LBNL-47807

Compressed bitmap indices for efficient query processing[*]

Kesheng Wu   Ekow J. Otoo   Arie Shoshani

Abstract

Bitmap indices are useful techniques for improving access speed of high-dimensional data in data warehouses and in large scientific databases. Even though the bitmaps are easy to compress, compressing them can significantly reduce the query processing efficiency. This is because the operations on the compressed bitmaps are much slower than the same operations on the uncompressed ones. To address this problem, we developed a new word-aligned compression scheme that uses a little more space, but is much faster than earlier methods. Using this compression technique, we evaluate several bitmap encoding schemes, including the equality encoding, the range encoding and the interval encoding. In our tests, the compressed bitmap indices are not only much smaller in size than their uncompressed versions, but are also just as fast in query processing as their uncompressed counterparts. In these tests, the compressed bitmap indices using different encoding schemes show different space and time trade-offs. To search for more useful choices, we also present several two-level schemes.

1 Introduction

The bitmap index is one of the most promising strategy for indexing high-dimensional data arising in such environments as data warehousing, decision support systems, and some scientific databases. One of the first database systems to use such a scheme is a system called Model 204 [10]. Most of the major commercial database systems now support some form of a bitmap index. In the research community, the earliest forms of the bitmap indices are known by such names as bit transposed files [15]. Some research results on signature files are also directly useful to enhancing the effectiveness of bitmap indices. Recently, a number of optimal bitmap schemes have also been proposed [3, 4, 18]. For high dimensional data, various tests have shown that bitmap schemes are faster than commonly used tree based indices [9, 14].

Our current work was initially motivated by the data management needs of a high-energy physics project called STAR[1] [12, 13]. The high-energy collisions, called "events", produce a large amount of data. The high energy physics experiment produces millions of events and each event is characterized by hundreds of attributes. In a typical use case, the user may specify some range criteria on a handful of the attributes and request that the data for the qualified events be retrieved. The bitmap index is a particularly promising strategy for these types of data accesses. There are some initial successes in using this strategy [12, 13]. Here we report our recent experience in using various compressed versions of the bitmap indices.

The main advantage for using a compressed bitmap index is to reduce the space requirement. The classical bitmap index produces one bitmap for each distinct value of the attribute being indexed. The size of the indices could be much larger than the size of the dataset. This is especially

[*]This work was supported by the Director, Office of Science, Office of Laboratory Policy and Infrastructure Management, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

[1]Information about the STAR project is available on the web at `http:://www.star.bnl.gov/STAR`.

true for scientific databases where most of the attributes have high cardinality. However, the bitmaps from the bitmap indices are often very sparse, i.e., they contain mostly zero bits. They are therefore prime candidates for compression [4, 6]. The common compression schemes, such as gzip [8] and bzip2 [11], aren't designed for compressing bitmap indices. If a bitmap index is compressed using such a scheme, the query processing usually takes much longer than using the uncompressed index. One solution to this problem is to use specially designed compression schemes.

Recently, a number of studies were performed on compression schemes especially designed for bitmap indices [5]. One of the most promising compressing scheme is the byte-aligned bitmap code (BBC) [1, 2]. This scheme permits efficient operations without decompression, thereby reducing both the disk space requirement and the memory requirement for performing operations. The question we address in this paper is whether a compressed bitmap index can outperform its uncompressed counterpart.

In this paper we introduce the word-aligned hybrid run-length code (WAH) and show that WAH is able to outperform other compression schemes by an order of magnitude using only 50% more space. It is even able to outperform uncompressed scheme in majority of the test cases. Using this compression scheme, we evaluated a three different encoding schemes, the equality encoding, the range encoding [15] and the interval encoding [4]. Tests on a set of real data from the STAR experiment show the the compressed indices not only takes significantly less space but also remains competitive with the uncompressed versions. When compressed, these encoding schemes exhibit different space and time trade-offs than in the uncompressed cases. In particular, none of them is the best in both space and time. To find better compromises in space and time, we also introduce a set of two-level bitmap index schemes.

The remainder of this paper is organized as follows. In Section 2, we briefly explain the terms used in this paper and review some of the commonly used bitmap indexing schemes. In Section 3, we describe the WAH compression scheme and demonstrate its effectiveness. Section 4 contains performance measures of the three encoding indices, with and without compression. We introduce the two-level schemes in Section 5 and present performance data for show their space and time trade-offs. A short summary is provided in Section 6.

2 Review of bitmap indices

In this section, we briefly review some common schemes of generating bitmap indices. The main purpose of this section is to introduce the terminology and the three optimal bitmap indexing schemes to be used later. A bitmap index is built for an individual attribute of a dataset. The process of generating a bitmap index from the values of the attribute typically involves three steps. The first step, which we call the binning step, assigns the tuples into bins to produces a list of integers that are indices to the bins. The second step, which we call the encoding step, encodes the results of the first into bit sequences or bitmaps. The final step, which we call the compression step, decides how to store these bit sequences.

For the first step, the classical strategy is to map each distinct value into one bin. This strategy is simple and effective if the attributes of the dataset have low cardinalities such as is typical in many attributes of data warehousing systems. However, for attributes with high cardinalities such as those from some scientific databases, this strategy generates too many bit sequences and requires too much space. One simple strategy to avoid this is to use only a small number of bins [12, 13, 14]. A number of binning schemes have been published in the literature [7, 14, 16, 18]. In this paper, we only use the most straightforward binning scheme, see Figure 1.

Now that each bin contains many values, the bitmap index is not able to give precise qualifying

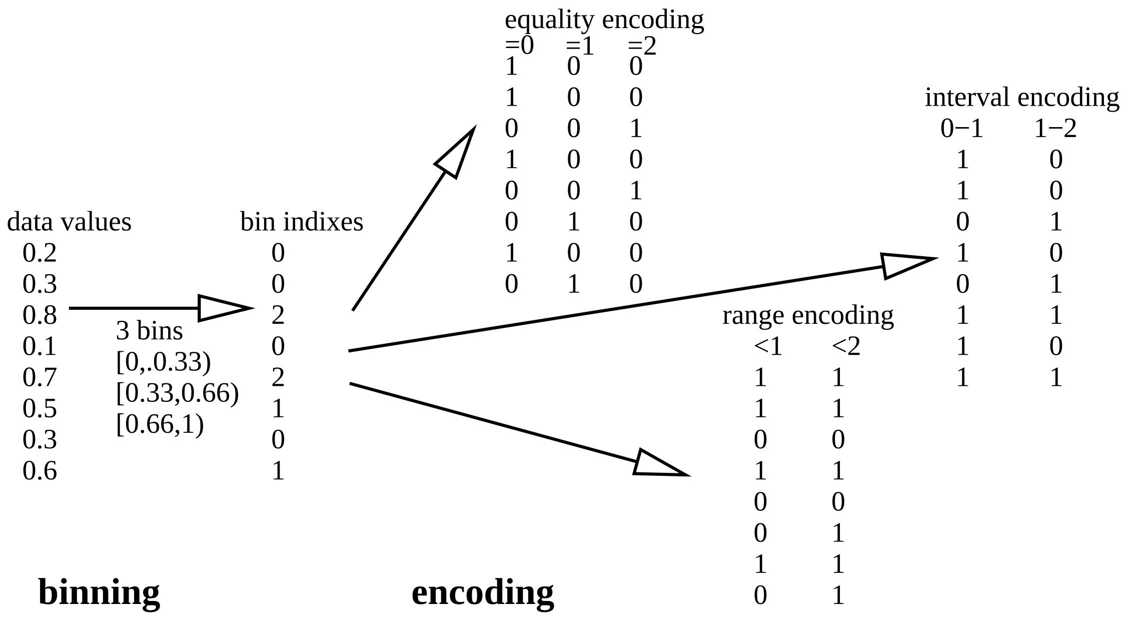


Figure 1: An illustration of the binning step and encoding step.

hits to some queries. The strategy we have taken to resolve this is to generate a superset and a subset of the qualifying hits. Those tuples that are in the superset but not in the subset needs to be resolved by another scheme. We perform comparisons directly on the data values to decide whether they satisfy the the query criteria. In the following discussions, we call the process of computing the superset and the subset as the *index operation* and the comparison step as the *final filtering operation.*

For the second step, i.e., the encoding step, one of the most comprehensive discussion is given by Wong et al [15]. A number of authors have recently proposed new ones that are provably optimal [3, 4, 18]. For the purpose of describing this step, we view the output from the binning step as a list of integers. The classical bitmap index maps each of these integers into one bit sequence. This is called the equality encoding in recent literature. Let A be the name of an attribute, an *equality query* is a query of the form $A = 2$ and a *membership query* is a query for the form $A \in \{6, 9, 11, 12\}$. The equality encoding is efficient for these queries because each bit sequence represents a relation of the form $A = i$ where i is an integer. The two other optimal schemes we use later are the range encoding [15] and the interval encoding [4]. Each bit sequence of the range encoding represents a one-sided range relation of the form $A < i$ and each bit sequence of the interval encoding represents a two-sided range relation of the form $i \leq A < i + \lceil b/2 \rceil$ where b is the number of bins used in the binning step and the function $\lceil x \rceil$ computes the ceiling of x. It is easy to see that the range encoding is optimal for one-side range queries and the interval encoding is optimal for two-sided range queries [4].

Figure 1 shows three different encoding schemes mentioned here. Given b bins, the equality encoding generates b bit sequences, the range encoding generates $b - 1$ bit sequences, and the interval encoding generates $b + 1 - \lceil b/2 \rceil$ bit sequences.

These three schemes are based on simple relations among the integers resulting from the binning step. Another general strategy for the encoding step is to decompose the integers as described in [3, 15]. Typically, these schemes generate less bit sequences, but the operations may involve most of them.

After generating the bit sequences, some data structures are needed to represent them in computer programs. We refer to these data structures as the bit vectors. The simplest bit vector stores the bit sequences literally without compression. We call this the *literal bit vector*. The main advantage of this approach is that the bitwise logical operations on the bit sequences are fast. This is an important property because the most common operations on the bit vectors are the bitwise logical operations. The bitmap indices using literal bit vectors are called the uncompressed bitmap indices. Those employing compressed bit vectors are called compressed bitmap indices. Compressed schemes are attractive because they can reduce the space requirement. Since the bit sequences generated from the bitmap indexing schemes are usually sparse, they are relatively easy to compress. However, because most of the generic compression algorithms don't support fast bitwise logical operations, the compressed bitmap indices are usually slower in processing queries compared to their uncompressed counterparts. To increase the efficiency of query processing, a number of specialized compression algorithms have been developed. The byte-aligned bitmap code (BBC) is an example of such a scheme [1, 2]. In the following section, we describe another specialized compression scheme called the word-aligned hybrid run-length code (WAH) [17]. It is an efficient scheme that significantly outperforms BBC. For this reason, it is our preferred method for compressing bitmap indices.

We demonstrate the efficiencies of the compressed bitmap indices through a number of experimental tests. The test results reported here are on the STAR data. We have also conducted some tests on a large set of synthetic data generated with various distributions. The tests on such synthetic data show similar relative performances among the various indexing schemes. For this reason, we present the results on the tests using the STAR data.

3 Word-aligned hybrid run-length code (WAH)

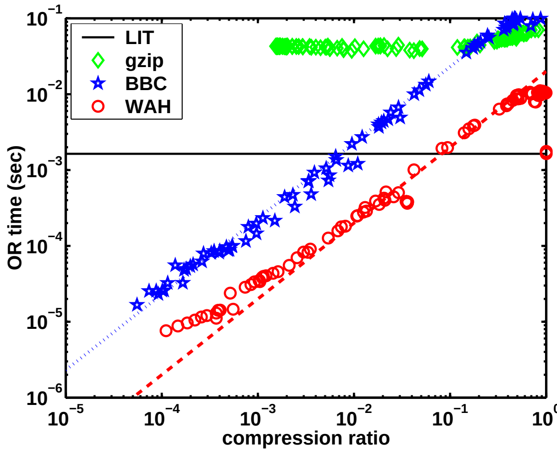
In this section, we briefly review the main characteristics of the compression algorithm used; namely the word-aligned hybrid run-length code or WAH for short. Because of space limitations we only give a short description of this coding scheme followed by some performance data. We don't describe how the logical operations are performed without decompression. Interested readers can find details in a technical report [17].

As the name suggests, this scheme is a variation on the run-length code. The essence of the run-length code is to represent a list of consecutive identical bits by its length and its bit value. As is common in the literature, we refer to a sequence of identical bits as a *fill*. The bit value of a fill is called the *fill bit*. The number of bits in a fill is called the *fill length*. To ensure efficient operations, WAH encodes the fill length and the fill bit in one whole word. Compared to the uncompressed scheme, this only reduces the space requirement if the fill is longer than a word. For fills that are shorter, WAH stores them literally. Altogether, there are two types of code words in WAH, those that contain literal bit values (called *literal words*) and those that contain fills (called *fill words*). In our current 32-bit implementation, we use the Leftmost Bit (LMB) of a word to distinguish between a literal word and a fill word, where 0 indicates a literal word and 1 indicates a fill word. The lower 31 bits of a literal word contains literal bit values. The second leftmost bit of a fill word is the fill bit and the 30 lower bits store the fill length. To achieve fast operation, it is crucial that we impose the word-alignment requirement on this scheme. The word-alignment requirement in WAH requires all fill lengths to be integer multiples of 31 bits (i.e., literal word size). Given this restriction, we represent fill lengths in multiples of literal word size. For example, if a fill contains 62 bits, the fill length will be recorded as two (2), see Figure 2.

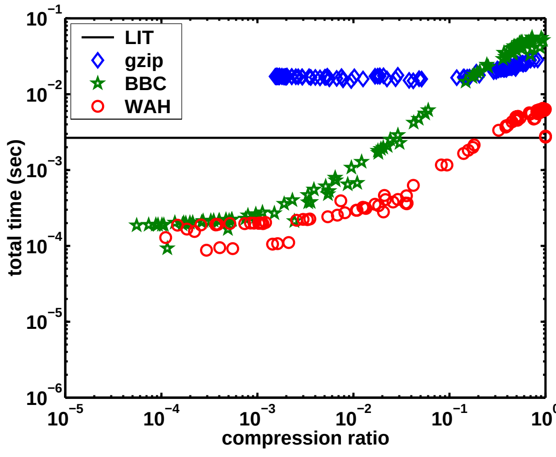
To see how well this compression scheme works, we have conducted some tests on a subset of

124 bits	1,20*0,3*1,79*0,21*1		
31-bit groups	1,20*0,3*1,7*0	62*0	10*0,21*1
groups in hex	40000380	00000000 00000000	001FFFFF
WAH (hex)	40000380	80000002	001FFFFF

Figure 2: A WAH bit vector. Each WAH code word (last row) represents a multiple of 31 bits from the bit sequence (first row).



(a) logical OR time



(b) total time

Figure 3: Time (seconds) to perform bitwise OR on bit vectors of an bitmap index.

the STAR data using the initial data of about 1 million events. We selected the 12 most commonly queried attributes from the total of over 500 and built an equality encoded bitmap index for each of the 12 attributes. The resulting bit sequences are stored in four different bit vector schemes, the literal scheme (LIT for short), the gzip scheme [8], the BBC scheme and the WAH scheme.

Figure 3 shows the timing results of the logical OR operations. The logical OR is selected as the representative of three different logical operations tested, because all three show similar relative performances among the various schemes. The horizontal axes of the plots are compression ratios. The *compression ratio* is defined as the ratio of the compressed bit vector size to its uncompressed counterpart. Each symbol displayed in Figure 3 represents the time to perform one logical operation on two bit vectors. The compression ratios shown in Figure 3 are the average compression ratios of the two operands of a logical operation.

The timing results reported here are from using a Sun Enterprise 450 machine[2] that is based 400 MHz UltraSPARC II CPUs. The test data were stored on a file system consisting of five disks connected to the internal UltraSCSI controller and managed by a VERITAS Volume Manager[3]. The VERITAS software distributes files onto the five disks to maximize the IO performance.

Figure 3(a) shows the logical operation time and Figure 3(b) shows the total time including the time to read the two bit vectors from files. If these bit vectors are used in an actual database system, it is likely that once they are read into computer memory, they will be kept in memory

[2]Information about the E450 is available at `http://www.sun.com/servers/workgroup/450`.

[3]Information about VERITAS Volume Manager is available at `http://www.veritas.com/us/products`.

LIT	gzip	BBC	WAH
1	0.16	0.20	0.29

Figure 4: Average compression ratios of all bit vectors from a bitmap index.

for as long as the space is not needed for other tasks. On the average, we expect the cost of a logical operation to be larger than what is shown in Figure 3(a) but no more than what is shown in Figure 3(b).

Because the bit sequences are relatively small, in some cases the IO overhead, which is about 2×10^{-4} seconds, dominates the total time. In these special cases, the total times are about the same for BBC and WAH. As the STAR experiment continues, the data size will grow significantly, and the IO overhead will become less significant in the future. Even with small data size, in the majority of the test cases, the word-aligned scheme is still significantly faster. When the compression ratio is one, the logical operations on WAH bit vectors are about 80 times faster than the same operations on BBC bit vectors. Even when the time to read two bit vectors is included, the WAH scheme is still about 20 times faster than BBC. If we sum up all the total time values from all test cases (including different logical operations), the sum for BBC is about 12 times that of WAH. In other words, on the average WAH is about 12 times as fast as BBC. If the IO time is not included, the differences are even larger. Compared to the literal scheme, the BBC scheme is faster in less than half of the test cases, WAH is faster in about 60% of the test cases.

Figure 4 shows the relative sizes, i.e., compression ratios, of the four bit vector schemes shown in Figure 3. On the average, WAH-compressed bit vectors use less than a third of the space required by the uncompressed scheme (LIT). Compared to BBC, WAH uses only about 50% more space. We believe it is worthwhile to trade this 50% more space for 12 fold increase in operation speed. The gzip scheme is based on an asymptotically optimal compress scheme [8]. Even compared again this optimal scheme, WAH scheme uses no more than twice as much space. For this extra space, WAH is able to perform logical operations several orders of magnitudes faster than gzip. Overall, we believe WAH is the most appropriate scheme for compressing bitmap indices.

4 Performance of compressed bitmap indices

In this section, we report the timing results of the three bitmap indexing schemes when compressed using the WAH scheme. The timing results are compared with their corresponding uncompressed versions and the sequential scan. The *sequential scan* refers to the process of sequentially reading each data value and performing comparisons to determine which tuples qualify a query. As in the previous section, the test data consists of the 12 most commonly queried attributes from the STAR data using just over 2 million events.

The test queries are randomly generated two-sided range queries. These are the most common form of queries on the given dataset. One common measure for these range queries is the *query box size*, which is defined to be the ratio of the range covered by a query to the domain size. We expect that the majority of the queries to have relatively small query boxes. In this test all queries involves only one attribute, we constructed three quarters of our random queries to have query box sizes less than a half and the rest have larger query boxes. Let b be the number of bins used in the binning step. On queries with query boxes less than $1/b$, the equality encoded index may be more efficient than the other two. On queries with larger query boxes, the equality encoded index is likely to be worse than the other two.

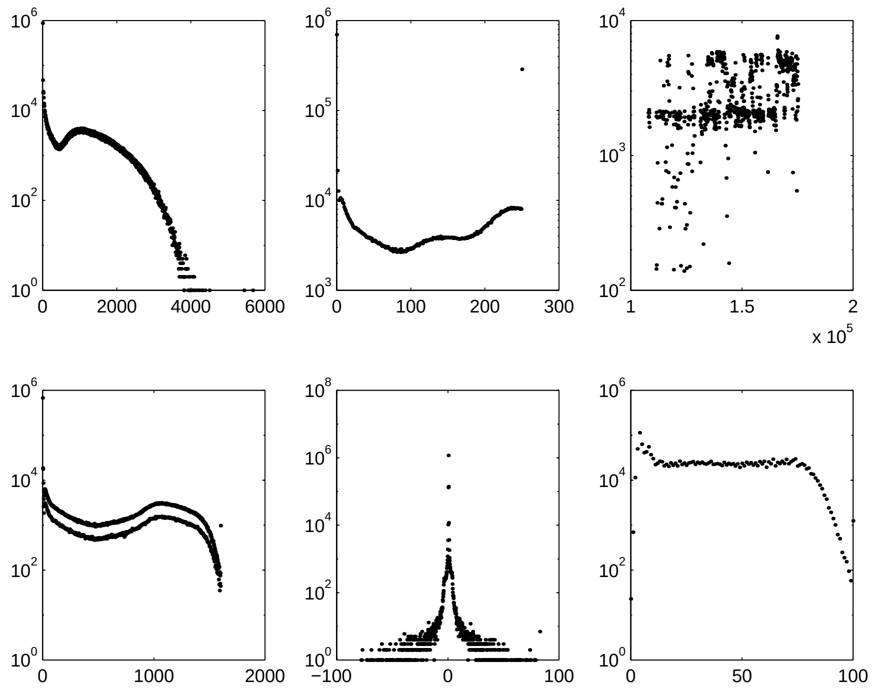
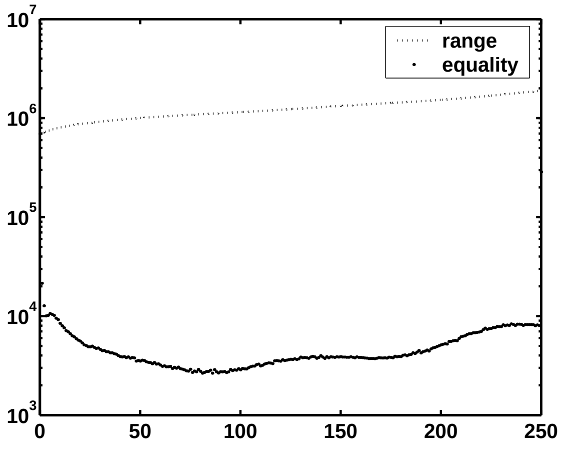
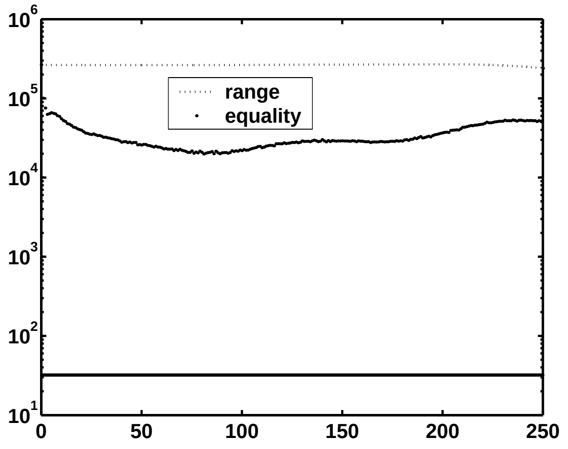


Figure 5: Data distribution of selected STAR data.



(a) events distribution

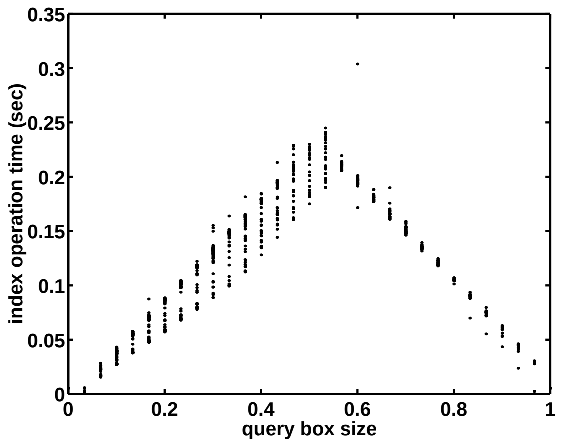


(b) bit vector size in bytes

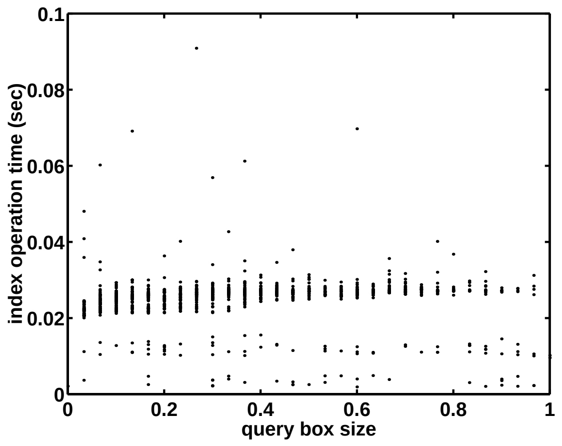
Figure 6: Size information about bit vectors of one bitmap index.

Figure 5 shows the data distribution of some attributes from the test data set. The domains of these attributes are divided into 1000 equal-sized bins and the vertical axes shows the number of events that fall into the bin. It appears that all attributes are distributed highly unevenly in their corresponding domains. Synthetic datasets are unlikely to have similar data distributions. In the rest of this section, we first present some performance information on one specific attribute, and then present some average measurements.

The first set of performance data in Figure 6 are about the sizes of the bitmap indices on one



(a) 30 bins, equality encoding



(b) 30 bins, range encoding

Figure 7: Time to process a single query on a single attribute.

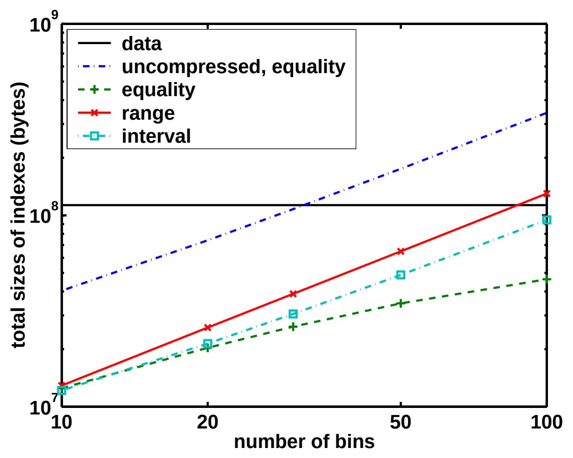
particular attribute. To avoid cluttering the graphs, we have shown only the information about the equality encoded index and the range encoded index. In general, each bit vector for the interval encoded index is slightly larger than a bit vector from the range encoded index, but there are only about half as many bit vectors in the interval encoded index as there are in the range encoded index.

The domain of this attribute ranges from 0 to 250. This domain is divided into 1000 bins. Figure 6(a) plots the number of ones in each bit vector for each of the 1000 bins. Figure 6(b) shows the sizes of the bit vectors. This figure appears to have three lines. The one on the top is for the range encoded index and the other two are for the equality encoded index. These two lines are actually formed from 1000 dots. The bottom one is formed from those bit vectors representing empty bins. They take 32 bytes each. In fact, all bit vectors for the equality encoded index are much smaller than those bit vectors from the range encoded index. Each bit vector for the range encoded index is nearly the same size as the uncompressed bit vector storing the same number of bits. In this case, we say the corresponding bit sequences are incompressible.

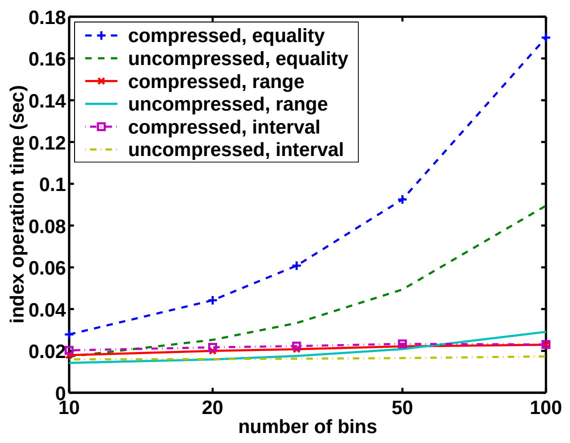
We have chosen this particular attribute to demonstrate the size differences between the two encoding schemes. Though the differences for other attributes may not be as dramatic as this one, it is generally true that the size of the range encoded index is large than the equality encoded index. If there are only a few bins and all bit vectors from both schemes are incompressible, then the range encoded one may take less space because it actually uses one less bit vector than the equality encoded one. Without compression, because the range encoded index uses one less bit vector, it always takes less space [3, 4, 18]. Clearly, with compression, the relative differences are more complicated.

The query processing time can be divided into two parts, the *index operation time* and the *final filtering time*. When the binning step uses the same number of bins, the three indices return the same results from their index operations and the final filtering time should be same. For this reason, the first few sets of timing results we present contain only the index operation time.

Figure 7 shows the index operation time of individual queries on one attribute. The index used to generate Figure 7(a) is based on the equality encoding and the index used to generate Figure 7(b) is based on the range encoding. Aside from the few outliers, the timing values in Figure 7(a) generally fall under a pyramid and the timing values in Figure 7(b) show no obvious



(a) sizes of the indices



(b) average index operation time

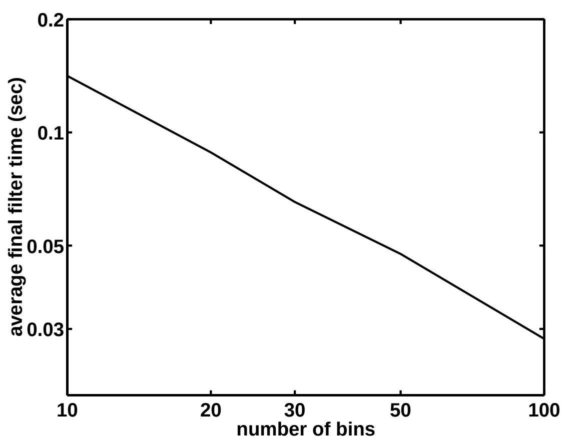
Figure 8: Size of the indices and time (seconds) to operate on them.

relation with the query box size. We didn't show the results for the interval encoded index because it is similar to the results for the range encoded index. As predicted in [3, 4, 18], the index operation times for both the range encoded index and the interval encoded index don't depend on the query box size. For the equality encoded index, as query box grows, more bit vectors have to be accessed during index operation. However, when the query box size is larger than a half, it is more efficient to process the complement of the query. As a result the time decreases as query box size approaches one.

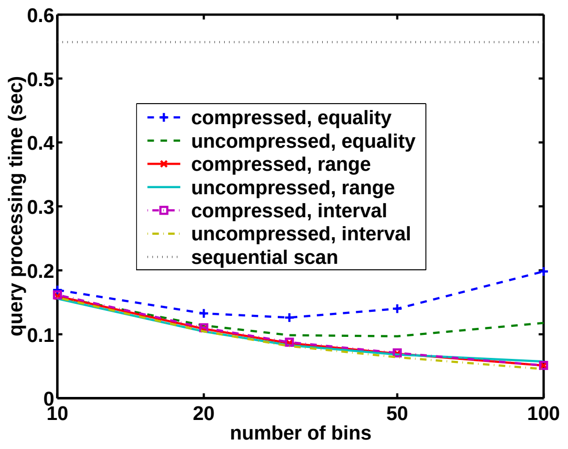
This figure only shows query processing time on one attribute. Different set of queries are generated for the other 11 attributes as well. To simplify the comparisons among the different indexing schemes, we use average time of different queries and different attributes in the following discussions.

Figure 8 shows some information about the overall space and time efficiencies of the three encoding schemes. Figure 8(a) shows the total sizes of all 12 bitmap indices built for the 12 selected attributes. The compression scheme reduces the space requirement significantly in all test cases. The compressed indices are less than a half of the size of the uncompressed ones. The indices based on equality encoding are smaller than the other two. Without compression, it is the one that use the largest amount of space and the index size for the interval encoding is about half of the size of the other two. However, with compression, the index for the equality encoding is the smallest and the one based on the interval encoding is about 20% smaller than the one based on the range encoding. In addition, as the number of bins increases, the growth in total size of the equality encoded index decreases and the difference between the equality encoded index and the other two increases. At 100 bins, the compressed equality encoded index is less than one seventh the size of the uncompressed one. The total sizes of the range encoded indices and the interval encoded indices all increase nearly linearly as number of bins.

Figure 8(b) shows the average index processing time for the three bitmap indices. The compressed versions and the uncompressed versions for all three encoding schemes are shown here. Overall, the time spent by the compressed versions and the uncompressed version are fairly close. This is particularly true for the indices based on the range encoding and the interval encoding. This defies the typical expectation that reducing size leads to more processing time.



(a) average final filtering time



(b) total query processing time

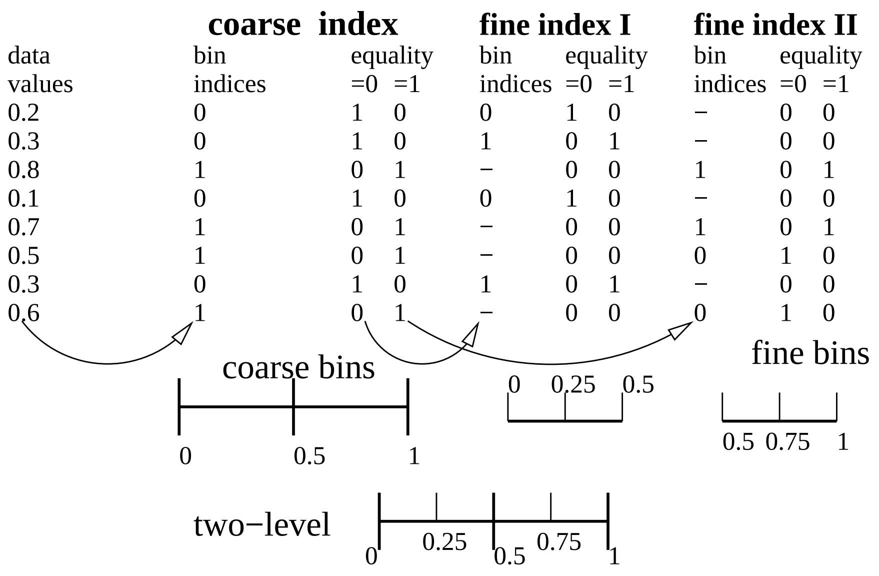
Figure 9: Average final filtering time and total query processing time when different number of bins are used.

Figure 9(b) shows the total query processing time including both the index operation time and the final filtering time. The timing results shown here are simply the sum of the index operation times in Figure 8(b) and the final filtering times shown in Figure 9(a). On the average, the final filtering time is nearly inversely proportional to the number of bins used, see Figure 9(a). This is because as more bins are used, the bitmap index is able to provide more accurate answers. In other words, the difference between the superset and the subset produced by the index operation is smaller. The final filtering step has less entries to compare and it takes less time.

For reference, Figure 9(b) also contains a line indicating the average time needed to perform the sequential scan to process the same queries. In other studies, simple scan the whole dataset was found to be among the fastest strategies for processing high-dimensional queries [9, 14]. In a typical relational database, all values of a tuple are stored together. In this case, comparing the value of one attribute requires one to read the entire tuple from disk. Our sequential scan is able to only access the needed values since we have vertically partitioned the database and stored values for each attribute separately.

The average time to scan the 2 million entries is 0.557 seconds. Given that most of the attribute values are 4 bytes each. This is equivalent to a reading speed of about 17 MB/s. The peak speed of the SCSI controller on the test system is about 80 MB/s. Considering the sequential scan also needs to perform comparisons and store the results in a compressed bit vector. This sequential scan is indeed quite efficient. Compared to this sequential scan, all versions of the bitmap indexing schemes can complete the same queries a number of times faster. It indicates that we have a high quality implementation of the indexing schemes.

Queries used in this test only involves one attribute. If the user queries typically involve more than one attribute, the final filtering step should take less time [13]. In this case, using a bitmap index can be even faster compared to such schemes as the sequential scan.



	coarse index			**fine index I**			**fine index II**		
data	bin	equality		bin	equality		bin	equality	
values	indices	=0	=1	indices	=0	=1	indices	=0	=1
0.2	0	1	0	0	1	0	−	0	0
0.3	0	1	0	1	0	1	−	0	0
0.8	1	0	1	−	0	0	1	0	1
0.1	0	1	0	0	1	0	−	0	0
0.7	1	0	1	−	0	0	1	0	1
0.5	1	0	1	−	0	0	0	1	0
0.3	0	1	0	1	0	1	−	0	0
0.6	1	0	1	−	0	0	0	1	0

Figure 10: An illustration of a two-level EE index.

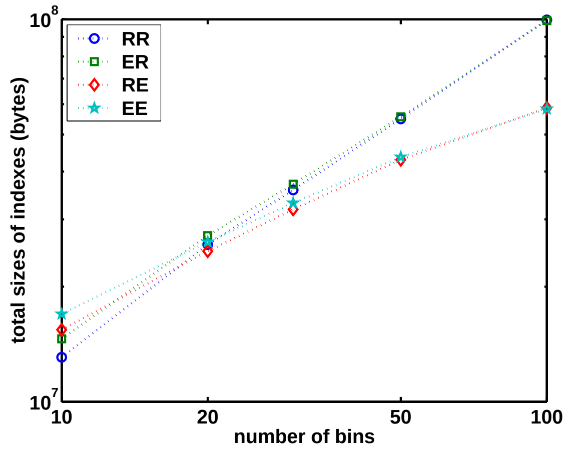
5 Two-level schemes

In the previous section, we see that the equality encoded index is more space efficient than the range encoded index, but the latter is more time efficient than the former. To find a compromise between the two, we present a set of two-level schemes in this section.

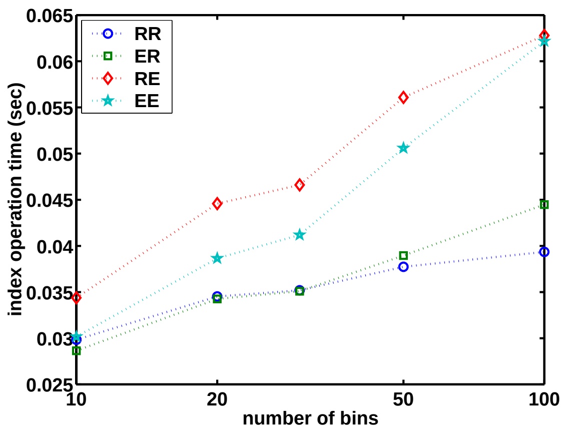
The range encoding doesn't need many operations because each bit vector contains information about a large portion of the attribute's domain. We view this as global information. This global information also causes the bit vectors to be larger. In contrast, each bit vector from equality encoded index contains only information about one bin and for this reason the bit vectors are smaller. The two-level schemes allow one to control how much of the global information to retain. This allows one to possibly find a median between the range encoded scheme and the equality encoded scheme.

Overall, the two-level schemes can be viewed as generating two sets of overlapping bins for the binning step from the index construction process. The example shown in Figure 10 demonstrates how we generate a two-level index based on equality encoding. In this example, we first divide the attribute's domain into two coarse bins and build an index as before. Each of the two coarse bins is then divided into two fine bins. An index is then built for each coarse bin separately. We refer to the index built on top of the coarse bins as the *coarse index* and the ones built on top of fine bins as the *fine indices*. The fine indices for each coarse bin can be encoded differently, but for simplicity we encode all of them the same way. This limits us to four different two-level indices depending on how we encode the two sets of indices. Let E denote the equality encoding and R denote the range encoding. Then the four schemes can be written as RR, RE, ER, and EE. In this short-hand notation, the first letter indicates the encoding type of the coarse index and the second letter indicates the encoding type of the fine index.

In the one-level cases, the total number of bins is the only parameter to consider when generating bitmap index on equal-sized bins. In the two-level cases, the total number of bins has a similar role as the total number of fine bins as they both control the accuracies of the answers generated from the index operations. There is an additional parameter, which is the number of coarse bins. If there



(a) sizes of indices



(b) index operation time

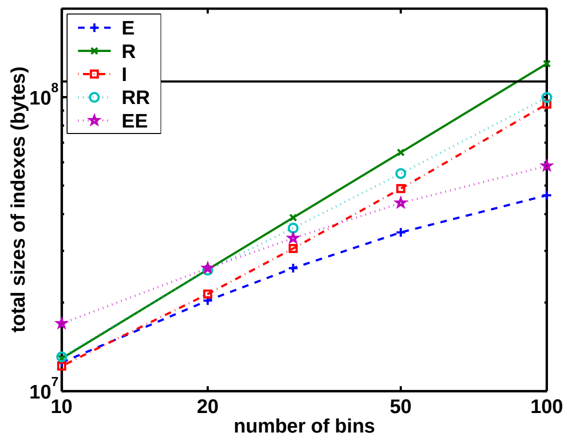
Figure 11: Comparison among the four two-level schemes.

are more coarse bins, it is more likely that during the index operation only the coarse index will be needed. In the two-level RR index, this reduces the average index operation time. The bit vectors of the coarse indices are typically larger than the bit vectors of the fine indices. This suggests that we should have only a few coarse bins. For the two-level EE index, if we assume every logical operation takes the same amount of time, the worst-case index operation time is proportional to $n_c + 2n_f/n_c$, where n_c is the number of coarse bins and n_f is the total number of fine bins. Given n_f, the above expression is minimized when $n_c = \sqrt{2n_f}$. We use this formula to choose the number of coarse bins in the following tests.

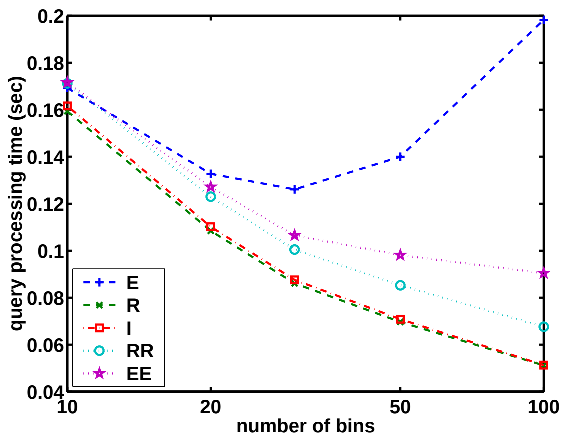
Figure 11 compares the two-level schemes. Figure 11(a) shows the sizes of the different indices. The horizontal axis in this case is the number of fine bins. When there are only 10 fine bins, the EE scheme uses the largest amount of space among the four and RR uses the least. With 20 or 30 fine bins, the four schemes use about the same amount of space. As more fine bins are used, the encoding scheme of the fine bins appears to determine the space requirement. The two that use the equality encoding for the fine bins take about the same amount of space and the two that use the range encoding for the fine bins take about the same amount of space. With 100 fine bin, schemes EE and RE take about 40% less space compared to RR and ER.

Figure 11(b) shows the time to perform the index operation during query processing. In general, the two with range encoding for the fine indices take less time. The differences increase as the number of bins increase. This is similar to what was observed for the one-level schemes.

We take RR and EE as representatives of the two-level schemes and show them together with the one-level schemes in Figure 12. In terms of space, see Figure 12(a), the scheme RR always uses slightly less space than the one-level range encoded bitmap index. The scheme EE always uses more space than the one-level equality encoded bitmap index. It uses about half the space of the one-level range encoded scheme when there are 100 fine bins. The differences in time, see Figure 12(b), are relatively small when there are only few bins. This is because the total query processing time is dominated by time in the final filtering step. If there are more bins, the differences among the various schemes are larger. With 100 fine bins, the two-level EE scheme uses about 10% more space than the one-level equality encoded scheme and it takes less than half of the time to process the same queries.



(a) sizes of indices



(b) query processing time

Figure 12: Comparing two-level schemes against the one-level ones.

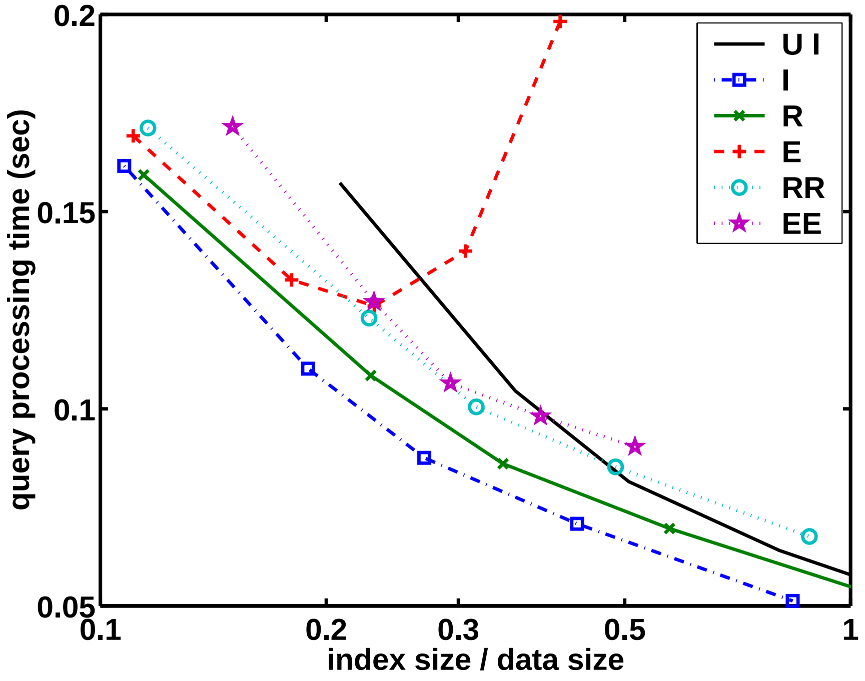


Figure 13: Average query processing time versus index sizes (relative to data size) for various schemes tested.

Overall, the equality encoding is still the most space efficient among the encoding schemes tested, and the range encoding and the interval encoding are the most time efficient. However, this comparison is based on fixing the number of bins. To a user of a database system, this parameter isn't as crucial as space usage or operation speed. For this reason, we directly plot the average query processing time and the index size in Figure 13.

Figure 13 contains the same data as in Figure 12. The difference is that we have removed the number of bins out of the picture. The horizontal axis is the relative sizes of the indices. This figure contains information from five compressed bitmap indices and an uncompressed one. The uncompressed one labeled "U I" is the uncompressed version of the interval encoded bitmap index. The compressed ones are the one-level equality encoded index (E), the one-level range encoded index (R), the one-level interval encoded index (I), the two-level schemes EE and RR. From this figure, we see that the the one-level interval encoded index (I) is the best among the schemes tested. Given the same space for indices, the one-level range encoded scheme uses less time to process an average query even though it has to use less bins to accommodate the space restriction.

6 Summary

Without compression, the relative strengths of the various bitmap indexing schemes are well understood [3, 4, 18]. However, the same is not true when compression is used. In this paper, we studied how to use one specific scheme, namely the word-aligned hybrid run-length code (WAH), to compress various bitmap indexing schemes. Our main conclusion is that the compressed indices are much smaller than the uncompressed ones and the average query processing times are about the same. This suggests WAH can be used to reduce the size of bitmap indices without decreasing the performance of the database systems. In many cases, the database administrator may want to limit the space used by the index when designing a database. Tests conducted in this paper can serve as references for this type of decision. Based on our experience, the following are reasonable initial choices.

allowed index size / data size	1/8	1/4	1/2
bins for range encoded index	10	20	40
bins for interval encoded index	10	25	60
bins for equality encoded index	10	30	100

Under the same space restriction, which encoding scheme produces the best bitmap index is likely to depend on the type of query it handles. For two-sided range queries, the most important fact is the query box size on individual attributes since the bitmap indices process each attribute separately. In this paper, we tested on a set of queries that have medium size query boxes. In this case, the compressed bitmap index based on interval encoding is the best. If the query boxes on each attribute is very small, or queries are equality queries or one-sided range queries, other encoding schemes might be better. We plan conduct further tests to determine the most appropriate choices for these queries in the future.

References

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

[2] G. Antoshenkov and M. Ziauddin. Query processing and optimization in orcale RDB. *The VLDB Journal*, 5:229–237, 1996.

[3] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *Proceedings of the 1998 ACM SIGMOD: International Conference on Management of Data*. ACM press, 1998.

[4] C. Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM*

SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA. ACM Press, 1999.

[5] T. Johnson. Performance measurements of compressed bitmap indices. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*. Morgan Kaufmann, 1999. A longer version appeared as AT&T report number AMERICA112.

[6] M. Jürgens and H.-J. Lenz. Tree based indexes vs. bitmap indexes - a performance study. In S. Gatziu, M. A. Jeusfeld, M. Staudt, and Y. Vassiliou, editors, *Proceedings of the Intl. Workshop on Design and Management of Data Warehouses, DMDW'99, Heidelberg, Germany, June 14-15, 1999*, 1999.

[7] Nick Koudas. Space efficient bitmap indexing. In *Proceedings of the ninth international conference on Information knowledge management CIKM 2000 November 6 - 11, 2000, McLean, VA USA*, pages 194–201. ACM, 2000.

[8] Jean loup Gailly and Mark Adler. *zlib 1.1.3 manual*, July 1998. Source code available at `http://www.info-zip.org/pub/infozip/zlib`.

[9] Volker Markl, Martin Zirkel, and Rudolf Bayer. Processing operations with restrictions in RDBMS without external sorting: The tetris algorithm. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 562–571. IEEE Computer Society, 1999.

[10] P. O'Neil. Model 204 architecture and performance. In *2nd International Workshop in High Performance Transaction Systems, Asilomar, CA*, volume 359 of *Springer-Verlag Lecture Notes in Computer Science*, September 1987.

[11] Julian Seward. *bzip2 and libbzip2*, March 2000. Source code available at `http://-sourceware.cygnus.com/bzip2`.

[12] A. Shoshani, L. M. Bernardo, H. Nordberg, D. Rotem, and A. Sim. Multidimensional indexing and query coordination for tertiary storage management. In *11th International Conference on Scientific and Statistical Database Management, Proceedings, Cleveland, Ohio, USA, 28-30 July, 1999*. IEEE Computer Society, 1999.

[13] K. Stockinger, D. Duellmann, W. Hoschek, and E. Schikuta. Improving the performance of high-energy physics analysis through bitmap indices. In *11th International Conference on Database and Expert Systems Applications DEXA 2000, London, Greenwich, UK*, September 2000.

[14] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In J. Widom, A. Gupta, and O. Shmueli, editors, *VLDB'98, Proceedings of 24th International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 194–205. Morgan Kaufmann, 1998.

[15] H. K. T. Wong, H.-F. Liu, F. Olken, D. Rotem, and L. Wong. Bit transposed files. In *Proceedings of VLDB 85, Stockholm*, pages 448–457, 1985.

[16] K.-L. Wu and P. Yu. Range-based bitmap indexing for high cardinality attributes with skew. Technical Report RC 20449, IBM Watson Research Division, Yorktown Heights, New York, May 1996.

[17] Kesheng Wu, Ekow J. Otoo, Arie Shoshani, and Henrik Nordberg. *Notes on Design and Implementation of Compressed Bit Vectors*. Berkeley, CA, 2001. Tech report.

[18] M.-C. Wu and A. P. Buchmann. Encoded bitmap indexing for data warehouses. In *Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 220–230. IEEE Computer Society, 1998.