The VLDB Journal (2026) 35:4
https://doi.org/10.1007/s00778-025-00950-8

REGULAR PAPER

An Evaluation of B-tree Compression Techniques

Sikang Sun[1] · **Chuqing Gao**[1] · **Shreya Ballijepalli**[1] · **Jianguo Wang**[1]

Received: 12 November 2024 / Revised: 24 October 2025 / Accepted: 27 October 2025 / Published online: 28 November 2025

Abstract

B-trees are widely recognized as one of the most important index structures in database systems, providing efficient query processing capabilities. Over the past few decades, many techniques have been developed to enhance the efficiency of B-trees from various perspectives. Among them, *B-tree compression* is an important technique introduced as early as the 1970s to improve both space efficiency and query performance. Since then, several B-tree compression techniques have been developed. However, to our surprise, we have found that these B-tree compression techniques were *never* compared against each other in prior works. Consequently, many important questions remain unanswered, such as whether B-tree compression is truly effective or not. If it is effective, under what scenarios and which B-tree compression methods should be employed? In this paper, we conduct an experimental evaluation of seven widely used B-tree compression techniques using both synthetic and real datasets. Based on our evaluation, we present lessons and insights regarding the use of B-tree compression that can be leveraged to guide system design decisions in modern databases.

Keywords B-trees · Database indexes · B-tree compression · Head compression · Tail compression

1 Introduction

Since the inception in the 1970s [17, 27], B-trees have played a fundamental role in database systems (and data management systems in general). This is due to many advantages that B-trees offer, such as simplicity, high performance, support for various query types (e.g., point queries and range queries), and compatibility with different hardware platforms (e.g., main memory, non-volatile memory, disks, SSDs, GPUs, and the cloud). As a result, almost all major database systems have implemented B-trees or their variants.[1]

Over the past few decades, there are many works proposed to optimize B-trees from various aspects, e.g., concurrency control [31], lock-free design [43], page layout [46], node size [34], compression [18], cache-aware optimizations [33, 37], and RDMA-optimized design [60]. Graefe provides an excellent survey on modern techniques of B-trees [32].

This paper revisits *B-tree compression*, a technique that was initially proposed in 1977 [18] aiming to reduce the space overhead and improve performance for B-trees. Two types of B-tree compression techniques were proposed in [18]: *Tail Compression* and *Head Compression* (see Section 2 for details). The main idea of the Tail Compression is to post a shorter separator to the parent node when splitting a leaf node, rather than selecting a full key (typically the middle one) from the leaf node. In contrast, the main idea of Head Compression is to compress the keys within a node [18]. As the keys in a B-tree are globally sorted, they tend to share some common prefix. With Head Compression, the common prefix will be factored out and stored only once to reduce the space overhead.

B-tree compression is expected to offer substantial benefits.

(1) *Reduce Space Overhead*: It is evident that the space overhead can be reduced. This is important because indexes are known to take considerable space overhead. According to Oracle blogs [26] and some research papers [58] , indexes can consume around 50-55% of the total database

[1] Note that we use B-trees to mean B+-trees in this work where the leaf nodes store all the keys.

✉ Jianguo Wang
csjgwang@purdue.edu

Sikang Sun
sun1017@purdue.edu

Chuqing Gao
gao688@purdue.edu

Shreya Ballijepalli
sballije@purdue.edu

[1] Purdue University, West Lafayette, Indiana, US

space. Moreover, many modern databases (e.g., Rockset [24] and AnalyticDB [55]) build indexes on *all* columns to enhance performance. Thus, it makes perfect sense to compress database indexes and in particular B-trees.

(2) *Reduce Monetary Cost*: The reduced space overhead can lead to reduced monetary costs when purchasing storage devices. Bhattacharjee et al. showed that the disk storage can take 24% $\sim$ 78% of the overall cost [19]. By minimizing the space overhead, B-tree compression can save millions of dollars for large-scale databases.

(3) *Improve Query Performance*: Another important advantage of B-tree compression is that it can achieve better query performance and insert performance. This is because some B-tree compression methods can support efficient query processing directly over compressed B-trees without the need for decompression, e.g., both Tail and Head Compression in [18]. By reducing the key sizes, the time spent on comparing keys for querying and insertion will be lower.

As a result, many real-world database systems have supported B-tree compression. Examples include DB2 [19], SAP HANA [21, 22], MongoDB WiredTiger [12], and MySQL (MyISAM) [9].

Motivation. However, to our surprise, those B-tree compression techniques were *never* compared against each other in the past. It is unclear whether the newer compression algorithms used in, for example, DB2 [19] or WiredTiger [12], can outperform the Head and Tail Compression techniques proposed in 1977 [18]. Furthermore, it remains unknown whether the Head and Tail Compression are indeed effective in improving uncompressed B-trees because the only available experimental results we are aware of were found in [18], which were conducted based on the hardware of the 1970s, a setup completely distinct from today's modern servers.

As a result, it is unclear whether modern database systems should adopt B-tree compression or not? If they should, then in which scenarios and which B-tree compression technique should be used?

Contributions. This paper answers the above questions by conducting the first comprehensive experimental evaluation of seven widely used B-tree compression techniques. We evaluate these compression methods using synthetic datasets with various distributions as well as real-life datasets regarding space overhead (compression ratio), search performance (including point queries and range queries), and insert performance. Moreover, we revisit techniques to further improve B-tree compression.

Based on the results, we refresh the understanding of B-tree compression techniques. In particular, we (1) present an up-to-date understanding of different B-tree compression techniques; (2) remedy misunderstandings and inaccurate conclusions made in prior works; and (3) clarify the scenarios in which compression algorithm should be employed for B-trees.

This article is an extended version of the conference paper presented at SIGMOD'24 [30], which is an experimental paper concluding that Head+Tail compression performs best. In this version, we present new contributions to explore techniques (including prefix vector representation, unrolled binary search, and key normalization) for further improving Head+Tail compression (in Section 6). We estimate **the new contribution to be at least 30%**.

Open-source. We open-source the code at https://github.com/chuqingG/BtreeComp.

Paper Organization. The paper is organized as follows. Section 2 explains the B-tree compression techniques evaluated in this work. Section 3 describes the experimental setup and implementation. Section 4 presents experimental results on synthetic datasets. Section 5 shows experimental results on real datasets. Section 6 revisits techniques to further improve B-tree compression. Section 7 reviews relevant work. Section 8 summarizes the main findings and concludes the work.

2 B-tree compression

In this section, we will review the existing B-tree key compression techniques. The main idea of these compression techniques is to reduce the individual key size in a B-tree node, either by truncating the prefix bytes or suffix bytes in a key.

2.1 Tail Compression

The Tail Compression technique was first introduced in 1977 [18]. This was one of the first compression techniques introduced for B-trees, and it aims toward reducing the size of separators in the non-leaf nodes. The main idea is to promote a shorter key to the parent node during leaf node splits by finding the shortest possible separator that can distinguish between the left and right nodes. Let us consider the example in Figure 1a and 1c, where we assume it to be a complete tree with only 4 nodes.

Instead of storing the full split key 'aecd' in node ①, Tail Compression promotes the shortest separator that can distinguish node ③ and ④, which is 'aec'. This separator is computed as the longest common prefix (LCP) between the last key of node ③ and the first key of node ④, followed by the next character in the first key of node ④. In this case, we have LCP("aeaf", "aecd") + 'c' = "ae" + 'c' = "aec".

The length of the separator can be further optimized by choosing a split point around the middle (not necessarily the middle point) that gives the shortest separator size. Let us consider the example in Figure 2 representing a leaf node

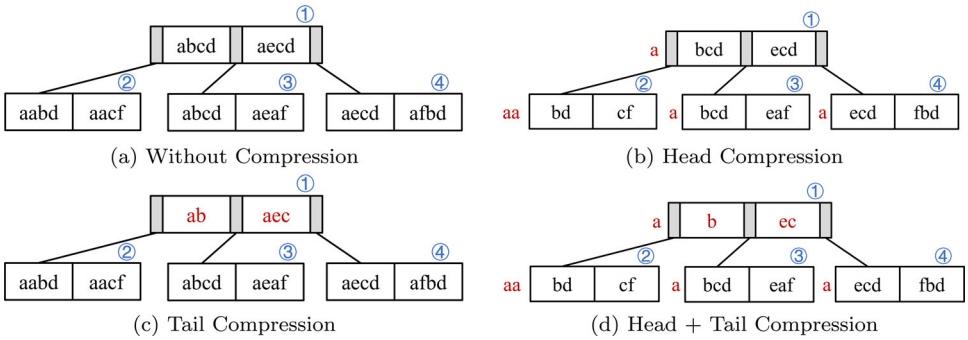


Fig. 1 Example of head and tail compression

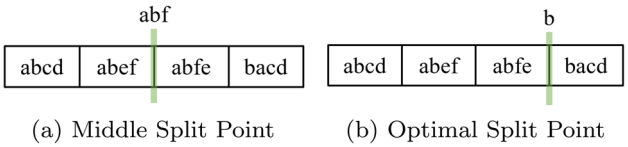


Fig. 2 Choosing a split point in tail compression

with four keys. If we choose to split the node in the middle, we get 'abf' as the separator, but choosing the split point after three keys gives a shorter separator 'b'. This optimization can also be extended to non-leaf nodes. A larger range of candidate keys increase insertion time while improving compression ratio. To balance them, [18] provides a recommended scope that ranges from the node and we use this scope in our implementation.

Tail Compression reduces the size of separators in non-leaf nodes. As a result, this increases the branching degree, reducing the tree's overall height. The smaller size of separators (leading to lower time for comparing keys) and the tree's smaller height improves the search time.

2.2 Head Compression

Head Compression was introduced in the same paper as Tail Compression in 1977 [18]. The intuition behind the Head Compression is that keys in the same B-tree node are "similar", i.e., they tend to share some common prefix, because the keys are sorted. Thus, we can factor out the common prefix among the keys. Specifically, for each node, we identify the largest lower bound and smallest upper bound from the parent nodes, and the common prefix between these bounds determines the prefix of the key. The upper bound is updated with the first key of the right sibling node during a split, while the lower bound is set as the smallest key. The leftmost and rightmost nodes of the entire tree do not have lower and upper bounds. Note that we do not calculate the prefix solely based on the current keys in the node. This ensures that adding new keys to a B-tree node does not need the re-computation of the prefix. This prefix is stored once per node, and all the keys in the node store only the suffix bytes.

Let us consider the example in Figure 1a and 1b, where the node ③ has the largest lower bound from its parent node ① as 'abcd' and the smallest upper bound as 'aecd'. Both bounds have a common prefix of 'a', thus node ③ can compress its new keys as 'bcd' and 'def'. Note that node ② and ④ are not compressed as they do not have a lower and upper bound.

Head Compression reduces the keys' size in leaf and non-leaf nodes if they have a common prefix. During the search operation, the key can be compressed using the node's prefix, and the comparison will be performed on only the suffix bytes without decompression. This reduces the comparison time between keys.

2.3 Head+Tail Compression

Head and Tail Compression can be simply combined to further reduce the space overhead, as shown in Figure 1d. On the basis of Figure 1c, it omits the common prefix within each page.

2.4 IBM DB2

IBM DB2 supports a different compression technique for B-trees [19]. It implements a variant of prefix compression where each key is represented using a (*prefix*, *suffix*) pair. It identifies subsets of keys with common prefixes and stores only the suffix bytes of the keys while storing the prefix only once. Figure 3a gives an example of the keys on a node, and Figure 3b shows how this is represented in DB2. h_n represents the header metadata that refers to the n_{th} key, including the offset and length of the key. We use the same notation h_n below as well. The (start:end) pair following a prefix indicates the range of the suffixes that have the prefix. In this example, the 0:2 following "ab" represents the suffixes with an index from 0 to 2 (i.e. "cd", "efg" and "exyz").

It triggers prefix optimization when the index page is almost full. If the current page shares a common prefix, the optimization greedily chooses the longest common prefix of the two current keys, adds suffixes to the prefix until it is no longer applicable, and then repeats the process. Otherwise, it applies two heuristics for optimizing the size of the prefixes

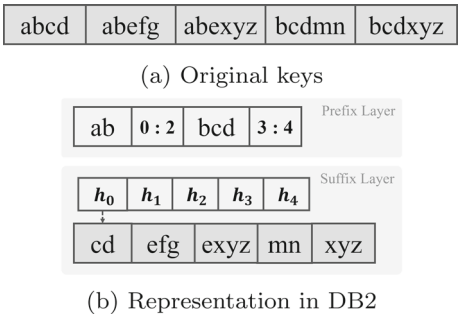


Fig. 3 Example of DB2 compression

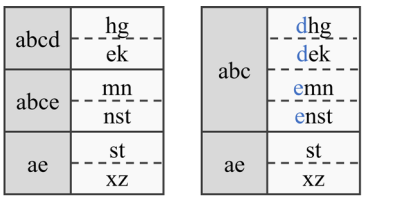


Fig. 4 DB2 prefix merge illustration

- Prefix Merge and Prefix Expansion, and chooses the one that offers greater space savings to install on the page.

2.4.1 Prefix Merge

Prefix Merge merges multiple prefix groups to reduce the size occupied by prefix metadata and increase the suffix bytes. It uses the concept of Closed Range (CR) for each prefix to identify segments that can be merged into a single prefix group. A CR for a prefix p_i is a group of prefixes that share the same prefix as the first key in the scope of p_i and the last key in the scope of p_{i-1}. When multiple choices for merge exist, the segment that gives the best space saving is chosen. For example, if we consider Figure 4, the CR of the prefix 'ae' is ['ac', 'ab'], which can be merged together to form a single group with the prefix 'a'.

2.4.2 Prefix Expand

Prefix Expansion expands prefixes across boundaries, increasing the size of prefixes and reducing the size of the suffixes. This heuristic is applied when the cost of increasing the prefix size is less than the space saved through the reduction in the size of suffixes. Figure 5 shows an illustration of prefix expansion where a subset of keys of the second and keys of the third prefix are merged to form a larger prefix 'abedc'.

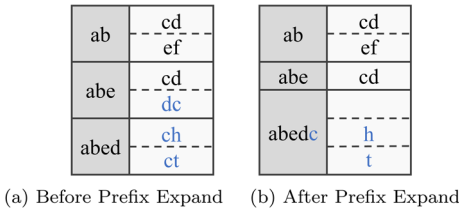


Fig. 5 DB2 prefix expand illustration

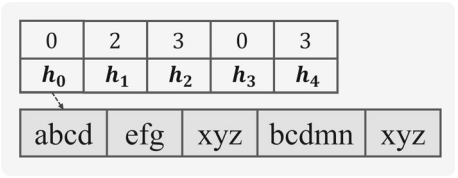


Fig. 6 Representation in wiredtiger

2.5 MongoDB WiredTiger

2.5.1 Prefix Compression

MongoDB WiredTiger's row-store storage format supports another variant of prefix compression [12] in the disk layout by identifying prefixes between adjacent keys, similar to delta-encoding. In this approach, each key stores only the suffix key bytes and the number of prefix bytes common with the previous key. The first key in the node is stored fully, and the following keys are stored by comparing with the previous key to identify a common prefix. Figure 6 shows how the example in Figure 3a is represented in WiredTiger. In addition to h_n, WiredTiger stores the length of the longest common prefix between the current key and the previous one. We use "prefix size" to denote it. The first key is "abcd" and does not have a predecessor in the node, so its prefix size is 0. The second key is "abefg", which shares a common prefix "ab" with "abcd".

Hence, the second key representation stores the suffix bytes as 'efg' and prefix size as 2. A similar approach is followed for the other keys.

In the most general case, a key is decompressed by moving backward till a fully instantiated key (or a key with prefix of length 0) is found and then walking forward to initiate the key.

2.5.2 Suffix Truncation

WiredTiger implements a suffix truncation technique [5] similar to the Tail Compression in [18]. If we consider the example in Figure 6 when the node is split between keys 'abfe' and 'bacd', based on suffix compression, 'b' is promoted to the parent node, instead of the entire key 'bacd'. In

subsequent parts of this paper, we use Tail Compression to refer to it in a uniform way.

2.5.3 Key Instantiation Techniques

One significant difference in its delta-encoding idea compared to other compression techniques is that the keys need to be decompressed using its predecessors in order to aid search or insertions. To accelerate this process, WiredTiger supports two types of key instantiation techniques that help to build the full key directly. Next, we introduce the ways the two techniques work. A further discussion about their applicability in the in-memory experiments will be covered in Section 3.3.

Best Prefix Group. The idea is to maintain a best slot whose base key can be used to decompress the most keys without scanning. WiredTiger defines the most-used page key prefix as the longest group of compressed key prefixes on the page that can be built from a single, fully instantiated key on the page. If the key falls under this prefix group, it can be directly initialized using the first key and the suffix bytes and therefore reduce the number of disk accesses.

Roll-forward Distance Control. WiredTiger also embeds an additional optimization by instantiating some keys in advance. The idea came from the observation that the search would be too slow in the case of a set of prefix-compressed keys requiring long roll-forward processing. For some worst cases, when we walk backwards through the page, each key would require processing every key appearing before it on the page. The method aims to help with the tree search during which the process may happen repeatedly. To control the maximum distance of roll-forward needed to decompress a key, WiredTiger sets a value, which is the number of a group of keys, and in this paper we call it skipping distance. For each set of keys of number skipping distance, WiredTiger instantiates the first key and therefore limits how far the cursor is forced to roll backward.

2.6 MySQL MYISAM

MySQL MyISAM supports prefix compression [9] by identifying prefixes between adjacent keys. It uses a representation similar to WiredTiger (Section 2.5), where each key stores only the suffix key bytes and the number of prefix bytes common with the previous key. One significant difference from the WiredTiger prefix compression is the search technique. MyISAM uses sequential search which starts from the beginning of the page, and therefore removes the need for key instantiation as every key can be constructed directly from the previous key.

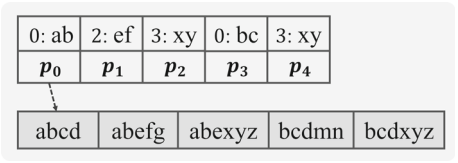


Fig. 7 Example of PkB representation

2.7 PkB-Tree

PkB-Trees were first introduced in [21] and are based on a partial-key approach. SAP HANA adopts this concept and uses a variant known as the CPB-tree [22]. To deal with the prefix, PkB introduced the concept of "base key": For the first key in the node, the base key is the key in the node's ancestor that is compared during the search. For any subsequent key in the node, its base key is the one immediately preceding it. In the partial-key approach, each key stores three parameters. (1) An offset representing the bit where the key differs from its base key (2) l bits of the key following the offset position (3) pointer to the data record. For the first key in the node, the base key is the key in the node's ancestor that is compared during the search. For the key in the following part, its base key is the previous one. In our in-memory implementation, we simply move all its structure to main memory, which introduces extra space overhead for it.

This pointer representation is based on the indirect key approach, where a pointer to the key is stored instead of the entire key. PkB was proposed to reduce CPU cache misses instead of saving the total space overhead. Since dereferencing the pointer increases cache misses, the offset and l bits are stored to aid key comparison. In scenarios where comparison is not resolved using the offset and l bits, the pointer is dereferenced. The search algorithm is designed such that it requires at most one pointer dereference per node. We apply a similar approach at the byte level for string data type and show how the example above is represented in PkB in Figure 7. p_i means the pointer refering to the i_{th} keys. In this example, we set l to its default value of 2 and assume that the node is the leftmost one of the tree, implying that the first key does not have a base key.

The original paper [21] uses data with a total size of 1MB, which is small enough to fit all its partial keys into the CPU cache. Despite its inability to save total space, we include PkB to see how it performs in modern scenarios and whether its idea can be migrated to disk-based scenarios.

The first key, 'abcd' is stored as 'ab', representing the first 2 bytes with offset = 0 and metadata pointing to the original key. The second key, 'abefg', is compared with its base key, 'abcd', and is stored as 'ef' representing the 2 bytes following the prefix 'ab' with offset = 2. The third key, 'bcdmn', has nothing in common with the previous one, so its first 2 bytes

'bc' is stored with offset = 0. The rest of the keys use a similar approach in their representation.

3 Experimental setup

In this section, we present the experimental setup in Section 3.1, evaluation metrics in Section 3.2, and implementation in Section 3.3 of various B-tree compression techniques.

3.1 Platform

3.1.1 Hardware Setup

The experiments were performed on an Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz based on an x86 architecture with 128GB DRAM. The CPU's L1, L2 and L3 cache sizes are 2.6MB, 70MB and 84MB. The disk-based experiments were run on a 1.6TB NVMe SSD, whose read/write bandwidth is 1GB/3GB per second and the read/write latency is 9/12 us.

3.1.2 Software Setup

Programming framework. All algorithms are implemented using C++ compiled using GCC 11.4.0 with O3 optimization enabled under Ubuntu 22.04. We did not include the values following other B-tree compression work. Their introduction would slightly reduce the compression ratio but would not change the overall conclusion. We evaluated both in-memory B-trees and disk-based B-trees. The implementation of these is further discussed in Section 3.3.

Datasets. We include both real world datasets (as shown in Table 1) and random-generated synthetic datasets. We use domain size to represent the size of the alphabet. For the synthetic data, strings are uniformly distributed by default, where each position is generated with equal probability for all characters in the alphabet. In Section 4.6, keys are treated as long numbers and are generated according to a normal distribution. See Section 4.6 for details. In terms of evaluation, we adopt a 20/80 warm-up/run policy. The first 20% is used only to bring the system to steady state (caches, buffers, background tasks) and is excluded from metrics. Throughput and latency are calculated on the remaining 80%.

Parameters. The following experiment results are averaged under three runs and the variance is usually within 1%. For the PkB implementation, we set the length of the partial key as 2 based on the optimal value reported in [21]. We model the data content of each key of the B-tree as a variable string, and all key comparisons are performed based on a byte-level comparison following prior works [9, 12, 18, 19, 21, 43, 52].

Table 1 Key Features in Different Datasets

Datasets	Num	Min Size	Max Size	Avg Size
TPC-H	60M	96B	156B	129.58B
WEBSPAM	25M	16B	2047B	112.33B
WikiTitles	25M	1B	255B	19.34B
MemeTracker	5.5M	8B	21.2KB	75.90B

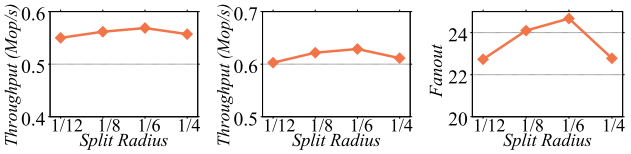


Fig. 8 Tail compression over different split Ranges

3.2 Evaluation Metrics

We measure the compression algorithms on three primary metrics:

1. **Compression Ratio**: Reducing the key size is the primary goal of compression techniques. Lower B-tree key sizes result in greater space savings.
2. **Search Time**: Search time allows us to analyze the potential benefits or overheads of applying compression on B-Tree nodes. This time is computed by issuing an exact search query on the B-Tree. By default, we focus on point queries, but we also evaluate range queries in Section 4.7.
3. **Insert Time**: Insert time evaluates the write performance, which is also important for B-trees. It demonstrates the impact of B-tree compression on insertion.

3.3 Implementation

As the source code is not available for some existing B-tree compression techniques, we try our best to implement each compression technique as close to its original implementation and as efficiently as possible. For all the methods, in cases where compressing non-leaf nodes is optional, we enable this feature to ensure a fair comparison. As for the split policy, Tail Compression selects the best split point from a given scope. We conducted a preliminary experiment on the impact of split range, as shown in Figure 8. A split radius r corresponds to the split range $[\frac{1}{2}-r, \frac{1}{2}+r]$. When r is small, the chosen separator may be long. When r is too large, it may lead to uneven splitting and increase the number of nodes, which further reduces the throughput. Therefore, we set r to $\frac{1}{6}$ in our experiments. In other algorithms where the split

point was not explicitly stated, we chose to split at the middle of the page.

Head, Tail, and PkB compression are implemented based on their original papers. The original DB2 paper [19] does not mention how the search is performed, so we implement a two-level binary search for it, where a binary search is first performed on the prefix layer and then on the suffix layer. MyISAM is implemented based on the design principles derived from the source code of the MyISAM storage engine in MySQL [8].

We implement the WiredTiger compression based on its source code [5]. The original WiredTiger B-Tree is based on both a disk and in-memory storage, we migrate the whole tree to memory when we conduct experiments on in-memory B-trees. As for its additional optimizations mentioned in Section 2.5, we explore their applicability in our benchmark by some preliminary experiments based on a workload of the default setting in Section 4.1. As for the best prefix group, it introduces a maintenance overhead during insert time due to the fact that each single insertion can invalidate the original metadata of best prefix group. Its extra branching logic also increases query time. In our experiments it resulted in a performance degradation of about 50% for insert while giving no significant improvement in query performance. As for roll-forward distance control, the average fan-out of non-leaf nodes is 41.45 as shown in Table 3. In the absence of suffix truncation, the average number of keys in a node should closely resemble that of MyISAM. The recommended value of skipping distance in [5] is 10, which implies that this instantiation technique may not be very helpful for purely in-memory workload . Also, the additional header metadata introduced by instantiated keys can further reduce the fan-out and the branching logic it introduce also makes it not always a positive optimization. Therefore, we only enable roll-forward distance control in disk experiments with the recommended value 10.

In disk-based scenarios, for most compression techniques, we store the non-leaf nodes in memory and move the complete pages in the leaf level to disk. For PkB, due to its characteristics, we move all the complete keys to the disk while keeping all B-tree levels in memory.

4 Results on synthetic datasets

In this section, we present the experimental results on evaluating the seven compression techniques on synthetic datasets. We use the default settings to show an overall result first, then discuss the effect of each factor on these compression techniques in detail.

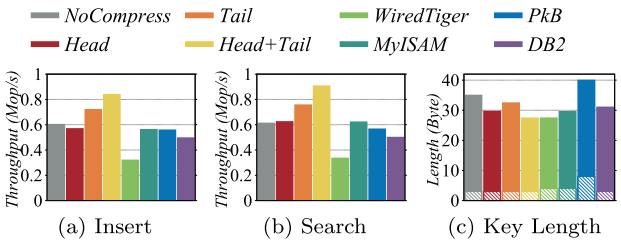


Fig. 9 In-memory results under default settings

Table 2 Default Experiments Settings

	Page Size	# of Keys	Length	Domain Size
Mem	512B	100M	32 Bytes	10
Disk	4096B			

4.1 Overall Results

4.1.1 Results in Memory

Figure 9 shows the overall results under our default settings, whose parameters are showed in Table 2. The keys are uniformly distributed random numbers.

Insert Performance. As shown in Figure 9a, Tail Compression and Head+Tail Compression perform better than other techniques. Tail Compression adds extra processing only at split time and does not introduce complex adjustment strategies like DB2 does. This advantage makes these two algorithms the only ones that improve insert performance compared to the uncompressed B-tree.

WiredTiger, PkB, and MyISAM, the three algorithms based on the delta compression, are 46.8%, 7.3% and 6.7% worse than the uncompressed B-tree respectively. The degradation results from an increase in decompression-like operations inside the node. WiredTiger, the most affected algorithm, uses 44.6% of the total insertion time for decompression. Without the existence of WiredTiger's additional instantiation techniques, MyISAM's sequential scanning reduces duplicate scans compared to WiredTiger's binary lookup, resulting in better performance.

DB2, on the other hand, its insertion performance is slowed down by two things, its aggressive optimization strategy performance at split time and slower searches, which we will discuss below.

Search Performance. Figure 9b shows the performance of point queries. Head+Tail Compression exhibits the highest throughput, surpassing the uncompressed B-tree by 48.2%, which is attributed to the synergistic effect of Head and Tail Compression. Head Compression increases the throughput by 2% by reducing the length of characters per comparison within a page. Tail Compression reduces separator length

in non-leaf nodes, thereby decreasing the B-tree height and resulting in a 23.5% increase in throughput.

Like insertion, WiredTiger's search performance is also degraded by its decompression time. MyISAM's query performance is slightly higher than the uncompressed B-tree. The performance of PkB is affected by the fact that the length of the partial key is sometimes insufficient for comparisons. Like MyISAM, PkB performs a sequential lookup within a node, but instead of returning when the result of comparison is equal, PkB returns a segment whose prefixes truncated by partial key mechanism match the target key value, and then continues to search in this segment using the complete keys when the length of segments is greater than one. And this process can introduce a certain amount of repeated comparisons.

DB2 shows lower throughput compared to Head Compression because of its multiple prefixes within a page. Our experiment shows that prefix search takes about 30% of the total time of in-page search, this conclusion is based on the total size of prefix metadata and data structure we introduced in Section 3.3, how to fine-tune these details to reduce prefix search overhead is beyond the scope of this paper.

Table 3 Statistics on B-trees with Different Techniques in Default Memory Setting

Techniques	Height	Fanout	# Nodes	# Non-leaf
Origin	9	9.27	13200400	1423920
Head	8	10.64	10527600	989764
Tail	6	28.51	12192500	427629
Head+Tail	6	42.87	9727250	226899
WiredTiger	6	41.45	10977100	264843
MyISAM	9	9.92	11709900	1180310
PkB	9	8.56	14445900	1688180
DB2	9	9.61	12321400	1282050

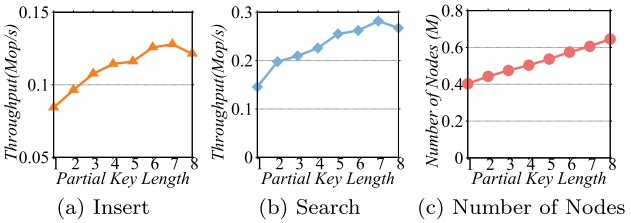


Fig. 10 Performance under different partial key length

Key Size. As shown in Figure 9c, we break down the key length into two components: The line-filled part represents the length of the header metadata for each item (which, in an uncompressed B-tree, includes the offset and key length); the solid part represents the length of the actual content, comprising both the key and the prefix (if any).

The average length is calculated across all nodes, rather than only those where the compression applied. Except for the overall result in this section, we use compression ratios instead of actual lengths to visualize the compression effect.

Head+Tail Compression and WiredTiger show most space saving because they utilize both prefix and suffix. They achieve compression ratios about 1.27x.

PkB shows a negative compression effect, because its structure is not optimized for in-memory scenarios. In in-memory experiments, it stores both the full key records and the compressed one, which makes it always need more space. To be specific, the total length equals original key length and length of header, which is a constant only affected by the pre-set partial key length.

Among other techniques, Tail Compression shows a relatively low space saving over all nodes because it only works for non-leaf nodes. As illustrated in Table 3, its truncation decreases the proportion of non-leaf nodes which increases fan-out. Among the other three techniques, Head Compression and MyISAM show similar compression ratio. DB2 takes up more space than Head Compression as their multiple prefixes require more header metadata, but for this dataset the saved space in suffix is not enough to compensate for it.

Furthermore, we gather supplementary spatial data to complement the compression ratio and display them in Table 3. Generally speaking, the ones with Tail Compression (Tail Compression, Head+Tail Compression and WiredTiger) significantly reduce the height of trees and thus shorten the search path. For average fan-out of non-leaf nodes, Head+Tail Compression and WiredTiger, the two techniques apply compression to both prefix and suffix are highest.

Overall, on the random dataset, Head+Tail Compression and WiredTiger achieve the highest compression ratios, with Head+Tail Compression showing better throughput because it does not need decompression for query processing.

4.1.2 Results on Disk

There are some differences between disk and memory experiments. A setting change is the length of the partial key. PkB was not initially designed for disk, and we reorganized the structure of its in-memory nodes for disk scenarios, as discussed in Section 3.3. Additionally, since PkB accesses the disk each time its partial keys are insufficient for comparison, the length of these partial keys is data-sensitive. It should be carefully chosen to maximize its ability to distinguish between keys without introducing excessive space overhead. Therefore, we would like to explore whether there is an optimal value other than the recommended one.

As shown in Figure 10, there is a pronounced improvement in performance when the length of the partial key increases

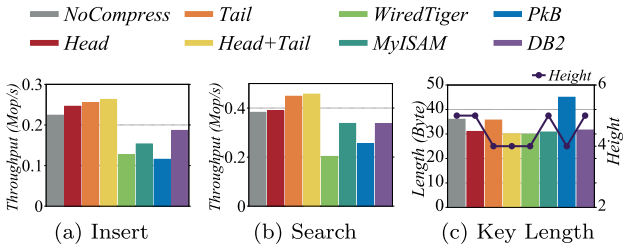


Fig. 11 Overall results on disk under default settings

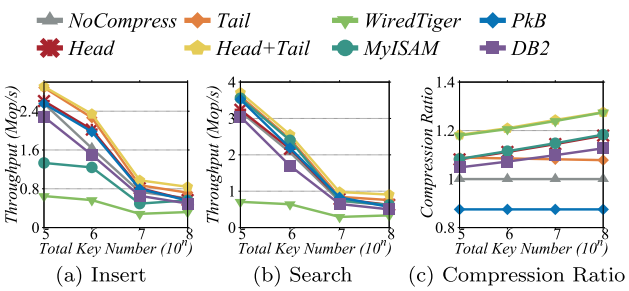


Fig. 12 Performance under different number of keys

from 1 to 4 bytes, implying that a large percentage of compressed keys can be differentiated by comparing only 2-4 bytes. However, performance begins to decline as the partial key length reaches 8 bytes. In the range of 5 to 7 bytes, we chose 5 bytes as the length of the partial key for the disk experiments below because the number of nodes (i.e., the total memory overhead) continued to grow steadily.

Another main change is the page size. We use 4KB, the size of an OS page, as our default value in disk-based scenarios. Larger pages can diminish the effectiveness of some compression techniques, which we discuss further in Section 4.4.

Figure 11 shows the overall results on disk. All techniques, except PkB, involve only one disk I/O for accessing the leaf node. This fixed overhead narrows down the relative improvement in query performance by compression, and the overall results show a trend consistent with the experiments in memory scenarios. The gap between WiredTiger and MyISAM is partially reduced because WiredTiger's roll-forward distance control plays a role. Under such a page size, all prefix-based methods show similar space savings. Similar to the memory scenario in Table 3, Tail Compression reduces the tree height from 5 to 4, making it also effective in improving throughput.

PkB, on the other hand, saves space in memory by moving all the complete keys to disk. Each key item only needs to store the header metadata, whose length is constant, resulting in a height that is also lower than that of the uncompressed tree. However, its sequential in-node search performance suffers significantly under such a large page size. Compared to MyISAM, which also employs sequential search, the extra indirect addressing (actually disk access) in PkB causes it to show lower throughput. Additionally, the results of partial key comparisons (i.e., matched length) are not utilized by the complete key, leading to a small number of duplicate comparisons.

4.2 Impact of Number of Keys

Figure 12 shows the results when the number of keys varies. For the randomly distributed keys on a given dictionary domain, the higher of the total number, the higher the similarity between neighboring ones in the sorted key. In other words, prefix-based compression techniques would show a more significant space saving. This is reflected in Figure 12c. As the number of keywords grows exponentially, the compression ratios of Head, Head+Tail Compression, WiredTiger, DB2, MyISAM show a near-linear improvement. For Tail Compression, the growing number of keys makes the separator of two adjacent longer, which reduces fan-out and thus increases the proportion of non-leaf nodes (from 3.055% to 3.507% when the number of keys grows from 1M to 100M). When we count the over all nodes, these two factors work together to stabilize the average key length, increasing it only by a tiny amount.

For search performance, the relative performance gap between the best Head+Tail and MyISAM/PkB increases as the number of keys grows. That is because high similarity between keys reduces the number of fully instantiated keys within a node, which causes the number of predecessors that need to be scanned to construct a full key to become larger. DB2 has a more obvious drop in search performance as the number of keys increases. It is partially because that the growing number of prefixes within a node has a greater impact on performance than the longer prefixes. In other words, the growing time for prefix search outweighs the time saved by the suffix comparison.

In short, for our distribution, the effectiveness of Tail Compression decreases as the number of keys increases. For the other methods , their growth in compression ratio have the same trend. Head+Tail still show a best performance overall.

4.3 Impact of Key Length

For a given distribution, when the number of keys is constant, the length of the common prefix of keywords in a interval of certain length is the same in probability. In other words, if the number of keywords within a node is constant, the length of the common prefix within a node is roughly the same for keywords of different lengths. However, when the total page size is fixed, increasing the keyword length reduces the number of keywords within the page. This leads to two effects, one is the number of nodes belongs to the same

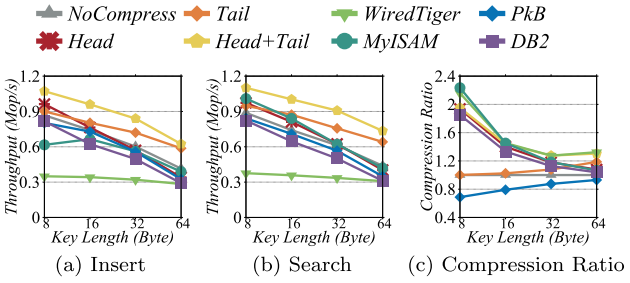


Fig. 13 Performance under different key lengths

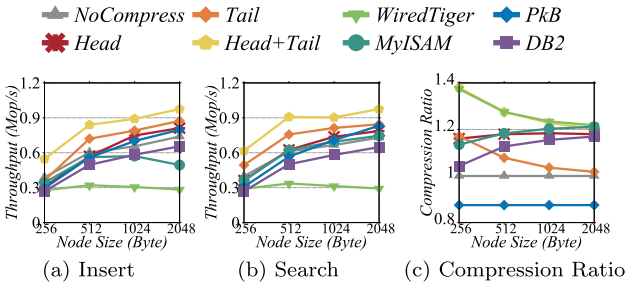


Fig. 14 Performance under different page sizes (Memory)

common prefix becomes smaller, another is that the common prefix of the fewer nodes would become longer. They two will have opposite effects on the compression ratio, and according to our experimental results in the Figure 13c, the former wins the game. Additionally, longer keys require more space to be set aside for page splitting, its effect on fan-out can be approximated by the following calculation. We denote the page size by S and the average key length (includes header metadata) by L, then assume that the total length of the new metadata written during the split time and the space to ensure that the next key can be inserted is about k times the average key length. Depending on the compression method, the value of k varies in (1, 3) approximately. Then when a node reaches the limit and need to split, the fan-out F roughly satisfies $F = \frac{S}{L} - k$. Therefore the size of the effectively utilized space on page is about $S - k * L$. As L gets larger, k leads to more unused space in the page.

On the other hand, some compression methods benefit from longer keys. For example, when the length of separators, as we discussed above, does not change much, thus it has significantly improved compression efficiency relative to the total length of keys. For techniques based on delta compression, decreasing in number of keys within a page shortens their path for decompression, which therefore narrows their gaps between Head Compression. Also, for the fix-length cost, the header metadata of PkB, longer keys reduce the relative overhead it imposed.

In a word, with a roughly stable length of common prefixes, the increase in key length leads the effects of prefix-based and suffix-based compression to vary in opposite trends, combining the two helps ensure applicability under different data.

4.4 Impact of B-tree Page Size

4.4.1 Results in Memory

We then explore the impact of page size on the compression techniques, for DB2, we always cap the total size of prefix metadata at 25% of the page size. Figure 14 shows the results. As the page size grows, the Tail Compression seem to become less effective in terms of compression ratio. However, this is an artifact caused by the small proportion of non-leaf nodes. As shown in Table 4, it actually increases fan-out and reduces the height effectively, because larger node sizes exacerbate the variability between non-leaf nodes and shorten the length of separators between non-leaf nodes. However, as the overall tree height decreases, it no longer has a significant advantage in shortening the search path. It can reduce the height from 13 to 8 when the page size is 256B, and only from 5 to 4 when the page size grows to 2048B. Accordingly, the query performance gap between the Tail Compression and uncompressed B-tree decreases as the page size increases.

Table 4 Tail Compression under Different Page Sizes

Page Size	Height	Fan-out	Non-leaf Nodes Ratio
256	8	14.0807	7.102%
512	6	28.5119	3.507%
1024	5	60.8383	1.644%
2048	4	134.845	0.740%

The compression ratio gap between DB2 and Head Compression also decreases as the page size grows and reverses at the page size of 2KB. This is due to the fact that DB2's prefix merge and prefix expand optimization strategies only work when there are multiple prefixes within a node, and when the number of prefixes is 1, the prefix optimization algorithm greedily creates new prefix groups within the node. However, multiple prefixes are not always preferable to a single prefix, especially when the number of keys within a node is small. The extra prefixes and their metadata may incur greater space overhead. However, as the page size grows, for example when there are hundreds of keys within a node, a common prefix for all keys may not effectively compress the length of the keys, thus the multiple prefixes of DB2 can be helpful.

Overall, as the page grows, most compression methods will improve performance due to shorter search paths. For delta compressions in particular, a larger page means a longer distance from the previous fully instantiated key. As the page size grows, MyISAM/PkB's search performance

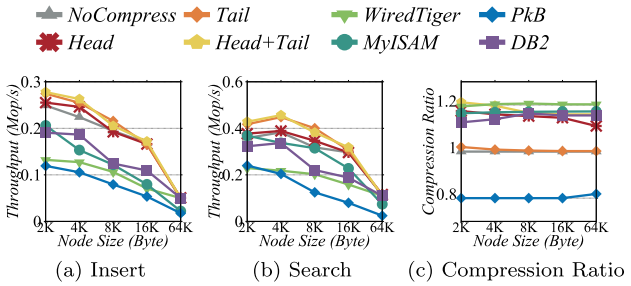


Fig. 15 Performance under different page sizes (Disk)

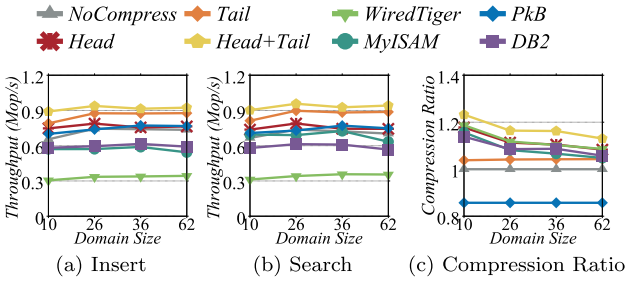


Fig. 16 Performance under different domain sizes

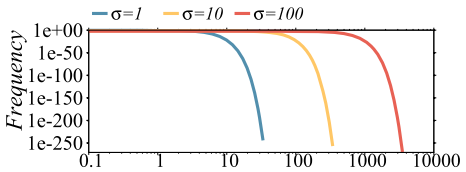


Fig. 17 Data frequency under different scales

increases, while WiredTiger hovers around the same throughput. Head+Tail holds the highest absolute performance across all experimented parameters.

To sum up, as the page size increases, the compression ratios of MyISAM and DB2 increase, while the Head and Tail Compression decreases. Tail Compression improves performance by shortening search paths across page sizes. MyISAM's insertion performance decreases at large page sizes due to the increased overhead of in-node decompression. DB2 suffers from the increasing time of prefix search.

4.4.2 Results on Disk

Compared with in-memory scenarios, page sizes in disk scenarios are larger in existing DBMS (e.g., SQLite uses a 4KB page size [4], and Postgres uses an 8KB page size [7]). We varied the page size from 2KB to 16KB, and the results are shown in Figure 15. Unlike the in-memory case, where the throughput initially rises and then falls, in disk scenarios without additional optimization for large page sizes like that in [54], the throughput degrades on larger pages for most compression techniques. One reason is that the larger page size cannot further reduce the tree's height but increases the in-node search cost. For all prefix-based compression techniques, the compression ratio stabilizes at around 1.2 for the given page size range. The throughput decreases gradually as the page size grows because each point query accesses the disk only once, and the larger page size cannot reduce disk I/Os. We observe the same pattern for large page sizes (e.g., 64KB). Page sizes larger than 64KB require a metadata size larger than two bytes to store pointer information, which will further decrease storage effectiveness and thus throughput.

4.5 Impact of Domain Size

We choose four different domain sizes, 10, 26, 36, and 62, which correspond to (1) pure numbers, (2) upper/lower case letters, (3) numbers and upper/lower case letters, (4) numbers, lower and upper case numbers Based on Section 4.4, we resize a page to 1KB, the maximum size at which the performance of the majority of these techniques has not started to degrade.

As shown in Figure 16, the results for compression ratios are consistent with the statistical intuition that the compression ratios of prefix-related techniques gradually get closer to 1 as the domain size increases. Tail Compression, on the other hand, is not much affected by domain size. As domain size grows from 10 to 62, its compression ratio is basically a horizontal line on the figure, and the average length of keys changes by less than 0.2 Byte. The situation is slightly different in terms of throughput. Most methods saw a small overall improvement in search performance as the domain size increased. This is due to the fact that for strings that are less similar to each other, fewer bytes need to be compared to get a comparison result.

4.6 Impact of Distribution of Keys

Figure 17 shows how the standard deviation (the same as the word 'scale' below) affects the degree of aggregation of the keys, which in turn affects the average prefix length. In this example, keys with a scale of 1 vary within 50, while those with a scale of 100 vary by around 5000. When adding all the keys with the same offset of 10000, keys with a scale of 1 are located in the range (10000, 10050) and therefore have a common prefix '100'. For keys with a scale of 100, they spread in the range (10000, 15000), which results in a shorter common prefix '1'. Therefore, the prefix length can be roughly controlled by varying the degree of aggregation. We generated 32-byte long keys in a similar manner. We also compare them with the keys of uniform distribution which is more spread out than a normal distribution with a standard deviation of 10^{31}.

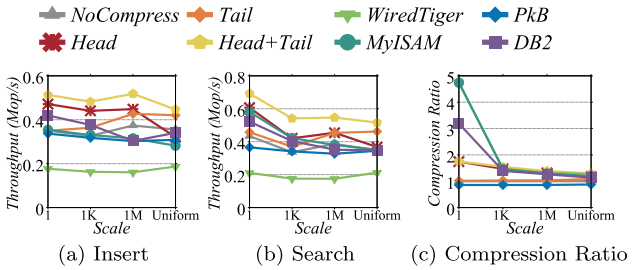


Fig. 18 Performance under different scales

As shown in Figure 18, in terms of compression ratios, the results under the different distributions show significant differences. As the scale decreases, the space-saving advantage of compression between neighboring keys over intra-node common prefix compression becomes more and more apparent. In particular, the compression ratios of MyISAM and WiredTiger growly rapidly at small scales (from 1K to 1) while those of Head and Head+Tail Compression increase slowly but steadily. This is due to the fact that the average node common prefix length has an approximately negative linear correlation with the exponent of scale, and this does not hold exactly for the length of common prefix between neighboring nodes. DB2, as result of the tradeoff between fine-grained and coarse-grained compression, shows a compression ratio roughly the average of Head Compression and MyISAM/WiredTiger. Tail Compression and the suffix part of WiredTiger have almost no effect when the scale is very small. When setting the scale to 1 and 1k, Tail Compression can only reduce the height of the B-tree by one instead of two in other cases.

Changes in throughput corresponds to those of the compression ratios. MyISAM, as a representative of the approaches based on delta compression, gradually converges to the same query performance as the uncompressed B-tree as the scale increases. Tail Compression, on the other hand, shows a significant performance degradation due to the longer search path when the scale is set to 1K, and rise again when the scale decreases to 1 because the comparison bytes become shorter. the performance gap between Tail Compression and the uncompressed B-tree widens as the scale increases. Insert performance demonstrates the tradeoff between index creation overhead and the performance gains of compressed keys. When the scale is less than 1M, DB2 and MyISAM outperform the uncompressed B-tree.

Overall, when data aggregation is super centralized, MyISAM and WiredTiger exhibit significantly higher compression ratios than the other methods, but there is no concomitant significant improvement in query performance.

Table 5 Search Throughput (Kop/s) Under Different Ranges

	Memory		Disk	
Range	10	100	10	100
Original	201.73	65.03	275.23	168.24
Head	177.85	63.06	215.25	100.57
Tail	237.57	74.99	278.66	171.79
Head+Tail	204.29	64.73	226.19	107.65
WiredTiger	112.57	42.92	207.18	101.96
MyISAM	173.30	53.92	234.82	110.78
PkB	174.31	64.63	30.66	16.10
DB2	162.99	71.77	202.49	96.90

4.7 Impact of Query Ranges

We then explore the performance of range query, where all keys in the query range need to be compressed. We vary the query range from 10 to 100 items. The results of both memory and disk are shown in Table 5. Tail Compression always shows the best performance because there is no decompression needed. Some compression techniques have metadata in the nodes that help to quickly locate whether the target range terminates within the current leaf node. For example, Head Compression maintains the upper bound of the node. However, in our tests, there was no significant difference in performance between skipping entire nodes and batch decompressing versus decompressing and comparing keys one by one. For the in-memory scenario, all methods that require decompression (or prefix-based) have lower throughput on range queries than the uncompressed tree. Disk experiments show similar results. In addition, it shows that migrating PkB directly to disk scenarios is not a good idea. Its design of separating partial keys and full keys makes localization disappear, so range queries require a lot of disk I/Os.

5 Results on real datasets

In this section, we present results on four real datasets: TPC-H [10], WEBSPAM-UK2007 [2], WikiTitles [11], and MemeTracker URL Links dataset [3]. Table 1 summarizes the key features in these datasets.

5.1 Results on TPC-H

TPC-H [10] is a popular decision support benchmark whose data is close to which for everyday use in the real world. We choose the largest LINEITEM table from it and set the scale factor to 10, which correspond to tables with 60 millions rows. As with most databases these days, in dealing

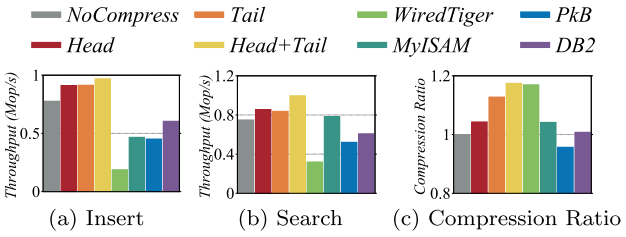


Fig. 19 Results over TPC-H LINEITEM

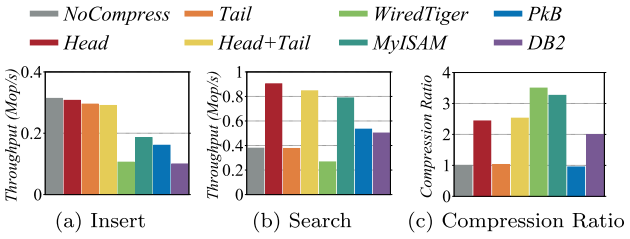


Fig. 20 Results over WEBSPAM-UK2007

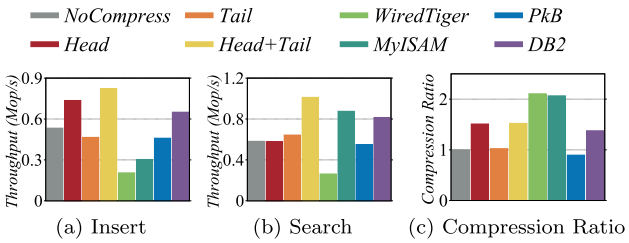


Fig. 21 Results over MemeTracker URLs

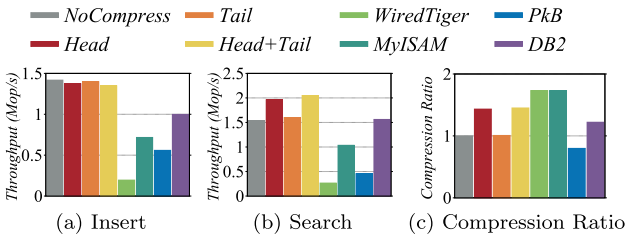


Fig. 22 Results over WikiTitles

with multi-column composite keys, we consider the combination of the involved columns in B-tree as a single column and implement compression on it.

Figure 19 shows the results in memory. Overall, similar to the results in synthetic dataset, Head+Tail shows the best result in both throughput and compression ratio. The first few columns in the LINEITEM table are mainly numeric characters with a small domain size, and the compression effect of prefix-based techniques are mainly (or only) related to the first column due to the existence of column separator. Thus, Head Compression, MyISAM and DB2 do not save much space compared to the average key length.

Consistent with the synthetic datasets, for such keys with low overlap, the three ones that truncate the suffix show higher compression ratio than others. For the simple keys in the first column, there are usually only 2 or 3 different values within a 1KB-length page. Accordingly, for MyISAM and WiredTiger, the length of common prefix between the current and previous keys in the key header only changes 1-2 times within a node. Thus few memory copying are needed to rebuild a prefix, allowing MyISAM that uses sequential scan to exhibit higher query performance than uncompressed B-tree despite a low compression ratio.

5.2 Results on WEBSPAM and MemeTracker

WEBSPAM [2] is a collection of 100M URLs of spam/non-spam hosts collected from the .uk domain. For the purpose of this experiment, we collect 25M of the URLs. Unlike randomized datasets, urls are naturally more amenable to prefix-based compression. Considering the lengths of the keys, we adjust the page size to 4KB. Figure 20 shows the results.

Overall, all techniques except Tail Compression show more effective results here compared to some of the randomized datasets mentioned above. In terms of compression ratio, techniques based on delta-compression (i.e. MyISAM and WiredTiger) save more space than those that compress the common prefix in a range (i.e, Head Compression and DB2). DB2 has a lower compression ratio than Head Compression, the main reason is that for urls, the multiple prefixes in a node often have long common prefix and differ only in the last few bytes, thus the extra metadata overhead is more than the space saved by further compression. The high compression ratio of MyISAM compensates for the high overhead of decompressing one by one. However, WiredTiger's in-node binary search still leads to unsatisfactory throughput due to the presence of duplicate decompression. As for DB2, consistent with the results we discussed in Section 4.3,

it suffers from the long keys in this datasets, which make the prefix search overhead a relatively large portion of the total query time. Therefore, the smaller number of keys makes the common prefixes in nodes longer and increases the share of prefix search in the total query time of DB2, leading to a larger gap between DB2 and Head Compression in both compression ratio and throughput.

Similar results can be obtained on other datasets with domain-specific high similarity prefixes.

For example, we also run the benchmark over another urls dataset, the MemeTracker URLs dataset. The results are shown in Figure 21. A main difference between the two datasets is the average length of keys. The growing number of in-node keywords makes DB2 perform better and reduces the efficiency of Head Compression.

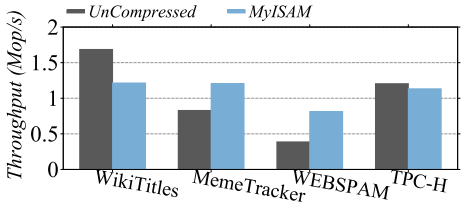
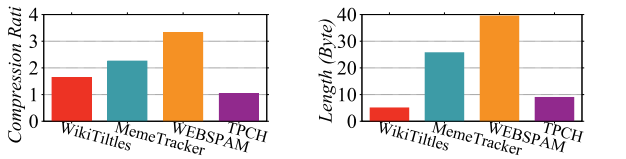


Fig. 23 Performance comparison between MyISAM and Uncompressed B-Tree on Real Datasets



(a) Compression Ratio (b) Average Comparison length

Fig. 24 Statistics on compression ratio and Keys' feature

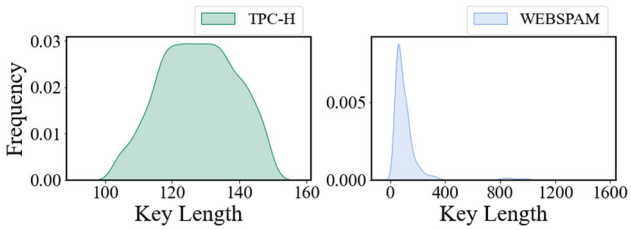


Fig. 25 Distribution of key length

5.3 Results on WikiTitles

We chose the English Wikipedia title dataset as a representation of the distribution of real-world texts in a linguistic perspective. WikiTitle's overall key length is short, along with the header metadata, the average length of each item in the uncompressed B-tree is only 23.73 Bytes. As shown in Figure 22, the constructional properties of English favor the prefix-based compression to some extent while the Tail Compression does not make a noticeable difference. Comparing Figure 19 and Figure 22, it shows that even though MyISAM has a higher compression ratio in WikiTitles, the relative throughput to the uncompressed version is much lower. When Head+Tail shows a stable and competitive performance, MyISAM's occasional competitiveness deserves further discussion. Based on this observation, we further explore some details about it.

5.4 Further Discussion

We sample 10M keys from each dataset above and run MyISAM and the uncompressed B-tree over them. The query performance are shown in Figure 23. We collect a series of valuable statistics as follows during the write/read process that may have an impact on performance:

Table 6 Uncompressed B-tree Statistic for Different Datasets

Datasets	Key Size	Fanout	Height	# Leaves
WikiTitles	25.19	57.42	4	2793
MemeTracker	79.65	22.11	6	10275
WEBSPAM	109.60	11.81	8	60598
TPC-H	131.69	13.77	6	61361

1. Average actual comparison length: The average length that is sufficient to distinguish between two keys, i.e., the number of bytes compared before return the comparison result.
2. Actual comparison proportion: the proportion of (1) to the average key length.
3. Suffix comparison ratio: During sequential scan, when the matched bytes are longer than the prefix of current key, a comparison on the suffix is needed to decide whether continue. We counted the percentage of keys where comparisons are needed during the scan.
4. Average suffix comparison length: The average length of the comparisons in (3).

To figure out factors affecting relative performance, we start from the performance of uncompressed B-tree, Table 6 displays some supporting information. As shown in the table, the search performance is strongly influenced by the height of trees, among which the throughput drops dramatically for WEBSPAM of the highest tree. TPC-H has longer keys on average than WEBSPAM, but has a lower tree height and a higher fan-out than it. The reason exists in the distribution of key length, as shown in Figure 25. The imbalance in key length leads to the inefficiency in uncompressed B-tree, which can be partially mitigated in MyISAM.

Between MemeTracker and TPC-H, the two have a same height, MemeTracker has lower throughput due to its long comparison length as shown in Figure 24b, even though its average key length is much shorter than TPC-H. From TPC-H to MemeTracker, the average comparison length changed from 8.92 to 25.67, resulting in an increase in proportion of comparison time in the total query time from 62.5% to 71.2% and thus the decrease in overall throughput.

Techniques that compress between neighboring keys, such as MyISAM, has high compression ratios for many real datasets and performs well at compression ratios of 2 or higher. However, for some easily distinguishable data, it cannot maintain the original high throughput rate.

6 Improving head+tail compression

As shown in Section 4 and Section 5, the Head+Tail compression consistently shows better insert and search through-

put while maintaining good compression ratios across various page sizes, key distributions, and domain sizes compared to other B-tree compression techniques. A detailed summary can be found in Section 8. In this section, we study more on the Head+Tail compression. In particular, we investigate techniques to further optimize it.

Lomet documented three specific optimization techniques for B-tree compression in [46]: *prefix vector representation*, *unrolled binary search*, and *key normalization*, which are discussed in the context of Head+Tail Compression specifically. However, that paper is a personal-experience-sharing style article that does not provide experimental results on the effectiveness of these optimizations. Moreover, that paper was published more than two decades ago, during which computer architecture has changed significantly. Thus, it is interesting to investigate whether and how these optimizations are effective in today's computing environment.

In this section, we implement these techniques (proposed in [46]) in the context of Head+Tail compression and evaluate their effectiveness.

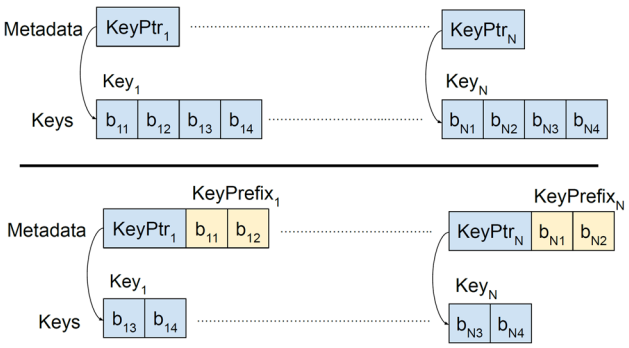


Fig. 26 Example of prefix vector Representation

Table 7 Prefix Vector Representation Performance Improvement

Mops/s	Insert	Search
No Compression	0.745 (+13.7%)	0.729 (+9.5%)
Head	1.026 (+37.2%)	1.008 (+36.8%)
Tail	0.841 (+6.5%)	0.872 (+7.3%)
Head+Tail	1.137 (+27.6%)	1.139 (+26.1%)
H+T Extra Nodes	+183 (+0.004%)	

6.1 Prefix Vector Representation

After Head+Tail compression, the keys in the B-tree may not have the same length (even if they did before compression). Performing binary search on variable-length keys incurs additional overhead due to redirection, as it requires lookups in a separate array for offsets and lengths. The main idea of the prefix vector representation (proposed in [46]) is to partition each key into two parts (shown in Figure 26): a fixed-length part and a variable-length part. All fixed-length parts go into the key prefix vector representation, while the variable-length parts are stored as they are. The length of the fixed-length key can be configured to two, four, or eight bytes. Binary search is applied on the fixed-length vector first, and only when the comparison result is equal do we need to traverse the suffix to determine the final comparison result.

The advantages of this optimization are at least two-fold: (1) Binary search on the fixed-length vector is faster than on variable-length keys without the redirection overhead; (2) The design is more cache-friendly, as the shorter fixed-length keys fit more easily into CPU caches. Lastly, this design is more suitable for an optimization to be introduced in Section 6.3.

6.1.1 Experimental Result

In this experiment, we set the page size to 1kB according to experimental setting in Section 4.1. We set the prefix size to 4 bytes (and study the impact of the prefix size in Section 6.1.2). Compared with the default in-memory settings in Section 4.1, the larger page size can further leverage the prefix vector's capability to reduce probe cost. We also evaluate the B-trees with and without Head+Tail Compression to further explore how the prefix vector representation works with Head+Tail, and the results are shown in Table 7.

Overall, the prefix vector representation shows an increase in throughput across all compression variants. Head+Tail remains the best-performing technique, with increases of **27%** and **26%** for insertion and search, respectively. By exiting key comparisons early, the prefix vector representation reduces probe costs. The experimental results indicate a promising performance enhancement for the B-tree optimization.

The most significant performance improvement is seen in Head Compression, with a **37%** and **36%** increase, which can be attributed to the great compatibility between the Head Compression technique and the prefix vector representation: both operate on prefixes. The page-level common prefixes lead to redundant comparisons for each key during binary search. Head Compression removes the common prefixes within a node, thus giving the prefix vector representation a greater chance to exit the search during prefix comparisons.

The performance improvement in Tail Compression is much less in comparison, to the point that its relative performance improvement from the prefix vector representation is lower than that of the uncompressed B-tree. This is mainly due to node-wide common prefixes. Moreover, the internal nodes in Tail Compression use the shortest keys as separators during node splitting. Sometimes this separator is shorter than the prefix size, which causes internal fragmentation in the metadata entries. This issue is more prevalent in higher-

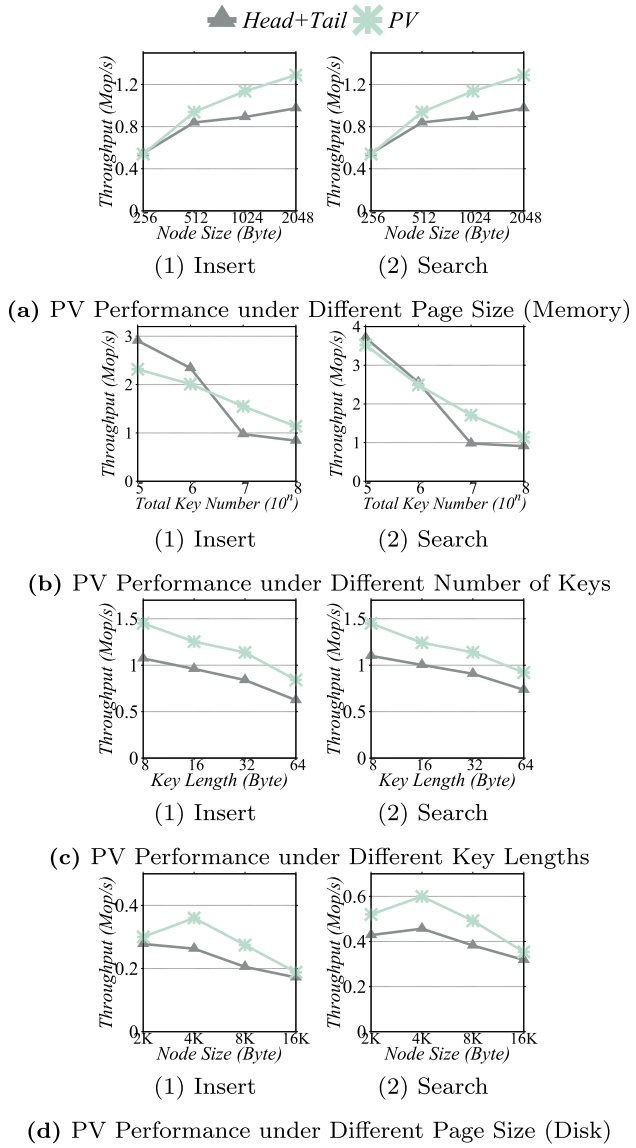


Fig. 27 PV performance under different parameters

level internal nodes (closer to the root) because the separator can often be only one or two bytes.

We also show the space overhead. According to the experiment, the prefix vector representation created **0.3%** more internal nodes (**0.004%** of total nodes) on Head+Tail Compression due to internal fragmentation (Table 7). These extra nodes may cause slight insertion performance losses and reduce compression effectiveness, but the percentage is negligible under our settings.

Figure 27 shows the performance of PV under different parameters. Generally, PV performs better than Head+Tail only, showing stable improvement over different page sizes and key lengths. PV and Head+Tail only share similar performance at a size of 256 bytes. The low number of stored keys per node means there are fewer chances of indirection. The entire node is likely to be pulled into the cache regardless of the optimization. We observe that PV provides an optimization for all page sizes up until 16KB. This suggests the node size is too large even for PV to optimize. Each binary search iteration is likely to result in a cache miss because even the prefix entries are too far apart.

Lastly, we see PV has worse throughput at 100K entries and below (Figure b). This may be because PV performs well if most internal nodes are shorter than the set PV size. Tail compression allows the minimum-length identifying key when splitting nodes. With a low number of input keys, the effect of PV's internal fragmentation on performance becomes more obvious. This, fortunately, can be resolved by lowering the set prefix size (e.g., 2 instead of 4). A dynamic system can be introduced that changes the prefix size according to the total number of stored keys. The required tree reformatting can be performed offline.

Table 8 Head+Tail Performance under Different Prefix Vector Hyperparameters

Mops/s	No PV	2 Bytes	4 Bytes	8 Bytes
Insertion	0.891	1.0632	1.137	1.057
Search	0.903	1.0632	1.139	1.090
Extra Nodes	baseline	+0.008%	+0.004%	+0.9%

6.1.2 Impact of Prefix Size

As mentioned above, the size of the prefix can affect performance and compression efficiency. In Section 6.1.1, 4 bytes was used as the default prefix size, and we study the impact of different prefix sizes in this experiment.

Table 8 compares the relative performance of different prefix sizes, specifically short (2 bytes) and long (8 bytes), against the Head+Tail B-tree with a prefix vector representation. The 4-byte prefix outperforms the other two sizes in both insertion and search. The results show that 2-byte prefixes provide some improvement over Head+Tail but fall short in throughput when compared to 4-byte prefixes. A similar pattern applies to 8-byte prefixes. This is because, although the larger prefix size allows for more frequent early exits, the longer metadata reduces cache locality, resulting in lower performance than 4-byte prefixes.

In terms of the space overhead due to fixed prefix length, the 8-byte prefix vector representation resulted in **0.9%** extra nodes compared to Head+Tail (Table 8). Although the extra nodes do not place a significant strain on search performance, they negatively affect insertion speed due to the additional node creation, splitting, and copying. This effect can be seen in the throughput difference between insertion and search for the 8-byte prefix.

Counterintuitively, the 2-byte prefixes had slightly more extra nodes than the 4-byte prefixes. After inspection, we believe the subtle difference between the prefix sizes affected the node layout and the Tail Compression's separator selection during the millions of insertions, consequently impacting the size and density of internal nodes. These effects then rippled throughout the entire tree in subsequent insertions. In our case, this anomaly caused a very small amount of storage overhead, merely **0.008%** of the total nodes. After observing the compression ratio in additional experiments, we found that a 2-byte prefix had nearly the same number of nodes as a normal B-tree from 10k to 10m keys (with at most 1 extra node), while using a 4-byte prefix consistently resulted in more internal nodes comparatively. This peculiarity hints at potential optimizations in Tail Compression, possibly in the form of split interval prediction during insertion or a version of post hoc tree node density balancing or separator recalculation.

6.1.3 Summary

Overall, the prefix vector representation proposed in [46] has been shown to improve Head+Tail compression by up to **27%** for insertion and search. The space overhead introduced is negligible.

6.2 Unrolled Binary Search

The drawback of conventional binary search is that it contains many hard-to-predict branches, which can affect performance. In [46], another optimization, termed "unrolled binary search" was mentioned to address this issue by removing as many branches as possible. Unrolled binary search is based on Shar's algorithm, which is first proposed in the book [39]. However, its effect on compressed B-trees was not studied in [46]. In this section, we conduct experiments to discuss the impact of unrolled binary search on B-tree compression.

6.2.1 Algorithm Description

Algorithm 1 shows Shar's algorithm [39]. The main loop body takes advantage of arrays whose sizes are powers of 2, where index calculations can be replaced by bit right-shifts. Shar's algorithm ensures that all arrays beyond the first comparison are of sizes that are powers of 2. There are several important constants concerning the array size N in Shar's algorithms defined as follows:

- $k = \lfloor \lg N \rfloor$.
- $i = 2^k$
- $l = \lceil \lg(N - i + 1) \rceil$.
- $i' = N + 1 - 2^l$.

Algorithm 1: Shar's Algorithm

```
   Input: Key, Node
   Output: index idx (with i, i', k, l precomputed)
1  Arr ← Node.keys // lowest key pointer
2  M ← i; idx ← i; cmp ← compare(Arr[idx], Key);
3  if cmp > 0 then
4  |  idx ← i'; M ← l;
5  else
6  |  if cmp < 0 then
7  |  |  idx ← idx − M/2;
8  |  else
9  |  |  return idx;
10 for M ← M/2 to 0 M/2 do
11 |  cmp ← compare(Arr[idx], Key);
12 |  if cmp > 0 then
13 |  |  idx ← idx + M;
14 |  else if cmp < 0 then
15 |  |  idx ← idx − M;
16 |  else
17 |  |  return idx;
18 return idx;
```

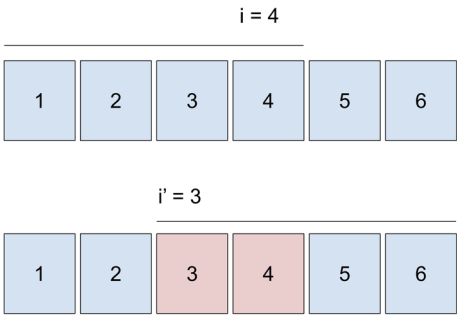


Fig. 28 Example of shar's algorithm. If the key is larger than 4, then i flip to i', and 3 and 4 become redundant

In line 2, the algorithm compares the input with $Arr[idx]$ where idx is initialized to the i, the largest power of two less than the length of the array. If the result indicates that the target lies to the left of $Arr[idx]$, then the binary search will be conducted normally with M initialized to i, which is a power of 2 and can be safely divided by two in each iteration. If the target lies to the right of $Arr[idx]$, then we set M as the smallest power of 2 larger than the length to the right of $Arr[idx]$ and conduct the same binary search on the remaining section.

Figure 28 shows an example of how l and M are adjusted before the loop body. In this case, we try to find 6 in an array of length of 6, so $M = i = 4$ (one-indexed for simplicity). Because the target 6 is larger than $Arr[i] = 4$, we flip to the right, then the $idx = i' + M = 5$ where $i' = 3$, $M = l = 2$. If such "flip" occurs, our new boundary can over-estimate to enforce lengths in powers of two. Notice that elements 3 and 4 are again within the search range that has not been

discarded by divide-and-conquer, indicated by i'. While this ensures correctness, it also means duplicated comparisons can exist in Shar's algorithm. Fortunately, the algorithm will never make more than $\lfloor \lg(n) \rfloor + 1$ comparisons because of divide-and-conquer. With that, Shar's algorithm is able to minimize the index calculation overhead and M can always shrink within a single right-shift instruction.

Algorithm 2: Unrolled Binary Search (UBS)

Input: Key, $Node$
Output: index (with i, i', k, l precomputed)

```
1  low ← Node.keys // lowest key pointer
2  M ← i;
3  if compare(low + M, Key) ≥ 0 then
4      low ← Node.keys + i' // flip to right
5      M ← l;
6  if M/2 ≥ 256 then
7      if (cmp = compare(low + 256, Key)) ≥ 0 then
8          low ← low + 256;
9  if M/2 ≥ 128 then
10     if (cmp = compare(low + 128, Key)) ≥ 0 then
11         low ← low + 128;
12 if M/2 = ... then
       // repeat for all power-of-two cases
13 if (cmp = compare(low + 1, Key)) ≥ 0 then
14     low ← low + 1;
15 return low − Node.keys;
```

According to [46], Shar's algorithm enables loop unrolling, which eliminates loop branching and index calculations entirely. Based on the brief description in [46], we implemented the unrolled binary search algorithm, which has no branching or index computation after the first probe and embeds displacements as constants, as shown in Algorithm 2. With the switch statement, the program jumps to the corresponding displacement and continues sequentially through all cases until each comparison is completed.

In addition, we employ another optimization technique: instead of accessing the index directly, we keep track of the lowest index, *low*, which points to the lowest element that has yet to be excluded by divide-and-conquer. With this new setup, only two conditions are required for each iteration: first, to check if the lowest index pointer needs to be increased (i.e., if the comparison result is greater than zero), and second, to check if the element has been found. This eliminates the need to update the pointer when the target element is in the lower half, as excluding elements is effectively achieved by halving the embedded constants with each iteration.

Note that the number of switch cases in Algorithm 2 depends on the page size. A larger page size will require a higher first case and more comparisons. The highest case

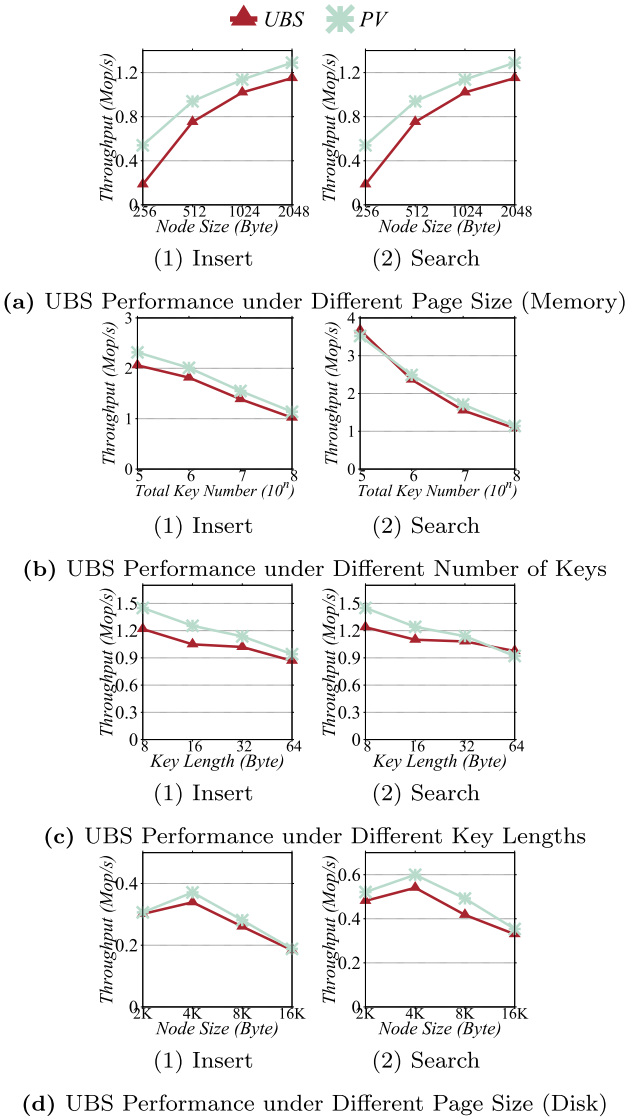


Fig. 29 UBS performance under different parameters

can be set to less than the page size by calculating the maximum number of keys a node can theoretically hold. By opting for a tighter theoretical upper bound, one can reduce the executable size and thereby increase performance. In our experiment, we use 256 as the upper bound for the 1024-byte page size. If the page size increases, manual adjustments to the number of cases in the switch statement may be required.

6.2.2 Experimental Result

Table 9 shows that UBS results in performance decrease both in search throughput (-10.3%) and insertion throughput (-5.92%). We analyze this behavior by examining the impact of unrolled binary search on the number of function calls made by the comparison function. As previously

Table 9 Unrolled Binary Search Result

$Mops/s$	Insertion	Search
Head+Tail+PV	1.137	1.139
Head+Tail+PV+UBS	1.021 (-10.3%)	1.0811 (-5.92%)

mentioned, Shar's algorithm may result in extra comparisons, where keys that could have been discarded remain due to "flip" bound over-estimation. The number of comparison function calls for insertion and search increased by **6.01%** and **5.83%**, respectively (Table 9). Since the percentage increases are approximately the same, we assume that duplicate comparisons in unrolled binary search uniformly affect both insertion and search throughput. Although unrolled binary search aims to eliminate loop branching statements, the resulting assembly still contains branch statements because the compiler hoists switch cases to take advantage of cache locality during optimization. This effect, however, should apply uniformly to both insertion and search (See Table 10).

UBS decreases the system throughput despite its complexity. We observed that UBS's assembly does not inline the comparison function as its original counterpart does. This means each switch case incurs function call overhead. Additionally, the switch statement does not have direct branching due to code hoisting, which means the program may go through multiple branches before reaching the correct case. Lastly, the additional comparisons mentioned earlier also put pressure on the program's throughput.

Figure 29 shows the performance of UBS under different parameters. The number of switch cases is adjusted accordingly for larger page sizes. We observe a uniform performance degradation across all settings when compared to PV.

6.2.3 Summary

Overall, unrolled binary search does not improve performance. The additional comparisons do not make UBS a good optimization under relative high comparison cost environments, such as in database systems.

6.3 Key Normalization

In [46], another optimization called "Key Normalization" (KN) was developed to reduce the cost of key comparison. Instead of comparing two keys in a byte-wise fashion, KN compares two keys word-wise to reduce the number of comparisons. However, on little-endian machines, we cannot directly convert two string keys into words due to correctness issues, requiring byte-order reversion.

The downside of the fixed-word comparison implementation is that if the length of the string is not divisible by the word length, then empty or null bytes must be padded to ensure correctness. This requirement often conflicts with compression techniques. In the case of head compression, extracting a common head byte may or may not reduce a key's size depending on its length. For example, in a worst-case scenario, extracting $L_{word} - 1$ bytes of common prefixes from all keys could result in *no compression effect* because all compression is offset by null padding bytes at the end of the keys. In this section, we conduct experiments to explore the effects of fixed-word comparisons on performance and storage.

For key normalization, we will study two variants: key-prefix only (**KP**) and full normalization (**FN**). Because the prefix vector representation already has word-length prefixes, normalizing them for word-size comparison will be straightforward. Full key normalization rounds up all key lengths that are not divisible by the word length and pads null bytes at the end of the key.

Due to the nature of Head Compression, implementing key suffix normalization becomes rather complex. Typically, Head Compression is achieved through a simple pointer shift: increasing the local key pointer by the node's prefix size provides the compressed key for the corresponding node. However, this mechanism is not compatible with key normalization, where the bytes are out of order. If one were to compress the search key before entering the search function, a linear string conversion would be required to place the bytes in the correct index each time the key enters a node with a compressed prefix.

Instead, we opt for a slightly different implementation. Rather than normalizing the incoming key before entering a node, a key is normalized only when it is inserted into the B-tree node. To facilitate word-sized comparisons, we use the BSWAP instruction[2] (available on x86 to convert the byte order from the little-endian format to the big-endian format) during the comparison function. With this approach, we can perform normal pointer arithmetic for head compression, apply BSWAP byte ordering locally, and compare it with normalized keys with minimal overhead. Finally, the normalized prefix of the incoming key can be stored locally at the start of a node search function, which avoids redundant BSWAP instructions during comparison. A word-length of null bytes is padded at the tail of the incoming keys to ensure that garbage values do not affect head-compressed word comparisons.

[2] https://www.felixcloutier.com/x86/bswap

Table 10 Number of Comparison Function Calls

# of Function Calls	Insertion	Search
Head+Tail+PV	2592998804	2102982597
Head+Tail+PV+UBS	2748905177 (+6.01%)	2225652709 (+5.83%)

Table 11 Key Prefix-only and Full Normalization Implementation over Prefix Vector Results

Mops/s	KP Insert	KP Search	FN Insert	FN Search
No Compression	0.771	0.735	0.794	0.784
Head	1.108	1.070	1.082	1.070
Tail	0.867	0.902	0.908	0.935
Head+Tail	1.189 (+4.6%)	1.173 (+2.9%)	1.192 (+4.9%)	1.169 (+2.6%)
H+T Extra Nodes	+0		+360260 (+7.3%)	

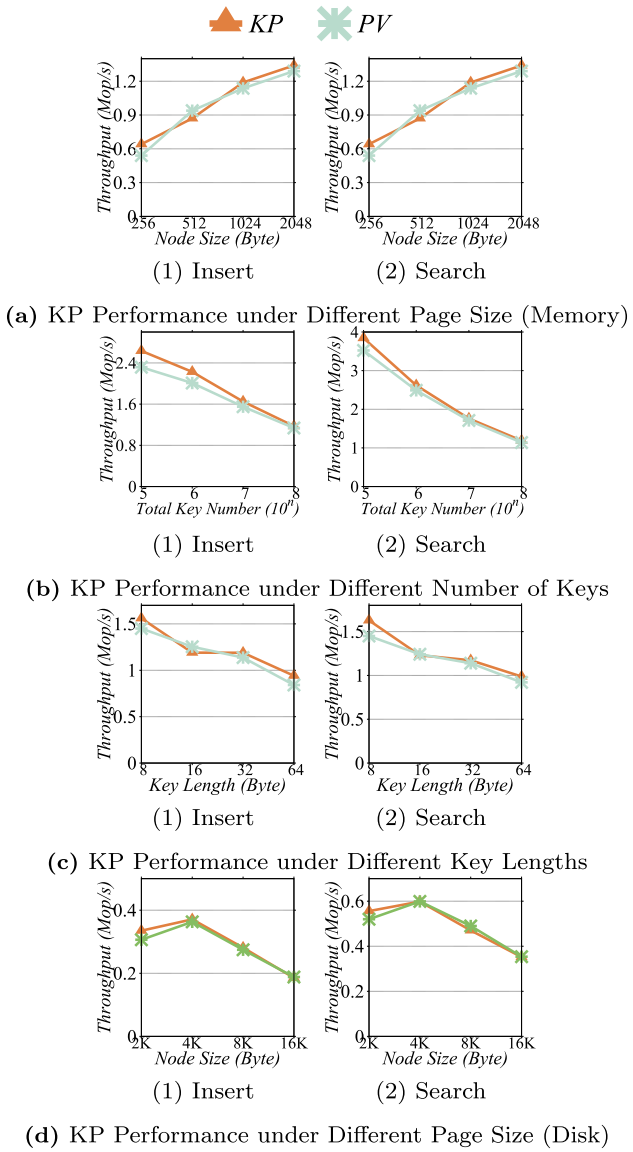


Fig. 30 KP performance under different parameters

6.3.1 Experimental Result

We implemented key prefix-only and full normalization on top of the prefix vector representation. Table 11 shows the results. Compared to Head+Tail+PV (prefix vector representation), prefix-only normalization improved insertion and search by **4.6%** and **2.9%**, and full normalization improved insertion and search by **4.9%** and **2.6%**. The performance increase between the two techniques is similar. We also find that full normalization increases the total number of nodes by **7.3%** for Head+Tail compression, a significant increase compared to the prefix vector representation. Prefix-only normalization does not affect the tree size (i.e. the space overhead).

Although FN theoretically has a lower comparison cost, the reduced compression ratio means FN will spend more time during binary search. Most searches exit during prefix comparisons (at most a few bytes into the suffix). Therefore, the reduction in comparison cost contributed by FN is realized only in a few cases, with the main exception being when the key is found in the leaf node. In cases where the comparison cost is high (i.e., no compression and Tail compression), FN provides a performance boost over KP, whereas KP and FN show similar throughput when the comparison cost is low (i.e., Head compression and Head+Tail).

We explain more of the implementation details. First, unlike KP, the keys during FN insertions are BSWAP'ed and inserted as a word instead of being copied one by one, which may increase insertion throughput. This may give FN a slight advantage over KP and PV (Prefix Vector Representation). For KP and FN, we constructed a constant-space (i.e., malloc-less) key-copying algorithm that handles the prefix vector, head compression, and key normalization simultaneously, instead of denormalizing the key into an allocated buffer and then reinserting it into the new node. This approach is similar to PV, where the keys are copied to the new node byte by byte.

Since FN does not provide additional optimization given the space tradeoff, we opted to test only KP's performance over different settings. Figure 30 shows the performance of KP versus PV under different parameters. The data suggests that KP tends to outperform PV across the board, although the degree of optimization is limited.

6.3.2 Summary

Overall, the prefix-only normalization does not improve performance much. Full Normalization requires null padding, which decreases the compression ratio and does not show a significant improvement over key prefix-only normalization. Unless tasks involve a large number of key suffix comparisons (i.e., high comparison cost), which is rare under Head+Tail compression, Full Normalization is not necessary.

7 Related work

In this section, we present relevant work to this paper.

Compression is an important topic in databases. In addition to B-tree compression, which is the main focus of this paper, there are many works on compression in other levels.

To achieve the best compression ratio, various compression techniques are developed for different types of data, such as dictionary encoding [25], run-length encoding [13], integer compression, bitmap compression [38, 41, 53], float compression [44, 48], and text compression [56, 57].

This paper focuses on traditional B-trees due to their widespread usage. Even many in-memory databases also use B-trees or their variants. For example, Microsoft's in-memory database Hekaton uses the Bw-tree [28]. SAP's main-memory database HANA uses the CPB+-tree, a compressed prefix B+tree [6]. H-store, another main-memory database, also uses a B-tree [36]. There are also some other indexes that have been designed with space-saving purposes in mind. Bumbulis and Bowman developed a compact B-tree [23], which uses a local Patricia tree (a variant of trie) to represent the keys inside each node instead of using an array. Many radix trees like ART [42] save space by sharing path information. Zhang et al. developed an order-preserving key compression for in-memory search trees [59]. That work uses a dictionary-based order-preserving compression strategy for strings, essentially reducing the size of keys before inserting them into the tree structure. Similar works can be found in [15, 16, 20, 45]. However, they are not as widely used in commercial database systems as B-trees because B-trees are simple and mature, support both point and range queries efficiently, and are compatible with both memory and disk.

This work is also relevant to some prior B-tree implementations, such as [49, 50], that have used certain forms of head and tail compression. Although [1] uses the same term, it is completely different from the key normalization described in this paper or in Lomet's paper [46]. The key normalization in [1] focuses on transforming different key index formats and data types into a single representation so that the results can be easily compared. The motivation is that some data types (e.g., datetime) have more complex and costly comparison functions. It transforms all such keys into a format usable by memcmp to reduce comparison cost. In contrast, the key normalization in Lomet's paper [46] involves switching the positions of bytes to allow multi-byte comparisons.

It is interesting to note that the prefix vector representation shares a similar design philosophy with the recent German-style string representation [35, 47], which uses a set of constant prefixes for variable-length strings to reduce space. However, the prefix vector representation maintains a small, fixed prefix size (typically 2 or 4 bytes) for all strings within a B-tree node, thereby improving search performance through direct binary search and better cache locality. Note that both techniques target different use cases. The prefix vector representation is specific to B-trees, where many keys are integers, floats, or small-sized strings. In contrast, the German-style string representation is a general-purpose string representation used in more complex settings, such as Apache Arrow and Apache DataFusion [35].

Another line of research is learned indexes [14, 29, 40] that use machine learning techniques to predict the key positions in B-trees. While this is innovative, it remains to address many practical issues, such as slow updates and concurrency control, to be widely adopted in real database systems.

While there have been prior surveys and experimental studies evaluating compression techniques for other data types, like bitmap [51], to the best of our knowledge, no study has compared the performance of various B-tree compression techniques. This comparison is the main contribution of our work.

8 Summary and conclusion

In this section, we summarize the main findings from all experiments on evaluating B-tree compression techniques.

8.1 Summary

Search Performance. It is interesting to see that the Head+Tail Compression, although proposed in the 1970s [18], consistently exhibits the best search performance among all B-tree compression techniques. It achieves a **25%** $\sim$ **120%** performance improvement over the uncompressed B-tree across different datasets. This is impressive, especially considering that the B-tree is a highly optimized and widely used index in database systems.

Moreover, Head+Tail Compression also outperforms other newer compression techniques for B-trees. For instance, it is **21.7%** $\sim$ **90.2%** faster than DB2 [19], **129%** $\sim$ **684%** faster than MongoDB WiredTiger [12], **7.3%** $\sim$ **139%** faster than MySQL MyISAM [8]. The high performance is attributed to the fact that Head+Tail Compression supports query processing directly on the compressed B-tree without needing decompression.

Insert Performance. For insertion performance, the Head+Tail Compression also shows the best performance compared to other B-tree compression techniques. It achieves a **10%** $\sim$ **30%** performance improvement over the uncompressed B-tree.

Furthermore, only the Head, Tail, and Head+Tail Compression techniques outperform uncompressed B-tree in insertion. Other compression techniques (including DB2, WiredTiger, MyISAM, and PkB) are slower than B-tree during insertion because of their compression complexity.

Space Overhead. For space overhead, Head+Tail and WiredTiger are the top two candidates, achieving a compression ratio ranging from **1.2X** $\sim$ **3.5X** across various datasets. Notably, in datasets with randomized keys, Head+Tail offers the most significant space savings in most cases. Meanwhile, for real-world datasets where keys have certain semantics, such as names, sentences, and URLs, WiredTiger and MyISAM are the most space-efficient.

Optimizations to Head+Tail Compression. Prefix vector representation can improve Head+Tail compression by up to **27%** for insertion and search. However, both unrolled binary search and key normalization do not enhance performance much (up to 3%).

8.2 Overall Recommendations

Overall, we recommend Head+Tail Compression for B-trees. It has the best search and insert performance while maintaining a decent compression ratio. More importantly, Head+Tail achieves faster performance than uncompressed B-trees. In addition, prefix vector representation is also recommended to further improve the Head+Tail compression.

Note that although WiredTiger and MyISAM can incur lower space than Head+Tail in some datasets, considering the lower search and insert performance, we still think Head+Tail is better.

8.3 Conclusion

In this paper, we conduct the first experimental study to compare the performance of widely-used B-tree compression techniques since they have never been compared against each other in prior studies. We find that the oldest technique, Head+Tail Compression, proposed in the 1970s, turns out to be the best across various datasets and scenarios in general. Also, prefix vector representation can further improve the Head+Tail Compression.

There are many interesting research directions to explore in the future. We plan to implement B-tree compression within database systems such as PostgreSQL and MySQL, and we expect enhanced performance. We also plan to improve B-tree compression in the following ways. For example, introducing Tail or Head compression to other compression algorithms (e.g., MyISAM) can lead to even better performance. It is also interesting to optimize B-tree compression for composite keys because simply combining the keys may not achieve a better compression ratio. Optimizing B-tree compression for various hardware platforms is also promising, given their different B-tree variants. Additionally, using machine learning to develop more efficient B-tree compression techniques is an interesting direction.

Acknowledgements Jianguo Wang acknowledges the support of the National Science Foundation under Grant Number 2337806.

References

1. Key Normalization (https://wiki.postgresql.org/wiki/Key_normalization)
2. Webspam-uk2007 dataset (2007). https://chato.cl/webspam/datasets/uk2007/
3. Snap memetracker dataset (2008). https://www.kaggle.com/datasets/snap/snap-memetracker
4. The default page size change of sqlite 3.12.0 (2022). https://www.sqlite.org/pgszchng2016.html
5. Source code of wiredtiger's b-tree implementation (2022). https://github.com/wiredtiger/wiredtiger/tree/develop/src/btree
6. CREATE INDEX Statement in SAP HANA (https://help.sap.com/docs/SAP_HANA_PLATFORM/4fe29514fd584807ac9f2a04f6754767/20d44b4175191014a940afff4b47c7ea.html) (2023)
7. Database page layout in postgresql 16 (2023). https://www.postgresql.org/docs/current/storage-page-layout.html
8. Myisam source code in mysql (2023). https://github.com/mysql/mysql-server/tree/a246bad76b9271cb4333634e954040a970222e0a/storage/myisam

9. Mysql reference manual (2023). https://dev.mysql.com/doc/refman/8.0/en/key-space.html#:~:text=Prefix%20compression%20is%20used%20on,when%20you%20create%20the%20table
10. Tpc-h benchmark (2023). https://www.tpc.org/tpch/
11. Wikipedia articles titles (2023). https://dumps.wikimedia.org/enwiki/latest/
12. Wiredtiger documentation (2023). https://source.wiredtiger.com/11.1.0/file_formats.html#file_formats_compression
13. Abadi, D.J., Madden, S., Ferreira, M.: Integrating compression and execution in column-oriented database systems. In: SIGMOD, pp. 671–682 (2006)
14. Al-Mamun, A., Wu, H., He, Q., Wang, J., Aref, W.G.: A Survey of Learned Indexes for the Multi-dimensional Space. CoRR **abs/2403.06456** (2024)
15. Antoshenkov, G.: Dictionary-Based Order-Preserving String Compression. VLDB J. **6**(1), 26–39 (1997)
16. Antoshenkov, G., Lomet, D., Murray, J.: Order Preserving String Compression. In: ICDE, pp. 655–663 (1996)
17. Bayer, R., McCreight, E.M.: Organization and Maintenance of Large Ordered Indices. Acta Informatica **1**, 173–189 (1972)
18. Bayer, R., Unterauer, K.: Prefix B-Trees. ACM Transactions on Database Systems (TODS) **2**(1), 11–26 (1977)
19. Bhattacharjee, B., Lim, L., Malkemus, T., Mihaila, G.A., Ross, K.A., Lau, S., McCarthur, C., Toth, Z., Sherkat, R.: Efficient Index Compression in DB2 LUW. VLDB **2**(2), 1462–1473 (2009)
20. Binnig, C., Hildenbrand, S., Färber, F.: Dictionary-based Order-preserving String Compression for Main Memory Column Stores. In: SIGMOD, pp. 283–296 (2009)
21. Bohannon, P., McIlroy, P., Rastogi, R.: Main-Memory Index Structures with Fixed-Size Partial Keys. In: S. Mehrotra, T.K. Sellis (eds.) SIGMOD, pp. 163–174 (2001)
22. Breddemann, L.: What is cpb+-tree in sap hana? (2020). https://www.lbreddemann.org/what-is-cpb-tree-in-sap-hana/
23. Bumbulis, P., Bowman, I.T.: A Compact B-tree. In: SIGMOD, pp. 533–541 (2002)
24. Canadi, I.: Converged index: The secret sauce behind rockset's fast queries (2019). https://rockset.com/blog/converged-indexing-the-secret-sauce-behind-rocksets-fast-queries/
25. Chen, Z., Gehrke, J., Korn, F.: Query Optimization In Compressed Database Systems. In: SIGMOD, pp. 271–282 (2001)
26. Christman, G.: Compression features included with oracle database enterprise edition (2022). https://blogs.oracle.com/dbstorage/post/compression-features-included-with-oracle-database-enterprise-edition
27. Comer, D.: The Ubiquitous B-Tree. CSUR **11**(2), 121–137 (1979)
28. Diaconu, C., Freedman, C., Ismert, E., Larson, P., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M.: Hekaton: SQL server's memory-optimized OLTP engine. In: SIGMOD, pp. 1243–1254 (2013)
29. Ding, J., Minhas, U.F., Yu, J., Wang, C., Do, J., Li, Y., Zhang, H., Chandramouli, B., Gehrke, J., Kossmann, D., Lomet, D.B., Kraska, T.: ALEX: An Updatable Adaptive Learned Index. In: SIGMOD, pp. 969–984 (2020)
30. Gao, C., Ballijepalli, S., Wang, J.: Revisiting B-tree Compression: An Experimental Study. Proc. ACM Manag. Data **2**(3), 169:1–169:25 (2024)
31. Graefe, G.: A Survey of B-tree Locking Techniques. TODS **35**(3), 16:1–16:26 (2010)
32. Graefe, G.: Modern B-Tree Techniques. Foundations and Trends in Databases **3**(4), 203–402 (2011)
33. Graefe, G., Larson, P.: B-Tree Indexes and CPU Caches. In: ICDE, pp. 349–358 (2001)
34. Hankins, R.A., Patel, J.M.: Effect of node size on the performance of cache-conscious b+-trees. In: Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '03, p. 283-294. Association for Computing Machinery, New York, NY, USA (2003). https://doi.org/10.1145/781027.781063
35. Hao, X., Lamb, A.: Using stringview / german style strings to make queries faster: Part 1- reading parquet (2024). https://datafusion.apache.org/blog/2024/09/13/string-view-german-style-strings-part-1/
36. Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S.B., Jones, E.P.C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., Abadi, D.J.: H-store: A High-performance, Distributed Main memory Transaction Processing System. Proc. VLDB Endow. **1**(2), 1496–1499 (2008)
37. Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A.D., Kaldewey, T., Lee, V.W., Brandt, S.A., Dubey, P.: FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In: SIGMOD, pp. 339–350 (2010)
38. Kim, S., Lee, J., Satti, S.R., Moon, B.: SBH: Super Byte-aligned Hybrid Bitmap Compression. Inf. Syst. **62**, 155–168 (2016)
39. Knuth, D.E.: The Art of Computer Programming, Volume 3: Sorting and Searching, *The Art of Computer Programming*, vol. 3, 2nd edn. Addison-Wesley, Reading, Massachusetts (1998)
40. Kraska, T., Beutel, A., Chi, E.H., Dean, J., Polyzotis, N.: The Case for Learned Index Structures. In: SIGMOD, pp. 489–504 (2018)
41. Lang, H., Beischl, A., Leis, V., Boncz, P.A., Neumann, T., Kemper, A.: Tree-Encoded Bitmaps. In: SIGMOD, pp. 937–967 (2020)
42. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: Artful indexing for main-memory databases. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE), pp. 38–49 (2013). https://doi.org/10.1109/ICDE.2013.6544812
43. Levandoski, J.J., Lomet, D.B., Sengupta, S.: The Bw-Tree: A B-tree for New Hardware Platforms. In: ICDE, pp. 302–313 (2013)
44. Liu, C., Jiang, H., Paparrizos, J., Elmore, A.J.: Decomposed Bounded Floats for Fast Compression and Queries. PVLDB **14**(11), 2586–2598 (2021)
45. Liu, C., Umbenhower, M., Jiang, H., Subramaniam, P., Ma, J., Elmore, A.J.: Mostly Order Preserving Dictionaries. In: ICDE, pp. 1214–1225 (2019)
46. Lomet, D.B.: The Evolution of Effective B-tree: Page Organization and Techniques: A Personal Account. SIGMOD Record **30**(3), 64–69 (2001)
47. Neumann, T., Freitag, M.J.: Umbra: A Disk-Based System with In-Memory Performance. In: Conference on Innovative Data Systems Research (CIDR) (2020)
48. Pelkonen, T., Franklin, S., Cavallaro, P., Huang, Q., Meza, J., Teller, J., Veeraraghavan, K.: Gorilla: A Fast, Scalable. In-Memory Time Series Database. PVLDB **8**(12), 1816–1827 (2015)
49. Schmeißer, J., Schüle, M.E., Leis, V., Neumann, T., Kemper, A.: B^2-Tree: Page-Based String Indexing in Concurrent Environments. Datenbank-Spektrum **22**(1), 11–22 (2022)
50. Theodorakis, G., Clarkson, J., Webber, J.: An Empirical Evaluation of Variable-length Record B+Trees on a Modern Graph Database System. In: International Conference on Data Engineering Workshops (ICDEW), pp. 343–349 (2024)
51. Wang, J., Lin, C., Papakonstantinou, Y., Swanson, S.: An Experimental Study of Bitmap Compression vs. Inverted List Compression. In: SIGMOD, pp. 993–1008 (2017)
52. 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: SIGMOD, pp. 473–488 (2018)
53. Wu, K., Otoo, E.J., Shoshani, A.: Optimizing Bitmap Indices with Efficient Compression. TODS **31**(1), 1–38 (2006)
54. Xu, H., Li, A., Wheatman, B., Marneni, M., Pandey, P.: Bp-tree: Overcoming the point-range operation tradeoff for in-memory b-trees. Proc. VLDB Endow. **16**(11), 2976-2989 (2023).https://doi.org/10.14778/3611479.3611502
55. Zhan, C., Su, M., Wei, C., Peng, X., Lin, L., Wang, S., Chen, Z., Li, F., Pan, Y., Zheng, F., Chai, C.: AnalyticDB: Real-time OLAP

Database System at Alibaba Cloud. VLDB **12**(12), 2059–2070 (2019)
56. Zhang, F., Wan, W., Zhang, C., Zhai, J., Chai, Y., Li, H., Du, X.: CompressDB: Enabling Efficient Compressed Data Direct Processing for Various Databases. In: SIGMOD, pp. 1655–1669 (2022)
57. Zhang, F., Zhai, J., Shen, X., Wang, D., Chen, Z., Mutlu, O., Chen, W., Du, X.: TADOC: Text Analytics Directly on Compression. VLDB J. **30**(2), 163–188 (2021)
58. Zhang, H., Andersen, D.G., Pavlo, A., Kaminsky, M., Ma, L., Shen, R.: Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In: SIGMOD, pp. 1567–1581 (2016)
59. Zhang, H., Liu, X., Andersen, D.G., Kaminsky, M., Keeton, K., Pavlo, A.: Order-Preserving Key Compression for In-Memory Search Trees. In: SIGMOD, pp. 1601–1615 (2020)
60. Ziegler, T., Vani, S.T., Binnig, C., Fonseca, R., Kraska, T.: Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In: SIGMOD, pp. 741–758 (2019)