Analysis of Data Placement Strategy based on Computing Power of Nodes on Heterogeneous Hadoop Clusters

by

Sanket Reddy Chintapalli

A project submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
December 12, 2014

Keywords: MapReduce, HDFS, Computing Capability, Heterogenous

Approved by

Xiao Qin, Associate Professor of Computer Science and Software Engineering
James Cross, Professor of Computer Science and Software Engineering
Jeffrey Overbey, Assistant Professor of Computer Science and Software Engineering

Abstract

Hadoop and the term 'Big Data' go hand in hand. The information explosion caused due to cloud and distributed computing lead to the curiosity to process and analyze massive amount of data. The process and analysis helps to add value to an organization or derive valuable information.

The current Hadoop implementation assumes that computing nodes in a cluster are homogeneous in nature. Hadoop relies on its capability to take computation to the nodes rather than migrating the data around the nodes which might cause a significant network overhead. This strategy has its potential benefits on homogeneous environment but it might not be suitable on an heterogeneous environment. The time taken to process the data on a slower node on a heterogeneous environment might be significantly higher than the sum of network overhead and processing time on a faster node. Hence, it is necessary to study the data placement policy where we can distribute the data based on the processing power of a node. The project explores this data placement policy and notes the ramifications of this strategy based on running few benchmark applications.

Acknowledgments

I would like to express my deepest gratitude to my adviser Dr. Xiao Qin for shaping my career. I would like to thank Dr. James Cross and Dr. Jeffrey Overbey for serving on my advisory committee.

Auburn University has my regard for making me a better engineer and I would like to thank all my professors and staff for shaping my intellect and personality. I would like to thank my friends and family for supporting me throughout my tenure at Auburn. I had great fun exploring various technologies, meeting interesting people, taking up challenges and expanding my perspective on software development and engineering.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data intensive applications are growing at a fast pace. The need for storing the data and processing them in order to extract value from skew data has paved way for parallel, distributed processing applications like Hadoop.

The ability of such applications to process petabytes of data generated from websites like Facebook, Amazon, Yahoo and search engines like Google and Bing have eventually lead to a data revolution where by processing each and every minute information relating to customers and users can add value, there by improving core competency.

Hadoop is an open source application first developed by Yahoo inspired from the Google File System. Hadoop comprises of task execution manager and a data storage unit known as HDFS (Hadoop Distributed File System) based on Google's Big Data Table [1]. Hadoop has two predominant versions available today namely Hadoop 1.x and 2.x. The 2.x has features pertaining to the improvement of the batch processing feature by introducing YARN (Yet Another Resource Negotiator). YARN comprises of a Resource Manager and Node Manager. The Resource Manger is responsible for managing and deploying resources and the Node Manager is responsible for managing the datanode and reporting the status of the datanode to the resource manager.

1.1 Scope

1.1.1 Data Distribution

We observed that data locality is a determining factor for MapReduce performance. To balance workload in a cluster, Hadoop distributes data to multiple nodes based on disk space

availability. Such a data placement strategy is very practical and efficient for a homogeneous environment, where computing nodes are identical in terms of computing and disk capacity. In homogeneous computing environments, all nodes have identical workloads, indicating that no data needs to be moved from one node to another. In a heterogeneous cluster, however, a high-performance node can complete local data processing faster than a low-performance node. After the fast node finishes processing the data residing in its local disk, the node has to handle the unprocessed data in a slow remote node. The overhead of transferring unprocessed data from slow nodes to fast ones is high if the amount of data moved is large. An approach to improve MapReduce performance in heterogeneous computing environments is to significantly reduce the amount of data moved between slow and fast nodes in a heterogeneous cluster. To balance the data load in a heterogeneous Hadoop cluster, we investigate data placement schemes, which aim to partition a large data set into small fragments being distributed across multiple heterogeneous nodes in a cluster. Unlike other data distribution algorithms, it takes care of replication and network topology before moving the data between the nodes.

1.2 Contribution

Data placement in HDFS. We develop a data placement mechanism in the Hadoop distributed system or HDFS to initially distribute a large data set to multiple computing nodes in accordance with the computing capacity of each node. More specifically, a data reorganization algorithm is implemented in HDFS.

1.3 Organization

This project is organized as follows. Chapter 2 explains in detail the Hadoop's architecture, HDFS and MapReduce framework in Hadoop along with the new YARN framework. Chapter 3 explains the problem and explains existing solutions. Chapter 4 describes the

design of the redistribution mechanism. Chapter 5 analyzes the results and performance of the balancer/redistribution algorithm.

Chapter 2

Hadoop

Apache Hadoop is an open source software project that enables the distributed processing of large data sets across clusters of commodity servers. It is designed to scale up from a single server to thousands of machines, with a very high degree of fault tolerance. Rather than relying on high-end hardware, the resiliency of these clusters comes from the softwares ability to detect and handle failures at the application layer. Hadoop enables a computing solution that is scalable, cost effective, flexible and fault tolerant [2] [3].

2.1 Hadoop Architecture

Hadoop is implemented using Client-Master-Slave design pattern. Currently there are two varied implementations of Hadoop namely 1.x and 2.x. Hadoop 1.x generally manages the data and the task parallelism through Namenode and JobTracker respectively. On the other hand Hadoop 2.x uses YARN (Yet Another Resource Negotiator).

In Hadoop 1.x there are two masters in the architecture, which are responsible for the controlling the slaves across the cluster. The first master is the NameNode, which is dedicated to manage the HDFS and control the slaves that store the data. Second master is JobTracker, which manages parallel processing of HDFS data in slave nodes using the MapReduce programming model. The rest of the cluster is made up of slave nodes which runs both DataNode and TaskTracker daemons. DataNodes obey the commands from its master NameNode and store parts of HDFS data decoupled from the meta-data in the NameNode. TaskTrackers on the other hand obeys the commands from the JobTracker and does all the computing work assigned by the JobTracker. Finally, Client machines are neither Master or a Slave. The role of the Client machine is to give jobs to the masters to load data

into HDFS, submit Map Reduce jobs describing how that data should be processed, and then retrieve or view the results of the job when its finished.

YARN uses ResourceManager which spawns multiple NodeManagers on subsequent nodes on the cluster. The NodeManager is responsible for deploying the tasks and reporting the status of the node to the Resource Manager [6].

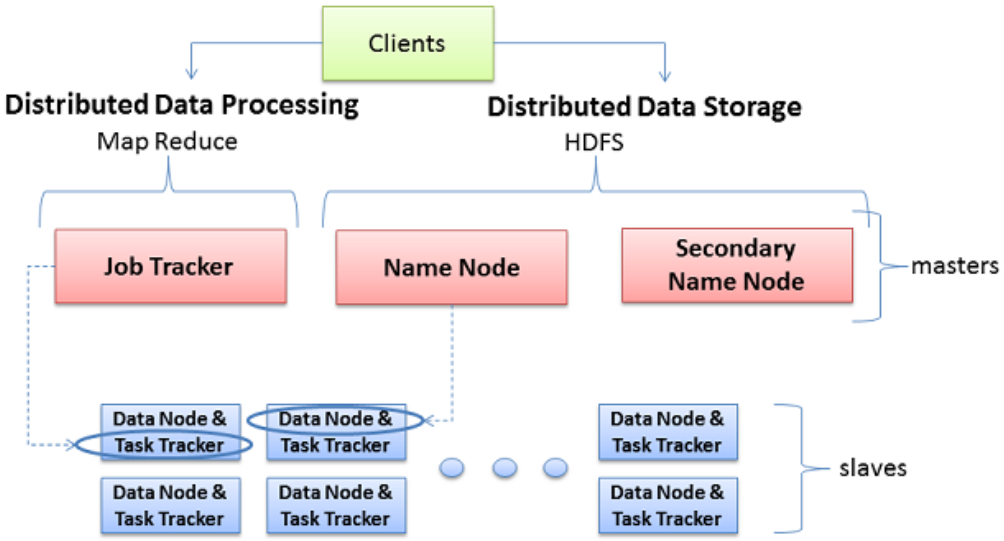

Figure 2.1: Hadoop Architecture [5]

Figure 2.1 shows the basic organization of the Hadoop cluster. The client machines communicates with the NameNode to add, move, manipulate, or delete files in HDFS. The NameNode in turn calls the DataNodes to store, delete or make replicas of data being added to HDFS. When the client machines want to process the data in the HDFS, they communicate to the JobTracker to submit a job that uses MapReduce. JobTracker divides the jobs to map/reduce tasks and assigns it to the TaskTracker to process it.

Typically, all nodes in Hadoop cluster are arranged in the air cooled racks in a data center. The racks are linked with each other with the help of rack switches which runs on TCP/IP.

2.2 MapReduce

In Introduction chapter we understood that MapReduce is a programming model designed for processing large volumes of data in parallel by dividing the work into a set of independent tasks. MapReduce programs are influenced by functional programming constructs used for processing lists of data. The MapReduce fetches the data from the HDFS for parallel processing. These data are divided in to blocks as mentioned in the section above.

2.2.1 JobTracker and TaskTracker

JobTracker is the master, to which the applications submit MapReduce jobs. The JobTracker gets the map tasks based on input splits and assigns tasks to TaskTracker nodes in the cluster. The JobTracker is aware of the data block location in the cluster and machines which are near the data. The JobTracker assigns the job to TaskTracker that has the data with it and if it cannot, then it schedules it to the nearest node to the data to optimize the network bandwidth. The TaskTracker sends a HeartBeat message to the JobTracker periodically, to let JobTracker know that it is healthy, and in the message it includes the memory available, CPU frequency and etc. If the TaskTracker fails to send a HeartBeat to the JobTracker, the JobTracker assumes that the TaskTracker is down and schedules the task to the other node which is in the same rack as the failed node [3].

The Figure 2.2 [4] shows the data flow of MapReduce in couple of nodes . The steps below explains the flow of the MapReduce [4].

1. Split the file: First the data in the HDFS are split up and read in *InputFromat* specified. InputFormat can be specified by the user and any InputFormat chosen would read the files in the directory, select the files to be split into InputSplits and give it to

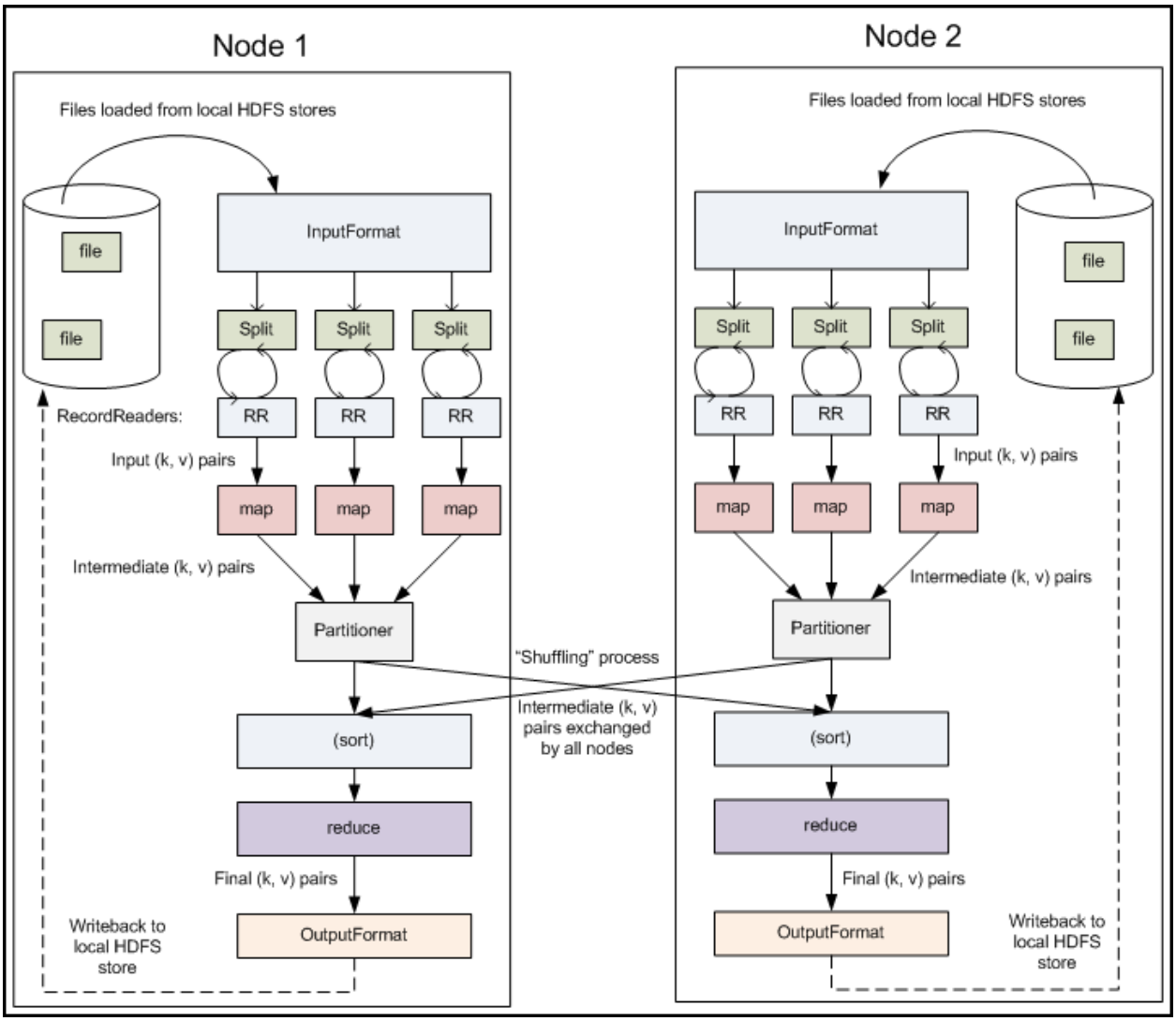

Figure 2.2: MapReduce data flow.

RecordReader to read the records in (key, value) pair that would be processed in further steps. Standard InputFormats provided by the MapReduce are TextInputFormat, SequenceInputFormat, MultipleInputFormat [3].

The InputSplit is the unit work that comprises a single map task in a MapReduce program. The job submitted by the client is divided into the number of tasks, which is equal to the number of InputSplits. The default InputSplit size is 64MB / 128MB and can be configured by modifying split size parameter. The input splits enable the parallel processing of MapReduce by scheduling the map tasks on other nodes in cluster at same time. When the HDFS splits the file into blocks, the task assigned to that node accesses the data locally.

2. Read the records in InputSplit: The InputSplit although is ready to be processed it still does not make sense to the MapReduce program as the input to it is not in key-value format. The *RecordReader* actually loads the data and converts it to key, value pair expected by the Mapper task. The calls to RecordReader calls map() method of Mapper.

3. Process the records: When the Mapper gets the key-value pair from the RecordReader, it calls the map() function to process the input key-value pair and output an intermediate key-value pair. While these mappers are reading their share of data and processing it in parallel fashion across the cluster, they do not communicate with each other as they have no data to share. Along with the key-value pair, the Mapper also gets couple of objects, which indicates where to forward the output and report the status of task [2].

4. Partition and Shuffle: The mappers output the key,value pair which is the input for the reducer. This stage the mappers begin exchanging the intermediate outputs and the process is called shuffling. The reducer reduces the intermediate value with the same key and it partitions all the intermediate output with the same key. The partitioner

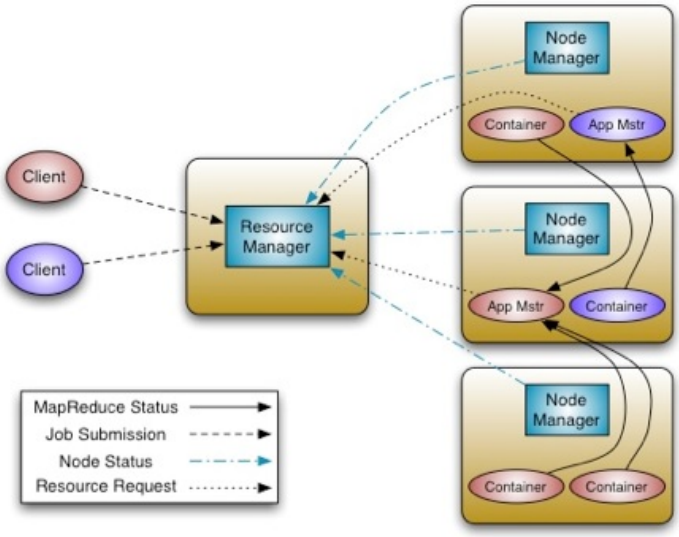

Figure 2.3: YARN

determines which partition a given ¡key,value¿ pair go to. The intermediate data are sorted before they are presented to the Reducer.

5. Reduce the mapper's output: For every key in the assigned partition in the reducer a reduce() function is called. Because the reducer reduces the partition with the same key, it iterates over the partition to generate the output. The OutputFormat will specify the format of the output records, and the reporter object reports the status. The RecordWriter writes the data to file specified by the OutputFormat [2].

2.2.2 YARN

YARN is a tool which is responsible for managing the job and data allocation.The fundamental idea of YARN is to split up the two major responsibilities of the JobTracker i.e. resource management and job scheduling/monitoring, into separate daemons: a global ResourceManager and per-application ApplicationMaster (AM) [6]. The ResourceManager and per-node slave, the NodeManager (NM), form the new,

and generic, system for managing applications in a distributed manner. Figure 2.3 gives an overview of the YARN architecture.

The ResourceManager is the ultimate authority that arbitrates resources among all the applications in the system. The per-application ApplicationMaster is, in effect, a framework specific entity and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the component tasks.

The ResourceManager has a pluggable Scheduler, which is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc. The Scheduler is a pure scheduler in the sense that it performs no monitoring or tracking of status for the application, offering no guarantees on restarting failed tasks either due to application failure or hardware failures. The Scheduler performs its scheduling function based on the resource requirements of the applications; it does so based on the abstract notion of a Resource Container which incorporates resource elements such as memory, cpu, disk, network etc.

The NodeManager is the per-machine slave, which is responsible for launching the applications containers, monitoring their resource usage (cpu, memory, disk, network) and reporting the same to the ResourceManager.

The per-application ApplicationMaster has the responsibility of negotiating appropriate resource containers from the Scheduler, tracking their status and monitoring for progress. From the system perspective, the ApplicationMaster itself runs as a normal container.

2.3 HDFS

Hadoop Distributed File System is the filesystem designed for Hadoop to store the large sets of data reliably and stream those data to the user application at the high

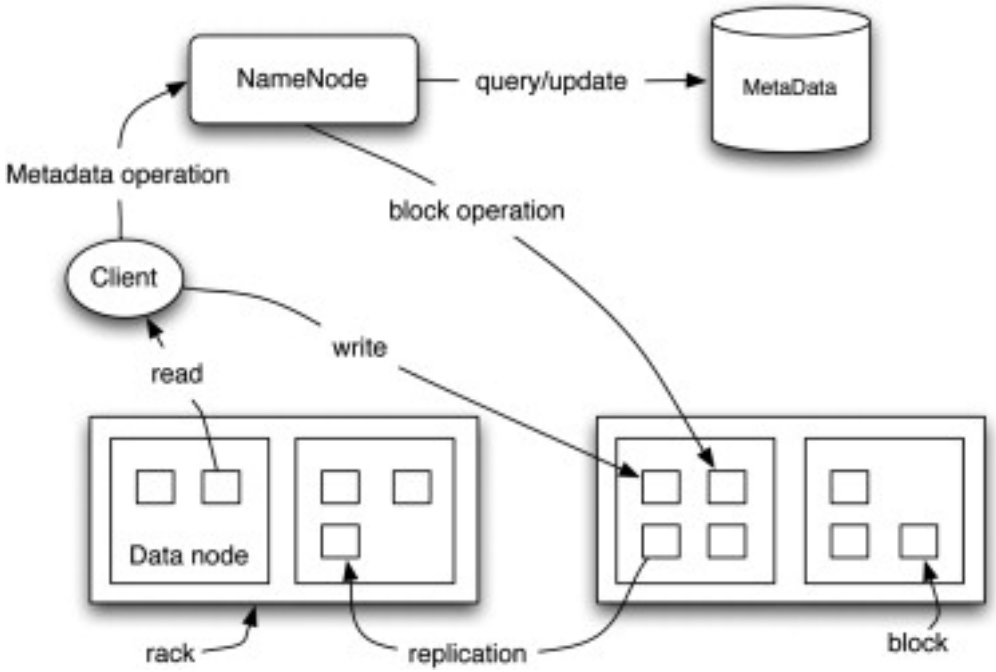

Figure 2.4: HDFS Architecture [3]

throughput rather than providing low latency access. Hadoop is designed in Java and that makes it incredibly portable across platform and operating systems. Like the other distributed file systems like Lustre and PVFS, HDFS too stores the meta data and the data separately. The NameNode stores the meta-data and the DataNodes store the application data. But, unlike Lustre and PVFS, the HDFS stores the replicas of the data to provide high throughput data access from multiple sources and also data redundancy increases the fault tolerance of HDFS [5] [3].

When the HDFS replicates it does not replicate the entire file, it divides the files into fixed sized blocks and the blocks are placed and replicated in the DataNodes. The default block size in Hadoop is 64MB and is configurable.

2.3.1 NameNode, DataNode and Clients

The Figure 2.4 shows the HDFS architecture in Hadoop which contains three important entities- NameNode, DataNode and Client. The NameNode is responsible for storing

the meta-data, and track the memory available and used in all the DataNodes. The client which wants to read the data in the HDFS first contacts the NameNode. The Namenode then looks for the block's DataNode which is nearest to the client and tells the client to access the data from it. Similarly, when the client wants to write a file to the HDFS, it requests the NameNode to nominate 3 DataNodes to store the replicas and the client writes to it in streamline fashion. The HDFS would work efficiently if it stored the files of larger size, at least size of a block because the HDFS stores the Namespace RAM. If it were all smaller files in HDFS then the inodes information would occupy the entire RAM leaving no room for other operations.

The NameNode would register all the DataNodes at the start-up based on the NamspaceID. The NameSpaceID would be generated when the NameNode formats the HDFS. The DataNodes are not allowed to store any blocks of data if the NamespaceID does not match with the ID of the NameNode. Apart from the registering the DataNodes in the start-up the DataNodes send the block reports to the NameNode periodically. The block report contains the block id, the generation report and the length of the each block that DataNode holds. Every tenth report sent from the DataNode is a block report to keep the NameNode updated about all the blocks. A DataNode also sends the HeartBeat messages that just notify the NameNode that it is still healthy and all the blocks in it are intact. When the NameNode does not receive a heartbeat message from the DataNode for about 10 seconds, it assumes that the DataNode is dead and uses it's policies to replicate the data blocks in the dead node to other nodes that are alive.

Similar to most conventional file systems, HDFS supports operations to read, write and delete files, and operations to create and delete directories. The user references files and directories by paths in the namespace. The user application generally does not need to know that file system metadata and storage are on different servers, or that blocks have multiple replicas.

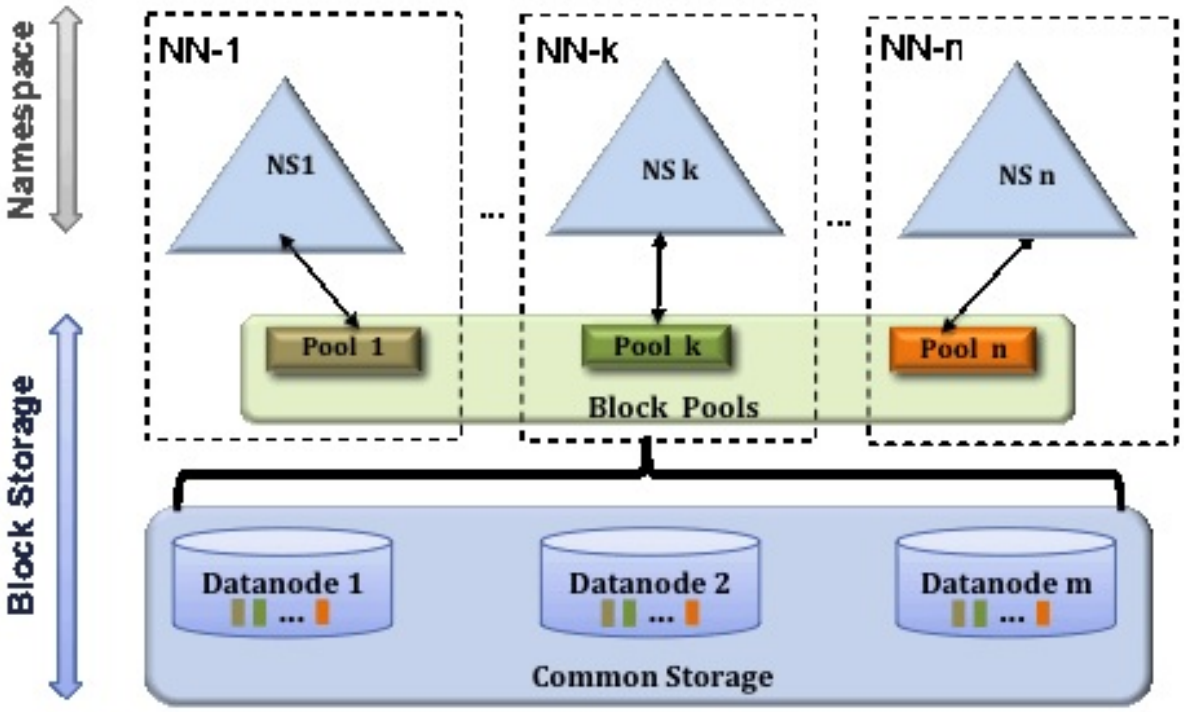

Figure 2.5: HDFS Federated Namenode

2.3.2 Federated Namenode

In hadoop 2.x a new concept know as federated namenode has been introduced. In order to scale the name service horizontally, federation uses multiple independent Namenodes/namespaces. The Namenodes are federated, that is, the Namenodes are independent and dont require coordination with each other. The datanodes are used as common storage for blocks by all the Namenodes. Each datanode registers with all the Namenodes in the cluster. Datanodes send periodic heartbeats and block reports and handles commands from the Namenodes. The Key benefits of a federated namenode are namespace scalability, performance and isolation. Figure 2.5 gives an overview about the architecture

6. Namespace Scalability: HDFS cluster storage scales horizontally but the namespace does not. Large deployments or deployments using lot of small files benefit from scaling the namespace by adding more Namenodes to the cluster

7. Performance: File system operation throughput is limited by a single Namenode in the prior architecture. Adding more Namenodes to the cluster scales the file system read/write operations throughput.

8. Isolation: A single Namenode offers no isolation in multi user environment. An experimental application can overload the Namenode and slow down production critical applications. With multiple Namenodes, different categories of applications and users can be isolated to different namespaces.

2.3.3 Backup Node and Secondary NameNode

The NameNode is the single point of failure for the Hadoop cluster, so the HDFS copies the of the Namespace in NameNode periodically to a persistent storage for reliability and this process is called checkpointing. Along with the NameSpace it also maintains a log of the actions that change the Namespace, this log is called journal. The checkpoint node copies the NameSpace and journal from NameNode to applies the transactions in journal on the Namespace to create most up to date information of the namespace in NameNode. The backup node however copies the Namespace and accepts journal stream of Namespace and applies transactions on the namespace stored in its storage directory. It also stores the upto-date information of the Namespace in memory and synchronizes itself with the NameSpace. When the NameNode fails, the HDFS picks up the Namespace from either BackupNode or CheckPointNode [5] [3].

2.3.4 Replica and Block Management

HDFS makes replicas of a block with a strategy to enhance both the performance and reliability. By default the replica count is 3, and it places the first block in the node of the writer, the second is placed in the same rack but different node and the third replica is placed in different rack. In the end, no DataNode contains more than one replica of a block and no rack contains more than two replicas of same block. The nodes chosen on the basis of proximity to the writer, to place the blocks.

There are situations when the blocks might be over-replicated or under-replicated. In case of over-replication the NameNode deletes the replicas within the same rack first and from the DataNode, which has least available space. In case of under-replication, the NameNode maintains a priority queue for the blocks to replicate and the priority is high for the least replicated blocks.

There are tools in HDFS to maintain the balance and integrity of the data. *Balancer* is a tool that balances the data placement based on the node disk utilization in the cluster. The *Block Scanner* is a tool used to check integrity using checksums. *Distcp* is a tool that is used for inter/intra cluster copying. The intention of this project is to modify the *Balancer* to take into consideration the computing capacity of the nodes as opposed to the space.

Chapter 3

Design

The project focuses on distributing the data based on computing capacity. In the case of a homogeneous environment, the computing capacity and disk capacity are indifferent. As a result, the data gets distributed based on the availability of space on the cluster. In a homogeneous environment, the transfer of data from one node to another in order to fill spaces will cause severe network congestion. Hadoop has a balancing functionality which balances the data before running the applications in the event of severe data accumulation on few nodes. The replication factor also plays a major role in the data movement and the balancer takes care of this aspect. We name the balancer as *CRBalancer*, Computation Ratio Balancer.

On the other hand, in a heterogeneous environment, it makes perfect sense to migrate the data from one node to another, as a faster node accounts for the overhead of data migration and task processing. As a result, the data placement policy suggested in the project tries to explore the ramifications of migrating large portion of data to a faster node in a heterogeneous environment. The data placement algorithm and the implementation details give a greater overview of the implementation that has been carried out to achieve the goal.

3.1 Data Placement: Profiling

Profiling in this scenario is done to compute the computation ratios of nodes within the cluster. In order to accomplish this task, we need run a set of benchmark applications namely Grep and WordCount. The computation ratio is calculated based on

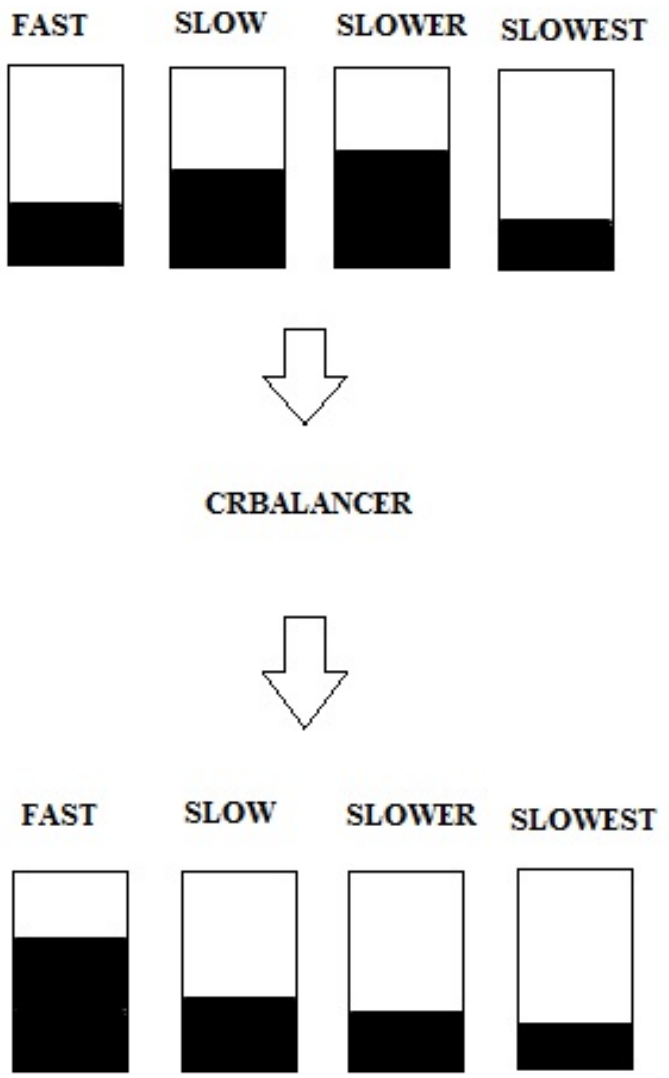

Figure 3.1: Motivation

the processing capacity of the nodes. Lets assume we have 3 nodes A, B and C with A being the fastest, then B and slowest being C. Lets assume it takes 10 seconds to run an application on A, 20 seconds on B and 30 seconds on C. Now, the computation ratio of A would be 6, 3 for B and 2 for C based on the least common multiple principle.

3.2 Implementation Details

The CRBalancer is responsible for migrating the data from one node to another. The Figure 3.2 describes the pattern and the methodology in which the CRBalancer migrates the data. It first takes into consideration the network topology, then calculates the under utilized and over utilized datanodes. Mapping between the under utilized and over utilized datanodes takes place by moving the blocks concurrently among the nodes. It also takes into consideration the replication factor by limiting the migration if the data is already present on the node for the corresponding file. After balancing takes place the benchmarks are run and tested for the results.

The CRBalancer uses CRNamenodeConnector application to connect to the namenode in order to get the information about the datanodes on the fly in order to decide the amount of data to be transferred to the under-utilized nodes from the over-utilized nodes. It also uses CRBalancingPolicy application to keep track of the amount of data within each node.

The CRBalancer can be run in the background while other applications are running on the nodes. The CRBalancer takes care of the network overhead while transferring the blocks. It steals some bandwidth without interrupting the processing of the applications. The expectation is that a similar application performs better when it executed again at some point of time.

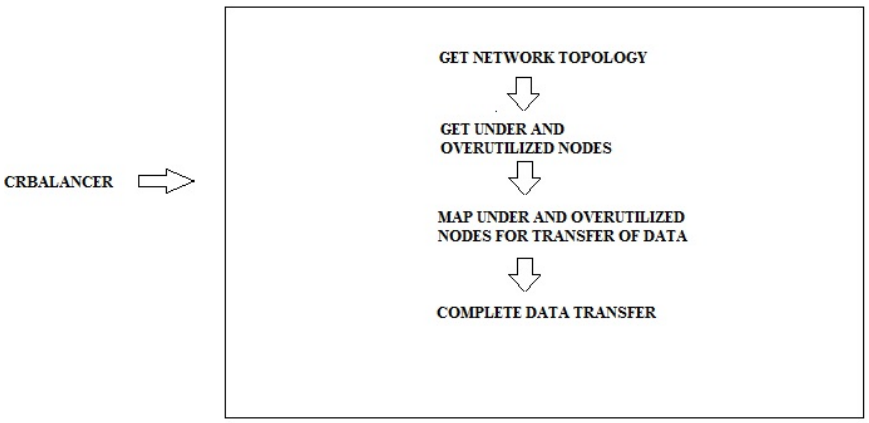

Figure 3.2: Computation Ratio Balancer

Chapter 4

Experiment SetUp

4.1 Hardware

The hardware setup consists of three nodes with a their respective configurations described in the Table 4.1 below.

Node	Processor	Speed	Cache	Storage
A	HP Xeon	4 core * 2.4 GHz	8MB	142.9 GB
B	HP Xeon	4 core * 2.4 GHz	8MB	142.9 GB
C	HP Celeron	1 core * 2.2 GHz	512KB	142.9 GB
D	HP Celeron	1 core * 2.2 GHz	512KB	142.9 GB

Table 4.1: Node Information in Cluster

4.2 Software

The software used in this experiment comprises of Hadoop application 2.3.0. A node is assigned the master status and other nodes are assigned the slave status. The master node contains the namenode which the computing ratio balancer uses to connect to the namenode. It reads block information from the namenode and runs the distribution algorithm in order to distribute the data based on the computing capacity of the nodes.

4.3 Benchmark:WordCount

The word count is a proved cpu intensive benchmark which reads the data-set and counts the number of words within the data-set based on a space delimiter per line.

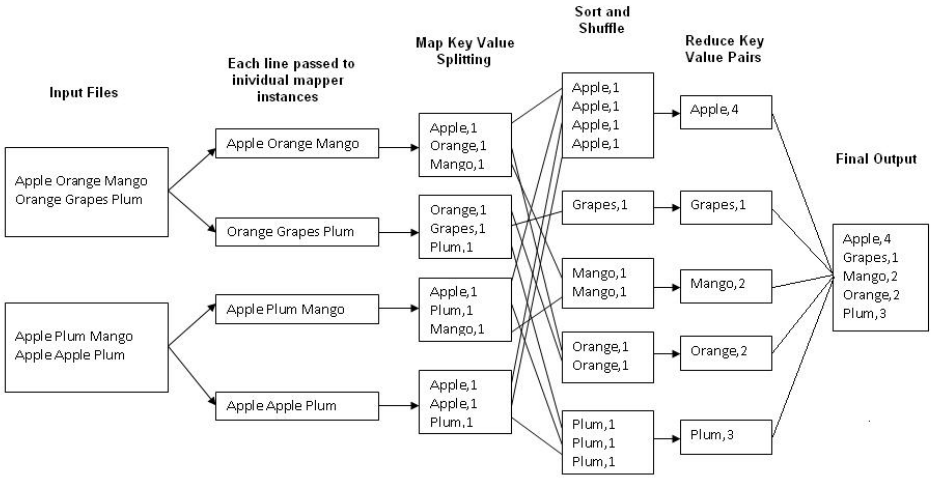

Figure 4.1: Word Count Descripion

The figure below describes the image corresponding to the mechanics of the mapper and reducer in order to obtain the word count within a data-set.

4.4 BenchMark:Grep

The grep is a proved cpu intensive benchmark which reads the data-set and counts the number of words within the data-set based on a search pattern mentioned as an input. The figure below describes the image corresponding to the mechanics of the mapper and reducer in order to obtain the search pattern within a data-set.

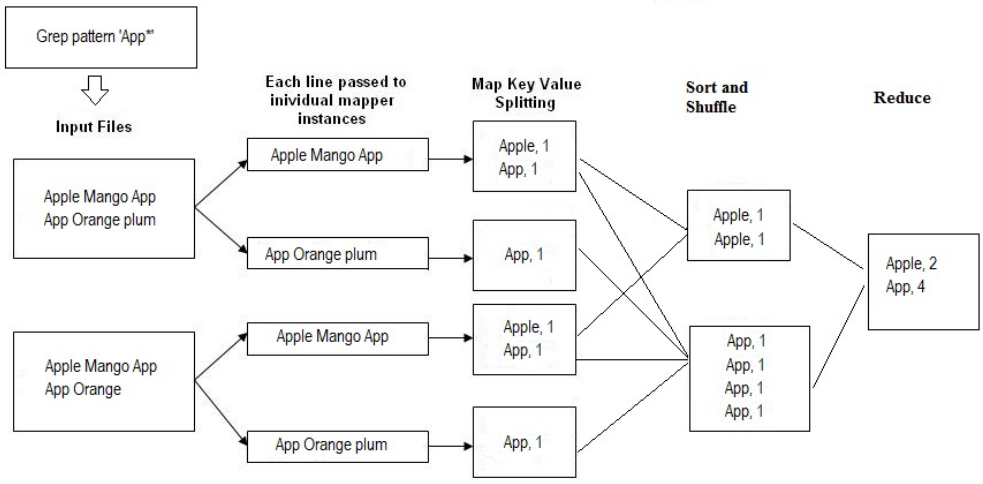

Figure 4.2: Grep description

Chapter 5

Performance Evaluation

The performance evaluation notably runs two benchmark tests namely WordCount and Grep. The computing ratios are calculated by the principle of least common multiple. There are two file sizes 1.1GB, 2.2GB that have been taken into consideration for testing. The time taken to run these files is indicated in the Figure 5.1 and Figure 5.3 corresponding to the two benchmarks. The observation clearly indicates that the time taken by individual nodes to process the file size is linear with respect to time. Hence, computing ratio is a very good metric to balance and process the data based on the computing power of the nodes. The Table 5.1 describes the calculation of the computation ratio. The calculated ratios are put into a configuration file and send as an argument to the balancer to migrate the data from under utilized nodes to over utilized nodes.

Node	**WordCount**	**Grep**
HP Xeon	0.36	0.4
HP Xeon	0.36	0.4
HP Celeron	0.14	0.1
HP Celeron	0.14	0.1

Table 5.1: Computation Ratio

5.1 WordCount

The WordCount Performance graphs indicate that Node A and B being the faster nodes tend to process the entire dataset faster individually. On the other hand Node C and D are considerably slower that the faster performing nodes. Hence, it signifies

that more amount of data has to be present in the faster performing nodes to process the application faster. Therefore, the balancer is run in order to make the distribution according to the computing power. This movement is carried out under the assumption there is sufficient space available on the individual nodes.

The Figure 5.2 shows that 6 different tests are performed. The tests are performed to test the impact of network overload and computing power of the nodes. The bar graphs with the title 'In A - Distributed' describes that the nodes B, C and D are switched off and the datasets are placed in A. After placing the datasets into node A, the nodes B, C and D switched back on. Now the data from A are distributed to B,C and D. Since the data gets distributed during the start up, there is a delay during the start of the cluster. Hence, the additional time during the start up indicates the distribution. Hence, the additional time is taken for processing when compared to 'All - Unbalanced'. Similarly, the tests are run by placing the datasets into B,C and D respectively.

'All - Unbalanced' signifies the Hadoop distribution which distributes the data evenly across all the nodes. 'All - Balanced' indicates the data distributed across all the nodes after running the CRBalancer application.

After the balancing takes place the Figure 5.2 shows a significant improvement of about 11 percent approximately with respect to the WordCount Application. The improvement tends to increase with larger dataset.

5.2 Grep

The Grep Performance graphs indicate that Node A and B being the faster nodes tend to process the entire dataset faster individually. On the other hand Node C and D are considerably slower than the faster performing nodes. Hence, it signifies that more amount of data has to be present in the faster performing nodes to process

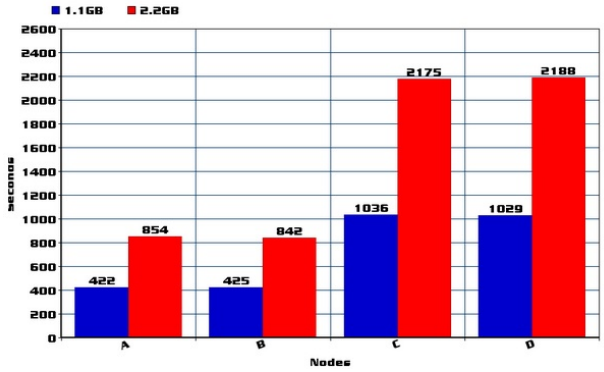

Figure 5.1: Calculating Computation Ratio for WordCount By Running On Individual Nodes

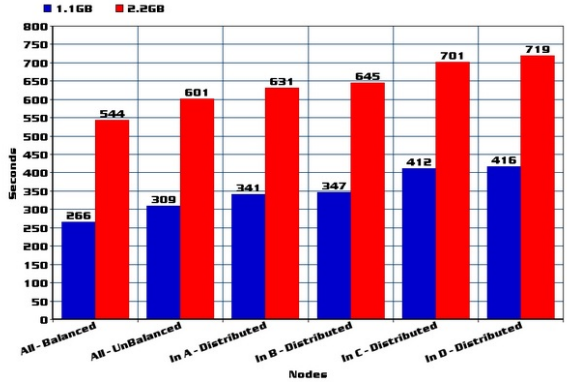

Figure 5.2: WordCount Performance Evaluation After Running CRBalancer

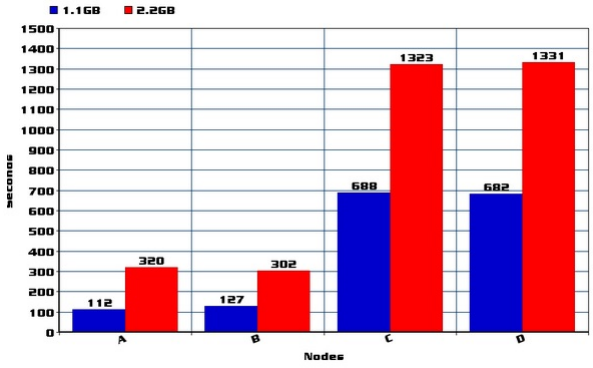

Figure 5.3: Calculating Computation Ratio for Grep By Running On Individual Nodes

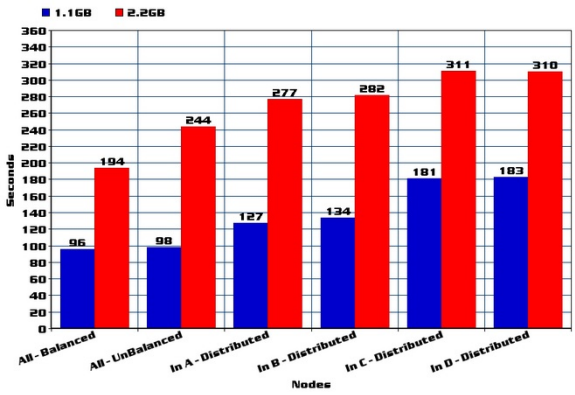

Figure 5.4: Grep Performance Evaluation After Running CRBalancer

the application faster. Therefore, the balancer is run in order to make the distribution according to the computing power. This movement is carried out under the assumption there is sufficient space available on the individual nodes.

The Figure 5.4 shows that 6 different tests are performed. The tests are performed to test the impact of network overload and computing power of the nodes. The bar graphs with the title 'In A - Distributed' describes that the nodes B, C and D are switched off and the datasets are placed in A. After placing the datasets into node A, the nodes B, C and D switched back on. Now the data from A are distributed to B,C and D. Since the data gets distributed during the start up, there is a delay during the start of the cluster. Hence, the additional time during the start up indicates the distribution. Hence, the additional time is taken for processing when compared to 'All - Unbalanced'. Similarly, the tests are run by placing the datasets into B,C and D respectively.

'All - Unbalanced' signifies the Hadoop distribution which distributes the data evenly across all the nodes. 'All - Balanced' indicates the data distributed across all the nodes after running the CRBalancer application.

After the balancing takes place the Figure 5.4 shows a significant improvement of about 25 percent with respect to the Grep Application for a larger dataset although there is no

significant variation when compared to an unbalanced dataset. This clearly indicates that with the increase in datasize the placement algorithm will gain more leverage until a trade-off is made with space and network overload. Hence, this distribution would be suitable for small to moderate workload conditions

5.3 Summary

In this Chapter, we described a performance problem in Hadoop Distributed File System on heterogeneous clusters. Motivated by the performance degradation caused by heterogeneity, we designed and implemented a data placement mechanism in HDFS. The new mechanism distributes fragments of an input file to heterogeneous nodes according to their computing capacities. Our approach significantly improves performance of Hadoop heterogeneous clusters. For example, the empirical results show that our data placement mechanism can boost the performance of the two Hadoop applications (i.e., Grep and Word- Count) by up to 25 and 11 percent approximately. The current project also deals with the replication and cluster network aware data placement. Future work has to deal with data placement with respect to multiple applications running a set of clusters. Also, we might have to deal with the storage space issue and the network overhead and device an efficient data placement algorithm which is dynamic in nature for optimum performance.

Bibliography

[1] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: a distributed storage system for structured data. In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7 (OSDI '06), Vol. 7. USENIX Association, Berkeley, CA, USA, 15-15.

[2] MapReduce: Simplified Data Processing on Large Clusters, Jeffrey Dean, Sanjay Ghemawat.

[3] Cooling Hadoop: Temperature Aware Schedulers in Data Centers, Sanjay Kulkarni, Xiao Qin.

[4] Yahoo Developer Network, https://developer.yahoo.com/hadoop/tutorial/module4.html.

[5] Hadoop Official Site, http://hadoop.apache.org/.

[6] HortonWorks Official Site, http://hortonworks.com