Summer 2016

Distributed indexing and scalable query processing for interactive big data explorations

Gheorghi Guzun
University of Iowa

DISTRIBUTED INDEXING AND SCALABLE QUERY PROCESSING FOR INTERACTIVE BIG DATA EXPLORATIONS

by

Gheorghi Guzun

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Electrical and Computer Engineering
in the Graduate College of
The University of Iowa

August 2016

Thesis Supervisor: Assistant Professor Guadalupe M. Canahuate

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Gheorghi Guzun

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Electrical and Computer Engineering at the August 2016 graduation.

Thesis Committee: _____
Guadalupe M. Canahuate, Thesis Supervisor

_____
Er-Wei Bai

_____
Jon G. Kuhl

_____
Mona K. Garvin

_____
Ricardo Mantilla

To Sebastian

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Guadalupe Canahuate who supported and guided me throughout my PhD. She is an extraordinary researcher, and a wellspring of ideas. Her invaluable input always helped me to finish the next task. Without her guidance, this dissertation would have not been brought to fruition.

I would also like to extend my gratitude to Professor Jon Kuhl, with whom I was lucky to work alongside, and whose courses I attended with great enthusiasm. His passion for teaching, work ethic, and challenging questions take Education to the next level. I want to acknowledge the other members of my dissertation committee, and thank them for their feedback after my first presentation of the research proposal: Dr. Ricardo Mantilla, Dr. Er-Wei Bai and Dr. Mona Garvin.

Many parts of this dissertation are a result of collaboration with other people. My first project on bitmap compression was a joint work with David Chiu, Jason Sawin and my advisor Guadalupe Canahuate. Joel Tosado worked alongside with me on various projects while developing the distributed bit-vector indexing and query systems, such as K-NN queries, and query approximation through slice shedding.

I would like to thank Cathy Kern and Dina Blanc who always went out of their way to help me with things within and outside the ECE Department.

Last but not least, I am thankful my family and friends. To my lovely wife

Africa who always supports and believes in me. To my parents who raised me, taught me the good values, and are always there for me.

ABSTRACT

The past few years have brought a major surge in the volumes of collected data. More and more enterprises and research institutions find tremendous value in data analysis and exploration. Big Data analytics is used for improving customer experience, perform complex weather data integration and model prediction, as well as personalized medicine and many other services.

Advances in technology, along with high interest in big data, can only increase the demand on data collection and mining in the years to come. As a result, and in order to keep up with the data volumes, data processing has become increasingly distributed. However, most of the distributed processing for large data is done by batch processing and interactive exploration is hardly an option. To efficiently support queries over large amounts of data, appropriate indexing mechanisms must be in place.

This dissertation proposes an indexing and query processing framework that can run on top of a distributed computing engine, to support fast, interactive data explorations in data warehouses. Our data processing layer is built around bit-vector based indices. This type of indexing features fast bit-wise operations and scales up well for high dimensional data. Additionally, compression can be applied to reduce the index size, and thus utilize less memory and network communication.

Our work can be divided into two areas: index compression and query pro-

cessing. Two compression schemes are proposed for sparse and dense bit-vectors. The design of these encoding methods is hardware-driven, and the query processing is optimized for the available computing hardware. Query algorithms are proposed for selection, aggregation, and other specialized queries. The query processing is supported on single machines, as well as computer clusters.

PUBLIC ABSTRACT

The past few years have brought a major surge in the volumes of collected data. More and more enterprises and research institutions find tremendous value in data analysis and exploration. Big Data analytics is used for improving customer experience, perform complex weather data integration and model prediction, as well as personalized medicine and many other services.

Advances in technology, along with high interest in big data, can only increase the demand on data collection and mining in the years to come. As a result, data processing has become increasingly distributed, in order to keep up with the data volumes. However, most of the distributed processing for large data is done by batch processing of text data. This means that the answers are often available only after some time when inquiring the data. Interactive exploration is hardly an option over big data, and that limits the value of the information. To efficiently support queries over large amounts of data, appropriate indexing mechanisms must be pre-computed.

This dissertation proposes an indexing and query processing framework, that can run on top of a distributed computing engine, to support fast, interactive data explorations in data warehouses. Our data processing layer is built around a bit-vector based index that enables for fast bit-wise operations and compression to reduce memory and network bandwidth utilization.

We achieve these results by developing two compression schemes for sparse

and dense bit-vectors. The design of these encoding methods is hardware-driven, and the query processing is optimized for the available computing hardware. We propose query algorithms for selection and aggregation and other specialized queries. The query processing is supported on single machines, as well as on map-reduce systems.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

LIST OF ALGORITHMS

Algorithm

CHAPTER 1
INTRODUCTION

1.1 Motivation for scalable indexing in distributed settings

The past few years have seen an explosion in stored data volumes and have produced a major change in how people and computing systems interact with data. Today, research institutions, enterprises, as well as small organizations collect very large amounts of data. They all do so with the goal of extracting valuable information for their businesses. The insights derived from the data are then used in the decision making process for solving a wide range of problems. To better understand customers and their behaviors and preferences, companies are keen to expand their traditional datasets with social media data, browser logs, as well as text analytics and sensor data to get a more complete picture of their customers [1](e.g. streaming content providers and online retail stores). In healthcare, with the advent of high-throughput genomics, life scientists are starting to encounter challenges in handling, processing and moving massive datasets, that were once the domain of astronomers and high-energy physicists [2]. Terabyte-sized datasets are now common in earth and space sciences, physics, finance, and security and law enforcement [3].

Given the large sizes of these datasets and the stalling processor speeds, increasingly more applications evolve towards distributed systems and organizations have to scale out to computer clusters. As a result, several new program-

ming models and data storage frameworks for cluster environments have been proposed. Google's MapReduce [4] presented a simple and general model for batch processing, that also handles faults. The MapReduce model gained popularity as it enabled computations over large datasets without requiring expensive supercomputers. Hadoop MapReduce [5] is the open source implementation of MapReduce, it has been used extensively by the open source big data community. Additionally, a number of specialized tools that run on top of MapReduce and its file system [6] have been proposed [7]–[12]. However, in the MapReduce world today, a great deal of distributed processing is still done by batch processing of flat files.

1.2 Indexing requirements

To get the "value" that organizations look for in their collections of data, there is often need for interactive data explorations and insight gathering. This means that the queries should be answered at near real-time rates. Centralized indices fall short when faced with the volume and dimensionality of modern data. Hence, appropriate data indexing techniques must be integrated with the existing distributed computing infrastructures.

In this work we argue that the bit-vector based indices can be applied in distributed environments and improve query execution times for a wide range of queries. Bit-vectors based indices, such as the Bitmap index [13], [14] and the Bit-sliced index (BSI) [15], have been used successfully and extensively in the context

of data warehouses and scientific databases. The main advantages of such indices compared to others are:

- Partitionable - Most conventional indices have been designed for centralized environments, and become inefficient when they do not fit into main memory, as it is often the case with large datasets. The bit-vector index is a binary representation of the data, and indexes each dimension independently. Thus can be easily partitioned horizontally, as well as vertically. By controlling the granularity of the index partitions it is easier to find a good trade-off between parallelism and network communication.

- Scalable for high dimensionality - Big Data is often high dimensional data, and hierarchical indices suffer from the curse of dimensionality [16]. This is, they become slower than sequential scan after a number of dimensions. Bit-vector based indices have proved to scale well for high dimensions [17].

- Fast read access - The vast majority of big data analysis exploration applications are read mostly [18]. Once the data is written, it is rarely updated. However new inserts are frequent. Fast read accesses facilitate faster query times and provide the user easier access to useful information.

- Compressible - It is desirable that the index does not exceed the actual raw data size. The smaller the index, the fewer disk accesses, less memory utilization and network communication are required. Ultimately this translates into faster query responses. The bit-vector indices are compressible. The

most popular bitmap compression techniques are based on run-length compression. Also, it is possible to control the level of data quantization, which is a lossy type of compression.

- Fast bitwise operations - The main advantage of the bit-vector based indices is that the bitwise operations are executed at the hardware level and are executed at clock cycle speeds. More sophisticated queries can be built from strings of bit-level operations such as AND, OR, XOR and NOT.

1.3 Summary description of the proposed system

In order to adapt bit-vector based indices to a distributed environment and in particular using the MapReduce framework, our work focuses on two complementary areas: index compression and query processing.

For index compression we propose two compression schemes for sparse and dense bit-vectors. For sparse vectors, we combine existing compression techniques into a single framework and allow different methods to coexist. For dense bit-vectors we propose a hybrid compression scheme that allows for verbatim (or uncompressed) vector to be combined with compressed vectors and the results are compressed or not depending on the benefits for query execution time. The design of these encoding methods is hardware-driven, and the query optimizer accounts for the available computing hardware.

For query processing, we first extend the types of queries supported by bit-vector based indices such as preference and top-k, K-Nearest Neighbors, and

collaborative filtering. Because aggregation across all dimensions dominates the query time in these applications when using the BSI, we propose distributed algorithms to support selection, aggregation, and other specialized queries using the MapReduce paradigm. Figure 1.1 shows the software stack being used for implementing the distributed query processing over the bit-vector index. The distributed BSI arithmetic layer represents the proposed work - the BSI query engine, and currently uses Apache Spark [19], [20] for distributing its tasks.

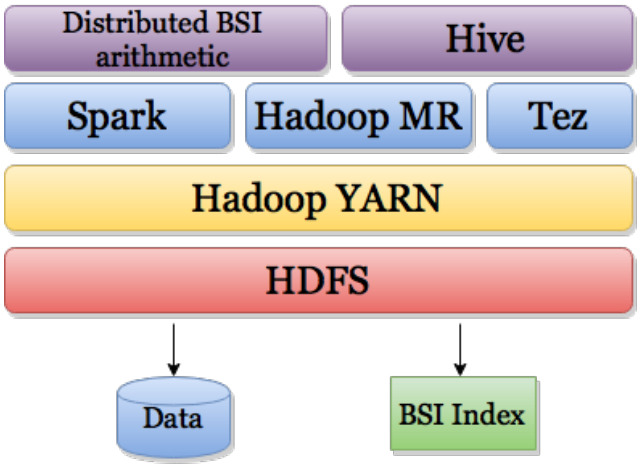


Figure 1.1: Overview of the software stack used in this work.

To prove the efficacy of the proposed approach, we implemented the indexing and the query logic necessary for performing complex queries such as: aggregations, preference queries, K-Nearest Neighbors(K-NN) and collaborative filtering on an Apache Spark cluster. Preliminary results show that the proposed approach outperforms Hive, a map-reduce based data warehouse, over Hadoop

Map-Reduce and the optimized query engine Tez, by at least an order of magnitude. It is also 25 times faster than SparkSQL, which uses the same distributed computation engine as the proposed index.

1.4 Dissertation plan

The rest of this dissertation is organized as follows. Chapter 2 describes the background and related work relevant to this dissertation. Chapter 3 presents our work on bitmap index compression. Chapter 4 presents the query algorithms for complex queries such as preference queries, and describes a query optimization method through hybrid compression for dense bit-vectors. Chapter 5 presents a distributed architecture of the distributed bit-sliced index. Further, chapter 6 describes our methods for answering top-K and K-NN queries using the distributed bit-sliced index. Additionally, we optimize result aggregations and provide cost estimations, and means for tuning the partition sizes and the level of parallelism. Chapter 7 summarizes and concludes the thesis.

CHAPTER 2
BACKGROUND AND RELATED WORK

Bit-vector based indices, also known as bitmap indices, are popular indexing techniques for enabling fast query execution over large scale datasets. Because bit-vector based indices can leverage fast bitwise operations supported by hardware, they have been extensively used for selection and aggregation queries in data warehouses and scientific applications.

2.1 Bitmap index

A bitmap index is used to represent a property or attribute value-range. For a simple bitmap encoding, each bit in the bitmap vector corresponds to an object or record in a table and bit positions are set only for the objects that satisfy the bitmap property. For categorical attributes, one bitmap vector is created for each attribute value. Continuous attributes are discretized into a set of ranges (or bins) and bitmaps are generated for each bin.

For example, consider a bitmap index for a relation of different objects in a spatial grid. Let us consider three attributes for this table: the object type and the X and Y coordinates where the object resides. Attribute *Type* is a categorical attribute and one bitmap vector is created for each object type. Attributes X and Y are continuous attributes. The desired resolution for the grid defines how many bins need to be created for X and Y. A grid with resolution 100×100, would require 100 bitmap vectors for attribute X and 100 bitmap vectors for attribute Y. A

query asking for objects located within the grid cell identified by x_i and y_j, can be answered by ANDing the corresponding bitmap vectors together. Any set bit in the resulting vector indicates an object located within the cell.

If the number of objects is fixed, increasing the resolution of the grid in the running example will increase the number of bitmap vectors producing a larger index but it will also increase the sparsity of the bit-vectors created. For this reason, bitmap indices are often compressed using *run-length* based encoding. A *run* refers to a set of consecutive bits with the same value, i.e., all 0s or all 1s. Sparse bitmap will have long runs of zeros.

The topic of bitmap indices was introduced in [15]. Several bitmap encoding schemes have been developed, such as equality [15], range [21], interval [21], and workload and attribute distribution oriented [22]. Several commercial database management systems use bitmaps.

	Raw Data		Equality Bitmaps						Bit-Sliced Index (BSI)				BSI SUM		
			Attrib 1			Attrib 2			Attrib 1		Attrib 2				
Tuple	Attrib 1	Attrib 2	=1	=2	=3	=1	=2	=3	$B_1[1]$	$B_1[0]$	$B_2[1]$	$B_2[0]$	$S[2]^3$	$S[1]^2$	$S[0]^1$
t_1	1	3	1	0	0	0	0	1	0	1	1	1	1	0	0
t_2	2	1	0	1	0	1	0	0	1	0	1	0	0	1	1
t_3	1	1	1	0	0	1	0	0	0	1	0	1	0	1	0
t_4	3	3	0	0	1	0	0	1	1	1	1	1	1	1	0
t_5	2	2	0	1	0	0	1	0	0	1	1	0	1	0	0
t_6	3	1	0	0	1	1	0	0	1	1	0	1	1	0	0

[1] $S[0]$=$B_1[0]$ XOR $B_2[0]$, C_0 = $B_1[0]$ AND $B_2[0]$
[2] $S[1]$=$B_1[1]$ XOR $B_2[1]$ XOR (C_0)
[3] $S[2]$=C_1=$Majority(B_1[1],B_2[1],(C_0))$

Figure 2.1: Simple example of equality encoded bitmaps and bit-sliced indexing for a table with two attributes and three values per attribute.

Figure 2.1 shows an example for a dataset with two attributes each one with 3 distinct values (cardinality=3). For the equality encoded bitmaps, one bitmap is

generated for each attribute value and the bit is set if the tuple has that value.

2.2 Bit-sliced index

BSI (Bit-Sliced Index) [15], [23] can be considered a special case of the encoded bitmaps [24]. With bit-sliced indexing, binary vectors are used to encode the binary representation of the attribute value. One BSI is created for each attribute.

For the BSIs in the example (Figure 2.1), since each attribute has three possible values, the number of bit-slices for each BSI is $\lceil log_2 3 = 2 \rceil$. The first tuple t_1 has the value 1 for attribute 1, therefore only the bit-slice corresponding to the least significant bit, $B_1[0]$ is set. For attribute 2, since the value is 3, the bit is set in both BSIs. The SUM for the two BSIs is also shown in the figure. In this case, the maximum value of the sum is 6 and the number of bit-slices is $\lceil log_2 6 = 3 \rceil$. The addition of the BSIs representing the two attributes is done using efficient bit-wise operations. First, the bit-slice $sum[0]$ is obtained by XORing $B_1[0]$ and $B_2[0]$ i.e. $sum[0] = B_1[0] \oplus B_2[0]$. Then $sum[1]$ is obtained in the following way $sum[1] = B_1[1] \oplus B_2[1] \oplus (B_1[0] \wedge B_2[0])$. Finally $sum[2] = Majority(B_1[1], B_2[1], (B_1[0] \wedge B_2[0]))$. BSI arithmetic for a number of operations is defined in [23]. BSI is the most compact representation of an attribute as only $\lceil \log_2 values \rceil$ vectors are needed to represent all values. However, their high density makes them hard to compress any further.

2.3 Compression of bit-vectors

Bit-vector indices are typically compressed using specialized run-length encoding schemes that allow queries to be executed without requiring explicit decompression. Byte-aligned Bitmap Code (BBC) [25] was one of the first compression techniques designed for bitmap indices using 8-bits as the group length and four different types of words. BBC compresses the bitmaps compactly but query processing is CPU intensive. Word Aligned Hybrid (WAH) [26] proposes the use of words instead of bytes to match the computer architecture and make access to the bitmaps more CPU-friendly ($w = 32 or 64$ bits). WAH divides the bitmap into groups of length $w-1$ and collapse consecutive all-zeros/all-ones groups into a fill word.

2.3.1 Word-Aligned Hybrid run-length code (WAH)

Let w denote the number of bits in a word. Assuming the most significant bit indicates the type of word we are dealing with, the lower $(w-1)$ bits of a literal word contain the bit values from the bitmap. If the word is a fill, then the second most significant bit is the fill bit, and the remaining $(w-2)$ bits store the fill length. WAH imposes the word-alignment requirement on the fills. This requirement is key to ensure that logical operations only access words.

Figure 2.2 shows a WAH bit vector representing 128 bits. In this example, we assume 32 bit words. Under this assumption, each literal word stores 31 bits from the bitmap, and each fill word represents a multiple of 31 bits. The second

128 bits	1,20*0,4*1,78*0,30*1			
31-bit groups	1,20*0,4*1,6*0	62*0	10*0,21*1	9*1
groups in hex	400003C0	00000000 00000000	001FFFFF	000001FF
WAH (hex)	400003C0	80000002	001FFFFF	000001FF

Figure 2.2: A WAH bit vector.

line in Figure 2.2 shows how the bitmap is divided into 31-bit groups, and the third line shows the hexadecimal representation of the groups.

Recently, several bitmap compression techniques that improve WAH by making better use of the fill word bits have been proposed in the literature [27]–[32].

2.3.2 Word-aligned Bitmap Code (WBC)

WBC or Enhanced WAH (EWAH) [27], [33], divides the bitmap into groups of w-bits, not $w-1$-bits. EWAH uses two types of words: marker words and literal words. Half of a marker word is used to encode the fills. The upper half (most significant bits) of the fill word encode the fill value, and the run length and the remaining bits are used to represent the number of literal words following the run encoded in the fill. This implicit word type eliminates the need for a flag bit for each word to identify the type of word.

Verbatim Bitmap (in hex)		400003C0	00000000 00000000 00000000	001FFFF0	000001FF
EWAH Bitmap (in hex)	0000 0001	400003C0	0003 0002	001FFFF0	000001FF

Figure 2.3: A verbatim bitmap and its EWAH encoding.

Figure 2.3 shows a verbatim bitmap and its EWAH encoding. The verbatim bitmap has 182 bits (6 32-bit words). The EWAH bit-vector always starts with a marker-word. The first half of a marker-word represents the header. For a 32-bit word length as in this case, the first 16 bits indicates the type and number of fill words. The second half of the marker-word, tells the number of literals that follow a marker-word (1 in the example). After the literal word comes another marker-word. The first bit (0) indicates a run of zeros and the value 3 is the number of words in the run. The second half of the marker word indicates that there are two literals following the fill.

2.4 Bit-vector based indices used in distributed environments

Bitmap-based indices have been used in distributed environments as filters for matching data and in the form of Bloom filters [34] to filter out entire partitions that do not contain the data of interest for a given query. A Bloom filter is a bit-array structure that allows testing whether a given value exists in a set, with controlled false positive rate. For example, the authors in [35] exploit Bloom filters to reduce the data communication costs in joins for hybrid warehouses. They do so by using Bloom filters to ensure that only the records that will participate in the join need to be transferred through the network.

A Bitmap Index for Database Service (BIDS) [36] have been proposed for indexing large scale data-stores. BIDS indexes the data using Equality Bitmaps and Bit-Sliced indices. Depending on the attribute cardinality of the data it decides

whether to use equality encoded bitmaps, bit-sliced index, or not to create an index for an attribute. The created index, either equality encoded bitmaps or bit-sliced index, is then compressed using WAH. However, the bit-vector indices are only used for point and range queries, and often retrieve only partial results. For more complex queries, it is necessary to employ an additional index such as Trojan [37]. BIDS also compresses its bit-vector indices using WAH, regardless of the density of the bit-vector, and as we show in [38], for high density bit-vectors, compression should be applied only in certain cases, otherwise it imposes an overhead.

For high dimensional data, where there is need for dimensionality reduction, Locality-sensitive Hashing (LSH) [39] is often used. LSH algorithms use data depending hashing so that similar items fall into the same "bucket". This way LSH approaches are extensively used in solving the K-Nearest Neighbors problem. We use bit-vector based indices to answer KNN queries.

Another line of work exploits intra-cycle CPU parallelism available at the bit level in modern processors. BitWeaving [40] focuses on running in-memory data scans and operates on multiple bits of data in a single cycle, processing bits from different columns in each cycle. Although an interesting concept, the index is limited to in-memory databases and does not scale across multiple machines.

In the next two chapters we present our contributions and proposed work for improved index compression and distributed query processing. Our compression can be integrated with most of the related work cited in this section and the query processing algorithms are integrated within the Hadoop software stack.

CHAPTER 3
BIT-VECTOR COMPRESSION

An effective mechanism for speeding-up queries is to build indices on the queried attributes. As many of the conventional indexing mechanisms fail to scale well, we opt for an index that is fast and suitable for parallelization. We pick bit-vector indexing as our indexing of choice as it features fast low-level bit-wise operations. Specifically, we employ the use of bitmap and bit-sliced indices.

As defined earlier, the bitmap indices are binary structures that identify the rows that falls into a certain criteria. For each category, the bitmap index generates a bit-vector, thus for high cardinality datasets the bitmap index can be very large, albeit rather sparse. To tackle this, we first analyze the current most popular bitmap compression schemes and design a system that selects the most appropriate encoding method, depending on the dataset. We then take on improving the existing word-aligned bitmap compressions by adding an additional segment length that changes based on the bit-density. For higher bit-densities, a shorter segment results in better compression. We call this compression method Variable Aligned Length (VAL), and it is described further in Section 3.2

To support a larger array of queries, we integrate the Bit-Sliced index (BSI) within our index. The BSI index stores a set of "bit-slices". A bit-slice is a bit-vector that contains one bit defined by its depth from each tuple of an attribute. Unlike the bitmap index, the BSI index is denser, and its size is generally smaller than the size of the raw data, even without applying compression. However we find that, in

certain situations, compression can be applied, and can improve query execution times. Thus we propose a query optimization method through compression of the BSI slices, as described in Section 3.3.

We have evaluated several word-aligned bitmap compression techniques in terms of compression ratio but more importantly query time for point and range queries. As expected, no encoding is better than all others in all cases. However, our results evidence that there are specific scenarios in which one method should be preferred over the others. Depending on one's needs, the preference could be towards a more compact bitmap index or faster queries. In this study we provide insights as to which encoding should be preferred depending on the dataset. For example, EWAH could be preferred for bitmaps that are hard-to-compress, that have short fills and mostly literals; PLWAH could be preferred for very sparse bitmaps where it compresses significantly better than WAH and may have faster query times.

Furthermore, we formalize a method for estimating relative query performance between two encodings before actually compressing the bitmaps. This method only requires a single traversal of the bitmaps for providing these estimations, and has proven to be accurate.

3.1 Compression and query performance analysis for word-aligned bitmap compression methods

Having already described the basic principles of the WAH bitmap encoding in chapter 2, we proceed with analyzing the probable bitmap sizes and query algorithm complexity of WAH.

WAH Compressed Bitmaps

Let d be the bit density of a uniform random bitmap. The probability of finding an uninterrupted group that is a 1-fill, i.e., $2w-2$ consecutive bits that are one, is d^{2w-2}. Where w is the word length. Similarly, the probability of finding a counting group that is a 0-fill is $(1-d)^{2w-2}$. Thus, the expected size of a WAH compressed bit-vector containing N bits is:

$$m_{\mathrm{WAH}}(d) = \frac{N}{w-1}(1-(1-d)^{2w-2}-d^{2w-2}).$$

For an attribute following a probability distribution, where the i^{th} value has a probability of p_i, the total size of the bitmap index compressed using WAH is:

$$s_N = \sum_{i=1}^{c} m_R(p_i) \approx \frac{N}{w-1}\Big(c-\sum_{i=1}^{c}(1-p_i)^{2w-2}-\sum_{i=1}^{c}p_i^{2w-2}\Big),$$

where N is the number of objects or bits in the bitmap, m_R is the expected number of words in a random bitmap, w is the word size, and c is the cardinality of the attribute [26].

WAH-encoded bitmaps never require more than $4N$ words. If $c < 0.01N$, then the maximum size of the WAH compressed bitmap for the attribute is about $2N$ words. A clustering factor f is computed as the average number of bits in the 1-fill runs. The maximum size of the WAH compressed bitmap for a clustering factor $f > 1$ and $c < 0.1N$ is:

$$s \approx \frac{N}{w-1}\left(1 + \frac{2w-3}{f}\right),$$

which is nearly inversely proportional to the clustering factor f.

A more detailed reasoning and proofs for the above formulas can be found in [26].

Let us denote by m_x and m_y, the number of words in the WAH compressed bitmap vectors X and Y, respectively. M is the number of words in the uncompressed bitmap ($M = \lceil \frac{N}{w-1} \rceil + 1$). In the generic algorithm 1, the main loop is executed $max(m_x, m_y) \leq I_w \leq min(m_x + m_y - 1, M)$ times. The *decode* method is called once per word: $m_x + m_y$ times. The *addLiteral* method is called when two literal words or a fill and a literal are operated together: αI_w, where α is the fraction of iterations that generate literals. The *addFill* method is called when two fills are operated together: $(1 - \alpha)I_w$ times.

Algorithm 1: General Bit-wise Logical Operation.

Input: bit-vector x, y - Input bit-vectors
Output: bit-vector z - the resulting compressed bit-vector after performing the logical operation $x \circ y$

```
1  bitVector z=new bitVector(M);
2  while x.vec and y.vec are not exhausted do
3      if x.activeWord is exhausted and there are more words in x then
4          x.decodeWord();
5      end
6      if y.activeWord is exhausted and there are more words in y then
7          y.decodeWord();
8      end
9      while x.activeWords and y.activeWords are not exhausted do
10         if x.activeWord.nSegments == 0 then
11             x.activeWord = x.nextWord();
12         end
13         if y.activeWord.nSegments == 0 then
14             y.activeWord =y.nextWord() ;
15         end
16         if x.activeWord.isFill and y.activeWord.isFill then
17             nSegments=Min(x.activeWord.nSegments, y.activeWord.nSegments);
18             if x.activeWord.fill ∘ y.activeWord.fill == 0 then
19                 z.vec.add(FillOfZeros + nSegments);
20             end
21             else
22                 z.vec.add(FillOfOnes + nSegments);
23             end
24             x.activeWord.nSegments-=nSegments;
25             y.activeWord.nSegments-=nSegments;
26         end
27         else
28             z.vec.add(x.activeWord.getLitValue() ∘ y.activeWord.getLitValue());
29             //GetLitValue decreases nSegments by 1
30         end
31     end
32 end
33 return z;
```

The size of the resulting bitmap is $m_z < min(m_x + m_y, M)$. This space is allocated before entering the loop to avoid dynamic memory allocation. Let us assume that the time to allocate m_z is $C_a(m_x + m_y)$. C_1 is the time to decode a word, C_l the time to execute appendLiteral, C_f the time to execute appendFill, and

C_0 is the loop overhead. The time to execute a query using Algorithm 1 is:

$$t = C_a(m_x + m_y) + C_1(m_x + m_y) + C_l\alpha I_w + C_f(1-\alpha)I_w + C_0 I_w \tag{3.1}$$

Because I_w is always smaller than $m_x + m_y$, it is easy to see from Equation 3.1 that Algorithm 1 has a complexity of $\mathcal{O}(m_x+m_y)$, which means that the query time is directly proportional with the size of the bitmap index. In order to estimate the query time however, this information is insufficient. A more precise approximation of I_w is required. Further in the next sections we show that I_w can be approximated with high accuracy, and so can the query time.

PLWAH/CONCISE Compressed Bitmaps

PLWAH and CONCISE exploit the fact that not all the bits in a word are used to store the run-length counter when compressing a fill word. Often runs are interrupted by a single set/unset bit, thus they use some of the bits in the fill word to store a "position list" of the set/unset bit. This enables a better compression when compared to WAH.

Let s be the maximum number of heterogeneous bits a fill word can store in its list of positions. The size of the list is $s\log_2(w)$ and the size of the counter is $w - 2 - s\log_2(w)$. A single fill word can thus represent up to $(2^{w-2-s\log_2(w)})$ groups of length $(w-1)$. All compression schemes have an overhead when representing incompressible bitmaps. For WAH and PLWAH, this overhead is one bit per word, so the compression ratio is $w/(w-1)$. A bitmap containing no homogeneous groups

will not have any fill words and will be incompressible. The upper and lower bounds of the PLWAH compression ratio are respectively: $(2^{w-2-s\log_2(w)}+1)(w-1)/w$ and $(w-1)/w$. As long as the upper bound is not reached, the worst PLWAH compression ratio is bounded by the WAH compression ratio. A more detailed description of PLWAH compressed bitmaps is provided in [28].

We estimate that the number of words in the PLWAH compressed bit-vector containing N bits, and a density d is:

$$m_{\mathrm{PLWAH}}(d) = \frac{N}{w-1}(1-(1-d)^{2w-2}-d^{2w-2}) - S_{\mathrm{PLWAH}},$$

where S_{PLWAH} is the number of sparse words that can be encoded within the PLWAH fills. The statistics about this variable and the other used as inputs for size estimations have to be collected when scanning the data.

PLWAH uses the same generic Algorithm 1 as WAH for performing bitwise logical operations. Thus a big-O analysis of WAH querying algorithms and PLWAH querying algorithms would lead us to equivalent upper bounds. Thus, performing performance measurements for each of the constants in Equation 3.1 could be a better way to differentiate between the two encodings. In the next section, we perform an exercise of measuring the performance for the constants defined in Equation 3.1, and then show that it is possible to predict the relative performance between PLWAH/CONCISE and WAH for different datasets.

The WAH and EWAH bitmap encodings have often a similar compression

ratio, however for some data distributions the difference in compressed size can be significant between WAH and EWAH. This is also reflected in the EWAH query time.

EWAH Compressed Bitmaps

In general, WAH and WBC (EWAH) encoding schemes are fairly similar. The difference between WBC (EWAH) and WAH is that the first one uses only half of the fill word to encode the 0-fill or the 1-fill, while the second half is used to mark the number of literals following the fill word. Because of this, EWAH may need more than one word to encode a fill, if the bitmaps are very sparse. Thus, when estimating EWAH compressed size of a bitmap, one should consider counting the number of run-lengths longer than EWAH can encode, and adjust the WAH size estimation equations for EWAH. Thereby, the expected size of a EWAH compressed bit-vector containing N bits with density d is:

$$m_R(d) \approx \frac{N}{w}(1 - (1-d)^{2w} - d^{2w}) + R_{ex},$$

where R_{ex} is the number of runs that have a length greater than $(w \times 2^{\frac{w}{2}}) - 1$.

WBC/EWAH compression scheme was designed with the goal of avoiding decoding literal words at query execution time. This reduces the number of memory accesses and can result in faster queries when compared to WAH. However at the same time, when compressing the resulting bit-vector EWAH has to increment the last counter of the following literal words. Thus every time EWAH performs a

literal append, it has to access the last marker-word to increment its literal counter, and also decrement the counters of the bitmap columns queried.

3.1.1 Performance estimations

As discussed in the previous sections, there is no best bitmap compression technique that can outperform the rest, for all types of data in both, query time and index size. Furthermore, since the query algorithms for all analysed techniques have the same upper bound complexity, it is difficult to estimate which one will perform better just by considering index sizes and the query big-O analysis.

In this section we estimate query times before actually running the queries for datasets compressed with different bitmap encoding schemes. Generally we want to estimate query times before even compressing a dataset. Estimating both the size of the compressed data and the query time can help deciding in choosing the most suitable bitmap index compression technique for the data.

Our goal is to formalise a method to estimate the performance of these different encoding techniques in order to select the best bitmap compression encoding for various datasets. As users have different preferences, "best" could have different meanings, thus it is important to estimate both, the compression size and expected query time. Then it is at the user's discretion to choose between better compression ratio or faster query time. We show that it is possible to estimate within a confidence level of 95% the relative performance between four word-aligned based bitmap compression schemes: WAH, EWAH and PLWAH/CONCISE.

Table 3.1: Measured Constants for query execution.

	WAH	EWAH	PLWAH/ CONCISE
Memory alloc(C_a)	x	x	x
Decode Literal(D_l)	10x	10x	16x
Decode Fill (D_f)	20x	20x	22.5x
Append Literal (C_l)	5x	7.5x	7.5x
Append Fill (C_f)	5x	5x	5x

First we compute the size of compressed bitmaps using the procedures from the previous section. As discussed in previous sections, the query time is proportional to the size of the two compressed bitmaps being queried, however the complexity of decoding and appending has a big impact too. Algorithm 1 traverses through bit-vector x and bit-vector y until both of them are exhausted. Being a generic algorithm, Algorithm 1 is applicable for all, WAH, EWAH and PLWAH. Thus Equation 3.1 is valid for these compression schemes as well. C_a, C_1, C_l and C_f are all constants and can be measured empirically. The decoding of a literal word requires less instructions than decoding a fill word. Thus, we split the decoding cost, C_1, into two components: D_l for decoding a literal word, and D_f for decoding a fill word. Therefore, the general query time for Algorithm 1 becomes:

$$t_G = C_a(m_x + m_y) + D_l(m_{x_l} + m_{y_l}) + D_f(m_{x_f} + m_{y_f}) + C_l\alpha I_w + C_f(1_\alpha)I_w + C_0 I_w, \quad (3.2)$$

where m_{x_l} and m_{y_l} represent the number of literals contained in the compressed bit-vector x and y respectively. Similarly, m_{x_f} and m_{yf} are the number of fills in x and y.

Table 3.1 shows measured values of the constants from Equation 3.2 for WAH, PLWAH/CONCISE and EWAH. The constants are measured in time units that are relative to the machine the queries are run on. The relative differences between them are expected to be preserved on a faster/slower computer. From Table 3.1 we can observe that C_l and C_f are not equal, and thus the number of fill appends and literal appends have to be estimated.

Another unknown remains the number of times Algorithm 1 executes the main loop, which is equal to the total number of appends. The number of iterations for the main loop is completely dependent on the sizes of the queried bit-vectors. More specifically, it depends on the size of the largest compressed bit-vector and the size ratio between the two bit-vectors. For two independent and highly compressed bit-vectors, there is small probability that the fill words of one are aligned with the other bit-vector. Thus Algorithm 1 requires two iterations to exhaust one word from that largest bit-vector. On the other hand, for a bit-vector that is not compressed at all, only one iteration is sufficient to consume one word from the largest bit-vector. At the same time, the size ratio between the two bit-vectors being queried determines the final number of iterations. If the better compressed bit-vector is significantly smaller than the other bit-vector, then the probability for two fill words being aligned is higher than when the two bit-vectors are relatively equal. Equation 3.3 gives the formula for estimating the total number of iterations

in the main loop of Algorithm 1:

$$I_w \approx [(1 - \max(CR_x, CR_y)) \times \frac{\min(m_x, m_y)}{\max(m_x, m_y)} + 1] \times \max(m_x, m_y), \tag{3.3}$$

where CR_x and CR_y are the compression ratios for bit-vectors x and y respectively:

$$CR_x = size(x)_{compressed}/size(x)_{uncompressed}$$

Every iteration of Algorithm 1 results in either a fill append or a literal append. Thus the total number of appends is equal to the number of iterations.

We estimate the number of fill appends (I_f) based on the probability of two fills being aligned in the compressed bit-vectors that are queried. The remaining appends are appends literals (I_l). The probabilities for estimating the number of fill appends differ based on the compression scheme used. In the next three subsections we estimate the query time performance for WAH, EWAH, and PLWAH/-CONCISE respectively.

Performance Estimations for WAH

For WAH, the total number of fill appends is the probability of two fill words being operated at the same time in Algorithm 1, multiplied by the number of iterations in Algorithm 1. The probability of operating two fill words within the same iteration, is at least the product of the probabilities of encountering a fill

word in each of the operated bit-vectors:

$$P_{\text{appendFill}} > \frac{\text{Fills}_x}{m_x} \times \frac{\text{Fills}_y}{m_y}.$$

In the equation above we have the "greater" comparison operand because even when it happens for two fill words to be operated at the same time, the probability of them having the same run-length encoded is close to zero. In fact, if the number of fill words would be equal between the two bit-vectors, the number of fill appends would be exactly double the product of the probabilities of encountering a fill word in each of the operated bit-vectors multiplied by the total number of appends. Meaning that, on average, it takes exactly two fill-words from the second bit-vector to consume a fill-word from the first fill-vector, and vice-versa. However, when the number of fill words differs between the two queried bit-vectors, then, on average, it will take more than two fill words from the larger bit-vector to consume one fill-word from the smaller bit-vector. Based on these, we estimate that the number of append-fills in WAH compression scheme can be approximated by:

$$I_f = \frac{\text{Fills}_x}{m_x} \times \frac{\text{Fills}_y}{m_y} \times I_w \times (3 - \frac{\min(m_x, m_y)}{\max(m_x, m_y)}), \tag{3.4}$$

where Fills_x and Fills_y are the number of fill words in bit-vector x and y respectively, while m_x and m_y are their respective sizes. Plugging I_f and I_l in Equation

3.2, we obtain:

$$t_{\text{WAH}} = C_a(m_x + m_y) + D_l(m_{x_l} + m_{y_l}) + D_f(m_{x_f} + m_{y_f}) + C_l I_l + C_f I_f + C_0 I_w. \quad (3.5)$$

The number of literal appends (I_l) can be approximated by subtracting the number of fill appends from the total number off appends $I_l = I_w - I_f$.

Performance Estimations for EWAH

EWAH query algorithm has some differences from WAH query algorithm. The theoretical advantage of EWAH over WAH is that it does not require any decoding for the literal words. On the other hand, maintaining the counter of the literal words when appending literals at query time adds cost to EWAH query processing time. Furthermore, some EWAH compressed bitmaps may have a larger size, due to its smaller fill encoding capacity, and thus EWAH may require more decodes for fills than WAH in these cases. The Equation 3.4 is applicable for EWAH as well, considering EWAH number of literals and fills in the compressed bit-vector.

Even if EWAH does not explicitly decode the literal words, it still has to access them once, and it also has to decrement the counter that tells how many literals are still following. This means that decoding literals for EWAH may be cheaper, however, it still comes at a cost. In general, the expected EWAH total

query time would be:

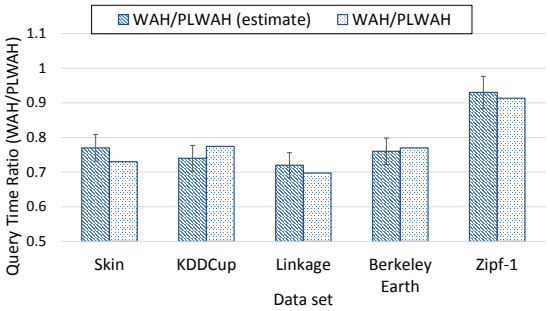
$$t_{\text{EWAH}} = C_a(m_x + m_y) + \frac{CR_x + CR_y}{2} \times D_l(m_{x_l} + m_{y_l}) + D_f(m_{x_f} + m_{y_f}) + C_l I_l + C_f I_f + C_0 I_w. \tag{3.6}$$

In Table 3.1, we set the Decode Fill time for EWAH the same as the one for WAH. However this is only true for the case when there are no fill words. As EWAH compresses better the bitmaps, the cost of decode literal decreases.

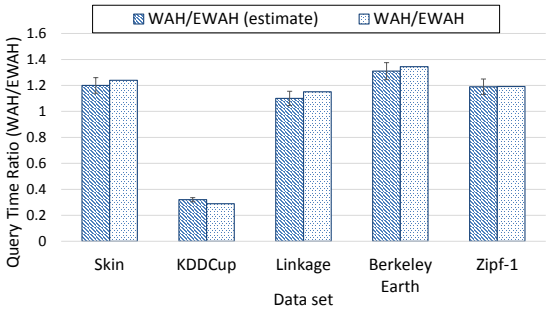
Performance Estimations for PLWAH/CONCISE

As discussed earlier, Algorithm 1 performs the same number of iterations for both, PLWAH and WAH. Because these two schemes use the same hybrid encoding, the number of fill appends and literal appends between the two schemes should be equal when performing the same query. Thus, the estimation results from Equation 3.4 are also valid for PLWAH. There are several differences between these two schemes. One of them is that PLWAH utilises less memory to store the compressed bitmap, and makes fewer memory accesses when decoding the bitmap. On the other hand, decoding a PLWAH word requires more time than decoding a WAH word due to its checks for sparse literals encoded within fills. Finally, when compressing the resulting bit-vector on the fly, at query time, every literal append should check whether or not the last word is a fill and if this is a sparse literal that can be encoded together with the fill.

These variations result in query time differences when comparing these two schemes. The ideal case for PLWAH is when it has a sparse literal between each



(a) Estimated vs recorded query time ratio WAH/PLWAH over real data



(b) Estimated vs recorded query time ratio WAH/EWAH over real data

Figure 3.1: Query time estimation over real data.

fill word. In this case PLWAH can exploit the better memory utilization and fewer bitmap accesses at maximum. The worst case is when there are no sparse literals encoded within the fills. In this case PLWAH still has a more complex query algorithm and append literal algorithm than WAH, while it does not benefit from the memory advantage.

We evaluated our estimation methods using synthetic data, as well as four real datasets, and predicted the relative performance of WAH versus PLWAH and WAH versus EWAH within a confidence level of 95%. The synthetic data was generated with a zipfian distribution. The four real datasets used in our evaluation are: KddCup[1] Berkeley Earth,[2] Linkage,[3] Skin.[4] The estimation results are shown

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

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

[3]http://archive.ics.uci.edu/ml/datasets/Record+Linkage+Comparison+Patterns

[4]https://archive.ics.uci.edu/ml/datasets/Skin+Segmentation

in Figure 3.1.

To summarize this section, we have evaluated several word-aligned bitmap compression techniques in terms of compression ratio but more importantly query time for point and range queries. As expected, no encoding is better than all others in all cases. However, these results evidence that there are specific scenarios in which one method should be preferred over the others. Depending on one's needs, the preference could be towards a more compact bitmap index or faster queries. In this study we provide insights as to which encoding should be preferred depending on the dataset. For example, EWAH could be preferred for bitmaps that are hard-to-compress, that have short fills and mostly literals; PLWAH could be preferred for very sparse bitmaps where it compresses significantly better than WAH and may have faster query times.

Furthermore, we formalise a method for estimating relative query performance between two encodings before actually compressing the bitmaps. This method only requires a single traversal of the bitmaps for providing these estimations, and has proven to be accurate. We evaluated our estimation method on synthetic data, as well as on four real datasets, and predicted the relative performance of WAH versus PLWAH and WAH versus EWAH within a confidence level of 95%.

3.2 Variable Aligned Length - VAL

Most modern bitmap compression techniques are variants of the Word-Aligned Hybrid (WAH) encoding, which uses w-bit words to align with the underlying CPU architecture, i.e., $w = 32$ or $w = 64$. While the word size w is fixed on physical constraints, there is no such requirement that the *segment length* s, i.e., the unit of compression, must be fixed at $s = w - 1$. Indeed, WAH-style fills and literals can easily be represented in $s = 7$ bit segments, which is packed into the physical unit of bytes rather than words [28].

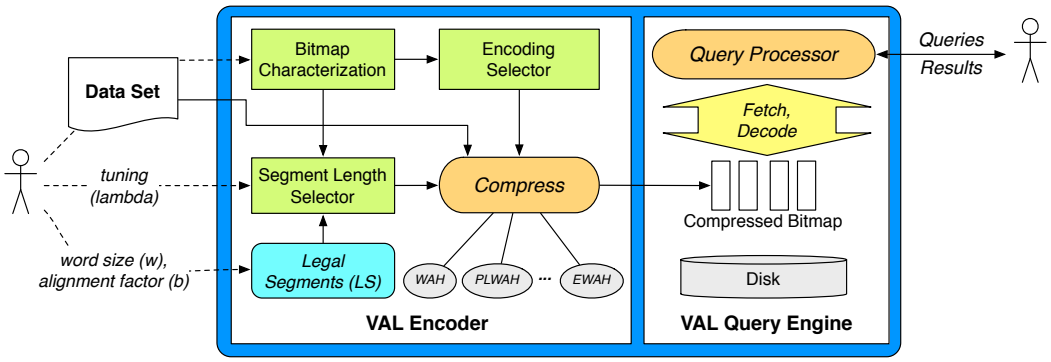


Figure 3.2: The VAL System Framework: Encoder and Query Engine.

The selection of the compression method, and independently, the segment length s, are both data and application-dependent. It is observable that there exists clear scenarios in which one method outperforms the others in time and/or space. We have identified two orthogonal aspects that can be generalized: (1) the encoding segment length s and (2) the encoding method used for compression. We propose a unified bitmap compression framework, Variable Aligned Length

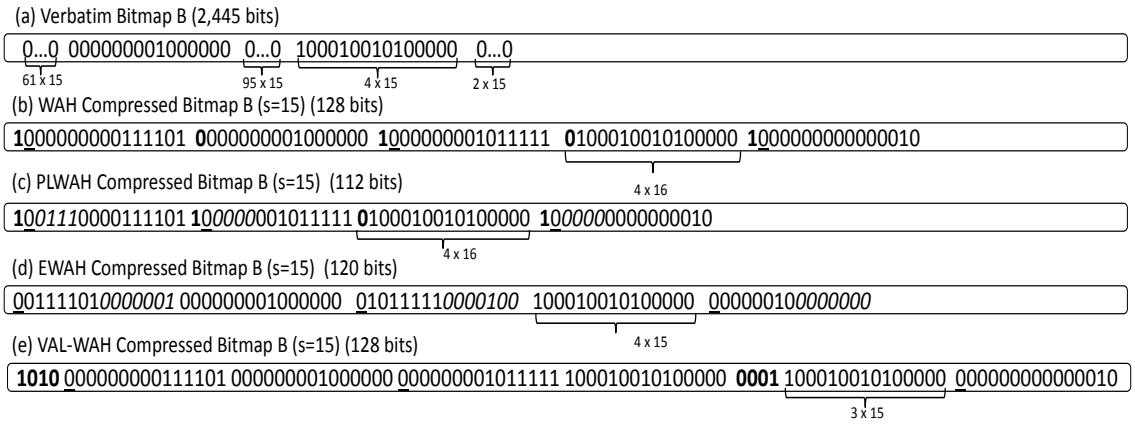


Figure 3.3: Examples of Various Word Aligned Encodings.

(VAL), where these variations can coexist. Our framework inputs user preference on the space-time trade-off, and automatically applies the optimal settings to improve performance.

The proposed VAL system framework is shown in Figure 3.2, comprising two main components: *VAL Encoder* and *VAL Query Engine*. The user inputs the data and a set of system-specific parameters. The input dataset is first characterized, e.g, by profiling the overall bit distribution and length of runs. This information is sent to the *Encoding Selector* and the *Segment Length Selector*. The former selector chooses an appropriate compression encoding scheme, and the latter decides on a segment length s to be used for encoding each bit vector. After compression, the compressed index is read by the *VAL Query Engine* which handles queries over the dataset. Queries can be executed over data compressed with different encoding techniques or segment lengths.

Bitmap Encoding Commonalities

To show the commonalities and generalizability of modern WAH-variant schemes, let us focus on the encodings for several techniques: Word-Aligned Hybrid (WAH) [26], Enhanced WAH (EWAH) [27], and Position-List WAH (PLWAH) [28]. Figure 3.3 (a) shows a verbatim, uncompressed, bitmap B that will be used to drive our examples. It contains $2,445$ bits divided into a 61×15-bit run of zeros, followed by 15 clear bits with a single set bit in position 7, then another 95×15-bit run of zeros, 4×15 bits of mixed zeros and ones, and finally, a run of 30 zeros. This example uses a $w = 64$ bit architecture and a *segment length* of $s = 15$, which means that each consecutive 15-bit segment in B will be considered at a time as units of compression.

Recall that the standard version of WAH is aligned to the machine's word size w, and thus uses a segment length of $s = w - 1$. Figure 3.3 (b), shows a WAH-encoded bitmap using a $s = 15$ *segment length*. Instead of a word, we assume a more generalized *block* of size $w' = 16$ bits. Note that in this scenario, four $w' = 16$-bit blocks can be physically encapsulated into a $w = 64$ bit word. For each w'-bit block, the most significant bit (shown in bold) is the flag bit, e.g, 1 for fills and 0 for literals. If a w'-bit block is a fill, the second most significant bit (underlined) denotes the fill bit. The remaining 14 bits in the fill block are then used to encode the run of consecutive of fill segments in the verbatim bitmap (61 for the first fill block). This is followed by a literal block, which stores the 15-bit literal segment with the 7th bit set to 1, and so on.

The next example we show is PLWAH, in Figure 3.3 (c). PLWAH's defining property lies in its ability to encode fills even with the presence of a single dirty bit, which would disrupt a run. Note that again, we are assuming $s = 15$, which imposes a $w' = 16$ block size. Notice that the single dirty bit (in the 7th position) in verbatim bit vector pollutes an otherwise longer run of 0s. Instead of encoding this dirty block as a literal, PLWAH uses the four *position* bits shown in *italic* to denote the position of the dirty bit, i.e, $(7)_{10} = (0111)_2$. If the *position* bits for a fill block are zeros, then no literal word is integrated. This determination requires some decode overhead when processing queries. PLWAH uses $p = \lceil \log_2 s \rceil$ bits to index the dirty bit position, and therefore the maximum run-length that can be encoded with a single fill block is $2^{s-p-1} \times s$. Longer fills will require two fill words to be encoded.

Figure 3.3d) shows the EWAH encoding for bit vector B. For fill blocks, $p = \lfloor \frac{s}{2} \rfloor$ bits (in *italics*) are used to indicate the number of literal blocks following the fill. The maximum run-length that can be encoded with a single fill word is thus $(2^{\frac{s}{2}-1}s)$. For this reason, EWAH typically does not compress as well as the other methods. However, its improved performance during query time is significant because literal words can be skipped or appended without decoding when operated with a fill.

Finally, Figure 3.3e) shows the bitmap B encoded using WAH within the VAL framework (VAL-WAH). VAL packs the segments into a word using w'-bit blocks and creates a word header (in bold) that stores the flag bits, one per block within each word. By placing this header in the front, we can reduce the decoding

overhead. The compression performance of VAL-WAH is the same as WAH for the same s. The compression improvements achieved by VAL come from the use of smaller segments which in general, will produce better compression than longer ones. The exception is the case where the bits used to represent a fill are not enough to represent long runs in a single block. For comparison purposes, the bitmap B from the example would require 320 bits after compression using WAH-64 ($s = 63$). For this example, WAH-64 would require $2.5\times$ more storage than VAL-WAH.

Bitmap Processing Commonalities

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

An *activeWord* can be interpreted as the following structure:

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

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

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

Due to the shared encoding similarities of the WAH variants, we observe

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

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

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

The Val Encoder

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

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

$$LS = \{2^i \times (b-1) \mid 0 \leq i \leq (\log_2 w - \log_2 b)\} \tag{3.7}$$

On a 64-bit architecture ($w = 64$) and alignment factor $b = 16$, the legal segment lengths are, $LS = \{15, 30, 60\}$. This definition of segment lengths ensures that larger segment lengths are always multiple of smaller segment lengths, and therefore the *activeBlocks* they create are always logically aligned. To further reduce the

overhead of query execution, the segments are also memory-aligned, i.e, segments never cross over two physical words. For instance, segment lengths $s = 15, 30,$ and 60 encapsulate four, two, and one block(s) into a single physical word, respectively. When needed, *blocks* are padded with zeros within the physical word. For example, recall that each *block* is encoded using $s + 1$ bits (the one extra bit is needed to flag the block as being either a literal or a fill).

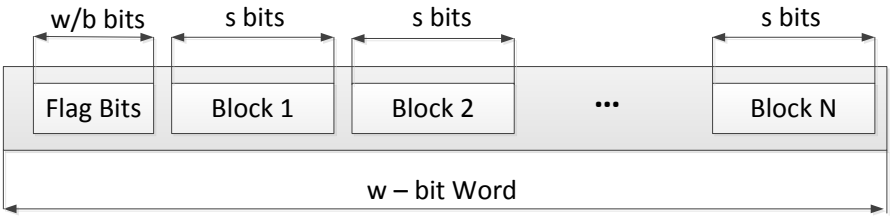


Figure 3.4: Word Encapsulation.

This word encapsulation scheme is shown in Figure 3.4. The number of blocks encapsulated into a word is given by $N = \frac{(b-1)\times w}{b\times s}$. The flag bit for each block is stored in the $\frac{w}{b}$-bit word header. The goal of the word header is to minimize the time required to align the segments between two bitmaps encoded using different block sizes. For example, two literal blocks with a VAL bitmap encoded using $s = 15$ can be directly operated with the corresponding literal block encoded using $s = 30$. It is worth noting that small alignment factors would have a significant number of unused bits for larger segments in LS. For example, for $b = 8$ and $w = 64$, the legal segment length $s = 56$ would have 7 unused bits per word. However, it

is worth noting that in many cases the increase in compression for the bit vectors encoded with smaller segment lengths will make up for these pad bits in the bit vectors encoded with larger segment lengths.

Let us consider again the system framework presented in Figure 3.2. Along with the dataset, the user also inputs the machine's word size w, the alignment factor b, and a tuning parameter λ (explained later). First, w and b are input into Equation 3.7 to determine the set of legal segment lengths, LS. Next, encoding a bitmap involves two major decisions: (1) the encoding method to use, and (2) the segment length $s \in LS$. To inform on these decisions, the *Bitmap Characterization* component passes over and profiles each bit vector from the input data. This profile is used as input into both the *Encoding Selector* and the *Segment Length Selector*.

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

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

$$s = \begin{cases} s_{c+i}, & \text{if } \frac{size(B,s_c)\cdot(1+\lambda)^{1+i+\lambda}}{i+1} \geq size(B, s_{c+i}) \\ s_c, & \text{otherwise} \end{cases} \tag{3.8}$$

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

$$s_c = \arg\min_{s_i \in LS} \{size(B, s_i)\} \tag{3.9}$$

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

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

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

in the cases where both bitmaps are compressed well, there are opportunities for faster query execution by processing whole compressed blocks.

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

VAL Implementation of WAH (VAL-WAH)

In VAL-WAH, each block in a word corresponds to a WAH fill or literal segment. As discussed previously, the flag bits for all blocks in the word are placed in the word header. The number of blocks in a word depends on the segment length used for encoding. Query execution for VAL-WAH follows the generic logical operation presented in Algorithm 1. The specialization for WAH consists in

the implementation of the *decodeNextWord()* method. Since several encoding segment lengths could be used, a complication during query processing is to execute queries involving bitmaps encoded using different segment lengths. This is done by parametrizing the decode method with an integer p that acts as a segment-length conversion factor. The segment length used for decoding is $2^p \times s$, where s is the segment length used for encoding. The possible values of p depend on the legal segments for encoding. For our current setup that allows segment lengths of 15, 30 and 60, p is in the interval $[-2, 2]$. There are three cases: $p < 0$, $p = 0$, and $p > 0$, which decode the current word into blocks using a smaller segment length, the same segment length, or a larger segment length than it was used for encoding. Specifically, $p = 0$ is used to decode segments with the same segment length as the encoding, $|p| = 1$ is used to convert segment lengths between 15-30 and 30-60, while $|p| = 2$ is used to convert between 15-60.

Converting Segment Lengths

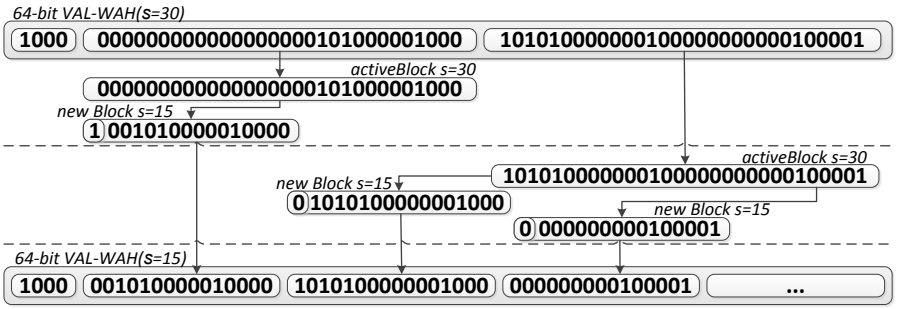


Figure 3.5: Example of converting a 64-bit VAL compressed bit vector using *s=30* down to a 64-bit VAL compressed bit vector using segment *s=15*.

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

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

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

literal is completed using the next word in the vector.

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

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

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

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

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

- Case 1 (Lines 3-7): The *activeBlock* is a literal. In this case, the *activeBlock* is divided into $2^{|p|}$ (for us, 2 or 4) literal blocks with smaller segment length and added to the *activeWord* as literals. The *activeWord* will be iterated in the main query processing function.

- Case 2 (Lines 8-10): The *activeBlock* is a fill. In this case, a single fill block is

added with same fill value and the number of segments is multiplied by $2^{|p|}$ using a shift-left operation.

As an illustration of Algorithm 2, consider the bit vector in Figure 3.5. The figure shows the conversion from $s = 30$ down to $s = 15$. First, the fill block with zeros is stored into a fill block with $s = 15$ and double the number of segments. Then the second block in the word is stored into two literals with $s = 15$. Note that, since decoding is done in memory only and $nSegments$ is a 64-bit number, the multiplication of the number of segments in the fills from larger segments lengths to smaller lengths never require extra segments to encode the fills.

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

- Case 1 (Lines 3-6): The *alignedBlock* is empty, , there are no bits left from the previous decoded block or word. If the *activeBlock* is a literal, then it is added to the *alignedBlock* as a literal and the alignedBlock is marked to be incomplete. The *alignedBlock* is not yet added to the *activeWord*.

- Case 2 (Lines 7-11): The *alignedBlock* is empty and the *activeBlock* is a fill,

then its fill is added to *allignedWord* with a factor of m less *nSegments*. The *alignedBlock* used to store the remainder bits from the division by m. They are stored as s bits in a literal.

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

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

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

- Case 3 (Lines 12-18): The *alignedBlock* is not empty, , there are leftover literal bits from a previous block, and *activeBlock* is also a literal. In this case, the *activeBlock* is appended to *alignedBlock*. If the *alignedBlock* is filled with $s \times m$

bits, then it is added to the *activeWord* and cleared.

- Case 4 (Lines 19-25): The *alignedBlock* is a literal and *activeBlock* is a fill. In this case, the *alignedBlock* is appended with literals from the *activeBlock* until it is filled with $s \times m$ bits. The *alignedBlock* is appended to *activeWord* as a literal block. Then the remaining blocks in *activeBlock* are appended to *activeWord* as a fill. If there are any leftover bits from dividing the remaining blocks by m, then they are stored as a literal in *alignedBlock*.

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

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

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

Because the harmonic mean emphasizes the smaller ratio, it captures the combined rate of speedup and compression more faithfully than an arithmetic mean. Fur-

thermore, we inverted H_M so that the larger values imply better performance, and the goal would be to show higher gain. The $gain$ across all datasets is presented in Figure 3.6. The combined gain of VAL-WAH is higher than the other encoding methods. The results for sorted and non-sorted bitmaps are shown in Figure 3.6 and Figure 3.7

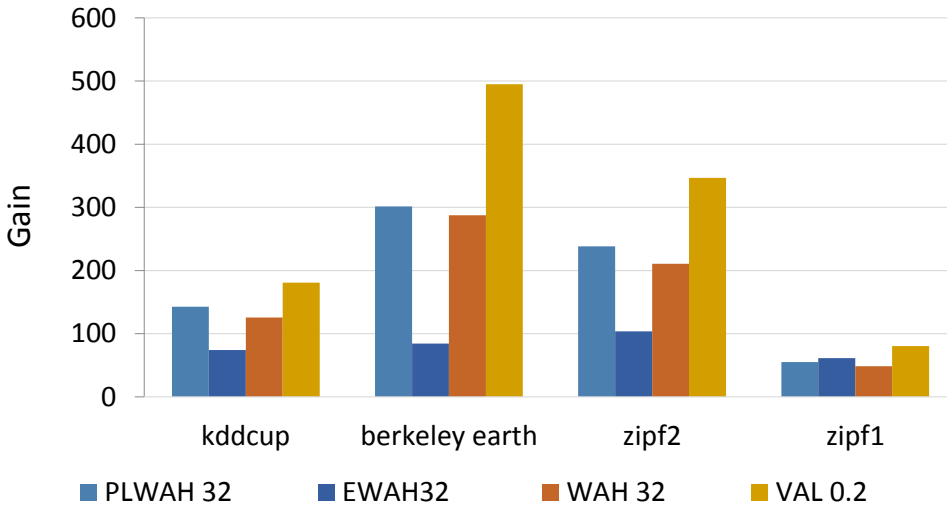


Figure 3.6: Combined Gain for Sorted Bitmaps.

To conclude, VAL enables several run-length aligned compression algorithms to coexist together and extends them to allow variable segment lengths. Efficient query execution algorithms over bitmaps compressed using different encoding lengths are proposed. A user-defined λ parameter allows users to adjust the trade-off between compressed index size versus expected query execution speed. As a proof of concept we implemented WAH within the framework and performed

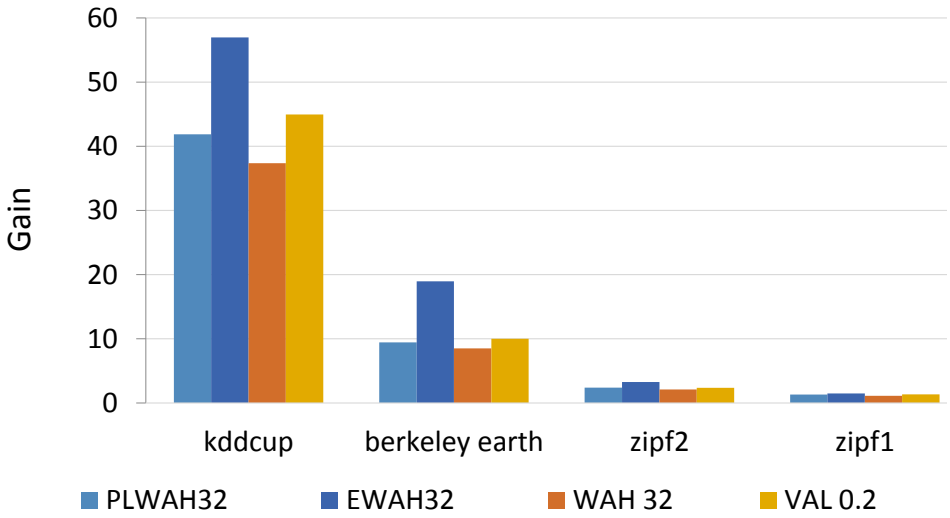


Figure 3.7: Combined Gain for Non-sorted Bitmaps.

an extensive performance evaluation. This VAL-WAH approach is very effective for sorted data, particularly for skewed data distributions and is able to outperform WAH. We also show that net gains can be obtained when applying this framework to non-sorted data especially for high-cardinality attributes. The flexibility of having variable segment lengths but still maintaining the alignment of the blocks and the segments in the bitmap are the key of the success of the proposed framework.

3.3 Hybrid compression for hard-to-compress bit-vectors

When considering compressing the bit-vectors, it is important to take into account the effects of compression. As shown in the previous section, highly compressible bit-vectors can exhibit faster query times than the non-compressed ones [41]. Nonetheless, the BSI bit-vectors are usually dense and hard-to-compress.

Therefore, compression would not always speed up queries and could add considerable overhead. In these cases, bit-vectors are often stored verbatim (non-compressed).

On the other hand, queries are answered by executing a cascade of bit-wise operations involving indexed bit-vectors and intermediate results. Often, even when the original bit-vectors are hard-to-compress, the intermediate results become sparse. It could be feasible to improve query performance by compressing these bit-vectors as the query is executed. In this scenario it would be necessary to operate verbatim and compressed bit-vectors together.

Considering the above, we propose to compress the BSI bit-slices using a hybrid compression scheme [38], which is a mix between the verbatim scheme and the EWAH/WBC bitmap compression. Bit-vectors are only compressed when the compression can improve the query time, otherwise the bit-vectors are left verbatim. The query optimizer described in [38] is able to decide at run time when to compress or decompress a bit-vector, in order to achieve faster queries. Queries can benefit from compression when the decoding of compressed bit-vectors is faster than reading a full verbatim vector, and also through less memory utilization. In the next chapter we describe in detail a way to speed-up basic operations between bit-vectors(such as intersections, unions and exclusions) by alternating the use of compressed and verbatim bit-vectors.

CHAPTER 4
QUERY PROCESSING AND OPTIMIZATION

For more complex queries such as aggregations, the BSI index is shown to perform the best in most cases [15]. Since a bitmap representing a set of rows is the simplest bit-sliced index, it is a straightforward way to determine multisets of rows resulting from the SQL clauses UNION ALL (addition), EXCEPT ALL (subtraction), and INTERSECT ALL (min). Also for the WHERE clause in SQL, for extracting values that are greater or less than a certain value, it is possible to only operate with a few bit-slices, and not the entire attribute. For this purpose we use and extend the bit-sliced index arithmetic described in [23], for operations such as sum, subtraction, min, max, average, count.

In a distributed environment, these operations are performed locally on multiple nodes in parallel. However, it is important to find the best trade-off between parallelism and network communication. In the next sections we describe the execution of aggregations and preference queries, first on a single machine, and then on a cluster.

This chapter first describes a methodology to optimize bit-wise operations between dense bit-vectors, such as those found in the BSI index. Then these atomic bit-wise operations are used to execute more complex queries in centralized environments.

4.1 Bitwise operation optimization using hybrid compression

In the previous chapter it was mentioned that dense bit-vectors could benefit from a compression scheme that alternates between verbatim and compressed bit-vectors. Further we describe a query optimizer that decides during query execution whether to compress the resulting bit-vector of a bitwise operation or not. It turns out that by compression only the best bit-vector candidates can help speed-up the query significantly.

Consider a relational database D with m attributes and n tuples. Bitmap indices are built over each attribute value or range of values and stored column-wise. Query processing over bitmaps is done by executing bit-wise operations over one or more bitmap columns.

A selection query is a set of conditions of the form $A\ op\ v$, where A is an attribute, $op \in \{=; <; \leq; >; \geq\}$ is the operator, and v is the value queried. We refer to point queries as the queries that use the equal operator ($A = v$) for all conditions and range queries to the queries using a between condition (e.g. $v_1 \leq A \leq v_2$).

The values queried, v_i, are mapped to bitmap bins, b_i, for each attribute. If the bitmaps correspond to the same attribute then the resulting bitmaps are ORed together, otherwise they are ANDed together. In the case of selection queries, the resulting bitmap has set bits for the tuples that satisfy the query constraints.

Equality encoded bitmaps can support selection queries and have been shown to be optimal for point queries. These bitmaps are sparse (even more for higher cardinalities), and therefore can benefit from compression. However, compression

comes with a cost during query processing since it requires more time to decode the columns. In some cases, however, compression can speed up the queries and also reduce the space footprint of the bitmaps. This is typically the case for highly compressed columns. For very hard to compress bitmaps, where the number of set bits is higher, compression may not able to reduce the size but still imposes an overhead during query time.

Consider for example a dataset uniformly distributed where each one of 100 attributes is divided into two bins. Each bitmap column individually is not able to compress at all (as the expected bit-density is 0.5). In this case, it may seem beneficial to store the bitmaps as verbatim bitmaps, uncompressed, therefore avoiding decoding overhead during query time. However, consider a point query over several of the dimensions for the same data-set. The expected density quickly decreases with each added dimension. For 4, 10, and 20 attributes queried the expected density for a point query would be 0.06, 0.001, and 0.000001, respectively. In this case, the use of compression would speed up the query considerable as the number of dimensions increases.

Our goal is to design a space where verbatim and compressed bit-vectors can be queried together and compression is used not only to reduce space but also to speed up query time.

In order to enable this hybrid query processing, the bitmap index representation is extended with a header word. The first bit in this word is the flag bit, e, to indicate whether the bitmap column is stored verbatim ($e = 0$) or compressed

(e = 1); and the remaining w – 1 bits store the number of set bits in the bitmap. The number of set bits is used to estimate the density of the resulting bitmap during query optimization as described in the next section.

In our hybrid space, there are three possibilities when operating two bit-vectors: both of them are compressed, both are verbatim, or one is verbatim and the other one is compressed. The first two cases (both compressed or both verbatim) present no challenge since the operations for these cases already exist. We developed query algorithms that operate an EWAH compressed bit-vector with a verbatim bitmap. We chose EWAH as our compression scheme because its word length is equal to the computer word length (w) used for verbatim bitmaps. Thus, by using EWAH we maintain the alignment with the verbatim bitmaps.

Algorithm 4 shows the pseudo-code for a general bit-wise operation $\circ$ between two bit-vectors: one compressed and the other verbatim, and produces a result that can be either verbatim or compressed. The input bit-vectors C and V correspond to the compressed and verbatim bit-vectors, respectively. While C has a more complex data structure and requires more sophisticated access to its words, V represents a simple array and its words can be accessed directly without any decoding required. This is the reason our hybrid model is able to speed up query execution of hard-to-compress bitmaps when compared to EWAH or other bitmap compression methods, as there is no decoding overhead for bit-vector V. The size of C is always smaller than or equal to the size of V, and thus Algorithm 4 will perform maximum $|C|$ iterations for the AND operations and maximum $|V|$ itera-

tions for the OR and XOR operations. Where $|C|$ is the size of C, and $|V|$ is the size of V in words.

Algorithm 4: Operation of a compressed bit-vector C with a verbatim bit-vector V.

```
   Input: C, V
   Output: R
1  pos = 0 ;
2  while C has more words do
3  |  R.appendWords(C.runningBit ∘ V.sub(pos, C.runLen), C.runLen);
4  |  pos+ = C.runLen;
5  |  for i=1 to C.NofLiterals do
6  |  |  R.append(C.activeWord ∘ V[pos]);
7  |  |  pos++;
8  |  end
9  end
10 return R
```

Having defined the input data for Algorithm 4 we will now proceed to describe its steps. Because EWAH starts always with a marker word, Algorithm 4 starts by processing this word. Recall that half of the marker word stores the fill count ($runLen$ in the Algorithm) and the other half stores the number of following literals ($NofLiterals$ in the Algorithm). The fill count and the bit value for run of fills together with the corresponding words from the verbatim bitmap are passed into the `appendWords` method (Line 3). This method, depending on the fill bit value and the logical operation to be performed $\circ$, will add a stream of homogeneous bits, or a stream of values resulting from operating the fill bit with the words from V. This number of consecutive words is equal to the fill count that was just

decoded. Next, the active position pos within V is updated (Line 4). Then, each of the following literal words in C (Line 5) is operated with the corresponding word in V (Line 6), and the active position within V is updated (Line 7). After the consecutive literals in C are exhausted, the next word in C will be again a marker word and Algorithm 4 iterates to Line 3 until all the words in the compressed bitmap C have been exhausted.

The append procedures (Lines 3 and 6) encode the resulting word(s) into the result bitmap R, which could be EWAH compressed or verbatim. The algorithm to decide whether the result should be compressed or not is discussed in the next section.

Note that the running time for the Hybrid algorithm described above is proportional to the size of the compressed bitmap for intersections (AND) and complement (NOT) operations, and proportional to the non-compressed bitmap for union (OR) and XOR operations.

Our goal in this work is to identify the cases where it is beneficial to operate a bit-vector in a compressed form or in a verbatim form. We shall compress if compression can improve, or does not degrade the query time. Our assumption is that if Algorithm 4 is called, the compressed bitmap is sparse (compression reduces the size considerably) and the verbatim bitmap is dense (marginal or no benefit from compression). In this case, operating a verbatim and compressed bitmap together would be more efficient than operating two-compressed or two-verbatim bitmaps together. Operating two verbatim bitmaps requires the traversal of the

entire bitmap regardless of the bit-density. Thus, the proposed hybrid algorithm opens the possibility to reduce the memory requirement during query processing and speed up the query at the same time. The next section describes how this hybrid scheme can be exploited to improve the query performance over bit-vectors.

4.1.1 Query optimizer

Let us identified the three cases when operating two bitmap columns as VV when both of them are verbatim, CC when both of them are compressed, and as VC the hybrid case, when one of them is verbatim and the other is compressed. Regardless of the input format, the result of operating two bitmap columns can be encoded either as a verbatim bitmap or as a compressed one. The decision of whether the result should be compressed or not is made by the query optimizer as described next.

Let us denote by n the number of bits in each bitmap, and by s_i the number of set bits in bitmap i. The bit-density is then defined as $d_i = \frac{s_i}{n}$. The bitmap encoding format, verbatim (0) or compressed (1), is denoted by the flag bit e_i.

Consider the bit-vectors B_1 and B_2 with bit-densities d_1 and d_2, respectively. Algorithm 5 shows the pseudo-code for query optimization of the binary bit-wise operations. Given two bitmaps B_1 and B_2 with bit-density and stored format (d_1, e_1) and (d_2, e_2), respectively and a bit-wise operation OP $\in \{AND, XOR, OR\}$, our query optimization algorithm estimates the resulting bit-density d and decides whether the result should be compressed ($e = 1$). The density parameters $\alpha, \beta,$ and γ

are used to indicate the maximum bit-density for which compressing the resulting bit-vector from AND, OR, and XOR operations, would be beneficial for subsequent operations.

Algorithm 5: Query optimization for AND, OR, and XOR bitwise operations.

Input: d_1, d_2, e_1, e_2, OP
Output: d, e

```
1  e = 0;
2  if (OP == AND) then
3  |   d = d1d2;
4  |   if (d < α ) ‖ d > 1 − α) then
5  |   |   e = 1;
6  |   end
7  end
8  else if (OP == OR) then
9  |   d = d1 + d2 − d1d2 ;  // d=d1+d2 for equality bitmaps from same attribute
10 |   if ((e1 == 1 & e2 == 1 & (d < β)‖(d > (1 − β)))) then
11 |   |   e = 1;
12 |   end
13 end
14 else if (OP == XOR) then
15 |   d = d1(1 − d2) + (1 − d1)d2;
16 |   if ((e1 == 1 & e2 == 1 & (d < γ)‖(d > (1 − γ)))) then
17 |   |   e = 1;
18 |   end
19 end
```

By default the encoding format of the result is verbatim (Line 1).

For ANDs (Lines 2-7), the result is compressed when the expected density of the resulting bitmap is smaller than α or it is larger than $1 - \alpha$. It is easy to see that for ANDs, the number of 1s in the result will always be less than or equal to the bitmap with smaller number of 1s.

For ORs (Lines 8-13), the result is compressed only when the two input bitmaps are compressed and the expected density of the result is smaller than β, or when the expected density of the result is greater than $1-\beta$. For ORs, the number of ones in the result is at least the number of ones in the bitmap with a larger number of 1s. If either one of the bitmaps is not compressed chances are that the result should not be compressed either. The other extreme is the case when the number of set bits in the results is so high that runs of 1s can be compressed efficiently. The comparison with $1-\beta$ allows us to compress bitmaps with a bit-density close to 1.

For XORs (Lines 14-19), the result is compressed only when the two input bitmaps are compressed and the expected density is smaller than γ or greater than $1-\gamma$ for the same reasons discussed previously for the OR operation. OR and XOR operations exhibit the same performance behavior.

For both, OR and XOR operations, both input bit-vectors have to be compressed in order to output a compressed result. Otherwise, there is an overhead to the query if the result is compressed at low bit-densities.

In the case of the NOT operation, the complement bitmap is kept in the same form as the input bitmap, i.e. if the input bitmap is compressed, the complement bitmap would also be compressed.

4.1.2 The Threshold Parameters α, β, and γ

The values for α, β, and γ should be sensitive enough to trigger the compression of the resulting bitmap without degrading the query performance.

The $append$ procedure from Algorithm 4, line 6, is responsible for appending a literal or a fill to the resulting bitmap. Given the decision taken in Algorithm 5, this procedure can append to a compressed bitmap or to a verbatim bitmap. Appending to a verbatim bitmap takes constant time. However appending to a compressed bitmap is more expensive. Previous evaluation of the EWAH open source algorithm [42] , shows that $appendLiteral$ and $appendFill$ for EWAH use on average 3 to 4 conditional branching instructions, and takes up to 5 to 15 times longer to execute than a simple append to a non-compressed bitmap [43]. The construction of the resulting bitmap in a bit-wise operation will be proportional to the size of the input bitmaps [27]. Considering this, we shall select α, β, and γ depending on the bit-density of the output bitmaps, which can be estimated from the input vector densities. The decision about compressing the result should be considered when the input bit-vectors have a density that allows to have a compression ratio of between 0.2 to 0.06 or better($\frac{1}{5}$ and $\frac{1}{15}$). Further we will work with bitmap densities instead of compression ratios because we operate with both, compressed and verbatim bitmap.

For a uniformly distributed EWAH bitmap vector x, the compression ratio can be estimated as:

$$CR_x \approx 1 - (1 - d_x)^{2w} - d_x^{2w}$$

where, CR_x is the compression ratio (compressed/non-compressed), d_x is the bit density for bitmap vector x, and w is the word length used for run length encoding. In this work we use w=64 to match the computer architecture of our ex-

perimental environment. To compensate for the overhead of compressing the bit-vector, the expected compression ratio should be between 0.2 and 0.06, obtained when d is between 0.0005 and 0.002.

For the unions, we choose the more conservative boundary. Thus, if the denser bit-vector has a density smaller than or equal to 0.0005, then the maximum density the result can have is 0.001 and the minimum can be 0.0005. The resulting bit-vector has the maximum possible density when none of the set-bit positions coincide in the operated bit-vectors. The minimum possible density for the result occurs when all the set-bits have the same positions for both bit-vectors being operated. Hence we pick β, and γ to be between these two values.

For the intersections, we want a more aggressive compression policy. If the sparser input bit-vector has a density smaller than or equal to 0.002, then the maximum probable density of the resulting bit-vector is 0.002. This is the case when both bit-vectors being operated have the same set-bit positions. Because this rarely happens, we can set a lower threshold for α and compress the result more aggressively.

4.1.3 Bit-Density Estimation

The bit-densities of the input bitmaps are used to decide whether the result should be compressed or left as a verbatim (uncompressed) bitmap. For the indexed bitmaps we store the number of set positions and the densities are easily computed. However, for the subsequent query operations involving the re-

sulting bitmaps, we use estimated bit-densities, as computing the actual densities can be expensive. The expected density of the resulting bitmap is estimated using the densities of the input bitmaps. In this work we assume that the distribution of the set bits in the two input bitmaps are independent and compute the expected density using the probability of set bits in the result bitmaps. It is typical for query optimizers to make this assumption in query selectivity estimations and even when for most real datasets not all the attributes are independent, this estimation gives reasonable performance as we will show in the experimental results over real datasets.

For the NOT operation, the bit density can be computed easily since the number of set bits in the complement of bitmap B_1 would be $n - s_1$ as:

$$d_{NOT} = 1 - d_1$$

The bit-density of the bitmap resulting from ANDing the two bit-vectors can be computed as the product of the two bit-densities. This is the probability that the same bit is set for both bit-vectors:

$$d_{AND} = d_1 \times d_2$$

Similarly, the bit-density of the bitmap resulting from ORing the two bit-vectors can be computed as the sum of the two bit-densities. This is the probability that the bit is set for either one of the bit-vectors minus the probability that the bit

is set in both bit-vectors:

$$d_{OR} = d_1 + d_2 - d_1 d_2$$

This is how the expected density is computed when bitmaps from different attributes are ORed together and for all BSI encoded bitmaps. For equality encoded bitmaps, however, since only one bit is set for all the bitmaps of an attribute, $d_1 d_2$ is known to be zero and the bit-density in the case when two bitmaps from the same attribute are ORed together is estimated by:

$$d'_{OR} = d_1 + d_2$$

For XOR, the bit-density of the resulting bitmap corresponds to the probability that only one bit is set between the two bitmaps and can be estimated as:

$$d_{XOR} = d_1(1 - d_2) + (1 - d_1)d_2$$

Evaluation of the optimizer

To measure the overhead of our query optimizer we set-up a scenario where the hybrid model should have exactly the same running time (if neglecting the overhead imposed by the optimizer) as when operating with only verbatim bitmaps or only compressed bitmaps. Multi-dimensional queries involving a variable number of bitmaps (between 2 and 10) were executed for AND, OR, and XOR operations and the results are shown in Figure 4.1.

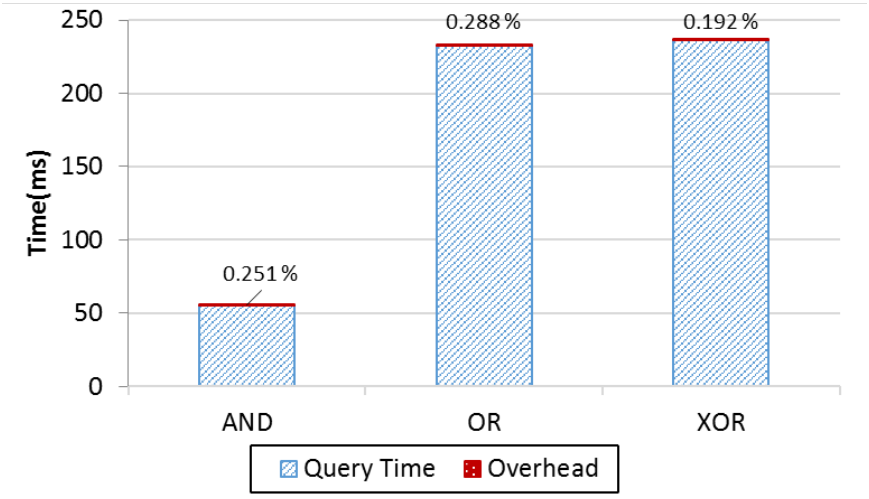


Figure 4.1: The query overhead created when the optimizer cannot improve query time.

For the AND queries we randomly generated uniform distributed low density bitmaps ($d = 10^{-4}$) with 100M bits. Since these bitmaps are highly compressible, the hybrid model starts only with compressed(EWAH) bitmaps. Because the AND optimization keeps the result compressed when the input is compressed, and the density is below the threshold, our optimizer always keeps the result as a compressed bitmap. We measured the time required by our hybrid model to execute 45 AND queries and run the same set of queries using only compressed bitmaps. The difference is reported as the optimization overhead.

Following the same methodology, for OR and XOR operations we generated uniformly distributed high-density bitmaps ($d \approx 0.05$) with 100M bits and present the cumulative query time of 45 queries. Because when performing OR and XOR operations over these bitmaps the results are not compressible, our query optimizer keeps the results as verbatim. We ran the same set of queries over verbatim

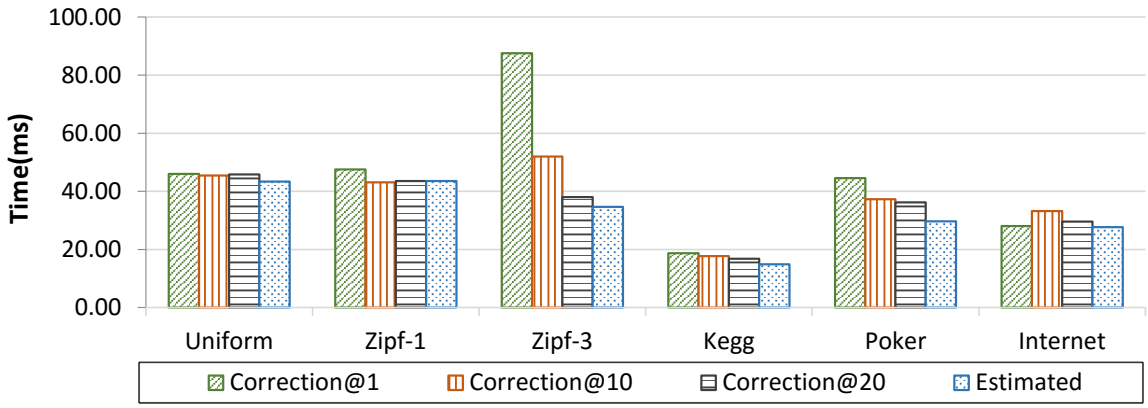


Figure 4.2: Top-K query times for datasets with average attribute correlation: [Uniform=0.0002]; [Zipf-1=0.0003]; [Zipf-3=0.00032]; [Keg=0.26]; [Poker=0.02]; [Internet=0.024].

bitmaps and report the difference as the optimization overhead.

As Figure 4.1 shows, the overhead that the query optimizer brings is very modest (around 0.2%). It is worth noting that this is a worse case scenario where the optimizer cannot improve the query time over the non-hybrid model. As we will see in the next experiments, the overhead is more than justified by the overall improvement in query time and memory utilization.

To evaluate the effect of our bit-vector independence assumption, we compare the decisions made by the optimizer using the estimated density against the decisions made using the densities of the intermediate results by computing them at query time. Then we count the number of times the estimations led to a different decision than the actual computed densities. Table 4.1 shows the number of mismatches for real datasets when performing the top-k queries.

One could argue that simply computing the bit-vectors density will never

Table 4.1: Percentage of mismatch optimization decisions when using estimated vs. measured density for the intermediate results.

Dataset	HIGGS	Kegg	Network
Total operations	823	252	311
Number of mismatches	4	8	18
Mismatch %	0.48	3.2	5.8

produce errors, and could offset the computing time by always taking the "right" decision, and producing faster queries. To verify if this is the case, we set to measure the top-k query times for six datasets (3 synthetically generated, and 3 real), one of which has high attribute correlation. The average attribute correlation for the `Kegg` dataset is 0.26, however the correlation between some attributes is as high as 0.7. Figure 4.2 shows the top-k query times for the six datasets. The `Uniform`, `Zipf-1`, and `Zipf-3` are synthetically generated datasets, while `Kegg`, `Poker`, and `Internet` are real datasets. The average attribute correlation is provided in the figure description. In this figure we compare the query time when using density estimation, and when computing the bit-vector density. We also measure the query time for a combination between the two approaches to determine the density of the resulting bit-vector. `Correction@10` means that if one of the bit-vectors being operated has its density resulted from previous 9 bit-wise operations using estimation, then the 10^{th} bit-wise operation will use a scan and compute. For `Correction@20` the density calculation is used every 20^{th} operation. We use this technique for diminishing the effects of density estimation error propagation. Even if we make use of the POPCNT CPU instruction when computing the bit-vectors cardinality,

the estimation still outperforms the other approaches, even for highly correlated datasets, where there may be a higher density estimation error.

4.1.4 Beyond bit-wise operations

The density estimation methods for the set of basic operations supported by bitmap indices shall be sufficient for computing the bit-vector densities in more complex queries. For a bit-vector based index, a more complex query is usually composed by a cascade of basic bit-vector operations (intersections, unions, complements, exclusions). For instance, a range query may contain a series of unions, or a point query may contain several intersections.

Our hybrid compression method is designed to work with hard-to-compress bit-vectors. The bit-sliced index is a bit-vector index that is hard-to-compress because of a high set-bit density. It is usually preferred over the bitmap index when working with high cardinality domains. Some of the queries supported by the bit-sliced index are: range queries, aggregations, top-k, and term matching queries.

The more complex queries such as top-k preference queries aim to retrieve a small portion of the data, and at the same time they involve a high number of basic bit-vector operations. This means that the final result, as well as many intermediate results from the bit-wise operations, produce low density bit-vectors. This is one of the cases when the hybrid query optimizer can help speeding up the query, and reduce the memory utilization, by compressing the sparse intermediate results.

In the following experiments we use three synthetically generated datasets

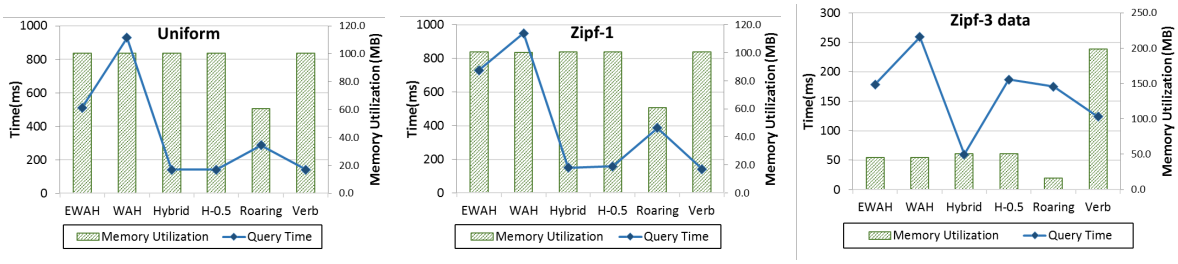
and also real datasets, to cover a large enough range of distributions and densities. We present the results in terms of top-k query time and memory utilization. We focus on memory utilization rather than on the index size because the BSI index is usually a high-density bitmap index and hard to compress. However, because the bit-slices are operated multiple times, the intermediate results can become sparser and there are opportunities for compression. We also add for comparison a naïve Hybrid method that compresses the intermediate result if it can achieve a compression ratio of 0.5 or better. This is denoted by H-0.5 in our figures. For the Hybrid method (H) we set the initial threshold T to 0.5. However this is application dependent and can be changed depending on the need for space or faster queries.

Figure 4.3 shows the results in terms of execution time and memory utilization over synthetic data when top k queries are executed over BSI indices. The datasets contain 5 attributes with normalized values with 6 decimal positions, 10 million rows, and generated using three different distributions: uniform, zipf-1 (f=1) and zipf-3 (f=3).

For the uniform data-set (Figure 4.3a), The index had 100 slices, an average of 20 slices per attribute. Only the Roaring bitmap was able to compress the initial index, with a compression ratio of 0.64, however in terms of Top-K query times it was two times slower than the Hybrid. The Hybrid scheme, also was faster than the verbatim by about 2%. In terms of memory utilization, the Hybrid performed similarly to WAH and EWAH, however it was several times faster (3.6x faster than

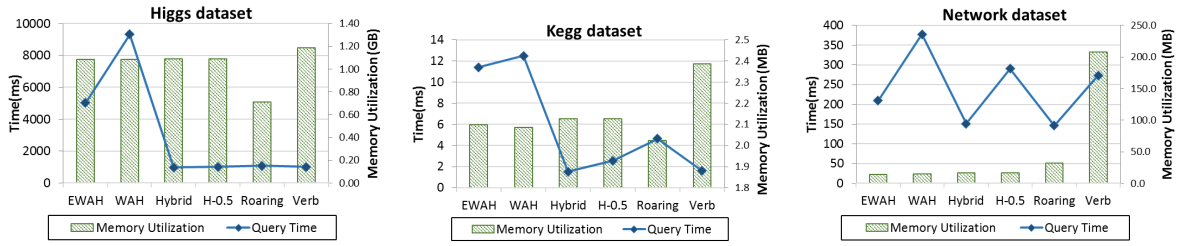
EWAH and 6.6x faster than WAH).

With the zipf-1 skewed dataset, results showed in Figure 4.3b, the results were similar to those Figure 4.3a.



(a) Uniform distribution (b) Zipf distribution (f = 1) (c) Zipf distribution (f = 3)

Figure 4.3: TopK queries over synthetic datasets with 5 attributes, 10M rows. Each attribute is represented by a BSI with 20 slices (normalized values with 6 decimal positions).



(a) Higgs dataset (b) Kegg dataset (c) Network dataset

Figure 4.4: TopK queries over real datasets.

For the zipf distribution with f=3 (Figure 4.3c), the skew in the data is significantly higher than in zip-1 and thus the bit-slices are highly compressible. Hav-

ing the compression threshold T set to 0.5, the Hybrid started with 79 compressed bit-slices out of 100. As shown in Figure 4.3c, even if the Verbatim scheme has a very forthright query algorithm without requiring any decoding/encoding, the execution time for the top-K queries is 2 times higher than the Hybrid scheme. This again underlines the potential of compression not just for storage-space purposes but also for speeding up processing. At the same time, The Hybrid was 2.8 times faster than the Roaring bitmap, even if the Roaring bitmap compressed the bit-slices better. The Hybrid scheme also used only 5% more memory than WAH and EWAH and was 4.3 times faster than WAH and 3 times faster than EWAH.

Furthermore, to show that our synthetically generated data-sets accurately represent the performance gains that can be obtained over real-world data, we perform top-K queries over the real data-sets described at the beginning of this section. The results in terms of of execution time and memory utilization for the real data-sets are shown in Figure 4.4.

For the `Higgs` dataset, the bit-sliced index contains 607 bit-slices(an average of 29 slices per attribute). The BSI-sum algorithm [38], together with the top-K algorithm performed 234 XOR operations, 345 AND operations and 244 OR operations for this dataset. From a total of 823 logical operations executed, 131 of their results were compressed and 692 were not. The Hybrid scheme was 5% faster and used 7% less memory than the Verbatim scheme, was 14% faster and used 33% more memory than the Roaring bitmap. When compared to WAH and EWAH, the Hybrid scheme, was 5 times faster than EWAH and an order of magnitude faster

than WAH, while using under 1% more memory than these two schemes.

The bit-sliced index for the `Kegg` dataset has 243 bit-slices, and the Hybrid compressed 44 of them, to start with. From a total of 252 logical operations performed, the Hybrid compressed 26 of their results, and 226 where not compressed. The BSI-sum algorithm [38], together with the top-K algorithm performed 86 XOR operations, 110 AND operations and 68 OR operations for this dataset. In terms of query time, the Hybrid outperformed all the other schemes. It was 5% faster than Verbatim, 3 times faster than Roaring, 1.7 times faster than H-0.5, 8.2 times faster than WAH, and 7.5 times faster than EWAH.

The `network` data-set is in a way similar to the zipf-3 data, its values having a higher skewing factor. The bit-sliced index for the `network` has 207 bit-slices, and the Hybrid compressed all of them, to start with. From a total of 311 logical operations performed, the Hybrid compressed 197 of their results, and 114 where not compressed. The BSI-sum algorithm [38], together with the top-K algorithm performed 104 XOR operations, 139 AND operations and 68 OR operations for this dataset.

Evaluations of the hybrid compression show that the hybrid approach always outperforms the use of verbatim only bit-vector in terms of both, query time and memory utilization. At the same time, the evaluations show the scenarios when this model can outperform compressed only bitmaps. For the top-K queries performed over real data-sets we achieved up to an order of magnitude faster queries when compared to the run-length compressed schemes, and up to 11 times

less memory used when compared to the verbatim scheme. However, further investigation regarding the value of the parameters α, β, and γ is necessary.

4.2 Complex queries using Bit-sliced Indices (BSI)

In addition to range and point selection queries we take on investigating the performance of bit-vector indices on more complex queries, such as aggregations, K-Nearest Neighbors(K-NN), collaborative filtering, and preference queries. As a proof of concept, we implement the necessary distributed BSI arithmetic logic for performing such queries over the BSI index.

4.2.1 Top-K Preference queries

Top-k (preference) queries are used in several domains to retrieve the set of k tuples that match the best with a given query. For high-dimensional spaces, evaluation of top-k queries is expensive, as data and space partitioning indices perform worse than sequential scan. An alternative approach is the use of sorted lists to speed up query evaluation. This approach extends performance gains when compared to sequential scan to about ten dimensions. However, data-sets for which preference queries are considered, often are high-dimensional.

We explore the use of bit-sliced indices (BSI) to encode the attributes or score lists and perform top-k queries over high-dimensional data using bit-wise operations [44]. Our approach does not require sorting or random access to the index. Additionally, bit-sliced indices require less space than other type of indices. The size of the bit-sliced index (without using compression) for a normalized data-set

with 3 decimals is 60 times smaller than the size of sorted lists. Furthermore, our experimental evaluation shows that the use of BSI for top-k query processing is more efficient than Sequential Scan for high-dimensional data. When compared to Sequential Top-k Algorithm (STA), BSI is one order of magnitude faster. Figure 4.6 shows the query times for executing top-k preference queries over the BSI index, and how it compares with existing state of the art approaches.

Consider a relation R with m attributes or numeric scores and a preference query vector $Q = \{q_1, \ldots, q_m\}$ with m values where $0 \leq q_i \leq 1$. Each data item or tuple t in R has numeric scores $\{f_1(t), \ldots, f_m(t)\}$ assigned by numeric component scoring functions $\{f_1, \ldots, f_m\}$. The combined score of t is:
$F(t) = E(q_1 \times f_1(t), \ldots, q_m \times f_m(t))$ where E is a numeric-valued expression. F is monotone if $E(x_1, \ldots, x_m) \leq E(y_1, \ldots, y_m)$ whenever $x_i \leq y_i$ for all i. In this paper we consider E to be the summation function: $F(t) = \sum_{i=1}^{m} q_i \times f_i(t)$. The k data items whose overall scores are the highest among all data items, are called the top-k data items. We refer to the definition above as **top-k weighted** preference query.

Let us denote by B_i the bit-sliced index (BSI) over each attribute i. A number of slices s is used to represent values from 0 to $2^s - 1$. $B_i[j]$ represents the j^{th} bit in the binary representation of the attribute value and it is a binary vector containing n bits (one for each tuple). The bits are packed into words, the storage requirement for each binary vector is n/w, where w is the computer architecture word size (64 in our implementation).

In order to compute the score for each data point we first multiply the at-

Algorithm 6: Preference query execution using bit-slices. B is the set of all BSIs, q is the query vector, and k is the desired number of results.

```
prefQuery (B,q,k)
1:  if (k < 0)
2:     Error ("k is invalid")
3:  S = Multiply (q_j, B_j) where j is the first non-zero weight in q
4:  for (i = j + 1; i ≤ m; i++)
5:        if q_i > 0
6:           S = SUM_BSI(S, Multiply (q_i, B_i))
7:  T = TopK (S, k)
8:  return T
```

tribute value by the query preference for that attribute using bit-wise operations. Given a query Q, the query vector is first converted to integer weights based on the desired precision. Let us denote by b, the number of bits used to represent a query preference. The preference query execution algorithm pseudo-code is given in Algorithm 6. The Multiply, SUM_BSI, and TopK operations called in this algorithm are performed over the BSI index, and are described in more detail in [23] and [44].

We identify three main parts in our main algorithm 6:

1. For all non-zero weights, multiply the BSI for the attribute with the corresponding query weight (Lines 3 and 6, and steps 1-6 in Figure 4.5).

2. Sum the partial scores produced by the Multiply algorithm into a BSI S (Line 6, and steps 7-9 in Figure 4.5).

3. Find the top k data points given the final BSI score S (step 10 in Figure 4.5).

Figure 4.5 illustrates the steps necessary to perform a weighted top-K query over the BSI index.

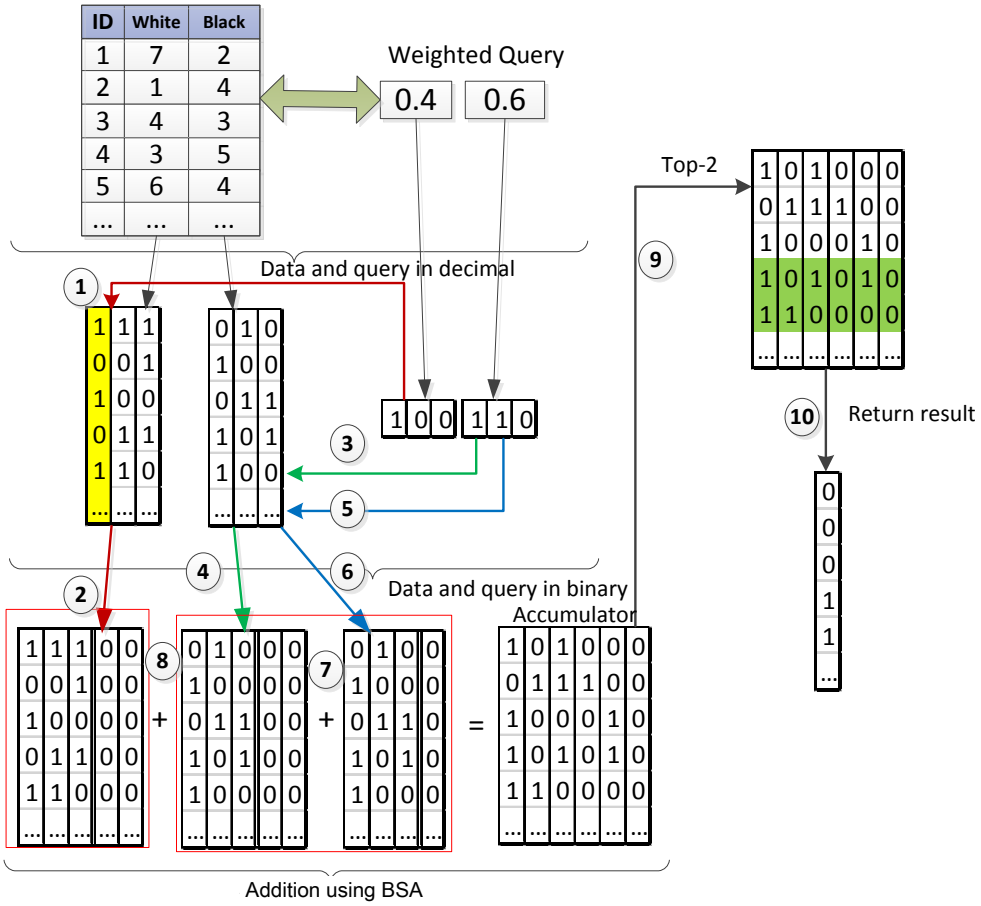


Figure 4.5: Example of BSI Arithmetic applied for finding top-2 tuples given a weighted preference query.

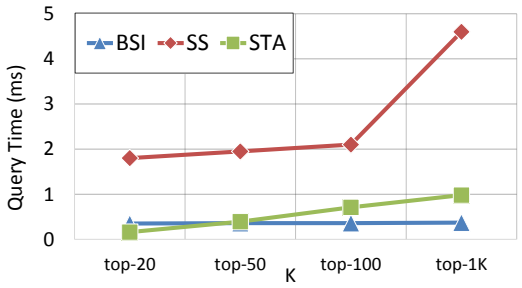
To evaluate the performance of the preference queries over the BSI index, we compared our approach with a sequential scan approach and the Sequential Top-k Algorithm (STA) [45]. The results are shown in Figure 4.6. More details regarding the experimental setup and dataset descriptions are provided in [44]. As

the figure shows, increasing k when extracting top-k candidates, does not impact significantly the query time for BSI. In fact, the query time increases by only 0.01 - 0.03 ms when changing k = 10 to k = 1000 for all four datasets.

The reason is that the top K BSI algorithm is only invoked once after the scores for all the tuples have been computed. As expected, SS and STA performance increases linearly with k.

(a) `coil2000` dataset. (Rows: 9822, Attributes: 86).

(b) `internet` dataset. (Rows: 10104, Attributes: 72).



(c) `kegg-metabolic` dataset.(Rows: 53413, Attributes: 24).

(d) `poker-hand` dataset. (Rows: 1,000,000; Attributes: 11).

Figure 4.6: Top-k weighted query on real data (K= 20 - 1,000).

In the BSI index the attributes are indexed independently, this technique for processing preference queries can take advantage of the columnar storage and

have the potential to be executed in parallel. We introduce several algorithms for processing top-k, and top-k weighted queries while exploiting the fast bit-wise operations enabled by the BSI index [44]. This approach is robust and scalable for high dimensional data. In our experimental evaluation we show that by increasing the dimensionality of the data, BSI query times increase only linearly-proportional to the number of attributes added. Moreover, the distribution of the data does not affect the query performance for BSI, while TA and other threshold algorithms using sorted lists are very sensitive to data distribution.

In the next section we define algorithms to perform parallel aggregations over the BSI index in a distributed environment. Efficient aggregations are very important for queries such as K-NN, collaborative filtering, and top-K preference queries. We execute these queries using the MapReduce paradigm.

CHAPTER 5
DISTRIBUTED BIT-SLICED INDEX

In the previous chapter it was showed that the bit-sliced index (BSI) is scalable for high-dimensional data and showed to outperform existing approaches for answering top-k preference queries. Due to independently indexed dimensions and no need for sorting, the BSI index can also be efficiently partitioned. The horizontal and vertical partitions are straight forward, assuming that the vertical partitions store whole attributes. However, we go a step further and break the attributes by vertical slicing, and store the bit-slices into partial BSI attributes. By doing so we enable for a finer control over the task parallelism, load balancing and network communication when answering queries using the distributed BSI index. Additionally, each bit-slice is compressed using the hybrid compression method described in the previous chapter. The compression further reduces the size of the index and, which translates into less memory consumption and network communication.

5.1 Structure of the distributed BSI index

Let us denote by B_i the bit-sliced index (BSI) over attribute i. A number of slices s is used to represent values from 0 to $2^s - 1$. $B_i[j]$ represents the j^{th} bit in the binary representation of the attribute value, and it is a binary vector containing n bits (one for each tuple). The bits are packed into words and each binary vector encodes $\lceil n/w \rceil$ words, where w is the computer architecture word size (64 bits in

our implementation). The BSI can be also compressed using the hybrid approach described in the previous section.

For every attribute i in R we create a Bit-sliced index B_i. In this work we want to address the problem of handling large datasets that do not fit into the memory of a single machine, and thus the BSI index has to be partitioned and distributed across the nodes of a cluster. With this goal in mind, we create a $BSIattribute$ class that can serve as a data structure for an atomic BSI element included in a partition. Each partition can include one or more $BSIattribute$ objects. A $BSIattribute$ object can represent all tuples of a attribute (in the case of vertical-only partitioning) or only a subset (in the case of horizontal, or vertical and horizontal partitioning). Furthermore, a $BSIattribute$ object can carry all of the attribute bit-slices or only a part of them. In this case we would create a number of partial $BSIattribute$s each containing a subset of bit-slices from the $BSIattribute$ that they represent.

Figure 5.1 shows the structure of the $BSIattribute$ class. The `signed` flag marks if the numbers represented by the attribute are signed or not. In case the attribute is signed and contains negative values, then it can be represented as two's complement or sign-magnitude, marked by the `twosComplement` flag. In this case the instance represents a $BsiSigned$ subclass.

The `decimals` field indicates the number of bits to the right of the decimal point, these variables are used for the execution of BSI arithmetic operations such as sum, difference and multiplication. The `numSlices` field stores the number of

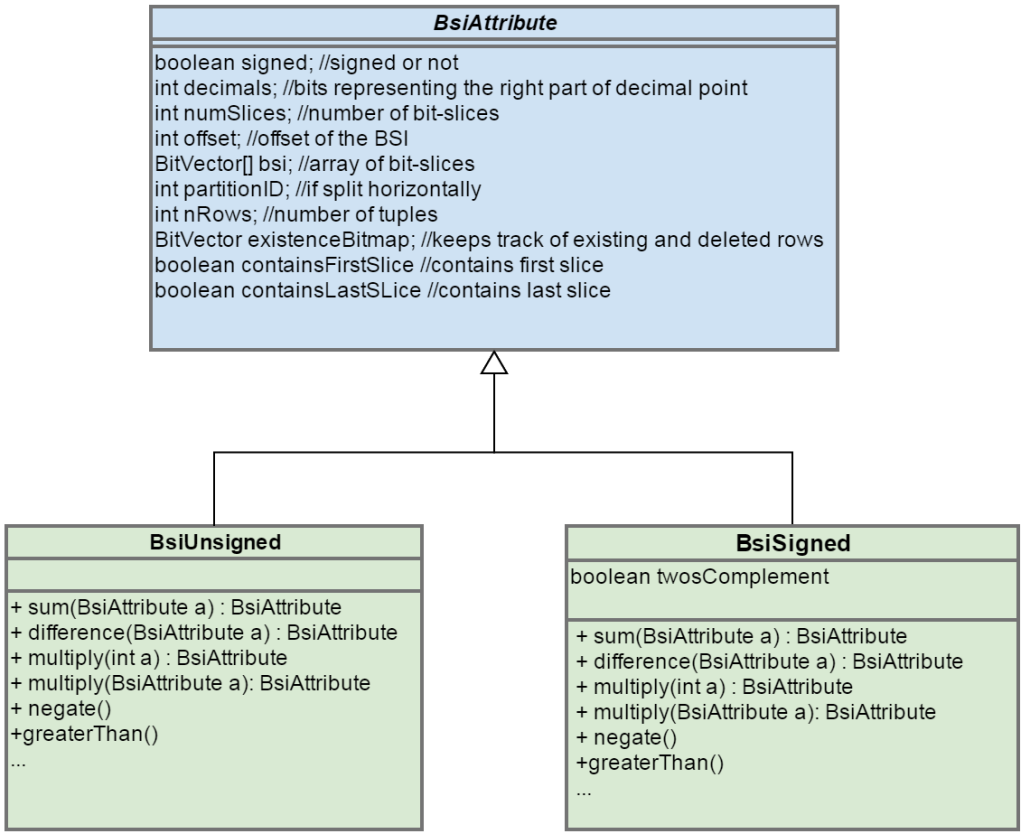


Figure 5.1: BsiAttribute class diagram.

bit-slices contained in the `bsi` array. The `offset` represents the number of positions the bit-slices within this $BSIattribute$ should be shifted to retrieve the actual value. The `partitionID` field stores the sequence of the $BSIattribute$ segment if the attribute is horizontally partitioned. The `partitionID` together with the `nRows`, which stores the number of rows in the $BSIattribute$ segment, help mapping the row IDs with the BSI values. The `existenceBitmap` is a bitmap that has set bits for existing tuples and zero for deleted rows or for the non-existing rows corresponding to the last bits in the BSI added to complete a full word.

Because each BSI attribute can be further partitioned vertically by mapping bit-slices or groups of bit-slices, it is important to track the first and last bit-slice of a BSI attribute that is partitioned using slice mapping. The `containsLastSLice` flag helps identifying the vertical partition of a BSI attribute that contains the last bit-slice. This helps while executing distributed operations for when sign extension is necessary, as the last slice represents the sign slice. Similarly, the `containsFirstSLice` indicates the presence of the first slice. The partition containing the first slice is handled differently in the case of negations or transformations between two's complement and sign-magnitude.

The $BSIattribute$ class also contains methods for various operations between attributes and transformations of the attribute. Figure 5.1 shows only a few of the methods that have been developed in our implementation.

5.2 Partitioning of the BSI index

The data partitions, as well as the index partitions, are stored on a Distributed File System (DFS) that is accessed by a cluster computing engine. We implemented the distributed BSI arithmetic (summation and multiplication) on top of Apache Spark, and used its Java API to distribute the workload across the cluster.

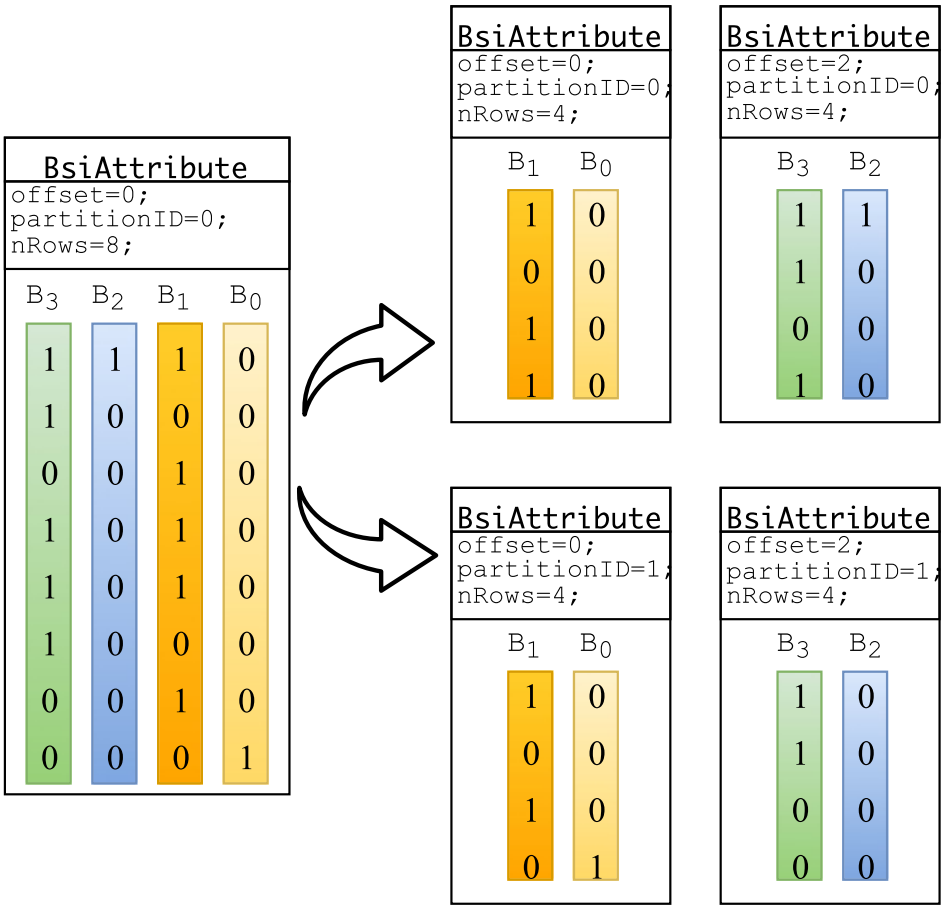


Figure 5.2: Example of vertical and horizontal partitioning of a BsiAttribute.

An example of how a BSI attribute can be partitioned and how some of the fields shown in Figure 5.1 are used to keep record of the resulting partitions, is shown in Figure 5.2.

When creating the BSI index, it is worthwhile to co-locate both the data partitions and index partitions on the same nodes. Co-locating the index and the data partitions helps to avoid data shuffling during the index creation, and also avoids network accesses, should the original data be accessed at any moment of the query execution.

The BSI index, as defined in [15] is a vertically partitioned index. However the bit-slices can also be segmented to allow for smaller index partitions that can fit on a disk page and main memory. We call this type of segmentation *horizontal partitioning*. In a distributed environment, the main concern is to minimize network throughput, while maximizing the parallelism, and horizontal partitioning can help in this regard.

Concatenation is straight forward, as each BSI in a partition has the same number of bits corresponding to the same rowIds.

Algorithm 7 shows the pseudo-code for concatenating the resulting list of BSI attributes in the case of vertical and horizontal partitioning. The $concatenate()$ operation on line 4 of the algorithm concatenates each bit-slice from t to the bit-slices of $finalSum$, where t is the next segment of the horizontal split BSI attribute. The $existenceBitmap$ is also concatenated, and the number of rows in $finalSum$ is updated.

Algorithm 7: Concatenation of BSI attributes from different horizontal partitions.

```
Input: List<BSIattribute> sumAtt
Output: BSIattribute finalSum
1 BSIattribute finalSum = new BSIattribute();
2 while sumAtt.hasNext() do
3 |   t = sumAtt.next();
4 |   finalSum.concatenate();
5 end
6 return finalSum
```

For queries involving all the attributes, as it is the case for topK queries, it may be inefficient to index a single attribute per partition. This would mean that all the index data would be shuffled during aggregation. On the other hand, grouping too many BSI attributes together creates very large partitions that may not fit into the nodes main memory. Hence, horizontal partitioning of the bit-slices can help lower the partition sizes, to allow for in-memory processing, while still minimizing network throughput by grouping multiple attributes together.

5.3 Data encoding

In the first section of this chapter we showed that the signed and unsigned attributes are treated differently. The signed data can be represented in two's complement encoding, which is suitable for addition and subtraction operations. However, for multiplications it is often more efficient to represent the data as sign-magnitude(i.e. only store the magnitude of the value, and keep record of the sign in the sign bit-vector).

We created several procedures within our $BSIattribute$ class that help with

the transformation from one encoding to another.

The $negate$ method changes the sign of each value encoded in the $BSIattribute$. The resulting $BSIattribute$ will always be a $BsiSigned$ instance encoded as either sign-magnitude or two's complement.

The abs method returns the absolute values encoded in a BSI attribute. It is always a instance of the $BsiUnsigned$ subclass.

Within the $BsiSigned$ subclass there are two possible transformations that will change the encoding of the attribute: $toSignMagnitude$ and $toTwosComplement$. These methods transform the encoding from two's complement to sign-magnitude and vice versa.

Algorithm 8: Changes the encoding from sign-magnitude to two's complement.

```
   Input: BSIattribute
1  for i=0 to numSlices do
2  |  bsi[i]=bsi[i] XOR sign;
3  end
4  if lastSlice then
5  |  addSlice(sign);
6  end
7  if firstSlice then
8  |  addOneSlice(sign);
9  end
10 setTwosFlag(true);
```

When operating BSI attributes encoded in two's complement, there is need for sign extension. However, because the attributes are often partitioned vertically, only the partitions containing the last slice (i.e. sign slice) should be sign extended.

Similarly, when changing the encoding from sign-magnitude to two's complement and vice versa, after flipping the bits, only the partitions containing the first slice will need the addition of the sign slice (Algorithm 8 line 8, and Algorithm 9 line 5).

Algorithm 9: Changes the encoding from two's complement to sign-magnitude.

```
   Input: BSIattribute
1  for i=0 to numSlices do
2  |  bsi[i]=bsi[i] XOR bsi[numSlices-1];
3  end
4  if firstSlice then
5  |  addOneSlice(sign);
6  end
7  setTwosFlag(false);
```

The unsigned BSI attributes don't require special handling of the first and last slices. Thus, every partition should be handled alike when performing parallel arithmetic operations such as summations. However, if encoded in two's complement, then the BSI arithmetic operations defined in [23] for centralized systems, must be modified to apply the sign extension only to the partitions containing the last bit-slice.

CHAPTER 6
DISTRIBUTED QUERY PROCESSING

Implementing a centralized solution to a distributed environment presents unique challenges. It is important to find a good balance between the task parallization and network communication. Load balancing is important as well. For example, in the case of summing BSI attributes in parallel for aggregation purposes, some attributes can have a much higher cardinality than others. The nodes operating those attributes can become strugglers, and impose a delay on the next scheduled stage in the processing of the query.

In the next sections we present our solution for a distributed execution of basic arithmetic over the BSI index and result aggregation across multiple dimensions, which is critical for complex analytical queries such as preference queries, K-NN (nearest neighbor), and collaborative filtering over the BSI index. We propose a two-phase map-reduce algorithm while grouping the BSIs by slice depth for the aggregation. We then estimate the amount of network communication, and define a number of parameters that should be taken into consideration given the query, dataset, and the cluster infrastructure.

6.1 Distributed top-k queries using BSI indexing

The key idea behind this work is the parallelization of basic BSI arithmetic operations, such as summation or multiplication. These operations and their algorithms on a centralized system are described in [23].

In this work we perform the top-k (preference) queries by executing the following steps:

1. First we apply the set of weights defined as the query preferences Q. These weights are applied in parallel for each dimension.

2. Then we aggregate the summation result across all dimensions in parallel. We propose a two-phase MapReduce algorithm that uses the bit-slice depth for mapping.

    - Map the bit-slices encoding the weighted dimensions to different mappers. The mapping key is the bit-slice depth within the attribute.
    - Aggregate locally the bit-slices by adding together all the bit-slices with the same depth from the same node, and obtain a partial sum BSI for every depth.
    - Complete the computation of the BSI sum by depth by shuffling and aggregating the partial sums.
    - Aggregate all the partial sums BSI by depth into one final aggregated BSI.

3. Finally, we apply the top-k algorithm over the resulting BSI using the algorithms showed in [23], [46]. This can be done on the master node, or in parallel, by splitting the sum attribute horizontally.

The proposed approach exploits the parallelism exhibited by the Bit-Sliced

Index (BSI), and avoids sorting in MapReduce when retrieving top-k tuples. However, implementing a centralized solution to a distributed environment presents unique challenges. For one, the BSI index can be partitioned vertically as well as horizontally. The vertical partitioning can be done not just by splitting the columnar attributes, but also by splitting the bit-slices within one attribute. The horizontal partitioning of the BSI index can be done by segmenting the bit-slices within the BSI.

For the top-k preference queries considered in this work, the top-k query processing algorithm consists of two MapReduce phases. The steps of the top-k preference query are depicted in Figure 6.1.

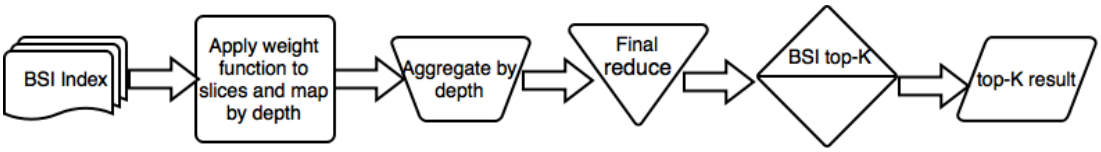


Figure 6.1: Top-k (preference) query stages using the two-phase BSI slice mapping method.

6.2 Distributed Nearest Neighbor queries using bit-vector indexing

6.2.1 K-NN queries using BSI indexing

Given a point in the feature space, the K-NN(nearest neighbor) query returns k closest points from a data set. Since K-NN queries are often used in machine learning for classification purposes, these queries are often run against a training data set. The k points returned by the query are then used to vote on

predicting the class for the point of interest.

Similarly to the top-k preference queries, if no dimensionality reduction is used, most existing data structures and indices used to answer K-NN queries fall into the curse of dimensionality. They become slower than a sequential scan approach. Because the BSI index is often smaller than the raw data, it is worth considering a sequential scan approach on the BSI index to answer K-NN queries for high dimensional data.

The steps required for answering the K-NN query over a BSI index are:

1. Compute the Manhattan distance between the query point and the indexed points.
    - Using the two's complement encoding for the BSI index, compute in parallel for every dimension the difference between the query point and the data points.
    - In parallel, extract the absolute value for the differences computed.
2. Aggregate point distances by executing the SUM operation.
3. Perform the top-k Min operation over the BSI attribute resulting from the previous step.

Data domain characteristics may indicate that certain attributes are more important than others in defining data object similarity. Applying different weights to each attribute can be used to more accurately reflect the real-world similarity.

This can be easily accommodated by including the multiplication by a constant, defined in [23], into the proposed kNN computation.

6.2.2 Distributed K-NN queries using equality bitmap indexing

Another way of answering K-NN queries with the help of bit-vector indices is by computing the Hamming distance [47] between different dimensions of the query and data points. In this case an equality encoded bitmap [48] index would be well suited as the index structure. The Hamming distance can be computed by only accessing one bitmap per dimension (i.e. the bitmap encoding the value corresponding to the query value).

6.2.3 Extended kNN-based queries

kNN over BSI indices can support powerful variations to the baseline similarity search by combining the proposed approach with bitmap index functionality. A number of these query types are described below:

Weighted Similarity Search

Data domain characteristics may indicate that certain attributes are more important than others in defining data object similarity. Applying different weights to each attribute can be used to more accurately reflect the real-world similarity. This can be easily accommodated by including the multiplication by a constant into the proposed kNN computation.

Projected Similarity Search

Bitmaps maintain dimensional independence between attributes. This makes it possible to compute the similarity only over some of the attributes. Therefore, object similarity within a given subspace can easily be computed without any special adaptation. In contrast to others multi-dimensional indexes that do not maintain dimensional independence, where similarity computations can not easily be computed over a subset of the indexed dimensions. For example, if using Locality Sensitive Hashing(LSH), every time a different projection is used, the hash tables have to be re-computed.

Constrained Similarity Search

The proposed approach can be combined with traditional bitmap functionality seamlessly. Bitmaps are naturally suited to efficiently perform selection queries over the attributes. Queries that are a combination of selection over a subset of attributes and similarity over another subset of attributes is easily performed by first performing the selection query to define those points that fall within the constraints (represented by an existence bitmap), and then compute the similarity for this subset of points.

Complex Similarity Search

Complex similarity searches refer to queries with more than one reference point. For example, a query that asks for images similar to a set of multiple examples. BSIs can be used to support such type of queries. The average distance

between the points and each query can be computed independently for each dimension and then added together as the similarity measure. The query result correspond to those objects with the greatest total similarity to the set of query objects.

6.3 Distributed aggregation using the BSI index

The most important step in answering complex queries in a distributed environment is the parallel operation execution result aggregation. There are multiple ways one can implement aggregations in map-reduce, and thus we describe three possible approaches, and then focus on the optimized method that is a two-phase map-reduce algorithm which uses bit-slice depth keys for mapping and then reducing keys.

6.3.1 BSI tree reduction aggregation

The simplest way to implement an aggregation in MapReduce is to use a tree-like reduction to add every pair of BSI attributes in parallel using $\lceil log_2 m \rceil$ reduce rounds, where m is the number of attributes in the dataset. For clarity, in each "round," the reduced output is fed into another iteration of map() and reduce(). The pseudo-code for this simple approach is provided in Algorithm 10, and as can be seen, a few lines of code can achieve this parallelization.

However, let us further evaluate this approach through an example. Consider a dataset with $m = 128$ attributes and a Hadoop cluster with 10 nodes. The aggregation would require 7 reduce rounds. The map tasks are trivial and simply emit the input data to the reduce tasks. The first round requires 64 reduce tasks,

and each subsequent round requires just half of the previous. Since the output of each round is used as the input for the next round, a large amount of data may need to be shuffled between nodes. It is also possible that stragglers or "lazy" nodes can slow down computation. Moreover, as the number of reduce tasks drops below the number of nodes, not all nodes can be used in the computation. Figure 6.2 illustrates an example of tree aggregation and the nodes involved in the aggregation at different stages.

Considering these limitations, we optimize this parallel SUM_BSI to reduce the amount of data shuffled and the number of rounds needed to reduce the tree. We call this optimization the SUM_BSI Group Tree Reduction and describe it in detail in the next subsection.

Algorithm 10: Tree reduction for BSI aggregation.

```
   Reduce():
   begin
       Input: RDD<BSIAttr> pSum1, pSum2
       Output: RDD<BSIAttr> sumAtt
1      sumAtt = pSum1.SUM-BSI(pSum2);
2      return sumAtt
   end
```

6.3.2 BSI group-tree reduction aggregation

To minimize the number of partial BSIs generated and shuffled between nodes, we take advantage of *data locality*. Because each task node typically also

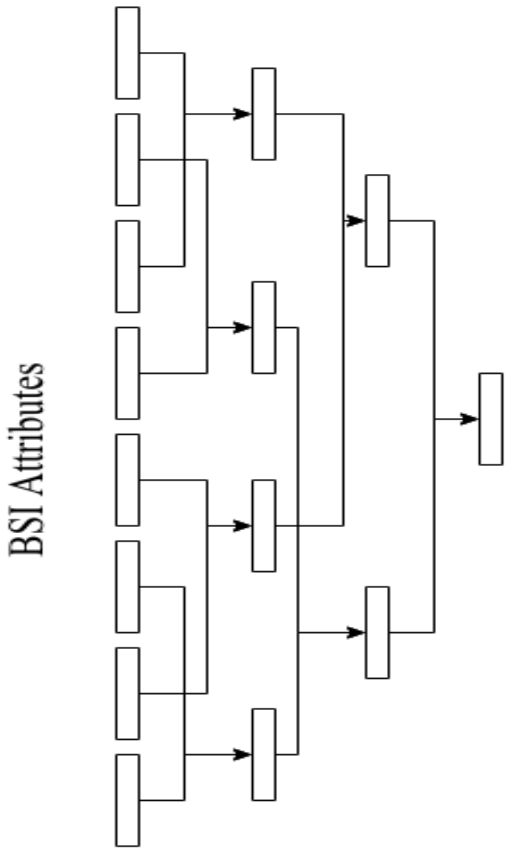


Figure 6.2: Aggregation using a single round tree reduction with the BSI sum operator.

serves as a data-node, we first aggregate a group of BSIs locally within each node. Next, we aggregate the partial results using the previous tree reduction. The size of the group is limited by the number of attributes in a partition and never exceeds it. Figure 6.3 shows the two rounds of aggregation, and how the data shuffle is reduced by "forcing" the first round to be performed on a local level first. The pseudo-code is listed in Algorithm 11

Consider again our previous example of a dataset with $m = 128$ attributes, and a 10 node cluster. If we define groups of size $p = 13$, then each node will add 13 BSIs (except for the last node, which adds 11 BSIs for a total of 128) in parallel.

Algorithm 11: Group tree BSI aggregation.

```
   Map(): //Maps attributes by their assigned executor (location)
   begin
       Input: RDD<BSIAttr> indexAtt //BSIAttr
       Output: RDD<Integer, BSIAttr> byExecutor
1      int executorID= assignExecutor();
2      return new Tuple(executorID, indexAtt)
   end
   ReduceByKey()://Group(local) aggregation
   begin
       Input: RDD<Integer, BSIAttr> byExecutor1, byExecutor2
       Output: RDD<Integer, BSIAttr> pSum
3      pSum = byExecutor1.SUM-BSI(byExecutor2);
4      return pSum
   end
   Map():
   begin
       Input: RDD<Integer, BSIAttr> pSumByExecutor
       Output: RDD<BSIAttr> pSum
5      pSum = pSumByExecutor._2();
6      return pSum
   end
   Reduce()://Final aggregation
   begin
       Input: RDD<BSIAttr> pSum1, pSum2
       Output: RDD<BSIAttr> sumAtt
7      sumAtt = pSum1.SUM-BSI(pSum2);
8      return sumAtt
   end
```

This produces 10 partial BSIs that can be reduced using the tree reduction in just $\lceil \log_2 10 = 4 \rceil$ rounds.

With this approach, the second phase must wait for the results from the first phase before starting. However, by reducing the height of the tree and therefore the network communication cost, we can still reduce the overall execution time. The biggest problem with this algorithm is the lack of load balancing. The number

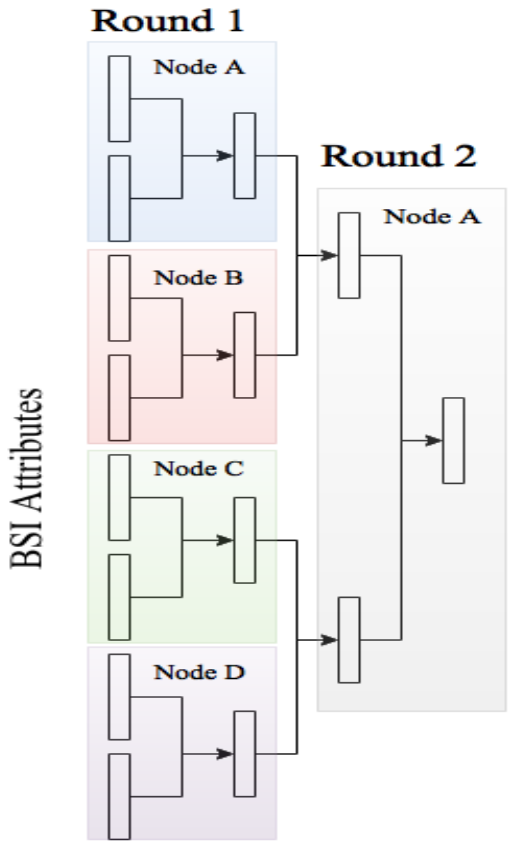


Figure 6.3: Aggregation using a two round group-tree reduction with the BSI sum operator.

of slices used for each attribute needs not to be the same, which in turn translates into variable execution times for adding a group of BSI attributes. Furthermore, with smaller numbers of partial products (*e.g.* one for each executor), not all the nodes are busy during the tree reduction.

In order to achieve load balancing, we should exploit the fact that each bit-slice is stored column-wise and make the *bit-slices*, not the attributes, the working units. Our proposed approach using the bit-slice depth as the mapping key is described next.

6.4 BSI two-phase slice mapping for distributed aggregations

It is true that the compact representation of the BSI makes the algorithms described in the previous two sections highly competitive versus their array counterparts. However, most of their performance gains, if not all, come from the reduced size of the BSI, and not necessarily because the algorithms are efficient. In this section, we propose a aggregation algorithm that promotes the bit-slices as the processing data units and applies the lessons-learned in computer arithmetic optimization to further improve the performance of the parallel aggregation. The basic idea of this approach lies in use the bit-slice depth as the mapped key and implement a two-phase algorithm, shown in Figure 6.4. In the first phase, the slices are added by bit-depth, producing a weighted partial sum BSI. In the second phase, all the partial sums are added together in a method similar to a carry-save adder.

Consider again our running example where $m = 128$ attributes are added using 10-nodes. Let us now assume that each attribute's value is within $1M = 2^{20}$, so every attribute i can be further partitioned into a set of 20 vertical bit-slices: $\{B_i[d] \mid 19 \geq d \geq 0\}$. In the proposed two-phase algorithm, the first task is to map all the bit-slices with the same depth (d) to a single node. Then addition is performed over 128 BSIs containing only 1 slice each, producing 20 partial sum BSIs. Each partial sum is in the range $[0, 128]$ and would require at most 8 slices. Next, these partial sums are added using their original depth d as their "weight." For example, the partial sum for the bit-slices of depth $d = 2$ would have a weight of $2^d = 4$. Because the weight is always a power a 2, this weighting scheme can

be done efficiently by bit-shifting. Since the BSIs are stored column-wise, this shift can be represented using an offset and never materialized.

It is also possible to perform the parallel aggregation using groups of bit-slices to reduce data shuffling. In the previous example, with a group size of $g = 2$, we could have slices 0 and 1 from all 128 attributes added together in the same node during the first stage. This ability to group the slices and divide the attributes (*e.g.*, half of the depth 0 slices added in one node and the other half in another), allows us to balance the load and keep all the nodes busy longer. In the remainder of this section, we formalize the proposed two-phase algorithm and analyze its cost in Section 6.5.

For clarity in describing our algorithms, we use the example illustrated in Figure 6.4. In the first MapReduce phase, every BSI attribute has its slices mapped locally to different mappers based on their depth d. The splitting of the BSI attribute in individual bit-slices allows for a finer granularity of the indexed data and for a more efficient parallelism during the aggregation phase. The pseudo-code of the mapping step is shown in the first Map() function of Algorithm 12. Every mapper has a $BSIAttr$ (containing multiple slices) as input, and outputs a set of $BSIAttr$s that contain *one* bit-slice each. These bit slices are mapped by their depth in the input $BSIAttr$. Although there is an overhead associated with encapsulating each bit-slice into a $BSIAttr$, by creating a higher level of parallelism, we also achieve better load balancing and resource utilization.

Still in the first phase, the aggregation is done by the ReduceByKey() func-

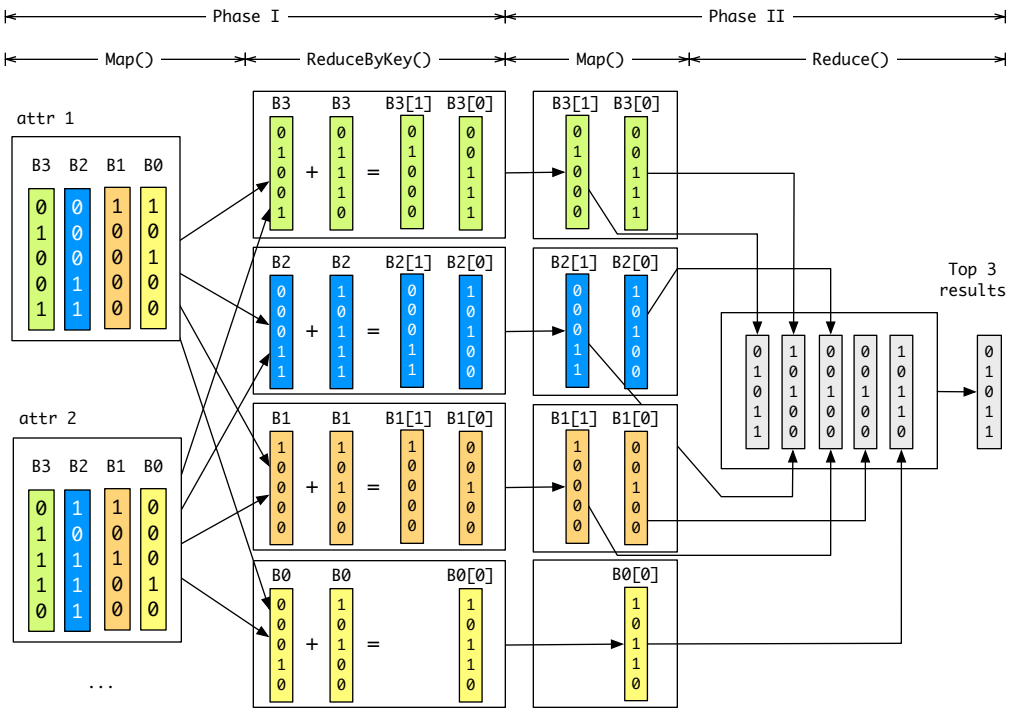


Figure 6.4: SUM_BSI Using Slice Mapping Example.

tion of Algorithm 12. In this step, all the bit-slices with the same key (depth) are aggregated into a $BSIAttr$. Line 9 of Algorithm 12 performs the summation of two BSIs. We use the same addition logic as the authors in [23]. However, we achieve a parallelization of the BSI summation algorithm by splitting the $BSIAttr$ into individual slices and executing their addition in parallel similarly to a carry-save adder. The offset of the resulting $BSIAttr$s are saved in the $offset$ field of each $BSIAttr$ object to ensure the correctness of the final aggregated result. Apache Spark optimizes the summation by aggregating the bit-slices on the same node first, then on the same rack, and then across the network. Thus, trying to minimize the network throughput. The aggregation by depth is done locally first.

After aggregating partial local results, the second MapReduce phase initiates to complete the aggregation by depth through shuffling the partial sums and reducing by their depths. The final step of the aggregation is done by reducing all the BSIs ($pSum$) produced in the previous ReduceByKey() stage, regardless of their key. The final result ($attSum$) of this reduce phase is a single BSI attribute in the case of vertical only partitioning, or a set of BSI attributes, that should be concatenated, in the case of vertical and horizontal partitioning. Concatenation is straight forward, as each BSI in a partition has the same number of bits corresponding to the same rowIds.

6.5 Cost estimations for two-phase map-reduce BSI aggregation

As showed in the work that uses the BSI index for top-K queries on single machines [44], the query time is dominated by aggregation. Thus, in this section we focus on estimating the complexity of the two-phase map-reduce aggregation, and the amount of data shuffling that it generates in a distributed environment. In this section we estimate the network data shuffle, and time complexity per horizontal partition. This estimation should be applied for each horizontal partition.

6.5.1 Data shuffle estimation

The mapping in the first phase (Figure 6.4), does not produce any shuffling since it aggregates only the slices from attributes found on the same node.

The mapping in the first phase (Figure 6.4), does not produce any shuffling since it aggregates only the slices from attributes found on the same node. Data

Algorithm 12: Two phase distributed BSI aggregation by slice depth.

```
   Map(): //Map slices by depth
   begin
       Input: RDD<BSIAttr> indexAtt
       Output: RDD<Integer, BSIAttr> byDepth
1      int sliceDepth=0;
2      while indexAtt has more slices do
3          bsi = new BSIAttr();
4          bsi.add(indexAtt.nextSlice());
5          byDepth.add(new Tuple(sliceDepth, bsi));
6          sliceDepth++;
7      end
8      return byDepth
   end
   ReduceByKey()://Reduce by depth - first reduce phase
   begin
       Input: RDD<Integer, BSIAttr> byDepth1, byDepth2
       Output: RDD<Integer, BSIAttr> pSum
9      pSum = byDepth1.SUM-BSI(byDepth2);
10     return pSum
   end
   Map():
   begin
       Input: RDD<Integer, BSIAttr> partSum
       Output: RDD<BSIAttr> pSum
11     pSum = partSum._2();
12     return pSum
   end
   Reduce(): //Second reduce phase
   begin
       Input: RDD<BSIAttr> pSum1, pSum2
       Output: RDD<BSIAttr> sumAtt
13     sumAtt = pSum1.SUM-BSI(pSum2);
14     return sumAtt
   end
```

shuffling occurs twice in our two-phase MapReduce aggregation. The first time is between the reducers of phase 1 and the mappers of the phase 2, and the second time data is shuffled between the mappers and reducers of the second phase. The amount of data shuffled depends on the number of nodes, partitions, tasks (or the number of attributes per task), and the number of slices per group. The number of slices per group can vary from 1 to s, where s is the highest number of slices per attribute in the dataset. In Figure 6.4 the slices are mapped into groups of one.

In order to determine the amount of data shuffled between the reducers of phase 1 and the mappers of phase 2, we should find first the number of outputs created by the reducers of phase 1. Given m attributes with s maximum slices per attribute, a attributes per node, and g slices per group, each node produces $\frac{s}{g}$ partial aggregations by depth. The size of each of these partial aggregations is in the worst case:

$$\lceil \log_2(g+a) \rceil \tag{6.1}$$

This represents the number of slices each partial aggregation by depth contains after the reduce phase 1. The total number of slices shuffled at this stage is:

$$Sh_1 = \left[\left(\min\left(\left\lceil \frac{s}{g} \right\rceil, \left\lceil \frac{m}{a} \right\rceil \right) - 1 \right] \cdot \left\lceil \frac{m}{a} \right\rceil \cdot \lceil \log_2(g+a) \rceil \tag{6.2}$$

The mappers of the second phase produce $\frac{s}{g}$ outputs, each with the size:

$$\lceil \log_2(g+a) \rceil + \left\lceil \log_2\left(\frac{m}{a}\right) \right\rceil = \left\lceil \log_2 \frac{(g+a)m}{a} \right\rceil \tag{6.3}$$

The total number of slices shuffled between the mappers and reducers of the second phase is:

$$Sh_2 = \left(\left\lceil \frac{s}{g} \right\rceil - 1\right)\left\lceil \log_2 \frac{(g+a)m}{a} \right\rceil \tag{6.4}$$

The total amount of data shuffled is the sum of the results from Equations 6.2 and 6.4:

$$Sh = Sh_1 + Sh_2. \tag{6.5}$$

The size of each bit-slice is given by the number of rows in each horizontal partition. If compression is applied, then bit-vector size estimation techniques such as in [43] should be considered in estimating the size of the bit-slices.

6.5.2 Time Complexity Analysis

Based on our estimations from the previous sub-section, the amount of data shuffled decreases as g - the number of slices per group increases, or as a - the number of attributes per node increases. However, less data shuffling means a higher load on individual tasks. We further analyze the time complexity for each individual task, and its impact on the total query time in the two-phase MapReduce aggregation.

The cost of summing two BSI attributes is linear on the number of slices and the number of rows in the attributes. If p is the number of slices of the attribute with a higher number of slices, then the cost of adding the two attributes is equal to the cost of executing p bitwise logical operations between two vectors. Given that the number of slices per group is a constant, g is the number of slices for

each depth-shifted attribute in the reduce phase 1. Adding all the depth-shifted attributes within one node has the following complexity:

$$T_1 = \sum_{i=1}^{\log_2 a} (g + i). \tag{6.6}$$

There are $\frac{m}{a}$ partial sums with the same key per task, to complete the aggregation of partial sums shifted by depth. Thus the cost of this aggregation is:

$$T_2 = \sum_{i=1}^{\lceil \log_2 m/a \rceil} (g + \lceil \log_2 a \rceil + i) \tag{6.7}$$

Finally, the cost of aggregating the partial sums shifted by depth into one final attribute, is given by:

$$T_3 = \sum_{i=1}^{\lceil \log_2 s/g \rceil} \left(g + \lceil \log_2 a \rceil + \left\lceil \log_2 \frac{m}{a} \right\rceil + i \right) \tag{6.8}$$

When taking into consideration the time complexities from Equations 6.6, 6.7, and 6.8, one must account for the different number of tasks executed in these three steps. For example, if T_1 has a weight of one, i.e. $W_{T_1} = 1$, then the number of tasks for T_2 and T_3 is different. For $W_{T_1} = 1$, the weight for T_2 is:

$$W_{T_2} = \frac{1}{\lceil \frac{m}{a} \rceil} \tag{6.9}$$

since there are fewer tasks for T_2 than T_1 by a factor of $\frac{m}{a}$. While the weight

for T_3 is:

$$W_{T_3} = \frac{1}{\lceil \frac{m}{a} \rceil \lceil \frac{s}{g} \rceil} \tag{6.10}$$

In this case, there are s/g fewer tasks than in the previous step.

Using the time complexities discussed above, together with the data shuffle estimations, it is possible to find the optimum values for the number of slices per group (g) and the number of initial tasks/attributes per task.

6.6 Evaluation of the two-phase map-reduce distributed BSI aggregation method

6.6.1 Scalability of the proposed indexing and querying approach

We test the scalability of the proposed two-phase slice-mapping for aggregation (Slice BSI) in terms of query time as data dimensionality and the number of computing nodes/CPU cores increases, as well as for increasing data cardinality.

Given that the BSI index is sensitive to data cardinality, we set up to measure the scalability of the two-phase slice-mapping aggregation algorithm when compared to the BSI tree-reduction and the two-round BSI tree-reduction (BSI group) methods. We also compare with a map-reduce method for aggregation implemented in Spark, that is similar to the Tree BSI without using the BSI index (Spark). Figure 6.5 shows the query times for sum aggregations when varying the data cardinality from 10^3 to 10^18, using from 10 to 64 bit-slices per attribute. As the figure shows, the Slice BSI method is up to two times faster than the other BSI methods, and up to 20X faster than the non BSI method for lower cardinality. The

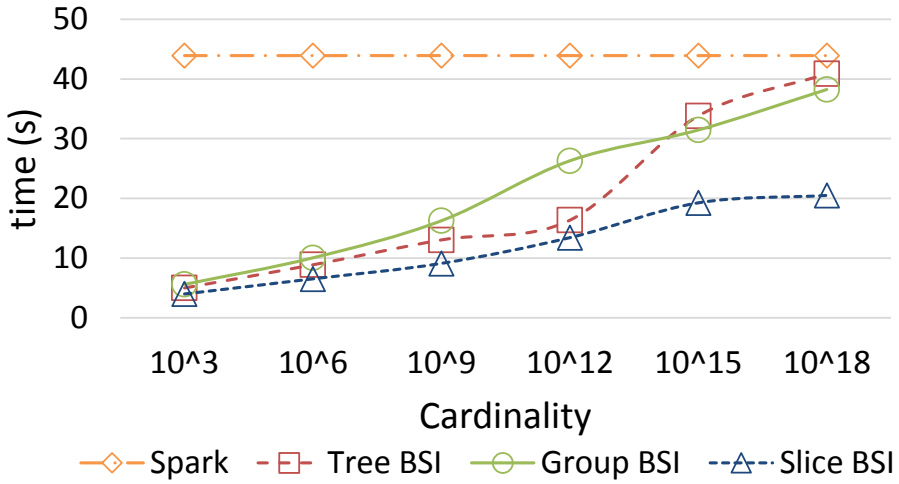


Figure 6.5: Aggregation query time with increasing data cardinality.(Synthetic dataset, uniformly distributed data over 260 attributes and 5 Million rows).

non BSI method is not sensitive to data cardinality.

Figure 6.6 shows the performance of the two-phase map-reduce for aggregations (Slice BSI) method against the two other BSI methods and the non-BSI method when increasing the number of rows. Slice BSI consistently outperforms the other approaches. However, another useful insight that can be collected from this experiment is that the time per row starts to increase once we reached 8 Million rows per partition (2.2 s/1M rows for 8M rows per partition vs. 1.9s/1M rows for 4M rows per partition). A potential explanation could be the limited CPU cache size. Partition size should be determined by considering the CPU cache size on the cluster, which should be subject to future research.

Figure 6.7 shows the Slice BSI aggregation times over the Rainfall[1] dataset as the number of dimensions increases from 2,000 to 8,758. The results are shown

[1]http://www.emc.ncep.noaa.gov/mmb/ylin/pcpanl/stage4/

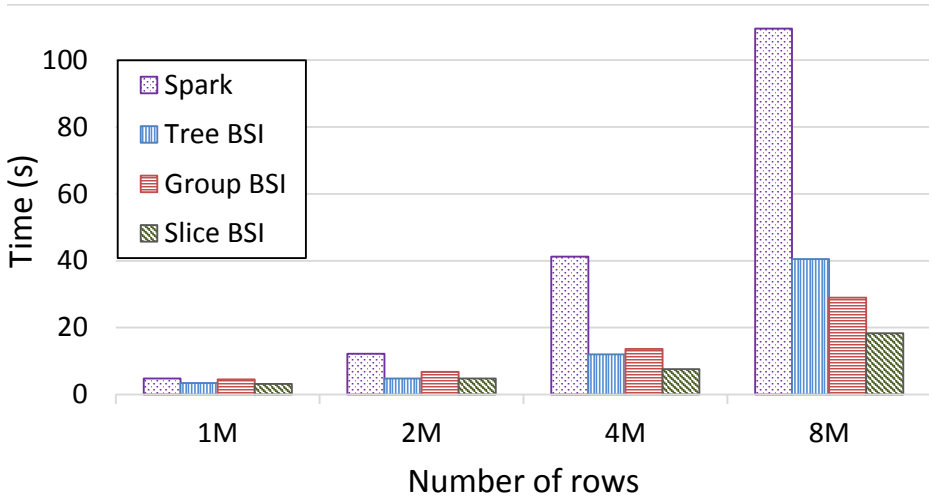


Figure 6.6: Aggregation performance for varying the number of rows. (Synthetic dataset, uniformly distributed data over 260 attributes with cardinality 10^{12}).

for 12, 24, 36, and 48 CPU cores allocated by the resource manager. Considering the available hardware infrastructure, we increase the number of CPU cores by 12 at a time (each datanode features 12 CPU cores). For this experiment we used a number of 25 bit-slices per dimension. Figure 6.7 shows good scalability of the two-phase slice-mapping algorithm, in terms of increasing data dimensionality, and in terms of increasing the number of CPU cores.

The Slice BSI method improves on the other two BSI methods by balancing the task complexity and the network communication. In the next section we further investigate how the task granularity/partition size impacts the query time.

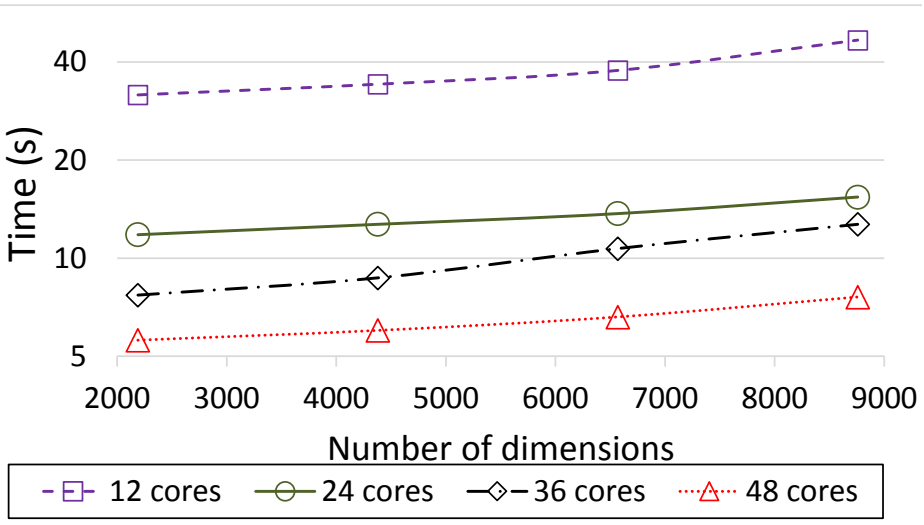


Figure 6.7: Aggregation using the Slice BSI method when varying the number of dimensions and the number of executors(cpu cores) over the Rainfall dataset (Dataset: Rainfall, 25 bit-slices per dimension).

6.6.2 Evaluation of cost estimations

Evaluation of data shuffle estimations

To evaluate the data shuffle and aggregation time estimations described in the previous sections, we use the Rainfall dataset with one horizontal partition. In the case of multiple horizontal partitions, the estimations should be applied to each horizontal partition. The size of each compressed bit-slice was computed using the estimations described by [43], in the section referring to EWAH/WBC. The sizes for the verbatim bit-slices is simply the number of bits given by the number of rows.

Figure 6.8 shows the estimated data shuffled in MB and the measured data shuffled when performing sum aggregations over the rainfall data. These measurements are shown for different values of g - slices per group. The BSI index

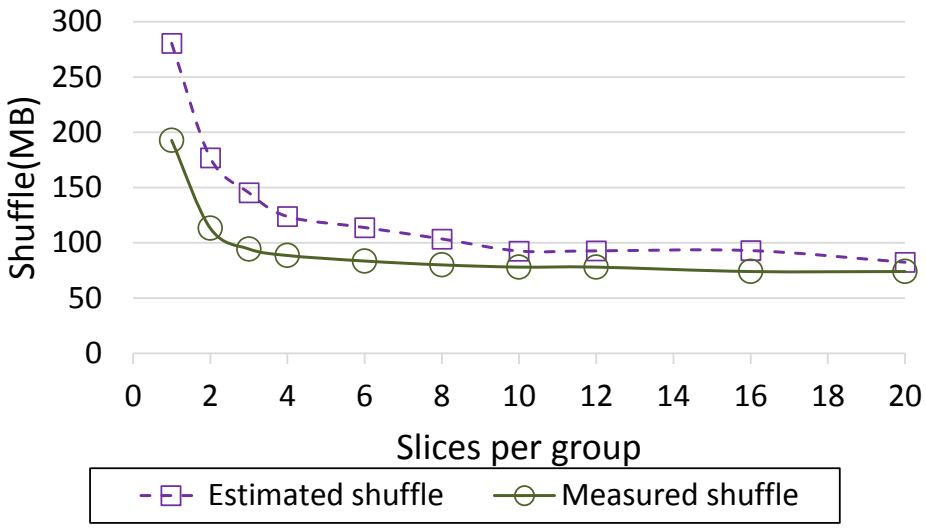


Figure 6.8: Estimated data shuffle compared to measured data shuffle, for the two phase aggregation method. (Dataset: Rainfall data, 20 slices per attribute, index size: 9.8 GB, index partitions: 94).

was partitioned into 94 vertical partitions. Both, the estimated data shuffle and the measured data shuffle, present the same patterns when increasing g. As expected, the estimated values are higher given that the equations described earlier reflect the upper bounds for data shuffling.

Evaluation of query time estimations

In this section we estimate the aggregation times in the two-phase slice BSI map-reduce method, when varying the number of bit-slices per group (g). The purpose of this experiment is to validate our estimations from earlier, and to develop a technique for finding the optimum value for g, prior to running any queries.

The exact query times depend on the hardware deployed for running the two-phase map-reduce method for aggregation. Thus in this section we normalize

the estimated and measured query times to present them on a scale from 0 to 1, using the following equation:

$$Norm(e_i) = \frac{e_i - E_{min}}{E_{max} - E_{min}}, \tag{6.11}$$

where E_{min} is the minimum value for series E and E_{max} is the maximum value for series E.

There is a trade-off between parallelism and network communication when executing this aggregation. Thus we estimate the query time based on both, data shuffle estimations and the time complexity estimations per task described in Section 6.5.

$$Time = Norm(Sh_i) + Norm(T_i), \tag{6.12}$$

where

$$T = W_{T_1} T_1 + W_{T_2} T_2 + W_{T_3} T_3. \tag{6.13}$$

Sh, W_{T_1}, T_1, W_{T_2}, T_2, W_{T_3}, and T_3 are defined in Section 6.5.

The results of this experiment are shown in Figure 6.9. In this case the optimum number of slices per group during the map-reduce phase is four, and the estimation also indicated that 4 slices per group would give the fastest result. It is worth noting that the estimated time does not consider the overhead associated with map-reduce scheduling and synchronization. Our goal is not to predict exe-

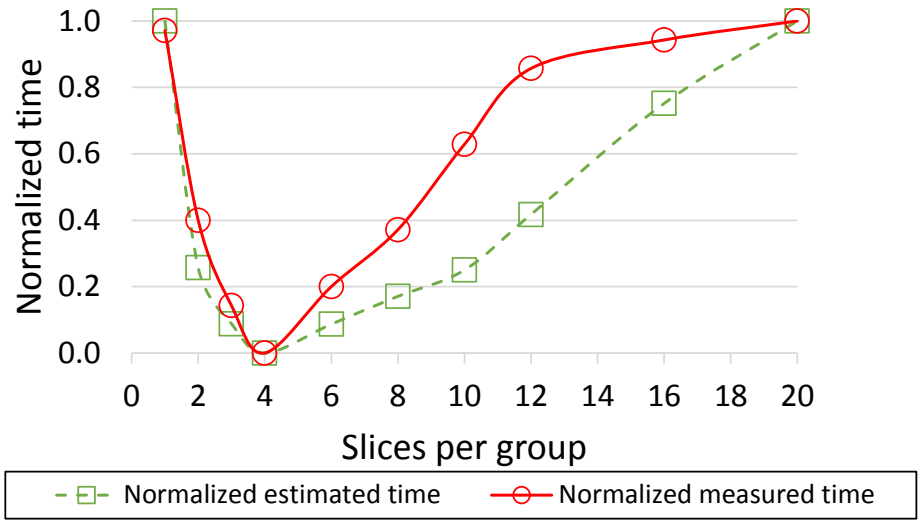


Figure 6.9: Estimated execution time compared to measured execution time, for the two-phase slice-mapping aggregation method. (Dataset: Rainfall data, 20 slices per attribute, index size: 9.8 GB, index partitions: 94).

cution time, but rather decide on the values of the run-time parameters that would offer the best trade-off between parallelism and network bandwidth.

In addition to the number of slices per group, using these estimations one can also tune a, the number of attributes per partition. By controlling the number and the size of partitions, one can increase or decrease the network communication, and also the load on each individual task.

6.6.3 Comparison against existing distributed data stores

To validate the effectiveness of the proposed index and query algorithms, we compare our query times against SparkSQL and Hive on Tez, and Hive on Hadoop Map-Reduce. We performed the top-k non-weighted preference query over the HIGGS dataset 5 times and averaged the query times. We also gener-

ated 5 random sets of weights with a 3 decimal precision and another 5 random sets of weights with a 6 decimal precision. We ran these queries on the proposed distributed BSI index, Hive on Hadoop MapReduce (MR), and Hive on Tez. We loaded the HIGGS dataset attribute values into Hive tables as float numbers. For the BSI index, we used 32 bit-slices per attribute to have a fair comparison against Hive and SparkSQL.

The query ran on Hive has the following syntax:

```
SELECT RowID,
(column1*weight1 +...+ columnN*weightN)
as 'AttributeSUM'  FROM table
ORDER BY AttributeSUM DESC LIMIT k
```

Figure 6.10 shows the query times of these top-k weighted and non-weighted queries against SparkSQL, Hive on Hadoop MR, Hive on Tez and the distributed BSI index. The results show that the BSI on Spark was 25 times faster than SparkSQL and one order of magnitude faster than Hive.

On this experiment, SparkSQL is slower than Hive on Hadoop MR and Hive on Tez because of some inefficiencies in its shuffle phase, one of which is the lack of shuffle file consolidation, as described in [49]. However, SparkSQL should see some improvements in the more recent releases. It is worth noting that BSI on Spark and SparkSQL use the same computation engine for distributing the work across the cluster.

We have laid the foundation for a distributed query engine using bitmap-based indices. Our goal is to further extend the BSI arithmetic using distributed algorithms to support more operations and to provide accurate estimations of their

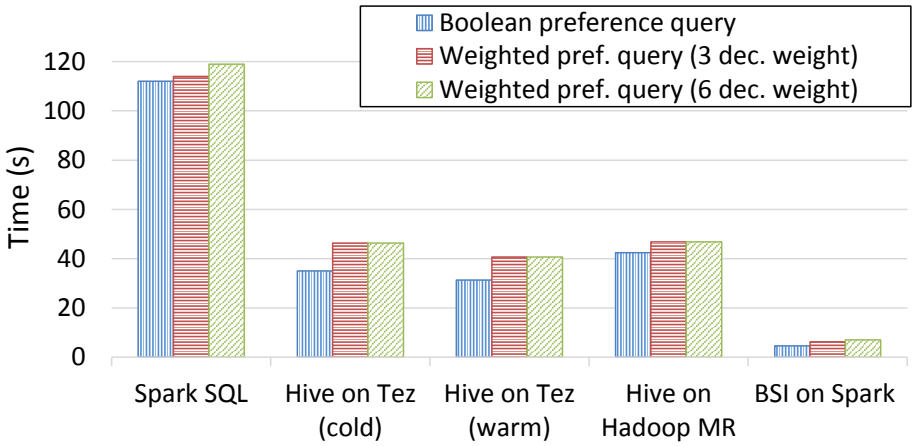


Figure 6.10: BSI top-K preference and top-K weighted preference query time using BSI slice-mapping compared to Hive on Hadoop map-reduce, and Hive on Tez (Dataset: HIGGS, 32 bit-slices per dimension).

cost. Many other types of queries such as constraints top k, skyline queries, and predictive queries can be supported as well.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

In this work we propose a distributed indexing and scalable query processing system for interactive Big Data explorations. Our indexing and query processing layer is integrated with an open-source cluster computing engine and is designed to handle large amounts of data. For indexing, we support traditional bitmap indices as well as an enhanced BSI (bit-sliced index). Queries are executed by operating vectors of bits in parallel.

For sparse bitmaps it is crucial to apply compression, thus we conduct a survey of the most popular bitmap compression methods. We propose a framework for estimating the performance and choosing the best compression method given a particular dataset. We improve existing word-aligned compression methods by enabling compression using variable aligned lengths (VAL). In our experiments we show that VAL can improve compression by up to 1.8 times and query times by up to 30%.

However, the VAL framework was not able to improve the performance of high-density bit-vectors such as the bit-sliced indices. The BSI index being a high density bit-vector index, typically performs better without compression. Nonetheless, during the execution of complex data exploratory queries, when the query answer is typically a very small portion of the data, the intermediate bit-vectors become sparse and compressible. Thus, exploiting these opportunities for compression can benefit the overall query speed and hardware resource utilization.

With this in mind, we propose a hybrid compression framework that can improve execution time of aggregations for preference queries over BSIs about 10%, and with up to 11 times less memory utilization when compared to non-compressed bit-vectors. In situations when memory availability is limited, the query times can degrade substantially without using compression by producing many more disk accesses.

In high-dimensional spaces, where traditional indexing and algorithms for processing complex queries such as preference or nearest neighbor queries, fail to scale and become slower than sequential scan approaches, we showed that by using BSI indices one can outperform sequential scan due to a smaller index size and a more efficient CPU utilization. Moreover, because the BSI indexes each dimension independently, they are scalable for high-dimensional data, and thus suitable for distributed settings.

We describe the structure of a distributed BSI index, that can be partitioned horizontally as well as vertically. In conjunction with bitmap vectors, used to filter data points, we describe how the distributed BSI can be used to answer complex queries extensively used in big data analytics applications. Because efficient execution of parallel BSI arithmetic operations and result aggregation is crucial, we extend the BSI index to support parallel operations and aggregations. Using a two-phase MapReduce algorithm for distributed arithmetic and result aggregations, together with fine control over the task sizes and network data shuffling, we aim for more efficient hardware resource utilization and results in faster processing times

for queries that seek to access to large amounts of data.

In a distributed computing system, there is always a tradeoff between parallelism and network communications. We analyze the costs for distributed operations such as summations over the BSI index. The costs are defined in terms of total CPU cost(level of parallelism) and network communication depending on partition sizes and cluster hardware availability. We assume equal weights for these two cost measures and optimize the partition size and group bit-slices into partial BSI attributes that do not exceed the defined partition size. Depending on the execution environment, the user can modify the weights for network communication or level of parallelism to achieve optimum performance.

Our contributions include important extensions to the bit-slices index including partitioning and distributed query algorithms. Our preliminary results show that the proposed approach outperforms Hive, a map-reduce based data warehouse, over Hadoop Map-Reduce and the optimized query engine Tez, by at least an order of magnitude in terms of execution time for preference queries. Furthermore, experiments show a 25 times execution time improvement over Spark-SQL, which uses the same distributed computation engine as the proposed approach.

As future work, the bit-vector based indices, have the potential to support more types of queries used in data analysis and exploration applications, such as constrained top-K queries, skyline queries, collaborative filtering, and others. The library of supported operations can be expanded.

We investigated and optimized the partitioning of the distributed BSI index in terms of tradeoff between parallelism and network communication. A further improvement in partitioning could potentially be considering the local CPU cache size to determine more efficient horizontal partition sizes.

We showed that using compressed bit-vector indices, not only improves the overall query speed, but also requires fewer computational resources. With this in mind, there is further need for researching the energy usage of our algorithms. Given our results, the intuition would be that using bit-vector indices can reduce the electricity usage when compared to conventional methods used in most cloud applications. Thus the proposed indexing and query processing layer in this dissertation has the potential to reduce the power consumption in data centers that specialize in mining of very large data.

REFERENCES

[1] Abhinandan S. Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: Scalable online collaborative filtering. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 271–280, New York, NY, USA, 2007. ACM.

[2] Vivien Marx. Biology: The big challenges of big data. *Nature*, 498(7453):255–260, 2013.

[3] Chris A Mattmann. Computing: A vision for data science. *Nature*, 493(7433):473–475, 2013.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[6] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[8] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[9] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, et al. F1: A distributed sql database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, 2013.

[10] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of*

the 2015 ACM SIGMOD International Conference on Management of Data, pages 1383–1394. ACM, 2015.

[11] Ting Liu and Margaret Martonosi. Impala: A middleware system for managing autonomic, parallel sensor systems. In *ACM SIGPLAN Notices*, volume 38, pages 107–118. ACM, 2003.

[12] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 5–5. ACM, 2009.

[13] Patrick E O'Neil. Model 204 architecture and performance. In *High Performance Transaction Systems*, pages 39–59. Springer, 1989.

[14] Patrick O'Neil and Goetz Graefe. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record*, 24(3):8–11, 1995.

[15] P.E. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 38–49. ACM Press, 1997.

[16] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, volume 98, pages 194–205, 1998.

[17] Ratko Orlandic, Jack Lukaszuk, and Craig Swietlik. The design of a retrieval technique for high-dimensional data on tertiary storage. *ACM SIGMOD Record*, 31(2):15–21, 2002.

[18] Azza Abouzied, Kamil Bajda-Pawlikowski, Jiewen Huang, Daniel J Abadi, and Avi Silberschatz. Hadoopdb in action: building real world applications. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1111–1114. ACM, 2010.

[19] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[20] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Sys-*

tems Design and Implementation, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[21] Chee-Yong Chan and Yannis E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, SIGMOD '99, pages 215–226, New York, NY, USA, 1999. ACM.

[22] Nick Koudas. Space efficient bitmap indexing. In *Proceedings of the Ninth International Conference on Information and Knowledge Management*, CIKM '00, pages 194–201, New York, NY, USA, 2000. ACM.

[23] Denis Rinfret, Patrick O'Neil, and Elizabeth O'Neil. Bit-sliced index arithmetic. In *ACM SIGMOD Record*, volume 30, pages 47–57. ACM, 2001.

[24] Ming-Chuan Wu and Alejandro P. Buchmann. Encoded bitmap indexing for data warehouses. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 220–230, Washington, DC, USA, 1998. IEEE Computer Society.

[25] G. Antoshenkov. Byte-aligned bitmap compression. In *DCC '95: Proceedings of the Conference on Data Compression*, page 476, Washington, DC, USA, 1995. IEEE Computer Society.

[26] K. Wu, E. J. Otoo, and A.Shoshani. Compressing bitmap indexes for faster search operations. In *Proceedings of the 2002 International Conference on Scientific and Statistical Database Management Conference (SSDBM'02)*, pages 99–108, 2002.

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

[28] F. Deliege and T. Pederson. Position list word aligned hybrid: Optimizing space and performance for compressed bitmaps. In *Proceedings of the 2010 International Conference on Extending Database Technology (EDBT'10)*, pages 228–239, 2010.

[29] Francesco Fusco, Marc Ph. Stoecklin, and Michail Vlachos. Net-fli: On-the-fly compression, archiving and indexing of streaming network traffic. *Proceedings of the VLDB Endowment*, 3(2):1382–1393, 2010.

[30] Alessandro Colantonio and Roberto Di Pietro. Concise: Compressed 'n' composable integer set. *Information Processing Letters*, 110(16):644–650, 2010.

[31] David Chiu Fabian Corrales and Jason Sawin. Variable length compresssion for bitmap indices. In *ACM International Conference on Database and Expert Systems Applications*, pages 381–395, 2011.

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

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

[34] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[35] Yuanyuan Tian, Tao Zou, Fatma Özcan, Romulo Goncalves, and Hamid Pirahesh. Joins for hybrid warehouses: Exploiting massive parallelism in hadoop and enterprise data warehouses. pages 373–384, 2015.

[36] Peng Lu, Sai Wu, Lidan Shou, and Kian-Lee Tan. An efficient and compact indexing scheme for large-scale data store. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 326–337. IEEE, 2013.

[37] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proceedings of the VLDB Endowment*, 3(1-2):515–529, 2010.

[38] Gheorghi Guzun and Guadalupe Canahuate. Hybrid query optimization for hard-to-compress bit-vectors. *The VLDB Journal*, pages 1–16, 2015.

[39] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.

[40] Yinan Li and Jignesh M Patel. Bitweaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 289–300. ACM, 2013.

[41] Gheorghi Guzun, Guadalupe Canahuate, Dereck Chiu, and Jason Sawin. A tunable compression framework for bitmap indices. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 484–495. IEEE, 2014.

[42] Daniel Lemire, Ownen Kaser, and Kamel Aouiche. Sorting improves word-aligned bitmap indexes. *Data and Knowledge Engineering*, 69:3–28, 2010.

[43] Gheorghi Guzun and Guadalupe Canahuate. Performance evaluation of word-aligned compression methods for bitmap indices. *Knowledge and Information Systems*, pages 1–28, 2015.

[44] Gheorghi Guzun, Joel Tosado, and Guadalupe Canahuate. Slicing the dimensionality: Top-k query processing for high-dimensional spaces. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XIV*, pages 26–50. Springer, 2014.

[45] Wen Jin and Jignesh M. Patel. Efficient and generic evaluation of ranked queries. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 601–612, New York, NY, USA, 2011. ACM.

[46] Denis Rinfret. Answering preference queries with bit-sliced index arithmetic. In *Proceedings of the 2008 C 3 S 2 E conference*, pages 173–185. ACM, 2008.

[47] Richard W Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[48] Chee-Yong Chan and Yannis E. Ioannidis. An efficient bitmap encoding scheme for selection queries. *SIGMOD Rec.*, 28(2):215–226, June 1999.

[49] Aaron Davidson and Andrew Or. Optimizing shuffle performance in spark. *University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep*, 2013.