Improving Bitmap Execution Performance Using Column-Based Metadata

Miguel Velez Jason Sawin
Department of Computer and Information Sciences
University of St. Thomas
St. Paul, MN. USA

Alexia Ingerson David Chiu
Department of Mathematics and Computer Science
University of Puget Sound
Tacoma, WA. USA

***Abstract*—Many data-intensive business and research applications rely on advanced database indexing techniques to ensure efficiency. A popular approach for read-only data sets is *bitmap indices*. They use fast machine bitwise operations when querying and are highly compressible. A commonly used compression technique used for bitmaps is Word-Aligned Hybrid (WAH). WAH is a run-length encoding scheme that greatly compresses sparse bitmaps and can directly query data without explicit decompression. In this paper, we present an algorithm that uses metadata, gathered at compression-time, to enable logical short-circuiting in WAH's query algorithm, thus reducing the number of required memory accesses. We also present a heuristic that identifies queries where the processing overhead of our algorithm will not likely be off-set by the number of eliminated memory accesses. In these cases, it is advantageous to process the query using the standard WAH algorithm. The results of our empirical study over both real and synthetic data sets show that our approach can realize speedups of average query times ranging from $1.3\times$ to $+84\times$ when compared to WAH. They also showed that the use of our heuristic reduced the number of times our approach underperformed WAH.**

I. Introduction

Scientific and analytical applications are data-intensive, requiring fast access to large data archives. Many indexing techniques continue to be developed to allow high-performing data access and query processing. Among these, the bitmap index [1], has experienced a resurgence in recent popularity in business intelligence and scientific applications [2,3,4,5,6, 7,8,9] due to two main factors. First, bitmap indices perform query processing using bitwise logical operations which are CPU-supported primitives and therefore fast. Second, due to the indices' sparseness, bitmap-compression techniques have yielded high compression ratios while still supporting fast logical operations.

Like most database indices, a bitmap is also a coarse summary of a relation. The attributes of the relation are first discretized into either distinct value-ranges or single value attributes. For each tuple's attribute, a 1 is used to indicate that the attribute applies to it. Conversely, when the attribute does not apply to the tuple, it is represented with a 0. More formally, a bitmap B is an $m \times n$ array where the n columns represent bins, and the m rows of the bitmap correspond to the m tuples in the indexed relation. A bit $b_{i,j} \in B = 1$, if the ith tuple falls into the specified value or range of the jth bin, and $b_{i,j} = 0$, otherwise.

Tuples	Bins								
	X				Y				
	x_0	x_1	...	x_k	y_0	y_1	y_2	y_3	y_4
t_1	0	1	...	0	0	0	1	0	0
t_2	0	0	...	0	0	1	0	0	0
t_3	0	0	...	1	0	0	1	0	0
...	...	...	...	...	...	...	...	...	...

TABLE I: An Example Bitmap

Suppose we know that the values of X are discrete integers uniformly distributed in the range $[0, k]$ for some constant $k > 0$. If k is relatively small, we can generate a bin x_j for every value of X, otherwise, we can generate bins for some value-ranges of X. Suppose we also know that the values of Y can be any real number, and its distribution is heavily skewed toward values near either side of 0. Because the values of Y are continuous and unbounded, its values must be discretized into ranges based on its distribution, and we correspond each range to a bin. Because we know that Y tends to oscillate around 0, we chose to use five bins for the example shown in Table I:

$$y_0 = (-\infty, -1]$$
$$y_1 = (-1, 0)$$
$$y_2 = 0$$
$$y_3 = (0, 1)$$
$$y_4 = [1, \infty)$$

Suppose we want to retrieve all tuples where $X < \gamma$ and $-0.5 \leq Y \leq 0$. The query processor can find the candidates by evaluating $v_{res} = (x_0 \vee \ldots \vee x_\gamma) \ \wedge \ (y_1 \vee y_2)$, which corresponds to the following set:

$$res = \{t \mid (t[X] < \gamma) \wedge (-1 < t[Y] < 1)\}$$

There may be false positives in res, which requires a bit more pruning, but only the tuples $t_i \in res$ with a corresponding bit $v_{res}[i] = 1$ must be retrieved from disk and examined to ensure they meet the selection criteria. All records r_j with a corresponding bit $v_{res}[j] = 0$ are pruned immediately, avoiding disk access.

Intuitively from this small example, "wider" bitmaps (*i.e.*, with high bin cardinality) correspond to higher selectivity. Highly selective bitmaps are usually quite sparse (with a

majority of 0-bits), which lends well to compression. Because query processing applies logical operations over multiple bit-vectors, the compression must occur at the granularity of a bit-vector. However, common compression techniques (gzip, 7-zip, etc.) cannot be applied in this context, because the inflation overhead would drastically slow-down query processing rates.

Therefore, bitmap-compression techniques must allow query processing to occur, without inflation, over the compressed bit-vectors. Among the most widely-used technique is the *word-aligned hybrid* code (WAH) [10]. It is the basis for other known techniques [5,11,12,13]. A detailed description of WAH will be given in Section II, but generally, it encodes a bit-vector into a sequence of words (32 or 64 bits, depending on the architecture). These words can hold either compressed run-length data or uncompressed data. To perform query processing, the logical operator is applied between two WAH words at a time, resulting in a time complexity that is linear to the number of encoded words.

A. Contributions

We observe that opportunities exist for optimizing query processing over compressed bitmaps through short-circuiting logical operations. For instance, boolean logic informs us that applying an AND between a 0 and x produces a 0, and therefore, if we had *a priori* knowledge of the contents of each bit-vector, then some words would not be required to be retrieved from memory, saving valuable time for query processing. We propose a framework that captures such *metadata* for compressed bitmap indices, and applies them to accelerate query processing.

This paper makes the following contributions:

- We present our Meta+WAH algorithm which integrates metadata into WAH's query processing to enable short-circuiting.
- Due to processing overhead, the short-circuiting algorithm will not be the most efficient approach for all queries. We present a hybrid approach that uses a heuristic to select between executing a query using either WAH or Meta+WAH.
- We provide a detailed comparison of query-processing performance with and without the use of metadata using several synthetic and real scientific data sets. The results show that Meta+WAH can realize average query speedups of $1.3\times$ to $+84\times$. Our Hybrid approach realized similar speedups but underperformed WAH fewer times.

The remainder of the paper is organized as follows. In Section II we provide an overview of the WAH compression algorithm and discuss row-orderings for bitmaps. Section III presents our algorithm of Meta+WAH and describes the heuristic we used to create the hybrid approach. Section IV presents the results of our empirical study. Related works are presented in Section V. We conclude and present future avenues for this work in Section VI.

II. Background

Depending on the number and cardinality of a relation's attributes, the bitmaps themselves can become too large to be stored in main memory. This section provides background on the compression and query processing mechanisms for the WAH bitmap-compression schemes.

A. Word-Aligned Hybrid Code (WAH)

The Word-Align Hybrid (WAH) is the most widely used bitmap-compression algorithm. The core encoding unit in WAH is a *word* (commonly 32 or 64 bits in today's architectures)—chosen for performance: words are the basic unit of transfer between cache memories and registers.

Let w denote a machine's word size. WAH defines just two types of words: *literal* and *fill*. Literals are encoded in the form $(flag, litval)$, where $flag = 0$ is the most-significant bit, with the bit-value of 0 signifying that the next $w-1$ bits representing $litval$ is an uncompressed bit sequence. Literals are used if the data is noisy, interspersed with both 0s and 1s. The other type is a fill word. It is encoded using the following triple $(flag, val, len)$, where $flag = 1$ is again the most-significant bit, with the bit-value of 1 signifying that the remaining $w-1$ bits encode a run-length of consecutive bits of value val. Given a run of r consecutive homogeneous bits, then $len = \lfloor r/(w-1) \rfloor$ encodes the number of consecutive WAH words containing val bits.

Consider the following example. Suppose we wish to compress a bit-vector containing the following sequence: the first $w-1$ bits alternate between 1 and 0, followed by a sequence of $(w-1) \times 4M$ zeroes, followed by another sequence of alternating $w-1$ bits. On a 64-bit machine, without compression, this vector requires over 2 million words of storage. Encoded in WAH, it requires just three words: $(0, 1010...01)$, $(1, 0, 2^{22}{}_{<2>})$, and $(0, 1010...01)$, where $x_{<2>}$ is the binary representation of x.

The bitwise logical operations of queries can be directly applied to WAH compressed bit-vectors. Consider the case of $A \wedge B$ where A and B are both WAH bit-vectors. The result of this query is a compressed bit-vector C which when uncompressed is the result of uncompressed $A\wedge$ uncompressed B. WAH linearly processes queries by analyzing word pairs, one word from each bit-vector operand. When a word has been fully processed, or exhausted, the next word from that column is retrieved. There are three possible pairing of operand word types:

1) **literal** $\wedge$ **literal**: In this case, a new $result$ literal word is added to C where $C.result.litval = A.operand.litval \wedge B.operand.litval$.
2) **fill** $\wedge$ **fill**: This case results in a new $result$ fill being added to C where $C.result.val = A.operand.val \wedge B.operand.val$ and $C.result.len = Min(A.operand.len, B.operand.len)$.
3) **fill** $\wedge$ **literal**: In this case, a new $result$ literal word is added to C. If $A.operand.val$ is 1 then $C.result.litval = B.operand.litval$ else $C.result.litval = 0$.

In cases 2 and 3, there is additional bookkeeping to track the number of words processed in each fill. This can be thought of as simply subtracting the number of processed words from the fill length of the fill words.

B. Row-Ordering Bitmaps

Previous studies have shown that reordering the rows of a bitmap can improve compression and query performance [14,15,16]. Finding the row ordering that produces the optimal total run length for a bitmap has been proven to be NP-Complete. Lexicographical and Gray code orderings are common heuristics. Figure 1 shows the effects of lexicographical and Gray code ordering on bitmaps containing 3 vectors v1, v2, v3. The white space represents 0s and the black represents 1s. Notice that both reordering algorithms tend to produce longer runs in the first few bit vectors, but deteriorate into shorter runs (and worse, a random distribution) of bits for the higher vectors. As can be seen, if low-order columns are queried with high-order columns there are numerous opportunities for short-circuiting.

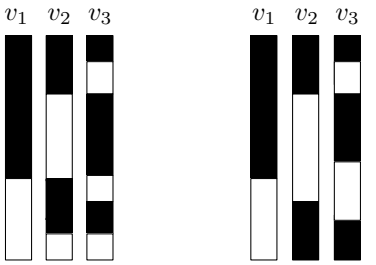


Fig. 1: Graphical representation of row ordering.

III. Enabling Short-Circuiting

As described above, traditionally processing the query of two compressed bit-vectors is accomplished by linearly comparing matched words from each of the vectors. In this manner, the process ensures that it is only comparing values in the same tuple. However, this approach does not take advantage of logical short-circuiting.

For an example of the benefits of short-circuiting, consider the query shown in Figure 2. Column X consists of a fill word which represents a run of 310 words of zeros and a literal word. X is being logically ANDed with column Y, which consists of 311 literal words. If WAH were to process this query, it would read the first word of X from memory. Even though the result is going to be 310 words of 0's, it would then read in 310 literals from Y, one at a time, and apply the logical operation each time until it had exhausted the run of X. WAH has to do this because it does not have advanced knowledge of the composition of Y. If it somehow knew that Y consisted of all literals, it could immediately write the result of 310 words of 0 and skip reading all but the last word of Y. By short-circuiting, WAH could skip 310 memory accesses.

To effectively short-circuit, the algorithm must have knowledge of the column structures. This information could come

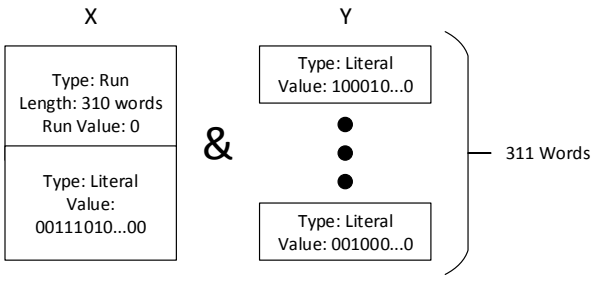


Fig. 2: Bitmap AND query example.

in several forms. For example, knowing on which bitmap row every 10th word in the compressed vector starts would allow the algorithm to skip up to ten memory accesses at a time. This type of structural information can be gathered at compression-time and stored in metadata files to be accessed at query-time. For this approach to be practical, the metadata being used must not significantly add to the space overhead of the compressed bitmap. It also should not significantly add to the processing overhead of the query algorithm as this would degrade any possible speedup.

For this paper, we are considering a metadata that records the number of literals following each fill word. This information is represented as a list. The first number in the list represents the number of leading literals in the corresponding column (0 if the columns first word is a fill). The second number in the list represents the number of literals following the first fill and so on. This information is collected for each compressed column and stored in a separate file. This form of metadata was selected because it requires only a slight modification to WAH's querying algorithm to enable short-circuiting, it is relatively compact, and it will be shown to perform well with Gray code ordered data sets.

Algorithm 1 presents a WAH bitwise AND operation that incorporates logical short-circuiting using the metadata lists described above. We are calling this approach **Meta+WAH**. There are four inputs to the operation: two compressed bit vectors A and B, and their corresponding metadata lists. The operation returns a compressed bit vector Z which is the product of $A \wedge B$. The algorithm continues as long as it has not exhausted both of the columns (line 1). Each iteration of the algorithms tests the current word of each column. If a word has been exhausted, the next word is retrieved (lines 2-13). If the newly loaded $currentWord$ is a fill, then the number of literals that follows it is fetched from the metadata list (lines 4-6 and 10-12).

Next, the $currentWord$s are processed (lines 14-45). If both $currentWord$s are fills, then the length of the shorter run is stored in $nWords$ (lines 15-16). A fill of length $nWords$ is added to the result column Z. The value of this new fill is the result of the logical AND operation being applied to columns A's and B's $currentWord$s fill values (lines 17-18). The fill lengths of both A's and B's $currentWord$ss are updated to reflect that $nWords$ have been processed (lines 19-20). This will exhaust the shorter fill causing that $currentWord$ to be

updated on the next iteration of the algorithm.

Short circuiting can take place if one of the $currentWord$s is a fill of 0s and the other is a literal (lines 23-38). Consider the case where column A's $currentWord$ is a fill of 0s and columns B's is a literal (line 19-27). In this situation, the metadata for the column B is consulted. It contains the count of literals until the next fill word in that column. The minimum of A's run length and the number of literals returned from the metadata is stored in $nJumpLen$ (lines 24-25). A fill of 0s with length $nJumpLen$ is added to Z (line 26) and the length of A's fill and B's metadata are modified to represent that $nJumpLen$ words have been processed (lines 27-28). Finally, column B's $currentWord$ is jumped $nJumpLen$ words.

If both of the $currentWords$ are literals, a bitwise AND is performed on their values and a literal is added to Z (Lines 40-41). In this case, both of the columns A's and B's metadata are updated to represent that the current word is now one literal closer to the next fill. After the algorithm has processed all the words in both columns it returns Z (line 47).

A. Hybrid Approach

As shown above, the approach to short-circuiting has a certain overhead expense associated with it. Accessing the metadata requires memory reads and determining if short-circuiting is possible requires additional branching. Short-circuiting is only beneficial if enough words can be circumvented to recoup the cost of the additional processing.

Though short-circuiting might be efficient for a significant number or even a majority of queries, it will not be well suited for all queries. In some cases, it will be more efficient to use the original WAH query algorithm. To this end, we have developed a lightweight heuristic which can be used to build a hybrid Meta+WAH\WAH approach. The heuristic is:

$$Algo = \begin{cases} 0, & \text{if } \left| \frac{\#Literal_{col1} - \#Literal_{col2}}{\#Words_{col1} + \#Words_{col2}} \right| \geq \Delta \\ 1, & \text{otherwise} \end{cases} \quad (1)$$

For two columns being ANDed together, the heuristic computes the difference between the number of literals. It then normalizes that result to the total number of words being processed. If the absolute value of the normalized result is greater than or equal to the user-specified Δ, (i.e. $Algo = 0$) Meta+Wah should be used. If $Algo = 1$, then WAH should be used. Intuitively, the heuristic is trying to limit the use of Meta+WAH to queries where there is a significant difference in the number of literals between the two operand columns. When such a difference occurs there is a higher chance that short-circuiting can occur (e.g. in a sorted bitmap querying a low-order column with a high-order column). By normalizing the difference to the total number of words being processed, the heuristic weights columns with fewer words as being more amenable to short-circuiting. Few words mean long runs and increased opportunities for short-circuiting and less overall processing time.

The results of the heuristic will, of course, vary for datasets. There are many factors the heuristic does not explicitly account for, including the length of literal chains, the average length of 0 runs, etc. By parameterizing the heuristic by Δ, the user is given the ability to select how conservative they would like the heuristic to be. If they set $\Delta < 0$ the heuristic will always select WAH and, conversely, Meta+WAH if $\Delta > 1$.

Algorithm 1: AND Operation with Short-Circuiting

```
   Input: Bit Vector A, B: Metadata List M_A, M_B
   Output: Bit Vector Z: The resulting compressed bit vector after
           performing the logical operation X ∧ Y
1  while A and B are not exhausted do
2      if A.currentWord is exhausted then
3          A.getNextWord();
4          if A.currentWord.isFill() then
5              M_A.getNextMeta();
6          end
7      end
8      if B.currentWord is exhausted then
9          B.getNextWord();
10         if B.currentWord.isFill() then
11             M_B.getNextMeta();
12         end
13     end
14     if A.currentWord.isFill() and B.currentWord.isFill() then
15         nWords = min(A.currentWord.nWords,
16                      B.currentWord.nWords);
17         Z.addFill(A.currentWord.fill
18                   ∧ Y.currentWord.fill, nWords);
19         A.currentWord.nWords -= nWords;
20         B.currentWord.nWords -= nWords;
21     end
22     else
23         if A.currentWord.isFill() and A.currentWord.fillValue()=0 then
24             nJumpLen = min(A.currentWord.nWords,
25                            M_B.numLiterals);
26             Z.addFill(0, nJumpLen);
27             A.currentWord.nWords -= nJumpLen;
28             M_B.numLiterals -= nJumpLen;
29             B.jumpWords(nJumpLen);
30         end
31         else if B.currentWord.isFill() and B.currentWord.fillValue()=0
           then
32             nJumpLen = min(B.currentWord.nWords,
33                            M_A.numLiterals);
34             Z.addFill(0, nJumpLen);
35             B.currentWord.nWords -= nJumpLen;
36             M_A.numLiterals -= nJumpLen;
37             A.jumpWords(nJumpLen);
38         end
39         else
40             Z.addLiteral(A.currentWord.getLitValue()
41                          ∧ B.currentWord.getLitValue());
42             M_A.numLiterals -= 1;
43             M_B.numLiterals -= 1;
44         end
45     end
46 end
47 return Z;
```

IV. Experimental Results

In this section, we present an evaluation of our metadata algorithm over real and synthetic data sets. Our experiments were run on a machine with two Intel Xeon E5-2630 processors (each having six 2.3 GHz cores with hyperthreading enabled), 128 GB DDR3 RAM. All algorithms used for comparison were written in Java. Each experiment was repeated five times: results from the first run were discarded to warm

the cache, and the average of the subsequent four runs are reported.

A. Data Sets

We prepared both synthetic and real data sets for testing. Here, we describe the data generation process.

Synthetic Data: The `uniform` data set was generated using a Zipf distribution generator. The generator assigns each bit a probability of: $p(k, n, skew) = (1/k^{skew})/\sum_{i=1}^{n}(1/i^{skew})$, where n is the number of elements determined by cardinality, k is their rank, and the coefficient $skew$ creates an exponentially skewed distribution. For uniform the $skew$ value was set to 0 to create a uniform distribution, the k value was set to 1, and n was set to 10. The resulting data set contains 32 million rows and 10 attributes. Each attribute was uniformly discretized using 10 equally-populated bins, resulting in 100 columns.

Real Data: We use the data set from KDD Cup'99 (labeled `KDD`), which captures network flow. The data set contains $4,898,431$ rows and 42 attributes [17]. Continuous attributes were discretized into 25 bins using Lloyd's Algorithm [18]. In total there were 475 bins. Another data set we use is Record Linkage (labeled `linkage`) which stores records from the Epidemiological Cancer Registry of the German state of North Rhine-Westphalia [19]. This data set contains $5,749,132$ rows and 12 attributes. The 12 attributes were discretized into 130 bins.

We compressed all data sets using WAH, and we gathered the metadata lists for each column.

B. Methods

To compare our algorithm, we constructed a query set that performs a bitwise operation over all possible pairs of columns,

$$Col_i \circ \ Col_j \ \ \forall\ i, j\ (i < j)$$

where $\circ$ is a boolean operation. Therefore, the total number of queries executed depends on the data set, but broadly, $\frac{n(n+1)}{2}$ queries are executed for a bitmap containing n columns. We used the bitwise AND operator ($Col_i \wedge \ Col_j$), because it is one of the most common operations in bitmap processing. The bitwise AND is also the target of optimization in our `Meta+WAH` scheme.

We ran several versions of our algorithm:

- `meta+wah-only`: Always consult the metadata while processing queries. This could lead to high overheads for queries between columns that may not contain many short-circuiting opportunities.
- `meta+wah-hybrid`: Only consult the metadata based on the heuristic presented earlier in Section III-A. Using this approach we could potentially avoid the overheads of reading metadata for columns in which the gains are predictably small.
- `meta+wah-opt`: Consider the optimal case of the hybrid approach by always selecting the algorithm that obtains the minimal execution time. This is a theoretical *a posteriori* result, and a useful line for comparison.

For each query we report the speedup $S(i, j)$ of the optimized schemes (`meta+wah-only`, `meta+wah-hybrid`, and `meta+wah-opt`) over normal WAH execution, which is expressed as:

$$S(i,j) = \frac{execution\_time_{WAH}(Col_i, Col_j)}{execution\_time_{Optimized}(Col_i, Col_j)}$$

For clarity in the presentation of our results, we have sorted the query-set (horizontal axis) in ascending order of $S(i, j)$. Intuitively, a $S(i, j) < 1$ means that using metadata was slower than simply running WAH, a $S(i, j) = 1$ means no gain over WAH, and $S(i, j) > 1$ means a $S(i, j)$-time performance improvement over WAH.

Parameterizing the Hybrid Algorithm: As described in Section III-A, `meta+wah-hybrid` is parameterized by Δ, which allows the user to select how conservative the heuristic is. Figure 3 shows how changing Δ affects the heuristic's selection ability to correctly choose between Meta+WAH and WAH for each of our data sets. The x-axis shows the values of Δ and the y-axis shows the percentage of queries that the heuristic was able to select the correct algorithm.

A lower $Delta$ value means that Meta+WAH is preferenced. For most of our data sets this is not a bad choice, however for `uniform_unsorted` this is not the case. Since uniform has very few runs, there will be few opportunities for short-circuiting so WAH will be faster for the vast majority of queries. Also, `uniform_unsorted` has very few differences so, with even a relatively small $Delta$, the heuristic exclusively selects WAH which is the correct choice. For the majority of the data sets, a Δ between .01 and .2 appears to be the ideal.

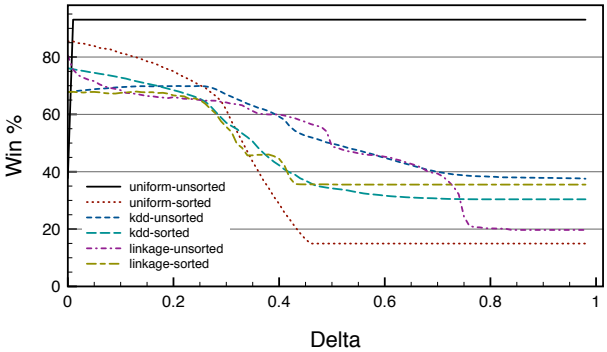


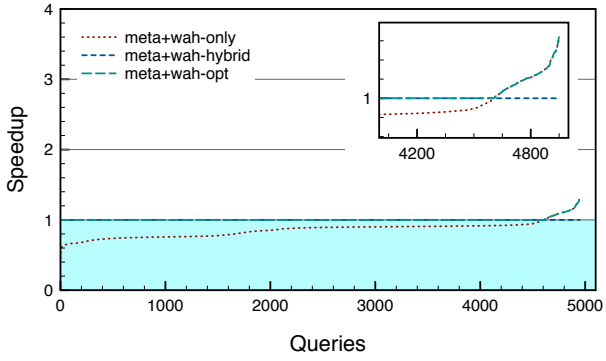
Fig. 3: Effect of Δ

Based on these results, we decided to fix $\Delta = 0.1$ for all remaining experiments.

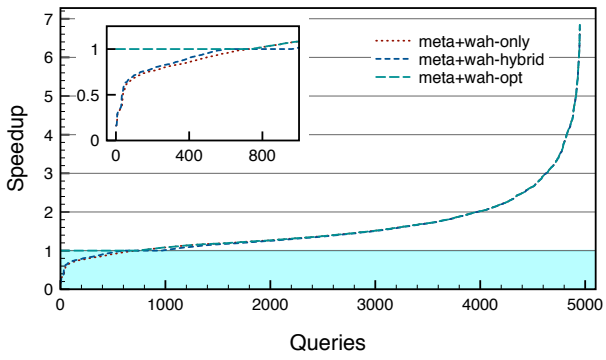
C. Summary of Results

Uniform (Synthetic): We first focus on the synthetic data set, `uniform`. In the unsorted case (shown in Figure 4(a)), 1s are uniformly distributed throughout the data set, rendering it dense and devoid of long runs of homogeneous bits. This

data set represents the worst case, and the poor results reflect this intuition. In total, there were 4950 queries run for this file. By always consulting the metadata (`meta+wah-only`), it underperforms pure WAH for 93.6% of all queries. Only 6.3% of the queries outperform WAH, for an average speedup per query of 0.86. The hybrid approach always conservatively selects WAH so it never underperformed. However, it missed the slight speedup of 1.007 realized by the optimal solution which selects Meta+WAH the few times short-circuiting is beneficial.



(a) Uniform (Unsorted)

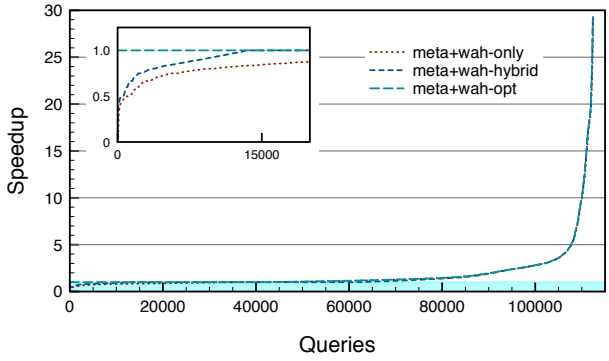


(b) Uniform (Sorted)

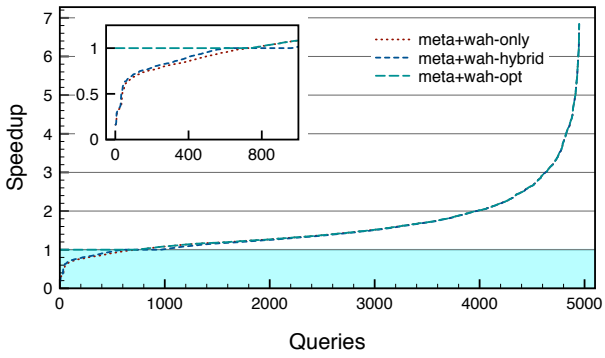
Fig. 4: Speedup over WAH (Uniform Dataset)

In Figure 4(b), we show the results for when `uniform` is sorted in Gray code order. The effects of sorting are drastic, as `meta+wah-only` outperforms pure WAH 85% of the time, and its average speedup per query is raised to 1.6. It achieves a maximal speedup of about 7×. This result is very nearly the optimal result, as `meta+wah-opt` achieves an average speedup of 1.63. For this data set, `meta+wah-hybrid` has the same overall performance as `meta+wah-only`.

KDD (Real): We now turn our focus on the unsorted version of `KDD`, shown in Figure 5(a). There are over 112500 queries, of which `meta+wah-only` outperforms WAH for 62.4% and underperforms 32.4% of the queries, respectively (the remaining queries were equal in time). The average speedup per query is 1.83. In a small percentage of the queries, we can observe relatively lucrative speedups (with the highest near 32×). This is nearly optimal as `meta+wah-opt` realizes an average speedup of 1.87. As shown `meta+wah-hybrid` underperformed pure WAH fewer times (28% of the queries versus `meta+wah-only`'s 32%). However, its increase in average speedup was negligible.



(a) KDD (Unsorted)



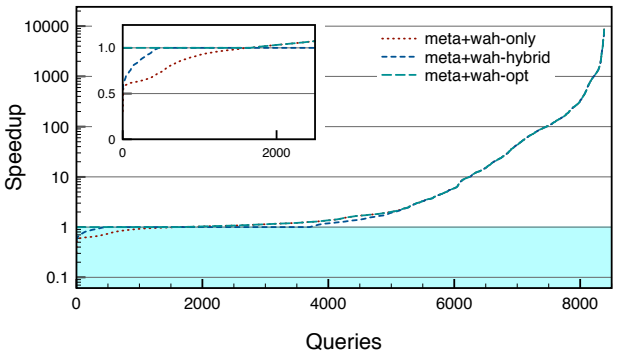
(b) KDD (Sorted)

Fig. 5: Speedup over WAH (KDD Dataset)

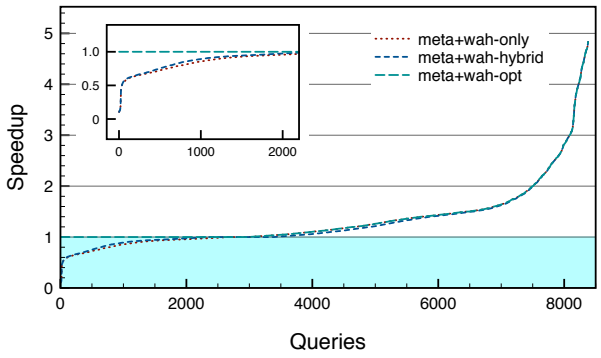
The results for the sorted version of `KDD` are shown in Figure 5(b). Here, the average speedup for `meta+wah-only` has been raised to 1.34, with 69.6% outperforming WAH. Again, the effects of sorting lead to better compression, which increases opportunities for short-circuiting. The maximum improvement is not as sharp as in the unsorted version because WAH's performance over the sorted data is also improving. For this data set, `meta+wah-hybrid` actually realized a slightly slower average speedup due to being overly conservative. The optimal average speedup was 1.38.

Linkage (Real): Finally, we shift our attention to the medical record linkage data set. The results for the unsorted version are shown in Figure 6(a). Here, `meta+wah-only` underperforms WAH for 19% of the queries whereas `meta+wah-hybrid` only underperforms for 11%. However, `meta+wah-hybrid`'s conservative nature meant that it only outperformed WAH for 61.3% of the queries. Conversely, `meta+wah-only` outperformed WAH for 80.3% of the queries. Both have an average speedup of approximately 84.8. The optimal speed up was 84.9. This data set demonstrates the potential benefits of short-circuiting as over 200 of the queries showed a +100× speedup.

The results for the sorted linkage data set are less dramatic because of the increased efficiency of WAH (Figure 6(b)).



(a) Linkage (Unsorted)



(b) Linkage (Sorted)

Fig. 6: Speedup over WAH (Linkage Dataset)

In this case, `meta+wah-only` underperformed WAH in 31.8% of the queries and outperformed in 64.5%. Similarly, `meta+wah-hybrid` underperformed for 27.5% of the queries and outperformed for 59.7%. They both realized a speedup of about 1.33. The optimal speedup was 1.38.

D. Size of Metadata

Data Sets	Unsorted			Sorted		
	Data	Meta	Ratio (%)	Data	Meta	Ratio
uniform	387	0.7	0.2	176	1.1	0.6
KDD	4.4	0.1	2.2	2.7	0.08	3.2
linkage	54.3	0.37	0.6	4.1	0.13	3.3

TABLE II: Compressed Data and Metadata Sizes (MB)

Table II shows the size of the WAH compressed data files and the corresponding metadata files for both the sorted and unsorted versions of the data sets. As shown,all metadata files are less than 3.5% the size of their corresponding compressed data files. In most cases, the increase in query efficiency easily justifies the slight increase in storage space.

V. Related Work

There are numerous other compression algorithms that are similar to WAH. Byte-aligned Bitmap Code (BBC) [20] was one of the earliest schemes applied to bitmap indices. BBC uses four types of byte-aligned atoms. The use of a 7-bit segment length allows BBC to compress compactly but is very CPU intensive when querying. WAH, which uses a word-aligned encoding, has been shown to typically use 60% more space but executes queries up to $12\times$ faster [21]. PLWAH [11] and Concise [22] are compression algorithms inspired by WAH. They both altered the formatting of the fill word. They reserve $\lceil \log_2 w \rceil$ bits, where w is the total number of bits used for a fill word. If the run being represented was terminated by a "near-fill" word (a word containing a single dirty bit) the reserve bits are used to indicate the position of that dirty bit. Thus, there is no need to create an addition literal word. PLWAH encodes near-fills that follow a run and Concise encodes those before a run. This approach can achieve $2\times$ the compression of WAH. Compax [5] introduced two new types of words in addition to WAH's literal and fill: fill-literal-fill (FLF) and literal-fill-literal (LFL). The new words encode short runs (run lengths that can be encoded in a byte) and literal words that differ from the fill by a single byte and appear in the patterns FLF or LFL. Compax was able to encode bitmaps using 60% less space than WAH. We believe that our metadata approach to short-circuiting is almost directly applicable to these types of algorithms and that they would experience speedups comparable to those of WAH.

Recent works have explored the advantages of using variable "word" lengths for encoding. Partitioned WAH (PWAH) [23] partitions 64-bit words into P segments where $P \in \{2, 4, 8\}$ and a header P bits long which indicates the type of each segment. For example, a word in PWAH-8 contains an 8-bit header which holds the flag bits for the segments. The remaining 56 bits are partitioned into 7-bit segments which can store literal or fill values. Similarly, Variable-Aligned Length code (VAL) [24] allows bitmap columns to be compressed using aligned encoding lengths (e.i. $15, 30, 60$). VAL maintains the variable-alignment to ensure that columns compressed using different encoding lengths can still be queried. Variable Length Compression (VLC) [12] is very similar to VAL. However, in VLC, the segment lengths can be arbitrary. Corrales, *et al.* showed that using truly arbitrary lengths was impractical when processing queries. Instead, they suggested limiting lengths to $m \times b$ where b is a common base. It was shown that the need to translate both columns could lead VLC to be 3 to 4 times slower than VAL [24]. PWAH does not propose to execute queries involving bitmaps of varying partitions sizes so it could directly benefit from our approach to short-circuiting. However, VAL and VLC present algorithms for efficient querying between columns compressed using different encoding lengths. To use our approach these algorithms would have to somehow translate the metadata on-the-fly to compensate for the varying compression lengths. It would be interesting to see if these algorithms could benefit from such an approach.

Our previous work showed how the use of metadata collected at compression-time could be used to increase the efficiency of range queries and point queries in VAL [25]. This work used metadata to dynamically select encoding-length translation algorithms for processing queries between columns compressed using different encoding lengths. It also explored

a variety of column orderings based on metadata to speed up the processing of range queries. It did not explore ways to use metadata to enable short-circuiting during query processing.

The work most closely related to ours is that of Enhanced WAH (EWAH) [13]. Like WAH, EWAH uses both literals and fills when compressing bitmaps. It differs from WAH, in the format of these words. Literal words no longer have a flag bit. Thus, all bits are used to encode verbatim the bitmap. Fill atoms are divided in half. The upper half (the most significant bits) are used to encode the value bit and run length, similar to WAH, but there is no longer a flag bit. The lower half is used to encode the number of literal atoms that follow the current fill. This additional information enables short-circuiting in EWAH in a manner similar to our approach. This allows EWAH to achieve similar speedups to Meta+WAH. However, since the metadata is encoded in the compressed data files, fewer bits are used to encode the run-length. Thus, EWAH may need to use two words to encode long runs compared to one word used in WAH, in some instances. Additionally, EWAH incurs a slightly increased processing cost when compared to WAH, since it must always process the metadata. By separating the metadata from the data files, our approach is more flexible and applicable to a wide range of bitmap compression algorithms. As shown above, our hybrid approach reduces the number of instances when the processing cost of the metadata is detrimental to query execution time.

There are other compression techniques for bitmap indices that are not based on run-length encodings. For example, Chambi, *et al.* introduced Roaring bitmap compression [26] which uses a data structure that stores bitmap set entries as 32-bit integers in a two-level index scheme. RIDBit [27] also uses a two-level indexing scheme in the form of a B-tree of bitmaps. When the bitmap is sparse, both of these approaches record the position of the dirty bits in lists. In this manner, they circumvent the need for short-circuiting, as they only process the positions of the dirty bits. In an empirical study, Roaring bitmaps were able to process AND queries $\times 4 - \times 5$ faster than WAH and Concise for synthetic data. It would be interesting to compare these approaches to Meta+WAH.

VI. Conclusion and Future Work

In this paper we introduced Meta+WAH, an algorithm that uses metadata, collected at compression-time, to enable logical short-circuiting when querying bitmaps compressed using WAH. The results of our empirical study showed that Meta+WAH realized average query speedups of $1.3\times$ to $+84\times$ when compared to WAH. We also presented a heuristic that identified queries where the processing overhead of Meta+WAH will not likely be off-set by the number of eliminated memory accesses. Using this heuristic we created a hybrid approach that realized similar speedups as Meta+WAH but reduced the number of times it underperformed WAH.

In the future, we plan to extend our technique to a variety of other bitmap compression algorithms. We also plan to explore different types of metadata that can facilitate short-circuiting.

References

[1] P. E. O'Neil, "Model 204 architecture and performance," in *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, (London, UK), pp. 40–59, Springer-Verlag, 1989.

[2] K. Stockinger and K. Wu, "Bitmap indices for data warehouses," in *In Data Warehouses and OLAP. 2007. IRM*, Press, 2006.

[3] "Apache Hive Project, http://hive.apache.org."

[4] R. R. Sinha and M. Winslett, "Multi-resolution bitmap indexes for scientific data," *ACM Trans. Database Syst.*, vol. 32, August 2007.

[5] F. Fusco, M. P. Stoecklin, and M. Vlachos, "Net-fli: On-the-fly compression, archiving and indexing of streaming network traffic," *VLDB*, vol. 3, no. 2, pp. 1382–1393, 2010.

[6] A. Romosan, A. Shoshani, K. Wu, V. M. Markowitz, and K. Mavrommatis, "Accelerating gene context analysis using bitmaps," in *SSDBM*, p. 26, 2013.

[7] Y. Su, G. Agrawal, J. Woodring, K. Myers, J. Wendelberger, and J. P. Ahrens, "Taming massive distributed datasets: data sampling using bitmap indices," in *HPDC*, pp. 13–24, 2013.

[8] Y. Su, Y. Wang, and G. Agrawal, "In-situ bitmaps generation and efficient data analysis based on bitmaps," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2015, Portland, OR, USA, June 15-19, 2015*, pp. 61–72, 2015.

[9] S. Shohdy, Y. Su, and G. Agrawal, "Load balancing and accelerating parallel spatial join operations using bitmap indexing," in *22nd IEEE International Conference on High Performance Computing, HiPC 2015, Bengaluru, India, December 16-19, 2015*, pp. 396–405, 2015.

[10] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," *ACM Trans. Database Syst.*, vol. 31, pp. 1–38, Mar. 2006.

[11] F. Deliege and T. Pederson, "Position list word aligned hybrid: Optimizing space and performance for compressed bitmaps," in *EDBT'10*, pp. 228–239, 2010.

[12] F. Corrales, D. Chiu, and J. Sawin, "Variable Length Compression for Bitmap Indices," in *DEXA'11*, pp. 381–395, 2011.

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

[14] T. Apaydin, A. c. Tosun, and H. Ferhatosmanoglu, "Analysis of basic data reordering techniques," in *SSDBM'08*, pp. 517–524, 2008.

[15] A. Pinar, T.Tao, and H. Ferhatosmanoglu, "Compressing bitmap indices by data reorganization," in *ICDE'05*, pp. 310–321, 2005.

[16] D. Lemire, O. Kaser, and E. Gutarra, "Reordering rows for better compression: Beyond the lexicographic order," *ACM Transactions on Database Systems*, vol. 37, no. 3, pp. 20:1–20:29, 2012.

[17] M. Lichman, "UCI machine learning repository," 2013.

[18] S. P. Lloyd, "Least squares quantization in pcm," in *IEEE Transactions on Information Theory*, vol. 28, pp. 129–137, 1982.

[19] M. Sariyar, A. Borg, and K. Pommerening, "Controlling false match rates in record linkage using extreme value theory," *Journal of Biomedical Informatics*, vol. 44, no. 4, pp. 648–654, 2011.

[20] G. Antoshenkov, "Byte-aligned bitmap compression," in *DCC '95: Proceedings of the Conference on Data Compression*, p. 476, 1995.

[21] K. Wu, E. J. Otoo, and A.Shoshani, "Compressing bitmap indexes for faster search operations," in *SSDBM'02*, pp. 99–108.

[22] A. Colantonio and R. Di Pietro, "Concise: Compressed 'n' composable integer set," *Information Processing Letters*, vol. 110, no. 16, pp. 644–650, 2010.

[23] S. J. van Schaik and O. de Moor, "A memory efficient reachability data structure through bit vector compression," in *ACM SIGMOD International Conference on Management of Data*, pp. 913–924, 2011.

[24] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin, "A tunable compression framework for bitmap indices," in *IEEE International Conference on Data Engineering (ICDE'14)*, 2014.

[25] R. Slechta, J. Sawin, B. McCamish, D. Chiu, and G. Canahuate, "Optimizing query execution for variable-aligned length compression of bitmap indices," in *International Database Engineering & Applications Symposium*, pp. 217–226, 2014.

[26] S. Chambi, D. Lemire, O. Kaser, and R. Godin, "Better bitmap performance with roaring bitmaps," *Software: Practice and Experience*, 2015.

[27] E. O'Neil, P. O'Neil, and K. Wu, "Bitmap index design choices and their performance implications," in *Database Engineering and Applications Symposium*, pp. 72–84, 2007.