203–402, 2011.

[GUW09] Garcia-Molina, J. W. H.; Ullman, J. D.; Widom, J.: DATABASE SYSTEMS The Complete Book Second Edition." 2009.

[Ha08] Harizopoulos, S.; Abadi, D. J.; Madden, S.; Stonebraker, M.: OLTP through the looking glass, and what we found there. In: Proceedings of SIGMOD 2008. Pp. 981–992, 2008.

[JC10] Jin, R.; Chung, T.-S.: Node Compression Techniques Based on Cache-Sensitive B+-Tree. In: 9th IEEE/ACIS ICIS 2010. Pp. 133–138, 2010.

[KM17] Khuong, P.-V.; Morin, P.: Array Layouts for Comparison-Based Searching. ACM Journal of Experimental Algorithmics 22/, 2017.

[KN11] Kemper, A.; Neumann, T.: HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: Proceedings of ICDE 2013. Pp. 195–206, 2011.

[Le18] Leis, V.; Haubenschild, M.; Kemper, A.; Neumann, T.: LeanStore: In-Memory Data Management beyond Main Memory. In: Proceedings of ICDE 2018. Pp. 185–196, 2018.

[LHN19] Leis, V.; Haubenschild, M.; Neumann, T.: Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. IEEE Data Eng. Bull. 42/1, pp. 73–84, 2019.

[LKN13] Leis, V.; Kemper, A.; Neumann, T.: The adaptive radix tree: ARTful indexing for main-memory databases. In: Proceedings of ICDE 2013. Pp. 38–49, 2013.

[MKM12] Mao, Y.; Kohler, E.; Morris, R. T.: Cache Craftiness for Fast Multicore Key-value Storage. In: Proceedings of EuroSys 2012. Bern, Switzerland, pp. 183–196, 2012.

[Mo68] Morrison, D. R.: PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. J. ACM 15/4, pp. 514–534, Oct. 1968.

[NF20] Neumann, T.; Freitag, M. J.: Umbra: A Disk-Based System with In-Memory Performance. In: Proceedings of CIDR 2020. 2020.

[RR99] Rao, J.; Ross, K. A.: Cache Conscious Indexing for Decision-Support in Main Memory. In: Proceedings of VLDB 1999. Pp. 78–89, 1999.

[SPB05] Samuel, M. L.; Pedersen, A. U.; Bonnet, P.: Making CSB+-Tree Processor Conscious. In: Workshop on Data Management on New Hardware, DaMoN 2005, Baltimore, Maryland, USA, June 12, 2005. 2005.

[Wa18] Wang, Z.; Pavlo, A.; Lim, H.; Leis, V.; Zhang, H.; Kaminsky, M.; Andersen, D. G.: Building a Bw-Tree Takes More Than Just Buzz Words. In: Proceedings of SIGMOD 2018. Pp. 473–488, 2018.