A Tunable Compression Framework for Bitmap Indices

Gheorghi Guzun [#1] Guadalupe Canahuate [#2] David Chiu [*3] and Jason Sawin [†4]

[#] *Electrical and Computer Engineering, The University of Iowa, Iowa City, IA, USA*

[1] gheorghi-guzun@uiowa.edu.edu, [2] guadalupe-canahuate@uiowa.edu

[*] *Engineering and Computer Science, Washington State University, Vancouver, WA, USA*

[3] david.chiu@wsu.edu

[†] *Computer and Information Sciences, University of St. Thomas, St. Paul, MN, USA*

[4] jason.sawin@stthomas.edu

***Abstract*—Bitmap indices are widely used for large read-only repositories in data warehouses and scientific databases. Their binary representation allows for the use of bitwise operations and specialized run-length compression techniques. Due to a trade-off between compression and query efficiency, bitmap compression schemes are aligned using a fixed encoding length size (typically the word length) to avoid explicit decompression during query time. In general, smaller encoding lengths provide better compression, but require more decoding during query execution. However, when the difference in size is considerable, it is possible for smaller encodings to also provide better execution time. We posit that a tailored encoding length for each bit vector will provide better performance than a one-size-fits-all approach.**

We present a framework that optimizes compression and query efficiency by allowing bitmaps to be compressed using variable encoding lengths while still maintaining alignment to avoid explicit decompression. Efficient algorithms are introduced to process queries over bitmaps compressed using different encoding lengths. An input parameter controls the aggressiveness of the compression providing the user with the ability to tune the trade-off between space and query time. Our empirical study shows this approach achieves significant improvements in terms of both query time and compression ratio for synthetic and real data sets. Compared to 32-bit WAH, VAL-WAH produces up to 1.8× smaller bitmaps and achieves query times that are 30% faster.

I. Introduction

From business analytics to scientific simulations, today's applications are increasingly data-driven, and their associated data stores are growing rapidly in density. To support efficient queries over such Big Data, appropriate indexing mechanisms must be precomputed. One popular indexing technique for enabling fast query execution over large scale read-only data sets is the *bitmap index* [1], [2]. Because bitmaps can leverage fast bitwise operations supported by hardware, they have been extensively used for selection and aggregation queries in data warehouses [3], [4] and scientific applications [5], [6], [7], [8].

In a simple bitmap, each row corresponds to a tuple, and a column (bit vector) corresponds to an attribute value-range (or a bin). Bit positions are set only for tuples that satisfy the bit vector property. For categorical attributes, one bit vector can be created for each attribute value. Continuous attributes can be discretized into a set of bins and a bit vector is generated for each bin. For example, consider a relation containing objects in a spatial grid. Suppose there are three attributes: object *type* and the *X*,*Y* coordinates in which the object resides. The attribute, *type*, is a categorical attribute and thus one bit vector is created for each object type. Attributes *X* and *Y* are continuous, so the desired resolution for the grid defines the number of bins needed to properly represent each grid cell.

A query requesting objects located within a grid cell can be answered by applying an `AND` between the corresponding bit vectors. Any set bit in the resulting vector indicates an object located within the cell. If the number of objects is fixed, increasing the resolution of the grid in the running example will increase the number of bit vectors, producing a larger index, but it will also increase the bit vectors' sparsity. For this reason, it is worthwhile to compress bitmap indices.

Modern bitmap compression techniques minimize the decoding overhead during query time by allowing operations to be applied directly over the compressed bitmaps. Most of these techniques are variants of the popular Word-Aligned Hybrid (WAH) encoding [9], [10]. To ensure memory-alignment on $w = 32$ or 64-word architectures, WAH splits the bit vector into a sequence of $(w-1)$ bit *segments*, which become the unit of compression. A $(w-1)$-bit segment containing heterogeneous values is encoded into a *literal* as follows. For a w-bit word, the most significant bit is flagged 0, and the remaining $(w-1)$ bits are copied over verbatim. Conversely, a sequence of r consecutive $w-1$ bit segments containing homogeneous bit values can be encoded into a *fill*. The most significant bit is flagged 1 to denote a run of segments, the next bit denotes the value of the run, and the remaining $(w-2)$ bits are used to represent the run-length of segments, r.

Using this encoding, a maximum run of $2^{w-2} \times (w-1)$ bits can be represented by a single WAH *fill* word. However, in practice the longest runs only require several bits to be encoded. It is this overhead in representing the fills that have motivated several new techniques. Recent efforts can be classified into two separate tracks: (1) enhanced variants of WAH , and (2) arbitrary unit segment lengths for compression. While these variants share basic encoding and processing

similarities, their differences optimize various scenarios. For instance, Concise [11] and PLWAH [12] seek to improve compression ratio when a single bit interrupts a long run, while VLC [13] proposed the use of arbitrary segment lengths to improve compression but increased query execution time due to the decoding cost overhead.

We believe that a unified bitmap compression and execution framework, which can negotiate the aforementioned trade-offs, would be beneficial to the Big Data community. In this paper, we propose a novel aligned variable segment-length compression framework to achieve better compression and query execution times. Within our framework, we allow various word-aligned compression techniques to coexist and operations between bitmap vectors compressed with different methods is performed on-the-fly circumventing explicit decompression overheads. Given the data set and several input parameters from the user, our framework characterizes the data to inform on an appropriate segment length and encoding scheme. The scheme can be chosen under the conditions it performs best and combined to achieve improved storage and query time.

This paper makes the following contributions:

- A novel *generalized* and *extendible* framework for bitmap indices called Variable Aligned Length (VAL) is designed to compress bit vectors using different encoding methods and variable segment lengths while maintaining the alignment of the encoding unit segment length.
- As proof-of-concept, we have implemented WAH within the VAL context (referred to as VAL-WAH). New, efficient algorithms operate WAH compressed bitmaps encoded using different segment lengths.
- This compression framework has been parametrized with an input *tuning parameter* λ that allows users to trade-off between space and querying time. Our VAL-WAH is more efficient than 64-bit WAH in compression (up to $3.4\times$ smaller) with comparable query time performance (only 3% overhead). Compared to 32-bit WAH, VAL-WAH produces up to $1.8\times$ smaller bitmaps and achieves query times that are 30% faster.
- An extensive analysis of VAL-WAH has been performed on real and synthetic data sets with different data distributions (uniform and zipf). Space-time comparisons are performed against bitmaps compressed using several well-known encoding. Among popular schemes, we show that our compressed index is up to $50\times$ smaller than 64-bit EWAH and up to 40% faster than 32-bit PLWAH. Using harmonic mean, we combine compression ratio and query time ratio into a single metric, *gain*. Experiments over both sorted and non-sorted data show that VAL-WAH produces a positive net gain when compared to other methods. Furthermore, the gain increases with bin cardinality when compared to fixed-length WAH.

The remainder of this paper is structured as follows. In Section II, we present background and related work on bitmap compression schemes. Section III describes the proposed Variable Aligned-Length framework in depth, detailing both compression and query processing algorithms. Section IV details a proof-of-concept integration of our framework with the popular encoding scheme, WAH. In Section V, we present our experimental results for both sorted and non-sorted data. Finally, we conclude and discuss future research directions in Section VI.

II. Related Works

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) [14] was one of the first compression techniques for bitmap indices operating with byte-blocks for alignment purposes. BBC compresses the bitmaps compactly and query processing is CPU-intensive. Word Aligned Hybrid (WAH) [15] proposes the use of words instead of bytes to match the computer architecture and reduce read latency. In practice, WAH uses roughly 60% more space than BBC and can execute logical operations $12\times$ faster.

Recently, several encodings optimizing WAH have been proposed in the literature [9], [12], [6], [11], [13], [16]. Enhanced WAH (EWAH), also seen in literature as Word-aligned Bitmap Code [9], focuses on improving query time by trading-off space. EWAH uses half of a word to encode fills. The upper half (most significant bits) of the fill word encodes the flag bit, the fill value, and the run-length. The remaining bits are used to represent the number of literal words following the run encoded in the fill. Query speedup is achieved because literal words can be skipped when operating with large fills. For example, literal values can be ignored when an `AND` is applied with a fill of 0s. However, since only half of the word is used to encode the fills, a bitmap with long runs may not be compressed as efficiently.

PLWAH [12] and Concise [11] improve compression in the cases where a single dirty bit interrupts a long run. Within a fill word, $\lceil \log_2 w \rceil$ bits are reserved to indicate the position of the fill's complement bit. For Concise, this near-fill segment is the literal word *before* the fill and for PLWAH is the literal word *after* the fill. In the best case, a bitmap compressed using PLWAH/Concise can be half the size of a WAH compressed bitmap. An extension to PLWAH [17] enables "polluted blocks" to appear multiple times in a fill block by introducing a new word type called draggled fill.

To handle high rates of append operations, Compax [6] introduced two new types of words in addition to WAH's literal and fill: fill-literal-fill (FLF) and literal-fill-literal (LFL). These 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. In these cases, Compax is able to compress three WAH words into a single 32-bit word: three bytes are used to encode the original WAH words and the remaining byte indicates the type of word and the byte position in the original WAH word of the literal byte(s). In their evaluation, Compax was able to encode bitmaps using 60% less space than WAH. Although Compax was designed for 32-bits words, it is possible to extend it to variable sizes (16 and 64-bit words) by adjusting the number of bits to index the byte in the literal words and therefore it could be included in the proposed framework as well.

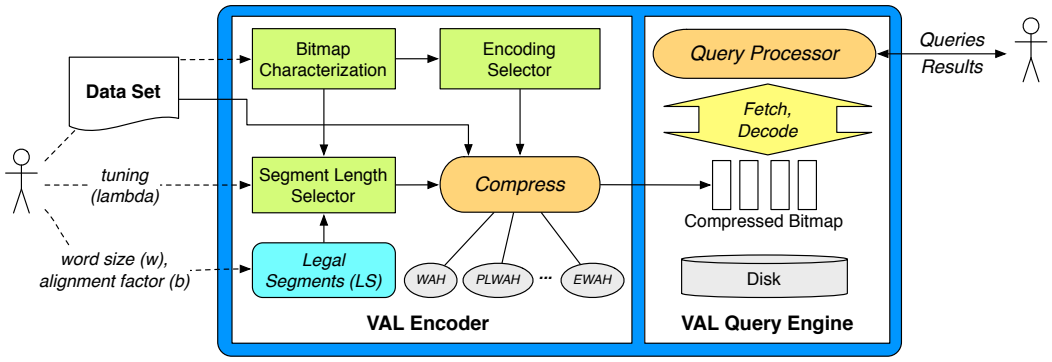


Fig. 1. The VAL System Framework: Encoder and Query Engine

Because tuples can be ordered arbitrarily in relations, reordering has been applied to maximize the length of the runs [18], [19], [20], [21], [22]. Finding an optimal order, however, has been shown to be NP-Complete [23], and different heuristics have been proposed, such as lexicographic order, Gray codes [19], and Hamming-distance-based [24], among others. Reordering produces longer runs for the first few bit vectors, but generally deteriorate into shorter runs (and worse, random noise) in the higher dimensions. Such a pattern means that WAH can achieve optimal compression for the first few columns, but the compression reduces for the later columns. This is the motivation for allowing varying segment lengths for each bit vector.

Previous efforts have also recognized the advantages in using variable segment lengths for encoding. In VLC [13], arbitrary segment lengths can be used to encode each bit vector. Performance in query execution degrades drastically when segment lengths are not aligned. Partitioned WAH (PWAH) [16] proposes to encode the bitmaps using several partitions within a 64-bit word. PWAH-8 for example, divides a word into 8 partitions and stores fills and literals using 7 bits. PWAH maintains a header in each word for all flag bits, enabling "extended fills", using shorter block lengths to represent longer runs. However, PWAH does not propose to execute queries involving bitmaps compressed using variable partition lengths. Since the partitions use unaligned segment lengths of 7, 15, or 31, queries involving bitmaps compressed with different encoding lengths will require explicit decompression.

As expected, the use of these techniques is application and data-dependent. We have designed a unified compression framework where these techniques can coexist and are used for the cases where they can improve performance the most. We developed efficient query processing algorithms over bitmaps compressed using different segment lengths, while maintaining the alignment of the segments. We have designed encoding and query processing interfaces that allow the integration of various aligned run-length compression techniques. To inform the selection of the method/segment-length encoding to be used for a particular bitmap vector, we introduce a λ parameter that captures the trade-off between compression and decoding time during query execution.

III. Variable Aligned Length (VAL) Framework

Most modern bitmap compression techniques are variants of the Word-Aligned Hybrid (WAH) encoding, which uses w-bit words to align with the underlying CPU architecture, *e.g.*, $w = 32$ or $w = 64$. While the word size w is fixed on physical constraints, there is no such requirement that the *segment length* s, *i.e.*, the unit of compression, must be fixed at $s = w - 1$. Indeed, WAH-style fills and literals can easily be represented in $s = 7$ bit segments, which is packed into the physical unit of bytes rather than words[12].

The selection of the compression method, and independently, the segment length s, are both data and application-dependent. It is observable that there exists clear scenarios in which one method outperforms the others in time and/or space. We have identified two orthogonal aspects that can be generalized: (1) the encoding segment length s and (2) the encoding method used for compression. We propose a unified bitmap compression framework, Variable Aligned Length (VAL), where these variations can coexist. Our framework inputs user preference on the space-time trade-off, and automatically applies the optimal settings to improve performance.

The proposed VAL system framework is shown in Figure 1, comprising two main components: *VAL Encoder* and *VAL Query Engine*. The user inputs the data and a set of system-specific parameters. The input data set is first characterized, *e.g.*, by profiling the overall bit distribution and length of runs. This information is sent to the *Encoding Selector* and the *Segment Length Selector*. The former selector chooses an appropriate compression encoding scheme, and the latter decides on a segment length s to be used for encoding each bit vector. After compression, the compressed index is read by the *VAL Query Engine* which handles queries over the data set. Queries can be executed over data compressed with different encoding techniques or segment lengths. An important contribution of this paper is the design of querying algorithms that can operate two bit vectors encoded with different segment lengths. These algorithms as well as the performance of the VAL Encoder are evaluated in Section V.

A. Bitmap Encoding Commonalities

To show the commonalities and generalizability of modern WAH-variant schemes, let us focus on the encodings for sev-

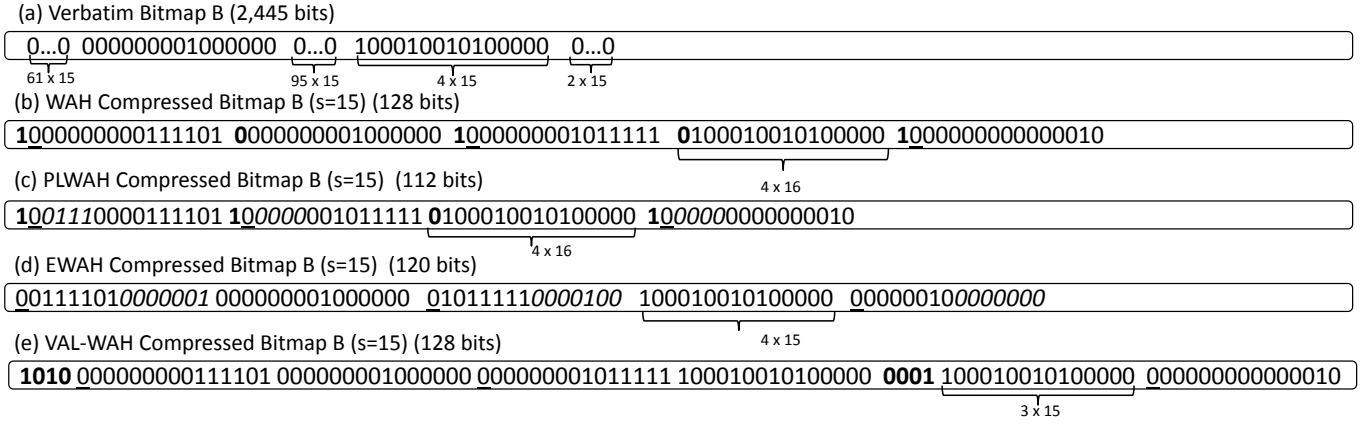


Fig. 2. Examples of Various Word Aligned Encodings

eral techniques: Word-Aligned Hybrid (WAH) [15], Enhanced WAH (EWAH) [9], and Position-List WAH (PLWAH) [12]. Figure 2 (a) shows a verbatim, uncompressed, bitmap B that will be used to drive our examples. It contains $2,445$ bits divided into a 61×15-bit run of zeros, followed by 15 clear bits with a single set bit in position 7, then another 95×15-bit run of zeros, 4×15 bits of mixed zeros and ones, and finally, a run of 30 zeros. This example uses a $w = 64$ bit architecture and a *segment length* of $s = 15$, which means that each consecutive 15-bit segment in B will be considered at a time as units of compression.

Recall that the standard version of WAH is aligned to the machine's word size w, and thus uses a segment length of $s = w - 1$. Figure 2 (b), shows a WAH-encoded bitmap using a $s = 15$ *segment length*. Instead of a word, we assume a more generalized *block* of size $w' = 16$ bits. Note that in this scenario, four $w' = 16$-bit blocks can be physically encapsulated into a $w = 64$ bit word. For each w'-bit block, the most significant bit (shown in bold) is the flag bit, *e.g.*, 1 for fills and 0 for literals. If a w'-bit block is a fill, the second most significant bit (underlined) denotes the fill bit. The remaining 14 bits in the fill block are then used to encode the run of consecutive of fill segments in the verbatim bitmap (61 for the first fill block). This is followed by a literal block, which stores the 15-bit literal segment with the 7th bit set to 1, and so on.

The next example we show is PLWAH, in Figure 2 (c). PLWAH's defining property lies in its ability to encode fills even with the presence of a single dirty bit, which would disrupt a run. Note that again, we are assuming $s = 15$, which imposes a $w' = 16$ block size. Notice that the single dirty bit (in the 7th position) in verbatim bit vector pollutes an otherwise longer run of 0s. Instead of encoding this dirty block as a literal, PLWAH uses the four *position* bits shown in *italic* to denote the position of the dirty bit, *i.e.*, $(7)_{10} = (0111)_2$. If the *position* bits for a fill block are zeros, then no literal word is integrated. This determination requires some decode overhead when processing queries. PLWAH uses $p = \lceil \log_2 s \rceil$ bits to index the dirty bit position, and therefore the maximum run-length that can be encoded with a single fill block is $2^{s-p-1} \times s$. Longer fills will require two fill words to be encoded.

Figure 2d) shows the EWAH encoding for bit vector B. For fill blocks, $p = \lfloor \frac{s}{2} \rfloor$ bits (in *italics*) are used to indicate the number of literal blocks following the fill. The maximum run-length that can be encoded with a single fill word is thus $(2^{\frac{s}{2}-1}s)$. For this reason, EWAH typically does not compress as well as the other methods. However, its improved performance during query time is significant because literal words can be skipped or appended without decoding when operated with a fill.

Finally, Figure 2e) shows the bitmap B encoded using WAH within the VAL framework (VAL-WAH). VAL packs the segments into a word using w'-bit blocks and creates a word header (in bold) that stores the flag bits, one per block within each word. By placing this header in the front, we can reduce the decoding overhead. The compression performance of VAL-WAH is the same as WAH for the same s. The compression improvements achieved by VAL come from the use of smaller segments which in general, will produce better compression than longer ones. The exception is the case where the bits used to represent a fill are not enough to represent long runs in a single block. For comparison purposes, the bitmap B from the example would require 320 bits after compression using WAH-64 ($s = 63$). For this example, WAH-64 would require $2.5\times$ more storage than VAL-WAH.

B. Bitmap Processing Commonalities

The similarities in encoding schemes also imply commonalities in query execution. Let us consider the WAH query processing algorithm and how it compares with other methods. Without loss of generality, the discussion that follows considers a query executed as the `AND` of two compressed bit vectors. An `AND` operation is performed by iterating over the words in the two bit vectors. For each WAH encoded word, the flag bit is read, and decoded into an *activeWord*. An *activeWord* is a structure that identifies the type of word (fill or literal). If the *activeWord* is a fill, then it also holds the fill bit value and the number of segments in the run. Two *activeWords* from each bit vector are queried together, until the number of segments

is exhausted. At this point, the next word is read from the cache and decoded. An *activeWord* can be interpreted as the following structure,

```
typedef struct {
   /* holds encoded word value */
   word_t val;
   /* fill-specific vars */
   bool fillBit
   int runLength;
} activeWord;
```

where `sizeof(word_t)` is equal to the machine's word size. The encoded value of the word is stored in `val`, and to determine whether an *activeWord* is a fill or literal is done by simply examining `val`'s most significant bit. Clearly, the values of `fillBit` and `runLength` are only assigned if the *activeWord* is determined to be a fill.

There are three cases when executing the `AND` between the two *activeWords*, `X` and `Y`. (Case 1) If `X` and `Y` are both fills, the result is a new fill word with its `fillBit` equal to the result of `X.fillBit & Y.fillBit`. The new fill word's `runLength` is assigned `abs(X.runLength-Y.runLength)`. (Case 2) If `X` and `Y` are both literals, then the result is a new literal word with `val` set to `X.val & Y.val`. (Case 3) Finally, if `X` is a literal and `Y` is a fill, then the number of segments in the fill word is first decremented by one: `Y.runLength--`. Afterwards, the result is a new literal word with `val` being set to the `&` result between `X.val` and the literal value of `Y.fillBit`. Bit vectors are never explicitly decoded one bit at a time. Considering each bit as a processing unit, operations of type (Case 1) observe a superlinear speedup, while operations of type (Case 2 and Case 3) observe an $s\times$ speedup, where s is the encoded segment length.

Due to the shared encoding similarities of the WAH variants, we observe that WAH's core processing algorithm can also be easily extended to process PLWAH, Concise, Compax, or EWAH with minor modifications. For PLWAH and Concise, decoding of the *activeWord* word could produce one fill and one literal when the *position* bits for the fill are not all zeros. This literal is the word either following or preceding the fill, respectively for PLWAH and Concise. The logic for query processing remains similar; the difference is to operate both the dirty literal and the fill before decoding the next word.

To query using Compax, there will be more branch operations, because it uses four types of words. For fill words, more decoding is required to decide whether the fill is of the form Fill-Literal-Fill (FLF) or Literal-Fill-Literal (LFL). In those cases, three active words will be created but the query processing logic still remains the same. The branching overhead is the trade-off for Compax's update friendly structure.

Because EWAH applies a different encoding for the fills, it does not generate multiple *activeWords* after decoding. It only stores the number of literal words following the fill, and this information is used for query optimization. When two fills are `AND`ed together and one of them is a zero fill, literal words can be skipped without decoding by incrementing the position of the vector iterator. Also, literals can be `AND`ed until the counter reaches zero without requiring any decoding. These translate into faster query execution. Nevertheless, the logic for operating literals and fill values remains relatively unchanged.

C. The Val Encoder

To generalize query processing over variable *segment lengths*, we introduce a more general *activeBlock* in lieu of an *activeWord*. An *activeBlock* shares the basic structure of an *activeWord*, except that the `activeBlock.val` considers sequences of s $(s \leq w)$ bits. When $s = w$, there is one block per word, and the structures and query processing reduce back to the original algorithm. However, for encodings using smaller segment lengths $s < w$, decoding of a physical word, would produce two or more *activeBlocks*. For example, when $w = 64, s = 15$, there would be four `activeBlocks` per physical word.

The goal of our framework is to improve compression without adversely affecting query performance. For this reason, the segment lengths s cannot be arbitrary, as we would lose the alignment benefits. Queries would suffer from considerable decoding overhead during query execution. Given the machine's word size w and an *alignment factor* b, $b \leq w$, we define the set of *Legal Segment Lengths* LS as,

$$LS = \{2^i \times (b-1) \mid 0 \leq i \leq (\log_2 w - \log_2 b)\} \quad (1)$$

On a 64-bit architecture ($w = 64$) and alignment factor $b = 16$, the legal segment lengths are, $LS = \{15, 30, 60\}$. This definition of segment lengths ensures that larger segment lengths are always multiple of smaller segment lengths, and therefore the *activeBlocks* they create are always logically aligned. To further reduce the overhead of query execution, the segments are also memory-aligned, *i.e.*, segments never cross over two physical words. For instance, segment lengths $s = 15, 30,$ and 60 encapsulate four, two, and one block(s) into a single physical word, respectively. When needed, *blocks* are padded with zeros within the physical word. For example, recall that each *block* is encoded using $s+1$ bits (the one extra bit is needed to flag the block as being either a literal or a fill).

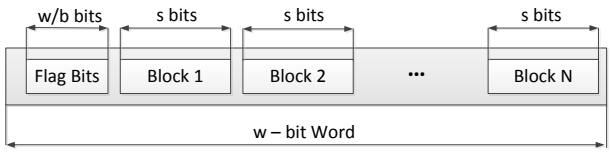


Fig. 3. Word Encapsulation

This word encapsulation scheme is shown in Figure 3. The number of blocks encapsulated into a word is given by $N = \frac{(b-1)\times w}{b\times s}$. The flag bit for each block is stored in the $\frac{w}{b}$-bit word header. The goal of the word header is to minimize the time required to align the segments between two bitmaps encoded using different block sizes. For example, two literal blocks with a VAL bitmap encoded using $s = 15$ can be directly operated with the corresponding literal block encoded using $s = 30$. It is worth noting that small alignment factors would have a significant number of unused bits for larger segments

in LS. For example, for $b = 8$ and $w = 64$, the legal segment length $s = 56$ would have 7 unused bits per word. However, it is worth noting that in many cases the increase in compression for the bit vectors encoded with smaller segment lengths will make up for these pad bits in the bit vectors encoded with larger segment lengths.

Let us consider again the system framework presented in Figure 1. Along with the data set, the user also inputs the machine's word size w, the alignment factor b, and a tuning parameter λ (explained later). First, w and b are input into Equation 1 to determine the set of legal segment lengths, LS. Next, encoding a bitmap involves two major decisions: (1) the encoding method to use, and (2) the segment length $s \in LS$. To inform on these decisions, the *Bitmap Characterization* component passes over and profiles each bit vector from the input data. This profile is used as input into both the *Encoding Selector* and the *Segment Length Selector*.

The *Encoding Selector* determines an encoding for a bit vector given its profile. For example, if the bit vector is very sparse, then PLWAH may be selected. EWAH may be preferred for noisy bitmaps that have a majority of literals and short fills. The *Segment Length Selector* uses the profile and LS to identify an appropriate $s \in LS$ to compress each bit vector. In general, bitmaps compressed using smaller segments will compress more aggressively, but may require more decoding and bookkeeping operations when executing queries. To exploit this trade-off, we allow users to input a *tuning parameter* λ, $0 \leq \lambda \leq 1$. As λ approaches 0, the system will attempt to achieve the best compression possible, while a λ approaching 1 prioritizes faster query execution time.

Given a bit vector B and λ, the *Segment Length Selector* will return

$$s = \begin{cases} s_{c+i}, & \text{if } \frac{size(B,s_c)\cdot(1+\lambda)^{1+i+\lambda}}{i+1} \geq size(B, s_{c+i}) \\ s_c, & \text{otherwise} \end{cases} \quad (2)$$

where $size(B, s)$ is the size of the bit vector B when compressed with segment length s. The term,

$$s_c = \arg\min_{s_i \in LS} \{size(B, s_i)\} \quad (3)$$

refers to the segment length that yields the most compressed bit vector. Similarly, s_{c+i} denotes the i^{th} legal segment length greater than s_c in *LS*.

After these parameters are selected, B is compressed. A *header byte* must be appended to the beginning of each compressed bit vector. The most significant four bits in the header are used to identify the multiplier m for the alignment factor b, such that $(m \times b) \in LS$ is the selected segment length. The remaining four bits encodes the method used, *e.g.*, WAH or PLWAH.

D. The VAL Query Engine

Enabling variable segment lengths complicates query processing. As discussed previously, queries are executed by performing logical bitwise operations between bit vectors. In general, the more compressed blocks are contained in the bit vector, the longer it takes to execute the query as compressed blocks need to be decoded. Using variable segment lengths can increase decoding costs. However, in the cases where both bitmaps are compressed well, there are opportunities for faster query execution by processing whole compressed blocks.

In our framework, two VAL bit vectors $X_{m\times s}$ and Y_s encoded using segment lengths $m \times s$ and s, respectively are operated together to produce bit vector Z_s, which stores the result of the bitwise logical operation $X_{m\times s} \circ Y_s$, where $\circ$ is a binary logical operator. Algorithm 1 shows the pseudocode for query processing. Each bitmap is still decoded one physical word (*currentWord*) at a time (Line 1-7). The parameter m (Line 3) indicates that the *currentWord* should be decoded into blocks of segment length $m \times s$. The decoded *currentWord* contains a number of *activeBlocks*. This *activeBlock* is tantamount to the *activeWord* structure used in WAH. The *currentWords* are iterated one block at a time (Lines 8-14) and are operated together until exhausted. Two fill blocks can be operated without explicit decompression (Lines 15-20). If one of the *activeBlocks* is a literal, then the values are operated together and the number of segments in the fill, *nSegments*, is decremented by 1 with each *getLitValue()* call (Lines 21-23).

Algorithm 1: General Bitwise Logical Operation

```
   Input: Bit Vector X, Y: (X.segLen=m × s and Y.segLen=s)
   Output: Bit Vector Z: The resulting compressed bit vector after
           performing the logical operation X ∘ Y
1  while X and Y are not exhausted do
2      if X.currentWord is exhausted and there are more words in X then
3          X.decodeNextWord(m);
4      end
5      if Y.currentWord is exhausted and there are more words in Y then
6          Y.decodeNextWord(1);
7      end
8      while X.currentWord and Y.currentWord are not exausted do
9          if X.activeBlock.nSegments = 0 then
10             X.activeBlock = X.nextBlock();
11         end
12         if Y.activeBlock.nSegments == 0 then
13             Y.activeBlock = Y.nextBlock();
14         end
15         if X.activeBlock.isFill() and Y.activeBlock.isFill() then
16             nSegments = min(X.activeBlock.nSegments,
17                             Y.activeBlock.nSegments);
18             Z.addFill(X.activeBlock.fill
19                             ∘ Y.activeBlock.fill, nSegments);
20             X.activeBlock.nSegments -= nSegments;
21             Y.activeBlock.nSegments -= nSegments;
22         end
23         else
24             Z.addLiteral(X.activeBlock.getLitValue()
25                             ∘ Y.activeBlock.getLitValue());
26         end
27     end
28 end
29 return Z;
```

Note the similarities between Algorithm 1 and the WAH processing algorithm described in Section III-A. The encoding-specific details are enabled through implementing the abstract method *decodeNextBlock()*. As a proof-of-concept, we implemented WAH within our generalized framework and optimized query execution over variable segment lengths.

IV. VAL Implementation of WAH (VAL-WAH)

In VAL-WAH, each block in a word corresponds to a WAH fill or literal segment. As discussed previously, the flag bits for all blocks in the word are placed in the word header. The number of blocks in a word depends on the segment length used for encoding. Query execution for VAL-WAH follows the generic logical operation presented in Algorithm 1. The specialization for WAH consists in the implementation of the *decodeNextWord()* method. Since several encoding segment lengths could be used, a complication during query processing is to execute queries involving bitmaps encoded using different segment lengths. This is done by parametrizing the decode method with an integer p that acts as a segment-length conversion factor. The segment length used for decoding is $2^p \times s$, where s is the segment length used for encoding. The possible values of p depend on the legal segments for encoding. For our current setup that allows segment lengths of 15, 30 and 60, p is in the interval $[-2, 2]$. There are three cases: $p < 0$, $p = 0$, and $p > 0$, which decode the current word into blocks using a smaller segment length, the same segment length, or a larger segment length than it was used for encoding. Specifically, $p = 0$ is used to decode segments with the same segment length as the encoding, $|p| = 1$ is used to convert segment lengths between 15-30 and 30-60, while $|p| = 2$ is used to convert between 15-60.

A. Converting Segment Lengths

In this section, we show how conversion between different segment lengths is performed efficiently. During query execution, this conversion is performed on-the-fly as the query is processed. The bitmaps are not re-encoded to a different segment length explicitly. As mentioned before, segment lengths in LS can be easily aligned during query execution since, larger lengths are always multiple of smaller ones. The *conversion factor* between different segment lengths can be expressed as $m = 2^{|p|}$.

Given a VAL-compressed bit vector X_s encoded using segment length s, we can easily convert X_s into $X_{s/m}$ in the following way. Consider the conversion using $p = 1, m = 2$ of X_{30} down to X_{15}. A literal block in X_{30} will translate into two literal blocks in X_{15}. Similarly, a fill block in X_{30} with segment count $nSegments$ will translate into a single block in X_{15}, with same fill bit and segment count equal to $nSegments \times m$. Note that multiplications/divisions can be done using shift-operations because m is a power of 2. The sign of p indicates whether we convert up ($p < 0$) to a larger or convert down ($p > 0$) to a smaller segment, and thus defining the direction of the shift operation.

Translating from smaller segments to larger ones is possible. For example $p = -1, m = 2$ would translate X_{30} into X_{60}. Two literal blocks in X_{30} will translate into one literal block in X_{60}. A fill block in X_{30} with segment count $nSegments$ will translate into one fill block in X_{60} with segment count equal to $nSegments \times m$. When the division has a remainder, a literal segment is generated with a literal representation of the fill value for m-fraction of the word, and the rest of this decoded literal is completed using the next word in the vector.

Algorithm 2 shows the pseudocode to decode a word from a bit vector X_s encoded using segment length s and produces the decoded *activeBlocks* using segment length m/s. There are two key data structures in this algorithm: $activeWord$ and $activeBlock$. The $activeWord$ is an array containing the new decoded blocks, *i.e.*, an $activeBlock$.

Algorithm 2: A Method for Decoding Down ($p > 0$).

Input: Compressed word containing blocks of length s; N: the number of blocks in the word; possible values 1,2, $m = 2^{|p|}$: factor of the new segment length
Output: $activeWord$ - VAL word containing decoded blocks using segments of length s/m

```
1  for i = 1 → N do
2      activeBlock = ith block;
3      if activeBlock.isLiteral() then
4          for j = 1 → m do
5              activeWord.addLiteral(activeBlock.value >>>
6                                     s × (N − i));
7          end
8      end
9      else
10         activeWord.addFill(activeBlock.value >>> (s − 1),
11                            activeBlock.nSegments ×m);
12     end
13 end
14 return activeWord;
```

For $p > 0$, when decoding is done to a smaller segment length, there are two possible cases for every block in the word being decoded:

- Case 1 (Lines 3-7): The *activeBlock* is a literal. In this case, the *activeBlock* is divided into $2^{|p|}$ (for us, 2 or 4) literal blocks with smaller segment length and added to the *activeWord* as literals. The *activeWord* will be iterated in the main query processing function.
- Case 2 (Lines 8-10): The *activeBlock* is a fill. In this case, a single fill block is added with same fill value and the number of segments is multiplied by $2^{|p|}$ using a shift-left operation.

As an illustration of Algorithm 2, consider the bit vector in Figure 4. The figure shows the conversion from $s = 30$ down to $s = 15$. First, the fill block with zeros is stored into a fill block with $s = 15$ and double the number of segments. Then the second block in the word is stored into two literals with $s = 15$. Note that, since decoding is done in memory only and $nSegments$ is a 64-bit number, the multiplication of the number of segments in the fills from larger segments lengths to smaller lengths never require extra segments to encode the fills.

Now let us consider the case where $p < 0$ for converting from a smaller segment length s up to a larger segment length $s \times m$. The pseudocode for this decoding algorithm is shown in Algorithm 3. Here, it is required to add one more structure that allows us to temporarily store partial words. *alignedBlock* also serves as a buffer for storing leftover bits from previous blocks, when the division of *nSegments* by m has a non-zero remainder. For every block contained in the word being decoded, we have several cases:

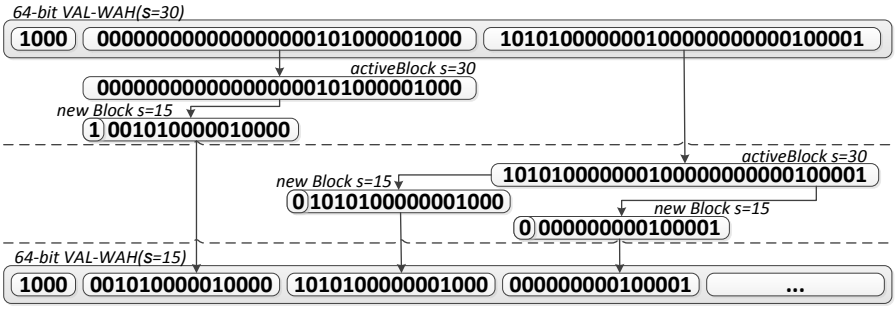


Fig. 4. Example of converting a 64-bit VAL compressed bit vector using *s=30* down to a 64-bit VAL compressed bit vector using segment *s=15*.

- Case 1 (Lines 3-6): The *alignedBlock* is empty, *i.e.*, there are no bits left from the previous decoded block or word. If the *activeBlock* is a literal, then it is added to the *alignedBlock* as a literal and the alignedBlock is marked to be incomplete. The *alignedBlock* is not yet added to the *activeWord*.
- Case 2 (Lines 7-11): The *alignedBlock* is empty and the *activeBlock* is a fill, then its fill is added to *allignedWord* with a factor of m less *nSegments*. The *alignedBlock* used to store the remainder bits from the division by m. They are stored as s bits in a literal.
- Case 3 (Lines 12-18): The *alignedBlock* is not empty, *i.e.*, there are leftover literal bits from a previous block, and *activeBlock* is also a literal. In this case, the *activeBlock* is appended to *alignedBlock*. If the *alignedBlock* is filled with $s \times m$ bits, then it is added to the *activeWord* and cleared.
- Case 4 (Lines 19-25): The *alignedBlock* is a literal and *activeBlock* is a fill. In this case, the *alignedBlock* is appended with literals from the *activeBlock* until it is filled with $s \times m$ bits. The *alignedBlock* is appended to *activeWord* as a literal block. Then the remaining blocks in *activeBlock* are appended to *activeWord* as a fill. If there are any leftover bits from dividing the remaining blocks by m, then they are stored as a literal in *alignedBlock*.

Aligning blocks with different compression lengths poses a small overhead in the VAL query processing algorithm. However, in general, VAL compresses better and often requires fewer iterations to complete the query, as described in Algorithm 1. This translates into performance benefits not only in terms of compression ratio, but also in terms of total query time execution when compared to WAH.

Algorithm 3: A Method for Decoding Up ($p < 0$).

Input: Compressed word containing blocks of length s; N: the number of blocks in the word; possible values 1,2, $m = 2^{|p|}$: factor of the new segment length
Output: $activeWord$ - VAL Word containing decoded blocks of length $s \times m$

```
1  for i = 1 → N do
2      activeBlock= i'th block;
3      if alignedBlock.nSegments=0 then
4          if activeBlock.isLiteral() then
5              alignedBlock.addLiteral(activeBlock.value)
6          end
7          else
8              activeWord.addFillBlock(activeBlock.fill,
               activeBlock.nSegments/m);
9              store the leftover bits in alignedBlock, if any
10         end
11     end
12     else
13         if activeBlock.isLiteral() then
14             alignedBlock.addLiteral(activeBlock.value);
15             if alignedBlock.isComplete() then
16                 activeWord.addLiteralBlock(alignedBlock.value)
                   alignedBlock.clear()
17             end
18         end
19         else
20             while alignedBlock.isNotComplete() do
21                 alignedBlock.addLiteral(activeBlock.fill)
                   activeBlock.nSegments- -
22             end
23             activeWord.addLiteralBlock(alignedBlock)
               alignedBlock.clear()
               activeWord.addFill(activeBlock.fill,
               activeBlock.nSegments/m) store the leftover bits in
               alignedBlock, if any
24         end
25     end
26 end
27 return activeWord
```

V. Experimental Results

In this section, the performance of VAL bitmap compression framework is evaluated over both real scientific and synthetically generated data. We first describe the experimental setup and the data sets used. Then we show the effectiveness of the tuning parameter λ to trade-off size against query time and then evaluate the impact on performance of the alignment imposed over the segment lengths by comparing to *Variable Length Coding* (VLC). We then compare the performance of VAL-WAH on sorted bitmaps with other bitmap compression schemes: *Position List Word Aligned Hybrid* (PLWAH), *Enhanced Word Aligned Hybrid* (EWAH), and *Word Aligned Hybrid* (WAH). Finally, we introduce a Gain metric computed as the harmonic mean of compression and query time ratios (normalized using the performance for verbatim or uncompressed bitmaps) to evaluate the combined performance.

A. Experimental Setup

The synthetic data sets were generated using two different distributions: `uniform` and `zipf`. These two distributions are representative of real-world bitmaps because continuous attributes and even high cardinality attributes are transformed using discretization (or binning) before creating the bitmap indices. The distribution of the data into bitmap indices depends on the method used for discretization. For example, if equi-populated bins (containing roughly the same number of objects) are used, the distribution of the bitmaps is uniform. However, if the discretization method is based on clustering, or on the density of data, then the bins would follow a skewed distribution. The zipf distribution generator assigns each bit a probability 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 data sets for $f = 1$ and $f = 2$.

Unless otherwise noted, the synthetic bitmap index contains 10 million rows and 100 bit vectors (4 attributes each with a binning cardinality of 25). We also included two real data sets in our experiments:

- `kddcup`[1]. 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. Continuous attributes were discretized into 25 bins.
- `berkeley earth`[2]. The Berkeley Earth Surface Temperature Study has created a preliminary merged data set by combining 1.6 billion temperature reports from 16 preexisting data archives. For our experiments we use a subset containing $14,786,160$ rows and 7 attributes. Each attribute was discretized with up to 25 bins.

Discretization over these real data sets was done using the least squares quantization method [25]. For sorted data experiments, Gray-code ordering [19] was used to reorder the bitmaps.

All experiments were executed over a machine with an Intel Core i7-2600 processor (8MB Cache, 3.20 GHz) and 8 GB of memory, running Windows 7 Enterprise. Our code was implemented in Java for relative performance measurements with open-source implementations of EWAH and Concise.

A set of 500 queries over two randomly selected columns (from different attributes) was generated. Each query applies a logical `AND` over the corresponding bitmaps. For query execution time, we do not take into account the time required to load the bitmaps into memory. Note that comparatively, this presents an advantage for the techniques that produce larger bitmaps (EWAH and WAH64). However, since all bitmaps fit into main memory, loading time can be done once and amortized over a large number of queries. The query set was executed six times, and the result for the first run was discarded to prevent Java's *just-in-time* compilation from skewing results. The times from the other five runs were averaged and reported.

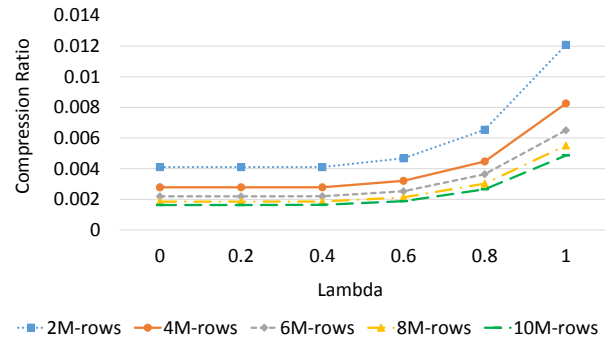
For all experiments, we assume $w = 64$ bits, the alignment factor $b = 16$, and λ parameter is set between 0 and 1 to choose among $s \in \{15, 30, 60\}$. The query processing involving two bitmap vectors encoded using different segment lengths would convert the larger encoding down to the smaller one, as presented in Algorithm 2.

[1]http://archive.ics.uci.edu/ml/datasets/KDD+Cup+1999+Data

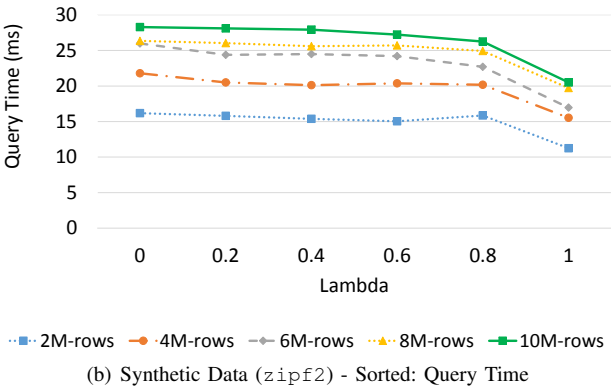
[2]http://berkeleyearth.org/dataset/

B. Tuning Parameter λ

The parameter λ allows users to tune the aggressiveness of our compression technique. Specifically, $\lambda \to 0$ prioritizes on the index size, while $\lambda \to 1$ prioritizes speed.



(a) Synthetic Data (`zipf2`) - Sorted: Compression Ratio



(b) Synthetic Data (`zipf2`) - Sorted: Query Time

Fig. 5. VAL: Effects of λ on Compression and Query Time: Synthetic Data

Figure 5 shows the effect of λ over the compression ratio and query time for a synthetic sorted data set with *zipf-2* distribution. Figure 5(a) shows the compression ratio as λ is varied from 0 to 1 by increments of 0.2. The compression ratio is computed as $size_{compressed}/size_{verbatim}$. As expected, compression ratio increases (*i.e.*, we lose compression) as λ increases, which translates into faster query times as captured in Figure 5(b). Similar results were also observed for the real data sets and for non-sorted data.

When $\lambda = 0.0$, VAL uses the smallest segment length ($s = 15$) for encoding over 70% of the columns, which reduces to little over 20% for $\lambda = 0.8$. Increasing λ allows more columns

to compress with larger segments (*e.g.*, 30 and 60), translating into speedups because less decoding is required, even when these columns are queried together with columns compressed using $s = 15$.

For the remainder of this section, to facilitate the interpretation of the results, we denote VAL with λ using VALλ, *e.g.*, VAL0 means $\lambda = 0$.

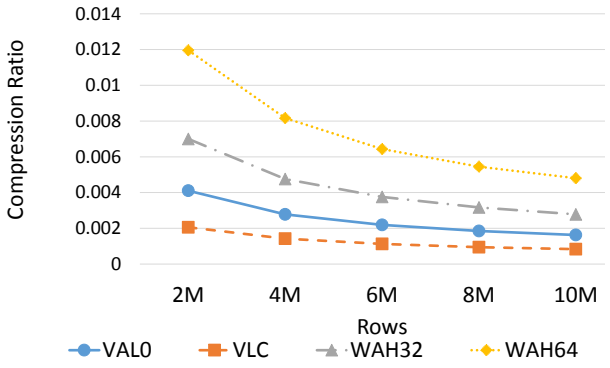
C. Evaluating Segment Alignment

This set of experiments demonstrates the importance of enforcing the logical alignment of bit vectors for query processing efficiency. Figure 6 shows the compression ratio and query execution time when comparing VAL0 to VLC and WAH (using 32-and 64-bit words). VLC is allowed to choose segment lengths $s \in \{7, 14, 21, 28\}$. As can be seen, VLC achieves the best compression due to the shorter length options. Its size represents only $55\% - 75\%$ of VAL0. However, during query time, VLC performs 3 to 4 times worse than VAL0. VAL0, on the other hand, achieves better compression performance than both WAH32 and WAH64 with comparable query times to WAH64, and always faster than WAH32. Note that in general, VAL-WAH can rarely be faster than WAH64 because WAH64 does not decode *activeBlocks*. VAL-WAH shows considerable improvement in compression ratio (less than half the size of both WAH32 and WAH64) without any degradation in query execution performance, which provides evidence for the effectiveness of our framework.

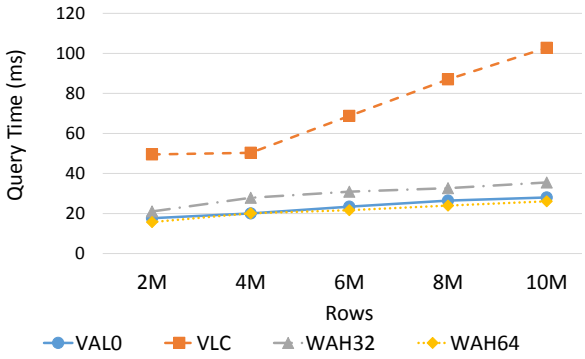
D. Comparison over Sorted Data

We now compare our approach against WAH 32-bit, WAH 64-bit, PLWAH 32-bit, EWAH 32-bit, and EWAH 64-bit. In these experiments, we only show VAL-WAH results for $\lambda = 0.2$. We compare using both real data sets, and the zipf1 and zipf2 synthetic data sets with 10 million rows. To provide a reader-friendly comparison, we normalized the results using the verbatim bitmaps for query time in the same way as compression ratio. We compute query time ratio as: $query\_time_{compressed}/query\_time_{verbatim}$. Therefore, for both compression ratio and query time ratio, smaller values imply better performance.

Figure 7 shows the comparison for sorted synthetic data. As can be seen, for all three distributions, VAL offers the best compression ratio. The greater improvement is for uniform distributions, because for skewed distributions, WAH and PLWAH are also able to compress effectively. VAL0.2 compressed bitmap sizes varying between $60\% - 80\%$ of the size of PLWAH and $55\% - 70\%$ of WAH32. Although EWAH does not compress well, Figure 7(b) shows that it offers the best query time for all distributions. In terms of query time, VAL0.2 is up to 25% faster than WAH32 and up to 15% faster than PLWAH. For larger values of λ, sometimes VAL-WAH can be faster than WAH64. For instance, when we chose $\lambda = 0.9$ on the kddcup sorted data set, VAL-WAH was 5% faster than WAH64, and this is due to a slightly smaller size of the compressed bitmap. Figure 8 shows the same experiment for the real data sets. The same patterns emerged for real

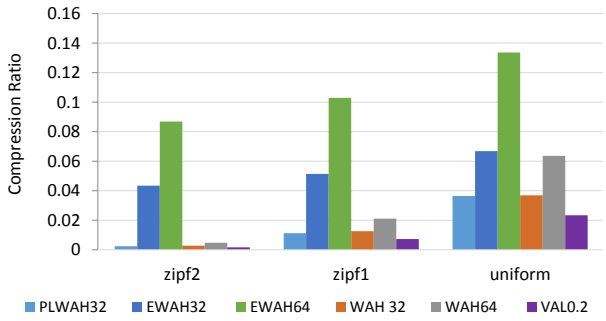


(a) Synthetic Data (zipf2) - Sorted: Compression Ratio

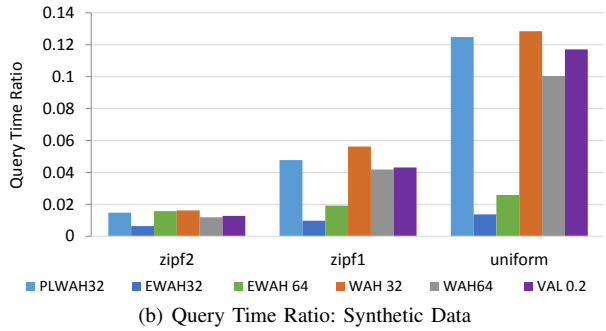


(b) Synthetic Data (zipf2) - Sorted: Query Time

Fig. 6. Effects of segment alignment on Compression and Query Time: Synthetic Data



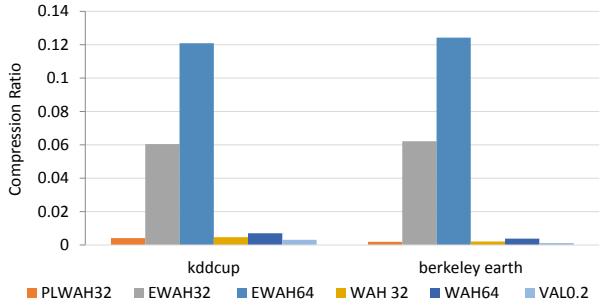
(a) Compression Ratio: Synthetic Data



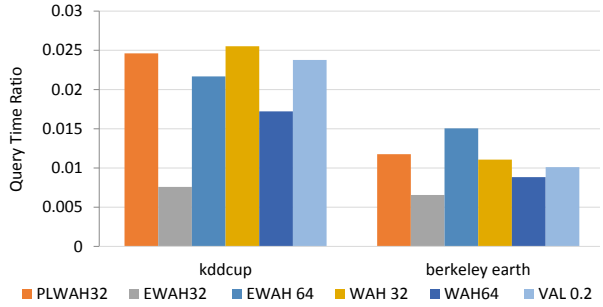
(b) Query Time Ratio: Synthetic Data

Fig. 7. Compression and Query Time Ratio Comparison - Synthetic Data: Sorted

data as for synthetically generated data. VAL-WAH offers the best compression and better query time than WAH32. EWAH has the best performance in terms of query time, however this is mostly due to a difference in implementation. In our experiments we used *ArrayLists* for storing the bit vectors in all encoding techniques except EWAH. For EWAH, we compared using the implementation offered by [18].



(a) Compression Ratio: Real Data



(b) Query Time Ratio: Real Data

Fig. 8. Compression and Query Time Ratio Comparison - Real Data: Sorted

To help simplify the discussion on trade-off, we combine compression ratio and query time ratio into a single metric, *gain*. Presuming that speedup and compression rates are equally weighed, we can use the *harmonic mean* H_M of the two ratios,

$$gain = \frac{1}{H_M} = \frac{query\_ratio + compression\_ratio}{2 \times query\_ratio \times compression\_ratio}$$

Because the harmonic mean emphasizes the smaller ratio, it captures the combined rate of speedup and compression more faithfully than an arithmetic mean. Furthermore, we inverted H_M so that the larger values imply better performance, and the goal would be to show higher gain. The *gain* across all data sets is presented in Figure 9. The combined gain of VAL-WAH is higher than the other encoding methods.

E. Results over Non-Sorted Data

Although the motivation for encoding bitmaps using different segment lengths came from observations of the run-length deterioration for sorted data, performance gains can also be obtained for non-sorted data, as seen in Figure 10. In this case, EWAH is clearly the best encoding, followed by VAL0.2. This is due to a better query execution time given that EWAH uses

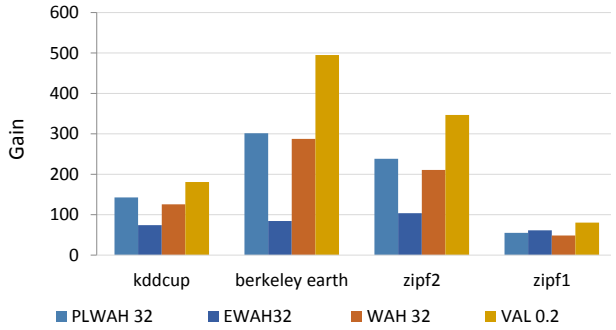


Fig. 9. Combined Gain for Sorted Bitmaps

Arrays for storing the bit vectors, while all the other techniques use *ArrayLists*. Because runs are very short, neither WAH nor PLWAH are able to compress very effectively.

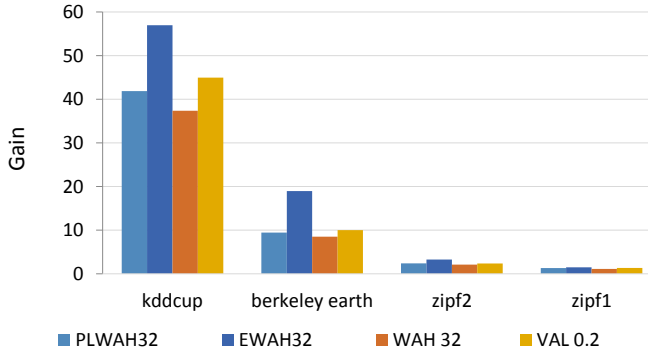


Fig. 10. Combined Gain for Non-sorted Bitmaps

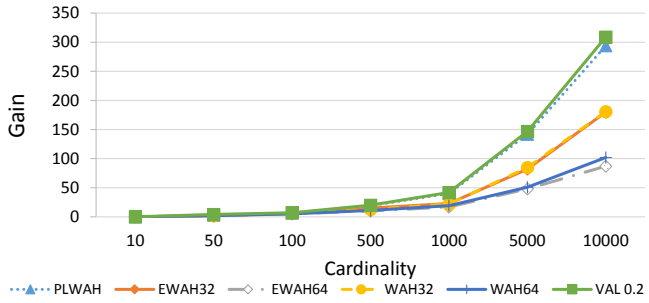


Fig. 11. Combined Gain for Non-sorted Bitmap over Cardinality

To evaluate the effect of cardinality on *gain*, we generated a zipf1 data set with 10 million rows and two attributes of increasing cardinality from 10 to 10K values. Figure 11 shows the gains obtained from this experiment. Larger cardinality produces bitmaps that are more sparse. As expected, EWAH has the smallest gain, followed by WAH. The reason is that, for sparse bitmaps, EWAH does not compress as effectively as WAH, and the query times are not much better than those of WAH. Because VAL0.2 and PLWAH are able to compress better, in

this case, they also require less decoding (fewer words result in fewer iterations in Algorithm 1). Still, `VAL0.2` has up to 3% more gain over `PLWAH32`.

In summary, no encoding is better than all others. However, these results provide evidence that there are specific scenarios in which one method should be preferred over the others. We have shown that by integrating WAH into VAL, we can improve compression without adverse implications for query performance. Similar improvements are expected for the other encodings once they are integrated into our VAL framework. Furthermore, all methods combined can achieve greater compression and better query time than when only individual methods are applied. We believe these results provide compelling evidence to support our position that a unified bitmap encoding and querying framework is desirable.

VI. Conclusions and Future Work

In this paper we introduced a tunable framework for variable aligned bitmap compression. Our framework enables several run-length aligned compression algorithms to coexist together and extends them to allow variable segment lengths. Efficient query execution algorithms over bitmaps compressed using different encoding lengths are proposed. A user-defined λ parameter allows users to adjust the trade-off between compressed index size versus expected query execution speed. As a proof of concept we implemented WAH within the framework and performed an extensive performance evaluation. This VAL-WAH approach is very effective for sorted data, particularly for skewed data distributions and is able to outperform WAH. We also show that net gains can be obtained when applying this framework to non-sorted data especially for high-cardinality attributes. The flexibility of having variable segment lengths but still maintaining the alignment of the blocks and the segments in the bitmap are the key of the success of the proposed framework.

For future work, we can implement other word-aligned bitmap compression techniques within VAL. We plan to improve the decision process for the method and segment length using the λ parameter and gathering bitmap statistics to make an informed decision considering the expected performance of the different encodings and estimated bitmap size with variable segment lengths. Given the improvements observed from incorporating WAH into the framework and the performance results for fixed-length PLWAH and EWAH in our experiments, we are confident that adapting other methods into this generalized framework will translate into further improvements in both compression size and query execution time.

References

[1] H. K. T. Wong, H. fen Liu, F. Olken, D. Rotem, and L. Wong, "Bit transposed files," in *Proceedings of VLDB 85*, pp. 448–457, 1985.

[2] I. Spiegler and R. Maayan, "Storage and retrieval considerations of binary data bases.," *Inf. Process. Manage.*, vol. 21, no. 3, pp. 233–254, 1985.

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

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

[5] K. W. *et al.*, "Fastbit: Interactively searching massive data," in *SciDAC*, 2009.

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

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

[8] 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.

[9] 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.

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

[12] F. Deliege and 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)*, pp. 228–239, 2010.

[13] F. Corrales, D. Chiu, and J. Sawin, "Variable Length Compression for Bitmap Indices," in *DEXA'11*, (Berlin, Heidelberg), pp. 381–395, Springer-Verlag, 2011.

[14] G. Antoshenkov, "Byte-aligned bitmap compression," in *DCC '95: Proceedings of the Conference on Data Compression*, (Washington, DC, USA), p. 476, IEEE Computer Society, 1995.

[15] K. Wu, E. J. Otoo, and 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)*, pp. 99–108, 2002.

[16] 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.

[17] D. K. Andreas Schmidt and M. Beine, "A proposal of a new compression scheme for medium-sparse bitmaps," *International Journal On Advances in Software*, vol. 4, no. 3 and 4, pp. 401–411, 2012.

[18] D. Lemire, O. Kaser, and K. Aouiche, "Sorting improves word-aligned bitmap indexes," *Data and Knowledge Engineering*, vol. 69, pp. 3–28, 2010.

[19] A. Pinar, T.Tao, and H. Ferhatosmanoglu, "Compressing bitmap indices by data reorganization," in *Proceedings of the 2005 International Conference on Data Engineering (ICDE'05)*, pp. 310–321, 2005.

[20] O. Kaser, D. Lemire, and K. Aouiche, "Histogram-aware sorting for enhanced word-aligned compression in bitmap indexes," in *ACM 11th International Workshop on Data Warehousing and OLAP*, pp. 1–8, 2008.

[21] T. Apaydin, A. c. Tosun, and H. Ferhatosmanoglu, "Analysis of basic data reordering techniques," in *International Conference on Scientific and Statistical Database Management*, pp. 517–524, 2008.

[22] 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.

[23] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in *Proceedings of Supercomputing*, 1999.

[24] H. H. Malik and J. R. Kender, "Optimizing frequency queries for data mining applications," in *International Conference on Data Mining*, pp. 595–600, 2007.

[25] S. P. Lloyd, "Least squares quantization in pcm," in *IEEE Transactions on Information Theory*, 1982.