Improving the Querying Efficiency of the PLWAH Bitmap Algorithm

Benjamin Taufen
Computer and Information Sciences
University of St. Thomas
St. Paul, MN, USA

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

David Chiu
Mathematics and Computer Science
University of Puget Sound
Tacoma, WA, USA

ABSTRACT

Bitmap indices are commonly used for accessing large, read-only data. A bitmap is a simplified model of the underlying data in secondary storage. Its coarse representation enables the use of fast CPU operations to answer common database queries. Additionally, bitmaps are very compressible. Several known compression algorithms allow the compressed form of the bitmap to be queried directly, and one of which is Position List Word-Aligned Hybrid (PLWAH). PLWAH is modified hybrid run-length encoding scheme that can achieve better compression than traditional schemes such as Word-Aligned Hybrid (WAH). This improved compression introduces an increased query processing cost, of which we address in this paper. We present a technique that uses metadata to allow PLWAH's query algorithm to exploit logical short-circuiting opportunities, reducing the cost of certain queries. In our empirical study, we found that our approach achieved an average speedup of 1.41× over PLWAH for real scientific data sets. For specific queries, our approach realized speedups as high as 8000×.

ACM Reference format:
Benjamin Taufen, Jason Sawin, and David Chiu. 2017. Improving the Querying Efficiency of the PLWAH Bitmap Algorithm. In *Proceedings of IDEAS '17, Bristol, United Kingdom, July 12-14, 2017,* 8 pages.
https://doi.org/10.1145/3105831.3105868

1 INTRODUCTION

Efficient query execution in modern database systems relies on advanced indexing schemes. Bitmap indices [15] is one such index commonly used in large industrial and scientific read-only databases. Such indices create a coarse binary representation of the data stored in underlying relations in the form of a sparse two-dimensional bitmap. The binary model allows queries to be processed using the fast primitive CPU logic operations, and the sparse nature of bitmaps has led to the creation of numerous run-length compression techniques [7–9, 27, 28]. These compression schemes greatly reduce the size of a bitmap index while also increasing query response time. Compressed indices can be queried directly without explicit decompression, thus reducing the number of memory accesses required to process a query. This paper presents a modification to a popular compression scheme that further reduces the number of memory accesses needed to answer queries.

IDEAS '17, July 12-14, 2017, Bristol, United Kingdom
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5220-8/17/07…$15.00
https://doi.org/10.1145/3105831.3105868

Tuples	Subject	# Pubs
t_1	Soft. Eng.	725
t_2	A.I.	55
t_3	DB	450
t_4	Graphics	420

Tuples	Subject				# Pubs			
	x_0	x_1	x_2	x_4	y_0	y_1	y_2	y_3
t_1	1	0	0	0	0	0	0	1
t_2	0	1	0	0	1	0	0	0
t_3	0	0	1	0	0	0	1	0
t_4	0	0	0	1	0	0	1	0

Table 1: Example Relation to Bitmap

Bitmaps are created by discretizing the attributes of a relation into a series of bins. Bins either represent value ranges or distinct values. Consider the simple example shown in Table 1. The left table shows a relation that consists of two attributes **Subject** and **# Pubs**. The right table presents a possible bitmap that could be created to represent the relation. In the bitmap, the values of the attributes for each tuple are assigned to bins. The x bins encode the distinct possible values for **Subject** (e.g., x_0 is *Soft. Eng*, x_1 is *A.I.* and so on). The y bins typify ranges of possible values for the **# Pubs**. Specifically, y_0 represents [0, 200), y_1 is [200, 400), y_2 is [400,600) and y_3 is [600, ∞). The values of the bitmap rows are derived through the binning process of the corresponding relation tuple. For example, the **Subject** value for tuple t_1 is *Soft. Eng.*. Thus in the bitmap row corresponding to t_1, a 1 is placed in the x_0 bin, and the remaining x bins receive a 0 value. Similarly, the **# Pubs** value for t_1 is 755 so a 1 is placed in the y_3 bin and the remain bins are zeroed out.

Bitmaps can be processed to either directly answer queries or significantly reduce the number of tuples that must be investigated to respond to a query. For example, suppose a user wants to find all topics that had at least 400 publications. The query could be processed by performing a bitwise *OR* operation between the bitmaps y_2 and y_3 bit vectors. Every row that contains a 1 in the resulting bit vector represents a tuple that has a publication count in the desired range.

Though bitmaps are a simplified representation of the raw data, they can grow to become too large for core memory. A variety of compression schemes have been designed to address this issue. One of the most widely-used bitmap compression techniques is *word-aligned hybrid* code (WAH) [27]. It compresses each bit vector into *words*. WAH uses two types of word representations, `fills` which are compressed runs of homogeneous bits and `literal` which capture the bit values of heterogeneous sequences. WAH was the inspiration for *position list word-align hybrid* code (PLWAH) [8]. Like WAH, PLWAH uses `fill`, and `literal` words that are system aligned. However, PLWAH's `fill` words incorporate a *position list*. This list is used to encode words that contain a single heterogeneous bit that follows a run. Experiments have shown that this enhancement can enable PLWAH to decrease compression size by 50% when compared to WAH [8].

Due to its *position list*, PLWAH's querying algorithm is more complicated than WAH's. This increased processing overhead can lead to slower query response times for certain bitmaps. We present an approach that can improve PLWAH's querying efficacy. Our approach implements logical short-circuiting through the use of metadata. The metadata records structural facts about PLWAH compressed bit vectors. The query processing engine uses this knowledge to avoid memory reads when it encounters long sequences of homogeneous bits that allow for short-circuiting. For example, if the engine is performing a logical *AND* between two columns, and it determines that one column contains a run of 6300 0s it does not need to read in the corresponding bits in the other column since they will not impact the result of the query.

The significant contributions of this paper are the PLWAH short-circuiting algorithm we present and our detailed evaluation of its performance. We explore the effects of our algorithm on PLWAH query times on both synthetic and real scientific data. Further, we compared our approach against the performance of WAH on the same data sets. The results of our study show our approach realized an average speedup of 1.11× over WAH and 1.41× over PLWAH on real data sets. We also found for certain queries our approach could achieve over an 8000× speedup when compared to PLWAH.

The remainder of the paper is organized as follows. In Section 2 we provide an overview of the PLWAH compression and query algorithms. Section 3 presents our enhanced query algorithm that enables short-circuiting, as well as, our specialized single-sided approach. Section 4 presents the results of our empirical study. We present related works in Section 5. We conclude and present plans for future work in Section 6.

2 POSITION-LIST WORD-ALIGNED HYBRID COMPRESSION (PLWAH)

This section provides an overview of PLWAH's compression and query algorithms. PLWAH has a flexible general form and can accommodate different system word sizes. This paper only considers a PLWAH version that uses a single *position list* and is configured for a 64-bit system.

PLWAH compresses bitmaps a single bit vector, or bin, at a time. To aid in the demonstration this process Figure 1(a) presents an example bit vector. First, PLWAH clusters the vector into consecutive groups of 63 bits. These groups are represented as rows in the figure. As shown, the first four homogeneous groups only contain the value 0. The fifth group represents a "nearly identical" group. That is a group that contains a single heterogeneous bit that follows a homogeneous group. The remaining group in the example includes several heterogeneous values.

$$\overbrace{\texttt{000000000000000000000000000000000000000000000000000000000000000}}^{63\ Bits}$$

```
000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
000010000000000000000000000000000000000000000000000000000000000
000000001000000000000000000000000000000001100000000000000000100
```

(a) Uncompressed Bit Vector

```
1|0|000101|00000000000000000000000000000000000000000000000000000100
0|000000001000000000000000000000000000000001100000000000000000100
```

(b) PLWAH Compressed Bit Vector

Figure 1: Example of PLWAH Compression

PLWAH uses two types of words when compressing a bit vector: *literals* and *fills*. A literal has the form $(flag, value)$, where $flag$ is 0 and $value$ represents an uncompressed bit sequence. A fill's form is $(flag, bitVal, posList, runLen)$ and is used to compress runs of homogeneous groups. A $flag$ value of 1 identifies the word as a fill and the $bitVal$ is a single bit that records whether the run is of 1's or 0's. The number of groups being compressed is recorded in the 56 bits of *runLen*. If the run is followed by a nearly identical group, the position of the heterogeneous bit in that group is recorded in *posList* which comprises 6 bits.

Figure 1(b) shows the PLWAH compressed version of the example bit vector. The compressed version only requires two words. The first is a fill word, as designated by the 1 value in the most significant bit. A $bitVal$ of 0 and a *runLen* value of 4 (shown in binary) encodes that the first 4 groups of 63 bits were all 0's. The *posList* value of 5 encodes that the fifth group was nearly identical and that the heterogeneous value was located in the 5th most significant bit. The other word in the compressed form is a literal as indicated by its $flag$ value being set to 0. The $value$ is identical to the bit sequence of the 6th group of bits in the uncompressed bit vector.

Queries of the compressed bitmap are of the form $A \circ B$, where A and B are bit vectors and $\circ$ is a bitwise logical operation. The result of a query is a compressed bit vector C which is equivalent to the result of uncompressed $A\circ$ uncompressed B. The query processing algorithm treats the operand bit vectors like stacks. It pops the first word from each vector. The paired words are analyzed until they are fully exhausted, e.g. entirely processed. When a word is exhausted the next word from that bit vector is popped. Consider an example of $A \wedge B$. There are three possible word type pairings:

(1) **literal** $\wedge$ **literal**: In this case, a new *result* literal word is added to C where $C.result.value = A.w_y.value \wedge B.w_z.value$, where w_y and w_z are the current words being processed from A and B respectively. This exhausts both $A.w_y$ and $B.w_z$ and the next words from both vectors are popped.
(2) **fill** $\wedge$ **fill**: Here the *result* added to C is a fill word. Specifically,

$C.result.bitVal = A.w_y.bitVal \wedge B.w_z.bitVal$ and
$C.result.runLen = Min(A.w_y.runLen, B.w_z.runLen)$

```
A { 1|0|000011|00000000000000000000000000000000000000000000000000100000
    0|000000001000000000000000000000000000000110000000000000000000100

                                  &

B { 0|000000100000011100000000000000000000000000000000000000000000000
    0|000000000000000000000000000000000000000001100000000000000000101
                                  ⋮
    0|000000000000010000000000000000000000000000000000000000000000001
    0|001100011111000000000000000000000000000000000000000000000000110

                                  =

C { 1|0|000000|00000000000000000000000000000000000000000000000000100001
    0|000000001000000000000000000000000000000000000000000000000000100
```

Figure 2: Example AND query

This will exhaust at least one of the words. Assume the exhausted word is A. If $A.w_y.posList > 0$, a new literal word is built with all bit values set to $A.w_y.bitVal$ except for $bit_{posList}$ which is set to $\overline{A.w_y.bitVal}$. This new literal is then pushed onto the top of A.

(3) **fill $\wedge$ literal**: in this case $result$ is a literal. Assume that $A.w_y$ is the fill word. If $A.w_y.bitVal$ is 1 then $C.result.value = B.w_z.value$, else $C.result.value = 0$.

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 run length of the fill words.

3 SHORT-CIRCUITING

In PLWAH queries, logical short-circuiting can occur when examining only one of the words being processed is sufficient to determine the result. The example query shown in Figure 2 will help demonstrate this concept. It displays two compressed bit vectors, A and B, participating in an *AND* query. The resulting bit vector, C, is also shown. A contains two words. The first is a fill representing a run of 32×63 0s. The *posList* of that word indicates that the run was followed by a nearly identical block that had a single 1 value in the third-bit position. The remaining word in A is a literal. In total, A represents 34 uncompressed blocks. B contains 34 literals. The resulting bit vector, C, is a fill word representing a run of 33×63 zeros and a literal word.

To process the above query, PLWAH first scans in the fill from A. As this fill represents 32 words of 0 it presents an opportunity for short-circuiting. Logically, PLWAH does not need to process the first 32 literals in B as any value *AND*ed with 0 is 0. Unfortunately, at query time, PLWAH does not have knowledge of the structure of compressed columns. It does not know that B has 32 literals to skip. Without this information, it has to linearly process the words in B until A's fill word is exhausted to ensure tuples are evaluated in the correct order. Once the run is exhausted, it creates a new literal to account for the encoded nearly identical word. The nearly identical word and the second to last literal in B are *AND*ed together, this results in a word of all zeros which is added to the fill result word. Finally, the last literal of A and B and processed and a new literal is added to the result.

We present an approach that uses metadata collected at compression time to enable PLWAH to take advantage of short-circuiting.

Algorithm 1: Short-Circuiting AND

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

This metadata records the number of initial literals in each vector and the number of literals following each run. In the example query above, this structural knowledge would have allowed PLWAH skip over the first 34 literals in B, thus saving 34 memory accesses.

Algorithm 1 presents a modified PLWAH *AND* operation that uses metadata to implement short-circuiting. The inputs to the algorithm are two compressed bit vectors, A and B and their respective metadata lists, M_A and M_B. It returns a compressed bit vector Z which encodes the solution to the query. The algorithm iterates over the words of the vectors until all of A and B are exhausted (line 1). Each iteration processes a single $currentWord$ from each column. Initially, $currentWord$ is set to exhausted for both bit vectors. Lines (2-21) assess the state of the bit vectors' $currentWord$

every iteration. If a word is found to be exhausted, a secondary check is performed (lines 3 and 13). This check determine if the newly exhausted word encoded a nearly identical word. Essentially, if the exhausted $currentWord$ is a fill and has a $position - list > 0$, a new literal is built to represent the encoded word. This new literal becomes the $currentWord$ (lines 4 and 14). If the exhausted word does not encode a near word, the next word from the same bit vector is retrieved and examined (lines 5-9 and 15-19). If the new word is a fill, then the metadata is updated to reflect the number of literals that follow it.

Lines 22-45 process the two $currentWords$. If both words are fills, a new *result* fill word is added to Z. The run length of the result word is the shorter of the two operand run lengths (line 23). The value of this result word is the logical *AND* of the $bitValues$ of the two $currentWords$ (line 24). The run lengths of the two operand words are adjusted to reflect that a portion of their encoded values has been processed (lines 25-26).

If one of the operand words is a fill of 0s and the other a literal, then the algorithm can employ short-circuiting (lines 28 and 40). Consider the case when $A.currentWord$ is a fill of 0s and $B.currentWord$ is a literal (lines 28-34). Here, the algorithm examines B's metadata which represents the number of literals from $B.currentWord$ to the next fill word. The minimum of B's metadata and A's run length is found and stored in $nJumpLen$ (line 29). This values represents the number of literal that can be short-circuited in B. A new 0 fill of length $nJumpLen$ is added to Z. Lines 31 and 32 update the number of words processed for $A.currentWord$ and B's metadata. Finally, B's current word is jumped $nJumpLen$ words (line 33), avoiding that many memory accesses.

Lines 40-44 address instances when no short-circuiting is possible because either both words are a literals or one is a fill of 1s. In such cases, a new literal is added to Z which is the result of literal value of the two operand words *AND*ed together. Finally, both sets of metadata are updated. The $isLit()$ function returns 1 if the $currentWord$ is a literal and 0 otherwise.

4 EVALUATION

In this section, we present an extensive evaluation of our metadata algorithm. Our experiments were run on a Windows machine with two Intel Xeon E5-2630 processors (each having six 2.3 GHz cores with hyperthreading enabled) and 128 GB DDR3 RAM. All techniques used for comparison were written in Java using consistent data structures and query-processing algorithms to ensure a fair comparison. Each experiment was repeated multiple times. The results from the first run were discarded to warm the cache, and the average of the subsequent runs are reported in the paper.

4.1 Experimental Setup

The following data sets were used for our evaluation. They are representative of the type of applications (*e.g.*, scientific, mostly read-only) that would benefit from bitmap indexing.

- `linkage` – contains anonymized records from the Epidemiological Cancer Registry of the German state of North Rhine-Westphalia [17]. This data set contains 5, 749, 132 rows and 12 attributes. The 12 attributes were discretized into 130 bins.
- `KDD` – this data set was procured from KDD Cup'99 and captures network flow traffic. The data set contains 4, 898, 431 rows and 42 attributes [12]. Continuous attributes were discretized into 25 bins using Lloyd's Algorithm [13], resulting in 475 bins.
- `energy` – contains measurements reported from 20 synchrophasors deployed over the Pacific Northwest power grid over approximately one month [2]. Data from all synchrophasors arrive at a rate of 60 measurements per second and are discretized into 1367 bins. There are 7, 273, 800 rows in this data set.
- `uniform` and `zipf2` – these data sets were generated to represent the worst-case and the common-case, respectively. Each data set contains 100 bins total (*i.e.,* ten attributes discretized into ten bins). In `uniform`, an attribute value has a 1/10 probability of falling into any bin, resulting in a data set that is noisy and hard to compress. In `zipf2`, an attribute value has a far higher chance (zipf distribution with $skew = 2$) of falling into a given bin. 32 million rows were generated for each data set.

Because the order of tuples is arbitrary, previous work has shown that sorting bitmaps using *greycode* ordering can help extend run-lengths and therefore improve compression ratios [16]. We, therefore, prepared two versions (unsorted and greycode ordered) of each data set for our comparisons.

We ran the following set of queries over each data set:

$$bin_i \wedge bin_j \; \forall \; i, j \; (i < j)$$

The bitwise AND operator ($bin_i \wedge bin_j$) was used, because it is considered the most common operation in bitmap-index processing. In the next subsection, we report results using the following labels.

- `wah`: Data sets were compressed and processed using the 64-bit version of WAH.
- `plwah`: Data sets were compressed using 64-bit PLWAH with one position list being supported.
- `plwah+full`: Data sets were compressed using 64-bit PLWAH with one position list. Metadata on the bit-distribution was collected for each bin in the bitmap.

4.2 Results

We present the results of our experimental evaluation in this subsection. Initially, we discuss the overall benefits of short-circuiting when used in conjunction with `plwah`. Next, we evaluate our algorithm over synthetic data sets and real scientific data sets.

4.2.1 Impact of Metadata-Informed Short Circuiting. Table 2 presents some overarching results on using our short-circuiting method. All data sets have been grey-code sorted. As can be seen, our approach significantly reduces the number of words decoded per query across all data sets. As one would expect, `uniform` data observes the fewest number of words skipped compared to all other data sets, since their data tend to be skewed.

However, we should be quick to note that a higher number of words skipped does not imply lower query-processing time, as we will show in the Section 4.2.2. This is because short skips involving only a few words may occur frequently in denser data sets (`uniform`). The short skips do not individually account for

Data Set	Queries	Avg Words	Avg Words Skipped
uniform	4949	463252	230490 (49.7%)
zipf2	4949	93120	53527 (57.4%)
KDD	112574	1295	837 (64.6%)
linkage	8384	7880	4815 (61.1%)
energy	933660	8430	5554 (65.8%)

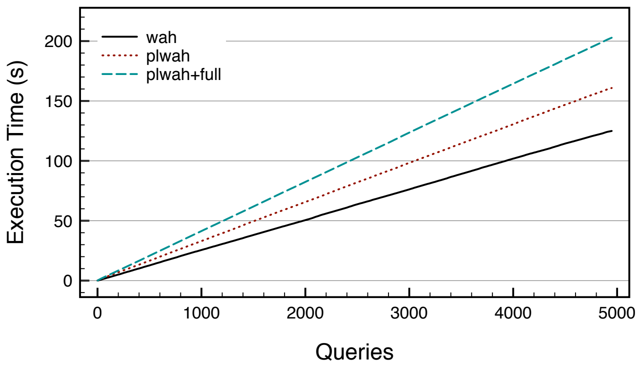
Table 2: PLWAH Words Short-Circuited

significant performance gains, but still add up to a high number of total skips used for computing the average words skipped per query. The high average words per query for `uniform` support this hypothesis.

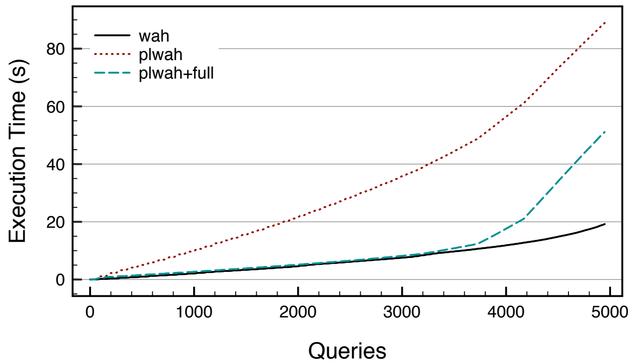
Conversely, performance gains are high when queries observe few long skips. Such long skips are prevalent where data is sparser by nature (`zipf2` and real data). We would therefore expect a range of modest to high speedups over PLWAH in query execution with these data sets.

4.2.2 Performance of Query Processing. Initially, we will evaluate the effect of our approach on synthetic data. Figures 3(a) and 3(b) respectively show the execution times for the unsorted and greycode-sorted versions of the `uniform` data set. First, consider the unsorted results shown in Figure 3(a). The uniform data set is inherently noisy, and opportunities for compressing runs are elusive. As expected, `wah` excels for unsorted uniform data. It is 1.3× faster than `plwah` and 1.6× faster than `plwah+full`. This is due to the presence of only shorter runs in the data, which `plwah` can compress just as efficiently, but suffers from the overhead of decoding the position-list position list during query processing. The absence of runs is even more deleterious to our approach `plwah+full` which under-performs both of the other algorithms. This lack of performance is because `plwah+full` adds an additional metadata-decoding overhead to `plwah`.

Lastly, we discuss the impact of row-sorting on the performance of all methods being evaluated. Consistent with previous works [11, 16], we show that row-sorting culminates in an overall speedup of 6.5 for `wah`, 1.8 for `plwah`, and 3.98 for `plwah+full`. Recall that the objective of greycode-sorting is to maximize runs, an attempt to eliminate the presence of disruptive bits – the ones that `plwah` encodes more efficiently. We therefore expect `wah` to deliver lucrative performance gains over `plwah` and `plwah+full`, which only observe modest speedup due to their decoding (and metadata-lookup) overhead. Indeed, for uniformly distributed data, we demonstrate that `wah` is the unequivocal winner, and should be used over `plwah` and our optimizations. However, `plwah+full` outperforms `plwah` by quite a large margin for the sorted version, which implies that opportunities for short-circuiting were more than enough to offset the metadata decoding overhead. Though `plwah+full` still under-performs `wah`, as shown in the Figure 3(b) there is really only a significant performance difference in later queries. This is due to a "quirk" of greycode-sorting. When the rows are sorted, long runs are manifested in the low-order columns but the high-order columns tend toward a uniform distribution of set bits. As shown, `plwah+full` has nearly identical performance to `wah` until high-order columns are queries against each other. In these cases, both



(a) Uniform (Unsorted)



(b) Uniform (Sorted)

Figure 3: Query Execution Time (uniform)

columns consist largely of literals and there are few opportunities for short-circuiting.

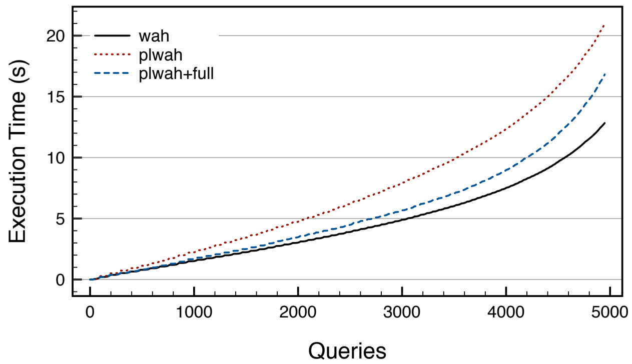


Figure 4: Query Execution Time (zipf2)

However, real data are seldom random and uniform. Many attributes tend to favor certain values or value-ranges (e.g., voltage on the power grid usually operates within normal range). Next, we evaluate our algorithms over skewed and real data sets. Furthermore, because we observe similar speedups for all other data sets, we will show only the results over greycode-sorted versions for the remainder of our evaluation.

Figure 4 displays the results for the zipf-distributed synthetic data set. Compared to `uniform`, one can observe the effects of data distribution on the various algorithms. Over 32 million rows, the 5000 queries finish in a fraction of the time compared to `uniform` data. `wah` still outperforms both `plwah` and `plwah+full`, but not by a wide margin (1.75× over `plwah` and 1.4× over `plwah+full`).

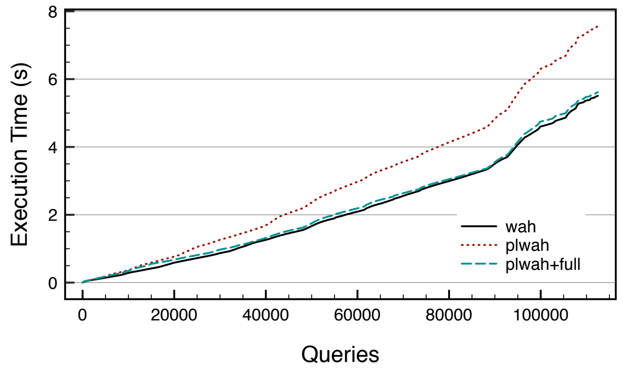


Figure 5: Query Execution Time (KDD)

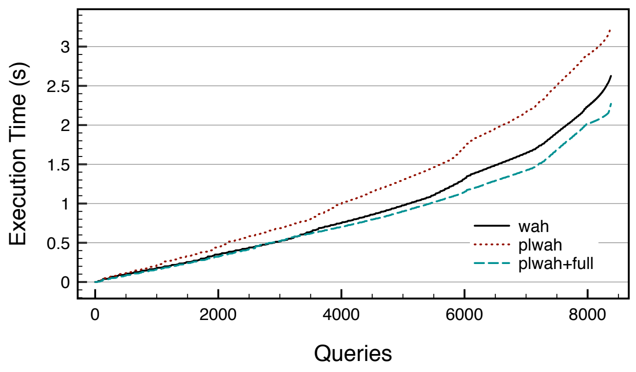


Figure 6: Query Execution Time (linkage)

Next, we discuss results for the real scientific data sets whose source and characteristics were explained in Section 4.1. Figures 5, 6, and 7 show the comparisons of query execution time for `kdd`, `linkage`, and `energy`, respectively. As expected in all three data sets expected `plwah` loses to `wah` due to its decoding overhead. The benefits of our optimization are also present, as `plwah+full` outperforms `plwah` by a significant margin with speedups of 1.35×, 1.43×, and 1.47×, respectively. Surprisingly, `plwah+full` not only tracks the performance of `wah`, it even outperforms `wah` in for two of the data sets that we tested, `linkage` (1.16×) and `energy` (1.2×). Where `plwah+full` loses (for the `kdd` data), it only experiences a 2% slowdown.

To better inform our results, we further investigate the behavior of the queries across our data sets. Figure 8 shows the percentage of queries where `plwah+full` outperformed over `wah` and `plwah`. We

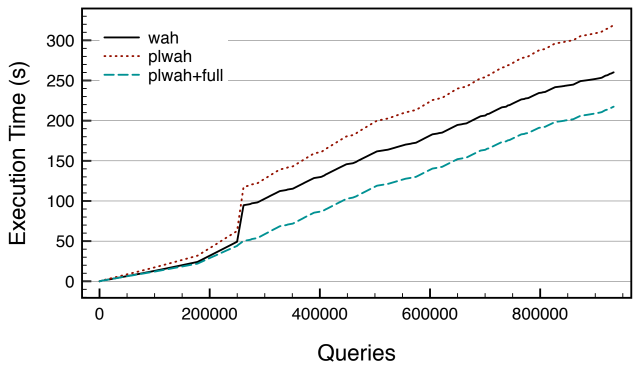


Figure 7: Query Execution Time (energy)

are again, only considering the sorted data sets. Has shown, for all but `kdd` `plwah+full` outperforms `plwah` on more than 50% of the queries performed. In the case of `energy`, `plwah+full` out performs `plwah` on over 80% of the queries. As expected, the results are less dramatic when `plwah+full` is compared to `wah`. It is interesting to note that though `plwah+full` only out performs `wah` on roughly 25% of the queries that is enough of an improvement to give the two techniques approximately the same performance.

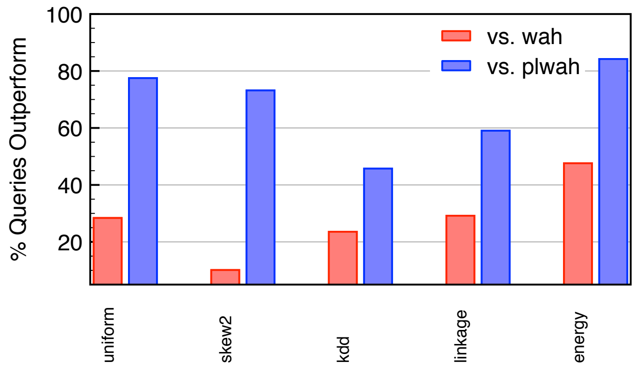


Figure 8: Query Outperformance Rates

Above, we discussed the results of our experimentation in the aggregate, however exploring the results of individual queries provided interesting results. For brevity, we will only consider the performance of `plwah+full` to `plwah` on the real-world data. For `kdd`, the best speedup realized by `plwah+full` for an individual query was 6.9×. For `linkage`, the maximum speedup was 17.4× and impressively it was 8038.4× for `energy`. One of the reasons the significant improvements can be seen by examining how many literals the queries were able to skip reading from memory due to short-circuiting. For `kdd` `plwah+full` was able to skip 6,701 liters, for `linkage` it was 35,407 and for `energy` it was 115,451 literals.

4.3 Size of Metadata

One of the concerns with using metadata in conjunction with compressed bitmaps is that the size of the metadata will significantly

Data Sets	Data	Meta	Overhead (%)
energy	43.9	1.74	3.94
KDD	2.3	0.11	4.78
linkage	3.9	0.17	4.35
uniform	387	0.13	.03
uniform (sorted)	176	1.22	.69
Zipf2	35.5	1.75	4.92

Table 3: PLWAH Compressed Data and Metadata Sizes (MB)

increase the memory footprint. Table 3 presents the size of the PLWAH compressed data sets as well as the size of the associated metadata files. For each of the data sets considered in this paper, the corresponding metadata files were less than 2 MB in total size and represented less than a 5% increase in space. This meant that for any given data set, the metadata easily fit in core-memory during query processing and that it did not significantly increase the needed hard-disk space.

5 RELATED WORK

The dominant cost incurred by highly-selective bitmap indices is space complexity. Indeed, it has been shown that space requirements for storing bitmaps can often exceed the database itself [18]. To deal with this issue, compressed-bitmap encodings have been proposed over the past several decades. One of the early approaches, Byte-aligned Bitmap Compression (BBC) [1], used byte-alignment to compress runs. Though seminal, BBC was later superseded by encodings that use *words* as the minimum unit of bitmap-data representation. Today, most compressed-bitmap encodings [6–10, 20, 24, 28] are variations of the popular word-aligned hybrid code (WAH) [26] and its implementation, FastBit [23]. It is worth noting that compressed-bitmap encodings not exclusively based on WAH are also in use today [3, 4]. A recent survey of known bitmap compression techniques was completed in [5].

As described above, Position-List WAH (PLWAH) [8] is one such variant of WAH PLWAH was proposed by Deliège and Pedersen to address WAH's principal weakness: a run of zeroes is often disrupted by a single dirty (set) bit, requiring a fill word to end prematurely. PLWAH cleverly encodes the position of the dirty bit using a set of reserve bits in the fill word, and therefore allowing the run of zeroes to continue. Colantonio and Di Pietro proposed Concise [6] independently and around the same time as PLWAH was introduced. Concise shares the same objectives as PLWAH of increasing compression by effectively encoding "near-fill" words. Concise also uses a position list, however that position list records the position of the dirty bit in near-fills that precedes the fill where as PLWAH encodes those that follow a fill. We would expect to see very similar speedups in query time if Concise implemented our metadata approach.

Several efforts have also explored the query optimization of established compressed-bitmap encodings. RIDBit [14] uses a two-level indexing scheme in the form of a B-tree of bitmaps. Sinha and Winslett proposed a multi-resolution bitmap index [18, 19], whereas a high resolution bitmap is further indexed by a less resolute bitmap (and so on, similar to a tree of bitmaps). This organization allows for fast accesses of hot subsets of large-scale data. Other related work show the effectiveness of multi-resolution bitmaps in large-scale scientific data applications [21, 22].

Another line of approaches toward query optimization is finer-grained, seeking to directly reduce the number of bitwise operations necessary. Enhanced WAH (EWAH) [28] splits a fill word into two halves. The most-significant half encodes a run much like WAH, but the least significant half encodes the number of succeeding literals. When performing the common bitwise & operation between two vectors, and the opposing vector contains a long run of zeroes, the information provided in the least-significant half of the current vector can be exploited to "short-circuit" the operation — allowing successive words to be skipped.

In a recent effort we collected various metadata (e.g., fill and literal patterns) per vector. The metadata, which contributed manageable space overhead, was used to inform the query processor on short-circuiting opportunities [25]. We exhibited and validated the performance benefits of EWAH, but our approach is generalizable to other encoding schemes. In this paper we (1) demonstrate the generalizability of the metadata scheme by extending it to PLWAH and (2) present two separate short-circuiting approaches specifically for PLWAH.

6 CONCLUSION AND FUTURE WORK

In this paper, we present a new PLWAH query algorithm. This algorithm uses metadata, collected at compression-time, to implement logical short-circuiting thus avoiding unneeded memory reads during the querying process. We conducted and empirical study that compared our algorithm to that of the original PLWAH and WAH query algorithm. The results of our empirical study showed that our approach outperforms PLWAH for all data sets excepted for `uniform`, which represents the worse case scenario. For the real-world data sets we studied, our approach realized an average speedup of 1.11× over WAH and 1.41× over PLWAH.

In the future, we plan to explore the effects of bitmap bit density on the efficacy of our approach. We also plan to investigate how short-circuiting can most effectively be leverage in range queries.

REFERENCES

[1] G. Antoshenkov. 1995. Byte-aligned bitmap compression. In *DCC '95: Proceedings of the Conference on Data Compression.* 476.

[2] bpa. Bonneville Power Administration, http://www.bpa.gov. (????).

[3] Samy Chambi, Daniel Lemire, Robert Godin, Kamel Boukhalfa, Charles R. Allen, and Fangjin Yang. 2016. Optimizing Druid with Roaring bitmaps. In *Proceedings of the 20th International Database Engineering & Applications Symposium, IDEAS 2016, Montreal, QC, Canada, July 11-13, 2016.* 77–86. https://doi.org/10.1145/2938503.2938515

[4] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2015. Better bitmap performance with Roaring bitmaps. *Software: Practice and Experience* (2015).

[5] Z. Chen, Y. Wen, J. Cao, W. Zheng, J. Chang, Yinjun Wu, G. Ma, M. Hakmaoui, and G. Peng. 2015. A survey of bitmap index compression algorithms for Big Data. *Tsinghua Science and Technology* 20, 1 (Feb 2015), 100–115. https://doi.org/10.1109/TST.2015.7040519

[6] Alessandro Colantonio and Roberto Di Pietro. 2010. Concise: Compressed 'n' Composable Integer Set. *Inform. Process. Lett.* 110, 16 (2010), 644–650.

[7] Fabian Corrales, David Chiu, and Jason Sawin. 2011. Variable Length Compression for Bitmap Indices. In *DEXA'11.* 381–395.

[8] F. Deliege and T. Pederson. 2010. Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps. In *EDBT'10.* 228–239.

[9] Francesco Fusco, Marc Ph. Stoecklin, and Michail Vlachos. 2010. Net-Fli: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic. *VLDB* 3, 2 (2010), 1382–1393.

[10] Gheorghi Guzun, Guadalupe Canahuate, David Chiu, and Jason Sawin. 2014. A Tunable Compression Framework for Bitmap Indices. In *IEEE International*

Conference on Data Engineering (ICDE'14).

[11] Daniel Lemire, Owen Kaser, and Eduardo Gutarra. 2012. Reordering rows for better compression: Beyond the lexicographic order. *ACM Transactions on Database Systems* 37, 3 (2012), 20:1–20:29.

[12] M. Lichman. 2013. UCI Machine Learning Repository. (2013). http://archive.ics.uci.edu/ml

[13] S. P. Lloyd. 1982. Least squares quantization in pcm. In *IEEE Transactions on Information Theory*, Vol. 28. 129–137.

[14] E. O'Neil, P. O'Neil, and K. Wu. 2007. Bitmap Index Design Choices and Their Performance Implications. In *Database Engineering and Applications Symposium*. 72–84.

[15] Patrick E. O'Neil. 1989. Model 204 Architecture and Performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*. Springer-Verlag, London, UK, 40–59. http://portal.acm.org/citation.cfm?id=645575.658338

[16] A. Pinar, T.Tao, and H. Ferhatosmanoglu. 2005. Compressing Bitmap Indices by Data Reorganization. In *ICDE'05*. 310–321.

[17] Murat Sariyar, Andreas Borg, and Klaus Pommerening. 2011. Controlling false match rates in record linkage using extreme value theory. *Journal of Biomedical Informatics* 44, 4 (2011), 648–654. https://doi.org/10.1016/j.jbi.2011.02.008

[18] Rishi Rakesh Sinha and Marianne Winslett. 2007. Multi-resolution bitmap indexes for scientific data. *ACM Trans. Database Syst.* 32, Article 16 (August 2007). Issue 3. https://doi.org/10.1145/1272743.1272746

[19] Rishi Rakesh Sinha, M. Winslett, Kesheng Wu, K. Stockinger, and A. Shoshani. 2008. Adaptive Bitmap Indexes for Space-Constrained Systems. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*. 1418–1420. https://doi.org/10.1109/ICDE.2008.4497575

[20] Ryan Slechta, Jason Sawin, Ben McCamish, David Chiu, and Guadalupe Canahuate. 2014. Optimizing Query Execution for Variable-aligned Length Compression of Bitmap Indices. In *International Database Engineering & Applications Symposium*. 217–226.

[21] Yu Su, Gagan Agrawal, Jonathan Woodring, Kary Myers, Joanne Wendelberger, and James P. Ahrens. 2013. Taming massive distributed datasets: data sampling using bitmap indices. In *HPDC*. 13–24.

[22] Yu Su, Yi Wang, and Gagan Agrawal. 2015. 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*. 61–72. https://doi.org/10.1145/2749246.2749268

[23] K Wu *et al.* 2009. FastBit: Interactively Searching Massive Data. In *SciDAC*.

[24] Sebastiaan J. van Schaik and Oege de Moor. 2011. A memory efficient reachability data structure through bit vector compression. In *ACM SIGMOD International Conference on Management of Data*. 913–924.

[25] Miguel Velez, Jason Sawin, Alexia Ingerson, and David Chiu. 2016. Improving Bitmap Execution Performance Using Column-Based Metadata. In *4th IEEE International Conference on Future Internet of Things and Cloud, FiCloud 2016, Vienna, Austria, August 22-24, 2016*. 371–378. https://doi.org/10.1109/FiCloud.2016.59

[26] K. Wu, E. J. Otoo, and A.Shoshani. 2002. Compressing bitmap indexes for faster search operations. In *SSDBM'02*. 99–108.

[27] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. 2006. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.* 31, 1 (March 2006), 1–38. https://doi.org/10.1145/1132863.1132864

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