©2011

Hrishikesh Gadre

ALL RIGHTS RESERVED

INVESTIGATING MAPREDUCE FRAMEWORK EXTENSIONS FOR EFFICIENT PROCESSING OF GEOGRAPHICALLY SCATTERED DATASETS

By

HRISHIKESH GADRE

A thesis submitted to the

Graduate School – New Brunswick

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree

Master of Science

Graduate program in Electrical and Computer Engineering

Written under the direction of

Professor Manish Parashar

And approved by

_____

_____

_____

New Brunswick, New Jersey

OCTOBER 2011

ABSTRACT OF THE THESIS

INVESTIGATING MAPREDUCE FRAMEWORK EXTENSIONS FOR EFFICIENT PROCESSING OF GEOGRAPHICALLY SCATTERED DATASETS

By HRISHIKESH GADRE

Thesis Director

Professor Manish Parashar

We observe two important trends brought about by the evolution of Internet in recent years. Firstly to improve end-to-end application performance in presence of bottlenecks in the wide-area Internet communication, modern day Internet services are designed in a decentralized fashion involving geographically distributed datacenters connected through the Internet. Secondly the pervasive nature of Internet services has resulted into an exponential growth in the size of digital information created, captured or replicated. Organizations are keenly interested in mining this information to uncover trends, statistics and other actionable information which can give them competitive advantage. These two trends necessitate the design of a large-scale data processing system which can operate efficiently in a distributed environment involving multiple datacenters connected through the Internet.

In recent years, MapReduce programming model and specifically its open source implementation Hadoop is gaining a lot of traction for performing large-scale data processing in a centralized environment. Our evaluation of different real-world usage scenarios of Hadoop deployments revealed that the organizations with the distributed

datasets are required to copy the entire dataset to a centralized location so that it can be efficiently processed by the Hadoop MapReduce framework. As the Internet evolves growth in the size of distributed datasets would outpace the improvements in the network bandwidth available in the Internet. At that point the approach of copying the entire dataset to a single location using Internet would become infeasible.

In this thesis, we have investigated the possibility of extending the MapReduce and specifically Hadoop framework to operate in a distributed environment involving multiple datacenters connected through the Internet. We also have proposed policies to improve the performance of Hadoop MapReduce framework in a distributed environment. We have observed that our policies improve the performance of Hadoop framework substantially.

ACKNOWLEDGEMENT AND DEDICATION

I would like to thank my advisor Professor Manish Parashar for giving me this opportunity to work on something practical and interesting, for his enthusiasm, his inspiration, his encouragement, his sound advice and great efforts during my research in The Applied Software Systems Laboratory (TASSL). I am grateful to my colleagues at TASSL and other friends at Rutgers University for their emotional support and help, which made my study at Rutgers enjoyable and fruitful. I would like to thank the staff at the Center for Autonomic Computing (CAC) and Department of Electrical and Computer Engineering for their assistance and support. I wish to thank my parents and my sister for their understanding, endless encouragement, and love.

Table of Contents

List of illustrations

1. Introduction

1.1 Motivation

Today evolution of Internet is radically improving every aspect of human society e.g. social networking sites like Facebook provide an opportunity to connect and socialize with other people in a way previously unimaginable or services like Amazon or eBay make the experience of shopping almost effortless.

Most of the modern enterprise applications and services on the Internet require rigorous end-to-end system quality. Even a small degradation in performance and reliability can have a considerable impact on business metrics such as application adoption and site conversion rates. A 2009 Forrester Consulting survey [11] found that a majority of online shoppers cited website performance as an important factor in their online store loyalty, and that 40% of consumers will wait no more than 3 seconds for a page to load before abandoning a site.

The performance and reliability of the Internet has a great influence on the end-to-end performance of any Internet service. Since the Internet is designed as a best-effort network, it cannot provide guarantees about end-to-end performance or reliability. So the services deployed in a centralized environment (i.e. in a single datacenter) are often subjected to bottlenecks in the wide-area Internet communication, resulting in poor performance.

To overcome this problem, modern Internet services are based on decentralized architecture so that they can be geographically distributed (using multiple datacenters).

The key benefit of this approach is the significant decrease in the data volume that needs to be moved, the subsequent network traffic, and the distance the data needs to travel which reduces transmission costs, latency and improves the quality-of-service (QoS).

Another interesting trend brought about by the evolution of Internet is the growth in the size of the digital information created, captured or replicated. A report published by IDC [9] states that overall 988 Exabyte of digital information (including structured as well as unstructured) was generated in year 2010 and predicts that the size of the total information would grow exponentially over coming years. Also over 95% of the digital information is available in unstructured (or semi-structured) form such as music, images, application logs or event streams generated by social networking applications such as Twitter or Facebook.

Organizations are keenly interested in mining this information to uncover trends, statistics and other actionable information which can give them competitive advantage. For example processing the click-stream data from the eCommerce website logs helps to figure out interesting trends in usage patterns and in turn customer behavior. These two trends necessitate a large-scale data processing system which can operate efficiently in a distributed environment.

In recent years, a MapReduce programming model [2, 3] is gaining a lot of traction for performing large scale data processing in a centralized environment. Inspired by the concepts of functional programming, this model divides the required computation into a set of small tasks that execute in parallel on multiple machines, and scales nicely to very large clusters of inexpensive commodity computers. The key benefits of this

programming model include its simple programming interface, ability to process large-scale unstructured (or semi-structured) datasets efficiently as well as ability to achieve massive scalability (and fault-tolerance) without requiring high-end computing infrastructure.

Hadoop [12] is a popular open source implementation of MapReduce programming model, primarily developed by Yahoo! and used by a number of organizations such as Facebook, Amazon, and LinkedIn etc. We noticed that the current design of Hadoop framework is targeted towards a centralized architecture involving a single datacenter. We studied the real-world usage scenarios for Hadoop deployments and observed that the organizations with dataset distributed across multiple datacenters (such as Rackspace [5]) are required to copy the entire dataset to a centralized location so that it can be efficiently processed by Hadoop MapReduce framework.

The feasibility of copying the entire dataset to a single datacenter depends upon a number of factors such that the total size of the data to be copied from each datacenter, total number of datacenters involved in the copy as well as capacity of inter datacenter network. But as the Internet evolves, the growth in the size of geographically scattered datasets would outpace the improvements in the network bandwidth available in the Internet. At that point the approach of copying the entire dataset to a single location using the Internet becomes infeasible.

1.2 Problem statement

As part of this research, we study the design assumptions behind the different components of Hadoop eco-system (specifically Hadoop Distributed File System (HDFS)

and MapReduce) which mandate the processing of MapReduce computations in a centralized environment. For each such design assumption, we propose and implement a policy to improve its performance in a distributed environment. Finally we compare and contrast our policies with the native Hadoop framework in an execution environment involving two datacenters, one at CAC Lab at Rutgers University and the other at Amazon EC2 for different classes of workloads.

1.3 Contributions

Specifically, the contributions of this research are as follows,

- Design and implementation of a WAN aware Hadoop Distributed File System (HDFS) replica placement policy which improves the performance of write operations with respect to the default policy in the range of 100% to 200% depending upon the size of the dataset being written.
- Design and implementation of an Adaptive Reduce Task Scheduling (ARTS) algorithm. This algorithm substantially reduces the total size of the Map phase output copied across the datacenters by taking into consideration the datacenter-locality of the Map phase output while scheduling Reduce tasks.

  Also in case of a shared Hadoop cluster, this algorithm adapts the start of the Reduce phase with respect to the type of the workload executed as well as the size of the job (such as batch or interactive). Hence this algorithm consistently outperforms default Reduce phase scheduling policies in Hadoop framework (such as early-start and delayed-start).

2. Background and related work

2.1 MapReduce Programming Model

To perform large-scale data analysis (e.g. log processing, building web indexes etc.) over a commodity cluster (comprising of thousands of machines); programmer has to make complex decisions such as effective parallelization of computation for speedup, data-distribution, fault tolerance as well as scalability etc. Without support of a suitable parallel programming model, these issues obscure the implementation of original computation with large amounts of complex code to deal with them.

To avoid such complexity, MapReduce programming model is designed, which enables the programmer to express intended computation without the messy details of parallelization, data-distribution, fault-tolerance and scalability. This programming model is inspired by the concepts of ‘map’ and ‘reduce’ primitives in the functional programming languages like LISP and ML.

Parallel processing is a technique to divide the required computation into a set of tasks so that they can be executed concurrently on multiple machines in a cluster. At the end, the results of all the tasks are merged together to produce final result. The framework is also responsible to implement necessary mechanisms for fault-tolerance as well as load-balancing. The complexity of parallel algorithms is estimated in terms of space (memory requirements), time (processor cycles) as well as the communication overhead between different subtasks.

The communication between different subtasks can be performed in different ways. In case of shared-state architecture, a central authority is responsible to maintain the state of the computation. Different parts of computation communicate with each other by updating this shared state. In order to maintain consistency of the underlying data-structures, the concurrent accesses to the shared-state are serialized by implementing appropriate locking protocols. This results into serialization of part of computation as well as extra communication overhead, leading to degraded performance. Also, systems designed with shared-state architecture tend to be difficult to scale.

On the other hand in case of shared-nothing architecture, each of the subtasks is independent and self sufficient, and there is not a single point of contention in the system. This helps to achieve high-degree of parallelism as well as lower communication overhead. Generally, the systems designed with shared-nothing architecture tend to be easy to scale.

2.1.1 Functional programming in the context of parallel processing

The imperative programming paradigm is characterized as having an implicit state that can be modified (i.e. side effected) by constructs in the source language. As a result, this paradigm has a notion of sequencing the constructs to permit a precise and deterministic control over the state.

In contrast, functional programming paradigm is characterized as having no implicit state (i.e. immutable data-structures and no shared state), and the computation is expressed solely as an evaluation of mathematical expressions. The state-oriented computations are accomplished by explicitly passing the state as an argument to the function.

Due to the requirement of immutable data-structures, the systems based on functional programming paradigm are not subject to data-race conditions and hence don't require elaborate locking schemes to ensure the consistency of their data-structures. Hence, we can say that this paradigm is based on the shared-nothing architecture - making it suitable for efficient parallel processing.

Another interesting property of functional programming paradigm is the notion of higher order functions, which accept one or more functions as arguments and optionally return a function as a result. This property is very useful in abstracting the common behavior of the system at one place.

Also, any computation expressed as a functional program is deterministic in nature regardless of the order in which the parts of the computations are performed (known by the Church-Rosser theorem [7]). This property along with the higher order functions can be used to build a software framework which can accept user defined functions as arguments and automatically execute them on a large data-set in a highly parallel fashion on a commodity cluster.

Finally functional programming is declarative in nature. Hence functional programs tend to be much more concise as compared to their imperative counterparts. This helps in improving the programmer productivity.

Thus, the MapReduce programming framework utilizes the concepts of functional programming to provide a concise interface in imperative languages (like C++, Java, Python etc.) to the application programmers to express their computations in the form of two functions 'map' and 'reduce' which are automatically executed in a highly parallel

fashion on a commodity cluster by taking into consideration the factors like fault-tolerance, data-distribution and scalability.

2.1.2 Map function

As shown in figure 1, Map function accepts a set of elements as an input and executes user specified conversion (or mapping) over every input element. It returns all the output elements as a set. Here the input and output elements are expressed as key-value pairs. The result of this mapping function forms the intermediate result of the MapReduce computation.

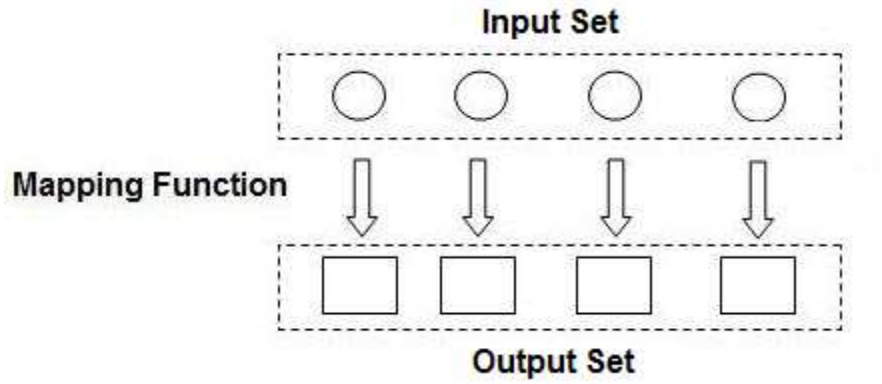

Figure 1: Map operation transforms every element in the input set using the user-specified conversion function

The independent execution of mapping function over the input elements helps to automatically divide the input dataset and perform the mapping operation in parallel over a cluster. After completion of Map operation on individual fraction of the input, system groups the intermediate result based on the key. At the end of Map phase (i.e. after

successful execution Map function over entire dataset), system merges the results of individual groups based on parallel-merge-all strategy [8].

The system then executes a reduce function over a set of intermediate values associated with each of the intermediate key. To improve the parallelization, the key-space for the intermediate keys is uniformly partitioned. This helps parallel execution of reduce function using multiple machines in the cluster.

2.1.3 Reduce function

As shown in figure 2, Reduce function accepts a set of elements as an argument and executes user defined accumulation function over input elements sequentially and returns the final value of the computation as a result (as shown in the figure above).

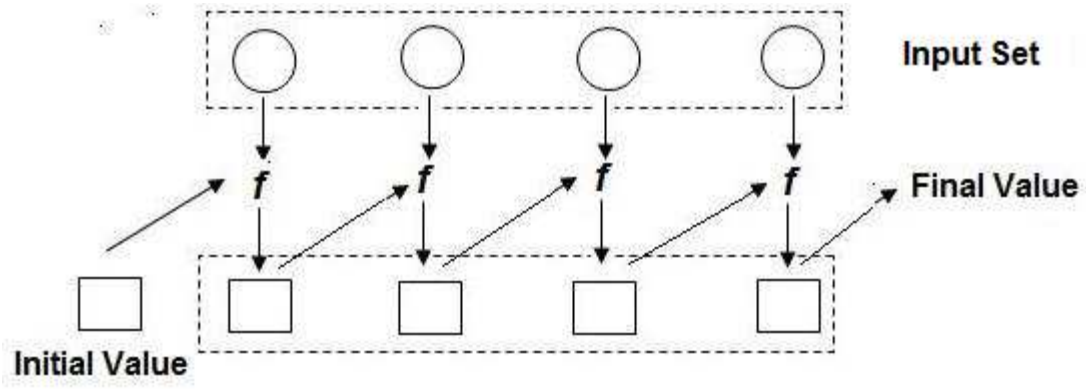

Figure 2: Reduce function iterates over the input set to produce an aggregated value.

2.1.4 Example MapReduce application

A MapReduce framework can be used to perform log processing to extract useful trends in the interaction between the users and the application. For example, one interesting

trend would be to find out distinct client machines interacting with the application along with their request count. Consider following sample of log statements,

66.249.65.107 - - [08/Oct/2010:04:54:20 -0400] "GET /support.html HTTP/1.1" 200 11179 "-" "Mozilla/5.0"

66.249.90.107 - - [08/Oct/2010:04:54:20 -0400] "GET /support.html HTTP/1.1" 200 11179 "-" "Mozilla/5.0"

66.249.90.107 - - [08/Oct/2010:04:54:21 -0400] "GET /contact.html HTTP/1.1" 200 11179 "-" "Mozilla/5.0"

The Map function would accept a set of log statements (as shown above). Here each input element would comprise of key (the line number in the log file) and value (the actual log statement).

The Map function would extract the client IP address from the log statement using the specified log format. The output of the Map operation would be client IP address (as a key) and request count (as a value). During this operation, the request count would always be 1.

```
Function Map (String key, String value)
   //Ignore key (which is a line number) in this case.
  String ipAddress = extractIpAddress(value);
  EmitIntermediate(ipAddress, “1”);
```

The result of applying the Map function on the sample log statements would be

[“66.249.65.107”, “1”]

[“66.249.90.107”, “1”]

[“66.249.90.107”, “1”]

The Reduce function accepts all the intermediate result entries with the same key (in this case client IP address). This function iterates over the input set and outputs the sum of request counts for each client IP address as described below.

```
Function Reduce(String key, Iterator values)
int result = 0;
for each v in values
result += parseInt(v);
Emit(key, AsString(result));
```

The result of reduce operation in case of the sample log statements would be,

[“66.249.65.107”, “1”]

[“66.249.90.107”, “2”]

2.2 Hadoop MapReduce framework

Hadoop is an open source implementation based on the original Google MapReduce [2] and Google File System [3] publications. Hadoop is created by Doug Cutting (the originator of Apache Lucene – a widely used text search library) while developing an open source web search engine called Apache Nutch.

Hadoop eco-system is based on Google Cluster Architecture [1]. This architecture differs from traditional High Performance Computing (HPC) environment in two key

dimensions. Firstly, the reliability of the system is built in the system software rather than depending upon server-class hardware. This helps to build a high-end computing cluster using commodity hardware at a much lower cost. Secondly, this architecture focuses on aggregate request throughput rather than peak server response time. The response time of the application is improved by increasing the parallelization of the request processing tasks.

Since Hadoop is designed to perform large-scale data processing in a clustered environment, it is very important to ensure to that processing nodes in the cluster have fast access to the required data, which ultimately depends upon availability of inter-node network bandwidth. Hence design of datacenter network architecture is one of the key elements to achieve acceptable performance for the given Hadoop cluster.

Basically there are two options for designing underlying communication substrate for a large-scale cluster. One option is to use specialized hardware and communication protocols (like InfiniBand or Myrinet). This option provides excellent scalability for clusters of thousands of machines with very high bandwidth. But due to the specialized hardware requirements, it is quite expensive.

On the other hand, using commodity Ethernet switches and routers to connect different machines in the cluster brings down the cost of datacenter networking setup considerably. This design also resonates with the Google Cluster Architecture [1] and hence used for most of the Hadoop deployments in practice.

The main drawback of using commodity networking equipment is that the aggregate network bandwidth provided across the cluster scales poorly with respect to the cluster

size. Also it is not economically feasible to achieve highest level of aggregate bandwidth as the cost increases non-linearly with respect to the size of the cluster.

To overcome this limitation, Hadoop ecosystem considers network bandwidth as a scarce resource and implements mechanisms to optimize its usage as far as possible. This is done by carefully modeling the topology of datacenter network being used in the Hadoop cluster.

2.2.1 Modeling network topology in Hadoop framework

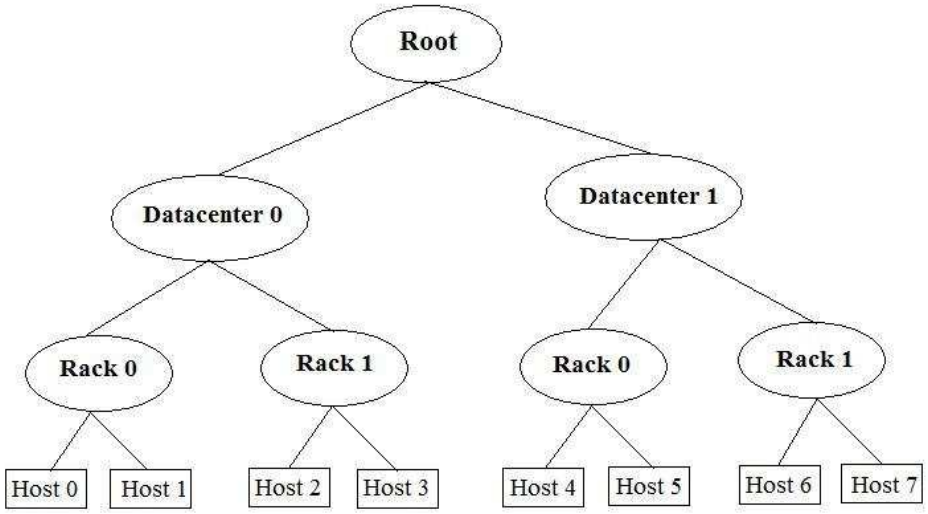

Figure 3: Network topology in Hadoop (Reference: HDFS rack awareness proposal)

As shown in the figure 3, Hadoop models the topology of datacenter network as a tree. The machines in the cluster represent leaves of the tree, which are connected to each other with the help of one or more racks. Typically each rack hosts around 30-40 machines connected to each other by a 1GB/s internal switch. Multiple racks in a single datacenter are connected to each other by core switch (which normally is 1GB/s or better).

The levels in the tree are not predefined and can be configured by the administrator to suit the needs of the deployment. In practice it is common to have levels in the tree which correspond to the datacenter, the rack as well as the machine in the cluster. The distance between any two machines is equal to the sum of their distances to their closest common ancestor.

Since the bandwidth available for each level decreases as we move up the tree from bottom to top, distance between nodes in a tree can be used as a measure of availability of bandwidth between given two machines in the cluster. Hadoop attempts to schedule operations such that the distance between source and destination of data is as small as possible.

The Hadoop infrastructure mainly comprises of following components,

- Hadoop Distributed File System (HDFS)
- Hadoop MapReduce

2.2.2 Hadoop Distributed File System (HDFS)

HDFS is a distributed file-system designed to operate over a cluster of commodity hardware catering high-throughput batch-oriented applications processing large-scale data (typically in the range of terabytes). HDFS is suitable to store small number of very large files (typically in the range of hundreds of gigabytes in size) with write-once and read-many-times access pattern.

A typical application generates dataset and stores it in HDFS once; and performs various analyses on the stored dataset over a period of time. Each analysis would normally

require scanning a large portion of the dataset. Hence a key system requirement is high read-throughput rather than low read-latency. Also, write-once semantic helps to avoid issues related to data coherency due to concurrent updates from multiple clients.

Another key design consideration for HDFS is based on the assumption that it is more efficient to move the computation closer to required data rather than moving data closer to the computation being performed. This helps avoiding network congestion due to large-scale data movement and improving the application performance. To this end, HDFS is designed to be location-aware. This means that HDFS provides necessary functionality to model the network topology of the cluster (in terms of racks, switches and datacenters etc.) as well as interfaces for the applications to discover the location of operated data. This is helpful for the applications (typically MapReduce jobs) to schedule computations over the cluster intelligently in order to minimize network traffic.

Since Hadoop cluster is deployed on a cluster of commodity hardware, a failure of one or more hardware components is a norm rather than exception. Hence a key goal for HDFS is to detect hardware failures and automatically recover from them.

Also, HDFS is developed using Java platform, which facilitates easy portability between different computing platforms.

2.2.2.1 Architecture

Every file stored in HDFS is composed of one or more data-blocks of fixed size. The size of block can be configured at the time of file creation (the default size is 64MB). To guard against data-loss due to hardware failure, every block is replicated on multiple machines in the cluster (default replication factor is 3 although it can be configured at the time of file creation).

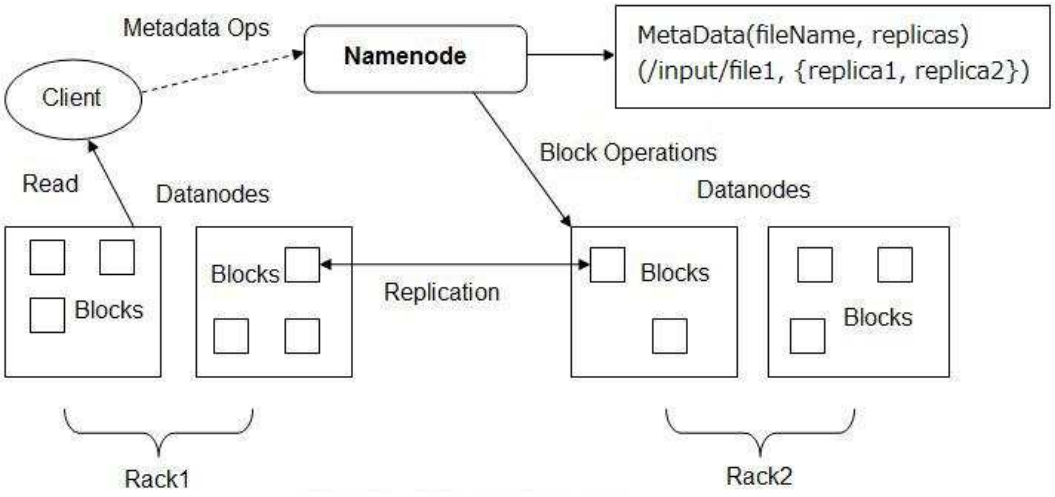

Figure 4: HDFS Architecture

As shown in figure 4, Hadoop File-System (HDFS) is based on Master/Slave architecture. It separates the functionality of management of file-system metadata from handling data-blocks so that each can be delegated to separate components in the system. Every HDFS cluster consists of a single meta-data server (called NameNode in Hadoop terminology) and a group of data-block servers (called DataNode in Hadoop terminology).

NameNode manages the namespace of Hadoop file-system and regulates access to files by clients. It exposes file system namespace operations (like opening, closing, and renaming files and directories) to the HDFS clients. It is also responsible for maintaining the mapping between files and their associated blocks. For every block in the file-system, it keeps an up-to-date information regarding set of DataNodes currently holding its replica. The NameNode also makes decisions related to block-placement and replication.

By having a single NameNode in the HDFS cluster, the design of the system is simplified. It also enables the NameNode to take sophisticated block-placement and replication decisions based on the current state of the cluster. The downside of this design

is the fact that NameNode is a single-point-of-failure in the system and also it limits scalability of Hadoop cluster.

In order to make sure that NameNode doesn't become a bottleneck during the file operations, HDFS never directs actual file read/write operations through NameNode. The client instead requests NameNode for the appropriate DataNode to contact (the one containing data in case of read operation or the one will contain data in case of write operation). The communication between client and the DataNode happen independently without any involvement of the NameNode.

HDFS cluster consists of a number of DataNodes, typically one per every machine in the cluster. DataNode is responsible to serve storage capability for the HDFS cluster using its local storage resources. The DataNode handles read/write requests from the clients as well as commands from the NameNode related to block creation, deletion and replication.

2.2.2.2 Important Operations in HDFS

Read Operation

The figure 5 describes the sequence of operations performed during Read operation using HDFS.

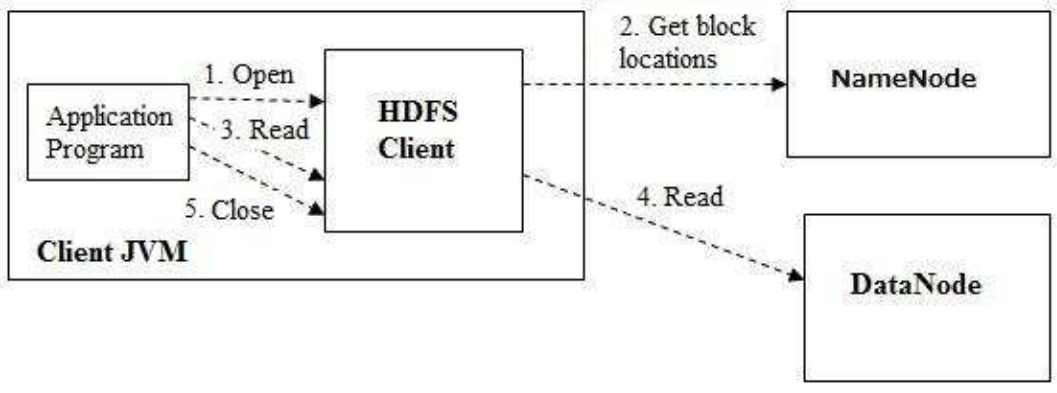

Figure 5: HDFS Read Operation

1. The application program requests the HDFS client to open the required file on its behalf. The HDFS client consults with NameNode to verify if the requested file exists in the system. Upon success, the HDFS client returns application with a file handle which is be used during subsequent file operations. The state of the file handle is maintained exclusively in HDFS client. This design helps keep NameNode completely stateless and hence can be scaled successfully in case of large clusters.
2. HDFS client requests NameNode for the information about the locations of replicas for the first few logical blocks in the file. For each logical block, NameNode ensures that the list of replicas returned is sorted according to its proximity with the requesting client host. This ensures that HDFS client can fetch the contents of file from the nearest possible location in the cluster.
3. Using the file-handle, the application program performs read operations on the file in a streaming mode. Internally HDFS client fetches the contents of entire logical block in one step and buffers them to improve the performance of read operation.
4. For each logical block, HDFS client sequentially contacts DataNodes in the order provided by the NameNode until the read operation succeeds. HDFS client also keeps a track of failed DataNodes and makes sure not involve them during reading subsequent blocks. At the end of read operation, it also verifies checksums of the data transferred from the given DataNode. If a corrupted block is found, it reports the logical block number and the associated DataNode information to the NameNode so that appropriate recovery could be performed.

5. At the end, the application program requests HDFS client to close the file opened for reading by passing the corresponding file handle.

Write Operation

The figure 6 describes sequence of actions performed during write operation using HDFS.

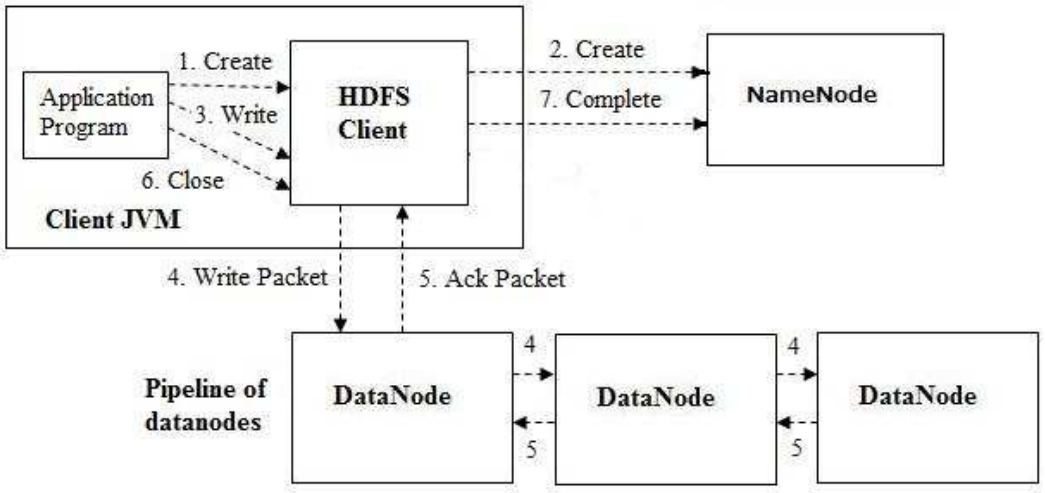

Figure 6: Write operation in HDFS

1. The application program requests HDFS client to create a new file of a given name in HDFS.
2. The HDFS client requests NameNode to create a new file of given name. The NameNode performs various checks to make sure that file doesn't already exist in HDFS as well as the client has necessary write permissions in the requested namespace. Upon verification, NameNode creates an entry for the requested file in the namespace and reports success to HDFS client. HDFS client returns corresponding file handle to the application program.

3. As the application performs write operation using the file handle, the HDFS client splits data being written into logical blocks. The size of each logical block can be configured per file basis. For each block being written, HDFS client requests NameNode to allocate new block in the HDFS cluster. NameNode uses configured replica-placement policy to select a set of DataNodes to hold the contents of the block. The client receives list of DataNodes from NameNode in sorted order based on their location proximity with the client.
4. For each block write operation, the given set of DataNodes form a pipeline among themselves such that each DataNode forwards block contents to the next one in order. This design helps to fully utilize each machine's outbound network bandwidth instead of dividing it amongst multiple recipients (in case of transferring data directly from client to a set of DataNodes). Also since DataNodes are sorted based on their location proximity, each entity in the pipeline has to transfer the block content only to the closest possible entity in the network which reduces the possibility of network bottlenecks.
5. The DataNodes also send back corresponding acknowledgements in the pipelined fashion back to the client. The client waits for acknowledgements from all the DataNodes before declaring write operation as successful.
6. At the end of file operation, the application program requests HDFS client to close the file. The HDFS client ensures that there are no pending acknowledgements and informs NameNode that file is complete.

2.2.2.3 HDFS replica placement policy

During the allocation of a new block, NameNode consults with the configured replica placement policy to determine set of DataNodes to be selected to host the replica of new block. The replica placement policy attempts to trade-off between read-bandwidth, write-bandwidth and the reliability. e.g. to optimize write-bandwidth utilization, the policy can host all the replicas in a single rack (thereby avoiding off-rack write traffic). But this results into lower reliability (e.g. if the rack loses network connectivity, the stored data would be unavailable) as well as lower read-bandwidth (since all the requests for the given block would be directed to a single rack).

The default replica selection policy provided in HDFS is suited for deployments in a single datacenter. This policy works as follows,

- If the client machine is hosting a DataNode process for the HDFS, it is used to place first replica. Otherwise policy selects any DataNode at random after verifying its suitability (e.g. DataNode should have sufficient space available to store the data and it should not be too busy handling other requests).
- The second replica is placed on a DataNode in different rack as compared to first one at random.
- Third replica is placed on a DataNode in the same rack as the second one but on different machine (as compared to second one) selected at random.
- If there are still more replicas needs to be placed, this policy selects DataNodes at random. Here the policy attempts to avoid placing too many replicas on the same rack.

This policy provides excellent balance in all aspects,

- Reliability (since blocks are stored on two different racks)
- Write-bandwidth (since data has to pass through a single network switch during write operation).
- Read-bandwidth (since the request load for a given block can be divided between two racks).

2.3 Hadoop MapReduce framework

Hadoop MapReduce framework is based on Master/Slave architecture. A master (called JobTracker in Hadoop terminology) accepts jobs submitted by the clients and performs 'data-aware' scheduling using a set of workers (called TaskTracker in Hadoop terminology). Master partitions the input data into a set of M splits and processes them in parallel using a set of workers. Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function. The number of partitions (R) is specified by the client.

Figure 7 demonstrates the flow of execution of a MapReduce job using Hadoop MapReduce framework.

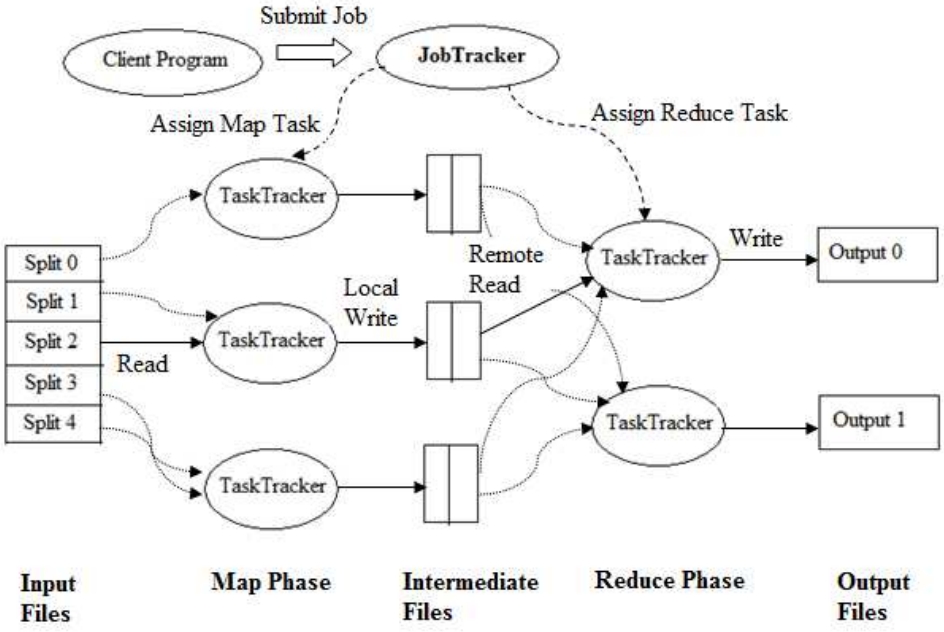

Figure 7: Overview of job execution in Hadoop MapReduce

- After client submits a MapReduce job to the JobTracker, it splits the input dataset into M pieces (each typically 64MB) and creates a Map task per input split.
- JobTracker attempts to schedule each Map task by taking into consideration data locality between the input split (stored on HDFS) and the idle TaskTracker (ready to execute a Map task) such that copy of input dataset over the network is reduced as far as possible.
- During the execution of Map task, the TaskTracker reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory. Periodically, these buffered key/value pairs are written to local disk, partitioned into R regions by the partitioning function. The TaskTracker informs JobTracker about locations of this intermediate output periodically.

- JobTracker inform Reducers the locations of Map phase output. The Reducer in turn copies the contents from the local disks of Mappers using HTTP protocol. When Reducer has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same Reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.
- The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user defined Reduce function. The output of the Reduce function is appended to a final output file for this Reduce partition.
- After successful completion, the output of the MapReduce job is available in the R output files (one per Reduce task).

2.3.1 Hadoop MapReduce Schedulers

In this section, we discuss different schedulers used by the typical Hadoop deployments in practice.

2.3.1.1 Hadoop Fair Share Scheduler

The Hadoop Fair Share scheduler [13] is developed by Facebook in collaboration with University of California, Berkeley. Fair scheduling is a method of assigning resources to jobs such that all jobs get, on average, an equal share of resources over time. When there is a single job running, that job uses the entire cluster. When other jobs are submitted, tasks slots that free up are assigned to the new jobs, so that each job gets roughly the same amount of CPU time. This allows short jobs to finish in reasonable time without

starving long jobs. It is also a reasonable way to share a cluster between multiple users. Finally, fair sharing can also work with job priorities - the priorities are used as weights to determine the fraction of total compute time that each job should get.

This scheduler organizes jobs into "pools", and shares resources fairly between these pools. The pools can be configured for each individual user or for each UNIX group. Within each pool, fair sharing is used to share capacity between the running jobs. Pools can also be given weights to share the cluster non-proportionally.

In addition to providing fair sharing, the Fair Scheduler allows assigning guaranteed minimum shares to pools, which is useful for ensuring that certain users, groups or production applications always get sufficient resources. When a pool contains jobs, it gets at least its minimum share, but when the pool does not need its full guaranteed share, the excess is split between other running jobs. This lets the scheduler guarantee capacity for pools while utilizing resources efficiently when these pools don't contain jobs.

The Fair Scheduler lets all jobs run by default, but it is also possible to limit the number of running jobs per user and per pool through the configuration. This can be useful when a user must submit hundreds of jobs at once, or in general to improve performance if running too many jobs at once would cause too much intermediate data to be created or too much context-switching.

2.4 Related Work

2.4.1 Cotemporary MapReduce frameworks

In this section we compare and contrast between Hadoop MapReduce framework and the other contemporary data processing systems.

2.4.1.1 Sector/Sphere

Sector/Sphere [10] is a software suite designed for high-performance distributed data storage and processing not only within a single datacenter but also across geographically distributed data centers over high speed wide-area-network (greater than 1 Gb/s). This system is originally designed at University of Illinois, Chicago and currently maintained as an Open source project.

Sector is a user-space distributed file system which uses local storage capacity of each node for data storage. Similar to HDFS, this file-system is network topology aware and hence can optimize the network bandwidth used for data transfer. Also it does not support concurrent write operations and is designed mainly for the read intensive workloads.

However the design of Sector differs from HDFS in a number of ways. Firstly, Sector is not a block-oriented file-system like HDFS. This means that it doesn't split the user files into blocks; instead a user file is stored intact on the local file-system of one or more slave nodes. Although this design helps to provide better robustness due to its simplicity and better performance over wide-area-networks; the maximum size of the file which can be stored in Sector cannot exceed the maximum file size supported by the individual nodes in the cluster. Secondly, it uses UDT (UDP based data transfer) protocol for data transfer unlike other distributed file-systems such as HDFS which use TCP protocol. This

enables Sector to provide high performance data I/O across geographically distributed data-centers.

Sphere is a parallel data processing engine integrated in Sector and it can be used to process data stored in Sector with the help of user-defined functions (UDFs) in parallel. The flexibility of specifying the UDFs for processing makes this system more generic to other MapReduce frameworks like Hadoop since MapReduce can be implemented as a special case of UDFs. The Sphere computing platform is also network topology aware and hence can optimize the network bandwidth consumed during computation by making data-locality aware scheduling decisions.

2.4.1.2 Twister

Twister [21], developed at Indiana University is a distributed in-memory MapReduce framework optimized for iterative MapReduce computations. It reads data from local disks of worker nodes and handles Map phase intermediate output in distributed memory of worker nodes. It users publish/subscribe messaging infrastructure for communication and data transfers. Twister assumes that the data read from the local disks are maintained as files and hence supports file based input format, which simplifying implementation of Twister runtime. This approach requires users to split their dataset into a number of files unlike Hadoop Distributed File System (HDFS) which can automatically partition the input dataset into a number of blocks. Twister also supports sending input data for Map tasks directly via messaging infrastructure which is extremely useful in iterative MapReduce computations since it avoids performing disk-write operation to save the results of Reduce phase for the next iteration. Instead the results of Reduce phase of last iteration are directly fed to the Map tasks of next phase via messaging infrastructure.

2.4.1.3 Comet MapReduce

Comet MapReduce framework [22] is being developed at Rutgers University. It is designed using Comet, a decentralized (peer-to-peer) computational infrastructure that extends Desktop Grid environments to support applications that have high computational requirements along with non linear communication requirements. Comet provides a decentralized and scalable tuple space, efficient communication and coordination support, and application-level abstractions that can be used to implement Grid applications. The tuple space is essentially a global virtual shared-space constructed from the semantic information space used by entities for coordination and communication. This information space is deterministically mapped, using a locality-preserving mapping, onto the dynamic set of peer nodes in the Grid system. The resulting structure is a locality preserving semantic distributed hash table (DHT) built on top of a self-organizing structured overlay. This MapReduce framework use the tuple-space provided by Comet infrastructure to propagate the Map phase intermediate output thereby avoid disk read/write operations. This is particularly useful for MapReduce computations involving small to medium scale input datasets.

2.4.2 Large-scale data transfer mechanisms over Internet

The problem of efficiently transferring large datasets over the Internet has been studied extensively in the past. Jim Gray [15], DOT (Data Oriented Transfer service) project [16] and the PostManet project [17] have proposed using postal service as an alternative option for large-scale data transfer over the Internet. In this approach, the entire dataset is loaded to a portable storage device which is shipped via postal service (such as USPS, FedEx or UPS) to the destination site. The dataset is uploaded at the destination site using

the storage device. This helps to completely avoid the copy of dataset over the Internet. Now even Amazon provides a similar mechanism named Import/Export [18] in its Cloud offerings. This approach is suitable in environments where a small set of source sites transfer bulk datasets to a single destination infrequently. But as the number of source sites or the frequency of data transfer increases, the cost of shipping the datasets increase substantially. Hence this approach is not feasible in such environments.

Pandora project [14] attempts to solve the problem of group-based data transfer in an environment involving multiple source sites each hosting a large dataset and single destination site, such that it reduces both total dollar costs incurred by the group as well as the total transfer latency of the collective dataset. It makes use of both the Internet as well as postal service (such as USPS, FedEx or UPS). The consolidated dataset at the destination site can later be used to perform computations using frameworks like Hadoop [12] or Dryad [19] locally.

The basic difference between this work and our proposal is in the fact that we do not attempt to consolidate the entire dataset to a single location. Each participating site hosts a part of Hadoop Distributed File System (HDFS) and the dataset created at the site is stored locally in HDFS thereby avoiding data transfer over the Internet as far as possible. During the execution of a MapReduce job, the computations moved to the participating sites hosting the datasets over the Internet.

Stork [18] is a scheduler for data placement activities developed for Grid environments executing data intensive scientific computations across wide-area networks. It attempts to solve the problem of efficient and reliable data placement over distributed Grid

environments by taking into consideration the semantics and characteristics of data placement tasks and implementing techniques to optimize them. The basic difference between this work and our proposal is in the fact that instead of moving the data towards computation through intelligent data placement techniques, we move the computation closer to the data sources and avoid the data transfer over wide-area networks as far as possible.

2.4.3 Delay scheduling

Zaharia et.al. [20] propose a Fair Share scheduler for Hadoop framework. The Fair Share scheduler attempts to resolve the conflict between fairness in scheduling and data locality with the help of a delay scheduling algorithm for the Map phase. The main idea here is that when the job that should be scheduled next according to fairness cannot launch a data-local task, it waits for a small time interval – letting other jobs to launch tasks itself. In this work, we have attempted to apply the concept of delay scheduling for the Reduce phase (i.e. an Adaptive Reduce Task Scheduling (ARTS) algorithm. Please refer to section 4.1 for details).

3. MapReduce Extensions

Today most of the modern enterprise applications and services on the Internet require rigorous end-to-end system quality and even small degradation in performance and reliability can have a considerable business impact. Since the Internet is designed as a best-effort network, it cannot provide guarantees about end-to-end performance or reliability. So the applications deployed in a centralized environment are often subjected to bottlenecks in the wide-area Internet communication resulting in poor performance.

To overcome this problem, modern enterprise applications are designed to be decentralized so that they can be deployed to the logical extremes of Internet instead of a centralized environment. The key benefit of this approach is the significant decrease in the data volume that needs to be moved, the subsequent network traffic, and the distance the data needs to travel which reduces transmission costs, latency and improves the quality-of-service (QoS).

A good example of such decentralized architecture is the email-hosting platform designed by a company named **Rackspace** (which provides managed hosting services for the enterprises). Its email hosting platform ‘Mailtrust’ currently host emails for more than one million users and thousands of companies using hundreds of email servers distributed across multiple geographies [8]. The decentralized nature of this platform helps to improve the performance and hence the user perception of the Email service provided.

During processing user requests, this platform generates considerable amount of logs (around 150GB per day) in various formats. The log-processing subsystem is required to aggregate this log information in a timely fashion, which can be used for different

purposes such as growth planning, understanding usage patterns as well as troubleshooting customer problems.

To facilitate this, Rackspace utilizes Hadoop to process the logs to build Lucene indexes that customer-support team can query. The actual mail server logs are stored using HDFS on a dedicated Hadoop cluster. The figure below represents the deployment scenario of this system.

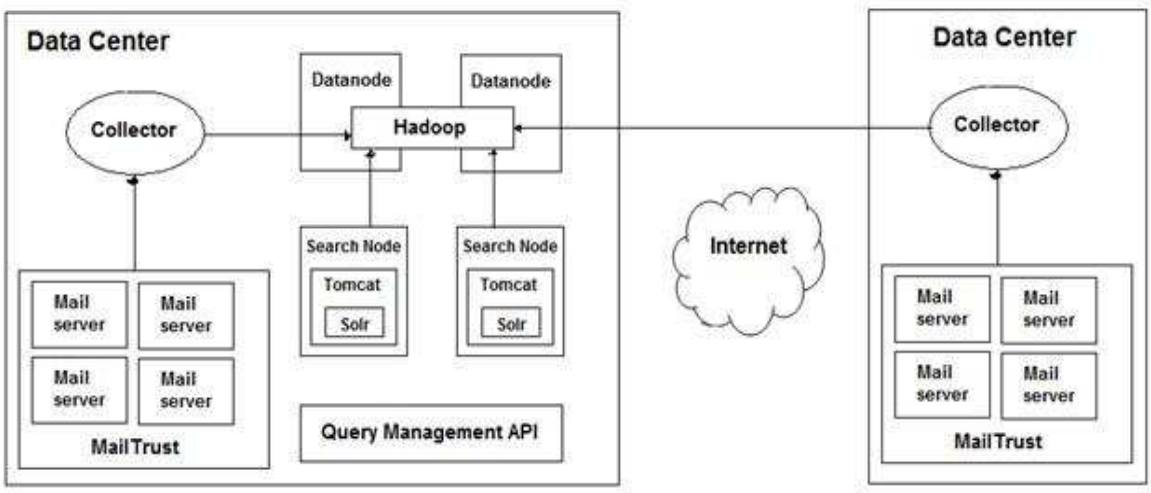

Figure 8: Architecture diagram for Log processing system at Rackspace

As shown in the figure 8, the email servers generating logs are distributed across multiple data centers. One of these datacenters also hosts a dedicated Hadoop cluster to perform the Log storage and processing. Within every datacenter, a variant of UNIX syslog is used to transfer the logs from email server to the Collector component. The Collector component in turn aggregates each type of log into a single stream and writes it to the dedicated Hadoop file system performing the log storage. Once the raw logs are placed in Hadoop file system, they can be processed by the MapReduce jobs. The result of this processing is the Lucene index, which is used to query various trends in the system

behavior such as the geographical sources of connections, average latency between specific machines, most effective spam rules etc.

There are many different MapReduce applications such as building inverted-index for a search engine, performing distributed Grep operation etc. which also may require operating on data distributed in multiple datacenters.

The feasibility of copying the entire dataset to a single datacenter depends upon a number of factors such that the total size of the data to be copied from each datacenter, total number of datacenters involved in the copy as well as capacity of inter datacenter network. But as the Internet evolves, the growth in the size of geographically scattered datasets would outpace the improvements in the network bandwidth available in the Internet. At that point the approach of copying the entire dataset to a single location using the Internet becomes infeasible.

In the past Jim Gray [15], DOT project [16] and the PostManet project [17] have suggested using postal service as an alternative option to large-scale data transfer over the Internet. But this alternative is not feasible in distributed environments involving a large number of data-sources frequently generating data to be processed.

Hence as part of this research, we have studied design assumptions behind different components of Hadoop eco-system which mandate the processing of Map/Reduce computations in a centralized environment. Specifically we focus on two aspects of Hadoop framework,

- Design of an underlying Hadoop Distributed File System (HDFS)
- Design of scheduling algorithm for the Reduce phase.

3.1 WAN aware Hadoop Distributed File System

The fundamental design assumption in a MapReduce framework is the availability of a single logical distributed file-system storing the input dataset. So in order to execute MapReduce computation across multiple datacenters, it is essential to have a distributed file system which can scale over the wide-area networks. As part of this research, we have experimented with Hadoop Distributed File System (HDFS).

In order to understand the implications of deploying a single HDFS cluster across multiple datacenters, we conducted an experiment to evaluate the performance of HDFS spanning multiple datacenters and observed that the write performance of HDFS degrades considerably for the large datasets. (Please refer to section 5.3 for details of this experiment.)

The cause of performance degradation in HDFS is that the default replica placement policy in HDFS does not take into consideration the datacenter-locality of the client while selecting the replicas. This results in unnecessary copy of data blocks over the Internet during the write operation.

To solve this problem, we have designed a WAN-aware replica placement policy which ensures that the replicas are placed in the same datacenter as the client, provided sufficient storage space is available in that datacenter. Using the proposed policy, HDFS clients in multiple datacenters would be able to share a unified namespace without the overhead of data transfer over the Internet. With the WAN aware replica placement policy, the performance of write-operation in HDFS improves from 100% up to 200%

depending upon the size of the dataset being written. (Please refer to section 5.3 for details of this experiment.)

We also observed that the network-topology awareness feature in HDFS ensures that during the read operation, the client accesses a datacenter-local copy of the block if available and hence the performance of the read operation does not suffer unnecessarily due to deployment of HDFS over the wide-area network. (In case when there is no datacenter-local copy of the block available, then the client would need to fetch it from one of the remote datacenters. But this cost is unavoidable and can be improved in future by using intelligent data-transfer protocols).

3.2 Intelligent Scheduling of Reduce phase

The execution time of a MapReduce job depends upon the time at which the Reduce phase starts. In Hadoop framework this can be configured by setting an appropriate value for ‘mapreduce.job.reduce.slowstart.completedmaps’ property (henceforth termed as slow-start threshold), which denotes a fraction of the total number of Map Tasks in the Job which should be complete before the Reduce phase can begin. After this threshold is reached, Hadoop selects any random node to execute the Reduce task.

Although this policy works well on dedicated Hadoop cluster in a centralized environment, it is very inefficient when Hadoop is deployed in either distributed environment involving multiple datacenters or in a shared environment in which multiple MapReduce jobs can execute concurrently in a single cluster. Existing literature [3, 4] does not offer any guidelines for intelligent Reduce phase scheduling in such environments.

Consider a distributed environment involving multiple datacenters connected through the Internet. As the MapReduce computation operates on scattered dataset in multiple datacenters, each datacenter hosts a fraction of total Map phase output. Now if the Reduce task is scheduled to a node in datacenter hosting only a small fraction of total Map phase output, it will require to copy substantial size of Map phase output from other datacenters over the Internet. This can degrade the response time of a MapReduce computation considerably. Hence the Reduce phase scheduling algorithm should take into consideration locality between the node executing Reduce task and the nodes storing Map phase output. This means that in a distributed environment, Reduce task should be scheduled to a node in datacenter hosting the maximum fraction of total data size for the corresponding partition.

On a dedicated cluster, Hadoop uses early-start (slow-start threshold = 0.05) policy by default. Early start of Reduce phase helps to overlap the compute intensive Map phase and I/O intensive Shuffle phase, resulting in improved response time for the Job. This configuration is especially suitable for clusters executing short and interactive jobs, where response time is very important. This is because short jobs suffer greater percentage degradation in the response time if the Reduce phase is delayed (Please refer to section 5.4 for details).

On the other hand, in case of a shared cluster concurrently executing both long running batch-processing jobs and short interactive jobs; starting the reduce phase early results in reducing the overall throughput of the system. This is because after the Reduce phase begins for a long running job, it occupies majority of Reduce slots in the cluster. As a result concurrent short jobs in the cluster are starved for the Reduce slots. This results in

degrading the response time for short jobs during the periods of high contention (Please refer to section 5.5.2 for details).

Although delaying the start of Reduce phase helps to improve the total throughput of the cluster; for an individual Job, it ends up having only a partial-overlap between the compute intensive Map phase and the I/O intensive Shuffle phase. Hence during the periods of low contention, response time suffers as the Reduce tasks have to wait longer for the Shuffle phase to complete (due to delayed start).

Since the total time required for the Shuffle phase is directly proportional to the size of the dataset generated during the Map phase, we experimented with different workloads using a realistic cluster configuration. During each experiment, we used early-start and delayed-start Reduce phase scheduling policies in order to understand its effect on the job execution time. During these experiments, we considered following two important classes of MapReduce workloads.

3.2.1 TeraSort

TeraSort is a write-intensive MapReduce benchmark, which sorts the input dataset using a MapReduce paradigm (Please refer section 5.1.3 for details). The size of the output generated at the end of Map phase is equivalent to the size of the input dataset (there is no reduction for this benchmark). Hence this benchmark characterizes all the workloads generating substantial amount of Map output as compared to original input dataset.

For this experiment, we used a large input dataset (around 4GB) and executed this benchmark on a 3-node Hadoop cluster for Reduce phase scheduling policies such as early-start (slow-start threshold = 0.05) and delayed-start (slow-start threshold =0.8).

From figures 9 and 10, we can see that a delayed-start policy for Reduce phase results in reducing the overlap between the Map and Shuffle phases, thereby increasing the total job execution time (around 586s). On the other hand early-start of Reduce phase results in substantial overlap between the Map and Shuffle phases, thereby reducing the total job execution time (around 564s). Hence considering response time, this class of workload is sensitive to the slow-start threshold value.

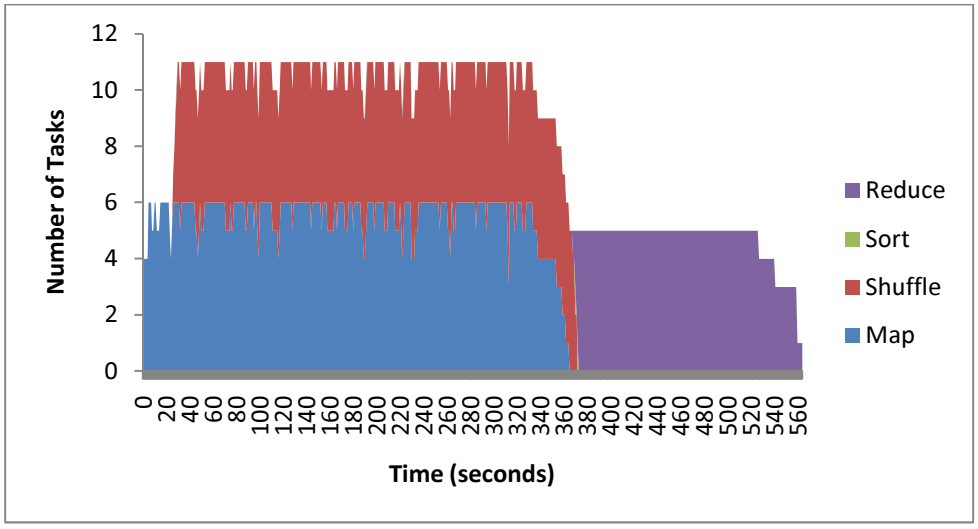

Figure 9: Task timeline for TeraSort benchmark using early-start of Reduce phase

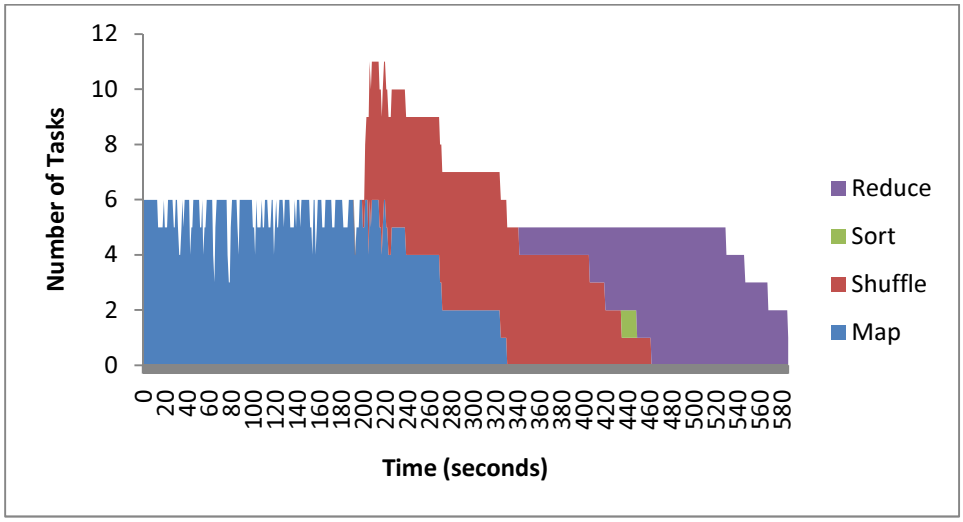

Figure 10: Task timeline for TeraSort benchmark using delayed-start of Reduce phase

3.2.2 WordCount

WordCount is a MapReduce benchmark which counts the total number of occurrences of all the distinct words available in the given input dataset (please refer to section 5.1.4 for details). The size of the Map phase output is substantially smaller as compared to the original dataset (of reasonable size). Hence this benchmark represents workloads performing aggregation over the input dataset.

For this experiment, we used a large input dataset (around 5GB of eBooks in text format) and executed this benchmark on a 3-node Hadoop cluster for Reduce phase scheduling policies such as early-start (slow-start threshold = 0.05) and delayed-start (slow-start threshold =0.8).

From figures 11 and 12, we can clearly see that increasing the delayed-start of Reduce phase does not result in increase in the job execution time as compared to early-start policy. Hence considering response time, this class of workload is not sensitive to the slow-start threshold value. Therefore we can state that the decision to start the Reduce phase should also depend upon the type of workload for a given Job as well as total size of input dataset (which would classify a Job as a long running or short/interactive). Considering only the total progress of Map phase (in terms of fraction of total Map tasks complete) is not sufficient to provide acceptable performance on a shared Hadoop cluster.

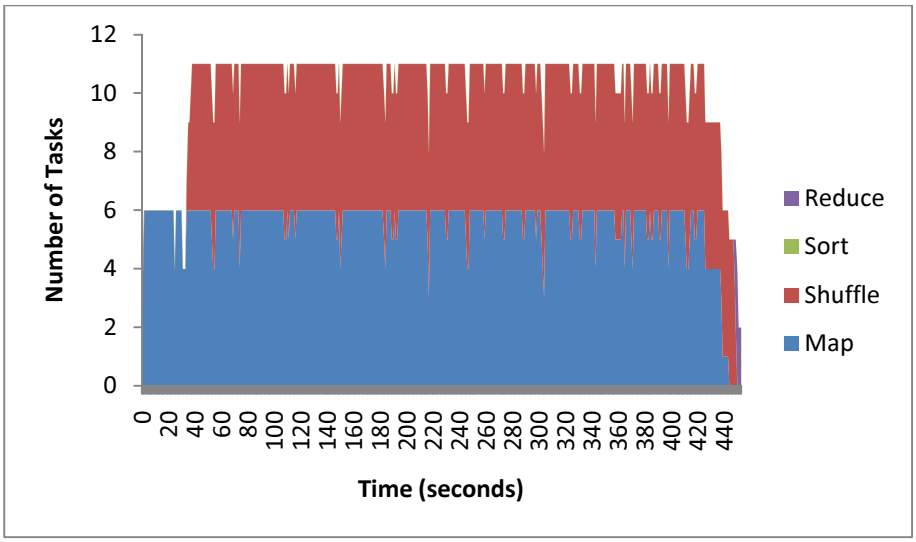

Figure 11: Task timeline for WordCount benchmark using early start of Reduce phase

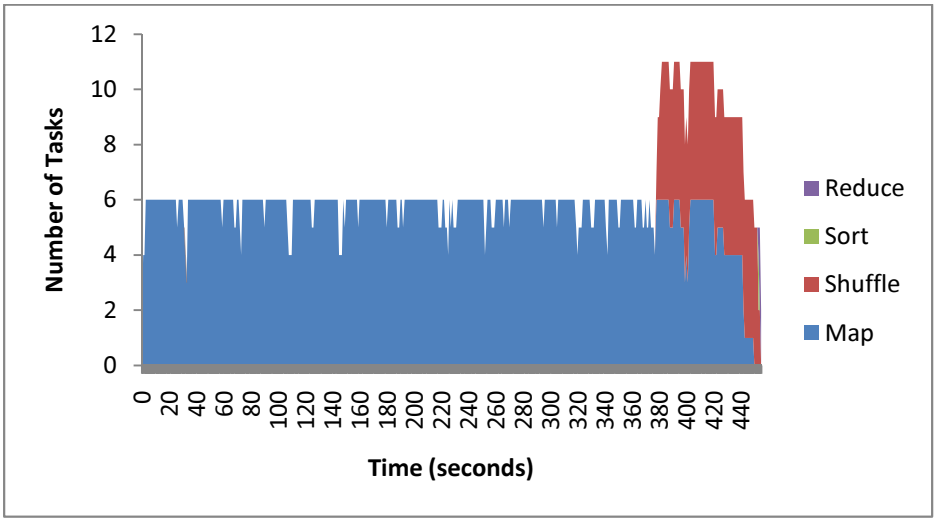

Figure 12: Task timeline for WordCount benchmark using delayed start of Reduce phase

4. System Design

4.1 Adaptive Reduce Task Scheduling (ARTS) algorithm

We have designed a new algorithm for scheduling Reduce phase in Hadoop which works well both in distributed environments involving multiple datacenters as well shared environments. This algorithm is based upon two important observations,

- A Shuffle phase cannot complete unless the entire Map phase is complete. Hence for any given partition, we can delay the execution of corresponding Reduce task if the time to copy the Map output for this partition is less than the estimated finish time for the Map phase. Also among multiple eligible partitions, the partition containing largest Map output needs to be scheduled earliest.

- At any point during the execution of a Map phase, the size of partial Map output can be used to infer the type of the workload being executed. This information can be used to decide if a Reduce task needs to be scheduled immediately or can be delayed.

This algorithm is orthogonal to the deployment environment (such as centralized or distributed) as well as scheduling policy configured for the Hadoop cluster (e.g. a Capacity Scheduler or a Fair Share scheduler etc.).

Figure 13 describes the proposed algorithm. During execution of MapReduce job, the Hadoop master (called JobTracker) keeps a track of total size of Map phase output segregated with respect to datacenters. This means that for every partition, it keeps a track of total size of Map output in each datacenter. This helps to identify the datacenter-

locality while scheduling Reduce phase in distributed environment. It is also used to classify the workload which helps in deciding the time when the Reduce phase could start.

We have implemented a mechanism to identify datacenter identity for any node based on its IP address or domain-name (e.g. all machines with domain name .edu are part of CAC Lab). This configuration is static in nature and needs to be performed manually during the initialization of Hadoop cluster. This algorithm is executed when the master (called JobTracker) identifies any worker (called TaskTracker) in the cluster with an available Reduce slot. Let's assume that the datacenter identifier for this worker is D_x. The important steps in this algorithm are as follows,

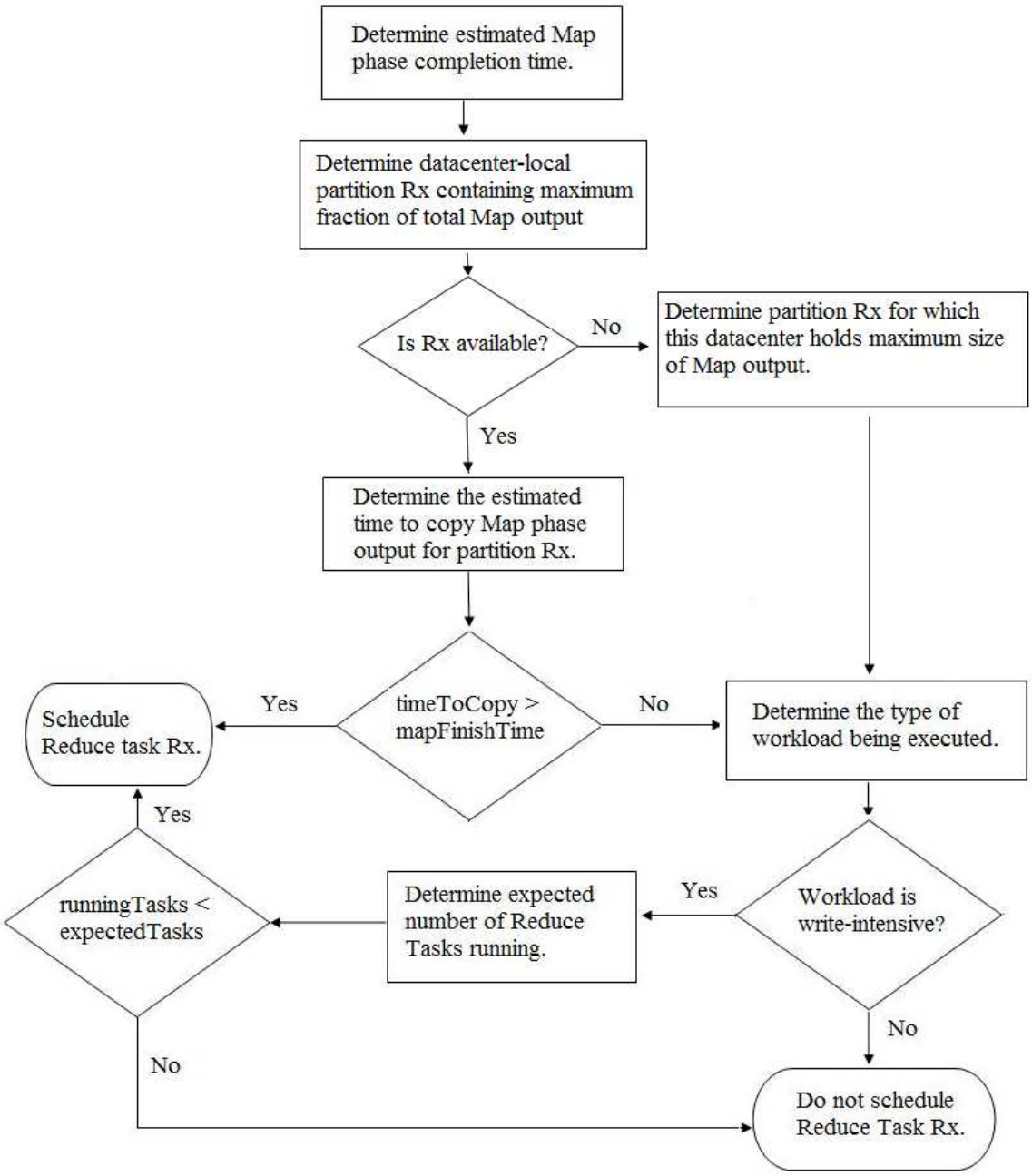

Figure 13: Adaptive Reduce Task Scheduling (ARTS) algorithm

- JobTracker calculates estimated time required to complete the Map phase using the information such that total number of Map tasks remained to be processing and average Map task completion time (please refer section 4.1.2 for details).

- Among all the unscheduled partitions, JobTracker attempts to find out a partition R_x for which datacenter D_x holds maximum fraction of total Map output compared to other datacenters.
- If no such partition could be found, then it selects any partition R_x such that datacenter D_x has maximum size of Map output as compared to all other unscheduled partitions. Please note that in this case partition R_x may not hold maximum fraction of total Map output as compared to other datacenters. This is just an optimization to reduce the cost of data copy from other datacenters.
- JobTracker calculates estimated time required to copy the entire Map output for selected partition R_x.
- If the estimated time to copy is larger than the Map phase completion time, then it schedules the partition R_x for execution on the identified worker.
- If the estimated time to copy is smaller than the Map phase completion time, then JobTracker classify the workload for job by using the total size of Map output for partition R_x and the total Map phase input processed (please refer section 4.1.3 for details).
- If the workload is not write-intensive, then it does not schedule the partition R_x for execution.
- If the workload is write-intensive, then it calculates estimated number of Reduce tasks executing at the given point of time using a liner model (explained in section 4.1.1). If the total number of Reduce tasks executing are lagging behind the expected number then it schedules the partition R_x for execution on the identified worker.

4.1.1 ARTS Algorithm for different types of workloads

The figure 14 depicts the task timeline for the Terasort benchmark using the ARTS algorithm. Here we observe that although the scheduling of the Reduce tasks start early (after completion of 5% Map tasks), not all the Reduce tasks are started immediately (like in case of early-start policy). Instead, the algorithm figures out the next best partition depending upon the size of Map phase output generated so far at each point of time.

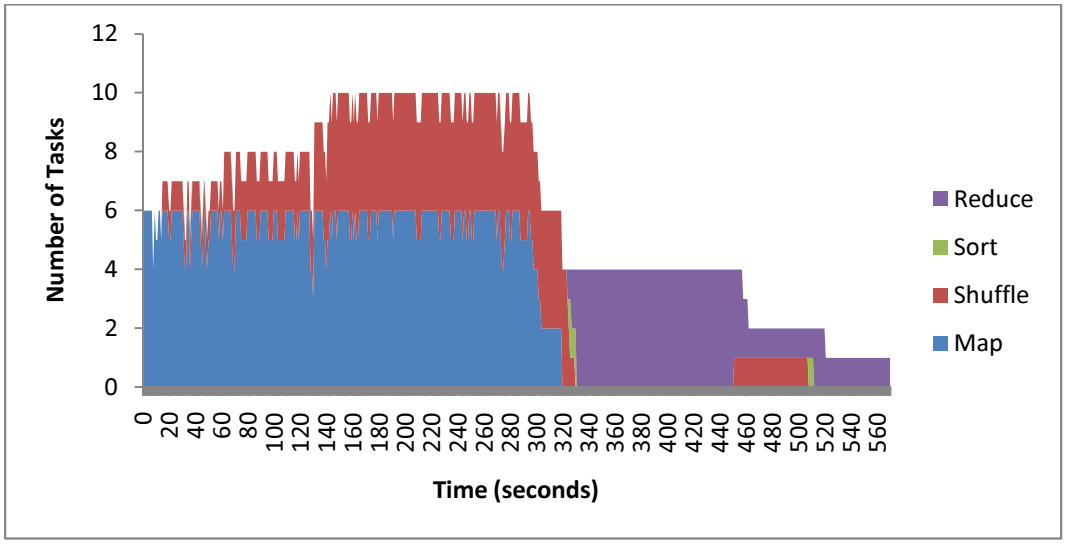

Figure 14: Task timeline for TeraSort benchmark using ARTS algorithm

Here we would like to note that the calculation of time-to-copy Map output depends upon the network bandwidth estimation provided by the user. In case this estimation is overly optimistic, it can result into delaying the start of the Reduce phase thereby degrading the performance of the Job. In order to avoid such scenarios, we also implement a linear step function which calculates expected number of Reduce tasks to be executing at any point of time (please refer section 4.1.4 for details). The algorithm uses this function to ensure that scheduling of the Reduce tasks is not lagging too far behind this expected number.

This helps to avoid the inefficient scheduling decisions made due to optimistic network bandwidth estimations.

In our experience such step function is required only in case of workloads similar to Terasort since they tend to generate large amount of intermediate Map output and hence are more vulnerable to incorrect network bandwidth estimates. In the future work, we are planning to automatically infer the network bandwidth available in the given Hadoop cluster.

The figure 15 depicts the task timeline for the Wordcount benchmark using the ARTS algorithm. Here we can see that the scheduling of the Reduce tasks start late (after completion of around 90% Map tasks) like in case of delayed-start policy while providing execution time comparable with the best case of early-start policy.

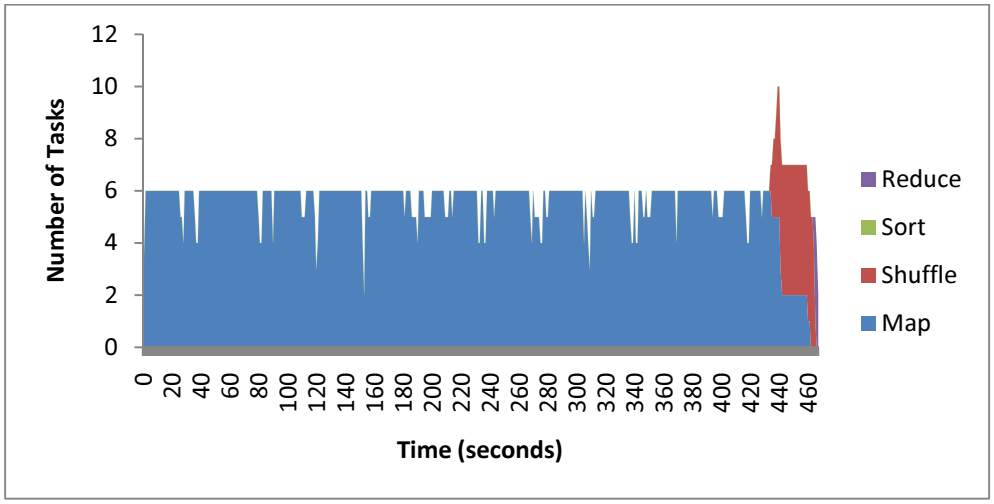

Figure 15: Task timeline for WordCount benchmark using ARTS algorithm

We have implemented the proposed ARTS algorithm in Hadoop framework. The figure 16 represents system architecture diagram for our implementation.

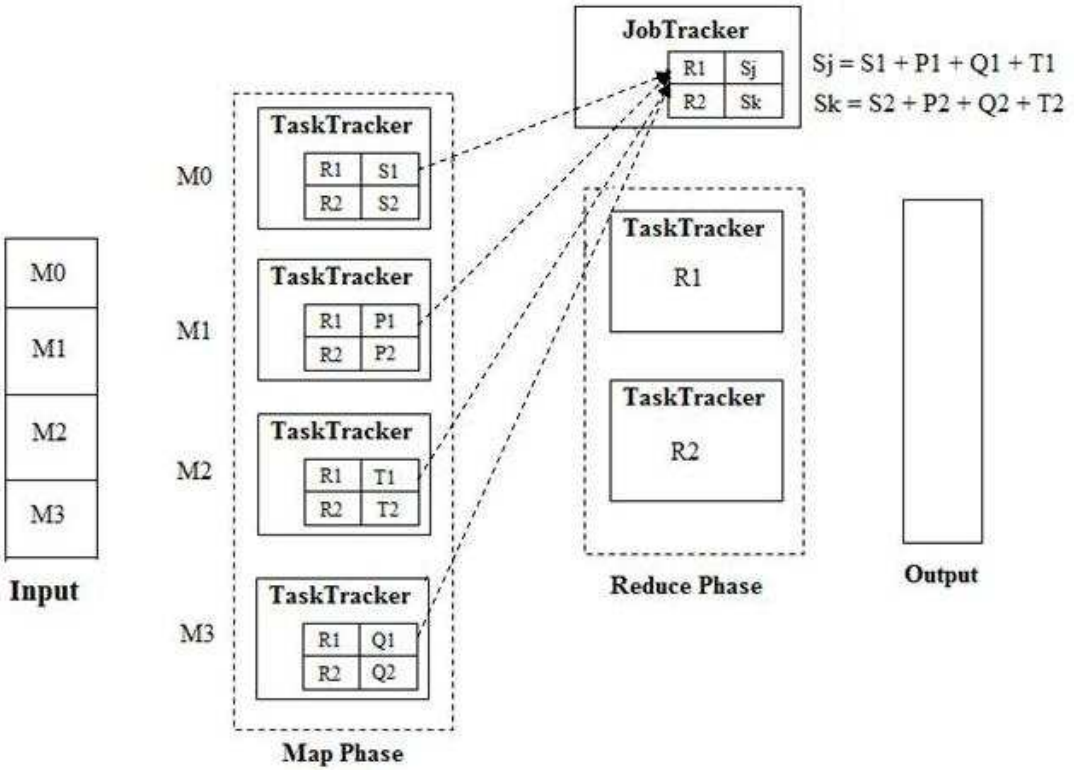

Figure 16: The system architecture for adaptive slow-start algorithm

As shown in figure 16, upon completion of a Map task, each worker in Hadoop cluster (called TaskTracker) notifies the master (called JobTracker) with the sizes of Map outputs for each of the partition. For each partition, the master performs aggregation (segregated with respect to datacenters). This helps master to keep a track of aggregate size of available Map output for each partition in each datacenter.

The master also has access to following information,

- Total number of Map tasks for this Job (totalTasks)
- Total number of Reduce tasks for this Job (totalReduceTasks)

- Total number of completed Map tasks (totalTasksCompleted)
- Total time spent during the Map phase (totalTimeInMapPhase)
- Total number of Map tasks being executed currently (runningTasks)
- Total number of Map slots currently allocated for this Job (mapSlotsAllocated)
- Estimated network bandwidth available inside each datacenter as well as across datacenters.
- For each partition, the total size of available Map output. (mapOutput[x] where x denotes the partition number).

Following subsections describe procedures used in the ARTS algorithm. Please refer to section 4.1 for details of the algorithm.

4.1.2 Calculation of estimated Map phase completion time

Following steps are used to calculate estimated Map phase completion time.

Step1: Calculate number of remaining Map tasks.

$$remainingTasks = totalTasks - totalTasksCompleted - runningTasks$$

Step2: Calculate average completion time for a Map task.

$$avgMapTime = totalTimeInMapPhase / totalTasksCompleted$$

Step3: Calculate estimated Map phase finish time.

$$mapFinishTime = \left(\lceil (remainingTasks/mapSlotsAllocated) \rceil \right) * avgMapTime$$

4.1.3 Determination of workload type of the job

Following steps are used to determine type of workload is executed.

```
totalMapOutput = 0;

For each partition Rx, totalMapOutput += mapOutput[x]
totalMapInputProcessed = totalTasksCompleted * hdfsBlockSize
outputFraction = totalMapOutput / totalMapInputProcessed;

If (outputFraction < Threshold)
   WorkLoad is of type Aggregating
Else
 Workload is of type Write-Intensive
```

4.1.4 Determination of expected number of Reduce tasks running

Following steps are used to determine the expected number of Reduce tasks in the Running state.

Step1: Calculate the step to be used during this calculation. It is calculated by using the values of total number of Map tasks (after the Reduce phase starts) and the total number of Reduce tasks.

$$step = \lceil ((numTasks - completedTasksForSlowStart)/numReduceTasks) \rceil$$

Step2: Calculate the expected number of Reduce tasks by using the values of total Map tasks completed after the slow-start and the value of the step calculated above.

$$value = totalTasksCompleted - completedTasksForSlowStart$$
$$expectedReduceTasks = \min (\lceil (value/step) \rceil , numReduceTasks)$$

5. Experimental Evaluation

5.1 Benchmarks

This section summarizes different benchmarks in Hadoop framework used for evaluation as part of this research.

5.1.1 TestDFSIO

TestDFSIO tests the I/O performance of HDFS by using a MapReduce job as a convenient way to read or write files in parallel. Each file is read or written in a separate map task, and the output of the map is used for collecting statistics relating to the file just processed. The statistics are accumulated in the reduce, to produce a summary.

5.1.2 TeraGen

TeraGen is a MapReduce application designed to generate official input dataset for the TeraSort benchmark. The user specifies the number of rows and the output directory in HDFS and this application executes a MapReduce job to generate the dataset. It divides the desired number of rows by the desired number of Map tasks and assigns ranges of rows to each Map task. The Map tasks generate and write the dataset in parallel in HDFS. The format of the data is as follows,

(10 bytes key) (10 bytes rowid) (78 bytes filler) \r \n

The keys are random characters from the set ' ' .. '~'.

The rowid is the right justified row id as an int.

The filler consists of 7 runs of 10 characters from 'A' to 'Z'.

Since this benchmark is write intensive, it can be used for stress testing HDFS write performance apart from generating input dataset for TeraSort benchmark.

5.1.3 TeraSort

This is a MapReduce application to sort the data generated by the TeraGen benchmark. The input is a set of hundred-byte long records stored in a HDFS in the format described in section 5.1.2. TeraSort is a standard MapReduce sort, except for a custom partitioner that uses a sorted list of N − 1 sampled keys that define the key range for each reduce. In particular, all keys such that sample [i − 1] <= key < sample[i] are sent to reduce i. This guarantees that the output of reduce i are all less than the output of reduce i+1. To speed up the partitioning, the partitioner builds a two level Trie data structure that quickly indexes into the list of sample keys based on the first two bytes of the key. TeraSort generates the sample keys by sampling the input before the job is submitted and writing the list of keys into HDFS.

Since this benchmark is read and write intensive, it is used for stress testing entire Hadoop eco-system including both HDFS and MapReduce.

5.1.4 WordCount

WordCount benchmark accepts a set of text files as an input and counts how often words occur. The output of the benchmark is a set of text files, each line of which contains a word and the count of how often it occurred, separated by a tab. Each Map task accepts a line as an input and breaks it into words. It then emits a key/value pair of the word and 1. Each Reduce task sums the counts for each word and emits a single key/value with the word and sum. As an optimization, the Reduce task implementation is also used as a

combiner on the map outputs. This reduces the amount of data sent across the network by combining each word into a single record.

5.2 Evaluation of baseline I/O performance of Hadoop file-system (HDFS) in different clusters

Since the configuration of hardware components is generally not uniform in multiple datacenters, we want to evaluate baseline I/O performance of Hadoop File System in different clusters.

In this experiment we setup a single node Hadoop cluster (psudo distributed mode) on CAC cluster at Rutgers University as well as on Amazon EC2 (using small instance type). The hardware configurations for both the nodes is given as follows.

Amazon EC2

1.7 GB memory

1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit)

160 GB instance storage

32-bit platform

CAC Lab (glitch.rutgers.edu)

Intel(R) Xeon(R) CPU 2.40GHz (4 processors)

3.6 GB memory

1TB storage

64-bit platform

For this experiment, we utilizied TestDFSIO benchmark provided by the Hadoop distribution (please refer section 5.1.1for details). In this case, we conducted following experiments.

- Executed TestDFSIO benchmark locally on CAC cluster (i.e. using both JobScheduler as well as HDFS in the CAC cluster).
- Executed TestDFSIO benchmark locally on EC2 cluster (i.e. using both JobScheduler as well as HDFS in the EC2 cluster).

Figures 17 and 18 show the performance of HDFS in above three combinations for read and write operations for different data sizes (250MB, 500MB and 750MB respectively). From this experiment, we can conclude that the CAC cluster provide much better I/O performance for HDFS operations as compared to the EC2 environment.

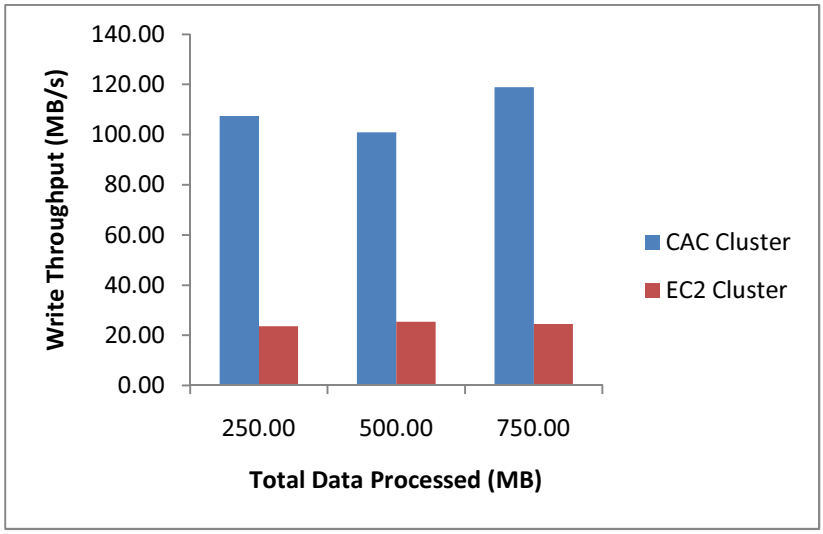

Figure 17: HDFS write performance for different sizes of datasets

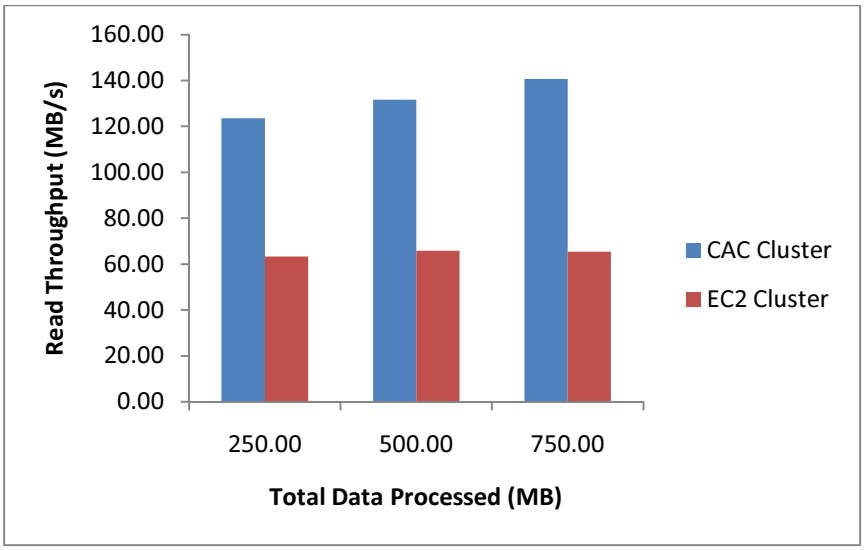

Figure 18: HDFS read performance for different sizes of datasets

5.3 Performance evaluation of HDFS write operation using different replica placement policies

The objective of this experiment is to evaluate the performance of write operation in HDFS using different replica selection policies. In this experiment we have experimented with two repica placement policies – (a) the default policy provided by the Hadoop framework. This policy is suitable only for the HDFS deployments in a single datacenter. (b) WAN aware replica placment policy designed as part of this research. We have used Hadoop TeraGen benchmark (described in section 5.1.2)

In this experiment we setup a six node Hadoop cluster using CAC cluster at Rutgers University as well as on Amazon EC2. The hardware configurations for both the nodes is given as follows.

Amazon EC2 (small instance)

1.7 GB memory

1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit)

160 GB instance storage

32-bit platform

CAC Lab (glitch.rutgers.edu/spring.rutgers.edu/darkwing.rutgers.edu)

Intel(R) Xeon(R) CPU 2.40GHz (4 processors)

3.6 GB memory

1TB storage

64-bit platform

We experimented with following three cluster configurations.

- Executed benchmark locally on EC2 cluster. For this experiment, we setup a six node Hadoop cluster on Amazon EC2. This experiment would help us to understand the baseline performance of TeraGen benchmark in a single datacenter environment.
- Executed benchmark using both nodes in both CAC and EC2 cluster using the Hadoop's default replica placement policy (termed as Hadoop Native henceforth). For this experiment, we divided the 6 node cluster equally into two parts such that each cluster would host a part of it (having 3 nodes each). This experiement would help us to understand the performance degradation due to the Hadoop's standard replica placement policy in a distributed environment.
- Executed benchmark using both CAC and EC2 cluster but with the newly designed WAN aware replica placement policy. This experiment would help us to understand the performance improvement gained with this policy.

The graph in figure 19 represents time required for completing the write operation using different replica placement policies. We have normalized the value of time required for the write operation with respect to the corresponding value in a single datacenter environment. This is calculated using following formula.

$T_{normalized} = ((T_{actual} - T_{base}) * 100) / T_{base}$

Where,

T_{actual} represents the value of time taken to complete the write operation in a distributed environment for a specific size of dataset.

T_{base} represents the value of time taken to complete the write operation in a centralized environment for the same size of dataset as above.

$T_{normalized}$ is the normalized value of time taken to complete the write operation in a distributed environment.

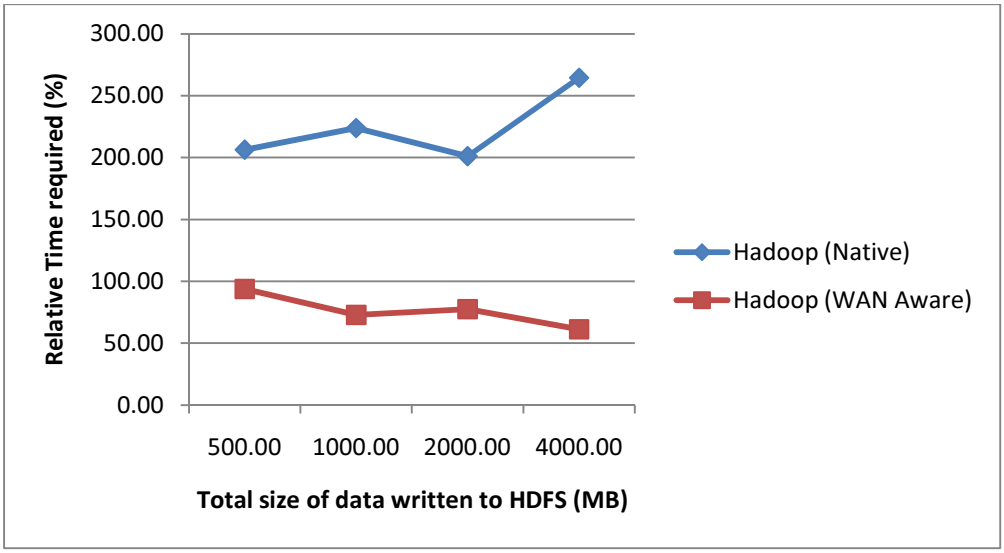

Figure 19: Relative performance of write-operations using different replica placement policies.

From figure 19, we can clearly see that in a distributed environment the default replica placement policy in Hadoop suffers from severe degradation in performance of write operation with respect to its performance in a centralized environment for any specific size of dataset (greater than 200%). The WAN aware replica placement policy on the other hand performs significantly better than the default policy (improvement in the range of 100% to 200%).

Another interesting point to note here is that for both policies time required to complete write operation fluctuates (e.g. it required less time to write 2GB data than 1GB using default policy). The reason behind this behavior is the allocation of Map tasks to different datacenters. The hardware configurations of nodes in multiple datacenters are not homogeneous (please refer to section 5.2 for details). Hence if a large number of Map tasks are allocated to a datacenter having better I/O performance then the time required to complete the operation is reduced considerably.

5.4 Evaluation of response time for a job for different Reduce phase scheduling algorithms

The goal of this experiment is to understand how different Reduce phase scheduling policies ultimately affect the response time of a job. In this experiment we use TeraSort benchmark (please refer section 5.1.3 for details) since it is sensitive to the Reduce phase start policy.

We conducted multiple iterations of this experiment by increasing the size of input dataset from 256MB up to 4GB. This helped us to infer the type of jobs (interactive or batch) which are more sensitive to a change in Reduce phase scheduling policy. For each size of input dataset, we measure response time for a job using both early-start and

delayed-start Reduce phase scheduling policies. We calculate percentage degradation in response time as follows,

% Degradation in response time = $[(T_{delayed\text{-}start} - T_{early\text{-}start}) / T_{delayed\text{-}start}] * 100$

Where,

$T_{delayed\text{-}start}$ represents response time using a delayed-start Reduce phase scheduling algorithm (slow-start threshold = 0.8)

$T_{early\text{-}start}$ represents response time using early-start Reduce phase scheduling algorithm (slow-start threshold = 0.05)

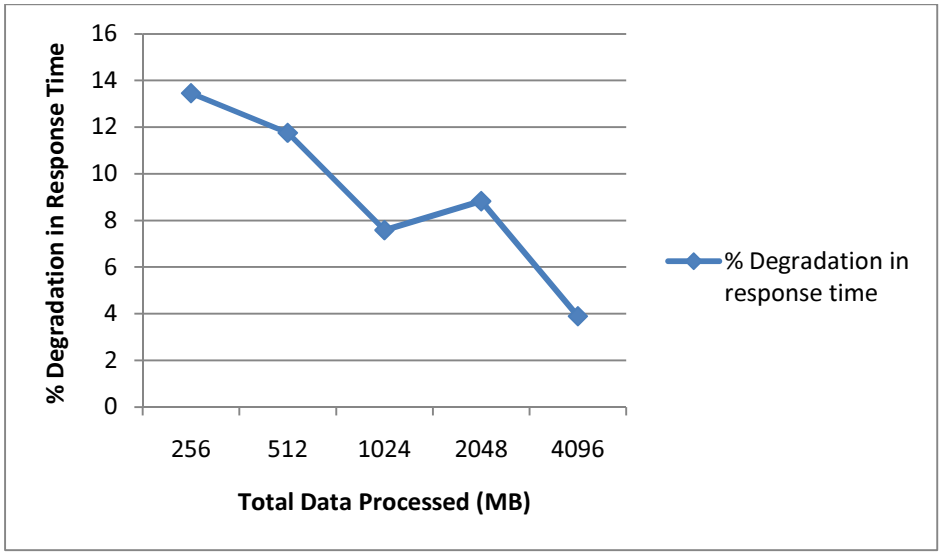

Figure 20: % degradation in response time after delaying the Reduce phase scheduling

From the figure 20, we can clearly see that the shorter jobs suffer larger percentage degradation in the response time. As the size of dataset increases, the percentage degradation in the response time becomes less dominant.

5.5 Performance evaluation of adaptive slow-start algorithm using different Hadoop schedulers

The adaptive slow-start algorithm attempts to schedule Reduce tasks such that an appropriate overlap between compute intensive Map phase and I/O intensive Shuffle phase can be achieved without penalizing short/interactive jobs (executing on a shared Hadoop cluster) on response time. The algorithm in itself is orthogonal to the design (and goals) of any specific scheduling algorithm (such as fair share scheduling or first-in-first-out scheduling) and hence can become a core component of the MapReduce framework.

We conducted a set of experiments to evaluate the feasibility of adaptive slow-start algorithm in different execution environments by experimenting with different Hadoop schedulers namely First-In-First-Out (FIFO) scheduler and the Fair Share scheduler. This will help us to determine the feasibilty of using the proposed algorithm as a core component of Hadoop MapReduce framework.

5.5.1 Performance evaluation of Adaptive Reduce Task Scheduling (ARTS) algorithm using First-In-First-Out (FIFO) Hadoop Scheduler

Hadoop FIFO Scheduler performs sequential scheduling of jobs such that atmost one job can execute in every phase. This means that there can be no two concurrent jobs competing for the Reduce slots. Hence in this scenario, an early start of Reduce phase (by configuring a low slow-start threshold) helps to improve the overlap between the compute and I/O intensive phases of a MapReduce job and as a result its response time.

The goal of this experiment is to evaluate the performance of adaptive slow-start algorithm using the First-In-First-Out (FIFO) Hadoop scheduler.

In this experiment we setup a three node Hadoop cluster on CAC cluster at Rutgers University. The hardware configurations for both the nodes is given as follows.

Intel(R) Xeon(R) CPU 2.40GHz (4 processors)

3.6 GB memory

1TB storage

64-bit platform

For this experiment, we used TeraSort benchmark (please refer section 5.1.3 for details) since it is sensitive to the changes in the slow start threshold. To verify that this algorithm is not sensitive to the size of the input dataset, we used datasets of different sizes starting from 256MB up to 4GB.

For every dataset, we measured response time for early-start scheduling (by setting a low value for slow-start threshold such as 0.05) as well as using the proposed adaptive slow-start algorithm. The results of this experiment are shown in figure 21.

From the graph in figure 21, we can see that the ARTS algorithm provides comparable results with respect to the early-start scheduling. Also the proposed algorithm is not sensitive to the size of the dataset (and hence the nature of the job for example a short-interactive job OR a long running batch-job) being executed.

Hence we can conclude that it is feasible to use the proposed ARTS algorithm in case of a FIFO Hadoop scheduler.

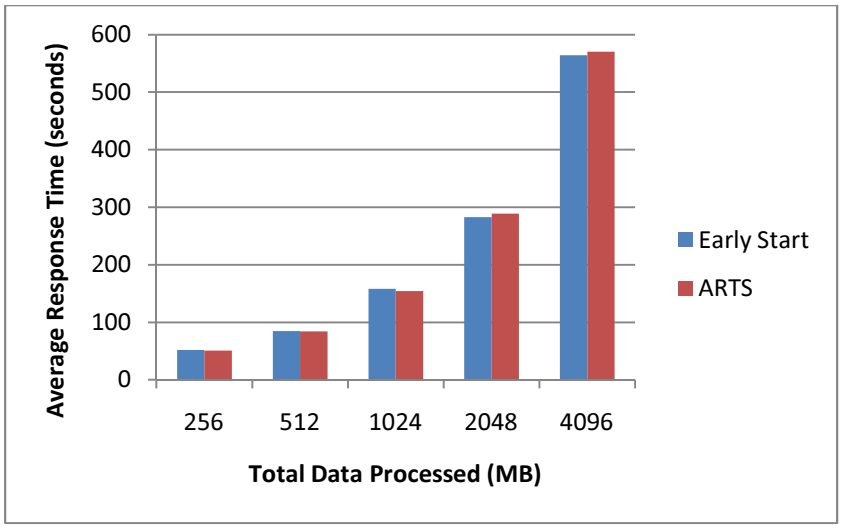

Figure 21: Average response time for TeraSort benchmark for different Reduce phase scheduling alternatives

5.5.2 Performance evaluation of Adaptive Reduce Task Scheduling (ARTS) algorithm using Hadoop Fair Share Scheduler

The Hadoop Fair Share scheduler (please refer to section 2.3.1.1) can be configured with multiple queues (typically one per user or per UNIX group) so that users can submit and execute multiple jobs concurrently. Each pool can be configured to have guaranteed capacity in terms of number of Map and Reduce slots in the cluster. When there are pending jobs in the specified pool, it gets at least the configured capacity; otherwise those slots are assigned fairly to other pools having active jobs. Fair sharing ensures that over time, each job receives roughly the same amount of resources. This helps shorter jobs to finish quickly without starving the longer jobs.

As explained in section 0, the scheduling decisions made for the Reduce phase can impact the response time (and hence the throughput) for individual jobs. This impact is

substantial for short (interactive) jobs. The goal of this experiment is evaluate the performance of adaptive slow-start algorithm using Hadoop Fair Share scheduler.

For this experiment, we used a 20 node Hadoop cluster on Amazon EC2 (using large instances). The cluster is configured such that each node has 2 Map slots and 2 Reduce slots. Hence the cluster has total number of 40 Map slots and 40 Reduce slots. The configuration each node is as follows,

7.5 GB memory

4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each)

850 GB instance storage

64-bit platform

In this experiment, we wanted to evaluate the performance of the proposed ARTS algorithm for different combinations of workloads executing concurrently on a Hadoop cluster configured with a Fair Share scheduler. Here our intention is to select one workload executing a large (batch-processing) job and one or more workloads executing short (interactive) jobs. Since TeraSort benchmark (explained in section 5.1.3) is sensitive to the slow-start threshold, we use it for short interactive jobs. For long batch-processing job, we alternate between WordCount (an aggregation workload – explained in section 5.1.4) and TeraSort.

Following table shows the configurations of different workloads used for this experiment.

TeraSort (Long batch-processing) job	30GB input dataset	434 Map Tasks 37 Reduce Tasks
TeraSort (Short interactive) job	512MB input dataset	11 Map Tasks 5 Reduce Tasks
WordCount (Long batch-processing) job	6GB input dataset	396 Map Tasks 37 Reduce Tasks

For this experiment, we configure a queue (named ‘production’) to execute batch jobs. This queue is configured to have minimum 20 Map slots and 29 Reduce slots. (These numbers are selected at random). The remaining capacity of the cluster is shared fairly among the other queues in the system.

For this experiment, the Hadoop cluster is configured to have five queues – one for long running batch jobs (named production) and other four for short interactive jobs. Users submit a job in the one of the configured queues and wait for it to complete before submitting the next job.

During each iteration of the experiment, we ensure that at least one batch job is always in progress by executing X number of batch jobs (X is a configurable parameter set to fairly high value).

For each experiment, we start with a single user executing N number of short jobs (N is a configurable parameter) and measure average response time for the short jobs. At this point, we also calculate average response time for batch jobs by considering all the batch jobs which were submitted during the time span of the iteration. Then we gradually

increase the number of users executing short jobs (by using a step of 1 and going up to 4). Again for each case, we calculate the average response time for the short jobs (irrespective of the associated queues) and the average response time for batch jobs. The graphs represented for this experiment are normalized with respect to the response time of the job in the best case (i.e. having an early start of the reduce phase in a dedicated cluster).

5.5.2.1 A combination of aggregating and write-intensive workloads

As per the description above, we conducted an experiment with an aggregating workload (A large WordCount job) and one or more short jobs executing write-intensive workload (A short TeraSort job).

The figure 22 depicts the average response time for a short job (relative to corresponding value on a dedicated cluster configured with early-start policy) against the total number of concurrent jobs in the system. Here the case of a single job represent the average response time for a job on a dedicated Hadoop cluster configured with early-start policy which is used to normalize all other values in the graph.

We can see that on a dedicated cluster (without any contention), early start of Reduce phase helps to improve the response time. In such scenario, the delayed start of Reduce phase results into degradation of average response time (approximately 10%).

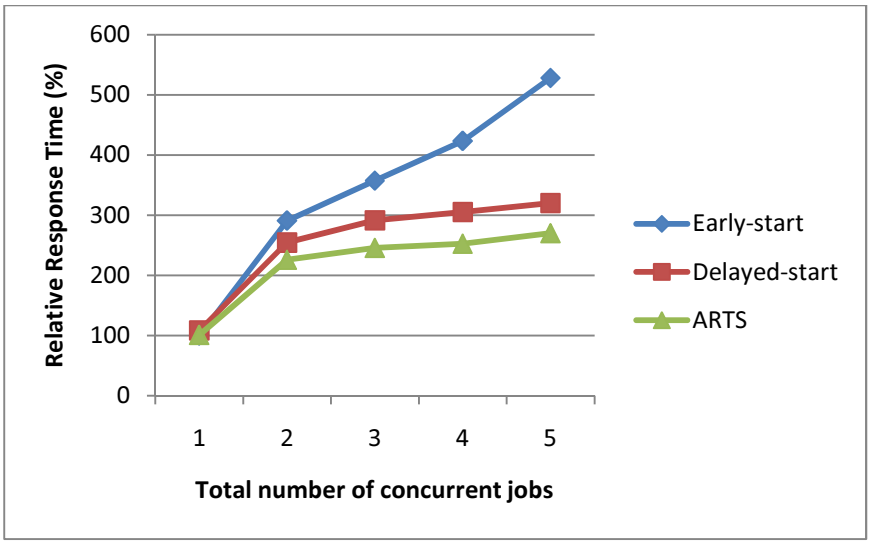

Figure 22: Average relative response time for short (interactive) jobs for different Reduce phase scheduling alternatives

But as the contention in the cluster grows, the early start of Reduce phase helps batch job to occupy the Reduce slots for longer duration and as a result the short jobs are starved for Reduce slots in the cluster. Hence in such scenario the average response time for short jobs degrades substantially (as shown in figure 21).

Delaying the start of Reduce phase helps to avoid the starvation of short jobs for the Reduce phase and as a result average response time improves with respect to early-start at the same level of contention. As shown in figure 22, the delayed start of Reduce phase improves the average response time for short jobs in the range of 4% to 20% (approximately) in a contented Hadoop cluster.

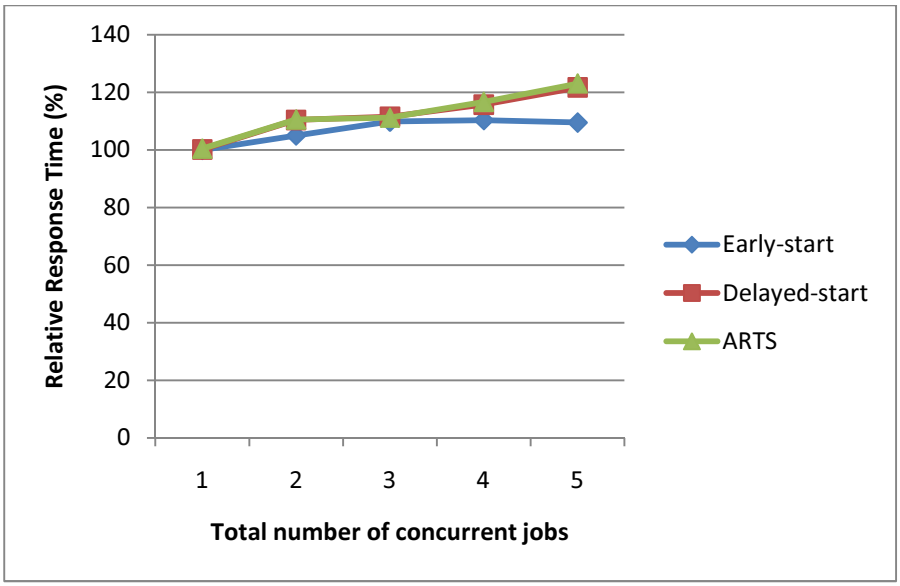

Figure 23: Average relative response time for long (batch) jobs for different Reduce phase scheduling alternatives

The figure 23 depicts the average response time for a batch job (relative to corresponding value on a dedicated cluster configured with early-start policy) against the total number of concurrent jobs in the system. Here the case of a single job represent the average response time for a batch job on a dedicated Hadoop cluster configured with early-start policy which is used to normalize all other values in the graph.

From figure 23, we can see that early-start of Reduce phase is especially helpful for batch jobs as their average response time doesn't degrade considerably (as compared to short jobs). On careful analysis we observed that while the short jobs are starved for Reduce slots in the cluster (after completing the Map phase), batch job gets access to more number of Map slots (since there is no contention in the cluster for the Map slots) resulting into an early completion of Map phase (compared to a contented cluster with multiple jobs active in Map phase). This behavior clearly violates the basic design goal of

the Hadoop scheduler (in this case Fair Share scheduler) to provide a fair share of Hadoop cluster to jobs submitted to different queues in the system.

As we delay the start of Reduce phase, the contention for Reduce slots decreases which results in an increase in the contention for the Map slots. This affects the time required to complete the Map phase and hence the response time for a batch job. From figure 23 we can see that as the contention in the system increases, the average response time for a batch job degrades in the range of 5% to 10% as the contention in the cluster increases.

The proposed ARTS algorithm attempts to find an optimum balance between these two alternatives (early-start vs. delayed-start) by taking into consideration factors such as class of workload as well as the size of job.

In case of a short job, the proposed algorithm recognizes the fact that it is a write-intensive workload and hence attempts to schedule the Reduce phase early. Hence when there is no contention in the cluster, the proposed algorithm provides comparable results with respect to early-start (thereby improving the overlap between I/O and computation in the cluster). But as the contention grows, it outperforms both early-start algorithm (in the range of 8% to 25%) and delayed- start algorithm (approximately 5%) in the average response time. (Please refer to figure 22)

In case of a batch job, the proposed algorithm recognizes the fact that it is an aggregating workload and hence attempts to delay the scheduling of Reduce as far as possible. (We observed that the Reduce phase is scheduled after approximately 95% completion of Map phase resulting into slow-start threshold of 0.95). From figure 23 we can see that the

adaptive slow-start algorithm provides comparable results with respect to the delayed-start algorithm.

Since the improvement in average response time for short jobs is not at the expense of corresponding degradation of batch job, we can conclude the proposed adaptive slow-start algorithm is able to improve the utilization of Hadoop cluster without compromising on the objectives of Hadoop scheduler.

5.5.2.2 Write intensive workloads in batch and interactive modes

In this experiment we use write-intensive workload for both the long running batch jobs (a TeraSort job operating on 30GB of input data) and one or more short jobs (a TeraSort job operating on 512MB of input data). The main goal of this experiment is to evaluate the performance of adaptive slow-start algorithm in case of write-intensive batch jobs (since this workload hints the algorithm to start the Reduce phase early).

The figure 24 depicts the average response time for a short job (relative to corresponding value on a dedicated cluster configured with early-start policy) against the total number of concurrent jobs in the system. Here the case of a single job represent the average response time for a batch job on a dedicated Hadoop cluster configured with early-start policy which is used to normalize all other values in the graph.

From figure 24 we can see that as in previous experiment (please refer to section 5.5.2.1), the proposed adaptive slow-start algorithm outperforms early-start algorithm by approximately 2% to 10% for short jobs on a contended cluster while providing comparable results when there is not contention.

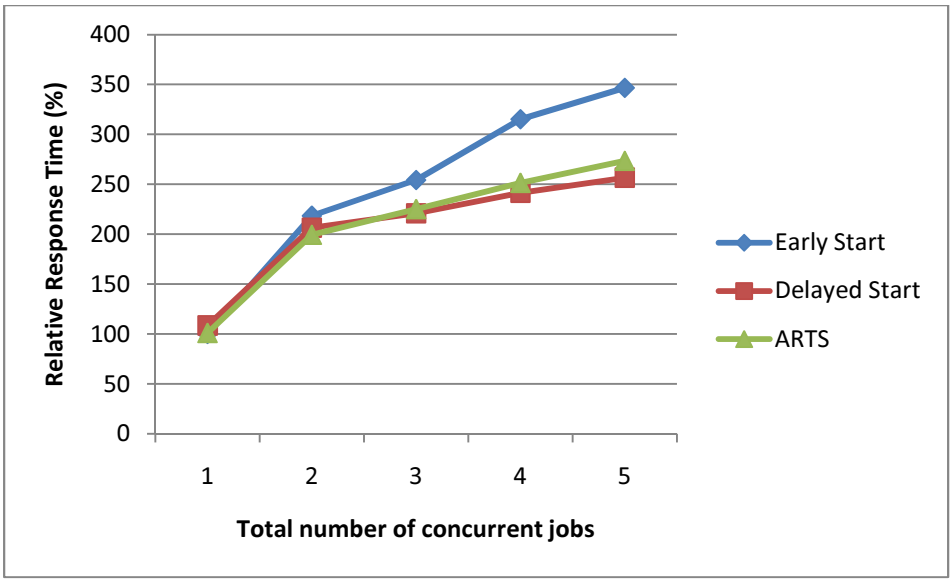

Figure 24: Average relative response time for short (interactive) jobs for different Reduce phase scheduling alternatives

For this combination of workloads, it provides comparable results with respect to delayed-start algorithm on a contented cluster (degradation in average response time is in the range of 1% - 2%). This is different from previous experiment (please refer to section 5.5.2.1) since the class of workload for the batch job is different (explained later in this section).

The figure 25 depicts the average response time (relative to corresponding value on a dedicated cluster configured with early-start policy) for a batch job against the total number of concurrent jobs in the system. Here the case of a single job represent the average response time for a batch job on a dedicated Hadoop cluster configured with early-start policy which is used to normalize all other values in the graph.

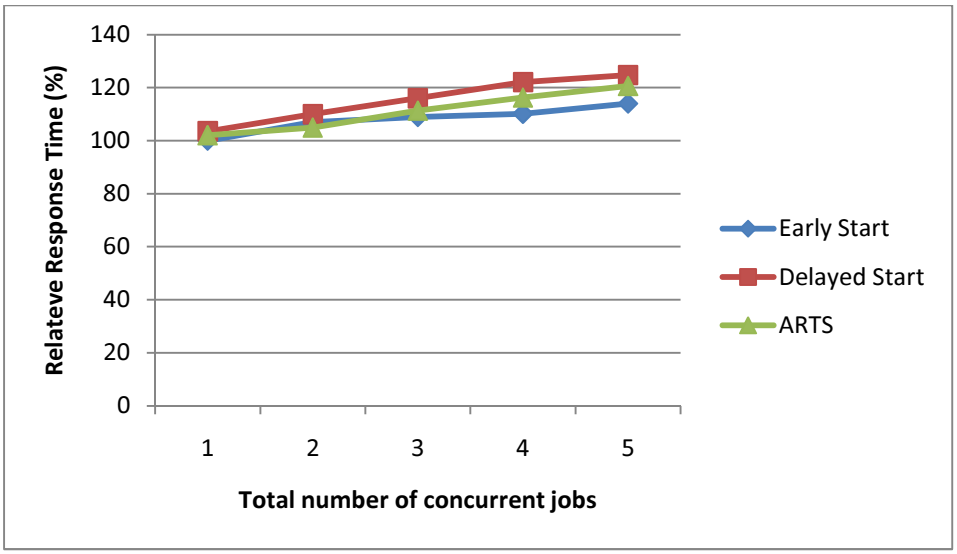

Figure 25: Average relative response time for long (batch) jobs for different Reduce phase scheduling alternatives

In case of a batch job, the proposed algorithm recognizes the fact that it is a write-intensive workload and hence attempts to schedule Reduce phase early. But unlike early-start algorithm, it doesn't start all the Reduce tasks immediately. At every point it intelligently selects a partition which would take maximum (estimated) time for the shuffle operation and the task is scheduled only if the time to copy exceeds (estimated) Map phase completion time (please refer section 0). We observed that after approximately 50% completion of Map phase, all the Reduce tasks are scheduled (resulting into a slow-start threshold of 0.5). Hence this algorithm outperforms the delayed-start algorithm (slow-start threshold of 0.8) by approximately 5% in average response time on a contended cluster.

The proposed algorithm uses a linear model to speed up the scheduling of Reduce tasks (please refer section 0). This helps to shield against the wrong estimation of network bandwidth value. We observe that in case of large job this linear model ends up forcing early start for too many Reduce tasks. This results into increased contention for Reduce

slots for short jobs (similar to early-start). We plan to experiment with different models which can help to reduce the contention for the Reduce slots. This may further improve the average response time for the short jobs.

5.6 Performance evaluation of Adaptive Reduce Task Scheduling (ARTS) algorithm in a distributed Hadoop cluster involving multiple datacenters

In this experiment we evaluate the performance of proposed Adaptive Reduce Task Scheduling (ARTS) algorithm in a distributed Hadoop cluster involving two datacenters one at CAC Lab at Rutgers University and the other at Amazon EC2. For this experiment, we use a TeraSort benchmark with 2GB input dataset. During the experiment we very the distribution of input dataset between two datacenters and measure the response time as well as total Map output copied over the Internet for both schemes (a) the proposed ARTS algorithm and (b) Native Hadoop Reduce task scheduling algorithm. To ensure that the distribution of Map output remains the same in both of these cases, we modified the Map phase scheduling algorithm to disable the execution of Map tasks across the datacenters.

In the first experiment we setup the entire 2GB input dataset in Amazon EC2 and executed the TeraSort benchmark for both of the Reduce task scheduling policies. The following table shows the results for this experiment,

	Native Hadoop	ARTS
Map Output - EC2	2129.92	2129.92
Map Output - CAC	0	0
Reduce Tasks - EC2	1	3
Reduce Tasks - CAC	2	0
Data Transfer over Internet (MB)	1418.08	0
AVG Runtime (s)	186.25	132.67

Here we can see that since the entire Map output is available in EC2 datacenter, the proposed ARTS algorithm schedules all the Reduce tasks in EC2 and as a result there is no copy of Map output across the datacenters. The native Hadoop scheme, on the other hand, does not understand the datacenter-locality of the Map output and hence scatters the Reduce tasks across the datacenters resulting into copy of substantial size of Map output.

In the second experiment, we divide the input dataset such that distribution is 4:5 (EC2: CAC). The following table shows the results,

	Native Hadoop	ARTS
Map Output - EC2	946.63	946.63
Map Output - CAC	1183.29	1183.29
Reduce Tasks - EC2	1	2
Reduce Tasks - CAC	2	1
Data Transfer over Internet (MB)	1022.03	1100.78
AVG Runtime (s)	208.25	171.33

From the table above we can see that in this case Native Hadoop scheduling algorithm performs better in terms of total data transfer over the Internet with respect to proposed ARTS algorithm. But in terms of execution time the ARTS algorithm outperforms the Native Hadoop scheduling algorithm. Careful analysis of job execution for both of these

policies reveal that the relative performance of Reduce phase depends upon the capabilities of the worker executing the Reduce tasks as well as the total amount of data transfer over the Internet. Since the workers in EC2 have better I/O performance (medium-large instances) than the workers in CAC lab, the ARTS policy provides better execution time (although this not by design). As a future work, we would like to incorporate the understanding of worker node profile in the scheduling algorithm (along with the total data transfer over the Internet).

6. Summary, conclusion and future work

6.1 Summary

The primary objective of this research was to study and evaluate design assumptions behind different components of Hadoop eco-system (specifically Hadoop Distributed File System (HDFS) and MapReduce) which mandate the processing of MapReduce computations in a centralized environment.

The study and evaluation of Hadoop MapReduce framework on different types on workloads in a distributed environment involving multiple datacenters (namely CAC Lab at Rutgers University and the Amazon EC2) reveled inefficiencies in the design of Hadoop Distributed File System (HDFS) as well as the Reduce phase scheduling algorithm in MapReduce.

The fundamental design assumption of Hadoop framework is the availability of a single logical distributed file-system storing the input dataset. Hence for executing MapReduce computation across multiple datacenters, it is essential for Hadoop Distributed File System (HDFS) to scale over the Internet. We observed that in such an environment performance of HDFS write operations suffer severely since the default replica placement policy does not take into consideration the datacenter-locality of the client during replica selection resulting into unnecessary copy of data blocks over the Internet.

To solve this problem, we have designed a WAN-aware replica placement policy which ensures that the replicas are placed in the same datacenter as the client, provided sufficient storage space is available in that datacenter. With this policy clients in multiple

datacenters would be able to share a unified namespace without the overhead of data transfer over the Internet.

We also observed that in Hadoop framework Reduce phase is started after completion of user-specified fraction Map phase for a job. Once the Reduce phase starts, Hadoop assigns a Reduce task to any node at random. This static task scheduling policy is very inefficient in distributed environment involving multiple datacenters as well as in a shared environment executing concurrent MapReduce jobs. We have designed an Adaptive Reduce Task Scheduling (ARTS) algorithm which performs well both in distributed and shared environments. This algorithm keeps a track of total size of Map phase output for each partition segregated with respect to datacenters. In a distributed environment, this information helps to identify the datacenter-locality while scheduling Reduce task. Also this information is used to classify the workload for the job. Adapting start time for Reduce phase with respect to type of workload as well as size job helps this algorithm to consistently outperform default Reduce phase scheduling algorithm in a shared Hadoop environment.

6.2 Conclusion

The investigation conducted as part of this research points out the necessity of a large-scale data processing framework for geographically scattered datasets. MapReduce being a popular programming model for large-scale data processing; we have proposed two extensions for Hadoop MapReduce framework which improve its performance in distributed environment involving multiple datacenters.

Firstly, proposed WAN aware replica placement policy in HDFS improves the performance of write operations in the range of 100% up to 200% over the default policy

in a distributed Hadoop cluster involving CAC cluster at Rutgers University as well as Amazon EC2.

Secondly, proposed Adaptive Reduce Task Scheduling (ARTS) algorithm significantly reduces the size of Map phase output copied across datacenters over the Internet by understanding the datacenter-locality of the Map phase output during the scheduling Reduce tasks. In addition in a shared Hadoop environment, this algorithm adapts the start time of the Reduce phase for a job by taking into consideration factors like type of workload as well as the size of job. This algorithm consistently outperforms default Reduce phase scheduling policies (namely early-start and delayed-start) in a shared Hadoop environment for different combinations of workloads.

6.3 Future Work

During our current evaluation we have not considered many design aspects such as Map phase scheduling, speculative execution etc. of the MapReduce framework. Improving efficiency of these aspects is critical to truly achieve the goal of extending MapReduce framework in a distributed environment involving multiple datacenters connected over the Internet.

Specifically, we plan to evaluate and improve following aspects of MapReduce framework.

- **Map phase scheduling**. Whenever a worker requests for a Map task, the Hadoop master attempts to find out if there is any Map task which is either data-local (i.e. having an input data block on the requesting node) or rack-local (i.e. having an input data block on one of the nodes in the same rack as the requesting node). If

no such Map task can be found, it assigns any Map task at random. In a centralized environment, this helps to improve the utilization of Hadoop cluster. But as we scale Hadoop cluster over the Internet, this scheduling may not be efficient as the cost to copy the data block to the remote node over the Internet may be substantial. Hence in such environment, Master should carefully weigh different criteria such as cost (and time) required to copy the data block to the remote location, cost (and time) required for the remote worker to perform the Map computation as well as the gain in the response time achieved by remote execution. We are planning to extend the Map phase scheduling framework with different policies which optimize factors such as cost of computation, cost of network bandwidth consumed over the Internet or user specified deadlines etc.

- **Speculative execution**. The speculative execution mechanism in Hadoop MapReduce improves the response time of a MapReduce job in the presence of a poorly performing faulty worker nodes (called stragglers) executing Map or Reduce tasks. This is achieved by speculating which worker nodes (and corresponding tasks) are stragglers and executing another copy of such tasks as a backup. The master accepts the result of whichever task completes first (either original or the backup) and other task is killed. Hadoop scheduler assumes that there is no cost of launching a speculative task on any idle worker available. This assumption may not be valid in a distributed Hadoop cluster. We are planning to evaluate this aspect of MapReduce framework.

We are also planning evaluate following aspects of Adaptive Reduce Task Scheduling (ARTS) algorithm,

- The linear model used during the design of ARTS algorithm (please refer to section 4.1 for details) is not optimized for scheduling Reduce tasks for write-intensive workloads (Please refer to section 5.5.2.2 for details). We are planning to evaluate different models which can improve its efficiency.
- At present the ARTS algorithm depends upon the value of network bandwidth provided by the user for calculating the time required for Shuffle phase. We are planning to automatically infer the network bandwidth available in the given Hadoop cluster.

7. References

[1] Luiz Barroso, Jeffrey Dean, and Urs H¨olzle. 2003. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro* 23, 2 (March 2003), 22-28.

[2] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In Proceedings of the 19th ACM symposium on Operating systems principles (SOSP '03). ACM, New York, NY, USA, 29-43.

[3] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: simplified data processing on large clusters. In Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6 (OSDI'04), Vol. 6. USENIX Association, Berkeley, CA, USA, 10-10.

[4] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments. In Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08). USENIX Association, Berkeley, CA, USA, 29-42.

[5] Hadoop - The Definitive Guide. Tom White (O'Reilly publication)

[6] http://www.rackspace.com/whyrackspace/network/datacenters.php

[7] Alonzo Church and John Barkley Rosser. "Some properties of conversion". Transactions of the American Mathematical Society, Volume 39, No. 3. (1936), 472–482.

[8] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. -I. Hsiao, and R. Rasmussen. 1990. The Gamma Database Machine Project. IEEE Trans. on Knowl. and Data Eng. 2, 1 (March 1990), 44-62.

[9] http://www.emc.com/collateral/analyst-reports/expanding-digital-idc-white-paper.pdf

[10] Yunhong Gu and Robert Grossman. Sector and Sphere: The Design and Implementation of a High Performance Data Cloud, Theme Issue of the Philosophical Transactions of the Royal Society A: Crossing Boundaries: Computational Science, E-Science and Global E-Infrastructure, 2009, vol. 367, no. 1897, page 2429-2445

[11] Forrester Consulting. eCommerce Web Site Performance Today: An Updated Look at Consumer Reaction to a Poor Online Shopping Experience. 2009.

[12] Hadoop, http://hadoop.apache.org/

[13] The Hadoop Fair Share scheduler.
http://hadoop.apache.org/common/docs/r0.20.2/fair_scheduler.html

[14] Brian Cho and Indranil Gupta. 2010. New Algorithms for Planning Bulk Transfer via Internet and Shipping Networks. In Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems (ICDCS '10). IEEE Computer Society, Washington, DC, USA, 305-314.

[15] Jim Gray and David Patterson, 2003. “A Conversation with Jim Gray”. Queue 1, 4 (June 2003), 8-17.

[16] Niraj Tolia, Michael Kaminsky, David G. Andersen, and Swapnil Patil. 2006. An architecture for internet data transfer. In Proceedings of the 3rd conference on Networked Systems Design \& Implementation - Volume 3 (NSDI'06), Vol. 3. USENIX Association, Berkeley, CA, USA, 19-19.

[17] Randolph Y. Wang, Sumeet Sobti, Nitin Garg, Elisha Ziskind, Junwen Lai, and Arvind Krishnamurthy. 2004. Turning the postal system into a generic digital communication mechanism. In Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '04). ACM, New York, NY, USA, 159-166.

[18] Tevfik Kosar and Miron Livny. 2004. Stork: Making Data Placement a First Class Citizen in the Grid. In Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04) (ICDCS '04). IEEE Computer Society, Washington, DC, USA, 342-349.

[19] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. SIGOPS Oper. Syst. Rev. 41, 3 (March 2007), 59-72.

[20] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In Proceedings of the 5th European conference on Computer systems (EuroSys '10). ACM, New York, NY, USA, 265-278.