Variable Length Compression for Bitmap Indices

Fabian Corrales David Chiu[†] Jason Sawin

Department of Mathematics and Computer Science, University of Puget Sound
[†] School of Engineering and Computer Science, Washington State University

Abstract. Modern large-scale applications are generating staggering amounts of data. In an effort to summarize and index these data sets, databases often use *bitmap indices*. These indices have become widely adopted due to their dual properties of (1) being able to leverage fast bit-wise operations for query processing and (2) compressibility. Today, two pervasive bitmap compression schemes employ a variation of run-length encoding, aligned over bytes (BBC) and words (WAH), respectively. While BBC typically offers high compression ratios, WAH can achieve faster query processing, but often at the cost of space. Recent work has further shown that reordering the rows of a bitmap can dramatically increase compression. However, these sorted bitmaps often display patterns of changing run-lengths that are not optimal for a byte nor a word alignment. We present a general framework to facilitate a variable length compression scheme. Given a bitmap, our algorithm is able to use different encoding lengths for compression on a per-column basis. We further present an algorithm that efficiently processes queries when encoding lengths share a common integer factor. Our empirical study shows that in the best case our approach can out-compress BBC by 30% and WAH by 70%, for real data sets. Furthermore, we report a query processing speedup of $1.6\times$ over BBC and $1.25\times$ over WAH. We will also show that these numbers drastically improve in our synthetic, uncorrelated data sets.

1 Introduction

Many research projects, vital for advancing our understanding of the world, have become prohibitively data-intensive. For example, exploration within bioinformatics generates terabytes of data per day [22]. In the field of high energy physics, the Large Hadron Collider at CERN is projected to generate 15 petabytes of data annually [7]. To facilitate data analysis, such projects may store their results in databases which employ advanced indexing techniques. Since the bulk of this scientific data is read-only, infrequently updated, and relatively easy to categorize into groups, bitmaps [18, 15] are highly amenable and widely used to index such data sets.

A bitmap index is a two dimensional array $B[m, n]$ where the columns denote a series of n *bins* and the rows correspond to m tuples in a relation. To transform a table into a bitmap, each attribute is first partitioned into a series of bins that might denote a point or a range of values. An element $b_{i,j} \in B = 1$ if the jth attribute in the ith tuple falls into the specified range, and 0 otherwise. While their representation can be large in terms of space, bitmaps can be queried using highly efficient low-level bitwise operations. To exemplify, consider an age attribute that might be partitioned into the following three bins: $a_1 = [0, 20]$, $a_2 = [21, 40]$, $a_3 = [41, \infty]$.

Table 1 shows a simple bitmap index on two attributes: age and $gender$ after binning.To find everyone under 21 or over 40, the processor can simply apply a bitwise OR of a_1 and a_3, then retrieve the succeeding tuples t_2 and t_3 from disk.

Tuples	**Bins**				
	Age			Gender	
	a_1	a_2	a_3	g_f	g_m
t_1	0	1	0	0	1
t_2	1	0	0	1	0
t_3	0	0	1	0	1

Table 1. An Example Bitmap

The tradeoff to supporting such fast querying, is storage costs. Bitmap indices can become very large. But fortunately, they are typically sparse and highly suitable for compression via run-length encoding (RLE), where bit sequences can be summarized using a nominal most significant bit (MSB) followed by the length of its run. A well-known problem of RLE is the fact that it may sometimes yield larger bitmaps. For instance, assuming a byte-based RLE, uniformly distributed bit patterns, such as (1010), will be "compressed" into $(10000001\ 00000001\ 10000001\ 00000001)$. Clearly, the original bit sequence could be more efficiently stored as a *literal*. Today's compression schemes utilize a run-length *hybrid*, i.e., a sequence of encoded bits can denote either a *literal* or a *run/fill* of n bits.

While the overhead of decoding initially appears to conflict with query performance, current memory-aligned compression schemes have found ways to improve both compression and query performance simultaneously [2, 20]. The Byte-aligned Bitmap Compression (BBC) [2] and the Word Aligned Hybrid Code (WAH) [20] are commonly used to compress and query bitmaps in databases. While both BBC and WAH are memory-aligned, they differ in granularity (byte versus word), which affects compression ratio and query execution time. As such, BBC typically generates smaller bitmaps, and WAH, due to its word-based representation, excels at compressing extremely long runs and allows for faster queries by amortizing multiple reads to extract bytes from words.

Because both BBC and WAH are run-length dependent, and tuple ordering is arbitrary in a database, efforts in row reorganization can be leveraged to produce longer runs and achieve better compression. One drawback to certain tuple reorganization schemes, such as Gray code ordering, is that average run length degrades as dimensionality increases [16]. In Gray code ordering the first several bit vectors of a bitmap may contain very long runs, but they become progressively shorter in each additional bit vectors. In fact, the highest dimensional bit vectors may still exhibit a uniform distribution of bits. We posit that it may be sensible to apply coarser encodings (e.g., WAH) to the initial bins for aggressive compression of longer runs. As dimensionality increases, and runs gradually become shorter, progressively finer encoding schemes may be used to achieve greater compression. In the highest dimensionality, where runs are even less infrequent, it may again be favorable to coarsen the encodings, because word-aligned encodings can store and process literals more efficiently.

This paper makes the following contributions:

– We have designed and implemented a novel *generalized* framework, Variable Length Compression (VLC), for encoding variably granular bitmap indices. Given a bitmap, our VLC is able to use different encoding lengths for compression on a per-column basis.
– We have designed an algorithm to compress bit vectors, which inputs a *tuning parameter* that is used to tradeoff encoding space and querying time.
– We have conducted an extensive analysis of VLC on several real data sets and compare results against state-of-the-art compression techniques. We show that VLC can out-compress Gray code ordered BBC and WAH by 30% and 70% respectively in the best case, on real data sets. We also report a speedup of $1.6\times$ over BBC and $1.25\times$ over WAH.

These contributions provide a method for faster querying on large databases as well as greater compression ratios. Our proposed VLC scheme also allows users tune the compression of their bitmap indices. For example, if certain columns are to be queried at higher rates they can be compressed using the larger encoding lengths to achieve faster queries. To maintain compression efficiency the less frequently queried columns can be compressed with smaller encoding lengths.

The remainder of this paper is organized as follows. In Section 2, we present the necessary background on bitmap compression, including BBC, WAH, and tuple reordering. Section 3 describes our Variable Length Compression framework in depth, detailing both compression and query processing algorithms. We present our experimental results in Section 4. Section 5 discusses related efforts in bitmap compression, and we conclude our findings in Section 6.

2 Background

Bitmap compression is a well-studied field, with its roots anchored in classic run-length encoding (RLE) schemes. However, traditional run-length techniques cannot be directly applied to bitmap indices because the bit vectors must first be decompressed to answer queries. This overhead would quickly dominate query processing time. Therefore, it is highly desirable to have run-length compression schemes that can answer queries by directly examining the bit vectors in their compressed state. In this section we present the background on current techniques used to compress bitmap indices that achieve this fast querying.

2.1 Byte-aligned Bitmap Code (BBC)

Run-length encoding schemes achieve compression when sequences of consecutive identical bits, or "runs", are present. BBC [2] is an 8-bit hybrid RLE representation in the form of a *literal* or a *fill*. The MSB, known as the *flag* bit, marks the encoding type. In turn, a byte $0xxxxxxx$ denotes that the least significant 7 bits is a literal representation of the actual bit string. In contrast, $1xnnnnnn$ encodes a fill which compactly represents runs of consecutive x's. Here, x is the *fill bit* which encodes the value of the bits in the run, and the remaining 6 bits are used for the length (in multiples of 7), e.g., 11001010 represents the sequence of 70 1's.

BBC is compelling in that the query execution time is directly proportional to the rate of compression. For example, suppose a database contains 77 rows and two bit vectors: v_1 and v_2. Assume that v_1 contains the literal 0101010 followed by a run of 70 consecutive 1's. Let v_2 contain a sequence of 70 0's followed by the literal 0100000. In BBC format, v_1 would be encoded as $(00101010\ 11001010)$ and similarly, $v_2 = (10001010\ 00100000)$. Now envision a query which invokes $v_1 \wedge v_2$. The query processor would read the first byte from both v_1 and v_2. By decoding the most significant bit, the query processor determines that it has read a 7-bit literal from v_1 and a run of $(10 \times 7) = 70$ 0's from v_2. Next, the literal from v_1 is `AND`'ed with a fill of seven 0000000 from v_2. Progressing further, the query processor reads and decodes the next byte from v_1. It is important to note that only seven 0's have been processed from the fill in v_2. Thus, all that is required is simply decrement of the fill count from 10 to 9. This demonstrates why BBC fills must be a multiple of 7. The next byte of v_1 is decoded as a run of 70 consecutive 1's. The next 9 `AND` operations can be carried out in one step by making the `AND` comparison once and reporting its results in the same compressed form. The run-length count for v_1 is updated to 1, and v_2 to 0. Thus $63 = (9 \times 7)$ bits have been compared without having to decode even once. After the $9th$ iteration, v_2's fills are exhausted, prompting a read of the next byte from v_2. Finally, the remaining 7 bits from both bins are `AND`'ed to complete the query. BBC's efficiency comes from the presence of fills, which effectively allows the processor to amortize the number of necessary memory accesses.

2.2 Word-Aligned Hybrid Code (WAH)

WAH [20, 19], unlike BBC, uses a 31 bit representation (32 bits including the flag bit). This representation offers several benefits over BBC—one being that for certain bitmaps, WAH can achieve significant speedup in query processing time when compared to BBC. This speedup is due to the fact that memory is typically fetched by the CPU a word at a time. By using a word-aligned encoding, WAH avoids the overhead of further extracting bytes within a word that is incurred by BCC. Thus, WAH not only compresses literals more efficiently than BBC (using 4 less flag bits per 31 bits), but it can also process bitwise operations much faster over literals by avoiding the overhead of byte extraction and parsing/decoding to determine if the byte is indeed a literal.

In terms of compressing runs, however, WAH typically pales compared to BBC. This is often due to the fact that WAH's fills can encode $2^{30} - 1$ multiples of 31 consecutive identical bits (i.e., a maximum fill length of 33,285,996,513). In practice, runs of this size are unlikely, which implies that many of the fill bits are unused. On the other hand, note that the maximum number of consecutive bits that a BBC fill can represent is $(2^6 - 1) \times 7 = 441$. In large-scale or highly sparse databases, it is likely that a run can continue far beyond this threshold, which means there can still be cases where WAH will yield more efficient encodings for runs.

2.3 Row Reordering of Bitmaps

As described above, WAH and BBC can achieve greater compression for bitmaps that contain longer average run-lengths.

Recent work has shown that the average run-length of a bitmap can be greatly improved by reordering the rows [12, 16, 10, 3]. Finding an optimal order of rows, however, has been proven to be NP-Complete, and lexicographical and Gray code ordering are widely used heuristics. Pinar, Tao, and Ferhatosmanoglu showed that compression ratio's can be improved by a factor of 10 for some bitmap indices if a Gray code (i.e., consecutive rows differ only by a single bit) ordering is applied [16]. Figure 1 shows the effects of lexicographical and Gray code ordering on bitmaps containing 3 vectors v_1, v_2, v_3. The white space represents 0's and the black represents 1's. Notice that both reordering algorithms tend to produce longer runs in the first few bit vectors, but deteriorate into shorter runs (and worse, a random distribution) of bits for the higher vectors.

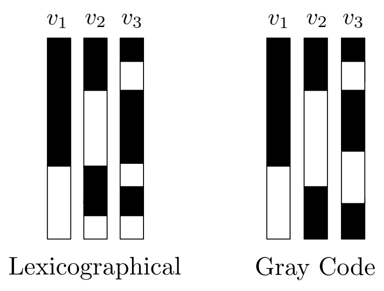


Fig. 1. Row Ordering Techniques

In situations like this, it would be desirable to employ a varying sized compression scheme for each bit vector. In this work, we assume that row reordering is a preprocessing step, and we implemented Gray code ordered bitmaps in our experimental results.

3 Variable Length Compression

In this section, we initially discuss how using variable bit-segment lengths presents opportunities to improve compression ratios beyond current state-of-the-art techniques, e.g., BBC and WAH. We then describe our compression technique, *Variable Length Compression (VLC)*, in detail.

Due to their use of fixed bit-segment lengths to encode bit vectors, neither WAH nor BBC generate optimal compression. To exemplify, recall that row reordered bitmaps produce long runs in the first several bit vectors, buts increasingly shorter runs in the later vectors. WAH's 31-bit *segment length* (32 bits including the 1 flag bit) is ideal for the first several bit vectors that potentially contain extremely long runs But after these first few vectors, the rest might tend to have an average run-length smaller than 62 (the shortest run-length multiple that WAH can compress), there is a higher likelihood that many shorter runs must be represented as WAH literals, which squanders compression opportunities. Conversely, BBC's maximum fill code, $1x111111$, can only represent a run of $63 \times 7 = 441$ x's. With its 7-bit fixed *segment length*, BBC cannot efficiently represent the long runs of the first several vectors. Any run longer than 441 would thus require another byte to be used.

We posit that we can attain a balanced tradeoff between these representations by using variably-sized bit *segment lengths*. To this end, we propose a novel run-length compression scheme *Variable Length Compression (VLC)* that can vary the segment lengths used for compression on a per bit vector basis. The flexibility of VLC enables us to compress the initial bit vectors of a row reordered bitmap using a longer segment length, while using a shorter length on later bit vectors. While a more robust compression can be expected using VLC, a challenge is maintaining efficient query processing speeds.

3.1 Variable Compression Scheme

For each bit vector in the bitmap VLC compression performs the following steps: (1) Determining segment length for bit-vector compression, (2) Vector segmentation, and (3) Word packing.

The goal of the *Segment Length Determination (SLD)* algorithm is to determine an optimal segment length for a given bit-vector. In general, given a bit vector, $v = (b_1, \ldots, b_n)$, SLD returns an integer value $seg\_len \mid L < seg\_len < H$. In this paper, we assume H to be the word-size, $H = 32$, and $L = 2$, since 2-bit segments cannot represent fills. In this work we consider two SLD approaches: `All Possible` is a brute force algorithm which simulates the compression of v using all possible segment lengths, $seg\_len = 3, \ldots, 31$. For each segment length, `All Possible` computes the number of words needed to store the compressed bit vector, and it returns the length that would achieve the best compression. If multiple segment lengths generate the same compression ratio, then the largest segment length is returned to reduce the amount of parsing required when the bit vector is queried—this follows the same insight behind WAH versus BBC.

Another SLD heuristic we consider is `Common Factor`, which is similar to `All Possible` in that it simulates the compression of individual bit vectors. However, `Common Factor` also takes as input an integer parameter, *base*, which is used to determine the set of segment lengths that will be used in the simulation. Specifically, it will only consider seg_len where $seg\_len \in \{x | 3 \leq x \leq 31 \wedge x \equiv 0 \ (mod\ base)\}$. The basis for this heuristic is to ensure that $gcd(v_1, v_2) \geq base$ any two compressed bit vectors v_1 and v_2. As it will become clear later, this property greatly improves query processing speed.

After seg_len is determined, the *Bit Vector Segmentation* process is applied. A compressed bit segment v_c is defined as a sequence of seg_len bits,

$$v_c = \begin{cases} 0 \bullet x_1 \bullet \ldots \bullet x_{seg\_len} & \text{(literal)} \\ 1 \bullet x \bullet n_1 \ldots \bullet n_{seg\_len-1} & \text{(fill)} \end{cases}$$

where $\bullet$ denotes concatenation. In the former case, the initial bit 0 denotes an uncompressed segment from v, and the succeeding sequence $x_1, \ldots, x_{seg\_len}$ denotes the literal. In the second case, v_c can represent a run by specifying 1 as the flag bit. The next bit x is the fill bit, and $n_1 \ldots, n_{seg\_len-1}$ is a number (base 2) denoting the multiple of a run of seg_len consecutive x bits. For example, if $seg\_len = 4$, the segment 10011 is the compressed representation of 000000000000, i.e., a run of $3 \times 4 (= 12)$ 0's.

Given this code representation, the algorithm proceeds as follows. Beginning with the first bit in an uncompressed vector v, we let v' denote the next seg_len bits in v. We initially encode v' as a literal, that is, $v_c = 0 \bullet v'$. Next, v' is assigned the subsequent seg_len bit sequence in v. If v' is a sequence of identical bits x, then we verify if it can be coalesced with v_c. If v' is not a run of seg_len bits, the v_c is first written to disk, and then again assigned the literal $0 \bullet v'$. If v_c indeed is a single literal containing a (seg_len)-bit run of x, then v_c is converted to a fill segment, $1 \bullet x \bullet 0 \ldots 010$. A more general case occurs if v_c is already a fill-segment. When this occurs, we simply add 1 to the run length portion. As v' continues to be assigned subsequent segments, the above steps are repeated.

Finally, *Word Packing* is used to reduce the parsing cost when executing a query over VLC segments. Segments (both runs and literals) are *fit* into words. For example, if $seg\_len = 3$, then VLC packs 8 segments (including their flag bits) in one 32 bit word. If $32 \not\equiv 0 \ (mod\ seg\_len + 1)$ then VLC appends $32\ mod\ seg\_len + 1$ 0's to the end of each word. These superfluous bits are called *pad bits* and they are ignored by the query algorithm. Our approach requires that each compressed bit vector be prefaced a header byte, which stores the segment encoding length used.

In Figure 2, we show two 32-bit words, in vectors X and Y, sharing a common gcd of 7. X is coded in $seg\_len = 14$, and Y in $seg\_len = 7$. Segment a is a fill, denoting a run of $seg\_len \times 87 (= 1218)$ 1's. In other words, a is 87 consecutive 14-bit segments of 1's. Because each 14-bit segment is coded using 15 bits, two such segments can be packed into a word, with the last remaining 2 bits b being padding. In the bottom word Y, segment c is a 7-bit literal 1111110, and d is a run of $seg\_len \times 48 (= 336)$ 0's. e simply denotes the remaining words in either vector.

3.2 Query Processing

Algorithms 1 and 2 are the query processing procedures over two bit vectors, X and Y from Figure 2. As a running example, we consider performing query using a logical op between X and Y.

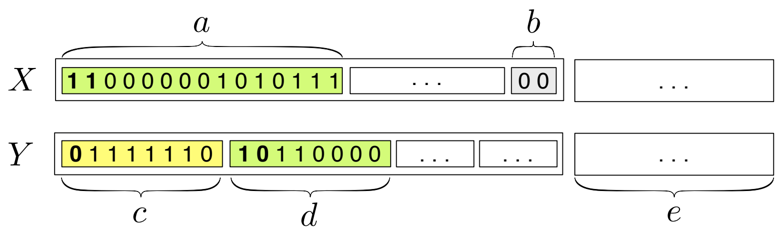


Fig. 2. Example of VLC with $seg\_len = 14$ and $seg\_len = 7$

Initially, we declare an empty vector Z to hold the results, and assign its base to $gcd(X, Y) = 7$ (Alg1:lines 1-3). Next, we loop through all segments of X and Y (Alg1:line 4), and in our example, X.active is assigned to a and Y.active is assigned c (Alg1:line 6). Next, both segments are decoded (see Algorithm 2), and this subprocedure invokes one of two cases. If the given segment seg is a fill, then its base *reduced* down to the gcd, effectively producing $(base/gcd)\times$ its current segment run-length (Alg2:lines 4-6). In the example, recall that a represents 87 consecutive 14-bit segments of 1's, so a.runLen = 87. However, since the gcd is 7, we must reduce a's base down to 7, which effectively produces $87 \times (14/7) = 174$ 7-bit segments in a. On the other hand, if a were a literal, then it is split into $(base/gcd)$ separate gcd-bit literals (Alg2:lines 3-9).

Algorithm 1 ColumnQuery(X, Y, op)

```
1: Let $Z$ be the result vector (compressed form)
2: $g \leftarrow gcd(X.seg\_len, Y.seg\_len)$
3: $Z.seg\_len \leftarrow g$
4: **while** $X$.hasNextSegment() **or** $Y$.hasNextSegment() **do**
5:     {*modify the segments to match the GCD*}
6:     Let $X$.active and $Y$.active be the next segment read from $X$ and $Y$
7:     decode($X$.active, $X$.seg_len, $g$); decode($Y$.active, $Y$.seg_len, $g$);
8:     **if** isFill($X$.active) **and** isFill($Y$.active) **then**
9:         $n \leftarrow$ min($X$.active.runLen, $Y$.active.runLen);
10:        $Z$.appendFill($n$, ($X$.active.fillBit $op$ $Y$.active.fillBit));
11:        $X$.active.runLen $\leftarrow$ $X$.active.runLen$-n$;
12:        $Y$.active.runLen $\leftarrow$ $Y$.active.runLen$-n$
13:    **else if** isFill($X$.active) **and** isLit($Y$.active) **then**
14:        $Z$.appendLit($Y$.getNextLiteral() $op$ $X$.active.fillValue);
15:        $Y$.litCount $\leftarrow$ $Y$.litCount $-1$; $X$.active.runLen $\leftarrow$ $X$.active.runLen $-1$;
16:    **else if** isLit($X$.active) **and** isLit($Y$.active) **then**
17:        $Z$.appendLit(($X$.active.getLiteral() $op$ $Y$.active.getLiteral()))
18:        $X$.litCount $\leftarrow$ $X$.litCount $-1$; $Y$.litCount $\leftarrow$ $Y$.litCount $-1$;
19:    **end if**
20: **end while**
21: return $Z$
```

Algorithm 2 decode(seg, $base$, gcd)

```
1: if isFill($seg$) then
2:     $seg$.runLen ← $seg$.runLen $\times(base/gcd)$
3: else
4:     $temp$ ← $seg$.getLiteral(); {* Binary representation of literal *}
5:     $seg.litCount \leftarrow base/gcd$;
6:     for each partition $p$ of $gcd$ consecutive bits in $temp$ do
7:         $seg$.addLiteral($p$);
8:     end for
9: end if
```

Returning to Algorithm 1, the above decoding procedure prepares the two vectors to share the same base, which allows for easy application of the bitwise op. Then there are three cases: (1) both X and Y are fill words (Alg1:lines 8-11), (2) only X is a fill and Y is a literal (Alg1:lines 12-14), and (3) both X and Y are literals (Alg1:lines 15-18).[⋆] In our example from Figure 2, Case 2 is invoked on segments a and c. We apply the bitwise op across X.fillValue and Y.nextLiteral(), which results in appending (1111111) op (1111110) to Z. Because we have only processed one literal, X's run-length counter is decremented by 1 (Alg1:line 15). However, while Y is subsequently assigned the next parsed segment d, X avoids this overhead, which can be significant if d happened to exit in the next word, causing Y to read from memory.

If Case 1 is invoked, that is, when both X and Y are fills, we can simply apply op to the single fill bit, and implicitly know that its result applies to all bits until the end of either X's or Y's current segment.s Thus, without corresponding memory accesses, we can process $min(X$.active.runLen, Y.active.runLen$)$ $\times$ gcd bits in $O(1)$ operation of updating the fill counts (Alg1:lines 8-12). Because of this property, the query processing time for extremely long runs can be done in sublinear time.

4 Experimental Evaluation

In this section, we evaluate the compression ratios among BBC, WAH, and our VLC variants over both real and synthetic data sets. We further analyze query processing performance. All experiments in this section were executed on a Java 1.6 implementation with `-Xmx1024m` set running on a 32-bit Windows 7 machine with 2.0 GB RAM and an Intel Dual Core 2.4GHz processor. The bitmap data sets on which we experiment in this study are described as follows.

– `HEP` (272MB) is from a real high-energy physics application containing 12 attributes. Each attribute was split into ranges from 2 to 12 bins, which results in a total of 122 columns comprised of 2,173,762 rows.
– `Histo` (21MB) is from an image database. The tuples represent images, and 192 columns have been extracted as color histograms of these images. The bitmap contains 112,361 rows and 192 columns.
– `Landsat` (238MB) is derived from real satellite images, whose SVD transformation produces 275,465 rows and 522 columns.

[⋆] Note that a 4th case exists also, which is the reverse of case (2), but it is redundant here.

– `Stock` (6.7MB) contains approximately 1080 days' worth of stock data for 6,500 companies, resulting in a high-dimensional bitmap containing 6,500 rows and 1080 columns.
– `Uni` (10.3MB) is a synthetic dataset generated with random bit distribution over 100,000 rows and 100 columns.

As an optimizing preprocessing step, all above data sets have initially been Gray code ordered using the algorithm presented in [16].

4.1 Data Compression Analysis

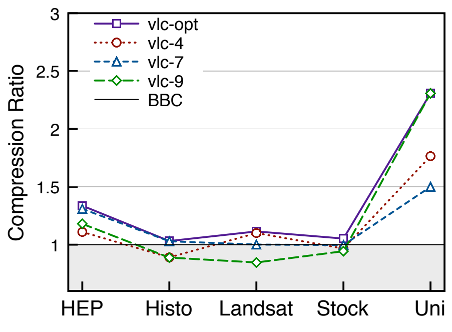
We implemented WAH, BBC, and VLC and compressed all aforementioned data sets. With VLC, we varied across all segment lengths $3 \leq seg\_len \leq 31$. However, due to space constraints, we only report four representative configurations: `vlc-opt`, `vlc-4`, `vlc-7`, and `vlc-9`. The `vlc-opt` setting corresponds to using the `All Possible` Segment Length Determination (SLD) algorithm to find the optimal segmentation length for each bit vector. The compression generated by `vlc-opt` represents the best possible compression our implementation of VLC can achieve. The `vlc-4`, `vlc-7`, and `vlc-9` results are produced by using the `Common Factor` SLD algorithm on a *base* of 4, 7, and 9 respectively. For instance, bitmaps compressed in `vlc-4` may contain bit vectors encoded in segment lengths of 4, 8, 12, 16, 20, 24, and 28.

	Compression (MB)						
Dataset	`Orig`	`WAH`	`BBC`	`vlc-opt`	`vlc-4`	`vlc-7`	`vlc-9`
`HEP`	272.0	2.251	1.552	1.161	1.398	1.185	1.315
`Histo`	21.4	1.05	0.59	0.572	0.663	0.573	0.664
`Landsat`	238.0	28.001	18.699	16.779	16.99	18.683	22.098
`Stock`	6.7	0.62	0.637	0.605	0.659	0.639	0.675
`Uni`	10.2	0.033	0.03	0.013	0.017	0.02	0.013

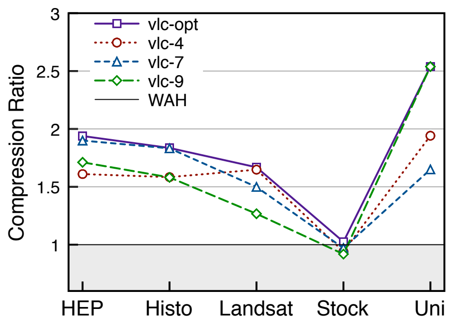
Table 2. Data Compression Size

Table 2 presents the size (in MB) of the compressed data sets. As expected, we observe that `vlc-opt` outperforms all configurations, but because the bit vectors are not *gcd*-aware, the misalignment of the segments will adversely affect query performance. Thus, we emphasize the "closeness" of the `vlc-⋆` versions to `vlc-opt` and that most `vlc-⋆` configurations also out-compress both WAH and BBC. For further comparison, we juxtapose the compression ratio of VLC over BBC and WAH in Figures 3(a) and 3(b) respectively. Expectedly, most `vlc-⋆` versions provide a modest improvement when compared to BBC for most data sets. We can observe that for `Histo`, `Landsat`, and `Stock`, BBC slightly out-compresses `vlc-9`. We believe this is due to BBC's aggressive compression of shorter runs, which appears frequently in these data sets.

Figure 3(b) depicts the comparison of VLC to WAH. Because of WAH's longer fixed segments, it is expected that all `vlc` versions should out-compress WAH, unless in the presence of massive amounts of *exponentially* long runs, which are rare. In the best case, we can observe improvement of $1.7\times$ (for real data) and $2.54\times$ (for synthetic data) the compression rates of WAH using VLC. We can observe the best results for the synthetic `Uni` dataset, where it becomes very clear that the optimal compression



(a) Versus BBC



(b) Versus WAH

Fig. 3. Comparison of Compression Ratio

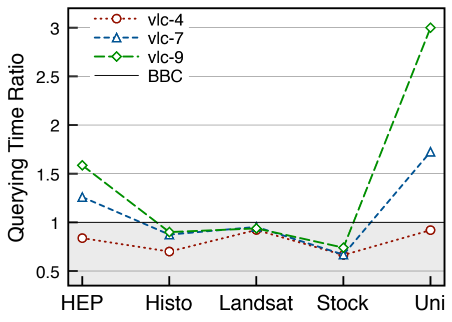
segment length is in between the extremes of BBC and WAH. Out of 100 bit vectors, the `vlc-opt` algorithm compressed 50 of these in base-9, 3 in base-13, and 47 in base-14. In `vlc-9`, which is near optimal, is compressing 50 bit vectors in base-9, 48 columns in base-18, and 2 columns in base-27.

An anomalous data set is `Stock`, where compression gains appear difficult to achieve. We believe there are two reasons contributing to this observation. Due to the `Stock` data's dimensionality (1080 columns), after the first several columns, Gray code tuple ordering eventually deteriorates and begins generating columns with very short runs, or worst case, uniformly distributed bits. Adding to this effect is that `Stock`'s number of rows is small, which implies the opportunities for longer runs is made further infrequent. This means most of the later columns are probably being represented as literals in all compression schemes. This theory is supported by the fact that WAH actually out-compresses most schemes (an outlier). Due to the fact that WAH only uses one flag bit per 31-bit literal, it is by far the most efficient way to store long literals in all schemes.

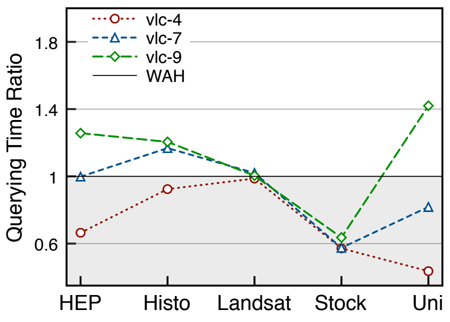
4.2 Evaluation of Query Processing Times

In this subsection, we present the query processing times. For each data set, a set of 10 queries, which vary in the amount of tuples retrieved, was generated. To execute these queries, we split each bitmap vertically into 4 equal-sized sets of bit vectors B_1, B_2, B_3, B_4. Each query inputs 2 bit vectors X and Y such that $\forall_{i,j}$: select $X \in B_i$, $Y \in B_j$ randomly. In other words, X and Y are randomly selected from each set of bit vectors, and we query against all combinations of bit vectors. We submitted the set of 10 queries repeatedly for all B_i, B_j combinations, and averaged the execution time to process all 10 queries.

This experimental protocol was selected in an attempt to avoid any segmentation length bias. By randomly selecting compressed bit vectors from different quadrants of the bitmap, we increase the likelihood that, under `vlc-`$\star$, the queried bit vectors would be in segmentation lengths. This protocol also ensures that not all the queried columns were selected from the first (or last) few bit vectors which would favor WAH, or conversely, BCC in inner regions.



(a) Versus BBC



(b) Versus WAH

Fig. 4. Comparison of Query Processing

Figures 4(a) and 4(b) display the query time ratios between `vlc-*` versions and BBC and WAH respectively. We did not include the results from `vlc-opt` in the graphs as it was, on average, 10 times slower than the next slowest algorithm. This was expected, since many of the column pairs used in the queries had $gcd = 1$, meaning they had to be fully decompressed before logical operation could be applied.

As can be seen in both Figure 4(a) and 4(b), `vlc-4` provides a less than optimal querying efficiency. Recall that this configuration can produce segment lengths of 4, 8, 12, 16, 20, 24, and 28. While this flexibility sometimes allows for robust compression rates, in terms of query performance, if X and Y vectors vary in segment lengths (which occurs frequently due to the large number of available base-4 options), they must be decoded to base-4 gcd, which is costly. This observation is supported by the performance gains from using the longer segment lengths of `vlc-7` and `vlc-9`. Because $seg\_len = 7$ and $seg\_len = 9$ only generate $\{7, 14, 28\}$ and $\{9, 18, 27\}$ bases, there will be higher probabilities where any two random bit vectors X and Y will match base, and thus *not* requiring gcd-base decoding.

Furthermore, for vectors that still require base decoding, the inherently larger bases of 7 and 9 allow bitwise operations to be amortized over a smaller base 4. These features combined with the increase in compression allowed `vlc-7` to match or improve upon WAH's querying times for 3 of the 5 data sets and `vlc-9` out-performs WAH on 4 of the data sets (Figure 4(b)). When compared to BBC, `vlc-7` and `vlc-9` had comparable query times for `Histo` and `Landsat`, and they performed significantly better on `Hep` and `Uni` (Figure 4(a)). For these latter data sets, `vlc-7` and `vlc-9` resulted in better compression than BBC and used larger segmentation lengths for most columns. The result is that much less parsing was required during query processing.

It is interesting to note that BBC querying actually out-performed WAH for `Histo` and `Landsat`. A closer investigation revealed that, for each of these data sets, a small number of the 10 queries were skewing the results. In both cases, the bit vectors in these outlier queries came from quadrants B_3 and B_4 of the bitmap. We surmise that these are the columns in a Gray code ordered bitmap that led to inefficient WAH compression. In these instances, it appears that WAH had to use a large number of literals to represent bit vectors. Essentially, this meant that WAH had to perform a logical operation for each bit. Conversely, due to BBC's ability to compress short runs it was able to perform fewer operations.

Summary and Discussion: In summary of our analysis, the results of our empirical study indicate that VLC can, on average, offer higher compression rates than either BBC and WAH. Although not initially expected, we also presented cases where VLC outperforms both BBC and WAH. Ignoring *vlc-opt* due to its prohibitive query processing times, in the *best case* for real data sets, `vlc-9` out-compresses BBC and WAH by a factor of $1.3\times$ and $1.71\times$ respectively. These numbers jump to $2.3\times$ and $2.54\times$ respectively for BBC and WAH for synthetic uniform data. In all of our compression experiments, the *worst case* was a 15% loss in compression, albeit this was rare. In terms of query performance, we managed a speedup factor of $1.6\times$ over BBC and $1.25\times$ over WAH in the *best case* for real data sets. These numbers grow to a $3\times$ speedup for BBC and $1.42\times$ speedup for WAH for our synthetic data. In the *worst case*, around a $0.6\times$ slowdown for `Stock` can be seen compared to either BBC or WAH. This suggests that VLC's gcd decoding renders it inefficient for high dimensional data sets with small numbers of rows, which are rare for large-scale data-intensive applications.

Our results also echo such efforts as [12], which sought a deeper understanding of how row ordering (as well as potentially many other optimizations) truly affects bitmap performance and compression. We view our findings as yet another example of how careful consideration of the segmentation length used for compression is especially important for Gray code ordered bitmaps.

5 Related Work

There is a large body of research related to bitmap indices and compression. In this section we focus on the class of techniques most relevant to our work.

Bitmap indices [18, 15], which are closely related to inverted files [14] (frequently occurring in information retrieval) have long been employed in traditional OLAP and IR systems. Over the years, seminal efforts have made practical the integration of bitmaps for general data management. For instance, works have addressed bitmap encoding issues, which include considerations on bit-vector cardinality and representation to optimize query processing [5, 17]. The focus in efforts are tangential to the work presented in this paper. We offer a generalized word aligned compression scheme.

Run-length encoding (RLE) techniques [9] were popularized early through the ubiquitous bit representations of data, and were particularly useful for compressing sparse bitmaps. For example, a bit vector can be transformed to a vector containing only the positions of 1's, and thereby implicitly compressing the 0 bits. When the sequence is further transformed to the differences of the positions, coding schemes, such as Elias's δ and γ codes and its variations [8, 13, 4], can be used to map the expectedly small integers to correspondingly small bit representations.

These efforts, however, do not consider the impact of memory alignment, which significantly slows the performance of bitwise logical operations, e.g., a vector which spans two bytes would require two separate reads. Such decoding overheads would not be very suitable for database query processing. Thus, effort towards generating memory-aligned, CPU friendly compressed bitmaps ensued. The Byte-aligned Bitmap

Code (BBC) [2] exploits byte alignment and allows direct bit-wise comparisons of vectors in their compressed state. Wu, *et al.* proposed Word-Aligned Hybrid Code (WAH) [20, 19], which is even more amenable for processing.

Due to WAH's success in accelerating query processing times, many variations of WAH have also been proposed. One example is the Word-aligned Bitmap Code (WBC), also seen in literature as Enhanced WAH (EWAH) [21]. This scheme uses a *header* word to represent fill runs and literal runs. A 31-bit header word (plus the 1 MSB for flagging) is split into two halves. The upper 16 bits following the flag bit is used to denote the fill, and the run length, just as before. The lower 15 bits are used to denote the run length of literals following the fill run. This optimization can be significant for query processing. For instance, long literal runs can be ignored (without accessing them) when AND'd with 0's. If a logical comparison is indeed required, the decoding phase would know *a priori* that the next n words are literals without parsing the flag bit.

Deliege and Pederson proposed a Position List extension to WAH (PLWAH), which exploits highly similar words [6]. Their insight is that, often, only a single bit can compromise a much longer run, and typically, it is highly unlikely that all fill bits are used in a word. Thus, their scheme first separates a bitmap into segments of $wordsize - 1$ bits. Words that are "nearly identical" to a fill word are identified and appended onto a fill, rather than representing it as a literal. The five most significant bits following the fill word's *fill* and *flag* bits are further used to identify the position of the bits in the nearly identical literal. We surmise that PLWAH can be used in conjunction to our VLC scheme, which is currently being implemented.

There has also been significant amounts of work on the issue of *bitmap row reordering*. Because tuple order is arbitrary in a database relation, its corresponding bitmap can thus be reordered to maximize runs. Pinar, Tao, and Ferhatosmanoglu explored the tuple reordering problem in [16]. They proved that tuple reordering is NP-Complete, and proposed the Gray code ordering heuristic. More recently, Lemire *et al.* presented an *extensive* investigation into reordering efforts (including column reordering, which is not being considered in our work) over large bitmaps [12]. Among various contributions, they made several interesting observations. For one, the authors experimented with 64-bit words and observed an interesting space-time tradeoff: while 64-bit word compression expectedly generates indices that are twice as large, the queries are slightly faster.

The above efforts are orthogonal to VLC — our technique allows bit vectors to be compressed and queried using varying segment encoding lengths. We have shown that VLC achieves greater compression in our experiments than both WAH and BBC in most cases, when the correct segment length is chosen. Our scheme thus provides an option to the user to encode a bitmap using specific encoding lengths to greater optimize compression, or to use encoding lengths that would allow for faster querying on certain columns that may be queried often. Thus, VLC is a *tunable* approach, which allows users to trade-off size and performance.

6 Conclusion and Future Work

Bitmap indices have become a mainstay in many database systems, popularized due to its fast query processing and compressibility. However, as high dimensional data continues to grow at today's astounding pace, bitmap compression becomes increasingly more important to minimize disk accesses when possible.

In this paper, we proposed a novel bitmap compression technique, *Variable Length Compression (VLC)*, which allows for robust run-length compression of ordered bitmaps. We offer two simple heuristics on selecting encoding length per given bit vector. Our query processing algorithm automatically decodes bit vectors to the same coding base so that queries can be carried out efficiently. We ran an experimental study on 4 sets of real data and 1 synthetic data set. We showed that VLC can out compress both BBC and WAH, two of today's state-of-the-art bitmap compression schemes, by around $2.5\times$ in the best case. We concede cases where VLC compression rates are less than BBC, but only marginally. In terms of query performance, the expectation was that VLC would lie somewhere between BBC and WAH, but remain competitive to WAH. While this assumption was shown to be true, we also observed interesting results that show clearly, there are cases where VLC outperforms and out compresses both BBC and WAH. Again, we acquiesce that there are opportunities for improvement on certain data sets.

During the evaluation process, some future work opportunities emerged. For example, we can (and should) *adapt* the segment encoding lengths to query history. Because longer segments tend to query much faster, we can dynamically relax compression rates for frequently queried columns (and conversely, compress infrequently queried columns more aggressively). To ensure word alignment, we currently *pad in* the last unused bits of a word, a necessary storage cost. An alternative could be to fit as many representations fit into one word, then begin a representation in one word and finish it in the following word, "stitching" the representation together. While we expect that the stitching may result in a slowdown in query times, but may also provide a substantial gain in compression. Some obvious experimental extensions would include using larger data sets, different row ordering algorithms combined with column re-ordering which has been shown to increase run-lengths [1, 11], and range queries.

Acknowledgments

We would like to thank Hakan Ferhatosmanoglu (Ohio State University), Guadalupe Canahuate (The University of Iowa), and Michael Gibas (Teradata) for their valuable insights and comments on this paper.

This work was generously supported in part by an Amazon Web Services (AWS in Education) Research Award to D. Chiu at Washington State University and a Martin Nelson Summer Award for Research to J. Sawin at the University of Puget Sound.

References

1. D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *ACM SIGMOD International Conference on Management of Data*, pages 671–682, 2006.
2. G. Antoshenkov. Byte-aligned bitmap compression. In *DCC '95: Proceedings of the Conference on Data Compression*, page 476, Washington, DC, USA, 1995. IEEE Computer Society.
3. T. Apaydin, A. c. Tosun, and H. Ferhatosmanoglu. Analysis of basic data reordering techniques. In *International Conference on Scientific and Statistical Database Management*, pages 517–524, 2008.
4. N. R. Brisaboa, S. Ladra, and G. Navarro. Directly addressable variable-length codes. In *String Processing and Information Retrieval - SPIRE*, pages 122–130, 2009.

5. C.-Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, SIGMOD '99, pages 215–226, New York, NY, USA, 1999. ACM.
6. 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)*, pages 228–239, 2010.
7. F. Donno and M. Litmaath. Data management in wlcg and egee. worldwide lhc computing grid. Technical Report CERN-IT-Note-2008-002, CERN, Geneva, Feb 2008.
8. P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, 1975.
9. S. W. Golomb. Run-Length Encodings. *IEEE Transactions on Information Theory*, 12(3):399–401, 1966.
10. 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*, pages 1–8, 2008.
11. D. Lemire and O. Kaser. Reordering columns for smaller indexes. *Information Sciences*, 181, 2011.
12. D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *Data and Knowledge Engineering*, 69:3–28, 2010.
13. A. Moffat and J. Zobel. Parameterised compression for sparse bitmaps. In *SIGIR*, pages 274–285, 1992.
14. A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14:349–379, 1996.
15. P. E. O'Neil. Model 204 architecture and performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 40–59, London, UK, 1989. Springer-Verlag.
16. 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)*, pages 310–321, 2005.
17. R. R. Sinha and M. Winslett. Multi-resolution bitmap indexes for scientific data. *ACM Trans. Database Syst.*, 32, August 2007.
18. H. K. T. Wong, H. fen Liu, F. Olken, D. Rotem, and L. Wong. Bit transposed files. In *Proceedings of VLDB 85*, pages 448–457, 1985.
19. K. Wu, E. Otoo, and A. Shoshani. An efficient compression scheme for bitmap indices. In *ACM Transactions on Database Systems*, 2004.
20. 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)*, pages 99–108, 2002.
21. K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg. Notes on design and implementation of compressed bit vectors. Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory, 2001.
22. M. J. Zaki and J. T. L. Wang. Special issue on bionformatics and biological data management. In *Information Systems*, pages "28:241–367", 2003.