Dynamic Data Organization for Bitmap Indices*

Tan Apaydin
Department of CSE
The Ohio State University
apaydin@cse.ohio-state.edu

Guadalupe Canahuate
Department of CSE
The Ohio State University
canahuat@cse.ohio-state.edu

Hakan Ferhatosmanoglu
Department of CSE
The Ohio State University
hakan@cse.ohio-state.edu

Ali Şaman Tosun
Department of CS
University of Texas
at San Antonio
tosun@cs.utsa.edu

ABSTRACT

Bitmap indices have been successfully used in scientific databases and data warehouses. Run-length encoding is commonly used to generate smaller size bitmaps that do not require explicit decompression for query processing. For static data sets, compression is shown to be greatly improved by data reordering techniques that generate longer and fewer runs. However, these data reorganization methods are not applicable to dynamic and very large data sets because of their significant overhead. In this paper, we present a dynamic data structure and algorithm for organizing bitmap indices for better compression and query processing performance. Our scheme enforces a compression rate close to the optimum for a target ordering of the data which results in fast query response time. For our experiments, we use Gray code ordering as the tuple ordering strategy. However, the proposed scheme efficiently works for any desired ordering strategy. Experimental results show that the proposed framework provides better compression and query execution time than the traditional approaches.

1. INTRODUCTION

Bitmap indices have been successfully implemented in commercial Database Management Systems such as Oracle [2, 3], Informix [9, 18], and have been used by many applications, e.g., data warehouses (OLAP), statistical and scientific databases [12, 21, 22]. Point and range queries are efficiently answered with bitwise logical operations directly supported by computer hardware. Although uncompressed bitmap indices involving a small number of rows and columns may work efficiently, large scale data sets require bitmap compression to reduce the index size while maintaining the advantage of fast bitwise logical operations [1, 2, 4, 11, 23]. The general approach is to utilize compression schemes that are based on *run-length* encoding[1]. The advantage of run-length encoding is that the compressed bitmaps do not require explicit decompression during query processing. The two popular run-length encoding based compression techniques are the Byte-aligned Bitmap Code (BBC) [2] and the Word-Aligned Hybrid (WAH) code [23].

Run-length encoding compression performs better with sorted data, therefore reordering techniques have been successfully applied to significantly improve the performance of run-length encoding based bitmap compression [10, 20]. However, finding an optimal data order to minimize the compressed size of a boolean table has been shown to be NP-hard through a reduction to Traveling Salesperson Problem (TSP) [10]. As an efficient TSP heuristic, Gray codes are shown to increase the lengths of runs in bitmap columns and improve the compression performance [20]. Gray code based techniques achieve comparable compression performance to more expensive TSP-based approaches while running considerably faster.

In typical scientific and data warehousing applications, massive volumes of data are frequently generated through experiments, measurements, or computer simulations. The updates are typically appends rather than deletions or value-changes. To make these massive data collections manageable for human analysts, efficient mechanisms to support the appends are vital. A typical application where bitmaps are widely utilized is a data warehouse, where facts are aggregated using several dimensions. As more transactions occur, new information is periodically inserted. Since bitmap updates are periodically done in batch mode, the new data is not available until the next scheduled update.

Using the current bitmap structures, a new tuple is inserted to the end of the index. As the ratio of the appended tuples increases, the overall compression efficiency is limited by the insertion order of the new tuples. In order to improve the compression efficiency, one could reorder the data periodically. However, the reordering schemes are known to be effective when the data fits in the main memory. For large data sets, scalable techniques are needed to handle insertions. Another alternative to maintain the data order and the compression performance would be to insert the new records to the appropriate location within the existing data. However, current bitmap structures do not efficiently support this approach due to the way data is organized. For instance, for a bitmap index that is compressed with run-length encoding, a single update (a bit flip from zero to one or vice versa) on a *run* will cause the run to be interrupted and more runs need to be created to compress the index. This would require the index to be reorganized since we need

*This work is partially supported by US NSF Grants IIS-0546713 and CCF-0702728.

Infoscale2008 June 4-6, 2008, Vico Equense, Napoli, Italy.

[1]Run-length encoding is the process of replacing repeated occurrences of a symbol by a single instance of the symbol and a count.

	Attribute I		Attribute II		
Tuple	b_1=f	b_2=m	b_1=1	b_2=2	b_3=3
t_1 = (f, 3)	1	0	0	0	1
t_2 = (m, 2)	0	1	0	1	0
t_3 = (f, 1)	1	0	1	0	0
t_4 = (f, 3)	1	0	0	0	1
t_5 = (m, 1)	0	1	1	0	0

Table 1: Simple bitmap for two attributes with 2 and 3 bins

original bits	1×1, 20×0, 3×1, 79×0, 21×1			
31-bit groups	1×1, 20×0, 3×1, 7×0	31×0	31×0	10×0, 21×1
groups in hex	40000380	00000000	00000000	001FFFFF
WAH(hex)	40000380	80000002		001FFFFF

Table 2: WAH compression for 124-bit vector

to shift all the following runs. In general, the recommendation is to make batch updates, i.e., drop the index, apply the changes and rebuild the index afterwards [5, 7, 15]. Obviously this approach consumes a lot of resources. Therefore, the traditional bitmap indices are accepted as an effective method only for static databases.

In this paper, we present a dynamic bitmap index scheme based on a structured partitioning that allows on-the-fly partial data reordering. By utilizing a dynamic structure, our goal is to improve the bitmap insertions further by keeping a given data order and also by targeting better compression and query execution performances. This way the applicability of bitmaps, along with the reordering methods, will be expanded to more domains. The proposed scheme efficiently works for any desired ordering technique. We also conduct an analysis of Gray code and lexicographical orderings.

The rest of the paper is organized as follows. In Section 2 we briefly cover the background on bitmaps. Section 3 provides the underlying technical motivation of our scheme. We discuss the main framework of our approach in Section 4, and Section 5 shows the experimental results. Finally, we conclude in Section 6.

2. BACKGROUND AND PRELIMINARIES

For an equality encoded bitmap index, data is partitioned into several bins, where the number of bins per each attribute could vary. 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. Table 1 shows an example with two attributes, which are quantized into 2 and 3 bins, respectively. The first tuple t_1 falls into the first bin of Attribute I, and the third bin of Attribute II. Note that after binning we can treat each tuple as a binary number, e.g., t_1 = 10001 and t_2 = 01010.

Bitmaps are compressed using run-length encoders not only to decrease the bitmap index size but also to enable efficient query execution performance while running the queries over the compressed bitmaps. The following subsections briefly describe the techniques for bitmap compression and updates on bitmaps.

2.1 Run-Length Based Compression

An earlier run-length encoding based bitmap compression scheme, BBC [2], stores the compressed data in bytes, therefore the computer memory is processed in a way that is not word-aligned, i.e., one byte at a time during most operations. Analysis shows that, for BBC, the time spent on bitwise logical operations is dominated by the time spent in CPU rather than in reading bitmaps from disk [24]. On a modern computer, accessing a byte takes the same amount of time as accessing a word, which is the main property that allowed WAH, a word-based compression scheme, to be designed in a *CPU-friendly* fashion. WAH is efficient since the bitwise operations can be performed on words without extracting individual bytes. There are two types of WAH words: *literal words* and *fill words*. In our implementation, it is the most significant bit that indicates the type of the word. 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. Table 2 depicts an example of WAH compression. The first row includes the original bits in a column of a bitmap table. In the last row, first and third words are the *literal* words, and the second is a *fill* word. WAH imposes the word-alignment requirement on the fills, which is the key to ensure that logical operations only access words. A comparison between WAH and BBC indicates that bit operations over the compressed WAH bitmap file are faster than BBC (2-20 times) [23] while BBC gives slightly better compression ratios. In this paper, we utilized WAH as the compression technique. However, our scheme efficiently works for any run-length based compression architecture, including BBC.

2.2 Gray Code Order (GCO)

The original Gray code order (GCO) is a reordering technique such that two adjacent binary numbers differ only by one bit. For instance (000, 001, 011, 010, 110, 111, 101, 100) is a binary Gray code. One can achieve GCO recursively as follows: i) Let S = $(s_1, s_2, ..., s_n)$ be a Gray code. ii) First write S forwards and then append the same code S by writing it backwards, so that we have $(s_1, s_2, ..., s_n, s_n, ..., s_2, s_1)$. iii) Append 0 at the beginning of the first n numbers, and 1 at the beginning of the last n numbers. For instance, take the Gray code (0, 1). Write it forwards and backwards, and we get: (0, 1, 1, 0). Then we add 0's and 1's to get: (00, 01, 11, 10). This approach is also referred as the *reflection* technique.

For a bitmap table, let $B(t_x, i)$ be the i^{th} bit of d-bit binary tuple t_x. The *Hamming distance* between two binary tuples t_x and t_y is given as follows: $H(t_x, t_y) = \sum_{i=1}^{d} |B(t_x, i) - B(t_y, i)|$. For example, the hamming distance between (11111) and (11001) is 2. Note that, for a GCO produced with the reflection technique, $H(t_i, t_{i+1}) = 1$.

For a boolean matrix with d columns, we define the *rank* of a d-bit binary tuple as the position of the tuple in GCO of the matrix. In Figure 1(b), e.g., the rank of t_3 is 0 and the rank of t_2 is 3.

As described in the previous section, run length encoding based schemes pack consecutive same-value-bits into runs, which does the actual job of compression, e.g., fill words for WAH. GCO technique has been proposed to improve the compression of runs in bitmaps [20]. Figure 1 illustrates the basic idea behind GCO. On the left matrix, there are 20 runs (6 on the first column and 5, 5, 4 on the following columns) whereas on the right matrix, reordering the tuples reduces the number of runs to 14. Figure 2 depicts the effect of running the GCO algorithm. Black and white segments represent the runs of ones and zeros respectively. On the left is the numerical (or lexicographical) order of a boolean matrix with 4 columns. GCO of the same matrix is presented on the right. As the figure illustrates, the aim of GCO is to produce longer and thus fewer runs than the lexicographic order.

The essential idea of traditional reordering techniques in batch periods is to keep the data in order so that the total compression and the query execution performance are improved. However, out-of-core and online algorithms are needed for these methods to be applicable in real-life settings where the data sets typically do not fit into main memory, and the data is updated mostly through appends.

$$\begin{matrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{matrix} \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \qquad \begin{matrix} t_3 \\ t_1 \\ t_5 \\ t_2 \\ t_6 \\ t_4 \end{matrix} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

(a) Original Table (b) Reordered Table

Figure 1: Example of tuple reordering

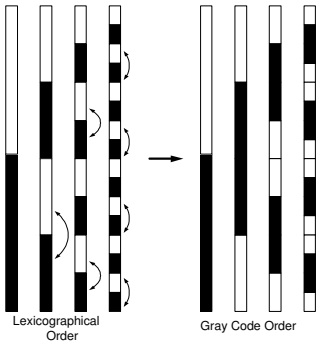


Figure 2: Gray Code Ordering

2.3 Bitmap Updates

A recent work concerning the efficient bitmap index updates is presented in [6]. In this study, each bitmap (or bin) is expanded by adding a single synthetic fill-word to the end, namely *0-fill-pad-word*, which can compress huge amounts of literal-words whose values are all zeros (similar to the second word at the last row in Table 2). For equality encoded bitmaps, a traditional row insertion for an attribute adds a 1 to the bin for which the new data value falls into. The remaining bins of the attribute are expanded with a 0. In [6], for an attribute, the idea is to only touch the bin that will receive a 1 and update the very last word that was synthetically added, and not to touch the other bins since they already have *0-fill-pad-words* at the end. This technique speeds up the updates on bitmap indices significantly, however tuple reordering is not taken into account. As with traditional bitmap encodings, this approach also appends the new tuples to the end of the indices, therefore both compression and query execution performance suffer from the order of insertions.

3. TECHNICAL MOTIVATION

Although the Bitmap GCO algorithm is proven to be effective for static databases, tuple insertions are not handled by the technique. Tuple appends to the end of the matrix will not obey the GCO and therefore, the matrix needs to be reorganized again to maintain the improved compression and query execution efficiency. An approach to preserve the GCO in a bitmap index against the tuple insertions might be as follows. When a new tuple arrives find the GCO *rank* of the row, assume r_i, and insert it in between the tuples whose ranks are r_{i-1} and r_{i+1}. For instance assume we want to insert t_7 = (1100) to the ordered matrix in Figure 1(b). Naturally, the proper place would be in between t_5 and t_2, in which case the total number of runs will still be 14 after the insertion. However bitmaps are stored and processed in column-wise compressed forms. Therefore the solution would be inefficient since one needs to decompress the bitmap first, then shift the bits to make room for inserting the new tuple in between the existing ones and then compress it again.

We aim to achieve an architecture-conscious data organization that effectively utilizes the main memory. We propose a dynamic bitmap scheme based on a horizontal partitioning of the bitmap table such that each partition can be managed within the main memory without any I/Os. To test its feasibility, we implemented a basic version of this idea where we uniformly partitioned a small subset of a data set and appended the remainder of the set, tuple by tuple, into the closest partition. In this simple approach, the new tuple is compared against the last rows (tuples) of the partitions, i.e., the smaller the hamming distance, the closer the two tuples are. We present the results in Table 3 where the values are the total number of words after WAH compression[2]. Table 3 reveals that even a simple technique of appending to different partitions instead of a single data set results in better compression, i.e., the number of WAH words drops to two third, compared to the brute-force update approaches where we always append to the end.

	Traditional Appends	Partition Appends
HEP data set	3,180,845	2,486,141

Table 3: Number of WAH words

3.1 Notation

In order to ease the presentation for the remainder of the paper, we provide the summary of utilized notation in Table 4.

Symbol	*Meaning*
GN(r)	GCO codeword with rank r
LN(r)	Lexicographic codeword with rank r
H(x,y)	Hamming distance of x and y
B(x,i)	i^{th} bit of x
G_d^k	Average hamming distance of d-bit GCO codewords whose ranks differ by k positions
L_d^k	Average hamming distance of d-bit lexicographic codewords whose ranks differ by k positions

Table 4: Notation

Next, we provide fundamental results for GCO and lexicographic order. Proposed scheme is motivated by these theoretical results that support the claim that GCO achieves better compression than lexicographic order. In addition, the results quantify the difference between the two orders.

3.2 Average Distance

We now investigate the tuple spacing for a table that is generated using the GCO reflection technique. This is basically the average hamming distance of the codewords whose ranks differ by a fixed number. The larger the fixed number is, the further apart the tuples are in the data set, and thus the larger the average hamming distance is, and this leads to worse compression performances. We derive the recursive formulation for both GCO and lexicographic code and prove the properties of these codes using the recursive formulation.

3.2.1 Gray Code Order

Let G_d^k denote the average hamming distance of all the d-bit Gray codes whose ranks differ by k, which is defined as follows

$$G_d^k = \frac{1}{2^d} \sum_{r=0}^{2^d} H(GN(r), GN((r+k) \bmod 2^d)) \qquad (1)$$

Following theorem shows the recursive formulation of G_d^k. Since GCO is defined recursively, following expression results in a recursive function.

[2]Detailed information about the data sets are presented in the experiments section.

THEOREM 3.1. *The values of G_d^k can be recursively computed as follows*

$$G_d^m = \begin{cases} G_{d-1}^{2k} & : \quad m = 4k \\ G_{d-1}^{2k+1} + 1 & : \quad m = 4k+2 \\ \frac{1}{2}G_{d-1}^k + \frac{1}{2}G_{d-1}^{k+1} + \frac{1}{2} & : \quad m = 2k+1 \end{cases} \quad (2)$$

PROOF. Let $G_{d,i}^k$ denote the contribution of bit i, which is formally defined as

$$G_{d,i}^k = \frac{1}{2^d}\sum_{r=0}^{2^d} |B(GN(r),i) - B(GN((r+k) \bmod 2^d),i)| \quad (3)$$

Using $G_{d,i}^k$ we can represent G_d^k as follows

$$G_d^k = \sum_{i=0}^{d-1} G_{d,i}^k = \sum_{i=0}^{d-2} G_{d,i}^k + G_{d,d-1}^k \quad (4)$$

Let T_d^k denote $\sum_{i=0}^{d-2} G_{d,i}^k$ in the above summation. T_d^k is the average difference in ranks for GCO excluding the last bit. For the 3-bit code $U = \{000, 001, 011, 010, 110, 111, 101, 100\}$, T_d^k excludes the last bit and considers the code $V = \{00, 00, 01, 01, 11, 11, 10, 10\}$. In codes considered for T_d^k every codeword is repeated twice. Using the same notation as G_d^k we have the following properties for T_d^k

$$T_d^m = \begin{cases} G_{d-1}^k & : \quad m = 2k \\ \frac{1}{2}G_{d-1}^k + \frac{1}{2}G_{d-1}^{k+1} & : \quad m = 2k+1 \end{cases} \quad (5)$$

Now let us look at $G_{d,d-1}^k$ which is the contribution of the last bit. We have

$$G_{d,d-1}^m = \begin{cases} 0 & : \quad m = 4k \\ 1 & : \quad m = 4k+2 \\ \frac{1}{2} & : \quad m = 2k+1 \end{cases} \quad (6)$$

Combining results for $G_{d,d-1}$ and T_d we get

$$G_d^m = \begin{cases} G_{d-1}^{2k} & : \quad m = 4k \\ G_{d-1}^{2k+1} + 1 & : \quad m = 4k+2 \\ \frac{1}{2}G_{d-1}^k + \frac{1}{2}G_{d-1}^{k+1} + \frac{1}{2} & : \quad m = 2k+1 \end{cases} \quad (7)$$

For the base case, $G_1^{2l} = 0$ and $G_1^{2l+1} = 1$. □

3.2.2 Lexicographic Order

Let L_d^k denote the average hamming distance of all the d-bit binary codes sorted in lexicographic order whose ranks differ by k. This is formally defined as follows

$$L_d^k = \frac{1}{2^d}\sum_{r=0}^{2^d} H(LN(r), LN((r+k) \bmod 2^d)) \quad (8)$$

Similar to G_d^k we can derive a recursive formulation for L_d^k. Having recursive formulations for both of them makes it easier to compare the values. Following theorem shows how to compute L_d^k.

THEOREM 3.2. *The values of L_d^k can be recursively computed as follows*

$$L_d^m = \begin{cases} L_{d-1}^k & : \quad m = 2k \\ \frac{1}{2}L_{d-1}^k + \frac{1}{2}L_{d-1}^{k+1} + 1 & : \quad m = 2k+1 \end{cases} \quad (9)$$

PROOF. Let $L_{d,i}^k$ denote the contribution of bit i, which is formally defined as

$$L_{d,i}^k = \frac{1}{2^d}\sum_{r=0}^{2^d} |B(LN(r,i) - B(LN((r+k) \bmod 2^d),i)| \quad (10)$$

Using $L_{d,i}^k$ we can represent L_d^k as follows

$$L_d^k = \sum_{i=0}^{d-1} L_{d,i}^k = \sum_{i=0}^{d-2} L_{d,i}^k + L_{d,d-1}^k \quad (11)$$

Let M_d^k denote $\sum_{i=0}^{d-2} L_{d,i}^k$ in the above summation. M_d^k is the average difference in ranks for the lexicographic order excluding the last bit. For the 3-bit code $U = \{000, 001, 010, 011, 100, 101, 110, 111\}$, M_d^k excludes the last bit and considers the code $V = \{00, 00, 01, 01, 10, 10, 11, 11\}$. In codes considered for M_d^k every codeword is repeated twice. Using the same notation as L_d^k we have the following properties for M_d^k

$$M_d^m = \begin{cases} L_{d-1}^k & : \quad m = 2k \\ \frac{1}{2}L_{d-1}^k + \frac{1}{2}L_{d-1}^{k+1} & : \quad m = 2k+1 \end{cases} \quad (12)$$

Now lets look at $L_{d,d-1}^k$ which is the contribution of the last bit. We have

$$L_{d,d-1}^m = \begin{cases} 0 & : \quad m = 2k \\ 1 & : \quad m = 2k+1 \end{cases} \quad (13)$$

Combining results for $L_{d,d-1}$ and M_d we get

$$L_d^m = \begin{cases} L_{d-1}^k & : \quad m = 2k \\ \frac{1}{2}L_{d-1}^k + \frac{1}{2}L_{d-1}^{k+1} + 1 & : \quad m = 2k+1 \end{cases} \quad (14)$$

For the base case, $L_1^{2l} = 0$ and $L_1^{2l+1} = 1$. □

3.2.3 Behavior for large d

In this section, we show that both G_d^m and L_d^m are nondecreasing functions of d, and for very large d GCO is better than lexicographic order for small values of m.

Following theorem shows that for fixed m, when d is increased G_{d+1}^m increases or stays the same.

THEOREM 3.3. $\forall m, d \geq 1, G_{d+1}^m \geq G_d^m$

PROOF. By induction

- *Base Case:* $G_2^m \geq G_1^m$.
  - *Case a:* $m = 4k$
    $G_2^m = G_1^{2k} \geq 0 = G_1^m$
  - *Case b:* $m = 4k+2$
    $G_2^m = G_1^{2k+1} + 1 \geq 1 \geq 0 = G_1^m$
  - *Case c:* $m = 2k+1$
    $G_2^m = \frac{1}{2}G_1^k + \frac{1}{2}G_1^{k+1} + \frac{1}{2} \geq 1 = G_1^m$
- *Inductive Hypothesis:* Assume $G_d^m \geq G_{d-1}^m$
- *Inductive Step:* Prove $G_{d+1}^m \geq G_d^m$
  - *Case a:* $m = 4k$
    $G_{d+1}^m = G_d^{2k} \geq G_{d-1}^{2k} = G_d^m$
  - *Case b:* $m = 4k+2$
    $G_{d+1}^m = G_d^{2k+1} + 1 \geq G_{d-1}^{2k+1} + 1 = G_d^m$
  - *Case c:* $m = 2k+1$
    $G_{d+1}^m = \frac{1}{2}G_d^k + \frac{1}{2}G_d^{k+1} + \frac{1}{2} \geq \frac{1}{2}G_{d-1}^k + \frac{1}{2}G_{d-1}^{k+1} + \frac{1}{2} = G_d^m$

□

Following theorem shows that for fixed m, when d is increased L_{d+1}^m increases or stays the same.

THEOREM 3.4. $\forall m, d \geq 1, L_{d+1}^m \geq L_d^m$

PROOF. By induction

- *Base Case:* $L_2^m \geq L_1^m$.

m	1	2	3	4	5	6	7	8
Lexicographic	2	2	3	2	$\frac{7}{2}$	3	$\frac{7}{2}$	2
GCO	1	2	2	2	$\frac{5}{2}$	3	$\frac{5}{2}$	2

Table 5: Average Distance in limit

  - *Case a:* $m = 2k$
    $L_2^m = L_1^k \geq 0 = L_1^m$
  - *Case b:* $m = 2k + 1$
    $L_2^m = \frac{1}{2}L_1^k + \frac{1}{2}L_1^{k+1} + 1 \geq 1 = L_1^m$

- *Inductive Hypothesis:* Assume $L_d^m \geq L_{d-1}^m$
- *Inductive Step:* Prove $L_{d+1}^m \geq L_d^m$
  - *Case a:* $m = 2k$
    $L_{d+1}^m = L_d^k \geq L_{d-1}^k = L_d^m$
  - *Case b:* $m = 2k + 1$
    $L_{d+1}^m = \frac{1}{2}L_d^k + \frac{1}{2}L_d^{k+1} + 1 \geq \frac{1}{2}L_{d-1}^k + \frac{1}{2}L_{d-1}^{k+1} + 1 = L_d^m$

□

Following theorem summarizes the behavior of average distance in the limit (for very large d). Similar properties for other values of m can be derived using the recursive formulation of G_d^m and L_d^m.

THEOREM 3.5. *Following properties hold*

- $m = 1$: $G_d^1 = 1$ *and* $\lim_{d\to\infty} L_d^1 = 2$
- $m = 2^n$: $G_d^{2^n} = 2$ *and* $\lim_{d\to\infty} L_d^{2^n} = 2$
- $m = 3$: $G_d^3 = 2$ *and* $\lim_{d\to\infty} L_d^3 = 3$
- $m = 5$: $G_d^3 = \frac{5}{2}$ *and* $\lim_{d\to\infty} L_d^5 = \frac{7}{2}$
- $m = 6$: $G_d^6 = 3$ *and* $\lim_{d\to\infty} L_d^6 = 3$
- $m = 7$: $G_d^7 = \frac{5}{2}$ *and* $\lim_{d\to\infty} L_d^7 = \frac{7}{2}$

Results in the limit are summarized in Table 5. As can be seen in the table for large d, GCO results in smaller or equal average distance compared to lexicographic order. A consequence of Theorem 3.5 is that one needs to apply GCO to as large data as possible since that is when it achieves its best performance gain. In fact, the best case is the global GCO considering the whole data set with all the 2^d possible number of tuples. A best of worlds method would preserve the global GCO (achieved by the off-line algorithm), but would use partitioning and work on local sets of data for efficiency and scalability. This constitutes the basis of our partitioning based solution where the boundaries of the partitions are decided considering the global GCO. The global GCO is achieved using a local ordering method. The details of the proposed method are described next.

4. DYNAMIC BITMAP SCHEME

The incremental organization of data is a well-known challenge in large-scale databases. Without a dynamic data organization, the data is usually kept in the order tuples are appended. An effective database solution is to utilize a dense index that dictates the data order. However, the insertions or updates of arbitrary bits in bitmaps are expensive enough to be simply avoided. Therefore, bitmaps are usually tailored for read-only environments. The common suggestion for bitmap updates is to perform a complete reorganization, i.e., drop the index, apply the changes and rebuild the complete index. We want to avoid reconstructing the entire bitmap index since it requires reading, reordering and building the index. At each rebuilding session, as the number of rows increases, the recreation time also increases. If the data set does not fit into main memory, one can apply the rebuilding process partially, and then utilize a merging mechanism, e.g., external sorting [13], to minimize the sorting cost. However, this does not reduce the complexity of the overall rebuilding process. The proposed technique is more efficient since it does not require the reorganization of the entire structure. Data is mapped to the tuned-size partitions and only local operations are performed. In this section, we first discuss our proposed framework that serves as a dynamic data organization scheme for bitmap indices. We then present our GCO Rank Algorithm that operates on a given bitmap tuple. Finally, we present the additional advantages and uses of the proposed technique.

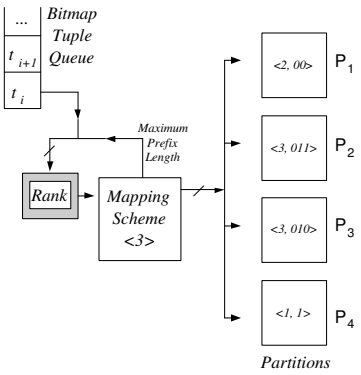


Figure 3: Main Framework

4.1 Dynamic Structure and Mapping Framework

The Dynamic Bitmaps (DB) framework is illustrated in Figure 3. On the top left is the queue for the tuple set that will be inserted into the existing topology. At the center are the Rank and Mapping Schemes whose main task is to point the new tuples to their corresponding partitions. For each partition, we define the following two parameters: $prefix\text{-}length$ (τ) and $prefix$. These are shown within the partitions in Figure 3. For instance, P_2 has τ = 3 and $prefix$ = 011. That means all the tuples in P_2 has the prefix 011. Within the Mapping Scheme, we keep the maximum τ among the partitions, i.e., τ_{max} = 3 in the figure.

The Rank algorithm in the framework should be tailored to the given tuple-ordering, which is GCO in our case. In our design, the rank function needs the number of bits as a parameter. For instance, since τ_{max} = 3 in Figure 3, the function takes only the 3 most significant bits and therefore the range of the mapping function will be [0, 7]. E.g., the 3-bit GCO rank value of a 5-bit tuple t_i = 01111 is 2, that is $Rank(t_i, \tau_{max})$ = 2. Next, the tuple is mapped to a partition based on its rank. For instance, let M denote the Mapping Scheme, the partition for t_i would be given by $M[Rank(t_i, \tau_{max})]$, which in this case is P_2.

4.1.1 Insertion Algorithm

We present the incremental insertion methodology to our dynamic structure in Algorithm 1. Given a tuple t_i, first line of the algorithm follows the mapping framework in Figure 3 and maps the tuple to the corresponding partition. In our implementation, we limit the size of the partitions in terms of the number of tuples that can fit into the memory. Note that there is always room for tuple t_i in a partition [line 2]. This is because a partition is split as soon as it becomes full [lines 3-12].

Algorithm 1 $Insert\ (t_i)$
Inserts a given tuple t_i to its corresponding partition.
M: Mapping, p: pointer to a partition

```
1: p ← M[Rank(t_i, τ_max)]
2: Append t_i to p
3: if p is full then
4:    Obtain a temporary space TS to store all tuples in p
5:    τ ← prefix length of p
6:    Obtain a new partition p'
7:    Set the prefix lengths of p and p' to τ+1
8:    if τ + 1 > τ_max then
9:       τ_max ← τ_max + 1
10:      Update M
11:   for each tuple t_q in TS do
12:      Insert(t_q)
```

4.1.2 Mapping Scheme Implementation

The task of pointing a given tuple to a partition based on its prefix is achieved by the Mapping Scheme in our framework. Distinct but related structures in the literature that can be adapted to our structure are [8, 14, 16]. There are several extensions provided by the framework for these design choices. Our structure is not necessarily limited to disk pages as is the case in these traditional approaches. The actual partitions can be represented by files. Furthermore, our scheme has the ability to enforce any user-specified order. In addition, our approach consumes memory linearly, as opposed to a contiguously allocated directory whose size changes exponentially. In order to utilize the memory efficiently, we consider two options. One solution is to change the order of the columns, and bring to front the columns that differentiate tuples in earlier bits. In order to achieve this, we sort the columns in increasing order based on the difference between the number of set bits (1s) and non-set bits (0s). Thus, the column with highest entropy, i.e., the column with almost equal number of ones and zeros, will now be the first column in the order. However, a disadvantage with this approach is that after a series of insertions, the order of the columns may need to change since a column can have more set bits inserted than non-set bits (or vice versa). This is impractical and in addition, some applications may not allow to change the order of columns in the current index. As an alternative solution, we adapt a *binary-tree-like* structure in our scheme which efficiently utilizes the memory.

4.1.3 Design Issues

Mapping Scheme can either be based on the most significant bits of the tuples or on the least significant bits. The challenge for the latter option is that the tuples that are actually distant in GCO can map to the same partition and this will affect the compression performance. As τ increases, these tuples need to be moved to different partitions, which will be costly. For instance, assuming τ_{max} = 3 in Figure 3, 5-bit tuples 00000 and 10000 would map to the same partition (namely P_1) since their *least-significant-3-bit* ranks are equal (i.e., 0), although their *least-significant-5-bit* ranks are 0 and 31 respectively (these will be clear in the next section). On the other hand, it is still possible to follow the *least significant bits option*, however that leads to a totally different ordering and the Mapping Scheme also needs to follow the same ordering. Depending on the user-specified order, any subset of bits in tuples can be utilized by the mapping. Without loss of generality, we use the most significant bits, and from now on *rank* will simply refer to the *most-significant-bits-rank*.

Note that bitmaps are file-resident and each column is stored individually. A typical DBMS accomplishes more advanced memory management and uses low level IO functions which are faster since they directly interact with the disk controllers. In addition, the authors of [17] investigate different design choices for modern computer architectures. In their RIDbit implementation, the sequence of rows on a table are broken into equal-sized fragments and each fragment is placed on a single disk page. Similar disk page allocation techniques can be adapted for our scheme to further enhance its performance.

GCO Decimal	GCO Binary	GCO-rank Binary	GCO-rank Decimal
0	00000	00000	0
1	00001	00001	1
3	00011	00010	2
2	00010	00011	3
6	00110	00100	4
7	00111	00101	5
5	00101	00110	6
.....	.....	.....	.....
21	10101	11001	25
23	10111	11010	26
22	10110	11011	27
18	10010	11100	28
19	10011	11101	29
17	10001	11110	30
16	10000	11111	31

Table 6: GCO Ranks for 5 bits

4.2 GCO Rank Algorithm

In this section, we discuss our linear GCO rank algorithm. To motivate the problem, a subset of GCO is presented for 5 bits in Table 6. The second column is the binary GCO produced by the reflection method as described in Section 2.2 and the first column includes the corresponding decimal values. The third and fourth columns tabulate the ranks both in binary and decimal. The function of the rank algorithm is to return the rank (fourth column) given a bit-string (second column). We now present our GCO Rank Algorithm, which returns the rank in the binary form (i.e., the third column)[3].

Algorithm 2 receives two parameters: a bit-string and the number of bits utilized to produce the rank. The reason for the second parameter is as follows. One can feed the algorithm with a long bit-string and analyze only the rank of a *prefix* of the string by ignoring the remaining bits. Note that Algorithm 2 is linear in the number of bits (b) utilized.

We now provide an example to go through the algorithm. Let's take t = (10101) as the input tuple, whose rank we are looking for will be (11001) (or 25 in decimal) in Table 6. Assume that we are interested in all 5 bits of the tuple. Therefore, *for-loop* of Algorithm 2 will be executed 5 times (line 3). Line 4 will evaluate to true because $hasSeenSetBit$ is initially false. Line 5 will be false since the first bit of t is one. Next, line 8 will initialize the variable *rank* to 1. Since we concatenated the *rank* with 1, the $hasSeenSetBit$ will be true (line 9). This means, in the following iteration of the loop we will flip the next bit of t. In the second iteration, line 12 will be executed and the current value of *rank* will be 11. Since we concatenated the *rank* with 1 again, the following iteration will also flip the next bit. In the third iteration, line 14 will set the current value of *rank* to 110. At the end of the fourth iteration, *rank* will be 1100 (line 6). Finally, the fifth iteration yields 11001 as the value of *rank* (line 8), which is actually what we were looking for as the output.

THEOREM 4.1. *For a bit-string s and c bits, Algorithm 2 produces the GCO-rank of s.*

[3] Other implementations that translate decimal values to different GCO Ranks and vice versa are also publicly available.

Algorithm 2 $GCRank\ (t, b)$
Given a bit-string (tuple) t and the number of bits needed b, the algorithm returns the rank of the tuple in GCO for b bits.
$B(t, i)$ − returns i^{th} bit of t
$x \bullet y$ − returns the concatenation xy

```
1: rank ← null
2: hasSeenSetBit ← false
3: for (i=1; i ≤ b; i++) do
4:    if (hasSeenSetBit == false) then
5:       if (B(t,i) == 0) then
6:          rank = rank • 0
7:       else
8:          rank = rank • 1
9:          hasSeenSetBit ← true
10:   else
11:      if (B(t,i) == 0) then
12:         rank = rank • 1
13:      else
14:         rank = rank • 0
15:         hasSeenSetBit ← false
16: return rank
```

PROOF. The proof is based on induction on the number of given bits. The inductive basis is for c = 1. Observe that for bit-strings 0 and 1 the algorithm produces the correct ranks, i.e., 0 and 1 respectively. Assuming the function produces the right answer for c = k, let's examine the correctness for c = k+1. For k+1 bits, we have 2^{k+1} possible ranks, that is from 0 to $2^{k+1} - 1$. Let's consider these values in four equal parts: [0, $2^{k-1} - 1$], [2^{k-1}, $2^k - 1$], [2^k, $2^k + 2^{k-1} - 1$], [$2^k + 2^{k-1}$, $2^{k+1} - 1$] and name them as part 1, 2, 3, 4 respectively. For the first two parts, the algorithm only adds a zero as a prefix to the rank variables. Since we assume it works for c = k, adding a zero to the beginning of a binary number will not change its decimal value and the ranks will also be right for c = k+1. For part 3, note that all the bit-strings start with 1, therefore the algorithm appends 1 to the rank variable and then flips the second bit of the input bit-string. This way, part 3 produces the same binary rank values as part 1 except now the first bits are 1. That is, the ranks of part 1 are repeated for part 3 by adding 2^k to the ranks of part 1. Similarly, ranks of part 2 are repeated for part 4 by adding 2^k to the ranks of part 2. Since all the bit-strings start with 1, the algorithm keeps 1 and flips the next bit, which are all zeros for part 4. Therefore, the algorithm produces the binary ranks of part 4 same as part 2 except that the first bits are 1 instead of 0. □

4.3 Additional Uses and Advantages

There are additional uses and advantages of the proposed framework, which we summarize shortly below.

i) Other Orderings: We presented the framework using GCO as the ordering strategy due to its better compression ratio against lexicographic ordering and also its efficient and effective performance when compared to other TSP heuristics. However, the scheme works with any user-specified order.

ii) Preserving Optimum Order: Besides keeping a dynamic structure, our scheme is also capable of achieving the optimum compression ratio. Targeting the overall reordering, the technique processes the data in the partitions locally. At any given batch time, by reorganizing all the partitions similar to the traditional rebuilding, the performance of the technique reaches the optimum case that is achieved by reorganizing the entire data globally, with a small overhead of few runs being split by partitioning.

iii) Prefixes of Partitions: Recall that all the tuples within a partition have the same prefix. In our framework, the prefixes constitute a redundancy so that they do not need to be stored. This allows us to save more space and time during index creation.

	Number of Rows	Number of Columns	Number of WAH words	
			Original	With GCO
Landsat	275,465	600	1,433,908	978,318
Z1	2,010,000	250	8,139,089	2,723,993
UNI	2,100,000	250	12,094,597	5,152,517
HEP	2,173,762	122	3,180,845	562,826

Table 7: Data Set Statistics

iv) Query Execution: Since the prefixes within a partition are equal and kept by the Mapping Scheme, DB can efficiently answer the queries that are seeking the bins in a prefix, without retrieving any actual data. For this reason, frequently queried bins should be placed early in the column order. Besides the prefixes, we experienced that there are many other bins for which all the tuples within a partition have the same value. These bins do not need to be stored and retrieved either, which would further improve the query performance. Note that, the conventional bitmap indexes do not allow partial retrieval of a column. Even though a column is composed of only a few set-bits, one needs to retrieve and apply the bitwise operations to the entire column in a traditional approach.

v) Deletions and Updates: For the scenarios where deletions also occur, instead of deleting every tuple literally, one can just mark a deleted tuple by utilizing an *Existence Bitmap* (EB) for each partition [19]. For the query execution, after the bitwise operations are applied, the resulting bitmap needs to be ANDed with EB[4]. Furthermore, we reorganize a partition right after a split occurs, which naturally allows us to make the literal deletions within partitions. This is much more dynamic and efficient compared to rebuilding the entire bitmap table since the deleted tuples in the latter approach will still be unnecessarily processed for the queries until the rebuilding occurs. Besides deletions, the tuple updates can easily be handled by a deletion plus an insertion.

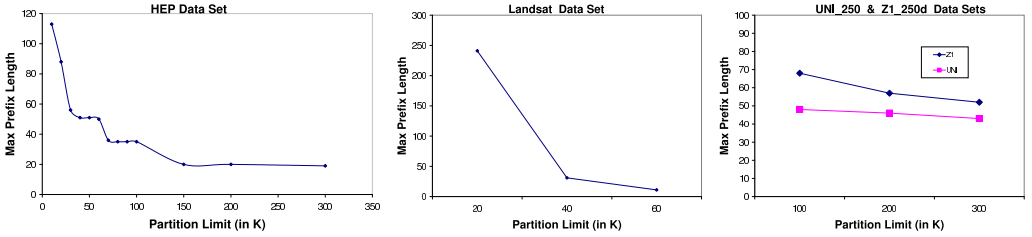
5. EXPERIMENTAL RESULTS

In this section, we discuss our experimental setup and present the empirical results. We performed experiments in order to quantify our scheme based on the number of partitions, prefix lengths, compressed storage size, and query execution time. We also compared our approach with a baseline technique where the bitmaps are split into main memory-sized chunks and the tuple ordering (GCO) is applied to each chunk independently. For a scenario where new insertions occur, we store the new tuples in a new chunk, and apply GCO to this new chunk once it becomes full. We call this approach *Chunk*.

5.1 Experimental Setup

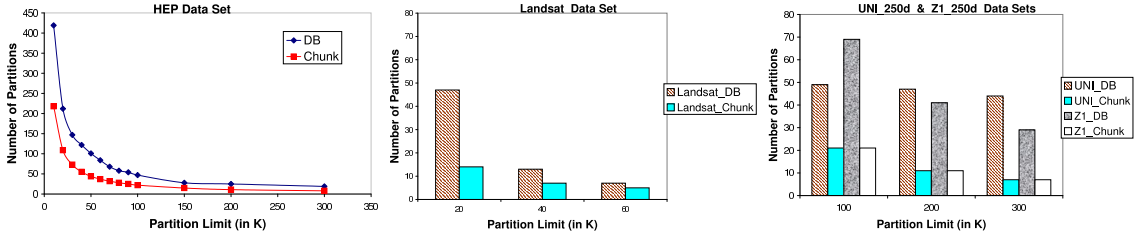
The experiments were performed with four data sets, three of which contain more than 2 million rows. HEP is a 12 attribute real bitmap data set generated from High Energy Physics experiments, and each attribute ranges from 2 to 12 bins, for a total of 122 bitmaps. Landsat data set is the SVD transformation of satellite images. UNI and Z1 are synthetically created data sets following uniform and zipf (with parameter set to 1) distributions respectively. The details are tabulated in Table 7. The synthetic data sets

[4]Consider a range query: Select * From X Where 1≤A≤5 AND 6≤B≤10. This requires 5 ORs for attribute A, 5 ORs for attribute B, and 1 AND accross A and B. Finally, *EB* adds one more bitwise AND operation to these 11 operations.



(a) (b) (c)

Figure 4: Maximum Prefix Lengths (τ_{max}) for All the Data Sets



(a) (b) (c)

Figure 5: Number of Partitions as a Function of Partition Limit

have varying number and varying cardinality of attributes but we only present the 250-dimensional cases in the table. The last two columns are the compressed sizes of the data sets in terms of the number of WAH-words (without and with GCO).

The experiments are based on Java implementations which were run on a Pentium IV 2.26 GHz processor machine with 1 GB of RAM using Windows XP Pro Operating System.

Recall that *Partition Limit* is the maximum number of tuples a partition can have, i.e. a partition splits as soon as it gets full. *Query Execution Time* is the time to run a combination of point and range queries using the appropriate bitmap query execution technique.

5.2 Prefix Length and Number of Partitions

Figure 4 illustrates the maximum prefix length (τ_{max}) as a function of the partition limit. For all the data sets, τ_{max} decreases as the partition limit is increased. However, for small partition sizes note that τ_{max} reaches high values. E.g., for HEP data, τ_{max} = 110 means a doubling mapping structure that is mentioned in Section 4.1.2 would require a directory that consists of 2^{110} pointer cells, which is clearly infeasible. On the other hand, a *binary-tree-like* architecture requires leaf pointers whose total number is linear in the number of partitions.

Figure 5 shows the number of partitions as the partition limit increases for all the data sets. Note that the Chunk approach has lower number of partitions on the average compared to DB. This is because the chunks (or partitions) for that technique are always full (except the last one), therefore the partition utilization is maximum.

5.3 Compressed Storage Size

We present the total number of WAH words required as the partition limit is varied in Figure 6 for all the data sets. For this experiment, besides keeping the partitions separate, we also concatenated the partitions into a single (large) partition and calculated the total number of words in this merged partition. *Chunk_Concat* and *DB_Concat* in the figure refer to this approach. In terms of total words, it is important to note that *DB_Concat* has actually the optimum performance one can achieve for a given reordering technique. In other words, it is the same as reordering the entire bitmap table without applying any partitioning.

Furthermore, the difference between *DB* and *DB_Concat* is an effect of partitioning (this is also valid for *Chunk* and *Chunk_Concat*). That is, the partition borders might end up cutting some border words (or runs) of the concatenated version into two separate words in the partitioned version. However, this overhead is minimal. In addition, Figure 6 shows that our technique, *DB*, performs very close to *DB_Concat* for all the data sets. Besides, *DB* is much more efficient than *Chunk* method, even though *Chunk* has fewer number of partitions in general (see Figure 5).

In order to observe the positive effect of tuple reordering in bitmap indices, it is also important to report the total number of words in the original bitmap table. Without any reordering and without any partitioning, for instance HEP data set has $3,180,845$ words in total (see Table 7). Both *DB* and *Chunk* approaches are much more efficient than that since they utilize reordering.

At this point it is important to note that the storage performance comparisons of *DB* and other techniques is done with the naïve implementation of *DB*, with no optimizations and all the bitmaps explicitly stored. However, the storage performance of *DB* would actually be further improved with the optimizations of Section 4.3.

5.4 Query Execution and Insertion Time

Figure 7(a) depicts how the query execution time compares using *Chunk_Concat* and *DB_Concat* approaches for HEP data set. Times are provided for a combination of 12 dimensional 100 point[5] and range queries using the indicated technique. Note that the results of total-number-of-words in Figure 6 reflect to the query execution performances in Figure 7(a). *DB_Concat* technique answers queries faster than *Chunk_Concat* since it has fewer words. For instance, for a partition limit of 10K in Figure 7(a), *DB_Concat* provides 37% improvement over *Chunk_Concat*.

Tuple reordering also has a significant effect on the query execution performance. Without applying any reordering and partitioning, just by appending the tuples to the end of the indices, the query execution time is 125.5 msec. Both *Chunk_Concat* and *DB_Concat* enable faster queries than this approach. For instance, for a partition limit of 10K, *Chunk_Concat* provides 72% improvement and *DB_Concat* provides 83% improvement.

[5]For bitmap indices, note that the point queries are just a special case for the range queries, i.e., with only AND operations.

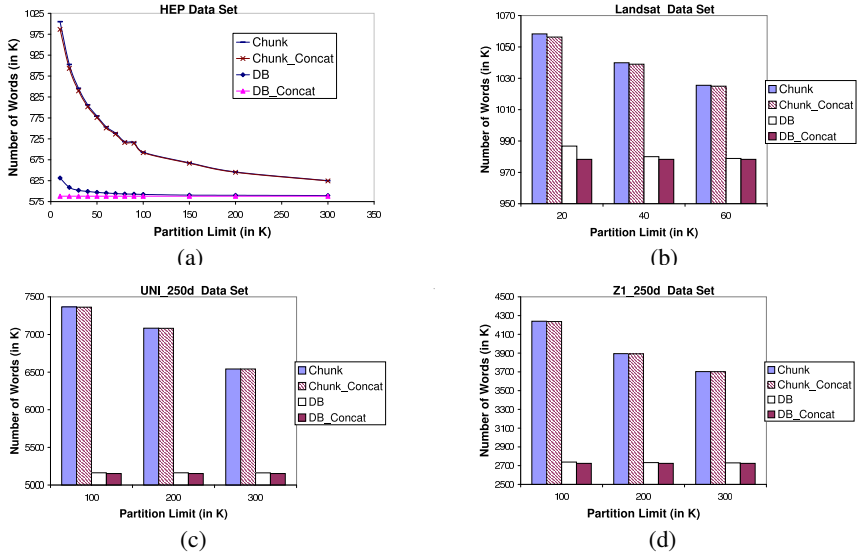


Figure 6: Total Number of WAH Words as a Function of Partition Limit

For a comparison between *Chunk* and *DB* techniques, we implemented *0-fill-pad-words* approach for both techniques, which is discussed in Section 2.3. In addition, for *DB* scheme we also implemented the optimization items iii and iv of Section 4.3.

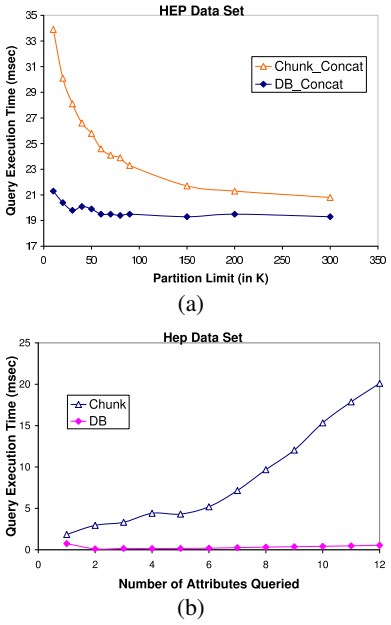


Figure 7: Query Execution Time

Figure 7(b) illustrates the query execution time comparison of *Chunk* and *DB*. For this experiment, our aim was to investigate the impact of number of attributes in the query. Therefore, we utilized from 1 to 12 dimensional 100 random[6] point queries[7], and the results are presented as averages. Note that, the larger the number of attributes in the queries, the more time it takes for the *Chunk* approach. This is because larger number of bitwise operations are used as we increase the number of queried attributes. On the other hand, the performance of *DB* is not affected by the number of queried attributes. The reason is the following. First of all, increasing the number of queried attributes decreases the number of matching partitions, therefore fewer number of partitions need to be accessed. In addition, *DB* approach doesn't process an entire bitmap (or column), instead only processes the part that is resident in a matching partition. Furthermore, thanks to the optimizations that *DB* enables, some parts do not need to be accessed, i.e., if all the rows have the same bit value for a bitmap in a partition.

We also experimented with the insertion time for new tuples. For this experiment, first we constructed the *Chunk* and *DB* structures using the entire HEP data set except the last 100 rows. Then we timed the insertion of these last 100 rows to the both structures. This took about 0.8ms for *Chunk* and 1.0ms for *DB*, which are comparable. We pay a little insertion overhead for *DB* but gain a lot from query execution performance.

5.5 Periodic Reorganization

In order to compare the proposed technique with a periodical reorganization approach, we followed a more feasible scenario than the procedure described in the beginning of Section 4. To the advantage of periodical reorganization approach, assume that updates occur only at certain period of times. We directly append the new rows to the end of the index while in the update-frequent session, and then in the infrequent session apply the rebuilding and reordering only to the newly inserted tuples. Figure 8 presents the results. For this experiment, we followed two different frequencies of reorganization. First, we started with $500K$ number of rows and built the traditional index with reordering. Then, step by step, we inserted $100K$ number of rows until the data set size reaches $1,000K$ rows (Figure 8(a)). At that point, we reorganized the inserted $500K$ rows for the traditional approach. Then we repeated the same process for the second $500K$ number of rows, and so on. In Figure 8(b), we repeated the same experiment but this time made a reorganization every other $300K$ rows. Figure 8 reveals that *DB* performs better than the periodical reorganization approach. For

[6] Queries are randomly selected from the data set, therefore the selectivity of the queries are at least 1 tuple.

[7] Range queries could also have been used for this experiment. To observe the true effect in such a case, the range of each attribute in the queries must be equal since larger ranges take more processing time than smaller ranges in general.

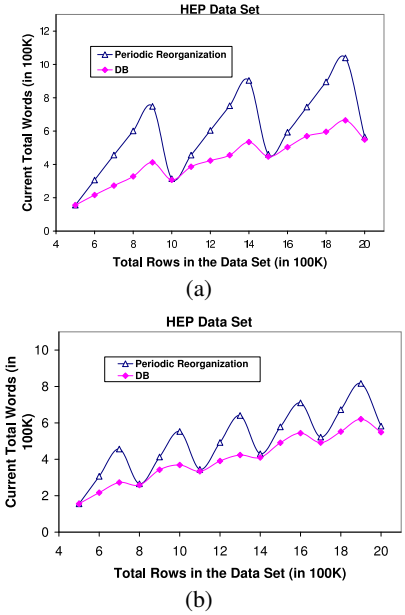


Figure 8: Periodic Reorganization

example, for 2 million rows in Figure 8(b), the periodical reorganization produces 582,293 number of WAH-words, whereas *DB* produces 548,918 WAH-words.

6. SUMMARY

We studied the problem of tuple appends to the ordered bitmap indices. For static data sets, it is known that the bitmap compression greatly improves by data reordering techniques. However, these data organization methods are not applicable to dynamic and very large data sets because of their significant overheads. We proposed a novel dynamic structure and algorithm for organizing bitmap indices to handle the tuple appends effectively. Given a user-specified order of the data set, our scheme enforces the optimum compression rate and query processing performance achievable for that order. We used Gray code ordering as the tuple ordering strategy for our experiments. However, the proposed scheme efficiently works for any desired ordering strategy. We aimed to keep a user-specified order of the data on bitmap indices and utilized a partitioning strategy tailored to our purposes. We conducted experiments to show that both compression and query execution are significantly improved with our technique.

7. REFERENCES

[1] S. Amer-Yahia and T. Johnson. Optimizing queries on compressed bitmaps. *The VLDB Journal*, pages 329–338, 2000.

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

[3] G. Antoshenkov and M. Ziauddin. Query processing and optimization in oracle rdb. *The VLDB Journal*, 5(4):229–237, 1996.

[4] T. Apaydin, G. Canahuate, H. Ferhatosmanoglu, and A.S. Tosun. Approximate encoding for direct access and query processing over compressed bitmaps. In *VLDB*, pages 846–857, Seoul, Korea, September 2006.

[5] D. K. Burleson. Oracle tuning: The definitive reference. *Rampant TechPress*, April 2006.

[6] G. Canahuate, M. Gibas, and H. Ferhatosmanoglu. Update conscious bitmap indices. In *SSDBM*, Banff, Canada, July 2007.

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

[8] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong. Extendible hashing: A fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979.

[9] Informix. Decision support indexing for enterprise datawarehouse. http://www.informix.com/informix/corpinfo/-zines/whiteidx.htm.

[10] D. Johnson, S. Krishnan, J. Chhugani, S. Kumar, and S. Venkatasubramanian. Compressing large boolean matrices using reordering techniques. In *VLDB 2004*.

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

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

[13] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley, 1998.

[14] P. Larson. Dynamic hashing. *BIT*, 18:184–201, 1978.

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

[16] W. Litwin. Virtual hashing: A dynamically changing hashing. In *VLDB*, pages 517–523, Berlin, 1978.

[17] E. O'Neil, P. O'Neil, and K. Wu. Bitmap index design choices and their performance implications. In *IDEAS*, Banff, Canada, 2007.

[18] P. O'Neil. *Informix and Indexing Support for Data Warehouses*, volume 10, pages 38–43. Database Programming and Design, February 1997.

[19] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 38–49. ACM Press, 1997.

[20] A. Pinar, T. Tao, and H. Ferhatosmanoglu. Compressing bitmap indices by data reorganization. *ICDE*, pages 310–321, 2005.

[21] 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.

[22] 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.

[23] 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.

[24] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006.