Dynamic Bitmap Index Recompression through Workload-Based Optimizations

Fredton Doan
Washington State University
fredton.doan@wsu.edu

David Chiu
Washington State University
david.chiu@wsu.edu

Brasil Perez Lukes
University of St. Thomas
pere2979@stthomas.edu

Jason Sawin
University of St. Thomas
jason.sawin@stthomas.edu

Gheorghi Guzun
The University of Iowa
gheorghi-guzun@uiowa.edu

Guadalupe Canahuate
The University of Iowa
guadalupe-canahuate@uiowa.edu

ABSTRACT

Many large-scale read-only databases and data warehouses use *bitmap indices* in an effort to speed up data analysis. These indices have the dual properties of compressibility and being able to leverage fast bit-wise operations for query processing. Numerous hybrid run-length encoding compression schemes have been proposed that greatly compress the index and enable querying without the need to decompress. Typically, these schemes align their compression with the computer architecture's word size to further accelerate queries.

Previously, we introduced *Variable Length Compression* (VLC), which uses a general encoding that can achieve better compression than word-aligned schemes. However, VLC's querying efficiency can vary widely due to mismatched alignment of compressed columns. In this paper, we present an optimizer which recompresses the bitmap over time. Based on query history, our approach allows the VLC user to specify the priority of compression versus query efficiency, then possibly recompress the bitmap accordingly. In an empirical study using scientific data sets, we showed that our approach was able to achieve both better compression ratios and query speedup over WAH and PLWAH. On the largest data set, our VLC optimizer compressed up to 1.73× better than WAH, and 1.46× over PLWAH. We also show a slight improvement in query efficiency in most experiments, while observing lucrative (11× to 16×) speedup in special cases.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Indexing Methods

Keywords

bitmap, indexing, compression, modeling, optimization

IDEAS'13 2013, October 9-11, Barcelona [Spain]

1. INTRODUCTION

Scientific research, vital for advancing our understanding of the world, is becoming increasingly data-intensive. For example, exploration within bioinformatics generates terabytes of data per day [1]. In high energy physics, the Large Hadron Collider at CERN is projected to generate 15 petabytes of data annually [2], and the STAR [3] project collects billions of colliding particle events. The mounting data sets, coupled with the growing memory gap, complicates our quest for fast analytics. To this end, such large-scale applications store their observations or simulations in data warehouses, which employ advanced indexing techniques.

One important technique used to represent big data is the *bitmap index* [4, 5, 6], adopted by popular systems including Oracle, IBM, and Hadoop Hive [7]. Bitmaps are leveraged for their efficient query processing through bit-wise operations, and they can moreover be compressed aggressively to ensure memory residency. To this end, specialized compression techniques have been designed to avoid explicit decoding during query execution. Most of these compression codes are *hybrid*, in that they combine run-length encoding with literal representation of bit strings. Hence, the compressed index usually consists of a set of *run* and *literal* codes [8]. Furthermore, current hybrid codes are word-aligned (*e.g.*, 32 or 64 bits) to further reduce access overhead [9, 10, 11].

Recently, we proposed the Variable Length Bitmap Code (VLC) [12], which uses variable-length codes (instead of restricting to the word size) to maximize compression. Query execution in this case is more expensive because misalignments need to be resolved at query execution time. However, there are still cases in which the queries executed using variable lengths outperform word-aligned approaches. There are two reasons for this anomaly. First, VLC may produce a much smaller, cache-friendlier index. Second, the frequently queried bitmap columns may be considerably compressed using encoding lengths that share a large greatest common divisor (*gcd*). The *gcd* is used to align the two bitmap columns without requiring explicit decompression of the bitmap. Experiments show that larger *gcd*'s translate to faster query processing but worse compression ratio.

Our work in this paper exploits opportunities for optimizing VLC's space-time tradeoff. The proposed framework allows users to specify the priority between compression ratio *vs.* query efficiency. We formalize this as a constrained bi-objective optimization problem and provide an efficient

heuristic that can adaptively adjust and recompress columns with an optimized VLC encoding length. The main contributions of this paper can be summarized as follows.

- We leverage query history for the dual purpose of compressing the infrequently queried columns more aggressively, and align the frequently queried columns for faster processing.
- We introduce a trade-off parameter that allows the user to prioritize between compression ratio and query efficiency.
- We define a novel constrained bi-objective optimization problem and propose an efficient greedy heuristic to approximate solutions in polynomial time.
- We evaluate our approach using data sets generated from real applications and compare it to two dominating compression schemes, WAH and PLWAH.
- We present an extensive discussion on lessons-learned and the limitations of our approach.

The rest of this paper is organized as follows. Section 2 presents background on bitmap compression and processing. Section 3 presents a formalization of the constrained optimization problem and the system models we employ. To this end, we propose a heuristic to approximate the optimal solution. In Section 4, we evaluate our approach with an extensive experimental analysis using real data sets. Related works are presented in Section 5. Finally, Section 6 presents our conclusions and outlines future work.

2. BACKGROUND

A bitmap index is a coarse representation of a database relation that can be queried directly. Essentially, it is a two-dimensional array $B[m, n]$ where the n columns represent value *bins* and the rows correspond to m tuples in a relation. To transform a table into a bitmap, each attribute is first partitioned into bins that might denote a value or a range of values. In the simple bitmap encoding, an element $b_{i,j} \in B = 1$ if the jth attribute in the ith tuple falls into the specified range and 0 otherwise.

One important property is that bitmaps can be queried directly using fast, hardware-supported bitwise operations. For example, consider Table 1. It shows a bitmap index for a very simple relation consisting of two attributes: *age* and *Car Owner*. The *age* attribute has been discretized into three bins: $a_1 = [0, 20]$, $a_2 = [21, 40]$, $a_3 = [41, \infty]$. The *car owner* attribute requires two bins: *yes* indicating the individual does own a car, and *no* indicating the opposite. To find everyone under 21 who does own a car, the processor can apply a bitwise AND of a_1 and *yes*, and then retrieve the matching tuple t_2 from disk.

Although a single bitmap is an abbreviated representation of the data, bitmaps can still become too large to be contained in core memory. To this end, bitmaps are often compressed using *hybrid* run-length encoding (RLE) schemes. A pure RLE represents sequences of similar data values (*i.e.*, *runs*) as a (run-length, data value) pair. For example, an RLE would represent the sequence $\langle 00000001011 \rangle$ as $\langle (7,0)(1,1)(1,0)(2,1) \rangle$. In this example, compression is only achieved for the longer run of zeros. RLE schemes can in fact generate larger codes if long runs are not abundant

Tuples	**Columns (Bins)**				
	Age			Car Owner	
	a_1	a_2	a_3	*yes*	*no*
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 Index

in the data. To this end, a *hybrid* RLE uses two encoding atoms *fills* and *literals*. A fill atom is used to represent long runs in the data, as a *(run-length, data value)* pair. The literal atoms are used to represent short runs and sequences of noisy, heterogeneous data values verbatim. A hybrid RLE might encode the above example as $\langle (F, 7, 0)(L, 1011) \rangle$ where F and L are used to identify fill atoms and literal atoms, respectively.

One popular hybrid RLE used for bitmap indices is Word-Aligned Hybrid (WAH) [9]. It compresses each column of a bitmap independently using an encoding format that is aligned with the computer system's architecture. For a system that has a 32-bit word, it first partitions a column into segments 31 bits long (in general WAH uses a segment length of $wordsize - 1$). Segments that contain heterogeneous bits are encoded in literal atoms of the form: $(0V_1...V_{31})$. In this encoding the most significant bit (MSB) is referred to as the *flag*-bit and is used to distinguish between literal and fill atoms. A 0 indicates that the atom is a literal and that $V_1...V_{31}$ is the actual bit string of the corresponding segment. WAH combines consecutive homogeneous segments and encodes them in fill atoms of the form $(1FV_3...V_{31})$. The flag-bit value of 1 identifies the atom as a fill, the F (called the fill-bit) encodes the homogeneous value of the bits and $V_3...V_{31}$ is the binary representation of the run length, *i.e.*, the number of segments that were compressed. For example, a WAH word containing the value $0xC0000003$ would indicate that it was a fill atom representing a run of three groups of 31 1s.

By aligning its segment length with the memory subsystem, WAH is very efficient in query processing. To perform an AND query between two compressed columns, WAH processes a word from each column at a time. To process the words, the query processor first examines the MSBs to determine the type of atom. (1) If they are both literals, WAH returns a literal that is a result of a bitwise AND of the two words. (2) If one word is a fill and the other is a literal, it inspects the fill-bit of the fill atom. If the fill is a run of 1s, WAH returns the literal, otherwise it returns 0. It then reduces the number of segments encoded in the fill by 1 and fetches a new word to replace the exhausted literal. (3) Finally, if the two words are both fills, WAH returns a new fill that encodes the same run-length as the shorter of the two queried words. The fill bit of the returned word is the result of a bitwise AND applied to the fill bits of the two queried words. WAH then performs the necessary bookkeeping and fetches a new word to replace the exhausted fill. It is important to note that query processing in this third case observes a superlinear speedup.

The work presented in this paper leverages a previous work proposed by the authors: the Variable Length Compression (VLC) scheme [12]. Like WAH, VLC uses fill and

literal atoms. However, VLC allows the segment length to vary from $wordsize - 1$ to 4 bits per column. For example, VLC could compress one column using a segment length of 28 and the next column using a length of 14. It is worth noting that VLC is general, *i.e.*, WAH can be represented with VLC by fixing all columns to be compressed using $wordsize - 1$. The use of varying segment lengths allows VLC to compress smaller runs than WAH but it also reduces the efficiency of querying.

For segment lengths less than $wordsize/2 - 2$, each word requires additional parsing to retrieve the individual atoms[1]. There is also an additional cost when columns that were compressed using different segment lengths are queried together. VLC uses the (gcd) of the two segment lengths to translate each column so that they are aligned. It was demonstrated that, when using uncorrelated segment lengths selected to optimize compression VLC could achieve 2.5× better compression than WAH for some data sets [12]. However, the querying cost became prohibitive for such cases. In this paper we, present a tunable model that allows the user to select a preference for efficient querying or maximizing compression ratio.

3. PROBLEM DEFINITION

As we discussed in the previous section, VLC allows bit vectors to be compressed using any feasible *segment length*. This variability of segment lengths means that the alignment between any two bit vectors is not guaranteed. When two vectors are queried together, the performance is at the mercy of the greatest common divisor gcd between the segment lengths used to compress the two vectors. Therefore, in the best-case, and the gcd between any two segment lengths is $wordsize - 1$, then that particular query runs as fast as WAH. In the worst-case, the gcd between two segment lengths is 1 and both bit vectors must first be decompressed before running queries, which betrays the goal of bitmap compression schemes.

In [12], we suggested that VLC should be constrained to use only like-base segments. For instance, VLC-7 would only allow bit vectors to be compressed using segment lengths 7, 14, 21, and 28 in a 32-bit system, keeping any two bit vectors relatively aligned (worst case $gcd = 7$). Even with these constrained options, the tradeoff can be significant: a vector compressed in segment 7 is typically smaller in size, but slower to query. Conversely, a vector compressed in segment 28 may be faster to query, but larger in size. The insight we make in this paper is that a bit vector need not be bound to any particular segment length. Instead, we argue that a bit vector's *assignment* to a particular segment length should vary over time given the current state of the query behavior.

Abstractly, we will initially use VLC to compress the index down as aggressively as possible. Over time, as columns are queried together, we will possibly recompress the index such that the frequently-queried columns become aligned with larger segment lengths, while the sparsely-queried columns are compressed down aggressively to save space. To guide this *dynamic recompression* scheme, we leverage the query history among all columns over time. We devise new models and define the *Workload-Driven Bitmap Recompression Optimization Problem*, which requires an exponential-time solver. To this end, we propose a greedy heuristics to approximate the optimal solution.

[1]For segment lengths that are not a factor of $wordsize$ VLC packs $\lfloor wordsize/seglen \rfloor$ atoms into a single word and places $wordsize \bmod seglen$ 0s in the remaining bits, which will be ignored during querying.

3.1 System Models

Our objective is to obtain, per bitmap column, an optimal segment length that is both efficient for querying and also produces the smallest possible file size. To solve this optimization problem, we formulate a mathematical model to characterized the degree of relationship between file compression size versus query time. This model produces the target objective encoding lengths for the bitmap columns. For clarity, we define all the terms used in our model in Table 2.

Notation	Description
$C = \{c_1, \ldots, c_n\}$	The set of n columns in the bitmap index.
$S = (1, \ldots, m)$	A vector of segment lengths, where m is the machine's word size.
τ	Optimization time intervals.
H_τ	Query history for all columns at time τ.
$\hat{h}_{j,k}(\tau) \in H_\tau$	Estimated pairwise query frequency of columns c_j and c_k at time τ.
$h_{j,k}(\tau)$	Observed pairwise query frequency of columns c_j and c_k at time τ.
A	A segment length assignment for the n columns.
$a_{i,j} \in A$	Segment-to-column assignment bit: 1 if c_j is compressed with segment length i, and 0 otherwise.
usz_j	Uncompressed size of column c_j.
$csz_{i,j}$	Compressed size of column c_j with segment length $i \in S$.
$WCR(A, H_\tau)$	Workload compression ratio given history H_τ and assignment A.
$WQ(A, H_\tau)$	Workload query-time ratio given history H_τ and assignment A.
$\lambda \in [0, 1]$	Size/speed tradeoff parameter. A small value prefers compression, and a large value prefers query efficiency.
$i_u \in S$	The initial user-input segment length for our heuristic.

Table 2: Notation Reference

Let a bitmap index be represented by a set of n columns $C = \{c_1, \ldots, c_n\}$. In VLC, a column can be compressed using a segment length $S = (1, 2, \ldots, m)$, where m is the machine's word size. In our proposed recompression scheme, we monitor our system using a discrete time model. The time interval of interest is $\tau \in \{1, 2, \ldots, T\}$ for an arbitrary large T. We assume the database system receives an arbitrary number of queries over time, and we let $h_{j,k}(\tau) \in [0, 1]$ denote the pairwise query frequency between columns c_j and c_k observed within time τ and $\tau - 1$. To capture the querying behavior, we store a history,

$$H_\tau = \{\hat{h}_{j,k}(\tau) \mid 1 \leq j, k \leq n\} \quad (1)$$

where $\hat{h}_{j,k}(\tau)$ is the moving average of the query frequency between columns c_j and c_k at time τ,

$$\hat{h}_{j,k}(\tau) = \alpha \cdot h_{j,k}(\tau - 1) + (1 - \alpha) \cdot \hat{h}_{j,k}(\tau - 1)$$

This function weighs the most recently observed query frequency $h_{j,k}(\tau - 1)$ most heavily, while the remaining history

observes an exponential decay in their contribution in generating the prediction. The use of exponential moving average is well-established to model temporal trends due to its $O(1)-$time update operation [13, 14]. In our formulation, we let $\hat{h}_{j,k}(0) = 0\ \forall j,k$ be the base history and $\alpha = 0.7$.

Each column c_j in the bitmap has an initial uncompressed size of usz_j. We let $csz_{i,j}$ represent the resulting size for column c_j when compressed with segment length $i \in S$. We further define:

$$A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{pmatrix} \tag{2}$$

as a *segment assignment*, where $a_{i,j} = 1$ if segment length i is used to compress column c_j, and $a_{i,j} = 0$ otherwise. It is important to note that, because a column can only be compressed using one segment length at any time, $\sum_{i=1}^{m} a_{i,j} = 1$ must hold for all j.

Given a segment assignment A, the compression ratio of a column c_j can thus be defined,

$$\frac{\sum_{i=1}^{m} a_{i,j} \cdot csz_{i,j}}{usz_j} \tag{3}$$

Specifically, it is the ratio between the term $csz_{i,j}$ (which is the compressed size of c_j using the segment length i as given by A) and usz_j, which is the uncompressed size of c_j. The total index compression ratio is given intuitively by the summation across all columns c_j,

$$\frac{\sum_{j=1}^{n}\sum_{i=1}^{m} a_{i,j} \cdot csz_{i,j}}{\sum_{j=1}^{n} usz_j} \tag{4}$$

However, we need a compression ratio that can additionally capture the query history. This metric should reward the cases where the infrequently queried columns are compressed more aggressively. Given history H_τ and segment assignment A we define the *workload-based compression ratio* as follows:

$$WCR(A, H_\tau) = 1 - \frac{\sum_{j=1}^{n}\sum_{i=1}^{m}(1 - \frac{1}{n}\sum_{k=1}^{n}\hat{h}_{j,k}(\tau)) \cdot a_{i,j} \cdot csz_{i,j}}{\sum_{j=1}^{n} usz_j} \tag{5}$$

This model is derived directly from Eq. (4), but we factor in $\hat{h}(\tau)$. For a given column c_j, the term $\frac{1}{n}\sum_{k=1}^{n}\hat{h}_{j,k}(\tau)$ is its expected query frequency. We subtract this term from 1 to reward smaller compression sizes for columns with low query frequencies and reduce the penalty for columns with high query frequencies. Initially, when there is no history (*i.e.*, $\hat{h}_{j,k}(\tau) = 0\ \forall\ j,k$), the term reduces to Eq. (4). Finally, we take the difference from 1 because we will seek to *maximize* this normalized value later.

The next step is to model the *workload-based query efficiency*. We do not assume any prior knowledge of query response times, and instead leverage a simple relationship between query efficiency and column alignment. Previous studies [15, 8, 12] have validated that query performance between two columns increases if they are aligned, *i.e.*, compressed using the same segment length (best case) or multiples of the same segment length. Performance generally increases when the aligned segment lengths approach the machine's word size m. Specifically, this observation translates to greater query efficiency if the greatest common divisor (*gcd*) between the segment lengths used to compress any two columns approaches m. Given a vector of segment lengths $S = (1, 2, \ldots, m)$ and a segment assignment A at time τ, we define the workload-based query efficiency ratio as,

$$WQ(A, H_\tau) = \frac{1}{\beta} \cdot \sum_{j=1}^{n}\sum_{k=j+1}^{n} \hat{h}_{j,k}(\tau)\Big[\sum_{i_j=1}^{m-1}\sum_{i_k=1}^{m-1} a_{i_j,j} \cdot a_{i_k,k} \cdot gcd(i_j, i_k)\Big] \tag{6}$$

where

$$\beta = \frac{\sum_j^n \sum_k^n \hat{h}_{j,k}(\tau) \cdot (m-1) \cdot n!}{2(n-2)!}$$

The outer double-summation iterates through each combination column pairs c_j and c_k. For each combination pair, the inner summations produce the *gcd* of the segment lengths prescribed to c_j and c_k by the given map A. This term is further factored with the query frequency between the two columns. The $\frac{1}{\beta}$ term normalizes WQ to a range between $[0, 1]$. Specifically, we divide by $n!/2(n-2)!$, the size of all column pair combinations, the maximum attainable segment length $m-1$, and the total column history, $\sum_j^n \sum_k^n \hat{h}_{j,k}(\tau)$. Initially when there is no history, WQ reduces to 0 for any input segment assignment A. Conversely, if all history elements $\hat{h}_{j,k}(\tau) = 1$, WQ produces 1 if A assigns $m-1$ segment length to all columns.

3.2 Optimization Problem

With the two models WCR and WQ, we can now define the *Workload-based Bitmap Recompression Problem*. Given a segment assignment A and the query history H_τ, we obtain a pair of values $P(A, H_\tau) = (WCR(A, H_\tau), WQ(A, H_\tau))$. Because the range of values from both models is normalized to 1, we need only to minimize the distance to the ideal pair $P^* = (1, 1)$. Thus, we obtain the following cost objective,

$$dist\big(P(A, H_\tau), P^*, \lambda\big) = \sqrt{\begin{aligned}&\big((1-\lambda) \cdot (WCR(A, H_\tau) - 1\big)^2 + \\ &\big(\lambda \cdot (WQ(A, H_\tau) - 1\big)^2\end{aligned}} \tag{7}$$

This cost objective is an augmented version of euclidean distance between the two pairs. The newly integrated term $\lambda \in [0, 1]$ is a tradeoff parameter that allows users to adjust their preference between compression size and query speed. A smaller λ gives preference to higher compression ratio, while a larger value prefers query efficiency.

The goal of this objective is to find an optimal segment assignment A^* that would yield a pair with the shortest λ-augmented distance to P^*. Specifically, our constrained optimization problem can be defined as follows,

$$\underset{\mathbf{A}}{\textbf{Minimize}}\ \ dist\big(P(A, H_\tau), P^*, \lambda\big) \tag{8}$$

$$\textbf{Subject to}: \sum_{i=1}^{m} a_{i,j} = 1,\ \forall j \tag{C8.1}$$

$$\sum_{j=1}^{n} a_{i,j} = 0,\ \forall i < 4 \vee i = m \tag{C8.2}$$

Constraint (C8.1) ensures that only one segment length is assigned to a column. Constraint (C8.2) is needed to require that only feasible segment lengths are used for compression.

For instance, VLC cannot use a segment length m, which is the machine's full word length. Similarly, a segment length $i < 4$ do not have enough bits to represent useful information. Solving for the optimal solution A^* is NP-Hard with an $O(n^m)$ complexity, since we must evaluate all of A's possible bit-permutations. To this end, we next describe the design of a greedy polynomial-time heuristic that can provide approximate solutions.

3.3 Algorithm Design

In this section, we describe a greedy heuristics that can solve for an approximate solution. In our approach, referred to as Workload-based Greedy Bitmap Optimization (`wg`), we allow the user to input a segment length, $i_u \in S$. The algorithm computes A while confined to using only multiples of i_u so that query alignment is enforced. Recall that queries typically run faster given large *gcd* between the queried columns' segment lengths. To do this, we augment the original objective defined in Eq. 8 by adding a third constraint,

$$\sum_{j=1}^{n} a_{i,j} = 0, \text{ if } (i \bmod i_u) \neq 0 \qquad \text{(C8.3)}$$

The compression scheme that is produced by this approach is illustrated in Figure 1. Notice that the user input i_u serves as the base segment length, and all columns are compressed using a multiple of i_u.

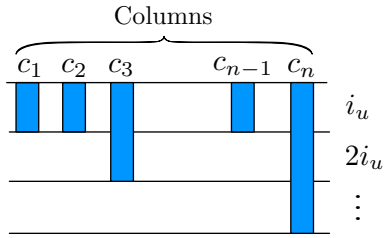


Figure 1: Segment Encoding Produced by `wg`

The heuristic to solve for A is presented in Algorithm 1. The algorithm inputs a set of columns $C = \{c_1, ..., c_n\}$, the user-input segment length i_u, the tradeoff parameter $\lambda \in [0, 1]$, and the current history H_τ then returns a segment assignment A. Presumably, the user enters i_u, $4 \leq i_u \leq m-1$ to meet constraint (C8.2). On Lines 2 and 3, we initialize the assignment $A[m, n]$ such that i_u is initially used to compress all columns. On Lines (4-6), we compute distance D, which is defined as the current-best distance to the ideal point P^*, as given in Eq. (7). Next, we iterate through each column $c_j \in CS$. To ensure that segment length i meets the new constraint (C8.3), i increments by i_u each iteration. We reset column j to use segment i by swapping the bit value with the previous iteration (Lines 12). The swapping must be done to ensure constraint (C8.1). Given this new setting, we compute the local distance d to the ideal again on (Lines 13-14). If this d yields is closer to D, then we save the current configuration into *temp* (Lines 15-18). After all the columns have been looped through, we return *temp* as the segment assignment for all columns (Line 22).

This algorithm is greedy because *temp* only holds the locally optimal assignment as we traverse over the columns.

Algorithm 1 `wg`(C, i_u, λ, H_τ)

```
1: {* Initialize segment assignment structure *}
2: A'[i, j] ← 0  ∀i, j
3: A'[i_u, j] ← 1  ∀j
4: P* ← (1, 1)
5: P ← (WCR(A', H_τ), WQ(A', H_τ))
6: D ← dist(P, P*, λ)
7: for all columns c_j ∈ C do
8:    i ← 2i_u
9:    while i ≤ m − 1 do
10:      {* update A' and compute the distance *}
11:      {* set column j to use segment i *}
12:      swap(A'[i − i_u, j], A'[i, j])
13:      P ← (WCR(A', H_τ), WQ(A', H_τ))
14:      d ← dist(P, P*, λ)
15:      if (d < D) then
16:         D ← d
17:         A ← A'
18:      end if
19:      i ← i + i_u {* consider only multiples of i_u *}
20:   end while
21: end for
22: return A
```

The outer-loop iterates over n columns, whereas the inner-loop iterates through to the maximum allowed segment length $m-1$ by step sizes of i_u. Each time through the inner-loop, we compute distance, which evaluates both $WQ(A, H_\tau)$ and $WCR(A, H_\tau)$. The computation for $WCR(A, H_\tau)$ is $O(mn)$ because the inner history summation can be stored incrementally. On the other hand, $WQ(A, H_\tau)$ is slighly more expensive: the computation of the normalizing factor η is $O(n)$ since it is dominated by computation of the factorial, again because the total history can be stored incrementally as a constant. To find the complexity of the main summations, we first note that we can precompute and store the possible *gcd* values in an array, therefore allowing us to ignore the gcd computation time. The computation of WQ is therefore $O(m^2n^2)$, resulting in an $O(m^3n^3)$ complexity for Algorithm 1. However, since m denotes the machine's word-length ($m = 32$ or 64 on today's machines), and because we iterate the inner-loop on orders of u_i, it can be held constant in asymptotic analysis, resulting in $O(n^3)$.

4. EVALUATION

In this section, we present an extensive evaluation of our approach over real data sets. First, we will explain the experimental setup and the data sets used for evaluation, then give an analysis of our experimental results.

We evaluate our system using the following real data sets. We have also evaluated our system using synthetic data, the results of which are presented in [16].

- `HEP` is generated by a real high-energy physics application. It contains 12 attributes, where each was discretized into ranges from 2 to 12 bins, resulting in a total of 122 columns and 2,173,762 rows in the bitmap. The uncompressed bitmap file is 285MB on disk.

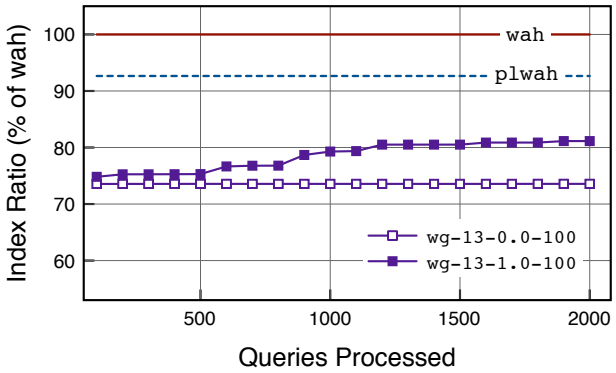
- `KDD99` is the full KDD Cup 99 data set [17], representing network traffic data for intrusion detection applications. It contains 4,898,430 rows and 42 attributes

that have been discretized into 474 columns. The uncompressed bitmap file is 2.37GB on disk.

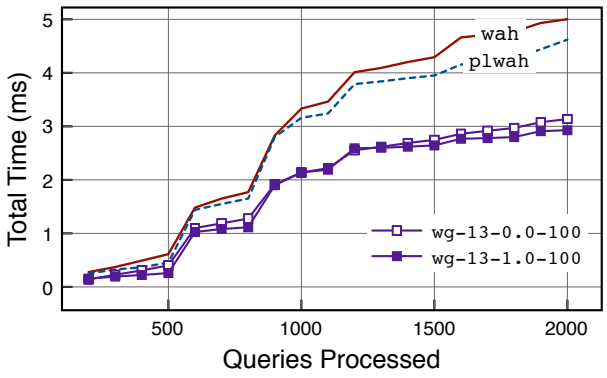
For the above data sets, we sorted the columns by order of access frequency in the query file, and sorted the row using grey code ordering to maximize runs [18, 19, 20, 21]. We generated query plans using the following method: Each query involves two columns c_j and c_k which are not necessarily distinct. These columns are drawn from a *zipf* distribution to observe the well-known *access skew*, *i.e.*, most queries access a small set of pages, which contain prime attributes, search-key attributes, etc [22, 23]. For `HEP`, we generated 2000 queries, and for `KDD99`, we generated 500 queries. All experiments were run on a Linux 3.4.6-2.10 machine with four AMD 48-core CPUs and 256GB of RAM.

4.1 Evaluating Cost Models and Optimization

In these first set of experiments, we are concerned with model validation. We compare our algorithm, Workload-based Greedy Bitmap Optimization (`wg`) to two popular bitmap compression schemes, Word-Aligned Hybrid (`wah`) [9] and Position-List WAH (`plwah`) [10]. In all plots, we label our algorithm using the convention `wg-x-y-z` to denote the experimental run's configuration: initial segment length i_u = `x`, λ = `y`, and τ = `z`. All experiments were executed six times. We discard the results from the first run to prevent Java's *just-in-time* compilation from skewing our results. We then average the results from the next five runs and record the values.



(a) Compression for $\tau = 100$ (`HEP`)



(b) Query Execution Time for $\tau = 100$ (`HEP`)

Figure 2: Evaluation on the `HEP` Data Set

4.1.1 HEP Data Set

Let us initially focus on the compression results for the `HEP` data set, shown in Figure 2(a). This figure shows the index size ratio as a percentage of `wah` along the vertical axis over the number of queries processed (horizontal axis). As expected, `plwah` compresses slightly better than `wah` (92%). The two lines `wg-13-0.0-100` and `wg-13-1.0-100` refer to our algorithm with user-input $\lambda = 0$ and $\lambda = 1$, respectively. To summarize and communicate the results succinctly, we fixed $\tau = 100$ (*i.e.*, optimize after every 100 queries), and we show $u_i = 13$ because it yielded the best results. As can be seen in Figure 2(a), `wg-13-0.0-100` compresses more aggressively due to $\lambda = 0.0$, reducing the index size to 73% of the size of *wah* index. The line `wg-13-1.0-100` reflects the results when $\lambda = 1.0$. It compresses the index in the range of 74.8% to 81.1% of the `wah` index size over time. This range indicates that the index was recompressed to grow over time to speed up query execution. In the worst case, we out-compress `wah` by 1.23× and `plwah` by 1.14×.

In Figure 2(b), we show the total time taken to process 2000 queries, optimizing after every $\tau = 100$ queries. The first 100 queries are used to both train our models and to warm-up the CPU cache. We therefore exclude the contribution of the first 100 queries from all experiments. `wah` and `plwah` completed the remaining queries in 5.00ms and 4.62ms, respectively. In comparison, our approach when λ is set for compression, `wg-13-0.0-100`, yields an execution time of 3.13ms, for a speedup of 1.6× over `wah` and 1.48× over `plwah`. When λ is set to prioritize queries, `wg-13-1.0-100` yields an execution of 2.92ms. This results in a speedup of 1.7× over `wah` and 1.58× over `plwah`.

We computed the average latency taken to process a single query, displayed in Table 3.

Algorithm	Latency (μs)
`wah`	2.63
`plwah`	2.43
`wg-13-0.0-100`	1.65
`wg-13-1.0-100`	1.54

Table 3: Average Query Latency (`HEP`)

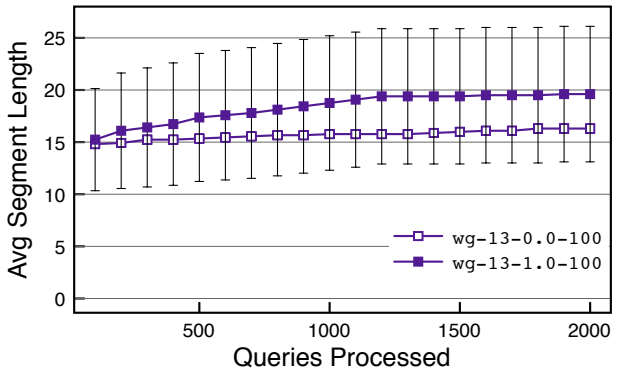


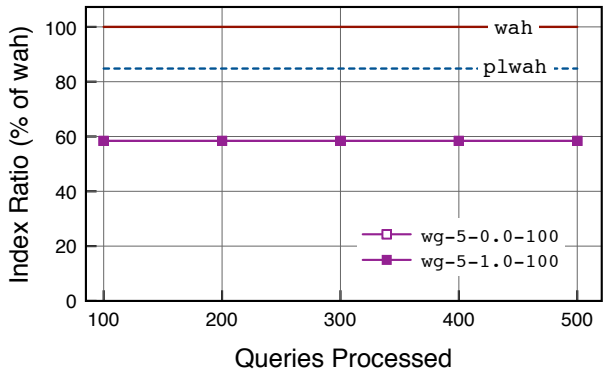
Figure 3: Average Segment Lengths (`HEP`)

To provide additional support for our models' effectiveness, we present the average segment lengths used to com-

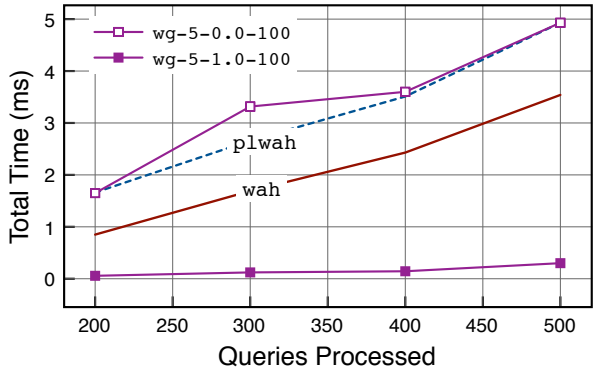
press the columns for the HEP data set in Figure 3. We display the average segment lengths generated using an initial segment length of $i_u = 13$ for $\lambda = 0.0$ and $\lambda = 1.0$. The vertical bars in the plot denote the standard deviations. In both settings, we can see that the average column segment lengths converge quite quickly. The standard deviations are shown only for wg-13-1.0-100, and these expectedly high due to a bimodal distribution: When $i_u = 13$, there are only two segment lengths (13 and 26) in which a word can be encoded. The distribution of HEP's 122 columns compressed using lengths 13 and 26 are 91 and 31, respectively after the final optimization. Take this result together with HEP results from Figure 2, we claim that our optimizer produces effective indices over time.

4.1.2 KDD99 Data Set

We now present the results from the same experiments executed over the KDD99 data set. Again for clarity, we fixed $\tau = 100$, and we show results for a single initial segment length, $u_i = 5$. Recall from earlier that the results from the first 100 queries are excluded for model training.



(a) Compression Comparison for $\tau = 100$ (KDD99)



(b) Queries Comparison for $\tau = 100$ (KDD99)

Figure 4: Evaluation on the KDD99 Data Set

Let us first focus on index size results, shown in Figure 4(a). First, note that plwah compresses the index to 85% of the wah index. Being a larger data set with nearly five million rows, opportunities for plwah to exploit its "position list" encoding abound, yielding greater compression. At first, it appears that the index sizes for both settings of our algorithm are equal due to the ostensible overlap in Figure 4(a), but there are in fact minor variations in the index size, varying from 58.388% to 58.39% of wah's index size. While the variations are negligible in the context of index size, we will see shortly that they significantly impact query performance. In comparison, both of our algorithms produces indices that were 58% the size of wah and 69% the size of plwah. Hence, we out-compress wah by 1.73× and plwah by 1.46×.

In Figure 4(b), we show the total query time, optimizing after every $\tau = 100$ queries. Again, the first 100 queries have been excluded due to initialization. wah and plwah completed the remaining queries in 3.54ms and 4.93ms, respectively. In comparison, our approach when λ is set for compression, wg-5-0.0-100, yields an execution time of 4.93ms. This is equal to the execution time of plwah, but is also 1.4× slower than wah. When λ is set to prioritize queries, wg-5-1.0-100 yields an execution of an astounding 0.3ms, resulting in an 11.8× speedup over wah and 16.4× over plwah. These results are surprising and dramatic even in this limited experiment of only 400 queries. We do not believe the speedup is a direct result of less decoding alone. Instead, we believe the adjustments in the compression over time rendered the index to be more cache-friendly. Clearly, a deeper treatment, including cache miss analysis, can validate this claim, which is planned for future work.

We computed the average latency taken to process a single query, displayed in Table 4.

Algorithm	Latency (μs)
wah	8.85
plwah	12.33
wg-13-0.0-100	12.33
wg-13-1.0-100	0.75

Table 4: Average Query Latency (KDD99)

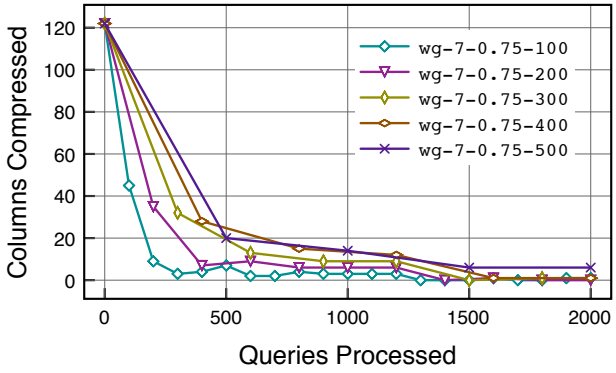
4.1.3 Optimization Interval (τ)

We now explore the effects of the optimization interval τ. We fixed $\lambda = 0.75$ to favor query speed. We also set the initial input segment length $i_u = 7$ due to its wider range of encodings (*e.g.*, 7, 14, 21, and 28). We vary the optimization window size τ over 100, 200, 300, 400 and 500 queries.

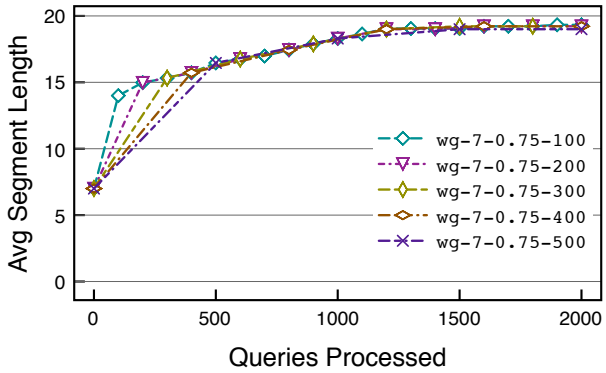
Figure 5(a) shows the number of columns for HEP that were (re)compressed at each optimization interval. Initially, all 122 columns are compressed using $s = 7$ as our design dictates. We can make two general observations: (1) As queries are processed, the number columns recompressed diminishes over time. This is due to two reasons. First, λ is configured to favor query performance, hence, as the columns are recompressed into a larger segment length, it will tend to stabilize. Second, the workload is drawn from a *zipf*-distribution, where a few dominating columns are queried the majority of the time. This skew causes the index to converge to a steady-state rapidly, *e.g.*, converging after roughly 500 queries. (2) The second observation is that a smaller τ tends to recompress more columns over time. For instance, after 500 queries, $\tau = 100$ has recompressed 68 columns, while $\tau = 500$ has only recompressed 20. This is due to the query history smoothing out over a longer period, leading to less columns requiring recompression.

The result in Figure 5(b), which shows the average segment lengths of the compressed index over the querying period, support the above arguments. We can observe the

steady increase in average segment length, converging at approximately 20. The distribution of the 122 columns compressed using lengths 7, 14, 21, 28 is 37, 73, 5, 13, respectively after the initial optimization, and 25, 37, 2, 58 after the final optimization. The curious observation is that τ does not appear to have much of an effect on the average segment length, as seen with convergence rates being equal after the initial 500 queries. This unexpected result suggests that we can optimize less frequently—a welcomed result, due to its high overhead costs. Although not shown due to space constraints, we also observed that the size of τ does not significantly impact total query times.



(a) Columns Recompressed Each Opt Interval (HEP)



(b) Average Segment Length Used in Index (HEP)

Figure 5: Optimization Interval τ

4.2 Overhead Costs

Given an initial segment length $i_u = 31$ our approach (*i.e.*, `wg-31-*-*`) reduces to that of `wah`. However, due to the segment-length generality of our compression framework, its querying logic is a bit more complex. Our approach must first compute the *gcd* between the columns' segments. Fortunately, because the maximum segment length is only the wordsize $m - 1$, we can precompute and memoize these values. In Table 5, we compare the total time to process 2000 queries over `HEP` between `wah`, `plwah`, and `wg-31-*-*`.

As can be seen, our approach is 20% and 31.8% slower than `wah` and `plwah`, respectively, in a head-to-head comparison using the `HEP` data set. However, as we showed in previous experiments, our framework's flexibility in allowing any segment length to be input can produce faster results if the initial i_u was properly selected.

Algorithm	Query Time (ms)
wah	5.67
plwah	5.11
wg-31-*-*	6.74

Table 5: Query Processing Overhead (HEP)

Our algorithm's $O(n^3)$ runtime is clearly affected by the column size n, but this is not the dominating cost. The significant time required to recompress columns is the primary concern. This factor is proportional to the number of columns recompressed into a different segment length, which is determined by our algorithm. In our experiments, we observed the optimization overheads shown in Table 6.

Data	Setting	Total Overhead (ms)
HEP	wg-7-0.0-100	1621.1
	wg-7-1.0-100	5047.0
	wg-9-0.0-100	697.2
	wg-9-1.0-100	3011.9
	wg-13-0.0-100	1327.4
	wg-13-1.0-100	3001.4
KDD99	wg-4-0.0-100	8189.0
	wg-4-1.0-100	9652.9
	wg-5-0.0-100	8536.3
	wg-5-1.0-100	9728.6
	wg-7-0.0-100	8318.1
	wg-7-1.0-100	9249.8
	wg-13-0.0-100	3728.3
	wg-13-1.0-100	3746.9

Table 6: Optimization Overhead

We show the overheads for $\tau = 100$, where the optimizer was invoked 20 times per run. As can be seen for both data sets, a lower value of λ tends to yield lower overheads because columns need not be recompressed often to suit queries. In general, it can also be seen that a larger segment length yields smaller overheads.

We show the results for $i_u = 13$ for `KDD99` to help exemplify this behavior. This may be due to two reasons: (1) less time is spent encoding when segment lengths are large and (2) there are less options (multiples) to use for encoding. For instance, given $i_u = 4$ the optimizer could recompress a column from 4 to 16. At the next interval, this column could again be recompressed to 20, 24, or 28, etc. In the case of $i_u = 13$, there are only two options 13 and 26. Therefore, the opportunities for recompression is reduced. Still, the overheads are significant compared to the total query times, which suggests that the optimizer should only be seldom executed. In the previous subsection, we showed that long optimization intervals did not yield significantly worse results. However, we will consider online optimization techniques in future work.

4.3 Limitations: Summary and Discussion

Based on our findings, we summarize the important insights that can be gleaned from our experimental results.

(1) The initial selection of i_u is crucial and produces nontrivial, data-dependent behavior. We showed that our results vary widely based on the selection of i_u. More interestingly, some i_u actually produced both greater

compression ratios *and* query efficiency. This result contradicts our assumption for constructing the WQ model. An informed, systematic way to select i_u is needed, and is a subject of our future work.

(2) The optimization interval τ should be dynamic. We showed that the optimization overhead can be quite high compared to the actual query processing time. However, once optimized, the remaining queries enjoy both a drastic speedup, and the index compression ratio likewise reduces. This suggests that, for *skewed-access* queries, the optimizer should be run once initially for the history to capture the skewed query pattern. This can be followed with less frequent, offline optimizations. Given a bursty or uniformly distributed query pattern, however, τ would have much more impact and should vary with respect to the characteristics of the accesses.

(3) Caching matters. Our query performance results, particularly for the large `KDD99` data set in Figure 4(b), showed the significant impact of caching. While the query history serves as a likely indicator of memory access patterns, our models do not account for cache performance, which breaks our original assumption that larger segment lengths (thus, larger indices) will always query faster. A cache-friendly optimization scheme is a major topic of our future work.

(4) Other limitations: Our proposed scheme is further limited as follows. If the query history is random (does not observe a skewed pattern), the optimization overhead will dominate. Fortunately, access skew can be expected in most realistic systems. Another limitation is that our models only capture two attributes queried at a time, and consider only exact-match queries (we did not experiment using range queries). Finally, our query sets are synthetic and small. To this end, we will employ real traces for future experiments. We will extend our proposed scheme to address these limitations in future works.

5. RELATED EFFORTS

In this section, we summarize several other bitmap compression schemes and the related area of workload-informed index selection.

Aside from the aforementioned WAH [9], and VLC [12], several other compression schemes have been designed for bitmap indices. Byte-aligned Bitmap Codes (BBC) [8] was the predecessor of WAH. It is a patented encoding that uses four types of byte-aligned atoms. It uses a 7 bit segmentation length encoding scheme. Wu, *et al.* [9] reports in an empirical study they found that WAH typically used 60% more space and that it could execute the logical operations of queries 12× faster than BBC.

The Enhanced WAH (EWAH) compression scheme is identical to WAH except in its encoding of fills [20]. In EWAH, a fill word is halved. The 17 most significant bits encode the flag bit, the fill value, and the run length. The lower 15 bits a represent the number of literal words following the run encoded in the fill. This information is used to optimize the query processing. For example, long runs of literal values can be ignored (not accessed) when ANDed with 0s.

COMPAX [11] is a compression scheme designed specifically for bitmaps representing network flow traffic. It compresses using a codebook of 4 types of words: 1) L, a word representing a literal of 31 bits, 2) 0F, encodes fills of consecutive runs of 31 zeros, 3) LFL, compactly represents a sequence of L-0F-L words where each of the three words consist of all 0s (excluding flag bits) except for a single byte which contains at least one 1, and 4) FLF, represents a sequence of 0F-L-0F words where, again, each word only consists of 1 "dirty byte". Essentially, the FLF and LFL words allow COMPAX to store three WAH words into one word. A study conducted by Fusco, *et al.* [11] showed that Compax could encode network flow bitmaps using 60% less space than WAH. However, their study was limited to bitmaps representing network flow traffic. It is not obvious that COMPAX would perform similarly on bitmaps representing information outside of that domain.

Recently, Deliége and Pedersen proposed the Position-List Word-Aligned Hybrid (PLWAH), which can achieve better compression than WAH by addressing the situations where single heterogeneous bit separates two runs [10]. These cases can occur quite often in sparse data sets. In their encoding of fill atom the two most significant bits are the fill and flag bits, it dedicates the following five bits to a position list. These bits are a binary representation of the heterogeneous bit's position in a "nearly identical" segment. In 32 bit word compression, a nearly identical segment is one in which all bits, except for one, have the same value as the preceding homogenous segment. Their experimental study showed that they were able to compress 50% better than WAH without a reduction of query processing efficiency. The authors also propose a fixed segment length of 7, 15 or 31, which is similar to VLC, but they apply the same length to the entire bitmap.

Our work is tangentially related to previous work in the areas of index selection and automatic tuning of databases. Notably, we propose a limited type of *feedback control loop* which monitors the online query activity and adjusts the compression of the bitmap index to achieve higher efficacy. The idea of a feedback control loop that monitor workloads and system metrics for database tuning has been well explored, *e.g.*, [24, 25, 26, 27, 28]. The works of Koudas [29] and Rotem *et al.* [30] are probably the most closely related work in this area. They both consider query workloads to try optimize the boundary selection of bins to minimize the cost of querying bitmap indices. However, to the best of our knowledge the work presented in this paper is the first to apply a workload based algorithms to optimize index segmentation length selection.

6. CONCLUSIONS AND FUTURE WORK

Bitmap indices pervade many applications due to their fast query processing and compressibility. However, as the amount data continues to grow rapidly in today's digital age, the scientific community is actively engaging to improve existing bitmap compression algorithms. In this paper, we proposed a novel bitmap optimization framework, which dynamically recompresses the index over time. We modeled query and compression ratios as a function of segment lengths and workload history. We defined a constrained optimization problem and showed that our algorithm can generate indices that significantly outperform both WAH and PLWAH over real data sets.

The results of our empirical study suggest that there are a few instances where our assumption that larger segments lengths always produces faster queries may not be true. In future works, we must refine and extend our models to further integrate cache consciousness. We will also investigate

the effects of having a dynamically varying optimization interval τ. Finally, in 64-bit architectures, there will be many unused bits in the fill word. We have already seen approaches, such as PLWAH, successfully take advantage of these bits to improve compression. Breaking a long word into variable PLWAH-based segments can prove beneficial and complementary.

7. REFERENCES

[1] M. J. Zaki and J. T. L. Wang, "Special issue on bionformatics and biological data management," in *Information Systems*, pp. "28:241–367", 2003.

[2] F. Donno and M. Litmaath, "Data management in wlcg and egee. worldwide lhc computing grid," Tech. Rep. CERN-IT-Note-2008-002, CERN, Geneva, Feb 2008.

[3] J. Harris, "Star collaboration 1994 nuclear physics a 566 277 285," 1994.

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

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

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

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

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

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

[10] F. Deliège and T. B. Pedersen, "Position list word aligned hybrid: optimizing space and performance for compressed bitmaps," in *In Proc. of EDBT*, pp. 228–239, 2010.

[11] F. Fusco, M. P. Stoecklin, and M. Vlachos, "Net-fli: on-the-fly compression, archiving and indexing of streaming network traffic," *VLDB'10*, vol. 3, pp. 1382–1393, Sept. 2010.

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

[13] D. Haynes, S. M. Corns, and G. K. Venayagamoorthy, "An exponential moving average algorithm," in *IEEE Congress on Evolutionary Computation*, pp. 1–8, 2012.

[14] S. Chitraganti, S. Aberkane, and C. Aubrun, "Statistical properties of exponentially weighted moving average algorithm for change detection," in *CDC*, pp. 574–578, 2012.

[15] K. Wu, E. Otoo, and A. Shoshani, "An efficient compression scheme for bitmap indices," in *ACM Transactions on Database Systems*, 2004.

[16] F. Doan, "Workload-Driven Bitmap Recompression for Real-Time Query Acceleration," Master's thesis, Washington State University, 2013.

[17] S. Rosset and A. Inger, "Kdd-cup 99: knowledge discovery in a charitable organization's donor database," *SIGKDD Explor. Newsl.*, vol. 1, pp. 85–90, Jan. 2000.

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

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

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

[21] E. Pourabbas, A. Shoshani, and K. Wu, "Minimizing index size by reordering rows and columns," in *SSDBM'12*, pp. 467–484, 2012.

[22] A. Dan, P. S. Yu, and J.-Y. Chung, "Characterization of database access skew in a transaction processing environment," in *Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS/PERFORMANCE'92, pp. 251–252, 1992.

[23] A. Dan, P. S. Yu, and J. yao Chung, "Characterization of database access pattern for analytic prediction of buffer hit probability," *VLDB Journal*, vol. 4, pp. 127–154, 1995.

[24] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback, "Self-tuning database technology and information services: from wishful thinking to viable engineering," in *VLDB*, pp. 20–31, 2002.

[25] K. P. Brown, M. Mehta, M. J. Carey, and M. Livny, "Towards automated performance tuning for complex workloads," in *VLDB*, pp. 72–84, 1994.

[26] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah, "Adaptive query processing: Technology in evolution," *IEEE DATA ENGINEERING BULLETIN*, vol. 23, no. 2, pp. 7–18, 2000.

[27] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil, "Leo - db2's learning optimizer," in *VLDB*, pp. 19–28, 2001.

[28] A. Mehta, C. Gupta, S. Wang, U. Dayal, *et al.*, "Automated workload management for enterprise data warehouses," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, no. 1, pp. 11–19, 2008.

[29] N. Koudas, "Space efficient bitmap indexing," in *Proceedings of the 9th international conference on Information and knowledge management*, CIKM '00, (New York, NY, USA), pp. 194–201, ACM, 2000.

[30] D. Rotem, K. Stockinger, and K. Wu, "Optimizing candidate check costs for bitmap indices," in *Proceedings of the 14th ACM international conference on Information and knowledge management*, pp. 648–655, 2005.