Adding Data Analytics Capabilities to Scaled-out Object Store

Cengiz Karakoyunlu[a], John A. Chandy[a], Alma Riska[b]

[a]*Department of Electrical and Computer Engineering, University of Connecticut, Storrs, CT, 06269*
[b]*NetApp, Inc.*

Abstract

This work focuses on enabling effective data analytics on scaled-out object storage systems. Typically, applications perform MapReduce computations by first copying large amounts of data to a separate compute cluster (i.e. a Hadoop cluster). However; this approach is not very efficient considering that storage systems can host hundreds of petabytes of data. Network bandwidth can be easily saturated and the overall energy consumption would increase during large-scale data transfer. Instead of moving data between remote clusters; we propose the implementation of a data analytics layer on an object-based storage cluster to perform in-place MapReduce computation on existing data. The analytics layer is tied to the underlying object store, utilizing its data redundancy and distribution policies across the cluster. We implemented this approach with Ceph object storage system and Hadoop, and conducted evaluations with various benchmarks. Performance evaluations show that initial data copy performance is improved by up to 96% and the MapReduce performance is improved by up to 20% compared to the stock Hadoop implementation.

Keywords: In-situ Data Analytics, Object Storage, Attribute-based Storage, MapReduce

Email addresses: cengiz.k@uconn.edu (Cengiz Karakoyunlu), john.chandy@uconn.edu (John A. Chandy)

1. Introduction

High-performance computing on large-scale data has become an important use case in recent years. There are various storage system solutions for end users to perform high-performance computation on large-scale data, while also providing data protection and concurrency between different users [1].

Clusters and cloud storage applications that work on large-scale data typically employ separate compute and storage clusters, since the requirements of the compute and storage tiers are different from each other. However, a serious drawback of this architecture is the need to move large amounts of data from the storage nodes to the compute nodes in order to perform computation and then to move the results back to the storage cluster. Today, many storage systems store petabytes of data for various applications, such as climate modeling, astronomy, genomics analysis etc., and the amount of data stored in these systems is projected to reach exabyte scale in the near future [2]. Therefore, moving big amounts of data between storage and compute nodes is not an efficient way of performing computation on large-scale data anymore. Additionally, storing data both at the storage and compute sites increases storage overhead and with data replicated multiple times at both sites for resiliency, this overhead becomes even worse. Moving data between storage and compute nodes also increases the total energy consumption and the network load.

On the other hand, there have been many efforts that have gone into improving storage interfaces and abstractions in order to store and access data more efficiently. Object-based storage [3, 4] is an important effort in this respect and many scaled-out storage systems today [5, 6, 7] are based on the object-based storage abstraction. Object-based storage is an alternative to the traditional block-based storage (i.e. SCSI, ATA). Data is stored in discrete containers, called *objects*, each of which is identified by a distinct numerical identifier. Each object stores data and data attributes that can be controlled by the user. Data attributes can be used to store metadata describing the data (i.e. size, name, replica locations etc.) and metadata management operations to query these attributes can be offloaded from dedicated servers to object storage for improved performance [8]. As a result, object-based storage increases the interaction between the storage system and the end-user and simplifies the data management of a storage system.

Using object-based storage features, the computational applications in a cluster or cloud application can benefit from the intelligence of the underlying

storage system and eliminate data movement while enabling in-place analytics capabilities. Consequently, the storage layer can be scaled while the computational layer remains lightweight. In this paper, we propose an example of this approach by implementing a computational framework, Hadoop [9], on Ceph object-based storage system [10]. We also conduct performance evaluations using *Grep* [11], *Wordcount* [12], *TestDFSIO* [13] and *TeraSort* [14] benchmarks with various redundancy and replication policies. The evaluation results indicate that initial data copy performance of Hadoop is improved by up to 96% and MapReduce performance is improved by up to 20%. It is important to note that, Hadoop and Ceph object storage system can still be used as stand-alone systems in this approach, meaning that their normal functionalities are not impacted.

The rest of this paper is organized as follows. Section 2 briefly introduces MapReduce and object-based storage, two main components of this work. Then, Section 3 discusses related studies in a number of categories: improving the performance of Hadoop as a stand-alone system, using a cluster file system as the backend storage of Hadoop and integrating the computation layer of Hadoop, MapReduce, with object storage systems for in-place computation. While presenting studies for the last category, their disadvantages against the method presented in this paper are discussed; namely, data is still transferred to HDFS, data management policies of the underlying storage system are overridden or data-compute locality is only provided through virtualization. Section 4 shows how to enable in-place analytics capabilities on large-scale data using Hadoop and Ceph object storage without transferring data from compute nodes to storage nodes and without changing how the underlying storage is managed. Section 5 gives the performance evaluation results of the proposed method from *Grep* [11], *Wordcount* [12], *TestDFSIO* [13] and *TeraSort* [14] benchmarks. Finally, Section 6 summarizes the findings of this work and discusses possible future research directions.

2. Background

This section gives a brief overview of the main components of the approach proposed in this work - MapReduce and object-based storage.

2.1. MapReduce

MapReduce is a parallel computational model developed originally by Google [15] and it is widely used for distributed processing of large datasets

over clusters. Data in MapReduce is represented with <key, value> pairs. The first step of an application using MapReduce is to partition its input data into blocks that are replicated across datanodes. This data is then processed in parallel with mappers that produce intermediate data from the input data. This intermediate data is then fed to reducers which process the intermediate data based on intermediate keys and combine intermediate values to form the final output data of the application.

Hadoop [9] is a commonly used open-source implementation of MapReduce and it consists of two layers - storage and computation. The MapReduce algorithm is implemented in the computational layer, whereas the storage layer is managed by the Hadoop Distributed File System (HDFS). HDFS provides redundancy by replicating data three times (by default) across the storage nodes while also trying to preserve the data locality of the system. One replica is stored locally, the second replica is located in another node in the same rack and the last replica is stored in another rack. Hadoop applications also follow a write-once-read-many workflow and as a result, they can benefit from the approach presented in this paper extensively, as data is not ingested from a remote storage cluster to the compute cluster.

2.2. Object-Based Storage

Object-based storage is a storage model that stores and accesses data in flexible-sized logical containers, called *objects*, instead of using the traditional fixed-sized, block-based containers. Objects store metadata either together with data or in dedicated object attributes. Metadata can be any type of data (i.e. size, access permissions, creation time etc.) describing the actual object data. Increasing interest in object-based storage led to the standardization of the T10 object-based storage interface [16]. There have been many examples of object-based storage systems in cluster file systems; such as PVFS [17] and Lustre [5], as well as scaled out cloud storage systems; such as Ceph [10], OpenStack Swift [7], and Amazon S3 [18]. These systems are typically designed as a software interface on top of an existing file system.

3. Related Work

This section introduces related studies on improving the performance of Hadoop and its integration with object storage.

There have been several research efforts that analyzed and tried to improve the performance of Hadoop without integrating it with an underlying

storage system. Shvachko et al. show the metadata scalability problem in Hadoop, by pointing out that a single namenode in HDFS is sufficient for read-intensive Hadoop workloads, while it will be saturated for write-intensive workloads [19]. Some related studies improved the performance of Hadoop by modifying its internal data management methods. *Scarlett* replicates data based on popularity, rather then creating replicas uniformly and causing machines containing popular data to become bottlenecks in MapReduce applications [20]. Porter analyzes the effects of decoupling storage and computation in Hadoop by using *SuperDataNodes*, servers that contain more disks than traditional Hadoop nodes, for the cases where the ratio of the computation to storage is not known in advance [21]. *CoHadoop* modifies Hadoop by co-locating and copartitioning related data on the same set of nodes with the hints gathered from the applications [22]. *Maestro* identifies map task executions processing remote data as an important bottleneck in MapReduce applications and tries to overcome this problem with a scheduling algorithm for map tasks that improves locality [23].

Hadoop is also integrated with cluster file systems in a number of studies, in order to analyze the outcomes of using cluster file systems for MapReduce applications. Tantisiriroj et al. integrate *PVFS* [17] with Hadoop and compare its performance to HDFS [24]. Ananthanarayanan et al. use *metablocks*, logical structures that support both large and small block interfaces, with *GPFS* to show that cluster file systems with metablocks can match the performance of Internet file systems for MapReduce applications [25]. *Lustre* can also be used as the backend file system of Hadoop [26].

More recent work integrates object storage with MapReduce for in-place data analytics. Rupprecht et al. integrates OpenStack Swift with MapReduce [27]; however, this work overrides the replication policy of OpenStack Swift and has performance loss due to the time reducers spend while renaming results. CAST [28] performs cloud storage allocation and data placement for data analytics workloads by leveraging the heterogeneity in cloud storage resources and within jobs in an analytics workload. SupMR [29] creates MapReduce input splits from data chunks rather than entire data, meaning that data is still copied to the HDFS. Nakshatra [30] uses pre-fetching and scheduling techniques to improve the performance of data analytics jobs that are executed directly on archived data; but, data is still read and ingested into HDFS. Similarly, VNCache [31] and MixApart [32] use pre-fetching and scheduling techniques to ingest data to a cache on compute cluster. However, data is still transferred from the storage cluster to the compute cluster and

mechanisms to maintain and clean the caches on compute nodes are needed. Rutman presents a method similar to the method we are proposing to integrate Hadoop with Lustre [33]; but, hard links are used for the intermediate output data of mappers and a fast network interconnect between the storage and compute tiers is assumed to be readily available. The method we are proposing is not dependent on the type of network interconnects and symbolic links are used while ingesting data to HDFS. Yu et al. [34] has a similar implementation, where in-situ data analytics on Lustre storage nodes is enabled. Unlike our approach, the map tasks in this method do not always work with local data and metadata server is queried for replica locations every time, which can be costly in terms of performance. Yu et al. [34] also co-locates data analytics with Lustre through virtualization; while they run on the same physical node in the method we are proposing. VAS [35] is another similar study; except that it does not always follow the replication policy of the underlying Lustre storage system and it co-locates data and computation using virtual machines. Wilson et al. present RainFS to integrate MapReduce with HPC storage [36]; however, network-attached remote storage is considered only.

4. Proposed Method Architecture

This section presents our approach to integrate Hadoop with an object-based storage system - Ceph is used as the demonstration platform, but any object-based storage system such as PVFS [17] or Lustre [5] could be used. As mentioned in Section 2.1, Hadoop consists of a computation layer, MapReduce, and a storage layer, HDFS, that manages the underlying storage system. This work modifies Hadoop to perform in-place computation on large-scale data without moving or transferring data anywhere and enables having a lightweight MapReduce computation layer, while scaling the underlying storage system.

A typical Hadoop implementation is shown in Figure 1. Storage cluster stores the data on which a MapReduce application will be executed. Compute cluster consists of nodes that form a master-slave relationship; where the master node is responsible for transmitting jobs or I/O operations and monitoring the status of slave nodes. Each Hadoop node has two process running corresponding to the computation (MapReduce) and storage (HDFS) layers respectively. On master nodes, the computation layer process is a jobtracker and the storage layer process is a namenode. On slave nodes, the

computation layer process is a tasktracker and the storage layer process is a datanode.

At the start of a MapReduce application, data is transferred from the storage cluster to the compute cluster through a network interconnect. In the storage layer, datanodes are responsible for replicating and storing this data as *blocks*. In the context of this discussion, a *block* is the smallest unit of data that can be accessed by Hadoop I/O operations. Each datanode periodically sends heartbeats and block reports to the namenode to keep its global view updated. Namenode is responsible for metadata operations; such as, keeping track of the block location on the datanodes, collecting status reports from the datanodes and choosing datanodes to perform I/O operations on. As an example; when a client wants to perform an I/O operation (read or write) in the system, it first communicates with the namenode to learn the data block locations for a read operation or to obtain a list of datanodes to store data blocks on for a write operation. As soon as the client has this information, it can communicate with the datanodes directly to perform the I/O operation. As mentioned earlier, datanodes replicate data blocks. During a read operation, the client chooses one of the replicas (usually the closest one in terms of network distance) to start reading from. On the other hand, the write operation is performed in a pipeline, i.e. data is written to the first replica location at first, then it is forwarded from the first replica location to the second replica location and so on. Client receives an acknowledgment at the end of successful I/O operation.

A namenode process is usually co-located with a jobtracker in the computation layer. Similarly, a datanode process is usually co-located with a tasktracker in the computation layer. Jobtracker is responsible for assigning and distributing tasks to the tasktrackers and keeping track of their status. Tasktrackers are responsible for performing actual map and reduce tasks with data stored in the storage layer, HDFS. Jobtracker receives an acknowledgment from all the tasktrackers when they are done with executing a given task.

4.1. Moving Hadoop Processes to Storage Cluster

The biggest disadvantage of the typical approach presented in Figure 1 is the need to transfer data between the storage and compute clusters before and after a MapReduce application. Considering the size of data being moved, the network interconnect can be easily maxed-out and the total energy consumption would increase as system resources are utilized for data

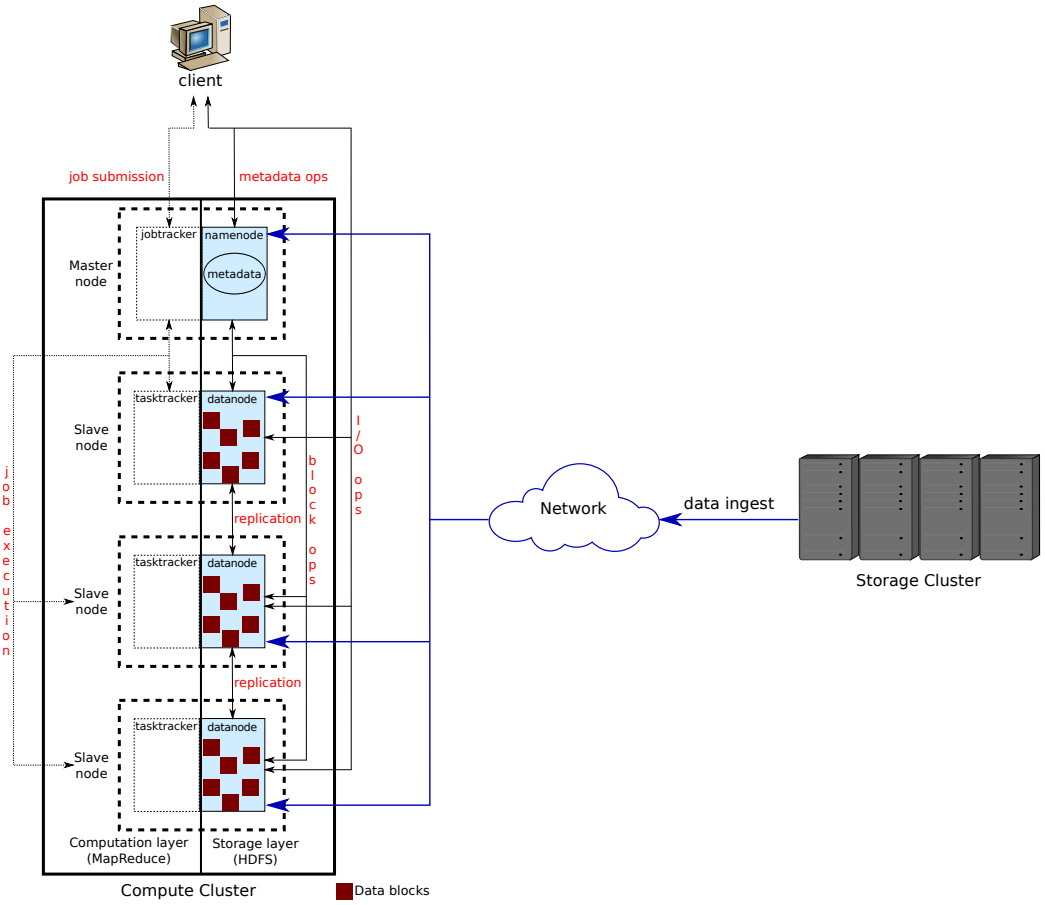

Figure 1: Typical Hadoop Implementation Architecture

transfer.

The proposed approach implements Hadoop on the compute cluster. By doing so, the network interconnect between separate clusters is completely eliminated and the compute cluster is actually a storage cluster at the same time. Ceph stores object data and it is tied to the datanode processes in Hadoop. This new architecture is shown in Figure 2.

4.2. Initial Object Store Scan

As shown in Figure 2, Ceph object store and Hadoop datanode processes are tied together and there can be multiple methods to integrate Hadoop with the underlying Ceph storage system. The most straight-forward approach would be to transfer object data directly from Ceph to HDFS. Another approach can be using Ceph as the back-end storage of Hadoop and implementing Hadoop data management policies there. Instead of these methods, this paper suggests an alternative approach, *co-locating Ceph object store with Hadoop processes on the same physical node and creating symbolic links in HDFS for data that already exists in Ceph*. Creating symbolic links is a fairly fast operation, eliminates the need to transfer data to HDFS and it will be discussed in Section 4.4. In order to create symbolic links for the existing data in Ceph, certain properties of Ceph objects must be known. The proposed approach makes these properties visible to HDFS, therefore makes Hadoop aware of existing Ceph data, through *Global Information File*, which is created during the initial object store scan.

Figure 3 shows the process of initial object store scan. As a first step, Hadoop namenode process communicates with Ceph Metadata Server (MDS) and it asks for the metadata information of all the existing objects in Ceph. Metadata information provided by Ceph to Hadoop is discussed in Section 4.3. After receiving the metadata from Ceph, Hadoop namenode saves this information in the *Global Information File* and writes it to HDFS (as if it is a regular file in HDFS), as shown in the second step in Figure 3. As mentioned earlier, Hadoop and Ceph preserve their stand-alone functionality in this approach and that makes it possible to copy *Global Information File* to HDFS.

The procedure to create and distribute the *Global Information File* is less than ten seconds for even 150 GB of data in Ceph and this is a one-time operation that is performed when Hadoop is initialized. Hadoop trusts the information in the *Global Information File* and once Hadoop daemons are successfully started, Ceph daemons do not even have to run anymore.

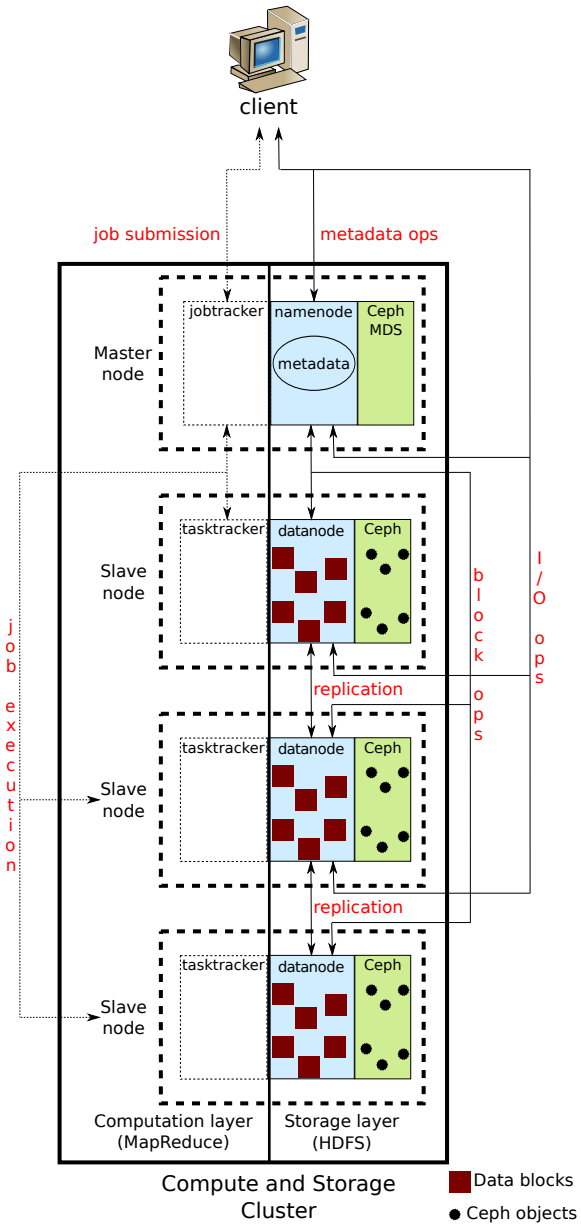

Figure 2: Proposed Hadoop Architecture

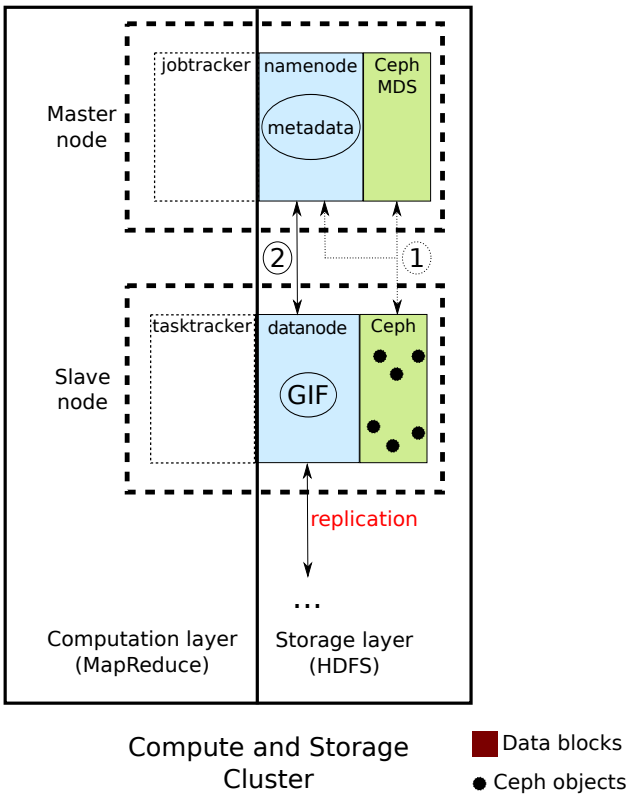

Figure 3: Initial Integration of Ceph Object Store with Hadoop Datanodes

If new data is available in Ceph, Ceph can be scanned again to regenerate the *Global Information File*. Hadoop daemons will pick up the updated information after a restart. It is important to note that incrementally updating the *Global Information File* with each write operation in a write-heavy workload will be expensive as the existing data blocks will be scanned over and over again. This work is geared towards updating the *Global Information File* asynchronously on already existing data in a write-once-read-many workload, which is the optimal use case for MapReduce applications.

4.3. Contents of the Global Information File

The *Global Information File* is created during the initial data scan, as discussed in Section 4.2, and it is crucial for the integration of Hadoop and Ceph object store. Following is the summary of information stored in the *Global Information File*.

- *File names* and *pre-calculated block names* are used by Hadoop to identify objects that already exist in Ceph. Pre-calculated block names are formed by hashing object name, object location and replication level provided by Ceph. If a block does not follow this naming convention, Hadoop will treat that block as a regular HDFS block and this will make it possible for Hadoop to preserve its normal functionality. Meanwhile, not all Ceph objects have to be visible to Hadoop. Any object that is not scanned will not be in the *Global Information File* and therefore, will not be visible to Hadoop applications. Any application or user can read the *Global Information File* and find the *pre-calculated block name* using a *file name*.

- *Replica locations* are of critical importance. As it is the case with any other storage system, Ceph has its own replica placement policy and it is preserved in the proposed approach, as the ultimate goal is to perform in-place computation on existing data without moving it anywhere else.

- *Absolute paths* are necessary while creating symbolic links to already existing Ceph data from Hadoop.

- As this work does not ingest any data into HDFS, Hadoop does not know about the sizes of existing Ceph objects. The *file sizes* are fed from the *Global Information File* to Hadoop for MapReduce operations to work properly.

Table 1 shows an example of two rows from the *Global Information File*.

File Name	Pre-calculated Block Name	Replica Location	Absolute Path	File Size
testfile1	blk_582040001401	node1	/mnt/ceph/dir1/obj1	65536
testfile2	blk_381980001533	node4	/mnt/ceph/dir1/obj2	65536

Table 1: Example Rows from the *Global Information File*

4.4. Creating Symbolic Links

Once the *Global Information File* is written to HDFS and distributed to each datanode, datanodes can read metadata from the *Global Information File* and create symbolic links for the existing Ceph data accordingly. This procedure is illustrated in Figure 4.

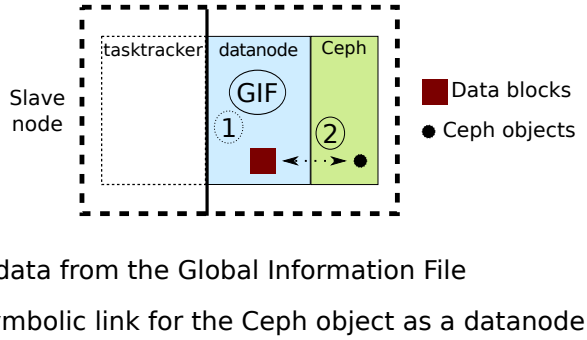

Figure 4: Creating Symbolic Links for Existing Ceph Data

As part of this work, symbolic link creation function in Hadoop is modified to accept two arguments; *file name* and *replica location*. At first, datanode reads a row from the *Global Information File*. In order to create a symbolic link, the datanode needs to know the *absolute path* of the Ceph object on that node (i.e. /mnt/ceph/dir1/obj1) and the *pre-calculated block name* (i.e. blk_381980001533) that will be assigned to the newly created symbolic link. Both of these parameters can be found from the *Global Information File* by finding the row that matches with the given *file name* (i.e. testfile1) and *replica location* (i.e. node1). After retrieving this information, the datanode creates the symbolic links and changes its metadata; such that the size of the symbolic link is equal to the actual size of the object. This is accomplished by using the *file size* (i.e. 65536) information from the *Global*

Information File. Setting the correct *file size* is important as MapReduce operations reading data through these symbolic links need to know the exact size of the data for the correctness of I/O operations.

It is important to note that symbolic links are created for data blocks only. Since data is not ingested into HDFS in the proposed approach, Hadoop creates a metadata block of negligible size for a Ceph object. Metadata creation is dominated by checksum calculation and reading no data from Ceph means having an empty checksum. Additionally, the *Global Information File* has most of the metadata needed for symbolic link creation. As a result, Hadoop metadata block creation overhead is negligible. Checksum implementation is left as a future work item as discussed in Section 6.

Another important design decision is to disable datanode block scans. Hadoop datanodes are responsible for scanning data blocks they own and they report bad blocks to the namenode. Symbolic links created for existing Ceph data are detected and invalidated during block scans and to prevent this from happening, datanode block scans have to be disabled. Configuration changes for datanode block scans are described in Section 5.2. Modifying the datanode block scan in order to prevent it catching symbolic links while resuming its normal functionality is also left as a future work item, as discussed in Section 6.

4.5. Performing a MapReduce Application

At this point, initial data scans and symbolic links creations are done and any MapReduce application can be executed on Ceph objects as if they are located in HDFS. Since the proposed approach ingests no data to HDFS and creates symbolic links only for the Ceph objects located in the same physical node, that means mappers have to work local data. Therefore, the scheduling policy of mappers has to be modified. This is accomplished by changing MapReduce task scheduler code; such that, it assigns local map tasks if a MapReduce operation is executed on symbolic links (existing Ceph data). This also requires MapReduce split size to be exactly the same with the split size of the underlying storage systems. Configuration changes for the MapReduce split size are also described in Section 5.2.

5. Performance Evaluation

This section describes the experimental setup first followed by explanation of the performance evaluation tests and the discussion of the test results.

5.1. Experimental Setup

Experimental evaluations are conducted using five Google Compute Engine [37] instances. Each instance has two Intel Sandy Bridge vCPUs, 7.5 GB of memory and 250 GB of storage. The instance are grouped in an instance group in *us-central1-a* zone to simulate a cluster consisting of five nodes. Each instance has internal and external network access configured and passwordless SSH connection enabled to other instances.

5.2. Hadoop Configuration Parameters

In order to have a fair comparison against stock Hadoop implementation, the configuration parameters (dfs and mapreduce) are preserved during all performance evaluation tests. Table 2 shows the configuration parameters used for the experimental evaluations.

Hadoop Configuration Parameter	**Values**
dfs.replication	2 or 3
dfs.datanode.max.receivers	81920000
dfs.datanode.socket.write.timeout	0
dfs.datanode.scan.period.hours	-1
mapred.child.java.opts	-Xmx6144m
mapred.task.timeout	0
mapreduce.map.output.compress	true
mapreduce.map.output.compress.codec	org.apache.hadoop.io.compress.GzipCodec
mapred.child.ulimit	unlimited
mapred.min.split.size	matches underlying storage (see below for explanation)

Table 2: Hadoop Configuration Parameters

Details on these configurations are available in the Hadoop documentation [38]. There are two important configuration parameters that should be further explained here - *dfs.datanode.scan.period.hours* and *mapred.min.split.size*. Since the proposed method in this work does not really write any data to HDFS; but, rather creates symbolic links to existing data, datanode scans catch these links. In order to make Hadoop work properly with symbolic links, datanode scans are disabled. Additionally, the minimum split size of MapReduce, *mapred.min.split.size*, matches that of the underlying storage system, so that they work on the same number of splits for a fair comparison.

5.3. Performance Tests

Experimental evaluations are conducted using Hadoop 1.1.2 stable version and Ceph 0.94(Hammer) release. The benchmarks used are *Grep* [11], *Wordcount* [12], *TestDFSIO* [13] and *TeraSort* [14]. These benchmarks are commonly used to evaluate Hadoop applications (i.e. Tantisiriroj et al. [24] uses *Grep*, Maestro [23] uses *Wordcount*, Kulkarni et al. [34] uses *TestDFSIO* and Ananthanarayanan et al. [25] uses *TeraSort*) and they have different characteristics in terms of the size of data they use or generate. *Grep* searches for a pattern in a potentially large file and generates a small set of output containing matches. *Wordcount* is similar, but it generates much larger output. *TestDFSIO* and *TeraSort* generate their own input data and perform their tests on the generated data. *TestDFSIO* performs basic I/O operations (read and write in this work) on the generated data. Number of files to perform I/O and size of the I/O operation are configurable parameters of *TestDFSIO*. *TeraSort* sorts data generated by the *TeraGen* benchmark and optionally, sorted data can be verified with *TeraValidate* benchmark. The size of data produced by *TeraGen* is also a configurable parameter.

5.4. Test Results

Table 3 shows the parameters used during the evaluations.

Test Parameter	**Values**
Total number of nodes	3, 5
Replication levels	2 replicas, 3 replicas
Benchmarks	Grep, Wordcount, TestDFSIO, TeraSort
Input size per file	Grep (25 MB, 242 MB, 2.4 GB) Wordcount (29 MB, 286 MB, 2.8 GB) TestDFSIO (500 MB, 5000 MB, 15000 MB, 25000 MB, 50000 MB) TeraSort (1 GB, 10 GB, 50 GB)

Table 3: Test Parameters

5.4.1. Grep

The results of the experimental evaluations for the *Grep* benchmarks are discussed first. This benchmark searches for a pattern in a given file and extracts the occurrences of the given phrase to the resulting output file - so, it generates output much smaller in size compared to its input.

In this test case, 100 files that are equal to each other in size (25 MB, 242 MB or 2.4 GB) are generated first. Stock Hadoop creates these files in HDFS and writes data to each. In the proposed implementation, these files already exist in Ceph and the only requirement is to make Hadoop aware of these existing files through symbolic links during its initialization. Following the data creation, *Grep* benchmark is executed on the newly created data. The test steps outlined above are performed for both 3 nodes with 2 replicas and 5 nodes with 3 replicas.

The upper sub-plot of Figure 5 shows the measurements of the total time it takes to copy data into HDFS. As can be inferred, the proposed Hadoop implementation takes significantly less time than stock Hadoop to copy data into HDFS; because, no data is actually ingested into HDFS. As the input size per file is increased from 25 MB to 2.4 GB, the time it takes to copy data into HDFS increases for stock Hadoop. As no data is written to HDFS, that time stays constant for the proposed implementation. An improvement of 95% in terms of initial data copy performance is achieved and as the input size becomes larger, this improvement will become even higher. Another conclusion from Figure 5 is that the number of replicas or total storage nodes in the system does not have a significant effect on the performance of the data copy phase.

Improving the performance of the MapReduce phase is not a primary goal of this work, as the optimizations are targeted for the initial data copy phase; but, as the input size per file becomes larger (i.e. 2.4 GB), nearly 5% of improvement in MapReduce performance is observed due to data-compute locality. This is achieved by using the object storage system to identify replicas and force Hadoop to co-locate compute threads with the appropriate replica. Note that, there are a variety of existing techniques to co-locate data and compute in Hadoop; but, they do not help improving the performance of data copy phase, which is the primary goal of this work. Similar to the data copy phase, the number of replicas or total storage nodes does not have a significant effect on the performance of the MapReduce phase.

5.4.2. Wordcount

This section gives the evaluation results for the *Wordcount* benchmark. *Wordcount* counts the number of occurrences of each word in a given file and saves these numbers in an output file that is of similar size as the input.

Test parameters of the *Wordcount* benchmark are exactly the same with those used for the *Grep* benchmark in Section 5.4.1; except for the file sizes

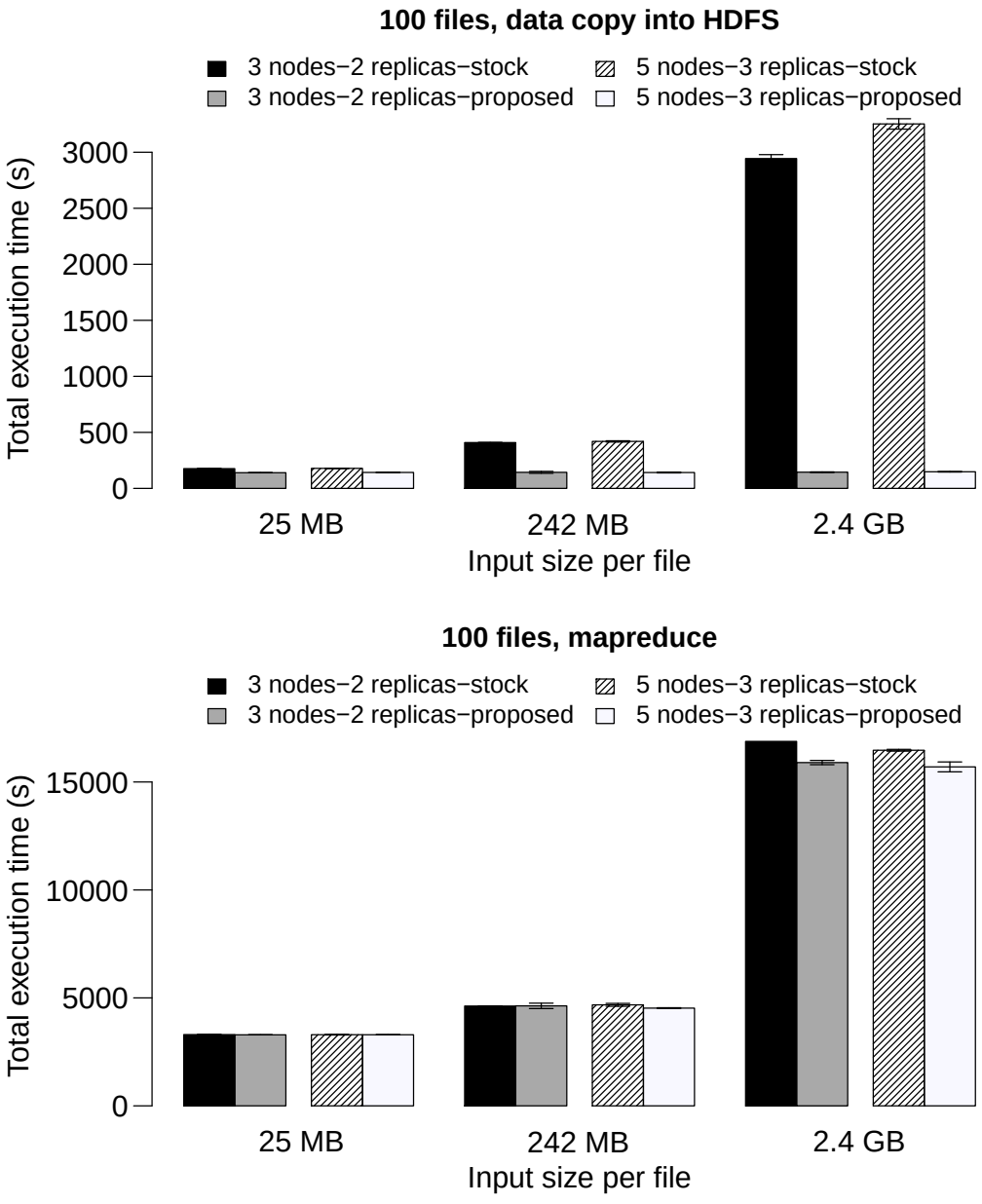

Figure 5: Evaluation results for *Grep*

(29 MB, 286 MB or 2.8 GB per file).

The upper sub-plot of Figure 6 shows the total time to copy data into HDFS. The performance improvement is nearly 95%, similar to the improvement observed for the *Grep* benchmark in Section 5.4.1. Additionally, increasing the input size per file, the number of replicas or total storage nodes has similar impacts and MapReduce performance is improved by nearly 5% as data and computation are co-located, with the number of replicas or storage nodes having no significant effect.

5.4.3. TestDFSIO

This section presents the experimental evaluation results for the *TestDFSIO* benchmark. This benchmark generates its own input with its *write* option using MapReduce, rather than the traditional data ingest method. It is possible to specify the number and size of files to generate with *TestDFSIO* benchmark. In this test case, *TestDFSIO* does not generate any input data, because its input already exists in the system. When a *TestDFSIO* run is completed, it dumps statistics about the benchmark performance (throughput, execution time, io rate etc.).

For the sake of simplicity and as the number of input files will not have a significant effect on the outcome of *TestDFSIO* tests, *TestDFSIO* benchmark is tested with a single file and the file size is varied between 500 MB and 50000 MB. Stock Hadoop creates these files with the *write* option of *TestDFSIO* and then performs a read operation on them. The implementation presented in this work performs a zero-length write that sets up symbolic links to already existing data, followed by a read operation. These test steps are performed for both 3 nodes with 2 replicas and 5 nodes with 3 replicas.

Figure 7 shows the data copy and MapReduce performance of the proposed implementation compared with that of stock Hadoop. First conclusion to draw is that regardless of the input data size and the number of replicas and storage nodes, our Hadoop implementation spends the same amount of time for the initial ingestion of data. On the other hand, the time it takes for stock Hadoop to create the data with the *write* option of *TestDFSIO* increases as the file size is increased from 500 MB to 50000 MB. At 50000 MB, the proposed implementation achieves a 96% improvement over the stock Hadoop implementation in terms of data copy performance. For the MapReduce phase, as the file size is increased, our Hadoop implementation performs better when compared to stock Hadoop. Since local map tasks are used, which means data and computation are co-located, the time it takes to

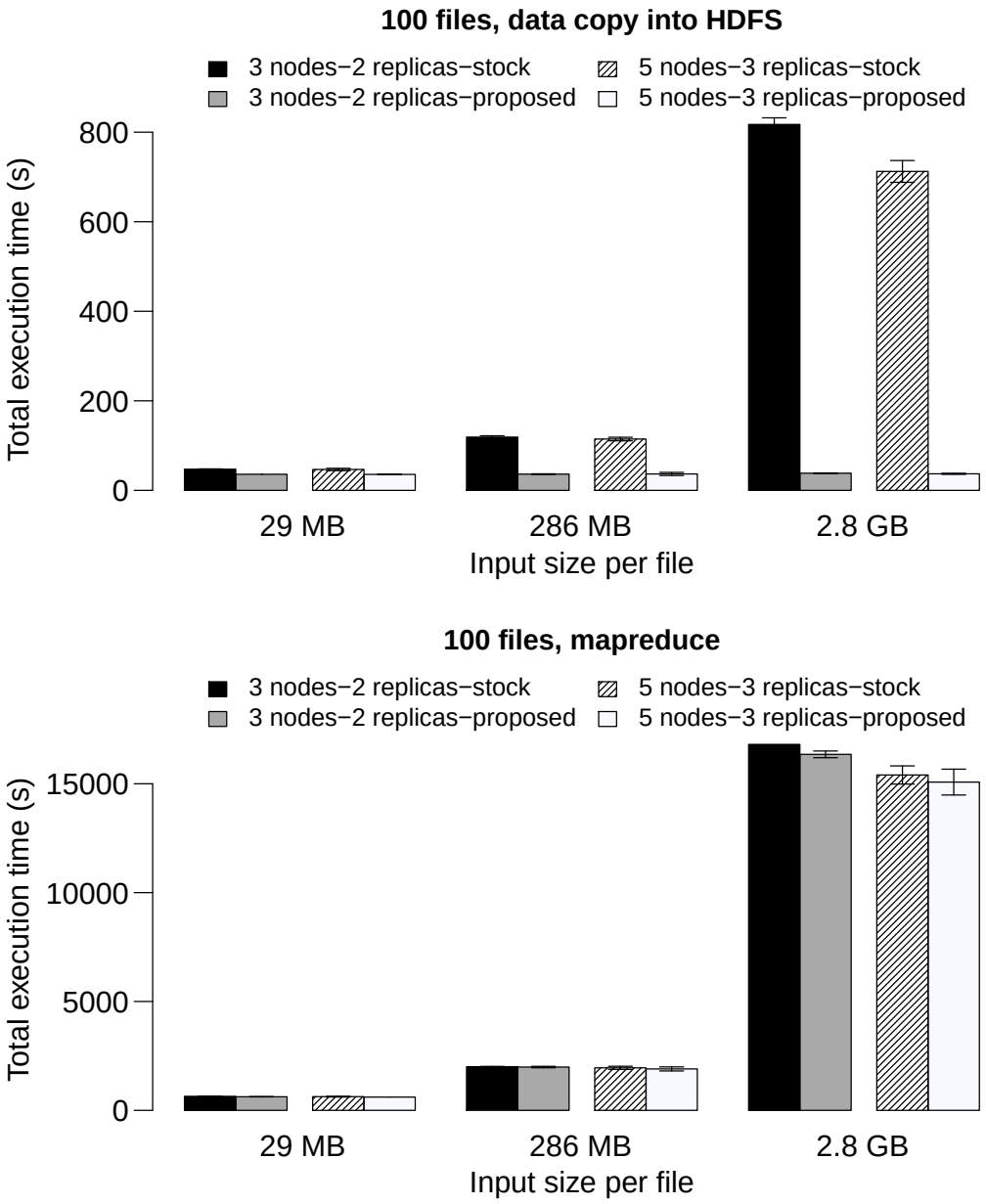

Figure 6: Evaluation results for *Wordcount*

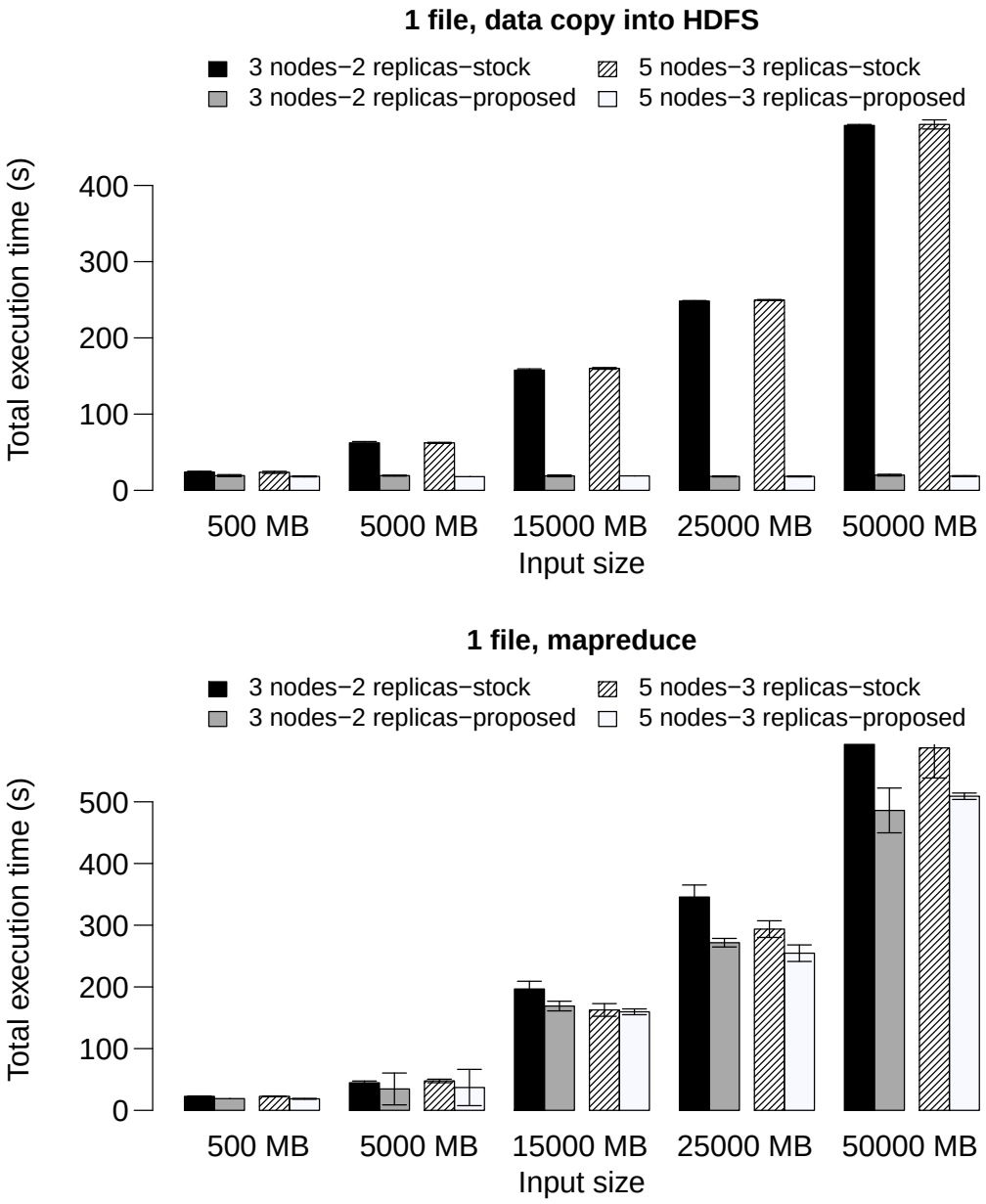

Figure 7: Evaluation results for *TestDFSIO*

shuffle mapper outputs to reducers is much smaller. Additionally, *TestDFSIO* MapReduce phase is dominated by reading the generated data which happens totally local, making the MapReduce improvement more apparent. As a result, MapReduce performance is improved by nearly 20% with the number of replicas or storage nodes having no significant effect. *TestDFSIO* test results are highly variant and this is a known issue with the *TestDFSIO* benchmark [39].

5.4.4. TeraSort

Finally, this section presents the experimental evaluation results for the *TeraSort* benchmark.

Similar to the *TestDFSIO* benchmark, *TeraSort* generates its own input with the *TeraGen* benchmark using MapReduce, rather than the traditional data ingest method. The input data generated by *TeraSort* consists of 100-byte rows. It is possible to specify the size of input data to be generated with *TeraGen*. While evaluating the proposed changes, *TeraSort* does not generate any input data as its input already exists.

The input size for *TeraSort* benchmark is varied between 5 GB, 10 GB and 50 GB during the performance evaluations. Stock Hadoop implementation creates files of these sizes with the *TeraGen* benchmark and then sorts them with *TeraSort*. Our Hadoop implementation performs a zero-length write when *TeraGen* is executed and creates symbolic links to already existing data. *TeraSort* sorts generated data and finally *TeraValidate* is executed to validate the sorted data. This test case is performed for both 3 nodes with 2 replicas and 5 nodes with 3 replicas.

The upper sub-plot of Figure 8 shows that data copy times increase as the input size is increased from 5 GB to 50 GB for both stock and proposed Hadoop. This is expected for stock Hadoop; but, for our implementation, data copy times increase because records created by the *TeraGen* are still processed one-by-one. *TeraGen* generates data in the format of 100-byte records (i.e. to generate 6553600 bytes of data, 65536 records are needed) and it writes data to each record individually. In the implementation presented in this work, these records are created; but, actual data is not committed to HDFS resulting in empty records. As the input size is increased from 5 GB to 50 GB, the number of records to create increases as well, which in turn increases data copy time. Still, a data copy performance improvement by up to 73% is observed, which becomes even bigger with larger input sizes. Additionally, the number of replicas and storage nodes does not af-

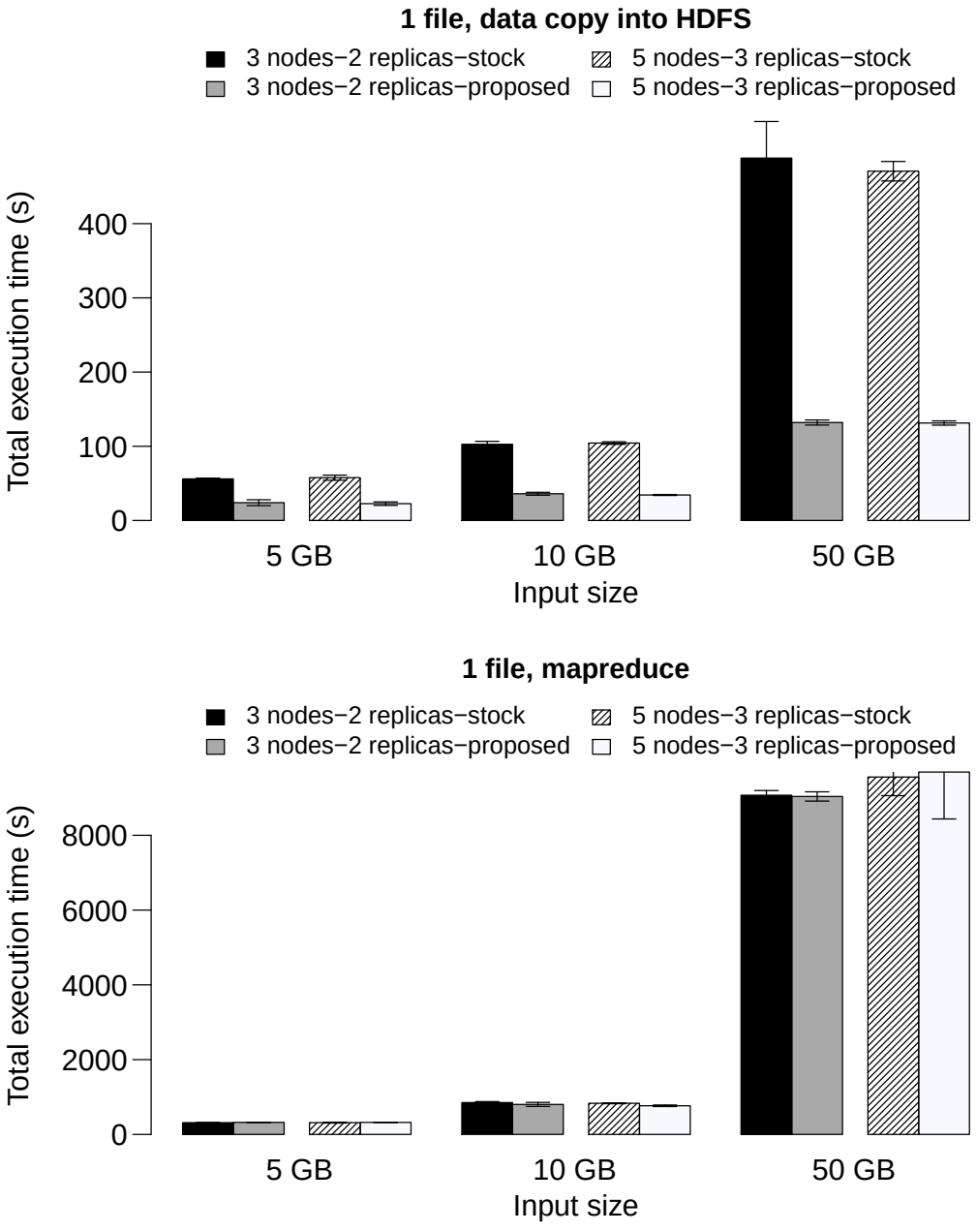

Figure 8: Evaluation results for *TeraSort*

fect the data copy performance significantly. For the MapReduce phase, nearly 5% improvement is observed due to data-compute locality, with the number of replicas or storage nodes having no significant effect. MapReduce performance improvements in this test case are not as high as those from *TestDFSIO*, as the MapReduce phase is not dominated by reads only.

6. Conclusions

In this paper, we presented an approach that performs computation on existing large-scale data in an object storage system without moving data anywhere and analyzed the outcomes of this approach. Experimental evaluations with Hadoop and Ceph object-based storage system show that it is possible to implement Hadoop on top of Ceph as a lightweight computational framework and to perform computational tasks in-place alleviating the need to transfer large-scale data to a remote compute cluster. Initial data copy performance is improved by up to 96% and MapReduce performance is improved by up to 20%.

Future research directions include implementing checksum calculation algorithms, tests with more data and larger storage systems and making datanode block scans work with symbolic links. The proposed approach in this paper does not read any data into HDFS and performing checksum calculations without reading any data into HDFS is tricky; because, Hadoop and the underlying storage system might have different checksum calculation algorithms. As an example; Hadoop uses CRC32; but, if the underlying storage system uses another checksum algorithm (i.e. MD5), this at least requires converting one checksum calculation method to another, which is a very expensive operation, even if no data is read into HDFS. Additionally, performance evaluation tests in this work are conducted with five nodes with three replicas at most. Further tests with different number of nodes and replicas would be helpful to prove the effectiveness of our approach. Finally, datanode block scans are disabled as they are detecting and invalidating the symbolic links created in the proposed approach. They can be changed to resume their normal functionality while not invalidating symbolic links; instead of disabling them all together.

7. Acknowledgement

This work was supported in part by a NSF High End Computing University Research Activity grant (award number CCF-0937879). Any opinions,

findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the NSF.

[1] Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2/.

[2] J. Gantz, D. Reinsel, The Digital Universe in 2020: Big Data, Bigger Digital Shadows, Biggest Growth in the Far East, IDC iView: IDC Analyze the Future (2012) 1–16.

[3] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, J. Zelenka, A cost-effective, high-bandwidth storage architecture, SIGPLAN Not. 33 (11) (1998) 92–103. doi:10.1145/291006.291029.
URL http://doi.acm.org/10.1145/291006.291029

[4] M. Mesnier, G. R. Ganger, E. Riedel, Object-based storage, IEEE Communications Magazine 41 (8) (2003) 84–90. doi:10.1109/MCOM.2003.1222722.

[5] Lustre, http://wiki.lustre.org/index.php/Main_Page.

[6] C. Maltzahn, E. Molina-Estolano, A. Khurana, A. J. Nelson, S. A. Brandt, S. Weil, Ceph as a Scalable Alternative to the Hadoop Distributed File System, ;login::The USENIX Magazine 35 (4) (2010) 38–49.

[7] Swift: Openstack object storage, https://wiki.openstack.org/wiki/Swift (2015).

[8] N. Ali, A. Devulapalli, D. Dalessandro, P. Wyckoff, P. Sadayappan, Revisiting the metadata architecture of parallel file systems, in: Petascale Data Storage Workshop, 2008. PDSW '08. 3rd, 2008, pp. 1–9. doi:10.1109/PDSW.2008.4811892.

[9] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, in: Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, 2010, pp. 1–10. doi:10.1109/MSST.2010.5496972.

[10] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, C. Maltzahn, Ceph: A scalable, high-performance distributed file system, in: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06, USENIX Association, Berkeley, CA, USA, 2006, pp. 307–320.
URL http://dl.acm.org/citation.cfm?id=1298455.1298485

[11] Hadoop Grep, https://wiki.apache.org/hadoop/Grep.

[12] Hadoop WordCount, https://wiki.apache.org/hadoop/WordCount.

[13] Benchmarking and Stress Testing an Hadoop Cluster with TeraSort, TestDFSIO & Co., http://www.michael-noll.com/blog/2011/04/09/benchmarking-and-stress-testing-an-hadoop-cluster-with-terasort-testdfsio-nnbench-mrbench/.

[14] Hadoop TeraSort, https://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/examples/terasort/package-summary.html.

[15] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113. doi:10.1145/1327452.1327492.
URL http://doi.acm.org/10.1145/1327452.1327492

[16] T10 Technical Committee of the InterNational Committee on Information Technology Standards, Object-Based Storage Devices - 3 (OSD-3), http://www.t10.org/.

[17] P. H. Carns, W. B. Ligon, III, R. B. Ross, R. Thakur, Pvfs: A parallel file system for linux clusters, in: Proceedings of the 4th Annual Linux Showcase & Conference - Volume 4, ALS'00, USENIX Association, Berkeley, CA, USA, 2000, pp. 28–28.
URL http://dl.acm.org/citation.cfm?id=1268379.1268407

[18] Amazon Simple Storage Service, http://aws.amazon.com/s3/.

[19] K. V. Shvachko, HDFS Scalability: The Limits to Growth, ;login::The USENIX Magazine 35 (2) (2010) 6–16.

[20] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, E. Harris, Scarlett: Coping with skewed content popularity

in mapreduce clusters, in: Proceedings of the Sixth Conference on Computer Systems, EuroSys '11, ACM, New York, NY, USA, 2011, pp. 287–300. doi:10.1145/1966445.1966472.
URL http://doi.acm.org/10.1145/1966445.1966472

[21] G. Porter, Decoupling storage and computation in hadoop with superdatanodes, SIGOPS Oper. Syst. Rev. 44 (2) (2010) 41–46. doi:10.1145/1773912.1773923.
URL http://doi.acm.org/10.1145/1773912.1773923

[22] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, J. McPherson, Cohadoop: Flexible data placement and its exploitation in hadoop, Proc. VLDB Endow. 4 (9) (2011) 575–585. doi:10.14778/2002938.2002943.
URL http://dx.doi.org/10.14778/2002938.2002943

[23] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, S. Wu, Maestro: Replica-aware map scheduling for mapreduce, in: Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on, 2012, pp. 435–442. doi:10.1109/CCGrid.2012.122.

[24] W. Tantisiriroj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, R. B. Ross, On the duality of data-intensive file system design: Reconciling hdfs and pvfs, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, ACM, New York, NY, USA, 2011, pp. 67:1–67:12. doi:10.1145/2063384.2063474.
URL http://doi.acm.org/10.1145/2063384.2063474

[25] R. Ananthanarayanan, K. Gupta, P. Pandey, H. Pucha, P. Sarkar, M. Shah, R. Tewari, Cloud analytics: Do we really need to reinvent the storage stack?, in: Proceedings of the 2009 Conference on Hot Topics in Cloud Computing, HotCloud'09, USENIX Association, Berkeley, CA, USA, 2009.
URL http://dl.acm.org/citation.cfm?id=1855533.1855548

[26] Using Lustre with Apache Hadoop, Sun Microsystems Inc., http://wiki.lustre.org/images/1/1b/Hadoop_wp_v0.4.2.pdf.

[27] L. Rupprecht, R. Zhang, D. Hildebrand, Big Data Analytics on Object Stores: A Performance Study, in: SC14: International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 2014.

[28] Y. Cheng, M. S. Iqbal, A. Gupta, A. R. Butt, Cast: Tiering storage for data analytics in the cloud, in: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '15, ACM, New York, NY, USA, 2015, pp. 45–56. doi:10.1145/2749246.2749252.
URL http://doi.acm.org/10.1145/2749246.2749252

[29] M. Sevilla, I. Nassi, K. Ioannidou, S. Brandt, C. Maltzahn, Supmr: Circumventing disk and memory bandwidth bottlenecks for scale-up mapreduce, in: Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International, 2014, pp. 1505–1514. doi:10.1109/IPDPSW.2014.168.

[30] A. Kathpal, G. Yasa, Nakshatra: Towards running batch analytics on an archive, in: Modelling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on, 2014, pp. 479–482. doi:10.1109/MASCOTS.2014.67.

[31] B. Palanisamy, A. Singh, N. Mandagere, G. Alatorre, L. Liu, Mapreduce analysis for cloud-archived data, 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing 0 (2014) 51–60. doi:http://doi.ieeecomputersociety.org/10.1109/CCGrid.2014.13.

[32] M. Mihailescu, G. Soundararajan, C. Amza, Mixapart: Decoupled analytics for shared storage systems, in: Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems, HotStorage'12, USENIX Association, Berkeley, CA, USA, 2012, pp. 2–2.
URL http://dl.acm.org/citation.cfm?id=2342806.2342808

[33] N. Rutman, Map/Reduce on Lustre (white paper). Technical report., Xyratex Technology Limited, Havant, UK, 2011.

[34] W. Yu, O. Kulkarni, Progress Report on Efficient Integration of Lustre and Hadoop/YARN, The Lustre User Group Conference, Miami, FL, USA, 2014.

[35] C. Xu, R. Goldstone, Z. Liu, H. Chen, B. Neitzel, W. Yu, Exploiting analytics shipping with virtualized mapreduce on hpc backend storage servers, Parallel and Distributed Systems, IEEE Transactions on PP (99) (2015) 1–1. doi:10.1109/TPDS.2015.2389262.

[36] E. Wilson, M. Kandemir, G. Gibson, Will they blend?: Exploring big data computation atop traditional hpc nas storage, in: Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on, 2014, pp. 524–534. doi:10.1109/ICDCS.2014.60.

[37] Google Cloud Platform - Compute Engine, https://cloud.google.com/compute/.

[38] Hadoop Cluster Setup, http://hadoop.apache.org/docs/r1.2.1/cluster_setup.html.

[39] Hadoop JIRA HDFS-941, https://issues.apache.org/jira/browse/HDFS-941.