A Two-Phase MapReduce Algorithm for Scalable Preference Queries over High-Dimensional Data

Gheorghi Guzun
Electrical and Computer Engineering
The University of Iowa

Guadalupe Canahuate
Electrical and Computer Engineering
The University of Iowa

David Chiu
Mathematics and Computer Science
University of Puget Sound

ABSTRACT

Preference (top-k) queries play a key role in modern data analytics tasks. Top-k techniques rely on ranking functions in order to determine an overall score for each of the objects across all the relevant attributes being examined. This ranking function is provided by the user at query time, or generated for a particular user by a personalized search engine which prevents the pre-computation of the global scores. Executing this type of queries is particularly challenging for high-dimensional data. Recently, bit-sliced indices (BSI) were proposed to answer these high-dimensional preference queries efficiently in a centralized environment.

As MapReduce and key-value stores proliferate as the preferred methods for analyzing big data, we set up to evaluate the performance of BSI in a distributed environment, in terms of index size, network traffic, and execution time of preference (top-k) queries over high-dimensional data. We implemented three MapReduce algorithms for processing aggregations and top-k queries over the BSI index: a baseline algorithm using a tree reduction of the slices, a group-slice algorithm, and an optimized two-phase algorithm that uses bit-slice mapping. The implementations are on top of Apache Spark using vertical and horizontal data partitioning. The bit-slice mapping approach is shown to outperform the baseline map-reduce implementations by virtue of using a reduced size index and by featuring a better control over task granularity and load balancing.

1. INTRODUCTION

The MapReduce framework [6] has proliferated due to the ease of use and encapsulation of the underlying distributed environment when executing big data analytical tasks. The open source software framework, Hadoop [25], implements MapReduce and provides developers the ability to easily leverage it in order to exploit any parallelizable computing tasks. It has further gained ground from the SQL-like interface on Hadoop such as Hive [24]. Nevertheless, MapReduce does have its shortcomings in handling SQL queries, as was indicated in [7]. Chief among these shortcomings is that it does not innately optimize an important database operator of interest, *top-k selection*. An efficient exact top-k approach optimized on MapReduce is still an open research question.

Top-k selection queries are ubiquitous in big data analytics. Intuitively, the top-k (preference) query refers to applying a monotonic scoring function S over the m attributes a_i (attributes of interest to the query) to each tuple under consideration. A weighted-sum function is a common scoring function, where a weight w_i is factored into each attribute: $S = \sum_{i=1}^{m} w_i a_i$. The weights are given by the user at query time, and the k tuples producing the highest S scores are returned to the user.

Given a distributed system where different subsystems control and store the data, and a query asking for the top k objects most similar to a query object, each subsystem computes a similarity score and the scores are then combined (weighted or not) to obtain the final answer. In the domain of information retrieval, consider a search engine tasked in retrieving the top-k results from various sources. In this scenario, the ranking function considers lists of scores based on word-based measurements, as well as hyperlink analysis, traffic analysis, user feedback data, among others, to formulate the top-k result [16, 21]. Moreover, since the size of many of these lists is large, they are distributed in a key-value store and processed using the MapReduce paradigm.

Top-k processing is a crucial requirement across several applications or domains. In this work we provide a solution for retrieving top-k results over high dimensional partitioned data. For column-stores, sets of attributes or dimensions are stored in different nodes. Our work indexes each partition independently and uses a novel distributed bit-slicing index arithmetic to combine partial results into a global answer to the query. Applications where time is considered as a dimension can also benefit from our work as it is the case for temporal sensor collected data. Consider for example weather data, where information from gauges and other sources are collected at regular intervals. There could be thousands or millions of these data sources distributed over a geographical area. Later in this paper we present results for top k queries over rainfall data collected in the United States using a $4\ km \times 4\ km$-resolution grid. The query is interested in the top k areas (grid cells) with the largest rainfall for a given day or month. This is a high-dimensional query will require the aggregation of $288 (= 24 \times 12)$ attributes or $8640 (= 30 \times 24 \times 12)$ attributes for a day and a month, respectively.

The applications described above make use of complex queries over large amounts of data. Such applications drive the need for such as Hive [24], offering SQL-like interfaces on top of MapReduce infrastructures. In addition to Hadoop MapReduce, there are a few other options that can serve as the computing engine in such

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

IDEAS '16, July 11 - 13, 2016, Montreal, QC, Canada

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4118-9/16/07...$15.00

DOI: http://dx.doi.org/10.1145/2938503.2938525

a system. Apache Spark [30] lets programmers construct complex, multi-step directed acyclic graphs (DAGs) of work, and executes those DAGs all at once, not step by step. This eliminates the costly synchronization required by Hadoop MapReduce. Spark also supports in-memory data sharing across DAGs, using RDDs [29]. Prior research on DAG engines includes Dryad [14], a Microsoft Research project used internally at Microsoft for its Bing search engine and other hosted services. Based on [14], the open source community created Apache Tez, which can be used by Hive as an alternative query engine. In this paper we compare our approach against Hive on Hadoop MapReduce and Hive on Tez.

The use of bit-sliced indices (BSI) to encode the score lists and perform top-k queries over high-dimensional data using bit-wise operations was proposed in [13] for the centralized case. The BSI method was shown to outperfom TA-based methods as well as sequential scan. In this paper, we set out to parallelize the topK queries using BSI. We implement three MapReduce algorithms for BSI over Spark [30] and evaluate the performance of top-k and weighted top-k queries over distributed high-dimensional data using MapReduce. We measure several aspects of scalability by varying the data dimensionality, cardinality, the number of rows to determine the best horizontal partition size. We also vary the number of executors/nodes in the cluster for scalability measurements.

To the best of our knowledge this is the first paper that implements bit-sliced index (BSI) arithmetic over MapReduce. We compare performance against Hive [24] running on MapReduce, Tez, and Spark SQL.

The primary contributions of this paper can be summarized as follows:

- We propose the usage of distributed BSI arithmetic for aggregations over large column stores. We enable both vertical partitioning as well as horizontal partitioning for the BSI index.
- We design and implement three MapReduce algorithms for BSI aggregation: a baseline tree reduction (Tree-BSI), a two-phase tree reduction using groups of slices (Group-BSI), and a two-phase reduction algorithm using the slice depth as mapping key (Slice-BSI).
- We formalize a cost model for the proposed method and apply it to dynamically determine the optimum partitioning and hardware resource allocation given the dataset and the available hardware in the cluster.
- We compare the performance of the three BSI algorithms and a column-store implementation (non BSI) over MapReduce. We analyze the cost of the proposed algorithm in terms of index size, network traffic, and execution time. We estimate these performance parameters and then compare with measurements over real datasets.
- We also compare the performance of the proposed distributed index against sequential scan over a row-store implemented in Hive, a widely used, MapReduce based data warehouse.

The rest of the paper is organized as follows. Section 2 presents background and related work. Section 3 describes the problem formulation and the proposed solution using vertical and horizontal partitioning in MapReduce. Section 4 shows experimental results over a Spark/Hadoop cluster. Finally, conclusions are presented in Section 5.

2. BACKGROUND AND RELATED WORK

This section presents background information for bit-sliced indices and related work for top-k query processing.

2.1 Bit-Sliced Indexing

Bit-sliced indexing (BSI) was introduced in [20] and it encodes the binary representation of attribute values with binary vectors. Therefore, $\lceil \log_2 values \rceil$ vectors, each with a number of bits equal to the number of records, are required to represent all the values for a given attribute.

Figure 1 illustrates how indexing of two attribute values and their sum is achieved using bit-wise operations. Since each attribute has three possible values, the number of bit-slices for each BSI is 2. For the sum of the two attributes, the maximum value is 6, and the number of bit-slices is $\lceil \log_2 6 \rceil = 3$. 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. For example, 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 $B_2[0]$: $sum[0] = B_1[0] \oplus B_2[0]$. 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 $sum[2]$, which is the carry, is $majority(B_1[1], B_2[1], (B_1[0] \wedge B_2[0]))$, where $majority(A, B, C) = (A \wedge B) \vee (A \wedge C) \vee (B \wedge C)$.

BSI arithmetic for a number of operations, including the addition of two BSIs, is defined in [23]. Previous work [13, 22], uses BSIs to support preference and top k queries efficiently. BSI-based top k for high-dimensional data [13] was shown to outperform current approaches for centralized queries. In this work we adapt this preference query processing to a distributed setting.

2.2 Bitmap Compression

Most types of bitmap (bit-vector) compression schemes use specialized run-length encoding schemes that allow queries to be executed without requiring explicit decompression.

Word-Aligned Hybrid Code (WAH) [26] proposes the use of words to match the computer architecture and make access to the bitmaps more CPU-friendly. WAH divides the bitmap into groups of length $w - 1$, where w is the CPU's word size. WAH then collapse consecutive all-zeros or all-ones groups into a fill word.

Recently, several bitmap compression techniques that improve on WAH by making better use of the fill word bits have been proposed in the literature [15, 27], and others. Previous work have also used varying segment lengths s within $s \leq w$ encoding [12].

In this work we use our recently proposed bit-vector compression scheme [10], which is a hybrid between the verbatim scheme and the EWAH/WBC [15] bitmap compression. This hybrid scheme compresses the bit-vectors if the bit density is below a user-set threshold. Otherwise the bit-vectors are left verbatim. In our experiments we begin with compressed bit-vectors if the compressed size for the bit-vector is 0.5 or smaller than the size of the uncompressed bit-vector. The query optimizer described in [10] is able to decide at run time when to compress or decompress a bit-vector, in order to achieve faster queries. We choose this compression scheme due to its capability of dealing with denser bitmaps, which is the case for the bit-vectors inside the bit-sliced index, and it allows for uncompressed bit-vectors to be operated with compressed ones. Nonetheless, it is possible to apply other compression models, such as the one proposed in [4]. The compression model is orthogonal to the contributions of this work.

2.3 Distributed Top-k Queries

Several approaches based on the Threshold Algorithm (TA) [8] have been proposed to efficiently handle top-k queries for multi-dimensional data in a distributed environment [3, 7, 8, 9, 17].

The Three Phase Uniform Threshold (TPUT) [3] algorithm was

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

Figure 1: Simple BSI example for a table with two attributes and three values per attribute.

shown to outperform TA by reducing the communication required and terminating in a fixed number of round trips. TPUT consists of three phases. The first phase collects the top-k at every node. It then establishes a threshold from the lower bound estimate of the partial sums of the top-k objects gathered from all the nodes. For high-dimensional data the computed threshold is not able to considerably prune the objects retrieved in the second phase as a large number of objects would satisfy the threshold. Using the retrieved objects, the lower bound estimate is refined and upper bounds are calculated for all the objects. Objects are pruned when the upper bound is lower than the refined lower bound estimate. The third phase collects the remaining information of the remaining candidate objects from the nodes and ultimately selects the top-k. Several approaches optimize threshold computation by storing data distribution information [28] or trading-off bandwidth for accuracy [18].

However, as shown in [13], TA-based algorithms are not competitive for high dimensional spaces and are outperform by sequential scan. Moreover, these techniques require index/clustered and/or expensive sorted access to the objects which limit their adaptation to the highly parallel system of Hadoop [2]. For these reasons, we only compare performance with sequential scan as the base line, implemented as a selection query over MapReduce using HiveQL [24], a SQL-like interface based on MapReduce jobs.

Other related work includes RanKloud [2], which proposes a "utility-aware" repartitioning and pruning of the data using runtime statistics to avoid all input from being processed. However, it does not guarantee a retrieval of the top-k results. Another recently proposed approach is to execute rank join queries over NoSQL databases [19]. While this paper deals with a different problem than the one addressed in this paper, namely top-k joins, we believe the top-k selection algorithm described in this paper can be adapted to support top-k joins as well. This extension and mapping from top-k selection to top-k join is left as object for future work.

3. PROPOSED APPROACH

3.1 Problem Formulation

We now formally define the *top-k weighted preference query problem*. For clarity, we define the notations in Table 1.

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:
$F(t) = E(q_1 f_1(t), \ldots, q_m f_m(t))$ where E is a numeric-valued expression. F is monotonic 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: $F(t) = \sum_{i=1}^{m} q_i f_i(t)$. The k data items whose overall scores are the highest among all data items, are called the top-k data items.

3.2 Index Structure and Basic BSI Arithmetic

Table 1: Notation Reference

Notation	Description
n	Number of rows in the data
m	Number of attributes in the data
s	Number of slices used to represent an attribute
w	Computer architecture word size
Q	Query vector
$\|q\|$	Number of non-zero preferences in the query
a	Attributes per node (in case of vertical partitioning)
g	Slices per group (groups are shuffled during aggregation)
d	Depth of a bit-slice, a number between 0 and $s-1$

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 B_i 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.

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 must be partitioned and distributed across the nodes of a cluster.

With this goal in mind, we create a $BSIAttr$ class that can serve as a data structure for an atomic BSI element included in a partition. Each partition can include one or more $BSIAttr$ object. A $BSIAttr$ object can represent all the attribute tuples (in the case of vertical-only partitioning) or only a subset (in the case of horizontal, or vertical and horizontal partitioning). Furthermore, a $BSIAttr$ object can carry all of the attribute bit-slices or only a subset of them.

The data partitions, as well as the index partitions, are stored on a distributed filesystem (HDFS), 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.

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.

3.3 Distributed Top-k Query Processing Using the BSI Index

We identify two main stages to implement the top-k query execution algorithm using BSI, as described in the centralized case [13].

The computation of the scores over *all* dimensions in parallel (aggregating all the attribute values in our case for SUM_BSI) and performing the top-k operation over the result BSI (done in a single node). One could implement the BSI attribute aggregation over MapReduce in several ways. In this section we explore three different algorithms for performing SUM_BSI in parallel and evaluate their strengths and limitations.

3.3.1 SUM_BSI using Tree Reduction

The simplest way to implement this 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 pseudocode for this simple approach is provided in Algorithm 1, 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.

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 1: 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
```

3.3.2 SUM_BSI Group Tree Reduction

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

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. 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 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.

3.3.3 SUM_BSI Using Slice Mapping

It is true that the compact representation of the BSI makes the algorithms described in Section 3.3.1 and Section 3.3.2 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 3. 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 3.4. For the 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 2.

For clarity in describing our algorithms, we use the example illustrated in Figure 3. 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 2. 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

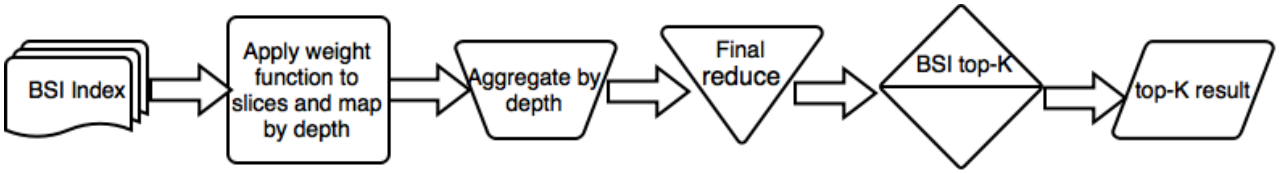

Figure 2: Top-k (preference) query stages using the two-phase BSI slice mapping method

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() function of Algorithm 2. In this step, all the bit-slices with the same key (depth) are aggregated into a $BSIAttr$. Line 9 of Algorithm 2 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.

3.4 Cost Estimations for Two-phase Map-reduce BSI Aggregation

As shown in the work that uses the BSI index for top-k queries on single machines [13], the query time is dominated by aggregation. Thus, in this section we focus on estimating the complexity of the two-phase MapReduce 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.

3.4.1 Data Shuffle Estimation

The mapping in the first phase (Figure 3), does not produce any shuffling since it aggregates only the slices from attributes found on the same node. Data 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 3 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:

$$\lceil \log_2(g+a) \rceil \tag{1}$$

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:

$$Sh_1 = \left[(\min\left(\left\lceil \frac{s}{g} \right\rceil, \left\lceil \frac{m}{a} \right\rceil \right) - 1 \right] \cdot \left\lceil \frac{m}{a} \right\rceil \cdot \lceil \log_2(g+a) \rceil \tag{2}$$

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

$$\lceil \log_2(g+a) \rceil + \left\lceil \log_2(\frac{m}{a}) \right\rceil = \left\lceil \log_2 \frac{(g+a)m}{a} \right\rceil \tag{3}$$

The total number of slices shuffled between the mappers and re-

Algorithm 2: 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
```

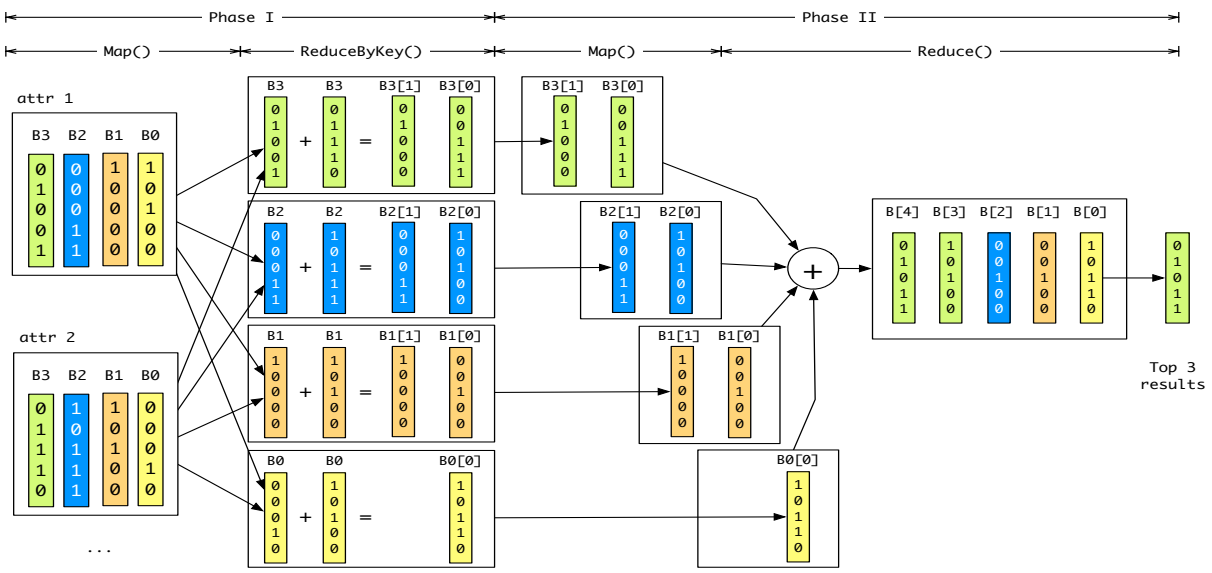

Figure 3: SUM_BSI Using Slice Mapping Example

ducers of the second phase is:

$$Sh_2 = \left(\left\lceil \frac{s}{g} \right\rceil - 1\right)\left\lceil \log_2 \frac{(g+a)m}{a} \right\rceil \tag{4}$$

The total amount of data shuffled is the sum of the results from Equations 2 and 4:

$$Sh = Sh_1 + Sh_2. \tag{5}$$

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 [11] should be considered in estimating the size of the bit-slices.

3.4.2 *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:

$$T_1 = \sum_{i=1}^{\log_2 a} (g+i). \tag{6}$$

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:

$$T_2 = \sum_{i=1}^{\lceil \log_2 m/a \rceil} (g + \lceil \log_2 a \rceil + i) \tag{7}$$

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

$$T_3 = \sum_{i=1}^{\lceil \log_2 s/g \rceil} \left(g + \lceil \log_2 a \rceil + \left\lceil \log_2 \frac{m}{a} \right\rceil + i\right) \tag{8}$$

When taking into consideration the time complexities from Equations 6, 7, and 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:

$$W_{T_2} = \frac{1}{\lceil \frac{m}{a} \rceil} \tag{9}$$

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:

$$W_{T_3} = \frac{1}{\lceil \frac{m}{a} \rceil \lceil \frac{s}{g} \rceil} \tag{10}$$

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. In the following section we evaluate these estimations, and other measures in our experiments.

4. EXPERIMENTAL EVALUATION

In this section we evaluate the BSI index on a Hadoop/Spark cluster, in terms of scalability, index size, compression ratio, and query times. We run each query five times and report the average time. We then compare the query times for weighted and non-weighted top-k preference queries against SparkSQL, Hive on Hadoop MapReduce and Hive on Tez. We perform the top-k preference queries using k=20. We do not analyze the impact of changing k,

as it was shown in [13], the change of k does not impact the total query time in a significant way. The authors of [19] show that the change in the value of k does not significantly impact the query time when running top-k on Hive either.

In addition, we evaluate the scalability of the proposed approach as the data dimensionality increases, and the scalability as the number of executors increases. Further, we analyse the network throughput of the proposed approach and how the partitioning and the grouping of the BSI attributes affects the network traffic load. We also evaluate our estimations of the shuffle and experiment with estimating the total running time of a top-k preference query using the BSI index.

4.1 Experimental Setup

We implemented the proposed index and query algorithms in Java, and used the Java API provided by Apache Spark to run our algorithms on an in-house Spark/Hadoop cluster. The Java version installed on the cluster nodes was 1.7.0_79, Spark version 1.1.0. and Hadoop version 2.4.0. As cluster resource manager we used Apache Yarn.

Our Hadoop stack installation is built on the following hardware: There is one Namenode (master) server (Two 6-core Intel Xeon E5-2420v2, 2.2GHz, 15MB Cache; 48 GB RAM at 1333 MT/s Max). The cluster also contains four Datanode (slave) servers (two 6-core Intel Xeon E5-2620v2, 2.1 GHz,15MB Cache; 64 GB RAM at 1333 MT/s Max). The namenode and datanodes are connected to each other over 1 Gbps Ethernet links over a dedicated switch. Unless otherwise noted, we use all the available hardware resources in this cluster for running the experiments.

In our experiments we used synthetically generated data as well as two real datasets to evaluate the proposed indexing and querying. The two real datasets used are described below:

- `HIGGS`[1]. This dataset was used in [1] and has been produced using Monte Carlo simulations. The first 21 features (columns 2-22) are kinematic properties measured by the particle detectors in the accelerator. The last seven features are functions of the first 21 features; these are high-level features derived by physicists to help discriminate between the two classes. This dataset has a high cardinality, as each attribute is represented by real numbers with with numeric precisions of 16. In a non-compressed form, this dataset has a size of 7.4 GB. Its distribution is close to being uniform.

- `Rainfall`[2] This Stage IV dataset contains rainfall measures for the United States using a 4Km x 4Km -resolution grid. The data corresponds to 1 year (2013) of hourly collected measurements. The number of cells in the grid is 987,000 (881 x 1121). Each cell was mapped to a bit position. A BSI was constructed for each hour generating 8,758 BSIs. This dataset has a lower cardinality than the previous described dataset. In a non-compressed form, this dataset has a size of 98 GB. This is a sparser dataset, as it reflects rainfall patterns.

4.2 Data Cardinality/ Number of slices per attribute

One of the features of the BSI index is that it is possible to trade some of the accuracy of the indexed data for faster queries. For example, it is possible to represent real numbers in less than 64 or 32 bits, by slicing some decimals off these numbers. Also, if the cardinality of the dataset does not require 32 or 64 bits, the BSI index uses only the required number of bits/bit-slices to represent the highest value across one dimension.

[1]http://archive.ics.uci.edu/ml/datasets/HIGGS/

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

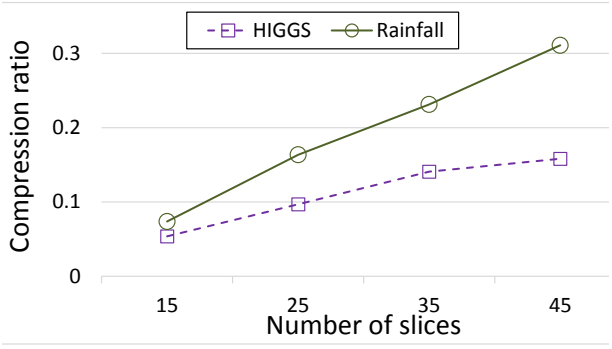

Figure 4: Compression ratio of the BSI index when increasing the number of bit-slices per attribute (Datasets: HIGGS, Rainfall)

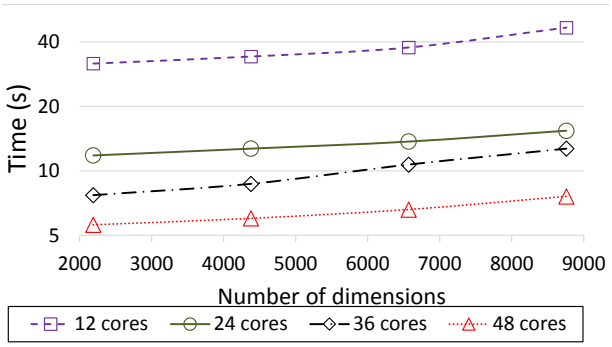

Figure 5: Top-k non-weighted preference query times using Slice-BSI when varying the number of dimensions and the number of executors (cpu cores). (Dataset: Rainfall, 25 bit-slices per dimension)

Figure 4 shows the compression ratio (C_{ratio}), which is the ratio between the BSI index size (BSI_{size}) and the original data size ($DataFile_{size}$) when varying the number of bit-slices to represent the attributes of the HIGGS and Rainfall datasets.

$$C_{ratio} = \frac{BSI_{size}}{DataFile_{size}}$$

We measure the size of a original dataset as the size on disk taken by a dataset in a non-compressed text/csv file, as this is one of the most common file formats used with the Hadoop systems.

In the case of our BSI index, the compression comes not only from using a limited number of bit-slices per attribute, but also from compressing each individual bit-slice (where beneficial) using a hybrid bitmap compression scheme [10].

As expected and can be seen in Figure 4, the compression ratio degrades as the number of slices per attribute grows, however it is still significant even for 45 bit-slices per attribute.

Because the Rainfall data had a lower cardinality to begin with, the compression ratios or HIGGS are better than those for the Rainfall dataset. However, it is worth noting that when representing the HIGGS dataset with less than 64 bit-slices per attribute, there is a minimal loss in the numerical precision. With 45 slices we are able to represent 12 decimal positions.

4.3 Scalability of the proposed indexing and querying approach

We test the scalability of the proposed two-phase slice-mapping in terms of query time as data dimensionality and the number of

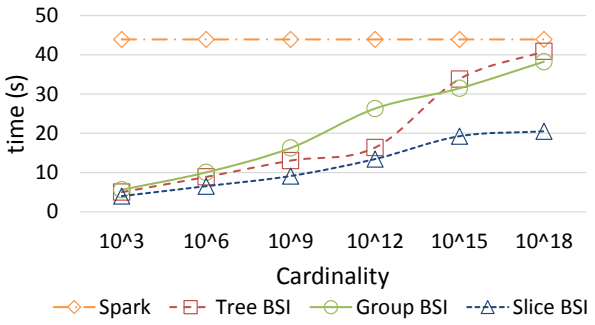

Figure 6: Top-K non-weighted preference query time as data cardinality increase. (Dataset: Uniform, 260 attributes, 5 Million rows)

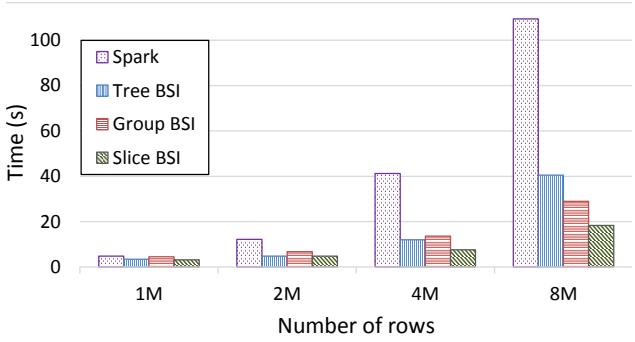

Figure 7: Top-k (K=20) non-weighted preference query time as the number of rows increases. (Dataset: Uniform, 260 attributes, 40 slices/attribute)

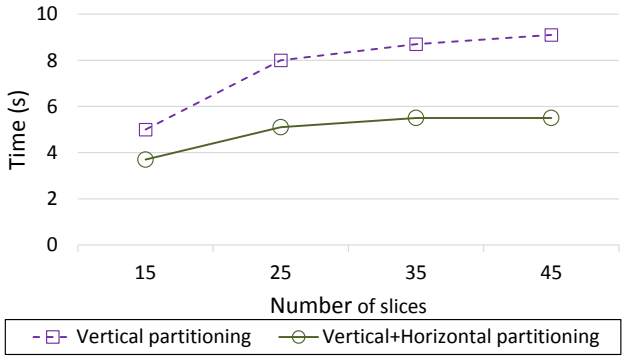

Figure 8: Top-k (K=20) non-weighted preference query using Slice-BSI with vertical and vertical+horizontal partitioning, while varying the number of bit-slices per attribute. (Dataset: HIGGS)

computing nodes/CPU cores increases, as well as for increasing data cardinality.

Figure 5 shows the non-weighted top-k preference query times over the Rainfall dataset as the number of dimensions increases from 2,000 to 8,758. The results are shown 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 5 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.

Given that the BSI index is sensitive to data cardinality, we set up to measure the scalability of the two-phase slice-mapping algorithm when compared to the BSI tree-reduction (labeled `Tree BSI`) and the two-round BSI tree-reduction (labeled `Group BSI`) methods. We also compare with a MapReduce method for aggregation implemented in Spark, that is similar to the Tree BSI without using the BSI index (labeled (labeled `Spark`)). Figure 6 shows the query times for top-K queries 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 non BSI method is not sensitive to data cardinality.

It is worth noting that the Group BSI method is sometimes slower than the straightforward Tree BSI method. This is somewhat counter-intuitive. However, it can be explained in the cases where the Tree BSI query is not dominated by network shuffling. In these cases, is not worth minimizing the network communication by splitting the work into a two round MapReduce job. By doing this, the result of the first round has to be available before starting the second round, and thus a lazy node can slow down the entire job.

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.

4.4 Partitioning, network throughput, and load balancing

Figure 7 shows the performance of the 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 reach 8M 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 will be a subject of future study.

Figure 8 shows the query times when performing a non-weighted top-K preference query over the HIGGS dataset using vertical partitioning only, and vertical + horizontal partitioning. The horizontal partitions contain 5 Million rows, thus splitting each attribute of the dataset into two horizontal partitions. The two-phase slice mapping improves the query time when using both horizontal and vertical partitioning. This is due to the partial BSI attribute being smaller(*i.e.*, smaller task size), and a reduced network communication by virtue of grouping several attributes into the same partition.

4.5 Evaluation of Data Shuffle Estimations

To evaluate the estimations described in Section 3.4.1, we use

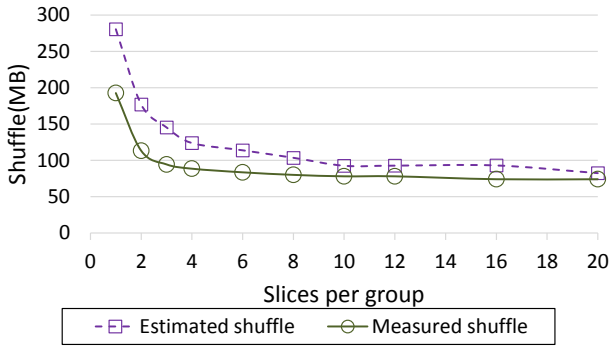

Figure 9: Estimated data shuffle compared to measured data shuffle, for the two phase aggregation method. (Dataset: Rainfall data, 20 slices/attribute, index size: 9.8 GB, index partitions: 94)

the Rainfall dataset with one horizontal partition. In the case of multiple horizontal partitions, the estimations from Section 3.4.1 should be applied to each horizontal partition. The size of each compressed bit-slice was computed using the estimations described by [11], 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 9 shows the estimated data shuffled in MB and the measured data shuffled when performing boolean top-k queries over the rainfall data. These measurements are shown for different values of g - slices per group. The BSI index 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 in Section 3.4.1 reflect the upper bounds for data shuffling.

4.6 Evaluation of Query Time Estimations

In this section we estimate the Boolean top-k query times when varying the number of bit-slices per group (g). The purpose of this experiment is to validate our estimations from Section 3.4.2, 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 MapReduce method for top-k. 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:

$$Norm(e_i) = \frac{e_i - E_{min}}{E_{max} - E_{min}}, \tag{11}$$

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

The top-k query time is dominated by the BSI aggregation, as we discussed in the previous sections. 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 3.4.

$$Time = Norm(Sh_i) + Norm(T_i), \tag{12}$$

where

$$T = W_{T_1} T_1 + W_{T_2} T_2 + W_{T_3} T_3. \tag{13}$$

Sh, W_{T_1}, T_1, W_{T_2}, T_2, W_{T_3}, and T_3 are defined in Section 3.4.

The results of this experiment are shown in Figure 10. In this case the optimum number of slices per group during the MapReduce 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 MapReduce scheduling and synchronization. Our goal is not to predict execution 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.

4.7 Comparison against existing distributed data stores

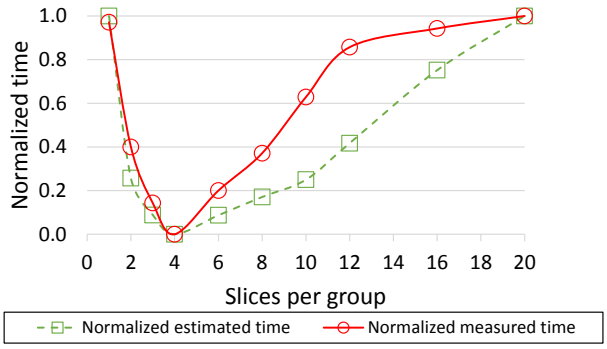

Figure 10: Estimated vs. measured execution time for the slice-BSI aggregation method. (Dataset: Rainfall, 20 slices/attribute, index size: 9.8 GB, index partitions: 94)

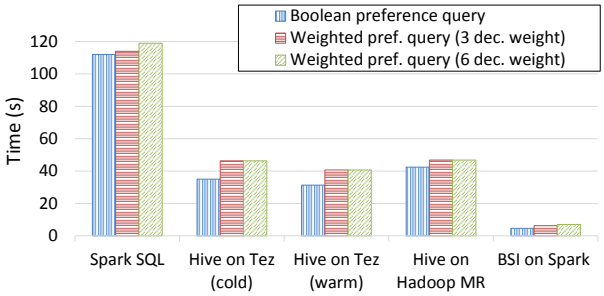

Figure 11: Top-K preference query time for slice-BSI compared to Hive on Hadoop MapReduce, and Hive on Tez. (Dataset: HIGGS, 32 slices/attribute)

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 generated 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 'score'
FROM table ORDER BY score DESC LIMIT k
```

Figure 11 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 [5]. 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.

5. CONCLUSION

In this work we have implemented preference (top k) queries over bit-sliced indices using MapReduce. We parallelized two BSI arithmetic operations: multiplication by a constant and addition of two BSIs. We show that the two-phase slice-mapping algorithm

proves to be more efficient than other two simpler MapReduce methods for aggregating BSI attributes, while all three methods outperform the non-BSI MapReduce aggregation over high dimensional columnar data. The proposed indexing and query processing works for both: horizontally partitioned (row-stores) and vertically partitioned (column-stores) data. This approach is robust and scalable for high dimensional data. We executed top-K over a dataset with over 8,000 dimensions. It also scales well when adding more computing hardware to the cluster. In our experiments, when increasing the number of CPU cores from 24 to 48, query time decreased by approximately 50% for all the preference queries executed over different number of dimensions. When decreasing the number of bit-slices per dimension, the index size and the query time also decrease linearly. Thus the datasets that have lower cardinality can benefit even more from better index compression and faster query times.

We provide the cost analysis of our approach, and accurately estimate the optimum parameters for achieving the best query times. We showed that the proposed approach outperforms Hive, a MapReduce based data warehouse, over Hadoop Map-Reduce and the optimized query engine Tez. It is also 25 times faster than SparkSQL, which uses the same distributed computation engine as the BSI index.

For the future, we plan to implement and support more types of queries on top of the distributed BSI index, such as constrained top-K queries, rank joins, skyline queries, and others. We also plan to investigate further the effects of the BSI attribute segment size, and how the CPU cache size, or the disk page size together with the segment size affect the query time.

References

[1] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Commun*, 5, 2014.

[2] K. S. Candan, P. Nagarkar, M. Nagendra, and R. Yu. Rankloud: A scalable ranked query processing framework on hadoop. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 574–577, New York, NY, USA, 2011. ACM.

[3] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 206–215, New York, NY, USA, 2004. ACM.

[4] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *arXiv preprint arXiv:1402.6407*, 2014.

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

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[7] C. Doulkeridis and K. Norvag. A survey of large-scale analytical query processing in mapreduce. *The VLDB Journal*, 23(3):355–380, June 2014.

[8] R. Fagin, A. L. Y, and M. N. Z. Optimal aggregation algorithms for middleware. In *In PODS*, pages 102–113, 2001.

[9] U. Guntzer, W.-T. Balke, and W. Kiesling. Optimizing multi-feature queries for image databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 419–428, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

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

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

[12] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin. A tunable compression framework for bitmap indices. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 484–495. IEEE, 2014.

[13] G. Guzun, J. Tosado, and G. 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.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[15] D. Lemire, O. Kaser, and E. Gutarra. Reordering rows for better compression: Beyond the lexicographic order. *ACM Transactions on Database Systems*, 37(3):20:1–20:29, 2012.

[16] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 129–140. VLDB Endowment, 2003.

[17] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, June 2004.

[18] S. Michel, P. Triantafillou, and G. Weikum. Klee: A framework for distributed top-k query algorithms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 637–648. VLDB Endowment, 2005.

[19] N. Ntarmos, I. Patlakas, and P. Triantafillou. Rank join queries in nosql databases. *Proc. VLDB Endow.*, 7(7):493–504, Mar. 2014.

[20] P. 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.

[21] M. Persin, J. Zobel, and R. Sacks-davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47:749–764, 1996.

[22] D. 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.

[23] D. Rinfret, P. O'Neil, and E. O'Neil. Bit-sliced index arithmetic. *SIGMOD Rec.*, 30(2):47–57, 2001.

[24] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[25] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[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] H. Yu, H.-G. Li, P. Wu, D. Agrawal, and A. El Abbadi. Efficient processing of distributed top-k queries. In *Proceedings of the 16th International Conference on Database and Expert Systems Applications*, DEXA'05, pages 65–74, Berlin, Heidelberg, 2005. Springer-Verlag.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[30] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. 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.