Revisiting B-tree Compression: An Experimental Study

CHUQING GAO, Purdue University, USA
SHREYA BALLIJEPALLI, Purdue University, USA
JIANGUO WANG, Purdue University, USA

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 the first 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 that can be leveraged to guide system design decisions in modern databases regarding the use of B-tree compression.

CCS Concepts: • **Information systems** → **Data access methods**; **Data compression**.

Additional Key Words and Phrases: B-trees, Database Indexes, B-tree Compression

ACM Reference Format:
Chuqing Gao, Shreya Ballijepalli, and Jianguo Wang. 2024. Revisiting B-tree Compression: An Experimental Study. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 169 (June 2024), 25 pages. https://doi.org/10.1145/3654972

1 INTRODUCTION

Since the inception in the 1970s [14, 25], 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, GPUs, and the cloud). As a result, almost all major database systems have implemented B-trees or their variants. Note that we use B-trees to mean B+-trees in this work where the leaf nodes store all the keys.

Over the past few decades, there are many works proposed to optimize B-trees from various aspects, e.g., concurrency control [28], lock-free design [38], page layout [41], node size [31], compression [15], cache-aware optimizations [30, 33], and RDMA-optimized design [52]. Graefe provides an excellent survey on modern techniques of B-trees [29].

This paper revisits *B-tree compression*, a technique that was initially proposed in 1977 [15] aiming to reduce the space overhead and improve performance for B-trees. Two types of B-tree compression techniques were proposed in [15]: *Tail Compression* and *Head Compression* (see Sec. 2 for details).

Authors' addresses: Chuqing Gao, gao688@purdue.edu, Purdue University, West Lafayette, Indiana, USA; Shreya Ballijepalli, sballije@purdue.edu, Purdue University, West Lafayette, Indiana, USA; Jianguo Wang, csjgwang@purdue.edu, Purdue University, West Lafayette, Indiana, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2836-6573/2024/6-ART169
https://doi.org/10.1145/3654972

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 [15]. As all keys in the same B-tree node are 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 a recent technical blog from Oracle, "Indexes often take up to 50% of the total database space and it is not uncommon to have 10–20 indexes on a single table" [24]. Moreover, numerous modern databases (e.g., Rockset [21] and AnalyticDB [48]) 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% ∼ 78% of the overall cost [16]. 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 [15]. By reducing the key sizes, the time spent on comparing keys for query and insert will be lower.

As a result, many real-world database systems have supported B-tree compression. Examples include DB2 [16], MySQL (MyISAM) [8] and MongoDB WiredTiger [10].

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 [16] or WiredTiger [10], can outperform the Head and Tail Compression techniques proposed in 1977 [15]. Furthermore, it remains unknown whether the Head and Tail Compression techniques proposed in 1977 [15] are indeed effective in improving uncompressed B-trees because the only available experimental results we are aware of were found in [15], which were conducted based on the hardware of the 1970s, a setup completely different 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, which is the main contribution of this work. 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.

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) the scenarios in which a compression algorithm should be employed for B-trees and determine which algorithm is most suitable.

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

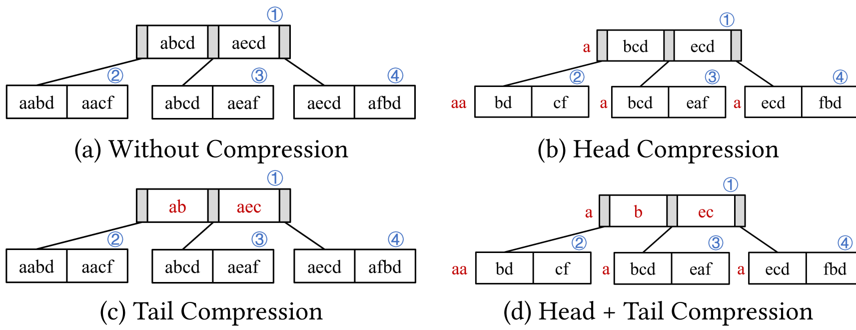


Fig. 1. Example of Head and Tail Compression

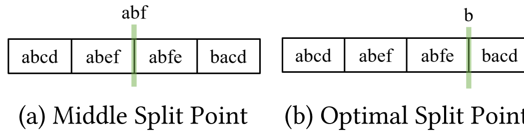


Fig. 2. Choosing a Split Point in Tail Compression

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.

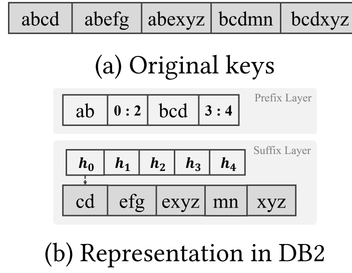
2.1 Tail Compression

The Tail Compression technique was introduced in 1977 [15]. 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 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.

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.



(a) Original keys

(b) Representation in DB2

Fig. 3. Example of DB2 Compression

2.2 Head Compression

Head Compression was introduced in the same paper as Tail Compression in 1977 [15]. 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.

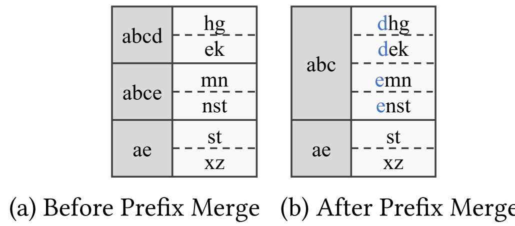
2.3 Head+Tail Compression

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 [16]. 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



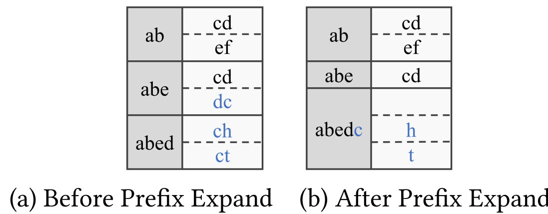
(a) Before Prefix Merge (b) After Prefix Merge

Fig. 4. DB2 Prefix Merge Illustration

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 - 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 ["abcd", "abce"], which can be merged to a single group with the prefix "abc".

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'.



(a) Before Prefix Expand (b) After Prefix Expand

Fig. 5. DB2 Prefix Expand Illustration

2.5 MongoDB WiredTiger

2.5.1 Prefix Compression. MongoDB WiredTiger's row-store storage format supports another variant of prefix compression [10] 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".

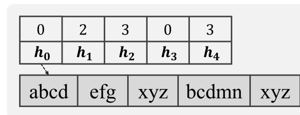


Fig. 6. Representation in WiredTiger

Hence, the second key stores the suffix bytes as "efg" and the 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 [4] similar to the Tail Compression in [15]. 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 Sec. 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 [8] by identifying prefixes between adjacent keys. It uses a representation similar to WiredTiger (Sec. 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.

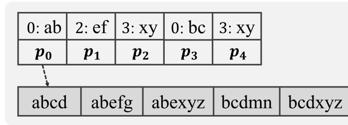


Fig. 7. Example of PkB Representation

2.7 PkB-Tree

PkB-Trees were first introduced in [18] and are based on a partial-key approach. SAP HANA adopts this concept and uses a variant known as the CPB-tree [19]. 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 in the metadata: (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.

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 partial key can help avoid the full comparison. In scenarios where the partial key is insufficient to obtain the result, 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 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.

The original paper [18] 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.

3 EXPERIMENTAL SETUP

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. For disk-based operations, we used a 1.6TB NVMe SSD. The read/write bandwidth is 1GB/3GB per second. 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 papers. 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 Sec. 3.3.

Datasets. In this paper, all the following experiments divide the datasets into two parts accounting for 20% and 80% unless specified. The first part is used in warmup phase while the second is

Table 1. Key Features in Different Datasets

Datasets	Num	Min Size	Max Size	Avg Size
TPC-H	60M	96B	156B	129.58B
WEBSPAM-UK2007	25M	16B	2047B	112.33B
WikiTitles	25M	1B	255B	19.34B
MemeTracker URLs	5.5M	8B	21.2KB	75.90B

for run phase and we only measure the run phase. For the search-like operations, we insert the 100% data before the search starts. We include both real world datasets (as shown in Table 1) and random-generated synthetic datasets.

Parameters. The following experiment results are averaged under three runs, and their 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 [18]. 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 [8, 10, 15, 16, 18, 38, 45].

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 Sec. 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 [16] 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 [7].

We implement the WiredTiger compression based on its source code [4]. 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

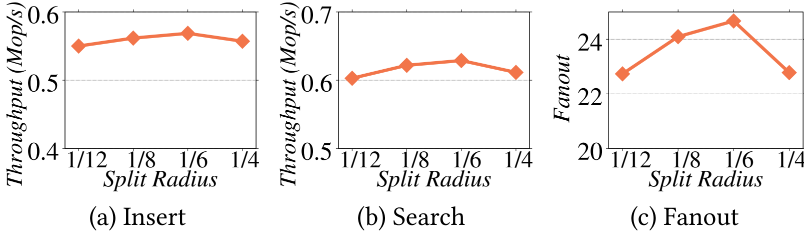


Fig. 8. Tail Compression over Different Split Ranges

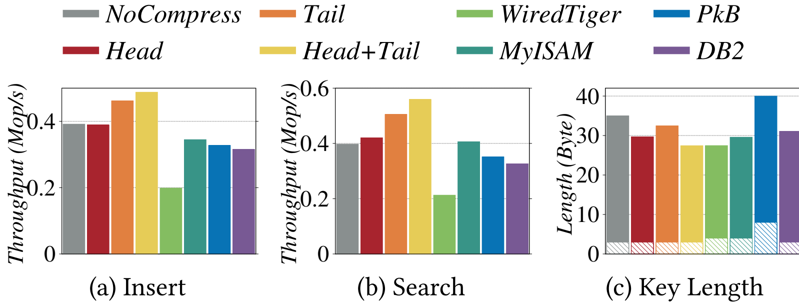


Fig. 9. Overall Results in Memory under Default Settings

in Sec. 2.5, we explore their applicability in our benchmark by some preliminary experiments based on a workload of the default setting in Sec. 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 gave 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 [4] is 10, which implies that this instantiation technique may not be very helpful for purely in-memory scenario. 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 levels of the B-tree 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.

4.1 Overall Results

Table 2. Default Experiments Settings

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

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, MyISAM and PkB, the three algorithms based on the delta compression, are 49.4%, 12.1% and 16.4% 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 40.7%, which is attributed to the synergistic effect of Head and Tail Compression. Head Compression increases the throughput by 5.7% 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 27.2% 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 Sec. 3.3, how to fine-tune these details to reduce prefix search overhead is beyond the scope of this paper.

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 when 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.

Table 3. Statistics on B-trees in Default Memory Setting

Techniques	Height	Fan-out	# Nodes	# Non-leaf Nodes
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

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.

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 Sec. 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 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.

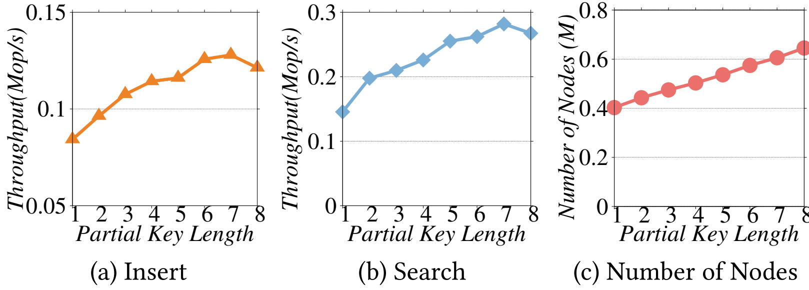


Fig. 10. Performance under Different Partial Key Length

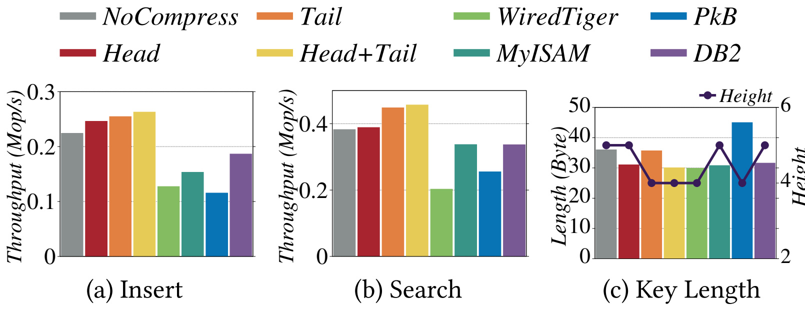


Fig. 11. Overall Results on Disk under Default Settings

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 Sec. 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

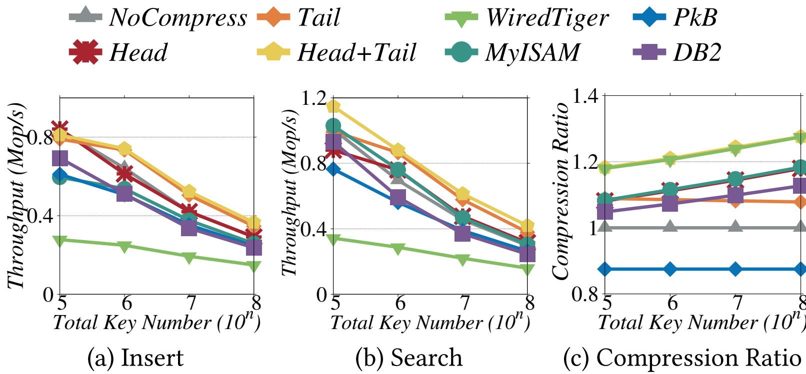


Fig. 12. Performance under Different Number of Keys

show a more significant space saving. This is reflected in Figure 12c. As the number of keys 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/WiredTiger 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 and finally is overtaken by PkB under the case of the highest number of keys, 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 a 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 within a node is roughly the same for keys of different lengths. However, when the total page size is fixed, increasing the key length reduces the number of keys within the page. This leads to two effects, one is the number of keys within a node becomes less, another is that the common prefix of the fewer keys would become longer. The 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, and therefore reduce page utilization.

On the other hand, some compression methods benefit from longer keys. For example, the length of separators, as we discussed above, does not change much. Thus, the compression efficiency in Tail Compression is significantly improved. For techniques based on delta compression, decreasing in number of keys within a page shortens their path for decompression, which therefore narrows

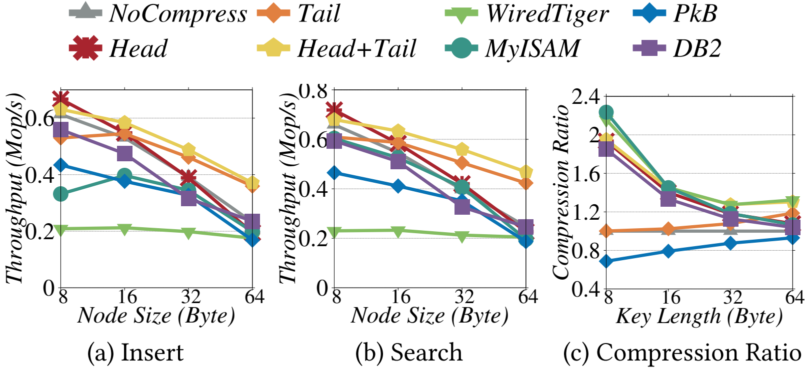


Fig. 13. Performance under Different Key Lengths

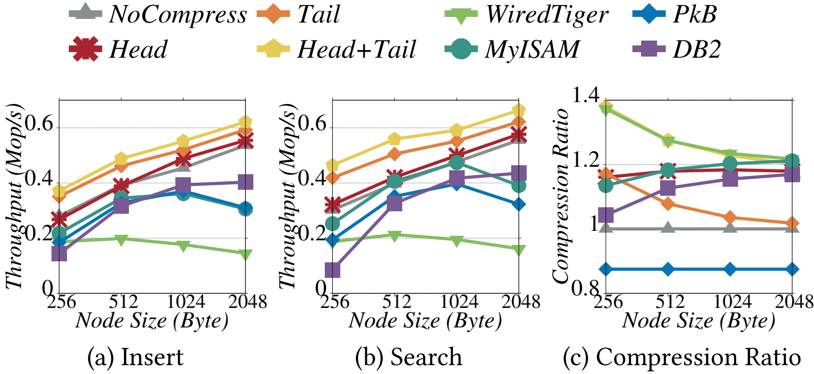


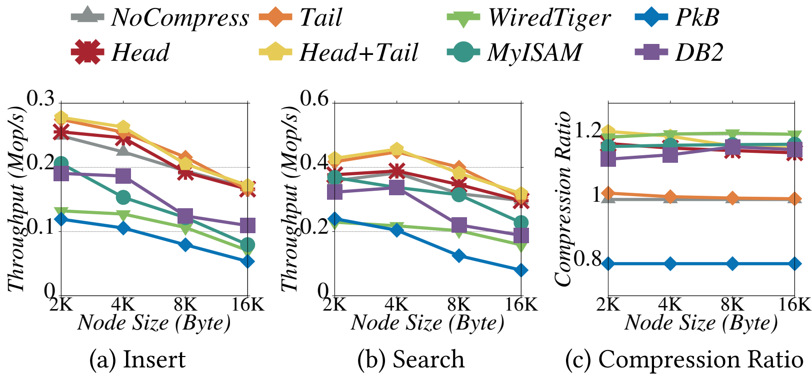
Fig. 14. Performance under Different Page Sizes (Memory)

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

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 of the shorter separators. 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.



(a) Insert (b) Search (c) Compression Ratio

Fig. 15. Performance Under Different Page Sizes (Disk)

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 optimization strategies only work when there are multiple prefixes within a node. When the number of prefixes is 1, the 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, a common prefix for all keys may not effectively compress the length of the keys, thus the multiple prefixes of DB2 can be helpful.

Table 4. Tail Compression under Different Page Sizes

Page Size	Height	Fan-out	Non-leaf 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%

Overall, as the page grows larger, most compression methods will improve performance due to shorter search paths, but those based on delta compression would decline later on. For them, a larger page also means a longer distance from the previous fully instantiated key. This, along with shorter search paths, makes these compression methods achieve optimal performance at some page size, rather than continually improving performance as the page size grows. 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 performance decreases at large page size due to the increased overhead of in-node decompression. DB2 suffers from the increasing time of prefix search.

Compared with in-memory scenarios, page sizes in disk scenarios are larger in existing DBMS (e.g., SQLite uses a 4KB page size [3], and Postgres uses an 8KB page size [6]). 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 [47], 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

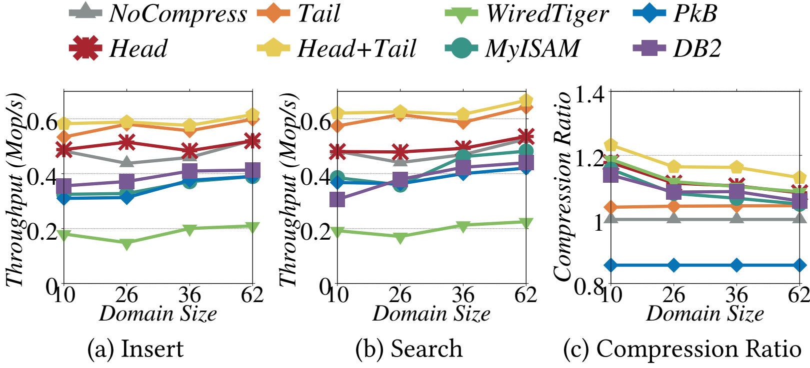


Fig. 16. Performance under Different Domain Sizes

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.

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 Sec. 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 actually changes by less than 0.2 Byte.

The situation is slightly different in terms of throughput. For most methods, there is a small overall improvement in search performance as the domain size increases. 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. For MyISAM, the decompression overhead is reduced a little which also accelerate the query process to a certain extent.

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}.

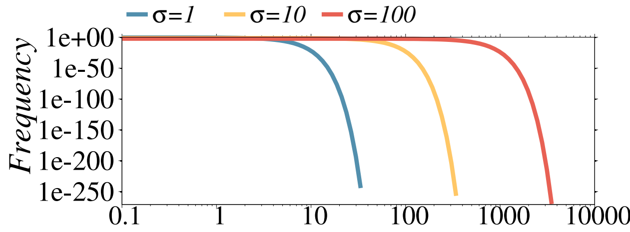


Fig. 17. Data Frequency under Different Scales

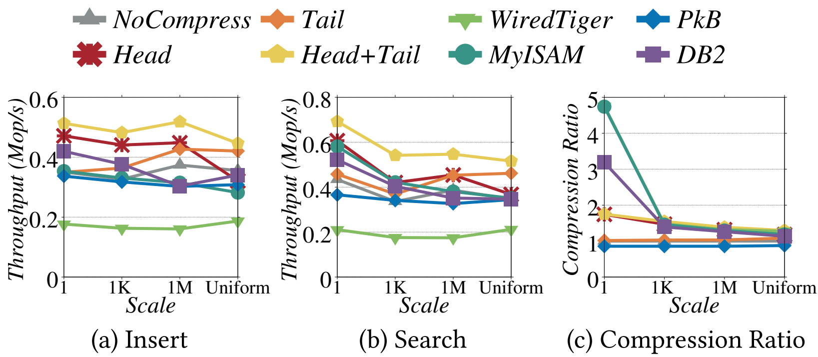


Fig. 18. Performance under Different Scales

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 reduced 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.

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

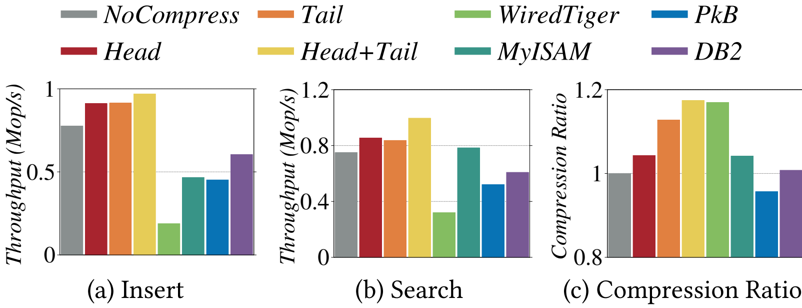


Fig. 19. Results over TPC-H LINEITEM (Memory)

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.

4.7 Evaluating Range Queries

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

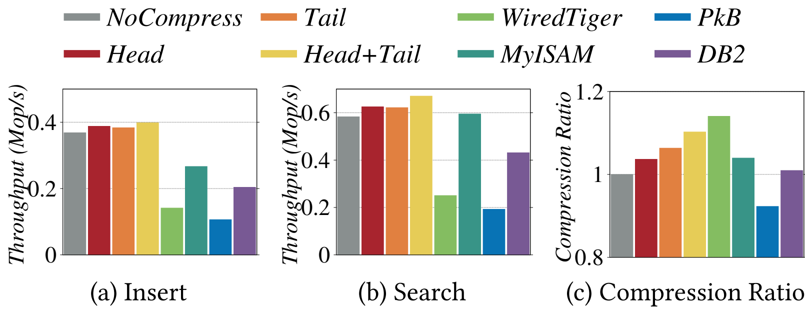


Fig. 20. Results over TPC-H LINEITEM (Disk)

5.1 Results on TPC-H

TPC-H [9] 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 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. We also ran the experiment on disk and the overall trend is consistent with the in-memory scenario, as shown in Figure 20.

5.2 Results on WEBSPAM and MemeTracker

WEBSPAM [1] is a collection of URLs of spam/non-spam hosts collected from the .uk domain. 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 21 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 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

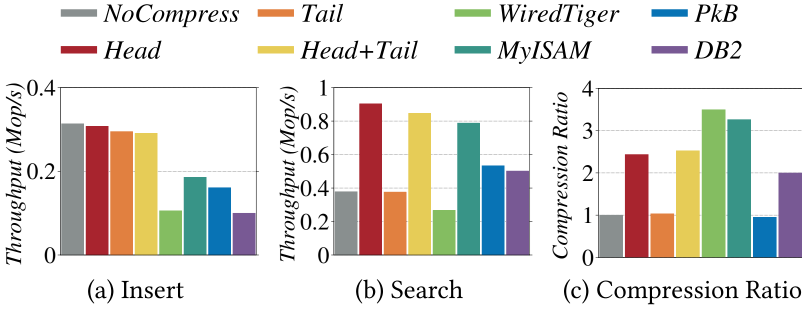


Fig. 21. Results over WEBSPAM-UK2007

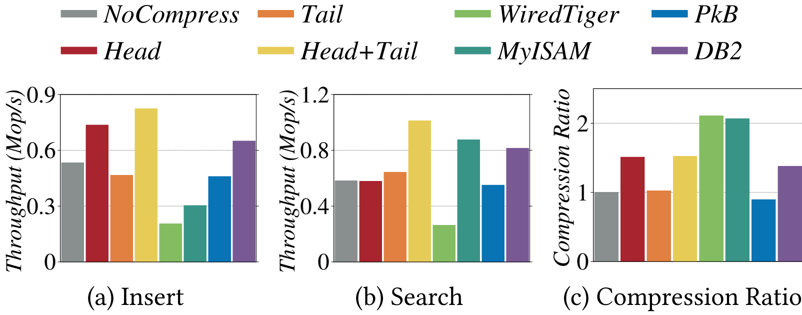


Fig. 22. Results over MemeTracker URLs

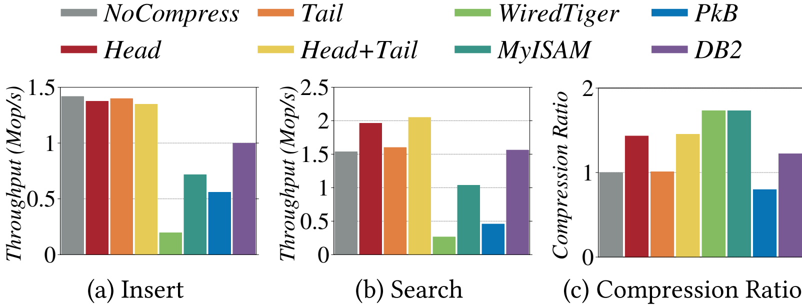


Fig. 23. Results over WikiTitles

DB2, same as Sec. 4.3, the long keys make the prefix search overhead a relatively large portion of the total query time. Therefore, the increasing share of prefix search in the total query time of DB2 leads 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 [2]. The results are shown in Figure 22. A main difference between the two datasets is the average length of keys. The growing number of in-node keys makes DB2 perform better and reduces the efficiency of Head Compression.

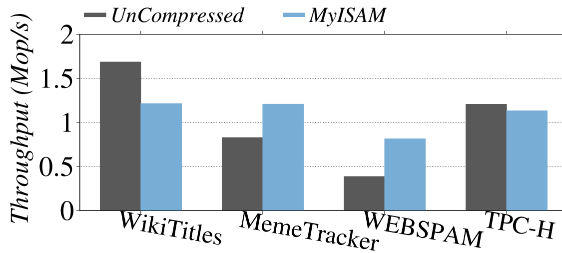


Fig. 24. Performance Comparison Between MyISAM and Uncompressed B-Tree on Real Datasets

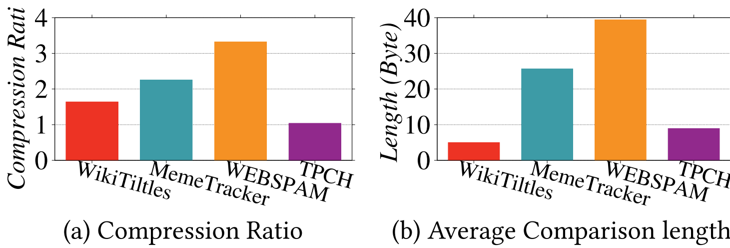


Fig. 25. Statistics on Compression Ratio and Keys' Feature

Table 6. Uncompressed B-tree Statistic for Different Datasets

Datasets	Key Length	Fan-out	Height	Leaf Numbers
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

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 23, 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 23, 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 24. We collect a series of valuable statistics among which the average actual comparison length, which is the average length that is sufficient to distinguish between two keys (i.e., the number of bytes compared before return the comparison result), has a large impact on the performance.

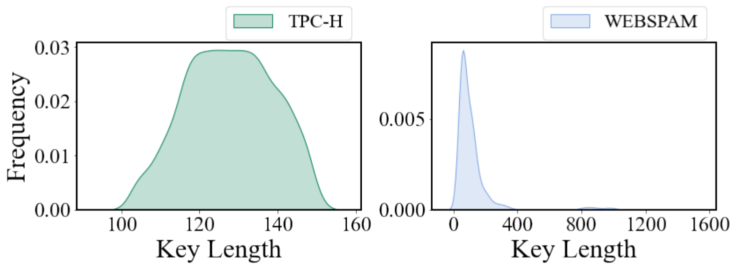


Fig. 26. Distribution of Key Length

Table 6 displays some supporting information about the uncompressed B-tree. 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 26. 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 25b, 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 RELATED WORK

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 [23], run-length encoding [11], integer compression, bitmap compression [34, 36, 44, 46], float compression [22, 39, 42], inverted index compression [43], and text compression [49, 50].

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 [26]. SAP's main-memory database HANA uses the CPB+-tree, a compressed prefix B+tree [5]. H-store, another main-memory database, also uses a B-tree [32]. There are also some other indexes that have been designed with space-saving purposes in mind. Bumbulis and Bowman developed a compact B-tree [20], 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 [37] save space by sharing path information. Zhang et al. developed an order-preserving key compression for in-memory search trees [51]. 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 [12, 13, 17, 40]. 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.

Another line of research is learned indexes [27, 35] 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 [44], no study has compared the performance of various B-tree compression techniques. This comparison is the main contribution of our work.

7 SUMMARY AND CONCLUSION

7.1 Summary

Search Performance. It is interesting to see that the Head+Tail Compression, although proposed in the 1970s [15], consistently exhibits the best search performance among all B-tree compression techniques. It achieves a **25%** ~ **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%** ~ **90.2%** faster than DB2 [16], **129%** ~ **684%** faster than MongoDB WiredTiger [10], **7.3%** ~ **139%** faster than MySQL MyISAM [7]. 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, Head+Tail Compression also shows the best performance compared to other B-tree compression techniques. It achieves a **10%** ~ **30%** performance improvement over the uncompressed B-tree.

Furthermore, only the Head, Tail, and Head+Tail Compression techniques outperform uncompressed B-tree in terms of 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** ~ **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.

7.2 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. As a result, we recommend modern database systems implement it.

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

7.3 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.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] 2007. WEBSPAM-UK2007 Dataset. https://chato.cl/webspam/datasets/uk2007/
[2] 2008. SNAP Memetracker Dataset. https://www.kaggle.com/datasets/snap/snap-memetracker
[3] 2022. The Default Page Size Change of SQLite 3.12.0. https://www.sqlite.org/pgszchng2016.html
[4] 2022. Source Code of WiredTiger's B-Tree Implementation. https://github.com/wiredtiger/wiredtiger/tree/develop/src/btree
[5] 2023. CREATE INDEX Statement in SAP HANA (https://help.sap.com/docs/SAP_HANA_PLATFORM/4fe29514fd584807ac9f2a04f6754767/20d44b4175191014a940afff4b47c7ea.html).
[6] 2023. Database Page Layout in PostgreSQL 16. https://www.postgresql.org/docs/current/storage-page-layout.html
[7] 2023. MyISAM Source Code in MySQL. https://github.com/mysql/mysql-server/tree/a246bad76b9271cb4333634e954040a970222e0a/storage/myisam
[8] 2023. MySQL Reference Manual. https://dev.mysql.com/doc/refman/8.0/en/key-space.html#:~:text=Prefix%20compression%20is%20used%20on,when%20you%20create%20the%20table.
[9] 2023. TPC-H Benchmark. https://www.tpc.org/tpch/
[10] 2023. WiredTiger Documentation. https://source.wiredtiger.com/11.1.0/file_formats.html#file_formats_compression
[11] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *SIGMOD*. 671–682.
[12] Gennady Antoshenkov. 1997. Dictionary-Based Order-Preserving String Compression. *VLDB Journal* 6, 1 (1997), 26–39.
[13] G. Antoshenkov, D. Lomet, and J. Murray. 1996. Order Preserving String Compression. In *ICDE*. 655–663.
[14] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indices. *Acta Informatica* 1 (1972), 173–189.
[15] Rudolf Bayer and Karl Unterauer. 1977. Prefix B-Trees. *TODS* 2, 1 (1977), 11–26.
[16] Bishwaranjan Bhattacharjee, Lipyeow Lim, Timothy Malkemus, George A. Mihaila, Kenneth A. Ross, Sherman Lau, Cathy McCarthur, Zoltan Toth, and Reza Sherkat. 2009. Efficient Index Compression in DB2 LUW. *PVLDB* 2, 2 (2009), 1462–1473.
[17] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based Order-preserving String Compression for Main Memory Column Stores. In *SIGMOD*. 283–296.
[18] Philip Bohannon, Peter McIlroy, and Rajeev Rastogi. 2001. Main-Memory Index Structures with Fixed-Size Partial Keys. In *SIGMOD*. 163–174.
[19] Lars Breddemann. 2020. What is CPB+-Tree in SAP HANA? https://www.lbreddemann.org/what-is-cpb-tree-in-sap-hana/
[20] Peter Bumbulis and Ivan T. Bowman. 2002. A Compact B-tree. In *SIGMOD*. 533–541.
[21] Igor Canadi. 2019. Converged Index: The Secret Sauce Behind Rockset's Fast Queries. https://rockset.com/blog/converged-indexing-the-secret-sauce-behind-rocksets-fast-queries/
[22] Xinyu Chen, Jiannan Tian, Ian Beaver, Cynthia Freeman, Yan Yan, Jianguo Wang, and Dingwen Tao. 2024. FCBench: Cross-Domain Benchmarking of Lossless Compression for Floating-Point Data. *PVLDB* 17, 6 (2024), 1418–1431.
[23] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. 2001. Query Optimization In Compressed Database Systems. In *SIGMOD*. 271–282.
[24] Gregg Christman. 2022. Compression Features Included with Oracle Database Enterprise Edition. https://blogs.oracle.com/dbstorage/post/compression-features-included-with-oracle-database-enterprise-edition
[25] Douglas Comer. 1979. The Ubiquitous B-Tree. *CSUR* 11, 2 (1979), 121–137.
[26] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*. 1243–1254.
[27] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned

Index. In *SIGMOD*. 969–984.
[28] Goetz Graefe. 2010. A Survey of B-tree Locking Techniques. *TODS* 35, 3 (2010), 16:1–16:26.
[29] Goetz Graefe. 2011. Modern B-Tree Techniques. *Foundations and Trends in Databases* 3, 4 (2011), 203–402.
[30] Goetz Graefe and Per-Åke Larson. 2001. B-Tree Indexes and CPU Caches. In *ICDE*. 349–358.
[31] Richard A. Hankins and Jignesh M. Patel. 2003. Effect of Node Size on the Performance of Cache-conscious B^+-trees. In *SIGMETRICS*. 283–294.
[32] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: A High-performance, Distributed Main memory Transaction Processing System. *PVLDB* 1, 2 (2008), 1496–1499.
[33] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *SIGMOD*. 339–350.
[34] Sangchul Kim, Junhee Lee, Srinivasa Rao Satti, and Bongki Moon. 2016. SBH: Super Byte-aligned Hybrid Bitmap Compression. *Information Systems* 62 (2016), 155–168.
[35] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. 489–504.
[36] Harald Lang, Alexander Beischl, Viktor Leis, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2020. Tree-Encoded Bitmaps. In *SIGMOD*. 937–967.
[37] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In *ICDE*. 38–49.
[38] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*. 302–313.
[39] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J. Elmore. 2021. Decomposed Bounded Floats for Fast Compression and Queries. *PVLDB* 14, 11 (2021), 2586–2598.
[40] Chunwei Liu, McKade Umbenhower, Hao Jiang, Pranav Subramaniam, Jihong Ma, and Aaron J. Elmore. 2019. Mostly Order Preserving Dictionaries. In *ICDE*. 1214–1225.
[41] David B. Lomet. 2001. The Evolution of Effective B-tree: Page Organization and Techniques: A Personal Account. *SIGMOD Record* 30, 3 (2001), 64–69.
[42] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *PVLDB* 8, 12 (2015), 1816–1827.
[43] Jianguo Wang, Chunbin Lin, Ruining He, Moojin Chae, Yannis Papakonstantinou, and Steven Swanson. 2017. MILC: Inverted List Compression in Memory. *PVLDB* 10, 8 (2017), 853–864.
[44] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An Experimental Study of Bitmap Compression vs. Inverted List Compression. In *SIGMOD*. 993–1008.
[45] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD*. 473–488.
[46] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. 2006. Optimizing Bitmap Indices with Efficient Compression. *TODS* 31, 1 (2006), 1–38.
[47] Helen Xu, Amanda Li, Brian Wheatman, Manoj Marneni, and Prashant Pandey. 2023. BP-tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-trees. *PVLDB* 16, 11 (2023), 2976–2989.
[48] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. 2019. AnalyticDB: Real-time OLAP Database System at Alibaba Cloud. *PVLDB* 12, 12 (2019), 2059–2070.
[49] Feng Zhang, Weitao Wan, Chenyang Zhang, Jidong Zhai, Yunpeng Chai, Haixiang Li, and Xiaoyong Du. 2022. CompressDB: Enabling Efficient Compressed Data Direct Processing for Various Databases. In *SIGMOD*. 1655–1669.
[50] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. 2021. TADOC: Text Analytics Directly on Compression. *VLDB Journal* 30, 2 (2021), 163–188.
[51] Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Order-Preserving Key Compression for In-Memory Search Trees. In *SIGMOD*. 1601–1615.
[52] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *SIGMOD*. 741–758.

Received October 2023; revised January 2024; accepted February 2024