Minimizing Index Size by Reordering Rows and Columns*

Elaheh Pourabbas[1], Arie Shoshani[2], and Kesheng Wu[2]

[1] National Research Council, Roma, Italy,
[2] Lawrence Berkeley National Laboratory, Berkeley, CA, USA

Abstract. Sizes of compressed bitmap indexes and compressed data are significantly affected by the order of data records. The optimal orders of rows and columns that minimizes the index sizes is known to be NP-hard to compute. Instead of seeking the precise global optimal ordering, we develop accurate statistical formulas that compute approximate solutions. Since the widely used bitmap indexes are compressed with variants of the run-length encoding (RLE) method, our work concentrates on computing the sizes of bitmap indexes compressed with the basic Run-Length Encoding. The resulting formulas could be used for choosing indexes to build and to use. In this paper, we use the formulas to develop strategies for reordering rows and columns of a data table. We present empirical measurements to show that our formulas are accurate for a wide range of data. Our analysis confirms that the heuristics of sorting columns with low column cardinalities first is indeed effective in reducing the index sizes. We extend the strategy by showing that columns with the same cardinality should be ordered from high skewness to low skewness.

1 Introduction

Bitmap indexes are widely used in database applications [4, 16, 25, 30, 34]. They are remarkably efficient for many operations in data warehousing, On-Line Analytical Processing (OLAP), and scientific data management tasks [6, 10, 22, 26, 28, 29]. A bitmap index uses a set of bit sequences to represent the positions of the values as illustrated in Figure 1. In this small example, there is only a single column **X** in the data table, and this column **X** has only seven distinct values 0, 1, ..., 5. Corresponding to each distinct value, a bit sequence, also known as a bitmap, is used to record which rows have the specific value. This basic bitmap index requires $C \times N$ bits for a column with C distinct values and N rows. In the worst case, where every value is distinct, the value of C is N, this basic bitmap index requires N^2 bits, which is exceedingly large even for modest datasets. To reduce the index sizes, a bitmap index is typically compressed [4, 23, 30].

* The authors gratefully acknowledge the suggestion from an anonymous referee that clarifies Lemma 1. This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

RID	**X**	bitmaps					
		b_0 =0	b_1 =1	b_2 =2	b_3 =3	b_4 =4	b_5 =5
1	0	1	0	0	0	0	0
2	1	0	1	0	0	0	0
3	1	0	1	0	0	0	0
4	2	0	0	1	0	0	0
5	3	0	0	0	1	0	0
6	4	0	0	0	0	1	0
7	4	0	0	0	0	1	0
8	5	0	0	0	0	0	1

Fig. 1. The logical view of a sample bitmap index with its seven bitmaps shown as the seven columns on the right. In this case, a bit is 1 if the value of **X** in the corresponding row is the value associated with the bitmap.

The most efficient compression techniques for bitmap indexes are based on run-length encoding (RLE) [3, 30], which can be significantly affected by ordering of the rows [21, 28]. The best known strategy for ordering the columns is to place the column with the lowest cardinality first [21]. However, much of the earlier work only analyze the worst case scenario and is only applicable to uniform random data. In this work, we provide an analysis strategy that works for non-uniform distributions and therefore provide more realistic understanding of how to order the columns. We demonstrate that our formulas produce accurate estimates of the sizes. Our analysis also leads to a new ordering strategy: for columns with the same cardinality order the columns with high skew first.

2 Related Work

The size of a basic bitmap index can grow quickly for large datasets. The methods for controlling index sizes mostly fall in one of the following categories, compression [3, 7, 30], binning [19, 33] and bitmap encoding [9, 27, 32]. In this paper, we concentrate on compression. More specifically, how column and row ordering affects the sizes of compressed bitmap indexes. In this section, we provide a brief review of common compression techniques for bitmap indexes, and reordering techniques for minimizing these index sizes.

2.1 Compressing bitmap indexes

Any lossless compression technique could be used to compress a bitmap index. However, because these compressed bitmaps need to go through complex computations in order to answer a query, some compression methods are much more effective than others. In a simple case, we could read one bitmap and return it as the answer to a query, for example, the bitmap b_1 contains the answer to query condition "$0 < \mathbf{X} < 2$." However, to resolve the query condition "$\mathbf{X} > 2$," we need to bring bitmaps b_3 through b_6 into memory and then perform three

bitwise logical OR operations. In general, we may access many more bitmaps and perform many more operations.

To answer queries efficiently, we need to read the bitmaps quickly and perform the bitwise logical operations efficiently. We can not concentrate on the I/O time and neglect the CPU time. For example, the well-known method LZ77 can compress well and therefore reduce I/O time, however, the total time needed with LZ77 compression is much longer than with specialized methods [17, 31].

Among the specialized bitmap compression methods, the most widely used is the *Byte-aligned Bitmap Code* (BBC) [3], which is implemented in a popular commercial database management system. In tests, it compresses nearly as well as LZ77, but the bitwise logical operations can directly use the compressed bitmaps and therefore require less memory and less time [2, 17]. Another method that works quite well is the *Word-Aligned Hybrid* (WAH) code [30, 31]. It performs bitwise logical operations faster than BBC, but takes up more space on disk. This is because WAH works with 32-bit (or 64-bit) words while BBC works with 8-bit bytes. Working with a larger unit of data reduces the opportunity for compression, but the computations are better aligned with the capability of CPUs. Due to its effectiveness, a number of variations on WAH have also appeared in literature [11, 12, 14].

The key idea behind both BBC and WAH is *Run-Length Encoding* (RLE) that represents a sequence of identical bits with a count. They represent short sequences of mixed 0s and 1s literally, and BBC and the newer variants of WAH also attempt to pack some special patterns of mixed 0s and 1s. In the literature, each sequence of identical values is called a *run*. To enable a concise analysis, we choose to analyze the bitmap index compressed with RLE instead of BBC or WAH. In a straightforward implementation of RLE, one word is used to record each run. Therefore, our analysis focuses on the number of runs.

The commonly used bitmap compression methods such as BBC and WAH are more complex than RLE, and their compressed sizes are more difficult to compute. The existing literature has generally avoided directly estimating the compressed index sizes [18, 20, 21, 28]. In this work, we take a small step away from this general practice and seek to establish an accurate estimation for the sizes of RLE compressed bitmap indexes. We will show that our formulas are accurate and amenable to analysis.

2.2 Data Reordering Techniques

Reordering can improve the compression for data and indexes [1, 21]. Some of the earliest work on this subject was designed to minimize the number of disk accesses needed to answer queries by studying the consecutive retrieval property [13, 15]. Since the bitmap index can be viewed as a bit matrix, minimizing the index sizes is also related to the consecutive ones property [8]. These properties are hard to achieve and approximate solutions are typically used in practice.

A widely used data reordering strategy is to sort the data records in lexicographical order [5, 20, 24]. Many alternative ordering methods exist, one well-known example is the *Gray* code ordering. No matter how sorting is done, a

common question is which column to use first. There is a long history of publications on this subject. Here is a brief review of a few of them.

One of the earliest publications on this subject was by Olken and Rotem [24]. In that paper, the authors investigated both deterministic and probabilistic models of data distribution and determined that rearranging data to optimize the number of runs is NP-Complete via reduction to the Traveling Salesman Problem (TSP). Under a probabilistic model, the optimal rearrangement for each attribute has the form of a double pipe organ. The key challenge for implementing their recommendation is that the computational complexity grows quadratically with N, the number of rows in the dataset.

Pinar et al. [28] similarly converted the problem of minimizing the number of runs into a Traveling Salesman Problem. They suggested Gray code ordering as an efficient alternative to the simple lexicographical ordering or TSP heuristics, and presented some experimental measurements to confirm the claim.

Apaydin et al. considered two different types of bitmap indexes under lexicographical ordering and Gray code ordering [5]. They found that Gray code ordering gives slightly better results for the Range encoded bitmap index.

In a series of publications, Lemire and colleagues again proved that minimizing the number of runs is NP hard and affirmed that sorting the columns from the lower cardinality is an effective strategy [18, 20, 21].

All these analyses focus on the bit matrix formed by a bitmap index and consider essentially the worst case scenario, therefore are mostly applicable to uniform random data. In real word applications, the data is hardly ever uniform random numbers. Our work address this nonuniformity by developing an accurate approximation that can be evaluated analytically.

3 Theoretical Analysis

To make the analysis tasks more tractable, we count the number of runs, which is directly proportional to the size of a RLE compressed bitmap index. The key challenge is this approach is that even though the definition of a run only involves values of each individual column, the number of runs for one column depends on the columns sorted before it. To address this challenge, we develop the concept of *leading k-tuple* to capture the dependency among the columns. As we show next this concept can capture the expected number of runs and allows us to evaluate how the key parameters of data affects the expected number of runs and the index sizes.

3.1 Counting k-tuples

Without loss of generality, we concentrate our analysis of a table of integers with N rows and M columns. To avoid the need to construct a bitmap index and count the number of runs, we introduce a quantity that can be directly measured as follows.

X	Y	Z
10	20	30
10	22	33
11	20	31
11	21	30
11	21	32

A) Sort **X** first

Z	Y	X
30	20	10
30	21	11
31	20	11
32	21	11
33	22	10

B) Sort **Z** first

Fig. 2. A small data table sorted in two different ways.

Definition 1. *A* chunk *is a sequence of identical values of a column in consecutive rows.*

For the examples shown in Figure 2, in the version that sorted **X** first, the values of **X** form two chunks, one with the value 10 and the other with the value 11. In the version sorted **Z** first, the values of **X** form three chunks, two chunks with the value 10 and a chunk with the value 11. Note that a chunk always includes the maximum number of consecutive identical values. We do not break the three consecutive values into smaller chunks.

The two tables shown in Figure 2 are sorted with different column orders and have different number of chunks. To capture this dependency on column order, we introduce a concept called *leading k-tuple*.

Definition 2. *A leading k-tuple is a tuple of k values from the first k columns of a row.*

Depending how the columns are ordered, the leading k-tuples will be different. For example, the first leading 2-tuple in Fig. 2A is (**X**=10, **Y**=20) and the second leading 3-tuple in Fig. 2B is (**Z**=30, **Y**=21, **X**=11). In this paper, when we refer to a *k-tuple*, we only refer to a leading k-tuple, therefore we usually use the shorter term.

Without loss of generality, we refer to the first column in our reordering as column 1 and the kth column as column k. In Fig. 2A, column 1 is **X**, while in Fig. 2B, column 1 is **Z**. Similarly, we refer to the jth smallest value of column k as the value j without regards to its actual content.

An critical observation is captured in the following lemma.

Lemma 1. *The number of chunks for column k is bounded from above by the number of distinct leading k-tuples* T_k.

Typically, T_{k-1} is much smaller than T_k and the number of chunks for column k is very close to T_k. For this reason, we count the number of k-tuples instead of counting the number of chunks. We will discuss the difference between T_k and the number of chunks in Section 3.2.

Assume that the data table was generated through a stochastic process and the probability of a leading k-tuple $(j_1, j_2, \ldots, j_k)$ appearing in the data table is $p_{j_1 j_2 \ldots j_k}$. After generating N such rows, the probability that a particular k-tuple $(j_1, j_2, \ldots, j_k)$ is missing from the data table is $(1 - p_{j_1 j_2 \ldots j_k})^N$. Let C_1

denote the number of possible values for column 1, and C_k denote the number of possible values for column k. The total number of distinct k-tuples is $C_1 C_2 \dots C_k$. Summing overall all possible leading k-tuples, we arrive at the number of missing k-tuples as $\sum_{j_1 j_2 \dots j_k} (1 - p_{j_1 j_2 \dots j_k})^N$, and the number of distinct k-tuples in the data table as

$$T_k = \prod_{i=1}^{k} C_i - \sum_{j_1 j_2 \dots j_k} (1 - p_{j_1 j_2 \dots j_k})^N . \tag{1}$$

Note that the above formula works with the probability of k-tuples and is applicable to any data set, even where the columns exhibit correlation. In most cases, the probability of the k-tuples $p_{j_1 j_2 \dots j_k}$ is harder to obtain than the probability of an individual column. To make use of the probability distribution of the columns, we assume the columns are statistically independent, and $p_{j_1 j_2 \dots j_k} = p_{j_1} p_{j_2} \dots pj_k$. The above expression of T_k can be rewritten as:

$$T_k = \prod_{i=1}^{k} C_i - \sum_{j_1 j_2 \dots j_k} (1 - p_{j_1} p_{j_2} \dots p_{j_k})^N \tag{2}$$

Our goal is to count the number of runs in the bitmap index for each column of the data table. For the bitmaps shown in Fig. 1, roughly each chunk in the values of **X** leads to two runs in some bitmaps. We can generalize this observation as follows.

Lemma 2. *For a column with T_k chunks and C_k distinct values, the number of runs in the bitmap index is $2T_k + C_k - 2$.*

Proof. For each chunk in column k, the corresponding bitmap in the bitmap index will have a sequence of 0s followed by a sequences of 1s. This leads to the term $2T_k$ runs for T_k chunks. For most of the C_k bitmaps in the bitmap index, there is a sequences of 0s at the end of the bitmap. Altogether, we expect $2T_k + C_k$ runs. However, there are two special cases. Corresponding the first chunk in the values of column k, there is no 0 before the corresponding 1s. Corresponding to the last chunk, there is no 0s at the end of the bitmap. Thus, there are two less runs than expected. The total number of runs is $2T_k + C_k - 2$.

3.2 Accidental chunks

As a sanity check, we next briefly consider the case where all columns are uniformly distributed. This also helps us to introduce the second concept we call the *accidental chunks* which captures the error of using the number of leading k-tuples to approximate the number of chunks for column k.

Assuming the probability of each value is the same, i.e., $p_{j_i} = C_i^{-1}$, we can significantly simplify the above formula as

$$T_k = \left(1 - \left(1 - \prod_{j=1}^{k} C_j^{-1} \right)^N \right) \prod_{j=1}^{k} C_j \tag{3}$$

Table 1. Example of sorting 1 million rows (N=1,000,000) with column cardinality from lowest to highest.

C	*Max chunks*	*Exp chunks*	*Max runs*	*Exp runs*	*Actual runs*	*Error (%)*
10	10	10	28	28	28	0
20	200	200	418	418	418	0
40	8000	8000	16038	16038	16038	0
60	480000	420233	960058	840524	841142	0.074
80	38400000	987091	76800078	1974260	1966758	-0.38
100	3840000000	999869	7680000098	1999836	1980078	-1.00

Table 2. Example of sorting 1 million rows (N=1,000,000) with column cardinality from highest to lowest.

C	*Max chunks*	*Exp chunks*	*Max runs*	*Exp runs*	*Actual runs*	*Error (%)*
100	100	100	298	298	298	0
80	8000	8000	16078	16078	16078	0
60	480000	420233	960058	840524	840584	0.007
40	19200000	974405	38400038	1948848	1934192	-0.0075
20	384000000	998699	768000018	1997416	1898906	-4.93
10	3840000000	999869	7680000008	1999746	1800250	-9.976

We generated one million rows of uniform random numbers and actually counted the number of chunks and number of runs; the results are shown in Tables 1 and 2. The actual observed chunks include identical values appearing contiguously crossing the k-tuple boundaries. Even though two k-tuples may be different, the values of the last column, column k, could be the same. For example, in Fig. 2B, the three row in the middle all have $\mathbf{X}$=1, even though the corresponding 3-tuple are different. This creates what we call *accidental chunks*.

Definition 3. *An accidental chunk in column k is a group of identical values for column k where the corresponding k-tuples are different.*

In general, we count a chunk for column k as an *accidental chunk*, if the values of the kth column are the same, but the leading $(k-1)$-tuples are different. After sorting, the leading $(k-1)$-tuples are ordered and the identical tuples are in consecutive rows. Since the column k is assumed to be statistically independent from the first $(k-1)$ columns, we can compute the number of consecutive identical values as follows.

Starting from an arbitrary row, the probability of the column k being j_k is p_{j_k} and the probability that there is only a single j_k (followed by something else) is $p_{j_k}(1-p_{j_k})$. The probability that there are two consecutive rows with j_k is $p_{j_k}^2(1-p_{j_k})$, and the probability for q consecutive j_k is $p_{j_k}^q(1-p_{j_k})$. The numbers of time j_k appears together is:

$p_{j_k}(1-p_{j_k}) + 2p_{j_k}^2(1-p_{j_k}) + 3p_{j_k}^3(1-p_{j_k}) + \ldots + (N-1)p_{j_k}^{N-1}(1-p_{j_k}) + Np_{j_k}^N$
$\cong p_{j_k}(1-p_{j_k}) \sum_{i=1}^{\infty} ip_{j_k}^{i-1} = p_{j_k}(1-p_{j_k})(1-p_{j_k})^{-2} = p_{j_k}(1-p_{j_k})^{-1}$,

where[3] $\sum_{i=1}^{\infty} ip^{i-1} = \sum_{i=1}^{\infty} \frac{\partial p^i}{\partial p} = \frac{\partial(\sum_{i=1}^{\infty} p^i)}{\partial p} = \frac{\partial(1-p)^{-1}}{\partial p} = (1-p)^{-2}$. The average times a value of column k repeats is $\mu_k = \sum_{j_1 j_2 \ldots j_k} \frac{p_{j_k}}{1-p_{j_k}}$.

The set of consecutive values in column k must span beyond the group of identical $(k-1)$-tuples in order to be counted as an accidental chunk. When the majority of the chunks for the first $(k-1)$ columns have only 1 row, the number of chunks for column k is reduced by a factor $1/\mu_k$.

For uniform random data, we can estimate the values of μ_k and check whether they agree with the observations from Tables 1 and 2. These tables show an example of sorting 1 million tuples with column cardinality ordered from the lowest to the highest and from the highest to the lowest, respectively. In Table 1, we see that about 99% of 5-tuples are distinct, more precisely, there are 987091 5-tuples for 1 million rows. This suggests that there might be a noticeable number of accidental chunks for column 6 shown in the last row in Table 1. In this case, the cardinality of column 6 is 100, $p_{j_6} = 1/100$ and $\mu_6 = \sum_{j_6=1}^{100} \frac{1/100}{1-1/100} = \frac{100}{99}$. The number of chunks observed should be T_6/μ_6, which is 1% less than the expected value of T_6. The number of runs in the bitmaps is 1% less the expected value in Table 1, which agree with our analysis. Similarly, in Table 2 about 97% of the 4-tuples are distinct, which suggests that there might be noticeable number of accidental chunks for columns 5 and 6. The cardinality of the last two columns are 20 and 10 respectively, and our formula suggests that the observed number of chunks would be 5% and 10% less than the expected value of T_5, T_6. In Table 2, we again see a good agreement with the predictions[4].

3.3 Asymptotic case

Assume the value of each $p_{j_1 j_2 \ldots j_k}$ to be very small, say $p_{j_1 j_2 \ldots j_k} \ll 1/N$. In this case, we can approximate the probability that the k-tuple $(j_1, j_2, \ldots, j_k)$ not appearing in our dataset as $(1 - p_{j_1 j_2 \ldots j_k})^N \approx 1 - Np_{j_1 j_2 \ldots j_k}$. This leads to the following approximation for the number of distinct k-tuples.

$$\begin{aligned} T_k &\approx \prod_{i=1}^{k} C_i - \sum_{j_1 j_2 \ldots j_k} (1 - Np_{j_1 j_2 \ldots j_k}) \\ &= \prod_{i=1}^{k} C_i - \sum_{j_1 j_2 \ldots j_k} 1 + N \sum_{j_1 j_2 \ldots j_k} p_{j_1 j_2 \ldots j_k} \\ &= N \sum_{j_1 j_2 \ldots j_k} p_{j_1 j_2 \ldots j_k} \quad = \quad N. \end{aligned}$$

[3] http://mathworld.wolfram.com/PowerSum.html

[4] Lemma 1 seems to suggest that the errors in Tables 1 and 2 can only be negative or zero, however there are a few positive numbers. The reason for this is that the number of chunks and runs given are based on the expected number of leading k-tuples, not the number of leading k-tuple observed in the particular test dataset.

In other word, every k-tuple will be distinct. If the probability of each individual k-tuple is very small, we intuitively expect all the observed tuples to be distinct. We generalize this observation and state it more formally as follows.

Definition 4. *A tuple $(j_1 j_2 \dots j_k)$ in a dataset with N tuples is a rare tuple if $p_{j_1 j_2 \dots j_k} < 1/N$.*

In most discussions, we will simply refer to such a tuple as a rare tuple, without referring to the number of rows, N.

***Conjecture on rare tuples**: A rare tuple in a dataset will appear exactly once if it does appear in the dataset.*

In a typical case, there is a large number of possible tuples and many of them with very small probabilities while a few tuples with larger probabilities. Therefore, we cannot apply the above estimate to the whole dataset. The implication from the above conjecture is that the rare tuples that do appear in a data set will be different from others. To determine the total number of distinct tuples, we can concentrate on those tuples that appear more frequently, which we call *common tuples*.

Definition 5. *A tuple $(j_1 j_2 \dots j_k)$ in a dataset with N tuples is a common tuple if $p_{j_1 j_2 \dots j_k} \geq 1/N$.*

In most discussions, we will simply refer to such a tuple as a common tuple, without referring to the number of rows, N.

3.4 Zipfian Data

In the preceding sections, we have demonstrated that our formulas predict the numbers of runs accurately for uniform data and rare tuples. Next, we consider the more general case involving data with non-uniform distribution and a mixture of rare tuples and common tuples. In order to produce compact formulas, we have chosen to concentrate on data with Zipf distributions. We further assume that each column of the data table is statistically independent from others.

Following the above analysis on rare tuples, we assume that all rare tuples that do appear in a dataset are distinct. A common tuple may appear more than once in a dataset, we say that it has duplicates. More specifically, if a k-tuple appears q times, then it has $(q-1)$ duplicates. Let D_k denote the number of duplicates, the number of distinct k-tuples in the dataset is $T_k = N - D_k$. This turns the problem of counting the number of distinct values into counting the number of duplicates. To illustrate this process, let us first consider a case of 1-tuple, i.e., one column following the Zipf distribution $p_{j_1} = \alpha_1 j_1^{-z_1}$, where $\alpha_1 = \left(\sum_{j_1=1}^{C_1} j_1^{-z_1}\right)^{-1}$ and z_1 is a constant parameter known the Zipf exponent.

By definition of p_{j_1}, the value j_1 is expected to appear Np_{j_1} times in that dataset. The common values are expected to appear at least once, i.e., $Np_{j_1} = N\alpha_1 j_1^{-z_1} \geq 1$. Since p_{j_1} is a monotonically decreasing function of j_1, common values are smaller than rare ones. Let $\beta_1 \equiv (N\alpha_1)^{1/z_1}$, we see that all j_1 values

less than C_1 and $\lfloor \beta_1 \rfloor$ (where $\lfloor . \rfloor$ is the floor operator) are common values. The number of duplicates can be expressed as

$$D_1 = \sum_{j_1=1}^{min(C_1, \lfloor \beta_1 \rfloor)} \left(N \alpha_1 j_1^{-z_1} - 1 \right). \quad (4)$$

For the convenience of later discussions, we define two functions

$$s_1 = \sum_{j_1=1}^{min(C_1, \lfloor \beta_1 \rfloor)} j_1^{-z_1}, \qquad r_1 = \sum_{min(C_1, \lfloor \beta_1 \rfloor)+1}^{C_1} j_1^{-z_1}.$$

By the definition of α_1, we have $\alpha_1 = r_1 + s_1$. Furthermore, the number of common values is $N\alpha_1 s_1$, the number of rare values is $N\alpha_1 r_1$, and the number of distinct values is $T_1 = min(C_1, \lfloor \beta_1 \rfloor) + N\alpha_1 r_1$. Among all possible values of β_1, when $\beta_1 < 1$, there is no common value and $T_1 = N$; when $\beta_1 \geq C_1$, all values are common values and $T_1 = C_1$.

In the more general case where the kth column has cardinality C_k and Zipf exponent z_k, we have $\alpha_k = 1/\sum_{j_k=1}^{C_k} j_k^{-z_k}$, the number of distinct k-tuples and the number of duplicate k-tuples are given by the following expressions,

$$T_k = N - D_k, D_k = \sum_{N p_{j_1 \ldots j_k} > 1} (N p_{j_1 \ldots j_k} - 1), p_{j_1 \ldots j_k} = \prod_{i=1}^{k} p_i = \prod_{i=1}^{k} \alpha_i j_i^{-z_i}. \quad (5)$$

Since the Zipf distribution is a monotonic function, it is straightforward to determine the bounds of the sum in Equation (5) as illustrated in the one-column case above. Given j_1, ..., j_{k-1}, the upper bound for j_k is given by $\left(N \alpha_k \prod_{i=1}^{k-1} \alpha_i j_i^{-z_i} \right)^{1/z_k}$. In many cases, there are a relatively small number of common tuples, which allows us to evaluate the above expression efficiently. From this expression, we can compute the number of k-tuples and therefore the number of runs in the corresponding bitmap indexes.

Theorem 1. *Let $C_1, C_2, \ldots, C_M$ denote column cardinalities of a data table. Assume all columns have the same skewness as measured by the Zipf exponents. To minimize the total number of runs in the bitmap indexes for all columns with sorting, the lowest cardinality column shall be sorted first.*

Proof. Let's first consider the case of 1-tuple. Given z_1, the summation $\sum_{j_1}^{C_1} j_1^{-z_1}$ increases as C_1 increases. The values of α_1 decreases as C_1 increases (as illustrated in Fig. 3A) which can be expressed as $\partial \alpha_1 / \partial C_1 < 0$. This leads to $\partial \beta_1 / \partial C_1 = \frac{1}{z_1} (N\alpha_1)^{(\frac{1}{z_1} - 1)} N \frac{\partial \alpha_1}{\partial C_1} < 0$, which means that β_1 decreases as C_1 increases (as shown in Fig. 3B). In the expression for D_1, the upper bound of the summation is the minimal of C_1 and $\lfloor \beta_1 \rfloor$.

When $C_1 \leq \lfloor \beta_1 \rfloor$, all C_1 values are common values. In these cases, by definition of the Zipfian distribution $\sum_{j_1=1}^{C_1} N \alpha_1 j_1^{-z_1} = N$, the number of duplicates $D_1 = N - C_1$ (see Eq. (4)) and the number of distinct values is C_1.

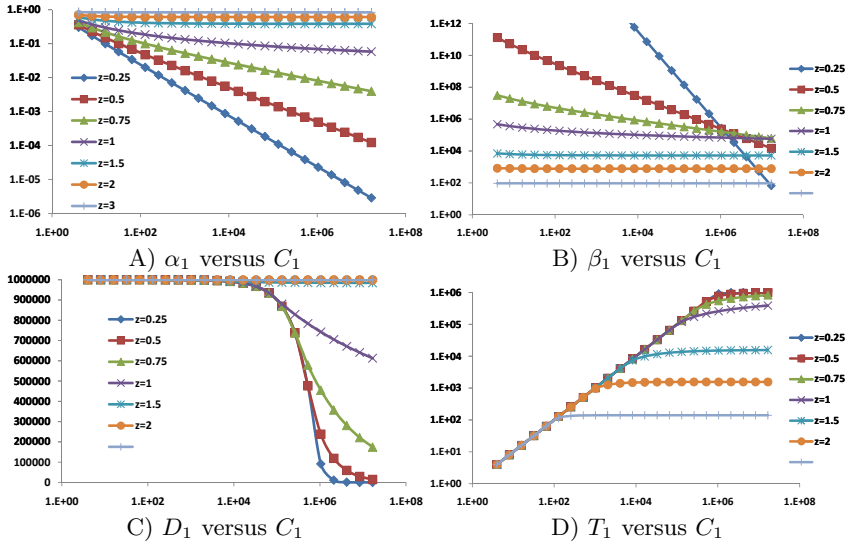


Fig. 3. How the values vary with column cardinality with fixed skewness (assuming 1 million rows).

In cases where $C_1 > \lfloor \beta_1 \rfloor$, the number of distinct values may be less than C_1. Because both α_1 and β_1 decrease with the increase of C_1, each term in the summation for D_1 decreases and the number of terms in the summation may also decrease, all causing D_1 to decrease as C_1 increases, as shown in Fig. 3C. In other words, as C_1 increases, there are fewer duplicates and more distinct values. As shown in Fig. 3D, the value of T_1 increases with C_1.

Now, we consider the case of k-tuple. In this case, the probability for a k-tuple is $p_{j_1 j_2 \ldots j_k} = \alpha_1 \alpha_2 \ldots \alpha_k j_1^{-z} j_2^{-z} \ldots j_k^{-z}$, the common tuples are those with $\alpha_1 \alpha_2 \ldots \alpha_k j_1^{-z} j_2^{-z} \ldots j_k^{-z} \geq 1/N$, or alternatively, $j_1 j_2 \ldots j_k \leq (\alpha_1 \alpha_2 \ldots \alpha_k N)^{1/z}$. Note that the values $j_1 j_2 \ldots j_k$ are positive integers.

Along with the conditions that $1 \leq j_1 \leq C_1, \ldots, 1 \leq j_k \leq C_k$, the number of duplicate tuples can be expressed as follows ($\beta_k \equiv (\alpha_1 \alpha_2 \ldots \alpha_k N)^{1/z}$):

$$D_k = \sum_{j_1=1}^{min(C_1, \lfloor \beta_1 \rfloor)} \sum_{j_1=1}^{min(C_2, \lfloor \beta_2 j_1^{-1} \rfloor)} \cdots \sum_{j_1=1}^{min(C_k, \lfloor \beta_k j_1^{-1} j_2^{-1} \ldots j_{k-1}^{-1} \rfloor)} \left(N \alpha_1 \alpha_2 \ldots \alpha_k j_1^{-z} j_2^{-z} \ldots j_k^{-z} - 1 \right). \quad (6)$$

In the above expression, the order of the column among the k-tuple does not change the number of distinct tuples. Therefore, when all columns are considered together, i.e., $k = M$, it does not matter how the columns are ordered. However, as soon as one column is excluded, say, $k = M - 1$, it does matter which columns

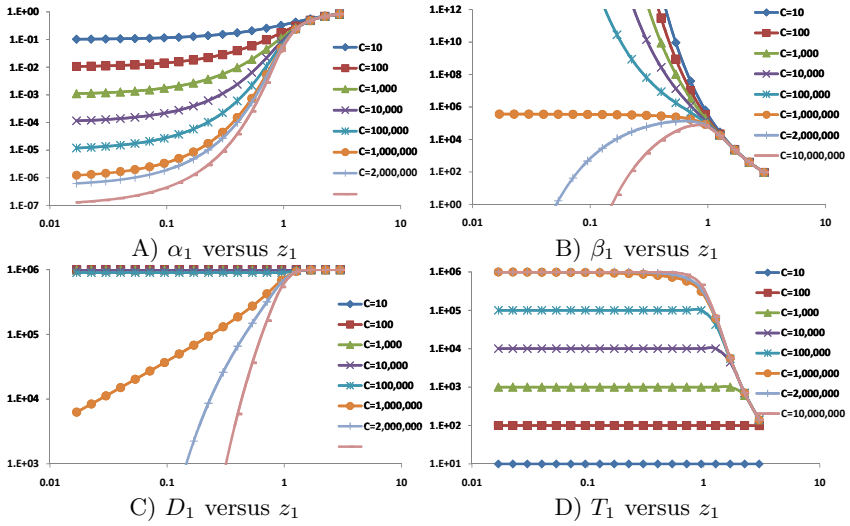


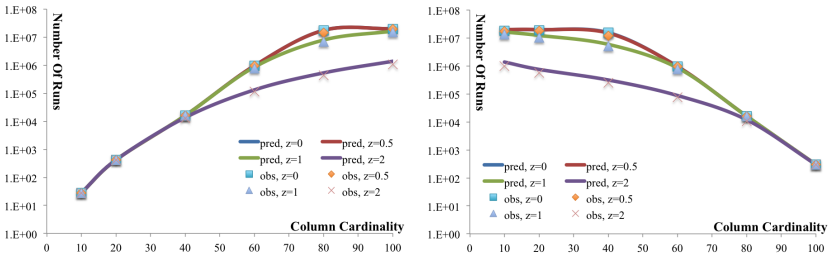
Fig. 4. How the values vary with skewness with fixed column cardinality (assuming 1 million rows).

are excluded. Assume that we have two columns to choose from, say columns A and B; and the only different between them is their column cardinalities, C_A and C_B. Without loss of generality, assume $C_A > C_B$, consequently, $\alpha_A < \alpha_B$ and $\beta_A < \beta_B$. In the formula D_k, a smaller β_A value indicates that the number of terms in the summation would be no more than that with a larger β_B. For each term, replacing the value of α_k with α_A will produce a smaller value than replacing it with α_B. Therefore, choosing the higher cardinality column decreases D_k, increases the number of distinct tuples and increases the number of runs in the corresponding bitmap index. Thus, sorting the lowest cardinality column first reduces the number of distinct k-tuples and minimizes the number of runs.

Theorem 2. *Let $C_1, C_2, \ldots, C_M$ denote the column cardinalities of a data table. Assume $C_1 = C_2 = \ldots = C_M$. To minimize the total number of runs in the bitmap indexes by sorting, the column with the largest Zipf exponent shall be ordered first.*

Proof. Instead of giving a complete proof here, we will outline the basic strategy. Based on the information shown Fig. 4, particularly Fig. 4B, we need to handle the cases with $C > N$ and $z < 1$ separately from the others. In the normal cases, α_1, β_1, D_1 and T_1 are all monotonic functions and it is clear that sorting columns with larger Zipf exponents first is beneficial.

In the special case with $C > N$ and $z < 1$, where the possible values to use is large and the differences among the probabilities of different values are relatively small, the number of runs for a column is the same as that of a uniformly random



A) Columns with the same skewness ordered from low cardinality to high cardinality

B) Columns with the same skewness ordered from high cardinality to low cardinality

Fig. 5. Predicted numbers of runs (lines) and the observed numbers of runs (as symbols) plotted against the column cardinality (10 million rows). The symbols are very close to their corresponding lines indicating that the predictions agree well with the observations.

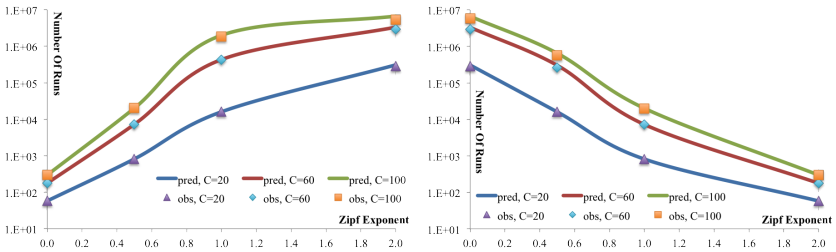
column. In which case, which column comes first does not make a difference, and we can follow the general rule derived for the normal cases. Thus, we always sort the column with the largest Zipf exponent first.

4 Experimental Measurements

In the previous section, we took the expected number of distinct k-tuple as an estimate of the number of chunks and therefore the number of runs in a bitmap index. In this section, we report a series of empirical measurements designed to address two issues: (1) how accurate are the formulas for predicting numbers of runs for Zipf data and (2) do the reordering strategies actually reduce index sizes? In these tests, we use a set of synthetic Zipfian data with varying cardinalities and Zipf exponents. The column cardinalities used are 10, 20, 40, 60, 80, and 100; the Zipf exponents used are 0 (uniform random data), 0.5, 1, and 2 (highly-skew data). The test data sets contain 10 million rows, which should be large enough to avoid significant statistical errors.

4.1 Number of runs

The first set of measurements are the numbers of runs predicted by Eq. (5) and the numbers of runs actually observed on a set of Zipf data. We also collected the expected and the actual numbers of runs with different ordering of the columns. We first organize the synthetic data into four tables where all columns in each table have the same Zipf exponent. In Fig. 5, we display the numbers of runs for each bitmap index (for an individual column) with the data table sorted in two different column orders, the lowest cardinality column first or the highest cardinality column first. In this figure, the discrete symbols denotes the observed



A) Columns with the same cardinalities are ordered from small Zipf exponent to large Zipf exponent

B) Columns with the same cardinalities are ordered from large Zipf exponent to small Zipf exponent

Fig. 6. Predicted numbers of runs (lines) and observed numbers of runs (symbols) plotted against the Zipf exponents (10 million rows). The symbols are very close to their corresponding lines indicating that the predictions agree well with the observations.

Table 3. Total number of runs (in thousands) of columns with the same skew in two different orders (N=10,000,000).

Skew	*Total numbers of runs*			
z	*Low cardinality first* (Small per Thm 1)		*High cardinality first*	
	predicted	observed	predicted	observed
0	38,559	38,382	56,281	53,545
0.5	38,506	35,266	55,904	48,988
1	25,254	22,328	35,629	29,523
2	2,065	1,639	2,557	1,892

values and the lines depict the theoretical predictions developed in the previous section. The number of runs vary from tens to tens of millions. In this large range of values, our predictions are always very close to the actual observations.

Fig. 6 shows similar predicted numbers of runs against observed values for data tables containing columns with the same column cardinality. Again we see that the number of runs vary from tens to millions, and the observed values agree with the predictions very well.

To see exactly how accurate are our predictions, in Table 3 and 4, we listed out the total number of runs for each of the data tables used to generate Fig. 5 and 6. In these two tables, the total numbers of runs are reported in thousands. As we saw in Table 1 and 2 for uniform random data, the predictions are generally slightly larger than the actually observed values. The discrepancy appears to grow as the skewness of the data grows or the column cardinality grows. We believe this discrepancy to be caused by the accidental chunks discussed in Section 3.2. We plan to verify this conjecture in the future.

Since the actual number of runs are different from the expected values, a natural question is whether the predicted advantage of column ordering still

Table 4. Total number of runs (in thousands) of columns with the same column cardinality ordering in two different ways (N=10,000,000).

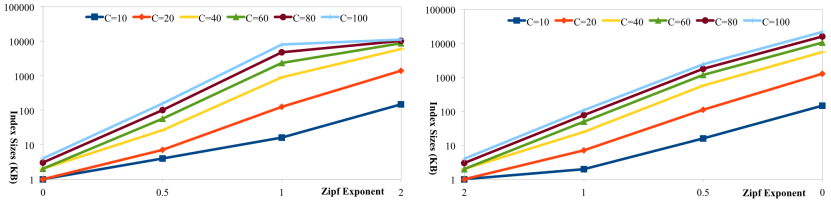
Cardinality	*Total numbers of runs*			
C	*Low skew first*		*High skew first* (Small per Thm 2)	
	predicted	observed	predicted	observed
10	22	22	22	22
20	326	301	326	301
40	1864	1636	1860	1622
60	3767	3335	3638	3173
80	6016	5274	5472	4776
100	8706	7274	7353	6331

Table 5. The total sizes (KB) of compressed bitmap indexes produced by FastBit under different sorting strategies. Each data table has 10 million rows and four columns with the same column cardinality but different skewness.

C	10	20	40	60	80	100
Low skew first	168	1,540	6,911	11,188	15,146	19,487
High skew first	166	1,393	6,125	11,870	17,913	23,878

observed. In Table 3, we see that the total number of runs of tables sorted with the lowest cardinality column first is always smaller than the same value for the same table sorted with the largest cardinality column first. This is true even for highly-skew data (with $z = 2$), where the predicted total number of runs is nearly 35% higher than the actual observed value ($2065/1639 = 1.26$, $2557/1892 = 1.35$). In this case, sorting the the highest cardinality column first produces about 15% more runs than sorting the lowest cardinality column first ($1892/1639 = 1.15$). The predicted advantage is about 24% ($2557/1639 = 1.24$). Even though it may be worthwhile to revisit the source of error in our predictions, the predicted advantage from Theorems 1 and 2 are clearly present.

From our analysis, we predicted that ordering the columns with the highest skew first is the better than other choices. In Table 4, we show the total numbers of runs from two different sorting strategies, one with the highest skew first and the other with the least skew first. For the columns with relative low column cardinalities, there are enough rows to produce all possible tuples. In this case, the prediction is exactly the same as the observed values. As the column cardinality increases, there are larger differences between the predictions and observations. With each column having 100 distinct values, the difference is almost 20%. However, even in this case, the predicted advantage of sorting the column with the largest Zipf exponent first is still observable in the test.



A) Columns with the same cardinalities ordered from small Zipf exponent to large Zipf exponent

B) Columns with the same cardinalities ordered from large Zipf exponent to small Zipf exponent

Fig. 7. Sizes of FastBit indexes ($N = 10,000,000$)

4.2 FastBit index sizes

Even though commonly used bitmap compression methods are based on RLE, they are more complex than RLE and therefore our predictions may have larger errors. To illustrate this point, we show the sizes of a set of bitmap indexes produced by an open-source software called FastBit [29] in Fig. 7 and Table 5. In these tests, our test data is divided into six tables with four columns each to produce the sizes shown in Fig. 6 and Table 4. According to Theorem 2, ordering highly-skewed column first is expected to produce smaller compressed bitmap indexes. In Table 5, we see that the prediction is true for three out of the six data tables, those with $C = 10$, $C = 20$, and $C = 40$.

For data tables with high column cardinalities, the predictions are wrong and the values shown in Fig. 7 offers some clues as why. In general, the last column to be sorted is broken into more chunks, the corresponding bitmap index has more runs and requires more disk space. This index typically requires significantly more space than those of the earlier columns, and therefore dominates the total index size. In Fig. 7B, we see that the index sizes grow steadily from the column being sorted first to the column being sorted last. However, in Fig. 7A, we see the index size for column sorted last did not grow much larger than the previous columns. This is especially noticeable for $C = 100$, $C = 80$ and $C = 60$. More work is needed to understand this trend.

5 Conclusions

In this paper, we developed a set of formulas for the sizes of Run-Length Encoded bitmap indexes. To demonstrate the usefulness of these formulas, we used them to examine how to reorder the rows and columns of data table. Our analysis extends the reordering heuristics to include non-uniform data. We demonstrated that the formulas are indeed accurate for a wide range of data. We also discussed the limitations of the proposed approach. In particular, because the practical bitmap compression methods are not simple Run-Length Encoding methods,

their index sizes deviate from the predictions in noticeable ways. We plan to explore options to capture these deviations in a future study.

References

1. Abadi, D., Madden, S.R., Ferreira, M.C.: Integrating compression and execution in column-oriented database systems. In: SIGMOD. ACM (2006)
2. Amer-Yahia, S., Johnson, T.: Optimizing queries on compressed bitmaps. In: VLDB. pp. 329–338 (2000)
3. Antoshenkov, G.: Byte-aligned bitmap compression. Tech. rep., Oracle Corp. (1994)
4. Antoshenkov, G., Ziauddin, M.: Query processing and optimization in oracle rdb. The VLDB Journal 5, 229–237 (1996)
5. Apaydin, T., Tosun, A.S., Ferhatosmanoglu, H.: Analysis of basic data reordering techniques. In: SSDBM. Lecture Notes in Computer Science, vol. 5069, pp. 517–524 (2008)
6. Bookstein, A., Klein, S.: Using bitmaps for medium sized information retrieval systems. Information Processing & Management 26, 525–533 (1990)
7. Bookstein, A., Klein, S., Raita, T.: Simple bayesian model for bitmap compression. Information Retrieval 1(4), 315–328 (2000)
8. Booth, K.S., Lueker, G.S.: Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. Journal of Computer and System Sciences 13(3), 335 – 379 (1976), http://dx.doi.org/10.1016/S0022-0000(76)80045-1
9. Chan, C.Y., Ioannidis, Y.E.: Bitmap index design and evaluation. In: SIGMOD. pp. 355–366 (1998)
10. Chaudhuri, S., Dayal, U., Ganti, V.: Database technology for decision support systems. Computer 34(12), 48–55 (2001)
11. Colantonio, A., Pietro, R.D.: Concise: Compressed 'n' composable integer set. Information Processing Letters 110(16), 644–650 (2010), http://dx.doi.org/10.1016/j.ipl.2010.05.018
12. Deliège, F., Pedersen, T.B.: Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In: EDBT '10: Proceedings of the 13th International Conference on Extending Database Technology. pp. 228–239. ACM, New York, NY, USA (2010)
13. Deogun, J.S., Gopalakrishnan, K.: Consecutive retrieval property–revisited. Information Processing Letters 69(1), 15 – 20 (1999), http://dx.doi.org/10.1016/S0020-0190(98)00186-0
14. Fusco, F., Stoecklin, M.P., Vlachos, M.: NET-FLi: on-the-fly compression, archiving and indexing of streaming network traffic. Proc. VLDB Endow. 3, 1382–1393 (September 2010), http://portal.acm.org/citation.cfm?id=1920841.1921011
15. Ghosh, S.P.: File organization: the consecutive retrieval property. Commun. ACM 15, 802–808 (September 1972), http://doi.acm.org/10.1145/361573.361578
16. Hu, Y., Sundara, S., Chorma, T., Srinivasan, J.: Supporting RFID-based item tracking applications in oracle DBMS using a bitmap datatype. In: VLDB'05. pp. 1140–1151 (2005)
17. Johnson, T.: Performance of compressed bitmap indices. In: VLDB'99. pp. 278–289 (1999)

18. Kaser, O., Lemire, D., Aouiche, K.: Histogram-aware sorting for enhanced word-aligned compression in bitmap indexes. In: DOLAP '08. pp. 1–8. ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1458432.1458434
19. Koudas, N.: Space efficient bitmap indexing. In: CIKM. pp. 194–201 (2000)
20. Lemire, D., Kaser, O., Aouiche, K.: Sorting improves word-aligned bitmap indexes. Data & Knowledge Engineering 69(1), 3–28 (2010), http://dx.doi.org/10.1016/j.datak.2009.08.006
21. Lemire, D., Kaser, O.: Reordering columns for smaller indexes. Information Sciences 181(12), 2550–2570 (2011), http://dx.doi.org/10.1016/j.ins.2011.02.002
22. Lin, X., Li, Y., Tsang, C.P.: Applying on-line bitmap indexing to reduce counting costs in mining association rules. Information Sciences 120(1-4), 197–208 (1999)
23. MacNicol, R., French, B.: Sybase IQ multiplex-designed for analytics. In Proceedings of 13th International Conference on Very Large Data Bases, VLDB'04, Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Rene J. Miller, J. A. Blakeley, and K. Bernhard Schiefer, Editors, August 31 - September 3 pp. 1227–1230 (2004)
24. Olken, F., Rotem, D.: Rearranging data to maximize the efficiency of compression. In: PODS. pp. 78–90. ACM Press (1985)
25. O'Neil, P.: Model 204 architecture and performance. In: 2nd International Workshop in High Performance Transaction Systems, Asilomar, CA, Springer Verlag Lecture Notes in Computer Science. vol. 359, pp. 40–59 (1987)
26. O'Neil, P.: Informix indexing support for data warehouses. Database Programming and Design 10(2), 38–43 (1997)
27. O'Neil, P., Quass, D.: Improved query performance with variant indices. In: SIGMOD. pp. 38–49. ACM Press (1997)
28. Pinar, A., Tao, T., Ferhatosmanoglu, H.: Compressing bitmap indices by data reorganization. In: ICDE'05. pp. 310–321 (2005)
29. Wu, K.: FastBit: an efficient indexing technology for accelerating data-intensive science. Journal of Physics: Conference Series 16, 556–560 (2005), http://dx.doi.org/10.1088/1742-6596/16/1/077
30. Wu, K., Otoo, E., Shoshani, A.: Optimizing bitmap indices with efficient compression. ACM Transactions on Database Systems 31, 1–38 (2006)
31. Wu, K., Otoo, E., Shoshani, A., Nordberg, H.: Notes on design and implementation of compressed bit vectors. Tech. Rep. LBNL/PUB-3161, Lawrence Berkeley National Lab, Berkeley, CA (2001), http://www-library.lbl.gov/docs/PUB/3161/PDF/PUB-3161.pdf
32. Wu, K., Shoshani, A., Stockinger, K.: Analyses of multi-level and multi-component compressed bitmap indexes. ACM Transactions on Database Systems 35(1), 1–52 (2010), http://doi.acm.org/10.1145/1670243.1670245
33. Wu, K., Stockinger, K., Shosani, A.: Breaking the curse of cardinality on bitmap indexes. In: SSDBM'08. pp. 348–365 (2008), preprint appeared as LBNL Tech Report LBNL-173E
34. Wu, M.C., Buchmann, A.P.: Encoded bitmap indexing for data warehouses. In: ICDE'98. pp. 220–230. IEEE Computer Society, Washington, DC, USA (1998)