CS490 Technical Report — Updateable Bitmap Indexes

Hyeong Kyun Park (park1119@purdue.edu); Winston Chen (chen2909@purdue.edu); Gregory Chininis (gchinini@purdue.edu); Daniel Hooks (hooksd@purdue.edu); Michael Lee (lee2965@purdue.edu);
https://github.com/h1yung/PyUpBit

1 INTRODUCTION

Bitmaps are common data structures used in database implementations due to having fast read performance. Often they are used in applications in need of common equality and selective range queries. Essentially, they store a bit-vector for each value in the domain of each attribute to keep track of large scale data files. However, the main drawbacks associated with bitmap indexes are its encoding and decoding performances of bit-vectors.

Currently the state of art update-optimized bitmap index, update conscious bitmaps, are able to support extremely efficient deletes and have improved update speeds by treating updates as delete then insert. Update conscious bitmaps make use of an additional bit-vector, called the existence bit-vector, to keep track of whether or not a value has been updated. By initializing all values of the existence bit-vector to 1, the data for each attribute associated with each row in the existence bit-vector is validated and presented. If a value needs to be deleted, the corresponding row in the existence bit-vector gets changed to 0, invalidating any data associated with that row. This new method in turn allows for very efficient deletes. To add on, updates are then performed as a delete operation, then an insert operation in to the end of the bit-vector.

However, update conscious bitmaps do not scale well with more data. As more and more data gets updated and inserted, the run time increases significantly as well. Because update queries are out-of-place and increase size of vectors, read queries become increasingly expensive and time consuming. Furthermore, as the number of updates and deletes increases, the bit-vector becomes less and less compressible.

This brings us to updateable Bitmaps (UpBit). According to the paper, *UpBit: Scalable In-Memory Updatable Bitmap Indexing*, researchers Manos Athanassoulis, Zheng Yan, and Stratos Idreos developed a new bitmap structure that improved the write performance of bitmaps without sacrificing read performance. The main differentiating point of UpBit is its use of an update bit vector for every value in the domain of an attribute that keeps track of updated values. This allows for faster write performance without sacrificing read performance.

Based on this paper, we implemented UpBit and compared it to our implementation of update conscious bitmaps to compare and test the performances of both methods.

2 APPROACH/IMPLEMENTATION

2.1 Data Structures

We used the bit-vector Python library [?] for generating integer bit vectors of values for indexing and converting for searching and updating. Here, we define bitmap as a dictionary taking a dictionary of attribute names and their domains. Value bit vectors and updating bit vectors were saved as lists. We decided to create our own bitmap for a singular attribute. A similar approach was followed by the UCB implementation.

2.2 Compression

WAH (Word Aligned Hybrid Bitmaps) In order to improve the memory usage of bitmaps, bit-vectors are encoded using Word-Aligned Hybrid compression (WAH), which is a modified form of run-length encoding. WAH compresses bit-vectors in groups of 31 bits by considering three cases. First, 31 bits contains a combination of 1s and 0s. Second, the 31 bits consists solely of 1s. Third, the 31 bits consists solely of 0s. An encoded word consists of 32 bits; the first, the signal bit, differentiates between the first case and the other two cases. In the first case, the signal bit is 0, followed by the 31 literal bits — that is, no compression is performed. In the second and third cases, the signal bit is a 1, followed by the fill bit, which differentiates between case two and three. In case two, the fill bit is a 1; in case three, the fill bit is a 0. The following 30 bits represent an integer value in binary denoting how many consecutive rows in the un-encoded bit-vector that consist of all 1s or all 0s. This compression algorithm is particularly useful for bitmaps since the value bit-vectors are inherently sparse; this allows a vast majority of the bit-vectors on average to be compressed, as we expect to see a high number of consecutive rows that consist of all 0s.

A major challenge that traditional bitmaps and update-conscious bitmaps face is the problem of needing to decompress bit-vectors, write to them, and then re-compress them. This can result in a significant amount of overhead in write read and write latency if the index is not designed to optimize these operations. UpBit aims to solve this problem by using fence-pointers, which provide a way to locate where a particular bit resides within the compressed encoding.

2.3 Functions

The first, and most important, function that was implemented is updating a row. In order to do so, UpBit flips the bit in the row's position of the update bit-vector for a specific value. Additionally, UpBit needs to flip the bit in the row's position of the update bit-vector for the old value, which will now be empty.

To implement the insertion of a new row, first it is necessary to find the bit-vectors for the value that we are inserting a new row for. UpBit then checks if the bit-vectors have padding space that will allow it to insert a new row. Once space is created, the bit-vector's size is increased by one, and the bit corresponding to the new row is set to 1 in the value bit-vector.

Next, is the function used to delete a row. First, UpBit finds the value in the bitmap that the row to-be-deleted corresponds to. Once found, then UpBit finds the update bit-vector that corresponds to

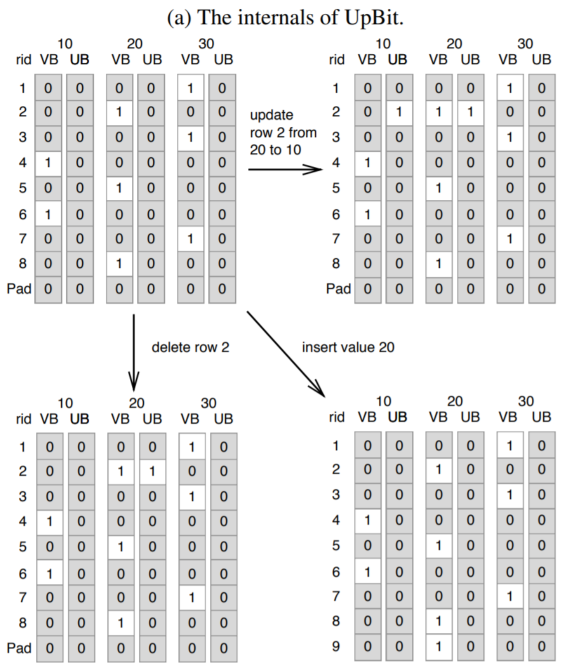

Figure 1: Update, Insert, and Delete Functions for UpBit. [1]

the value found in the last step. Finally, the bit in the update bit-vector for the specified value at the position that will be deleted is flipped.

Lastly, the function used to search or query for a bit-vector is described. The purpose of searching is to discover if a value exists in a column and what row it exists at. The first step is locating the bit-vectors that correspond to the queried value. Next, UpBit checks whether the update bit-vector has been altered, or if it contains all zero values. If the update bit-vector contains all zero values, then search returns the value bit-vector. However, if the update bit-vector contains any updates at all, then search returns the bitwise XOR of the value bit-vector and the update bit-vector.

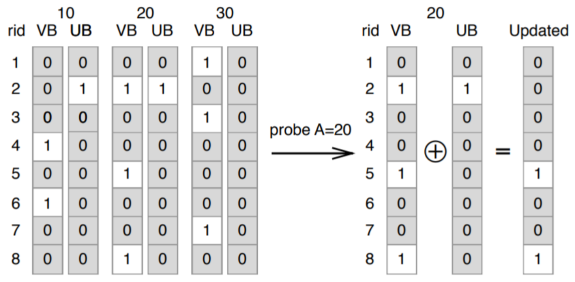

Figure 2: Search Function for UpBit. [1]

2.4 Dataset Generation

To test our data structure's correctness and performance, we generated various synthetic datasets consisting of integer data sampled from two different distributions. For simplicity, we use single-attribute datasets, since the bitmap index works on a per-attribute basis. Namely, we vary three parameters for dataset creation: n, the total number of records in the dataset, d, the cardinality of the attribute's domain, and the distribution from which the integers are sampled from, either a uniform distribution or a zipfian distribution. We experiment with n values of different orders of magnitude, ranging from one thousand to one hundred thousand, and d values of either 10 or 100. The two distributions impact the compressibility and therefore memory usage and overall performance of the data structure. More specifically, while the uniform distribution could be seen as a baseline, we expect to see the performance on the zipfian dataset to be much better. When sampling from the zipfian distribution, the majority of values fall within the first few values of the attribute's domain. This means that the remaining values in the attribute domain will be extremely sparse, leading to higher compressibility as we see even more consecutive 0s in their corresponding value bit-vectors than that of the uniform dataset.

3 EXPERIMENTATION

Because we implemented our version of UpBit in Python and the original implementation was in C++, we knew that the original implementation would perform better. Therefore, we decided to test our implementation by performing relative comparisons to our own implementation of update conscious bitmaps (UCB) in Python. Primarily, we compared the read times of UpBit and UCB to perform for all four functions (Update, Insert, Delete, Search). In performing our tests, we decided to use a synthetic dataset from a uniform distribution with n = 100,000 and our cardinality of the attribute's domains d = 100. Additionally, the values we obtained from testing are averages across 10 runs. Below are our results:

As one can see from Figure 3, our implementation of UpBit actually performed only slightly worse for the Delete and Insert functions than the implementation of UCB. In particular, for the Delete function, the read time for UCB was 0.000001 seconds, while the read time for UpBit was 0.00024 seconds. Seeing as these two functions are often difficult to handle with Bitmaps, and UpBit has an additional bit-vector that needs to be addressed, it can be understood why there would be slightly worse performance.

From Figure 3, we can also see that our implementation for Search also was slightly slower. However, in Figure 3, one can observe that our implementation of Update with UpBit was significantly quicker than that for UCB. Since the improvement of the Update function is one of the main focuses of UpBit, it is good to see this result.

Next, we decided to scale up and increase to value of n to 1,000,000 records in our synthetic dataset.

As we can see from Figure 4, our implementations for Delete and Insert with UpBit performed relatively the same as our previous tests. However, as one can observe from Figure 4, the read time for our Search function in UpBit decreased from 0.16 seconds to 0.1 seconds. This may have been a result of how sparse the dataset was. Lastly, we can see that our implementation of the Update

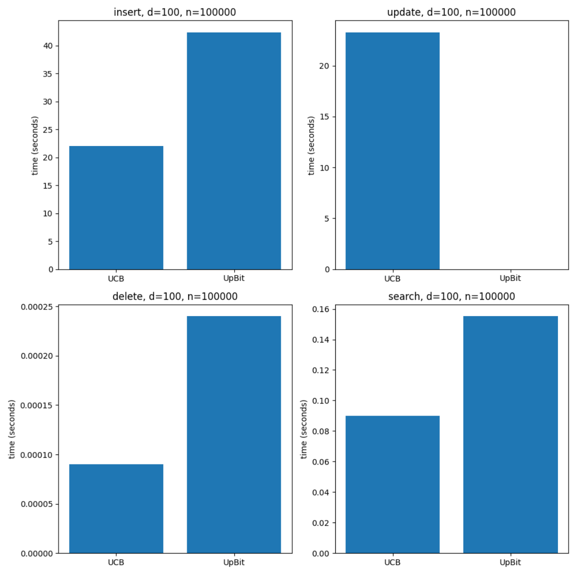

Figure 3: Runtimes for Delete, Insert, Search, and Update - UCB vs. UpBit - n = 100,000

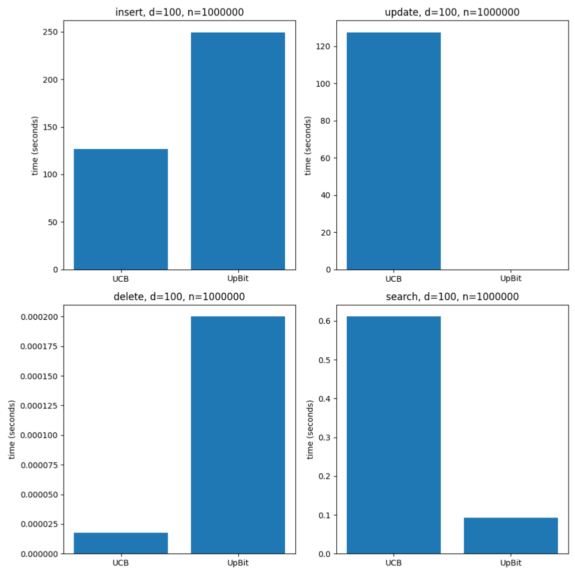

Figure 4: Runtimes for Delete, Insert, Search, and Update - UCB vs. UpBit - n = 1,000,000

function for UpBit continued to perform significantly better than the implementation for UCB.

We also ran the same tests using our datasets generated from zipfian distributions, but due to the challenges we faced in implementing fence pointers and efficient decompression, we did not (and did not expect to) see better results. As described in section 2.4, UpBit would benefit from being used on a zipfian dataset due to the added sparsity inherent in zipfian distributions, which in turn leads to higher compressibility of the majority of the value bit-vectors. Without compression and fence pointers implemented as we hoped, our results on zipfian datasets is more or less the same as our results on uniform datasets.

4 CONCLUSION

From the tests that we conducted, we can conclude that UpBit has significantly quicker read times for both the Update and the Search functions. Meanwhile, our implementation of UpBit performed slightly worse than expected for the Delete and Insert functions. In terms of the Insert function, we believe that the slightly worse read time is due to our struggles with handling the compression aspect of UpBit. Overall, some aspects of our implementation of UpBit did not quite meet the expectations that we had due to challenges that we encountered along the way with compression, however some aspects of our implementation of UpBit, particularly the most important one in the Update function, did relatively live up to the expectations of the UpBit data structure.

REFERENCES

[1] M. Anthanassoulis, Z. Yan, and S. Idreos. UpBit: Scalable In-Memory Updatable Bitmap Indexing. In *SIGMOD '16: Proceedings of the 2016 International Conference on Management of Data*, pages 1319–1332, 2016.