Efficient Indexing of Hashtags using Bitmap Indices

Lawan Thamsuhang Subba
Aalborg University
lawbba@cs.aau.dk

Christian Thomsen
Aalborg University
chr@cs.aau.dk

Torben Bach Pedersen
Aalborg University
tbp@cs.aau.dk

ABSTRACT

The enormous amounts of data being generated regularly means that rapidly accessing relevant data from data stores is just as important as its storage. This study focuses on the use of a distributed bitmap indexing framework to accelerate query execution times in distributed data warehouses. Previous solutions for bitmap indexing at a distributed scale are rigid in their implementation, use a single compression algorithm, and provide their own mechanisms to store, distribute and retrieve the indices. Users are locked to their implementations even when other alternatives for compression and index storage are available or desirable. We provide an open source, lightweight, and flexible distributed bitmap indexing framework, where the mechanisms to search for keywords to index, the bitmap compression algorithm used, and the key-value store used for the indices are easily interchangeable. We demonstrate using Roaring bitmaps for compression, HBase for storing key-values, and adding an updated version of Apache Orc that uses bitmap indices to Apache Hive that although there is some runtime overhead due to index creation, the search of hashtags and their combinations in tweets can be greatly accelerated.

CEUR-WS.org/Vol-2324/Paper09-LSubba.pdf

1 INTRODUCTION

Social media platforms like Facebook, Instagram and Twitter have millions of daily active users. On a daily basis, users upload content onto the platforms on a petabyte scale. To keep its user base engaged and active on their platforms, it is critical for such platforms to ensure that users can find content that is relevant to them quickly. Restrictions are not placed on how much information users can upload due to ever decreasing storage costs. Therefore, efficient retrieval from data warehouses becomes just as important as storage. Most social media platforms support hashtags, a keyword containing numbers and letters preceded by a hash sign (#). They allow users to add specific targeted keywords to contents they upload on social media platforms, allowing other users in turn to find them. Its simplicity and lack of formal syntax have allowed for its widespread adoption on multiple platforms. Efficiently finding relevant hashtags and their combinations at the Big data scale is a challenge.

The volume, velocity and variety of data arriving every second means that distributed file systems like Hadoop Distributed File System (HDFS) [19] are preferred for Big data. HDFS supports several file formats like text/comma separated values (CSV) files, column-oriented storage formats like Orc [15], Parquet [16] and row-oriented storage formats like Avro [1]. The difference between row-oriented and column-oriented storage formats lies in how they store contiguous blocks of data. Row-oriented storage formats store successive rows contiguously, whereas column-oriented storage formats ensure that all values of a column are stored contiguously. The former is suitable for transactional (OLTP) workloads, while the latter is suitable for analytical workloads (OLAP). In order to read only relevant data, column-oriented storage formats like Orc and Parquet support predicate pushdown, where the search predicates using =, <, >, <=, >= or != are pushed down to the storage level and are evaluated against its aggregate based indices holding the minimum and maximum values for each column in each block. Such aggregate based indices work fine when the data is ordered, but when the data appears unordered or is skewed, they are prone to false positive results. To alleviate this problem [20] proposed columnar imprints which scans the entire column to create bit vectors for every cache line of data. A bit is set within a bit vector if at least one value occurs within the corresponding bin. As a result, a unique imprint of the cache line is created providing a coarse-grained view for the entire column. However, neither the default aggregate based indices of Orc nor the column imprint index supports the indexing of substrings like hashtags that can exist within non-numeric columns. For queries like SELECT tweet FROM table WHERE string LIKE "%#hashtag%" there is no alternative but to read every single shard of the dataset, which is not a practical approach for large datasets as it is incredibly time consuming.

The compressible nature of bitmap indices, the ability to perform hardware assisted logical operations (AND, OR, XOR) on them and their lightweight nature when compared to tree-based indices make them ideal for indexing hashtags under such conditions. Distributed bitmap indexing frameworks are not a new concept, and several have been developed [10, 13, 17]. However, they are rigid in the use of a specific bitmap compression algorithm and provide their own implementation to store, distribute and retrieve the bitmap indices. A flexible system where the compression algorithm to use and the mechanism to store, distribute and retrieve bitmaps can be easily swapped to the state of the art systems is desirable. Such a system allows for the reuse of existing technologies, and better alternatives can be swapped in when available. With this paper, we develop and evaluate a lightweight and flexible bitmap indexing framework which can be incorporated into existing big data technologies. In this paper, our main contributions are

(1) An open source, lightweight and flexible distributed bitmap indexing framework for big data which integrates with commonly used tools incl. Apache Hive and Orc. Users can easily plug-in their desired functions to find keys to index, bitmap compression algorithm and key-value store. Otherwise, they may use the default setup consisting of Roaring bitmap index, HBase as the key-value store and provide their specific method to find indexable keys.
(2) A demonstration of how the search for substrings like hashtags in tweets can be greatly accelerated by using our bitmap indexing framework. The storage costs for bitmap indices are minimal, however there is runtime overhead due to index creation.

The paper is structured as follows. Section 2 provides background information on the technologies used in our framework. Section 3 presents the related work. Section 4 describes the implementation details for our indexing framework. Section 5 presents

© 2019 Copyright held by the author(s). Published in the Workshop Proceedings of the EDBT/ICDT 2019 Joint Conference (March 26, 2019, Lisbon, Portugal) on CEUR-WS.org.

the experiments conducted with our distributed indexing framework. Finally, Section 6 concludes our work.

2 BACKGROUND

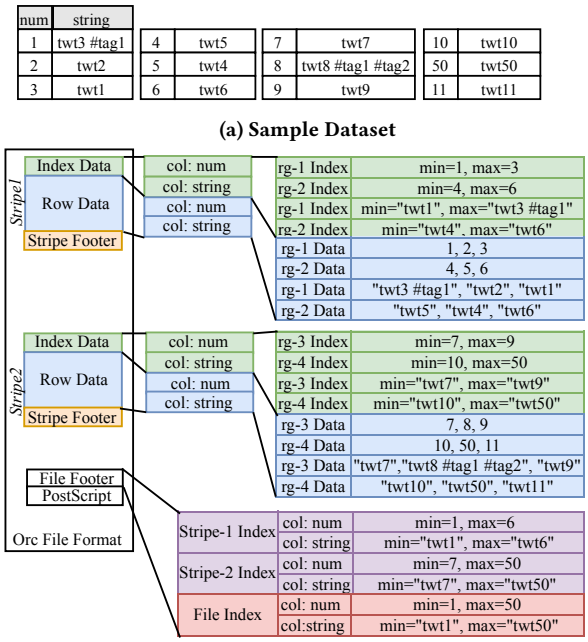
2.1 Apache HBase

A key-value database stores data as a series of keys-values where the key acts as a unique identifier and maps to a value in the database. Both the key and value can be either simple types or complex types as supported by the key-value database. The three major operations that define a key-value database are put(key, value), get(key), delete(key). Apache HBase [9] is an open-source distributed wide column store providing key-value store operations on top of HDFS. HBase is used for low latency read/write operations on large datasets. Logically, data in HBase is organized as labeled tables containing rows and columns, each row is defined by a sorting key and an arbitrary number of columns. Several other open source key-values databases are available [5, 11, 14, 18].

2.2 Apache Orc

Apache Orc is a columnar-oriented file format for Hadoop that is both type aware and self-describing [15]. It is optimized for reading data and creates indices at multiple levels to find relevant data efficiently. A low-level example of the Orc file format and the three levels of its aggregate based indices are shown in Figure 1. We use a small dataset in Figure 1a with 12 records (each has two attributes num and string) to illustrate how a dataset is stored as an Orc file, including its aggregate based indices. An Orc file comprises of independent units called stripes, each having the default size of 64MB. Each stripe is further composed of groups of rows called rowgroups with the default size of 10,000 rows. In our example, a rowgroup size of three is used resulting in an Orc file containing two stripes and four rowgroups. For every stripe, the index data streams denoted by green blocks store index information about columns for every rowgroups within the stripe. For both numeric and non-numeric attributes, the (min, max) values within the rowgroup are used as indices. In case of non-numeric attributes, the values are compared lexicographically to determine the max and min value. The rowgroup data streams denoted by the blue blocks contain the actual data stored in rowgroups within stripes. Each stripe contains a stripe footer with information on the index stream, data stream and encoding information for each column in the stripe. The rowgroup indices within a stripe are used to produce the indices at stripe level denoted by the purple blocks, and the stripe level indices are used to generate the file level index denoted by red blocks. The file level and stripe level indices are stored in the file footer section of the Orc file. Additionally, information about the datatype of every column in the file, number of stripes and the number of rowgroups in every stripe are stored in the file footer. Lastly, the information regarding how to interpret an Orc file including the version of the file, the length of the file footer and the compression used is stored in the postscript section.

In our example, the file level index contains (min=1, max=50) for the column num and (min="twt1", max="twt50") for the column string. Queries with filter predicates like SELECT * FROM table WHERE num = 51 or string = "twt51" search for keys outside the min-max range and are stopped immediately. However, in cases where data is unordered or skewed, aggregate based indices are prone to false positives. For instance, in our example,



(a) Sample Dataset

(b) Sample Dataset stored as Orc file

Figure 1: Orc File Structure

rowgroup-4 data contains skewed data, and its index information is distorted for both columns. Queries searching for num = 25 or string = "twt25" end up reading data streams from the Orc file even though no stripes or rowgroups in the Orc file hold those values. [20] proposed the columnar imprint index to solve this problem for numeric columns. However, neither the default aggregate based indices of Orc or the columnar imprint index supports the indexing of substrings that can exist within non-numeric columns. For example, our sample dataset contains tweets and some rare hashtags. There is no way to accelerate queries like SELECT * FROM table WHERE string LIKE "%#tag1%" or (AND and OR) operations of hashtags.

In addition to aggregate based indices, Orc also supports Bloom filters [2] on its columns. Bloom filters are highly space efficient probabilistic data structure for determining set membership. Compared to aggregate based indices and Columnar imprints, the probabilistic nature of bloom filter means that false positives are possible but false negatives are not. Although the false positive probability is configurable on Orc, bitmap indices do not suffer from this problem.

3 RELATED WORK

A bitmap index is a special kind of index where if a dataset contains N records and an attribute A has D distinct values, the bitmap index generates D bitmaps having N bits each. Each bit in the bitmaps is set to "1" if the record contains that value otherwise, the bit is set to "0" [21]. Bitmap indices can be compressed significantly and require less space than other conventional tree-based indices. In addition, hardware supported bitwise operations (AND, OR, NOT and XOR) can be utilized on bitmap indices in order to speed up queries. Based on run-length encoding (RLE) bitmap compression schemes WAH [22] and PLWAH [6] have been proposed to reduce the space occupied by bitmap indices. [22] proposed the Word-Aligned Hybrid (WAH) bitmap compression, where a sequence of consecutive bits of ones or zeros can be

represented with their bit value and a count indicating the length of the sequence. WAH runs are comprised of a fills and tails. A fill is a set of similar bits that is represented as a count plus a bit value indicating whether it is a zero fill or a one fill. Next, the tail is a mixture of zeros and ones, which are represented without compression. [6] observed that WAH compression runs were never long enough to use all the bits allocated for the run-length counter. Hence, they proposed the Position List Word Aligned Hybrid (PLWAH) compression, which uses those unused bits to hold the position list of set/unset bits that follow a zero or one run. Thus, if a tail following a fill differs only by a few bits, the fill word can encode the difference between the tail and fill. The size of PLWAH bitmaps are often half that required for WAH bitmaps, and PLWAH was found to be faster than WAH. While compression algorithms based on run-length encoded compression perform better on sparse bitmaps, where most of the bits are 0's, they are not so effective on dense bitmaps where most of the bits are a combination of 0's and 1's. In addition, they have slow random access and cannot skip sections of the bitmap. Therefore, [3] developed a hybrid compression technique called Roaring bitmaps that uses packed arrays and uncompressed bitmaps in a two-level index. It separates the data and divides it into sparse and dense chunks. The dense chunks are stored using bitmaps while the sparse chunks are stored using a packed array of 16 - bit integers. It supports fast random access, and in experiments where it was compared to WAH, compresses several times better and was found to be faster. However, compared to RLE compression algorithms, Roaring bitmap had limitations regarding the compression of long compressible runs. Therefore, a third type of container was added to support such runs of consecutive values [12] making Roaring several times faster than the RLE based (WAH) while also compressing better.

While bitmap indices were introduced to expedite queries in traditional centralized systems, there are challenges when applying them to Big data platforms utilizing distributed file systems (DFS). In such situations, local indices will be created on every computing node, and a mechanism is required to create and maintain a global index for expediting queries. [10, 13, 17] have proposed scalable distributed bitmap indexing frameworks. [13] proposes the Bitmap Index for Database Service (BIDS) framework for large-scale data stores. The framework utilizes an adaptive indexing technique, which uses either WAH, bit-sliced encoding or partial indexing depending on the data characteristics to reduce index size. Their indexing scheme favors the creation of the maximum number of indexed attributes so that a wide variety of queries can be supported. The compute nodes are organized according to the Chord protocol, and the indexes are distributed across the nodes using a load balancing mechanism. In Apache Hive [10], data is stored as logical tables, the tables themselves are stored as files distributed in HDFS and the metadata is stored in the Hive metastore. The bitmap indices for Hive tables are stored in index tables with columns containing the indexed column, the name of the block storing the data, offset within the block and bitmap index for the column values. Hive uses its metastore and both tables to process queries on its bitmap indexed tables. It uses an enhanced version of WAH to compress bitmaps and does not support the indexing of substrings from string columns. The work that closely resembles ours is the distributed bitmap indexing framework Pilosa [17]. Pilosa uses a modified version of Roaring bitmap based on 64-bit integers and divides each bitmap index into frames, views and fragments. Pilosa runs as a cluster of one or more nodes, and there is no designated master node. The executor processes the call for all relevant slices on the local node and concurrently issues requests to process the call for slices which reside on remote nodes in the cluster. Once local and remote processing has ended, it performs any aggregation or reduction work and returns the results.

All the previous indexing frameworks use a fixed compression algorithm, but a flexible framework where the compression algorithm can be substituted is desirable as better compression algorithms are developed and released. Also, all of them lock users to their specific implementation to store, distribute and retrieve bitmap indices in a distributed setting. However, a framework where users can use any key-value store allows greater flexibility as state of the art key-value stores can be utilized. For example, HBase and Hive are regularly used together on the same Hadoop cluster and Hive provides storage handlers that allow Hive statements to access HBase tables.

4 SYSTEM

In this section, we present our bitmap indexing framework. We will look at how the index creation process takes place for datasets in Hive, and how the bitmap indices are used during query processing. Finally, how the indexing framework can be used on Hive is discussed.

4.1 System Architecture

Hive [10] is a data warehouse solution running on Hadoop [8] that allows users to use the query language HiveQL to write, read and manage datasets in distributed storage structures. It supports the Orc file format as one of its underlying storage formats. The system architecture for Hive and our indexing framework is shown in Figure 2. HiveQL queries are submitted to Hive through the Hive clients. The queries are received by the driver, then the compiler is used to parse, type check and semantically analyze the queries with schema information in the metastore. An execution plan is created for the query and an optimizer optimizes the execution plan using solutions like column pruning, predicate pushdown (PPD) and pipelining. Finally, the executor executes the optimized execution plan as jobs on Hadoop. Hive supports query execution via three execution engines (MapReduce, Tez and Spark). The job scheduling and resource management tasks are handled by YARN [23]. If Orc use is enabled, all the engines create jobs that use the Orc reader/writer to read and write files in HDFS. As aggregate based indices are prone to false positives, we added the bitmap indexing framework to the Orc reader/writer to support more accurate indices. The framework is agnostic to the execution engine. Users can use our implementation that searches for hashtags to generate <key, bitmap> values, uses the state of the art Roaring bitmap for compression and HBase as its key-value store. Alternatively, users can easily replace the default implementation of the indexing framework with their desired bitmap compression algorithm, key-value store and functions to search for keys to index by uploading a custom Jar file to the working list of Hive's Jar file.

As bitmap indices are key-value pairs where the key is the search key to be indexed and the value is its bitmap representation, the indexing framework uses a key-value store for persistent storage of the indices for Orc files. For our implementation, HBase [9] was chosen as the key-value store due to its low input/output latency and interoperability with Hadoop, but other key-value stores can be easily swapped in. When Hive stores datasets as Orc files, the indexing framework uses functions defined by the

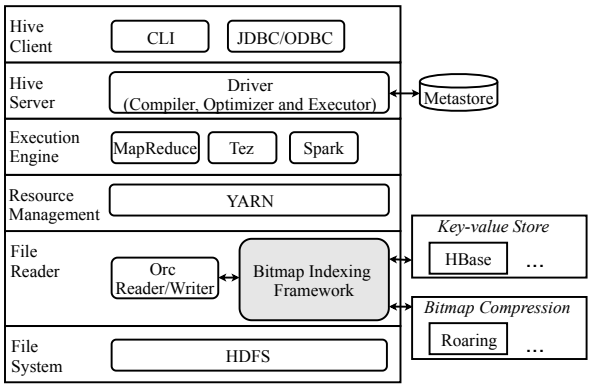


Figure 2: Orc Bitmap Indexing Framework

user to find keys, generates state of the art Roaring bitmap indices and stores them in HBase. For Hive queries that read from tables stored as Orc files, the indexing framework uses the search predicate passed during predicate pushdown to retrieve and process relevant bitmap indices from HBase. After processing the bitmaps, the indexing framework allows Hive to skip irrelevant stripes and rowgroups. Note that our indexing framework only controls which stripes and rowgroups are accessed, the query processing itself is done entirely by Hive's own execution engine.

4.2 Bitmap index Creation

As Orc files are composed of stripes and each stripe is composed of rowgroups, a bit in a bitmap represents the presence or absence of a key in a tuple at a certain row number in a rowgroup within a stripe. The stripe number and rowgroup number can be determined from a tuple's row number provided that the maximum number of rows that can fit into a rowgroup is known [default for Orc: 10,000] and the maximum number of rowgroups per stripe *(mrgps)* is consistent across the Orc file. However, by default Orc uses a stripe size of 64MB and depending on the nature of the dataset, the number of rowgroups across stripes is unlikely to be consistent. In order to ensure consistency across all stripes, *ghost* rowgroups can be added to stripes that contain a smaller number of rowgroups than the maximum number of rowgroup per stripe. Ghost rowgroups do not exist in the Orc files but are added only during the bitmap index creation process. Once the number of rowgroups across the stripes has been made consistent, the maximum rowgroups per stripe (*mrgps*), rows per rowgroup (*rprg*) and row number for a particular tuple (*rn*) can be used in the integer divisions in equation (1) to determine the containing stripe number (*str*) and rowgroup number (*rg*).

$$\begin{aligned} str &= rn/(mrgps * rprg) \\ rg &= (rn \bmod (mrgps * rprg))/rprg \end{aligned} \tag{1}$$

The approach is similar to the Hakan factor [7] for bitmap indices used by the Oracle DBMS, where the Hakan factor refers to the maximal number of rows a data block can store and is used to determine the data block which contains a row.

The bitmap index creation process documented in Algorithm 1 takes place concurrently with the process of writing datasets into a Hive table as Orc files. When the dataset is being written into a Hive table, several mappers are run and each mapper processes a shard of the dataset, the shard size ranges between 50 and 1000 MB under default MapReduce and Hive settings. Our algorithm takes as input a shard being processed by the

Algorithm 1: Bitmap Index Creation

input : *shard* shard for mapper, *col* column to be indexed, *udf* user defined function to find keys
output : *kBms* list of all keys their bitmaps
Algorithm `createIndex(`*shard, col, udf*`)`

```
1  uKeys=List<String>                          /*unique keys*/
2  rnR=bitmap()                                /*rownumbers in bitmap*/
3  prnR=bitmap()                               /*padded rownumbers in bitmap*/
4  lstR=List<bitmap>                           /*list of bitmaps*/
5  kBms=List<String,bitmap>                    /*list of keys and bitmaps*/
6  for i ← 1 to shard.size do
       /*Default writing process of Orc*/
7      createRowNrBitmap(i, shard.get(col, i))
8  addGhostRowgroups()
9  createBitmapIndex()
11 return kBms
```

Procedure `createRowNrBitmap(`*rowNr, col*`)`

```
1  nr=0
2  bm = new bitmap()
3  keys = udf(col)
4  if keys!=null then
5      foreach key in keys do
6          if uKeys.exists(key) then nr = uKeys.get(key)
7          else nr = uKeys.add(key).getSize()
8          bm.add(nr)
9      lstR.add(bm)
10     rnR.add(rowNr)
```

Procedure `addGhostRowgroups()`

```
1  rownr=0
2  for j ← 1 to shard.size do
3      if rnR.contains(j) then prnR.add(rownr++)
4      if isStpBnd(j) then rownr += addPadding (j, mrgps,
         rprg)
```

Procedure `createBitmapIndex()`

```
1  for k ← 1 to prnR.size do
2      setBits = lstR.getAt(k).toArray()
3      foreach setbit in setBits do
4          key = uKeys.get(setbit);
5          if kBms.exists(key) then
6              kBms.add(key, kBms.get(key).add(k))
7          else kBms.add(key, new bitmap(k))
```

mapper, the column name to be indexed and the user-defined function (UDF) set by the user to find keys within the column. In lines 6-7, as the columns of the dataset are being written to an Orc file, *createRowNrBitmap* is executed on the column to be indexed. In line 3 of *createRowNrBitmap*, the UDF set by the user is used to find keys in the column. Then in lines 5-8, each unique key is stored in a list and a roaring bitmap is created to identify the position of the keys in the unique list. Finally, the roaring bitmap identifier of keys is added to the list *lstR* and the row number of the column containing the keys is added to the roaring bitmap *rnR*. The default value of 10,000 is used for rows per rowgroups (*rprg*) and the maximum rowgroups per stripe (*mrgps*) is determined at the end of the default writing process

rownr	tweet	rownr	tweet
0	twt0 #tag1	11	twt11
1	twt1	12	twt12
2	twt2	13	twt13
3	twt3	14	twt14
4	twt4	15	twt15
5	twt5	16	twt16 #tag1#tag2
6	twt6	17	twt17
7	twt7 #tag2		
8	twt8		
9	twt9		
10	twt10		

(a) Sample dataset

blk0	str0	rg0	0	twt0 #tag1
			1	twt1
		rg1	2	twt2
			3	twt3
		grg0	4	
			5	
	str1	rg2	6	twt4
			7	twt5
		rg3	8	twt6
			9	twt7 #tag2
		grg1	10	
			11	
blk1	str2	rg4	12	twt8
			13	twt9
		rg5	14	twt10
			15	twt11
		rg6	16	twt12
			17	twt13
	str3	rg7	18	twt14
			19	twt15
		rg8	20	twt16 #tag1 #tag2
			21	twt17
		grg2	22	
			23	

(b) Sample dataset in Orc including ghost rowgroups

block	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
stripe	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3
rowgroup	0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1	2	2
rownr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
#tag1	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0
#tag2	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0

(c) Bitmap representation including ghost rowgroups

Key	Value	
	WorkerNode-OrcFilename	..
mrgps	3	..
#tag1	Roaring(**1**,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,**1**,0,0,0)	..
#tag2	Roaring(0,0,0,0,0,0,0,0,**1**,0,0,0,0,0,0,0,0,0,0,0,**1**,0,0,0)	..

(d) Key and bitmaps with compression algorithm

Figure 3: Orc Index creation process.

of Orc. After all the columns of the shard have been written, ghost rowgroups are added in *addGhostRowgroups* to ensure that the number of rowgroups per stripe is consistent. In lines 2-4, the row numbers from the roaring bitmap *rnR* are moved to the padded roaring bitmap *prnR*. *addPadding* is used to calculate the padding at the stripe boundaries for stripes that contain fewer rowgroups than *mrgps*. Next, in *createBitmapIndex*, all the keys and the row numbers found are converted to key-bitmap pairs and stored in the list *kBms*, and finally returned by *createIndex*.

Filling a Hive table with data ends with Orc files being created for the shards processed by each mapper. Next, the keys identified by the user-defined function and the bitmaps associated with each key are stored in a HBase table. The index keys are used as HBase keys and the bitmap for the keys are stored as values using column identifiers. Using a column identifier helps identify which worker node the Orc file resides and what Orc file the bitmap is associated with. Its use allows bitmaps for keys from other Orc files to be stored in the same HBase table. The maximum number of rowgroups per stripe *(mrgps)* for an Orc file is also stored in HBase using column identifiers.

Figure 3 shows an example of the bitmap index creation process. In Figure 3a, the sample input dataset consists of two columns (rownr, tweet) and 18 tuples. Figure 3b, shows the dataset stored as an Orc file in HDFS as two HDFS blocks {blk0, blk1}, containing four stripes {str0, str1, str2, str3} and a total of 9 rowgroups. In order to create a file level bitmap index, the maximum number of rowgroups per stripe across the Orc file is determined by reading the Orc file footer. For the sample dataset there are 4 stripes and stripe str2 with its 3 rowgroups has the maximum number of rowgroups. Therefore, ghost rowgroups are added to stripes str0, str1 and str3 to make them consistent with stripe str2. In Figure 3c, we see the bitmap representation for the hashtags #tag1 and #tag2 where the shaded portions represent the ghost rowgroups. Finally, the keys, their bitmaps , the column identifier, and the maximum rowgroups per stripe is stored in an HBase table as shown in Figure 3d.

4.3 Bitmap Index Processing

Based on the queries submitted to Hive, it can use our indexing framework to retrieve and process bitmap indices stored in a key-value store to prevent access of irrelevant stripes and rowgroups

Algorithm 2: Bitmap Index Processing

```
input  : ast search predicates in abstract syntax tree, mrgps
         maximum rowgroups per stripe, rprg rows per
         rowgroup, stripes list of stripes being processed;
output : strrg stripes and rowgroups to read;
Algorithm useIndex(ast, mrgps, rprg, stripes)
1   rbm = ProcessPredAST(ast, stripes.start, stripes.end);
2   foreach setBit in resultBM do
3       rownr = getRowNr(setBit);
4       strrg.add(getStripe (rownr), getRg (rownr));
5   return strrg;
```

from the underlying Orc files. HiveQL queries run by users on Hive are translated to MapReduce jobs [4] and each mapper will process a split of the Orc files. To improve performance, Hive employs a hybrid splitting strategy to process Orc files. If the number of Orc files are less than the expected number of mappers, to improve parallelism, Orc file footers are read to provide each mapper a split of the stripes to process. However, if the average file size is less than the default HDFS block size, each Orc file will be treated as a split and a mapper will receive all its stripes. Hive calls the former strategy ETL, while the latter is called BI. As our indexing framework creates bitmaps at the file level, for the BI strategy, each mapper is processing the entire Orc file and bitmaps also cover the entire Orc file. However, for the ETL strategy mappers are processing only a portion of the stripe from Orc files. Therefore, parts of the bitmap covering non-relevant stripes need to be removed.

The bitmap index usage is described in Algorithm 2. As input the algorithm takes the predicates to search for in the form of an abstract syntax tree (AST), the maximum rowgroups per stripe, rows per rowgroup and information about the stripes being processed by the mapper. In line 1, the search predicate AST is traversed using *ProcessPredAST*, the procedure executes recursively pulling bitmaps for each predicate using the column identifier from HBase. Next, the bitmaps are sliced using the stripe start/end information if Hive is processing the query under ETL mode, under BI mode no slicing occurs as the mapper is processing all stripes of an Orc file. Then, logical operators (LIKE,

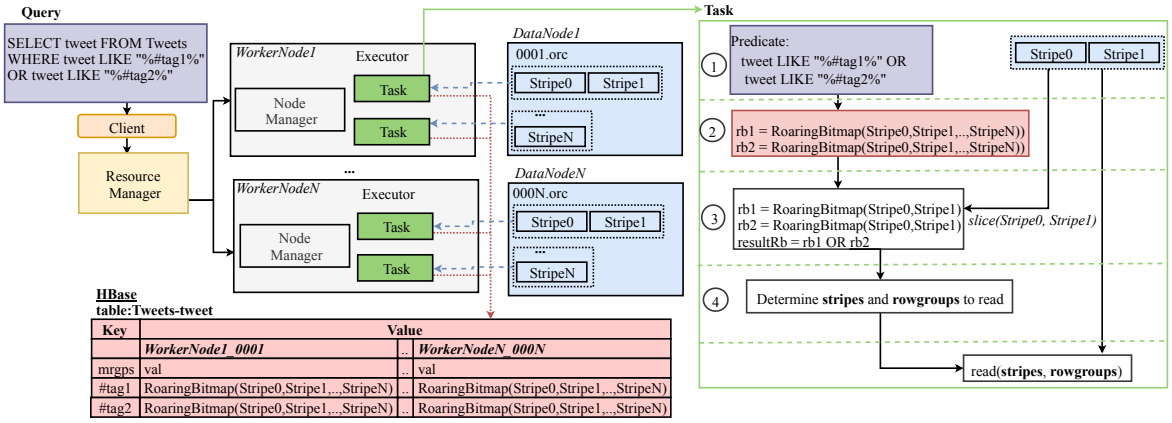


Figure 4: Orc Index Processing.

OR, AND or XOR) between predicates are applied to the bitmaps to retrieve the result bitmap *rbm*. Finally, in lines 2-4, for all set bits in the result bitmap, the row numbers representing the set bits are used in *getStripe* and *getRg* to determine the stripes and rowgroups to read and stored in the list *strrg* and returned in line 5.

Figure 4 provides an example of how the indexing processing framework enables Hive to skip irrelevant blocks during query execution if bitmap indexing is enabled. The example shows a cluster containing Hadoop, Hive and HBase with N worker nodes. Tasks are executed by the worker nodes in the cluster to answer queries. Hive contains a table Tweet stored into N Orc files and the HBase table (Tweets-tweet) contains the indexed keys, their bitmaps and the maximum stripes per rowgroups for each Orc file in the worker nodes. A select query is executed on the tweet table in Hive with two predicates. Each task processing the query will receive and pass the predicates to the underlying storage structure. Hive is running using its ETL strategy and each task will process only a portion of the stripes. One of the tasks is running in the executor of WorkerNode1 and is processing stripe0 and stripe1 of the Orc file 0001.orc from DataNode1. The task uses the keys (#tag1 and #tag2) from the predicates and the column identifier WorkerNode1_0001 to retrieve the bitmap values for the keys from HBase. As the task is not processing all the stripes within an Orc file, the bitmaps retrieved from the key-value store are sliced to cover only the stripes being processed. Next, any logical operation between the keys are applied to their bitmaps to retrieve the result bitmap. Finally, the stripes and rowgroups to read are determined by applying equation (1) to the set bits of the result bitmap, allowing the skipping of irrelevant stripes and rowgroups.

4.4 Hive Integration

The indexing framework is made publicly[1] available for use under the Apache License 2.0[2]. The index creation and processing functionality are integrated into the *WriterImpl* and *RecordReaderImpl* classes of Hive's Orc writer and reader components. During data insertion to a Hive table that uses Orc file format for storage, Orc files are created across the cluster storing table data. If bitmap indexing has been enabled, the bitmap index creation process of our framework discussed in 4.2 hooks into the Orc file writing process to create and store indices for predefined fields of the table. Similarly, during query execution, if bitmap indexing has been enabled, the bitmap index processing component of our framework discussed in 4.3 runs for the pushed predicates on the Hive table and the stripes and rowgroups to read from each Orc file is determined. To use the indexing functionality in a new Hive installation, the default Orc reader and writer packages in Hive will have to be replaced with ones containing our indexing functionality. As Hive allows users to write and upload their custom built Jar files, the bitmap indexing framework can be uploaded alongside the Hive Jar to its list of working Jar files, and the Orc reader and writer can reference it.

The interface of our bitmap indexing framework is provided in Listing 1. Users can use our implementation or plugin their specific implementation. In line 3, the function *findKeys* is used by the framework to find indexable keys in non-numeric columns. Our implementation returns hashtags as keys. In line 5, the boolean function *isProcessable* is used to determine if a predicate is processable by the framework or not. In case the predicate cannot be processed, Hive's default query processing is run. In line 7, the function *createBitmap* is used by users to decide which bitmap compression algorithm to use to index their data and also implements Algorithm 1. Our implementation uses Roaring bitmaps for compression. Finally, in lines 9 and 11, functions *storeKeyBitmap* and *getKeyBitmap* are used to store the key-bitmap pairs into a key-value store and to return a byte value of a bitmap for a particular key. Our implementation uses Roaring for compression and HBase as the key-value store. To replace the default implementation, users need to override our implementation, rebuild the indexing framework and deploy the Jar file to Hive's working list of Jar files.

Listing 1: Interface for Indexing framework

```
1  public interface IBitmapIndexingFramework {
2      /* find indexable keys in column fields */
3      String[] findKeys(String column);
4      /* determine if search predicate is usable by framework */
5      boolean isProcessable (String ast);
6      /* create bitmap index from rownumber and column */
7      boolean createBitmap(int rowNr, String column);
8      /* store all key-bitmap pairs in key-value store */
9      boolean storeKeyBitmap(String[] args);
10     /* get bitmap index for a single key */
11     byte[] getKeyBitmap(String[] args);
12 }
```

Listing 2 shows how users can use the Hive console to use our indexing framework. The first statement in line 2 creates a

[1]https://github.com/lawansubba/lbif
[2]https://www.apache.org/licenses/LICENSE-2.0

Hive table with two columns with Orc as the underlying storage structure. In lines 3-6, a flag is set enabling bitmap indexing, the Hive table with the column to index is declared, and what bitmap indexing implementation of Listing 1 to use is declared. Finally, an insert statement like line 6 will fill the Orc based table, while our indexing framework uses the set bitmap indexing implementation to find keys and creates <key, bitmap> pairs, which are stored in the predetermined key-value store. How the bitmap indices can be used is shown in lines 8-11. Lines 8-10 enable predicate push down, the use of indices based filtering and bitmap indexing functionality. Lastly, a select query like in line 11 will use the search key #tag1 in Algorithm 2 to return only relevant results. These settings can also be defined in the configuration file of Hive so that users don't have to specify them every time.

Listing 2: HiveQL for Bitmap Index creation/use

```
1  /* bitmap index creation */
2  CREATE TABLE tblOrc(id INT, tweet VARCHAR) STORED AS ORC;
3  SET hive.optimize.bitmapindex=true;
4  SET hive.optimize.bitmapindex.format=tblOrc/tweet/;
5  SET hive.optimize.bitmapindex.framework='com.BIFramework';
6  INSERT INTO tblOrc SELECT id, tweet FROM tblCSV;
7  /* bitmap index usage */
8  SET hive.optimize.ppd=true;
9  SET hive.optimize.index.filter=true;
10 SET hive.optimize.bitmapindex=true;
11 SELECT * FROM tblOrc WHERE tweet LIKE '%#tag%';
```

5 EVALUATION

The indexing framework is integrated into the Orc reader and writer components of Hive 2.2.0 and then installed in a fully distributed cluster on Microsoft Azure with one node acting as master and seven nodes as slaves. All nodes are in the East US region and use Ubuntu OS with 4 VCPUS, 8 GB memory, 192 GB SSD. HDFS 2.7.4 is used for distributed file system and HBase 1.3.1 for persistent storage of key-value stores. Details about the datasets used for our experiments are provided in Table 1. All three datasets contain tweets collected from the Twitter API for different months in 2013. The schema for the dataset contains 13 attributes [tweetYear, tweetNr, userIdNr, username, userId, latitude, longitude, tweetSource, reTweetUserIdNr, reTweetUserId, reTweetNr, tweetTimeStamp, tweet]. The sizes of datasets are 55, 110 and 220 GB respectively and the size of the datasets determine the number of tuples, the total number of hashtags and the total number of unique hashtags found in each dataset. The number of Orc files the dataset is stored into is determined by the MapReduce configurations like number or mappers, the number of cores available for processing and amount of RAM available for each mapper. The three datasets are stored as 66, 128 and 224 separate Orc files respectively across the cluster, each file containing a different number of stripes and rowgroups.

If a query returns a significant portion of the dataset, at a certain threshold, the indexed scan will be just as or more time consuming than a full scan of the dataset. Therefore, for any indexing system, it is important to investigate when this threshold is reached. The three datasets are stored in Hive as Orc based tables, and their indices are stored in HBase. In order to investigate the threshold, indices for each dataset are analyzed to find hashtags for queries that access tuples in a geometric sequence (1,2,4,8,..) until the maximum sequence number is found. If a hashtag does not exist that accesses tuples for a sequence number, the hashtag accessing the closest higher sequence number is used. The discovered hashtags are used in LIKE queries to record execution times from Hive tables under the default mode and using bitmap indices. Next, the very same hashtags are OR'd together successively in queries to determine execution times for OR-LIKE queries. Lastly, hashtags are discovered and used in self JOIN queries. There are not enough common hashtags between tweets to perform AND operations and test for the threshold. Therefore, AND operations have been excluded from the experiments. Each query was run a total of five times, and the median value was taken as the execution time. Hive runs queries on all datasets using ETL strategy. Note that the experiments show execution times/stripes and rowgroups accessed by the (LIKE, OR-LIKE and JOIN queries) and the number of matching tuples accessed before a group by operation is performed. The three types of queries used in our experiments are shown below.

LIKE: SELECT tweetSource, COUNT(*) as Cnt
FROM TableName
WHERE tweet LIKE '%hashtag1%'
GROUP BY tweetSource;

OR-LIKE: SELECT tweetSource, COUNT(*) as Cnt
FROM TableName
WHERE (tweet LIKE '%hashtag1%'
OR tweet LIKE '%hashtag2%',...)
GROUP BY tweetSource;

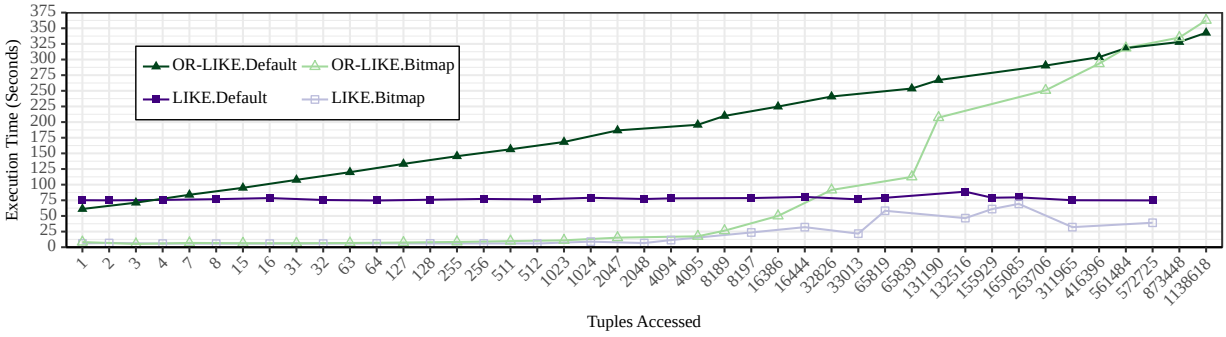
JOIN: SELECT t1.tweetSource, COUNT(*) as Cnt
FROM TableName AS t1 JOIN TableName AS t2
ON (t1.tweetNr = t2.reTweetNr)
WHERE t1.tweetNr != -1
AND (t1.tweet LIKE '%hashtag1%')
AND (t2.tweet LIKE '%hashtag1%')
GROUP BY t1.tweetSource;

Figure 5a shows the execution times for the LIKE and OR-LIKE queries for the largest dataset Tweets220 using both the default mode and bitmap indices. The results for the other two datasets (Tweets55 and Tweets110) show similar results and do not add new information and are not included here. Under default mode, all stripes and rowgroups of the dataset are read and processed. In case of LIKE queries, a single like comparison is done on the tweet column and the tweetSource is used in the group by only if the tweet contains the hashtag. Therefore, the execution times remains nearly constant for LIKE queries accessing between 1 and 572,725 tuples. In contrast, for OR-LIKE queries an increasing amount of LIKE operations are performed on the tweet, and then an OR operation is performed between the results. Therefore, as more LIKE conditions are added in the OR-LIKE query, the execution time for OR-LIKE queries increases. Compared to the default mode, we observe that if the queries are highly selective, our indexing framework can accelerate execution times for both LIKE and OR-LIKE queries. However, as more tuples are accessed, more stripes and rowgroups are accessed from the Orc files, and as a result there is an increase in execution time.

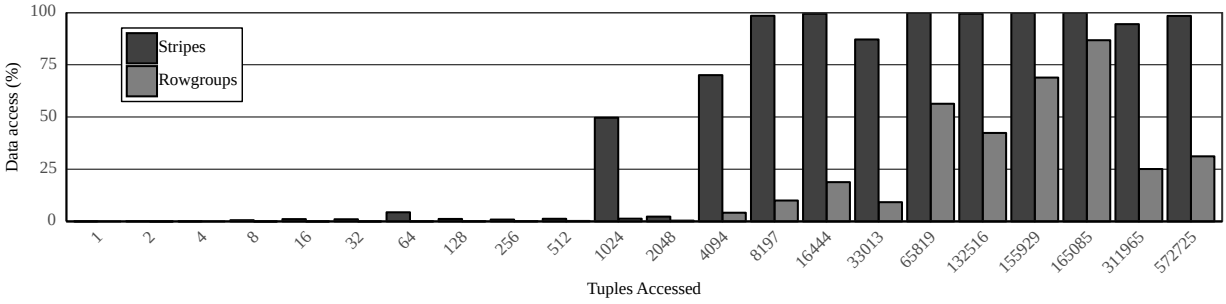
Figure 5b and Figure 5c show the percentage of stripes and rowgroups accessed by the LIKE and OR-LIKE queries when bitmap indices are used. We can summarize that response times when bitmap indices are used are influenced more by the number of rowgroups accessed than the number of stripes accessed. A significant portion of the queries read nearly all the stripes, but only a few queries read almost all the rowgroups and the execution time for those queries are nearly equal to the execution time in default mode. A similar pattern is observed in Figure 5c for OR-LIKE queries when bitmap indices are used. The last three

Table 1: Dataset details

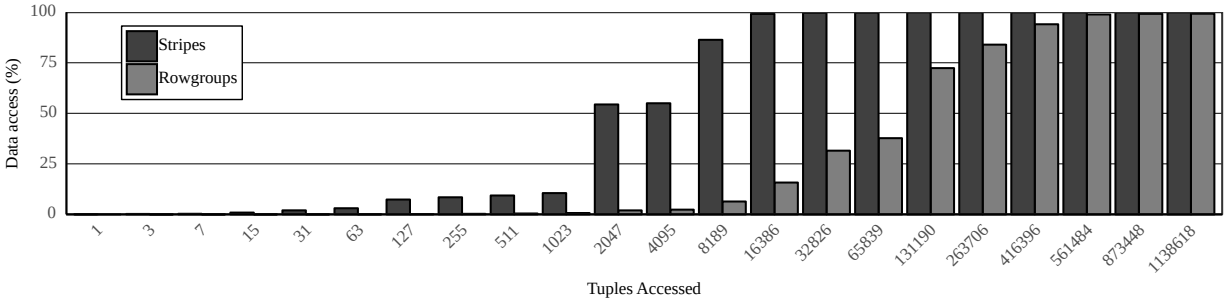
Dataset	Tuples	Total HashTags	Unique Hastags	Orc Files	Stripes	Rowgroups
Tweets55	192,665,259	32,534,370	5,363,727	66	285	19,360
Tweets110	381,478,160	62,281,496	9,063,962	128	624	38,351
Tweets220	765,196,395	126,603,736	16,149,621	224	1342	76,918



(a) Execution times for LIKE and OR-LIKE queries



(b) Stripes/Rowgroups accessed by LIKE queries



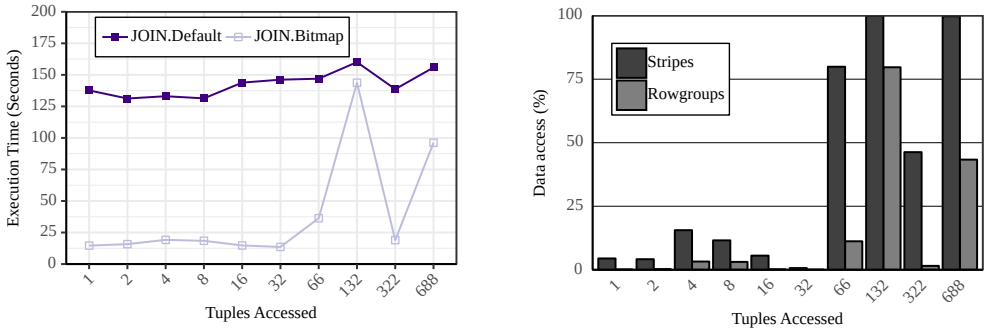
(c) Stripes/Rowgroups accessed by OR-LIKE queries

Figure 5: Query execution times and stripes/rowgroups accessed by LIKE and OR-LIKE queries on Tweets220.

queries access almost all the stripes and rowgroups of the dataset, and the execution time exceeds the default implementation. In such cases, a full scan is preferable to an indexed scan.

The results of the last experiment involving self JOIN queries using both the default mode and bitmap indices are shown in Figure 6a. Similar to our previous findings, we find that query execution times can be greatly reduced for highly selective JOIN queries by using bitmap indices. The amounts of data involved in the JOIN operation from the left table and right table is greatly reduced and thus the improvement in execution times. As the queries involve self JOINs, Figure 6b shows the percentage of stripes and rowgroups accessed in either side of the left and right table of the join. An interesting observation is that the query accessing 132 tuples is accessing significantly more stripes and rowgroups than the query accessing 322 tuples. The reason is that the hashtag used in the former query is much more common than the latter one and exists throughout the Tweets220 dataset, but only 132 tuples exist that satisfy the join condition (t1.tweetNr = t2.reTweetNr). This explains the sudden spike in execution time in Figure 6a for the query that accesses 132 tuples. However, even in this case the execution time using bitmap indices is better than the default mode as the indexed solution is able to skip some irrelevant rowgroups.

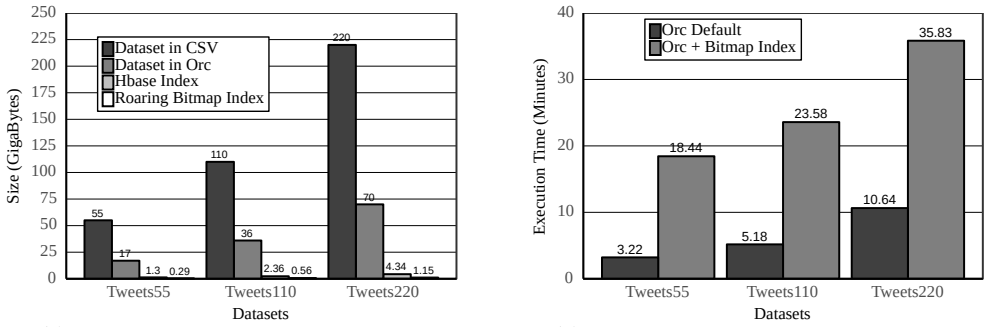
The size of the three datasets in CSV format, their sizes when stored as Orc based tables, the size of their indices when stored in HBase and the size of Roaring bitmap indices are shown in Figure 7a. Compared to the CSV format, the Orc formats using their encoders for the different column types can significantly reduce the storage footprint of each dataset by more than half. The sizes of the Roaring bitmap indices and the HBase tables



(a) Execution times for JOIN queries

(b) Stripes/Rowgroups accessed by JOIN queries

Figure 6: Query execution times and stripes/rowgroups accessed by JOIN queries on Tweets220.



(a) Tweets datasets and their Index sizes

(b) Index creation time for Tweets datasets

Figure 7: Tweets datasets their index sizes and index creation times.

where they are stored are a fraction of size of the datasets in the CSV format and Orc format. However, the index creation process comes with an initial index building cost as shown in Figure 7b. Compared to the default table creation process which stores the datasets as Orc files, our indexing framework scans the datasets for hashtags, creates bitmap indices for each Orc file and stores them in HBase resulting in a 4 to 6 times more expensive table creation process.

6 CONCLUSION

In this paper, a lightweight, flexible and open source bitmap indexing framework is proposed to efficiently index and search for keys in big data. The framework provides a function to search for hashtags, uses Roaring bitmap for bitmap compression and HBase for storing key-values. However, all three components can be easily swapped with other alternatives. The indexing framework was integrated into Hive and tested on a Hadoop, Hive and HBase cluster. Experiments on the cluster using three datasets of different sizes containing tweets demonstrates that the execution times can be significantly accelerated for queries of high selectivity.

ACKNOWLEDGEMENTS

This research has been funded by the European Commission through the Erasmus Mundus Joint Doctorate "Information Technologies for Business Intelligence Doctoral College" (IT4BI-DC) and Aalborg University. All experiments were performed on Microsoft Azure using a sponsorship granted by Microsoft.

REFERENCES

[1] Apache Avro. 2018. Apache Parquet Home. (2018). Retrieved 2018-10-20 from https://avro.apache.org/

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

[3] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2016. Better bitmap performance with roaring bitmaps. *Software: practice and experience* 46, 5 (2016), 709–719.

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

[5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 205–220.

[6] François Deliège and Torben Bach Pedersen. 2010. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *Proceedings of the 13th international conference on Extending Database Technology*. ACM, 228–239.

[7] Hakan Factor. 2018. Oracles Hakan Factor. (2018). Retrieved 2018-10-20 from https://www.databasejournal.com/features/oracle/oracles-hakan-factor.html

[8] Apache Hadoop. 2018. Welcome to Apache Hadoop! (2018). Retrieved 2018-10-20 from http://hadoop.apache.org/

[9] Apache HBase. 2018. Apache HBase Home. (2018). Retrieved 2018-10-20 from https://hbase.apache.org/

[10] Apache Hive. 2018. Apace Hive TM. (2018). Retrieved 2018-10-20 from https://hive.apache.org/

[11] Riak KV. 2018. Redis KV Home. (2018). Retrieved 2018-10-20 from http://basho.com/products/riak-kv/

[12] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. 2016. Consistently faster and smaller compressed bitmaps with roaring. *Software: Practice and Experience* 46, 11 (2016), 1547–1569.

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

[14] Memcached. 2018. memcached - a distributed memory object caching system. (2018). Retrieved 2018-10-20 from http://www.memcached.org/

[15] Apache ORC. 2018. Apache ORC, High-Performance Columnar Storage for Hadoop. (2018). Retrieved 2018-10-18 from https://orc.apache.org/

[16] Apache Parquet. 2018. Apache Parquet Home. (2018). Retrieved 2018-10-20 from https://parquet.apache.org/

[17] Pilosa. 2018. Pilosa Home. (2018). Retrieved 2018-10-20 from https://www.pilosa.com/

[18] Redis. 2018. Redis Home. (2018). Retrieved 2018-10-20 from https://redis.io/

[19] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee, 1–10.

[20] Lefteris Sidirourgos and Martin Kersten. 2013. Column imprints: a secondary index structure. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 893–904.

[21] Kurt Stockinger and Kesheng Wu. 2007. Bitmap indices for data warehouses. In *Data Warehouses and OLAP: Concepts, Architectures and Solutions*. IGI Global, 157–178.

[22] Kesheng Wu, Ekow J Otoo, and Arie Shoshani. 2006. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems (TODS)* 31, 1 (2006), 1–38.

[23] Yarn. 2018. Apache Hadoop YARN. (2018). Retrieved 2018-10-20 from https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html