LBNL-54673

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

March 5, 2004

Abstract

It is well established that bitmap indices are efficient for read-only attributes with a small number of distinct values. For an attribute with a large number of distinct values, the size of the bitmap index can be very large. To overcome this size problem, specialized compression schemes are used. Even though there is empirical evidence that some of these compression schemes work well, there has not been any systematic analysis of their effectiveness. In this paper, we analyze the time and space complexities of the two most efficient bitmap compression techniques known, the Byte-aligned Bitmap Code (BBC) and the Word-Aligned Hybrid (WAH) code, and study their performance on high cardinality attributes. Our analyses indicate that both compression schemes are optimal in time. The time and space required to operate on two compressed bitmaps are proportional to the total size of the two bitmaps. We demonstrate further that an in-place OR algorithm can operate on a large number of sparse bitmaps in time linear in their total size. Our analyses also 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. We also verify the analytical results with extensive tests, and identify an optimal way to combine different options to achieve the best performance. The test results confirm the linearity in the total size of the compressed bitmaps, and that WAH outperforms BBC by about a factor of two.

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 [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. This index is only efficient for low cardinality attributes where there are only few distinct values. Many strategies have been devised to extend the bitmap index to high cardinality attributes by using more compact encoding strategies [5, 6, 10, 13], binning [11, 12, 14, 19], and compression [2, 3, 16, 17]. This paper presents analyses and performance tests on two of the most efficient compression schemes, and shows that with compression, the bitmap indexing scheme is 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, i.e., its cardinality is c, then there are c bitmaps with N bits each. Each bitmap represents a particular value. It has a position set to 1 if that position has that particular value. For example, in a bitmap index for an integer attribute in the range of 0 ... $(c-1)$, the ith bit of the jth bitmap is 1 if the attribute's value in the ith tuple equals to j. Such a bitmap requires $cN/8$ bytes to store it without compression. 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 $3-4$ times of 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 "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 of them are based on run-length encoding. They both represent a long sequence of 0s or 1s using a counter, and represent a mixture of 0s and 1s in their literal sequence.

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 worst case behavior of the two compression schemes. 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, the most stringent definition of optimality is that its time complexity is linear in the number of hits. Using this definition, the compressed bitmap index is also optimal for high cardinality attributes because the total size of the bitmaps involved in answering a query is proportional to the number of hits.

We also show extensive performance tests to verify the analyses. To implement an efficient software, an important consideration is how to operate on many bitmaps efficiently, more specifically when to use the in-place algorithm. Among the various options studied is a strategy to combine them so that we always achieve the best performance. Tests show that the bitmap indices compressed with both WAH and BBC scale linearly with the total size of bitmaps involved. A WAH compressed index typically uses about half the time required by a BBC compressed index to answer the same query. In addition, a WAH compressed index always outperforms the projection index [10], but a BBC compressed index takes more time in some test cases.

The remaining of this paper is organized as follows. Sections 2 and 3 contain our analyses of the worst case sizes of bitmap indices compressed with WAH and BBC. Section 4 has a discussion on the time complexity of bitwise logical operations on two bitmaps. Section 5 has the case of bitwise logical operations on a large number of bitmaps, and shows 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 combine various options of performing 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 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 between the two types. In a literal word, the remaining bits are used to store literal bit values. To improve operational efficiency, WAH requires each fill to contain an integer multiple of bits stored in a literal word. For example, if a word on a machine contains 32 bits, a literal word can store 31 bits from the bitmap, then each fill must contain a multiple of 31 bits. In a fill word, the length of a fill is recorded as the multiple of literal word size. Thus, on a 32-bit machine, a fill of 93 bits would be recorded to have length 3. In a fill word, one bit is used to distinguish it from the 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 to 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 $2^{30} - 1$. 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 into literal words, say, 31 bits each. If there are two or more consecutive groups with only 0s (or 1s), these consecutive groups form one fill and can be represented in one fill word. All remaining groups have to be represented literally. Let us call a fill followed by a group of literal bits as a *run* and call the literal bits in a run the *tail*. As in BBC, we allow the leading fill to be empty to accommodate consecutive literal groups. A run in a bitmap takes at most two words to represent it; 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 to use two words for it. 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 $n + 1$ run. 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 in an index 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, the maximum number of words required by the entire 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 not 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.

If each bitmap in the index spans all tuples of the relation, the number of bitmaps is the number of distinct values, i.e., the attribute cardinality, $b = c$. This unsegmented index has the least number of bitmaps and is usually smaller than the segmented indices. When a bitmap index is segmented, the number of bitmaps b is larger than c. The only exception is when $c = N$, then b is also N. When c is much smaller than the number of tuples in each segment, then it is likely that each segment needs c bitmaps, and a total of cg bitmaps are needed, where g is the number of segments. Generally, the more segments there are, the more bitmaps would be used. However, there are also strong reasons for segmenting an index. A robust implementation of any bitmap indexing scheme must carefully balance these considerations. For theoretical analyses, the increase in index size due to segmentation is reasonably straightforward to account for. The increase of the index size is strictly proportional to the increase of the number of bitmaps. To ease the analyses and comparisons, we concentrate our effort mostly on unsegmented indices from now on.

3 Sizes of BBC compressed bitmap indices

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 of the run. 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 addition to dividing runs according to the length of their fills, BBC also identifies a special type of tail bytes. A special tail byte is one that has one bit different from the rest, and the rest of the bits are the same as the bits of the preceding fill. A tail may have one such a special byte. In this case the tail 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 can be very common in sparse bitmaps. In its simplest form, a BBC compression scheme divides runs into four types [2, 8]. Run type 1 has a short fill with up to 15 tail bytes that are not special. Run type 2 has a short fill and one special tail byte. Run type 3 has a long fill and up to 15 tail bytes, and run type 4 has a long fill and one special tail byte.

For attributes with relatively small cardinalities, say $c < 100$, users are confident that the bitmap index is more efficient than the alternative indexing schemes, e.g., a B-tree. For this reason, we will only examine the size of a compressed bitmap index for an attribute with relatively high cardinality. In the following discussion, we define a *sparse bitmap* to be one that has fewer than 1 set bit out of 100 bits. As in a WAH compressed bitmap, except the last run, any other run has at least one set bit. The worst case size of BBC can be computed by assuming that all BBC runs, except the last one, contain exactly one set bit. This leads to a maximum of $N + b$ runs in the entire index. Since the total number of 1s in a bitmap index is fixed, if any run contains multiple 1s, the number of runs will be smaller, and the total bytes required to store the compressed bitmap index would also be smaller. This basic idea is formalized in the following theorem.

Theorem 3 *Let N denote the number of bits in a bitmap and let n denote the number of set bits. If $n < N/100$ and if each run has one set bit in its tail, then the BBC compression uses the maximum amount of storage.*

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, B and C be three runs of a sparse bitmap and suppose run B comes right before run C. To start with, we assume all of them are type 4 runs with 0-fills followed by special tail bytes. Each of such runs will take at least two bytes; one for the header and one for the counter. 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, 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.

Runs B and C will merge to form a larger fill. Let f_B and f_C denote the fill lengths of B and C. 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 the 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. Next, we examine the special case where the combined fill takes more bytes to represent.

In some cases, the combined fill needs one more byte than the larger one of the two old counters, but never more than one byte. 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 run B and run C are one-byte long, but the combined fill has 132 bytes and requires a two-byte counter. Under the assumption that both B and C are type 4 runs, this should not increase the total size required. However, if either B or C has only a short fill, i.e., the run requires only one byte to represent,

[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.

then moving the set bit from B to A 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 used by run 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 have a long fill, This increases the total storage required. Therefore, the sparse bitmap that has the maximum BBC compressed size must not have any short fills. ∎

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

Let f be 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 $\underline{f} = 128^{m-1} + \delta$. 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 size 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 $\gamma \equiv (\overline{f} - f)/(f - \underline{f})$ 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

$$\frac{\gamma(1 + m) + 2 + m}{\gamma + 1}.$$

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

For ease of estimation, let us assume the index is not segmented. Thus the number of bitmaps b is the number of distinct values c. Given that there are c bitmaps, 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 from 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, then when $c > 100$, the maximum number of bytes in a BBC compressed bitmap index is approximately*

$$(N + c)\left(\frac{\gamma(1 + m) + 2 + m}{\gamma + 1}\right),$$

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 $(N + c)(1 + m)$.

Proof. To prove this theorem, we observe that each fill constructed in the above discussion has the minimum length to use a given size multi-byte counter. Taking a byte away from any run will decrease the number of bytes used to store the run, adding a byte to any fill will not make it use more space. Therefore the bitmaps with the above bit pattern must take the largest possible number of bytes to represent with the BBC compression. ∎

To verify the accuracy of the above formula, we show the sizes of actual bitmap indices against the maximum values given by Theorems 2 and 4. The results are shown in Figures 1 and 2, where Figure 1 shows the measured sizes on some synthetic data and Figure 2 shows the measured 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 one the formula given in Theorem 4. These predicted maximum sizes are achieved with bitmap indices on uniform random attributes. Most other measured sizes are much smaller than the predicted maximum sizes, in other word, indices on attributes with non-uniform distributions or with some skewness are indeed smaller than the predicted maximum values.

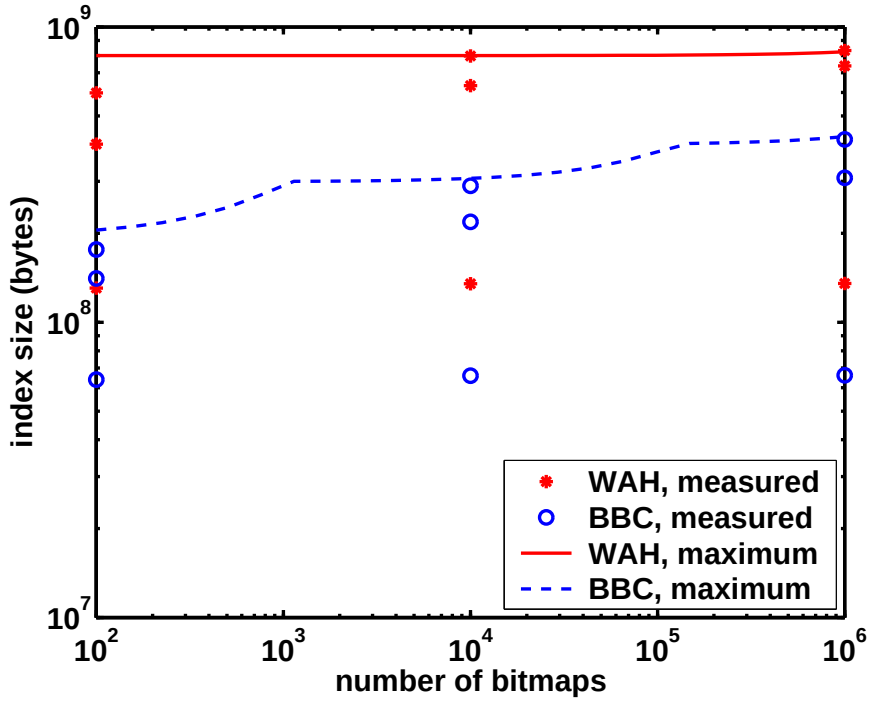

Figure 1: Sizes of compressed bitmap indices on a synthetic dataset ($N = 10^8$).

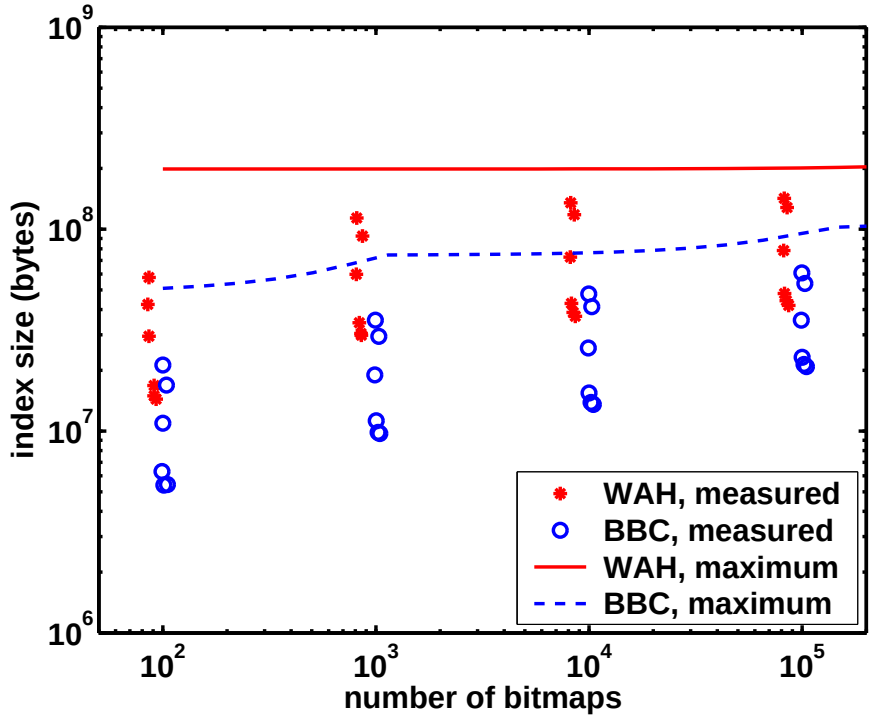

Figure 2: Sizes of compressed bitmap indices on a combustion dataset ($N = 2.5 \times 10^7$).

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

In both WAH and BBC, compressing and decompressing of a run can be done independently of all other runs. Because of this, it is easy to perform bitwise logical operations directly on the compressed data. The procedure generally consists of a loop over all words (or runs) of the compressed operands. In each iteration through the loop, the operations can be broken down into three steps, (1) decode the content of the next compressed word (or run) from each operand, (2) determine the output from the current contents of the two operands, and (3) append the new bits generated to the result. Next we show that each of these steps have an upper bound in time that is independent of the content of the bitmaps.

The first step takes a fixed number of machine level instructions. With WAH compression, it needs one test to determine whether the next word is a literal or a fill word. We can make this a simple test if we use the most significant bit of a word to distinguish a literal from a fill word. In case of a fill, it needs to extract the bit value of the fill and the length of the fill. These operations require a constant amount of time. The same operations for a BBC compressed run is more complex, but since the maximum number of bytes in the multi-byte counter is at most five (for fills less than 2^{35} bytes long), the maximum amount of time required for decoding a BBC run is also bounded by a constant.

The second step is to determine bit values to be generated given the contents of the two operands. For literal bytes or words, the machine supported bitwise logical operations can be used, which typically take one clock cycle each. For fills, the rules for determining the output bits are very simple. For a mixture of fill and literal bits, depending the logical operation being performed, the result can also be easily determined. For example, for bitwise AND operation, if the fill is a 1-fill, the result is simply the other operand. In short, these operations can be carried out within a maximum time that is independent of the bitmaps.

The last step is to append the new bits into the result bitmap. For BBC, this requires one to examine the last run, and for WAH, this requires one to examine the last word. In each case, there is only a small number of machine level instructions required to complete the append process. For example, in case of WAH, to append a literal word to an existing compressed bitmap, one simply appends the literal word to the list of words in the WAH compressed bitmap. To append a fill, one needs to examine if the last word in the current compressed bitmap represent a fill of the same type. If yes, one increases the fill length by the length of the incoming fill. Else, one starts a new fill word to represent the incoming fill.

Overall, each iteration of the bitwise logical operation on compressed bitmaps takes a fixed amount of time, the maximum number of iterations is proportional to the total size of two input bitmaps, and the worst case total time is proportional to the total size of the two input bitmaps. This is optimal because one has to examine both operands of an arbitrary logical operation to determine the results.

In the worst case, the result of a bitwise logical operation has as many bytes as the total bytes of two input operands. We can prove this by examining the details of the algorithms and counting how many runs are generated in the worst case. To save space, we offer the following arguments instead of a rigorous proof.

When both input bitmaps are sparse, each run contains only one set bit. Because these bits are far apart, the result of the operations will also have a single set bit in each run. In this worst case, the result would also have the same number of runs as the total number of runs in the two operands of the operation. With WAH compression, the size of a bitmap is directly proportional to the number of runs, therefore the size of the result would be the same as the total size of the two operands. With BBC compression, the average number of bits in a run of the result is smaller than those of the input bitmaps, therefore the number of bytes required by the multi-byte counters for lengths might use less bytes. However, in the worst case, the number of bytes required by the counters would be the same in all three bitmaps. Since each run takes the same number of bytes to represent it, the size of the result must equal the total size of the two input bitmaps.

We have measured the result size of a large number of bitwise OR operations. The results are displayed in Figure 3. 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 very sparse bitmaps, i.e., those with very 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

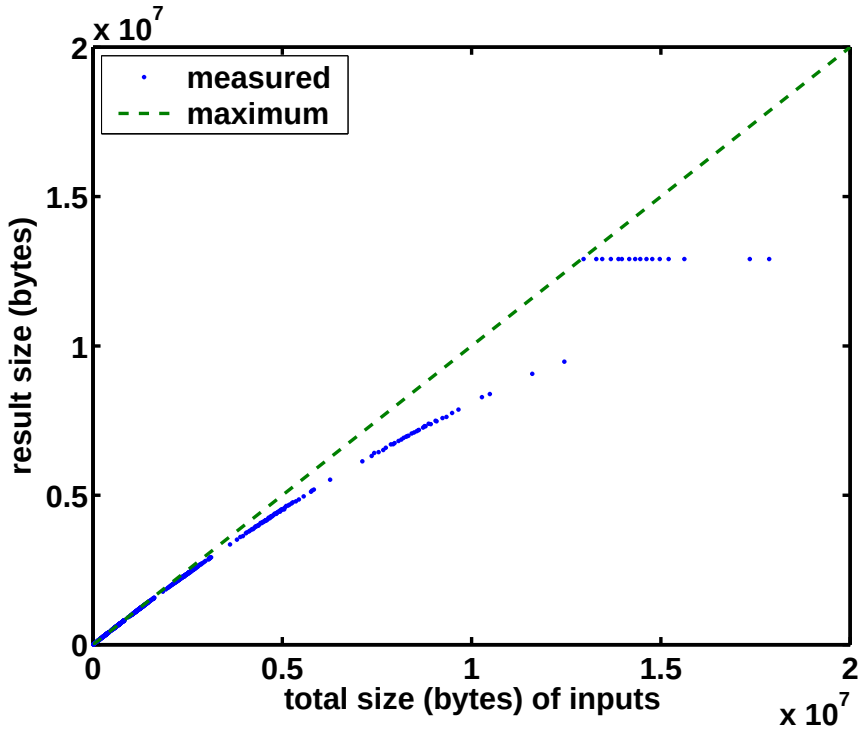

Figure 3: The result size of a logical operation plotted against the total size of two operands ($N = 10^8$).

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 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.

We have shown 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 Bitwise logical OR on many bitmaps

For a high cardinality attribute, the basic bitmap index contains many bitmaps. To answer a query such as "find all records with value less than 100" may require one to OR a large number of bitmaps. These operations might take a long time. We propose that the compressed bitmap index is still efficient. One evidence for 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 will be involved. If more than a half of the bitmaps are involved, it is easy to use the remaining bitmaps to compute the complement of the solution. Because of this, we never need to access more than a half of the compressed bitmaps, i.e., no more than N words. In many data warehousing applications, the projection index[4] 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 always accesses all N words, but using a compressed bitmap index one accesses N words only in the worst cases.

In addition to having relatively small sizes, logical OR operations on these bitmaps can be performed efficiently as well. Next we discuss five different strategies to perform OR operations on many compressed bitmaps. For convenience of discussion, let us call the bitmaps involved b_1, b_2, ..., b_k, and denote their sizes in bytes as s_1, s_2, ..., s_k. The first option is a simple application of the binary logical OR operation, which can be expressed as a simple `for` loop, where the variable r denotes the result and the operator $|=$ denotes

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

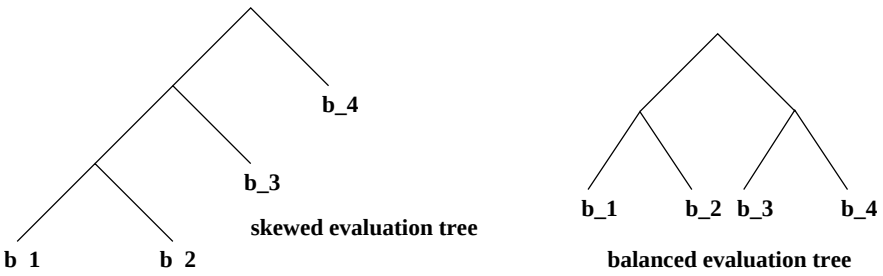

Figure 4: Two evaluation trees. A balanced tree is shorter and reduces the evaluation cost.

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` $r \mid= b_i$.

To see the worst case behavior of this approach, we assume the result of any bitwise OR operation has the maximum possible size, i.e., it has as many bytes as the total bytes of the two input bitmaps. 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 the $k-1$ logical operations is

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

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 for the operation. 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 of the operations, 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 expression for 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

$$t_3 = C_c s k \log_2(k). \tag{2}$$

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 bitmaps including the input bitmaps and the intermediate results. The priority queue puts the smallest bitmap on the top of the queue. Every binary OR operation is then performed on two bitmaps from the top of the priority queue. This ensures that the cheapest operations are performed first. It effectively implements the balanced evaluation tree without explicitly maintaining a balanced binary tree. In the following tests, we refer to this as option 3.

Both option 2 and 3 require to 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 input bitmap size may be required near the end of the process. If we free the input bitmaps immediately after they are used, this factor goes down from three to two. This amount of space is required to store the last two intermediate results and the final result, not depending on how the input bitmaps are accessed.

All previous approaches use compressed bitmaps as the result of bitwise logical operations, which requires bitmaps to be generated and destroyed for each new intermediate result. The cost of which may become a

option	description	memory requirement		time complexity
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 5: 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.

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 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 do not use separate data structure to store uncompressed bitmaps. Instead, we use the same data structure for both compressed and uncompressed bitmaps. If we are using WAH for compression, the uncompressed bitmap is also stored in the data structure for WAH compressed bitmaps. We pay a small percentage of storage overhead for doing this, however this allows use to operate efficiently between uncompressed and compressed bitmaps. When the left-hand size 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 implementation is only possible when the left-hand side of operator $|=$ is uncompressed, if the left-hand side is compressed, new storage has to be allocated for the new bitmap generated from the logical 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 should be proportional to the size of the compressed operand. 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 large number of bitmaps to be ORed, each bitmap must be very sparse because the total number of 1s must be equal to the number of tuples. 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 bitmap. 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 the others, when a large number bitmaps are 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 decompress a 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 5 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. Next we examine their relative performance and determine a way to optimally combine them to always achieve the best performance.

The optimality of a searching algorithm is measured in terms of the size of the search result. An optimal search algorithm should have a complexity that is linear in the number of hits. Let h denote the number of hits, the optimal complexity is $\mathcal{O}(h)$. Using a bitmap index, the number of hits is the number of 1s in the result bitmap. Using one of the first three options, in the worst case, the size of the result bitmap and the total size of the bitmaps involved in answering the query are both proportional to the number of hits. Therefore, the space complexity of a compressed bitmap index is optimal using these options. The time complexities of these options are not optimal, but the time complexity of option 3 is fairly close to optimal. Because the options 4 and 5 use an uncompressed bitmap to hold the intermediate result, they use $\mathcal{O}(N)$ space which is not optimal. However, their time complexities should be considered optimal because they are proportional to the total size of the bitmaps involved, which is proportional to the number of hits ($sk \propto h$).

6 Performance Measurements and Optimization

For measuring the performance of the five different options outlined in the previous section, we tested a large number of 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 bitmaps much more compressible. In Figure 2, we see that most of the observed sizes are much lower than the expected maximum sizes.

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 nearly 40 MB/s. To scan 100 million records of a projection of one attribute, 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 values with some following a uniform distribution and other following different Zipf distributions. Their sizes are shown in Figure 1. 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 6 and 7. Each point in the plots shows one 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 5. 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 are 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 6 is $t = 0.05 + 1.1 \times 10^{-8} S$, and the same trend line in Figure 7 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 8 shows the best options for WAH compressed bitmaps and Figure 9 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, for a small number

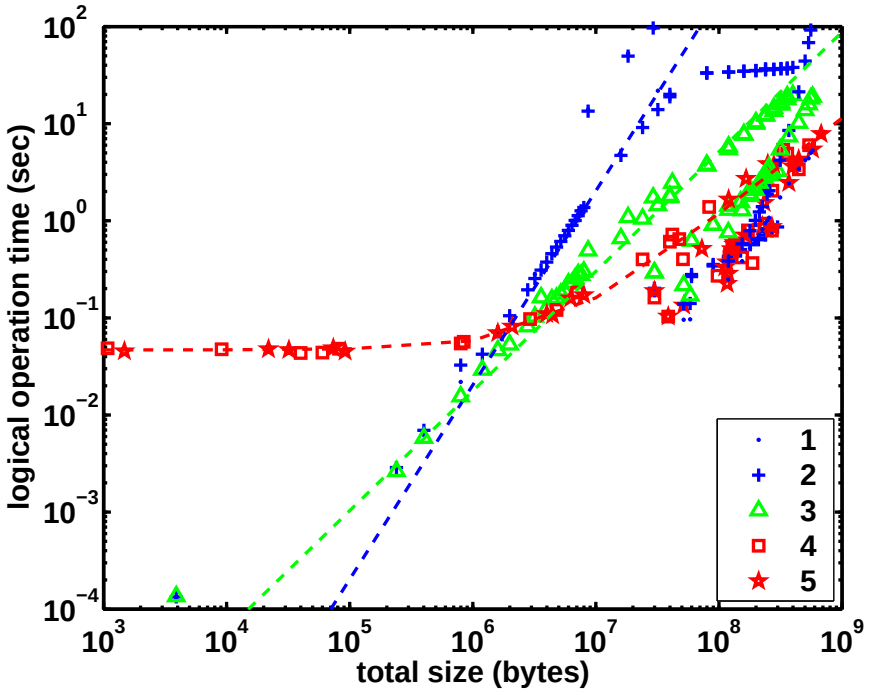

Figure 6: Time to OR many bitmaps compressed with WAH. Dashed trend lines are defined by formulas given in Figure 5.

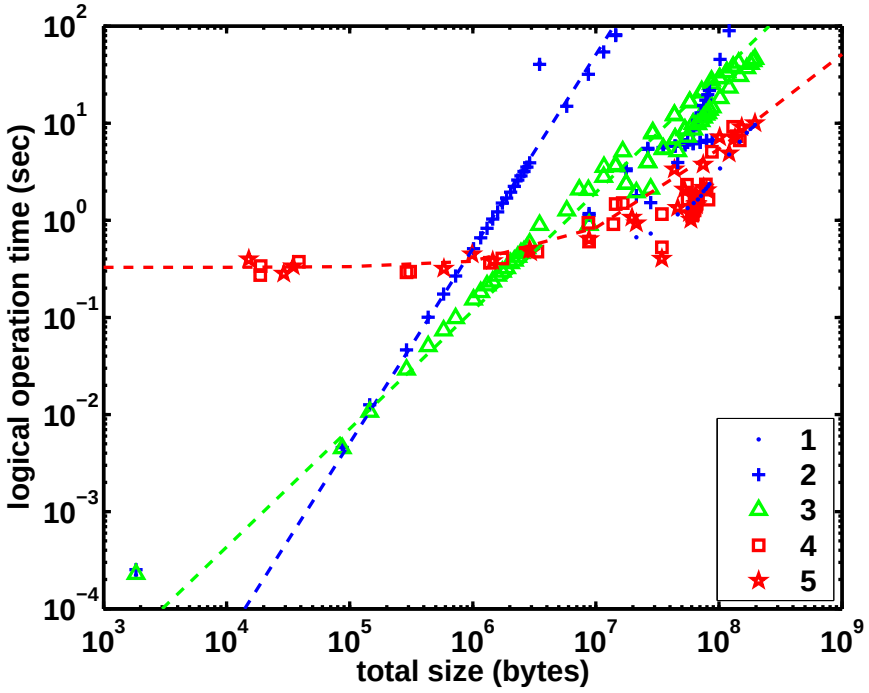

Figure 7: Time to OR many bitmaps compressed with BBC. Dashed trend lines are defined by formulas given in Figure 5.

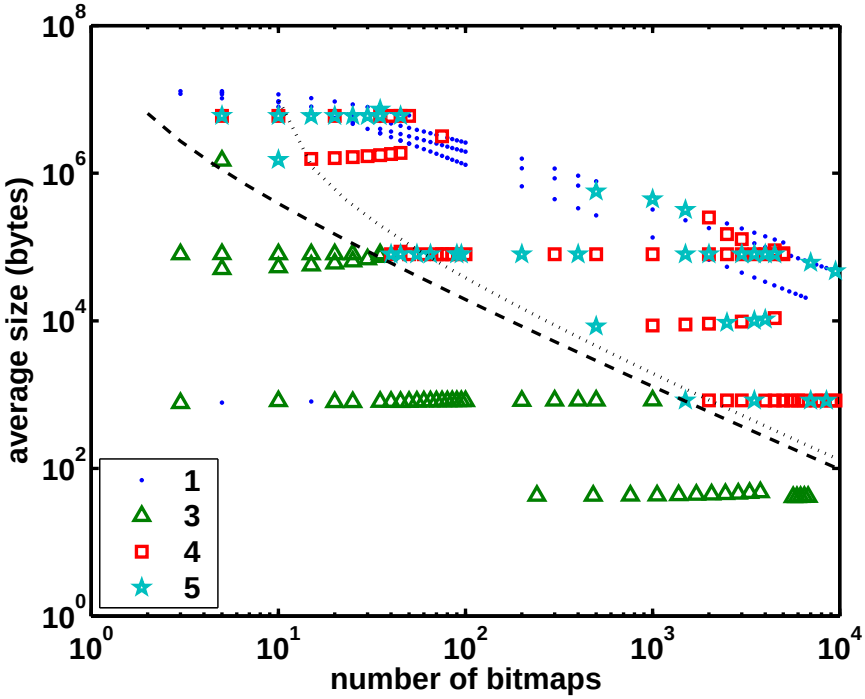

Figure 8: The best option to perform bitwise OR on WAH compressed bitmaps.

of bitmaps, where option 1 is the best. We have examined these cases and found the number of bitmaps to be very small and the performance differences between option 1 and option 3 are 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 results produced from operating on these 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. We have also identified that 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 8 and 9 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 the 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 8. 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.

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

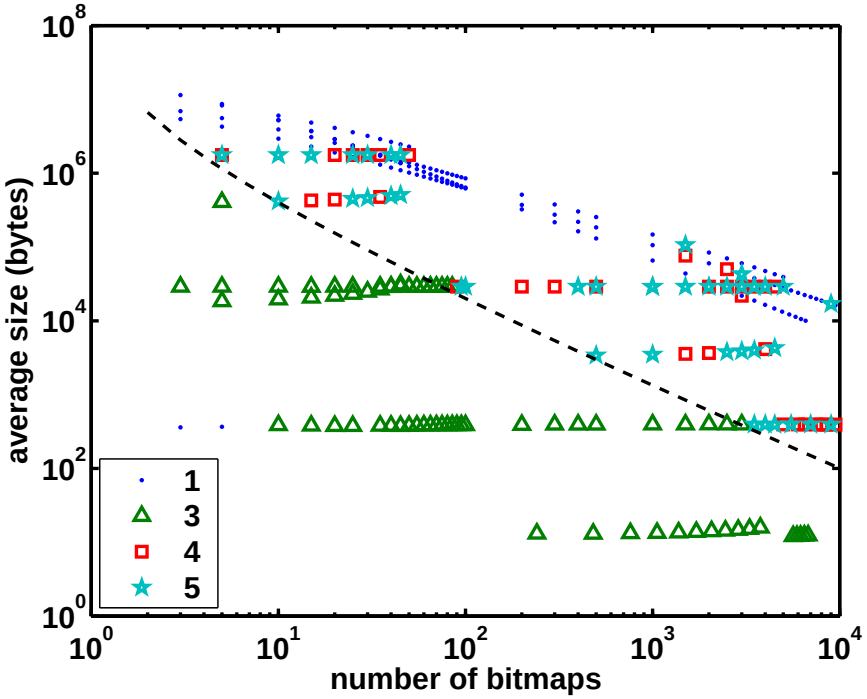

Figure 9: The best option to perform bitwise OR on BBC compressed bitmaps.

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 are 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 8 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.

From our earlier tests[18], we have found that when performing a bitwise logical operation on two bitmaps, 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. In other word, option 4 is more appropriate. With two bitmaps, the dividing line for using 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 10 plots the time measurements against the number of hits using the random dataset. In this figure, the total time refers to the total query processing time, including the time to operate 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 11 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.

Figure 12 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 worst than the projection index, but the BBC compressed indices may take more time.

The relative performance difference between WAH and BBC compressed bitmap indices in these tests is

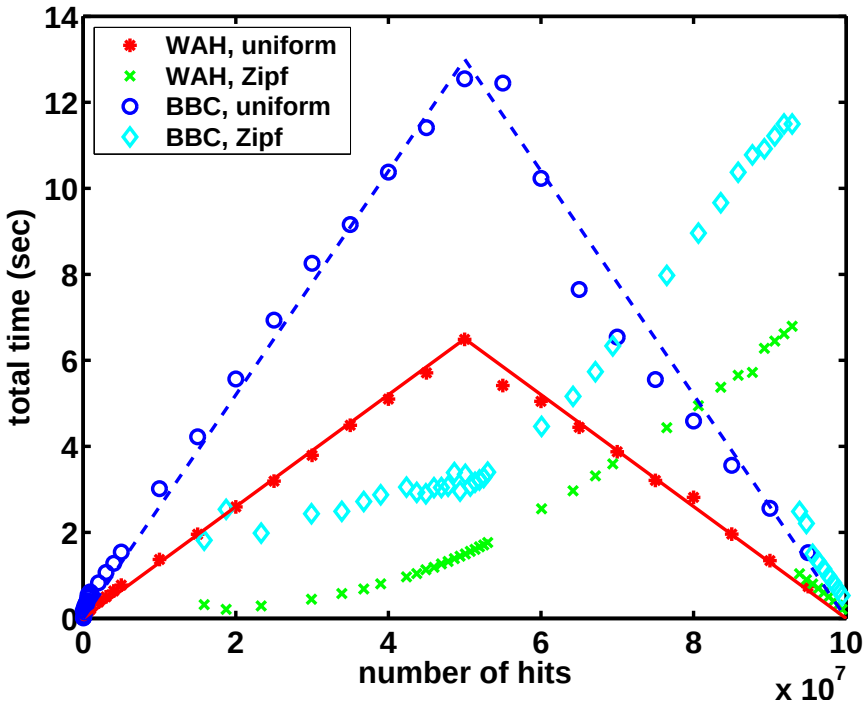

Figure 10: The total query processing time plotted against the number of hits for two type of random attributes, uniform and Zipf($1/x$).

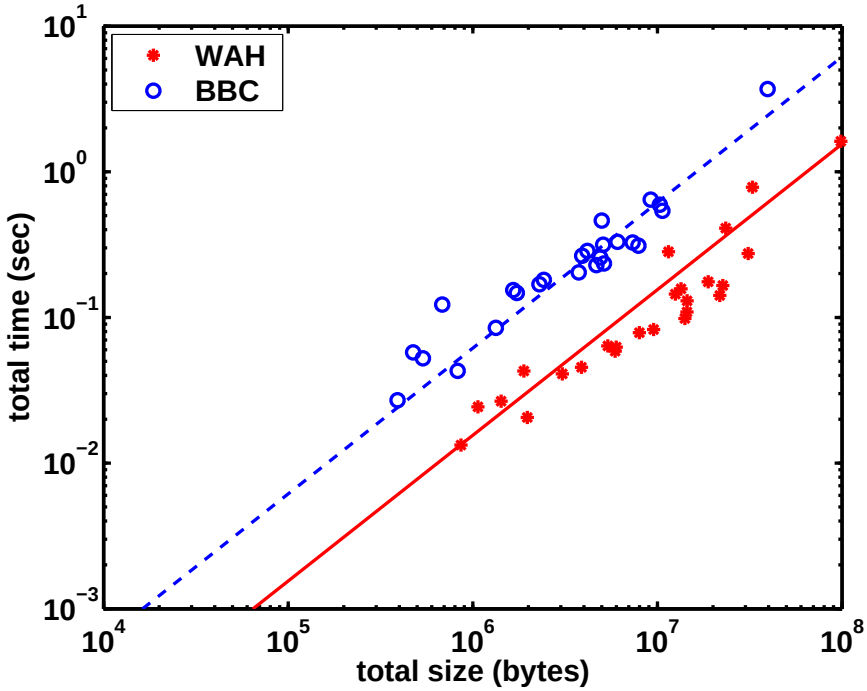

Figure 11: The total query processing time plotted against the total size of bitmaps used for the combustion dataset.

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

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. To demonstrate their effectiveness for high cardinality attributes, we analyze the space and time complexities of WAH and BBC compressed bitmap indices. 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 show an in-place algorithm to OR many bitmaps in time complexity that is linear in the total size of the bitmaps involved. In the worst case, the total size of the bitmaps involved is proportional to the number of hits. This means that using compressed bitmap to search on one attribute is optimal, and this optimality is confirmed with timing results from a set of random data and a set of real application data.

In these sets of tests, WAH compressed bitmap indices are about twice as fast as BBC compressed indices. On average, both indices can perform search operations much faster than 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 more time.

In the future, we plan to implement a robust version of our compressed bitmap indexing software based on WAH compression. The software should include segmented indices mentioned earlier. In addition, we intend to decide whether or not to use the complement of a search condition based on the size of the bitmaps involved rather than the number of bitmaps as in the current prototype implementation. Since the sizes of the bitmap indices for the attributes with Zipf distributions are smaller than those of uniform random attributes, the queries involving the attributes with Zipf distributions should take less time to answer than those involving uniform random attributes. In Figure 10, we see that this is mostly true, however, there are exceptions because our prototype bitmap index program decides whether or not to use complement based on the number of bitmaps. In a robust implementation, we should decide this based on the total size of the bitmaps involved.

Acknowledgments

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.

References

[1] Sihem Amer-Yahia and Theodore 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] Timothy C. Bell, Ian H. Witten, and John 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] Kurt 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] Kesheng Wu, Wendy Koegler, Jacqueline Chen, and Arie Shoshani. Using bitmap index for interactive exploration of large datasets. In *SSDBM 2003*, pages 65–74, 2003.

[16] Kesheng Wu, Ekow J. Otoo, and Arie 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] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Compressing bitmap indexes for faster search operations. In *SSDBM'02*, pages 99–108, 2002. LBNL-49627.

[18] Kesheng Wu, Ekow J. Otoo, Arie Shoshani, and Henrik 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] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.