IC-Data: Improving Compressed Data Processing in Hadoop

Adnan Haider, Xi Yang, Ning Liu, Xian-He Sun
Computer Science Department
Illinois Institute of Technology
Chicago, United States of America
{ahaider3,xyang34,nliu8}@hawk.iit.edu, sun@iit.edu

Shuibing He
Computer School
Wuhan University
Wuhan, China
heshuibing@whu.edu.cn

***Abstract*—As dataset sizes for data analytic applications and scientific applications running on Hadoop increases, data compression has become essential to store this data within a reasonable storage cost. Although data is often stored compressed, currently Hadoop takes 49% longer to process compressed data compared to uncompressed data. Processing compressed data reduces the amount of task parallelism and creates uneven workload distribution both of which are fundamental issues the MapReduce parallel programming paradigm should alleviate. In this paper, we propose the design and implementation of a Network Overlapped Compression scheme, NOC, and Compression Aware Storage scheme, CAS. NOC reduces data load time and hides compression overhead by interleaving network I/O with compression. CAS increases parallelism by dynamically changing a file's block size based on compression ratio. Additionally, we develop a MapReduce Module which recognizes the characteristics of compressed data to improve resource allocation and load balance. Collectively, NOC, CAS, and the MapReduce Module decrease job execution time on average by 66% and data load time by 31%.**

***Index Terms*—Big Data Processing; Data Compression; Hadoop;**

I. Introduction

As we continue to move forward in the big data era, system administrators are using data compression to reduce storage consumption and thus big data applications often have their datasets compressed. The MapReduce paradigm, used for running these applications, has continued to increase in popularity [1]. Hadoop MapReduce, an open source implementation of MapReduce, is in use for both scientific and data analytic applications [2]. Currently, Hadoop processes compressed data on average 49% slower than uncompressed data. However, data compression reduces the amount of network and disk I/O required for processing an application and thus has potential to reduce job runtime, instead of increase it. Both scientific and data analytic applications often have their datasets compressed.

Large scale scientific data processing using MapReduce is common in Biomedical Imaging, Bioinformatics, and High Energy Physics [3] [4] [5]. Bioinformatics applications, in particular Next-Generation Sequencing (NGS), have terabyte sized datasets. The parallelization from Hadoop has allowed these applications to achieve better performance, but as the size of datasets increases, compression needs to be deployed to limit storage consumption. CloudBurst is a prime example of a Bioinformatic application which needs to process large amounts of data [6]. Compression has also become widely adopted by companies who manage large amounts of data.

Large scale data warehouses, such as those at Facebook, compress all their data. Facebook receives 600 TB of uncompressed data as of 2014, which would lead to adding a new rack of storage every three days. But assuming a modest compression factor of 6, the amount of data needing to be stored diminishes to 100 TB which is significantly more reasonable [7]. Without compression, storing the vast amount of data processed in data warehouses, which is expected to reach 8.6 zettabytes by 2018 [8], would be practically infeasible. Twitter, another company dealing with large amounts of data, went so far as to modifying the LZO compression scheme to better fit their needs. This modified scheme is now running 24 hours a day at Twitter [9]. Both Facebook and Twitter use Hadoop to process their compressed data.

Because compression is used across multiple fields of science and industry, applications have their input data compressed or in other words they process compressed data. Many optimizations have been proposed for processing compressed data. Parallel Database Management systems (PDMS) have sophisticated mechanisms to compress data and process compressed data, such as parallel compression during data loading and a compression engine which allows processing of compressed data without decompression [10]. Compared to these complex techniques, Hadoop merely recognizes compressed data so that it can decompress it. This simple mechanism has been shown to perform worse than processing uncompressed data in Hadoop [11], but the reason for the performance decrease has not been studied. Often, the worse performance is falsely attributed to the decompression latency which is needed before running the application code on the data. Although this does incur some overhead, we found that there are other issues which lead to the degraded performance, and when overcome, processing compressed data can be faster than processing uncompressed data because of the reduction in the amount of data needing to be read and transferred.

Two fundamental benefits of the MapReduce parallel programming paradigm, parallelism and load balance, are greatly reduced when processing compressed data. The performance issues are caused by two current design components. First, the Hadoop Distributed File System (HDFS) does not treat the storage of compressed data any differently than uncompressed data even though the processing requirements are quite different. Second, MapReduce's resource allocation is based on compressed data size, when in fact execution time and amount of needed resources depends on the decompressed data size. In detail, processing compressed data has three core issues which lessen the benefits provided by Hadoop:

- The number of map tasks created is determined by the compressed dataset size. Thus, the number of tasks decreases as compression ratio decreases and the amount of data each task must process increases as compression ratio decreases. This results in less tasks which individually process more data resulting in decreased parallelism and increased job runtime.
- The variation between each map task's runtime increases since the duration of a map task depends on the amount of data it has to process which in turn depends on the varying compression ratio for different files. Runtime variation results in long running tasks which prolong job execution time and reduce load balance.
- Compressing data is a significant overhead in warehouse clusters as is the time to load data to clusters and datacenters. Currently, compression and data transfer is done sequentially which further increases the time till users can process their data.

Because of these issues, we propose IC-Data which **I**mproves **C**ompressed **Data** processing in Hadoop, hides compression overhead, and reduces data load time. Our main contributions are:

- We identify, formalize, and quantify the reasons for why a job processing compressed data takes on average 49% longer to execute than processing uncompressed data. The causes are a reduction in the number of parallel tasks and an increase in map task runtime skew.
- We develop a Network Overlapped Compression scheme, which overlaps data compression with network transfer in order to hide compression latency, limit network I/O, and reduce data load time. In addition, we develop a Compression Aware Storage scheme which allows HDFS to recognize compressed data and dynamically change the block size of a file based on the compression ratio. These two schemes are bundled into an HDFS Module which reduces data load time by an average of 31%.
- We develop a MapReduce module which consists of a Compressed Task Divider and a Compressed Block Reader. Together, they allow for efficient processing of compressed data by using the uncompressed data size to determine the number of resources for a job. The module reduces job execution time by an average of 66% across 6 benchmarks and one real application.

The paper is organized as follows: Section II discusses the background information and related work. Section III describes the problems with compressed data processing. The design and implementation of IC-Data is in Section IV. Section V provides the experimental evaluations and Section VI is the conclusion and future work.

II. Background and Related Work

A. Background

Hadoop consists of the Hadoop Distributed File System (HDFS) and Hadoop MapReduce, which processes the data stored in HDFS. HDFS stores files by dividing them into blocks, which by default are 128 MB. Each block is replicated onto multiple datanodes (storage nodes), which by default is 3. A Hadoop MapReduce job can be divided into a map phase and reduce phase. The map phase consists of multiple map tasks which each process a portion of the entire job's input data. Generally, each task processes a single HDFS block or multiple blocks. Once a map task finishes, its output is sent across the network to reduce tasks. Once every map task finishes, the reduce phase can start. The reduce phase calls the user's reduce function and then stores the output to HDFS.

Data compression is used to reduce storage consumption and I/O. Essentially, compression reduces I/O time and increases compute time, a valuable tradeoff in the big data era. Data compression can be used for map input data, map output data, and reduce output data. This paper focuses on compressed map input data, which requires compressed data processing. The amount of reduction in I/O depends on the compression ratio which depends on the patterns in the data and the compression algorithm. We calculate compression ratio as $\frac{CompressedData}{UncompressedData}$. A lower ratio means less storage consumption and reduced disk and network I/O for a job.

If input data is compressed, a map task decompresses the data and then calls the user's map function. There are two types of compression codecs (algorithms). The first type are splittable codecs, meaning if input data is compressed with this type of codec more than one task can process the data in parallel. However if the codec is not splittable, then one task processes the entire input data. This is required because a single task may need data from other portions of the file in order to decompress the data. Non-splittable codecs can cause significant performance issues when the dataset is large. In this paper, we do most tests with Bzip2 since it is one of the few splittable codecs available, has low compression ratios, but also has high decompression and compression overhead.

B. Related Work

Data compression is widely used in both Parallel Database Management Systems (PDMS) and in high performance computing (HPC). In addition, there has been some work dealing with compression in the Hadoop framework.

PDMS have advanced software when it comes to compressed data processing. In 1991, data compression was analyzed for benefits other than reduction in storage consumption [12]. An in production database management system, Vertica, has a compression engine which can process compressed data without decompression [10]. Another major PDMS, can reduce execution time by 50% when processing compressed data [11]. In the HPC field, compression was used to limit network I/O by utilizing idle CPU resources [13]. Another work interleaved data compression with I/O to hide the compression overhead for HPC systems [14].

Compression in Hadoop has achieved lesser attention than its PDMS and HPC counterparts, but there have been few works. One research group analyzed the impact of data compression on power savings [15]. They showed how the ratio of input to intermediate to output data can affect the benefits of compression and how compression ratio should be considered when deciding to compress data. Another work compared the overall performance of Hadoop and PDMS [11]. They mention that Hadoop's compressed data processing was much worse than PDMS's, but the reason for the degradation was not discussed. Facebook created an Optimized Row Columnar (ORC) file format partly to improve compressed data processing [16]. This work is designed for structured data and runs on Hive, which focuses on data queries [17]. Our work focuses on Hadoop, meaning large scale scientific and data analytic jobs. However, the newly identified problems still exist in Hive even with ORC files since Hive uses HDFS, where the issues discussed in the next section stem from.

III. Problems with Compression in Hadoop

In order to better understand the problems associated with compressed data processing, we present a simple mathematical model and then quantify the performance issues.

TABLE I: The mean and standard deviation of compression ratios (2nd and 3rd column). Amount of uncompressed data (MB) processed based on amount of compressed data read (4th and 5th column).

File Type	Mean	Std.Dev.	128MB	256MB
Fatsa	0.253	0.018	472.10≈544.16	942.21≈1077.25
Log	0.172	0.069	530.81≈1256.40	1061.27≈2495.21
CSV	0.212	0.065	461.42≈871.96	922.84≈1741.92

A. *Formulating the Compression Problem*

We present the issues with current compressed data processing using a simple model. Suppose there exists a job which needs to process f amount of uncompressed data. In addition, assume the data is stored compressed with a compression ratio c, where the ratio is calculated as in Section II. Lastly, suppose the amount of compressed data distributed to each task is a, which is often equal to the HDFS block size.

The number of tasks, t, will be equal to $\frac{cf}{a}$ since $c * f$ is the amount of compressed data stored and each task receives the same amount of data. λ is the time it takes to process a given amount of data and equals $\frac{TimeToProcess}{AmountOfDataProcessed}$. The denominator is the processing amount (decompressed data size). We make an assumption that λ is constant for a single job. In actuality, λ varies based on node hardware or cluster utilization. λ is constant to ensure all causes of suboptimal performance is due to compressed data processing. Generally, λ is larger for CPU bound jobs than I/O bound jobs.

We define $S(t)$ as the sum of the latency required to schedule/create t tasks and it is a constantly increasing function with the number of tasks. Lastly, jobs can execute in iterations when the number of available resources, r, is less than the number of tasks, t. We calculate the number of iterations as $\lceil \frac{t}{r} \rceil$, where r

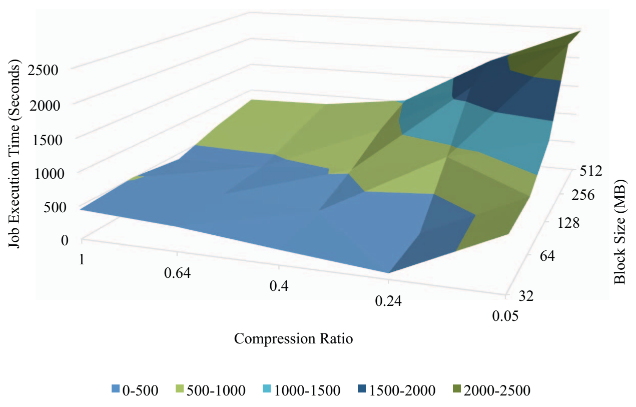

Fig. 1: Relationship between compression ratio and block size.

can be represented as the number of map slots available [18]. The duration of each map task will then be $\lambda\cdot$ *decompressed size* $= \lambda \cdot \frac{f}{cf/a} = \frac{\lambda a}{c}$. Using $t = \frac{cf}{a}$, the total map phase time is represented in Equation 1.

$$\lceil \frac{cf}{ra} \rceil \cdot \frac{\lambda a}{c} + S(cf/a) \tag{1}$$

Equation 1 captures the following relationships:

1) As compression ratio decreases, it leads to a longer processing time for each task and a decrease in the amount of parallel tasks. But also there are benefits, the number of iterations will decrease and the scheduling and task creation delay will decrease.
2) The impact of compression ratio can be offset by adaptively altering the amount of data given to each map task, a. For example, a task receiving data of low compression ratio should receive less data so that the uncompressed size is not too large for a single task.

Table 1 shows how tasks assigned different types of compressed data can process a much larger amount than they read due to decompression. The data was randomly selected from multiple sources including the 1000 Genome Project [19]. The fourth and fifth columns show the decompressed data size of 128 MB and 256 MB compressed blocks respectively. Each entry in the fourth and fifth column represents a range of one standard deviation from the mean compression ratio, which in a normal distribution would be the middle 68% of the data. A single task assigned a 128 MB block could actually call the user's map function on up to 1256 MB of data and up to 2495 MB of data for 256 MB blocks.

Figure 1 shows job runtime with different compression ratios and block sizes. The blue region in the figure (execution time less than 500 seconds) decreases in size as compression ratio decreases, and completely vanishes for the best compression ratio (.05). Thus, the best compression ratio providing the smallest storage consumption is not optimal for compressed data processing.

Another trend is the effect of block size on compressed data. Based on the sampled data in Table 1, a 0.24 compression ratio is a common value in real world text files. But referring to Figure 1, when using the default block size of 128 MB with a 0.24 compression ratio, the execution time is 4x longer than the shortest runtime. But with a smaller block size of 32 MB, the execution time is the shortest out of all runs. Thus, Hadoop does not capture the potential performance improvement provided by compressed data. Performance seems to be better for uncompressed data (compression ratio of 1) than most of the compressed data runs, but this is because the reduction in disk and network I/O is being masked by the reduction in parallel tasks. In order to design and implement an improved compressed data processing scheme, we first quantify the current causes of the degraded performance.

B. *Parallelism*

Hadoop MapReduce deploys a key technique which has allowed it to become widely used as a parallel programming framework. It allows an application to be automatically parallelized by Hadoop with minimal work from the user. When processing compressed data, the parallelization is not adequate because Hadoop allocates too few resources.

The number of tasks for a job directly depends on the amount of input data. A large job will have many tasks executing concurrently while a small job will have a few tasks. Most warehouse clusters, such as those at Facebook and Yahoo, have more than 80% of their data set sizes on the order of gigabytes and around 50% of the input data is on the order of megabytes [20]. When compressing datasets of these sizes, the amount of data on-disk will decrease. Thus when processing this data, the number of tasks created, which depends on the on-disk length, will decrease. However, the amount of data which will be processed is the same before and after compression. Although there will be less tasks, each task will process more data than it reads since the data must be decompressed before processing. This results in less tasks which each process more data resulting in a reduction in parallelism.

Figure 2 shows the individual map task time for each task in a Wordcount job which processes 3 GB of uncompressed data. For the uncompressed data run, there are 39 tasks (shown in black) each reading and processing 128 MB worth of data. After compressing this data with Bzip2 and achieving 5x reduction in storage consumption, we ran the application again which resulted in 9 tasks (shown in red). This is a 4x reduction in parallelism. The data amount given to each task remained the same as the uncompressed data run. Each task processed more than 500 MB of data and total map phase time increased by a factor of 2.4. The parallelism issue occurs because task creation is based on on-disk data size not uncompressed data size.

C. *Map Task Skew*

A well known issue when processing data in MapReduce is the variation in task execution time (map task skew). Often, such variation prolongs job runtime and leads to straggling tasks [1]. This problem arises based on faulty hardware, low performing hardware, data skew, and crossrack traffic [18].

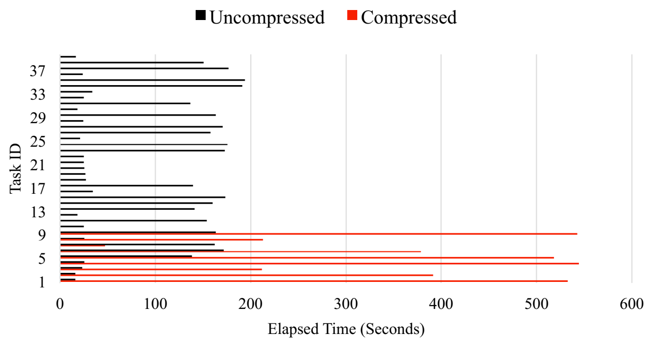

Fig. 2: The execution time for each task.

The issue also arises when processing on heterogeneous hardware [21]. We found that map task skew also arises when processing compressed data but at an even greater magnitude. For example, if a task is running on a node with low performing hardware and is processing compressed data, the runtime will be further prolonged since processing compressed data causes a task to process a larger data amount than normal.

In addition to these known causes of map task runtime variation, there are two new causes of variation which only arise when processing compressed data. First, if there are multiple files with different compression ratios as shown in Table 1, a task's runtime will depend on which file's data it receives since the decompressed data size will vary as in columns four and five. Second, slight variations in the amount of compressed data given to each map task can result in large variations in map task time.

Referring back to Figure 2, there is over a 300 second difference (longest task minus shortest task) in runtime for the compressed data run while only about 150 second difference for the uncompressed run. In this case, the variation is caused by slightly different amounts of data given to each map task. For example, suppose a task receives 20 MB less data than all the other tasks, then it will process 100 MB less data since there is a compression ratio is 0.2 for the data in Figure 2. This slight variation in task input data reduces load balance and increases the application runtime since the reduce phase can only start until after the last map task finishes.

One solution to solve map task skew distributes the data that is left to process of a straggling task to other tasks [22]. This solution cannot work when processing compressed data since when processing compressed data, a task must decompress the data in the same granularity in which it was compressed. For example, if each HDFS block was compressed individually, then the decompression must occur on each block and cannot be split arbitrarily within a block. Because of this, other tasks cannot simply start processing a straggling task's data without decompressing the entire block. This can waste both CPU and network resources, and thus it would be more advantageous if a method can detect compression ratio and task input amount variation before tasks are scheduled.

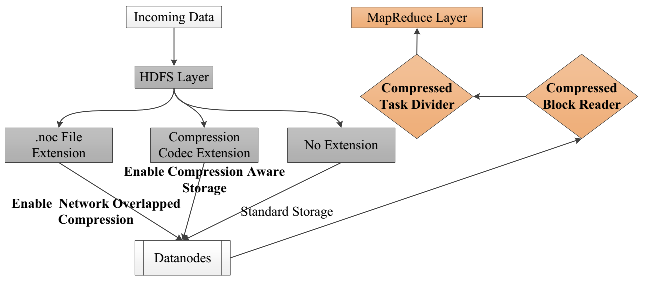

Fig. 3: Design of IC-Data contains two modules in gray and brown. Bold text represents the new techniques provided by IC-Data.

D. Compression Overhead and Data Load Time

The time to load data into a cluster is worth decreasing because it determines how quick users are able to run their applications. Data is often transferred between clusters within the same datacenter as well as between datacenters. For example, one of Facebook's clusters receives 30 TB of data everyday and thus is bottlenecked by network. In addition, the cluster compresses the data after it has been stored on disk incurring compression overhead [23]. Long load times can be mitigated by compressing the data, but currently compression and data loading are considered separately and done serially. By compressing data either before or after the data is loaded, the compression overhead is not hidden and is costly [11].

IV. IC-Data Design and Implementation

We present the design of a compression processing scheme, IC-Data, which consists of the HDFS Module and MapReduce Module, shown in Figure 3. The HDFS module provides two new methods to store data into HDFS and the MapReduce module provides two components to improve compressed data processing. Overall, the HDFS Module seeks to improve how compressed data is stored, so when the data is processed the MapReduce Module can use the modified storage layout to help decrease job runtime. Besides the performance goals, we abstract the complexity of IC-Data from the user.

A. HDFS Module

The HDFS module provides two new methods of data storage, Network Overlapped Compression (NOC) and Compression Aware Storage (CAS). Previously both compressed and uncompressed data were stored the same way. NOC compresses data while being loaded into HDFS. CAS improves storage of data that has already been compressed at a remote destination by dynamically changing the HDFS block size. The methods are differentiated based on file extension.

NOC and CAS have two design components which will alleviate the issues described in Section III. First, NOC and CAS increase parallelism by not reducing the number of HDFS blocks when the data is compressed. Second, they allow for constant decompressed data size, meaning that all blocks, possibly belonging to different files, when decompressed will have an equal size. This allows each task to process a constant amount of data, thus eliminating the issues of individual tasks processing large amounts of data and map task skew caused by compressed data..

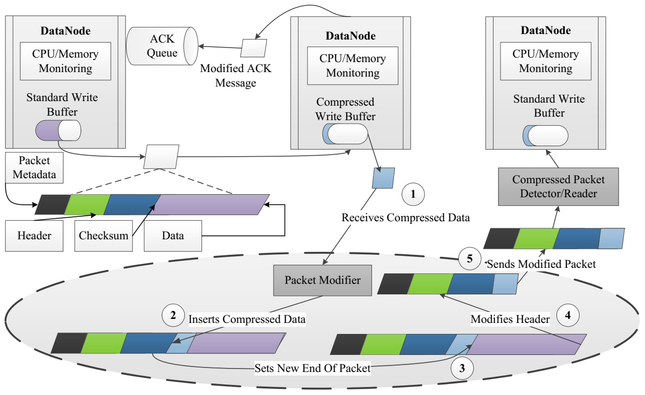

Fig. 4: The Network Overlapped Compression (NOC) design.

1) Network Overlapped Compression (NOC): Network Overlapped Compression (NOC) compresses data during data loading by interleaving compression with network transfer. It uses the overhead of replicating blocks to multiple datanodes to hide the compression latency. By overlapping compression with block transfer, the network traffic caused by loading large amounts of data and disk I/O is reduced.

When a file is loaded into HDFS, the file is divided into multiple blocks which are transmitted using 64 KB packets in a pipeline fashion to multiple datanodes for the purpose of fault tolerance. Typically, the data is not modified in the pipeline and is simply copied from node to node. NOC compresses the data, while in the pipeline.

For example, Figure 4 explains NOC. Here, the second datanode in the write pipeline decides to compress the block because it has available resources. Thus, for every packet of the block the second datanode receives, it compresses the packet. The Packet Modifier is responsible for modifying the packet with the compressed data as shown in Figure 4. The second datanode then writes the compressed data to its disk. The third node detects the compressed data and reads a shorter amount than the original packet length. The third node can simply write the smaller amount of data to disk resulting in a reduction in disk I/O without the compression overhead. The second datanode, also sends a modified ack message to the first datanode to notify that the block has compressed replicas.

NOC hides compression overhead and reduces data load time in two ways. First, compression occurs in parallel with other packet transfers and writes to disk. Second, after compression all nodes further in the pipeline will have reduced network transfer and disk I/O time, or in other words the benefits of compression trickle down to nodes deeper in the pipeline. NOC provides two features to improve processing of compressed data.

First, it allows the decompressed data size for each map task to be equal to the original HDFS block size since each block is compressed individually. For example, a datanode decides, based on resource utilization, to compress data on the first

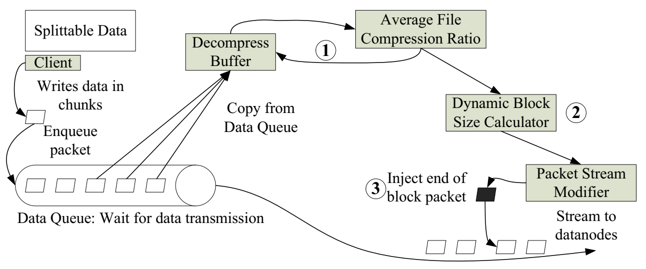

Fig. 5: The Compression Aware Storage (CAS) consists of a compression ratio calculation phase (1), block size calculation phase (2), and a stream modifying phase (3).

packet it receives for a block. If it decides to compress, all packets for the block will be compressed. Thus when the block is decompressed, the size must equal the original block size. However, the on-disk (compressed) length of the block will be smaller than the original block size.

Second, NOC maintains parallelism. Unlike standard compression where the number of blocks decreases when the data is compressed, NOC will have the same amount of blocks as the uncompressed data but each block will be smaller. The parallelism issue is improved since compression does not decrease the amount of blocks and thus there is more opportunity to divide work between tasks.

Although NOC compresses data packet by packet, the overall storage reduction remains the same as normal compression. This is because after 128 KB buffer sizes compression ratio does not improve significantly [24]. We fill this buffer size by increasing the default packet size from 64 KB to 128 KB.

2) Compression Aware Storage (CAS): Currently, HDFS does not differ in how it stores compressed data. Due to the strong coupling between HDFS and Hadoop MapReduce, how the data is stored greatly affects the job execution time when the data is processed. Thus, in order to reduce the job execution time when processing compressed data, we modify how compressed data is stored in HDFS.

The drawback of NOC is that it can only be applied to data which is not compressed. Often times data being loaded is already compressed. Receiving compressed data is common since transferring compressed data decreases data load time. Compression Aware Storage (CAS) serves the purpose of efficiently storing data which already arrives compressed in HDFS. Because of the difference in processing requirements for the two types of codecs, CAS stores data compressed with splittable and non-splittable codecs differently.

As shown in Figure 5 for splittable codecs, CAS decreases the block size dynamically based on a sampled compression ratio. CAS modifies the stream of data packets before the packets reach the datanodes. In Step 1, CAS copies the compressed data portion of several packets in the data queue into the decompression buffer. Then, CAS decompresses the data in order to calculate the compression ratio. Once calculated, the compression ratio contributes to the average file compression ratio. To minimize decompression overhead, CAS uses the packet's wait time in the data queue to overlap with

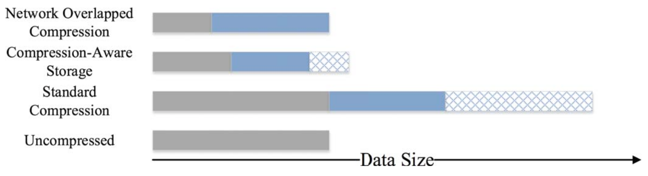

Fig. 6: The compressed data length (in gray) and uncompressed data length (gray plus blue) and decompressed size variation (striped) is shown for different schemes.

decompression. CAS could continue copying more packets in order to more accurately calculate the file's compression ratio but incur a larger overhead, or the process moves to Step 2.

Step 2 consists of a simple metric to calculate the new block size based on a given compression ratio. In order to keep a constant decompressed data size, the block size must decrease by an amount dependent on the compression ratio. Specifically, the new block size is calculated as *old block size* $*$ *compression ratio*. If the sampled compression ratio is accurate, when this block is decompressed for processing the uncompressed size will be approximately equal to *old block size*. In order to guard against the new block size being too small and resulting in short reads, the Dynamic Block Size Calculator contains a *threshold* which when crossed the *old block size* used in the calculation will increase by a power of two. After the new block size is calculated, it is sent to the Packet Stream Modifier.

The Packet Stream Modifier injects an end of block packet when the block boundary has been reached. Instead of using the *old block size* for injecting an end of block packet, the Packet Stream Modifier contains a counter of how much data has been sent to the datanodes. Once the counter has reached the new block size, the Packet Stream Modifier injects an end of block packet earlier than normal in order to signal to downstream datanodes the end of the block has been reached.

For non-splittable codecs, CAS stores the entire compressed file into one block. Since this compressed file will be processed by one task, the task will be able to process the entire file locally. If the file was broken into multiple blocks, then the single task will have to connect to multiple nodes in order to remotely read the data causing network overhead. Fault tolerance is maintained since CAS only reduces the number of blocks to one but not the number of replicas.

CAS increases parallelism by reducing the block size which increases the number of blocks for a file and thus provides more opportunity to divide work between tasks. It also provides approximate decompressed size by using compression ratio to determine block size.

Figure 6 describes how the different compression/storage mechanisms compare in terms of decompressed size of a single block. The gray blocks represent the on-disk (compressed) length. The blue plus gray block represents the uncompressed size. The striped block represents the variation in decompressed size which can be caused by different compression ratios between files. NOC allows the decompressed size to be constant, while CAS causes some variation due to inaccurate

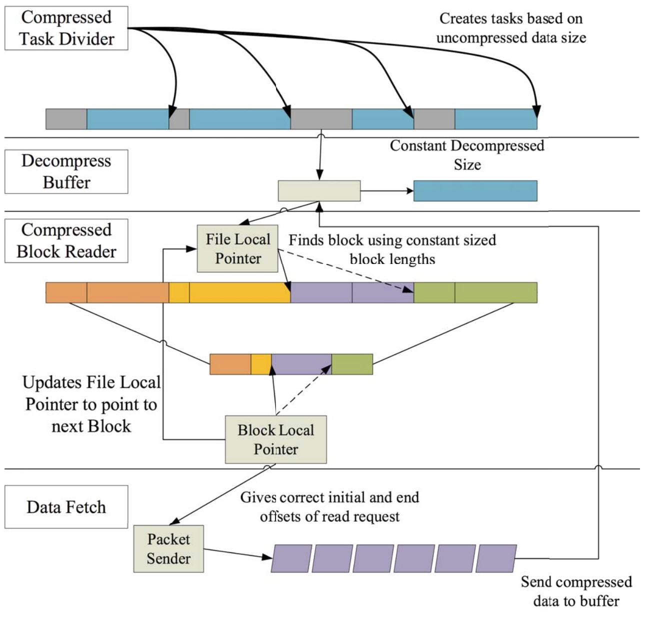
Compressed Task Divider
Creates tasks based on uncompressed data size
Decompress Buffer
Constant Decompressed Size
Compressed Block Reader
File Local Pointer
Finds block using constant sized block lengths
Updates File Local Pointer to point to next Block
Block Local Pointer
Data Fetch
Gives correct initial and end offsets of read request
Packet Sender
Send compressed data to buffer

Fig. 7: The MapReduce Module design.

compression ratio sampling. Standard compression, by storing the compressed data in one full block, has a large decompressed size as well as a larger variation. NOC and CAS can cause a single file to have blocks of different on-disk sizes. The MapReduce Module abstracts the complexity created by varied block lengths and improves resource allocation when processing compressed data.

B. MapReduce Module

The MapReduce Module contains two components, the Compressed Task Divider and Compressed Block Reader. The former modifies the number of tasks (resources) by using the uncompressed data size instead of traditional compressed data size to determine the number of tasks for a job. It also detects the type of compression mechanism, NOC, CAS, or standard, in order to improve parallelism. The latter supports the Hadoop I/O API for reading blocks of varied lengths.

1) Compressed Task Divider: Compared to traditional Hadoop which packs as much on-disk data that fits into a split, the Compressed Task Divider packs as much uncompressed data into a split. As shown as the first layer in Figure 7, the gray block is the compressed data size and the blue block plus the gray block is the uncompressed data size. A split is the amount of data assigned to a task. Standard Hadoop would fit as much of the gray blocks into a single split, which would lead to two splits (two tasks) of large, varying decompressed size. Instead, we pack as much uncompressed data (gray plus blue) into 4 splits (4 tasks) in Figure 7. This results in a constant and user-tunable (changing split size) data processing amount for each task. In this way, the Compressed Task Divider allocates sufficient resources.

In order for the Compressed Task Divider to use the uncompressed data size, the data must be stored with NOC or CAS. The Compressed Task Divider simply assigns tasks to a single gray block which would typically be much less than the default 128 MB split size. During execution, each task will read an amount equal to the gray block size and process an amount equal to gray plus blue. The value of NOC and CAS for processing data comes from their assurance of a constant (NOC) or approximate (CAS) decompressed data size, which would be the gray plus blue blocks. In addition by not reducing the number of blocks after compression (gray blocks), there will be the same number of parallel tasks as the uncompressed data.

The Compressed Task Divider also allows for increased parallelism by detecting data stored with NOC. Since NOC compresses each block individually, each block can be decompressed without needing data in another block. Thus, NOC allows all compressed data to be splittable for all codecs. Gzip, which is also used by Facebook [24], is a non-splittable codec but can now be split and processed in parallel. In order to split data that is from a non-splittable codec, the Compressed Task Divider ensures each split contains all the data from a single compressed block. This allows a job to have its data decompressed in parallel and thus increase overall parallelism.

2) Compressed Block Reader: The Compressed Block Reader supports the Hadoop I/O API for NOC and CAS because they cause varying compressed (on-disk) block lengths. HDFS is designed such that blocks for the same file should have the same size except for possibly the last block in the file. In addition, tasks are assigned to data based on the assumption each block is of constant size. In order to circumvent this assumption, the Compressed Block Reader provides the allusion that the blocks stored using NOC and CAS are constant sized.

As shown in Figure 7 in the Compressed Block Reader layer, in order to put compressed data into the decompression buffer, two file pointers are created. The File Local Pointer presents a logical uncompressed view of the file. In this view, each block (pictured in different colors) has a constant size and is uncompressed. If a seek for a particular offset in the file occurs, the Local File Pointer is redirected to the initial offset of the block which contains the desired file offset. Thus, the File Local Pointer supports seeks between different blocks of a file and hides the complexity of varied block lengths.

To read data, the Local File Pointer is set to the initial offset of the block which needs to be read. Then, a Local Block Pointer is created to fulfill the request. A Local Block Pointer is aware of the compressed data length of the block and is used to point to the compressed data bytes. The Local Block Pointer has two main responsibilities. One, it has to update the Local File Pointer once the whole block has been read. Second since the initial request came in terms of the uncompressed file view, it has to translate the end and start offsets from the initial request to match the compressed block length.

In the Data Fetch layer, the Local Block Pointer sends the corrected start and end offset to the Packet Sender. This will correctly reduce the amount of packets which will be sent and potentially also reduce network I/O if the request was a

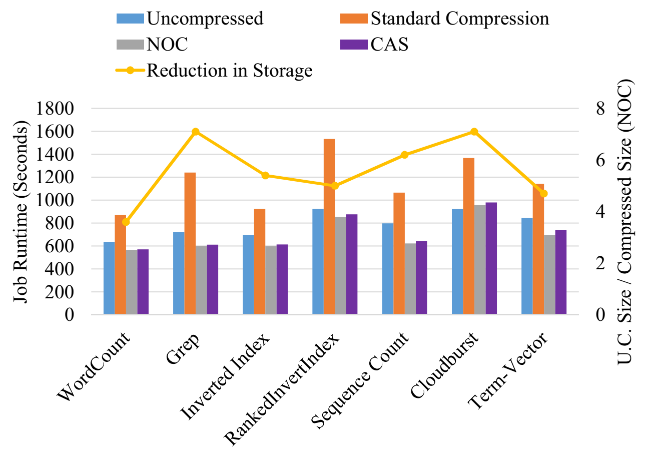

Fig. 8: The performance comparison between NOC, CAS (1 iteration), uncompressed, and standard compressed data processing is shown by the four bars which correspond to the left axis. Storage savings of NOC compared to uncompressed (U.C) for each application's dataset is shown by the yellow line which corresponds to the right vertical axis. The higher this value the more storage savings.

remote read request. Once in the decompression buffer, the data is decompressed and processed by the map task.

Overall, the MapReduce Module provides three features which improve how Hadoop processes compressed data. First, it allows tasks to be assigned based on uncompressed data instead of compressed data thus ensuring a sufficient number of parallel tasks for a single job. Second, it allows all compressed data to be split and processed in parallel. This also increases the number of codecs which are reasonable to use with Hadoop. Third, it abstracts the complexity of varying block lengths.

C. Implementation

IC-Data was implemented in the latest Hadoop release Hadoop-2.6.0. File type detection, packet modification, packet stream modification, varying packet sizes, compression buffers, decompression buffers, and compression ratio calculation were the major components which needed to be added for the HDFS module. This required modifications in *DFSOutputStream*, *PacketReceiver*, *BlockReader*, and *DFSInputStream* java classes. For the MapReduce Module, the Compressed Task Divider was implemented in *FileInputFormat* and Compressed Block Reader's two pointer types were implemented in *BlockReader*, *BlockSender*, and *DFSInputStream* classes.

V. Experimental Evaluation

We will examine the overall performance improvement using IC-Data and examine the efficiency of NOC and CAS as well as the reduction in data load time. Lastly, we will evaluate the improvements in parallelism and map task skew.

All experiments were conducted on a 42 node Sun Fire Linux Cluster. Each node has a 2.3GHz Opteron quad-core processor, 8GB memory, and a 250 GB 7200 RPM SATA hard drive. All nodes are connected through a gigabit ethernet connection. All tests were done with Hadoop YARN enabled.

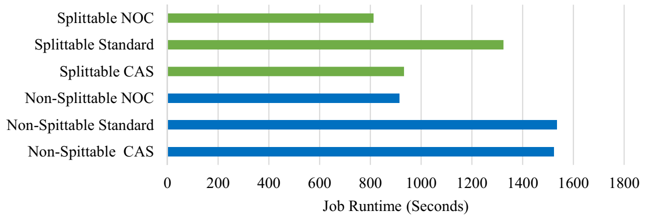

Fig. 9: The performance for different codec types running Terasort.

A. Overall Performance Improvement

In order to analyze the performance benefits of IC-Data, we compare its performance against processing uncompressed data and processing standard compressed data. Comparing against uncompressed data shows that users can achieve storage savings without sacrificing performance, unlike before.

We compare the performance against 7 different benchmarks. Wordcount and Grep are I/O and CPU intensive respectively. Inverted Index, Ranked Inverted Index, Sequence Count, and Term Vector come from the Puma Benchmarks and were chosen because they are similar to real data intensive applications which might run at data warehouses [25]. They also have slightly different input data formats, thus lending to different compression ratios and a thorough performance analysis. Lastly, Cloudburst, a Bioinformatics application, was tested to understand the performance on Fatsa file formats. The input data sizes were roughly 40 GB and exhibited 4x to 7x storage savings depending on dataset. For most runs, the compression codec used was Bzip2 since it is one of the few codecs which allows input data to be split into multiple tasks.

In Figure 8, NOC reduces on average job execution time by 66% compared to standard compressed data processing. When compared to uncompressed data, it reduces runtime by 14% and allows 5.6x less storage, as can be seen from the yellow line graph which corresponds to the right vertical axis of Figure 8. This allows users to achieve *both* performance and storage savings unlike standard compression. CAS achieves similar results. The performance advantage is more significant for CPU intensive applications since they benefit more from the improved parallelism provided by IC-Data. Although the parallelism for both IC-Data and uncompressed is the same, the benefits of less network and disk I/O outweigh the overhead of decompression. However for Cloudburst, IC-Data performs slightly worse, which we believe is due to the binary file format the input data is in. Using a compression codec with less CPU overhead than Bzip2 should provide improvements.

Figure 9 shows the improvement in processing data from the two codec types while running a sorting benchmark. Splittable NOC and CAS provide a 35% reduction in execution time compared to splittable standard. Non-splittable CAS does not provide much improvement even though it provided 100% local reads. This is because the remote reads for non-splittable standard do not provide a significant overhead compared to the decompression overhead. Even though 60% of the data is read remotely for non-splittable standard, performance doesn't

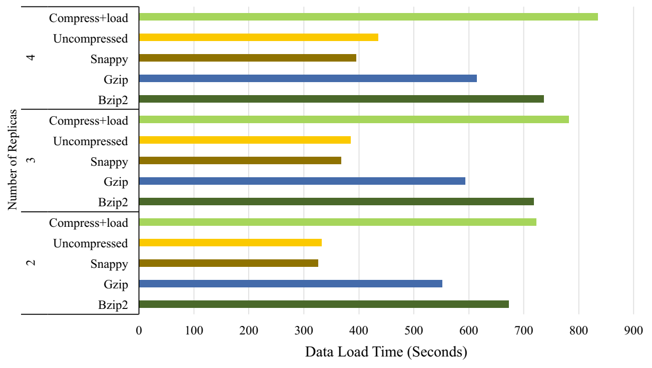

Fig. 10: Improvements in data load time for NOC data.

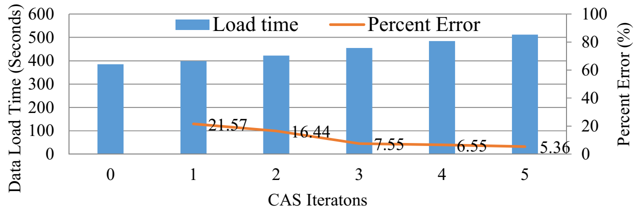

Fig. 11: The efficiency and accuracy of CAS for different iterations.

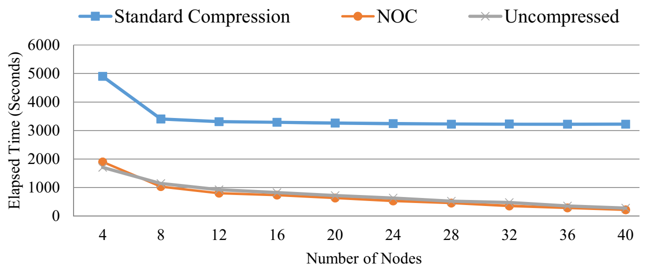

Fig. 12: The scalability of the compression techniques.

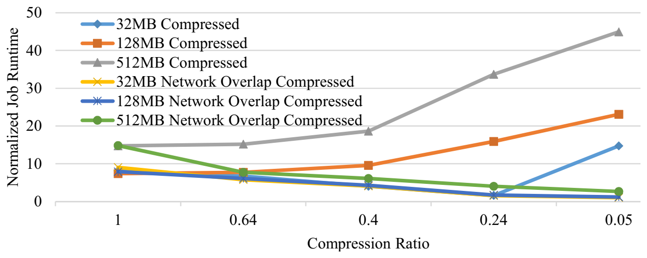

Fig. 13: Normalized job runtime for different ratios and block sizes.

suffer. Thus, when processing compressed data the priority for achieving local reads should be less than the priority of running jobs on nodes which can provide high decompression speeds.

B. Improvements in Data Load Time

We tested NOC using three compression codecs to measure data load time in Figure 10. Bzip2 has the most overhead, followed by Gzip, and with the least overhead but also least compression ratio is Snappy. Besides using NOC with three codecs, we also measured load time for the original uncompressed 40 GB file and the time to compress the data with Gzip and then load the data serially (compress + load), which is the current data loading technique.

Overlapping compression with data transfer using NOC provided on average a 31% improvement from the conventional compress+load. Snappy was able to beat the time to load the uncompressed data. Since Snappy has faster compression times, it was able to hide all the latency associated with data compression using the reduced disk and network I/O and reduced storage size to 48% of the original 40 GB file.

1) Compression Aware Storage Efficiency: CAS can be measured using two metrics, accuracy and efficiency. Accuracy is the percent error between the sample compression ratio and the file's actual compression ratio. The sampled ratio can be made accurate by decompressing more packets, but the overhead increases with the number of decompressed packets.

Figure 11 shows the data load time for different numbers of sampling iterations (the number of packets decompressed). The test was conducted on a 40 GB taken from the log data also shown in Table 1. The data loaded was compressed with Bzip2. From the figure, an iteration of 0 represents standard data loading without CAS measurements. With a 3% increase in runtime, CAS can get within a 22% error using one iteration. It can get within a 8% error while incurring a 17% increase in load time. A lower percent error results in less map task skew and improved load balance.

C. Improved Parallelism

In Figure 12, we tested a CloudBurst job processing 40 GB of data with a 0.2 compression ratio. Standard compression processing does not decrease in runtime after 8 nodes, while both NOC and uncompressed data processing continue to decrease in runtime. Hadoop YARN allows an upper bound on how many resources can be allocated on a single node simultaneously. Thus, when the number of required resources is greater than the number of available resources the tasks must wait and thus more nodes provide improvement. After 8 nodes, the amount of available resources was already larger than the amount of needed resources for the standard compressed data and thus more nodes didn't provide improvement.

When the number of nodes was only 4, NOC and uncompressed still performed better than standard compression even though there were a large number of tasks which had to wait to be scheduled. This occurs for two reasons. First, the amount of intermediate data per task is larger for standard compression. Since each task processes a larger amount of data, it must write a larger amount of intermediate data. This large amount of intermediate data requires a single task's internal buffer to fill multiple times resulting in multiple spills to disk. NOC and uncompressed will have less spills per task.

Second, map phase is overlapped with data shuffling to the reduce tasks. Although the reduce phase cannot start until the last map task finishes, the completed map tasks can have their intermediate data sent to the reduce tasks. Since each task takes longer to process, the amount of shuffling that can be overlapped with the map phase is less for standard compressed.

Figure 13 shows how different compression ratios affect performance. Lower compression ratios, although better for storage consumption, limit the amount of parallelism. As seen in Figure 13, as compression ratio decreases the performance actually worsens for standard compressed data, but improves for NOC due to reduction in network and disk I/O.

TABLE II: The standard deviation (Seconds) in map task runtime for different storage schemes and causes of skew.

Cause of Skew	U.C. Data	NOC	CAS	Compressed
Input Data Var.	69.21	79.24	123.45	181.29
Compression Ratio Var.	X	85.93	142.56	224.67
Node Hardware	113.54	200.21	286.46	375.45

D. Reduction in Map Task Skew

We measure the variations in map task duration caused by different compression ratios, node hardware, and amounts of input data in Table II.

The variation in map task duration using standard compressed data storage is consistently the largest followed by CAS. Recall Figure 6, CAS does not allow constant decompressed data size only an approximate estimation. The accuracy depends on the number packets sampled for the compression ratio. In this example, we took only one sample (i.e. the first packet). Surprisingly, NOC has more variation than uncompressed data (U.C. Data) even though it allows constant decompression size. We believe this is caused by the decompression latency which can become a source of significant variation when node hardware performance is altered.

VI. Conclusion and Future Work

Due to the need for storing compressed data to limit storage consumption, data is stored compressed. Yet, processing compressed data takes on average 49% longer. In this paper, we first analyzed the current problems with compressed data processing, which were decrease in parallelism, increase in map task skew, and serial execution of compression and data loading. After formulating, analyzing, and quantifying these issues, we developed the HDFS Module and MapReduce Module. The HDFS Module improved compressed data storage by allowing constant/approximate decompressed data size. It decreased data load time by 31% by overlapping compression with data loading. By allocating tasks based on uncompressed data size, the MapReduce Module decreased job runtime by 66% compared to standard compressed data processing and 14% compared to uncompressed data processing, allowing users to achieve *both* performance and storage savings.

As future work, we will extend these solutions into in-memory big data processing engines. In these frameworks, due to limited memory capacity, achieving both performance and storage savings will be even more essential.

VII. Acknowledgment

The authors would like to thank the partial support from NASA AIST 2015 funding. This research is also supported in part by NSF under NSF grants CNS-0751200, CNS-1162540, and CNS-1526887.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplied Data Processing on Large Clusters," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[2] Apache Software Fundation. (2014, Mar.) Apache Hadoop Project. [Online]. Available: http://hadoop.apache.org

[3] C. Zhang, H. Sterck, A. Aboulnaga, H. Djambazian, and R. Sladek, "Case study of scientific data processing on a cloud using hadoop," in *High Performance Computing Systems and Applications*, 2010.

[4] M. Gaggero, S. Leo, S. Manca, F. Santoni, O. Schiaratura, and G. Zanetti, "Parallelizing bioinformatics applications with mapreduce," in *CCA-08: Cloud Computing and its Applications*, 2008.

[5] J. Ekanayake, S. Pallickara, and G. Fox, "Mapreduce for data intensive scientific analyses," in *IEEE Fourth International Conference on eScience*, 2008.

[6] M. Schatz, "Cloudburst: highly sensitive read mapping with mapreduce," in *Bioinformatics*, 2009.

[7] http://www.enterprisetech.com/2014/04/11/facebook-compresses-300-pb-data-warehouse/, 2014.

[8] "Cisco global cloud index: Forecast and methodology, 2013-2018." http://cisco.com/c/en/us/solutions/collateral/serviceprovider/global-cloud-indexgci/Cloud Index White Paper.html, [Last accessed November 2014].

[9] http://blog.cloudera.com/blog/2009/11/hadoop-at-twitter-part-1-splittable-lzo-compression/, 2009.

[10] "Vertica," http://www.vertica.com/.

[11] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *SIGMOD'09*, 2009.

[12] G. Graefe and L. Shapiro, "Data compression and database performance," in *Proceeding of the 1991 Symposium on Applied Computing*, 1991.

[13] B. Welton, D. Kimpe, J. Cope, C. Patrick, K. Iskra, and R. Ross, "Improving i/o forwarding throughput with data compression," in *International Conference on Cluster Computing (CLUSTER)*, 2011.

[14] E. R. Schendel, S. Pendse, J. Jenkins, D. A. Boyuka II, Z. Gong, S. Lakshminarasimhan, Q. Liu, H. Kolla, J. Chen, S. Klasky, R. Ross, and N. F. Samatova, "Isobar hybrid compression-i/o interleaving for large-scale parallel i/o optimization," in *Proc. International Symposium on High-Performance Parallel and Distributed Computing, HPDC 12*, 2012.

[15] Y. Chen, A. Ganapathi, and R. Katz, "To compress or not to compress - compute vs. io tradeoffs for mapreduce energy efficiency," in *Proc. of the first ACM SIGCOMM workshop on Green networking*, 2010.

[16] https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC, 2015.

[17] "Hive," https://hive.apache.org/, 2010.

[18] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *OSDI*, 2010.

[19] G. McVean, "An integrated map of genetic variation from 1,092 human genomes," *Nature Publishing Group*, 2012.

[20] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," in *Proc. of VLDB*, 2012.

[21] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *OSDI*, 2008.

[22] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating skew in mapreduce applications," in *SIGMOD'12*, 2012.

[23] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. S. Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *SIGMOD'10*, 2010.

[24] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu, "Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems," in *ICDE*, 2011.

[25] F. Ahmad, S. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Tarazu: Optimizing MapReduce on Heterogeneous Clusters," in *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.