Scalable preference queries for high-dimensional data using map-reduce

Gheorghi Guzun
Electrical and Computer Engineering
The University of Iowa
Iowa City, USA
gheorghi-guzun@uiowa.edu

Joel E. Tosado
Electrical and Computer Engineering
The University of Iowa
Iowa City, USA
joel-tosadojimenez@uiowa.edu

Guadalupe Canahuate
Electrical and Computer Engineering
The University of Iowa
Iowa City, USA
guadalupe-canahuate@uiowa.edu

***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 preference queries efficiently in a non-distributed environment for data with hundreds of dimensions.**

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 data with thousands of dimensions. Indexing is implemented on top of Apache Spark for both column and row stores and shown to outperform Hive when running on Map-reduce, and Tez for top-k (preference) queries.

I. Introduction

The MapReduce[1] framework 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 [2], implements MapReduce and provides developers the ability to easily leverage it in order to exploit any parallelizable computing tasks. Hadoop has further gained ground from SQL-like interfaces such as Hive [3]. Nevertheless, it has its shortcomings as was indicated in [4]. Among these shortcomings, it does not innately optimize our operator of interest, top-k selection. An efficient exact top-k approach optimized on MapReduce is still an open question.

Top-k selection queries are ubiquitous in big data analytics. The top-k (preference) query here refers to applying a monotonic scoring function S_i over the A scoring attributes a_i (attributes of interest to the query) to each object i under consideration. Furthermore, a weighted sum function is a common scoring function where a weight w_i is multiplied to each attribute a_i. The weights are defined at query time. That is $S_i = \sum_{t=1}^{A} w_i \times a_i$. The top k objects with the highest scores is the result for the top-k selection query.

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 its top-k result [5, 6]. 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.

The need for complex queries, particularly for analytics, leads to the rise of data warehousing systems, such as Hive [3], offering SQL-like interfaces. These systems are built on top of the Apache Hadoop stack, which uses Hadoop MapReduce as its processing engine.

More recently, alternatives have been introduced to Hadoop MapReduce. Apache Spark [7] 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 [8]. Prior research on DAG engines includes Dryad [9], a Microsoft Research project used internally at Microsoft for its Bing search engine and other hosted services. Based on Dryad, 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 [10] for the centralized case. In this paper we implement BSI over Spark [7] and evaluate the performance of top-k and weighted top-k queries over distributed high-dimensional data using map-reduce. We measure two aspects of scalability: scalability as data and dimensionality increase, and scalability as the number of executors increases.

To the best of our knowledge this is the first paper that implements BSI arithmetic over MapReduce. The primary contributions of this paper can be summarized as follows:

	Raw Data		Bit-Sliced Index (BSI)				BSI SUM		
			Attrib 1		Attrib 2				
Tuple	Attrib 1	Attrib 2	$B_1[1]$	$B_1[0]$	$B_2[1]$	$B_2[0]$	$sum[2]^3$	$sum[1]^2$	$sum[0]^1$
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

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

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

- We design efficient map-reduce algorithms to evaluate top-k queries over bit-sliced indexing.
- We implement the algorithms over both: horizontally partitioned (row-stores) and vertically partitioned (column-stores) data.
- We analyze the cost of the proposed algorithms in terms of index size, network traffic, and execution time over real datasets.
- We compare the performance of the proposed distributed index against existing, widely used, map-reduce based data warehouses.

The rest of the paper is organized as follows. Section II presents background and related work. Section III describes the problem formulation and the proposed solution for both row- and column-stores using map-reduce. Section IV shows experimental results over a Hadoop cluster. Finally, conclusions are presented in Section V.

II. Background and Related Work

This section presents background information for bit-sliced indices and related work for top-k query processing. For clarity, we define the notations used further in this paper in Table I.

A. Bit-Sliced Indexing

Bit-sliced indexing (BSI) was introduced in [11] 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 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. 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, including the addition of two BSIs is defined in [12]. Previous work [10, 13], uses BSIs to support preference and top k queries efficiently. BSI-based top K for high-dimensional data [10] was shown to outperform current approaches for centralized queries. In this work we adapt this preference query processing to a distributed setting.

B. 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. Byte-aligned Bitmap Code (BBC) [14] 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 and query processing is CPU intensive. Word Aligned Hybrid (WAH) [15] proposes the use of words instead of bytes to match the computer architecture and make access to the bitmaps more CPU-friendly. WAH divides the bitmap into groups of length $w-1$ and collapse consecutive all-zeros/all-ones groups into a fill word.

Recently, several bitmap compression techniques that improve WAH by making better use of the fill word bits have been proposed in the literature [16, 17, 18], and others. Previous work has also used different segment lengths for encoding [19].

A bit-vector compression scheme, which is a hybrid between the verbatim scheme and the EWAH/WBC [18] bitmap compression, was proposed recently in [20]. This Hybrid scheme compresses the bit-vectors if the bit density is below a user set threshold, otherwise the bit-vectors are left verbatim. The query optimizer described in [20] is able to decide at run time when to compress or decompress a bit-vector, in order to achieve faster queries. The Roaring bitmaps [21], combines position list encoding with the

verbatim scheme, they split each bit-vector into chunks and each chunk can be represented as an array of positions, or verbatim.

The last two compression schemes described above, are of particular interest for us, since they are capable of dealing with denser bitmaps, which is the case for the bit-vectors inside the bit-sliced index. We choose to compress the bit-vectors using the methods described in [20], however the compression proposed by [21] could also be feasible.

C. Distributed Top-k Queries

Several approaches based on the Threshold Algorithm (TA) [22] have been proposed to efficiently handle top-k queries in a distributed environment [4, 23, 24, 25, 26].

The Three Phase Uniform Threshold (TPUT) [23] algorithm was 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 [27] or trading-off bandwidth for accuracy [28].

However, as shown in [10], TA-based algorithms are not competitive for high dimensional spaces. 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 [29].

RanKloud [29] proposes a "utility-aware" repartitioning and pruning of the data using run-time statistics to avoid all input from being processed. However, it does not guarantee a retrieval of the top-k results

III. Proposed Approach

A. Problem Formulation

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.

Table I: Notation Reference

Notation	Description
n	Number of rows in the data
m	Number of attributes in the data
s, p	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

B. Key Ideas of the Proposed Approach

The proposed approach exploits the parallelism exhibited by the Bit-Sliced Index (BSI), and avoids sorting in map-reduce 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. We refer to the position of each bit-vector in the BSI for a given attribute as the depth of the bit-vector. In this work we perform the top-k (preference) queries by executing the following steps:

- First we apply the set of weights defined as the query preferences Q. These weights are applied in parallel for each dimension.
- Then we map the bit-slices encoding the weighted dimensions to different mappers. The mapping key is the bit-slice depth within the attribute.
- Next, we aggregate the bit-slices by adding together all the bit-slices with the same depth and obtain a partial sum BSI for every depth.
- We then aggregate all the partial sums BSIs into one final aggregated BSI.
- Finally, we apply the top-K algorithm over the resulting BSI using the algorithms showed in [13, 30].

C. Index Structure and System Architecture

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. The depth of bit-vector $B_i[j]$ is j. Each binary vector contains 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

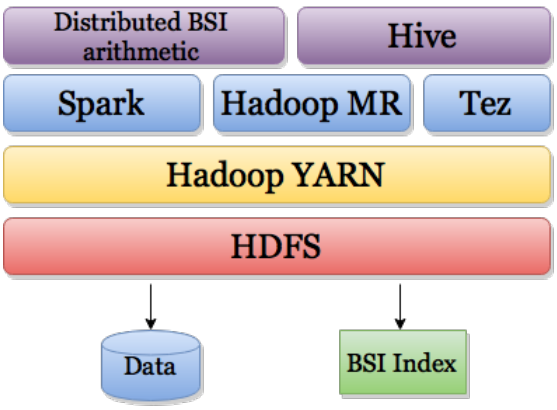


Figure 2: Overview of the software stack used in this work

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 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 carry 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 $BSIattribute$ object can carry all of the attribute bit-slices or only a part of them.

Listing 1 shows the structure of the $BSIattribute$ class. The `size` field stores the number of 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 rows at the end of BSI that were added to complete the last 64-bit word.

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. Figure 2 shows the system stack used in this work for executing top-K preference queries. 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 a good idea to co-locate 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.

Listing 1: BSIattribute Class

```
class BSIattribute {
    int size; //number of bit-slices
    int offset; //offset of the BSI
    BitVector[] bsi; //array of bit-slices
    int partitionID; //if split horizontally
    int nRows; //number of tuples
    BitVector existenceBitmap; //keeps
     //track of existing and deleted rows
}
```

D. Vertical and horizontal partitioning

The BSI index, as defined in [11] 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 or in the CPU cache. In this paper we call this type of segmentation $horizontalpartitioning$. In a distributed environment, the main concern is to minimize network throughput and horizontal partitioning can help in this regard.

Top-K preference queries require aggregation of all the attributes, and thus it may be inefficient to index a single attribute per partition. This would mean that all the index data will 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.

E. Index Compression

When considering compressing the bit-vectors within each BSI, it is important to take into account the effects of compression. Highly compressible bit-vectors can exhibit faster query times than the non-compressed ones [19]. 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 choose to compress the BSI bit-slices using a hybrid compression scheme [20], which is a mix between the verbatim scheme and the EWAH/WBC bitmap compression. It compresses the bit-vectors if the bit

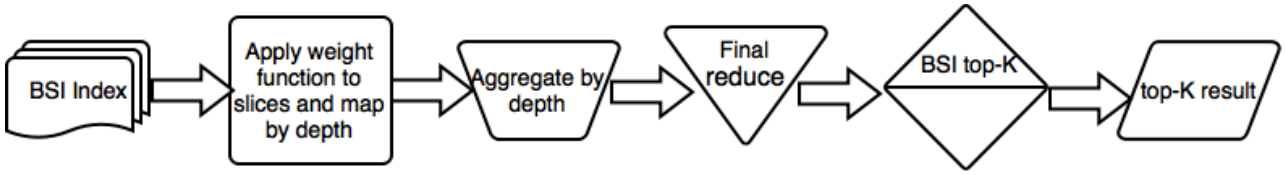


Figure 3: Top-K (preference) query stages

density is below a user set threshold, otherwise the bit-vectors are left verbatim. The query optimizer described in [20] is able to decide at run time when to compress or decompress a bit-vector, in order to achieve faster queries.

F. Distributed Top-k Query Processing

For preference queries considered in this work, the top-k query processing algorithm consists of one mapping stage and two reduce stages. The steps of the top-k preference query are showed in Figure 3.

In the mapping phase, every BSI attribute has its slices mapped to different mappers based on their depth (i.e. the position within the bit-vector array). 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 stem is shown in the Map() function of Algorithm 1. Every mapper has a $BSIattribute$ as input, and outputs a set of $BSIattribute$s that contain one bit-slice each. These bit slices are mapped by their depth in the input $BSIattribute$. Although, there is an overhead associated with encapsulating each bit-slice into a BSIattribute, as we show later, by creating a higher level of parallelism, we also enable for a better load balancing and resource utilization.

The first step of the aggregation is done by the ReduceByKey() function of Algorithm 1. In this step all the bit-slices with the same key (depth) are aggregated into a BSIattribute. Line 9 of Algorithm 1 does the summation of two BSIs. In our implementation we use the same logic as the authors in [12]. In this work, we achieve a parallelization of the BSI summation algorithm by splitting the BSI attribute into individual slices, and executing their addition in parallel similarly to a carry-save adder. The offset of the resulting BSI attributes are saved in the $offset$ field of each $BSIattribute$ 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 second and final step of the aggregation is done by aggregating all the BSIs ($partSum$) 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

Algorithm 1: Distributed BSI aggregation

```
   Map():
   begin
       Input: RDD<Integer, BSIattribute> indexAtt
              //attribute with the query weights
       Output: RDD<Integer, BSIattribute> byDepth
1      int sliceDepth=0;
2      while indexAtt has more slices do
3          bsi = new BSIattribute();
4          bsi.add(indexAtt.nextSlice());
5          bsi = bsi.multiply(weight) //in case of
           weighted preference queries;
6          byDepth.add(new Tuple2(sliceDepth, bsi));
7          sliceDepth++;
8      end
9      return byDepth
   end
   ReduceByKey():
   begin
       Input: RDD<Integer, BSIattribute> byDepth1,
              byDepth2
       Output: RDD<Integer, BSIattribute> partSum
10     partSum = byDepth1.SUM-BSI(byDepth2);
11     return partSum
   end
   Reduce():
   begin
       Input: RDD<Integer, BSIattribute> partSum1,
              partSum2
       Output: RDD<Integer, BSIattribute> sumAtt
12     sumAtt = partSum1.SUM-BSI(partSum2);
13     return sumAtt
   end
```

BSI attributes, that should be concatenated, in the case of vertical and horizontal partitioning.

Algorithm 2 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.

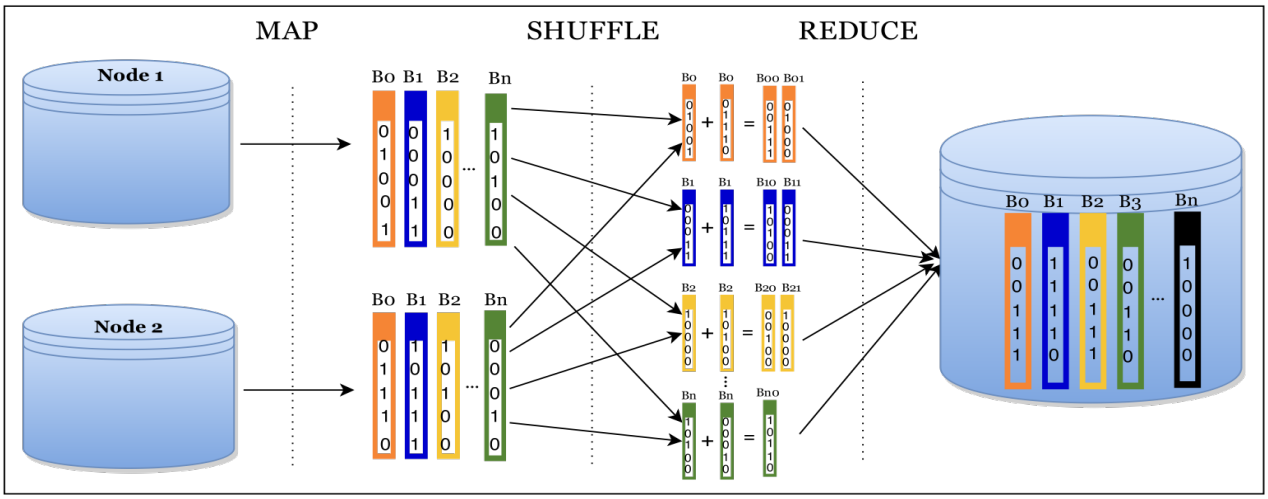


Figure 4: BSI aggregation using map-reduce

Algorithm 2: Concatinates BSI attributes from different horizontal partitions

Input: List<BSIattribute> $sumAtt$
Output: BSIattribute $finalSum$

```
BSIattribute finalSum = new BSIattribute();
while sumAtt.hasNext() do
    t = sumAtt.next();
    finalSum.concatenate();
end
return finalSum
```

Figure 4 shows a simple example of BSI summation between two BSI attributes using the map-reduce paradigm. The mapping phase maps the bit-slices with the same depth from the two BSIs to the same reducer. Splitting the BSI into bit-slices helps simplify and parallelize the summation of the BSIs later in the aggregation/reduce phase. In the reduce phase, the bit-slices with the same depth are added together creating an intermediate result or partial product BSI with the depth as offset. The total number of partial products is n, where n is the maximum number of slices between the two BSI attributes being added. Adding a BSI with only one slice together with another BSI is very efficient (logical XORs and logical ANDs for each slice, all involving only two inputs). In the final phase of the reduce, these n intermediate results are added together to form the final aggregated BSI. In this example when only two BSI attributes are added, it does not seem like an optimization to have $n-1$ BSI additions instead of just one. However, for high dimensional data, where the number of attributes is much higher than n (the maximum number of slices across all dimensions), this optimizes execution time as will be evidenced in the experimental results.

To finally answer the top-K query, we perform the top-K algorithm over a BSI attribute, described and used in [10, 13]

IV. Experimental Evaluation

In this section we evaluate the BSI index on a Hadoop/Spark cluster, in terms of index size, compression ratios, 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 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 the extraction of top-K values from the aggregated BSI attribute is done on a single node (the master node) and as shown in [10], the change of K does not impact the total query time in a significant way. The authors of [31] 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 analyze the network throughput of the proposed approach and how the partitioning and the grouping of the BSI attributes affects the network traffic load.

A. 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, the Spark version was 1.1.0. and the Hadoop version was 2.4.0. As cluster resource manager we used Apache Yarn.

Our Hadoop stack installation is built on the following hardware:

- One ”namenode (master)” server (Two 6-core Intel Xeon E5-2420v2, 2.2GHz, 15MB Cache; 48 GB RAM at 1333 MT/s Max)
- 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.

We used two real datasets in our experiments to evaluate the proposed indexing and querying:

- `HIGGS`[1]. This dataset was used in [32] 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 numeric precisions of 16. In a non-compressed form, this dataset has a size of 7.4 GB
- `Rainfall`[2] This Stage IV dataset consists of 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.

B. 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 to represent the highest value across one dimension.

Figure 5 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.

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

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

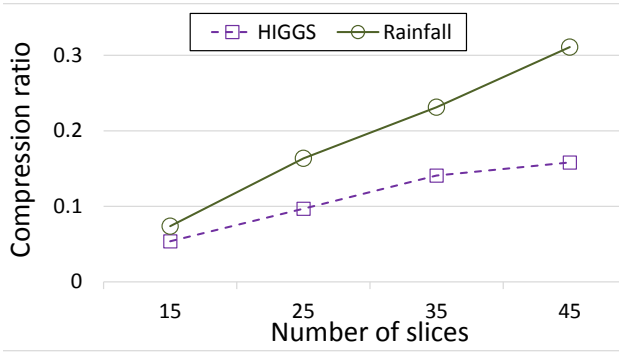


Figure 5: Compression ratio of the index when increasing the number of bit-slices per attribute (Datasets: HIGGS, Rainfall)

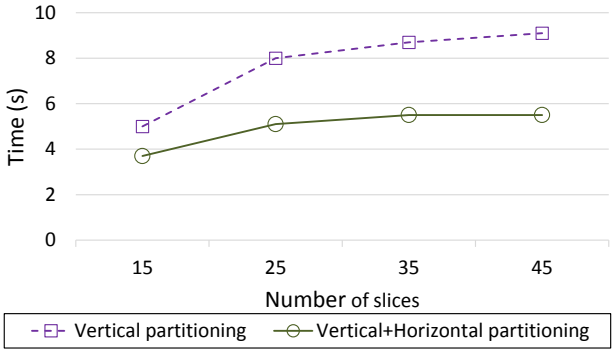


Figure 6: Top-K (K=20) non-weighted preference query over the HIGGS dataset with vertical and vertical+horizontal partitioning, while varying the number of bit-slices per attribute (Dataset: HIGGS).

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 [20].

As expected and can be seen in Figure 5, the compression ratio degrades as the number of slices per attribute grows. 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.

Figure 6 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. We set the partition size limit to 64 MB to allow for a higher granularity of the index and a better load balancing. Because of this, when performing only vertical partitioning of the index, there is only one BSI attribute per partition for a higher number of bit-slices per

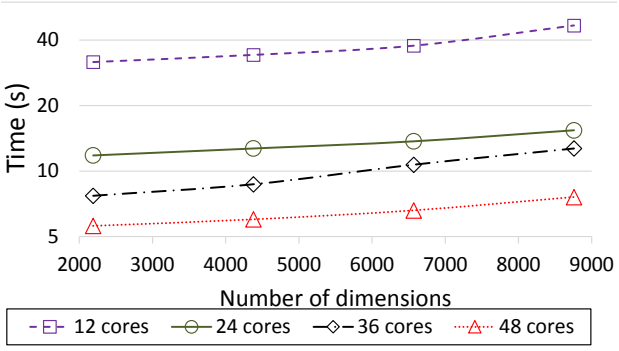


Figure 7: Top-K non-weighted preference query times when varying the number of dimensions and the number of executors(cpu cores) over the Rainfall dataset. (Dataset: Rainfall, 25 bit-slices per dimension)

partition. Having only one BSI attribute per partition leads to higher network throughput, and this is reflected in the query times, where horizontal+vertical partitioning outperforms vertical partitioning only. Thus, for datasets that have a high cardinality, and/or large number of rows, it is advisable to perform both, vertical and horizontal partitioning.

C. Scalability of the proposed indexing and querying approach

We test the scalability of the proposed approach in terms of query time as both data dimensionality and the number of computing nodes/CPU cores increases.

Figure 7 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. 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.

D. Partitioning, network throughput, and load balancing

For a better understanding of how partitioning and grouping of the BSI attributes affects the network throughput, and ultimately the query time, we 'force' a fixed number of BSI attributes to be indexed together within one index partition. For this experiment we used the Rainfall dataset, as it has a high number of dimensions/attributes. Because the number of tuples in this attribute is not very high, this a relatively small size per attribute, we only partition this dataset vertically. We vary the number of dimensions per partition from 1 to 50. We also vary the number of bit-slices to represent each dimension from 15 to 45.

Figures 8a and 8b show the read and write shuffle of the data as percentage of the total BSI index size, as the number of BSI attributes per partition increases from 1 to 50. As these two figures show, the amount of data shuffling is constant as the number of slices per attribute increases. Figure 8c shows the read and write data shuffling as the number of dimensions per partition increases. The least amount of data shuffling is achieved with the maximum number of dimensions per partition. Eventually the data shuffling tends towards zero as the number of dimensions partitioned together grows towards the total number of dimensions of the dataset. Note that Figure 8 show the amount of data shuffling as percentage of the index size, which is usually only a fraction of the original data.

Zero data shuffling, however does not necessarily mean faster queries. There is a trade-off between the partition size and the number of partitions. Thus grouping together multiple dimensions helps reducing network throughput, yet, if the partition sizes become too large, it is possible to partition horizontally as well, without adding significant data shuffling during aggregation required for top-K preference queries.

E. Comparison against existing distributed data stores

To validate the effectiveness of the proposed index and query algorithms, we compare our query times against Hive - a popular data warehousing system. 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.

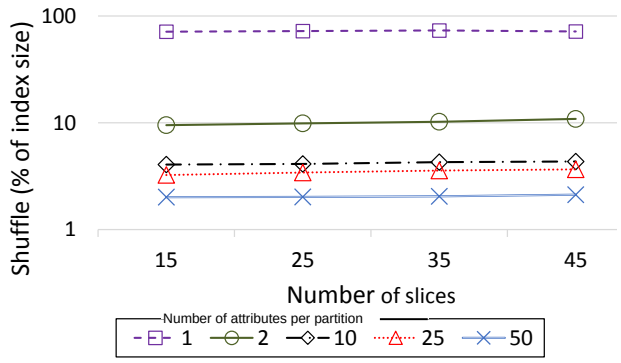
The query ran on Hive has the following syntax:

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

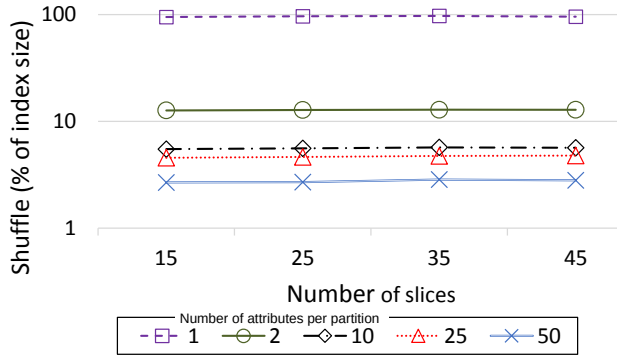
Figure 9 shows the query times of these top-k weighted and non-weighted queries against Hive on Hadoop MR, Hive on Tez and the distributed BSI index. The results show that the BSI was one order of magnitude faster than Hive.

V. Conclusion and Future Work

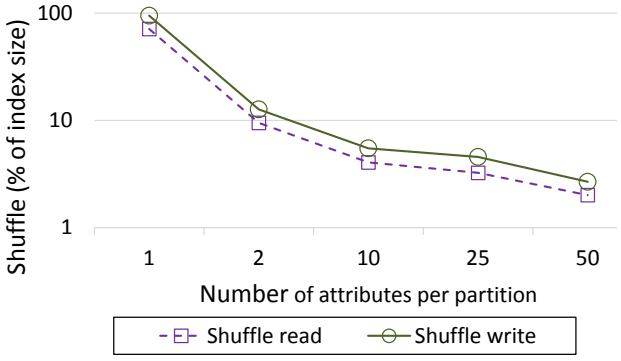
We have implemented preference (top k) queries over bit-sliced indices using map-reduce. We parallelized the aggregation of the BSI attributes. The proposed indexing and query processing works for both: horizontally partitioned (row-stores) and vertically partitioned (column-stores) data. This approach is robust and scalable for high dimensional data. We executed top k queries 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, the



(a) Shuffle read



(b) Shuffle write



(c) Data shuffling vs number of attributes per partition

Figure 8: Data shuffling as percentage of the total index size depending on the number of dimensions per partition and the number of slices per dimension. (Dataset: Rainfall)

preference 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 showed that the proposed approach outperforms Hive over Hadoop Map-Reduce and Hive over Tez even when the BSI index has the same number of bits per

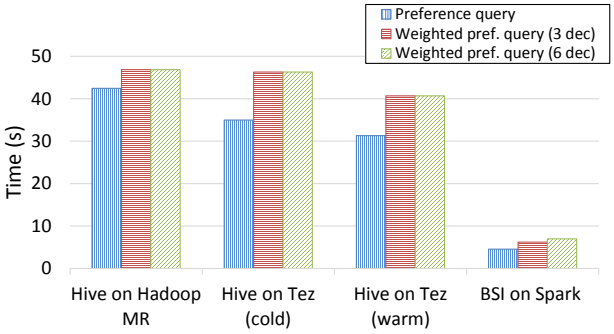


Figure 9: BSI top-K preference and top-K weighted preference query time compared to Hive on Hadoop map-reduce, and Hive on Tez (Dataset: HIGGS, 32 bit-slices per dimension).

attribute as the Hive data.

This is our first attempt on using bit-sliced indexing techniques within a distributed environment. For future work, we plan to investigate the exact amount of data shuffling created by various vertical and horizontal partition sizes and how this affects the total query time. This can help on determining the optimal number of bit-slices that should be grouped together during the map-reduce aggregation. 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. In addition, 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.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[2] T. White, *Hadoop: The definitive guide*, 2012.

[3] 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," *PVLDB*, vol. 2, no. 2, pp. 1626–1629, 2009.

[4] C. Doulkeridis and K. Norvag, "A survey of large-scale analytical query processing in mapreduce," *The VLDB Journal*, vol. 23, no. 3, pp. 355–380, Jun. 2014.

[5] 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*, ser. VLDB '03, 2003, pp. 129–140.

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

[7] 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*, vol. 10, 2010, p. 10.

[8] 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*, ser. NSDI'12, 2012, pp. 2–2.

[9] 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*, vol. 41, no. 3. ACM, 2007, pp. 59–72.

[10] 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*. Springer, 2014, pp. 26–50.

[11] 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*, 1997, pp. 38–49.

[12] D. Rinfret, P. O'Neil, and E. O'Neil, "Bit-sliced index arithmetic," *SIGMOD Rec.*, vol. 30, no. 2, pp. 47–57, 2001.

[13] D. Rinfret, "Answering preference queries with bit-sliced index arithmetic," in *Proceedings of the 2008 C 3 S 2 E conference*. ACM, 2008, pp. 173–185.

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

[15] 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)*, 2002, pp. 99–108.

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

[17] S. J. van Schaik and O. de Moor, "A memory efficient reachability data structure through bit vector compression," in *ACM SIGMOD International Conference on Management of Data*, 2011, pp. 913–924.

[18] D. Lemire, O. Kaser, and E. Gutarra, "Reordering rows for better compression: Beyond the lexicographic order," *ACM Transactions on Database Systems*, vol. 37, no. 3, pp. 20:1–20:29, 2012.

[19] 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*. IEEE, 2014, pp. 484–495.

[20] G. Guzun and G. Canahuate, "Hybrid query optimization for hard-to-compress bit-vectors," in *The VLDB Journal*, in press.

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

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

[23] 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*, ser. PODC '04, 2004, pp. 206–215.

[24] U. Güntzer, W.-T. Balke, and W. Kiebling, "Optimizing multi-feature queries for image databases," in *Proceedings of the 26th International Conference on Very Large Data Bases*, ser. VLDB '00, 2000, pp. 419–428.

[25] R. Fagin, "Combining fuzzy information from multiple systems (extended abstract)," in *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '96, 1996, pp. 216–226.

[26] A. Marian, N. Bruno, and L. Gravano, "Evaluating top-k queries over web-accessible databases," *ACM Trans. Database Syst.*, vol. 29, no. 2, pp. 319–362, Jun. 2004.

[27] 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*, ser. DEXA'05, 2005, pp. 65–74.

[28] 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*, ser. VLDB '05, 2005, pp. 637–648.

[29] 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*, ser. EDBT/ICDT '11, 2011, pp. 574–577.

[30] D. Rinfret, P. O'Neil, and E. O'Neil, "Bit-sliced index arithmetic," in *ACM SIGMOD Record*, vol. 30, no. 2. ACM, 2001, pp. 47–57.

[31] N. Ntarmos, I. Patlakas, and P. Triantafillou, "Rank join queries in nosql databases," *Proc. VLDB Endow.*, vol. 7, no. 7, pp. 493–504, Mar. 2014.

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