Update Conscious Bitmap Indices

Guadalupe Canahuate, Michael Gibas, Hakan Ferhatosmanoglu
The Ohio State University
Department of Computer Science
canahuat,gibas,hakan@cse.ohio-state.edu

Abstract

Bitmap indices have been widely used in several domains such as data warehousing and scientific applications due to their efficiency in answering certain query types over large data sets. However, their utilization has been largely limited to read-only data sets or to static snapshots of data due to the cost associated with the update and append of new data. Typically, several bitmaps are associated with each indexed attribute in a table, i.e. one for each attribute value, bin, or range. Each one of these bitmaps needs to be updated to reflect a new, appended row. Since a given table could be represented by hundreds or even thousands of bitmaps, the insertion of a single record can be prohibitively costly. In order to transfer the fast query response times offered by bitmap indices to dynamic database domains, we propose an update conscious bitmap index that provides a mechanism to quickly update bitmaps to reflect dynamic database changes. For an insert operation only the bitmaps that represent the values being inserted need to be updated. We formalize the insert and delete operations of the proposed technique and provide a cost model for bitmap updates. We compare the update conscious bitmaps to traditional bitmaps in terms of storage space, update performance, and query execution time.

1 Introduction

Bitmap indices are widely used in data warehouses and scientific applications to efficiently query large-scale data repositories. They are successfully implemented in commercial Database Management Systems, e.g., Oracle [1, 2], IBM [14], Informix [7, 12], Sybase [6, 8], and have found many other applications such as visualization of scientific data [10, 17, 18]. Bitmap indices can provide very efficient performance for point and range queries thanks to fast hardware supported bit-wise operations over the bitmaps.

Traditionally, the use of bitmaps has been restricted to read-only or read-mostly (data warehouses) data environments. Even though commercial RDBMS, such as Oracle, support updates to bitmap indexed tables, the general concensus is that updating bitmapped indexes take a lot of resources and time and the recommendation is to use nightly batch updates, drop the index, apply the changes, and rebuild the bitmaps [3, 5, 11].

The reason that the use of bitmaps has been confined to largely static domains is that bitmaps were not originally designed to handle updates efficiently. The main goal of a bitmap index is to execute queries fast and efficiently for large volumes of data. Within a bitmap index, a single bitmap is a bit vector that identifies the tuples that have attribute-values that match the value, category, or range associated with the bitmap. The entire bitmap index is made up of multiple bitmaps for each indexed attribute. The main reason why updates are so costly is that all the bitmaps, often hundreds of them, need to be updated when a new row is inserted. Insertion is the most common update operation.

As a sample application, consider High Energy Physics (HEP) experiments consisting of accelerating sub-atomic particles to nearly the speed of light and forcing their collision. Sequences of events notable for physicists are stored with all the details. The number of events stored and analyzed in one year is on the order of several millions and the number of attributes per each event is above 100 [16]. Bitmap indices provide efficient query support for such data [17]. New events are periodically appended to the database as more experiments are performed. Bitmap update is done periodically in batches. This means that the new data is not accessible from the index structure until the next scheduled update of the bitmap index is done. In order to update the bitmap index, the new data needs to be encoded into the bitmap, compressed, and appended to the existing bitmaps.

For the most commonly used bitmap encoding, where bits are only set for tuples that match the bitmap value, and in the case where updates were performed in real time, the insertion process would need to access all the bitmaps for the table, add a '0' for those bitmaps that are not relevant for the value(s) inserted and a '1' on the relevant bitmaps.

Considering this domain, the number of bitmaps per table makes this insertion process prohibitively costly.

However, if it were possible to propagate database changes to only those relevant bitmaps and still maintain consistency between index and database, then the cost of bitmap updates would be greatly reduced. By updating bitmaps efficiently, we can expand the domain of applications for which bitmap indices can be useful. We could essentially make the same number of changes to the bitmap index structure as would be necessary for an equivalent set of B+-tree indices over the attributes and maintain index-data consistency.

In this paper, we propose a bitmap index design for equality encoded bitmap indices for which new records can be added by updating a single bitmap per indexed attribute. Using compressed bitmaps, we add an extra word as the last word of each bitmap. This extra word is a compressed large run of 0's that pads the number of rows encoded in the bitmap to a fixed number, much larger than the actual number of rows in the table. When we insert a new row, we set the bit in the corresponding position of the relevant bitmaps, and compress the last words accordingly. Deletions are handled by maintaining a compressed existence bitmap (EB) [13]. When a record is deleted, the appropriate bit is set to 0 in the EB. Query execution is performed in the same manner as traditional bitmaps with one additional AND bit operation with the existence bitmap. Updates of existing data are handled by performing a delete operation followed by an insertion.

These changes expand the applicable domain of bitmap indices. For record insertions, the number of changes that need to be made to the index structure are reduced by a factor equal to the average cardinality of the indexed attributes of the dataset. These changes make bitmap indices more feasible for dynamic database applications and makes them more attractive for high cardinality data domains.

The rest of this paper is organized as follows. Section 2 provides some background on bitmap indices and bitmap compression. Section 3 presents the update conscious bitmap approach. Section 4 provides the cost model for bitmap updates and query execution. Section 5 shows the experimental results and the performance comparison with traditional bitmap indices. We conclude in Section 6.

2 Background

Bitmap tables are a special type of bit matrices. Each binary row in the bitmap table represents one tuple in the database. The bitmap columns are produced by quantizing the attributes in the database into categories or bins. Each tuple in the database is then encoded based on which bin each attribute value falls into.

	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. Simple bitmap example for a table with two attributes and three bins per attribute.

For the simple bitmap encoding (also called equality encoding) [13], if a value falls into a bin, this bin is marked "1", otherwise "0". Since a value can only fall into a single bin, only a *single* "1" can exist for each row of each attribute. After binning, the whole database is converted into a huge 0-1 bitmap, where rows correspond to tuples and columns correspond to bins. Figure 1 shows an example of the equality encoded bitmap using a table with two attributes, each partitioned into three bins. The first tuple t_1 falls into the first bin in attribute 1, and the third bin in attribute 2.

Bitmap indices can provide very efficient performance for point and range queries thanks to fast bit-wise operations over the bitmaps, which are efficiently supported by hardware.

With equality encoded bitmaps a point query is executed by ANDing together the bit vectors corresponding to the values specified in the search key. For example, finding the data points that correspond to a query where Attribute 1 is equal to 3 and Attribute 2 is equal to 5 is only a matter of ANDing the two bitmaps together. Equality Encoded Bitmaps are optimal for point queries [4]. Range queries are executed by first ORing together all bit vectors specified by each range in the search key and then ANDing the answers together. If the query range for an attribute queried includes more than half of the cardinality then we execute the query by taking the complement of the ORed bitmaps that are not included in the range query.

No matter which bitmap encoding we use, the bitmap index table is a 0-1 table. This table needs to be compressed to be effective on a large database. General purpose text compression techniques are clearly not suitable for this purpose since they significantly reduce the efficiency of queries [9, 20]. Specialized bitmap compression schemes have been proposed to overcome this problem. These schemes are based on run-length encoding, i.e., they replace runs of 0's or 1's in the columns by a single instance of the symbol and a run count. These methods not only compress the data but also enable fast bitwise logical operations, which translates into faster query processing.

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)	400003C0	80000002	001FFFFF	000001FF

Figure 2. An example of a WAH compressed bit vector.

Run length encoding [15] can therefore be used over every column to compress the data when long runs of "0" or "1" blocks become available. Pure run length encoding is not a good strategy because of its accessing inefficiency. The two most popular compression techniques for bitmaps are the Byte-aligned Bitmap Code (BBC) [1] and the Word-Aligned Hybrid (WAH) code [21]. Unlike traditional run length encoding, these schemes mix run length encoding and direct storage. BBC stores the compressed data in Bytes while WAH stores it in Words. WAH is more CPU efficient because it only has two types of words: literal words and fill words. In our examples it is the most significant bit that indicates the type of word we are dealing with. Let w denote the number of bits in a word, the lower $(w - 1)$ bits of a literal word contain the bit values from the bitmap. If the word is a fill, then the second most significant bit is the fill bit, and the remaining $(w-2)$ bits store the fill length. WAH imposes the word-alignment requirement on the fills. This requirement is key to ensure that logical operations only access words.

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, we assume 32 bit words. Under this assumption, each literal word stores 31 bits from the bitmap, and each fill word represents a multiple of 31 bits. The second line in Figure 2 shows how the bit vector is divided into 31-bit groups, and the third line shows the hexadecimal representation of the groups.

The last line shows the values of WAH words. Since the first and third groups do not contain greater than a multiple of 31 of a single bit value, they are represented as literal words (a 0 followed by the actual bit representation of the group). 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 1, followed by the 0 fill value, followed by the number of fills in the last 30 bits). The first three words are regular words, two literal words and one fill word. The fill word 80000002 indicates a 0-fill of two-word long (containing 62 consecutive zero bits). The fourth word is the active word, and it stores the last few bits that could not be stored in a regular word. Another word with the value 9, not shown, is needed to store the number of useful bits in the active word. Logical operations are performed over the compressed bitmaps resulting in another compressed bitmap.

3 Update Conscious Bitmaps (UCB)

3.1 The General Idea

Let us consider the update operations over a traditional bitmap indexed attribute A of table T. We denote as $B = \{b_1, b_2, ..., b_C\}$ the set of bitmaps over the domain of A and as n the number of tuples in T. The simple uncompressed bitmap encoding would have n bits in each bitmap. To insert a new tuple with value corresponding to b_i, one would need to add a '1' at the end of bitmap b_i and a '0' at the end of all other bitmaps.

To delete a record, we use the Existence Bitmap (EB) [13]. Originally, the EB was proposed to handle non-existent rows. The bits corresponding to non-existing tuples are set to zero in all bitmaps. Therefore, the EB needed to be used (ANDed together with another bitmap) after a NOT operation to make sure that only existing rows are reported as answers.

In the uncompressed bitmaps, to update a row we need to change two bitmaps. The old value bitmap to unset the corresponding bit and the new value bitmap to set the corresponding bit to 1. However, given the amount of space they require, it is not feasible to store the bitmaps in an uncompressed form. Several compression techniques have been proposed for bitmap indices. While these techniques effectively reduce the space required by the bitmaps, they have the disadvantage that there is no longer a direct mapping between the bit position in a bitmap and the position of the row in the table. In this case, for example, updates become even more expensive because we need to scan (decompress) the bitmap in order to locate the corresponding bit in the two bitmaps. This is the reason why we handle updates by performing a delete followed by an insert operation.

We modify the current equality encoded bitmaps in the following ways:

- To each bitmap we add a number of non-set bits, called pad-bits, that would be used for the new tuples. By padding the bitmaps with zeros we can insert a new tuple by setting the corresponding bit only in the bitmap that corresponds to the inserted value.
- We maintain a total bit counter, TBC, for the number of non-pad bits currently in the bitmaps.
- We also pad the Existence Bitmap (EB) with non-set bits to indicate non-existing rows. As in the original

approach, the EB is ANDed with the resulting bitmap after a NOT operation to make sure that only existing rows are reported as answers.

In the following section we detail the update conscious bitmap index approach. From here on, we refer to the equality encoded WAH-compressed bitmaps as original or traditional bitmaps.

3.2 Bitmap Creation

We based our implementation of the Update Conscious Bitmaps on equality encoded bitmaps using Word-Aligned Hybrid (WAH) compression. WAH has been proven to provide faster query execution (2X-20X faster) than Byte-aligned Bitmap Code (BBC) with a smaller compression ratio (compressed bitmaps are typically 40-60% larger) [19]. WAH is currently considered the start-of-the-art in bitmap compression due to its simplicity and efficiency.

Bitmaps are created as usual and compressed using WAH. For the rest of this paper the word size is set to $w = 32$ bits. The number of words encoded in each bitmap is $T_0 = \lceil n/(w-1) \rceil$, where n is the number of rows in the table. We divide by $w-1$ because with WAH compression, one bit is reserved to indicate the type of word, whether a literal or fill word. The compressed size of each bitmap can be smaller than T_0 words, however, a total of T_0 words are encoded in each bitmap. We then add a final word, called a pad-word, which is the fill representation of words of 0s. With one pad-word, each bitmap encodes $T_1 = 2^{w-2} - 1$ words. The pad-word would be the 0-fill word of $T_2 = T_1 - T_0$ words. In case more bits are needed, one could add one or more pad-words to encode T_1 more words for each pad-word. For the rest of paper we assume that T_1 words are enough to encode all the rows in the table, i.e. at any given time we have less than $T_1 * (w-1)$ rows in the indexed table (including the rows that would be inserted in the future).

Figure 3 shows an example of a compressed bit vector for both the original WAH compressed bitmap (3rd line) and the update conscious bitmap (4th line). It is worth noting that we changed the encoding of the active word in the WAH implementation so incoming bits of zeros do not require a bitmap update. In the original implementation the active word would be represented as 000001FF, in which case, adding a zero would update this word to be 000003FE. However, by encoding the rows from the most significant bits, the active word 7FC00000 in the UCB would remain unchanged when we add a zero. The last word in the update conscious bitmap (BFFFFFFA), represents the pad-word, i.e. a 0-fill word for a run of $33,285,996,358$ zeros. The total bit counter (TBC), not shown in this example, is set to 133.

3.3 Insertions

Let us consider the insertion of a new row where attribute A is appended with the value encoded by bitmap b_i. In this case, only b_i needs to be accessed to perform the insertion. The basic idea is to use the TBC and the number of words in the pad word to decide how many zeros we need to put before the new 1 in the current bitmap. We also update the number of words encoded by the pad word so the total number of words in the bitmap remains the same. We change the EB to include the new row and increment the TBC by 1.

We have two cases when inserting a new row for attribute A depending on whether the inserted value is encoded by an existing bitmap. In the first case, when the value is already encoded by a bitmap b_i, we compute the number of words encoded by the non-pad words of b_i, $T_0 = T_1 - T_2$. Note that even when T_0 is initially the same for all the bitmaps, as we insert new rows, T_0 remains the same for all the unchanged bitmaps. If all the bitmaps were updated on an insertion then the number of words encoded by the bitmaps would be $T_0' = \lceil \text{TBC}/(w-1) \rceil$ and T_0 would always be equal to T_0'. In the case when $T_0 = T_0'$ we only need to change the bit corresponding to position (TBC mod $(w-1)$) in the active word. In the case when $T_0 < T_0'$, which is the case when many insertions have been performed in other bitmaps, we add a 0-fill word to make $T_0 = T_0' - 1$, add an active word with the bit corresponding to position (TBC mod $(w-1)$) set to 1, and update $T_2 = T_1 - T_0$. The details of the implementation are given in Algorithm 1. To simplify the pseudocode, we introduce some variable names and their corresponding values in Table 1. The extra steps found in the pseudocode which are not described above are necessary to guarantee that the resulting bitmap is correctly compressed in WAH format.

As an illustration, consider the following example. In Figure 4 we present the last 7 compressed words for the bitmap index over the attribute *sex*, which only have two possible values M or F, for a table with 200 rows. Both bitmaps have 6 regular words, 1 active word and 1 pad-word (BFFFFFF8). The TBC is 200, and the number of words needed is 7 ($T_0' = \lceil 200/31 \rceil$). Now consider the insertion of a new row where $sex = M$. TBC is updated to 201. We only access the bitmap corresponding $sex = M$ and update the active word (W_1) to be 3FCF0000 (since the number of words needed remains to be 7, i.e. $T_0 = T_0'$). After 16 more insertions of rows where $sex = M$, TBC = 217, and $W_1 =$ 3FCFFFFF. When a new insertion comes where $sex = M$, TBC $=$ 218 and the words needed are now 8 ($T_0 < T_0'$), so we need to add one word and subtract 1 from the total words in W_0. Since (218 mod 31 $=$ 1), the new word is 40000000 and W_0 is updated to be BFFFFFF7.

Now consider the insertion of a new row where $sex = F$. TBC $=$ 219 and the number of words needed is 8. Since

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	
Orig (hex)	400003C0	80000002	001FFFFF	000001FF	
UCB (hex)	400003C0	80000002	001FFFFF	7FC00000	BFFFFFFA
	Literal Word	*Fill Word*	*Literal Word*	*Last Word*	*Pad Word*

Figure 3. WAH compressed bit vector for original bitmaps and Update Conscious Bitmaps (UCB).

Symbol	Description
W_0	The pad word in a UCB
W_1	The last non-pad word in a UCB
W_2	The previous to the last non-pad word in UCB
W_i.value or just W_i	The literal value of the i^{th} word
W_i.isFill	Indicates whether the i^{th} word is a fill or not
W_i.fill	The fill value of the i^{th} word
W_i.nWords	The number of words encoded by the i^{th} fill word
w	The word size, we set it to 32
TBC	The global total bit counter
$word[i]$	The literal word with only the i^{th} bit set.
T_1	The total number of words encoded in each bitmap, including the pad word, in our examples $T_1 = 2^{w-1} - 1$
T_2	The number of words encoded by the pad word, $T_2 = W_0.nWords$
T_0	The number of words encoded by the non-pad words in this bitmap, $T_0 = T_1 - T_2$
T_0'	The number of literal words needed to encode TBC rows, $\lceil \text{TBC}/(w-1) \rceil$

Table 1. Notation list and their meaning.

there are only 7 words encoded in the bitmap ($T_0 < T_0'$), we need to add one word to the bitmap and update the word count in W_0. Since (219 mod 31 = 2), the new word is 20000000 and W_0 is updated to be BFFFFFF7.

For the second and special case of insertions, when the value being inserted is not encoded by an already existing bitmap, i.e. the new row is the first row with such a value, we do not need to access any of the bitmaps for attribute A, we just add a new bitmap with at most two non-pad words. The first word corresponds to the fill words for (almost) all the previous rows and the second one is the active word with a 1 in the position corresponding to (TBC mod $(w-1)$). We also add as many pad-words as needed, e.g. 1 in our experiments.

3.4 Deletions

We propose to handle deletes by unsetting the corresponding bit in the EB and leaving the rest of the bitmaps unchanged. Since the bit corresponding to the deleted row remains set in some bitmaps, when we execute the query we need to AND the EB with the final result to ensure that no deleted row is retrieved as an answer.

4 Bitmap Index Cost Model

The cost to modify a set of indices associated with a database update is the cost to propagate all database changes to the set of indices such that all possible operations over the indices exactly reflect the state of current database. This section describes the bitmap modifications that need to occur so that the indices are up to date after a record insertion for both traditional bitmaps and our update conscious bitmaps.

In our final update cost estimates, we assume that reads and writes dominate the cost of updates compared to the bit comparisons and bit operations that need to be performed. We also assume that I/O operations have a consistent cost, but they may in fact differ based on disk locality. The experimental results will show to what extent these assumptions are valid.

4.1 Insertions

4.1.1 Traditional Bitmaps

For traditional bitmaps, when a new row is inserted *every* bitmap for each indexed attribute needs to be updated in order to maintain consistency with the database.

Assuming the bitmap index is not in memory and bitmap B_i is a pointer to a binary file that contains the compressed bitmap, the insertion routine involves reading at most the last two words of the bitmap file, making the appropriate modifications, and writing the updated file to disk. All the bitmaps, including the existence bitmap (EB), need to be accessed and updated.

		Bitmap						
	Value	W_6	W_5	W_4	W_3	W_2	W_1	W_0
Initially	M	7C3B773F	6D75B7FF	38F427F8	7E95B5FA	6FFFE3AF	3FCE0000	BFFFFFF8
(200 rows)	F	03C488C0	128A4800	470BD807	016A4A05	10001C50	40300000	BFFFFFF8
Insert 1 row	M	7C3B773F	6D75B7FF	38F427F8	7E95B5FA	6FFFE3AF	**3FCF0000**	BFFFFFF8
(sex = M)	F	03C488C0	128A4800	470BD807	016A4A05	10001C50	40300000	BFFFFFF8
Insert 16 rows	M	7C3B773F	6D75B7FF	38F427F8	7E95B5FA	6FFFE3AF	**3FCFFFFF**	BFFFFFF8
(sex = M)	F	03C488C0	128A4800	470BD807	016A4A05	10001C50	40300000	BFFFFFF8
Insert 1 row	M	6D75B7FF	38F427F8	7E95B5FA	6FFFE3AF	3FCFFFFF	***40000000***	**BFFFFFF7**
(sex = M)	F	03C488C0	128A4800	470BD807	016A4A05	10001C50	40300000	BFFFFFF8
Insert 1 row	M	6D75B7FF	38F427F8	7E95B5FA	6FFFE3AF	3FCFFFFF	40000000	BFFFFFF7
(sex = F)	F	128A4800	470BD807	016A4A05	10001C50	40300000	***20000000***	**BFFFFFF7**

Figure 4. Insertion operations for Update Conscious Bitmap Index over Attribute sex.

```
UpdateBitmap (Bitmap Index $B$, Total Bit Counter TBC)

1:  read $W_2, W_1$, and $W_0$ from $B$
2:  if $T_0 = T_0'$
3:      $W_1 = W_1 \mid word$[TBC mod $(w-1)$]
4:      if (TBC mod $(w-1)$ = 0) and ($W_1$ is all 1s)
5:          if $W_2$.isFill and $W_2$.fill=1
6:              $W_2$.nWords++
7:              delete $W_1$
8:          else if $W_2$ is all 1s
9:              $W_2$.isFill = true, $W_2$.fill=1, $W_2$.nWords=2
10:             delete $W_1$
11: else if $T_0 < T_0'$
12:     $W_0$.nWords=$(T_1 - T_0')$
13:     nWords = $T_0' - T_0$
14:     if nWords = 1
15:         add $word$[TBC mod $(w-1)$] before $W_0$
16:     else if nWords = 2
17:         add words 0 and $word$[TBC mod $(w-1)$]
              before $W_0$
18:     else
19:         $W$.isFill = true, $W$.fill=0, $W$.nWords=(nWords-1)
20:         add words W and $word$[TBC mod $(w-1)$]
              before $W_0$
```

Algorithm 1: Insertion of a new row - Update Conscious Bitmap Indices.

The cost to perform this update is:

$$t_{ins} = (tbm + 1) * (rt + ut + wt)$$

where t_{ins} is the estimated time to update the complete bitmap index upon a record insert, tbm is the total number of bitmaps excluding the EB, rt is the time to read the last words of the bitmap file, ut is the time to update the last words of the bitmap to be consistent with the record insertion, and wt is the time to write the bitmap file.

4.1.2 Update Conscious Bitmap

For update conscious bitmaps, when a new row is inserted, only *one* bitmap for each indexed attribute needs to be updated to maintain consistency with the database.

With the same assumptions as with the traditional bitmaps, insertions using UCB involve reading at most the last three words of the bitmap file, making the appropriate modifications, and writing the updated file to disk. One bitmap per indexed attribute, as well as the existence bitmap (EB), needs to be accessed and updated.

The estimated cost associated with a record insert is:

$$t_{ins} = (atts + 1) * (rt + ut + wt)$$

where $atts$ is the number of attributes.

Assuming that the read, write, and bit manipulation operations require similar times for the traditional and update conscious bitmaps, the speedup for the insertion of a new row is:

$$speedup = (tbm + 1)/(atts + 1)$$

4.2 Deletions

4.2.1 Traditional Bitmaps

The brute force schema for handling row deletes in traditional bitmaps would be to remove the bit corresponding to the deleted row in all bitmaps. However, this is clearly very inefficient since it involves decompressing all the bitmaps and altering the (w-1)-bit groups created to compress the bitmaps. A more clever way would be to use the EB to flag the deleted row as inexisting. There would then be two alternatives. The first one is to keep the rest of the bitmaps unchanged in which case we would need to exclude the deleted rows during query execution by AND-ing the final bitmap with the EB. The second one is to change the corresponding bits to 0 in the relevant bitmaps in which case query execution is not affected since deleted rows are 0s in all bitmaps. In order to maintain the query execution performance of traditional bitmaps, we chose the second alternative for traditional bitmaps. When a row is deleted, the bitmaps corresponding to the values of each indexed attribute in the row need to be updated, i.e. the set bit needs to be unset, and then, the bit corresponding to the deleted row in the existence bitmap needs to be unset as well. Now, recall that since the bitmaps are compressed, the bitmaps need to be scanned (decompressed) and the change in the bit value could mean increasing or decreasing the size of the bitmap by at most three words.

The estimated cost to perform a delete is:

$$t_{del} = atts * (rt_{bc} + dt + ut + ct + wt_{bc})$$

where t_{del} is the estimated time required to update the entire index upon a record deletion, rt_{bc} is the time it takes to read the bitmap file, dt is the time required to decompress/scan the bitmaps to locate the relevant bit, ct is the time to update the file, and wt_{bc} is the time needed to write the file to disk. These times will vary depending on the compressed length of the bitmap.

Note that one could choose the first alternative to handle deletes with traditional bitmaps, in which case the deletion cost and the query execution cost would be the same for both, traditional and update-conscious bitmaps.

4.2.2 Update Conscious Bitmaps

For the update conscious bitmaps, we decided to only modify the existence bitmap and handle the extra set bits during query execution by ANDing the final results with the existence bitmap in case there are deletions. Assuming the existence bitmap is not in memory, the operation requires reading the existence bitmap, scanning it, changing the bit associated with the deleted row to '0', compressing the bitmap, and writing it to disk. The estimated cost is:

$$t_{del} = rt_{bc} + dt + ut + ct + wt_{bc}$$

Assuming similar times associated with the compressed existence bitmap and the average bitmap size, the estimated speedup for deleting a row is:

$$speedup = atts$$

4.3 Query Execution

For both approaches, when a bitmap is negated during query execution the final answer needs to be ANDed with the EB in order to prune nonexistent rows. However, for update conscious bitmaps, this additional operation needs to be performed for every query if there are deletions.

The following analysis refers to queries where no bitmap is negated and there are deletions, i.e. the update conscious bitmaps require one bit-wise operation more than the traditional bitmaps. Note that in other cases, the UCB will exhibit the same query execution performance as the traditional bitmap. For simplicity, and without loss of generality, we will express the queries in terms of the number of bitmaps that need to be accessed to answer the query.

For point queries, the number of bitmaps that need to be accessed is equal to the number of attributes queried.

For range queries, the number of bitmaps that need to be accessed depends on the range of each attribute queried. However, it is bound by half of the bitmaps for each attribute.

4.3.1 Traditional Bitmaps

For traditional bitmaps, the time required to execute a query over b bitmaps is:

$$t_{ex} = (b - 1) * t_{bo}$$

where t_{ex} is the estimated time to execute a query, b is the number of bitmaps that need to be accessed to answer the query, and t_{bo} is the time to perform a bitwise operation. The time t_{bo} will vary depending on the size of the bitmaps.

4.3.2 Update Conscious Bitmaps

For update conscious bitmaps, query processing is identical as before with the addition that the final answer obtained from the bitwise operations is ANDed with the existence bitmap. The time to perform a query is:

$$t_{ex} = b * t_{bo}$$

And the speedup (slowdown) is:

$$speedup = (b - 1)/b$$

5 Experimental Results

5.1 Experimental Setup

We performed experiments in order to measure the impact on bitmap update time, query execution time, and index storage for update conscious bitmaps compared to traditional bitmaps.

Four data sets were used in the experiments. Each data set contains 2 million rows and attributes of varying cardinalities. The HEP data set is a set of real bitmap data generated from High Energy Physics experiments with 12 attributes. Each attribute has between 2 and 12 bins, for a total of 122 bitmaps. The UR data set is a synthetically generated data set of uniformly distributed random data. The Z1 and Z2 data sets are synthetically generated data sets

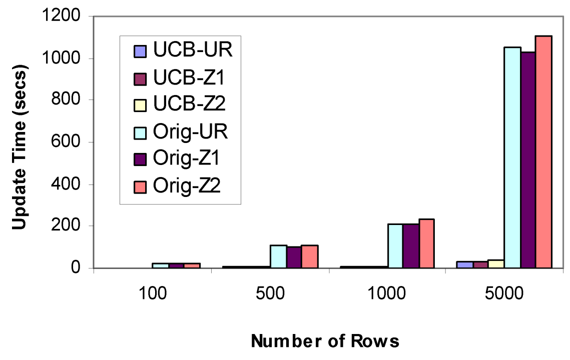

Figure 5. Index Update Time vs. Number of Rows Inserted.

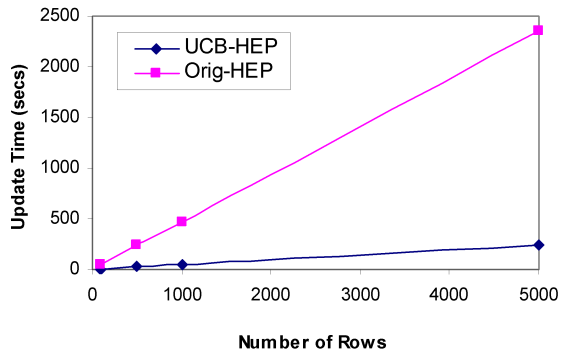

Figure 6. Index Update Time vs. Number of Rows Inserted.

following the zipf distribution with parameters 1 and 2 respectively.

Experiments are performed using a Windows XP Professional Operating System on a Pentium 4 2.26 GHz processor with 512 MB of RAM. A Java implementation of traditional bitmaps and update conscious bitmaps was developed to measure the relative differences between techniques and to compare to predicted results. Each bitmap is stored in a separate binary file. The bitmap update time includes time to read the relevant bitmap files, update the last word(s) of the file, and write the updated file back to disk. Query execution time is the time to run point queries using the appropriate bitmap query execution technique. For HEP data, the query points are randomly selected from the last 100,000 rows which were not used during bitmap creation. For synthetic data, the point queries correspond to randomly generated data points following the same distribution as the data set.

5.2 Update Time

5.2.1 Insertion Time versus Rows Appended

Figure 5 shows the index update time required as the number of records appended is varied for the synthetic data sets. Each append is done individually, not in a batch fashion. In this experiment we append one attribute with cardinality of 50. For both the original bitmap representation and the UCBs, the update of the Existence Bitmap is included in this experiment. Both techniques exhibit linear performance with respect to the number of rows appended. The data sets, which vary in compressibility, have little effect on the append times. The ratio between the update times for the original technique and the UCBs is close to the predicted value. Using our experimental setup, on the average, one insertion is done in 7 ms for UCB as opposed to 212 ms for traditional bitmaps.

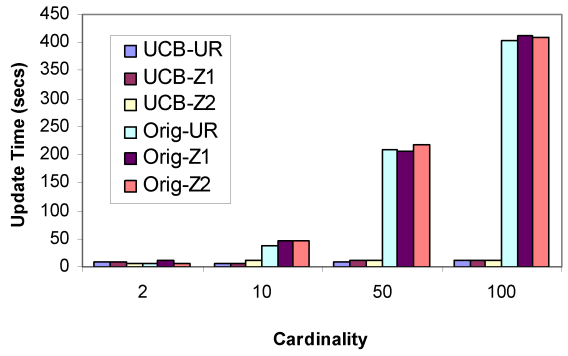

Figure 7. Index Update Time vs. Cardinality of the Attribute Indexed.

Figure 6 shows index update time for the real HEP data set. This graph shows the linear performance of insertions and includes the time to append all 12 attributes of an inserted record. Appends to the real data show the same pattern of append time as the synthetic data. To insert one new row, the average time is 50 ms using UCB as opposed to 470 ms using traditional bitmaps. The speedup is 9.46 (123/13).

5.2.2 Insertion Time versus Attribute Cardinality

Figure 7 shows the record append time for the synthetic data as attribute cardinality varies. For each stated cardinality, $1,000$ rows of a single attribute are appended. The UCBs shows constant update time with respect to attribute cardinality while the original bitmaps exhibit update times that linearly increase with respect to attribute cardinality.

Figure 8 shows the relationship between append time and attribute cardinality for the real HEP data set. It shows similar performance to the synthetic data sets.

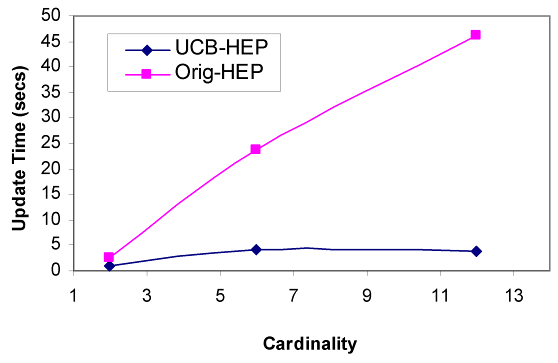

Figure 8. Index Update Time vs. Cardinality of the Attribute Indexed.

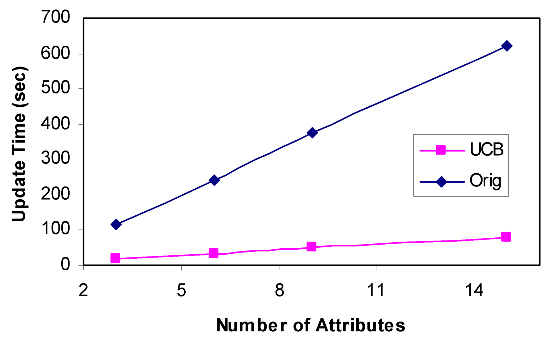

Figure 9. Index Update Time vs. Number of Attributes Indexed.

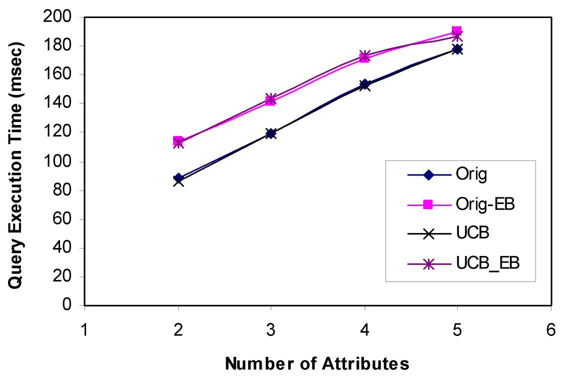

Figure 10. Point Query Execution Time vs. Number of Attributes Queried.

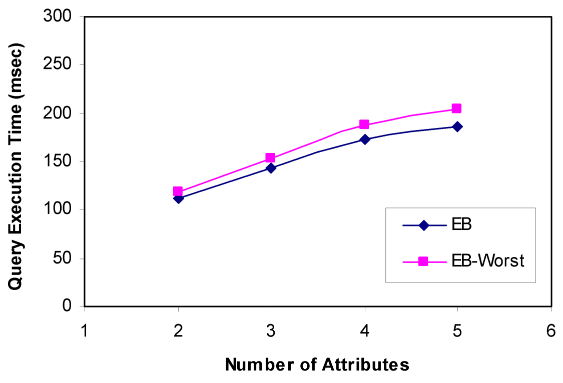

Figure 11. Point Query Execution Time for best case EB and worst case EB.

5.2.3 Insertion Time versus Number of Attributes

Figure 9 shows how insertion time is affected by the number of attributes indexed. Insertion time measures the time needed to append $1,000$ rows with a varying number of attributes. The attributes involved are equally taken from the UR, Z1, and Z2 data sets and all have cardinality 10. As expected, the performance for both the original and UCBs is linear with respect to the number of indexed attributes and the ratio between original bitmap and UCB append time is as predicted. The real HEP data performance is similar and is not shown here due to space considerations.

5.3 Query Execution Time

Figure 10 shows how the average query execution time compares using original bitmaps and UCB. Times are provided for performing 100 point queries using the indicated bitmap type as the number of attributes involved in the query vary. Times are provided for both the original bitmaps and UCB with and without an extra AND operation with a compressed existence bitmap. UCB and original bitmap query execution time is nearly identical when both use existence bitmap and when both do not.

5.4 Worst Case Existence Bitmap

As deletes occur over time, the existence bitmap becomes less compressible (the 0's associated with deleted records interrupt runs of 1's). Since query execution time over bitmaps is dependent on the compressed length of those bitmaps, queries that involve the existence bitmap can become more time consuming as the existence bitmap increases in size. Figure 11 demonstrates this effect and the magnitude of the effect. We compare the results using a best case (all 1's existence bitmap) to a worst case existence bitmap (alternating 1's and 0's, no WAH-compression possible). The graph displays results for the Z2 dataset using cardinality 10 attributes. The results show that a worst case existence bitmap adds less than 20 ms in query execution time for this experiment. Other data sets provide similar results.

5.5 Compressed Size

Considering the case when we only add one pad-word, there is no overhead in terms of storage space when the last word of the compressed bitmap is zero, since this word would be subsumed by the pad-word. If the last word of the compressed bitmap has a one, then we are increasing the compressed size by 1 word. In general, if we add p pad-words, the overhead would be between $|B| * (p-1) * w$ and $|B| * p * w$ bits, where $|B|$ is the number of bitmaps for the given table. For example, if we add one 32-bit pad-word to each bitmap and the number of bitmaps for the table is 1,000 bitmaps, in the worse case we are increasing the size of the bitmap indices by 4 KB. This is clearly a small overhead when compared with the overall benefits achieved using this technique.

6 Conclusion

In this paper, we have introduced an update conscious bitmap index structure. In order to insert a record, only those bitmaps associated with attribute-value pairs that match the new tuple need to be updated. This means that for each attribute appends are performed in constant time regardless of the attribute cardinality. This contrasts to traditional bitmaps which would require all bitmaps to be updated in order to maintain the index up-to-date. The speedup of bitmap index insertion using an update conscious bitmap index is on the order of the average indexed cardinality. In addition, query performance using UCB is as efficient as traditional bitmaps when the updates are in the form of appends. The storage overhead is minimal since the compressed size of each bitmap is increased by at most one word. Even in the case when there have been so many deletes and modifications of existing records that the Existence Bitmap becomes incompressible, there is only a minor impact on query execution time. Furthermore, there is nothing in the update conscious bitmap architecture that prevents periodic rebuilds of the bitmap index to mitigate this effect.

Update conscious bitmaps are particularly a good choice for domains where new data is added to the data set at a rate which allows for updates to take place, and it is desirable to keep the index up-to-date in real time. This approach expands the applicable realm of bitmap indices to include a wide range of dynamic data domains, such as scientific applications where data is appended as experiments are conducted. To the best of our knowledge, this is the first work that addresses the update of bitmap indices and proposes a solution to handle appends of new data efficiently.

References

[1] G. Antoshenkov. Byte-aligned bitmap compression. In *Data Compression Conference*, Nashua, NH, 1995. Oracle Corp.

[2] G. Antoshenkov and M. Ziauddin. Query processing and optimization in oracle rdb. *The VLDB Journal*, 1996.

[3] D. K. Burleson. *Oracle Tuning: The Definitive Reference*. Rampant TechPress, April 2006.

[4] C.-Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *SIGMOD*, pages 215–226, 1999.

[5] B. Consulting. Oracle bitmap index techniques. http://www.dba-oracle.com/oracle_tips_bitmapped_indexes.htm.

[6] H. Edelstein. Faster data warehouses. Information Week, December 1995.

[7] I. Inc. Informix decision support indexing for the enterprise data warehouse. http://www.informix.com/informix/corpinfo/-zines/whiteidx.htm.

[8] S. Inc. *Sybase IQ Indexes*, chapter 5: Sybase IQ Release 11.2 Collection. Sybase Inc., March 1997.

[9] T. Johnson. Performance measurement of compressed bitmap indices. In *VLDB*, pages 278–289, 1999.

[10] J. C. K. Wu, W. Koegler and A. Shoshani. Using bitmap index for interactive exploration of large datasets. *In Proceedings of SSDBM*, 2003.

[11] J. Lewis. Understanding bitmap indexes. http://www.dbazine.com/oracle/or-articles/jlewis3, 2006.

[12] P. O'Neil. Informix and indexing support for data warehouses, 1997.

[13] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD*, pages 38–49, 1997.

[14] M. Ramakrishna. In *Indexing Goes a New Direction.*, volume 2, page 70, 1999.

[15] D. Salomon. *Data Compression 2nd edition*. Springer Verlag, New York, 2000.

[16] SciDAC. Scientific data management center. http://sdm.lbl.gov/sdmcenter/, 2002.

[17] K. Stockinger, J. Shalf, W. Bethel, and K. Wu. Dex: Increasing the capability of scientific data analysis pipelines by using efficient bitmap indices to accelerate scientific visualization. *In Proceedings of SSDBM*, 2005.

[18] K. Stockinger and K. Wu. Improved searching for spatial features in spatio-temporal data. In *Technical Report. Lawrence Berkeley National Laboratory. Paper LBNL-56376.* http://repositories.cdlib.org/lbnl/LBNL-56376, September 2004.

[19] K. Stockinger, K. Wu, and A. Shoshani. Strategies for processing ad hoc queries on large data warehouses. In *Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP*, pages 72–79, 2002.

[20] K. Wu, E. Otoo, and A. Shoshani. A performance comparison of bitmap indexes. In *Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management*, pages 559–561, Atlanta, Georgia, November 2001.

[21] K. Wu, E. J. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *SSDBM*, pages 99–108, Edinburgh, Scotland, UK, July 2002.