Tree-Encoded Bitmaps

Harald Lang
Technical University of Munich
harald.lang@tum.de

Alexander Beischl
Technical University of Munich
beischl@tum.de

Viktor Leis
Friedrich Schiller University Jena
viktor.leis@uni-jena.de

Peter Boncz
Centrum Wiskunde & Informatica
boncz@cwi.nl

Thomas Neumann
Technical University of Munich
thomas.neumann@in.tum.de

Alfons Kemper
Technical University of Munich
alfons.kemper@in.tum.de

ABSTRACT

We propose a novel method to represent compressed bitmaps. Similarly to existing bitmap compression schemes, we exploit the compression potential of bitmaps populated with consecutive identical bits, i.e., 0-runs and 1-runs. But in contrast to prior work, our approach employs a binary tree structure to represent runs of various lengths. Leaf nodes in the upper tree levels thereby represent longer runs, and vice versa. The tree-based representation results in high compression ratios and enables efficient random access, which in turn allows for the fast intersection of bitmaps. Our experimental analysis with randomly generated bitmaps shows that our approach significantly improves over state-of-the-art compression techniques when bitmaps are dense and/or only barely clustered. Further, we evaluate our approach with real-world data sets, showing that our tree-encoded bitmaps can save up to one third of the space over existing techniques.

ACM Reference Format:
Harald Lang, Alexander Beischl, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2020. Tree-Encoded Bitmaps. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3318464.3380588

1 INTRODUCTION

Bitmap indexes have a long history in database systems and information retrieval [8, 11, 16, 37, 45, 53, 57]. They have many applications, such as efficiently evaluating predicates [42, 45, 46] and have been used to accelerate join [44] and aggregation [9, 46] queries. For medium or high cardinality columns, bitmap indexes consist of many individual bitmaps that are sparsely populated with 1-bits. Therefore, plain bitmaps consume large amounts of space, and compression is essential.

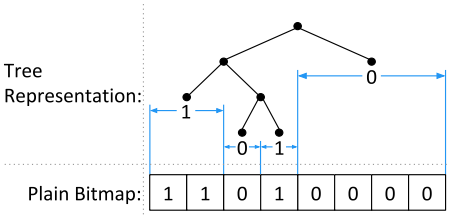


Figure 1: The key idea is to represent bitmaps as full binary trees. Longer runs are mapped to tree nodes closer to the root, and vice versa.

Consider the case of a bitmap index on an attribute A consisting of $|A|$ individual bitmaps of length n, where $|A|$ is the number of distinct values of A and n the number of tuples in the corresponding relation. The total number of 1-bits in the index is also n, whereas each bitmap receives $\frac{n}{|A|}$ 1-bits on average. A high number of distinct values, or the presence of skew, results in bitmap indexes with many sparsely populated bitmaps. Sparsity implies that these bitmaps mostly consist of consecutive 0-bits, i.e., *0-runs*. Having long runs of identical bits offers great compression potential, which all existing bitmap compression schemes try to exploit.

One simple, but fairly effective bitmap compression scheme is the Word-Aligned Hybrid [63] (WAH) approach, whose compression is based on run-length encoding (RLE). A WAH-compressed bitmap is a sequence of machine words, typically 32 or 64 bits in size. Each word either encodes a run or represents a small part of the original bitmap as is. The first is called a *fill word* and the latter a *literal word*. While WAH offers significantly better performance than its predecessor the Byte-Aligned Bitmap Compression [2] (BBC), its compression effectiveness suffers from two major weaknesses: (i) runs need to be rather long for the RLE-based compression to be effective and (ii) WAH has linear space overhead (one bit

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3380588

per word) for distinguishing between fill and literal words. In particular, the first weak point impairs compression when some random bits (also called *dirty bits* or *odd bits*) disrupt long runs. Over the years, several extensions to WAH have been proposed to solve this issue, i.e., PLWAH [18], Concise [15], VAL-WAH [22], EWAH [33], and SBH [27].

All the aforementioned compression techniques are based on RLE and therefore share another disadvantage, namely the linear time complexity of random access. Supporting efficient random access directly affects the efficiency of logical operations like bitwise AND, which are common operations in analytical queries.

Chambi et al. identified this problem and proposed the Roaring Bitmap format [10]. In contrast to the aforementioned compression techniques, Roaring Bitmap does not rely on RLE. Instead it partitions the input bitmap into equally sized chunks of length 2^{16} bits, where each chunk is physically stored in a separate container, as illustrated in Figure 2. Roaring implements three different container types and each container type represents the corresponding part of the bitmap differently. Depending on the number of bits set and on the presence of 1-runs, Roaring chooses the container type that consumes the smallest amount of memory. More precisely, if the number of 1-bits is less than or equal to 4096, an *array container* is used that stores a sorted list of 16-bit integers, one for each set bit. The integer values correspond to the positions of those bits within the current partition. If the number of set bits exceeds 4096, Roaring either employs a plain *bitmap container* or a *run container* [32]. A bitmap container stores the partition as is. A run container on the other hand stores the 1-runs as a list of 16-bit integer pairs $\langle a, b \rangle$, where $[a, b]$ is the range spanned by the 1-run.

Overall, Roaring is a very lightweight approach in terms of compression, as it only relies on integer arrays to represent bitmaps. Integer values are thereby truncated to 16 bits as every container encodes 2^{16} bits of the bitmap. Nevertheless, it results in significantly lower space consumption compared to RLE-based techniques in most scenarios. Due to the fact that the bit positions, the runs, and the containers themselves are sorted, a random access can be performed in logarithmic time, which significantly improves the performance of bitwise operation and thus of analytical queries [9].

It is worth mentioning that in principle Roaring is an extendable format, as it could employ any bitmap compression technique at the container level; including the tree-encoded bitmaps, we present in this work.

At the time of writing, Roaring was available in 11 programming languages and was widely used in Apache projects like Druid, Hive, Kylin, Lucence, Spark, and other systems[1]. This shows that today's applications not only demand high

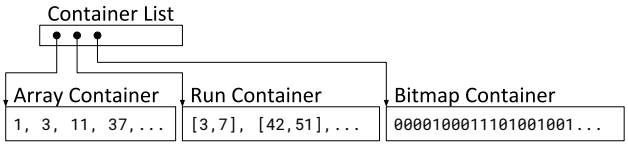


Figure 2: Roaring partitions the bitmap and stores each partition using the best suitable container type.

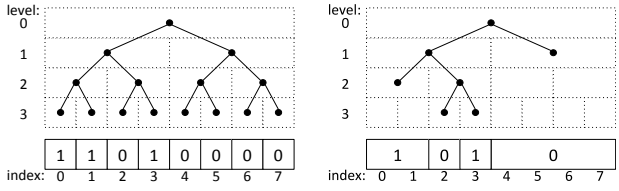
compression ratios but also efficient logical operations on compressed bitmaps. Further, we see a trend in database systems towards denser bitmaps—in particular, when bitmap indexes use histogram-based binning or are constructed to support range queries [11, 12]. In both cases, the resulting bitmaps exhibit higher bit densities compared to simple bitmap indexes as described at the beginning of this section[2].

With this work, we contribute a novel method to compress bitmaps. The compressed representation, which we call a *tree-encoded bitmap*, provides high compression ratios paired with logarithmic access time. Its primary strengths are the abilities (i) to compress both long and short runs and (ii) to significantly improve the compression ratios with denser bitmaps over existing techniques. The major conceptual difference compared to other compressed bitmap formats is that our approach employs a binary tree to represent bit runs of various lengths as illustrated in Figure 1. Tree nodes in the upper tree levels (closer to the root) thereby correspond to longer runs, and tree nodes in the lower levels to shorter runs. The low space requirement is achieved by using a succinct tree encoding and additional space optimizations that truncate balanced parts of the tree structure from the compressed representation. A key insight is that although our approach initially triples the size of a given bitmap to establish the tree structure, it does not only amortize this overhead, but also ultimately offers overall better compression ratios than RLE-based compression methods or the state-of-the-art Roaring Bitmap in a wide spectrum of moderately populated and clustered bitmaps. Using a collection of real-world data sets, we empirically found that tree-encoded bitmaps offer the best compression in 7 out of 8 cases, saving up to $^1/_3$ space in comparison with the second best solution.

Notation. Throughout the paper, we let n denote the length of a bitmap. Further, since compression heavily depends on the data distribution, we use the following two metrics to characterize individual bitmaps: (i) The *bit density* denoted as d refers to the fraction of bits set to 1, where $0 \leq d \leq 1$. The total number of set bits in a bitmap is therefore $d \cdot n$. (ii) The *clustering factor* denoted as f, with $1 \leq f \leq n$, indicates the degree of clustering of the 1-bits in a bitmap, i.e., how likely a 1-bit is followed by another 1-bit. Formally, it is defined as the average length of the 1-runs in a bitmap [63]. For

[1] We refer the reader to the official web site [31] for more details.

[2] In Section 5 we give a brief overview on the design space of bitmap indexes.



(a) initial state **(b) after pruning**

Figure 3: A bitmap represented as a binary tree. Initially, each leaf node is assigned a single bit (label). Sibling leaf nodes with identical labels are then pruned and the label is assigned to their parent. After pruning, the prior parent node becomes a leaf and represents multiple consecutive bits, a 0-run or a 1-run.

instance, the bitmap `01110010` (with $d = 0.5$) contains two 1-runs, one of length 3 and one of length 1. The clustering factor f therefore equals to 2. As both d and f refer to the set bits, they are dependent and the following restrictions apply: The clustering factor cannot exceed the total number of bits set ($f \leq d{\cdot}n$). Further, when the bit density exceeds 50%, the smallest possible value for f increases as well. E.g., given the bitmap `01010101` with $d = 0.5$ and $f = 1$; when the leftmost 0-bit is toggled (`11010101`), d increases to 0.625 and f to 1.25. In that particular case, 1.25 is the smallest possible clustering for a bitmap of length $n = 8$ and $d = 0.625$. In the general case, the smallest possible clustering is $\max(1, d/(1-d))$. Clustered bitmaps can be synthetically generated using a two-state Markov process, which we describe in the evaluation section.

2 TREE-ENCODED BITMAPS

In this section, we present our **T**ree-**E**ncoded **B**itmaps (TEB). The key idea behind TEB is to represent bitmaps as binary trees, which enables efficient navigation and therefore fast random access. The data structure is best explained by describing the construction algorithm. We therefore first present the tree-based compression algorithm. Later in this section, we describe how the tree is encoded space efficiently.

2.1 Compression

A TEB is constructed in two phases. In the first phase, a perfect binary tree is established on top of a given bitmap, as shown in Figure 3a. Each bit in the bitmap is associated with a single leaf node of the binary tree. Only leaf nodes carry a payload, which we refer to as *labels*. A label can either be a 0-bit or a 1-bit.

In the second construction phase, the binary tree is pruned bottom-up. Thereby, the algorithm removes all sibling leaf nodes with identical labels l, and the label l is assigned to the parent node. The pruning process stops when all pairs of sibling leaf nodes have different labels. Figure 3b depicts a fully pruned tree. The important thing to note here is that the newly created leaf nodes in the upper tree levels no longer represent individual bits of the bitmap; instead they represent consecutive bits that form either a 0-run or a 1-run. For instance, the leftmost node in Figure 3b represents a 1-run of length 2, starting at index 0 and the rightmost node represents a 0-run of length 4, starting at index 4.

With every single pruning step, two nodes are eliminated from the tree structure and one bit from the labels. Bottom-up pruning can therefore be considered a *lossless compression* method. Compressing the tree structure is a crucial part of TEB because the space overhead of the tree structure needs to be amortized. The tree initially consists of $2n - 1$ nodes, assuming n is a power of two. When the tree structure is encoded using one bit per node, then the space consumption of a TEB, including the labels, is initially, and in worst case, $3n - 1$ bits. Even though the worst case space consumption is relatively high, we will show that our tree-based representation of bitmaps often achieves significantly lower space usage than other compression schemes.

2.2 Encoding

An important part of TEB is the space-efficient way the tree structure is stored. We employ a *level-order binary marked* representation [24], which requires one bit per tree node. The encoded tree itself therefore is a sequence of bits (a bitmap).

We have to differentiate between the tree data structure that is used during compression and the *encoded* tree that is eventually stored in a TEB. For the tree-based compression, we temporarily make use of an implicit data structure [59] that allows for fast modifications, but occupies a constant amount of space – constant in the sense that its size does not change when nodes are removed. The level-order binary marked representation, on the other hand, is static but requires less space once the tree has been pruned. Thus, *encoding* is the process with which we transform the pruned tree into a more compact form.

To encode the pruned tree structure we traverse it in breadth-first left-to-right order (or level-order) and for each visited node a single bit is emitted, a 1-bit for inner nodes and a 0-bit for leaf nodes. These bits are appended to the bit sequence that represents the encoded tree, denoted as T. The labels of the leaf nodes are stored as a separate bit sequence to which we refer as L. When a leaf node is observed during traversal, its label bit is appended to L. For instance, the tree in Figure 3b is encoded as $T = \texttt{1100100},\ L = \texttt{0101}$.

To support efficient random access and bitwise operations, it is necessary to traverse the tree. Internally, the most important primitive operation is to determine the two child nodes of some given tree node, i.e., navigating downwards the tree. Within the encoded tree, each tree node is identified by its position in the bit sequence T. The sequence starts with the

root node at position 0. For any given tree node i, the child nodes can then be determined as follows [24]:

$$\text{left-child}(i) := \text{right-child}(i) - 1$$
$$\text{right-child}(i) := 2 \cdot \text{rank}(i)$$

where rank(i) refers to the number of 1-bits (inner nodes) in T within the range $[0, i]$.

Computing the rank of a node is a linear-time operation, and navigating from the root to any leaf node is therefore an $O(n \cdot \log n)$ operation. However, the rank operation can be turned into an $O(1)$ operation at the cost of additional space consumption [24]. TEB uses an implementation similar to the one used in [68], which pre-computes the rank on 512-bit block granularity and stores the values in an auxiliary integer array; which results in a 6.25% increased memory footprint. The rank is then computed as

$$\text{rank}(i) := R[\lfloor i/512 \rfloor] + \text{popcount}(T, \lfloor i/512 \rfloor \cdot 512, i)$$

where R refers to the array with the pre-computed values at block level and popcount counts the 1-bits in the last block up to index i.

Using an additional integer array populated with pre-computed ranks (a lookup table) is a common approach [20, 21, 43, 69] and changing the granularity of the lookup table offers a space/time trade-off. The more coarse-grained the lookup table is, the lower its space requirement and the higher the costs for counting the 1-bits within the last block; and vice versa. For TEB, we empirically determined that a granularity of 512 bits offers competitive performance at a reasonable space overhead. On a reasonably modern 64-bit hardware, a navigational operation in the tree therefore requires at most eight population count instructions (four on average) and one array lookup.

Besides the downward navigation, the rank of a tree node is further required to determine the node's label. If the node i is a leaf, then the position of the label within L is equal to the number of 1-bits in T preceding node i, which corresponds to the non-inclusive rank of i. However, because only leaf nodes have labels, we can use the inclusive[3] rank from above, because $T[\text{rank}(i)]$ is guaranteed to be a 0-bit. In summary, a label is accessed as follows:

$$\text{label}(i) := L[i - \text{rank}(i)]$$

Let us close by mentioning that the chosen encoding requires the tree structure to be a full binary tree, i.e., each node has either zero or two child nodes. It is easy to show that this holds for the tree structure of a TEB: Since the initial binary tree is perfect, and pruning always affects two sibling leaf nodes, the resulting tree structure remains full binary.

[3]We chose the inclusive rank as it results in fewer arithmetic instructions.

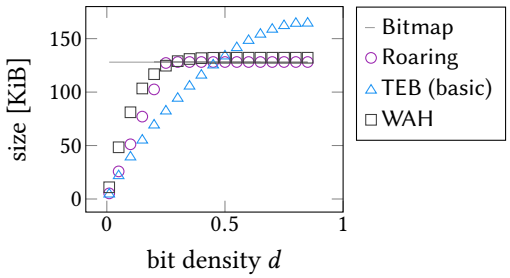


Figure 4: Size comparison for varying bit densities and a fixed clustering factor of 8.

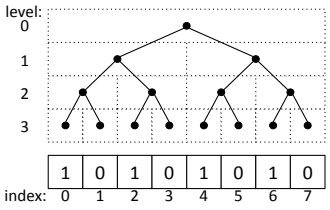


Figure 5: In worst case, the tree cannot be pruned (compressed) and the resulting TEB consumes approximately three times the space of the original bitmap.

2.3 Optimizations

The basic idea of TEB we have presented so far already shows promising results with regard to compression ratios. For instance, Figure 4 shows a space comparison of the TEB approach with two state-of-the-art bitmap compression techniques, Roaring and WAH. The compressed size (y-axis) depends on the ratio of 1-bits in the original bitmap (x-axis). Sparsely populated bitmaps offer higher compression potentials than densely populated bitmaps. In that particular case, if more than ~25% of the bitmap is populated with 1-bits, Roaring and WAH do not offer any compression at all. Both fall back to an uncompressed (literal) representation. TEB, on the other hand, is able to compress bitmaps with a bit density of up to ~45%.

The downside of the basic TEB approach is that in corner cases it can significantly exceed the size of the plain bitmap. In contrast to Roaring and WAH, our approach does not support an alternative representation to which it could fall back. In the following, we show that it is in fact not necessary to switch between different representations to address the high space consumption of TEB in unfavorable cases. It just requires a few minor modifications to the data structure and the compression algorithm, which we discuss in the following.

Implicit Tree Nodes. We motivate our first space optimization by considering the worst-case scenario for TEB. Figure 5 illustrates such a case. The depicted alternating bit sequence does not offer any compression potential. All pairs of sibling leaf nodes have different labels and therefore bottom-up

pruning cannot remove any tree nodes. The resulting TEB would consist of $n-1$ 1-bits for the inner nodes, followed by n 0-bits for the leaf nodes, and n label bits. In this extreme case, the label bits in L are identical to the uncompressed bitmap. Thus, storing the encoded tree structure is pure overhead.

Our first space optimization is to omit the leading 1-bits as well as the trailing 0-bits of the encoded tree structure. Only the intermediate bits of the tree structure are stored in the physical representation of a TEB. We refer to the omitted nodes as *implicit* tree nodes, and to the remaining as *explicit* tree nodes.

With regard to the worst case, this simple modification allows for the elimination of the entire tree encoding from the physical representation. Only the n label bits remain:

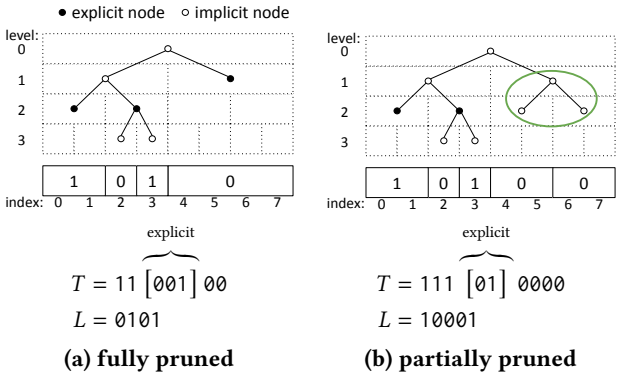
$$T = \underbrace{1111111}_{\text{leading 1-bits}}\,\underbrace{00000000}_{\text{trailing 0-bits}}, \quad L = 10101010$$

As mentioned before, the labels in L are identical to the original bitmap, i.e., the TEB degraded into an uncompressed bitmap. Thus, the size of the TEB is equal to the size of the plain bitmap, except for a small overhead that is caused by metadata.

However, further optimizations are needed, as this minor modification only mitigates the high space consumption of TEBs when the plain bitmap is poorly compressible. The TEB size may still significantly exceed the size of the uncompressed bitmap, i.e., the worst case has shifted. The modification, however, has two important implications:

(i) The encoded tree structure T is an optional part of the physical TEB data structure, as the entire tree may be implicit.
(ii) The space minimal TEB instance does not necessarily contain a fully pruned tree.

We give an example for (ii) in Figure 6a. The depicted TEB consists of three explicit tree nodes and four labels. Thus the space requirement is $3 \cdot 1.0625 + 4 = 7.1875$ bits, where the factor 1.0625 is to incorporate the space consumption of the rank helper structure (cf. Section 2.2). Figure 6b shows the TEB instance with the minimum size. The difference between the two TEB instances is that in Figure 6a the tree is fully pruned, whereas in 6b the two sibling leaves in the highlighted subtree have been preserved. The second instance therefore comprises a larger tree, but even though the total number of tree nodes and labels are higher, the second instance occupies less space ($2 \cdot 1.0625 + 5 = 7.125$ bits), as fewer tree nodes need to be stored explicitly. The circumstance that a fully pruned tree, in general, no longer corresponds to the smallest TEB instance requires a modification to the bottom-up pruning algorithm: Instead of returning the fully pruned tree, the algorithm needs to return the smallest tree instance observed during pruning, where the size is computed based on the number of explicit nodes, rather than the total number of nodes.



(a) fully pruned **(b) partially pruned**

Figure 6: Two different tree representations of the bitmap `11010000`. The fully pruned tree (a) occupies more space than the partially pruned tree (b), as more tree nodes need to be stored explicitly.

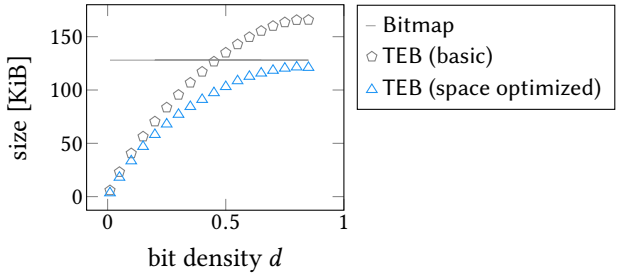


Figure 7: Size comparison of basic and space optimized TEBs using a clustering factor of 8.

Implicit Labels. Our second modification is to omit leading and trailing 0-labels in the physical TEB representation, similarly to implicit tree nodes. Omitting the leading 0-labels reduces the space consumption in particular with very sparse bitmaps. The tree representation of a sparse bitmap typically consists of a few leaf nodes with 1-labels at the deepest tree level $\log_2(n)$. But most of the leaf nodes with 0-labels can be found in the tree levels 1 to $\log_2(n) - 1$. Due to the tree being encoded in level order, the label bit sequence L tends to start with a long run of 0-labels, which we do not need to store explicitly. Trailing 0-labels on the other hand can occur when the length of the input bitmap is not a power of two. In that case, a TEB internally rounds up to the next power of two and fills the range $\left[n, 2^{\lceil \log_2(n) \rceil}\right)$ with 0-bits. Omitting these trailing 0-bits ensures that the number of stored labels never exceeds the length of the original bitmap.

The presented modifications reduce the overall space usage, as shown in Figure 7. In particular, the worst-case space consumption reduced from $3n-1$ to n bits, excluding the

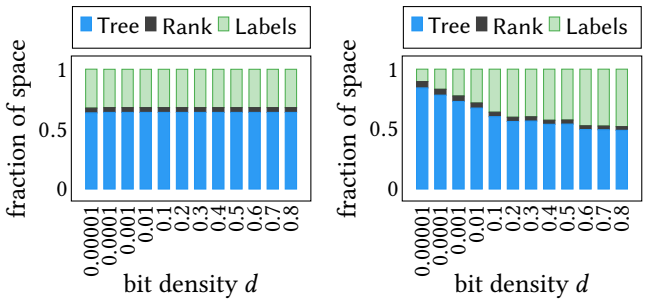
(small) metadata. We observe that in an optimized TEB the fraction of space occupied by the tree, the rank helper structure, and the labels is no longer fixed; compare Figures 8a and 8b. With sparse bitmaps, the labels occupy significantly less space. With denser bitmaps, on the other hand, we see that the fraction of space occupied by the tree structure decreases. Figure 9 shows how the implicit tree nodes and the implicit labels optimizations contribute to the space savings. The implicit labels optimization is most effective with sparse bitmaps and the implicit tree node optimization, on the other hand, favors denser bitmaps.

An important implication is that the space optimizations balance the upper part of the tree structure, as the example in Figure 6 has shown. The partially pruned tree in Figure 6b is perfectly balanced until level two, whereas the fully pruned tree in Figure 6a is only perfectly balanced until level one. Thus in general, the tree can be split into an upper balanced and a lower imbalanced part. This property allows for the reduction of the cost of navigational operations. We exploit the fact that within a perfect binary tree we can directly address the individual tree nodes, i.e., without computing ranks. If the number of the upper *perfect levels* is known, these levels of the tree can be logically cut off, and only the remaining sub-trees need to be considered. In our case, we can directly compute the number of perfect levels u based on the number of implicit inner nodes c that are already known when the space optimizations have been applied: $u := \lfloor \log_2(c+1) \rfloor + 1$. The corresponding node IDs for the last perfect level are within the range $[t_{\text{begin}}, t_{\text{end}})$, with $t_{\text{begin}} := 2^{u-1} - 1$ and $t_{\text{end}} := 2^u - 1$. Each of these nodes, or the sub-trees rooted at these nodes, respectively, span a range of length $2^{\log_2(n)-u-1}$ in the original bitmap. Thus, it can be considered as a uniform partitioning scheme, similar to the one used in Roaring Bitmaps, but with the major difference that the partition size is chosen adaptively.

The number of perfect tree levels is correlated with the effectiveness of the tree-based compression. The less effective the compression, the larger the number of perfect levels, and vice versa. In worst case, the entire tree is implicit and the number of perfect levels corresponds to the tree height. In other words, TEBs gradually degrade into literal bitmaps, but unlike Roaring and WAH, TEBs remain homogeneous and do not need to switch between different encodings or representations.

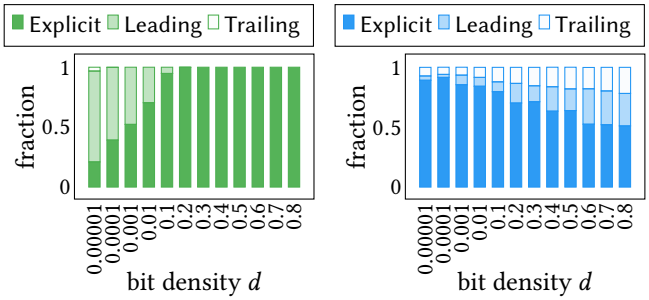
3 OPERATIONS

In this section, we describe the operations supported by TEB. Fundamentally, a TEB supports two access methods: (i) a point lookup and (ii) a 1-run iterator. High-level functionalities, like decompressing a bitmap or logical operations are implemented on top of the 1-run iterator.



(a) basic TEB (b) optimized TEB

Figure 8: The fraction of space occupied by the tree, the rank helper structure, and the labels.



(a) labels (b) tree nodes

Figure 9: The fraction of explicitly stored labels (a) and tree nodes (b).

Algorithm 1: Point lookup

Input : The bit index k to test
Returns: `true` if the k^{th} bit is set, `false` otherwise

```
// Determine the tree node at the last perfect level.
t_offset ← k >> (tree_height − perfect_levels − 1)
i ← t_begin + t_offset
j ← tree_height − 1 − perfect_levels − 1
// Navigate downwards until a leaf node is observed.
while i is an inner node do
    direction ← extract j^th bit from k
    i ← left-child(i) + direction
    j ← j − 1
end
return label(i)
```

3.1 Point Lookup

A point lookup is a straightforward operation that navigates downward the tree until a leaf node is reached. The index k of the bit to look up thereby specifies the path to take within the tree. For performance reasons, the downward navigation starts at the last perfect tree level rather than at the root node. The details are shown in Algorithm 1.

3.2 Run Iterator

The iterator interface allows for efficient iteration over a TEB. Unlike the iterators implemented in Roaring and WAH, the TEB iterator does not iterate over the individual 1-bits, instead it iterates over the 1-runs of a bitmap. A 1-run is thereby represented as two integer values $\langle begin, end \rangle$, pointing to the position of the first 1-bit and to the position one past the last 1-bit. The iterator traverses the tree in depth-first left-to-right order. To navigate down the tree, the functions left-child() and right-child() are used, as described in Section 2.2. To navigate upwards, the iterator makes use of a small stack that is populated during downward navigation. Other data structures like SuRF [68] implement upwards navigation using the *select* primitive, the counterpart to rank. For TEB, we prefer a classic stack-based approach as it is significantly faster in practice and saves space.

During tree traversal, the iterator needs to keep track of its position (and level) within the tree structure. This information is required to determine the start index and length of a 1-run when the iterator reaches a leaf node with label 1 and thus needs to produce an output. The iterator therefore maintains a *path* variable that encodes the path from the root to the current node using a single integer. The initial (and minimum) value of the path variable p is 1. During downwards navigation, a 0-bit is shifted in when navigating to the left child $p := (p << 1)$ and a 1-bit when navigating to the right child $p := (p << 1) \mid 1$. The index of the most significant 1-bit (the *sentinel bit*) indicates the level of the corresponding tree node:

$$\text{level}(p) := \text{sizeof}(p) \cdot 8 - 1 - \text{lzcount}(p)$$

where sizeof(p) refers to the size of the variable p in bytes and lzcount(p) to the number of leading zeros in p. A tree node that is identified by its path p then represents a run that starts at position

$$\text{pos}(p) := (p \oplus (1 << \text{level}(p))) << (tree\_height - \text{level}(p))$$

with length

$$\text{length}(p) := n >> \text{level}(p) \;\hat{=}\; 2^{\log_2(n) - \text{level}(p)}.$$

Similarly to the point lookup access method, the upper perfect levels of the tree are skipped. The iterator only considers the sub-trees rooted in $[t_{\text{begin}}, t_{\text{end}})$, as described in Section 2.3. Algorithm 2 shows how the iterator is forwarded to the next 1-run.

As mentioned earlier, a time-critical operation is to *fast-forward* the iterator to a desired position, thereby skipping all set bits in between. Thanks to the navigable tree structure, the operation can be performed in logarithmic time. Nevertheless, to achieve competitive performance in practice, we optimize the skip operation so that unnecessary navigation steps are avoided. The primary decision that is to be made is whether to (i) navigate up the tree to the common ancestor of the current and the destination node, and then downwards in the right sub-tree to the desired position, or (ii) start at the root node (or at the corresponding tree node in the last perfect level) and navigate only downwards until the desired position has been reached. Depending on the source and destination nodes, one option might be more efficient than the other. The two options may differ in the number of required navigation steps. But we also need to consider that navigating upwards is less costly in terms of issued CPU instructions than navigating downwards. The asymmetrical costs are mostly caused by the rank primitive, which is significantly more costly than accessing the stack. We experimentally determined that a downward step is approximately 9× more expensive than an upward step (~55 cycles vs. ~6 cycles).

Our decision logic works as follows: We start with a fast test to determine whether the destination position is outside of the current sub-tree:

$$pos >> (h - u - 1) \;!=\; to\_pos >> (h - u - 1)$$

where h refers to the tree height and u to the number of perfect levels. If the expression evaluates to true, we can directly go to the corresponding node at the last perfect level and navigate downwards until the desired position has been reached. Otherwise, if the destination node is within

Algorithm 2: Forward the iterator to the next 1-run.

```
while t < t_end do
    while stack is not empty do
        // Pop tree node i and its path p from the stack.
        ⟨i, p⟩ ← stack.pop()
        while i is an inner node do
            // Push right child on stack and go to left child.
            i ← left-child(i)
            p ← p << 1
            stack.push( ⟨i + 1, p | 1⟩ )
        end
        // Reached a leaf node.
        if label(i) = 0 then continue
        // Found a 1-run. Update the iterator state.
        level ← sizeof(p) · 8 − 1 − lzcount(p)
        begin ← (p ⊕ (1 << level)) << (tree_height − level)
        end ← begin + (n >> level)
        return
    end
    t ← t + 1
    p ← (t − t_begin) | (1 << (perfect_levels − 1))
    stack.push( ⟨t, p⟩ )
end
begin ← end ← n   // Reached the end.
return
```

the current sub-tree, we (i) determine the common ancestor node (ii), estimate the navigational costs for both options, and (iii) pick the cheaper path.

It is worth mentioning that an iterator with skip support is not the most efficient way to decompress (rather than intersect) a TEB. For these cases we provide an alternative iterator to which we refer to as *scan iterator*. Unlike the regular iterator, the scan iterator's seek function operates in $O(n)$, but it offers a significantly higher read throughput, as it (i) decodes the tree in batches and (ii) does not rely on the rank primitive to traverse the tree.

3.3 Tree Scan

In this section, we present a tree traversal algorithm that is optimized for modern x86 hardware. The algorithm takes a level-order encoded binary tree and iterates over all leaf nodes in left-to-right order. We refer to the algorithm as *tree scan*. The tree scan is the basic building block for the TEB scan iterator.

Generally speaking, navigating from one leaf node to the next one is a 3-step process: (i) navigate up the tree until a left child is observed, (ii) go to its right sibling, and (iii) walk down the tree to the leftmost leaf node. The key idea behind our solution is to have multiple lightweight bit iterators for the encoded tree structure T, one iterator per tree level, and then scan the bit sequence T in parallel. We denote the bit iterators in T as t_l with $0 \leq l < h$. Initially, all iterators point to the first bit in T at their corresponding level l. We expose the values each iterator points to as an integer value, denoted as α. The bits in $\alpha := b_{h-1} \dots b_1 b_0$ are populated with $b_l = *t_l$, where $*$ denotes the dereference operator. A path variable p identifies the position and the level within the tree, as described earlier. Initially, p points to the leftmost leaf node. Using the two values α and p, we can efficiently iterate over all leaf nodes in left-to-right order, cf., Algorithm 3. Thereby, p determines the number of upward steps and α determines the number of downward steps to perform in each iteration.

The bit iterators are implemented using the AVX-512 SIMD instruction set as follows. We use a 512-bit SIMD register to buffer the tree structure. The register is interpreted as 32×16-bit integers, i.e., the register is split into 32 *lanes*. Thereby, each SIMD lane corresponds to a tree level. For each level we load up to 16 bits from the encoded tree T. For instance, Figure 10 illustrates a buffer that contains the tree from Figure 6b. To consume the buffered tree bit by bit, we use a second SIMD register to which we refer as *read mask*. The read mask again consists of 32 lanes, and a single bit is set within each lane. Initially, the least significant bit is set to 1. The position of that bit represents the current read position in the corresponding buffer lane. Thus, the read mask represents the state of all (up to) 32 lightweight bit iterators. The increment of an iterator is then implemented as a left shift of the corresponding lane. The implementation has the advantages that we can work with multiple iterators in parallel and that the most important operations can be performed in a single instruction. For instance, multiple iterators can be incremented using a single masked shift instruction (`_mm512_mask_slli_epi16`[4]) and all iterators can be dereferenced in parallel to retrieve the aforementioned α value; cf., Figure 10.

Algorithm 3: Tree scan

```
p // The current path. Initially points to the leftmost leaf.
do
    // Produce an output, if the label of the current node is 1.
    ...
    // Walk upwards until a left child is found.
    up_steps ← tzcount(~p)
    last ← level(p) + 1
    p ← p >> up_steps
    p ← p | 1    // Go to the right sibling.
    first ← level(p)
    increment the iterators t_first to t_last and update α
    // Walk downwards to the leftmost leaf in that sub-tree.
    down_steps ← tzcount(~(α >> level(p)))
    p ← p << down_steps
while not done
```

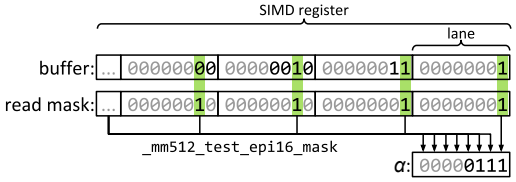


Figure 10: AVX-512 allows for the instantiation of up to 32 lightweight bit iterators (one for each tree level) using only two SIMD registers: The first is used to buffer the encoded tree level by level and the second represents the iterators' read positions.

The presented algorithm is used in the TEB scan iterator that is supposed to be used when efficient skip support is not required, e.g., when decompressing an entire TEB. With regard to performance, the scan iterator benefits from the predictable memory access pattern, as well as from the reduced number of memory loads, due to buffering. However, a problem not mentioned above is that we need to know the start offset in T for each tree level. Unfortunately, determining these offsets is a linear time operation. Therefore, we store the offsets as part of the TEB metadata, which now is logarithmic in size. For brevity we have omitted some of the

[4]We refer the reader to the Intel Intrinsics Guide for more details on the SIMD instruction set architectures: https://software.intel.com/sites/landingpage/IntrinsicsGuide

Algorithm 4: Next function of the AND iterator.

```
Input: Run iterators a and b.
while !(a.begin != n || b.begin != n) do
    begin_max ← max(a.begin, b.begin)
    end_min ← min(a.end, b.end)
    overlap ← begin_max < end_min
    if overlap then
        if a.end ≤ b.end then a.next()
        if b.end ≤ a.end then b.next()
        begin ← begin_max   // Update the iterator state.
        end ← end_min
        return
    else
        if a.end ≤ b.end then a.skip_to(b.begin)
        else b.skip_to(a.begin)
    end
end
begin ← end ← n   // Reached the end.
```

implementation details, such as how buffers are refilled and how labels are buffered and accessed; which works similarly to the buffering of the tree structure. We invite the interested reader to examine the source code of TEB[5].

3.4 Logical Operations

As mentioned earlier, high-level functionality is implemented on top of the 1-run iterator. Operations like a bitwise AND are themselves implemented as iterators and can therefore be arbitrarily chained and combined to evaluate complex expressions. Algorithm 4, for instance, shows how two bitmaps are intersected using the iterator API. In contrast to the implementations in Roaring and WAH, the iterator approach does not produce a compressed bitmap. We think this is not a disadvantage because producing compressed intermediate results when evaluating complex compressions could harm performance. For instance, when bitmap indexes are used to evaluate multi-dimensional selection predicates, it is sufficient to identify the ranges (or pages) that contain qualifying tuples; an intermediate bitmap would be discarded afterwards anyhow.

3.5 Updates

Data structure design in general is a trade-off between read, update, and memory overheads. The RUM conjecture [3] states that when optimizing (reducing) two of these overheads, it impairs the third one. TEBs are optimized for efficient read access and low memory consumption, and similarly to existing RLE-based compression schemes, the static nature of TEBs does not allow for in-place updates. In the following, we discuss various approaches that can be combined with TEBs to achieve updatability.

[5]TEB source code: https://db.in.tum.de/research/publications/#teb

	WAH	EWAH	Concise	Roaring	**TEB**
Census Income	3.4	3.3	2.9	2.6	**2.1**
Census Income (sorted)	0.66	0.64	0.55	0.6	**0.36**
Census 1881	34.4	33.8	25.6	15.1	**12.6**
Census 1881 (sorted)	3.0	2.9	2.5	2.1	**1.5**
Weather	6.8	6.7	5.9	5.4	**4.2**
Weather (sorted)	0.55	0.54	0.43	0.34	**0.26**
WikiLeaks	11.1	10.9	10.2	5.9	**5.4**
WikiLeaks (sorted)	2.9	2.7	2.2	**1.7**	1.7

Table 1: Space usage in bits per attribute value.

The naïve and costly way to support random updates is to decompress the bitmap, perform the update on the uncompressed representation, and (re-)compress it again afterwards. Prior work [4] proposed to reduce the update costs by staging updates in an auxiliary differential data structure and to apply these pending updates in batches, rather than one-by-one. Thereby, another compressed bitmap is used as a differential data structure. While this approach greatly reduces the number of decompression/compression cycles, it also causes redundancies (slightly higher memory consumption) and requires the differential data structure to be consulted (XORed) during read access.

Roaring bitmap applies a different strategy. Due to the fixed size partitioning, an update affects only a single container, rather than the entire bitmap. Thus, in worst-case, 2^{16} bits need to be re-compressed during updates. Updates can therefore be performed in constant time[6], even though the constant is quite large. Nevertheless, the partition size has been chosen sufficiently small to fit in an L1 cache to enable efficient decompression/compression cycles.

Both approaches can be used with TEBs. Partitioning could further be combined with differential updates so that a separate diff is maintained per partition. We will show in the later evaluation section that the combined approach offers the highest throughput regarding updates, with minor compromises regarding reads.

4 EXPERIMENTAL ANALYSIS

In the following, we evaluate our approach with regard to its compression ratio and performance. We begin by using a number of real-world data sets before performing a detailed evaluation using synthetic data.

4.1 Real-World Data

We evaluate TEBs with bitmaps from bitmap indexes constructed from four real-world data sets that have been previously used in the experimental evaluation of Roaring Bitmaps [32]. The data sets, namely Census Income, Census 1881,

[6]Assuming the corresponding containers reside in heap memory. Modifications to the serialized format would still be in linear time.

Weather, and WikiLeaks, come in two flavors: *as is* and *sorted*. The latter relies on a-priori sorting of the raw input data, which leads to significantly better compression ratios [33, 34, 47]. Following the prior work, we compress the individual bitmaps, 200 per data set, and report the average number of bits per attribute value. We compare TEB with Concise, EWAH, Roaring, and WAH. The results for Concise and EWAH are taken from [32]. We reproduced the results for Roaring with very minor differences with the sorted Census 1881 and WikiLeaks data. But we observed a higher discrepancy for WAH. Among our experiments, we observed a slightly higher space usage than reported earlier, except for Census 1881 where we observed a significantly better compression ratio (34.4 vs. 43.8 bits per element). We attribute these discrepancies to the fact that we use a different implementation [60] of WAH. Please note that EWAH and WAH use 32-bit words; we omit the results for the 64-bit implementations, as those have a higher space consumption among all tested workloads.

Table 1 summarizes the experimental results. TEB offers the best compression ratios, except for the sorted WikiLeaks data, where Roaring is slightly better (1.667 vs. 1.677 bits per element). TEB saves up to 22% space on unsorted data and up to 34.6% on sorted data compared to the second best compression technique, which in most cases is Roaring.

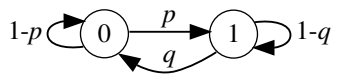
The rank lookup table (LuT) thereby accounts for 2.2% to 4.4% of the TEB size (3.7% geo. mean, among all real-world data sets). As mentioned earlier, changing the resolution of the LuT offers a space/time trade-off. A fine-grained LuT with one entry per 64 bit offers the best performance. We observe a 30% lower execution time for computing bitmap intersections. The memory overhead of the LuT thereby increases significantly to up to 27%, which almost cancels out the improvements in compression. Decreasing the LuT resolution to 2048 bits on the other hand reduces the TEB size by up to 2.8% but also causes the intersection time to increase by up to 10%. Table 2 shows how the space consumption of TEBs changes for varying rank resolutions compared to Roaring. Throughout our experiments, we found that a 512-bit resolution offers a reasonable space/time trade-off, which we use as our default setting in the following. Nevertheless, it is noteworthy that the rank LuT could be omitted when TEBs are written to persistent storage, and could be recomputed on-the-fly when TEBs are loaded back into main memory, allowing one to save additional disk space and I/O (cf., rightmost column in Table 2).

	Rank LuT resolution [bits]					no
	64	128	256	512	2048	LuT
Census 1881	*1.10*	0.95	0.87	0.83	0.81	0.80
Census 1881 (sorted)	0.87	0.76	0.71	0.69	0.67	0.66
Census Inc.	0.93	0.86	0.82	0.81	0.79	0.79
Census Inc. (sorted)	0.76	0.66	0.62	0.60	0.58	0.58
Weather	0.93	0.84	0.80	0.77	0.76	0.75
Weather (sorted)	0.97	0.84	0.79	0.76	0.74	0.73
WikiLeaks	*1.18*	*1.02*	0.95	0.91	0.89	0.88
WikiLeaks (sorted)	*1.25*	*1.11*	*1.04*	*1.01*	0.98	0.98

Table 2: Relative size of TEB compared to Roaring ($^{\texttt{TEB size}}/_{\texttt{Roaring size}}$) for varying rank resolutions.

4.2 Synthetic Data

For an in-depth analysis we generate random bitmaps, where the individual 1-bits are either uniformly distributed or clustered. *Uniform* random bitmaps are random bitmaps where each bit is independently generated following an identical probability distribution [63], i.e, each bit is set with probability d. *Clustered* random bitmaps on the other hand are generated using a two-state Markov process



with the transition probabilities p and q set to

$$p := \frac{d}{(1-d)\cdot f}, \text{ and } q := \frac{1}{f}$$

with $0 < d < 1$ and $1 \leq f \leq n$. We make a minor change over the definition given in [63]; which is that we choose the initial state randomly with a probability of 0.5, whereas in [63] the initial state is ①, meaning that a randomly generated bitmap would always start with a 1-run.

We generate bitmaps of length $n = 2^{20}$ and report the averaged results over 10 independent experiments. We compare TEB with WAH [63], which is the most popular RLE-based bitmap compression scheme, and with Roaring Bitmap [32], which is the state-of-the-art with regard to performance and compression ratio. The thorough study of Wang et al. [57] found Roaring to be superior over other bitmap compression techniques such as Concise [15], WAH, EWAH [33], VAL-WAH [22], PLWAH [18], and SBH [27]. We therefore limit our evaluation to Roaring and WAH.

For the experiments we use FastBit [60, 61] v2.0.3, which provides a C++ implementation of WAH, and CRoaring [5] v0.2.60 (unity build), the official C/C++ implementation of Roaring Bitmap. The `dynamic_bitset` from the Boost C++ libraries [7] v1.67.0 is used for uncompressed bitmaps. We compile with GCC v8.3.0 (`-O3 -march=native`) and execute on an Intel Core i9-7900X CPU @ 4 GHz.

4.2.1 Compression.

Uniform Bitmaps. In the following, we examine the compression ratios with uniform random bitmaps with varying bit densities. The results in Figure 11 show that TEB and Roaring are on par in the case of sparse bitmaps ($d < 0.005$). With an increasing d, TEB shows the lowest space usage.

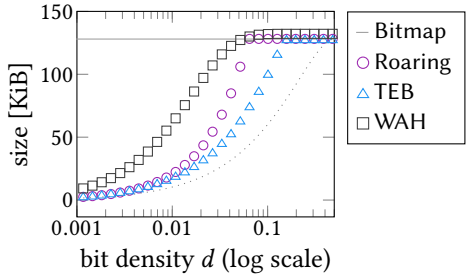


Figure 11: Size of uniform random bitmaps with varying bit densities. The dotted line refers to the information theoretic minimum.

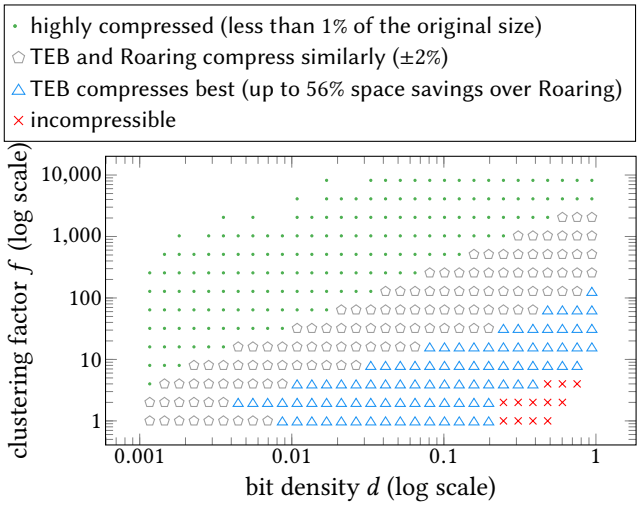


Figure 12: Summary of our findings when compressing clustered bitmaps.

When more than 13% of the bitmap is populated, TEB is no longer able to compress; Roaring and WAH already stop at 5%. With dense bitmaps ($0.5 < d \leq 1$) we observed symmetrical results for TEB and WAH, only Roaring requires a density of more than 97% for the compression to work again (rather than 95%). This is attributed to the different containers being used in Roaring, and the fact that Roaring encodes 0-runs and 1-runs differently, which is in contrast to TEB and WAH.

Clustered Bitmaps. With our third experiment, we examine the compression ratios with clustered bitmaps, using varying bit densities d and clustering factors f. We start with an exploration of the space spanned by d and f. Thereby, we consider the ranges $0.0001 \leq d < 1$ and $1 \leq f \leq n$. We make the following observations:

- When the input bitmaps are very sparsely populated or exhibit a strong clustering, all bitmap compression techniques under test perform well. In the dotted area (·) in Figure 12, the compressed bitmaps occupy less than 1% of the space of the uncompressed bitmap, irrespective from the employed compression scheme.
- TEB offers better compression ratios than WAH throughout all measurements; and only in some rare cases does WAH compress slightly better than Roaring.
- When comparing TEB and Roaring, TEB does not always offer the best compression ratios. However, in these cases, the differences in size are marginal. The largest difference in size we observed throughout all experiments is 1.6% of the original bitmap size. In the area marked with ⬠ in Figure 12, TEB and Roaring perform similarly.
- TEB in contrast, shows significantly higher compression ratios with denser bitmaps and bitmaps with lower clustering, cf. the area marked with △ in Figure 12. In comparison to Roaring, we observed a difference in size of up to 56% of the plain bitmap size, in favor of TEB. Figure 13 shows a qualitative side-by-side comparison.

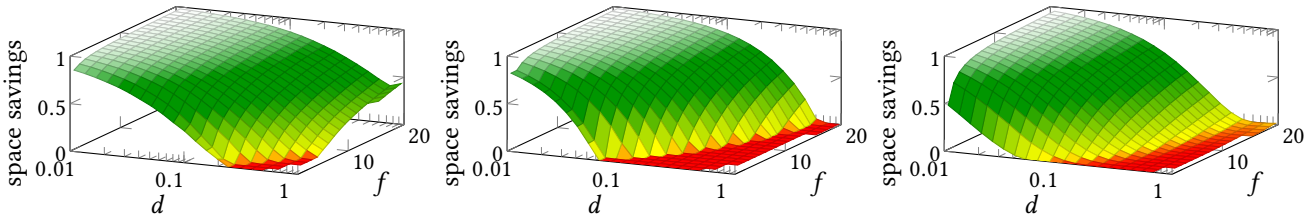
Figure 14 gives a detailed view on how the size of the compressed bitmaps change for varying d and fixed f. Figure 14a shows that the TEB approach is able to exploit short 1-runs in sparse bitmaps, resulting in up to ~50% space savings over Roaring. With a moderate clustering, as shown in Figure 14b, our approach is also able to compress dense bitmaps. Figure 14c, on the other hand, reveals that our approach has a slightly higher space usage than Roaring with strongly clustered bitmaps, which implies that Roaring can encode longer runs more space efficiently.

Figure 15 illustrates how f affects the compression ratios. Figures 15a and 15b show that already a slight clustering can lead to significant space savings with TEB. Roaring requires a significantly higher clustering to be competitive. With sparser bitmaps, TEB falls slightly behind Roaring (see Figure 15c), whereas WAH cannot compete.

4.2.2 *Performance.*

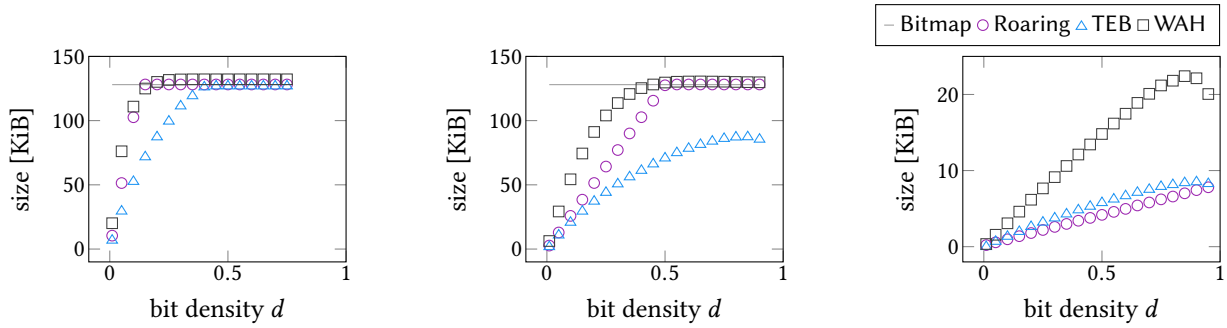
In the following, we evaluate the read and update performance of TEB, and show how it compares to Roaring and WAH.

Read Access. We first investigate the read (or decompression) throughput. We thereby iterate over all 1-runs of a bitmap and measure the duration in wall-clock time. In our initial performance experiment, we again explore the space spanned by d and f. Thereby we observe that an uncompressed bitmap performs better than the compressed formats when $16 \leq f \leq 128$ and $0.01 \leq d < 1$. It should be noted that the `dynamic_bitset` implementation, which we use for uncompressed bitmaps, is very straightforward and does not include any hardware specific optimizations. Thus, we expect a performance-optimized implementation to dominate an even larger space. When we consider only the performance of compressed bitmaps, we observe that the



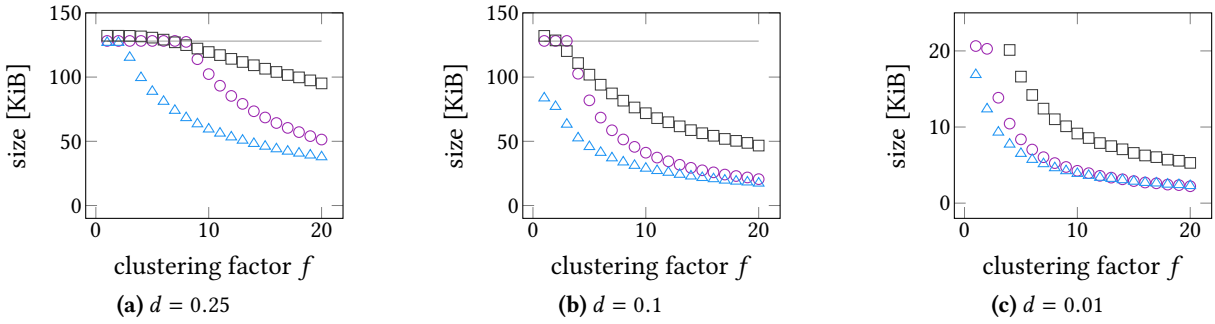
(a) TEB **(b) Roaring** **(c) WAH**

Figure 13: Space savings $\left(1 - \frac{\text{compressed size}}{\text{uncompressed size}}\right)$ for varying d and f.



(a) $f = 4$ **(b)** $f = 16$ **(c)** $f = 512$

Figure 14: Compressed bitmap size for varying bit densities and fixed clustering factors.



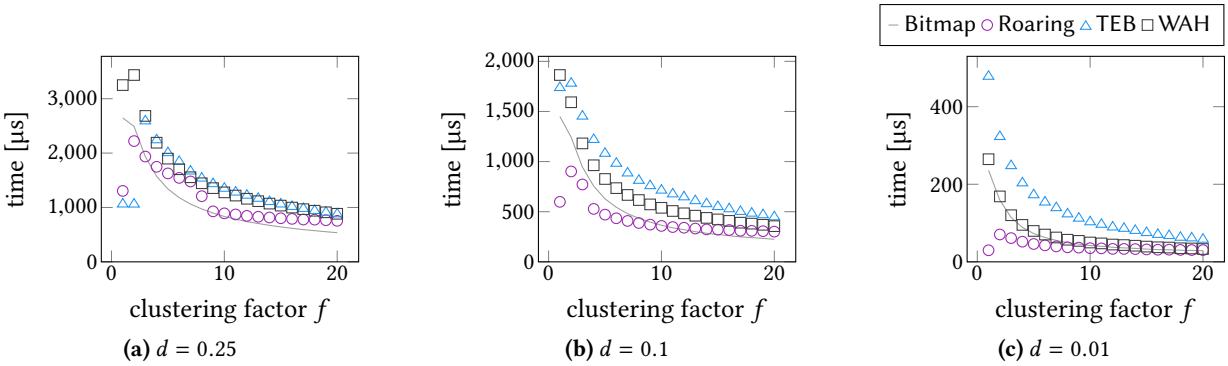
(a) $d = 0.25$ **(b)** $d = 0.1$ **(c)** $d = 0.01$

Figure 15: Compressed bitmap size for varying clustering factors and fixed bit densities.

clustering mostly determines the best performing compression technique: Roaring is dominant when $f \leq 16$, followed by WAH until f is approximately 128. TEB requires an evenly higher clustering ($f > 128$) to outperform Roaring and WAH.

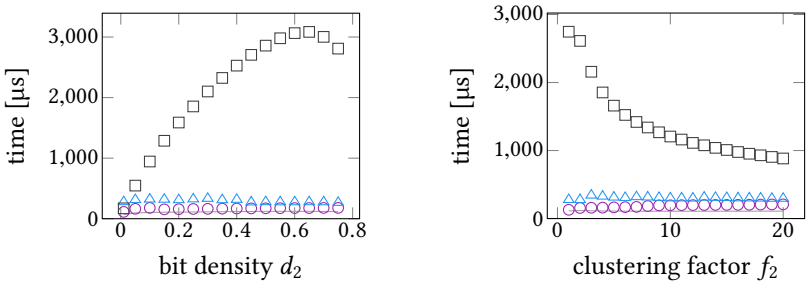
In Figure 16, we compare the performance for reasonable values of d and f, which we expect to occur in practice. We fixed d to $\{0.25, 0.1, 0.01\}$ and varied f within the range $[1, 20]$. We observe that the time to read the bitmap decreases with an increasing f, which is due to the smaller size of the input and due to less branching; the higher f is, the lower the number of 1-runs to iterate over. Figure 16a, with d set to 0.25, shows that TEB offers a similar performance as WAH, and that both are close to the performance of Roaring. Still, a plain bitmap performs best in most cases. The outliers at $f = 1$ and $f = 2$ are due to specialized code paths that are taken when the bitmaps are not compressed (or just barely compressed). In the Figures 16b and 16c, with bit densities reduced to 0.1 and 0.01, we observe that the absolute time to read a bitmap decreases for all implementations under test (note the different y-axis scales), but also that TEB falls behind relative to Roaring and WAH, indicating that the average cost per 1-run increases with lower d. Naturally, this is an expected result, as lower bit densities result in sparse and imbalanced trees, which in turn increases the number of tree levels that need to be traversed (cf., Section 2).

In our second experiment, we evaluate the effectiveness of efficient tree navigations within logical operations. We intersect (bitwise AND) two bitmaps with different characteristics.



(a) $d = 0.25$ **(b)** $d = 0.1$ **(c)** $d = 0.01$

Figure 16: Read performance for varying clustering factors and fixed bit densities.



(a) $d_1 = 0.01, f_1 = 8, f_2 = 4$ **(b)** $d_1 = 0.01, f_1 = 8, d_2 = 0.25$

Figure 17: Intersection performance.

The density and the clustering in the first bitmap is thereby fixed to $d_1 = 0.01$ and $f_1 = 8$. In Figure 17a, we fix the clustering in the second bitmap to $f_2 = 4$ and vary the density d_2. We observe that the density of the second bitmap only has a minor impact on the overall intersection time, except for WAH. The intersection of uncompressed bitmaps, with constant time random access, is fastest in this setting. Roaring takes ~1.5× the time of the plain bitmap intersection, and TEB ~1.9× the time of Roaring. In Figure 17b, we fix the density of the second bitmap to $d_2 = 0.25$ and vary f_2. Again, only WAH is sensitive to the varying clustering factor and thus to the size of the second bitmap. On average, Roaring needs ~1.8× the time of the plain bitmap intersection, and TEB ~1.6× the time of Roaring.

Differential Updates. In our final experiments, we extend TEB and the other bitmap compression techniques under test by a differential data structure and evaluate the update performance. Our experiments revealed that WAH is not well suited as a differential data structure. We found that Roaring significantly outperforms WAH in that regard, because (i) the partitioned in-memory layout of Roaring offers significantly faster updates and (ii) the better compression ratios of Roaring reduce the amount of memory occupied by pending updates. We therefore use Roaring as a differential data structure in the following and omit the results for WAH.

Compression method	avg. time per update [ns] non-partitioned	partitioned
TEB	599	218
Roaring	480* / 574	121* / 216
WAH	17634	794

* using the in-memory layout (non serialized)

Table 3: The average time to apply an update.

We measure the update throughput by applying 100k point updates to a compressed bitmap (with n=2^{20}, d=0.1, f=8) and report the average execution time. The number of pending updates is limited to 20k; i.e., a merge is triggered when this threshold is reached. Further, we examine how partitioning affects the execution time of point updates. We partition the bitmap into chunks of 2^{16} bits, whereas each chunk has its own diff. The results in Table 3 show that TEB and Roaring are on par, whereas WAH is several times slower. WAH suffers from the linear time complexity of point lookups that are involved with updates. Data partitioning helps to reduce the access latency significantly, but the average time of an update is still 3.6× higher. The performance of Roaring on the

other hand could be improved by using its in-memory layout and its specialized XOR implementations for the individual container combinations (cf., the results marked with * in Table 3). The optimization is enabled by the fact that both the value bitmap and the differential bitmap are Roaring bitmaps. In a pure in-memory setting, Roaring therefore outperforms TEB by up to 1.8× and WAH by more than 6× in terms of update latency (in the partitioned case).

Pending updates naturally impair read latency. We observed a 30% penalty for TEB and Roaring with 20k pending updates (20% with WAH), irrespective of partitioning. For more general information on the trade-offs involved with differential updates, we refer the reader to UpBit [4].

5 RELATED WORK

Throughout the paper, we already covered the related work regarding bitmap compression techniques [2, 4, 15, 18, 22, 27, 32, 33, 57, 63], except for the HICAMP bitmap [56] which is designed for a special kind of memory system [13]. In the following, we discuss other related work.

Bitmap Indexes. Bitmap indexes and bitmap compression are orthogonal topics, as bitmap indexes may also be constructed with verbatim bitmaps. However, in practice, compression is commonly used to reduce space consumption and to improve query performance. Thus, the term bitmap index often refers to a *compressed* bitmap index. Compression, however, is just one aspect of a bitmap index. Other techniques that are involved when a bitmap index is constructed are (i) *binning* [28, 65, 66] which groups multiple attribute values together and (ii) *encoding* [11, 12, 46] which translates the bins into a set of bitmaps [64]. Thereby, an encoding scheme is chosen that best supports the query workload. Common encodings are equality encoding, range encoding and interval encoding, whereas the latter two allow for arbitrary range queries by accessing at most two bitmaps. Optionally, an attribute value may be *decomposed* into multiple components that are individually assigned to bins afterwards. A single attribute value may therefore map to multiple bins. An extreme case is the bit-sliced index [46, 50], where the attribute values are decomposed bit-by-bit, and the number of bins (and bitmaps) is equal to the bit-width of the attribute.

Binning, encoding, and decomposition influence the characteristics of the individual bitmaps [64] of an index. Consequently, they affect the overall index size and eventually the query performance [25, 62]. A thorough evaluation of TEBs within the large design space of bitmap indexes is therefore beyond the scope of this work.

Succinct Data Structures. The space efficiency of TEBs is founded on the idea of mapping tree nodes to integer values [30] and the foundational work on succinctly encoded binary trees [24] that efficiently support the necessary navigational operations using the rank and select primitives. Both primitives require a helper structure to lower the time complexity of tree navigations from linear to constant time. Several implementations have been been proposed [20, 21, 43, 55, 69] to achieve the performance of pointer-based tree structures. A key to success, in terms of performance, was the introduction of the population count instruction, which unfortunately was quite late in wide-spread x86 processors (AMD 2007, Intel 2008). Over the years, other succinct tree encodings have been proposed [6, 14, 17, 39, 40, 49] that support a richer set of operations or being updatable [41]; both, however, would incur higher space consumption and/or lower performance with TEB.

Lightweight Indexing. Space-efficient secondary index structures, in general, have attracted a lot of interest in database research. Many lightweight data structures have been proposed to accelerate table scans by skipping (i) blocks of tuples [1, 38, 51, 52, 54, 67], (ii) scan ranges within blocks [29], or (iii) (parts of) individual tuples [19, 23, 35, 36, 48]. Other index structures were designed to support specific kinds of queries, e.g., queries with a LIMIT clause [26], or for specific kinds of data, e.g., observational data [58]. Most of these index structures rely on lightweight statistical data that is easy to maintain and query. The more heavyweight approaches either store approximations of the indexed columns [23, 51] or even require a different storage layout [19, 36].

6 CONCLUSION

The Tree-Encoded Bitmap (TEB) is a novel approach for compressing bitmaps. Its tree-based compression algorithm maps 0- or 1-runs of various lengths to binary tree nodes, where the depth of a node implicitly determines its run length. The resulting tree structure is then encoded using a succinct physical data structure that supports logarithmic access time and therefore allows for efficient logical operations (such as intersections) on compressed data. We experimentally showed that TEB saves considerable space compared to other compressed bitmap formats—in particular at higher bit densities, i.e., those cases where memory consumption would otherwise be fairly high. In terms of access speed, TEB is quite fast for intersection operations: almost as fast as the competing approach Roaring, and much faster than WAH. In the data distributions where TEB is strongest in saving space, its raw scan performance is also close to Roaring. As such, TEB encoded chunks could also be used as a worthwhile addition to the adaptive Roaring approach, significantly improving compression in the most difficult data distributions, while preserving performance.

Acknowledgements. This work was supported by the DFG project KE401/22.

REFERENCES

[1] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. 2013. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *PVLDB* 6, 14 (2013), 1714–1725. http://www.vldb.org/pvldb/vol6/p1714-kossmann.pdf

[2] G. Antoshenkov. 1995. Byte-aligned bitmap compression. In *Proceedings DCC '95 Data Compression Conference*. 476–. https://doi.org/10.1109/DCC.1995.515586

[3] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016.* 461–466. https://doi.org/10.5441/002/edbt.2016.42

[4] Manos Athanassoulis, Zheng Yan, and Stratos Idreos. 2016. UpBit: Scalable In-Memory Updatable Bitmap Indexing. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016.* 1319–1332. https://doi.org/10.1145/2882903.2915964

[5] The RoaringBitmap authors. [n.d.]. Roaring Bitmap. https://github.com/RoaringBitmap/RoaringBitmap. [Online; accessed 27-May-2019].

[6] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. 2005. Representing Trees of Higher Degree. *Algorithmica* 43, 4 (2005), 275–292. https://doi.org/10.1007/s00453-004-1146-6

[7] Boost.org. [n.d.]. Boost C++ Libraries. https://www.boost.org/. [Online; accessed 04-Jun-2019].

[8] Michael Cain and Kent Milligan. 2011. IBM DB2 for i indexing methods and strategies. IBM White Paper.

[9] Samy Chambi, Daniel Lemire, Robert Godin, Kamel Boukhalfa, Charles R. Allen, and Fangjin Yang. 2016. Optimizing Druid with Roaring bitmaps. In *Proceedings of the 20th International Database Engineering & Applications Symposium, IDEAS 2016, Montreal, QC, Canada, July 11-13, 2016.* 77–86. https://doi.org/10.1145/2938503.2938515

[10] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2014. Better bitmap performance with Roaring bitmaps. *CoRR* abs/1402.6407 (2014). arXiv:1402.6407 http://arxiv.org/abs/1402.6407

[11] Chee Yong Chan and Yannis E. Ioannidis. 1998. Bitmap Index Design and Evaluation. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.* 355–366. https://doi.org/10.1145/276304.276336

[12] Chee Yong Chan and Yannis E. Ioannidis. 1999. An Efficient Bitmap Encoding Scheme for Selection Queries. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.* 215–226. https://doi.org/10.1145/304182.304201

[13] David R. Cheriton, Amin Firoozshahian, Alex Solomatnikov, John P. Stevenson, and Omid Azizi. 2012. HICAMP: architectural support for efficient concurrency-safe shared structured data access. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012.* 287–300. https://doi.org/10.1145/2150976.2151007

[14] David R. Clark and J. Ian Munro. 1996. Efficient Suffix Trees on Secondary Storage *(SODA '96)*, Vol. 96. Society for Industrial and Applied Mathematics, USA, 383–391.

[15] Alessandro Colantonio and Roberto Di Pietro. 2010. Concise: Compressed 'n' Composable Integer Set. *Inf. Process. Lett.* 110, 16 (2010), 644–650. https://doi.org/10.1016/j.ipl.2010.05.018

[16] Oracle Corporation. 2005. Bitmap Index vs. B-tree Index: Which and When? https://www.oracle.com/technetwork/articles/sharma-indexes-093638.html. [Online; accessed 14-Jun-2019].

[17] Pooya Davoodi, Rajeev Raman, and Srinivasa Rao Satti. 2017. On Succinct Representations of Binary Trees. *Mathematics in Computer Science* 11, 2 (2017), 177–189. https://doi.org/10.1007/s11786-017-0294-4

[18] François Deliège and Torben Bach Pedersen. 2010. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings.* 228–239. https://doi.org/10.1145/1739041.1739071

[19] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. 2015. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015.* 31–46. https://doi.org/10.1145/2723372.2747642

[20] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From Theory to Practice: Plug and Play with Succinct Data Structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014).* 326–337.

[21] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. 2005. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA).* 27–38.

[22] Gheorghi Guzun, Guadalupe Canahuate, David Chiu, and Jason Sawin. 2014. A tunable compression framework for bitmap indices. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014.* 484–495. https://doi.org/10.1109/ICDE.2014.6816675

[23] Brian Hentschel, Michael S. Kester, and Stratos Idreos. 2018. Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018.* 857–872. https://doi.org/10.1145/3183713.3196911

[24] Guy Jacobson. 1989. Space-efficient Static Trees and Graphs. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989.* 549–554. https://doi.org/10.1109/SFCS.1989.63533

[25] Theodore Johnson. 1999. Performance Measurements of Compressed Bitmap Indices. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK.* 278–289. http://www.vldb.org/conf/1999/P29.pdf

[26] Albert Kim, Liqi Xu, Tarique Siddiqui, Silu Huang, Samuel Madden, and Aditya G. Parameswaran. 2016. Speedy Browsing and Sampling with NeedleTail. *CoRR* abs/1611.04705 (2016). arXiv:1611.04705 http://arxiv.org/abs/1611.04705

[27] Sangchul Kim, Junhee Lee, Srinivasa Rao Satti, and Bongki Moon. 2016. SBH: Super byte-aligned hybrid bitmap compression. *Inf. Syst.* 62 (2016), 155–168. https://doi.org/10.1016/j.is.2016.07.004

[28] Nick Koudas. 2000. Space Efficient Bitmap Indexing. In *Proceedings of the 2000 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA, November 6-11, 2000.* 194–201. https://doi.org/10.1145/354756.354819

[29] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016.* 311–326. https://doi.org/10.1145/2882903.2882925

[30] C. C. Lee, D. T. Lee, and C. K. Wong. 1986. Generating Binary Trees of Bounded Height. *Acta Inf.* 23, 5 (1986), 529–544. https://doi.org/10.1007/BF00288468

[31] Daniel Lemire. [n.d.]. Official Roaring Bitmap website. https://roaringbitmap.org. [Online; accessed 27-May-2019].

[32] Daniel Lemire, Gregory Ssi Yan Kai, and Owen Kaser. 2016. Consistently faster and smaller compressed bitmaps with Roaring. *Softw., Pract. Exper.* 46, 11 (2016), 1547–1569. https://doi.org/10.1002/spe.2402

[33] Daniel Lemire, Owen Kaser, and Kamel Aouiche. 2010. Sorting improves word-aligned bitmap indexes. *Data Knowl. Eng.* 69, 1 (2010), 3–28. https://doi.org/10.1016/j.datak.2009.08.006

[34] Daniel Lemire, Owen Kaser, and Eduardo Gutarra. 2012. Reordering rows for better compression: Beyond the lexicographic order. *ACM Trans. Database Syst.* 37, 3 (2012), 20:1–20:29. https://doi.org/10.1145/2338626.2338633

[35] Yinan Li, Craig Chasseur, and Jignesh M. Patel. 2015. A Padded Encoding Scheme to Accelerate Scans by Leveraging Skew. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 1509–1524. https://doi.org/10.1145/2723372.2737787

[36] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: fast scans for main memory data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. 289–300. https://doi.org/10.1145/2463676.2465322

[37] Roger MacNicol and Blaine French. 2004. Sybase IQ Multiplex - Designed For Analytics. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. 1227–1230. https://doi.org/10.1016/B978-012088469-8.50111-X

[38] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*. 476–487. http://www.vldb.org/conf/1998/p476.pdf

[39] J. Munro and V. Raman. 2001. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM J. Comput.* 31, 3 (2001), 762–776. https://doi.org/10.1137/S0097539799364092 arXiv:https://doi.org/10.1137/S0097539799364092

[40] J. Ian Munro and Venkatesh Raman. 1997. Succinct Representation of Balanced Parentheses, Static Trees and Planar Graphs. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*. 118–126. https://doi.org/10.1109/SFCS.1997.646100

[41] J. Ian Munro, Venkatesh Raman, and Adam J. Storm. 2001. Representing dynamic binary trees succinctly. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA*. 529–536. http://dl.acm.org/citation.cfm?id=365411.365526

[42] Parth Nagarkar, K. Selçuk Candan, and Aneesha Bhat. 2015. Compressed Spatial Hierarchical Bitmap (cSHB) Indexes for Efficiently Processing Spatial Range Query Workloads. *PVLDB* 8, 12 (2015), 1382–1393. http://www.vldb.org/pvldb/vol8/p1382-nagarkar.pdf

[43] Gonzalo Navarro and Eliana Providel. 2012. Fast, Small, Simple Rank/Select on Bitmaps. In *Experimental Algorithms - 11th International Symposium, SEA 2012, Bordeaux, France, June 7-9, 2012. Proceedings*. 295–306. https://doi.org/10.1007/978-3-642-30850-5_26

[44] Patrick O'Neil and Goetz Graefe. 1995. Multi-table Joins Through Bitmapped Join Indices. *SIGMOD Rec.* 24, 3 (Sept. 1995), 8–11. https://doi.org/10.1145/211990.212001

[45] Patrick E. O'Neil. 1987. Model 204 Architecture and Performance. In *High Performance Transaction Systems, 2nd International Workshop, Asilomar Conference Center, Pacific Grove, California, USA, September 28-30, 1987, Proceedings*. 40–59. https://doi.org/10.1007/3-540-51085-0_42

[46] Patrick E. O'Neil and Dallan Quass. 1997. Improved Query Performance with Variant Indexes. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*. 38–49. https://doi.org/10.1145/253260.253268

[47] Ali Pinar, Tao Tao, and Hakan Ferhatosmanoglu. 2005. Compressing Bitmap Indices by Data Reorganization. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*. 310–321. https://doi.org/10.1109/ICDE.2005.35

[48] Orestis Polychroniou and Kenneth A. Ross. 2015. Efficient Lightweight Compression Alongside Fast Scans. In *Proceedings of the 11th International Workshop on Data Management on New Hardware, DaMoN 2015, Melbourne, VIC, Australia, May 31 - June 04, 2015*. 9:1–9:6. https://doi.org/10.1145/2771937.2771943

[49] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms* 3, 4 (2007), 43. https://doi.org/10.1145/1290672.1290680

[50] Denis Rinfret, Patrick E. O'Neil, and Elizabeth J. O'Neil. 2001. Bit-Sliced Index Arithmetic. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*. 47–57. https://doi.org/10.1145/375663.375669

[51] Lefteris Sidirourgos and Martin L. Kersten. 2013. Column imprints: a secondary index structure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. 893–904. https://doi.org/10.1145/2463676.2465306

[52] Lefteris Sidirourgos and Hannes Mühleisen. 2017. Scaling column imprints using advanced vectorization. In *Proceedings of the 13th International Workshop on Data Management on New Hardware, DaMoN 2017, Chicago, IL, USA, May 15, 2017*. 4:1–4:8. https://doi.org/10.1145/3076113.3076120

[53] Rishi Rakesh Sinha and Marianne Winslett. 2007. Multi-resolution bitmap indexes for scientific data. *ACM Trans. Database Syst.* 32, 3 (2007), 16. https://doi.org/10.1145/1272743.1272746

[54] The PostgreSQL Global Development Group. [n.d.]. Block Range Index (BRIN) in PostgreSQL. https://www.postgresql.org/docs/11/brin.html. [Online; accessed 01-Jul-2019].

[55] Sebastiano Vigna. 2008. Broadword Implementation of Rank/Select Queries. In *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*. 154–168. https://doi.org/10.1007/978-3-540-68552-4_12

[56] Bo Wang, Heiner Litz, and David R. Cheriton. 2014. HICAMP bitmap: space-efficient updatable bitmap index for in-memory databases. In *Tenth International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014*. 7:1–7:7. https://doi.org/10.1145/2619228.2619235

[57] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An Experimental Study of Bitmap Compression vs. Inverted List Compression. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. 993–1008. https://doi.org/10.1145/3035918.3064007

[58] Sheng Wang, David Maier, and Beng Chin Ooi. 2014. Lightweight Indexing of Observational Data in Log-Structured Storage. *PVLDB* 7, 7 (2014), 529–540. http://www.vldb.org/pvldb/vol7/p529-wang.pdf

[59] J. W. J. Williams. 1964. Algorithm 232: Heapsort. *Commun. ACM* 7, 6 (1964), 347–348.

[60] John Wu and Kurt Stockinger. [n.d.]. FastBit: An Efficient Compressed Bitmap Index Technology. https://sdm.lbl.gov/fastbit/. [Online; accessed 27-May-2019].

[61] Kesheng Wu, Sean Ahern, E Wes Bethel, Jacqueline Chen, Hank Childs, Estelle Cormier-Michel, Cameron Geddes, Junmin Gu, Hans Hagen, Bernd Hamann, et al. 2009. FastBit: interactively searching massive data. In *Journal of Physics: Conference Series*, Vol. 180. IOP Publishing, 012053.

[62] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. 2004. On the performance of bitmap indices for high cardinality attributes. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. 24–35. https://doi.org/10.1016/B978-012088469-8.50006-1

[63] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. 2006. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.* 31, 1 (2006), 1–38. https://doi.org/10.1145/1132863.1132864

[64] Kesheng Wu, Arie Shoshani, and Kurt Stockinger. 2010. Analyses of multi-level and multi-component compressed bitmap indexes. *ACM Trans. Database Syst.* 35, 1 (2010), 2:1–2:52. https://doi.org/10.1145/1670243.1670245

[65] Kun-Lung Wu and Philip S. Yu. 1998. Range-Based Bitmap Indexing for High Cardinality Attributes with Skew. In *COMPSAC '98 - 22nd International Computer Software and Applications Conference, August 19-21, 1998, Vienna, Austria*. 61–67. https://doi.org/10.1109/CMPSAC.1998.716637

[66] Ming-Chuan Wu and Alejandro P. Buchmann. 1998. Encoded Bitmap Indexing for Data Warehouses. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*. 220–230. https://doi.org/10.1109/ICDE.1998.655780

[67] Jia Yu and Mohamed Sarwat. 2016. Two Birds, One Stone: A Fast, yet Lightweight, Indexing Scheme for Modern Database Systems. *PVLDB* 10, 4 (2016), 385–396. http://www.vldb.org/pvldb/vol10/p385-yu.pdf

[68] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. 323–336. https://doi.org/10.1145/3183713.3196931

[69] Dong Zhou, David G. Andersen, and Michael Kaminsky. 2013. Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*. 151–163. https://doi.org/10.1007/978-3-642-38527-8_15