Upbit with Parallelized Merge

Chigullapally Sriharsha
15111013
sharsha@iitk.ac.in

Piyush Kumar
15111026
piyushcs@iitk.ac.in

Akshay Jindal
16111029
akshayaj@iitk.ac.in

ABSTRACT

Bitmap indexes use bit arrays or bit vectors and answer queries by performing bitwise logical operations on these bitmaps, which speed-up query processing. In order to minimize storage space, bitmap indexes are stored in compressed form. Consequently, to perform insert operation, each one of the bitmaps needs to be updated to reflect a new row, which can be prohibitively costly. With regard to this bottleneck, researchers have come up with models like Update Conscious Bitmaps(UCB) [3], which uses Word Aligned Hybrid Encoding(WAH) [3]. UCB brings the concept of Existence Bitvector, which comes with a high probability of becoming uncompressible. Also it requires a lot of pointer maintenance for the updated rows. To overcome the above issues [2] introduces Upbit, which uses Update vectors for each bit vector. This makes updation easy as compared to the delete followed by insert technique used in UCB. It also increases the compressiblity of the bit vector becoming uncompressible by regularly merging the value and update bit vectors. But whenever this merging is done, that particular query is delayed. We propose to parallelize the task of merging value bit vectors and corresponding update bit vectors.

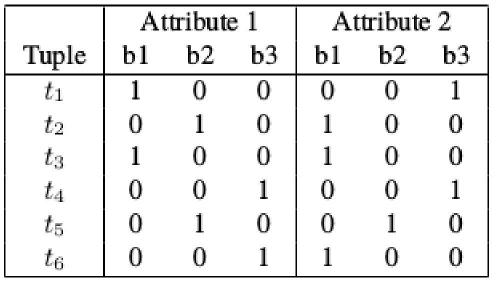
1. INTRODUCTION

1.1 Bitmap Indexes

For each distinct value in the column of an attribute, the bitmap scheme has a bit vector or a bit array. Each bit in a bit vector corresponds to a particular row. Lets take the example of Figure 1. Attribute 1 and 2, each have 3 distinct values b_1, b_2, b_3. Now bit vector b_1 has "1" against t_1. This means tuple t_1 has value b_1 in the column of Attribute 1.

1.2 Compression Techniques

Numerous bitmap compression schemes have been proposed to overcome this problem. The most relevant to our work are the Byte-aligned Bitmap Code (BBC) [1] and the Word-Aligned Hybrid (WAH) code [3].

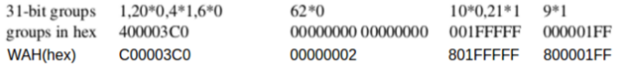


	Attribute 1			Attribute 2		
Tuple	b1	b2	b3	b1	b2	b3
t_1	1	0	0	0	0	1
t_2	0	1	0	1	0	0
t_3	1	0	0	1	0	0
t_4	0	0	1	0	0	1
t_5	0	1	0	0	1	0
t_6	0	0	1	1	0	0

Figure 1: A Bitmap for a table with two attributes and 6 bit vectors
Source: [3]

1.2.1 Word Aligned Hybrid(WAH)

Out of the two compression techniques, the latter one is more CPU efficient, because (as the name signifies) it ensures memory alignment on modern architectures. WAH uses two classes of words: literal words and fill words. It splits bits into words (say 32 bits each), which become the unit of compression. The most significant bit indicates the type of word being dealt with. We have taken 1 for literal and 0 for fill.
Consider 32 bit word a compression unit. So if a compression unit does not have all 0's or all 1's, then it is considered a literal word. Due to this, it cannot be encoded. It is used for direct storage of data. Fill words represent the the number of words (i.e. chunks of 31 bits) we have which are filled with runs of 0 or 1. The second bit contains the value (0 or 1) whose runs are stored in the words.



133 bits	1,20*0,4*1,78*0,30*1			
31-bit groups	1,20*0,4*1,6*0	62*0	10*0,21*1	9*1
groups in hex	400003C0	00000000 00000000	001FFFFF	000001FF
WAH(hex)	C00003C0	00000002	801FFFFF	800001FF

Figure 2: An example of a WAH compressed bit vector
Source:[3]

Figure 2 shows a WAH compressed bit vector representing 133 bits. The first line is the original bit vector. The vector starts with one 1, followed by twenty 0's, four 1's,seventy eight 0's, and thirty 1's. In this example, the compression unit is 32 bits. So our literal word will store 31 data bits and fill word will indicate the number of runs of 31 homogeneous (all 0's or all 1's) data bits. In the second line, 133 bits are broken down into chunks of 31 bits. The third line shows the hexadecimal notation of the chunks formed above. The last line shows the WAH representation. The first group contains both 1's and 0's and hence will be a literal word. The fourth group is less than 31 bits and thus is stored as a literal. The second group contains a multiple of 31 0's and therefore is represented as a fill word (a 0, followed by the 0 fill value, followed by the number of fills in the last 30 bits). The fourth word is the active word. The leftover bits which could not form any group of 31 bits, are grouped in active word. As the leftover bits are less than 31 bits always, so we need to keep a count of the number of valid bits (which actually account as data bits). We are doing this using a number which is written after the active word in square brackets [ ].

1.3 Bitmap Updates:Issue

In-place updates in Bitmap indexes are a costly affair. This is because if we need to update the value for column A in row i from val_1 to val_2 we will first decode the bit vector corresponding to the value val_1 inorder to reach the i^{th} bit, flip the bit and encode the bit vector. This is followed by decoding the bit vector corresponding to the value val_2, setting the i^{th} bit and then finally encoding it. This is very costly operation when the bitvectors are big.

2. RELATED WORK

2.1 Update Conscious Bitmaps

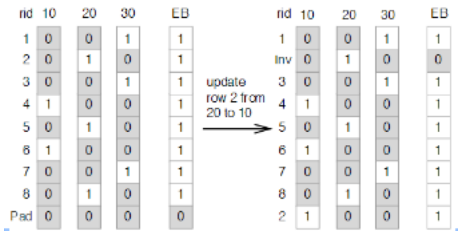
To handle the update problem, a number of approaches have been proposed. In [3] the authors have proposed Update Conscious Bitmaps (UCB). UCB model consists of another bit vector called Existence Bit vector. The purpose of the existence bit vector is to store the valid/invalid status of the bit in the value bit vector such that checking i^{th} tuple for a particular value in UCB would require a bitwise AND between the i^{th} entry of the value bit vector and the i^{th} entry in the existence bit vector. If it evaluates to 1, then only that entry in the value bit vector is considered valid. So in order to delete a value from a particular row, reset the corresponding entry in the existence bit vector, which will invalidate that entry in the value bit vector. Now a tuple with i^{th} entry in original bit vector invalidated and a valid (AND of value and existence bit vector at i^{th} position evaluates to 1) i^{th} entry in the bit vector corresponding to new value will be inserted. In addition to this a mapping is kept between the invalidated entry and the newly inserted entry. Figure 3 demonstrates the above text with an example of updation of 2nd row from value 20 to 10.

2.1.1 *Issues with UCB*

First, each update operation will leave a 0 followed by a 1 in the existence bit vector, due to which it's compressibility will degrade. Apart from this, being a bitmap index, the updation would require repeated decoding and encoding operations. Lastly, updation in UCB brings a overhead of pointer maintenance between the deleted(invalidated) row and newly inserted value. In [2] Upbit is proposed as a solution to the above three issues.

2.2 Upbit

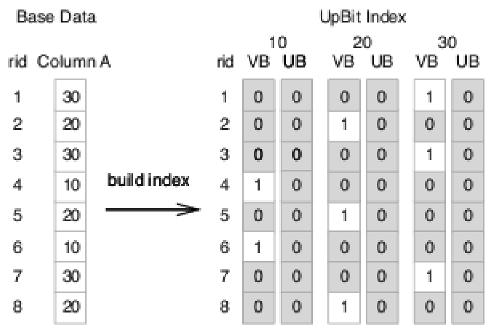
If there are n different bit vectors in our index, then the upbit model raises the count of bit vectors by n. These additional bit vectors are known as Update bit vectors. Each of the n value bit vectors has a corresponding update bit vector. Reading from upbit would require a bitwise XOR between the particular bit of value bit vector and that of update bit vector. Also as the name signifies, each update bit vector holds the updates of each value bit vector. This improves the update performance because of numerous reasons. First, it does not require the pointer maintenance between the invalidated row and newly inserted value. Secondly, now each update bit vector will store the updates corresponding to its value bit vector only, in contrast to Existence bit vector in UCB, which stores information about updates to all columns. This drastically reduces the risk of update bit vector becoming less compressible. In addition to this, the upbit model keeps merging the update and value bit vectors after a threshold T number of updates are done in an update bit vector. (Here on, UB stands for Update bitvector and VB stands for Value bitvector)



rid	10	20	30	EB
1	0	0	1	1
2	0	1	0	1
3	0	0	1	1
4	1	0	0	1
5	0	1	0	1
6	1	0	0	1
7	0	0	1	1
8	0	1	0	1
Pad	0	0	0	0

rid	10	20	30	EB
1	0	0	1	1
Inv	0	1	0	0
3	0	0	1	1
4	1	0	0	1
5	0	1	0	1
6	1	0	0	1
7	0	0	1	1
8	0	1	0	1
2	1	0	0	1

Figure 3: Update Conscious Bitmaps Source:[3]



rid	Column A
1	30
2	20
3	30
4	10
5	20
6	10
7	30
8	20

	10		20		30	
rid	VB	UB	VB	UB	VB	UB
1	0	0	0	0	1	0
2	0	0	1	0	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	0	1	0	0	0
6	1	0	0	0	0	0
7	0	0	0	0	1	0
8	0	0	1	0	0	0

Figure 4: UpBit Source:[2]

2.2.1 *Upbit Contributions*

UpBit model has the following contributions:-

- **Update bit vectors:-**Upbit comes with the idea of an update bit vector corresponding to each value bit vector to accomodate the updates. The benefits of this idea is two fold. First, the update load is distributed among them. Secondly, we can re-initialize the UBs after merging it with VBs, so we can get the compressibility of the update bit vector back after some updates. Since UBs here have high compressiblility, they remain smaller in size and update cost will be less.

- **Fence Pointers**:- The whole idea behind fence pointers is to get the k^{th} bit without decoding each word before its word. For every value bit vector, a fence pointer index is maintained. We can find address of

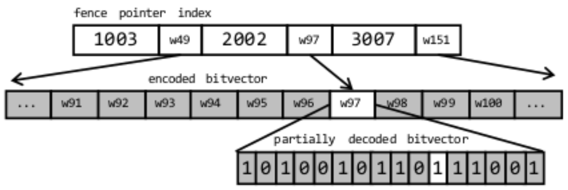


Figure 5: Fence Pointers Source:[2]

nearest possible WAH compressed word using this index, when we are searching for value of particular (row) bit in the bitvector. For eg:- Say we have to find the position of bit number 62073. If we compress our bit vectors in the chunks of 31 bits, then it will be in 62073/31 = 2002^{th} uncompressed word and in the word its offset would be 62073mod31 = 11. Referring to the fence pointer index gives us w97 as the encoded word containing 62073. So, now we can skip reading first 96 words and start from 97^{th} word. In this word, 11^{th} bit in the 97^{th} is the required bit. Fence pointers are maintained for value bit vectors, not update bit vectors, because update bit vectors are very small compared to value bit vectors and are reinitialised regularly to keep them more compressible. Moreover, an update in bit vector would require recalculation of fence pointers. Since updates are stored in Update bit vectors only, fence pointers can be calculated for Value bit vectors. They are recalculated whenever value bit vector is merged (xor-ed) with corresponding update bit vector. Read cost is reduced by fence pointers.

2.2.2 Negatives

UpBit has three major parameters that are exposed for tuning: (i) the UB-VB merging threshold, (ii) the fence pointer granularity, and (iii) the level of parallelism used. Statically defined values deliver robust performance for various workloads. UpBit, however, allows fine tuning of these knobs in the case of a workload that may require different tuning.
Merging of UB and VB happens after a threshold 'T' number of updates are done. So, for every bit vector reaching T number of updates, a read query will be delayed. If merge operations of several bit vectors are done at a time and parllelized, then latencies of further read queries can be eliminated (those queries, at which individual bit vectors reach threshold).

3. OUR TECHNICAL CONTRIBUTION

Along with threshold T, we select a secondary threshold 2T. We would defer the merging of Value bit vector and corresponding Update bit vector of a column even it its number of updates cross threshold T, until all or majority of other bit vectors also reach near T. We will then perform merging of bitvectors of all the columns in parallel using multiprocessing. If number of updates reach 2T, we would perform merge operation without waiting for other bit vectors to reach near T. With this strategy, although one read operation may be delayed a little more, latencies of many further read operations will be eliminated. We proved experimentally that the proposed technique improves the performance of UpBit.

4. IMPLEMENTATION

We implemented the proposed technique using C++ and Python languages. Bitvectors are generated for the data and WAH compression is done as described above. Value bit vectors and corresponding fence pointers are calculated and Update bit vectors are initialized. We implemented insert, delete, getbit, update and read operations. Getbit operation is implemented in two ways. To get value of bit at a particular row in UB, we proceed checking words until it. In case of VB, fencepointers are used. We use mmap package of python to directly reach the nearest possible word. Merging of value bit vector and corresponding update bit vector was implemented in two ways. One is merging of two bit vectors as in Upbit and the other way is parallelized merging as is proposed. The driver program sets up all the bit vectors - both VBs and UBs and fence pointers and starts execution of queries. Queries are read from a file one by one and time latency is recorded. Total time latency for all the queries to be executed is also calculated. A flag is used which tells whether the execution is to be done in upbit method or the proposed method.

4.1 Upbit

A map is maintained which stores the number of updates being performed on each bitvector. Whenever a bit is flipped in a bitvector, the count is increased by 1. When the count reaches threshold then it is merged with corresponding value bit vector. Merging is nothing but the xor operation. Value bit vector is xor-ed with its update bit vector and the result becomes new VB. UB is re-initialized according to new VB. The count of number of updates of UB is reset.

4.2 Parallelized Upbit(proposed method)

In addition to the map, we maintain a variable which stores number of bitvectors which crossed the threshold and not crossed secondary threshold. Let this count be N. When N reaches more than half the total number of UBs, all the bitvectors are merged in parallel. This is acheived using multiprocessing - multiple processes are spawned wherein each process merges a pair of UB and VB. This is implemented in python using the package 'multiprocessing' (it uses API similar to threading). This improves performance in multicore processors since each process can be allocated to a core. Improvement is observed to be more in big size data.

5. EVALUATION

5.1 Experimental Methodology

We evaluated our technique on 8-core system with RAM of 16GB. Processor is Intel core i7-4770 CPU @ 3.40 GHz. Operating System is of 64-bits. We generated custom data with various kinds of skews and queries with various combinations of operations. Improvement in performance is observed and graphs are plotted for time latencies in both techniques. Time for executing all the queries using two techniques is recorded and compared. Graphs are plotted for the same.

Both techniques are compared for various values of thresholds, various secondary thresholds, different number of queries, different combination of queries. This is in anticipation of patterns in the performance improvement. We selected dataset and queries as the following: *Dataset* - the dataset contains more than 140 million rows and two columns. Each

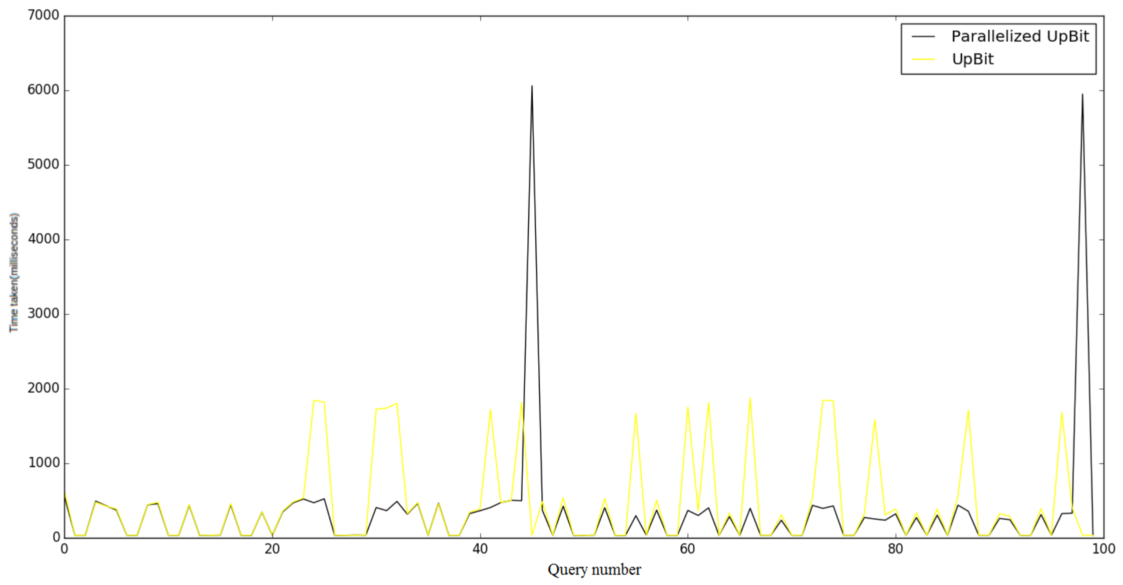


Figure 6: Latencies of queries

entry in a column could be one of eight distinct values. Occurence of each value is equally probable since random sampling was used. Therefore, there are eight VBs for each column. We use a parameter to insert fill words at regular intervals, wherein sampling would not be random (just for these words). To be explicit, every 800 entries generated through random sampling were followed by 80 entries of same value, one among the eight distinct values, so that fill words are generated at that place during compression. *Queries* - We evaluated on various queryfiles, containing varying number of queries - 100 to 500 queries - and recorded latencies of individual queries and overall latency. The columns and bitvectors queried are distributed using random sampling in the queries. We used following combinations of queries: Update - Read, Update - Read - Delete, Update - Read - Delete - Insert. All these operations occur with same probability. For ease of observation we used 8-bit word size for compression - same results would be observed for any word size.

We evaluated changing the number of queries from 100 to 500, threshold from 5 to 20, secondary threshold from 1.5*Threshold to 2.25*Threshold, for four values in each range mentioned. The dataset used is the same for all the cases and is the one that is described above.

5.2 Results

Time latencies of various queries can be observed in figure 6. The yellow curve is for UbBit and black curve is for parallelized UpBit. We can observe peaks in both curves. These occur when merging of VB and UB takes place. The peaks in yellow curve occur at individual mergings in UpBit and the peaks in parallelized UpBit occur when merging of all bit vectors happen in parallel.

One may observe that parallelized method performs better in most of the queries, especially around the peaks of yellow curve i.e, individual mergings in UpBit. Time latency for query when parallel merging happens is higher than peaks in yellow curve. But latencies are less than UpBit until next peak(of black curve). This results in lower overall time latency for the query-set, in Parallelized UpBit. The bigger the query set is, the more is frequency of parallel operations, the more is improvement in performance. The behaviour and pattern observed experimentally is as expected.

Performance improvement is observed in almost every case. In plots in figure 7 and figure 8, queries include only update and read operations. Time latencies are plotted for both Upbit and Parallelized Upbit for thresholds ranging from 5 to 20. In figure 9, first graph is for 300 queries of update-read-delete operations and second graph is for 400 queries of update-insert-read-delete operations.

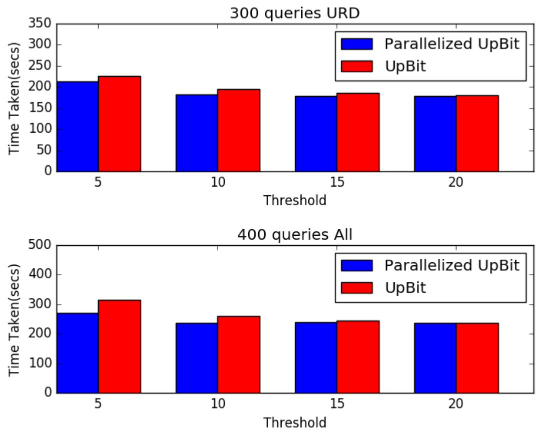


Figure 7: URD and URDI query sets

Following patterns were observed in the overall latency: For the purely randomized data and purely randomized queries optimal secondary threshold was observed to be 1.75 * threshold. We call this 1.75 multiplier. We observe that perfor-

Table 1: Overall latency of 300 URD queries(secs)

Threshold	5	10	15	20
Upbit	225.30	194.36	185.46	180.02
Parallelized Upbit	213.82	182.52	178.93	179.55

mance keeps improving until 1.75 value of multiplier and decreases after that. The pattern may be observed in figure 10. The curves are for different number of queries as shown in the legend.

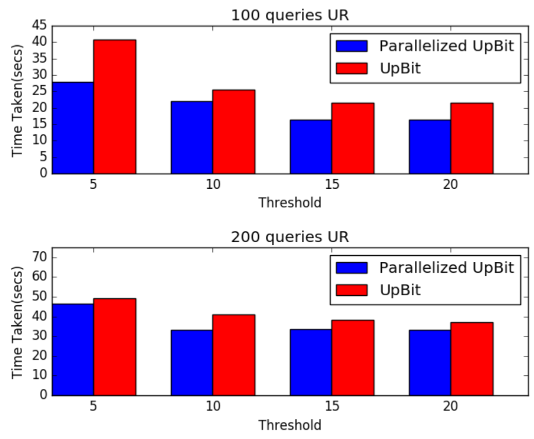


Figure 8: Update-Read queries

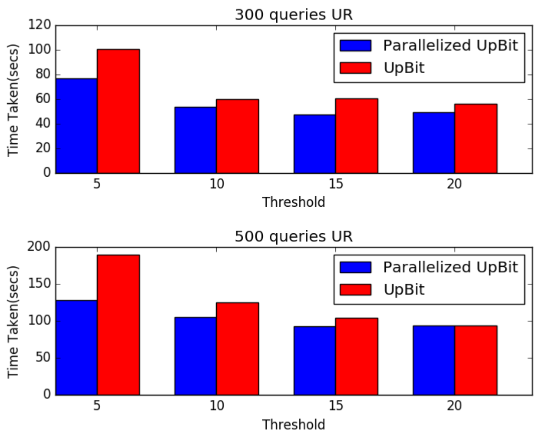


Figure 9: Update read queries

Tables 1 and 2 give latencies observed for various thresholds when multiplier is fixed at 1.75. The first table is for query set of 300 update-read-delete queries and the second one is for query set of 400 update-read-delete-insert queries. We can see that performance improvement is more in bigger query-sets.

It is observed that performance improvement is reduced by delete operations. It can also be seen that performance improvement is higher for lower thresholds. If threshold is high, then frequency of majority of Update Bit vectors reaching threshold is reduced, which reduces frequency of parallelized mergings. It reduces the improvement in performance.

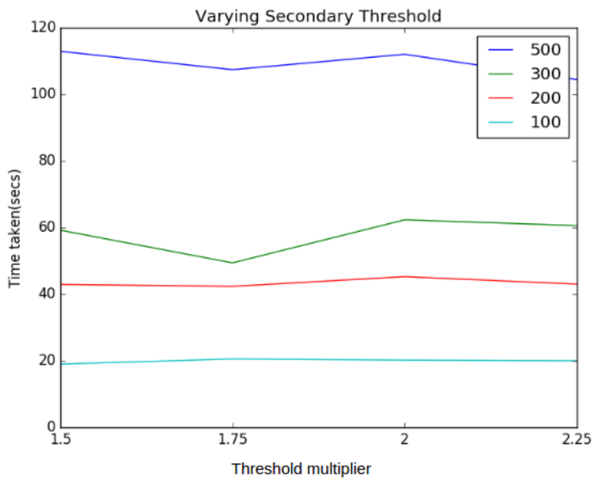


Figure 10: Multiplier vs latency

Table 2: Overall latency of 400 URDI queries(secs)

Threshold	5	10	15	20
Upbit	314.52	259.53	243.63	236.21
Parallelized Upbit	270.82	237.24	239.96	237.72

6. CONCLUSIONS

In this project, it has been shown that parallelizing the merging of bitvectors improves the performance of UpBit. We use a secondary threshold and defer the merging even if a bitvector crosses threshold as long as it is below the secondary threshold. When majority of bitvectors come in such range, we call the merge operation in parallel. It has been observed that for randomized dataset and randomized query set, optimal secondary threshold multiplier is 1.75 i.e, 1.75*Threshold is the optimal secondary threshold. It is also observed that delete operations reduce the performance improvement. It is noted that improvement is better for smaller threshold. The technique proposed works best for bigger datasets and bigger query-sets. It has also been noted that spawning multiple processes had been efficient than spawning multiple threads. Optimal value of secondary threshold depends both on the skew in the data and patterns in the queries. Further work can be done in correlating the value of optimal multiplier with various patterns in queries for data with different kinds of skews.

7. REFERENCES

[1] Topics in data warehousing. *https://view.officeapps.live.com/op/view.aspx?src=http://web.stanford.edu/class/cs345/slides/Lecture9.ppt*, 2004.

[2] M. Athanassoulis, Z. Yan, and S. Idreos. Upbit: Scalable in-memory updatable bitmap indexing. *SIGMOD '16, San Francisco, CA, USA*, 16, June 2016.

[3] G. Canahuate, M. Gibas, and H. Ferhatosmanoglu. Update conscious bitmap indices. *SSDBM*, pages 15–25, 2007.