ABSTRACT
Indexing is a fundamental mechanism for efficient data access. Recently, we proposed the Variable-Aligned Length (VAL) bitmap index encoding framework, which generalizes the commonly used word-aligned compression techniques. VAL presented a variable-aligned compression framework, which allows columns of a bitmap to be compressed using different encoding lengths. This flexibility creates a tunable compression that balances the trade-off between space and query processing time. The variable format of VAL presents several unique opportunities for query optimization.

In this paper we explore multiple algorithms to optimize both point queries and range queries in VAL. In particular, we propose a dynamic encoding-length translation heuristic to process point point queries. For range queries, we propose several column orderings based on the bitmap’s metadata: largest segment length first (lsf), column size (size), and weighted size (ws). In our empirical study over both real and synthetic data sets, we show that our dynamic translation selection scheme produces query execution times only 3.5% below the optimal. We also found that the weighted size column ordering significantly and consistently out-performs other ordering techniques. Finally, we show that algorithms scale to data sets that are row-ordered.

Categories and Subject Descriptors
H.2.4 [Systems]: Query Processing

Keywords
bitmap indices, bitmap compression, query execution

1. INTRODUCTION
Popular applications including, but not limited to, web search, online shopping, and social media, are increasingly data-intensive, and billions of users rely on fast interactions with massive data stores. To offer high-performing data access and query processing
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
IDEAS ’14 July 07 - 09 2014, Porto, Portugal
Copyright 2014 ACM 978-1-4503-2627-8/14/07...$15.00
http://dx.doi.org/10.1145/2628194.2628252.
length of consecutive segments. Similarly, 64-bit literal atoms are \((flag, v)\), where \(flag = 0\) denotes a literal and \(v\) is the \(s\)-bit verbatim segment. Intuitively, because word-aligned schemes impose a fixed segment length of \(s = w - 1\), it can efficiently compress extremely long runs of segments. It can also be seen that shorter runs can be represented more efficiently given smaller segment lengths.

To address segment size variability, we recently proposed the Variable Aligned Length (VAL) encoding scheme [9]. Unlike WAH, VAL enables varying segment lengths, so as long as they are multiply-aligned. For example, VAL might compress a noisier column by separating it into segments of length \((s = 15)\) to efficiently represent shorter runs. A sparse column could be compressed using segment length \(s = 60\) to emulate word-aligned encoding. We developed systematic methods to process queries between columns encoded using different segment lengths.

A particular challenge intrinsic to the VAL’s query execution algorithm is to reconcile different segment lengths when applying a logical operation between two columns \(V_i\) and \(V_j\). For instance, if we assume \(V_i.seglen = 15\) and \(V_j.seglen = 60\), it is unclear whether it is more efficient to decode \(V_i\) up to a single 60-bit segment, or conversely, decode \(V_j\) down to four 15-bit segments, before applying the logical operation.

This paper explores the unique challenges and opportunities that VAL’s variable alignment represents for efficient query execution of point and range queries. We explore several novel range query algorithms for VAL. This study makes the following contributions:

- We evaluate and identify the tradeoff between two distinct segment length translation algorithms for query execution: decode up vs. decode down. We present a metadata-driven heuristic that can select the ideal translation algorithm with a high rate of accuracy for point queries.
- For range query execution, we gather the columns’ metadata (e.g., column size, segment length, etc.), which is then used to impose a total ordering in range query processing. We show that ordering impacts the efficiency of range query execution in VAL for several different types of data.
- We present a nuanced experimental study that uses both real and synthetic data sets to evaluate the effects of our ordering on five unique range query processing algorithms. The results of this study show that a column ordering based on size weighted by column composition is the most efficient. It also suggests that an accumulation approach to query processing is ideal for uniform data and a reentrant priority queue approach is better suited for skewed data.

The remainder of the paper is organized as follows. Section 2 provides an overview of the VAL bitmap compression scheme. Section 3 presents our proposed optimizations to VAL’s range query algorithm. Section 4 presents the results of our empirical study. Related works are presented in Section 5. We conclude and present future avenues for this work in Section 6.

### 2. VAL COMPRESSION

The popular Word-Aligned Hybrid code (WAH) excels in compressing and accelerating the query processing of bitmaps containing super sparse columns. In general, depending on the application and data distribution, a bitmap can take on a spectrum of sparsity, which may not always favor WAH. In contrast, VAL analyzes each bit vector to be compressed, noting the bit distribution and run-lengths, then chooses an appropriate segment length and encoding algorithm. Like WAH, the bit vector is compressed by VAL into atoms, which are limited by hardware constraints, but the significant difference is that VAL can specify a segment length into which the word is broken and encoded. These words are then compressed using the specified compression method, determined by VAL during the analysis of the bit vector.

![Figure 1: VAL Encoding Example](image)

Consider the example shown in Figure 1, where the segment length is set to 15 with a word size of 64. The uncompressed bit vector (top) is 1155 bits long. Each word in the VAL compressed bit vector (bottom) will have a header of 4 bits representing whether the respective segment is a fill or a literal, reducing some of the overhead from decoding during query execution. The first and last segments are literals and represent the 15 verbatim bits from the original bit vector. The middle two segments are fills and are encoded such that the first bit is their fill bit (underlined in the figure) and the remaining bits encode the run-length of segments contained. Here, VAL was able to compress 1155 bits down to only a single 64-bit word. In contrast, 64-bit WAH would have require five words, or 320 bits. It can be observed even in this simple and contrived example that VAL’s flexibility from being able to select different segment lengths for each bit vector can improve the compression of shorter runs.

To minimize the translation overhead, VAL only allows for segments that are multiply-aligned. For example, with alignment factor 15 and word size 64, the possible segments lengths are 15, 30, and 60. Consider two VAL bit vectors, \(A_{m \times s}\) and \(B_s\), encoded using segment lengths \(m \times s\) and \(s\), respectively. The operation \(A_{m \times s} \circ B_s\), where \(\circ\) is a binary logical operator. Algorithm 1 shows the pseudocode for query processing. For each bit vector, one physical word (currentWord) is decoded at a time (Line 1-7). The parameter \(m\) (Line 3) indicates that the currentWord should be decoded into blocks, representing atoms, of segment length \(m \times s\). Given the multiply-aligned restriction on segment lengths, \(m\) is always a power a 2. The decoded currentWord contains a number of activeBlocks. This activeBlock is tantamount to the activeWord structure used in WAH. The currentWords are iterated over one block at a time (Lines 8-14) and are operated together until exhausted. Two fill blocks can be operated together 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, \( n\text{Segments} \), is decremented by 1 with each \( \text{getLitValue()} \) call (Lines 21–23).

**Algorithm 1: General Bitwise Logical Operation**

**Input:** Bit Vector \( A, B \) (\( A.\text{segLen} = m \times s \) and \( B.\text{segLen} = s \))

**Output:** Bit Vector \( Z \) The resulting compressed bit vector after performing the logical operation \( X \odot Y \)

1. while \( A \) and \( B \) are not exhausted do
2.   if \( A.\text{currentWord} \) is exhausted and \( \exists \) words in \( A \) then
3.     \( A.\text{decodeNextWord}(m) \);
4.   end
5.   if \( B.\text{currentWord} \) is exhausted and \( \exists \) words in \( B \) then
6.     \( B.\text{decodeNextWord}(1) \);
7.   end
8.   while \( A.\text{currentWord} \) and \( B.\text{currentWord} \) are not exhausted do
9.     if \( A.\text{activeBlk}.n\text{Segments} \) is 0 then
10.    \( A.\text{activeBlk} = A.\text{nextBlk} \);
11. end
12. if \( B.\text{activeBlk}.n\text{Segments} \) is 0 then
13.    \( B.\text{activeBlk} = B.\text{nextBlk} \);
14. end
15. if \( A.\text{activeBlk}.\text{isFill}() \) and \( B.\text{activeBlk}.\text{isFill}() \) then
16.   \( n\text{Segments} = \min(A.\text{activeBlk}.n\text{Segments}, \) \( B.\text{activeBlk}.\text{activeBlk}.\text{Segments}) \);
17.  \( Z.\text{addFill}(A.\text{activeBlk}.\text{fill}, n\text{Segments}) \);
18.  \( A.\text{activeBlk}.\text{activeBlk}.\text{Segments} -= n\text{Segments}; \)
19.  \( B.\text{activeBlk}.\text{activeBlk}.\text{Segments} -= n\text{Segments}; \)
20. end
21. else
22.   \( Z.\text{addLiteral}(A.\text{activeBlk}.\text{getLitValue}()) \);
23.   \( B.\text{activeBlk}.\text{getLitValue}()(); \)
24. end
25. end
26. end
27. return \( Z \);

Let us focus on Lines 3 and Line 5 of the algorithm. Due to segment length variation across bit vectors, it is necessary to align the bits appropriately before performing logical operations on them. Given \( A_{m \times s} \) and \( B_{s} \), there are two scenarios: the segment lengths between them are the same \( (m = 1) \), or they can differ \( (m > 1) \). In the former case, the bit vectors are already aligned, and the logical operation can proceed normally. In the latter case where the segment lengths differ \( (e.g., A_{30} \text{ and } B_{15}) \), the segments must be reconciled so that they are in alignment. Here, we have a choice: (1) the words in \( A_{30} \) could be split into segments of length 15. We name this method DecodeDown. (2) Conversely, the words in \( B_{15} \) can be merged to form segments of length 30. We refer to this method as DecodeUp. Note that the decision of the decoding direction occurs only once, prior to running Algorithm 1 which also defines the segment length of the result bit vector \( Z \).

We now discuss DecodeDown (Algorithm 2) and DecodeUp (Algorithm 3). When decoding down there are two cases depending on whether the block being decoded is a fill or literal. If the block is a fill, then the new run length is equal to:

\[
\text{runLength}_{\text{new}} = \frac{\text{segLen}_{\text{old}}}{\text{segLen}_{\text{new}}} \times \text{runLength}_{\text{old}}
\]

Recall from before that, because \( m \) is a power of 2, the DecodeDown operation can be reduced to shifting bits to the right. Figure 2(a) shows an example where the bits representing the number of segments represented can not translate into a single segment when decoding down from 60 to 15. Thus, two segments are created such that together they represent the same number of zeros in the bit vector. A literal segment is decoded down by dividing it into the smaller segment length and appending it onto the word. In this case a literal with segment length 60 would be broken down with segment length 15 into four literals.

**Algorithm 2: DecodeDown()**

**Input:** Compressed word containing blocks of length \( s \); \( N \): the number of blocks in the word; \( m \): the conversion factor to the new segment length

**Output:** \( \text{activeWord} - \text{VAL}\text{ word containing decoded blocks using segments of length } s/m \)

1. for \( i = 1 \rightarrow N \) do
2.   if \( \text{activeBlk}.\text{isLiteral}() \) then
3.     \( \text{activeWord}.\text{addLiteral}(\text{activeBlk}.\text{val} >> (s - (N - i))); \)
4.   end
5.   else
6.     \( \text{activeWord}.\text{addFill}(\text{activeBlk}.\text{val} >> (s - 1), \) \( \text{activeBlk}.n\text{Segments} \times m); \)
7.   end
8. end
9. return \( \text{activeWord}; \)

Cost-wise, DecodeUp consists of more cases than DecodeDown. There is the use of an additional data structure called the alignedBlock which stores intermediate data such as remainder bits or partial words. There are four scenarios when decoding down depending on the activeBlock and the alignedBlock values. First, consider the case where the activeBlock is a fill and the alignedBlock is empty. The activeBlock is added to the currentWord with the number of segments being a fraction of the new segment length with any remaining bits being appended to the alignedBlock as a literal. The second scenario is when the activeBlock is a literal and the alignedBlock is empty. The literal is appended to the alignedBlock. The third scenario is when the alignedBlock is not empty and the activeBlock is a literal. The alignedBlock is then filled with bits from the literal in activeBlock until it is a complete segment. The fourth and final scenario is when the alignedBlock is not empty and the activeBlock is a fill. This is the example that is presented in Figure 2(b). Literals are subtracted from the fill to complete the alignedBlock. In this example, all segments in the fill are needed to complete the block. In the case there are remaining segments in the activeBlock, they are added to the activeWord as a fill.

**Figure 2: Segment Length Resolution Example**
In this section, we present our proposed optimizations to VAL's query engine. We first discuss a heuristic for choosing a translation method when processing queries between columns with different segment lengths. Next, we present the various column orderings we propose to investigate for carrying out range queries. Finally, we discuss how the metadata can be collected at compression time to facilitate our optimizations.

3.1 Point Queries: Single Logical Operations

For fixed-length bitmap compression algorithms such as WAH, processing a query by applying a single logical operation is straightforward. The variable-length approach of VAL presents several unique challenges. The first is translating a column compressed with segment length \( s \) to that of a column compressed with segment length \( m \times s \), or vice versa. The previous section presented two solutions to this problem, i.e., \( \text{DecodeDown} \) and \( \text{DecodeUp} \). The existence of two possible approaches for translation provides a second challenge: when to use one translation over the other.

Accurate selection of the most efficient decoding method used to answer a query of the form \( A \circ B \) where \( \circ \) is a bitwise operator, relies on \( a \text{ priori} \) knowledge of the characteristics of \( A \) and \( B \), including: the segment lengths \( a \) and \( b \), the number of literal atoms, the number of fill atoms and the length of the runs they encode. Because gathering this information at query time would require both reading the columns into memory and completely parsing them, it is clear that any potential gains made by selecting the most efficient decoding method would be lost. We propose a heuristic that uses information gathered at compression-time to estimate the efficiency of each decoding method.

Our heuristic assumes that the efficiency of a decoding method depends on the amount of parsing required. Since no additional translation cost is incurred when both columns are compressed using the same segment length, we assume \( a < b \). Calculating the number of parses required by \( \text{DecodeDown} \) to process a query is straightforward. Before any translation penalty is incurred, each atom in both \( A \) and \( B \) will require one parse operation. Since \( B \) has the larger segment length, it will require additional parsing, incurred by the literal atoms: each literal atom of \( B \) must be parsed into \( b/a \) smaller atoms. \( B \)'s fill atoms will require no further parsing, their run lengths are simply adjusted to reflect the new \( \text{segLen} \).

The following formula calculates the parsing costs of processing a query using \( \text{DecodeDown} \):

\[
p_{\text{down}} = \left( \frac{b}{a} \times B_{\#\text{fills}} \right) + A_{\#\text{atoms}} \quad (2)
\]

where \( B_{\#\text{fills}} \) and \( A_{\#\text{atoms}} \) represent the number of fill and literal atoms in \( B \) respectively. \( A_{\#\text{atoms}} \) represents the total number of atoms in column \( A \).

Calculating the parsing cost required by \( \text{DecodeUp} \) is more complex. Since column \( B \) has the larger segment length, it will require no further parsing past isolating its individual atoms. Column \( A \) will incur the translation costs. Uniquely, \( \text{DecodeUp} \) actually presents opportunities to reduce the number of parses needed for literals in column \( A \) to under 1. This is due to VAL's word format: all flag bits for the atoms contained in a word are placed in the most significant bits. Thus, if a word in a column with a segment length of 15 contains only literal atoms, the 4 most significant bits of the word would be 0000 and the remaining bits is a segment of 60 literal bits. If this column were to be translated to segment length 60, it may be possible to translate it without further parsing, essentially saving four parses. Note that this is a specialization of Algorithm 3 which is not shown in the pseudocode for brevity. It will not be possible to realize these savings for all such "literal-words" because a previous word might not have translated exactly and thus the alignedBlock would contain leftover segments. In this situation, literals would have to be parsed from a literal-word to compensate for the leftovers.

Where the cost of parsing a literal is slightly reduced, using \( \text{DecodeUp} \) adds a slight cost to the parsing of the fill atoms of \( A \). Consider a fill atom from \( A \) which encodes a run length of \( R \). If \( R \mod b/a \neq 0 \), then the atom cannot be directly translated to a run of \( b \) sized segments. A total of \( R \mod b/a \) segments will need to be removed from the run and treated as literals added to the alignedBlock. We use the following formula to estimate the parsing costs of processing a query using \( \text{decodeUp} \):

\[
p_{\text{up}} = \gamma \left( \frac{b}{a} \times A_{\#\text{fills}} \right) + \tau A_{\#\text{atoms}} + B_{\#\text{atoms}} \quad (3)
\]

where \( 1 \geq \gamma, \tau \geq (a/b) \). A value of 1 for \( \gamma \) implies that every fill-value in \( A \) encodes run-lengths that are congruent to \( b/a - 1 \) mod \( b/a \). This would be the worse-case and requires the most parsing. A lower \( \gamma \) value implies less parsing is needed due to fewer incongruent run-lengths. \( \tau \) value of \( a/b \) implies that all literals in \( A \) are stored in literal-words and no additional parsing is needed to translate them to \( b \). A higher value implies the existence of fewer literal-words. These parameters could be thought of as tuning variables. Their ideal values would change from column to column.

Our Dynamic Translation Selection heuristic selects the decode...
method that is estimated to require the least number of parses using the above formulas. We evaluate its performance in Section 4.

3.2 Range Queries

For discretized or binned columns, range queries are implemented using the or-of-or’s form.

\[(A_1 \lor b_1 \lor ... \lor A_1 \lor b_m) \lor ... \lor (A_n \lor b_1 \lor ... \lor A_n \lor b_m)\]

where \(A_i, b_j\) denotes the \(j\)th bin of attribute \(i\). A previous study on WAH showed that the order in which the OR operations are applied to the columns of a range query can impact the overall execution time [10]. When columns \(A\) and \(B\) are OR-ed together, the resulting column can be at most \(\text{size}(A) + \text{size}(B)\) large. Thus, given a series of WAH columns that generates the worse case, each successive operation produces a column that has a size equal to the sum of the sizes of the operands. Ordering the column in increasing order of size provides an optimal query execution time [10][11].

However, it is not obvious what effect column ordering would have on VAL range queries, because the assumption that execution time is proportional to column size does not necessarily hold for VAL. For example, an operation applied to two columns with a segment length of 15 might take longer than the same operation being applied to two columns with a segment length of 60 even though the 60 columns might be larger. This discrepancy is due to the increased parsing incurred by columns compressed with shorter segment lengths. Another factor that must be considered when ordering columns is the translation cost associated when applying an operation to columns with different segment lengths.

Another design decision involves the column holding the intermediate (and final) result. One way (Accumulative) is to allocate temporary column \(T\) to hold the result column. When a column \(A\) is dequeued, the logical operation is applied directly \(T \leftarrow A \circ T\). This process continues until the priority is empty. Another processing method (Reentrant) is to dequeue two columns \(A\) and \(B\), and apply the logical operation. The resulting column is then re-enqueued into the priority queue. This is the approach taken by Wu, et al. [10]. The Reentrant approach can, in the worst case, require that \(\log_2(n)\) temporary result columns be kept in memory, where \(n\) is the total number of columns being queried. The temporary columns that are generated by both approaches will require the needed metadata (e.g., size) be collected while they are being generated. We illustrate these two approaches in Figure 3.

3.3 Column Metadata

The above querying algorithms rely on a priori knowledge about the bitmap columns being processed. We propose a process of gathering column metadata at compression time and storing it on disk. Gathering the information has a very low overhead and would not significantly impact the time needed to compress. The size of the metadata file will be proportional to the number of columns in a bitmap. However, depending on the amount of data being collected, the size of the metadata file will be insignificant when compared to the size of the compressed bitmap. For example, our point query algorithm requires 4 pieces of information per column: segment length, column size, number of literal atoms, and number of run atoms. This information can be stored in a byte for the segment length, and 4 32-bit integers for the remaining information. Prior to query processing, the metadata file can be read into memory and referenced as needed. This does slightly increase the memory footprint. However, this is much smaller than reading all the columns into memory, which is required to order by size without metadata.

4. EXPERIMENTAL RESULTS

In this section, we present an evaluation of our algorithms over both real and synthetic data sets. First, we describe the experimental setup. All experiments were performed on a machine equipped with two Intel Xeon E5-2630 processors (six 2.3 GHz cores with hyperthreading enabled), 128 GB DDR3 RAM, running Windows 7 Professional. Each experiment was repeated six times: results from the first run were discarded to warm the cache, and the average of the subsequent five runs are reported.

4.1 Data Preparation

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

**Synthetic Data:** We generated 8 data sets, each containing 32 million rows and 10 attributes. The attributes’ cardinalities vary between 10, 20, 40, and 80 resulting in bitmaps containing 100, 200, 400, and 800 columns, respectively. For each of these settings, two different distributions were used: uniform and zipf. The uniform distribution emulates real data sets that are discretized using equally-populated bins. In this approach, the data is binned such that each bin contains roughly the same number of objects, thus creating a uniform distribution of 1’s in the bitmap. The zipf distributions represent a clustered approach to discretization. In this process, the density of data is represented in the bitmap, creating a skewed distribution. The zipf distribution generator assigns each bit a probability of:

\[p(k, n, \text{skew}) = (1/k^{\text{skew}})/\sum_{i=1}^{n}(1/i^{\text{skew}})\]

where \(n\) is the number of elements determined by cardinality, \(k\) is their rank, and the coefficient \(\text{skew}\) creates an exponentially skewed distribution. The distribution is uniform when \(\text{skew} = 0\). We generated data sets for \(\text{skew} = 0\) (labeled uniform) and \(\text{skew} = 2\) (labeled zipf2).

**Real Data:** For our real data experiments, we use the data set...
from KDDCup’99 (labeled KDDCup), which captures network flow. The data set contains 4,898,431 rows and 42 attributes. Continuous attributes were discretized into 25 bins using Lloyd’s Algorithm [12]. In total, there were 475 bins/columns. We run the range queries shown in Table 2.

All data sets were compressed using segment lengths \( s \in S = \{15, 30, 60\} \) in round-robin fashion, i.e., \( A_1, b_1 \) compressed with \( s = 15 \), \( A_1, b_2 \) with \( s = 30 \), \( A_1, b_2 \) with \( s = 60 \), etc. To ensure an equal distribution of segment lengths used.

4.2 Point Queries

The first set of experiments evaluates the performance of point query execution. We performed point queries (e.g., AND/OR operations) over two randomly selected columns from our data sets. To test the performance of DecodeUp and DecodeDown, we only selected columns with mismatched segment lengths. We ensured an even distribution: 1/3 of our queries were of each variety, i.e., 15:60, 15:30, and 30:60.

Figure 4: Point Query Execution Time

The run-settings are defined as follows: (a) All Up is when DecodeUp() is used to execute every query, (b) All Down refers to when DecodeDown() is used for every query, (c) Dynamic refers to the use of our equations from Section 3.1 to inform on decoding directions, and (d) Optimal is when the all decode directions minimize execution time.

Let us first consider the results for zipf2. The results for executing AND and OR 5000 queries are shown in the left-hand and right-hand side of Figure 4(a), respectively. Notice that AND queries are much faster to process than OR queries. This is due to 0s, the predominating value in skewed data, short-circuiting the logic. Our heuristic provided the closest performance to the optimal, coming only within 1.8% and 3.5% of the optimal for AND and OR queries, respectively. The next closest setting, using All Up, came within 6.9% and 4.6% of the optimal. We ran 1000 queries over KDDCup, and show the results in Figure 4(b). Our heuristic came within 17.6% and 18.2% of the optimal, while the next closest setting (again, All Up) came within 23.8% and 23.1% of the optimal for AND and OR queries, respectively. Although our heuristic was less accurate over this data set, it still obtained significantly higher performance than simply using a uni-directional scheme. We leave the calibration of our heuristic for future work.

In both data sets, we can see that All Down suffers for both types of queries. The reason for this can be seen by again investigating our formulas for estimating the parsing costs of DecodeUp and DecodeDown (Section 3.1). If the worse case is assumed for DecodeUp, i.e., \( \gamma = \tau = 1 \), the formula for when DecodeDown is preferable, \( P_{up} > P_{down} \), can be simplified to \( A_{opt} > B_{opt} \). This implies that DecodeDown is only optimal when the columns with the shorter segment length \( A \) has more fill atoms than \( B \) does literal atoms. This relationship was observed for all instances where DecodeDown was more efficient. In general, it is expected that a compressed bitmap would contain more literals than fills. If a run is interrupted, it is likely to result in at least one fill-literal pair. In the entire compressed version of KDDCup, there were 750K literal atoms and only 189K fill atoms.

4.3 Range Queries

We are interested in understanding the effects of our metadatadriven ordering algorithms for computing range queries. Recall the ordering schemes from Section 3 where columns \( A, b \) can be ordered by size (size), largest segment length first (lsf), or weighted size (ws). We compare these orderings with a random column ordering (random). Each of these orderings was evaluated under six query processing algorithms: DecodeUp Accumulation, DecodeUp Reentrant, DecodeDown Accumulation, DecodeDown Reentrant, Dynamic Accumulation, and Dynamic Reentrant. The DecodeDown algorithm decodes down in translation, but due to its performance substantially slower than both Dynamic and DecodeUp algorithms (also observed in the Point Query evaluation), its results are not included in the interest of space.

4.3.1 Synthetic Data

We first used range queries over: uniform and zipf2. Using Eq. 4 for data sets with bin cardinality 10, 20, 40, and 80, we order together 25, 50, 100, and 200 columns, respectively.

In Figure 5(a) and Figure 5(b) we show the range query execution time for DecodeUp Accumulation and Dynamic Accumulation, respectively, over uniform data. It is interesting to note that for both of these approaches, size generates the slowest results. This is because the smallest columns of uniform all have a segment length of 15. Unfortunately, the size difference between any two columns is not large. In uniform with cardinality 80, the largest column is only 2.5x larger than the smallest column. In contrast, the largest column in zipf2 with the same cardinality is 705x larger than the smallest column.

The size ordering favors columns with segment length of 15, which have a few more fill atoms than other columns, but those gains are lost after applying multiple logical OR operations. Because the result column will have a segment length of 15 until it is queried with a column of a greater segment length, it will be roughly the same size as the rest of the columns, but will require
Figure 5: Execution Time (uniform)

Figure 6: Execution Time (uniform, Row-Ordered)
much more parsing. The other three orderings all achieve approximately the same times. This is because they all have a column with a segment length of 60 as one of their first three columns. Hence, the result column will have a segment length of 60 for the majority of the operations. Since the column sizes are so close, this reduction of parsing appears to be the key in query speedup. Dynamic selection of the translation method does provide a slight advantage for uniform offering a speedup of 1.14× for ws, the most efficient ordering.

The results for DecodeUp, Reentrant applied to uniform data are shown in Figures 5(c). The Dynamic, Reentrant results are shown in Figure 5(d). The timings for Dynamic, Reentrant are actually slower than its Dynamic, Accumulation counterpart for all orderings and all cardinalities. This suggests that our translation selection heuristic is not well-suited for the Reentrant approach. We believe this is because, even though DecodeDown might be most efficient for two columns, it can have negative ramifications on the remaining unprocessed columns. The consequences of a wrong decoding choice is magnified when conducting a range query. By examining the decoding method chosen by the Dynamic approach, we see that for the Dynamic, Reentrant, DecodeDown was selected more times than in the Dynamic, Accumulation approach.

Several previous studies have shown that row reordering can drastically improve compression and query performance [13][14][15]. One popular ordering schemes is graycode, i.e., two adjacent rows differ by only one bit. In general, graycode maximizes runs in the first several columns of the data, and then the runs degrade into shorter runs before deteriorating into noise in the last several columns. To evaluate graycode’s effect on our query algorithms, we sorted the rows using graycode ordering before compressing. The graphs shown in Figure 6(a) present the execution times of our range query algorithms on graycode sorted uniform data.

With row ordering, we see more variance in the execution times of the column orderings. Figure 6(a) presents the results for DecodeUp, Accumulation applied to graycode-ordered uniform. As can be seen, ws slightly outperforms size and both are approximately 1.3× to 1.5× faster than random ordering and 1.2× to 1.25× faster than lsf. This speedup is due to the size differential of the columns. Because graycode creates very long runs in the first few columns, some of the most compressed columns have a segment length of 60. By decoding up, the answer column will become 60 and reduce the total amount of parsing needed. The reason size and ws are beating lsf is because some of the columns compressed using a segment length of 30 and 15 are small enough to offset the increased parsing cost. Since lsf orders first by segment length and then by size, it does not realize the benefits of these very small columns. As shown in Figure 6(b) the Dynamic, Accumulation algorithm was less efficient than DecodeUp, Accumulation for all orderings.

There is very little difference in the timing results of ws for DecodeUp, Reentrant (Figure 6(c)) and approaches that use Accumulation. Though the Reentrant algorithm changes the ordering of the columns for the graycoded uniform data, the overall order is not radically different than that of the Accumulation approach. For cardinality 80, a typical temporary result column is used within 3 to 8 logical operations of its creation. This implies that the result columns of the logical operations are generating roughly the same number of atoms as the operand columns. The size ordering is performing slightly worse under DecodeUp, Reentrant. This is because with larger cardinality, size again begins to favor columns of segment length 15 and its performance degrades. Again, Dynamic, Reentrant (Figure 6(d)) is slower than Accumulation, Reentrant. This relationship of DecodeUp being faster than

Next, we move on to skewed data. Figure 7(a) shows the results of DecodeUp, Accumulation applied to zipf2, without row-ordering. Again, we see that ws is the most efficient ordering and is 1.5× faster than random and 1.2× faster than size, for high cardinality. The size ordering is only slightly faster than lsf. In Figure 7(b) we observe the first significant speedup achieved by the Reentrant scheme. On ws, Reentrant achieves a speedup of 2.09× over Accumulation for cardinality of 80. This speedup occurs because, unlike the uniform data set, the growth of the result column does not remain approximate to the size of the operands. When ws is applied to cardinality 80, the first temporary result column is not removed from the queue until 32 operations had been processed. The fact that size is doing almost as well as ws indicates that many of the columns with shorter column lengths are aggressively compressed.

The lsf order does not see a substantial improvement (only 1.24× speedup) for the Reentrant approach. This is because after all of the columns of segment length 60 are processed, the Reentrant scheme essentially becomes Accumulation. Since DecodeUp always produces a result column with a segment length of 60 for lsf, after all of the columns of segment length 60 are processed, the first column polled from the priority queue will be the result column. This is the exact pattern of Accumulation for lsf. Figure 8 shows the results of our algorithms applied to graycode-ordered zipf2. The pattern of the results mimic those observed in unordered zipf2, only faster. The ordered data set produced results for ws that were over twice as fast as unordered.
4.3.2 KDDCup'99 Data

Table 2 shows a set of real queries posed over our KDDCup data set. The Selection Criteria column describes the data being selected for each query, and Cols displays the number of bitmap columns that must be processed to answer the query.

<table>
<thead>
<tr>
<th>ID</th>
<th>Selection Criteria</th>
<th>Cols</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>≤ 50% of connections to different hosts for the same service and ≤ 10% REJ and ≤ 10% SYN error. This query could indicate that the host is accepting the connections and servicing them.</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>≤ 50% of connections to the same host (or &gt; 50% to different hosts) were for the same service and less than 10% have REJ errors and &lt; 10% SYN error. A client is primarily checking other hosts for a specific service and is not getting rejected often.</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>Connections with REJ or SYN errors to the same host, but not necessarily the same service</td>
<td>39</td>
</tr>
<tr>
<td>4</td>
<td>src_bytes=2048 and dst_bytes &lt; 1843. Indicates at least a 10% loss of data in the connection.</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>≤ 10% SYN and &lt; 10% REJ error for the same service and the same hosts. This could be used to see if some kind of connection that is an attack is actually getting through.</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 2: Queries over KDDCup Data Set

Figures 8(a) and 8(b) show the time it took our different orderings to answer the queries using both DecodeUp, Accumulation and DecodeUp, Reentrant algorithms. Again, we observed that Dynamic selection did not significantly improve query performance, so we will not include those results. For queries 1 and 2, we see that size is actually slower than random when using the Accumulation algorithm. In these two queries, the smallest columns had a segment length of 15, but many were not compact enough to compensate for the increased processing costs. The lsf ordering is also slower than random for query 1. This is because several of the 60 columns in this query were some of the larger columns being processed. This meant that the result column of lsf grew very quickly. Similar to the synthetic data results, the ws ordering outperformed all others for all queries.

As shown, there is only minor differences between the times of Accumulations vs. Reentrant for the fastest ordering, ws. This reaffirms our observation that significant speed up from the Reentrant algorithms is only realized for high cardinality queries. Though, it is interesting to note that the size ordering did perform better under the Reentrant scheme, though still not out performing ws.

4.4 Summary of Results

The salient findings from our experiments are as follows. For point queries, our dynamic translation selection heuristic consistently came within 3.5% of the optimal time for 1000 OR queries, and was more efficient than simply always using DecodeUp or DecodeDown. However, our heuristic did not extend well to range queries. For range queries, always selecting DecodeUp appears to be the most efficient processing method. Our study suggests that for data sets observing uniform distribution, the Accumulation approach to range query processing is the most efficient. The overhead of a Reentrant approach is not always surpassed for such data. However, for skewed data, the Reentrant approach appears to be much more efficient, providing up to a 2× speedup for higher cardinality data. Regardless of the type of data and the processing algorithm used, the ws column ordering almost always produced the fastest times.

5. RELATED WORK

Our work relates to bitmaps indices compressed using VAL, however, there are many other bitmap compression algorithms. One of the earliest schemes was the Byte-aligned Bitmap Code (BBC) [16]. BBC is a patented encoding that uses four types of byte-aligned
atoms. The use of a 7-bit segment length allows BBC to compress compactly but is very CPU intensive when querying. The Word-Aligned Hybrid (WAH), which uses a word-aligned encoding, has been shown to typically use 60% more space but executes queries up to 12× faster [8].

Several variants of WAH have emerged recently. PLWAH [17] and Concise [18] both modify WAH’s version of a fill atom. They reserve \( \lfloor \log_2 w \rfloor \) bits, where \( w \) is the total number of bits used for a fill atom. If the run being represented was interrupted by a near-fill segment, a segment containing a single dirty bit, the reserve bits are used to indicate the position of that dirty bit. Thus, there is no need to create an addition literal atom. PLWAH encodes near-fills that follow a run and Concise encodes those before a run. This approach can achieve \( 2 \times \) the compression of WAH. Enhanced WAH (EWAH) also modifies the formatting of fill atoms [19]. EWAH divides the fill atoms in half. The upper half (the most significant bits) are used to encode the flag bit, value bit and run length in the same manner as WAH. The lower half is used to encode the number of literal atoms that follow the fill. When processing queries, this extra information allows EWAH to skip literal words when comparing to large runs. This extra information does mean that, in some instances, EWAH will need to use two words to encode long runs, whereas WAH would only use one.

We recently introduced Variable-Aligned Length code (VAL) [9] and compared it to WAH, PLWAH, and EWAH, and found that for skewed and sorted data, VAL achieved the best compression. We also compared timings of point query operations, using only DecodeDown algorithm, which our current study has suggested is slower than DecodeUp for most queries. Variable Length Compression (VLC) [20] could be considered a generalization of VAL. Like VAL, it allows each bitmap column to be compressed using a separate segment length. Unlike VAL, the segment lengths can be arbitrary. Corrales, et al. showed that using truly arbitrary lengths was impractical when processing queries [20]. Instead, they suggested limiting lengths to \( m \times b \) where \( b \) is a common base. When two columns compressed using different segment lengths are queried, both are translated to the greatest common divisor of the two lengths. It was shown that the need to translate both columns could lead VLC to be 3 to 4 times slower than VAL. This high translation cost suggests that VLC may be amenable to query algorithms that clusters columns compressed using the same segment length similar to that of our last ordering.

Our work was inspired by Wu, et al. [10]. They evaluated the effect of ordering on range queries for both WAH and BBC. They also implemented a priority queue algorithm similar to the one we used. Our study varies from theirs in several significant aspects. We addressed the issue of column translation which is unique to variable-aligned compression schemes. We investigated orderings which are unique to VAL. We also used metadata to reduce the memory foot print of our query engine and evaluated our algorithms on both real and synthetic data. Wu, et al. proposed an in-place algorithm where one uncompressed columns is used to store all the intermediate results from an aggregate range query to avoid compression costs. We plan to evaluate such an approach for VAL in the future.

6. CONCLUSION AND FUTURE WORK

Variable Aligned Length (VAL) Compression is a tunable framework that allows for bitmap indexes to be compressed using varying segment lengths. In this paper we explored optimizations to VAL’s query engine. When VAL performs a logical operation on two columns that were compressed using different encoding lengths, one of the columns must be translated. We evaluated benefits of using two translation algorithms DecodeDown and DecodeUp. Our translation select heuristic determined which algorithm to use by analyzing parsing costs. The heuristic performed better than always using either DecodeDown or DecodeUp, and was within 3.5% of the optimal for or queries on skewed data, slightly besting DecodeUp. For range queries, we implemented column orderings that were dependent on metadata gathered at compression time. The \( \times f \) ordering, which weights size by column composition, method was tested under five query processing algorithms, and was consistently able to outperform other orderings. The results of our study also suggests that an accumulation approach to query processing is ideal for uniform data and a reentrant priority queue approach is better suited for skewed data.

In the future we plan to extend our translation selection heuristic to make it more amenable to range queries. We think it would be interesting to investigate the effects of column orderings on sequences of logical AND operations. Additionally, there may be other range query algorithms which could be implemented in VAL.

7. REFERENCES