Approximate Encoding for Direct Access and Query Processing over Compressed Bitmaps

Tan Apaydin
The Ohio State University
apaydin@cse.ohio-state.edu

Guadalupe Canahuate
The Ohio State University
canahuat@cse.ohio-state.edu

Hakan Ferhatosmanoglu *
The Ohio State University
hakan@cse.ohio-state.edu

Ali Şaman Tosun †
University of Texas at San Antonio
tosun@cs.utsa.edu

ABSTRACT

Bitmap indices have been widely and successfully used in scientific and commercial databases. Compression techniques based on run-length encoding are used to improve the storage performance. However, these techniques introduce significant overheads in query processing even when only a few rows are queried. We propose a new bitmap encoding scheme based on multiple hashing, where the bitmap is kept in a compressed form, and can be directly accessed without decompression. Any subset of rows and/or columns can be retrieved efficiently by reconstructing and processing only the necessary subset of the bitmap. The proposed scheme provides approximate results with a trade-off between the amount of space and the accuracy. False misses are guaranteed not to occur, and the false positive rate can be estimated and controlled. We show that query execution is significantly faster than WAH-compressed bitmaps, which have been previously shown to achieve the fastest query response times. The proposed scheme achieves accurate results (90% -100%) and improves the speed of query processing from 1 to 3 orders of magnitude compared to WAH.

1. INTRODUCTION

Bitmap indices are widely used in data warehouses and scientific applications to efficiently query large-scale data repositories. They are also successfully implemented in commercial Database Management Systems, e.g., Oracle [2, 4], IBM [32], Informix [15, 28], Sybase [12, 16]. Bitmaps have found many other applications such as visualization of scientific data [19, 38, 39]. The idea of a bitmap is basically to partition the data into bins. In general, the bit in row *i* and column *j* is a 1 if the data in row *i* falls into bin *j*. Each column is then stored as a bitmap. Query execution uses fast bit-wise operations over the bitmaps which are supported by hardware. The result of the query is a bitmap where the bits set correspond to the rows that satisfy the query, i.e., the fifth bit in the final result is 1 if the fifth record satisfies the query criteria. Bitmaps inherently involve redundant representation and are compressed for faster retrieval. Many of the proposed compression techniques [2, 18, 24, 34, 45] use run-length encoding, which replaces the continuous streams of 0's or 1's in a bitmap by a single instance of the symbol and a run count. One of the most effective compression techniques is Word Aligned-Hybrid (WAH) [43, 44, 46] which has been recently shown to provide the fastest query execution when compared with other techniques.

While space is gained by compressing the bitmap, the ability to directly locate the row identifier by the bit position in the bitmap is lost. Now the fifth bit in the compressed bitmap does not necessarily correspond to the fifth record. This is acceptable when queries do not involve the row identifier, or a range of identifiers, as part of the query, i.e. all records are considered as potential answers to the query. However, many queries are executed over a subset of data, which is usually identified by a set of constraints. For example, a typical scientific visualization query focuses on a certain region or time range. In a data warehouse, queries are specified over the dimensions (or the higher level dimensions in the hierarchy) such as date (or month, quarter, year), account (or customer, customer class, profession), or product (or product type, vendor). In many cases, queries pose multiple constraints and the result of executing one of the constraints is a list of candidate row identifiers to be further processed. In a multi-dimensional range query, the rows that do not satisfy a condition in one of the dimensions do not need to be processed in other dimensions. The performance of the current bitmap encoding schemes would suffer under all these scenarios where relevant rows are provided as part of the query. Moreover, the task of handling queries with WAH that ask for only a few rows needs extra bit operations or decompression of the bitmap.

Queries that only ask for a few rows are very common. The row number could represent time, product, or spatial position. In fact, the row number can indicate any attribute as long as the data set is physically ordered by it. For example, consider a data warehouse where the data is physically ordered by date. A query that asks for the total sales of every Monday for the last 3 months would effectively select twelve rows. Similarly, a query that asks for the sales of the last seven days is asking for seven rows. Moreover, the row number in the bitmap can represent more than one attribute. For example, we could map the x, y, and z coordinates of a data point to a single integer by using a well-known mapping function

* Partially supported by DOE grant DE-FG02-03ER25573 and NSF grant CNS-0403342.
† Partially supported by Center for Infrastructure Assurance and Security at UTSA

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '06, September 12-15, 2006, Seoul, Korea.
Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09

or a space-filling curve and physically order the points by three attributes at the same time. When users ask for a particular region, a small cube within the data space, we can map all the points in the query to their index and evaluate the query conditions over the resulting rows.

While many other approaches, including compressed bitmaps, compute the answer in $O(N)$ time, where N is the number of points in the grid, we want to be able to compute the answers in the optimal $O(c)$ time, where c is the number of points in the region queried. Our *goal* is to provide a structure that enables direct and efficient access to any subset of the bitmap, just as the decompressed bitmaps can, and which does not take as much space, i.e. it is stored in compressed form.

We propose an approximate bitmap encoding that stores the bitmaps in compressed form while maintaining efficient query processing over any subset of rows or columns. The proposed scheme inserts the set-bits of the bitmap matrix into an Approximate Bitmap (AB) through hashing. Retrieval is done by testing the bits in the AB, where any subset of the bitmap matrix can be retrieved efficiently. It is shown that there would be no false negatives, i.e., no rows that satisfy the query constraints are missed. False positive rate can be controlled by selecting the parameters of the encoding properly. Approximate query answers are tolerable in many typical applications, e.g., visualization, data warehousing. For applications requiring exact answers, false positives can be pruned in a second step in query execution. Thus, the recall is always 100% and the precision depends on the amount of resources we are willing to use. For example, a visualization tool can allow some margin of imprecision, and exact answers can be retrieved for a finer granularity query.

Contributions of this paper include the following:

1. We propose a new bitmap encoding scheme that approximates the bitmaps using multiple hashing of the set bits.
2. The proposed scheme allows efficient retrieval of any subset of rows and columns. In fact, retrieval cost is $O(c)$ where c is the cardinality of the subset.
3. The AB parameters can be specified in two ways: setting a maximum size, in which case the AB is built to achieve the best precision for the available memory; or setting a minimum precision, where the least amount of space is used to ensure the minimum precision.
4. The proposed scheme can be applied at three different levels of resolution: Building one AB for each attribute makes compression size independent of the cardinality of the attributes. Building one AB for the whole data set makes the compressed size independent of the dimensionality of the data set. Building one AB for each column makes each AB size dependent only on the number of set bits.
5. The approach presented in this paper can be combined with other structures to further improve the query execution time and the precision of the results using minimal memory.
6. The approximate nature of the proposed approach makes it a privacy preserving structure that can be used without database access to retrieve query answers.

The rest of the paper is organized as follows. Section 2 presents the related work in this area. Section 3 describes the proposed scheme.

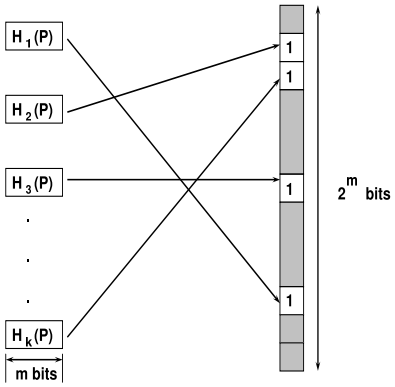


Figure 1: A Bloom Filter

Section 4 gives the theoretical analysis of the proposed scheme. Sections 5 and 6 present the experimental framework and results. We conclude in Section 7.

2. BACKGROUND

2.1 Bloom Filters

Our solution is inspired by Bloom Filters [5]. Bloom Filters are used in many applications in databases and networking including query processing [7, 11, 23, 25, 26], IP traceback [35, 36], per-flow measurements [10, 21, 22], web caching [13, 14, 40] and loop detection [41]. A survey of Bloom Filter (BF) applications is described in [6]. A BF computes k distinct independent uniform hash functions. Each hash function returns an m-bit result and this result is used as an index into a 2^m sized bit array. The array is initially set to zeros and bits are set as data items are inserted. Insertion of a data object is accomplished by computing the k hash function results and setting the corresponding bits to 1 in the BF. Retrieval can be performed by computing the k digests on the data in question and checking the indicated bit positions. If any of them is zero, the data item is not a member of the data set (since member data items would set the bits). If all the checked bits are set, the data item is stored in the data set with high probability. It is possible to have all the bits set by some other insertions. This is called a *false positive*, i.e., BF returns a result indicating the data item is in the filter but actually it is not a member of the data set. On the other hand, BFs do not cause *false negatives*. It is not possible to return a result that reports a member item as a non-member, i.e., member data items are always in the filter. Operation of a BF is given in Figure 1.

2.2 Bitmaps

Bitmap indexes were introduced in [29]. Several bitmap encoding schemes have been developed, such as equality [29], range [8], interval [9], and workload and attribute distribution oriented [20]. Several commercial database management systems use bitmaps [4, 16, 27]. Numerous performance evaluations and improvements have been performed over bitmaps [8, 37, 42, 45, 47, 48]. While the fast bitwise operations afforded by bitmaps are perhaps their biggest advantage, a limitation of bitmaps is the index size.

2.2.1 Bitmap Compression

A major disadvantage of bitmap indices is the amount of space they require. Several compression techniques have been developed in order to reduce bitmap size and at the same time maintain the advantage of fast operations [1, 3, 37, 43].

The two most popular run-length compression techniques are the Byte-aligned Bitmap Code (BBC) [3] and the Word-Aligned Hybrid (WAH) code [43]. BBC stores the compressed data in bytes while WAH stores it in words. WAH is simpler because it only has two types of words: *literal words* and *fill words*. In our implementation, it is the most significant bit that indicates the type of the word we are dealing with. Let w denote the number of bits in a word, the lower $(w-1)$ bits of a literal word contain the bit values from the bitmap. If the word is a fill, then the second most significant bit is the fill bit, and the remaining $(w-2)$ bits store the fill length. WAH imposes the word-alignment requirement on the fills. This requirement is key to ensure that logical operations only access words. Bit operations over the compressed WAH bitmap file are faster than BBC (2-20 times) [43] while BBC gives better compression ratio.

Recently, reordering has been proposed as a preprocessing step for improving the compression of bitmaps. The objective with reordering is to increase the performance of run length encoding. By reordering columns, compression ratio of large boolean matrices can be improved [17]. However, matrix reordering is NP-hard and the authors use traveling salesman heuristics to compute the new order. Reordering idea is also applied to compression of bitmap indices [31]. The authors show that tuple reordering problem is NP-complete and propose gray code ordering heuristic.

3. PROPOSED SCHEME

The goal of this work is to encode bitmaps in a way that provides fast and integrated querying over compressed bitmaps with direct access and no need to decompress the bitmap. In addition, an efficient extraction of any subset of the index is desired. Bitmaps can be considered to be a special case of boolean matrices. In general, the solution can be applied to boolean matrices. First, we describe how the proposed scheme can be applied to encode boolean matrices and then describe how it can be used to retrieve any subset of the matrix efficiently. Next, we describe the particular solution and its variations for bitmap encoding.

3.1 Encoding General Boolean Matrices

Consider the boolean matrix M in Figure 2 as an example. To compress this boolean matrix, we encode it into a binary array AB using multiple hash functions. For each bit set in the matrix, we construct a hashing string x as a function of the row and the column number. Then, k independent hash functions are applied over x and the positions pointed by the hash values are set to 1 in the binary array. The insertion algorithm is provided in Figure 3.

Collisions can happen when a hash function maps two different strings to the same value or two different hash functions map different strings to the same value.

Figure 4 presents the 32-bit AB that encodes M with $F(i,j) = concatenate(i,j)$, $k = 1$, and $H_1(x) = x \ mod \ 32$.

Definition: A query $Q = \{(r_1, c_1), (r_2, c_2), ..., (r_l, c_l)\}$ is a subset of a boolean matrix M and query result $T = \{b_1, b_2, ..., b_l\}$ is a set of bits such that $b_i = 1$ if $M(r_i, c_i) = 1$ and $b_i = 0$ if $M(r_i, c_i) = 0$.

To obtain the query result set T using AB we process each cell (r_i, c_i) specified in the query. We compute the hash string $x = F(r_i, c_i)$ and for all the k hash functions obtain the hash value

	1	2	3	4	5	6
1	0	0	0	1	0	1
2	0	1	0	0	1	0
3	0	0	0	0	0	0
4	0	0	1	0	0	1
5	1	0	0	0	0	0
6	0	0	0	0	1	0
7	0	1	0	0	0	0
8	0	0	1	0	0	1

Figure 2: Example of a Boolean Matrix M

```
insert(M)
01 for i = 1 to columns
02     for j = 1 to rows
03         if M(i,j) == 1
04             x = F(i,j)
05             for t = 1 to k
06                 b = H_t(x)
07                 set AB[b] to 1
```

Figure 3: Insertion Algorithm

$h_t = H_t(x)$. If all the bits pointed by h_t, $1 \leq t \leq k$, are 1, then the value of (r_i, c_i) is reported as 1. If a bit pointed by any of h_t values is 0, the value of (r_i, c_i) is reported as 0. The retrieval algorithm is given in Figure 5.

Let $Q_1 = \{(3,1), (3,2), (3,3), (3,4), (3,5), (3,6)\}$ be a query example over M. This query is asking for the values in the third row. The answer to this query is $T_1 = \{0,0,0,0,0,0\}$. Answering this query using the AB in Figure 4 would produce $T_1^{'} = \{0,0,1,0,0,0\}$. The third element in the answer set is a false positive set by cell $(6,5)$. Similarly, $Q_2 = \{(1,6), (2,6), (3,6), (4,6), (5,6), (6,6), (7,6), (8,6)\}$ which is asking for column 6, would produce the approximate result $T_2^{'} = (1,0,0,1,0,0,1,1)$, where the seventh element is a false positive. A second step in the query execution can prune the false positives later. The scheme is guaranteed not to produce false negatives.

We want to retrieve any subset of M efficiently. A structure that stores M by rows would retrieve a row efficiently but not a column, and vice versa. By using this encoding we can retrieve *any* subset efficiently, even a diagonal query for which other structures would need to access the entire matrix. The time complexity of the retrieval algorithm is $O(c)$, where c is the cardinality of the subset being queried. For Q_1 and Q_2 the cardinalities are 6 and 8 respectively. Using the hash values, AB can be accessed directly to approximate the value of the corresponding cell.

3.2 Approximate Bitmap (AB) Encoding

Consider the boolean matrix in Figure 6. This matrix represents the bitmaps for a table T with three Attributes A, B, C and 8 rows. Each attribute is divided into 3 bins. Columns 1, 2, and 3 correspond to Attribute A; Columns 4, 5, and 6 correspond to Attribute B; and the rest to Attribute C. Note that only one bit is set for each attribute column. This bit corresponds to the value of that attribute for the

0100 0001 0001 0101 0010 0100 1000 1000

Figure 4: AB(M) for $F(i,j) = concatenate(i,j)$, $k = 1$, and $H_1(x) = x \ mod \ 32$.

```
retrieve(Q)
01 T = ∅
02 for each (i,j) in Q
03     in = true
04     x = F(i,j)
05     for t = 1 to k
06         if AB[H_t(x)] == 0
07             in = false
08             T = T ∪ {0}
09             break loop
10     if in == true
11         T = T ∪ {1}
return(T)
```

Figure 5: Retrieval Algorithm

given row. Bitmaps are stored by columns, therefore there are nine bitmap vectors for table T.

	1	2	3	4	5	6	7	8	9
	A_1	A_2	A_3	B_1	B_2	B_3	C_1	C_2	C_3
1	1	0	0	0	0	1	0	0	1
2	0	1	0	0	1	0	0	1	0
3	0	0	1	1	0	0	1	0	0
4	0	0	1	0	0	1	0	0	1
5	1	0	0	1	0	0	1	0	0
6	1	0	0	0	1	0	1	0	0
7	0	1	0	0	1	0	0	1	0
8	0	0	1	0	0	1	0	0	1

Figure 6: Example of a bitmap table

In the case of bitmap matrices we can apply the encoding scheme at three different levels:

- One AB for the whole data set.
- One AB for each attribute.
- One AB for each column.

Construction of the AB for bitmaps is very similar to the construction for general boolean matrices. Depending on the level of encoding, we define different hash string mapping functions and different hash functions to take advantage of the intrinsic characteristics of the bitmaps.

3.2.1 Hash String Mapping Functions

The goal of the mapping function F is to map each cell to a different hash string x. Mapping different cells to the same string would make the bitmap table cells map to the same cells in the AB for all the hash functions, increasing the number of collisions.

First, we assign a global column identifier to each column in the bitmap table. Then, we use both the column and the row number to construct $x = F(i, j)$ where i is the row number and j is the column number.

When we construct one AB per data set or one AB per attribute, we define $F(i, j) = i << w||j$, where w is a user-defined offset. This string is in fact unique when w is large enough to accommodate all j. In the case when we construct one approximate bitmap per column we use $F(i, j) = i$, since the column number is already encoded in the AB itself.

3.2.2 Hash Functions

The hash functions used to build the AB have an impact in both the execution time and the accuracy of the results. We explore two hash function approaches. In the first one, we compute a single large hash function and use k partial values from the hash output to decide which bitmap bits to set. The motivation is to prevent the overhead introduced when computing several hash functions. In addition, such functions usually have hardware support which makes them very fast. In the second approach, we try k independent hash functions and use their hash values to set the bits in the approximate bitmap.

Single Hash Function. The main advantage of using a single hash function is that the function is called once and then the result is divided into pieces and each piece is considered to be the value of a different hash function. Table 1 illustrates this approach. However, the hash function selected should be secure in the sense that no patterns or similar runs should be found in the output. The reason is that the mapping of the inputs will point to different bits of the AB, and the chances of collisions in the AB will be smaller. For the single hash function approach we adopt the Secure Hash Algorithm (SHA), developed by National Institute of Standards and Technology (NIST), along with the NSA, for use with the Digital Signature Standard (DSS) [33].

Independent Hash Functions. Hash functions can be designed to take into account the characteristics of the bitmaps being encoded: each row number appears exactly once per attribute and only one column per attribute is set per row. Some hash functions would perform better than others. For example, a hash function that ignores the column number such as $F(i, j) = i$, would perform poorly when encoding the bitmaps as one AB per data set or one AB per attribute. Using this function, all bits are set in the insertion step since each row i has at least one 1 in it. The retrieval algorithm would always find all the bits set and the answer would have a false positive rate of 1, i.e., every cell considered in the query would be reported as an answer. We define two hash functions, namely Column Group and Circular Hash, in Section 5.2.2.

3.2.3 Bitmap Query Processing with AB

Generally, the type of queries executed over bitmaps are *point and range queries* over columns. Other type of queries would require more bitmap operations or decompression of the final bitmap because of the column-wise storage and the run-length compression which prevents the direct access of a given bit in the bit vector. Our technique can support these types of queries as well because the compression and storage of the bitmap is not done column-wise. For comparison with the current bitmap techniques we define rectangular queries over the AB and logic for interval evaluation.

Definition: A query $Q = \{(A_1, l_1, u_1), ..., (A_{qdim}, l_{qdim}, u_{qdim}), (R, r_l, ..., r_x)\}$ is a bitmap query over a bitmap table B and query result $T = \{b_1, b_2, ..., b_x\}$ is a set of bits such that

$$b_k = \bigwedge_{i=1}^{qdim} \left(\bigvee_{j=li}^{ui} B(k, A_{ij}) \right)$$

Let $Q_3 = \{(A, 1, 2), (R, 4, 5, 6, 7, 8)\}$ be a query example over bitmap table in Figure 6. This query is asking for the rows between 4 and 8 where Attribute A falls into bin 1 or 2. Traditional bitmaps would apply the bit-wise OR operation over the

Hash Function	H_0	H_1	H_2	...	H_{10}
Bits	$159-144$	$143-128$	$127-112$	...	$15-0$
SHA Output	0100100010001010	1000010100100001	0111100011100010	...	0000010101110011

Table 1: Single Hash Function 160-bit output split into 10 sets (k=10) of 16 bits (AB Size = 2^{16}).

```
retrieve(Q)
01 T = ∅
02 for each (i) in R
03     andpart = true
04     for A = 1 to qdim
05         orpart = false
06         for j = A(l_A) to A(u_A)
07             x = F(i,j)
08             in = true
09             for t = 1 to k
10                 if AB[H_t(x)]==0
11                     in = false
12                     break loop
13             orpart = orpart OR in
14             if orpart == true
15                 break loop
16         andpart = andpart AND orpart
17         if andpart == false
18             T = T ∪ {0}
19             break loop
20     if andpart == true
21         T = T ∪ {1}
21         break loop
return(T)
```

Figure 7: Retrieval Algorithm for Bitmap Queries

bitmaps corresponding to Attribute A value 1 and Attribute A value 2. Then, it would have to scan the resulting bitmap to find the values for positions 4 through 8 or perform a bit-wise AND operation with the resulting bitmap and an auxiliary bitmap which only has set positions 4 through 8 to provide the final answer. The exact answer of this query would be $T = \{0, 1, 1, 1, 0\}$ which translates into row numbers 5, 6, and 7. To execute this query using AB we first find the approximate value for the first cell of the query $(4, 1)$. If the value is 1 then we can skip the rest of the row since the answer is going to be 1. If the value is 0, we approximate the value for the next cell $(4, 2)$ and OR them together. We then continue processing rows 5, 6, 7, and 8 similarly. Q_3 is a one dimensional query because it is only asking for one attribute. $Q_4 = \{(A, 1, 2), (B, 2, 3), (R, 4, 5, 6, 7, 8)\}$ is a two dimensional query asking for the rows between 4 and 8 where Attribute A falls into bin 1 or 2 and Attribute B falls into bin 2 or 3. Here each interval is evaluated as a one dimensional query and the result is ANDed together. The retrieval algorithm for bitmap queries using AB is given in Figure 7.

4. ANALYSIS OF PROPOSED SCHEME

In this section, we analyze the false positive rate of the AB and provide the theoretical foundation to compute the size and the precision for each level of encoding.

4.1 False Positive Rate

In this section we analyze the false positive rate of the AB and discuss how to select the parameters. We use the notation given in Table 2.

Symbol	*Meaning*
N	Number of rows in the relation
C	Number of columns in the relation
d	Number of attributes
s	Number of bits that are set
k	Number of hash functions
n	AB Size
m	Hash Function Size ($\log_2$n)
α	AB Size Parameter ($\lceil s/n \rceil$)

Table 2: Notation

Assuming the hash transforms are random, the probability of a bitmap cell being set by a single hash function is $\frac{1}{n}$, and not being set is $1 - \frac{1}{n}$. After s elements are inserted into the AB, the probability of a particular bit being zero (or non-set) is given as

$$(1 - \frac{1}{n})^{ks} \approx e^{-\frac{ks}{n}}$$

The probability that all the k bits of an element are already marked (false positive) is

$$(1 - (1 - \frac{1}{n})^{ks})^k \approx (1 - e^{-\frac{ks}{n}})^k$$

Since most of the large scientific data sets are read-only, we know the parameter s, and we have control over the parameters k and n. Ideally we want a data structure whose size depends only on number of set bits s. For sparse matrices, such as most bitmaps, the size of the data structure should be small. Assume that we use an AB whose size n is αs, where α is an integer denoting how much space is allocated as a multiple of s. The false positive rate of the AB can be expressed as

$$(1 - (1 - \frac{1}{\alpha s})^{ks})^k \approx (1 - e^{-\frac{ks}{\alpha s}})^k = (1 - e^{-\frac{k}{\alpha}})^k$$

False positive rate is plotted in Figure 8. As α increases false positive rate goes down since collisions are less likely in a large AB. For a fixed α, false positive rate depends on k. The value of k that minimizes the false positive rate can be found by taking derivative of false positive rate and setting it to zero. The change in false positive rate for some values of α is given in Figure 9.

4.2 Size vs. Precision

In this section, we formulate the size and the precision of AB, as a function of the input parameters.

Let $D = \{A_1, A_2, ..., A_d\}$ be a data set with N records and let C_i be the cardinality of attribute A_i. The number of set bits s under various scenarios can be computed as follows:

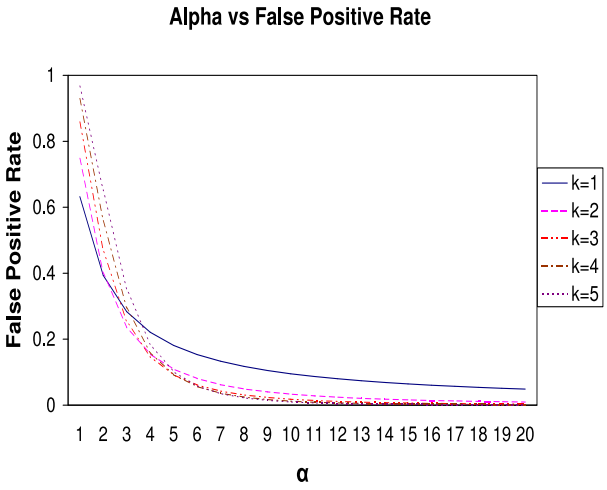


Figure 8: False Positive Rate as a function of α

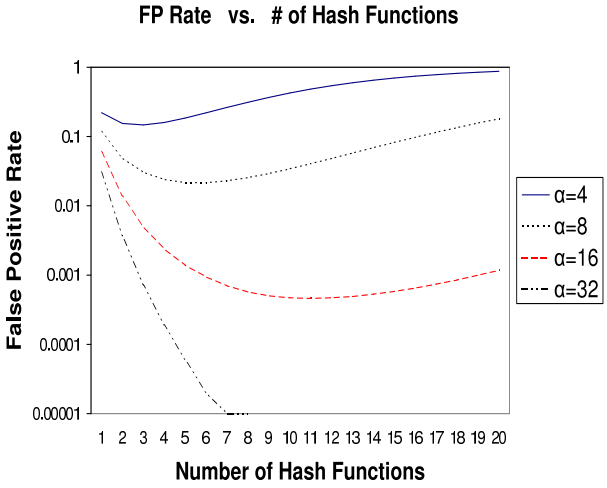


Figure 9: False Positive Rate as a function of k

- *One AB for the whole data set:* In this case we only construct one approximate bitmap and the number of set bits $s = dN$. False positive rate of this AB is $(1 - e^{\frac{-k}{\alpha_1 dN}})^k$.
- *One AB for each attribute:* In this case we construct d approximate bitmaps and the number of set bits $s = N$ for each one. The false positive rate of each AB is $(1-e^{\frac{-k}{\alpha_2 N}})^k$. Compared with above scenario, we can use an AB that has $\alpha_2 = \frac{\alpha_1}{N}$ and still achieve the same false positive rate.
- *One AB for each column:* In this case we construct $\sum_{i=1}^{d} C_i$ approximate bitmaps, one for each attribute value pair, and the number of set bits (s) may be different for each bitmap depending on the number of records that have that particular value.

Let the size of the approximate bitmap be 2^m. Using the notation from the previous section, we can derive m to be

$$m = \lceil (log_2(s\alpha)) \rceil \quad (1)$$

Precision (P) is defined as a function of the false positive rate (FP) as follows

$$P = 1 - FP = 1 - (1 - e^{-\frac{k}{\alpha}})^k$$

Given a maximum AB size $2^{m_{max}}$, α can be computed using Equation 1 above. Largest possible AB size is chosen since large ABs are preferable for their low false positive rate. We then compute the minimum k that maximizes P (minimizes FP).

Given a minimum precision P_{min} the AB size can be computed using α, which can be computed as a function of k as the following

$$\alpha = \frac{-k}{ln(1 - e^{\frac{ln(1-P_{min})}{k}})}$$

In order to decide whether to build one AB per data set or one AB per attribute, one needs to compare the AB sizes: AB per data set ($2^{\lceil log_2(dN\alpha) \rceil}$) vs. AB per attribute ($2^{\lceil log_2(N\alpha) \rceil} d$). In our case, where we always make α to be a power of 2, it is the same to compare $2^{\lceil log_2(dN) \rceil}$ with $2^{\lceil log_2 N \rceil} d$. If $2^{\lceil log_2(dN) \rceil}$ is less than $2^{\lceil log_2 N \rceil} d$, then one AB per data set would give better compression. It can be seen that if n is a power of 2 and d is not, then it is always better to construct one AB per attribute.

On the other hand, to decide whether to build one AB per attribute or one AB per column, one will need the number of set bits for each column and will need to compare $2^{\lceil log_2(N\alpha) \rceil} d$ with

$$\sum_{i=1}^{d} \sum_{j=1}^{C_d} 2^{\lceil log_2(s_{ij}\alpha) \rceil}$$

In our experience, if the attribute is uniformly distributed it is better to build one AB per column. However, for highly skewed data it is better to build one AB per attribute.

5. EXPERIMENTAL FRAMEWORK

5.1 Data Sets

For the experiments we use two real and one synthetic data sets. Table 3 gives a description for each data set. Uniform is a randomly generated data set following a uniform distribution, HEP is data from high energy physic experiments, and Landsat is the is the SVD transformation of satellite images. The uniformly distributed data set is a representative data set for bitmap experiments, since generally data need to be discretized into bins before constructing the bitmaps and it is well known that having bins with the same number of points is better than having bins with the same interval size. The intuition is that evenly distributing skewed attributes achieve uniform search times for all queries. Effectively, any data set can be encoded into uniformly distributed bitmaps by dividing the attribute domain into bins that roughly have the same number of data points.

5.2 Hash Functions

5.2.1 Single Hash Function (SHA-1)

SHA is a cryptographic message digest algorithm similar to the MD4 family of hash functions developed by Rivest (RSA). It differs

Data set	Rows	Attributes	Bitmaps	Setbits	Uncompressed Bitmap Size (bytes)	WAH Size (bytes)	Compression Ratio
Uniform	100,000	2	100	200,000	1,290,800	922,868	0.71
Landsat	275,465	60	900	16,527,900	31,993,200	30,103,376	0.94
HEP	2,173,762	6	66	13,042,572	18,512,472	12,021,328	0.65

Table 3: Data Set Descriptions

Data set	Number of ABs	$\alpha = 2$	$\alpha = 4$	$\alpha = 8$	$\alpha = 16$
Uniform	1	65,536	131,072	262,144	524,288
Landsat	1	4,194,304	8,388,608	16,777,216	33,554,432
HEP	1	4,194,304	8,388,608	16,777,216	33,554,432

Table 4: AB Size (in bytes) as a function of α. One AB per data set

in that the entire transformation was designed to accommodate the DSS block size for efficiency.

The Secure Hash Algorithm takes a message of any length and produces a 160-bit message digest, which is designed so that finding a text which matches a given hash is computationally very expensive.

We utilize SHA-1 [33] algorithm in our single hash function implementation. SHA-1 was a revision to SHA to overcome a weakness, and the revision corrected an unpublished defect in SHA.

5.2.2 Independent Hash Functions

The purpose of using different hash functions is to measure the impact of the hash function in the precision of the results. Here we describe the hash functions we use in the experiments. The rest of the hash functions used in this work come from the General Purpose Hash Function Algorithms Library [30] with small variations to account for the size of the AB.

- **Column Group.** This hash function splits the AB into groups. The number of groups is the cardinality of the attribute. The group number is selected based on the column number and the offset is computed using the modulo operation over the row number. We only use this hash function when we construct one AB per data set or one AB per attribute. $H(i,j) = jn + (i \bmod n)$.
- **Circular Hash.** This hash function constructs a unique number using the row and column number and maps the number to a cell in the AB using the modulo operation. In the case when we have one approximate bitmap per column only the row number is used. $H(x) = x \bmod n$.

5.3 Queries

As the query generation strategy we used sampling. For sampled queries there is at least one row that match the query criteria. This fact is important for AB experiments because if the number of actual query results is 0, the precision of the AB would always be 0. Queries producing 2 or $1,000$ false positives would have the same precision. The input parameters for the query generator are:

- *Number of queries (q)*: The total number of queries to be generated. We set this parameter to 100.
- *Data set (D= $\{A_1, A_2, ...A_d\}$)*: This is the data set for which we generate the queries. The number of attributes d, each attribute A_i where $1 \leq i \leq d$, the cardinality of each attribute C_i, and the number of rows N can all be derived from D.

Data set	$qdim$	sel	r
Uniform	2	6	.1,.5,1,5,10
Landsat	2	20	.04,.2,.4,2,4
HEP	2	25	.005,.01,.05,.1,.5

Table 7: Parameter Values for Query Generation

- *Query dimensionality ($qdim$)*: The number of attributes in the query. $1 \leq qdim \leq d$
- *Attribute selectivity (sel)*: The percentage of values from the attribute cardinality that constitute the interval queried for that attribute.
- *Percent of rows (r)*: The percentage of rows that is selected in the queries. A hundred percent indicates that the query is executed over the whole data set. The range for the rows is produced using the row number, i.e., the physical order of the data set. The lower value l is picked randomly between 1 and N. The upper bound u is computed as $l + (r * N)$, but in the case when this value is greater than N, we set $u = N$.

For the query generation, we randomly select q rows from the data set. Let us identify those rows by $\{r_1, r_2, ..., r_q\}$. Each query q_j is based on r_j. $qdim$ distinct attributes are selected randomly for each query. For each A_i picked where $\{1 \leq i \leq qdim\}$, the lower bound l_i is given by the value of A_i in r_j. The upper bound u_i is computed as $l_i + (sel * C_i)$, but in the case this value is greater than C_i, we set $u_i = C_i$.

5.4 Experimental Setup

To generate the approximate bitmaps we set α to be a power of 2 between 2 and 16, and k to be between 1 and 10. The results presented in the next section are using the largest α for the given data set for which the AB Size is smaller or comparable to the WAH bitmap size.

Table 7 gives the list of parameter values for query generation. We adjust the parameters to have 2 dimensional queries of 4 columns each, and varying the number of rows from 100 to 10,000 (100,500, 1K,5K,10K) for all data sets.

6. EXPERIMENTAL RESULTS

In this section we measure size, precision, and execution time of the Approximate Bitmaps. We compare size and execution time against the WAH-Compressed Bitmaps [43, 44, 46]. Results are given varying the appropriate parameters:

Data set	Number of ABs	$\alpha = 2$		$\alpha = 4$		$\alpha = 8$		$\alpha = 16$	
		Single AB	All ABs	Single AB	All ABs	Single AB	All ABs	Single AB	All ABs
Uniform	2	32,768	65,536	65,536	131,072	131,072	262,144	262,144	524,288
Landsat	60	131,072	7,864,320	262,144	15,728,640	524,288	31,457,280	1,048,576	62,914,560
HEP	6	1,048,576	6,291,456	2,097,152	12,582,912	4,194,304	25,165,824	8,388,608	50,331,648

Table 5: AB Size (in bytes) as a function of α. One AB per attribute

Data set	Number of ABs	$\alpha = 2$		$\alpha = 4$		$\alpha = 8$		$\alpha = 16$	
		Avg AB	All ABs	Avg AB	All ABs	Avg AB	All ABs	Avg AB	All ABs
Uniform	100	574	57,344	1,147	114,688	2,294	229,376	4,588	458,752
Landsat	900	6,809	6,127,616	13,617	12,255,232	27,234	24,510,464	54,468	49,020,928
HEP	66	67,986	4,487,048	135,972	8,974,096	271,943	17,948,194	543,885	35,896,388

Table 6: AB Size (in bytes) as a function of α. One AB per column

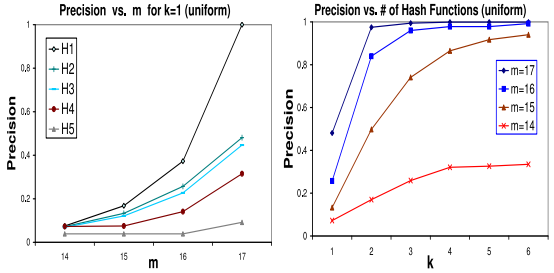
- The AB size, changing α and m.
- The number of hash functions k.
- The number of rows queried.

The results presented in the following subsections are averages over 100 queries of the same type using independent hash functions instead of SHA-1. At the end of this section, SHA-1 results are discussed. We show that by setting the experimental parameters such that the AB size is less than or at least comparable to the WAH-compressed bitmaps, we obtain good precision (more than 90% in most cases) and execution time from 1 to 3 orders of magnitude faster than WAH.

6.1 AB Size

Tables 4, 5, and 6 show the AB sizes in bytes for the cases when we construct *one AB per data set*, *one AB per attribute*, and *one AB per column*, respectively. The size of each AB is calculated based on the discussion in Section 4.2, i.e. finding the lowest power of 2 that is greater or equal to $s\alpha$. For example, in Table 4 the value for Landsat data for $\alpha = 4$ is calculated as follows. The number of set bits (s) for Landsat is 16,527,900. The lowest power of 2 that is greater or equal to $s\alpha$ is 67,108,864 in bits, and 8,388,608 in bytes. Note that this is also the size we obtain for HEP data, since we are restricting ourselves to powers of 2. Similarly, in Table 5 the Landsat data has 60 ABs. In this case, the parameter s is equal to the number of rows, which is 275,465 (given in Table 3). Therefore, the lowest power of 2 we are seeking per an AB (for $\alpha = 4$) is 2,097,152 in bits, and 262,144 in bytes. However, for 60 ABs we have 15,728,640 bytes in total. The values in Table 6 are calculated similarly with the only difference that the number of set bits s varies for each column. Recall that the number of set bits for each column would be the number of rows that fall into the corresponding bin. Therefore the size is not the same for all ABs. Table 6 presents the average and total AB size.

- For Uniform data, the best size is obtained when constructing *one AB per column*. The reason is that the difference between the data set dimensionality (2) and the attribute cardinality (50) is enough to achieve the same precision (α) by constructing one AB per column or one AB for the whole data set. However, the former uses less space than the latter, e.g., 524,288 bytes in Table 4 vs. 458,752 bytes in Table 6.
- For Landsat and HEP data sets, which have more attributes, constructing one AB for the whole data set requires less space than constructing one AB per column (refer to Tables 4 and 6).



(a) *Different hash functions* (b) *Different # of hash functions*

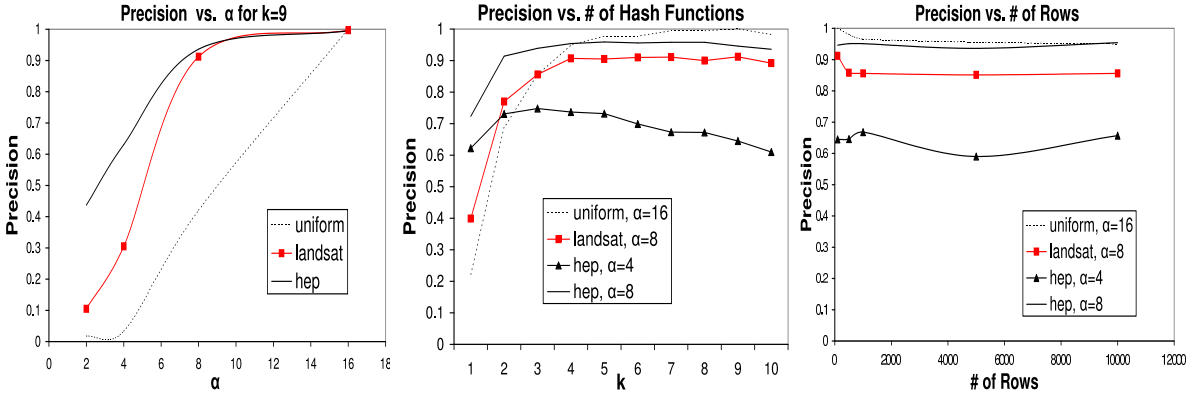
Figure 10: The effect of utilizing more space. ($m = log_2 n$)

Experiments show that AB can always be constructed using less space than WAH. The following are the α for which AB compresses better or comparable to WAH compressed bitmaps.

- For uniform data, when we construct one AB per column for $\alpha = 16$, the AB total size is less than half of the total size required by the WAH-compressed bitmaps.
- For HEP data, $\alpha = 4$ produces AB whose total size is about two thirds of the WAH-compressed bitmaps. For $\alpha = 8$ the AB size is comparable to the WAH size needing only one third more of the space and the AB size is still smaller than the uncompressed bitmaps. Since the theoretical false positive rate for $\alpha = 4$ is high (given in Figure 9), the results for HEP data for $\alpha = 8$ are also included in the following subsections.
- For Landsat data, $\alpha = 8$ produces ABs whose total size is about half the size of the WAH-compressed bitmaps.

6.2 Precision

Figure 10(a) supports our initial claim that the selection of the hash functions have a significant impact in the accuracy of the results if only one hash function is utilized. The Figure sketches that the precision varies for the same m with different hash functions. As m increases, the precision also increases since there are less collisions as a hash function can map elements to more values. For H_1, the precision is 1 when there are enough bits to accommodate all rows.



(a) *function of* α (b) *function of # of hash functions* (c) *function of # of rows*

Figure 11: Precision Results

However, as we increase k the false positive rate decreases and the impact of using one set of hash functions over another is not evident as all of them perform similarly with only small variations. Figure 10(b) shows the precision as a function of the number of hash functions, i.e., k. As can be seen, the precision increases for larger k. However, after certain k, the improvement is not very significant and it may even start to decrease. The optimal k can be computed using the theoretical foundation given in Section 4.

Figure 11(a) illustrates the precision for all the data sets as α increases. Note that the precision increases steadily as α increases and it is very close to 1 for larger α.

For a specific α of each data set, Figure 11(b) depicts the precision as a function of the number of hash functions (k). Up to the optimal k, the precision increases as k increases. After the optimum point, the precision starts decreasing because a large number of hash functions produces more collisions.

Figure 11(c) shows that the precision is independent of the number of rows queried remaining constant for each data set. The small variations in the precision are caused by having different columns queried in each case. For uniform data and queries of 10K rows, on the average, the number of tuples retrieve by each query is 59 and AB returns only 3 more tuples on the average. For queries of 100 rows, the results are much more accurate, i.e., 100 different queries using WAH returned 170 tuples and the same queries using AB returned 179 tuples. For landsat data and queries of 10K rows, the number of tuples returned by each query is 723 and the number of tuples returned by AB is 821, and for queries of 100 rows, the average matches per query is 8.98 for WAH and 9.85 for AB. For HEP and queries of 10K rows, WAH retrieved 3861 and AB retrieved 4039 tuples, and for queries of 100 rows, the average number of results is 42 for WAH and 44 for AB.

6.3 Execution Time

Figure 12 shows the CPU clock time in milliseconds (msec) for different α. As α increases the execution time decreases because the false positive rate gets smaller. Execution time is the same for all the data sets being proportional to the number of tuples returned that match the query.

Figure 13 shows the CPU clock time in msec as a function of the number of hash functions (k). As k increases the execution time increases linearly. That is because the time for iterations of the utilized hash functions are close, and the total consumed time is the addition of them.

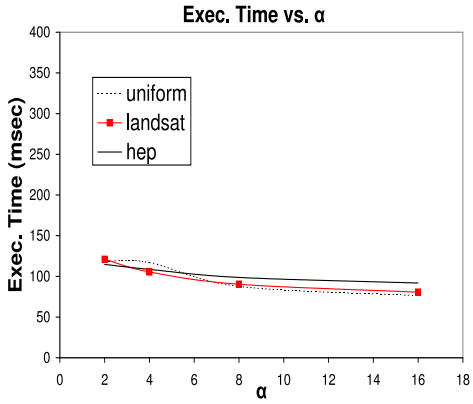
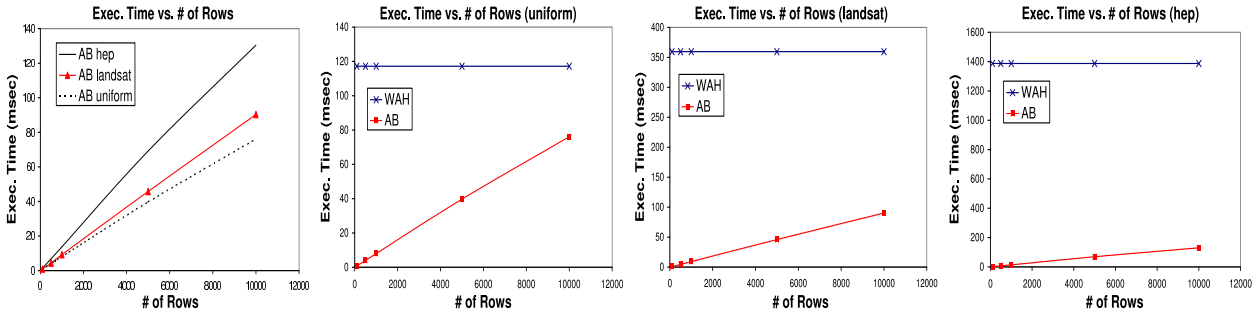


Figure 12: Execution Time as a function of α.

Figure 14 depicts the CPU clock time by varying the number of rows queried. The reason for the execution time of HEP data being longer than the other data sets, in Figure 14(a), is that the number of tuples that match the query for HEP data is higher than the others (as given in the last part of the previous section). For WAH, only the time it takes to execute the query without any row filtering is measured. This is the reason why the WAH execution time is constant for any number of rows. The final answer can be computed by performing an extra bitmap operation or by locating the relevant rows in the compressed bitmap. On the other hand, AB execution time is linear in the number of rows queried. In Figure 14(b), for a query targeting 10K rows of the uniform data, AB execution time is 76.09 msec, which is two thirds of WAH execution time. In Figure 14(c), AB time is one fourth of WAH time for the same number of rows. In Figure 14(c), AB time is one tenth of WAH for the 10K rows. In addition, for lower number of rows, the AB execution time improvement over WAH is in 2 orders of magnitude for uniform and landsat data sets, and 3 orders of magnitude for HEP data set. Experiments show that, for all the data sets, executing a query that selects up to around 15% of the rows by using AB is still faster than using WAH-compressed bitmaps.



(a) *all data sets* (b) *uniform*, α=16 (c) *landsat*, α=8 (d) *hep*, α=8

Figure 14: Execution Time Performance: WAH vs. AB

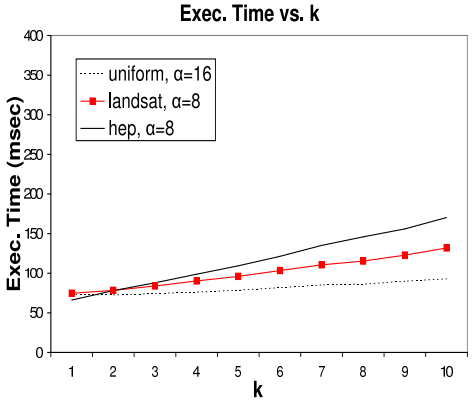


Figure 13: Execution Time as a function of k.

6.4 Single Hash Function vs. Independent Hash Functions

In terms of precision, SHA-1 results are very similar to the results obtained by using the independent hash functions. However, as the main purpose of SHA-1 is to have a secure hash function, the computation cost is very expensive and thus SHA-1 is slower than the other hash functions used in this work. Furthermore, the performance of SHA-1 can be enhanced by incorporating hardware support, which is out of the scope of this paper.

7. CONCLUSION

We addressed the problem of efficiently executing queries with no overhead over compressed bitmaps. Our scheme generates an Approximate Bitmap (AB) and stores only the set bits of the exact bitmaps using multiple hash functions. Data is compressed and directly accessed by the hash function values. The scheme is shown to be especially effective when the query retrieves a subset of rows.

The Approximate Bitmap (AB) encoding can be applied at three different levels *one AB per data set*, *one AB per attribute*, and *one AB per column*. Precision of the AB is expressed in terms of α. For the same α, the precision is the same for all levels. The size of the AB depends on the number of set bits and α. Skewed and uniform distributions in the bitmaps would benefit from constructing *one AB per attribute* and *one AB per column*, respectively. For high dimensional databases and high cardinality attributes *one AB per data set* offers the best size. In all cases, allocating more bits to the AB increases the precision of the results since the number of collisions will be less in a larger size AB. In general, precision increases with higher number of utilized hash functions, however, after an optimal point the precision starts to degrade. That point can be driven using the theoretical analysis provided in the paper. Another advantage of AB is that the precision is not affected by the number of rows queried.

We compared our approach with WAH, the state-of-the-art in bitmap compression for fast querying. The size parameter of AB, for the experiments, is chosen in such a way that the total space used by our technique remains less than or comparable with the space used by WAH. AB achieves accurate results (90%-100%) and improves the speed of query processing from 1 to 3 orders of magnitude compared to WAH. The performance can be further improved by incorporating hardware support for hashing.

8. REFERENCES

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

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

[3] G. Antoshenkov. Byte-aligned bitmap compression. In *Data Compression Conference*, Nashua, NH, 1995. Oracle Corp.

[4] G. Antoshenkov and M. Ziauddin. Query processing and optimization in oracle rdb. *The VLDB Journal*, 1996.

[5] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[6] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. In *Proceedings of the 40th Annual Allerton Conference on Communication, Control, and Computing*, pages 636–646, 2002.

[7] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks. Proceedings of ACM SIGCOMM, August 2002, pp. 47-60, August 2002.

[8] C.Y. Chan and Y.E. Ioannidis. Bitmap index design and evaluation. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 355–366. ACM Press, 1998.

[9] C.Y. Chan and Y.E. Ioannidis. An efficient bitmap encoding scheme for selection queries. *SIGMOD Rec.*, 28(2):215–226, 1999.

[10] Wu chang Feng, D.D. Kandlur, D. Saha, and K.G. Shin. Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness. In *Proc. of INFOCOM*, volume 3, pages 1520 – 1529, April 2001.

[11] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. Compactly encoding a function with static support in order to support approximate evaluations queries. Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms, January 2004.

[12] H. Edelstein. Faster data warehouses. Information Week, December 1995.

[13] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *IEEE/ACM Transactions on Networking*, Canada, 2000.

[14] L. Fan, P. Cao, J. Almeida, and A. Broder. Web cache sharing. Collaborating Web caches use bloom filter to represent local set of cached files to reduce the netwrok traffic. In *IEEE/ACM Transactions on Networking*, 2000.

[15] Informix Inc. Informix decision support indexing for the enterprise data warehouse. http://www.informix.com/informix/corpinfo/-zines/whiteidx.htm.

[16] Sybase Inc. *Sybase IQ Indexes*, chapter 5: Sybase IQ Release 11.2 Collection. March 1997.

[17] D. Johnson, S. Krishnan, J. Chhugani, S. Kumar, and S. Venkatasubramanian. Compressing large boolean matrices using reordering techniques. In *VLDB 2004*.

[18] T. Johnson. Performance measurement of compressed bitmap indices. In *VLDB*, pages 278–289, 1999.

[19] J. Chen K. Wu, W. Koegler and A. Shoshani. Using bitmap index for interactive exploration of large datasets. *In Proceedings of SSDBM*, 2003.

[20] N. Koudas. Space efficient bitmap indexing. In *Proceedings of the ninth international conference on Information and knowledge management*, pages 194–201. ACM Press, 2000.

[21] A. Kumar, J.J. Xu, and J. Wang L. Li. Algorithms: Space-code bloom filter for efficient traffic flow measurement. In *Proceedings of the 2003 ACM SIGCOMM conference on Internet measurement*, October 2003.

[22] A. Kumar, J.J. Xu, L. Li, and J. Wang. Measuring approximately yet reasonably accurate per-flow accounting without maintaining per-flow state. Proceedings of the 2003 ACM SIGCOMM conference on Internet measurement, 2003 October.

[23] P. Mishra and M.H. Eich. Join processing in relational databases. In *ACM Computing Surveys (CSUR)*, March 1992.

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

[25] J.K. Mullin. Estimating the size of joins in distributed databases where communication cost must be maintained low. In *IEEE Transactions on Software Engineering*, 1990.

[26] J.K. Mullin. Optimal semijoins for distributed database systems. In *IEEE Transactions on Software Engineering*, volume 16, pages 558 – 560, 1990.

[27] P.E. O'Neil. Model 204 architecture and performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 40–59. Springer-Verlag, 1989.

[28] P.E. O'Neil. Informix and indexing support for data warehouses, 1997.

[29] P.E. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 38–49. ACM Press, 1997.

[30] A. Partow. General purpose hash function algorithms library. http://www.partow.net/programming/hashfunctions/index.html, 2002.

[31] A. Pinar, T. Tao, and H. Ferhatosmanoglu. Compressing bitmap indices by data reorganization. *ICDE*, pages 310–321, 2005.

[32] M.V. Ramakrishna. In *Indexing Goes a New Direction.*, volume 2, page 70, 1999.

[33] M.J.B. Robshaw. Md2, md4, md5, sha and other hash functions. technical report tr-101, version 4.0. RSA Laboratories, July 1995.

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

[35] A.C. Snoeren. Hash-based IP traceback. In *ACM SIGCOMM Computer Communication Review*, 2001.

[36] A.C. Snoeren, C. Partridge, L.A. Sanchez, C.E. Jones, F. Tchakountio, B. Schwartz, S.T. Kent, and W.T. Strayer. IP Traceback to record packet digests traffic forwarded by the routers. IEEE/ACM Transactions on Networking (TON), December 2002.

[37] K. Stockinger. Bitmap indices for speeding up high-dimensional data analysis. In *Proceedings of the 13th International Conference on Database and Expert Systems Applications*, pages 881–890. Springer-Verlag, 2002.

[38] K. Stockinger, J. Shalf, W. Bethel, and K. Wu. Dex: Increasing the capability of scientific data analysis pipelines by using efficient bitmap indices to accelerate scientific visualization. *In Proceedings of SSDBM*, 2005.

[39] K. Stockinger and K. Wu. Improved searching for spatial features in spatio-temporal data. In *Technical Report. Lawrence Berkeley National Laboratory. Paper LBNL-56376.* http://repositories.cdlib.org/lbnl/LBNL-56376, September 2004.

[40] J. Wang. Caching proxy servers on the world wide web to improve performance and reduce traffic, October 1999.

[41] A. Whitaker and D. Wetherall. Detecting loops in small networks. 5th IEEE Conference on Open Architectures and Network Programming (OPENARCH), June 2002.

[42] K. Wu, E.J. Otoo, and A. Shoshani. A performance comparison of bitmap indexes. In *Proc. Conf. on 10th International Conference on Information and Knowledge Management*, pages 559–561. ACM Press, 2001.

[43] K. Wu, E.J. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *SSDBM*, pages 99–108, Edinburgh, Scotland, UK, July 2002.

[44] K. Wu, E.J. Otoo, and A. Shoshani. An efficient compression scheme for bitmap indices. Technical Report 49626, LBNL, April 2004.

[45] K. Wu, E.J. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. Technical Report LBNL-54673, Lawrence Berkeley National Laboratory, March 2004.

[46] K. Wu, E.J. Otoo, and A. Shoshani. Optimizing bitmap indexes with efficient compression. ACM Transactions on Database Systems (To appear), 2006.

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

[48] M.C. Wu. Query optimization for selections using bitmaps. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 227–238. ACM Press, 1999.