On the Performance of Bitmap Indices for High Cardinality Attributes*

Kesheng Wu and Ekow Otoo and Arie Shoshani

Lawrence Berkeley National Laboratory, Berkeley, CA 94720
Email: {KWu, EJOtoo, AShoshani}@lbl.gov

Abstract

It is well established that bitmap indices are efficient for read-only attributes with low attribute cardinalities. For an attribute with a high cardinality, the size of the bitmap index can be very large. To overcome this size problem, specialized compression schemes are used. Even though there are empirical evidences that some of these compression schemes work well, there has not been any systematic analysis of their effectiveness. In this paper, we systematically analyze the two most efficient bitmap compression techniques, the Byte-aligned Bitmap Code (BBC) and the Word-Aligned Hybrid (WAH) code. Our analyses show that both compression schemes can be optimal. We propose a novel strategy to select the appropriate algorithms so that this optimality is achieved in practice. In addition, our analyses and tests show that the compressed indices are relatively small compared with commonly used indices such as B-trees. Given these facts, we conclude that bitmap index is efficient on attributes of low cardinalities as well as on those of high cardinalities.

*The authors gratefully acknowledge the help received from Dr. Kurt Stockinger for review a draft of this paper. 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. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

1 Introduction

Bitmap indexing scheme of one kind or another have appeared in all major commercial database systems. This is a strong indication that the bitmap index technology is indeed efficient and practical [9]. The basic bitmap index scheme builds one bitmap for each distinct value of the attribute indexed, and each bitmap has as many bits as the number of tuples. The size of this index can be very large for a high cardinality attribute where there are thousands or even millions of distinct values. Many strategies have been devised to reduce the index sizes, such as, more compact encoding strategies [5, 6, 10, 13], binning [11, 12, 14, 19], and compression [2, 3, 16, 17]. In this paper, we study how compression schemes improve the bitmap indices. A number of empirical studies have shown that some compression schemes can reduce the index sizes as well as the query response time [8, 16, 17]. In this paper, we present analyses and performance tests on two of the most efficient compression schemes, and show that the compressed bitmap indices are efficient for attributes of any cardinality.

Let N denote the number of tuples in a relation, the basic bitmap index for any attribute of the relation has N bits in each bitmap; one corresponding to each tuple. If an attribute has c distinct values, where c is a short hand for its cardinality, then there are c bitmaps with N bits each. For example, in the bitmap index for an integer attribute in the range of $0 \ldots (c-1)$, the ith bit of the jth bitmap is 1 if the attribute's value in the ith tuple equals to j. Without compression, this bitmap index requires $cN/8$ bytes to store. If the attribute values are 4-byte integers, a typical B-tree index from a commercial database system is observed to use $10N$ to $15N$ bytes which is about 3 to 4 times the original data. If the cardinality is high, the bitmap index can be much larger than the B-tree index and the original data. One effective way to overcome this size problem is to compress the bitmaps.

There are many general-purpose text compression schemes, such as LZ77 [4, 8, 20], that can compress bitmaps. However these schemes are not efficient for answering queries[8, 16]. To answer a query, the most common operations on the bitmaps are bitwise logical operations, such as AND, OR and NOT. For example, for an integer attribute "I" with values ranging from 0 to 99, to answer a query for the form "$I < 50$", 50 of the bitmaps corresponding to values from 0 to 49 will be ORed together. Bitwise logical operations on bitmaps compressed with a typical text compression algorithm are generally much slower than the same operations on the uncompressed bitmaps [8, 16]. To improve the speed of operations, a number of specialized bitmap compression schemes have been developed. Two of the most efficient schemes are the Byte-aligned Bitmap Code (BBC) [2] and the Word-Aligned Hybrid code (WAH) [17]. Both are based on the run-length encoding. They represent a long sequence of 0s or 1s using a counter, and represent a mixture of 0s and 1s literally.

There are a number of empirical studies that indicate that these two compression schemes are efficient [1, 8, 16, 17]. In this paper, we systematically analyze the performance of answering range queries using compressed bitmap indices. The first result of the analyses is that the sizes of the compressed bitmap indices are relatively small compared with the typical B-tree indices. This is true even for attributes with very high cardinalities. The second result is that the time and space required to perform a bitwise logical operation on two compressed bitmaps are at worst proportional to the total size of the two. Furthermore we show that bitwise OR operations on a large number of bitmaps can be performed in time linear in the total size by using an in-place algorithm. This is optimal because it has the same complexity as reading the same bitmaps once. For a searching algorithm, one stringent definition of optimality is that its time complexity is linear in the number of hits. Using this definition, the compressed bitmap index is optimal for high cardinality attributes because the total size of the bitmaps involved in answering a query is proportional to the number of hits.

Depending on the number of bitmaps involving in answering a query, different algorithms achieve the optimal performance. Guided by the above analyses, we developed a simple yet effective strategy to select the appropriate algorithms to ensure the best performance in practice. Tests show that the bitmap indices, compressed with both WAH and BBC, scale linearly with the total size of bitmaps involved as predicted by the analyses. In the tests reported in this paper, a WAH compressed index typically uses about half the time required by a BBC compressed index to answer the same query. When querying high-dimensional datasets, the projection index is often the best option [10]. In out tests, a WAH compressed index always outperforms the projection index, a BBC compressed index may take more time than the projection index.

The rest of this paper is organized as follows. Sections 2 and 3 contain our analyses of the worst case sizes with WAH and BBC compression. Section 4 has a discussion on the time complexity of bitwise logical operations on two bitmaps. We discuss the options to operate on a large number of bitmaps in Section 5, and show that the in-place OR algorithm is a linear algorithm. A number of measurements are shown in these sections to verify the analyses. However, the bulk of performance measurements are shown in Section 6, where we also discuss how to select the best options to perform logical operations on many bitmaps. Finally, a short summary is given in Section 7.

2 Sizes of WAH Compressed Bitmap Indices

The Word-Aligned Hybrid (WAH) code is much simpler than the Byte-aligned Bitmap Code (BBC) and much easier to analyze. For this reason, we start with WAH. WAH is a hybrid of the run-length encoding and the literal bitmap [16, 17]. It contains two types of code words, *literal words* for storing literal bits and *fill words* for storing fills. In general, a *fill* is a consecutive group of bits of the same value. A group of 0s is a *0-fill* and a group of 1s is a *1-fill*. Both WAH and BBC require their fills to be of specific lengths. This causes short groups of bits with mixed 0s and 1s to be left out. Both schemes store these left out bits literally. We say a bitmap is *uncompressible* if all of the bits have to be stored literally. A bitmap is *uncompressed* if all bits are stored literally.

Since there are two types of words in WAH, one bit is required to distinguish them. In a literal word, the remaining bits are used to store raw bit values. To improve operational efficiency, WAH requires each fill to contain an integer multiple of bits stored in a literal word. For example, on a machine that uses 32-bit words, a literal word can store 31 bits from the bitmap, therefore each fill must contain a multiple of 31 bits. The length of a fill is recorded as the multiple of literal word size. For example, a fill with 93 bits would be recorded as length 3. In a fill word, one bit is used to distinguish it from a literal word, one bit is needed to record the bit value of the fill, and the remaining bits are used to store the fill length. On a 32-bit machine, the maximum fill length is $2^{30} - 1$, which represents a fill with $31 \times (2^{30} - 1)$ bits.

In an implementation of the bitmap index, the index is typically segmented. In this case, a bitmap index can be viewed as having a number of smaller indices each for a subset of the tuples of the relation indexed. This is necessary to reduce the size of each bitmap, improve the flexibility of the index generation process, and reduce possible access conflicts during update. Under this arrangement, a bitmap might contain only a few thousand bits or a few million bits. The maximum fill length is usually much smaller than 2^{30}. With this observation, we can safely assume that *a fill of any length can be represented in one fill word.* This significantly simplifies the analysis of WAH compressed bitmaps.

Using WAH to compress a bitmap, we first divide the input bits into groups that fit in literal words. If there are two or more consecutive groups with only 0s (or 1s), these groups form one fill and can be represented in one fill word. All remaining groups are represented literally. Let us call a fill followed by a group of literal bits a *run* and call the literal bits in a run the *tail*, see Figure 1 for an example. A run takes at most two words to represent; one fill word for the leading fill and one literal word for the tail. The only run that might not have a tail must be the last run of a bitmap. Typically, the last few bits of a bitmap do not use up a literal word, however a whole word has to be used to store them. Even though the last run might not have a tail, we always need at least a literal word. All together, the number of words in a WAH compressed bitmap is at most twice the number of runs, which proves the following theorem.

Theorem 1 *Let r denote the number of runs in a bitmap, the WAH compressed version of it requires at most $2r$ words.*

A bit with value 1 is also known as a *set bit.* All runs of a bitmap, except the last one, must contain at least one set bit. If a bitmap has n set bits, then it can have at most $n+1$ runs. Using the above theorem, the WAH compressed bitmap would use at most $2n+2$ words.

The number of 1s in any particular bitmap depends on the characteristics of the attribute. However, the total number of 1s of the entire bitmap index must equal the number of tuples N. Because of this, the maximum total size of all bitmaps is $2N+2b$, where b is the number of bitmaps used. This proves the following theorem.

Theorem 2 *Let N be the number of tuples in a relation, and let b denote the number of bitmaps used in the basic bitmap index for an attribute, using WAH compression, the maximum number of words required by the compressed bitmap index is $2N + 2b$.*

In the extreme case where every value of the attribute is distinct, the number of bitmaps is the number of tuples, i.e., $b = N$. In this case, a total of $4N$ words are required for the bitmap index. This extreme value is close to the typical size of a B-tree index. Therefore, with WAH compression, even in the most extreme case the bitmap index size is no larger than the commonly used B-tree index. As long as the attribute cardinality is much smaller than N, the bitmap index size is about half of that of a B-tree.

When a bitmap index is segmented, more bitmaps are used than the unsegemented index. This will increase the total size of the index. However, this increase is straightforward to account for. To ease the analyses and comparisons, we concentrate on unsegmented indices from now on.

type	**fill**	**tail**	**encoding**
1	short	normal	header, tail
2	short	special	header
3	long	normal	header, counter, tail
4	long	special	header, counter

Figure 2: The four types of BBC runs. The special tail is one byte long and has only one bit that is different from the fill just before it. A normal tail may have up to 15 bytes of any value. A fill is considered short if it has no more than three or seven bytes, see footnote 1.

3 Sizes of BBC Compressed Bitmap Indices

In this section, we derive an upper bound for the sizes of BBC compressed bitmap indices. To do this, we first outline the BBC compression scheme, then compute the maximum number of runs required to index a high cardinality attribute and the average size of the runs.

3.1 Outline of the BBC Compression

The Byte-aligned Bitmap Code (BBC) was developed by Antoshenkov and is used in a commercial database product [2]. There are two main variants of this scheme. One is designed primarily to compress 0-fills and the other to compress both 0-fills and 1-fills. The former is known as the 1-sided BBC and the later the 2-sided BBC. For sparse bitmaps, the 1-sided variant compresses slightly better than the 2-sided variant.

The BBC compression scheme breaks a bitmap into bytes. A BBC *fill* is a consecutive group of bytes that contains the same bits. In BBC, a *run* contains a fill followed a number of literal bytes. BBC encodes a bitmap one run at a time. A header byte is always used for each run. The length of a fill is recorded in the number of bytes. For short fills[1], its length is recorded in the header byte. For a longer run, a multi-byte counter is used to record the fill length. Each byte of the counter reserves one bit to indicate whether there are more bytes in the counter, and the remaining seven bits are concatenated in the order of their appearance to form a binary integer. This integer plus an offset[2] δ, is the actual number of bytes in the fill. Each byte of the multi-byte counter is basically a digit of a base 128 integer. For a fill with f bytes ($f \geq \delta$), the multi-byte counter uses $1 + log_{128}(f - \delta)$ bytes[3]. In the example shown in Figure 1, using the 2-sided BBC, the first two runs have short fills and the rest have long fills. The counters for runs 3 and 4 are one-byte long with value 8 indicating the fills have 12 bytes and 96 bits. The

[1]For a 1-sided variant, a short fill can have up to seven bytes; for a 2-sided variant, a short fill can have up to three bytes.

[2]For a 1-sided variant, the offset δ is eight; for a 2-sided variant, the offset δ is four.

[3]When $f = \delta$, one byte is used.

sample bits	30*1, 1*1, 8*0, 9*1, 100*0, 100*1, 1060*0, ...				
WAH	run 1	run 2	run 3	run 4	run 5
	30*0,1*1;	8*0,9*1,14*0;	62*0; 24*0,7*1;	93*1;	1054*0, ...
	4 bytes	4 bytes	8 bytes	4 bytes	4+ bytes
BBC	run 1	run 2	run 3	run 4	run 5
	30*0,1*1,1*0;	7*0,9*1;	96*0,4*0,4*1;	96*1;	1060*0,...
	type 2	type 1	type 3	type 3	type 3/4
	1 byte	3 bytes	3 bytes	2 bytes	3+ bytes

Figure 1: A sample bitmap

.

fill in run 5 has 132 bytes, which requires a two-byte counter. The counter records the value of 128, which is 1 0 in base 128 integer.

BBC also identifies a special tail. This special tail is one byte long and has only one bit different from the majority. The majority of the bits are the same as the preceding fill. This special byte is not explicitly stored. Instead three bits of the header byte are used to store the position of the bit that is different. Clearly, this special byte is very common in sparse bitmaps. In its simplest form, a BBC compression scheme divides runs into four types as illustrated in Figure 2 [2, 8].

3.2 Number of Runs

In this paper, we only analyze sparse bitmaps. Let N denote the number of bit in a bitmap and n denote the number of set bits. A *sparse bitmap* satisfies the relation $n < N/100$. As with WAH compression, a run must have at least one set bit, except the last run which might not have any set bit. The worst case size of BBC can be computed by assuming that all BBC runs, except the last one in each bitmap, contain exactly one set bit. This leads to a maximum of $N + b$ runs in the entire index.

The key to compute an upper bound of the index size is to show that having the maximum number of runs indeed requires more space. This basic idea is formalized in the following theorem.

Theorem 3 *For sparse bitmaps ($n < N/100$), the maximum size of a BBC compressed bitmap is achieved when each each run has at most one set bit and the last run has no set bit.*

Proof. We prove this theorem by contradiction, i.e., by showing that merging set bits together would not increase the storage required.

Let runs A and B be two arbitrary runs of a sparse bitmap, and let C be the run immediately following B. Runs A and C may be the same. If the set bit in B is moved to the tail byte of run A, then run A changes from type 4 to type 3 or from type 2 to type 1, because the tail byte is no longer a special byte. This causes the tail byte of A to be explicitly stored and the new run A requires one more byte than before.

To see what happens to the runs B and C, we first assume both of them are type 4 runs. After removing the set bit from run B, runs B and C will merge to form a longer fill. Let f_B and f_C denote the fill lengths of B and C before merging. The length of the combined fill is $f_B + f_C + 1$, since the tail byte of B is now part of the fill. The multi-byte counter for run B has $(1 + \lfloor \log_{128}(f_B - \delta) \rfloor)$ bytes, and the multi-byte counter for run C has $(1 + \lfloor \log_{128}(f_C - \delta) \rfloor)$ bytes. In most cases, the length of the combined fill can be represented using the same number of bytes as the larger of these two counters. In these cases, the space used to store the shorter of B and C is removed. This removes at least two bytes. Since only one extra byte is used by run A, moving the set bit from B to A reduces the total size by one or more bytes.

If the set bit is moved from another run to run A again, the size of A will not change, and the total size will definitely decrease. In general, if a run has multiple set bits, the total size of the compressed bitmap would be smaller than if all the runs have only one set bit.

The above is for the normal cases where combining B and C does not increase the size of the counter. Next, we examine the special cases where the combined fill takes more bytes to represent. An important observation is that the combined fill needs at most one more byte than the larger one of the two old counters. This can happen when both f_B and f_C are smaller than an integer power of 128, but $f_B + f_C + 1 - \delta$ is not. For example, with the 2-sided BBC, when $f_B = 66$ and $f_C = 65$, counters for both runs are one-byte long, but the combined fill has 132 bytes which requires a two-byte counter. In this case, moving the set bit from B to A may reduce the total size of the compressed bitmap if the shorter one of B and C takes more than two bytes. It does not change the total size if the shorter one of B and C takes exactly two bytes.

If either B or C has only a short fill, i.e., the run requires only one byte to represent, then removing the set bit of B actually increases the total size of the compressed bitmap by one byte. Let us assume C has a short fill, in order for the combined fill to use one more byte than that of B, the fill length of B

must be fairly close to an integer power of 128. In this case, some fill bytes can be moved from B to C to make run C a long fill, this increases the total storage required. Therefore, the sparse bitmap that has the maximum compressed size must not have any short fills. Similarly, if B is the last run of the bitmap and C does not exist, the total size of the compressed bitmap can be increased by moving some fill bytes from B to C. Therefore the last run of a compressed bitmap with the maximum size should have no set bit. ■

In a bitmap index, there is always a few bitmaps whose last run contain set bits. These bitmaps will be slightly smaller than the maximum computed here.

3.3 Average Size of a Run

If the set bit from B were moved to the middle of A's fill, run A would be split in two. This does not change the number of runs, but changes the distribution of 0-fills. Next we construct a way to rearrange the 0-fills of an infinite bitmap to maximize the average number of bytes required to represent each run.

If a multi-byte counter is m-byte long, the minimum fill length[4] it represents is $128^{m-1} + \delta$. If each fill is the minimum length for the counter, then the number of bytes required by the counters would be maximized. This is the core idea behind the following construction.

If f is the average number of bytes in a fill, the size of the multi-byte counter is $m = 1 + \lfloor \log_{128}(f - \delta) \rfloor$. To use the same size counter, the minimal fill length is $\underline{f} = 128^{m-1} + \delta$. We can take away $f - \underline{f}$ bytes from each fill without decreasing the size of the counter used, and these bytes can be added to some fills to increase the sizes of their counters. The number of bytes required to make an average fill use $(m+1)$ byte is $\overline{f} = 128^m + \delta$, which needs $\overline{f} - f$ new bytes. We need to take the excess bytes from γ $(\equiv (\overline{f} - f)/(f - \underline{f}))$ runs in order to make one run have a larger counter. After this redistribution, we have γ runs with $(1+m)$ bytes for every run with $(2+m)$ bytes. The average number of bytes needed to represent a run is

$$(\gamma(1+m) + 2 + m)\,/(\gamma + 1).$$

It easy to see that there is no fill that takes m or less bytes, because some bytes can be taken away from a run with $2+m$ bytes and make many shorter ones with $1+m$ bytes.

In the above construction, we assume that the bitmap has infinite number of bits, otherwise there may not be enough 0-fills to increase the bytes used by any counter. This indicates that the average number of bytes used by an infinite bitmap is an upper bound for a finite bitmap.

3.4 Size of a Compress Bitmap

[4]For $m = 1$, the minimum fill length is δ, not $1 + \delta$ as the formula shown. This will cause the average fill size computed to be less accurate for denser bitmaps, particularly those with $n \sim N/100$.

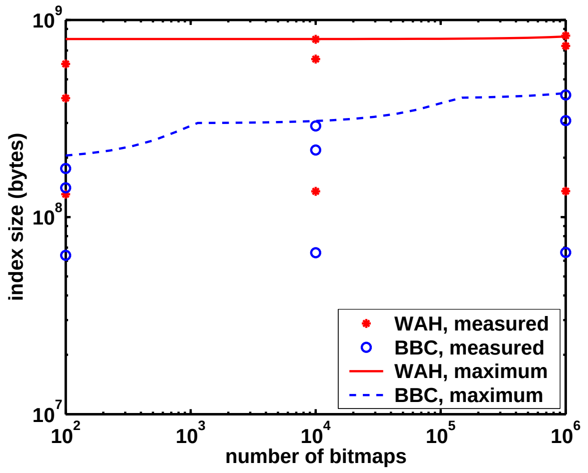

Figure 3: Sizes of compressed bitmap indices on a synthetic dataset ($N = 10^8$).

For ease of estimation, let us assume the index is not segmented. Thus the number of bitmaps b is the attribute cardinality c. The total number of bits in the entire bitmap index is Nc, the total number of runs is $N + c$, and the average number of bits in a run is about $Nc/(N + c)$. Plugging in the formula for the average number of bytes per run, we have the following theorem for the maximum size of a compressed index.

Theorem 4 *Let N denote the number of tuples in a relation, and let c denote the cardinality of the attribute indexed ($c > 100$), then the maximum number of bytes in a BBC compressed bitmap index is approximately*

$$(N + c)\,(\gamma(1+m) + 2 + m)\,/(\gamma + 1),$$

where $\gamma = (\overline{f} - f)/(f - \underline{f})$, $\underline{f} = 128^{m-1} + \delta$, $\overline{f} = 128^m + \delta$, $m = 1 + \lfloor \log_{128}(f - \delta) \rfloor$, $f = \frac{Nc}{8(N+c)} - 1$.

When $f = \underline{f}$, the maximum size is $(N + c)(1 + m)$.

To verify the above formulas, we show the sizes of actual bitmap indices against the maximum values given by Theorems 2 and 4. The results are shown in Figures 3 and 4. Figure 3 shows the index sizes on some synthetic data and Figure 4 shows the index sizes on a set of real application data. The solid lines are based on the formula given in Theorem 2 and the dashed lines are based on the formula given in Theorem 4. These predicted maximum sizes are achieved with bitmap indices on uniform random attributes. Indices for other attributes are smaller than the predicted maximum.

4 Logical Operation on Two Bitmaps

It was observed that the time complexity of a bitwise logical operation is proportional to the total size of two compressed operands [16, 17]. It is straightforward to count the number of operations required by

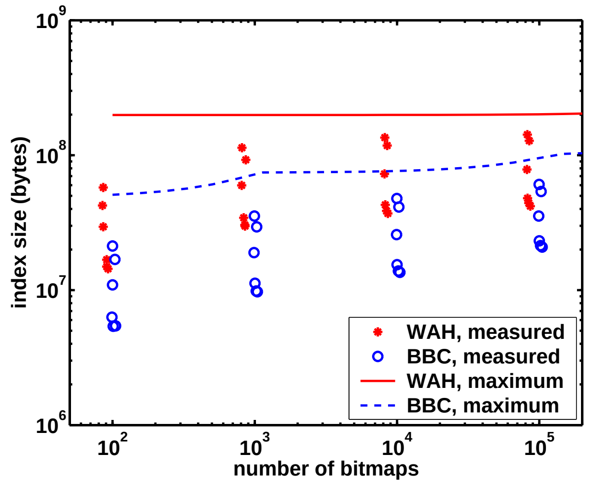

Figure 4: Sizes of compressed bitmap indices on a combustion dataset ($N = 2.5 \times 10^7$).

examining the algorithms used to perform these operations. However, due to space limitation we would not discuss the details. Instead, we describe the major steps performed in these operations and provide the upper bound for the time and space complexities of the algorithms.

With either WAH or BBC, a bitwise logical operation on two sparse bitmaps basically need to decode the compressed bitmaps, determine the result bitmap one piece at a time, and put the pieces together to form the compressed result. It is easy to see the time required to decode the operands is proportional to the size of input bitmaps. With an appropriate algorithm, the pieces produced at each step should be as large as possible. For sparse bitmaps, the result contains at most the same number of runs as the total number of runs in the two input bitmaps. Therefore, the size of the result, and the time required to compute and compress the result are also proportional to the total size of the input bitmaps. Overall, both the time and space complexity of a bitwise logical operation is linear in the total size of the input bitmaps. This is optimal because an arbitrary logical operation has to at least examine every byte of the input operands.

We have measured the result sizes of many bitwise OR operations. The results are displayed in Figure 5. In this plot, the dashed line along the diagonal shows that the result size is exactly equal to the total size of the two input bitmaps. Each dot is a test case on two random bitmaps. For sparse bitmaps, i.e., those with small total sizes, the result size is very close to the total size of input bitmaps. As the sizes of the input bitmaps become larger, the size of the result becomes less than the total size of the input bitmaps. Larger bitmaps have more runs, therefore more literal tails. This increases the likelihood of two literal bits from the input bitmaps being located close enough to each other to produce tails that contain multiple 1s. This reduces the number of runs, and consequently reduces

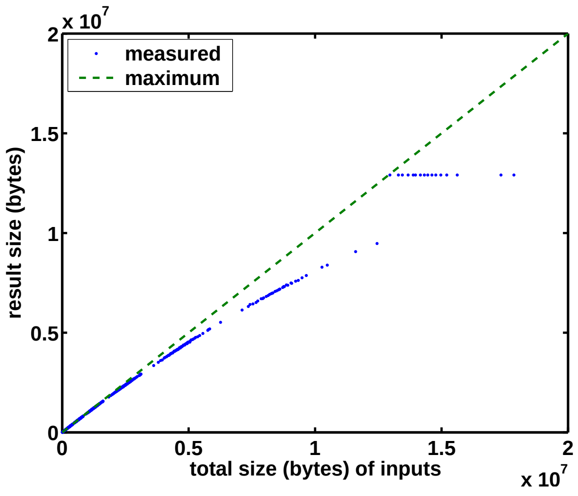

Figure 5: The result size of a logical operation plotted against the total size of two operands ($N = 10^8$).

the size of the resulting bitmap. The result with the maximum size is an uncompressed bitmap. In this case, its size is about 13 MB. When the total size of two input bitmaps are larger than 13 MB, the result is always 13 MB.

The above shows the worst case behavior of binary logical operations. Though these worst cases are often achieved, an actual operation may be a lot faster. For example, when performing an AND operation, if one of the operands is a single 0-fill, then the result is also a single 0-fill no matter what the other operand is.

5 Algorithms for Many Bitmaps

For a high cardinality attribute, the basic bitmap index contains many bitmaps. To answer a query such as "find all records with attribute I less than 100", one may need to OR a large number of bitmaps. Without compression, the execution time could be much longer than scanning the projection of the attribute I. We propose that the compressed bitmap index is efficient in this case. One evidence supporting this proposition is that the total size of a compressed bitmap is relatively small. With WAH compression, the compressed index sizes is about $2N$ words for high cardinality attributes where $c \ll N$. The size of a BBC compressed bitmap index is even smaller. In any series of OR operations, at most half of the bitmaps are involved. If more than a half of the bitmaps are required, it is easy to use the remaining bitmaps to compute the complement of the solution. Using WAH compressed index, we need to access no more than N words. In many data warehousing applications, the projection index[5] is considered the most efficient scheme when the attribute cardinality is high [10]. For an attribute whose values take one word each to store, the projection index requires N words. Using the projection index one

[5]The projection index is typically implemented as a sequential scan of a materialized view of a projection.

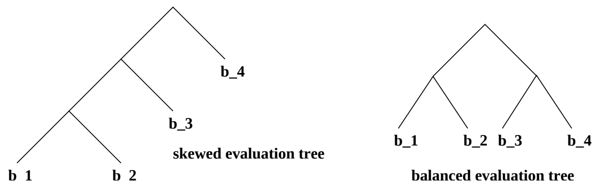

Figure 6: Two evaluation trees. A balanced tree is shorter and reduces the evaluation cost.

always accesses all N words, but using a compressed bitmap index one accesses N words only in the worst cases.

Previously we discussed operations on two compressed bitmaps. Next we discuss five different strategies to perform OR operations on many compressed bitmaps. The goal is to find a strategy that performs the best.

Let us denote the bitmaps $b_1, b_2, \ldots, b_k$, and denote their sizes in bytes as $s_1, s_2, \ldots, s_k$. The first option is a simple application of the binary logical OR operator, which can be expressed as a simple `for` loop, where the variable r denotes the result and the operator $|=$ denotes a bitwise OR operation that stores the result back to the variable on the left hand side.

```
$r = b_1$;
for $i = 2$ to $k$, do $r \mathrel{|}= b_i$.
```

To see the worst case behavior of this approach, we assume the result of a bitwise OR operation always has the maximum size. In addition, we assume that the time required to perform a bitwise OR operation is exactly proportional to the total size of the two input bitmaps. Let C_c be the proportional constant. The total time required to complete $k-1$ logical operations is

$$\begin{aligned} t_1 &= C_c(s_1+s_2) + C_c(s_1+s_2+s_3) + \ldots \\ &= -C_c s_1 + C_c \sum_{i=1}^{k}(k+1-i)s_i. \qquad (1) \end{aligned}$$

If all the compressed bitmaps have exactly the same size s, then the above equation simplifies to

$$t_1 = C_c s(k+2)(k-1)/2.$$

This option is very easy to implement and requires the minimal number of bitmaps in memory. In the following tests, we refer to this as option 1.

Since the multiplier in front s_i decreases gradually, we can order the bitmaps so that the smaller ones are in the front to decrease the overall cost. This approach of sorting the bitmaps according their sizes is referred to as option 2 in the following tests. Clearly, sorting the bitmaps does not change the worst case complexity, which is still quadratic in k.

Pictorially, the evaluation process of options 1 and 2 can be depicted as a skewed binary tree, which we call the *evaluation tree.* It is easy to see that balancing the evaluation tree will reduce the multipliers in front of the variables s_i in the expression for the total time. If all the bitmaps have the same size s, and $k = 2^h$, where h is an integer, the total time required using a balanced evaluation tree would be

$$t_3 = C_c sk \log_2(k). \qquad (2)$$

As k becomes large, this approach clearly is better than the two previous options. For our implementation, we use a priority queue to hold all input bitmaps and intermediate results. The priority queue puts the smallest bitmap on the top. Every binary OR operation is then performed on two bitmaps from the top of the queue. This ensures that the cheapest operations are performed first. It effectively implements the balanced evaluation tree without explicitly maintaining a tree. We refer to this as option 3.

Both option 2 and 3 require the sizes of all input bitmaps before any operation can be carried out. Typically, this means the input bitmaps have to be read into memory. Therefore, these options require more memory than option 1. If these input bitmaps are held in memory during the whole process, a maximum of three times the total size of the input bitmaps may be required near the end of the process. If we free the input bitmaps immediately after they are used, the factor goes down from three to two. This amount of space is required to store the last two intermediate results and the final result.

All previous approaches use compressed bitmaps as the result of bitwise logical operations, which requires bitmaps to be generated and destroyed for each intermediate result. The cost of which may become a significant portion of the total execution time. One way to avoid this is to use an uncompressed bitmap to store the result. In fact, one uncompressed bitmap can be used for all intermediate results and the final result. Since the uncompressed bitmap is not deleted or allocated repeatedly, it might reduce the overall cost of the operations especially for a large number of bitmaps. We have implemented two variations of this approach, which we call option 4 and 5. Option 4 decompresses the first input bitmap, and option 5 decompresses the largest input bitmap. Normally, in any compressed bitmap, fills are stored in a compact form. The decompression procedure explicitly forces all fills to be stored literally, therefore turns a compressed bitmap into an uncompressed one.

In our implementations of options 4 and 5, we use the same data structure for both compressed and uncompressed bitmaps. Since we use compressed data structures to store uncompressed bitmaps, we pay a small percentage of storage overhead. However, this allows us to efficiently operate between uncompressed and compressed bitmaps. When the left-hand side of the operator $|=$ is an uncompressed bitmap, it always writes back the result into the uncompressed bitmap without allocating any new storage. In later discussions, we refer to this as the *in-place OR operation* or the in-place operation.

The in-place OR operation can be implemented very efficiently. In tests, we observed that the time required for these functions are linear in the total size of the compressed bitmaps. A detailed analysis is beyond the scope of this paper, here we give the key ideas that support the observations. The in-place OR operation can be viewed as a function that modifies the uncompressed bitmap to add more 1s. These new 1s are from the compressed operand. With either BBC or WAH, it is straightforward to determine the position of these 1s and the cost of determining these positions is proportional to the size of the compressed bitmaps. The number of words or bytes that need to be modified is determined by the number of runs in the compressed operand. If there are lots of bitmaps to be ORed, each bitmap must be very sparse. For sparse bitmaps, it is very rear to have 1-fills. For each run in the compressed bitmap, only one word or a small number of bytes of the uncompressed bitmap need to be modified. The number of runs in a compressed bitmap is proportional to its size. Overall, the time required to modify an uncompressed bitmap with a compressed bitmap is linear in the size of the compressed bitmaps. The constant term in the linear expression comes from the initial time required to generate an uncompressed bitmap with only 0s. Since this initial cost depends on N, it does not qualify to be a constant in the strictest sense. However, because this initial cost is so small compared with others, when a large number bitmaps is involved, this initial cost is negligible, and the total execution time is indeed proportional to the total size of the input bitmaps.

We can express the time required by option 4 as

$$t_4 = C_d + C_i \sum_{i=1}^{k} s_i, \tag{3}$$

where C_d is the constant time required to generate a uncompressed bitmap and C_i is the per byte cost of performing the in-place logical operation. For option 5, there is an extra cost of finding the largest bitmap, which should be relative small. Since it is also proportional to k, it does not change the theoretical complexity.

Figure 7 contains a summary of the five options discussed. Clearly, for a large number of bitmaps, the best option is either option 4 or 5, and for a small number of bitmaps, one of the first three options might be better. In next section, we examine their relative performance and determine a way to combine them to always achieve the best performance. In the rest of this section, we briefly compare these algorithm against a theoretical optimal one.

Without considering the setup cost, the minimal cost of any searching algorithm is proportional to the size of the search result, because it needs to enumerate the result. Let h denote the number of hits, a hypothetical optimal search algorithm would have both time and space complexity of $\mathcal{O}(h)$. Using the basic bitmap index to answer a range query, the number of hits is the number of set bits in all bitmaps involved. In the worst case, with both BBC and WAH compression, the sizes of the bitmaps are proportion to the number of set bits, $sk \propto h$. This shows that all complexity expressions of the form $\mathcal{O}(sk)$ are optimal. More specifically, the space complexities of options 1, 2 and 3 are optimal and the time complexities of options 4 and 5 are optimal.

6 Selecting the Best Algorithm

From analyses, we know that different algorithms have different performance characteristics, to achieve the best performance in practice, we need to dynamically select the best algorithm for answering a particular query. To address this issue, we start by measuring the performance of all five options. By analyzing their relative performances, we come to a simple combined strategy for ensuring the best overall performances.

For measuring the performance of the five different options outlined in the previous section, we tested a large number of range queries on two sets of data, a set of random integers with various distributions and a set from a combustion simulation [7, 15]. Most of the tests are performed on the random data set because their bitmap indices are closer to the predicted worst case sizes. The real application data show significant skewness which makes the bitmap indices much smaller, see Figure 4.

The timing tests are conducted on a Linux machine with 2.8 GHz Pentium IV Xeon processor and a small hardware RAID with two SCSI disks. The machine has 1 GB RAM and the maximum reading speed of the disk system is about 80 MB/s. For sequential scan of large amounts of data, it actually sustains a read speed of about 40 MB/s. To scan 100 million records of a projection index, 400 MB in size, it takes about 10.3 seconds. This is the performance of projection index, which we use as the yard stick to measure the performance of other indices.

The random data set contains 100 million tuples of discrete random attributes some following a uniform distribution and other following different Zipf distributions. Their sizes are shown in Figure 3. The attributes with the uniform distribution have the largest bitmap indices compared with other attributes of the same cardinalities. Timing results from bitmap indices on these attributes also follow the formulas more closely.

We have generated bitmap indices with both WAH and BBC compression. The time used to perform bitwise OR on different number of bitmaps are shown in Figures 8 and 9. Each point in the plots shows one

option	description	memory		time
1	unordered bitmaps, compressed result	three compressed bitmaps	$\mathcal{O}(sk)$	$\mathcal{O}(sk^2)$
2	ordered bitmap, compressed result	all input bitmaps plus two intermediate results	$\mathcal{O}(sk)$	$\mathcal{O}(sk^2)$
3	priority queue, compressed result	all input bitmaps plus many intermediate results	$\mathcal{O}(sk)$	$\mathcal{O}(sk\log_2(k))$
4	decompressed first bitmap, uncompressed result	one uncompressed bitmap plus one compressed	$\mathcal{O}(N)$	$\mathcal{O}(sk)$
5	decompressed largest bitmap, uncompressed result	one uncompressed bitmap plus one compressed	$\mathcal{O}(N)$	$\mathcal{O}(sk)$

Figure 7: Summary of the five options used to OR many compressed bitmaps, where N is the number of bits in a bitmap, s is the average size (bytes) of the bitmaps involved and k is the number of bitmaps.

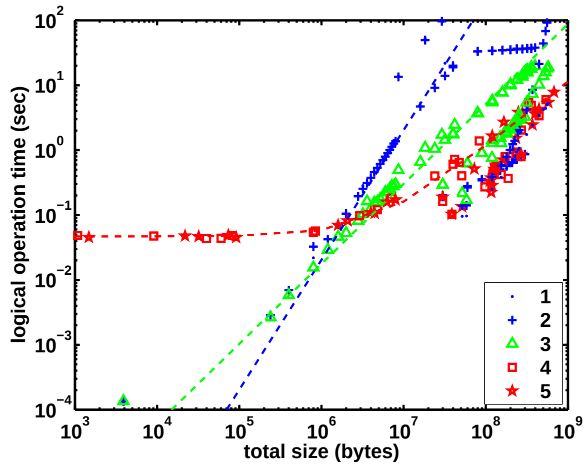

Figure 8: Time to OR many bitmaps compressed with WAH. Dashed trend lines are defined by formulas given in Figure 7.

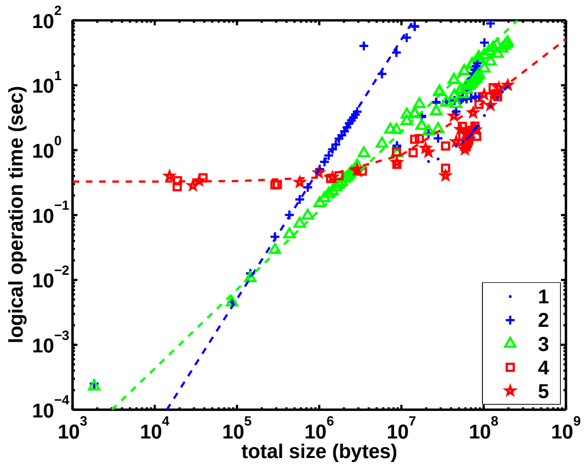

Figure 9: Time to OR many bitmaps compressed with BBC. Dashed trend lines are defined by formulas given in Figure 7.

timing measurement. The total size of the bitmaps is used as the horizontal axis. Assuming the bitmaps are the same size, the time required for the various options should follow the complexity formulas given in Figure 7. The dashed lines in the figures are the *trend lines* defined by these complexity formulas. The bitmaps from the indices for the uniform random attributes are about the same size. The time required to operate on these bitmaps basically follow the trend lines.

Of the five options, options 1 and 2 use about the same amount of time in many cases; options 4 and 5 take about the same amount of time in every case. This suggests that option 4 should be used since option 5 needs to find the largest bitmap. The cost to decompress a bitmap dominates the execution time when the total size of the input bitmaps is small. To decompress 100 million bits, it takes about 0.05 seconds with WAH compression and 0.33 seconds with BBC compression. Let S denote the total size (bytes) of the input bitmaps, the trend line for options 4 and 5 drawn in Figure 8 is $t = 0.05 + 1.1 \times 10^{-8}S$, and the same trend line in Figure 9 is $t = 0.33 + 5.1 \times 10^{-8}S$.

We also notice that there is a significant number of test cases that are far from the trend lines. This is largely because the trend lines are established for the worst cases.

To find out which option to use for a particular set of bitmaps, we plot the best options for the sets of bitmaps tested. Figure 10 shows the best options for WAH compressed bitmaps and Figure 11 shows the best options for BBC compressed bitmaps. As expected, option 3 is better for a small number of bitmaps and bitmaps with small sizes, but options 4 and 5 are better for a large number of bitmaps and bitmaps with large sizes. In each plot, we have drawn a dashed line to separate the region dominated by option 3 from the rest. The dashed lines separate the regions fairly cleanly. However, there are some cases with a small number of bitmaps, where option 1 is the best. We have examined these cases and found the performance differences between option 1 and option 3 to be very small. For ease of implementation, we will only use option 3 in these cases.

Option 1 also showed up in many test cases with very large bitmaps. In these cases, the first few bitmaps are relatively large and the intermediate results produced from operating on these bitmaps

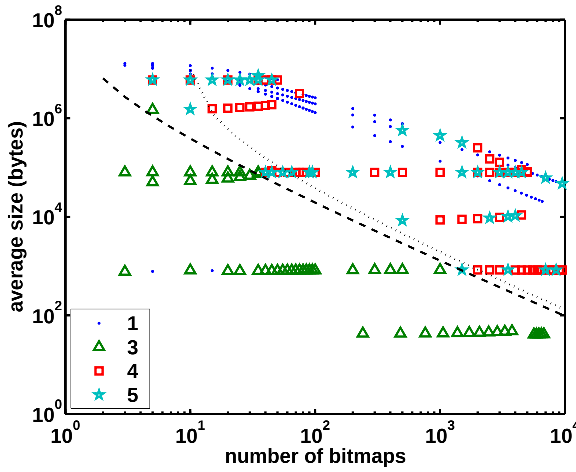

Figure 10: The best option to perform bitwise OR on WAH compressed bitmaps.

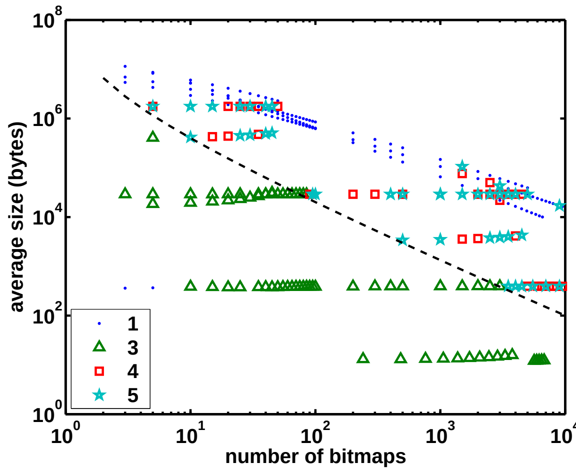

Figure 11: The best option to perform bitwise OR on BBC compressed bitmaps.

quickly become uncompressible. This effectively turns option 1 into option 4 because the same operator |= is used in the implementation of both options. This suggests that if the total size of first two bitmaps is larger or equal to the size of an uncompressed bitmap, option 1 should be used to avoid explicitly decompressing any bitmap.

In cases where option 1 works well, we expected option 2 to do even better. This turns out not to be the case because option 2 delays the generation of the uncompressible results and increase the time spent in generating intermediate results.

To perform operations on a small number of bitmaps, say two or three, clearly, it is best to use option 1. In cases where the first two bitmaps are very large we should also use option 1. Outside of these cases, the two primary options to consider are option 3 and option 4. The dashed lines shown in Figures 10 and 11 suggest a way to choose between the two. All cases above the lines should use option 4 and all those below the lines should use option 3. Next, we explain how we draw the lines.

Since we have derived the estimated time for all options, one way to decide whether to use option 3 or option 4 is to compare their expected time t_3 and t_4, and use the one with a smaller expected execution time. In this case, the divider would be defined by equation $t_3 = t_4$. Let s denote the average size of the bitmaps and let k denote the number of bitmaps. The dividing line is given by the following equation.

$$s = \frac{C_d}{k(C_c \log_2(k) - C_i)}. \tag{4}$$

To use this equation, we need to estimate three parameters, C_c, C_d and C_i. We have computed C_c as the average of $t_3/(nk\log_2(k))$ for all the test cases, and used a linear regression to compute the parameters C_d and C_i from the measured results. The line for WAH is plotted as the dotted line in Figure 10. It is easy to see that the dotted line does not divide the points cleanly. This is because Equations 2 and 3 are derived for the worst case scenarios. More importantly, to use Equation 4, we have to estimate three parameters. For this reason, our actual implementation uses the following equation to decide whether to use option 3 or option 4.

$$sk\log_2(k) = C, \tag{5}$$

where C is the size of one uncompressed bitmap. We use this equation to define the divider because it works well for both compression schemes and there is no parameter to estimate.

There is a small number of cases where the triangles representing option 3 fall on the wrong side of the line defined by Equation 5. However, the difference between using option 3 and 4 is relatively small in these cases. For example, for the triangle that is above the line in Figure 10 with 5 bitmaps (total size about 7.4 MB), the time spent using option 3 is 0.11 seconds and using option 4 is 0.12 second. In these cases, using either option 3 or option 4 gives reasonable performance.

When performing a bitwise logical operation on two bitmaps, we have found that if the total size is greater than that of one uncompressed bitmap, it is faster to decompress one operand and use the in-place operation to produce an uncompressed result [18]. This indicates that with two bitmaps the dividing line between option 3 or 4 is $s_1 + s_2 = C$. Equation 5 can be viewed as an extension of this observation based on the expected execution time, see Equation 2.

Analyses show that the time required to answer a query using the compressed bitmap indices is proportional to the total size of bitmaps involved and to the size of the search result. Figure 12 plots the time measurements against the number of hits in the synthetic dataset. In this figure, the total time refers to the total query processing time, including the time to operate

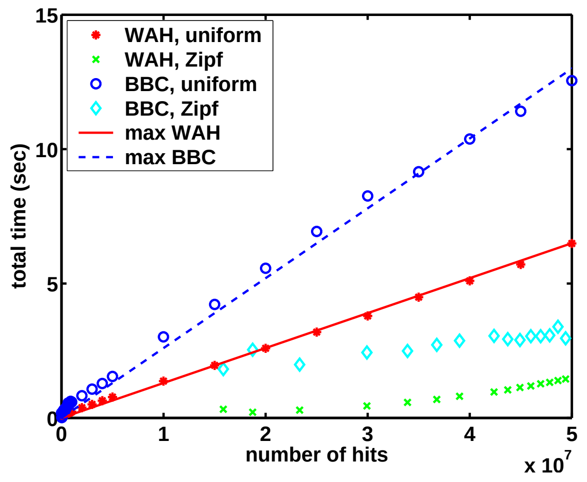

Figure 12: The total query processing time plotted against the number of hits for two type of random attributes, uniform and Zipf$(1/x)$.

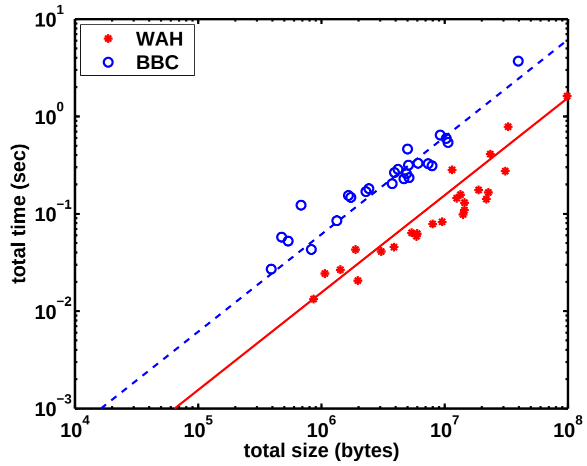

Figure 13: The total query processing time plotted against the total size of bitmaps used for the combustion dataset.

on the bitmaps and the time to read the bitmaps from disk. The total time shown here is measured using the combined option for operating on many bitmaps. The linear relation between the total time and the number of hits is clearly evident from this plot. Because of the use of the complement when more than half of the bitmaps are involved, the query processing time for uniform random attributes actually decreases when more than half of the records are hits. The time required using WAH compressed indices is about half of that using BBC compressed indices.

Figure 13 shows the timing results on the combustion dataset against the total size of the bitmaps involved. The total time is measured using the combined option and also includes time for all IO operations. The solid line and the dashed line shown are the average cases assuming the total time is proportional the total size of the bitmaps involved. Even though linearity is only expected for some cases, we see that the timing results follow the linear relation fairly closely.

random, $N = 10^8$		
	average	**max**
projection	10.3	10.3
WAH	1.2	6.8
BBC	2.8	12.5
combustion, $N = 2.5 \times 10^7$		
	average	**max**
projection	2.6	2.6
WAH	0.2	2.7
BBC	0.4	3.9

Figure 14: The average and worst case time (seconds) used by various searching schemes.

Figure 14 shows how the bitmap indexing schemes compare with the projection index. On the average, both types of compressed bitmap indices are significantly faster than the projection index. In the worst cases, the WAH compressed bitmap indices are no worse than the projection index, but the BBC compressed indices may take longer because operations on BBC compressed bitmaps are slower.

The relative performance difference between WAH and BBC compressed bitmap indices in these tests is at the low end of the performance differences measured in previous tests [17]. This is consistent with the fact that most of the bitmaps used in these tests are very sparse. On denser bitmaps, the performance difference can be much larger.

7 Summary

The effectiveness of the bitmap indexing scheme for low cardinality attributes is well accepted. There are also evidences that compressed bitmap indices can work well for high cardinality attributes [16, 17]. To fully understand their effectiveness for high cardinality attributes, we analyze the space and time complexities of WAH and BBC compressed bitmap indices for answering one-dimensional range queries. The analyses show that the total sizes of the compressed bitmap indices are fairly modest even for attributes with very high cardinalities. For most high cardinality attributes, where $c \ll N$, the WAH compressed indices use about $2N$ words, which is about half the size of a typical B-tree index. The BBC compressed indices are even smaller. We also develop a strategy to select the best algorithm to operate on a large number of bitmaps. This strategy is important for us to achieve predicted optimal speed in practice. Timing measurements confirm this optimality because the query response time is indeed linear in the number of hits for uniform random attributes. The query response time on other types of attributes is much lower than that for uniform random attributes.

Event though bitmap indices compressed with both BBC and WAH are theoretically optimal. On the average, WAH compressed bitmap indices are about twice as fast as BBC compressed indices in our tests. Because the projection index is often the best option for searching high dimensional datasets, we also measured its performance. On the average, bitmap indices compressed with both BBC and WAH can significantly outperform the projection index. In the worst cases, the WAH compressed indices take no more time than the projection index, but the BBC compressed indices may take longer because they require more CPU time.

We currently have a prototype software that can use both WAH and BBC compressions. In the future, we plan to implement a more robust version based solely on WAH compression. The software would include segmented indices mentioned earlier. The prototype implementation currently makes a number of decisions based on the number of bitmaps involved rather than on the total size of the bitmaps involved. For example, on attributes with non-uniform distributions, this may lead to wrong decisions on when to compute the complement of the query conditions. According to the analyses presented in this paper, the decision of when to use the complement should be based on the total size of the bitmaps involved.

References

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

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

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

[4] T. C. Bell, I. H. Witten, and J. Cleary. *Text Compression.* Prentice Hall, 1989.

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

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

[7] H. G. Im, J. H. Chen, and C. K. Law. Ignition of hydrogen/air mixing layer in turbulent flows. In *Twenty-Seventh Symposium (International) on Combustion, The Combustion Institute*, pages 1047–1056, 1998.

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

[9] P. O'Neil. Model 204 architecture and performance. In *2nd International Workshop in High Performance Transaction Systems, Asilomar, CA*, volume 359 of *Lecture Notes in Computer Science*, pages 40–59, September 1987.

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

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

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

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

[14] 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.

[15] K. Wu, W. Koegler, J. Chen, and A. Shoshani. Using bitmap index for interactive exploration of large datasets. In *SSDBM 2003*, pages 65–74, 2003.

[16] K. Wu, E. J. Otoo, and A. Shoshani. A performance comparison of bitmap indexes. In *Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management, Atlanta, Georgia, USA, November 5-10, 2001*, pages 559–561. ACM, 2001.

[17] K. Wu, E. J. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *SSDBM'02*, pages 99–108, 2002. LBNL-49627.

[18] K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg. Notes on design and implementation of compressed bit vectors. Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory, Berkeley, CA, 2001.

[19] M.-C. Wu and A. P. Buchmann. Encoded bitmap indexing for data warehouses. In *ICDE 1998*, pages 220–230. IEEE Computer Society, 1998.

[20] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.