Hybrid Query Optimization for Hard-to-Compress Bit-vectors

Gheorghi Guzun · Guadalupe Canahuate

Abstract Bit-vectors are widely used for indexing and summarizing data due to their efficient processing in modern computers. Sparse bit-vectors can be further compressed to reduce their space requirement. Special compression schemes based on run-length encoders have been designed to avoid explicit decompression and minimize the decoding overhead during query execution. Moreover, highly compressed bit-vectors can exhibit a faster query time than the non-compressed ones. However, for hard-to-compress bit-vectors, compression does not speed up queries, and can add considerable overhead. In these cases, bit-vectors are often stored verbatim (non-compressed).

On the other hand, queries are answered by executing a cascade of bit-wise operations involving indexed bit-vectors and intermediate results. Often, even when the original bit-vectors are hard-to-compress, the intermediate results become sparse. It could be feasible to improve query performance by compressing these bit-vectors as the query is executed. In this scenario it would be necessary to operate verbatim and compressed bit-vectors together. In this paper, we propose a hybrid framework where compressed and verbatim bitmaps can coexist and design algorithms to execute queries under this hybrid model. Our query optimizer is able to decide at run time when to compress or decompress a bit-vector. Our heuristics show that the applications using higher density bitmaps can benefit from using this hybrid model, improving both their query time and memory utilization.

G. Guzun
The University of Iowa
4016 Seamans Center for the Engineering Arts and Sciences
Tel.: +1 319-335-5197
Fax: +1 319-335-6028
E-mail: gheorghi-guzun@uiowa.edu

G. Canahuate
The University of Iowa
4016 Seamans Center for the Engineering Arts and Sciences
Tel.: +1 319-335-5197
Fax: +1 319-335-6028
E-mail: guadalupe-canahuate@uiowa.edu

1 Introduction

From business analytics to scientific analysis, today's applications are increasingly data-driven, proliferating in data stores that are rapidly growing. To support efficient *ad hoc* queries, appropriate indexing mechanisms must be in place. Bit-vector based indices, also known as bitmap indices, are popular indexing techniques for enabling fast query execution over large scale data sets. Because bit-vector based indices can leverage fast bit-wise operations supported by hardware, they have been extensively used for selection and aggregation queries in data warehouses and scientific applications.

A bitmap index is used to represent a property or attribute value-range. For a simple bitmap encoding, each bit in the bitmap vector corresponds to an object or record in a table and bit positions are set only for the objects that satisfy the bitmap property. For categorical attributes, one bitmap vector is created for each attribute value. Continuous attributes are discretized into a set of ranges (or bins) and bitmaps are generated for each bin. Selection queries such as point and range queries are executed by means of intersections and/or unions of the relevant bitmaps.

When simple encoding is used and a large number of bit-vectors are created for a data set, the index grows larger but also the bit-vectors are sparser and therefore amenable for compression. Bitmap indices are compressed using *run-length* based encoding. A *run* refers

to a set of consecutive bits with the same value, i.e., all 0s or all 1s. Sparse bitmaps will have long runs of zeros. Current bitmap compression techniques minimize the compression overhead in query time by allowing operations to work directly over the compressed bitmaps [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Most of them can be considered as extensions of Word Aligned Hybrid (WAH) encoding[4, 11]. These approaches are effective for sparse bitmaps but impose an overhead for harder-to-compress bitmaps, as the compression is not significant when compared to the verbatim bitmap and there is a decoding overhead during query processing.

Depending on the quantization resolution, the bitmaps can have a higher or lower density. With lower quantization resolutions, the bitmaps become denser and thus harder to compress. Another example of high density bit-vector index is the bit-sliced index (BSI) [12] which encodes the binary representation of the attribute value using one bit-vector to represent each binary digit. Since BSI bitmaps exhibit a high bit-density, i.e. a large number of bits is set in each bitmap, they do not compress well. An example of applications using BSI indices are the top-k (preference) queries, which was proposed in [13, 14] and shown to outperform threshold-based algorithms and sequential scan.

The denser bit-vector indices are hard to compress. Compressing them does not save significant storage space. While decoding the compressed bitmap during the query execution imposes a penalty. Because of these reasons, many applications using dense bit-vector based indices do not apply any type of compression. However, many of the real-world queries involve a cascade of bitwise logical operations over the bit-vector index. In many cases these queries aim to retrieve only a very small percentage of the tuples. Thus the intermediate results of the bitwise operations become sparse and can benefit from compression. Compressing the sparse intermediate results can help reduce the memory utilization, and often speed-up the query. One critical condition is that these compressed intermediate bit-vector results should be operable with either other compressed or verbatim bit-vectors.

In this paper, a hybrid query processing and optimization is proposed where queries are executed over both verbatim and compressed bitmaps together. Heuristics are defined to decide when a bitmap column should be compressed/decompressed at run-time. We evaluate our approach for selection and top-k queries. Selection queries are executed using synthetically generated bit-vectors with varying densities, and top-k queries are executed over real and synthetically generated data, using bit-sliced indices (BSI).

The benefit of this hybrid model is the avoidance of decoding overhead by storing hard-to-compress bitmaps as verbatim and still be able to reduce the index space by compressing the sparse bitmaps.

The primary contributions of this paper can be summarized as follows:

- This is the first work that performs queries using both verbatim and compressed bit-vectors operating them together.
- We defined query optimization heuristics to decide at run-time whether the resulting bitmap column should be compressed or not.
- We evaluate the proposed approach for selection and top-k (preference) queries and over equality encoded bitmaps and bit-sliced indices.
- We perform experiments over both synthetic and real data-sets.

The rest of the paper is organized as follows. Section 2 presents background and related work. Section 3 describes the queries and algorithms to operate verbatim and compressed bitmaps together. Section 4 presents the hybrid query optimization and the rationale behind the optimization heuristics. Section 5 shows the experimental results over both real and synthetic data-sets. Finally, conclusions are presented in Section 6.

2 Background and Related Work

This section presents background information for bitmap indices and bitmap compression.

The topic of bitmap indices was introduced in [15]. Several bitmap encoding schemes have been developed, such as equality [15], range [16], interval [16], and workload and attribute distribution oriented [17]. Several commercial database management systems use bitmaps. BSI [15, 18] can be considered a special case of the encoded bitmaps [19]. With bit-sliced indexing, binary vectors are used to encode the binary representation of the attribute value. One BSI is created for each attribute.

Figure 1 shows an example for a data set with two attributes each one with 3 distinct values (cardinality=3). For the equality encoded bitmaps, one bitmap is generated for each attribute value and the bit is set if the tuple has that value.

For the BSIs, since each attribute has three possible values, the number of bit-slices for each BSI is $\lceil log_2 3 \rceil = 2$. The first tuple t_1 has the value 1 for attribute 1, therefore only the bit-slice corresponding to the least significant bit, $B_1[0]$ is set. For attribute 2, since the value is 3, the bit is set in both BSIs. The

	Raw Data		Equality Bitmaps						Bit-Sliced Index (BSI)				BSI SUM		
			Attrib 1			Attrib 2			Attrib 1		Attrib 2				
Tuple	Attrib 1	Attrib 2	=1	=2	=3	=1	=2	=3	$B_1[1]$	$B_1[0]$	$B_2[1]$	$B_2[0]$	$S[2]^3$	$S[1]^2$	$S[0]^1$
t_1	1	3	1	0	0	0	0	1	0	1	1	1	1	0	0
t_2	2	1	0	1	0	1	0	0	1	0	1	0	0	1	1
t_3	1	1	1	0	0	1	0	0	0	1	0	1	0	1	0
t_4	3	3	0	0	1	0	0	1	1	1	1	1	1	1	0
t_5	2	2	0	1	0	0	1	0	0	1	1	0	1	0	0
t_6	3	1	0	0	1	1	0	0	1	1	0	1	1	0	0

[1] $S[0]=B_1[0]$ XOR $B_2[0]$, $C_0 = B_1[0]$ AND $B_2[0]$
[2] $S[1]=B_1[1]$ XOR $B_2[1]$ XOR (C_0)
[3] $S[2]=C_1=Majority(B_1[1],B_2[1],(C_0))$

Fig. 1: Simple example of equality encoded bitmaps and bit-sliced indexing for a table with two attributes and three values per attribute.

SUM for the two BSIs is also shown in the figure. In this case, the maximum value of the sum is 6 and the number of bit-slices is $\lceil log_2 6 \rceil = 3$. The addition of the BSIs representing the two attributes is done using efficient bit-wise operations. First, the bit-slice $sum[0]$ is obtained by XORing $B_1[0]$ and $B_2[0]$ i.e. $sum[0] = B_1[0] \oplus B_2[0]$. Then $sum[1]$ is obtained in the following way $sum[1] = B_1[1] \oplus B_2[1] \oplus (B_1[0] \wedge B_2[0])$. Finally $sum[2] = Majority(B_1[1], B_2[1], (B_1[0] \wedge B_2[0]))$. BSI arithmetic for a number of operations is defined in [18] and top-k (preference) query processing over BSIs is defined in [13, 14]. BSI is a compact representation of an attribute as only $\lceil \log_2 values \rceil$ vectors are needed to represent all values. However, their high density makes them hard to compress any further.

Other types of bitmap indices are typically compressed using specialized run-length encoding schemes that allow queries to be executed without requiring explicit decompression. Byte-aligned Bitmap Code (BBC) [1] was one of the first compression techniques designed for bitmap indices using 8-bits as the group length and four different types of words. BBC compresses the bitmaps compactly and query processing is CPU intensive. Word Aligned Hybrid (WAH) [2] proposes the use of words instead of bytes to match the computer architecture and make access to the bitmaps more CPU-friendly. WAH divides the bitmap into groups of length $w - 1$ and collapse consecutive all-zeros/all-ones groups into a fill word. Recently, several bitmap compression techniques that improve WAH by making better use of the fill word bits have been proposed in the literature [3, 4, 5, 6, 20, 21]. Previous work has also used different segment lengths for encoding [7, 20, 21] and there exists a unified query processing framework using variable length and in theory, different compression methods based on WAH [7].

However, because WAH-based methods require one bit for each word to indicate the type of word, the groups created are not aligned with the verbatim bitmaps where the bit-vector is divided into groups of w-bits. Operating a compressed bitmap with a verbatim bitmap would be very inefficient as the miss-alignment of the row identifiers would need to be resolved at query time.

Luckily, there exists one method, WBC or Enhanced WAH (EWAH) [4, 11], that divides the bitmap into groups of w-bits, not $w-1$-bits. EWAH uses two types of words: marker words and literal words. Half of a marker word is used to encode the fills. The upper half (most significant bits) of the fill word encode the fill value and the run length, and the remaining bits are used to represent the number of literal words following the run encoded in the fill. This implicit word type eliminates the need for a flag bit for each word to identify the type of word.

Figure 2 shows a verbatim bitmap and its EWAH encoding. The verbatim bitmap has 182 bits (6 32-bit words). The EWAH bit-vector always starts with a marker-word. The first half of a marker-word represents the header. For a 32-bit word length as in this case, the first 16 bits indicates the type and number of fill words. The second half of the a marker-word, tells the number of literals that follow a marker-word (1 in the example). After the literal word, another marker-word follows. The first bit (0) indicates a run of zeros and the value 3 is the number of words in the run. The second half of the marker word indicates that there are two literals following the fill.

In this work EWAH is used as the compression method to support an efficient hybrid query processing involving verbatim and compressed bitmaps.

An example of systems that could benefit from the Hybrid query optimization is the Bitmap Index for Database Service (BIDS) [22] proposed for indexing large scale data-stores. BIDS indexes the data using Equality Bitmaps and Bit-Sliced indices. Depending on the attribute cardinality of the data it decides whether to use equality encoded bitmaps, bit-sliced index, or

Verbatim Bitmap (in hex)		400003C0	00000000 00000000 00000000	001FFFF0	000001FF
EWAH Bitmap (in hex)	0000 0001	400003C0	0003 0002	001FFFF0	000001FF

Fig. 2: A verbatim bitmap and its EWAH encoding.

not to create an index for an attribute. The created index, either equality encoded bitmaps or bit-sliced index, is then compressed using WAH. Even when the bit-sliced indices are not effectively compressed it is still necessary to compress them using WAH in order to enable operations between the equality encoded bitmaps and the bit-sliced indices during query processing. This is where BIDS can benefit from the proposed hybrid model. If integrated into BIDS, bitmaps would only be compressed when the compression is effective and no overhead would be imposed to process the higher density bitmaps. Our query processing is able to operate compressed and verbatim bitmaps together. Furthermore, our query optimizer decides at query time whether the intermediate results should be compressed or not.

Roaring Bitmaps [10] proposes an alternative organization to run-length compression. It proposes to divide the bitmaps into chunks of 2^{16} bits and if sparse, store the row ids as an offset within the block and if dense, store as a verbatim bitmaps. This approach is very effective for sparse bitmaps but defaults to verbatim bitmaps for denser bitmaps. We compare our approach to roaring bitmaps and show that the hybrid approach outperforms roaring for hard-to-compress bitmaps.

The next section introduces the proposed hybrid query processing and optimization over verbatim and EWAH-compressed bitmap indices. For clarity, we define the notations used further in this paper in Table 1.

Term	Description
w	Computer architecture word size
n	Number of tuples or rows in the data
m	Number of attributes or dimensions in the data
c	Attribute cardinality, number of equality bitmaps created for the attribute
p	Number of slices used to represent an attribute in a BSI bitmap
s	Number of set bits in a bitmap
$d = \frac{s}{n}$	Density of a bitmap
e	Bit that indicates the storage format of a bitmap. $e = 0$ for verbatim bitmaps and $e = 1$ for compressed bitmaps.
V	A verbatim bitmap
C	A compressed bitmap
H	Hybrid- operation which could involve a verbatim and a compressed bitmap
T	The threshold compression ratio starting at which the bitmaps are stored in a compressed form

Table 1: Notation Reference

3 Hybrid Query Processing

Consider a relational database D with m attributes and n tuples. Bitmap indices are build over each attribute value or range of values and stored column-wise. Query processing over bitmaps is done by executing bit-wise operations over one or more bitmap columns.

A selection query is a set of conditions of the form $A\ op\ v$, where A is an attribute, $op \in \{=; <; \leq; >; \geq\}$ is the operator, and v is the value queried. We refer to point queries as the queries that use the equal operator ($A = v$) for all conditions, and range queries to the queries using a between condition (e.g. $v_1 \leq A \leq v_2$).

The values queried, v_i, are mapped to bitmap bins, b_i, for each attribute. If the bitmaps correspond to the same attribute then the resulting bitmaps are ORed together, otherwise they are ANDed together. In the case of selection queries, the resulting bitmap has set bits for the tuples that satisfy the query constraints.

Equality encoded bitmaps can support selection queries and have been shown to be optimal for point queries. These bitmaps are sparse (even more for higher cardinalities), and therefore can benefit from compression. However, compression comes with a cost during query processing since it requires more time to decode the columns. In some cases, however, compression can speed up the queries and also reduce the space footprint of the bitmaps. This is typically the case for highly compressed columns. For very hard to compress bitmaps, where the number of set bits is higher, compression may not able to reduce the size but still imposes an overhead during query time.

Consider for example a data set uniformly distributed where each one of 100 attributes is divided into two bins. Each bitmap column individually is not able

to compress at all (as the expected bit-density is 0.5). In this case, it may seem beneficial to store the bitmaps as verbatim bitmaps, uncompressed, therefore avoiding decoding overhead during query time. However, consider a point query over several of the dimensions for the same data-set. The expected density quickly decreases with each added dimension. For 4, 10, and 20 attributes queried the expected density for a point query would be 0.06, 0.001, and 0.000001, respectively. In this case, the use of compression would speed up the query considerable as the number of dimensions increases.

More recently, bit-sliced indexing (BSI) was proposed to support top-k (preference) queries [13, 14]. Queries are executed using bit-wise operations to produce the products, additions, and find the top k scores. As an example of such algorithms, we include the SUM operation as presented in [14, 18]. Algorithm 1 shows the pseudo-code for the addition of two BSIs, A and B, with a number of slices q and p, respectively. The result is another BSI S with MAX(p, q)+1 slices. A "Carry" bit-slice CS is used in Algorithm 1 whenever two or three bit-slices are added to form S_i, and a non-zero CS must then be added to the next bit-slice S_{i+1}. An example of a BSI sum was shown earlier in Figure 1.

Together with a MULTIPLY algorithm (not shown in this paper), the SUM algorithm produces a BSI encoding, the weighted sum for all the attributes, while the Top-K algorithm (also not shown) generates the resulting bitmaps with set bits for the top k tuples. With these algorithms, top-k queries can be executed over BSI using AND, OR, XOR, and NOT operations over the bitmaps. Compression was not used in [14]. The reason might be that BSIs would not be able to compress considerably due to their high density. However, one can expect that the bitmap representing the carry bit (Lines 2, 6, 13, and 20 in Algorithm 1) would be very sparse. Another highly compressible bitmap corresponds to the all-zeros bitmap used when shifting during the multiplication operation. Moreover, if the user specifies a constraint for the query with high selectivity, then all the slices could benefit from compression before computing the SUM.

Our goal in this paper is to design a space where verbatim and compressed bitmap columns can be queried together and compression is used not only to reduce space but also to speed up query time.

In order to enable this hybrid query processing, the bitmap index representation is extended with a header word. The first bit in this word is the flag bit, e, to indicate whether the bitmap column is stored verbatim ($e = 0$) or compressed ($e = 1$); and the remaining $w - 1$ bits store the number of set bits in the bitmap. The number of set bits is used to estimate the density of the resulting bitmap during query optimization as described in the next section.

Algorithm 1: Addition of BSI's. Given two BSI's, A and B, we construct a new sum BSI, S = A + B, using the following pseudo-code. We must allow the highest order slice of S to be $S[MAX(s, p) + 1]$, so that a carry from the highest bit-slice in A or B will have a place.

```
   Input: A, B
   Output: S
1  S[0] = A[0] XOR B[0] ;  // bit on in S[0] iff bit on
                                  // either A[0] or B[0]
2  CS = A[0] AND B[0] ;    // CS is "Carry" bit-slice
3  for i = 1; i <MIN(q, p); i++ ;       // While there are
                                  // bit-slices in A and B
4  do
5  |  S[i] = (A[i] XOR B[i] XOR CS) ;       // one (or
   |  three)
   |                          // bit on gives bit on in S_i
6  |  CS =Majority(A[i], B[i], C) ;      // two (or more)
   |  bits
   |                          // on gives bit on in CS
7  end
8  if q > p ;         // if A has more bit-slices than B
9  then
10 |  for i = p + 1; i ≤ q; i++ ;            // loop until
   |                                  // last bit-slice
11 |  do
12 |  |  S[i] = (A[i] XOR CS) ; // 1 bit on gives bit
   |  |  on
   |  |                       // in S[i]; CS might be ∅
13 |  |  CS = (A[i] AND CS) ;  // two bits on gives
   |  |                          // bit on in CS
14 |  end
15 end
16 else
   |  // B has at least as many bit-slices as A
17 |  for i = q + 1; i ≤ p; i++ ;         // continue loop
   |                             // until last bit-slice
18 |  do
19 |  |  S[i] = (B[i] XOR CS) ;   // one bit on gives
   |  |                          // bit on in S[i]
20 |  |  CS = (B[i] AND CS) ;  // two bits on gives
   |  |                          // bit on in CS
21 |  end
22 end
23 if CS is non-zero ;          // if still non-zero Carry
                                // after bit-slices end
24 then
25 |  S[MAX(q, p) + 1] = C ;  // put Carry into final
   |                                // bit-slice of S
26 end
```

In our hybrid space, there are three possibilities when operating two bit-vectors: both of them are compressed, both are verbatim, or one is verbatim and the other one is compressed. The first two cases (both compressed or both verbatim) present no challenge since the operations for these cases already exist. We developed query algorithms that operate an EWAH compressed bit-vector with a verbatim bitmap. We chose EWAH

as our compression scheme because its word length is equal to the computer word length (w) used for verbatim bitmaps. Thus, by using EWAH we maintain the alignment with the verbatim bitmaps.

Algorithm 2 shows the pseudo-code for a general bit-wise operation $\circ$ between two bit-vectors: one compressed and the other verbatim, and produces a result that can be either verbatim or compressed. The input bit-vectors C and V correspond to the compressed and verbatim bit-vectors, respectively. While C has a more complex data structure and requires more sophisticated access to its words, V represents a simple array and its words can be accessed directly without any decoding required. This is the reason our hybrid model is able to speed up query execution of hard-to-compress bitmaps when compared to EWAH or other bitmap compression methods, as there is no decoding overhead for bit-vector V. The size of C is always smaller than or equal to the size of V, and thus Algorithm 2 will perform maximum $|C|$ iterations for the AND operations and maximum $|V|$ iterations for the OR and XOR operations. Where $|C|$ is the size of C, and $|V|$ is the size of V in words.

Algorithm 2: Operates a compressed bit-vector C with a verbatim bit-vector V

```
   Input: C, V
   Output: R
1  pos = 0 ;
2  while C has more words do
3  |  R.appendWords(C.runningBit ∘
   |  V.sub(pos, C.runLen), C.runLen);
4  |  pos+ = C.runLen;
5  |  for i=1 to C.NofLiterals do
6  |  |  R.append(C.activeWord ∘ V[pos]);
7  |  |  pos++;
8  |  end
9  end
10 return R
```

Having defined the input data for Algorithm 2 we will now proceed to describe its steps. Because EWAH starts always with a marker word, Algorithm 2 starts by processing this word. Recall that half of the marker word stores the fill count ($runLen$ in the Algorithm) and the other half stores the number of following literals ($NofLiterals$ in the Algorithm). The fill count and the bit value for run of fills together with the corresponding words from the verbatim bitmap are passed into the `appendWords` method (Line 3). This method, depending on the fill bit value and the logical operation to be performed $\circ$, will add a stream of homogeneous bits, or a stream of values resulting from operating the fill bit with the words from V. This number of consecutive words is equal to the fill count that was just decoded. Next, the active position pos within V is updated (Line 4). Then, each of the following literal words in C (Line 5) is operated with the corresponding word in V (Line 6), and the active position within V is updated (Line 7). After the consecutive literals in C are exhausted, the next word in C will be again a marker word and Algorithm 2 iterates to Line 3 until all the words in the compressed bitmap C have been exhausted.

The append procedures (Lines 3 and 6) encode the resulting word(s) into the result bitmap R, which could be EWAH compressed or verbatim. The algorithm to decide whether the result should be compressed or not is discussed in the next section.

Note that the running time for the Hybrid algorithm described above is proportional to the size of the compressed bitmap for intersections (AND) and complement (NOT) operations, and proportional to the non-compressed bitmap for union (OR) and XOR operations.

Our goal in this work is to identify the cases where it is beneficial to operate a bit-vector in a compressed form or in a verbatim form. We shall compress if compression can improve, or does not degrade the query time. Our assumption is that if Algorithm 2 is called, the compressed bitmap is sparse (compression reduces the size considerably) and the verbatim bitmap is dense (marginal or no benefit from compression). In this case, operating a verbatim and compressed bitmap together would be more efficient than operating two-compressed or two-verbatim bitmaps together. Operating two verbatim bitmaps requires the traversal of the entire bitmap regardless of the bit-density. Thus, the proposed hybrid algorithm opens the possibility to reduce the memory requirement during query processing and speed up the query at the same time. The next section describes how this hybrid scheme can be exploited to improve the query performance over bit-vectors.

4 Hybrid Query Optimization

Let us identified the three cases for operating two bitmap columns as: VV when both of them are verbatim, CC when both of them are compressed, and as VC the hybrid case, when one of them is verbatim and the other is compressed. Regardless of the input format, the result of operating two bitmap columns can be encoded either as a verbatim bitmap or as a compressed one. The decision of whether the result should be compressed or not is made by the query optimizer as described next.

Let us denote by n the number of bits in each bitmap, and by s_i the number of set bits in bitmap i. The bit-density is then defined as $d_i = \frac{s_i}{n}$. The bitmap encoding format, verbatim (0) or compressed (1), is denoted by the flag bit e_i.

Consider the bit-vectors B_1 and B_2 with bit-densities d_1 and d_2, respectively. Algorithm 3 shows the pseudo-code for query optimization of the binary bit-wise operations. Given two bitmaps B_1 and B_2 with bit-density and stored format (d_1, e_1) and (d_2, e_2), respectively and a bit-wise operation OP $\in \{AND, XOR, OR\}$, our query optimization algorithm estimates the resulting bit-density d and decides whether the result should be compressed ($e = 1$). The density parameters $\alpha, \beta,$ andγ are used to indicate the maximum bit-density for which compressing the resulting bit-vector from AND, OR, and XOR operations, would be beneficial for subsequent operations.

Algorithm 3: Query optimization for AND, OR, and XOR bitwise operations

```
   Input: d1, d2, e1, e2, OP
   Output: d, e
 1 e = 0;
 2 if (OP == AND) then
 3 |   d = d1 d2;
 4 |   if (d < α ) || d > 1 − α) then
 5 |   |   e = 1;
 6 |   end
 7 end
 8 else if (OP == OR) then
 9 |   d = d1 + d2 − d1 d2 ;      // d=d1+d2 for equality
   |                              // bitmaps from the same attribute
10 |   if ((e1 == 1 & e2 == 1 & (d < β)||(d > (1 − β))))
   |   then
11 |   |   e = 1;
12 |   end
13 end
14 else if (OP == XOR) then
15 |   d = d1(1 − d2) + (1 − d1)d2;
16 |   if ((e1 == 1 & e2 == 1 & (d < γ)||(d > (1 − γ))))
   |   then
17 |   |   e = 1;
18 |   end
19 end
```

By default the encoding format of the result is verbatim (Line 1).

For ANDs (Lines 2-7), the result is compressed when the expected density of the resulting bitmap is smaller than α or it is larger than $1 - \alpha$. It is easy to see that for ANDs, the number of 1s in the result will always be less than or equal to the bitmap with smaller number of 1s.

For ORs (Lines 8-13), the result is compressed only when the two input bitmaps are compressed and the expected density of the result is smaller than β, or when the expected density of the result is greater than $1 - \beta$. For ORs, the number of ones in the result is at least the number of ones in the bitmap with a larger number of 1s. If either one of the bitmaps is not compressed chances are that the result should not be compressed either. The other extreme is the case when the number of set bits in the results is so high that runs of 1s can be compressed efficiently. The comparison with $1 - \beta$ allows us to compress bitmaps with a bit-density close to 1.

For XORs (Lines 14-19), the result is compressed only when the two input bitmaps are compressed and the expected density is smaller than γ or greater than $1 - \gamma$ for the same reasons discussed previously for the OR operation. OR and XOR operations exhibit the same performance behavior.

For both, OR and XOR operations, both input bit-vectors have to be compressed in order to output a compressed result. Otherwise there is an overhead to the query if the result is compressed at low bit-densities.

In the case of the NOT operation, the complement bitmap is kept in the same form as the input bitmap, i.e. if the input bitmap is compressed, the complement bitmap would also be compressed.

4.1 Bit-Density Estimation

The bit-densities of the input bitmaps are used to decide whether the result should be compressed or left as a verbatim (uncompressed) bitmap. For the indexed bitmaps we store the number of set positions and the densities are easily computed. However, for the subsequent query operations involving the resulting bitmaps, we use estimated bit-densities, as computing the actual densities can be expensive. The expected density of the resulting bitmap is estimated using the densities of the input bitmaps. In this paper we assume that the distribution of the set bits in the two input bitmaps are independent and compute the expected density using the probability of set bits in the result bitmaps. It is typical for query optimizers to make this assumption in query selectivity estimations and even when for most real data sets not all the attributes are independent, this estimation gives reasonable performance as we will show in the experimental results over real data sets.

For the NOT operation, the bit density can be computed easily since the number of set bits in the complement of bitmap B_1 would be $n - s_1$ as:

$$d_{NOT} = 1 - d_1$$

The bit-density of the bitmap resulting from ANDing the two bit-vectors can be computed as the product

of the two bit-densities. This is the probability that the same bit is set for both bit-vectors:

$$d_{AND} = d_1 \times d_2$$

Similarly, the bit-density of the bitmap resulting from ORing the two bit-vectors can be computed as the sum of the two bit-densities. This is the probability that the bit is set for either one of the bit-vectors minus the probability that the bit is set in both bit-vectors:

$$d_{OR} = d_1 + d_2 - d_1 d_2$$

This is how the expected density is computed when bitmaps from different attributes are ORed together and for all BSI encoded bitmaps. For equality encoded bitmaps, however, since only one bit is set for all the bitmaps of an attribute, $d_1 d_2$ is known to be zero and the bit-density in the case when two bitmaps from the same attribute are ORed together is estimated by:

$$d'_{OR} = d_1 + d_2$$

For XOR, the bit-density of the resulting bitmap corresponds to the probability that only one bit is set between the two bitmaps and can be estimated as:

$$d_{XOR} = d_1(1 - d_2) + (1 - d_1)d_2$$

These bit-densities are only approximations of the actual resulting bitmap vectors bit-densities, given that we assume independence between the operated bitmaps. However, in section 5 we compare the decisions made by the optimizer using the estimated density against the decisions made using the densities of the intermediate results by computing them at query time. We show that the difference is usually under 5%.

4.2 The Threshold Parameters α, β, and γ

The values for α, β, and γ should be sensitive enough to trigger the compression of the resulting bitmap without degrading the query performance.

The *append* procedure from Algorithm 2, line 6, is responsible for appending a literal or a fill to the resulting bitmap. Given the decision taken in Algorithm 3, this procedure can append to a compressed bitmap or to a verbatim bitmap. Appending to a verbatim bitmap takes constant time. However appending to a compressed bitmap is more expensive. Previous evaluation of the EWAH open source algorithm[9] , shows that *appendLiteral* and *appendFill* for EWAH use on average 3 to 4 conditional branching instructions, and takes up to 5 to 15 times longer to execute than a simple append to a non-compressed bitmap[23]. The construction of the resulting bitmap in a bit-wise operation will be proportional to the size of the input bitmaps [4]. Considering this, we shall select α, β, and γ depending on the bit-density of the output bitmaps, which can be estimated from the input vector densities. The decision about compressing the result should be considered when the input bit-vectors have a density that allows having a compression ratio between 0.2 to 0.06 or better($\frac{1}{5}$ and $\frac{1}{15}$). Further we will work with bitmap densities instead of compression ratios given that we operate with both, compressed and verbatim bitmap.

For a uniformly distributed EWAH bitmap vector x, the compression ratio can be estimated as:

$$CR_x \approx 1 - (1 - d_x)^{2w} - d_x^{2w} \quad (1)$$

where, CR_x is the compression ratio (compressed/non-compressed), d_x is the bit density for bitmap vector x, and w is the word length used for run length encoding. In this paper we use w=64 to match the computer architecture of our experimental environment. To compensate for the overhead of compressing the bit-vector, the expected compression ratio should be between 0.2 and 0.06, obtained when d is between 0.002 and 0.0005 (Equation 1). A smaller bit-vector contains fewer words, however it takes longer to decode and operate them.

Given the above reasoning, in Algorithm 3, for the unions, we choose the more conservative boundary. Thus, if the denser bit-vector has a density smaller than or equal to 0.0005, then the maximum density the result can have is 0.001 and the minimum can be 0.0005. The resulting bit-vector has the maximum possible density when none of the set-bit positions coincide in the operated bit-vectors. The minimum possible density for the result occurs when all the set-bits have the same positions for both bit-vectors being operated. Hence we pick β, and γ to be between these two values. In our evaluations we use $\beta = \gamma = 0.001$

For the intersections, if the sparser input bit-vector has a density smaller than or equal to 0.002, then the maximum probable density of the resulting bit-vector is 0.002. This is the case when both bit-vectors being operated have the same set-bit positions. Because this rarely happens, we can set a lower threshold for α and compress the result more aggressively. We choose a value between 0.002 and 0.00004, which is the probable density in the case when both bit-vectors being operated have the same density.

These values for the threshold parameters α, β, and γ are true for a 64-bit CPU. For a CPU with a different word length, these values will change, based on Equation 1. However, the steps followed in this section for determining these three parameters are the same. We later set up an experiment in section 5.2, where we

vary the input bitmap densities and execute all possible scenarios that Algorithm 3 can encounter. The goal of the experiment is to validate the limits discussed in this section for these three parameters.

4.3 The big picture

The density estimation methods for the set of basic operations supported by bitmap indices shall be sufficient for computing the bit-vector densities in more complex queries. For a bit-vector based index, a more complex query is usually composed by a cascade of basic bit-vector operations (intersections, unions, complements, exclusions). For instance, a range query may contain a series of unions, or a point query may contain several intersections.

Our hybrid compression method is designed to work with hard-to-compress bit-vectors. The bit-sliced index is a bit-vector index that is hard-to-compress because of a high set-bit density. It is usually preferred over the bitmap index when working with high cardinality domains. Some of the queries supported by the bit-sliced index are: range queries, aggregations, top-k, and term matching queries.

The more complex queries such as top-k preference queries aim to retrieve a small portion of the data, and at the same time they involve a high number of basic bit-vector operations. This means that the final result, as well as many intermediate results from the bit-wise operations, produce low density bit-vectors. This is one of the cases when the hybrid query optimizer can help speeding up the query, and reduce memory utilization, by compressing the sparse intermediate results.

In the next section we evaluate the performance of our proposed approach using synthetically generated data sets, to cover different densities and distributions, as well as real data sets. We perform top-k preference queries over bit-sliced indices to evaluate the hybrid scheme and the query optimizer over real data.

5 Experimental Evaluation

In this section we show the feasibility and efficiency of the proposed hybrid model. We first describe the experimental setup and the data sets used, which include a set of synthetic data sets as well as five real data sets. Then we evaluate the derived values for parameters α, β and γ for the different bit-wise operations. Then we show that the overhead incurred by the query optimization at run time is negligible when compared to an all verbatim or all compressed bitmap set up. Finally, we compare the performance of the Hybrid model with verbatim bitmaps, and compressed bitmaps with WAH, EWAH and Roaring Bitmaps, for point, range, and top-k queries.

5.1 Experimental Setup

The synthetic data sets were generated using two different distributions: `uniform` and `zipf`. The attribute cardinality of the generated data varies from 10 to 10,000, and the number of rows from 10,000,000 to 100,000,000 depending on the experiment. The `zipf` distribution is representative for many real-world data sets, it is widely assumed to be ubiquitous for systems where objects grow in size or are fractured through competition [24]. These processes force the majority of objects to be small and very few to be large. Income distributions are one of the oldest exemplars first noted by Pareto [25] who considered their frequencies to be distributed as a power law. City sizes, firm sizes and word frequencies [24] have also been widely used to explore the relevance of such relations while more recently, interaction phenomena associated with networks (hub traffic volumes, social contacts [26], [27]) also appear to mirror power-law like behavior. The zipf distribution generator uses a probability distribution of:

$$p(k, n, f) = \frac{1/k^f}{\sum_{i=1}^{n}(1/i^f)}$$

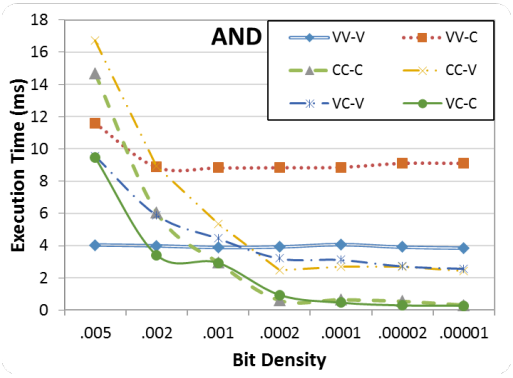
where n is the number of elements determined by cardinality, k is their rank, and the coefficient f creates an exponentially skewed distribution. We generated a few data sets for f varying from 0 to 3. Further, we varied the number of rows, number of attributes as well as their cardinality to cover a large number of possible scenarios.

The bit-sliced index does not follow the exact same distribution as the data set, however, the distribution of the data set is reflected in the cardinality of the bit-vectors. For instance in a zipf-1 data set the bit-slices with higher position index will have a lower cardinality than those on the same position index in a uniformly distributed data set.

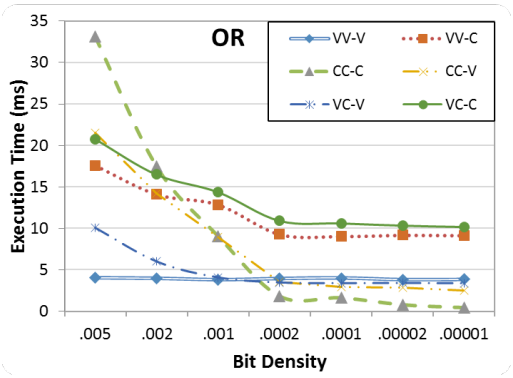
We identified that using our hybrid model over a BSI index when performing top-K queries can improve the query times as well as the memory utilization (the size of the index plus the size of the generated intermediate results) when compared to other bitmap compression systems. Thus, in this section we use a set of real data sets to support our results obtained with synthetic data. These data sets are part of the UCI Machine Learning Repository [28]:

– `Higgs.` This data-set was used in [29] and contains 11 million simulated atomic particles collision events. We use the first 21 attributes that are the kinematic properties measured by the particle detectors in the accelerator. These are all real numbers and we considered 8 significant digits.
– `Kegg.` This is the KEGG Metabolic Relation Network (Directed) data-set. It is a graph data, where Substrate and Product compounds are considered as Edges while enzyme and genes are placed as nodes. There are 53,414 tuples in this data set, and 24 attributes, with real and integer values. For the real values we considered 8 significant figures.
– `Network.` This data-set was used for The Third International Knowledge Discovery and Data Mining Tools Competition, which was held in conjunction with KDD'99. The data set contains 4,898,431 rows and 42 attributes with categorical and integer values. It contains nine weeks of raw TCP dump data for a local-area network (LAN) simulating a typical U.S. Air Force LAN.
– `Internet.` This data comes from a survey conducted by the Graphics and Visualization Unit at Georgia Tech October 10 to November 16, 1997. We use a subset of the data that provides general demographics of Internet users. It contains over 10,000 rows and 72 attributes with categorical and numeric values.
– `Poker.` In this data set each record is an example of a hand consisting of five playing cards drawn from a standard deck of 52. Each card is described using two attributes (suit and rank), for a total of 10 predictive attributes, plus one Class attribute that describes the "Poker Hand". The data set contains 1,025,010 instances and 11 attributes with categorical and numeric values.

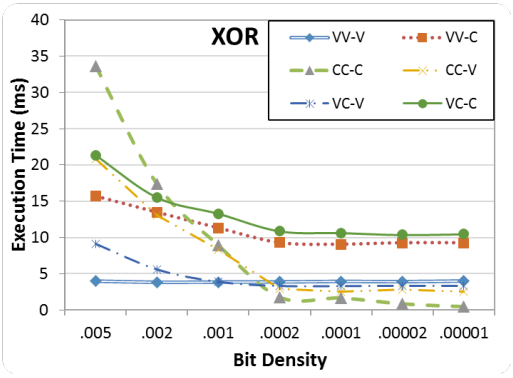
All experiments were executed over a machine with a 64-bit Intel Core i7-3770 processor and 16 GB of memory, running Linux 3.4.11-2.16. Our code was implemented in Java, and the Java version used for these experiments is 8.25 with the server mode enabled. The main reason for implementing this hybrid query optimization in Java is because we ultimately want to integrate it with the open source software stacks, that are Java based. During the measurements, the queries were executed at least 10 times, and the results for the first run were discarded to avoid measuring the just-in-time compilation. The following runs were averaged and reported.



(a) AND operation.



(b) OR operation.



(c) XOR operation.

Fig. 3: Execution time for different bit-wise operations over two bit-vectors as the bit-density of the bit-vectors decreases. Each alternative is identified with the encoding format of the operands and the result (e_1e_2-e). For example, VV-C indicates the case where both operands are verbatim and the result is stored compressed.

5.2 Setting the query optimization parameters

The query optimization parameters, α, β, and γ, refer to the maximum bit-density at which compression of the resulting bitmap would be beneficial for AND, OR, and XOR operations, respectively.

In order to find suitable values for these parameters, we randomly generated bitmap columns with varying bit densities and perform bit-wise AND, OR, and XOR operations using all possible scenarios for the input/output encoding:

- Both inputs are verbatim bitmaps and the result is stored verbatim (VV-V) or compressed (VV-C)
- Both inputs are compressed bitmaps and the result is stored verbatim (CC-V) or compressed (CC-C)
- One input is verbatim, the other is compressed, and the result is stored verbatim (VC-V) or compressed (VC-C)

The goal of this experiment is to find the values of α, β, and γ for which subsequent operations exhibit better performance than verbatim bitmaps.

Figure 3 shows the execution time for the three bit-wise operations over bitmaps with 100M tuples for all six scenarios. The densities of the input bitmaps vary from 0.005 to 10^{-5}.

Figure 3a shows the execution time of an AND operation between two columns with the given bit-density. The bit-density of the resulting bitmap is expected to be lower than the input bitmaps. However, when both input bitmaps are verbatim, there is a considerable overhead to compress the result as can be seen by the line VV-C in the figure. Therefore we pick a small enough value for α to guarantee that the overhead is justified by the performance obtained in future operations.

In the case of ORs and XORs (Figures 3b and 3c), the expected bit-density of the result is higher than the input bitmaps so β and γ are used to ensure that only bitmaps with enough low bit-density are compressed.

As seen in the figure, in all cases, at low bit-densities the execution time for the bit-wise operations involving at least one compressed bit-vector is faster than verbatim bitmaps. For an AND operation, the smallest penalty for compressing the result between two verbatim results occurs at densities smaller or equal to 0.002 for the input vectors. This makes for a density smaller or equal than 0.002, or on average 0.0004 for the result. Which confirms our pick for α in the previous section. For OR and XOR operations it is worth to maintain a compressed result only if the input bitmap densities are lower than 0.001 (at 0.001 CC-C becomes faster than CC-V). Thus the density of the result should be between 0.001 and 0.002 at this point. We choose the more conservative value and pick $\beta = \gamma$=0.001. A bit density of 0.001 means that there is one set bit for every 1,000 bits in the bitmap. For 100M, this is an average of 100K set bits per bitmap column. Further in our experiments we use $\alpha = 0.0004$ and $\beta = \gamma = 0.001$. These values are within the intervals estimated in Section 4.2, and thus validate our estimations.

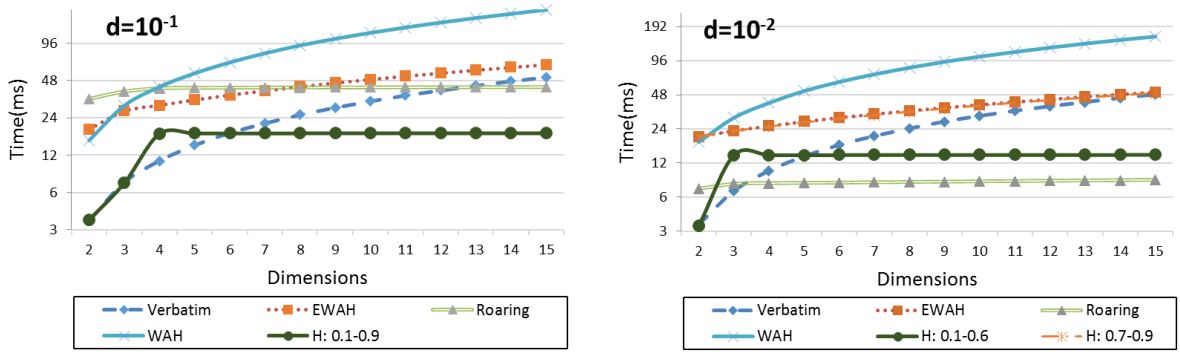
Because our hybrid model can query verbatim with compressed bitmaps, it makes sense not only to compress/decompress on-the-fly during query execution but also to start with a index that is a mix of verbatim and compressed bitmaps. The decision about whether the bitmap index should be compressed or not initially is application dependent. It is a matter of whether the trade-off between query time and memory utilization is justified. We use a threshold T to decide whether to start with a compressed or a verbatim bitmap. If the bitmap compression ratio is smaller than T, then the Hybrid starts with a compressed bitmap. Otherwise with a non-compressed bitmap.

5.3 Point queries

Point queries only involve AND operations between bitmaps from different attributes. To measure the effect of query dimensionality (i.e. number of bitmaps involved in the point query), we varied the number of bitmaps ANDed together from 2 to 15. Because these are AND operations, the density of the resulting bitmap vector decreases as the point query dimensionality grows. We perform this experiment using uniformly generated data sets, with densities varying from 10^{-1} to 10^{-2}. Each bitmap has 100 million bits. The query times are showed in Figure 4.

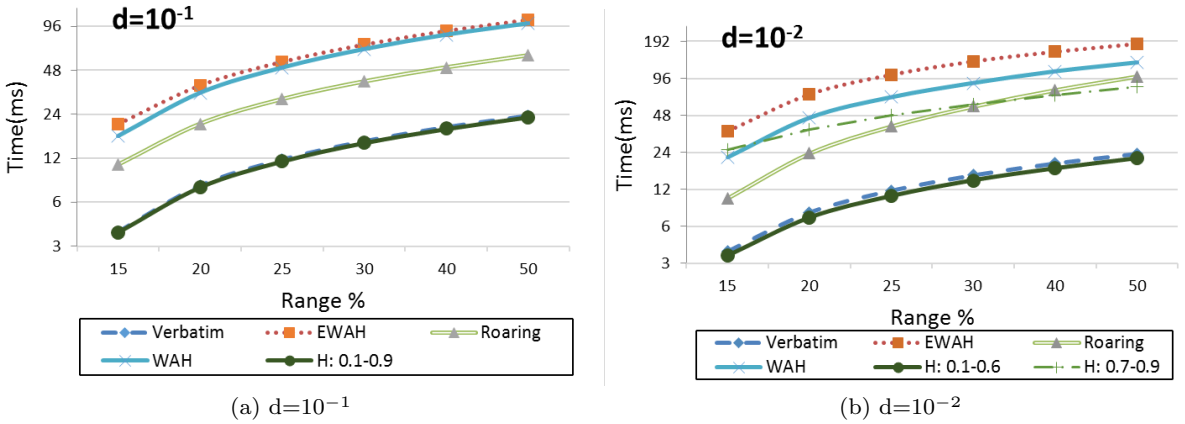
We compare five different methods in this experiment: `Verbatim`, `EWAH`, `WAH`, `Roaring bitmaps` and `Hybrid`. `Hybrid` refers to the proposed scheme, where the initial compression threshold T varies between 0.1 and 0.9 (each indexed bitmap is compressed if the compression ratio is smaller or equal to the compression threshold, e.g. H:0.1-H:0.9). In our figures, the Hybrid is denoted by H. The number following the colon sign denotes the compression threshold T, that determines whether to start with a compressed or a verbatim bitmap. Note that for the synthetic data sets all the bitmaps have the same density within an index, and thus, even if the decision is made for each bitmap individually, all the bitmaps should start with the same decision: compressed or verbatim.

Figure 4a shows the point query times for a uniform randomly generated data set with the bit density equal to 10^{-1}, as the dimensionality of the query increases from 2 to 15. Because the set-bit density is so high, WAH and EWAH cannot compress, the Roaring bitmap however achieves a compression ratio of 0.5. The Hybrid, being based on EWAH, starts with a non-compressed index regardless of the initial compression threshold selected. As figure 4a shows, the resulting



(a) d=10^{-1} (b) d=10^{-2}

Fig. 4: Performance of point queries as query dimensionality increases for different densities



(a) d=10^{-1} (b) d=10^{-2}

Fig. 5: Execution time for 1-dimensional range queries with increasing the percentage of the range queried.

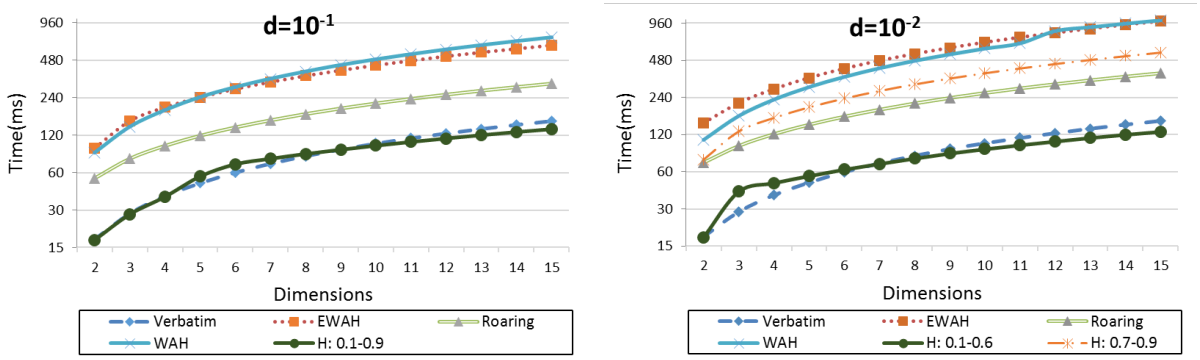
bitmap after the first three dimensions, for the Hybrid, is verbatim. After operating the fourth dimension, the result gets compressed, and the Hybrid pays a small penalty for this conversion. However as the dimensionality grows, the Hybrid operates the compressed result from the previous operation with the verbatim bitmap indexed. This way, from the fourth dimension to the 15'th, the Hybrid takes about 0.2 ms to complete the and operations, while Roaring takes 1.8, EWAH takes 34.5 and EWAH 136.4 ms. Overall for this density, the Hybrid is faster than all of the other methods we compare to, except Verbatim at 4 dimensions.

In Figure 4b the bit-density of the generated bitmap is 10^{-2}. Because the density is lower, when starting with verbatim bitmaps (H:0.1-H:0.6), the Hybrid starts to compress the result after the third dimension is being ANDed. Once the Hybrid starts compressing the result, it execute the AND operations much faster as the result becomes sparser. The time elapsed between ANDing the third and the 15'th dimension for the Hybrid was 0.16 ms, while for Roaring was 0.66 ms. If the Hybrid selects the compressed bitmaps to start with (H:0.7-H-0.9), then it performs exactly as EWAH.

5.4 Range queries

The range queries are performed, when it is necessary to retrieve all records between a lower and a higher boundary, within a bitmap index it means a cascade of OR operations between the bitmap columns of the attribute queried. Because these are OR operations, the density of the result is expected to grow with the increase in the range. We perform this experiment using uniformly generated data sets, with densities varying from 10^{-1} to 10^{-2}. Each bitmap has 100 million bits. Figure 5 shows the query times for range queries when the percentage of queried bitmap columns for the attribute varies from 10% to 50%.

For range queries, because the density of the resulting bitmap increases, the Hybrid chooses to keep the result as verbatim for higher densities. In Figure



(a) d=10^{-1} (b) d=10^{-2}

Fig. 6: Execution time for multi-dimensional range queries as the number of dimensions increases.

5a the Hybrid starts with non-compressed bimtaps and it performs similarly to the verbatim scheme. In Figure 5b if T is smaller than 0.7 then the Hybrid starts with non-compressed bitmaps, and performs similarly to Figure 5a. Otherwise the Hybrid starts with compressed bitmaps. However, because the density of the resulting bitmap increases as the range increases, the Hybrid doesn't compress the result and outperforms EWAH in terms of query time.

Range queries involving more than one attribute are called multi-dimensional range queries. This type of query is closer to a real life query scenario, involving a string of OR and AND operations. Figure 6 shows the execution time for multi-dimensional range queries as the number of attributes queried varies from 2 to 15 with a fixed percentage of the attribute range queried set to 20%. We perform this experiment using uniformly generated data sets, with densities varying from 10^{-1} to 10^{-2}. Each bitmap has 100 million bits.

The range queries lead to an increase in density for the intermediate result, however the AND operations then decrease their density. Because of that, after a few dimensions, the Hybrid is able to compress the intermediate result and perform AND operations using Algorithm 2. The compression of the intermediate results helps the Hybrid to outperform the verbatim scheme as the dimensionality grows in Figure 6 (when starting with non-compressed bitmaps). When starting with compressed bitmaps, the Hybrid performs similarity to EWAH, or better.

5.5 Top-k queries

As the previous experiments show, the Hybrid scheme competes well with the existing bitmap compression methods. However it outperforms all of them, including the verbatim (no- compression) scheme for high density bitmaps. With this in mind, we identified that applying our Hybrid compression to a bit-sliced index can benefit its applications in terms of query time even when compared to a non-compressed bit-sliced index. Moreover, in terms of memory utilization, the Hybrid is close to the performance exhibited by EWAH while being up to an order of magnitude faster than EWAH.

Top-K or preference top-K queries over a bit-sliced index is an important application that can benefit from our Hybrid scheme. Top-K queries are performed over a BSI index as defined in [14]. In a Top-K query where all attributes are weighted equally, the values are added together and tuples with the top k values are reported as the query answers as showed in Algorithm 1 and then the top-K algorithm described in [14]. Depending on the number of slices per attribute, Algorithm 1 can perform a number of AND, XOR and OR operations (at least 4-XOR, 5-AND, and 2-OR operations per added attribute). The top-K algorithm performs 2-AND, 1-OR and 1-NOT operation for each slice in the aggregated attribute. In the following experiments we computed the top-20. However as shown in [14], k does not have a significant impact on the algorithm running time, since it operates with bit-slices.

In the following experiments we use three synthetically generated data sets and also real data sets, to cover a large enough range of distributions and densities. We present the results in terms of top-k query time and memory utilization. We focus on memory utilization rather than on the index size because the BSI index is usually a high-density bitmap index and hard to compress. However, because the bit-slices are operated multiple times, the intermediate results can become sparser and there are opportunities for compression. We also add for comparison a naïve Hybrid method that compresses the intermediate result if it can achieve a com-

pression ratio of 0.5 or better. This is denoted by H-0.5 in our figures. For the Hybrid method (H) we set the initial threshold T to 0.5. However this is application dependent and can be changed depending on the need for space or faster queries.

Figure 7 shows the results in terms of execution time and memory utilization over synthetic data when top k queries are executed over BSI indices. The data sets contain 5 attributes with normalized values with 6 decimal positions, 10 million rows, and generated using three different distributions: uniform, zipf-1 (f=1) and zipf-3 (f=3).

For the uniform data-set (Figure 7a), The index had 100 slices, an average of 20 slices per attribute. Only the Roaring bitmap was able to compress the initial index, with a compression ratio of 0.64, however in terms of Top-K query times it was two times slower than the Hybrid. The Hybrid scheme, also was faster than the verbatim by about 2%. In terms of memory utilization, the Hybrid performed similarly to WAH and EWAH, however it was several times faster (3.6x faster than EWAH and 6.6x faster than WAH).

With the zipf-1 skewed data set, results showed in Figure 7b, the results were similar to those Figure 7a.

For the zipf distribution with f=3 (Figure 7c), the skew in the data is significantly higher than in zip-1 and thus the bit-slices are highly compressible. Having the compression threshold T set to 0.5, the Hybrid started with 79 compressed bit-slices out of 100. As shown in Figure 7c, even if the Verbatim scheme has a very forthright query algorithm without requiring any decoding/encoding, the execution time for the top-K queries is 2 times higher than the Hybrid scheme. This again underlines the potential of compression not just for storage-space purposes but also for speeding up processing. At the same time, The Hybrid was 2.8 times faster than the Roaring bitmap, even if the Roaring bitmap compressed the bit-slices better. The Hybrid scheme also used only 5% more memory than WAH and EWAH and was 4.3 times faster than WAH and 3 times faster than EWAH.

Furthermore, to show that our synthetically generated data-sets accurately represent the performance gains that can be obtained over real-world data, we perform top-K queries over the real data-sets described at the beginning of this section. The results in terms of of execution time and memory utilization for the real data-sets are shown in Figure 8.

For the `Higgs` data set, the bit-sliced index contains 607 bit-slices(an average of 29 slices per attribute). The BSI-sum algorithm (Algorithm 1), together with the top-K algorithm performed 234 XOR operations, 345 AND operations and 244 OR operations for this data set. From a total of 823 logical operations executed, 131 of their results were compressed and 692 were not. The Hybrid scheme was 5% faster and used 7% less memory than the Verbatim scheme, was 14% faster and used 33% more memory than the Roaring bitmap. When compared to WAH and EWAH, the Hybrid scheme, was 5 times faster than EWAH and an order of magnitude faster than WAH, while using under 1% more memory than these two schemes.

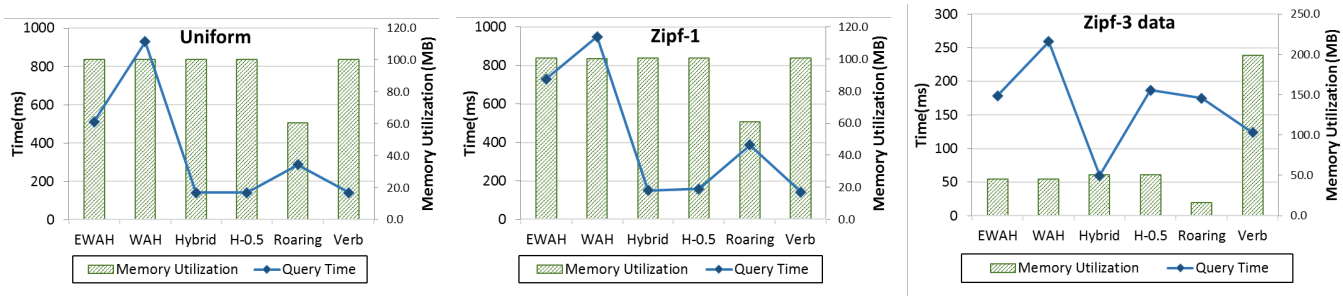
The bit-sliced index for the `Kegg` data set has 243 bit-slices, and the Hybrid compressed 44 of them, to start with. From a total of 252 logical operations performed, the Hybrid compressed 26 of their results, and 226 where not compressed. The BSI-sum algorithm (Algorithm 1), together with the top-K algorithm performed 86 XOR operations, 110 AND operations and 68 OR operations for this data set. In terms of query time, the Hybrid outperformed all the other schemes. It was 5% faster than Verbatim, 3 times faster than Roaring, 1.7 times faster than H-0.5, 8.2 times faster than WAH, and 7.5 times faster than EWAH.

The `network` data-set is in a way similar to the zipf-3 data, its values having a higher skewing factor. The bit-sliced index for the `network` has 207 bit-slices, and the Hybrid compressed all of them, to start with. From a total of 311 logical operations performed, the Hybrid compressed 197 of their results, and 114 where not compressed. The BSI-sum algorithm (Algorithm 1), together with the top-K algorithm performed 104 XOR operations, 139 AND operations and 68 OR operations for this data set.

5.6 Evaluation of the optimizer

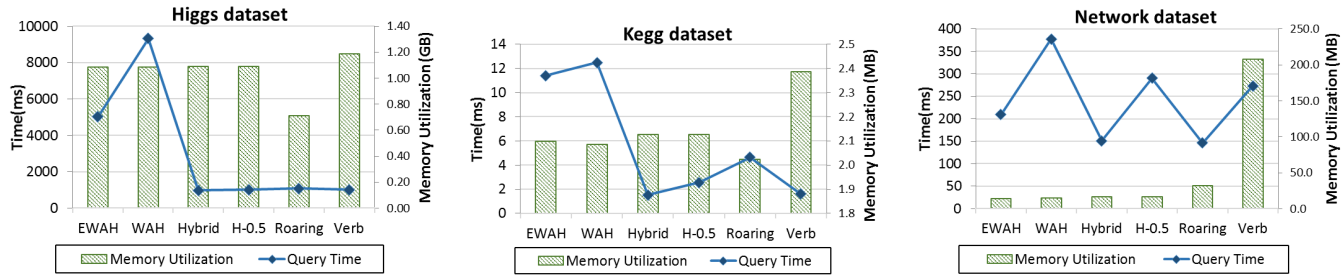
To measure the overhead of our query optimizer we set-up a scenario where the hybrid model should have exactly the same running time (if neglecting the overhead imposed by the optimizer) as when operating with only verbatim bitmaps or only compressed bitmaps. Multi-dimensional queries involving a variable number of bitmaps (between 2 and 10) were executed for AND, OR, and XOR operations and the results are shown in Figure 9.

For the AND queries we randomly generated uniform distributed low density bitmaps ($d = 10^{-4}$) with 100M bits. Since these bitmaps are highly compressible, the hybrid model starts only with compressed(EWAH) bitmaps. Because the AND optimization keeps the result compressed when the input is compressed, and the density is below the threshold, our optimizer always keeps the result as a compressed bitmap. We measured the time required by our hybrid model to execute 45 AND queries and run the same set of queries using only



(a) Uniform distribution (b) Zipf distribution ($f = 1$) (c) Zipf distribution ($f = 3$)

Fig. 7: TopK queries over synthetic data sets with 5 attributes, 10M rows. Each attribute is represented by a BSI with 20 slices (normalized values with 6 decimal positions).



(a) Higgs data set (b) Kegg data set (c) Network data set

Fig. 8: TopK queries over real data sets.

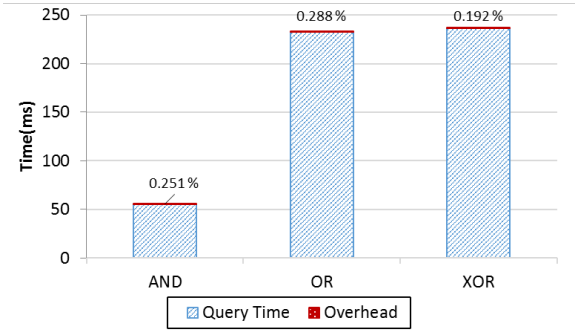


Fig. 9: The query overhead created when the optimizer cannot improve query time

compressed bitmaps. The difference is reported as the optimization overhead.

Following the same methodology, for OR and XOR operations we generated uniformly distributed high-density bitmaps ($d \approx 0.05$) with 100M bits and present the cumulative query time of 45 queries. Because when performing OR and XOR operations over these bitmaps the results are not compressible, our query optimizer keeps the results as verbatim. We ran the same set of queries over verbatim bitmaps and report the difference as the optimization overhead.

As Figure 9 shows, the overhead that the query optimizer brings is very modest (around 0.2%). It is worth noting that this is a worse case scenario where the optimizer cannot improve the query time over the non-hybrid model. As we will see in the next experiments, the overhead is more than justified by the overall improvement in query time and memory utilization.

To evaluate the effect of our bit-vector independence assumption, we compare the decisions made by the optimizer using the estimated density against the decisions made using the densities of the intermediate results by computing them at query time. Then we count the number of times the estimations led to a different decision than the actual computed densities. Table 2 shows the number of mismatches for real data sets when performing the top-k queries.

One could argue that simply computing the bit-vectors density will never produce errors, and could offset the computing time by always taking the "right" decision, and producing faster queries. To verify if this is the case, we set to measure the top-k query times for six data sets (3 synthetically generated, and 3 real), one of which has high attribute correlation. The average

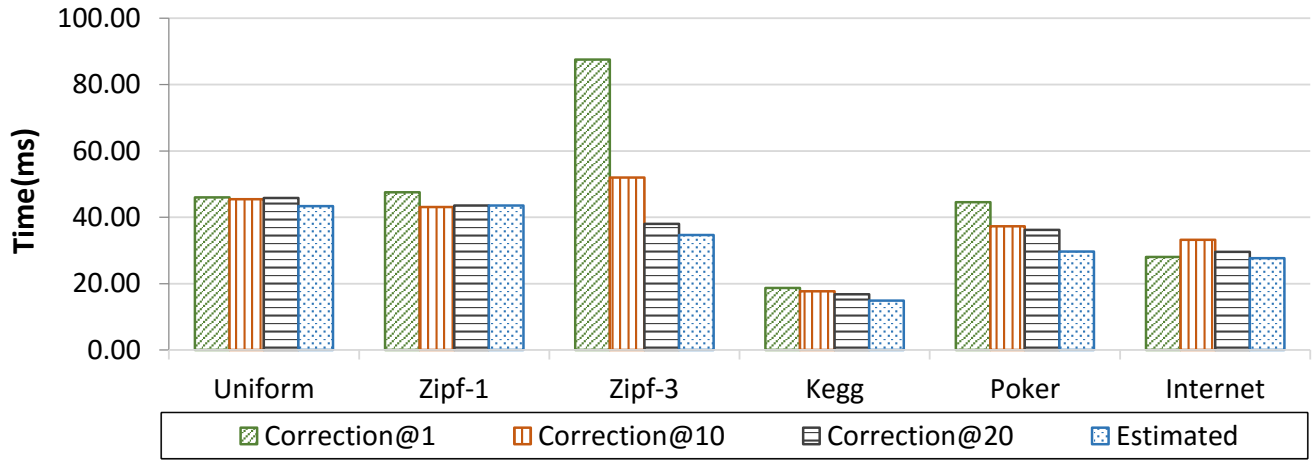


Fig. 10: Top-K query times for data sets with average attribute correlation: [Uniform=0.0002]; [Zipf-1=0.0003]; [Zipf-3=0.00032]; [Keg=0.26]; [Poker=0.02]; [Internet=0.024]

Table 2: Percentage of mismatch optimization decisions when using estimated vs. measured density for the intermediate results

Data set	HIGGS	Kegg	Network
Total operations	823	252	311
Number of mismatches	4	8	18
Mismatch %	0.48	3.2	5.8

attribute correlation for the `Kegg` data set is 0.26, however the correlation between some attributes is as high as 0.7. Figure 10 shows the top-k query times for the six data sets. The `Uniform`, `Zipf-1`, and `Zipf-3` are synthetically generated data sets, while `Kegg`, `Poker`, and `Internet` are real data sets. The average attribute correlation is provided in the figure description. In this figure we compare the query time when using density estimation, and when computing the bit-vector density. We also measure the query time for a combination between the two approaches to determine the density of the resulting bit-vector. `Correction@10` means that if one of the bit-vectors being operated has its density resulted from previous 9 bit-wise operations using estimation, then the 10^{th} bit-wise operation will use a scan and compute. For `Correction@20` the density calculation is used every 20^{th} operation. We use this technique for diminishing the effects of density estimation error propagation. Even if we make use of the POPCNT CPU instruction when computing the bit-vectors cardinality, the estimation still outperforms the other approaches, even for highly correlated data sets, where there may be a higher density estimation error.

To conclude, in this section, performed a series of experiments and showed that the proposed Hybrid compression and query optimization scheme, can be efficient over hard-to-compress bit-vectors. While the overhead added by the query optimizer is very low, it opens the possibility to exploit compression for the intermediate results, even when the initial index is not compressible. This often translates in faster queries and less memory utilization during query processing.

6 Conclusion

We have introduced a novel algorithm to perform bitwise operations over compressed and verbatim bitvectors. The proposed hybrid model exploits the fast query execution over verbatim bitmaps avoiding the decoding overhead of hard-to-compress bitmaps and still is able to reduce the memory utilization by compressing the sparse bitmaps. Furthermore, the bitmaps are only compressed when the use of compression will further improve execution time of subsequent operations.

With the proposed technique, we are able to compress and decompress the bitmaps at run time using the estimated density as a performance predictor. We show that the hybrid approach always outperforms the use of verbatim only bit-vector in terms of both, query time and memory utilization. At the same time, we show the scenarios when this model can outperform compressed only bitmaps. For the top-K queries performed over real data-sets we achieved up to an order of magnitude faster queries when compared to the run-length compressed schemes, and up to 11 times less memory used when compared to the verbatim scheme.

Future work involves the evaluation of the hybrid model over other types of queries for which the usage

of bit-vectors and bitmap indices have been proposed, such as skyline queries and term matching queries [30].

References

1. G. Antoshenkov, Byte-aligned bitmap compression, in: DCC '95: Proceedings of the Conference on Data Compression, IEEE Computer Society, Washington, DC, USA, 1995, p. 476.
2. K. Wu, E. J. Otoo, A.Shoshani, Compressing bitmap indexes for faster search operations, in: Proceedings of the 2002 International Conference on Scientific and Statistical Database Management Conference (SSDBM'02), 2002, pp. 99–108.
3. F. Deliege, T. Pederson, Position list word aligned hybrid: Optimizing space and performance for compressed bitmaps, in: Proceedings of the 2010 International Conference on Extending Database Technology (EDBT'10), 2010, pp. 228–239.
4. K. Wu, E. J. Otoo, A. Shoshani, H. Nordberg, Notes on design and implementation of compressed bit vectors, Tech. Rep. LBNL/PUB-3161, Lawrence Berkeley National Laboratory (2001).
5. A. Colantonio, R. Di Pietro, Concise: Compressed 'n' composable integer set, Information Processing Letters 110 (16) (2010) 644–650.
6. F. Fusco, M. P. Stoecklin, M. Vlachos, Net-fli: On-the-fly compression, archiving and indexing of streaming network traffic, Proceedings of the VLDB Endowment 3 (2) (2010) 1382–1393.
7. G. Guzun, G. Canahuate, D. Chiu, J. Sawin, A tunable compression framework for bitmap indices, in: Data Engineering (ICDE), 2014 IEEE 30th International Conference on, IEEE, 2014, pp. 484–495.
8. K. Wu, E. J. Otoo, A. Shoshani, A performance comparison of bitmap indexes, in: CIKM 2001, 2001, pp. 559–561.
9. D. Lemire, O. Kaser, K. Aouiche, Sorting improves word-aligned bitmap indexes, Data and Knowledge Engineering 69 (2010) 3–28.
10. S. Chambi, D. Lemire, O. Kaser, R. Godin, Better bitmap performance with roaring bitmaps, arXiv preprint arXiv:1402.6407.
11. K. Wu, E. J. Otoo, A. Shoshani, Optimizing bitmap indices with efficient compression, ACM Trans. Database Syst. 31 (1) (2006) 1–38. doi:10.1145/1132863.1132864.
    URL `http://doi.acm.org/10.1145/1132863.1132864`
12. P. O'Neil, D. Quass, Improved query performance with variant indexes, in: ACM Sigmod Record, Vol. 26, ACM, 1997, pp. 38–49.
13. D. Rinfret, Answering preference queries with bit-sliced index arithmetic, in: Proceedings of the 2008 C3S2E Conference, C3S2E '08, ACM, New York, NY, USA, 2008, pp. 173–185. doi:10.1145/1370256.1370286.
    URL `http://doi.acm.org/10.1145/1370256.1370286`
14. G. Guzun, J. Tosado, G. Canahuate, Slicing the dimensionality: Top-k query processing for high-dimensional spaces, TLDKS 14.
15. P. O'Neil, D. Quass, Improved query performance with variant indexes, in: Proceedings of the 1997 ACM SIGMOD international conference on Management of data, ACM Press, 1997, pp. 38–49. doi:http://doi.acm.org/10.1145/253260.253268.
16. C.-Y. Chan, Y. E. Ioannidis, An efficient bitmap encoding scheme for selection queries, in: Proceedings of the 1999 ACM SIGMOD international conference on Management of data, SIGMOD '99, ACM, New York, NY, USA, 1999, pp. 215–226. doi:http://doi.acm.org/10.1145/304182.304201.
    URL `http://doi.acm.org/10.1145/304182.304201`
17. N. Koudas, Space efficient bitmap indexing, in: Proceedings of the Ninth International Conference on Information and Knowledge Management, CIKM '00, ACM, New York, NY, USA, 2000, pp. 194–201. doi:10.1145/354756.354819.
    URL `http://doi.acm.org/10.1145/354756.354819`
18. D. Rinfret, P. O'Neil, E. O'Neil, Bit-sliced index arithmetic, SIGMOD Rec. 30 (2) (2001) 47–57. doi:http://doi.acm.org/10.1145/376284.375669.
19. M.-C. Wu, A. P. Buchmann, Encoded bitmap indexing for data warehouses, in: ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering, IEEE Computer Society, Washington, DC, USA, 1998, pp. 220–230.
20. D. C. Fabian Corrales, J. Sawin, Variable length compresssion for bitmap indices, in: ACM International Conference on Database and Expert Systems Applications, 2011, pp. 381–395.
21. S. J. van Schaik, O. de Moor, A memory efficient reachability data structure through bit vector compression, in: ACM SIGMOD International Conference on Management of Data, 2011, pp. 913–924.
22. P. Lu, S. Wu, L. Shou, K.-L. Tan, An efficient and compact indexing scheme for large-scale data store, in: Data Engineering (ICDE), 2013 IEEE 29th International Conference on, IEEE, 2013, pp. 326–337.
23. G. Guzun, G. Canahuate, Performance evaluation of word-aligned compression methods for

bitmap indices, Knowledge and Information Systems (2015) 1–28.
24. A. Clauset, C. R. Shalizi, M. E. J. Newman, Power-law distributions in empirical data, 2009. doi:10.1137/ 070710111.
URL http://dx.doi.org/10.1137/070710111
25. V. Pareto, Manual of political economy (1906).
26. A. lászló Barabási, R. Albert, Emergence of scaling in random networks, Science.
27. A.-L. Barabasi, The origin of bursts and heavy tails in human dynamics, Nature 435 (2005) 207.
URL http://www.citebase.org/abstract?id=oai:arXiv.org:cond-mat/0505371
28. M. Lichman, UCI machine learning repository (2013).
URL http://archive.ics.uci.edu/ml
29. P. Baldi, P. Sadowski, D. Whiteson, Searching for exotic particles in high-energy physics with deep learning, Nature Commun 5.
30. D. Rinfret, Term matching and bit-sliced index arithmetic, Ph.D. thesis, University of Massachusetts Boston (2002).