In-Place Updates in Tree-Encoded Bitmaps

Marcellus Prama Saputra
saputra@campus.tu-berlin.de
Technische Universität Berlin
Germany

Eleni Tzirita Zacharatou
elza@itu.dk
IT University of Copenhagen
Denmark

Serafeim Papadias
s.papadias@tu-berlin.de
Technische Universität Berlin
Germany

Volker Markl
volker.markl@tu-berlin.de
Technische Universität Berlin, DFKI
Germany

ABSTRACT

The Tree-Encoded Bitmap (TEB) is a novel bitmap compression scheme that provides a high compression ratio and logarithmic read time. It uses a tree-based compression algorithm that maps runs in the bitmap to leaf nodes of a binary tree. Currently, TEBs perform updates using an auxiliary differential data structure. However, consulting an additional data structure at every read introduces both memory and read overheads. To mitigate the shortcomings of differential updates, we propose algorithms to update TEBs in place. To that end, we identified two types of updates that can occur in a TEB: run-forming and run-breaking updates. Run-forming updates correspond to leaf nodes at the lowest level of the binary tree. All other updates are run-breaking. Each type of update requires different handling. Through experimentation with synthetic data, we determined that in-place run-forming updates are 2-3× faster than differential updates, while run-breaking updates cannot be efficiently performed in place. As a result, we propose a hybrid solution that performs run-forming updates in place while storing run-breaking updates in a differential data structure. Our experiments using synthetic data show that our hybrid solution is faster than differential updates as long as run-forming updates occur in a given workload. For instance, when 7% of all updates are run forming, our hybrid solution is 15% faster than differential updates.

Artifact Availability: The source code has been made available at https://github.com/marcellus-saputra/Thuja.

ACM Reference Format:
Marcellus Prama Saputra, Eleni Tzirita Zacharatou, Serafeim Papadias, and Volker Markl. 2022. In-Place Updates in Tree-Encoded Bitmaps. In *Proceedings of International Conference on Scientific and Statistical Database Management (SSDBM '22).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Bitmap indexes have had a long history in database systems [5, 7, 8]. Traditionally, they are used in read-heavy workloads and are particularly suited for indexing low cardinality attributes. Bitmap indexes are usually sparse, with few 1-bits interspersed between 0-bits. Additionally, the size of a bitmap index scales with the number of rows as well as the cardinality of the indexed attribute. As a result, to minimize the memory overhead, bitmap indexes typically employ compression schemes.

The Tree-Encoded Bitmap (TEB) [6] is a novel bitmap compression scheme that represents bitmaps as binary trees. Specifically, TEBs map 1-runs and 0-runs to leaf nodes, where the proximity of a leaf node to the root indicates the length of the run. Then, they encode the binary tree into two bitmaps, T and L. T represents the structure of the tree and L contains the labels of the leaf nodes. A label is either a 1-bit or a 0-bit, indicating a 1-run and a 0-run, respectively. TEBs boast better compression ratios than the state-of-the-art Roaring bitmap [4] while also having logarithmic read time. However, their low read and memory overheads come at the cost of high update overheads, as the RUM conjecture [1] indicates.

Currently, TEBs only support differential updates, i.e., they use an auxiliary differential data structure to store the updates. Typically, the differential data structure is also a compressed bitmap. Once the differential data structure reaches a certain number of stored updates, it is merged with the TEB. Differential updates offer a high upfront update performance, but this comes at the cost of read and memory overhead. For every read, the differential data structure needs to be consulted first before accessing the TEB. Moreover, as more updates are stored in the differential data structure, it becomes more complex and thus less compressible, leading to further increased read and memory overhead, which can only be mitigated by merging the differential data structure with the TEB.

In-place updates avoid the use of differential data structures along with their associated read and memory overheads. Specifically, to perform an in-place update in a TEB, we need to navigate and directly modify the T and L bitmaps. In this paper, we provide algorithms for performing updates in place as well as a mechanism that combines in-place updates with differential updates to achieve the best of both worlds. In summary, after presenting the key concepts of TEBs (Section 2) we make the following contributions:

(1) We identify two types of updates that can occur in TEBs, i.e., *run-forming* and *run-breaking* updates. An update is run-forming if it affects a leaf node that resides at the lowest possible level of the binary tree; otherwise it is run breaking. We handle each type accordingly (Section 3).

(2) We propose a hybrid approach for updating TEBs that combines in-place with differential updates. In this approach, incoming

SSDBM '22, July 06–08, 2022, Copenhagen, Denmark
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

updates are first checked to determine whether they are run forming. If they are, then they are performed in place; otherwise, they are stored in a differential data structure (Section 4).

(3) We validate our approach through experiments on synthetic data to show that our approach is faster than plain differential updates; the number of run-forming updates that are performed in a given workload determines the degree of the achieved speedup.

We then discuss related work in Section 6, where we describe existing differential update methods for bitmaps in the context of bitmap indexes, and conclude the paper in Section 7.

2 TREE-ENCODED BITMAPS

In this section, we discuss the basic workings of TEBs; how they are constructed, navigated, and how they are currently updated. For more details about TEBs, we refer the reader to [4].

Construction. A TEB is constructed in two steps. First, a perfect binary tree is constructed on top of the original bitmap. If the length of the original bitmap is not a power of 2, then 0-bits are appended to the original bitmap until its length is equal to the next nearest power of 2. Next, the binary tree is pruned bottom-up; if a pair of sibling leaf nodes possess the same label, they are removed and their label is assigned to their parent node. After the entire tree has been appropriately pruned, it is encoded into two bitmaps: T and L. Namely, while traversing the tree in level order, if a leaf node is visited, then a 0-bit is appended to T and its label is appended to L; otherwise, a 1-bit is appended to T while nothing is appended to L.

Navigation. The ID of the right child of a given inner node i can be calculated using the following formula: $right-child = 2 \cdot rank(i)$, where the rank of i indicates the inclusive cumulative number of 1-bits preceding i in T. Counting all the 1-bits in the first i bits in T is linear to the size of T. However, TEBs also implement a so-called rank lookup table, which contains the cumulative number of 1-bits in T in 512 bit intervals. The granularity of the rank lookup table is modifiable, but 512 bits per block was found to strike a good balance between performance and additional memory overhead [6]. A point lookup operation is performed by navigating the TEB until a leaf node is found, and then returning it. The navigation is guided by the binary representation of the searched position.

Differential Updates. Currently, TEBs perform updates by storing them in a differential data structure, which is typically another compressed bitmap. Storing updates works similarly to UpBit [2]; every bit in the differential data structure denotes whether its position has been updated. The actual value of a given position p can then be obtained by XORing the value at p in the TEB with the bit at p in the differential data structure. As updates are accumulated within the differential data structure, it becomes more complex and thus, less compressible. Therefore, after the differential data structure reaches a certain size, or after a certain number of updates have been stored, merging can be performed. The merge operation applies all stored updates into the TEB, and resets the differential data structure in the process. Currently, merging is done by first decompressing the TEB, followed by bitwise XORing the TEB with the differential data structure, and finally constructing a new TEB using the resulting bitmap. The Roaring bitmap was found to be the most suitable differential data structure [6].

3 IN-PLACE UPDATES

In this section, we discuss our approach for performing in-place updates in TEBs. Note that only point updates are discussed, i.e., updates to a single position. Range updates are not within the scope of this paper.

To update a TEB, the first step is to find the leaf node that corresponds to the updated position, for which we can use a point lookup. Let P be this node. Depending on whether P resides at the lowest possible level of the tree or not, we classify updates into two types that require different handling: run-forming and run-breaking updates.

3.1 Run-Forming Updates

A run-forming update is characterized by P residing at the lowest possible level of the tree. This is important, as at the lowest possible level of the tree, every leaf node represents a single individual bit in the original bitmap. Therefore, to perform a run-forming update in place, we only need to change P's label bit in L. This is done by retrieving the position of P's label bit and negating it. Note that run-forming updates may lead to additional pruning opportunities. Specifically, after we change the label of P, it may have the same label as its sibling, which means they could be pruned to save additional space. Similarly, after pruning P and its sibling, P's parent node may also be assigned the same label as its sibling, which leads to more pruning opportunities.

Pruning a TEB in place is challenging for several reasons. First, we must modify the T and L bitmaps directly. However, the original TEB implementation by Lang et al. [6] assumes static TEBs and thus does not contain any functionality to modify T and L. Second, to prune a pair of leaf nodes, we need to know the node ID of the parent node. However, TEBs only support efficient downward navigation from the root to the leaves using the rank lookup table. Therefore, to obtain the parent node, we need to traverse the TEB from the root, which is a linear time operation. Finally, removing/inserting bits from/into T and L is a linear time operation, since T and L are stored as arrays of 64-bit integer words. In the worst case, every word needs to be adjusted after removing or inserting even a single bit. As a result, a naive in-place pruning algorithm that traverses the TEB from the root and inserts/removes bits from/into T and L for every pair of pruned nodes incurs a quadratic complexity.

To perform run-forming updates efficiently in place, we use the following two key insights. First, a TEB remains correct after performing a run-forming update even without pruning. Second, if we know in advance all the positions in T and L that are affected by pruning a pair of leaf nodes, we can batch all the changes to the T and L bitmaps into a single operation instead of performing a series of removals and insertions. The first insight allows to prune the TEB only after performing several updates instead of pruning after every update, similar to merging in differential updates. To leverage our second insight, prior to the actual pruning, we traverse the TEB and mark the positions that require modifications. Afterwards, we can perform all changes to T and L in a single operation, thereby optimizing the runtime. Finally, we note that the run-forming updates themselves only require a point lookup operation followed by negating a single bit in L, and thus, they are extremely lightweight. Overall, our algorithm performs run-forming updates in place while delaying pruning and batching changes to T and L. That way, as we show in Section 5, in-place run-forming updates can be performed 2-3× faster than differential updates.

3.2 Run-Breaking Updates

An update is run breaking if P does not reside at the lowest level, which means P is a leaf node that represents a run. Therefore, to accurately represent the updated position, the run needs to be broken. A broken run can only be represented by a subtree, which means that P needs to be expanded into a subtree. This is achieved by first turning P into an inner node by setting its bit in T to a 0-bit. Nodes are then inserted in pairs into the TEB to represent P's descendants; the last pair of nodes are leaf nodes.

However, early experiments show that our implementation of in-place run-breaking updates is not efficient. This is because unlike run-forming updates, the majority of run-breaking updates cannot be performed in place without heavily modifying T and L. Additionally, TEBs allocate space statically, and thus it impossible to guarantee that there is enough allocated space to perform every run-breaking update in place. Finally, we argue that a run-breaking update requires at least linear time, unless T and L can be directly modified in sub-linear time. This could be achieved by first marking all positions in T and L that need to be changed using a point lookup-like operation, followed by performing all necessary modifications to T and L in a single iteration.

4 HYBRID UPDATES

In-place run-forming updates are significantly faster than differential updates, whereas in-place run-breaking updates are slower. To achieve the best of both worlds, we devise a hybrid approach that combines in-place and differential updates. In essence, our hybrid approach for updating TEBs first checks whether an update is run forming. If it is, then it performs the update in place. Otherwise, it stores it in a differential data structure.

Let us now describe how our hybrid approach for updating TEBs works. Let p be the position of the bitmap that we want to update with value v, B be the base TEB, and D be the differential data structure. First, we perform a point lookup for p on both D and B to determine the actual value at position p by XORing $D[p]$ with $B[p]$. From the point lookup on B we also obtain node n, the leaf node that represents p. If the actual value is equal to v, then the update is redundant and is canceled. Otherwise, we check n to see whether it resides at the lowest possible level. If n indeed resides at the lowest possible level, then the update is run forming. Therefore, we negate $B[p]$ to perform the update. Otherwise, if n does not reside at the lowest possible level, then we perform the update as a differential one. To do that, we negate $D[p]$.

Our hybrid approach brings the following benefits. First, since some updates are not stored in the differential data structure, its size increases at a slower pace, which reduces the associated memory overhead. Second, since fewer differential updates are performed, the differential data structure is merged with the TEB less frequently. As a result, fewer merge operations are needed compared to using pure differential updates in the long run.

We must also note that the above-mentioned benefits of the hybrid approach only apply for workloads that perform several run-forming updates. Yet, there is no guarantee that any number of run-forming updates are possible in a given TEB, i.e., that the TEB has leaf nodes at the lowest possible level, and that many such leaf nodes exist. Additionally, even if there are leaf nodes at the lowest possible level, a given sequence of updates might not affect these leaf nodes. In this case, our approach performs identically to normal differential updates. However, we argue that the above-mentioned situation is a corner case as it would require an extremely clustered bitmap, e.g., a bitmap for a sorted attribute.

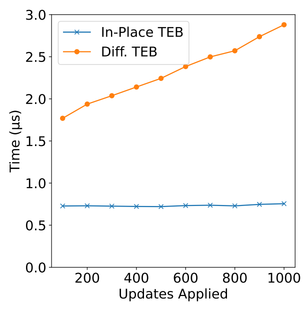

Figure 1: Update Latency.

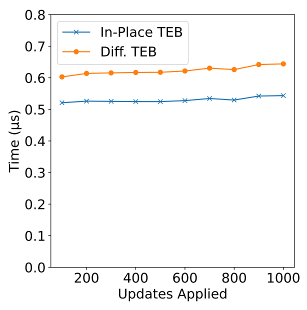

Figure 2: Lookup Latency.

5 EXPERIMENTAL EVALUATION

In this section, we first present our experimental setup and then evaluate our approach with respect to its update latency, lookup latency, and space consumption.

5.1 Experimental Setup

The experiments were performed on a machine equipped with an Intel® Core™ i7-8700K CPU @ 3.70GHz processor and 16GB RAM. In our experiments, we use synthetic bitmaps that we randomly generated based on two parameters, the bit density d and the clustering factor f. The bit density $d \in [0, 1]$ is the probability of an arbitrary position to contain a 1-bit. The clustering factor f represents the likelihood that a 1-bit is followed by another 1-bit. To generate the bitmaps for the experiments in Section 5.2, we perform a Bernoulli test in every position using d as the test parameter to determine the bit value. In addition to the bit density d, in Section 5.3 we also use the clustering factor f as a parameter to generate bitmaps that are similar to the ones used by Lang et al [6].

5.2 Run-Forming Updates

This experiment measures the performance of in-place run-forming updates. Specifically, we construct two TEBs using a randomly generated bitmap, one that performs differential updates with Roaring as its differential data structure, and one performing in-place updates. In this case, the bitmap is 1 million bits long and has a bit density of 0.1. Run-forming updates are then randomly generated and applied to the TEB. We report the update time after performing a certain cumulative number of updates. For instance, in Figure 1 the 400 "Updates Applied" indicates the time to perform the 400^{th} update in the sequence. Figure 1 shows the update performance of in-place run-forming updates and differential updates. Even from the first update, run-forming updates can be performed significantly

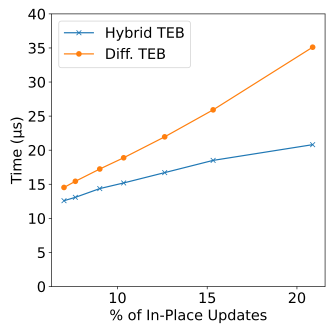

Figure 3: Average Update Latency for 100K Hybrid Updates.

faster in place than differential updates. In-place run-forming updates are 2.43 times faster than differential updates for the first 100 updates. At 1000 updates, in-place run-forming updates are 3.81 times faster than differential updates.

We also compare the point lookup performance between a TEB without and a TEB with a differential data structure. After applying a certain number of updates, we perform a point lookup on every position in both TEB instances and measure the average time. Figure 2 shows that point lookups are also faster without a differential data structure. Moreover, differential updates also perform worse over time, owing to the differential data structure becoming more complex with every update stored. Meanwhile, the performance of in-place run-forming updates remains constant throughout the experiment. We also measured the space occupied by both: after 1000 run-forming updates, the TEB that uses differential updates was 10% larger than the TEB that uses in-place updates, due to the extra space occupied by the differential data structure.

5.3 Hybrid Updates

This experiment compares the performance between differential updates and hybrid updates. We randomly generate several bitmaps that are 1 million bits long in a way that varies the likelihood that a random update is run forming. This is done by varying the bit density and clustering factor. Specifically, an update to a bitmap with high bit density and low clustering is more likely to be run forming than one to a sparsely populated bitmap with high clustering. For every bitmap, two TEBs are constructed: one that uses hybrid updates, and one that uses differential updates with Roaring as the differential data structure. Furthermore, we randomly generate 100K updates and apply them to both TEBs without checking beforehand whether they are run-forming updates or not.

We can see from Figure 3 that even when only 7% of all updates are run-forming, hybrid updates are on average 15% faster than differential updates. This gap only widens as more run-forming updates are possible; when 20% of all updates are run forming, hybrid updates are 69% faster. We also note that as a result of performing run-forming updates in place, the overall TEB size was reduced by 4-9%. Furthermore, note that the average update latency increases as more updates are performed in place. This occurs in both the TEB with hybrid updates as well as the TEB with plain differential updates. If more run-forming updates are possible in a given TEB state, that means there are more leaf nodes at the lowest level. As a result, the point lookup operation that precedes both a differential and a hybrid update needs to navigate to the lowest level, and thus take longer to perform.

6 RELATED WORK

In this section, we focus on established approaches for efficient updates to bitmap indexes.

Update Conscious Bitmaps. UCBs [3] are updated using a delete-then-insert mechanism. This is achieved by utilizing an auxiliary bitvector called the *existence bitvector* (EV). Each element in the EV is mapped to a row in the bitmap index. When a row is updated, its corresponding position in the EV is invalidated by setting it to 0, and the updated row is appended to the bitmap index.

UpBit. UpBit [2] uses a similar approach to UCBs, but at a more granular level. Specifically, it maintains one *update bitvector* (UV) per domain value. UVs are also compressed and remain highly compressible throughout their lifetime. As a result, their memory overhead is negligible. When a row is updated, both the corresponding row in the UV of the previous value and the row in the UV of the updated value are changed. Updating UVs increases their complexity, which reduces their compressibility. When the UVs become significantly large, they are merged with the bitmap index by applying all updates to the bitmap index and resetting all UVs.

7 CONCLUSION

In this paper, we proposed in-place updates for Tree-Encoded Bitmaps (TEBs). We achieved this by identifying the two types of updates that can occur in a TEB, i.e., run-forming and run-breaking updates, and handling each type differently. Through our experiments, we found that run-forming updates can be performed significantly faster than the current approach that uses differential updates. However, this was not the case for run-breaking updates. As a result, we proposed a hybrid solution that performs run-forming updates in place where possible, while other updates are performed as differential updates. Depending on the number of run-forming updates that are performed in a given sequence of updates, our hybrid approach is 15-69% faster than differential updates, while being identical to differential updates in the worst case.

REFERENCES

[1] 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 *EDBT*. 461–466. https://doi.org/10.5441/002/edbt.2016.42
[2] Manos Athanassoulis, Zheng Yan, and Stratos Idreos. 2016. UpBit: Scalable In-Memory Updatable Bitmap Indexing. In *SIGMOD*. 1319–1332. https://doi.org/10.1145/2882903.2915964
[3] Guadalupe Canahuate, Michael Gibas, and Hakan Ferhatosmanoglu. 2007. Update Conscious Bitmap Indices. In *SSDBM*. 15. https://doi.org/10.1109/SSDBM.2007.24
[4] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2015. Better bitmap performance with Roaring bitmaps. *Software: Practice and Experience* 46, 5 (2015), 709–719. https://doi.org/10.1002/spe.2325
[5] Chee-Yong Chan and Yannis E. Ioannidis. 1998. Bitmap Index Design and Evaluation. In *SIGMOD*. 355–366. https://doi.org/10.1145/276304.276336
[6] Harald Lang, Alexander Beischl, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2020. Tree-Encoded Bitmaps. In *SIGMOD*. 937–967. https://doi.org/10.1145/3318464.3380588
[7] Patrick E. O'Neil. 1989. MODEL 204 architecture and performance. In *High Performance Transaction Systems*. 39–59.
[8] Eleni Tzirita Zacharatou, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. 2015. RUBIK: efficient threshold queries on massive time series. In *SSDBM*. 18:1–18:12. https://doi.org/10.1145/2791347.2791372