RESEARCH-ARTICLE

B-Trees Are Back: Engineering Fast and Pageable Node Layouts

MARCUS MÜLLER, Technical University of Munich, Munich, Bayern, Germany

LAWRENCE BENSON, Technical University of Munich, Munich, Bayern, Germany

VIKTOR LEIS, Technical University of Munich, Munich, Bayern, Germany

Proceedings of the ACM on Management of Data, Volume 3, Issue 1 (February 2025)
https://doi.org/10.1145/3709664
EISSN: 2836-6573

B-Trees Are Back: Engineering Fast and Pageable Node Layouts

MARCUS MÜLLER, Technical University of Munich, Germany
LAWRENCE BENSON, Technical University of Munich, Germany
VIKTOR LEIS, Technical University of Munich, Germany

Large main memory capacity and even larger data sets have motivated hybrid storage systems, which serve most transactions from memory, but can seamlessly transition to flash storage. In such systems, the data structure of choice is usually a B-Tree with pageable nodes. Most academic B-Tree work considers only fixed size records, making them unsuitable for most practical applications. Given the prevalence of B-Trees, surprisingly few available implementations and benchmarks of optimized B-Trees cover variable-sized records. In this paper, we describe an efficient B-Tree implementation supporting variable-sized records containing six known node layout optimizations. We evaluate each optimization to guide future implementations, and propose an optimized adaptive layout that can even compete with pure in-memory structures for many workloads. Our results show that well-engineered B-Trees can efficiently handle both in-memory and out-of-memory workloads.

CCS Concepts: • **Information systems → Data layout**; **B-trees**; **Slotted pages**; **Variable length attributes**; **Fixed length attributes**; *Main memory engines*.

Additional Key Words and Phrases: Performance Analysis; B-Tree

ACM Reference Format:
Marcus Müller, Lawrence Benson, and Viktor Leis. 2025. B-Trees Are Back: Engineering Fast and Pageable Node Layouts. *Proc. ACM Manag. Data* 3, 1 (SIGMOD), Article 14 (February 2025), 26 pages. https://doi.org/10.1145/3709664

1 Introduction

Out-Of-Memory vs. In-Memory Indexing. Index structures are crucial for database performance, and for many decades, the B-Tree has dominated indexing [18]. The falling cost of main memory has caused the focus of research to temporarily shift from disk-optimized B-Trees to purely in-memory data structures such as ART [37], HOT [14], and Wormhole [55]. Now, with cheap and fast flash storage, stagnating main memory prices [29], and ever-growing data sets, out-of-memory index structures are becoming relevant again. A new generation of database systems [20, 21, 32, 35, 42, 50] aims to provide performance competitive with in-memory systems, while seamlessly transitioning to flash storage for larger working sets. Data structures developed purely for in-memory are unsuitable for this, as they do not support efficient paging.

Homecourt Advantage. Another reason to favor B-Trees over more recent in-memory structures is their long history of use in database systems. Many features desirable for database systems

Authors' Contact Information: Marcus Müller, Technical University of Munich, Munich, Germany, muelm@cit.tum.de; Lawrence Benson, Technical University of Munich, Munich, Germany, lawrence.benson@tum.de; Viktor Leis, Technical University of Munich, Munich, Germany, leis@in.tum.de.

ACM 2836-6573/2025/2-ART14
https://doi.org/10.1145/3709664

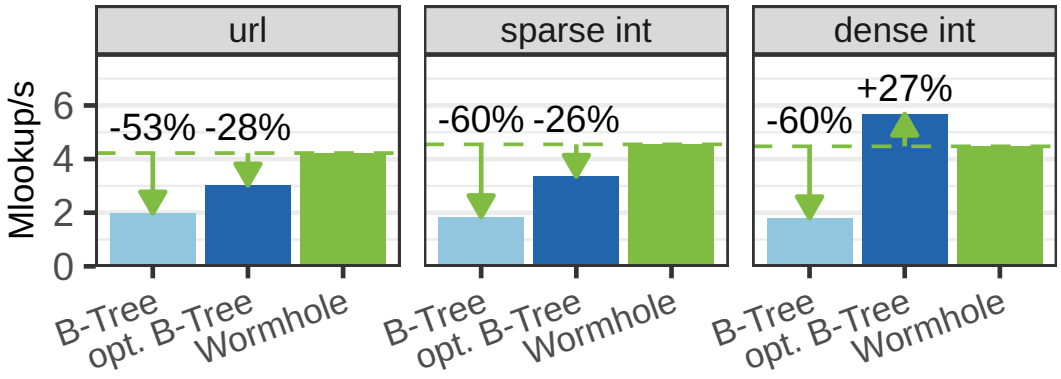

Fig. 1. Lookup throughput of B-Tree vs. Wormhole.

such as phantom protection and recoverability have well known solution for B-Trees as a result. Guidance on how to integrate B-Trees into database systems is widely available, for instance in Graefe's "Modern B-Tree techniques" [25] and its recent update [26].

Narrow Focus on Fixed-Sized Records. Given the prevalence and long history of B-Trees in databases, one might assume that detailed guidance for an efficient B-Tree implementation is readily available. However, this is not the case. A lot of research focuses on simple B-Trees where values are stored in an array in each node [16, 47, 53, 56]. These only work for keys with a fixed size, notably excluding strings. String keys are very common in database workloads [41, 51, 52]. A study of real world use of OLAP systems found around half of all values to be strings [52], many of which are used not merely as payload, but as a key. Given that indexes often span multiple key columns, many will contain at least one string column, making fixed-sized structures inapplicable.

Missing Focus on Basics. Surprisingly, there is also a general lack of research on the in-memory performance of B-Trees. Most modern research instead focuses either on special compute hardware or on optimizing I/O. The first group contains B-Trees utilizing persistent memory [17, 45, 58] and GPUs [10, 59]. The second group [29, 31, 48] is mostly spurred by the transition from disk to flash. However, the simplest case, in-memory performance of B-Trees on commodity hardware, remains neglected. One might also look for guidance in textbooks. However, these do not commonly contain a detailed performance evaluation, instead focusing on qualitative trade-offs. We provide both general advice and detailed information on the performance of specific techniques under specific conditions.

Optimizing In-Memory B-Trees. While the widespread use of B-Trees is motivated by their out-of-memory functionality, our evaluation focuses on in-memory performance for three reasons: First, with significant amounts of available memory and effective caching, many transactions can be served purely from memory. Second, modern flash offers tremendous I/O throughput (e.g., 12M IOPS on a single server [30]), making index performance a potential bottleneck even for out-of-memory workloads. Third, just like storage has changed, so have processors. The performance characteristics of today's processors are vastly different from the time when B-Trees last received major research focus [25, 39].

Evaluated Optimizations. The algorithmic logic of a B-Tree can conceptually be split into inter-node operations on the tree structure (split, merge, traversal) and intra-node operations that concern only a single page (insert, delete, lookup). This insight allows decoupling the former from the latter. In this work, we focus on the intra-node part, which is more affected by recent processor developments. Our optimizations could be combined with tree-level optimizations such as alternative node merging strategies [9, 23]. Specifically, we optimize the representation of nodes

in memory, as modern processors are often limited by memory access performance. We evaluate six optimizations: prefix truncation, heads, hints, fingerprinting, semi dense leaves and fully dense leaves. The first two are common and have been used for decades, the next two were proposed more recently. To the best of our knowledge, dense leaves have never been discussed in the literature, though similar approaches may be used in real world systems.

Closing the Gap. Based on our implementation of the above optimizations and a novel adaption strategy, we show that B-Trees are competitive with in-memory structures. As Figure 1 shows, a traditional B-Tree achieves less than half the lookup performance of the state-of-the-art in-memory Wormhole [55] data structure. Using the techniques described in this paper, we cut this gap in half, in the case of dense integer keys even surpass the state-of-the-art in-memory Wormhole data structure. This is achieved while keeping a fixed node size (4 KiB by default) and thus enabling fast and transparent caching using high-performance buffer managers based on pointer swizzling [28, 35, 42] or virtual memory [34].

Outline and Contributions. We start by defining the baseline B-Tree in Section 2 and discuss six optimizations in Section 3. We make the following contributions:

1) In Section 4, we perform an extensive evaluation to determine the impact of each of the six optimizations.
2) Based on our results, in Section 5, we present an adaptive B-Tree that selects the best node layout at runtime.
3) We benchmark several state-of-the-art in-memory indexes with respect to throughput and memory footprint and compare them with our B-Tree in Section 6.
4) In Section 7, we integrate our adaptive B-Tree into the multi-threaded storage engine vmcache [34] to show that the proposed data structure optimizations improve overall system performance in multi-threaded and out-of-memory scenarios.

Finally, we give a brief overview of related work in Section 8 and conclude in Section 9.

2 Background: Basic B-Tree

B-Tree. A B+-Tree is a comparison-based search tree where each path from the root node to a leaf node has equal length. In contrast to the original B-Tree [11], records are only stored in leaves. The inner nodes serve to guide the search to the correct leaf. For brevity, we will use the term B-Tree to refer to B+-Trees. Each node covers a specific range of the key space, with the root covering the full key range. For each inner node, its children partition its own range. The bounds of a node's range are known as its fence keys or fences [25].

Logical Node Structure. Each node maintains a sorted sequence of key-value-pairs. For leaves, these are the records stored in the leaf. For inner nodes, the values are pointers to the children. The key of each value is the upper fence of its associated child, or equivalently the lower fence of the next child. Logically, the keys lie between the children, so there is one more child than there are keys per node.

B-Trees using Templates. Many open-source B-Tree implementations use language features like templates or generics to support arbitrary fixed-size key types with some ordering relation. This includes the popular C++ library TLX [13], the Rust standard library [6], and public B-Tree implementations by Google for Go [2] and C++ [5]. However, many use cases require support for variable-sized keys such as strings. A generic B-Tree will usually resort to indirection to store these: The data is allocated on the program heap and a pointer stored in the tree. This hurts cache locality and promotes memory fragmentation.

Variable-Sized Records. To avoid these issues, many database systems store variable-sized records directly inside the tree. Instead of template type parameters, both keys and values are variable-length byte sequences. Many other key types can be converted to byte sequences while preserving ordering, a technique known as key normalization [25]. As records have variable size, there is no strict bound on the number of keys in each node.

Physical Node Structure. To store the variable-sized records, a slotted page layout as in Figure 2 is common. Each node consists of three parts, the *header*, the *slots*, and the *heap*. At the start of the node is its header, containing the metadata fields shown on the left-hand side of Figure 2. It begins with a 1 byte tag indicating the type of the node. This could be an inner node or leaf node, or it could be one of the alternative layouts discussed in Section 3. The field `upperChild` is used by inner nodes to store a pointer to the maximum child node. Likewise, `prefixLength` is used only if prefix truncation (Section 3.1) is enabled. The heap acts as a bump allocator for the variable-sized entries and fences. Between the header and the heap lies a variable-sized array of slots, whose size is stored in the `count` field. The slot array and heap grow toward each other in opposite directions. Each slot refers to an entry on the heap and stores its offset, key-length, and value-length as 16-bit integers. The slot array is sorted by key for efficient searching.

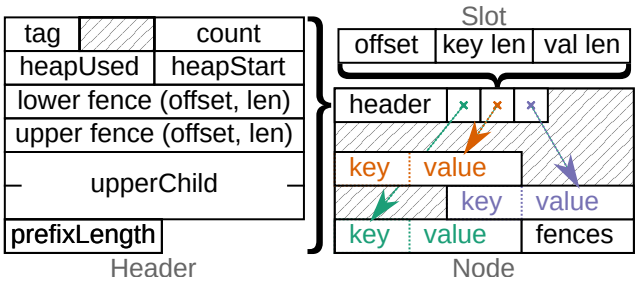

Fig. 2. Node with slotted page layout (right), its header (left), and a slot (top right). Each header line is 4 bytes.

B-Tree operations. We consider the following operations:

- `insert(key_ptr, key_len, val_ptr, val_len)`
- `lookup(key_ptr, key_len) -> (val_ptr, val_len)`
- `remove(key_ptr, key_len)`
- `scan(key_ptr, key_len, callback)`

The scan invokes the callback for records that are greater than or equal to the given key, in order, until the callback returns false.

Insert. To insert a record into the tree, first the appropriate leaf is located. Then, a new record is placed at the end of the unused region between slots and heap and a slot referencing it is inserted. If there is not enough space, the node is split and the insert retried. To split a leaf, a separator key is computed that divides the leaf near its median. This separator becomes a fence of both nodes resulting from the split. Then, the records are distributed to the two leaves and the separator along with a pointer to the newly allocated node is inserted into the parent. This may in turn cause the parent to split, cascading up the tree. Unlike leaves, splitting an inner node does not create a new separator. Instead, one of the keys in the node is not moved to either node resulting from the split, but to the parent. If the root node needs to be split, a new root is created with the old root as its only child before performing a normal split.

Separator Selection. When splitting a leaf, a separator must be chosen to divide the node. It is common practice to attempt to minimize the length of separator keys [12, 25]. Little motivation for this is provided in literature and given that these separators will later be used for prefix truncation (Section 3.1), a case could be made for instead trying to maximize common prefixes of adjacent separators. We will nevertheless follow this common practice. To this end, we consider a range of $\frac{\texttt{count}}{16}$ keys centered around the median. The longest common prefix length of this range is

determined. Then, the first byte past the common prefix is inspected for each key in the range. The first key to differ from the lowest key in this byte is used to form the separator. Any bytes past the differing byte are truncated to minimize length.

Remove. To remove a record, its slot is removed from the slot array. The data on the heap remains, though it is no longer referenced. The heap is compacted lazily during insertion. During compaction, all records are packed at the end of the heap and unreferenced records are discarded. The total size of live objects on the heap is tracked by the `heapUsed` field for this purpose. If two adjacent nodes have at least $\frac{3}{4}$ of their space unused after a remove, we merge them. Remove operations are similar to insert operations: Both need to locate a record and then shift half of all slots on average. They differ in that removes do not write to the heap, cannot cause compaction, and cause merges instead of splits. As remove operations perform very similarly to inserts and are comparatively rare in common workloads, we focus on lookups, inserts, and scans.

Multithreading and Out-Of-Memory. We use an unsynchronized in-memory implementation for most of our measurements. However, support for multithreading and larger than memory data sets is often needed in practice. For B-Trees both are achievable with relatively small overhead and implementation complexity. Optimistic lock coupling can be used to synchronize B-Trees efficiently [9, 36]. All our optimizations work on the node level and are orthogonal to this style of synchronization, which works on the tree level. B-Trees are naturally amenable to paging by using page-sized nodes and storing all values inline. In Section 7, we port our implementation to vmcache [34] to add support for both multi-threading and out-of-memory workloads and analyze overall system performance.

3 Optimizations

We present seven optimizations for the baseline B-Tree presented in Section 2. The optimizations are *prefix truncation, heads, hints, fingerprinting, semi dense leaves and fully dense leaves*. The first two are common and have been used for decades, the next two were proposed more recently. To the best of our knowledge, dense leaves have never been discussed in an academic publication, though similar approaches are likely used in some real world systems.

3.1 Prefix Truncation

Prefix truncation is a method of key compression almost as old as B-Trees themselves [12]. It builds on the observation that all keys within a node share the common prefix of its fences. Keys are stored with this prefix omitted, and search keys skip the prefix during comparisons. Figure 3 shows an example set of keys. The common prefix of fences `"https://"` is truncated. Even though all keys have an `"a"` following the prefix, it is not truncated, as the upper fence does not share it. The advantage is that local node operations never affect the prefix, which is determined during splits and merges. We store fences untruncated to simplify split and merge operations.

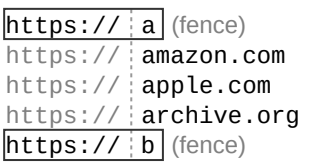

Fig. 3. Prefix Truncation.

Effects. A major benefit of prefix truncation is saving space. This reduces storage cost, and better utilizes caches. Computing the length of the common fence prefix is comparatively cheap and only occurs when nodes are created during splits and merges. The main cost is the increased implementation complexity of transferring keys between nodes, where it may be necessary to truncate or restore prefixes. At runtime this has negligible cost, as splits and merges are infrequent. During a workload of 100% inserts in random order, node splitting accounts for approximately 9% of CPU cycles.

3.2 Heads

During binary search within a node, often only the first few bytes of a key are compared. As the keys on the heap are stored in no particular order, each comparison entails a random memory access. By keeping a copy of the first 4 bytes of each key in its slot, cache locality is improved. The idea of storing a fixed size prefix for improved cache locality is described by Graefe as "poor man's normalized keys" [27], who attributes it to an even earlier use in the context of sorting [43]. We refer to this copy as the key's *head*. Heads are compared as integers in a single instruction instead of using `memcmp`. It is possible to omit this head from the key stored on the heap. However, doing so complicates the implementation significantly and slightly decreases throughput. We therefore choose to keep heads as copies at a slight expense to space use.

Slot Alignment. In addition to the heads, the slots contain three 16-bit integers to encode offset and key and value lengths. The resulting 10 byte size forces us to choose between adding 2 additional bytes of padding or storing the heads unaligned. The change in throughput between the two alternatives is less than 3% either way and depends on the specifics of the workload. We use a packed representation with unaligned access for its better space efficiency as we found the performance impact of unaligned accesses to be negligible.

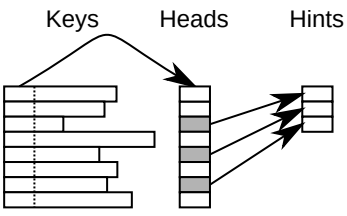

Fig. 4. Heads and Hints.

3.3 Hints

Figure 4 shows the heads optimization along with the next optimization, the hint array. Heads from evenly-spaced slots are copied into a fixed-size array in the node header. The hint at position i in the hint array is taken from slot $\left\lfloor \frac{\texttt{slotCount}}{\texttt{hintCount + 1}} \right\rfloor * (i + 1)$. This divides the slot array into $N + 1$ partitions using N hints. A linear search on the hints is used to narrow down the starting range of the subsequent binary search on the heads. This way, the number of cache misses can be reduced substantially. In line with LeanStore [8], where this technique was first proposed, we use a hint array spanning 16 hints, or 64 bytes. After modifications, the hint array needs to be updated. We update only the entries at or after the insertion point when possible. If the spacing of sampled entries $\left\lfloor \frac{\texttt{slotCount}}{\texttt{hintCount + 1}} \right\rfloor$ changes, the entire array is recreated.

3.4 Fingerprinting

FPTree [45] uses one byte hashes of keys (*fingerprints*) to efficiently locate candidate positions for records in unsorted leaves. It targets persistent memory, where keeping records ordered is expensive. We present an alternative leaf layout based on this idea. Figure 5 shows an example node. An array of fingerprints is stored on the node's heap. To perform a point lookup, we compute the key's hash and search the fingerprint array linearly using SIMD comparisons.

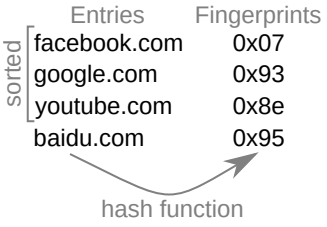

Fig. 5. Fingerprinting Leaf.

Sorting. As range scans must return entries in order, the node is sorted lazily before each scan. This introduces the possibility of scan operations modifying nodes, which may complicate concurrent implementations. To avoid unnecessary sorting, we partition the array of entries into a sorted and an unsorted range. Insertions append to the unsorted range and thus avoid shifting entries. Before each `scan`, the entries of the unsorted range (if any)

are sorted and merged into the sorted range. When sorting, we should apply the same permutation to the fingerprint array, as we do to the slot array. However, sorting routines that operate on entries split across multiple memory locations are not widely available. We use `std::sort` on the slot array without rearranging the fingerprints and recompute the fingerprint array from scratch. In addition to scans, sorting can also be triggered by node splits. While these technically only need partitioning, sorting takes up only a small fraction of total run time.

Prefix Truncation and Hashing. The truncated prefix is excluded from hashing, which reduces the amount of data that needs to be hashed, but requires recomputing fingerprints when the prefix length changes, e.g., during splits.

3.5 Dense Leaves

The idea behind dense leaves is to implement the node as plain array of values if possible. While the keys of the B-Tree are arbitrary-byte sequences, they are often produced from applying key normalization to a tuple of column values. If the last column of the tuple holds dense integers, e.g., when it is an ID column, frame-of-reference encoding can be used to represent keys as offsets from some reference key. We then use these offsets to index an array and locate the matching record without any key comparisons. This approach is restricted to key sets that are sufficiently dense, as space is otherwise wasted on arrays with most slots left empty. Therefore, one needs to use a second type of leaf that can handle the general case of sparse keys. One ends up with a B-Tree where only the leaves holding dense keys use the more specialized dense layout, while the others use the fallback. While frame-of-reference encoding in B-Trees has been previously discussed in academic publications [15, 24], we could not find any that combine it with such an array representation.

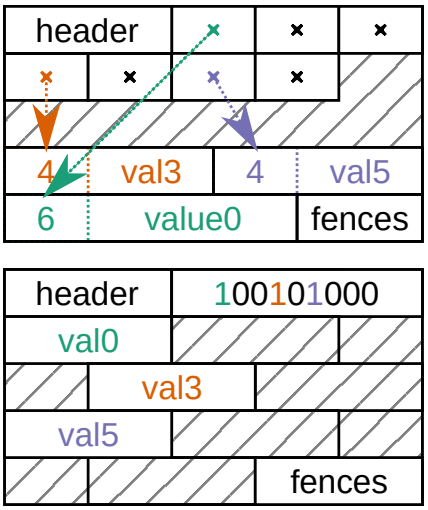

Fig. 6. Semi dense leaf (top) and fully dense leaf (bottom) with 3 records each.

Numeric Part Extraction. As we do not handle tuple keys, but normalized keys, we cannot apply frame-of-reference encoding directly to the last component. Instead, we treat the whole key as a large integer. This works because normalized keys are created by concatenating the normalized representations of each individual field. For efficient implementation, we impose further restrictions on the set of keys: All keys within the node must have equal length and differ only in their four least significant bytes. This allows us to determine offsets based only on the last four bytes, which can be loaded directly as a big endian integer. Keys shorter than four bytes are treated as if they contained additional zero bytes at the start. In principle, more tail bytes could be considered. However, this complicates the logic for handling shorter keys for little gain. We refer to the four least significant bytes of a key as the *numeric* part of the key. The remainder is said to be the *non-numeric* part. We emphasize that keys in different dense leaves may still have different sizes and the B-Tree as a whole does not become limited to a fixed key size. A practical example of this are keys formed by concatenating a string and an integer column. In this case, if there are sufficiently many integer values for a given string, then there must exist a leaf where all keys differ only in the integer part and therefore have equal length.

Offset Computation. We map the numeric parts to offsets by subtracting the numeric part of the reference key. Before locating a key in this way, we must verify its non-numeric part matches using

the lower fence. The reference key of a node is the minimum possible same-length key greater than the lower fence. We store its numeric part in the node header.

Variants. We propose two implementations of this concept: semi dense leaves and fully dense leaves. Both variants require that the lower fence key contains the entirety of the non-numeric part and that all keys have equal length and share the non-numeric part. Either variant has additional conditions. The creation of a dense node is attempted whenever a comparison-based leaf would otherwise be split. If all the layout's preconditions are fulfilled, and the set of present records fits (the offsets must not be too large), the node is converted. The resulting node may still have to be split after conversion to accommodate the newly inserted key.

Semi Dense Leaves (SDLs). This is a type of slotted page layout. The top half of Figure 6 shows an example. One slot is allocated for each possible offset between the fence keys. Each holds a single 16-bit integer, pointing to a length-prefixed value on the node's heap. Zero is used to represent the absence of a record. To compute the number of required slots, the SDL needs the upper fence to share the non-numeric part of the lower fence.

Space Use. Assuming insertion of random keys, we expect SDLs to appear within the tree as soon as they are more space efficient than the comparison-based leaves. With an average value size of 8 bytes and a truncated key size of 2 bytes, this occurs when 20% of possible (equal length) keys are present. At this density, a fully utilized 4 KiB SDL has around 1000 slots and holds 200 records, about the same as a full comparison-based leaf would. With 100% dense keys, around 330 records are possible.

Fully Dense Leaves (FDLs). This layout additionally assumes that all values have equal size, as might be the case for a secondary index storing tuple identifiers as values. With this assumption, values can be stored in an array with no additional indirection. With 8 byte values and dense keys, around 500 records can be stored per 4 KiB node. The bottom half of Figure 6 shows an FDL. A bitmap directly after the header stores, for each slot, if a record with the corresponding key is present. Most of the remaining space is used to store the array of values. Slots in the array for which no record is present do not hold meaningful data.

Dense Leaf Split. The set of keys that can be stored is uniquely determined by the lower fence key together with the sizes of keys, values, and upper fence. As a result, it does not make sense to adjust the number of slots depending on the range between the fence keys, like the semi dense layout does. Instead, we create FDLs even when their capacity is less than the number of keys that lie between the fence keys. This allows us to efficiently handle the insertion of sorted keys, where dense leaves will fill up starting from the lower fence. For instance, a node might be able to fit 400 entries and have fence keys ('wiki', 12) and ('wiki', 3000). The slots in the array would then correspond to the keys ('wiki', 13) — ('wiki', 412). If the key ('wiki',500) were inserted into the tree, it would be routed to this node. As there is no slot available for this key, the node would have to be split. This split, however, is special in that it does not distribute any keys to the new node. As each key has its own slot, the presence of one key has no effect on the ability to store another. Therefore, splitting at the median to evenly distribute records makes no sense. Instead, a new empty leaf is created to handle keys past the last available slot and the upper fence of the dense leaf is adjusted. Depending on how far past the maximum slot the new key lies, we create either another dense leaf or a general purpose leaf. To be able to change the upper fence of a dense leaf without touching the keys, we allocate enough space at node creation to hold an upper fence with the same length as the keys of the node. Theoretically, this strategy can result in a tree with only one record per leaf. This can be mitigated by converting back to a comparison-based leaf instead of splitting if too few records are present.

Partition Detection. This pattern handles the insertion of dense keys in ascending order efficiently. A closely related case is the insertion of keys from multiple non-overlapping ranges, where keys

within each range are inserted in order. In practice this occurs when keys are generated from multiple sources and each source draws sequential IDs from a different range. With the baseline separator selection strategy of splitting nodes near the median, the node at the boundary of two such ranges can often not use the dense layout, as it contains keys from both ranges. Even as the boundary node is split repeatedly by insertion of keys in the lower range, some keys from the upper range always remain in the boundary node. Thus, the good insert performance of FDLs would be lost. We therefore implement an alternative separator selection algorithm when using FDLs. Whenever a comparison-based leaf is to be split, we find the maximum split point such that the lower of the two new nodes could use the dense layout. If this split point is greater or equal to the median, we use it. Otherwise, we proceed with normal separator selection. Repeatedly applying normal separator selection on the boundary node quickly moves the boundary past the median. Thus, this scheme reliably separates ID generation ranges.

4 Evaluation

In this section, we evaluate the proposed optimizations by comparing B-Tree configurations with different sets of optimizations enabled. We benchmark seven different combinations of optimizations. Starting from the baseline presented in Section 2, we cumulatively add prefix truncation, heads, and hints. Three more configurations are obtained by applying all the above and one of the alternative leaf layouts: fingerprinting leaves, semi dense leaves (SDLs), or fully dense leaves (FDLs). We evaluate each optimization by comparing the pair of configurations that differ only in the inclusion of that optimization.

Experimental Setup. For each configuration, we run a synthetic workload with keys from one of four sets and 8 byte values, representing tuple identifiers. The key sets we use are: URLs averaging 62 bytes length (`urls`), Wikipedia titles averaging 23 bytes (`wiki`), 32-bit integers $[0, N)$ (`dense`), and random 32-bit integers (`sparse`). We chose the number of records such that the total data size is 300 MB. In each benchmark, we first insert 90% of the records, then benchmark insertion on the remaining 10%. Insertion order is random. We then run 5 million lookups and scans each on keys chosen according to a Zipfian distribution ($p \propto \frac{1}{k^\alpha}$). Following YCSB [19], we use $\alpha = 0.99$. For each scan, the number of records requested is chosen uniformly from [1,50]. We run each configuration five times and present median values. We perform our evaluation on an AMD Ryzen 9 7950X with frequency boost disabled at 4.5Ghz. The system has 32 KiB of L1, 1 MiB of L2, and 32 MiB of L3-cache available to each core and runs Linux 6.2.0. In the following, we dedicate one subsection to each optimization.

4.1 Prefix Truncation

We first compare prefix truncation with the baseline implementation. The top row of Figure 8 shows the increase in throughput. For most key sets, the change is minor. For `urls`, which often share long common prefixes, there is a larger increase. The effect is greatest for scan operations. This is expected, as the number of records per leaf directly influences how many leaves need to be scanned. For `urls`, the average number of records per leaf increases from 36 to 96. The increased density reduces the frequency of L1-cache misses by 35% for scans and 20% for inserts and lookups. We do not observe a consistent decrease in CPU instructions per operation, indicating that skipping prefix comparison has at best a minor effect. In addition to increasing throughput, prefix truncation reduces space use by 7 — 64%. Figure 7 shows the reduction for each key set.

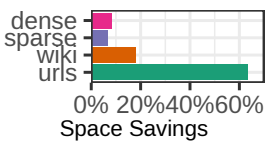

Fig. 7. Space reduction for each key set.

As we see, `urls` is again affected the most. The trend of `urls` being most affected aligns well with the intuitive expectation for URLs to share long common prefixes due to their hierarchical nature. To confirm this intuition, we conduct another experiment using the same data sets. We record the average length of the truncated prefix after insertion of all keys. We find them to be 49 bytes for `urls`, 7.0 bytes for `wiki`, 2.3 bytes for `dense`, and 1.5 bytes for `sparse`. This aligns well with our findings on throughput and space use.

4.2 Heads

Next, we evaluate key heads by comparing against the configuration with only prefix truncation enabled. The center row of Figure 8 shows the increase in throughput.

Point Operations. For lookups and inserts, throughput increases by 16 — 64%. The cost of maintaining heads during inserts is outweighed by improved cache locality. The increase is greatest for the two integer key sets. These keys fit completely within the heads, so comparison of the full keys on the heap is almost always avoided. As a result, we see both CPU-instructions and cache misses per lookup reduce by around half for integers. The effect for the strings is smaller, though still generally positive. Heads are more effective for `wiki` than for `urls`, because the truncated `wiki` keys are more likely to differ within their first bytes. Out of all pairs of adjacent keys within leaves, 38% have distinct heads for `urls`, while 61% do for `wiki`. For inner nodes, the difference is even greater (35% vs. 71%). In the root node, the heads are completely useless for `urls`, as all keys begin with `http`.

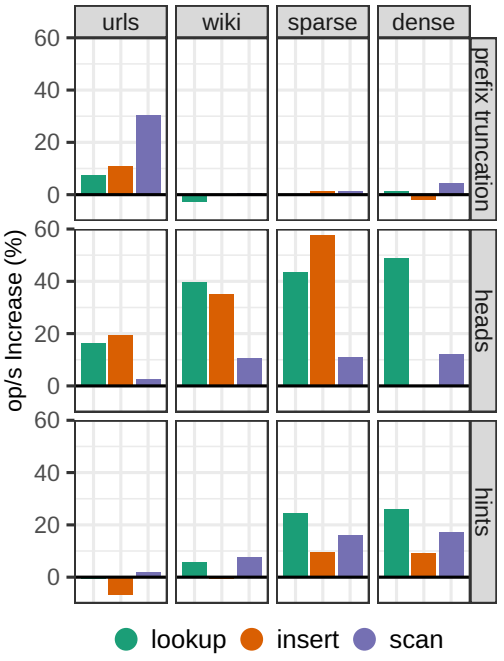

Fig. 8. Throughput increase of prefix truncation, heads, and hints. Each optimization is applied on top of the previous.

Scans. For scans, heads provide less of a benefit. Performance improves by 3 — 12%. This is because locating an entry, which is where heads help, is only a small part of the scan process. While iterating through the entries of a leaf, the heads provide no benefit. In fact, the space overhead of heads reduces the average number of records per leaf by 12 — 19%, slowing down iteration. This detracts from the performance gains of improved cache locality.

Space Use. While heads generally help performance, this comes at the cost of space use. Figure 9 shows the relative space overhead of heads for each key set. For `dense`, each record uses an additional 4.5 bytes. For all other key sets, the increase is around 5.7 bytes. The smaller increase for `dense` is an artifact of uneven tree growth. For random insertions, leaf node splits in a B-Tree occur in waves and consequently the load factor in leaves fluctuates [23]. Without heads, a wave of node splits occurs near the end for `dense`, decreasing the space efficiency of the prefix truncation configuration.

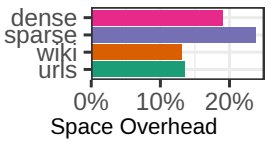

Fig. 9. Space overhead of heads for each key set.

> **Insight 1:** Heads increase throughput by up to 64% at the cost of space use. Their efficacy is affected strongly by the type of key used.

4.3 Hints

The bottom row of Figure 8 shows the change in throughput obtained by applying the hint array on top of the two previous optimizations. Like the heads optimization it is based on, it is most effective for the integer key sets. Lookup performance for integers is boosted by 25 — 26%. For strings, the change ranges from -0.6 to +6%. Like for heads, the efficacy of the hint array depends on how reliably keys can be distinguished by their starting bytes. The average size of the narrowed search range varies from 7% (`dense`) to 30% (`urls`) of entries. Low selectivity and the cost of updating the hint array result in reduced insert performance for the string keys. The number of CPU instructions per insert operation increases by around 20%. For integers, this cost is outweighed by improved cache locality. The number of L1 misses per insert is reduced by 30% for integers but only by 17% and 8% for `wiki` and `urls` respectively.

Downsides. In contrast to heads, the space overhead of the hint array is marginal at 1.5 — 2%. This is consistent with the ratio of hint array size to page size. Therefore, unless a workload is dominated by inserts, a hint array should be employed. In-place updates need not update the hint array and would not incur a performance penalty for string keys.

4.4 Fan-Out

More space efficient layouts not only reduce the overall number of nodes in the tree, but also increase the number of children inner nodes can have, further reducing the height of the tree. Before we move on to leaf specific optimizations, we consider the effect the previous optimizations have on fan-out. Average fan-out in the tree fluctuates substantially as records are inserted, just as the load factor of leaves does. We therefore conduct another experiment where we scale the number of records inserted by 16^{-R} with uniformly distributed $R \in [0, 1)$. Prefix truncation roughly doubles average fan-out for `urls`, increases it by 10% for `wiki` and `dense` and by only 1% for `sparse`. Heads decrease fan-out by around 8% for `urls` and by 15 – 20% for the remaining data sets. Hints decrease fan-out by less than 2%.

4.5 Fingerprinting Leaf

We next compare fingerprinting leaves against comparison-based leaves with heads and hints. Note that both configurations use heads and hints in inner nodes, i.e., the difference is only in the leaves. Figure 10 shows the change in throughput from using fingerprinting leaves. Fingerprinting is beneficial for strings but detrimental for integers. Both of these facts can be attributed to cache misses. For strings, the fingerprint array produces far fewer cache misses, as only entries with matching fingerprint cause a heap access. As a result, lookup throughput increases by 13 – 22%.

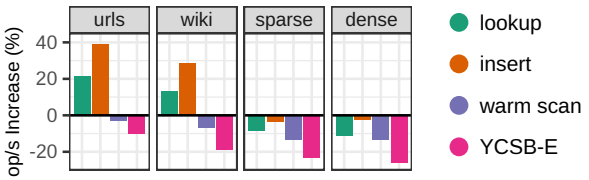

Fig. 10. Change in throughput with fingerprinting leaves.

Integer Keys. For integer key lookups, the fingerprinting array produces more cache misses. As the records are small, many entries fit in each leaf and the fingerprinting array itself spans many cache lines. Given the larger number of records, more fingerprint collisions are to be expected.

With the comparison-based leaf, the heads and hints lead to much fewer cache misses than for the strings.

Scans. Scan performance decreases for all key sets due to two reasons: lazy sorting and the lack of heads and hints. To separate these two effects, we replace the scan benchmark with two variations: *warm scan*, where we sort all leaves before the benchmark, and YCSB-E [19], which consists of 95% scans and 5% inserts. We see that throughput decreases for both, though to different degrees. This confirms that both factors contribute to the slowdown. The decrease due to lack of heads and hints (measured by warm scan), is larger for integer keys.

Insert. Not having to maintain ordering improves insert performance. Appending an entry accesses only the end of the slot array, while the comparison-based leaves need to bring half of all slots into cache on average. A leaf with hint array on average holds 84 records with `urls` and 130 records with `dense`. With a slot being 10 bytes, 5 bytes per record need to be moved on each insertion on average. For string keys, the performance advantage of fingerprinting is roughly doubled compared to lookups. For integers, the performance disadvantage is mostly negated.

Space Use. Although fingerprinting leaves are not universally faster than comparison-based leaves, the omission of heads makes them more space efficient. They save about 4 bytes per record. This amounts to space savings of 9 — 10% for string keys and 15% for integers.

> **Insight 2:** Fingerprinting leaves allow up to 20% faster lookups than heads and hints for string keys. For integers they are around 10% slower, though they save 15% space.

4.6 Dense Leaves

In this section, we evaluate the two dense leaf layouts, again by comparing against the hints-array configuration. For string keys and sparse integers, leaf nodes never reach sufficient density to be converted to a dense layout, so we end up with the same tree as we would with the hints configuration. The additional checks come at a slight performance cost, throughput decreases by around 1%. We only consider dense integers for the remainder of this section.

Key Set Density. In practice, even sequentially generated keys are often not perfectly dense. This has little effect on the other leaf types, but is substantial for dense leaves. Therefore, we generalize our `dense` key set. We use N integers, chosen uniformly from the range $[0, M)$ and call $\frac{N}{M}$ the *density* of the key set. We fix N at 25 million and vary density from 4% to 100% in 2% increments. Again, we report medians of five runs. Figure 11 shows the increase in throughput over hints depending on density. In general, both variants become more effective as density increases.

Lookup. Both variants increase throughput substantially if density is sufficiently high. At 100% density, throughput increases by 71% over hints with FDLs and by 37% with SDLs. SDLs first achieve a 10% throughput increase at 28% density, while FDLs do so only at 44%. The reason FDLs are effective only at higher density is that vacant slots are more costly for these, each using the same amount of space as an occupied slot. FDLs surpass SDLs at 58% density. The point at which this occurs depends on the size of payloads, with larger payloads being more favorable to SDLs.

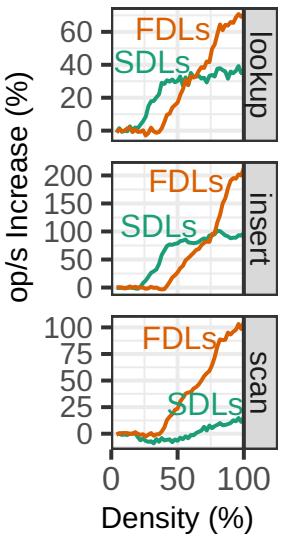

Fig. 11. Throughput increase of dense leaves over hints depending on key density.

Insert. As with lookups, FDLs achieve a higher maximum throughput but become effective only at higher density. At 100% density, throughput increases by 213% with FDLs and by 97% with SDLs.
Scans. For scans, FDLs increase throughput by up to 105%, while SDLs only do so by at most 16%. At lower densities, SDLs decrease throughput by up to 9%. This poor performance is caused by frequent branch misses. The maximum of 30 misses per scan is reached at 42% density. In comparison, hints and FDLs average at most 11 misses per scan at any density. The branch misses are caused by the record callback being conditionally invoked for each slot, depending on if it is occupied. Using SIMD, these could likely be reduced substantially.

> **Insight 3:** Dense leaves work well for dense integer keys without compromising performance for other keys. If possible, the fully dense variant should be used.

Space Use. In addition to increased throughput, dense leaves also reduce space use. Figure 12 shows the space use per record depending on density. Either kind of dense leaf uses less space than hints and becomes more efficient as the key set becomes denser. Again, we see that higher density is more favorable to FDLs. Compared to hints, space use is reduced by 52% for FDLs and by 40% for SDLs at 100% density.

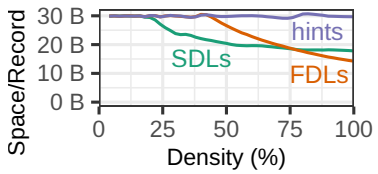

Fig. 12. Space use depending on density.

Partitioned Sequential Insert. We consider a special insertion pattern modeling 64-bit IDs generated sequentially from multiple non-overlapping ranges (*partitions*). Keys are generated by first choosing uniformly at random from one of N partitions.
For each partition, we generate sequential IDs using a separate counter, beginning at zero. The key is constructed by concatenating the 32-bit partition and counter IDs. We vary the number of partitions and generate a total of 10 million keys. The keys are inserted in order of generation, i.e., keys within each partition are inserted in order, but keys from different partitions are interleaved.

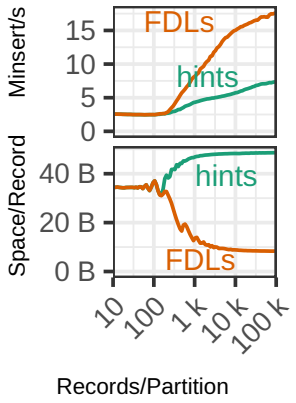

Fig. 13. Insert throughput and space use per record depending on the number of records per partition

SDLs. SDLs are ineffective in this benchmark. The upper fence of the maximum leaf of each partition typically lies in the subsequent partition or is close to it. The node's range is therefore too large to create an SDL. As all insertions happen on the last leaf of a partition, no dense leaf is ever created. FDLs are specifically designed to avoid this issue by converting nodes even when the dense layout cannot support the full key range. Moreover, sequential inserts results in very compact trees, as the split point is chosen such that the left node is fully utilized. This alternative approach to managing search ranges and splitting is natural for FDLs. As the fill factor of the node does not affect its capacity to hold other entries, the typical pattern of splitting nodes around the median does not make sense. Although SDLs do not require this approach like FDLs do, it could be implemented to improve sorted insert performance.

> **Insight 4:** Dense leaves that do not over-commit on their search range are ineffective for sorted insertions.

Table 1. Impact of optimizations on lookup performance.

	urls	wiki	sparse	dense
Prefix Truncation	+	≈	≈	≈
Heads	+	++	++	++
Hints	≈	≈	+	+
Fingerprinting	++	++	-	-
Dense Leaves	≈	≈	≈	+++

Fully Dense Leaves. Figure 13 shows insert throughput and space use per record depending on the number of records per partition for hints and FDLs. We see that for small partitions below around 200 keys, both configurations perform identically. This is because the partitions are too small to convert any leaf to a dense layout. As we increase the number of keys per partition, throughput improves for both configurations. This is due to improved cache locality, as only one leaf per partition needs to remain in cache to handle the insertion of new keys. The relative performance advantage of FDLs over hints rapidly increases between 200 and 10k keys per partition up to a maximum of around 2.7 times higher throughput. It then slowly declines as the number of active leaves becomes small enough to avoid cache misses even for hints. The right-hand side plot shows the space use per record. As a greater fraction of leaves uses the more compact dense layout, space use drops. For hints, space use increases as the partition size increases. This is because ordered inserts lead to a lower fill factor than random inserts with our near-median splitting strategy. With many small partitions, insertion order is practically random.

4.7 Discussion

We have shown that the performance impact of each optimization depends both on the set of keys used and the operations considered. Table 1 summarizes the impact of each optimization on lookup performance for each data set. Throughput changes for other operations are qualitatively similar. The exception to this are fingerprinting leaves, which perform better for inserts and worse for scans. As a result, they harm scan performance for all key sets, and do not significantly lower insert throughput for any key set. The first three optimizations (*prefix truncation, heads, hints*) are generally beneficial and have little downsides for in-memory performance. However, the optimal leaf layout depends heavily on the workload and data to make any general recommendations. To still benefit from all optimizations, we propose an adaptive node layout in the following section.

5 Adaptive B-Tree

As our experiments show, there is no universal best set of optimizations. Manually configuring a B-Tree for a particular use case, while effective, is often not practical. Many database systems have additional information available that may be used for automatic configuration. For instance, an index with string keys may use fingerprinting leaves, while one with integer keys uses dense leaves. However, the information available varies by system. For instance, SQLite tables are dynamically-typed [7], i.e., a column may contain multiple different types of values. We propose automatic configuration without complicating the abstraction offered by the tree.

5.1 Concept and Implementation

We present a scheme to automatically use a suitable leaf type depending on the keys stored in a node and the relative frequency of scan operations. Comparison-based leaves with hint arrays, fingerprinting leaves, and fully dense leaves are considered. Figure 14 summarizes the possible transitions between leaf types. As dense leaves improve throughput by a great degree when they

are applicable, they should be used whenever possible. The core difficulty the adaptive B-Tree addresses is choosing between fingerprinting and comparison based leaves. As we previously found, fingerprinting leaves speed up point accesses on string keys but reduce performance for integer keys and scans. Therefore, this choice depends both on the keys held by the leaf and the operations performed on it. We address these two challenges using two separate mechanisms: key-adaption and operation-adaption.

Key-Adaption. Key adaption is performed only during node splits and merges. These operations are infrequent enough that analyzing the set of keys in the leaf does not significantly reduce overall performance. On the other hand, the nature of keys stored changes only with insert and remove operations, which will typically trigger splits or merges and therefore key-adaption. Moreover, both key-adaption and splits and merges involve copying records between pages, so it is convenient to perform both at the same time. If remove and insert operations balance out, it is possible for the set of keys in a leaf to be entirely replaced without a single split or merge. We consider this scenario unlikely in practice. It could be addressed by additionally performing key-adaption as a background activity.

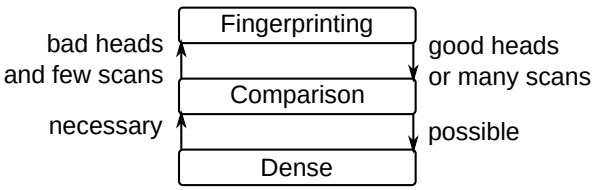

Fig. 14. Leaf layout transitions.

Detecting String Keys. Key-adaption should choose the fingerprinting leaf if keys are strings. With the four key sets we use, it is clear which keys are strings and which are not. However, to automatically pick a layout given arbitrary keys, we must first define which keys should be considered strings. We found two plausible explanations as to why keys being strings matters. First, integer keys have unique heads, which makes comparison-based search more efficient. Second, integer keys are smaller and therefore allow more records per page. This increases the cost of the linear search on the fingerprint array. To decide how to best handle large records with mostly unique heads, we conduct an experiment with integer keys and large payloads. We find that the hint array outperforms fingerprinting, therefore we use the uniqueness of key heads as the criterion. To evaluate head quality, we consider the number of pairs of adjacent keys which have equal heads. If this number does not exceed $\frac{1}{16}$ of the total record count, the heads are considered *good*. Strict inequality is used for this threshold, so that in cases where there are less than 16 keys and no collisions, the comparison based layout is used and a transition to dense leaves is possible. Benchmarking thresholds from $\frac{1}{64}$ to $\frac{16}{64}$ of records suggests that the exact value used has little effect on performance for our key sets and is robust. When averaging throughput over all key sets and operations, the best performing threshold is only 0.6% faster than the worst.

Operation-Adaption. If our keys are found to be string-like, the choice of layout comes down to the frequency of scan operations. We find that for string keys, fingerprinting leaves increase the time per scan by three times more than they reduce the time per lookup (averaging the differences in time per operation for both text key sets). Therefore, we consider scans "common" for this purpose, if they occur at least one third as often as point operations. To determine the relative frequency of scan operations, we introduce a one byte counter into both comparison-based leaves and fingerprinting leaves. It is incremented with probability 15% at each scan operation and decremented with probability 5% at each point operation. The counter ranges from 0 to 3 and saturates at either end of this range. Changing the probabilities of adjustment by a factor of 0.5× to 2× does not change performance significantly, nor does increasing the range of the counter up to 8, i.e., both are robust.

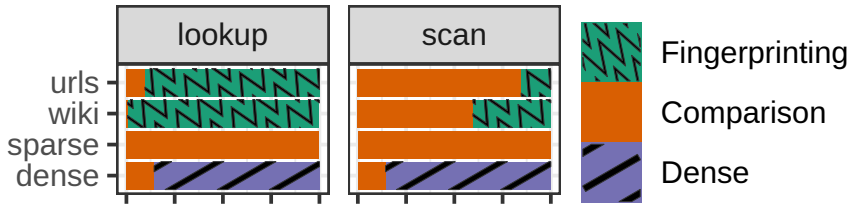

Fig. 15. Distribution of leaf types for each key set and workload.

Counter Details. The value of the counter influences layout choice during splits and merges, but it also triggers immediate conversion if the counter reaches the end of its range. This ability to perform adaption on reads is necessary, as the workload could transition from lookup-only to scan-only without a single write operation occuring. If the conversion is not possible due to space constraints, it is aborted, so read operations never modify the tree structure. Recall that the frequency of scans only matters if keys are string-like. Thus, a counter triggered conversion must be inhibited if they are not. To this end, the counter is enabled or disabled during splits and merges depending on the result of key-adaption. The counter is disabled by setting it to 255. The saturating increment and decrement operations use a single unsigned or signed comparison respectively to honor both the bounds of the counter range and this sentinel value. During splits and merges, fingerprinting leaves are used if the value of the counter is less than 2. That is, scans are rare, and the counter is not disabled due to good heads.

5.2 Evaluation

We repeat the experiment from Section 4 with the adaptive B-Tree.

Leaf Distribution. Figure 15 shows the distribution of leaf types depending on key set after each of the two reading workloads. The implementation correctly selects the type of leaf we would expect to perform best in each situation. The occurrence of fingerprinting leaves on `urls` and `wiki` after scan is due to incomplete adaption. At the beginning of the scan workload, the operation counters indicates that scan operations are rare. As the counters are incremented only with low probability, converting all leaves to comparison-based leaves takes many operations. After 50 million operations (rather than 5 million), fingerprinting leaves make up less than 0.01% of nodes. The use of comparison-based leaves for lookups on `urls` is due to genuinely good heads, such as from IDs or timestamps in URLs.

Throughput. Figure 16 shows the change in throughput compared to fully dense and fingerprinting leaves for each key set and operation. In all cases but one, the adaptive B-Tree achieves at least 98% the throughput of the better of the two configurations. The exception is scans on `wiki`, where throughput is 8% lower due to incomplete adaption. If 50 million operations are performed instead, the throughput gap shrinks to only 2%.

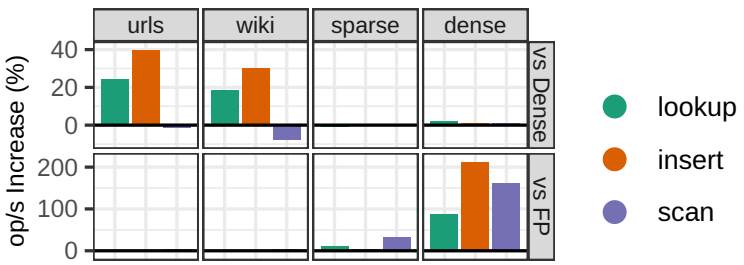

Fig. 16. Throughput increase of adaptive B-Tree vs fully dense (Dense) and fingerprinting leaves (FP).

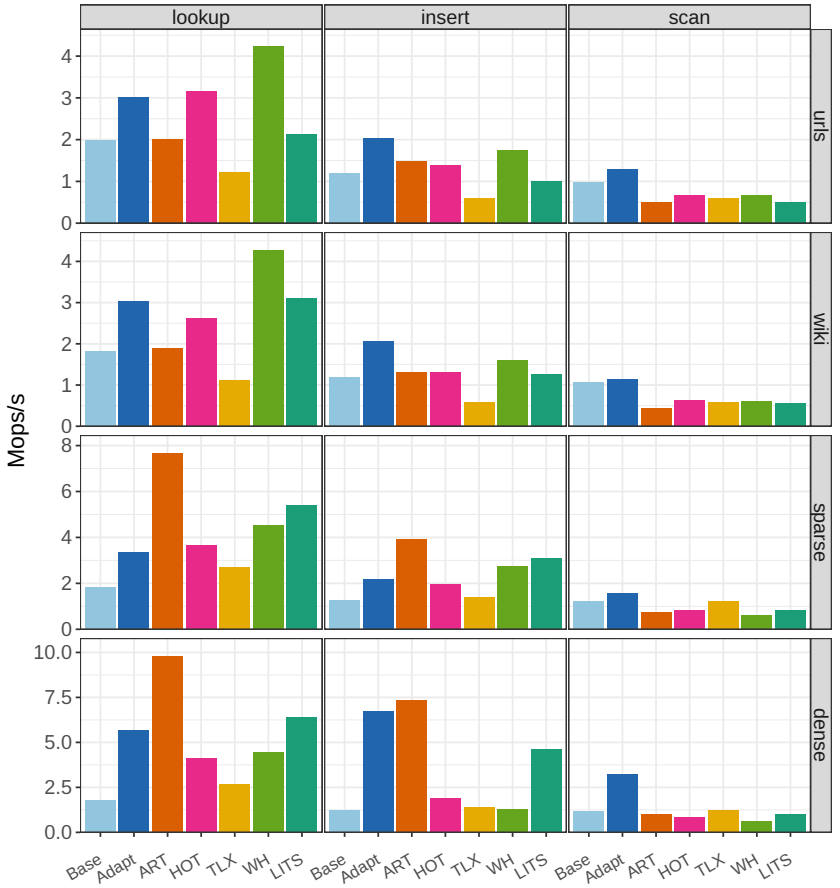

Fig. 17. Throughput of baseline B-Tree, adaptive B-Tree, ART, HOT, TLX's B-Tree, Wormhole (WH), and LITS.

6 B-Tree vs. in-memory structures

In this section, we compare the baseline and adaptive B-Tree against four state of the art in-memory data structures, ART [37], HOT [14], Wormhole [55], and LITS[57]. We chose these for their support of variable sized keys and scans and the availability of open source implementations. As they are specific to in-memory workloads (not supporting paging), we expect them to outperform the baseline B-Tree. We expect to close the gap with our optimizations. To demonstrate that variable size key support is a crucial feature, we also compare against the B-Tree implementation of the popular TLX[13] library, which supports fixed size keys only. We expect TLX to be comparatively slow for strings. Figure 17 shows the throughput for each data structure.

6.1 ART and HOT

ART [37] and HOT [14] are radix trees. Like B-Trees and unlike hash-based approaches, they support efficient range scans. They are generally thought to perform better than B-Trees for lookups. The improved lookup performance coupled with the ability to perform scans makes them strong choices for in-memory systems. We use the single threaded versions of both data structures.

Tuple storage. ART and HOT are not primarily intended for storage, but for use in a secondary index. Both store a 63-bit value for each key and require that the full key can be computed given

the value. Usually, tuple identifiers are stored and storage of tuples is delegated to some other data structure. Being able to extract keys from values is required because both store only parts of the keys in some cases. The lookup on the tree is performed optimistically with the assumption that the key is present. If the key is not present, a value associated with a different key may be returned. Consequently, lookups must be validated by comparing against the key in the tuple. To implement the same map-like interface offered by the B-Tree, we therefore store pointers to heap allocated tuples. The tuples contain a copy of both the key and value and their respective lengths.

Key Length. HOT expects a maximum key length to be set at compile time. We use a length of 4 for the integer key sets and a length of 256 for the string key sets.

String Keys. For lookups of strings, HOT is 5% faster to 14% slower than the adaptive B-Tree. ART is 34 — 37% slower, achieving roughly the performance of the baseline B-Tree. Both tries perform about 10 — 20% faster than the baseline for insertions of strings. The adaptive B-Tree is about 70% faster than the baseline.

Integer keys. ART is 129% faster than the adaptive B-Tree for lookups on `sparse` and 73% faster on `dense`. HOT is 10% faster for `sparse`, but 27% slower for `dense`. For inserts, ART is 9% and 80% faster, while HOT is 10% and 72% slower than the adaptive B-Tree for sparse and dense integers respectively.

Scans. For scans, even the baseline B-Tree is 18 — 73% faster than the better of the two tries. This is expected, as iteration for the B-Trees spans at most two leaves, whereas the tries need to walk many nodes per scan. Prefix truncation, heads, hints, and dense leaves improve further on this. The adaptive B-Tree outperforms the tries by 85 — 219%.

> **Insight 5:** B-Trees can compete with both tries for string keys but fall behind ART for integer keys. For scans, they are always superior.

Benchmarking Secondary Indexes. While we find HOT to achieve 52% lower throughput than ART for lookups of random 32-bit integers, the paper proposing HOT reports performance competitive to ART [14]. It is important to note that the HOT paper was measuring a slightly different operation: The retrieval of values from tuples was omitted, as it would be identical between all compared indexes. In contrast, B-Trees come with the advantage of handling the storage of tuples, so we implement this on top of ART and HOT for fairness. Our experiment differs in three more ways: First, we use 32-bit integers, instead of 64-bit. HOT is designed specifically to address ART's issues with sparse keys. Thus, it gains an advantage as keys grow longer and therefore more sparse. Second, the HOT paper packs integer keys into the tuple identifier, i.e., it assumes that the keys are equal to the tuple identifiers to simplify benchmarking. Third, we insert only half as many records, 25 million. Matching the original experiment in these three respects, we find HOT to be 39% slower than ART for integer lookups.

6.2 TLX B-Tree

TLX is a C++ library aiming to provide commonly needed functionality lacking in the STL. It includes among other template based collections a B-Tree implementation. This implementation was previously known as STX B-Tree, but has since been merged into TLX. In this section we compare it against our B-Tree. We use `tlx::btree_map<T,std::vector<std::uint8_t>>`, where `T` is the type of the key. We use `std::vector<std::uint8_t>` for strings and `std::uint32_t` for integers.

Comparison. For strings, TLX performs significantly worse than the baseline. This is expected given the additional indirection of `std::vector`. The baseline is 62 — 108% faster, the adaptive B-Tree is 148 — 259% faster. For integers, TLX has the advantage of not handling variable-sized

keys and comparing integers directly. The baseline is 32% slower than TLX for lookups, 9% slower for inserts, and 0 — 3% slower for scans. The adaptive B-Tree beats TLX by 24 — 58% for `sparse` and by 112 — 390% for `dense`.

Insight 6: Widely used generic B-Trees perform poorly.

6.3 LITS

LITS is a recently proposed learned index optimized for string keys[57]. We use the publicly available implementation referenced in the paper. As this implementation support only bytes in the range $[1, 127]$ in keys, we apply an order preserving escaping scheme that represents values outside the range $[2 - 125]$ using two bytes. Like ART and HOT, LITS supports only a 64-bit tuple identifier as value. As LITS already stores each record in a separate allocation and retains full keys, We store our 8 byte payloads directly in LITS. We expect that modifying LITS to support variable size values would reduce throughput only slightly. Inserting a tuple identifier like we do for ART and HOT substantially reduces LITS performance.

Comparison. The adaptive B-Tree achieves similar lookup throughput as LITS for `wiki` and surpasses it by 42% for `urls`. Non-ascii characters, on which LITS should perform worse due to escaping, are less common in `urls` than in `wiki`. For both kinds of integers LITS outperforms the adaptive B-Tree, though not by as much as ART does. LITS performs worse than either B-Tree for scans.

LITS Training Set Size. We find that the performance of LITS depends strongly on the size of the training set supplied at index creation. In this measurement we present 50% of all keys to the index before insertion, as the authors of LITS do. If we reduce this to 5%, the throughput achieved by LITS changes by -69% to +41%.

6.4 Wormhole

Wormhole [55] is a B-Tree where the inner nodes have been replaced with a hybrid data structure that combines a trie and a hash table. This enables very fast point lookups while still allowing scan operations (unlike hash tables, which offer even faster point lookups). For our benchmark we use the `whunsafe` API to not incur synchronization overhead and integrate with Wormhole's memory allocator to avoid unnecessary copies.

Lookup. The baseline B-Tree is 53 — 60% slower than Wormhole for lookups. Our adaptive B-Tree is still 26 — 28% slower. The exception to this is `dense`, where fully dense leaves make it 27% faster.

Insert. For insertion of strings, the adaptive B-Tree is 17-27% faster than Wormhole. It is 21% slower for `sparse` and 429% faster for `dense`, which seems to be a particularly bad case for Wormhole. Here, Wormhole achieves only 46% of the `sparse` throughput.

Scan. Like the other structures, Wormhole is much slower than the B-Trees for scans. Like TLX, Wormhole does not inline records into leaf nodes, but merely store pointers to them, hurting cache locality. As a result, the baseline B-Tree is 45 — 103% faster and the adaptive B-Tree 88 — 430% faster.

6.5 Space Use

To estimate the memory footprint of each structure, we determine the space use of the process before constructing the data structure and after completing the benchmark using `/proc/<pid>/statm`. Figure 18 shows space used by each of the data structures. Again, data sets are sized such that total key and value size is approximately 300 MB. The adaptive B-Tree uses the least space in all cases. TLX uses by far the most due to the 24 byte overhead of `std::vector`. Other than that, all

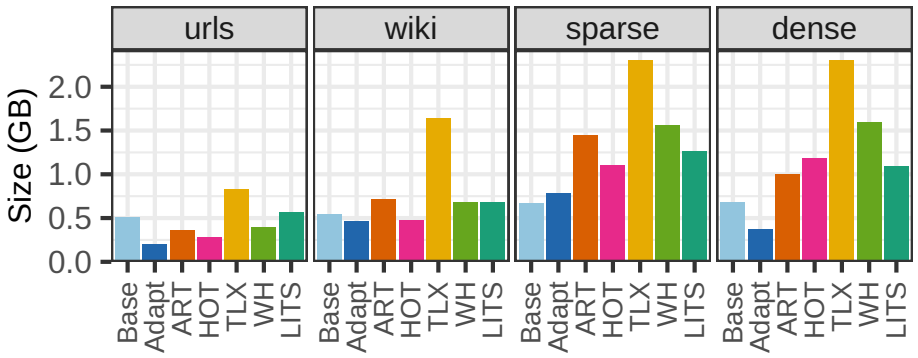

Fig. 18. Size of Data Structure.

in-memory indexes have very similar space use. This is because we include the space used for tuple storage, which is similar across all indexes. This makes up the bulk of their space use, even though we use relatively small 8 byte values. In contrast, the authors of HOT, LITS, and Wormhole measure space use of only the index structure and show significant space savings compared to competitors included in our experiment.

Wormhole Case Study. We analyze Wormhole's memory handling on `dense` in detail as an example. The total of all allocations made is only around 1.1 GB, substantially less than the observed 1.6 GB process size change. We attribute the difference to memory allocator overhead. Records themselves make up 65% of allocated space. Leaf nodes, which hold pointers to records and an indirection vector for in-order traversal, take up 32%. In summary, around one third of space use can be attributed to allocator overhead and a third of the remainder to pointers to tuples. As all in-memory indexes store records in separate allocations, we expect them to behave similarly.

Allocator Overhead. As the B-Trees allocate only full pages, they do not suffer from memory allocator overhead in the same way. The growth in process size is only 2% higher than the total size of nodes in the tree. Instead, memory fragmentation occurs within the tree in the form of partially filled pages and causes similar overhead.

Prefix Truncation. For the string keys, the space advantage of the adaptive B-Tree comes from prefix truncation. This is why the advantage is greater for `urls` than for `wiki` and why the baseline B-Tree performs similar to the in-memory indexes. While tries naturally share prefixes among keys, this applies only to the search structure. All in-memory indexes store the full key in each record.

Pointer Overhead. Storing records directly inside the page grants one more advantage to B-Trees: Records can be referenced using 16-bit offsets rather than 64-bit pointers. For the integer keys, where this overhead is significant, even the baseline B-Tree uses less space than the in-memory indexes. For `sparse`, the adaptive B-Tree suffers space overhead from key heads, while dense leaves increase the space advantage for `dense`.

6.6 Skew

As the parameter α of the Zipf-distribution of lookup keys increases, lookups generally become faster due to higher cache locality. However, throughput increases to different degrees for different data structures. We vary α from 0.5 to 1.5 and perform 10 million lookups for each data structure and key set. We use medians of five runs. Figure 19 shows the lookup throughput of in-memory data structures relative to the adaptive B-Tree. As α increases, ART, TLX, and Wormhole perform better in comparison to our B-Tree. In contrast, HOT becomes worse. HOT makes extensive use of

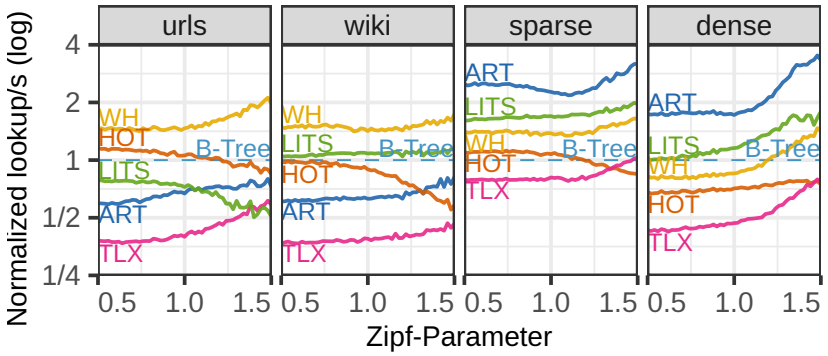

Fig. 19. Lookup throughput relative to adaptive B-Tree depending on Zipf-parameter used for key sampling.

prefetching to reduce the impact of cache misses. Therefore, it does not benefit from reduced cache misses to the same degree. LITS internally uses HOT in some cases, leading to mixed results.

6.7 Discussion

Overall, we see that the baseline B-Tree is generally slower than the in-memory structures for point operations. The adaptive B-Tree is competitive, reducing the gap in some cases and gaining an advantage in others. The TLX B-Tree storing strings using a separate heap allocation is by far the slowest of the options. For scans, B-Trees take the lead, making even the TLX B-Tree competitive.

7 System Integration

All experiments so far were performed single-threaded and in an in-memory setting. To demonstrate the real world benefits of the presented techniques, we integrate our B-Tree into the open source vmcache storage engine [34], which provides synchronization based on optimistic lock coupling [36] and transparent support for paging to storage. Because our B-Tree relies on fixed-size pages, the integration into a buffer manager such as vmcache is straightforward and does not require any changes to the data structure.

7.1 Synchronization

We first look at a multi-threaded in-memory workload. We repeat the experiment from Section 4 with a single thread and with 24 threads. Rather than executing a fixed number of operations, we execute lookups and scans for 30s each. Figure 20 shows the lookup throughput per thread for the original and the vmcache implementation. As expected, the throughput decreases slightly with the additional overhead of vmcache. Increasing the number of threads again slightly decreases throughput per thread. The adaptive B-Tree maintains a significant performance advantage over the baseline, demonstrating its scalability. Experiments with larger volumes of data, more threads, and the remaining B-Tree configurations indicate that the results from Section 4 transfer well to the multi-threaded benchmark, with two exceptions. First, fingerprinting leaves provide an even greater advantage to inserts, as unordered insertions consume significantly less memory bandwidth. Second, dense leaves achieve a smaller speedup, as the handling of locks adds significant overhead to the very fast dense leaf operations.

7.2 Out Of Memory

To evaluate out-of-memory performance, we make some changes to the synchronization experiment. First, we increase the number of records inserted to 600 million for integers and 120 million for `urls`. We discard the `wiki` keys, as there are not enough Wikipedia titles. We switch to a larger set

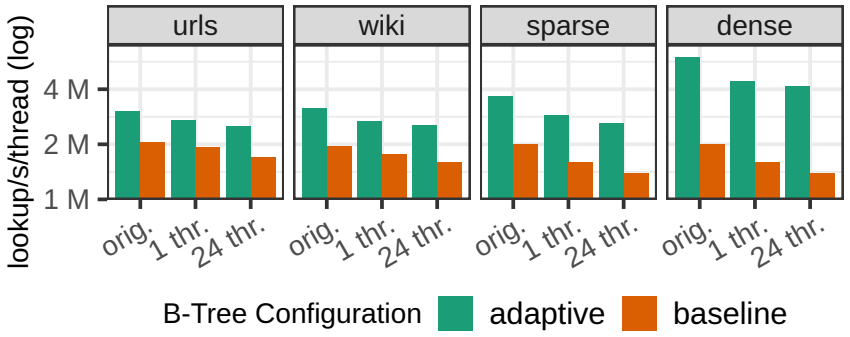

Fig. 20. Lookup throughput per thread for the original (unsynchronized) version and the vmcache integrated version with 1 and 24 threads.

of URLs averaging 90 bytes long and generate additional artificial URLs by prepending a byte from the range [0-127] to each URL. This first byte will be removed by prefix truncation, giving very similar behavior to real URLs in the lower levels of the tree. Second, to accommodate the larger number of keys, we use 64-bit integers for both `dense` and `sparse` keys. This yields a total size of roughly 9 GiB for the integers and 11 GiB for the URLs. We use a smaller data size for integers to compensate for the per record overhead of the B-Tree. Most importantly, we limit the size of the in-memory-buffer pool, which forces vmcache to evict pages to SSD as the tree grows. We vary this limit from 32 GiB down to 1 GiB. To issue more read requests to the SSD in parallel, we use 256 threads. We use a Samsung 980 PRO 1TB SSD. We run the benchmark for 300s and compute the average throughput.

OOM Performance. Figure 21 shows the lookup throughput of the adaptive and baseline B-Trees with varying buffer pool sizes. The x-axis shows the ratio of the sizes of the input data and the buffer pool. Performance drops sharply as this ratio exceeds some threshold and requests need to access evicted pages. Starting from this point, space efficiency dwarfs the in-memory speed considerations, as it determines the probability of having to read from SSD. The adaptive B-Tree performs slightly worse for random integers (due to the space overhead of heads) but significantly better for `urls` and `dense` integers. At the smallest buffer pool size of 1 GiB, 661k – 777k reads from SSD are performed per second. This is 61% – 72% of the read throughput achieved in a benchmark with `fio` (1080k).

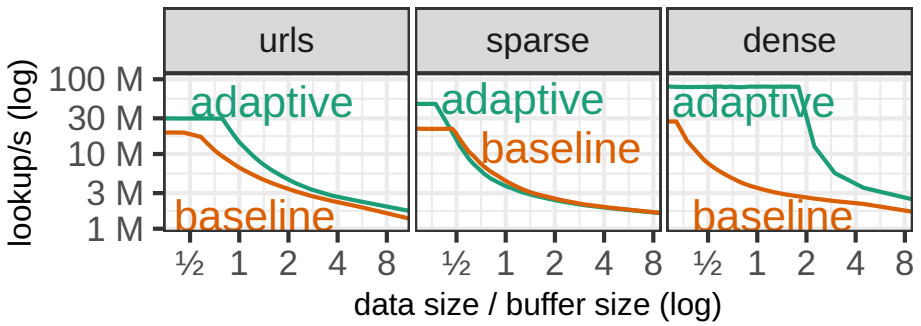

Fig. 21. Lookup throughput depending on buffer pool size.

7.3 Contention Split

Our B-Tree performs locking on a page granularity. If multiple frequently written keys reside in the same leaf, this leads to unnecessary contention. We implement contention split, a scheme to split contended nodes for our fingerprinting leaves [9]. It can be implemented just the same for the other two leaf types. We split leaves if at least $\frac{1}{30}$ of sampled writes encounter contention. For evaluation, we construct a B-Tree containing 5000 Wikipedia titles, and perform updates according to a Zipfian distribution ($\alpha = 1$) without index shuffling. With these parameters, 57% of writes are expected to hit one of the first 100 keys. We vary page size from 2 KiB to 32 KiB. We limit keys to 450 bytes to enable 2 KiB pages. Figure 22 shows throughput with and without contention split. Without contention split, throughput drops as page size increases and a larger share of writes contends for the first leaf. With contention split, throughput is relatively constant, indicating that the downside of coarse locking can be mitigated.

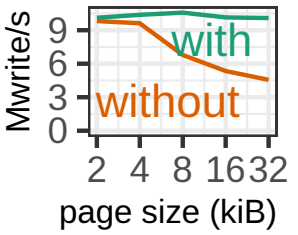

Fig. 22. Write throughput with and without contention split

7.4 Discussion

We have integrated our B-Tree into vmcache, which provides a synchronization framework and transparent support for paging to storage. We have shown that the adaptive B-Tree scales to many threads just as well as the baseline does. Though the optimizations are targeted at in-memory performance, the adaptive B-Tree often outperforms the baseline even out-of-memory.

8 Related Work

An extensive literature on B-Trees has accumulated since their inception in 1970 [18, 25, 26, 39].

B-Tree optimizations. Most optimizations discussed in this paper have been proposed previously. This includes prefix truncation [12], heads (termed *poor man's normalized keys*) [27], hints [8], and fingerprinting (though optimized for persistent memory) [45]. Frame-of-reference encoding has been proposed multiple times as a compression method [15, 24]. However, using the resulting offset to index an array in a B-Tree has to the best of our knowledge not been discussed in an academic publication. Such an array representation has been discussed in the context of implementing Equi-Joins[49], though without frame-of-reference encoding.

In-Memory B-Trees. While there are many papers concerned with optimizing B-Trees for in-memory use, these reduce the size of nodes to a degree that makes paging infeasible [33, 40, 46, 47].

MassTree. MassTree is a trie with a span of 8 bytes, where each trie node is a B-Tree [40]. We do not compare against MassTree, as it is significantly slower than Wormhole [55].

BP-Tree. BP-Tree [56] is a B-Tree with an alternative leaf layout that has good performance for point operations on larger leaves. It does not support variable sized keys, though its working principle could be transferred to the slot array. However, this would make the find operation proposed very cache unfriendly. Moreover, larger leaves are less desirable when using fingerprinting leaves and come with downsides for out-of-memory workloads[29].

Bw-Tree. Bw-Tree [38] is a lock-free B-Tree optimized to avoid cache invalidations in concurrent settings. The techniques discussed are orthogonal to our optimizations. However, later research finds that in-memory the Bw-Tree is inferior to a more conventional B-Tree synchronized using optimistic lock coupling [54].

Compression Alternatives. To increase the space efficiency of the B-Tree, we use both prefix truncation and truncated separator keys. A recent analysis compared alternative compression

techniques and found that for some data sets, other schemes can achieve a higher compression ratio [22]. All of these are variations of delta compression, where the prefix shared with the previous key is omitted from each key. Such schemes are employed among others by WiredTiger, RocksDB, and MyISAM. However, the authors of the study still recommend an approach that aligns with our implementation, as the alternatives have too high a performance cost.

LSM-Trees. LSM-Trees [44] are an alternative to B-Trees that organize data into runs according to how recently it has been written. This reduces write amplification at the cost of read performance. Contemporary LSM trees employ compression in their less frequently written levels to save space at the cost of access time. For instance, RocksDB uses delta compression. Many of the optimizations we present should transfer well to delta compressing LSM-trees. Dense leaves are a good fit, reducing both space use and access time. Fully dense leaves, however, are not compatible with value compression. Hints could be applied with little downsides and would likely improve access time. Sampling hints from the restart points of delta compression seems advisable. For Fingerprinting, two byte fingerprints should likely be used to reduce the probability of false hits, which cause costly key decompression. Heads are likely not worthwhile due to their space cost and the sorted nature of blocks. Apache Cassandra applies general purpose compression schemes to its LSM-Trees [4]. Our optimizations can be applied here, though access time for cold blocks will be dominated by decompression.

9 Conclusion

In this paper, we discuss and evaluate six optimizations for in-memory performance of B-Trees. The structure of keys and the type of operations performed have a strong effect on which techniques are beneficial. Accordingly, we design an adaptive B-Tree that selects one of three leaf node layouts at runtime. B-Trees can close the performance gap to pure in-memory structures using this technique. We integrate the B-Tree into the vmcache storage engine demonstrating significant full-system performance improvements both in and out of memory. To make our results reproducible, we make both the unsynchronized B-Tree [1] and the vmcache-integrated one [3] public.

References

[1] [n. d.]. btree-cpp. https://github.com/m-mueller678/btree-cpp/tree/sigmod25
[2] [n. d.]. BTree implementation for Go. https://pkg.go.dev/github.com/google/btree
[3] [n. d.]. btree24. https://github.com/m-mueller678/btree24/tree/sigmod25
[4] [n. d.]. Cassandra Documentation. https://cassandra.apache.org/doc/stable/cassandra/operating/compression.html
[5] [n. d.]. cpp-btree. https://code.google.com/archive/p/cpp-btree/
[6] [n. d.]. std::collections::BTreeMap. https://doc.rust-lang.org/std/collections/struct.BTreeMap.html
[7] 2022. SQLite. https://sqlite.org/datatype3.html
[8] Adnan Alhomssi, Michael Haubenschild, and Viktor Leis. 2023. The Evolution of LeanStore. In *BTW (LNI, Vol. P-331)*. 259–281.
[9] Adnan Alhomssi and Viktor Leis. 2021. Contention and Space Management in B-Trees. In *CIDR*.
[10] Muhammad A. Awad, Serban D. Porumbescu, and John D. Owens. 2022. A GPU Multiversion B-Tree. In *PACT*. ACM, 481–493.
[11] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indices. *Acta Informatica* 1 (1972), 173–189.
[12] Rudolf Bayer and Karl Unterauer. 1977. Prefix B-Trees. *ACM Trans. Database Syst.* 2, 1 (1977), 11–26.
[13] Timo Bingmann. 2018. TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers. https://panthema.net/tlx, retrieved Oct. 7, 2020.
[14] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2022. Height Optimized Tries. *ACM Trans. Database Syst.* 47, 1 (2022), 3:1–3:46.
[15] Philip Bohannon, Peter McIlroy, and Rajeev Rastogi. 2001. Main-Memory Index Structures with Fixed-Size Partial Keys. In *SIGMOD Conference*. 163–174.
[16] Anastasia Braginsky and Erez Petrank. 2012. A lock-free B+tree. In *SPAA*. ACM, 58–67.

[17] Hokeun Cha, Moohyeon Nam, Kibeom Jin, Jiwon Seo, and Beomseok Nam. 2020. B^3-Tree: Byte-Addressable Binary B-Tree for Persistent Memory. *ACM Trans. Storage* 16, 3 (2020), 17:1–17:27.

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

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

[20] Justin A. DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. 2013. Anti-Caching: A New Approach to Database Management System Architecture. *PVLDB* 6, 14 (2013), 1942–1953.

[21] Ahmed Eldawy, Justin J. Levandoski, and Per-Åke Larson. 2014. Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database. *PVLDB* 7, 11 (2014), 931–942.

[22] Chuqing Gao, Shreya Ballijepalli, and Jianguo Wang. 2024. Revisiting B-tree Compression: An Experimental Study. *Proc. ACM Manag. Data* 2, 3 (2024), 169.

[23] Nikolaus Glombiewski, Bernhard Seeger, and Goetz Graefe. 2019. Waves of Misery After Index Creation. In *BTW (LNI, Vol. P-289)*. 77–96.

[24] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *ICDE*. 370–379.

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

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

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

[28] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi A. Kuno, Joseph A. Tucek, Mark Lillibridge, and Alistair C. Veitch. 2014. In-Memory Performance for Big Data. *PVLDB* 8, 1 (2014), 37–48.

[29] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*.

[30] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage Engines. *PVLDB* 16, 9 (2023), 2090–2102.

[31] Ferenc Havasi. 2011. An Improved B+ Tree for Flash File Systems. In *SOFSEM (Lecture Notes in Computer Science, Vol. 6543)*. 297–307.

[32] Bernhard Höppner, Ahmadshah Waizy, and Hannes Rauhe. 2014. An Approach for Hybrid-Memory Scaling Columnar In-Memory Databases. In *ADMS@VLDB*. 64–73.

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

[34] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *SIGMOD* 1, 1 (2023), 7:1–7:25.

[35] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. 185–196.

[36] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019), 73–84.

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

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

[39] David B. Lomet. 2001. The Evolution of Effective B-tree: Page Organization and Techniques: A Personal Account. *SIGMOD Rec.* 30, 3 (2001), 64–69.

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

[41] Ingo Müller, Cornelius Ratsch, and Franz Färber. 2014. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *EDBT*. 283–294.

[42] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.

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

[44] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

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

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

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

[48] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. 2012. B+-tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives. *CoRR* abs/1201.0227 (2012).

[49] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD Conference*. ACM, 1961–1976.
[50] Radu Stoica and Anastasia Ailamaki. 2013. Enabling efficient OS paging for main-memory OLTP databases. In *DaMoN*. 7.
[51] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. *PVLDB* 17, 11 (2024), 3694–3706.
[52] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *DBTest@SIGMOD*. ACM, 1:1–1:6.
[53] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *SIGMOD Conference*. ACM, 1033–1048.
[54] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD Conference*. 473–488.
[55] Xingbo Wu, Fan Ni, and Song Jiang. 2019. Wormhole: A Fast Ordered Index for In-memory Data Management. In *EuroSys*. ACM, 18:1–18:16.
[56] Helen Xu, Amanda Li, Brian Wheatman, Manoj Marneni, and Prashant Pandey. 2023. BP-tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-trees. *PVLDB* 16, 11 (2023), 2976–2989.
[57] Yifan Yang and Shimin Chen. 2024. LITS: An Optimized Learned Index for Strings (An Extended Version). *CoRR* abs/2407.11556 (2024).
[58] Bowen Zhang, Shengan Zheng, Liangxu Nie, Zhenlin Qi, Hongyi Chen, Linpeng Huang, and Hong Mei. 2024. Revisiting PM-Based B$ˆ{+}$+-Tree With Persistent CPU Cache. *IEEE Trans. Parallel Distributed Syst.* 35, 5 (2024), 796–813.
[59] Weihua Zhang, Chuanlei Zhao, Lu Peng, Yuzhe Lin, Fengzhe Zhang, and Jinhu Jiang. 2022. High performance GPU concurrent B+tree. In *PPoPP*. ACM, 443–444.

Received July 2024; revised September 2024; accepted November 2024