Compressing Bitmap Indices by Data Reorganization*

Ali Pınar
Computational Research Division
Lawrence Berkeley National Laboratory
apinar@lbl.gov

Tao Tao
Department of Computer Science
U. Illinois at Urbana-Champaign
taotao@cs.uiuc.edu

Hakan Ferhatosmanoglu
Department of Computer Science
The Ohio State U.
hakan@cis.ohio-state.edu

Abstract

Many scientific applications generate massive volumes of data through observations or computer simulations, bringing up the need for effective indexing methods for efficient storage and retrieval of scientific data. Unlike conventional databases, scientific data is mostly read-only and its volume can reach to the order of petabytes, making a compact index structure vital. Bitmap indexing has been successfully applied to scientific databases by exploiting the fact that scientific data are enumerated or numerical. Bitmap indices can be compressed with variants of run length encoding for a compact index structure. However even this may not be enough for the enormous data generated in some applications such as high energy physics. In this paper, we study how to reorganize bitmap tables for improved compression rates. Our algorithms are used just as a preprocessing step, thus there is no need to revise the current indexing techniques and the query processing algorithms. We introduce the tuple reordering problem, which aims to reorganize database tuples for optimal compression rates. We propose Gray code ordering algorithm for this NP-Complete problem, which is an inplace algorithm, and runs in linear time in the order of the size of the database. We also discuss how the tuple reordering problem can be reduced to the traveling salesperson problem. Our experimental results on real data sets show that the compression ratio can be improved by a factor of 4 to 7.

1 Introduction

Advances in technology have enabled the production of massive volumes of data through observations and simulations in many scientific applications such as biology, high-energy physics, climate modeling, and astrophysics. In computational high-energy physics, simulations are continuously run, and events that are notable for physicists are stored with all the details. The number of events that need to be stored in one year is in the order of several millions [20]. In astrophysics, technological advances enabled devoting several telescopes for observations, results of which need to be stored for later query processing [21]. Genomic and proteomic technologies are now capable of generating terabytes of data in a single day's experimentation [28]. These new data sets and the associated queries are significantly different than those of the traditional database systems, most importantly due to their enormous size and high-dimensionality (more than 500 attributes in high-energy physics experiments). These new data sets and the associated queries pose a new challenge for efficient storage and retrieval of data and require novel indexing structures and algorithms.

Most of the scientific databases of practical interest are read-only, i.e., large volumes of data are stored once and never updated. Further use of the data is typically by means of selection queries. Various types of queries, such as partial match and range queries, are executed on these large data sets to retrieve useful information for scientific discovery. As an example, a user can pose a range query to retrieve all events with energy less than 15 GeV, and the number of particles less than 13. When the data are large and read-only, as in the case of scientific databases, indexing technolo-

[*] This work was funded by he Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract DE-AC03-76SF00098.

gies are well-known to significantly improve the performance of query and data analysis, thus developing index structures tailored for scientific data is crucial to effectively explore such data. Due to the scale and high dimensionality of these databases, simple extensions of traditional indexing strategies are inadequate: R-trees and its variants are well-known to lose effectiveness for high dimensions; hashing-based indices lack storage efficiency; and transformation based approaches are not effective for partial match and range queries. Furthermore, most of the indexing approaches do not focus on the size of the index structure itself. However, due to the huge data volume in a typical scientific database, the size of the indexing structure becomes as important as other parameters and must be taken into account.

Focusing on the major characteristics of scientific data, such as being read-only, having special access patterns and numerical attributes, researchers have managed to develop indexing techniques that are feasible for high dimensional scientific databases. Bitmap indexing, which has been effectively utilized in many major commercial database systems [2, 14, 27], has also been the most popular approach for scientific databases [3, 15, 22, 24, 25, 27]. Several techniques have been proposed exploiting the bitmap indexing approach for scientific data. The general idea is to organize the data as a two dimensional table. Events are stored rowwise as tuples. Every attribute is partitioned to several bins, and these bins form the columns of the table. A table entry is 1, if the tuple of this row is in the bin of the column, and "0" otherwise. Thus, the index table is a 0-1 table. This table needs to be compacted to be effectively used on a large database. General purpose text compression techniques are clearly not suitable for this purpose since they significantly reduce the efficiency of queries [11, 24]. Specialized bitmap compression schemes have been proposed to overcome this problem. The two most effective schemes in the literature are Byte-aligned Bitmap Code (BBC) [2] and Word-Aligned Hybrid Code (WAH) [1, 11, 24, 25, 26]. Both of these schemes, like many others [3, 27], are based on run-length encoding, i.e., they both replace repeated runs of 0s or 1s 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 to faster query processing.

Run-length encoding and its variants exploit uniform segments of a sequence, thus their performances depend directly on the presence of such uniform segments. Their effectiveness varies for different organizations of the database tuples, since ordering of tuples affect uniform segments in the columns. In this paper, we study how to reorder tuples of a database to achieve higher compression rates. Our techniques are used as a preprocessing step before compression, only to improve the performance, without affecting algorithms used for compression and querying. We state this tuple reordering problem as a combinatorial optimization problem, and propose heuristics for effective solutions for this NP-Complete problem [16]. We show a reduction of the tuple reordering problem to the traveling salesperson problem, which is a well-studied combinatorial optimization problem. However, given the enormous sizes of the databases, we are only restricted to memory and time efficient heuristics, which takes away the applicability of most frequently used techniques such as simulated annealing. In this paper, we propose Gray code sorting to order the rows of a bitmap table for larger segments of uniform 1s. Our algorithm is linear, in the size of the database, and an in-place algorithm, i.e., does not require any auxiliary memory allocation. Theoretically, we prove that our algorithm is optimal, when all cells of a bitmap table are full. In practice, our experiments on scientific data showed significant improvements in compression rates. In many instances, compressed file size for the reordered file less than half the compressed size of the original file. We have also observed a 5.36 times reduction in compresses file size on data set HEP1, bitmap table for which has has 122 columns and 2,173,762 rows.

The remainder of this paper is organized as follows. In the next section, we present compression algorithms for bitmap tables. Section 3 discusses the tuple reordering problem. We first define the problem, and introduce Gray code ordering, which is tailored for the tuple reordering problem. Next, we discuss the reduction to the traveling salesperson problem. Experimental results are presented in Section 4. Finally, we discuss future work and conclude with Section 5.

2 Compressing Bitmap Tables

The data that comes from scientific experiments is composed of attributes that are numerical or enumerated. Unlike conventional databases, a data record in a scientific database involves many more attributes, up to order of a hundreds. And the number of tuples is huge due to the technological advances that make it possible to generate huge volumes of data on a daily basis. High energy physics simulations generate millions of events to be stored in a single year. Due to such large data volume, even simple queries are extremely slow without an effective index structure in place. However, neither the well-known multidimensional indexing techniques [19, 10] nor their extensions [13, 12, 5, 7, 6] have been successful in scientific database systems, partly due to the effects of the infamous dimensionality problems [4, 23] and the massive scale of these systems.

Most practical approaches for indexing scientific data are based on bitmap indexing strategies [2, 27, 24, 14, 3, 22, 8, 9, 25, 15, 1, 11, 26]. For example, Wu, Otoo, and Shoshani proposed an effective bitmap indexing technique for large-scale high energy physics data [26]. This technique uses a compression technique called word-aligned hybrid (WAH) to compress the index structure to conveniently small sizes without losing accessing efficiency. Exploiting the fact that each attribute is numeric or enumerated, data are 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 a binning example with three attributes, each partitioned into two bins. The first tuple t_1 falls into the first bins in the attributes 1 and 2, and the second bin in attribute 3. Note that after binning we can treat each tuple as a binary number. For instance $t_1 = 101001$ and $t_2 = 010101$.

Table 1. Bitmap example

Tuple	Attribute 1		Attribute 2		Attribute 3	
	bin1	bin2	bin1	bin2	bin1	bin2
t_1	1	0	1	0	0	1
t_2	0	1	0	1	0	1
t_3	1	0	0	1	1	0
t_4	1	0	1	0	0	1
t_5	1	0	1	0	1	0
t_6	0	1	0	1	1	0

Binning method itself cannot compress the size, and instead, might even increase the size [3]. However, it converts the original table to a more concise format with only two different values: "0" and "1". Run length encoding [18] can therefore, be used over every column to compress the data when long runs of pure "0" or pure "1" blocks becomes possible. Pure run length encoding is not a good strategy for indexing because of its accessing inefficiency.

Unlike traditional run length encoding, WAH mixes run length encoding and direct storage. For instance, if the word length is 32, every column is partitioned to many length-31 blocks. If a block is a mixture of both "0" and "1", mark the most significant bit of encoded word "0" to indicate this word is *literal word* and copy the block to left 31 bits directly. Otherwise, without losing generality, assuming the block filled with all "1", we continue to scan and count the number of consecutive blocks which are filled in with all "1". To encode, the most significant bit is marked "1" to indicate this word is a *fill word*, and second significant bit is marked "1" to indicate the block is filled with "1"s. The remaining bits are used to store the number of blocks. Table 2 presents an example. The first row is a column from the original bitmap, which starts with a 1, continues with 20 0s, followed by 3 1s, 79 0s, and ends with 21 1s. The second column partitions it into 4 segments, each of which has 31 bits. Row 3 lists the hex representation of those segments, and row 4 is its WAH encoding. The first word is a literal word mixing 0 and 1, thus there is no change to its encoding. The second and third word are "fill word" with all 0. We then put them together. The encoding therefore is 80000002. The fourth word is another literal word.

3 Improving Compression Rates by Tuple Reordering

Run-length encoding and its variants exploit uniform segments of a sequence, thus their performances depend directly on the presence of such uniform segments. Their effectiveness can be improved by aligning data for longer uniform segments. In this section, we study the problem of reorganizing bitmap tuples for more efficient run-length encoding. In the next subsection, we describe the problem, which we call the *tuple reordering problem*. Then we discuss feasibility of reorganization, and requirements for an effective reordering algorithm. Finally, we discuss solution techniques. First, we propose exploiting Gray codes for ordering. Then we present a reduction of the problem to the traveling salesperson problem.

3.1 Problem Formulation

Our objective in reordering is to increase the performance of run-length encoding by having longer uniform segments and thus fewer number of blocks. Recall that run-length encoding, when used on bitmaps, packs each segment of "1"s into a block and stores a pointer to each block together with the length of the block. Thus its efficiency depends on the number of such blocks. Consider two consecutive tuples in the bitmap table. If the tuples are on the same bin for an attribute, then they will be packed to the same block. If not, then a new block should start. Efficiency can be enhanced by reordering tuples so that they fall into the same bins

Table 2. WAH compression

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

as much as possible. An example is illustrated in Figure 1. In this example, the original table has 12 blocks, whereas the reordered table requires only 7 blocks.

Let diff(t_i, t_j) be the number of attributes that tuple t_i and tuple t_j fall in different bins. Notice that diff(π_i, π_{i+1}) gives how many new blocks start at the ith tuple after reordering when run-length encoding is used, where π_i denotes the ith tuple in ordering π. An example for computing the diff values is illustrated in Figure 2. For example diff$(t_1, t_2) = 2$, since tuples t_1 and t_2 fall into different bins for the first two attributes. We can now formally define the tuple reordering problem.

Definition 1 (Tuple reordering problem) *Let π be an ordering of m tuples so that π_i denotes the ith tuple in the ordering. Tuple reordering problem is finding an ordering π that minimizes*

$$\sum_{i=1}^{m-1} \text{diff}(\pi_i, \pi_{i+1}). \tag{1}$$

In Equation 1, we sum diff values over all consecutive tuples to attain how many new blocks start for the whole table. The first tuple requires starting a block for each attribute. Thus the number of blocks can be computed as $A + \sum_{i=1}^{m-1} \text{diff}(\pi_i, \pi_{i+1})$, where A is the number of attributes. Thus finding an ordering that minimizes Equation 1 minimizes number of blocks in the reordered table. For instance, Equation 1 returns $2 + 2 + 2 + 1 + 2 = 9$ for the initial ordering, which means with the addition of A the number of attributes there will be $9 + 3 = 12$ blocks in the compressed table. Whereas for the reordered table in Figure 1, Equation 1 returns $0 + 1 + 1 + 1 + 1 = 4$, which means only 7 blocks in the compressed file.

3.2 Heuristics for Tuple Reordering

In this section we propose techniques to reorder database tuples for better compression rates. First we discuss feasibility of reorganizing a database and what is necessary for an ordering algorithm to be effective. We propose two approaches for tuple reordering. The first approach exploits the Gray codes for tuple reordering. We show that this technique is optimal under certain conditions. The second approach reduces the problem to the well-studied traveling salesperson problem.

3.2.1 Feasibility of Tuple Reordering

Databases are seldom reordered, since their enormous sizes make even moving data to implement a specified reordering a big challenge. Thus one needs to be careful while designing algorithms to find such reorderings. For an ordering algorithm to be applied to a database, it needs to be memory efficient. The memory requirement needs to be at least linear in the order of tuples. Preferably, the algorithm is *in-place*, which means it should not use any auxiliary memory. Also, it will be computationally inefficient, if not infeasible, to apply a technique to the whole database. An effective technique should be local, i.e., it must be sufficient to apply our techniques to the portions of the database to improve compression rates. This locality provides scalability to a technique, since it can be applied to databases of arbitrary sizes.

Reordering database tuples has only local effects, thus it is easy to localize reordering algorithms to only portions of the database. Reordering larger portions of the database is expected to yield better performance, thus it is still important to limit the memory requirement of the ordering algorithm to order larger portions of the database. The Gray code ordering proposed in the subsequent section is an in-place algorithm and thus optimal in terms of memory requirement. It can even be applied to the whole database, since it has a regular access pattern and requires a small number of passes over the bitmap table. The last section describes a reduction to the traveling salesperson problem, one of the most well-studied combinatorial optimization problems and a testbed for various optimization techniques. This reduction enables adoption of a wide variety of techniques to the tuple reordering problem, however these techniques almost invariably require additional storage, which is often superlinear in the number of tuples.

$$\begin{array}{c|cccccc|} t_1 & 1 & 0 & 1 & 0 & 0 & 1 \\ t_2 & 0 & 1 & 0 & 1 & 0 & 1 \\ t_3 & 1 & 0 & 0 & 1 & 1 & 0 \\ t_4 & 1 & 0 & 1 & 0 & 0 & 1 \\ t_5 & 1 & 0 & 1 & 0 & 1 & 0 \\ t_6 & 0 & 1 & 0 & 1 & 1 & 0 \end{array}$$

(a) Original Table

$$\begin{array}{c|cccccc|} t_1 & 1 & 0 & 1 & 0 & 0 & 1 \\ t_4 & 1 & 0 & 1 & 0 & 0 & 1 \\ t_5 & 1 & 0 & 1 & 0 & 1 & 0 \\ t_3 & 1 & 0 & 0 & 1 & 1 & 0 \\ t_6 & 0 & 1 & 0 & 1 & 1 & 0 \\ t_2 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}$$

(b) Reordered Table

Figure 1. Example for tuple reordering

$$\begin{array}{c|cccccc|} t_1 & 1 & 0 & 1 & 0 & 0 & 1 \\ t_2 & 0 & 1 & 0 & 1 & 0 & 1 \\ t_3 & 1 & 0 & 0 & 1 & 1 & 0 \\ t_4 & 1 & 0 & 1 & 0 & 0 & 1 \\ t_5 & 1 & 0 & 1 & 0 & 1 & 0 \\ t_6 & 0 & 1 & 0 & 1 & 1 & 0 \end{array}$$

(a) Original Table

	t_1	t_2	t_3	t_4	t_5
t_6	3	1	1	3	2
t_5	1	3	1	1	
t_4	0	2	2		
t_3	2	2			
t_2	2				

(b) Difference values between tuples

Figure 2. Function diff on an example

3.2.2 Gray Code Ordering

A Gray code is an encoding of numbers so that adjacent numbers have only a single digit differing by 1. For binary numbers two adjacent numbers differ only by one digit. For instance $(000, 001, 011, 010, 110, 111, 101, 100)$ is a binary Gray code. Binary Gray code is often referred to as the "reflected" code, because it can be generated by the reflection technique described below.

1. Let $S = (s_1, s_2, \ldots s_n)$ be a Gray code.
2. First write it forwards and then append the same code writing it backwards. That is $(s_1, s_2, \ldots, s_n, s_n, \ldots, s_2, s_1)$.
3. Append 0 at the beginning of the first n numbers, and 1 at the beginning of the last n numbers.

As an example, take the Gray code $(0, 1)$. Write it forwards, then add the same sequence backwards, and we get: $(0, 1, 1, 0)$. Then we add 0's and 1's to get: $(00, 01, 11, 10)$. We can use this new sequence as an input to our algorithm. After the reflection step we get $(00, 01, 11, 10, 10, 11, 01, 00)$. We add the first digits to attain: $(000, 001, 011, 010, 110, 111, 101, 100)$. It is worth noting that Gray codes are not unique, and different orders on the same numbers might satisfy the Gray code property. We use the term *fundamental Gray code* to refer to a Gray code generated by the reflection technique described above with using $(0, 1)$ as the initial sequence. We will also refer to ordering a set of numbers with respect to fundamental Gray codes or shortly *Gray code ordering*, which we describe next.

Definition 2 (Gray code rank) *The Gray code rank $g(s)$ of an n-bit binary number s is the rank of this number in an n-bit fundamental Gray-code.*

For instance, $g(0000) = 1$, since it is the first number in the 4-bit fundamental Gray code. And $g(0001) = 2$, since it follows 0000, in the fundamental Gray code.

Definition 3 (Gray code sorting) *A sequence $S = (s_1, s_2, \ldots, s_m)$ is Gray code sorted iff*

$$g(s_i) \leq g(s_{i+1})$$

for $i = 1, 2, \ldots m - 1$, where $g(s_i)$ refers to the Gray code rank of s_i.

The sequence $(0001, 0010, 0101, 1100, 1110, 1011)$ is Gray code sorted because $g(0001) = 2 \leq g(0010) = 4 \leq g(0101) = 7 \leq g(1100) = 9 \leq g(1110) = 12 \leq g(1011) = 14$.

This brings the question of how to efficiently order a set of numbers to be Gray code sorted. We can reverse the fundamental Gray code generation process, to sort numbers with respect to the fundamental Gray code. As the first step, we can divide numbers as those that start with 0 and those that start with 1. Clearly those that start with 0 will precede others in the ordering. Then we can recursively order those that start with 0. The same can be applied to the second group but we need to reverse their ordering due to the reflective property of the Gray code. In Algorithm 1, we present the pseudo-code of this algorithm. In this algorithm, $S(A, i, j)$ denotes the jth significant bit of the ith tuple in table A. Note that the reversion does not need to be a separate step in the algorithm, but we present it separately for clarity of the presentation.

```
GC-sort (A, $start, end, b$)
$i \leftarrow start$
$j \leftarrow end$
while $i < j$ do
    Decrement $j$ until $S(j, b) = 0$
    Increment $i$ until $S(i, b) = 1$
    if $i < j$ then
        Swap the $i$th and $j$th tuples on the table
    end if
end while
if $b <$ no_of_bits then
    GC-sort $(A, 1, j, b + 1)$
    GC-sort $(A, j + 1, end, b + 1)$
    Reverse $(j + 1, end)$
end if
```

Algorithm 1: An in-place Gray code sorting algorithm. GC-sort (A, $start, end, b$) sorts numbers between indices *start–end* in A according to their least significant b bits in Gray code order. $S(A, i, j)$ denotes the jth significant bit of the ith number in table A

Lemma 1 *Algorithm 1 orders numbers in A to be Gray code sorted, when initially invoked with GC-sort* $(A, 1, m, n)$, *where* m *is the number of tuples, and* n *is the number of bits.*

Proof *The proof is based on induction on the number of bits. First observe that recursive calls respect the previous orderings, since after one pass, the recursive calls only operate on the segment of tuples that all start with the same bit prefix.*

The inductive basis is for $n = 1$, *when it is easy to observe the correctness of the algorithm. It is also easy to see that numbers that start with* 0 *should precede those that start with* 1 *for Gray code sorting. By the inductive hypothesis, the numbers that start with* 0 *are sorted correctly by the algorithm according to their last* $n - 1$ *bits, and adding* 0 *does not affect their Gray code precedence. Similarly, numbers that start with 1 are Gray code sorted recursively according to their last* $n - 1$ *bits, however putting* 1 *at the beginning requires the reflected order, which we achieve by Reverse* $(j + 1, end)$.

Figure 3 illustrates this algorithm. It is important to note that Algorithm 1 is an in-place algorithm, which is important for our application since we have to deal with very large datasets.

Recall that since consecutive numbers differ at only one bit, Gray code numbers have maximum bit-level similarity between consecutive numbers. This observation can be used for ordering database tuples, since every tuple in the database can be considered as an n-bit binary number. By Gray code sorting, we can impose similarity between consecutive numbers. And if all distinct tuples exist, i.e., if all cells of the bitmap table are full, Gray code sorting will produce an optimal ordering. We formalize this claim with the following theorem.

Theorem 1 *Gray code ordering provides an optimal solution for the tuple reordering problem, if all cells of the bitmap table are full.*

Proof *The algorithm orders identical tuples consecutively. Thus at most one bit differs between two consecutive tuples, which implies optimality.*

By the result of Theorem 1, Algorithm 1 gives an optimal solution when all cells are full, however in practice this will rarely happen, and the solution may not be optimal. Gray code ordering is more effective when most of the cells are full, which means it is more effective with increasing number of rows, and thus larger databases. Its performance also depends on the number of attributes, and the number of bins per attribute. Increasing these two terms increases the number of cells in the bitmap table, making the table more sparse. Nevertheless, even when the bitmap table has a lot of empty cells, Gray code ordering imposes bit-level similarity between consecutive tuples very effectively as evidenced by the experimental results.

3.2.3 Reduction to the traveling salesperson problem

In this section, we describe a reduction of the tuple reordering problem to the traveling salesperson problem (TSP). TSP is a very well-studied problem, and many effective heuristics have been proposed in the literature [17], and has been a testbed to demonstrate the effectiveness of optimization methods such as simulated annealing and genetic algorithms. However our target application is database reorganization where the number of tuples (vertices of the TSP graph) may be easily in the order of millions, and the enormous sizes of these problems require memory- and time-efficient heuristics.

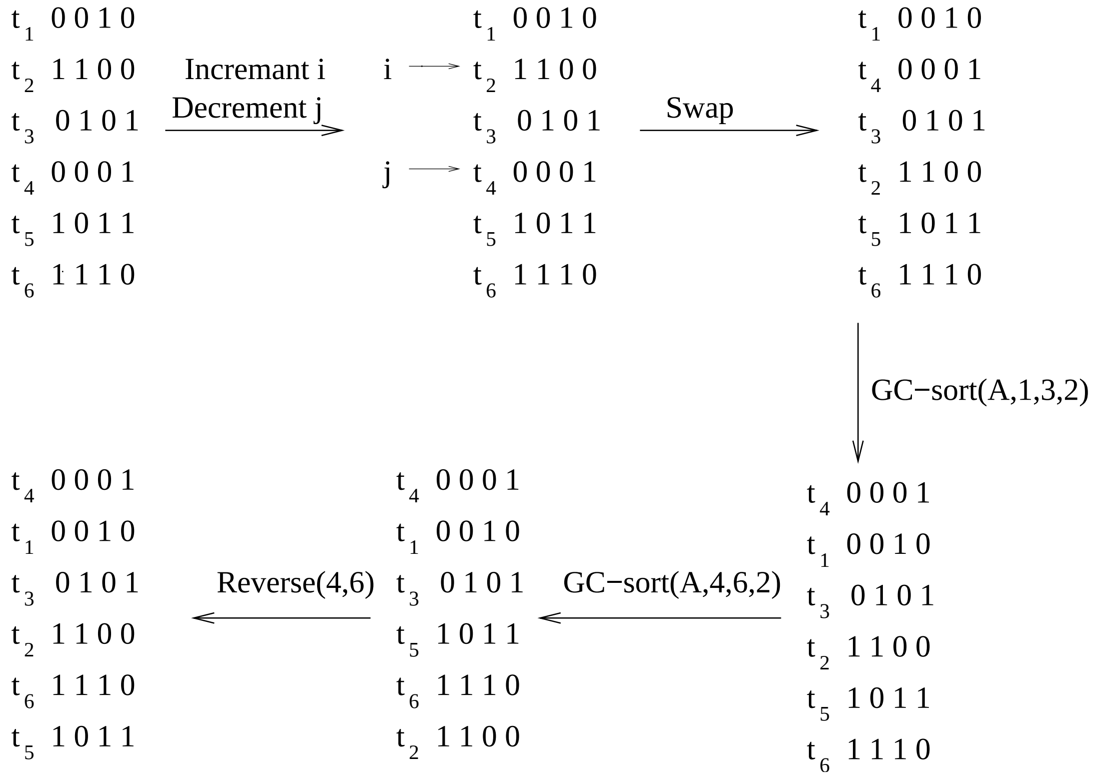

Figure 3. Illustration of Algorithm 1.

Traveling salesperson problem can be intuitively defined as finding a shortest path that visits all cities in a given map. In a graph theoretical formulation, cities correspond to vertices of a graph, and a weight function is defined on edges that connect vertices. The objective is to find a path visiting all vertices that minimizes the sum of weights of the edges between successive vertices. We describe a graph model to reduce the tuple reordering problem to the TSP.

Since we are seeking an ordering of tuples, we will have vertices to represent tuples and define a weight function so that an optimal solution to the TSP problem minimizes the number of blocks in run-length encoding. Given, a bitmap B as a set of tuples, define its graph $G_B = (V, E)$ so that each tuple t_i in B is represented by a vertex v_i, and each pair of vertices v_i and v_j is connected by an edge (v_i, v_j) in E. Define the weight of an edge (v_i, v_j) as diff(t_i, t_j) as defined in Section 3.1.

Theorem 2 *Given a bitmap B, define graph $G_B = (V, E)$ so that each tuple t_i is represented by a vertex $v_i \in V$. All pairs of tuples t_i and t_j are connected by an edge with weight $diff(t_i, t_j)$. Optimal TSP solution on G_B, gives an optimal solution to the tuple reordering problem.*

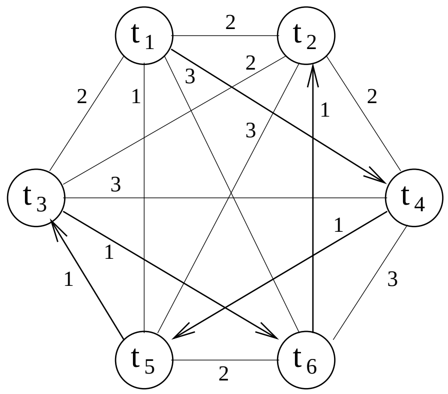

Figure 4. Reduction to TSP. TSP graph for the bitmap Table in Figure 1. Dark arrowed edges indicate an optimal TSP solution.

Proof *TSP ordering gives traversal of vertices that minimizes the sum of edge weights between consecutive vertices. When we replace vertices with tuples we get an ordering of tuples that minimizes the diff values between consecutive tuples. Thus minimizing the total edge weight corresponds to minimizing Equation 1, thus* *the number of blocks.*

TSP heuristics can be used to construct an ordering, or improve a given ordering. However, explicit construction of the TSP graph is not feasible for reordering database tuples. The TSP graph has $\binom{n}{2}$ potential edges. We can drop edges whose weights are zero, but even then number edges will be $O(n^2)$ for a bitmap table. Infeasibility of constructing the TSP graph restricts us to simple greedy strategies where edge weights can be computed on the air during the course of the algorithm. In our experiments, we used a 2-switch technique, which repeatedly seeks for a pair of vertices switching positions of which decreases the solution value. To further improve efficiency, we restricted the search for pairs to only those within a specified distance. It will be worthwhile to observe performances of other TSP heuristics from the literature, but it should be noted that one can use only a limited selection due to the very large sizes of the problems, and more importantly Gray code is already very effective and an in-place algorithm.

A similar problem has been studied by Pinar and Heath in the context of increasing memory performance of sparse matrix-vector multiplication [16]. The conventional data structures for sparse matrices require one memory indirection (extra load operation), during matrix-vector product operations. Pinar and Heath described how to reduce the number of memory indirections by exploiting nonzeros in consecutive positions in a column, and proposed a reordering method to reorder rows to align nonzeros of the matrix to consecutive positions in columns. Their method is based on a graph model that reduces the problem to the TSP. Tuple reordering problem is similar, since a bitmap can be considered as a sparse matrix, with tuples corresponding to rows and bins for all attributes corresponding to columns. We have a nonzero at row i and column j iff ith tuple is in bin j. However, the practical aspects of these two problems are significantly different, hence require different solution techniques. Sparse matrices arising in many applications define systems of linear equations and are square. Rectangular matrices arise especially in optimization, but even then the number of columns and the number of rows are close, at least in the same order. In databases however, the number of tuples, which corresponds to rows in a sparse matrix, is several orders of magnitude larger than the number bins, which corresponds to number of columns in a sparse matrix. Sparse matrices are much smaller in dimension compared to number tuples in a database.

4 Experimental Results

In this section, we discuss our empirical work to validate our proposed methods. We applied our reordering techniques to several data sets from various applications to observe the decrease in the sizes of the bitmap tables. As we will soon present in detail, we have observed significant improvements, which should directly translate into improvements in query processing times. Remember that scientific databases, which is the main motivation for our research are mostly read-only, thus reorganization needs to be done only once, for faster processing times in all future queries. Nevertheless, we also present the running times and scalability of our methods to prove the feasibility of application of our methods on very large databases.

It is also worth noting that our methods are used as a preprocessing step before actual compression algorithms, to align 1s in the bitmap table into consecutive positions. Thus, any compression algorithm can be employed to compress our reorganized data. In our experiments we used WAH compression algorithm [26].

We present the effectiveness of our methods based on the *improvement factor*, which we compute as the ratio of the compressed bitmap table size of the original data to the compressed bitmap table size of the reordered data, i.e,

$$\text{improvement factor} = \frac{\text{compressed size of original}}{\text{compressed size of reordered}}$$

Thus, an improvement factor of 5 means, compressed reordered data takes 5 times less space than the compressed original.

Table 3 reveals the effectiveness of our Gray code reordering algorithm on 6 data sets from various applications. In this table, the first three columns give the name of the problem, number of tuples, and number of columns in the bitmap table, respectively. The next two columns present the sizes of the compressed bitmap tables for the original and reordered data, respectively. The last column presents the improvement factor. Out of the 6 data sets, the first two data sets (HEP1 and HEP2) are from high energy physics applications. The third data set, histobig, comes from an image database with 112,361 images. Images are collected from a commercial CD-ROM and 64-dimensional color histograms are computed as feature vectors. The fourth data set, stock, is a time-series data which contains 360 days stock price movements of 6500 companies, i.e., 6500 data points with dimensionality 360. Histogram data set is partially correlated, whereas the stock data set is highly correlated. The last two data sets are composed of document feature vectors from 20 newsgroups based

Table 3. Improvement in compression of real data sets

	Bitmap table		Compressed size (bytes)		Improvement
Name	$\#columns$	$\#rows$	Original	Reordered	factor
HEP1	122	$2,173,762$	$3,149,590$	$587,773$	5.36
HEP2	907	$2,173,762$	$11,482,527$	$7,008,601$	1.64
histobig	64	$112,361$	$209,066$	$54,605$	3.83
stock	360	$6,500$	$156,980$	$22,904$	6.85
irvector16	160	$19,997$	$14,952$	$2,971$	5.03
irvector32	320	$19,997$	$17,135$	$11,064$	1.55

on tf/idf followed by SVD reduction.

As seen in Table 3, compression rates are magnified when the tuples are reordered with respect to Gray code ordering in all problem instances from all applications. The compressed index size for data stock is 7 times less than the original after reordering. The improvement factors are 5.36 and 1.64 for high energy physics data sets HEP1 and HEP2, respectively. Comparing the results for these two data sets, we see that, as expected, improvements are more significant, when the number of columns is smaller. Fewer number of columns means more room for improvement for a reordering algorithm, since more tuples are likely to fall into the same bins, and thus it is possible to order tuples so that consecutive tuples fall into same bins in a lot of attributes. A similar trend can be observed in information retrieval data sets irvector16 and irvector32, where the improvement factors are 5.03 and 1.55 respectively. Nevertheless, improvements are significant even for larger numbers of columns. It should also be noted that the Gray code ordering technique can be applied to arbitrary data sizes, since it is an in-place algorithm. This means the effectiveness of our techniques will only get better, as we apply these techniques to larger data sets.

As already discussed, our proposed techniques are preprocessing steps for conventional compression algorithms and associated query running techniques, and thus these query running techniques can be used as is, together with our algorithms. For this reason, we are not presenting any results on query run times, since it has been already reported that query run times are linearly dependent on the compressed bitmap table sizes. We expect our improved compression rates to translate directly into improved query run times. Notice that the effects of our improved compression rates will be even more dramatic under limited resources, which is typical in large-scale systems. Compacted index structures will grant better locality for algorithms, providing a second source of improvement.

In the second set of experiments, we have tested the performance of Gray code ordering for varying numbers of columns. We fixed the number of rows at 1,000,000 and tested the performance of our algorithm by varying the number of bins per attribute to change the number of columns to be 50, 100, 150, 200, 250, and 300. The results of our experiments are presented in Figure 5. In this figure original corresponds to the size of the compressed bitmap tables for the original data, whereas reordered corresponds to the size for compressing reordered data. As observed in this figure, compressed data sizes grow with increasing number of columns. Reordering significantly decreases compressed index size in all cases. The improvement factor is 2.52 2.08, 1.64, 1.92, 1.68 and 1.68, when the number of columns is 50, 100, 150, 200, 250, and 300, respectively. Fewer number of columns leaves more room for improvement for reordering due to increased likelihood of tuples in the same bins, which is nicely exploited by our Gray code ordering algorithm.

In the next set of experiments, we tested the run time performance of our algorithm. We run experiments on a Linux machine with 2.4GHz CPU and 1GByte memory. We used the irvector data from an information retrieval application, which has 19,996 tuples and 32 attributes, as our base data set, and randomly selected tuples, and attributes for our scalability studies. The results presented in Figures 6–8 are the averages of five runs on different problems of the same size. That is the run time of the algorithm for 1,00 rows is reported as the average run time for 5 randomly selected row sets of size 1,000.

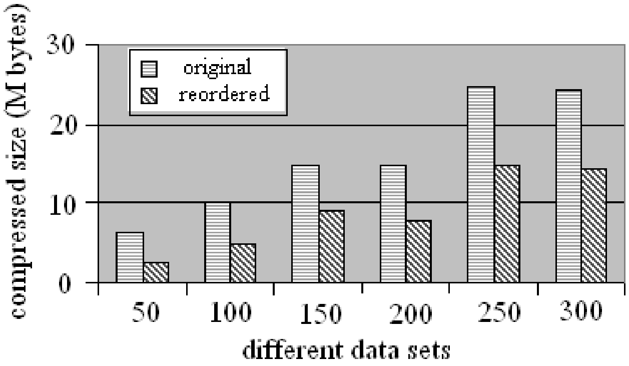

Figure 5. Performance for varying numbers of columns

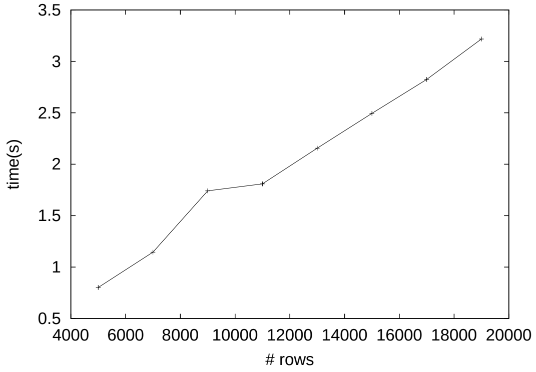

Figure 6. Algorithm scalability on the number of rows

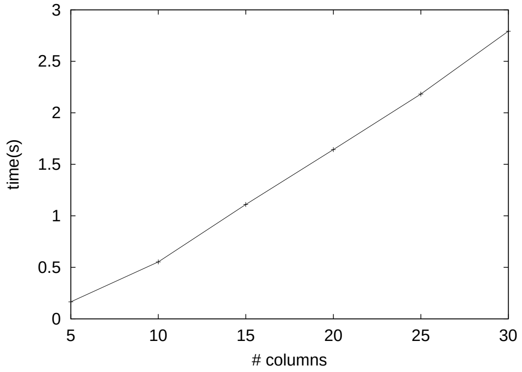

Figure 7. Algorithm scalability on the number of attributes

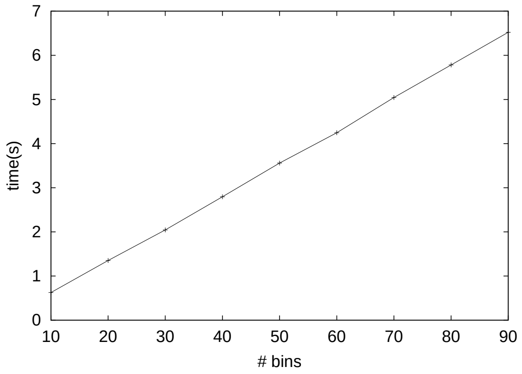

Figure 8. Algorithm scalability on bins per attribute

Figure 6 studies the effect of number of rows in the run time. For these runs, we used 30 attributes all of which are partitioned into 10 bins. The number of objects vary from 5000 to 19000. In Figure 6, the x-axis is the number of rows, and the y-axis is the run time in seconds, and the results clearly show the linear relation between the number of rows, and the runtime. Similarly, Figures 7 and 8, observe the effect of numbers of attributes and bins per attribute on the run time. In Figure 7, we fix the number of objects as $19,000$, the number of bins per attribute as 40. In Figure 8, we fix the number of objects as $19,000$ and the number of attributes as 30. All results confirm the linear relation between the runtime of our algorithm and the bitmap table size.

In the final set of experiments, we applied the 2-switch heuristic described in Section 3.2.3 on the TSP graphs for tuple reordering. As expected the runtimes were orders of magnitude slower compared to Gray code ordering. For instance, Gray code ordering on HEP1, which has 122 columns and 2,173,762 rows took only 43.4 seconds, whereas the 2-switch heuristic on the TSP graph took over 1,600 seconds. We have observed some improvement in the compression (around only 1%), but the huge gap in run time was daunting. We have observed similar results in the other data sets.

5 Conclusions and Future Work

We studied the problem of improving bitmap index compression rates by reorganizing data layout. Our algorithms reorder database tuples so that consecutive tuples are likely to fall into same bins to boost the performance of run-length encoding based compression schemes. We defined the tuple reordering problem, which aims to find an ordering of tuples that maximizes the similarity (measured by being in the same bin), between consecutive tuples. We proposed Gray code ordering technique for the tuple reordering problem, which exploits the idea of Gray codes. Our algorithm runs in linear time in the size of the database, and does not require any extra storage. This provides the applicability of our algorithm to very large data segments, even to the whole database. We also presented a reduction of the tuple reordering problem to the well-known, well-studied traveling salesperson problem(TSP). However, enormous sizes of the problems hinder applicability of frequently used TSP techniques for the tuple reordering problem. Our experiments showed that bitmap compression rates can be magnified by reordering database tuples. In many instances, compressed file size for the reordered file less than half the compressed size of the original file. We have also observed a 5.36 times reduction in compresses file size on data set HEP1, bitmap table for which has has 122 columns and 2,173,762 rows.

This paper shows the incontestable advantages of data reorganization for elevating bitmap index compression and introduces an important problem, which we call the tuple reordering problem. While our techniques are very effective in decreasing compressed bitmap indices, they are only our first steps in this direction, and leaves much for further research. The performance of Gray code sorting algorithm is affected by the order, in which we process the columns, and thus finding a good ordering of columns will be another interesting research project. Also, the literature in TSP is extremely rich, a more detailed study on adopting TSP techniques for the tuple reordering problem is worth investigating. Although enormous problem sizes hinder most of the techniques, a thorough study into TSP literature might be able to produce techniques, which avoid explicit construction of the TSP graph and might be applied to smaller segments of the data. Finally, existing compression algorithms are tuned for unordered data, whereas our algorithms provide long uniform segments in the data. We expect significant additional improvements in compression rates by tuning existing compression algorithms to reorganized data. In general, an interesting avenue will be better integration of ordering and compression algorithms, where ordering algorithms are tuned for the compression algorithm to be used, and the compression algorithms are tuned for the reordered data.

Acknowledgments

We are grateful John Wu from Lawrence Berkeley National Laboratory for his provision of some of the data sets, and insightful discussions.

References

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

[2] G. Antoshenkov. Byte-aligned bitmap compression. Technical Report, Oracle Corp., 1994. U.S. Patent number 5,363,098.

[3] A.Shoshani, L.M.Bernardo, H.Nordberg, D.Rotem, and A.Sim. Multidimensinal indexing and query coordination for tertiary storage management. In *SSDBM*, pages 214–225, 1999.

[4] S. Berchtold, C. Bohm, D. Keim, and H. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proc. ACM Symp. on Principles of Database Systems*, pages 78–86, Tuscon, Arizona, June 1997.

[5] S. Berchtold, D. Keim, and H. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 28–39, Bombay, India, 1996.

[6] C. Bohm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33:322–373, 2001.

[7] K. Chakrabarti and S. Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. In *Proc. Int. Conf. Data Engineering*, pages 440–447, Sydney, Australia, 1999.

[8] C.-Y Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *SIGMOD*, pages 355–366, 1998.

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

[10] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30:170–231, 1998.

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

[12] K. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal*, 3:517–542, 1995.

[13] D. B. Lomet and B. Salzberg. The hb-tree: A multi-attribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, December 1990.

[14] P. O'Neil. Model 204 architecture and performance. In *2nd International Workshop in High Performance Transaction Systems*, pages 40–59, Asilomar, CA, September 1987.

[15] Ekow J. Otoo, Arie Shoshani, and Seung won Hwang. Clustering high dimensional massive scientific dataset. In *SSDBM*, pages 147–157, Fairfax, Virginia, July 2001.

[16] A. Pınar and M. Heath. Improving performance of sparse matrix-vector multiplication. In *Proc. of Supercomputing 99*, 1999.

[17] G. Reinelt. *The traveling salesman: computational solutions for TSP applications.* Springer-Verlag, Lecture Notes in Computer Science, Vol: 840, 1994.

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

[19] H. Samet. *The Design and Analysis of Spatial Structures.* Addison Wesley Publishing Company, Inc., Massachusetts, 1989.

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

[21] SNAP. Supernova acceleration probe. http://snap.lbl.gov/, 2004.

[22] K. Stockinger. Bitmap indices for speeding up high-dimensional data analysis. In *DEXA*, 2002.

[23] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 194–205, New York City, New York, August 1998.

[24] K. Wu, E. J. 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.

[25] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Compressing bitmap indexes for faster search operations. In *SSDBM*, pages 99–108, Edinburgh, Scotland, UK, July 2002.

[26] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. An efficient compression scheme for bitmap indices. Technical Report 49626, LBNL, April 2004.

[27] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. On the performance of bitmap indices for high cardinality attributes. Technical Report 54673, LBNL, March 2004.

[28] M. J. Zaki and J. T. L. Wang. Special issue on bioinformatics and biological data management. *Information Systems*, 28:241–367, 2003.