Performance Measurements of Compressed Bitmap Indices

Theodore Johnson
AT&T Labs – Research
johnsont@research.att.com

Abstract

Bitmap indices are commonly used by DBMS's to accelerate decision support queries. A bitmap index is a collection of bitmaps in which each bit is mapped to a record ID (RID). A bit in a bitmap is set if the corresponding RID has property P (i.e., the RID represents a customer that lives in New York), and is reset otherwise. A significant advantage of bitmap indices is that complex logical selection operations can be performed very quickly, by performing bit-wise AND, OR, and NOT operations. Bitmap are also compact representations of densely populated sets. By using bitmap compression techniques, they are also compact representations of sparsely populated sets.

In spite of the great interest in bitmap indices, little has been published about the comparative performance of bitmap compression algorithms (i.e., compression ratios and times for Boolean operations) in a DBMS environment. We have implemented the three main bitmap compression techniques (LZ compression, variable bit-length codes, and variable byte-length codes) and built a generic bitmap index from them. We have tested each of these compression techniques (and their variants) for their compression ratio on a wide variety of synthetic and actual bitmap indices. Because bitmap indices are valuable for complex selection conditions, we evaluate four methods for performing a Boolean operation between compressed bitmaps, including methods that use compressed or partially uncompressed bitmaps directly.

Our results show that the best bitmap index compression technique and the best Boolean operation algorithms strongly depend on the bitmaps being compressed or operated on and the operations being performed. These results are a step towards understanding the space-time tradeoff in *adaptive compressed bitmap indices*, developing a bitmap index design methodology for compressed bitmaps, and optimizing Boolean expression evaluation on compressed bitmaps.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

1 Introduction

A *bitmap index* is a bit string in which each bit is mapped to a record ID (RID). A bit in the bitmap index is set if the corresponding RID has property P (i.e., the RID represents a customer that lives in New York), and is reset otherwise. One advantage of bitmap indices is that complex selection predicates can be computed very quickly, by performing bit-wise AND, OR, and NOT operations on the bitmap indices. This property of bitmap indices has led to considerable interest in their use in Decision Support Systems (DSS).

In a recent paper, O'Neil and Quass [10] provide an excellent discussion of the architecture and use of bitmap indices. Typically, a bitmap index is created for each unique value of an indexed attribute. Each bitmap is broken into fixed size fragments and stored in an index structure. O'Neil and Graefe [9] show that bitmap indices can be used as join indices for evaluating complex DSS queries on star schemas. O'Neil and Quass [10] point out that bitmap indices not only accelerate the evaluation of complex Boolean expressions, but can also be used to answer some aggregate queries directly. Several DBMS vendors have incorporated bitmap index technology into their products [12, 11].

A problem with using uncompressed (*verbatim*) bitmap indices is their high storage costs and potentially high query costs when the indexed attribute has

a high cardinality. Recent research has investigated methods for organizing bitmap indices to solve these problems. O'Neil and Quass [10] proposed *bit-slice* indices, in which a bitmap is created for each binary digit of the range of values of the indexed attribute. Chan and Ioannidis [3] propose *attribute value decomposition*, which generalizes the bit-slice index to use multiple radixes. They present an improved algorithm for making range queries on attribute-value indices, and for optimizing the design of attribute value decomposition indices. Wu and Buchmann [13] propose methods for improving the performance of hierarchy range queries on bit-slice indices.

An alternative method for dealing with the problem of using bitmap indices on high-cardinality attributes is to compress the bitmaps. For example, in a secondary index B-tree each unique key value might be shared by many records in a relation. Oracle uses compressed bitmaps (BBC encoded) to represent these sets of records [11]. A considerable body of work has been devoted to the study of bitmap index compression (see [7]). The use of bitmap compression has many potential performance advantages. Less disk space is required to store the indices, the indices can be read from disk into memory faster, and more indices can be cached in memory. However, the use of bitmap compression can introduce some problems. Performing Boolean operation requires the decompression or interpretation of the compressed bitmap, and this overhead might outweigh any savings in disk space or bitmap loading time. In addition, storing the bitmap in compressed form can make updates more expensive.

Using compression introduces many complications into bitmap index design, but much of the existing bitmap index design research [13, 3] applies to uncompressed bitmaps only. Some of the issues include:

- The compressibility of the bitmap depends strongly on the bit patterns in the bitmap.
- The compressibility of a bitmap depends on the bitmap compression algorithm. Which of the compression algorithms achieves the highest compression depends strongly on the bit patterns in the bitmap.
- There are several options for performing Boolean operations between compressed bitmaps. The default method is to uncompress the both bitmaps, then perform bitwise Boolean operations one word at a time. However other algorithms are possible (including direct operations on the compressed bitmaps) and they might be considerably faster.

Clearly, the best index design depends strongly on the nature of the data being indexed and the types of queries being performed. For bitmap indices on low cardinality attributes, compression might be effective on some but not all of the component bitmaps. For high cardinality attributes, an attribute value decomposed bitmap index [3] is likely to require more storage space than a compressed but non-decomposed bitmap index. However, such an index will evaluate range queries faster than a regular but compressed index if the query mix contains many large range queries. In some cases, it might be appropriate to use the techniques discussed in [13, 3] and compress the resulting bitmaps. But in this case an accurate model of bitmap compression ratios and Boolean operation costs should be used in the design.

A principled design of bitmap indices that makes use of compression requires an understanding of the performance of bitmap compression algorithms and their interaction with evaluating Boolean expressions. In this paper, we present a performance measurement study of several aspects of bitmap index compression, making the following contributions:

- **Evaluate bitmap compression algorithms in a DBMS setting.** We evaluate three representative approaches to bitmap compression: LZ compression using the widely available `zlib` library, variable bit length encoding using the ExpGol algorithm, and variable byte length encoding using the BBC codes. We evaluate these algorithms for their compression ratios on a wide variety of synthetic and actual bitmaps with widely varied selectivities and degrees of clusteredness.
- **Evaluate algorithms for Boolean operations on compressed bitmaps.** In a DBMS setting, one typically uses bitmap indices to evaluate complex selection predicates. For example, "New York, New Jersey, or Connecticut residents who are married, have three to five children, own a house, and work in a different state than their residence". The time to evaluate this expression depends on the time to perform all of the necessary Boolean operations. One option for performing the operations is to decompress each of the bitmaps involved, and then evaluate the expression. However, other algorithms for performing operations between compressed bitmaps are possible, including algorithms that operate directly on the compressed bitmaps. We evaluate the performance of four algorithms for performing Boolean operations on compressed bitmaps using synthetic bitmaps with a wide variety of selectivities and degrees of clusteredness.

The results that we present are necessary for a variety of follow-up research activities, including:

- Developing "adaptive compressed bitmaps" that choose the best compression algorithms for a given bitmap (based on compression ratios, time to perform Boolean operations, or both).

- Developing Boolean expression evaluation optimizers that rearrange the evaluation tree and choose algorithms for performing Boolean operations in order to minimize the total expression evaluation time.

- Developing a theory of optimal compressed bitmap indices similar to that described in [13, 3], but accounting for compression ratios and Boolean operation execution times.

The paper proceeds as follows. In Section 2, we discuss the three bitmap compression techniques evaluated in this paper. In Section 3.1, we evaluate the performance of these bitmap compression algorithms on a variety of synthetic and actual bitmap indices. In Section 3.2, we evaluate four Boolean operator evaluation algorithms using a variety of compression techniques and synthetic bitmaps. In Section 4, we discuss the impact of bitmap compression on bitmap index design. We discuss our conclusions in Section 5.

2 Bitmap Index Compression Algorithms

In our search of the literature, we have found a variety of techniques for compressing bitmap indices. The simplest method is to convert the bitmap into a *Run-Length Encoding* (RLE), which is a list of differences in the positions of successive set bits (Model 204 [8] uses a related method). If four byte integers are used to represent the run lengths, then the RLE representation uses less space than the uncompressed (verbatim) representation if fewer than 1 bit in 32 is set. The representation of run lengths as four-byte integers contains a great deal of redundancy, and typically can be compressed much further. We do not test RLE by itself as one of the encoding methods, but it is a component of the methods we do test.

We settled on three approaches to bitmap index compression as being representative of the methods that can be employed. The first method is to use the widely available LZ compression algorithm, which compresses repeated sequences of symbols. The second method compresses the RLE using variable bit length codes. The third method, the Byte-aligned Bitmap Codes (BBC) uses a variable byte-length representation of the RLE in places where the bitmap is sparse, and transcribes the bitmap where the bitmap is dense (verbatim codes).

In this section, we describe the bitmap index compression techniques used in this study. Each of the compression algorithms accepts a block of bitmap data and returns a block of compressed bitmap codes. By encoding a large bitmap one block at a time and indexing the compressed bitmap blocks, one can uncompress only those bits in a desired range.

2.1 LZ encoding

Lempel-Ziv encoding searches for long repeated strings in a body of text and replaces them with short compression codes. High-quality LZ compression software is easily available, both as the `gzip` file compression tool, and as the `zlib` data compression library [6]. Because LZ software is so readily available, the natural default choice of a compression engine is LZ even though the readily available implementation (`zlib`) is designed for text compression rather than for bitmap compression.

We implemented two variations of LZ bitmap index compression. The first method, *LZ*, compresses the verbatim bitmap, while *LZ-RLE* first converts the bitmap into a RLE representation, then compressed the RLE using LZ.

2.2 ExpGol encoding

A number of variable bit length techniques for compressing the run length encodings of bitmaps have been developed in the Information Retrieval literature. In [7], Moffat and Zobel present a unifying description of many of these coding techniques and a performance comparison of them. We chose the ExpGol code as representative of these techniques, as it is reported to have the best performance of the codes that do not rely on Huffman trees (LZ uses Huffman trees and the ExpGol algorithm has performance competitive with the highest compression algorithms discussed in [7]). The discussion of the ExpGol code we present here is based on the presentation given in [7].

A basic variable bit length representation of integers is a *gamma* code. The gamma code of integer n, $\gamma(n)$ is $\lfloor \log_2(n) \rfloor$ zero bits followed by the least significant $\lfloor \log_2(n) \rfloor + 1$ bits of the binary representation of n. Note that the truncated binary representation of n will always start with a 1. For example, $\gamma(1) = 1$, $\gamma(2) = 010$, $\gamma(3) = 011$, $\gamma(4) = 00100$, and so on. Interpreting a gamma code is done by counting the number of bits from the starting position to the first 1 bit, then copying the truncated binary representation of n to a location where it can be interpreted as an integer.

Fraenkel and Klein [5] have observed that a large class of bitmap encodings can be fit into a simple framework. Let V be a list of positive integers v_i. To encode n, we find the k such that

$$\sum_{j=1}^{k-1} v_j < n \leq \sum_{j=1}^{k} v_j$$

Let

$$d = n - \sum_{j=1}^{k-1} v_j - 1$$

To encode n, we write k in some encoding, followed by d using $\lceil \log_2(d) \rceil$ bits. For example, a γ code encodes k in unary (a string of zero bits followed by a 1 bit) using the following vector: $V = (1, 2, 4, 8, 16, \ldots)$.

Moffat and Zoebel find that the following extension to the ExpGol code gives the best compression. Let b be an integer that is representative of gap lengths in a bitmap. Then, use a gamma code to represent k and set:

$$V = (b, 2b, 4b, 8b, 16b, \ldots)$$

Moffat and Zoebel found that setting b to the median gap length, or to the geometric average of the gap lengths, to be effective. We implemented both approaches (*ExpGol median* and *ExpGol mean*, respectively).

2.3 Byte-aligned Bitmap Codes

Antoshenkov [2, 1] has proposed the use of *Byte-aligned Bitmap Codes* (BBC). The claimed advantage of BBC codes is their speed, since all operations occur locally on full bytes. The BBC encoding algorithm is a 1-pass algorithm, which permits incremental bitmap compression. Antoshenkov proposes logical operations on bitmaps that use only the compressed bitmap codes, which can be substantially faster than operating on uncompressed bitmaps. Finally Antoshenkov shows that his BBC codes achieve better compression than do gamma-delta codes (but there was no comparison to the considerably more efficient ExpGol codes).

BBC codes can be *one-sided* or *two-sided*. We first discuss one-sided codes. Every BBC code consists of two parts, a *gap* and an *ending*. The gap specifies the number of zero bytes that precede the ending. The ending can either be a *bit* (a byte with a single bit set), or it can be a *verbatim* sequence of bitmap bytes. Bit endings are used where the bitmap is sparse, while verbatim endings are used where the bitmap is dense. BBC codes use a clever packing to minimize space use; if the gap is short, a bit code is expressed in one byte. Long gaps are expressed with multi-byte codes. Two-sided codes are similar, except that the gap can be either zero-filled or one-filled.

In the course of our experiments, we found that a few simple modifications to Antoshenkov's encoding scheme resulted in a substantial improvement in space compression. All of the BBC related results in this paper use the improved BBC codes. We do not have space in this paper to discuss the improvements, but will do so in the full paper.

3 Performance

The most important aspects of compressed bitmap performance are the *compression ratio* (size of the compressed bitmap divided by the size of the uncompressed bitmap), and the time required to perform Boolean operations. Secondary performance considerations include the time to uncompress a bitmap, whether the compressed representation can be incrementally updated, storage management issues, and so on. Because of space constraints, we must defer a discussion of these issues to the full paper. However, Figures 7 and 8 illustrate the timne to uncompress.

After performing the experiments, we found that the LZ-RLE algorithm never had the best performance. To avoid cluttering our charts, we do not present LZ-RLE results. We also found that ExpGol Mean and ExpGol Median have nearly identical performance. We present results for ExpGol Mean only, and refer to them as ExpGol.

We built a generic compressed bitmap index to support our experiments. The core of the index is a list of pointers to compressed bitmap blocks, and the associated metadata. Each compressed bitmap block represents a fixed-size block of uncompressed bitmap. The minimal metadata associated with a compressed bitmap block is the length of the block. Additionally, one can store the type of compression used (to support multiple compression methods) and so on. We ran our experiments by generating a test bitmap, compressing and storing the compressed representation in the index, and then operating on the uncompressed representation. Throughout this study, we used 32 Kbyte blocks (which is the size of the compression window in zlib).

All experiments were carried out on a 225 Mhz Ultrasparc. We note that timing measurements are highly dependent on processor architecture and coding optimizations. However, we ported our code to an SGI Challenge and to an Intel Pentium Pro, and obtained nearly identical relative performance. We made efforts to optimize the ExpGol and BBC coding and decoding algorithms (by using pre-computed values, minimizing data copies, etc.) but an extensive tuning effort would probably yield faster code. We used zlib as it was provided. While small differences in execution speed are not significant, we feel that we have captured the relative performance of the algorithms well enough that large differences in execution speed are significant.

3.1 Bitmap Compression Ratio

In this section, we present the compression ratios achieved by the bitmap compression algorithms with a variety of input bitmaps. In our first experiments, we generate uniform random bitmaps as test input. A bit is set with probability p independently of all other bits in the bitmap. This model describes bitmaps that are uncorrelated with the sort attributes of the data set. We refer to the proportion of set bits in the bitmap as the *bit density*, which we represent with the symbol p. In all of the charts, the X axis is the bit density. We generated an 8 Mbyte bitmap (64 Mbits), and compressed in blocks of 32 Kbytes each. We

measured the compression ratio to be the size of the compressed bitmap divided by the size of the uncompressed bitmap.

Figure 1 shows the compression ratio of the four compression algorithms as the bit density varies between .0001 and .9999. All of the algorithms achieve a good degree of compression when p is close to zero, but only the two-sided algorithms (LZ and BBC 2s) achieve good compression when p is close to 1. the LZ coding has the best compression on dense bitmaps (roughly, $.2 \leq p \leq .98$). To minimize buffer use, we modified the ExpGol encoders to return a verbatim bitmap instead of an encoded bitmap if the encoded bitmap is larger than the verbatim bitmap. For this reason, their compression ratio is 1 when $p > .2$.

One can expect that many bitmaps are sparse, whether because the indexed attribute has a high cardinality, or because the data distribution is highly skewed. In Figure 2, we compare the compression ratios of the algorithms as p ranges from .0001 to .1. To better illustrate relative compression ratios, we present the size of the compressed bitmap as a multiple of the size of the ExpGol compressed bitmap. The ExpGol algorithms achieve significantly better compression than the other algorithms on sparse bitmaps, for example occupying one third the space of the LZ compressed bitmap when $p = .0001$. The ExpGol algorithms produce a 90% space reduction when $p = .01$, and a 99.8% space reduction when $p = .0001$.

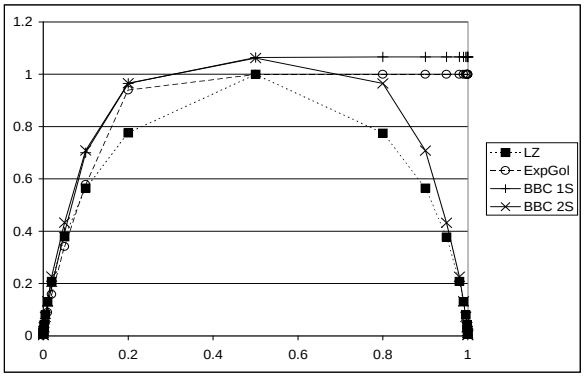

Figure 1: Bitmap compression ratio vs. bit density, uniform random.

If the indexed attribute is correlated with the sort order of the data set, then the bitmap is likely to be bursty. We model this type of burstiness with a *recursive biased distribution* (RBD) [4]. Each bit b_i in the range $i = 0 \ldots 2^l - 1$ is assigned a probability of being set, p_i, and each bit is set or reset independently of the other bits. Given a bias b, we set

$$q_i = b^{ones(i)}(1-b)^{1-ones(i)}$$

where $ones(i)$ is the number of ones in the binary representation of i. The value q_i represents the chance of receiving a ball in a ball-and-urn model. We throw R balls into the 2^l urns. We compute p_i to be the probability that urn i is non-empty.

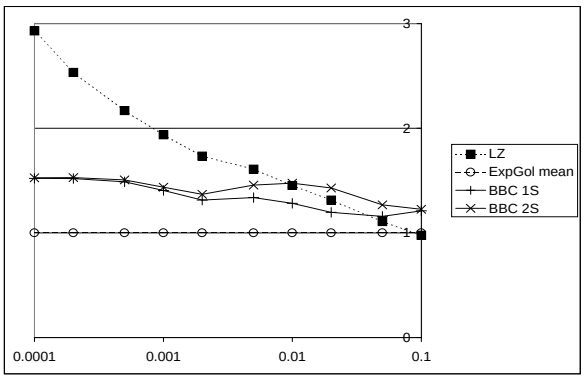

Figure 2: Compression ratio relative to ExpGol vs. bit density, unif. random.

We have found that a RBD is a good model of a clustered bitmap [4]. A bias of $b = .5$ produces uniform random bitmaps, while larger biases produce increasingly clustered bitmaps. We generated synthetic data by computing p_i for $i = 0 \ldots 2^l - 1$, repeating this pattern for the entire synthetic bitmap. Then each bit $i + k * 2^l$ is set with probability p_i. We adjusted R to adjust the bit density in the bitmap.

We present the relative compression ratio (Figure 3) for a bias of .8. The charts (and charts for other values of the bias) show that the relative performance of the algorithms has not changed significantly. The compression ratio of LZ and the BBC codes improve relative to ExpGol, with the crossover point moving from a bit density of .1 closer to a bit density of .01. The changes are more accentuated as the bias increases.

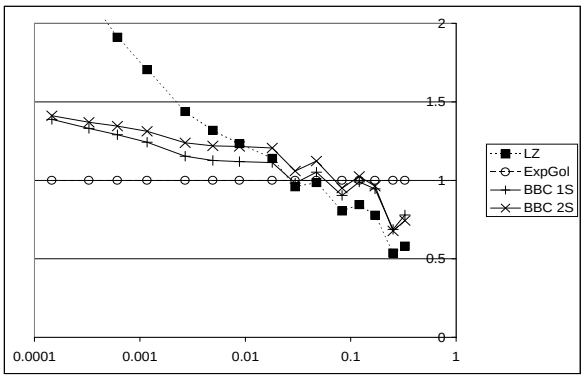

Figure 3: Relative compression ratio vs. bit density, bias=.8

While synthetic data is an excellent tool for the controlled testing of algorithm performance, we also need to evaluate the algorithms using actual bitmap data. We collected data from AT&T data warehouses and generated bitmap indices on the data. In the data set A, the attribute range is 682 values, and the most common value appears in about 28% of the attributes. In the data set B, the attribute range is 50 values, and the the most common value appears in about 13% of the tuples. In data set C, the attribute range is 11 values, and the most common value value occurs in about 80% of the tuples. The indexed attribute is correlated with the sort order of data set A, and is uncorrelated with the sort order of data sets B and C. The data sets were about .5 Mbytes in size each. We catenated the bitmaps to obtain 8 Mbyte bitmaps, for an easier comparison to the results on the synthetic data.

The compression ratio for data set A is shown in Figure 4. The compression performance is excellent, as the BBC 2S encoding of all 682 bitmaps uses 1.04 bits per tuple, while BBC 1S uses 1.5 bits per tuple, ExpGol encoding uses 1.75 bits per tuple, and LZ uses 2.25 bits per tuple. The relative performance of the algorithms is similar to that obtained with a highly biased RBD data set. The data set tends to contains runs of 1's as well as runs of 0's even in low density bitmaps. Because the BBC 2S code can represent runs of 1's succinctly, it is particularly effective at compressing this data set. The ExpGol code obtains the best compression, on the sparse data sets.

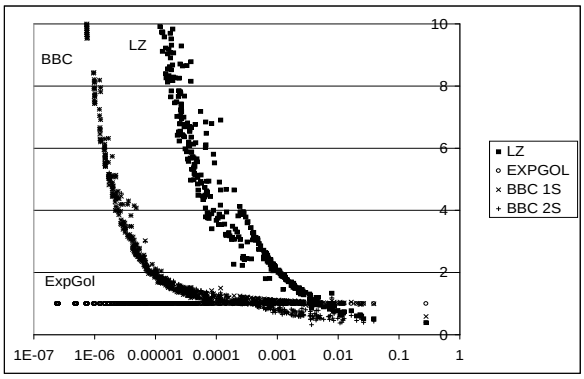

Figure 4: Relative compression ratio vs. bit density, data set A

The compression ratio for data set B is shown in Figure 5. The ExpGol algorithms require about 7.2 bits per tuple, the BBC codes require 8.5 bits per tuple, and LZ requires 8.6 bits per tuple to represent all 50 compressed bitmaps. The relative performance of the compression algorithms is similar to that obtained with uniform random synthetic data.

The compression ratio for data set C is shown in Figure 6. The ExpGol algorithms require about 2.3

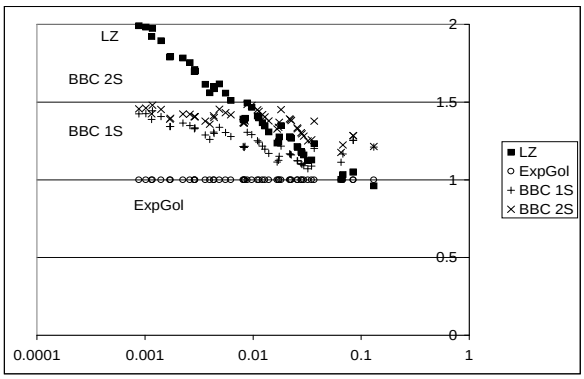

Figure 5: Relative compression ratio vs. bit density data set B

bits per tuple, the BBC codes require 2.5 bits per tuple, and LZ requires 2.3 bits per tuple to represent all 11 compressed bitmaps. The relative performance of the algorithm is similar to that obtained with uniform random synthetic data. The ExpGol algorithm generally obtains the best compression.

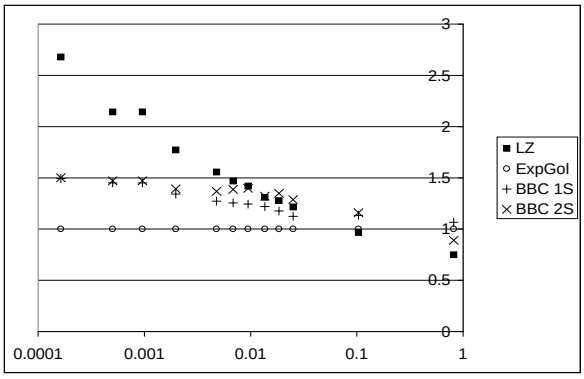

Figure 6: Relative compression ratio vs. bit density data set C

3.1.1 Summary

Existing bitmap compression algorithms can achieve very good compression ratios. As our results indicate, the best compression algorithm depends on the bitmap to be compressed. For dense bitmaps (i.e., density larger than about .1) usually the LZ algorithm will give the best compression, although BBC 2S can work better for clustered dense bitmaps.

For non-dense bitmaps, the choices are more complex. In Table 1 we list the best compression algorithm depending on whether the bitmap is uniform or clustered, sparse or non-sparse. Non-sparse means that the bit density is between .002 and .1, but these

boundaries are not precise.

sparse		non-sparse	
uniform	clustered	uniform	clustered
ExpGol	ExpGol	ExpGol	LZ, BBC 2S

Table 1: Best compression algorithms for non-dense bitmaps.

3.2 Boolean Operation Performance

Clearly, compressing bitmap indices confers many performance advantages, including reducing disk space usage and decreasing the time to load the index into memory. The primary advantage of using bitmap indices in the first place is the fast evaluation of complex selection predicates. The overhead of uncompressing the bitmaps before performing the Boolean operations can negate any advantage of using compression.

However, many Boolean operations can be performed using the compressed or partially uncompressed bitmaps. For some operations on some bitmaps, using compressed bitmaps can lead to a speed increase. In these experiments, we assume that a foundset has been partially computed, and we need to perform a Boolean operation between the foundset and a compressed bitmap to create an updated foundset. We investigate the following four algorithms for performing a Boolean operation:

Basic: We are given a verbatim foundset and a compressed bitmap. The bitmap is uncompressed, and the bitwise operations between the two bitmaps are performed. We used the largest possible word size for these operations (e.g., 64 bit words).

Inplace: The Inplace algorithm [8] operates on the foundset without materializing the bitmap. The OR operation is performed by setting bits in the foundset, while the AND operation zeros out inter-bit gaps and perform AND operations between the foundset and the bitmap bits.

Merge: We are given a foundset in RLE format, and a compressed bitmap. We partially uncompress the bitmap into RLE format. Then, it is a simple matter to merge these two lists into an output list while performing the desired Boolean operation. In our implementation, we operate on lists of sorted set bit positions instead of lists of run lengths.

Direct: The Direct algorithm takes a compressed foundset and a compressed bitmap, performs a Boolean operation, and produces a compressed foundset for output. The BBC codes best support this algorithm [2, 1]. The main idea is simple. Each BBC code expresses a gap and an ending. We scan through the two BBC code blocks keeping track of the current position in the codes. We accumulate an output gap, then an output ending. When the output code is finished (a gap occurs, or the verbatim ending is too long), we create an output BBC code word from the gap and ending description.

The actual implementation is quite complex and contains many special cases. For this reason, we implemented the Direct algorithm only for the BBC 1S code.

We implemented each of the algorithms for two Boolean operations, AND and OR, which can exhibit different performance characteristics. Other Boolean operations can be implemented in a similar fashion, and will have similar performance. For example, OR NOT (e.g. P OR NOT Q) is similar to AND, while XOR and AND NOT are similar to OR. The NOT operation will perform well only on verbatim bitmaps (or on two sided compression codes). However, many NOT operations can be combined with other operations to form operations that can be performed quickly from compressed bitmaps (e.g., the AND NOT and OR NOT operations). It is also possible to compute aggregate functions (e.g., count) on compressed or partially uncompressed bitmap representations. However, we do not address this issue here.

We note that some bitmap compression techniques do not support all of the four Boolean operation algorithms. The LZ encoding supports only the Basic algorithm, and we have implemented the Direct algorithm only for the BBC 1S algorithm.

For our experiments, we generated uniform and clustered bitmaps (where the bias $b = .8$). For each experiment, we assume that we have been given a foundset (in an appropriate form) and a compressed bitmap, and we want to perform a Boolean operation (AND, OR) on them. We measure only the time to perform the operation, not the time to generate the foundset.

In Figures 7 and 8, we show the time to perform an AND operation on an 8 Mbyte compressed bitmap using the Basic evaluation algorithm (the performance charts for the OR operation are identical). In these experiments, we repeatedly fetched a 32 Kbyte block from the foundset, uncompressed a 32 Kbyte block from the compressed bitmap, then performed the Boolean operation, until the entire bitmap was processed. Because we operate on small chunks of data, the bitmaps are cached at the time of the Boolean operation giving a very fast operation time (about .05 seconds). If we had uncompressed the entire bitmap before performing the operation, the overhead would have been significantly larger (about .5 seconds). We ran all of our experiments in this way, on the assumption that one would want to take advantage of cache locality to the greatest extent possible, especially when evaluating complex Boolean predicates.

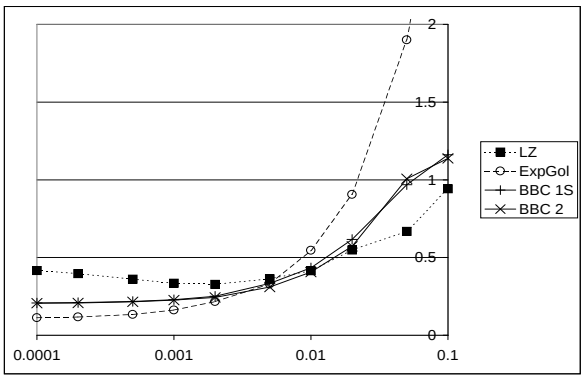

Figure 7: Basic AND operation time (secs) vs. bit density, uniform.

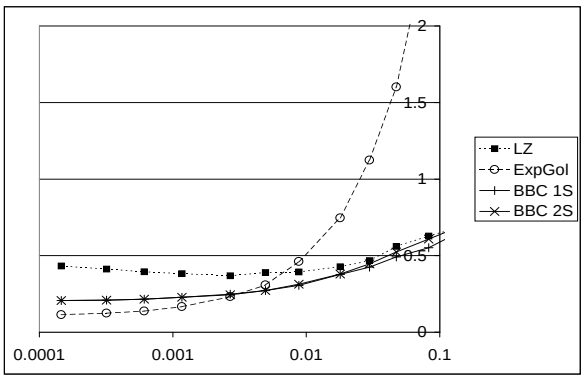

Figure 8: Basic AND operation time (secs) vs. bit density, bias=.8.

We note that these charts also represent the performance of decompressing bitmaps for each of the compression algorithms. We found that the decompression time charts for data sets A, B, and C are similar to those for the synthetic uniform data (B and C) or the biased synthetic data (data set A). To save space, we present timing results for the synthetic data only.

We next examined the performance of the Inplace algorithm, relative to that of the Basic algorithm. In Figures 9 and 10, we show the time to perform an AND operation using uniform and RBD bitmaps. In Figures 11 and 12, we show the time to perform an OR operation. These charts (and also Figures 13 through 16) show the time to perform an operation as a multiple of time for the Basic algorithm with the fastest encoding for the identical bitmap. We use this convention because we are interested in relative performance, and we can tell at a glance whether an evaluation algorithm has better performance than the Basic algorithm.

Performing an Inplace AND operation generally takes about the same amount of time as a Basic AND operation. Although we save on a memory copy, the logic for performing the operation is more complex and therefore slower. However, performing an Inplace OR operation using sparse bitmaps can be significantly faster than the Basic OR operation because only a small fraction of the output bytes must be modified. For a highly clustered bitmap compressed with a BBC code, the Inplace OR is faster than the Basic OR even for fairly dense bitmaps.

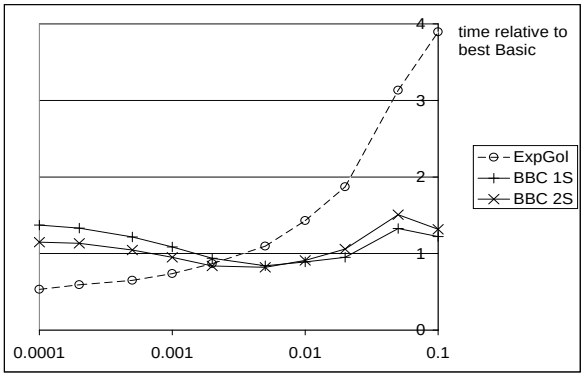

Figure 9: Inplace AND operation time (relative to best Basic) vs. bit density, uniform.

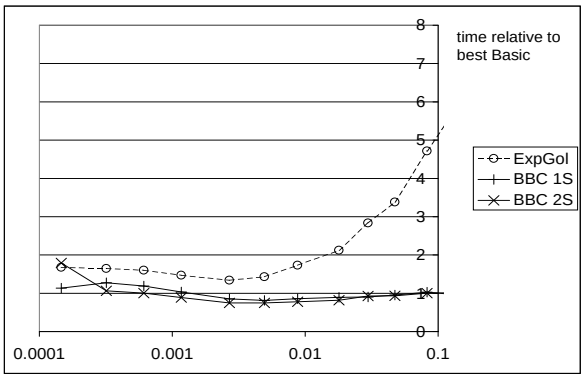

Figure 10: Inplace AND operation time (relative to best Basic) vs. bit density, bias=.8.

Finally, we test the performance of the Merge and the Direct algorithms. The foundset that is operated on has a significant impact on the operation performance (i.e., because the foundset is stored as a list of RLEs). In Figures 13 and 14 we use a uniform random bitmap with $p = .0001$ and show the time to perform an AND operation using uniform and RBD bitmaps. The OR operation is slower (because the result is larger), but the performance is similar. With

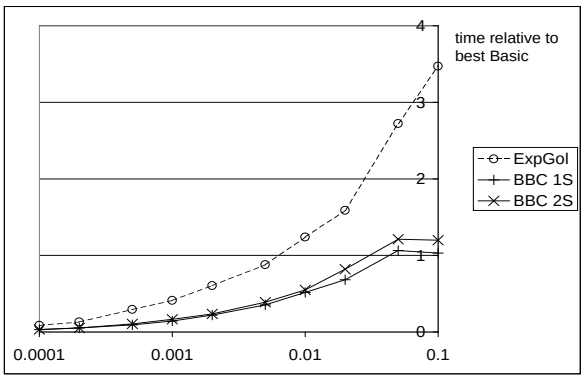

Figure 11: Inplace OR operation time (relative to best Basic) vs. bit density, uniform.

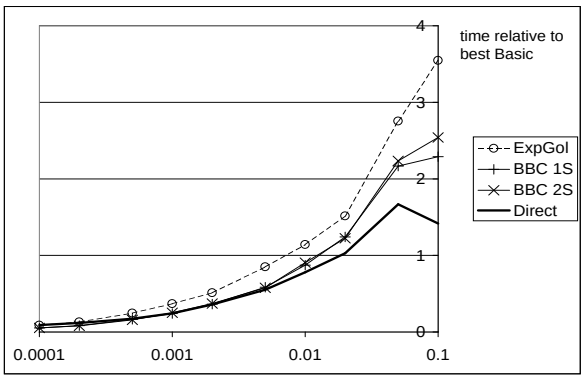

Figure 13: Merge and Direct AND operation (relative to best Basic) vs. bit density, unif. on unif. $p = .0001$.

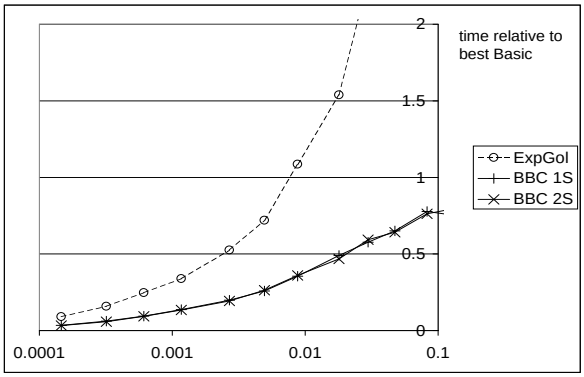

Figure 12: Inplace OR operation time (relative to best Basic) vs. bit density, bias=.8.

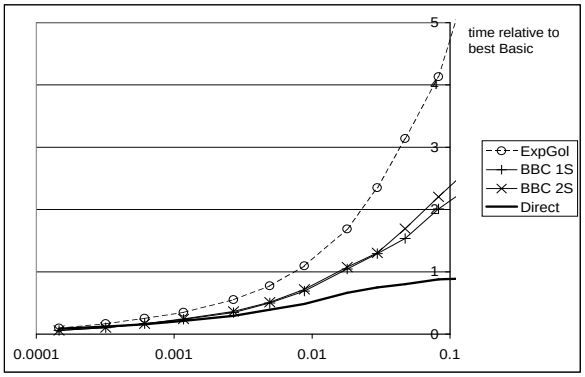

Figure 14: Merge and Direct AND operation (relative to best Basic) vs. bit density, bias=.8 on unif. $p = .0001$.

uniform bitmaps, the RLE AND operation is faster than the Basic algorithm for BBC codes and sparse bitmaps. The Direct algorithm is the fastest on clustered bitmaps, and the BBC codes are faster than the Basic algorithm even for moderately sparse bitmaps. In Figures 15 and 16, we perform the operations on a uniform random bitmap with $p = .1$. In general the operations become considerably slower when using the Merge or Direct algorithms than when using the Basic algorithm.

3.2.1 Summary

The best algorithm for performing an operation depends on the operator, the density of the existing foundset, and the density of the compressed bitmap. The best algorithm to use in each case is listed in Table 2. The computation cost of the algorithms depend on the amount of data touched and also on the complexity of the logic for performing the operation. Because the Basic algorithm is so simple, it performs surprisingly well. However, operations between sparse bitmaps using the Inplace, Merge, or Direct algorithms can occur 50 times faster than using Basic algorithm.

We note that in some cases, Boolean operations can be performed faster with compressed bitmaps than with with a verbatim bitmap. As we noted, the time to perform a Boolean operation between verbatim bitmaps is about .05 seconds in this study. The Merge algorithm is faster if the foundset and the compressed bitmap have a bit density of .001. The Inplace algorithm is faster if the foundset has a bit density of .002 or less.

4 Implications for Index Design

In Sections 3.1 and 3.2, we have seen that the performance of the alternatives for compressing bitmap indices varies considerably with the density of the

operation	foundset type	sparse foundset		non-sparse foundset	
		sparse bmp	non-sparse bmp	sparse bmp	non-sparse bmp
AND	uniform	Merge BBC, Direct	Basic	Inplace ExpGol	Basic
AND	clustered	Merge BBC, Direct	Direct	Basic	Basic
OR	uniform	Inplace BBC	Basic	Inplace BBC	Basic
OR	clustered	Inplace BBC	Basic	Inplace BBC	Basic

Table 2: Best Boolean operation evaluation algorithms.

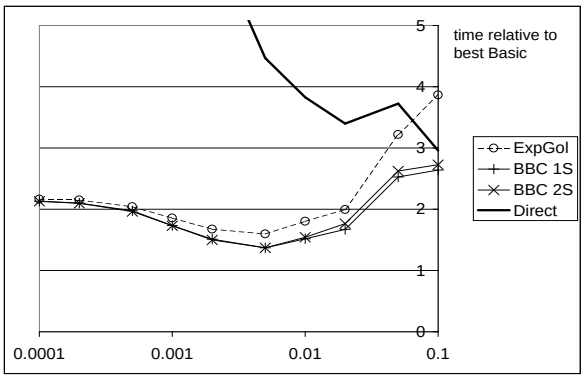

Figure 15: Merge and Direct AND operation time (relative to best Basic) vs. bit density, unif. on unif. $p = .1$.

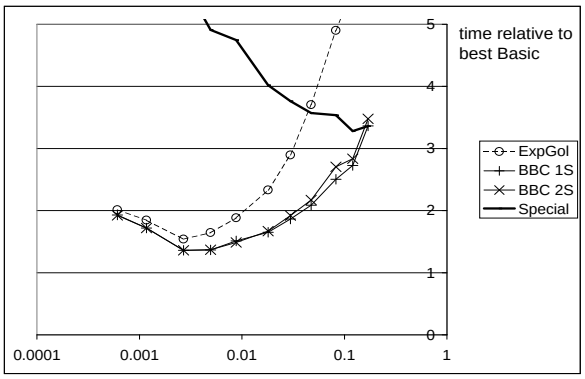

Figure 16: Merge and Direct AND operation time (relative to best Basic) vs. bit density, bias=.8 on unif. $p = .1$.

bitmap, the clusteredness of the bitmap, and the operations to be performed on the bitmaps. In this section, we discuss the implications of our measurement study on bitmap index design.

Indices on Attributes:

When bitmaps are compressed, we can observe that *zeros are cheap*. For example, AT&T data set A has a range of 682 values, but the total size of all 682 compressed bitmaps is slightly more than one bit per tuple. Data set B has a smaller range, but the bitmaps are uncorrelated with the sort position. Still, all fifty bitmaps require less than 8 bits per tuple, comparing favorably with a projection index. This property has been exploited, e.g. in [11]

An efficient bitmap compression algorithm such as ExpGol achieves a compression ratio close to the entropy encoding [7] on uniformly randomly generated bitmaps. On clustered bitmaps, the compression ratios are considerably better, as can be seen in Figures 17 and 18. These figures plot the number of compressed bitmap bits required to represent each set bit in the uncompressed bitmap, for data sets A and B respectively. We use the algorithm with the highest compression for each bitmap. We note that the flat part at the right hand side of Figure 17 is due to the clustering of set bits, which can make ones inexpensive to represent also (in the LZ and 2-sided BBC codes).

Several recent papers have addressed the issue of bitmap index design, with one goal being to minimize space usage. This has been accomplished by reducing the number of bitmaps required to represent a range of values, but increasing their density. However, if the goal of designing a bitmap index on an attribute is *only* to minimize total space use, the bit-slice indices discussed in [10, 13] and the range decomposed bitmaps discussed in [3] are unlikely to significantly reduce total space usage as compared to compressing the per-value bitmaps. An uncompressed bit-slice index (i.e., the most compact range decomposed bitmap index) on data sets A, B, and C will use 10, 6, and 4 bits per tuple, respectively, while the best compression algorithms used 1.04, 7.2, and 2.3 bits per tuple for the collection of per-value bitmaps.

However, an advantage of attribute value decomposed bitmap indices (including bit-slice indices) is the ability to express range queries by accessing only a few bitmaps. Algorithms for extracting ranges from bit-slice or range decomposed indices have been proposed in [10, 13, 3]. Chan and Ioannidis [3] show that at most $4n - 2$ bitmaps must be accessed to evaluate a two-sided range predicates, where n is the number of levels of decomposition. However, n bitmaps must be accessed for equality predicates.

The relative performance of attribute value decomposed bitmap indices versus regular but compressed

bitmap indices clearly depends on the query workload. regular indices have the advantage for equality queries, small ranges, or for selecting based on membership in moderate sized non-contiguous sets; but a disadvantage for large range or set membership queries. When the regular bitmap index is compressed, a performance comparison cannot be made only on the basis of the number of bitmaps scanned because the time to load or to perform a Boolean operation on a compressed bitmap is highly data and operation dependent, as charts 7 through 18 show. Given a sample of the bitmap indices and a sample of the workload, it is clear that an estimate of relative performance can be obtained. Unfortunately, a full treatment of this subject is beyond the scope of this paper.

Another dimension of designing bitmap indices on table attributes is the possibility of compressing attribute value decomposed and/or range encoded bitmap indices. This possibility is briefly explored in [3]. However, this subject is made difficult again because of the highly data dependent nature of the size and Boolean operation time of compressed bitmaps. The bitmap index optimization work [13, 3] assumes that each bitmap occupies the same space and requires the same amount of time to perform a Boolean operation. However this assumption clearly does not hold when bitmaps are compressed. Considering just space use, there are two problems. First, attribute value decomposition has the potential to increase space use instead of decreasing space use because of the increased number of set bits. For an example, we decomposed data set B using radixes (5, 10) (which is space-optimal for a 2-level decomposition) without changing the order of the bitmaps and using equality encoding. While the 50 original bitmaps required only 7.15 bits per tuple using the best compression, the 15 bitmaps in radix decomposition form required 9.01 bits per tuple using the best compression. A second problem is that different methods of grouping per attribute-value bitmaps into summary bitmaps will give different compression ratios. For example, by making a few experiments in which we reorder the per attribute-value bitmaps, we adjusted the space use of the attribute value decomposed bitmap from 8.31 bits per tuple to 9.12 bits per tuple.

Boolean Expression Evaluation Plans:

Current work in optimizing Boolean expression evaluation on bitmaps [3, 13] assumes that every Boolean operation on every bitmap requires the same amount of time to execute. However, our experimental results show that this is clearly not the case, and that the method by which the expression is evaluated has a significant effect on performance. For example, one should perform ORs on the sparse bitmaps using the Merge algorithm until the result bitmap becomes dense. Then one should uncompress the result bitmap and use the Inplace or Basic algorithm. It might be possible to rearrange the evaluation order, use multiple result bitmaps, and delay the changeover to the Inplace and Basic algorithms for as long as possible.

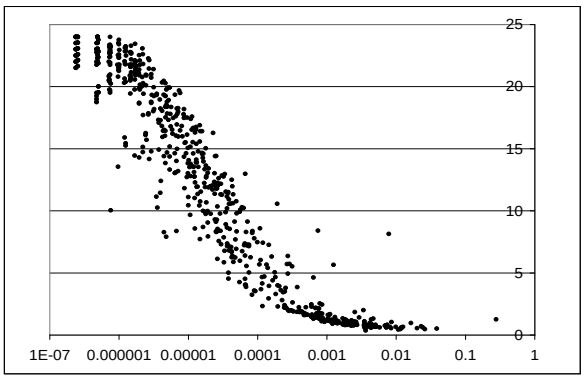

Figure 17: Bits per tuple, data set A

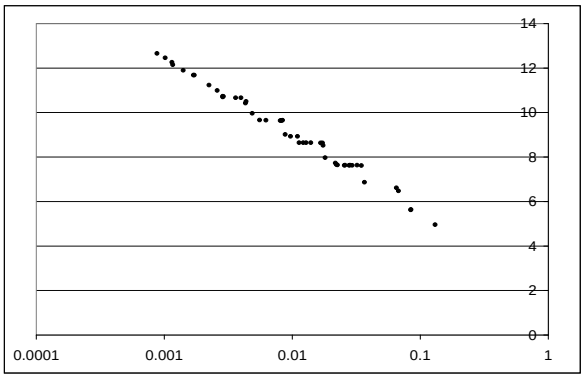

Figure 18: Bits per tuple, data set B

Adaptive Bitmap Compression:

In the previous sections, we have seen that different compression techniques work best on bitmaps with different characteristics. It is easy to design an index structure which can manage multiple compression algorithms (i.e., we built one for the experiments in this paper). Therefore, we have the flexibility to choose the compression algorithm on a bitmap-by-bitmap, or even on a block-by-block basis.

Suppose that we want to minimize compressed bitmap storage. By using the compression algorithm that minimizes space use, we can reduce the space use of the bitmap indices of data set A to .93 bits per tuple (11% space reduction), of data set B to 6.9 bits per tuple (2% space reduction), and of data set C to 2.01 bits per tuple (11% space reduction).

Another goal can be to minimize the time to evaluate a Boolean function of the bitmaps. A full treatment of this issue requires a consideration of the workload and of the typical Boolean expression evaluation plan. For the purpose of an example, we make some simplifying assumptions; that only ExpGol compression is used, that only Inplace evaluation is used, that OR operations occur three times as often as AND operations, and that the bitmaps are uniform. We further assume that the database contains 64 million tuples. For any bitmap, we want to determine whether or not we should compress it.

The time to perform a Boolean operation between the foundset and a bitmap is the sum of the time to load the bitmap from disk and the time to perform the operation once the bitmap is loaded into memory. If the disk transfer rate is 6 Mbytes/sec, then performing an operation with an uncompressed bitmap requires

$$(8 \text{ Mbytes})/(6 \text{ Mbytes/sec}) + .05 \text{ sec} = 1.38 \text{ sec}$$

Evaluating the time to perform an operation with an uncompressed bitmap is similar, but the size and the operation time depend on the bit density. By using the data from Figures 2 and 11, we can determine that any bitmap with a bit density less than .06 should be compressed, all others should be verbatim. If the disk transfer rate is 10 Mbytes/sec, the cutoff is .04.

5 Conclusions

Bitmap indexing has received new attention recently because of its application in OLAP and data warehouses. In many cases, bitmap compression can reduce space usage and possibly Boolean operation evaluation time, and can be a useful adjunct to index design. However, bitmap compression introduces new complications, and its performance has not been studied in a DBMS setting.

In this paper, we present a performance measurement study of algorithms for compressed bitmap indices. We evaluate the compression ratio of representative bitmap compression algorithms on a variety of synthetic and actual bitmap indices. Boolean expression evaluation time depends on the time to perform Boolean operations on the compressed bitmaps (i.e., rather than on the decompression time). For this reason, we evaluate four methods for performing a Boolean operation between a compressed bitmap and a (possibly compressed) foundset. We find that the various compression methods and Boolean operation evaluation algorithms have regions of best performance, which we summarize.

Based on our measurements, we find three areas of future research.

- *Adaptive compressed bitmaps* that choose the best compression scheme for each bitmap.
- Optimizing Boolean expression evaluation plans.
- Accounting for compression in bitmap index design.

Acknowledgements

We'd like to thank Pat O'Neil for his informative discussions concerning compressed bitmap indexes, and for suggesting the use of adaptive compressed bitmaps.

References

[1] G. Antoshenkov. Byte-aligned data compression. U.S. Patent number 5,363,098.

[2] G. Antoshenkov. Byte-aligned bitmap compression. Technical report, Oracle Corp., 1994.

[3] C.-Y. Chan and Y. Ioannidis. Bitmap index design and evaluation. In *SIGMOD '98*, pages 355–366, 1998.

[4] C. Faloutsos and T. Johnson. Accurate block selectivities using the recursive biased distribution. In submission, 1998.

[5] A. Fraenkel and S. Klein. Novel compression of sparse bit-strings – preliminary report. In *Combinatorial Algorithms on Words*, pages 169–183. Springer-Verlag, 1985. NATO ASI Series F.

[6] J.-L. Gailly and M. Adler. Zlib home page. http://quest.jpl.nasa.gov/zlib/.

[7] A. Moffat and J. Zobel. Parameterized compression of sparse bitmaps. In *Proc. SIGIR Conf. on Information Retrieval*, 1992.

[8] P. O'Neil Model 204 Architecture and Performance In *2nd Int. Workshop on High Performance Transactions Systems*, Springer-Verlag Lecture Notes in Computer Science 359, pages 40–59, 1987.

[9] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record*, 24:8–11, 1995.

[10] P. O'Neil and D. Quass. Improved query performance with variant indices. In *SIGMOD '97*, pages 38–49, 1997.

[11] Rdb7: Performance enhancements for 32 and 64 bit systems. http://www.oracle.com/products/servers/rdb/html/fs_vlm.html.

[12] Sybase iq indexes. In *Sybase IQ Administration Guide*, Sybase IQ Release 11.2 Collection, Chapter 5., 1997. http://sybooks.sybase.com/cgi-bin/nph-dynaweb/siq11201/iq_admin/1.toc.

[13] M.-C. Wu and A. Buchmann. Encoded bitmap indexing for data warehouses. In *Int. Conf. on Data Engineering*, pages 220–230, 1998.