A Comparative Study and Optimization of the Hadoop Scheduler

A project report submitted to

M. S. Ramaiah Institute of Technology

An Autonomous Institute, Affiliated to

Visvesvaraya Technological University, Belgaum

in partial fulfillment of the requirements for the degree of

Bachelor of Engineering in Computer Science & Engineering

Submitted by

Kumudha K N	1MS08CS043
Tejala T	1MS08CS128
Veena S Pilli	1MS08CS133

Under the guidance of
Jayalakshmi D. S
Associate Professor
Department of CSE
MSRIT

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

M. S. RAMAIAH INSTITUTE OF TECHNOLOGY
(Autonomous Institute, Affiliated to VTU)
BANGALORE-560054
www.msrit.edu

May 2012

Department of Computer Science & Engineering

M. S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
BANGALORE-560054

CERTIFICATE

This is to certify that the project work titled "A *Comparative Study and Optimization of the Hadoop Scheduler*" is carried out by

Kumudha K N	1MS08CS043
Tejala T	1MS08CS128
Veena S Pilli	1MS08CS133

in partial fulfillment for the award of degree of Bachelor of Engineering in Computer Science and Engineering during the year 2012. The Project report has been approved as it satisfies the academic requirements with respect to the project work prescribed for Bachelor of Engineering Degree. To the best of our understanding the work submitted in this report has not been submitted, in part or full, for the award of any diploma or degree of this or any other University.

(Jayalakshmi D. S)
Guide

(Dr.R. Selvarani)
Head, Dept. of CSE

(Examiners)

DECLARATION

We hereby declare that the entire work embodied in this report has been carried out by us at M. S. Ramaiah Institute of Technology under the supervision of Jayalakshmi D. S, Associate Professor. This report has not been submitted in part or full for the award of any diploma or degree of this or any other University.

Kumudha K N	1MS08CS043
Tejala T	1MS08CS128
Veena S Pilli	1MS08CS133

Abstract

The current cloud infrastructures face challenges to support cloud-based data-intensive applications that are not only latency-sensitive but also require strong timing guarantees. Hadoop is a general-purpose system that enables high-performance processing of data over a set of distributed nodes. It is a multi-tasking system that can process multiple data sets for multiple jobs for multiple users at the same time. Multiprocessing gives Hadoop the opportunity to map jobs to resources in a way that optimizes their use.

Up until 2008, Hadoop supported a single scheduler where jobs were submitted to a queue, and the Hadoop infrastructure simply executed them in the order of receipt. But recent development has led to the making of pluggable schedulers such as Fair scheduler and Capacity Schedulers. This allows use of new scheduling algorithms to help optimize jobs that have specific characteristics. A further advantage to this change is the increased readability of the scheduler, which has opened it up to greater experimentation and the potential for a growing list of schedulers to specialize in Hadoop's ever-increasing list of applications.

An empirical and exhaustive comparison of the existing Hadoop scheduler with Fair scheduler and Capacity scheduler for data intensive applications is made and the results are analyzed to infer which scheduler is better for which type of workload. Different data intensive applications are developed to run on Hadoop and timing results are documented when they are run under different schedulers.

The existing Hadoop scheduler does not take into account data locality when scheduling tasks. Data locality problems are overcome by implementing a Delay Scheduler which achieves the real-time objective of data locality awareness. A comparative analysis of the Delay Scheduler and the Fair Scheduler is done and the results are analyzed graphically.

ACKNOWLEDGEMENTS

We wish to express our hearty and sincere gratitude to our college M.S Ramaiah Institute of Technology and our Principal Mr. N.V.R Naidu for giving us the opportunity to pursue the UG course in Computer Science and Engineering.

We would like to thank the Department of Computer Science and Engineering and the Head of Department Dr. R Selvarani for giving us the necessary support to execute the project.

Our sincere thanks to our project guide, Mrs. Jayalakshmi D. S., Associate Professor, for her invaluable guidance. She has been a major source of inspiration for us. We would also like to acknowledge the support received from other faculty members of the Computer Science Department. We would like to thank our seniors who have always guided us and helped us whenever we had problems relating to the project.

We also thank our friends for providing constant encouragement, support and valuable suggestions during the development of the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 General Introduction

The MapReduce paradigm supports the execution of data-intensive applications. Each problem is divided into numerous jobs and they are scheduled in a distributed manner. Scheduling of these jobs using the right scheduling policy helps improve the overall performance.

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using a simple programming model. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than relying on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures [1].

1.2 Current Scenario

Hadoop is a general-purpose system that enables high-performance processing of data over a set of distributed nodes. It is a multi-tasking system that can process multiple data sets for multiple jobs for multiple users at the same time. Multiprocessing gives Hadoop the opportunity to map jobs to resources in a way that optimizes their use.

Up until 2008, Hadoop supported a single scheduler where jobs were submitted to a queue, and the Hadoop infrastructure simply executed them in the order of receipt (FIFO). But recent development efforts have led to the making of pluggable schedulers such as Fair scheduler and Capacity Schedulers. This allows use of new scheduling algorithms to help optimize jobs that have specific characteristics. With this change, Hadoop is now a multi-user data warehouse that supports a variety of different types of processing jobs, with a pluggable scheduler framework providing greater control. This framework allows optimal use of a Hadoop cluster over a varied set of workloads.

The Fair scheduler works on the principle that resources must be assigned to jobs such that on average over time, each job gets an equal share of the available resources. The result is that jobs that require less time are able to access the CPU and finish intermixed with the execution of jobs that require more time to execute.

The Capacity scheduler was defined for large clusters, which may have multiple, independent consumers and target applications. Capacity scheduling provides greater control as well as the ability to provide a minimum capacity guarantee and share excess capacity among users.

Following a strict queuing order hurts data locality and forces a job with no local data to be scheduled. We can address the problem through a simple technique called delay scheduling. When a node requests a task, if the head-of-line job cannot launch a local task, we skip it and look at subsequent jobs.

1.3 Statement of the Problem

Hadoop uses FIFO as its default scheduler, which measures a job's importance based on when it was submitted. A comparative study of the Hadoop default scheduler with Fair Scheduler and Capacity Scheduler is done and the results are analyzed to show which scheduler is best suited to handle different type’s workloads.

Optimization of the existing Hadoop Scheduler using data-locality aware scheduling policy is done for a stream of Hadoop jobs continuously submitted to a cloud cluster. Delay scheduling policy is used to optimize the Fair scheduler which overcomes the data-locality problems.

1.4 Objectives of the Project

- Conduct an exhaustive comparative study of the Hadoop default scheduler with Fair scheduler and Capacity Scheduler for different data-intensive applications under different workload conditions.
- Catalogue the results and analyze them to determine which scheduling policy is better for data-intensive applications.

- Aim to optimize the existing Hadoop scheduler and the Fair scheduler so as to achieve better real-time efficiency of scheduling MapReduce jobs by overcoming data-locality problems.

1.5 Current Scope

MapReduce applications are now becoming increasingly latency-sensitive, operating under demanding workloads that require fast response times for data-intensive computations under high data rates. These include online log processing, processing of geographical data, traffic simulation, Google translator systems, personalized recommendations, advertisement placement, social network analysis, real-time web indexing, processing of bioinformatics and continuous web data analysis.

Hadoop framework supports the execution of MapReduce jobs. Efficient scheduling of these jobs is an important factor in performance. We consider alternative scheduling policies such as Fair and Capacity scheduling polices and analyze their performance on a cluster of three nodes. The problems in data locality are addressed by using delay scheduling policy.

Chapter 2

Literature Survey

2.1 Cluster Computing

2.1.1 Introduction

The needs and expectations of modern-day applications are changing in the sense that they not only need computing resources but also the ability to remain available to service user requests almost constantly. These needs and expectations of today's applications result in challenging research and development efforts in both the areas of computer hardware and software. As applications evolve they inevitably consume more and more computing resources. To some extent we can overcome these limitations by creating faster processors and install larger memories. But future improvements are constrained by a number of factors, including physical ones, such as the speed of light and the constraints imposed by various thermodynamic laws, as well as financial ones, such as the huge investment needed to fabricate new processors and integrated circuits.

The obvious solution to overcoming these problems is to connect multiple processors and systems together and coordinate their efforts. The resulting systems are popularly known as parallel computers and they allow the sharing of a computational task among multiple processors. Parallel supercomputers have been in the mainstream of high-performance computing. However, their popularity is waning. The reasons for this decline are many, but include being expensive to purchase and run, potentially difficult to program, slow to evolve in the face of emerging hardware technologies, and difficult to upgrade without, generally, replacing the whole system. The decline of the dedicated parallel supercomputer has been compounded by the emergence of commodity-off-the-shelf clusters of PCs and workstations.

The emergence of cluster platforms was driven by a number of academic projects, such as Beowulf, Berkeley NOW, and HPVM [13]. These projects helped to prove the advantage of clusters over other traditional platforms. Some of these advantages included, low-entry costs to access supercomputing-level performance, the ability to

track technologies, an incrementally upgradeable system, an open source development platform, and not being locked into particular vendor products. Today, the overwhelming price/performance advantage of this type of platform over other proprietary ones, as well as the other key benefits mentioned earlier, means that clusters have infiltrated not only the traditional science and engineering marketplaces for research and development, but also the huge commercial marketplaces of commerce and industry. It should be noted that this class of machine is not only being used as for high-performance computation, but increasingly as a platform to provide highly available services, for applications such Web and database servers [13].

A cluster is a type of parallel or distributed computer system, which consists of a collection of inter-connected stand-alone computers working together as a single integrated computing resource. The components of a cluster are usually connected to each other through fast local area networks, each node running its own instance of an operating system. Clusters are usually deployed to improve performance and availability over that of a single computer, while typically being much more cost-effective than single computers of comparable speed or availability. Computer clusters have a wide range of applicability and deployment, ranging from small business clusters with a handful of nodes to some of the fastest supercomputers in the world such as the K computer. The key components of a cluster include, multiple standalone computers (PCs, Workstations, or SMPs), an operating systems, a high performance interconnect, communication software, middleware, and applications [13].

2.1.2 Flavors of Cluster

Clusters come in the following major flavors depending on their purpose [12]:

1. *High Performance Computing Flavor*. An example is a Beowulf. The purpose is to aggregate computing power across nodes to solve a problem faster. For example, high performance scientific computing typically spreads portions of the computation across the nodes of the cluster and uses message passing to communicate between the portions.
2. *High Throughput Computing Flavor.* These clusters harness the ever-growing power of desktop computing resources while protecting the rights and needs of their

interactive users. These systems are a logical extension of the batch job environments on old mainframes. Eg: Condor.

3. *High Availability Computing Flavor*. These clusters are designed to provide high availability of service. Many vendors provide high availability (HA) systems for the commercial world. Most of these systems use redundancy of hardware and software components to eliminate single points of failure. A typical system consists of two nodes, with mirrored disks, duplicate switches, duplicate I/O devices and multiple network paths.
4. *High Performance Service Flavor*. A cluster of nodes is used to handle a high demand on a web service, mail service, data mining service or other service. Typically, a request spawns a thread or process on another node.

2.1.3 Benefits of Clusters

- Clusters allow trickle-up: hardware and software technologies that were developed for broad application to mainstream commercial and consumer markets can also serve in the arena of high performance computing. It was this aspect of clusters that initially made them possible and triggered the first wave of activity in the field.
- Clusters permit a flexibility of configuration not ordinarily encountered through conventional MPP systems. Number of nodes, memory capacity per node, number of processors per node, and interconnect topology are all parameters of system structure that may be specified in fine detail on a per system basis without incurring additional cost due to custom configurability.
- Further, system structure may easily be modified or augmented over time as need and opportunity dictates without the loss of prior investment. This expanded control over system structure not only benefits the end user but the system vendor as well, yielding a wide array of system capabilities and cost tradeoffs to better meet customer demands.
- Clusters also permit rapid response to technology improvements. As new devices including processors, memory, disks, and networks become available, they are most likely to be integrated in to desktop or server nodes most quickly allowing clusters to be the first class of parallel systems to benefit from such

advances. Clusters are best able to track technology improvements and respond most rapidly to new component offerings [12].

2.1.4 Hardware Components

The key components comprising a commodity cluster are:

- The nodes performing the computing
- The dedicated interconnection network providing the data communication among the nodes.

Significant advances for both have been accomplished over the last half decade. While originally clusters were almost always assembled at the user site by staff local to the user organization, now increasingly clusters are being delivered by vendors as turnkey solutions to user specifications [12]. Where once such systems were packaged in conventional “tower” cases, now manufacturers are providing slim, rack mounted, nodes to deliver more capacity per unit floor area.

Cluster Node Hardware

A node of a cluster provides the system computing and data storage capability. It is differentiated from nodes of fully integrated systems such as an MPP in that it is derived from fully operational standalone computing subsystems that are typically marketed as desktop or server systems.

Cluster Network Hardware

Commodity clusters are made possible only because of the availability of adequate inter-node communication network technology. Interconnect networks enable data packets to be transferred between logical elements distributed among a set of separate processor nodes within a cluster through a combination of hardware and software support. Clusters incorporate one or more dedicated networks to support message packet communication within the distributed system.

2.1.5 Software Components

While the rapid advances in hardware capability have propelled commodity clusters to the forefront of next generation systems, equally important has been the evolving capability and maturity of the support software systems and tools. The software components that comprise the environment of a commodity cluster may be described in two major categories:

- *Programming tools*: Programming tools provide languages, libraries, and performance and correctness debuggers to construct parallel application programs.
- *Resource management system software*: Resource management software relates to initial installation, administration, and scheduling and allocation of both hardware and software components as applied to user workloads.

2.1.6 Application Areas for Clusters

- Web serving
- Audio processing (voice based email)
- Data mining
- Network simulation
- Image processing.

2.2 MapReduce

MapReduce is a framework for processing highly distributed problems across huge datasets using a large number of computers (nodes), collectively referred to as a cluster (if all nodes use the same hardware) or a grid (if the nodes use different hardware). Computational processing can occur on data stored either in a file system (unstructured) or in a database (structured) [2].

Characteristics of Map Reduce:

- Data locality consideration is important in MapReduce for efficient processing. However, MapReduce frameworks do not preserve data locality in consecutive operations due to its inherent natures.

- Map/Reduce does any filtering and/or transformations while each mapper is reading input data split by InputFormat.
- It has a simple API.
- It does automatic parallelization and distribution of data.
- It incurs not only intensive communication cost but also unnecessary processing cost.

A MapReduce job is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information. Hadoop runs the job by dividing it into tasks, of which there are two types: map tasks and reduce tasks.

- "Map" step: The master node takes the input, partitions it up into smaller sub-problems, and distributes them to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes the smaller problem, and passes the answer back to its master node.
- "Reduce" step: The master node then collects the answers to all the sub-problems and combines them in some way to form the output – the answer to the problem it was originally trying to solve.

The Map and Reduce functions of MapReduce are both defined with respect to data structured in (key, value) pairs. Map takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

Map (k1, v1) → list (k2, v2)

The Map function is applied in parallel to every item in the input dataset. This produces a list of (k2, v2) pairs for each call. After that, the MapReduce framework collects all pairs with the same key from all lists and groups them together, thus creating one group for each one of the different generated keys. The Reduce function is then applied in parallel to each group, which in turn produces a collection of values in the same domain:

Reduce (k2, list (v2)) → list (v3)

Each Reduce call typically produces either one value v3 or an empty return, though one call is allowed to return more than one value. The returns of all calls are collected as the desired result list.

There are two types of nodes that control the job execution process: a jobtracker and a number of tasktrackers [2]. The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job. If a task fails, the jobtracker can reschedule it on a different tasktracker. Hadoop divides the input to a MapReduce job into fixed-size pieces called input splits, or just splits. Hadoop creates one map task for each split, which runs the user defined map function for each record in the split [6].

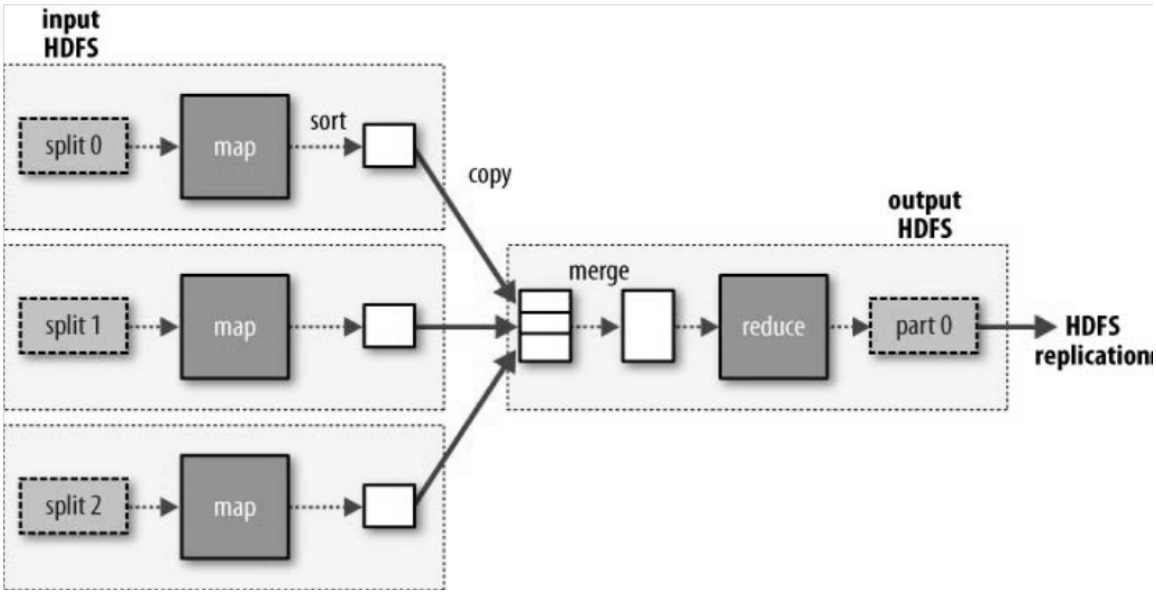


Fig 2.1: MapReduce data flow with a single reduce task

Having many splits means the time taken to process each split is small compared to the time to process the whole input. So by processing the splits in parallel, the processing is better load-balanced if the splits are small, since a faster machine will be able to process proportionally more splits over the course of the job than a slower machine. Even if the machines are identical, failed processes or other jobs running concurrently make load balancing desirable, and the quality of the load balancing increases as the splits become more

fine-grained. On the other hand, if splits are too small, then the overhead of managing the splits and of map task creation begins to dominate the total job execution time. For most jobs, a good split size tends to be the size of a HDFS block, 64 MB by default, although this can be changed for the cluster (for all newly created files), or specified when each file is created.

Hadoop does its best to run the map task on a node where the input data resides in HDFS. This is called the data locality optimization. The optimal split size is the same as the block size: it is the largest size of input that can be guaranteed to be stored on a single node. If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks, so some of the split would have to be transferred across the network to the node running the map task, which is clearly less efficient than running the whole map task using local data.

Map tasks write their output to local disk, not to HDFS. Map output is intermediate output: it's processed by reduce tasks to produce the final output, and once the job is complete the map output can be thrown away. So storing it in HDFS, with replication, would be unnecessary. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to recreate the map output.

Reduce tasks don't have the advantage of data locality—the input to a single reduce task is normally the output from all mappers. The output of reduce task is normally stored in HDFS for reliability. For each HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes. Thus, writing the reduce output does consume network bandwidth, but only as much as a normal HDFS write pipeline consumes.

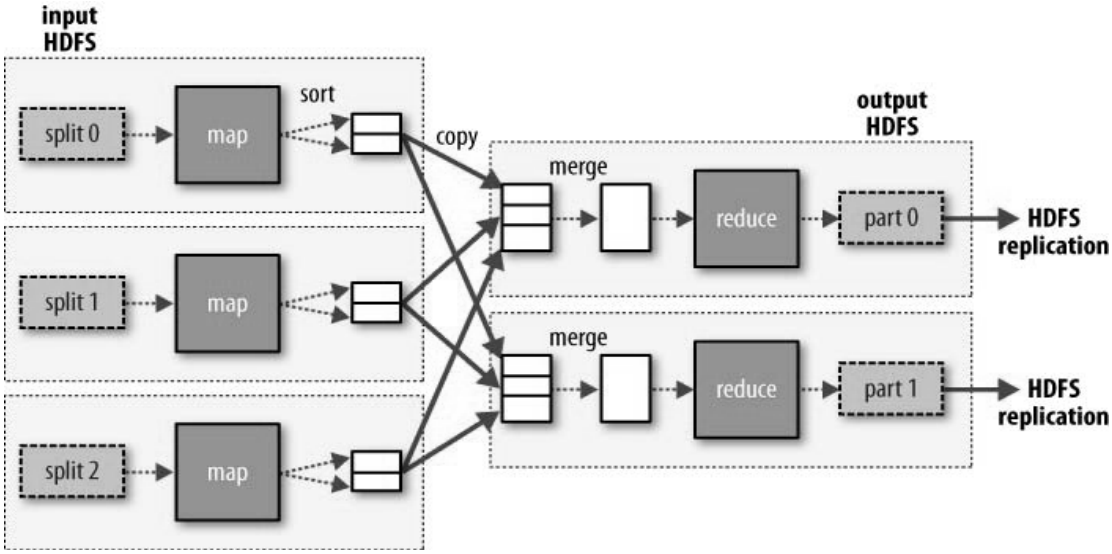


Fig 2.2: MapReduce data flow with multiple reduce tasks

When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for every key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner—which buckets keys using a hash function—works very well.

Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks. Hadoop allows the user to specify a combiner function to be run on the map output—the combiner function's output forms the input to the reduce function. Since the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call it for a particular map output record, if at all. In other words, calling the combiner function zero, one, or many times should produce the same output from the reducer.

The parallelism offered by MapReduce provides the possibility of recovering from partial failure of servers or storage during the operation: if one mapper or reducer fails, the work can be rescheduled – assuming the input data is still available. Thus the MapReduce framework transforms a list of (key, value) pairs into a list of values.

2.3 Hadoop

Hadoop was created by Doug Cutting, the creator of Apache Lucene, the widely used text search library [6]. The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using a simple programming model. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

Hadoop provides a distributed file system and a framework for the analysis and transformation of very large data sets using the MapReduce paradigm. An important characteristic of Hadoop is the partitioning of data and computation across many (thousands) of hosts, and executing application computations in parallel close to their data.

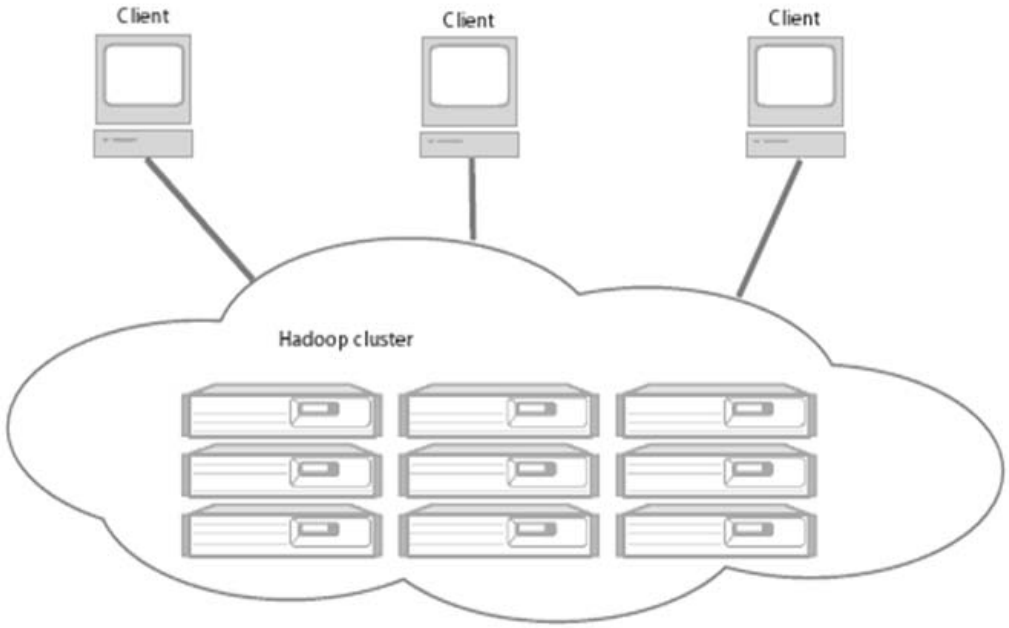


Fig 2.3: Basics of Hadoop

Fig 2.3 illustrates how one interacts with a Hadoop cluster. As you can see, a Hadoop cluster is a set of commodity machines networked together in one location. Data storage and processing all occur within this “cloud” of machines. Different users can submit computing

"jobs" to Hadoop from individual clients, which can be their own desktop machines in remote locations from the Hadoop cluster.

A Hadoop cluster scales computation capacity, storage capacity and IO bandwidth by simply adding commodity servers. Hadoop is a collection of related subprojects that fall under the umbrella of infrastructure for distributed computing [6]. They are as shown below:

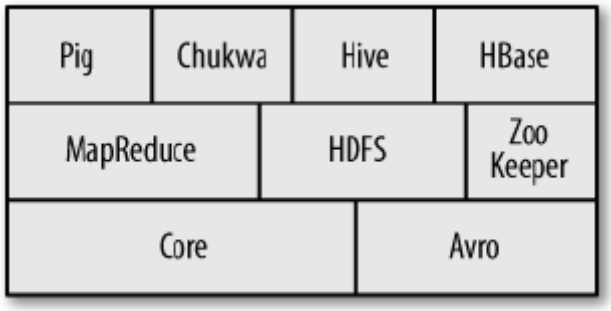


Fig 2.4: Hadoop Subprojects

Pig	Dataflow language and parallel execution framework
Chukwa	System for collecting management data
Hive	Data warehouse infrastructure
HBase	Column-oriented table service
MapReduce	Distributed computation framework
HDFS	Distributed file system
ZooKeeper	Distributed coordination service
Avro	Data serialization system

Table 2.1: Description of Hadoop Subprojects

Hadoop is an Apache project; all components are available via the Apache open source license. Yahoo! has developed and contributed to 80% of the core of Hadoop (HDFS and MapReduce). HBase was originally developed at Powerset, now a department at Microsoft. Hive was originated and develdeveloped at Facebook. Pig, ZooKeeper , and Chukwa were originated and developed at Yahoo! Avro was originated at Yahoo! and is being co-developed with Cloudera [1][8].

The key distinctions of Hadoop are that it is

- Accessible—Hadoop runs on large clusters of commodity machines or on cloud computing services such as Amazon's Elastic Compute Cloud (EC2).
- Robust—Because it is intended to run on commodity hardware, Hadoop is architected with the assumption of frequent hardware malfunctions. It can gracefully handle most such failures.
- Scalable—Hadoop scales linearly to handle larger data by adding more nodes to the cluster.
- Simple—Hadoop allows users to quickly write efficient parallel code.

Hadoop's accessibility and simplicity give it an edge over writing and running large distributed programs. Its robustness and scalability make it suitable for even the most demanding jobs at Yahoo and Facebook. These features make Hadoop popular in both academia and industry.

In Hadoop the data set will be divided into smaller (typically 64 MB) blocks that are spread among many machines in the cluster via the Hadoop Distributed File System (HDFS). With a modest degree of replication, the cluster machines can read the data set in parallel and provide a much higher throughput.

2.4 Hadoop Distributed File System (HDFS)

HDFS was originally built as infrastructure for the Apache Nutch web search engine project. HDFS is part of the Apache Hadoop project, which is part of the Apache Lucene project. HDFS is the file system component of Hadoop. The interface to HDFS is patterned after the UNIX file system.

The Hadoop Distributed File System (HDFS) is designed to run on commodity hardware, store very large data sets reliably, and to stream those data sets at high bandwidth to user applications. In a large cluster, thousands of servers both host directly attached storage and execute user application tasks. By distributing storage and computation across many servers, the resource can grow with demand while remaining economical at every size.

HDFS features:

- Highly fault-tolerant.
- Designed to be deployed on low-cost hardware.
- Provides high throughput access to application data.
- Suitable for applications that have large data sets.
- Relaxes a few POSIX requirements to enable streaming access to file system data.
- Stores file system metadata and application data separately.
- All servers are fully connected and communicate with each other using TCP-based protocols [10].

As in other distributed file systems, like PVFS, Lustre and GFS, HDFS stores metadata on a dedicated server, called the NameNode. Application data are stored on other servers called DataNodes. The file content is replicated on multiple DataNodes for reliability. While ensuring data durability, this strategy has the added advantage that data transfer bandwidth is multiplied, and there are more opportunities for locating computation near the needed data.

NameNode and DataNode

The HDFS namespace is a hierarchy of files and directories [8]. Files and directories are represented on the NameNode by inodes, with record attributes like permissions, modification and access times, namespace and disk space quotas. The file content is split into large blocks and each block of the file is independently replicated at multiple DataNodes. The NameNode maintains the namespace tree and the mapping of file blocks to DataNodes.

HDFS keeps the entire namespace in RAM. The inode data and the list of blocks belonging to each file comprise the metadata of the name system called the image. The persistent record of the image stored in the local host's native files system is called a checkpoint. The NameNode also stores the modification log of the image called the journal in the local host's native file system. For improved durability, redundant copies of the checkpoint and journal can be made at other servers. During restarts the NameNode restores the namespace by reading the namespace and replaying the journal.

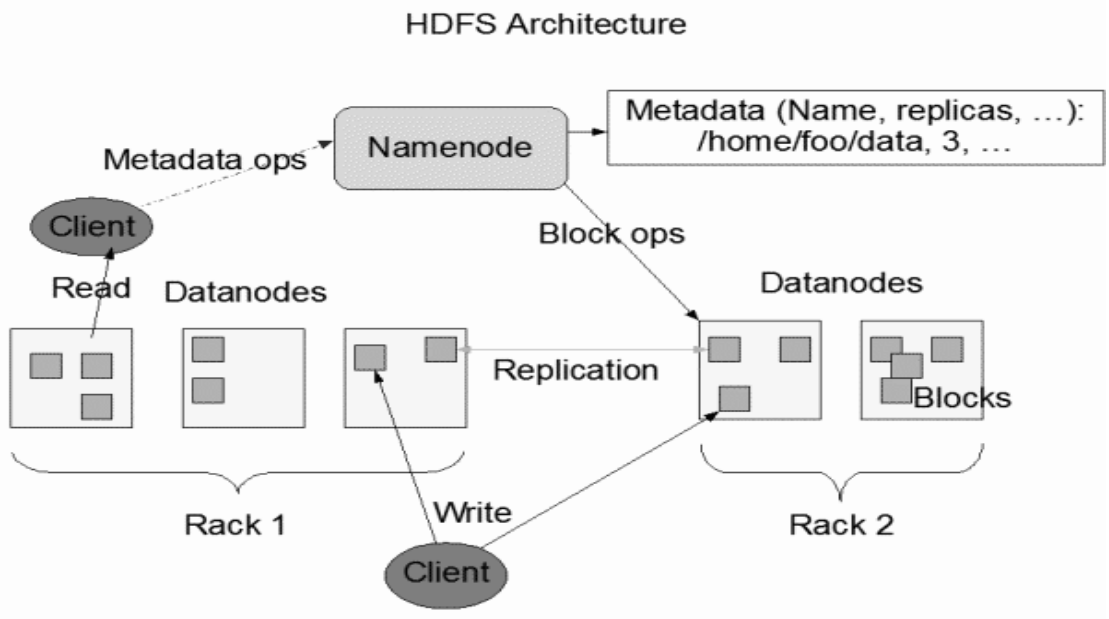


Fig 2.5: HDFS Architecture

Each block replica on a DataNode is represented by two files in the local host's native file system. The first file contains the data itself and the second file is block's metadata including checksums for the block data and the block's generation stamp. The size of the data file equals the actual length of the block and does not require extra space to round it up to the nominal block size as in traditional file systems. Thus, if a block is half full it needs only half of the space of the full block on the local drive.

During startup each DataNode connects to the NameNode and performs a handshake. The purpose of the handshake is to verify the namespace ID and the software version of the DataNode. If either does not match that of the NameNode the DataNode automatically shuts down. After the handshake the DataNode registers with the NameNode. DataNodes persistently store their unique storage IDs. The storage ID is an internal identifier of the DataNode, which makes it recognizable even if it is restarted with a different IP address or port.

A DataNode identifies block replicas in its possession to the NameNode by sending a block report. A block report contains the block id, the generation stamp and the length for each block replica the server hosts. Subsequent block reports are sent every hour and provide

the NameNode with an up-to date view of where block replicas are located on the cluster. During normal operation DataNodes send heartbeats to the NameNode to confirm that the DataNode is operating and the block replicas it hosts are available. The default heartbeat interval is three seconds. If the NameNode does not receive a heartbeat from a DataNode in ten minutes the NameNode considers the DataNode to be out of service and the block replicas hosted by that DataNode to be unavailable.

The NameNode does not directly call DataNodes. It uses replies to heartbeats to send instructions to the DataNodes. The instructions include commands to:

- replicate blocks to other nodes;
- remove local block replicas;
- re-register or to shut down the node;
- Send an immediate block report.

These commands are important for maintaining the overall system integrity and therefore it is critical to keep heartbeats frequent even on big clusters. The NameNode can process thousands of heartbeats per second without affecting other NameNode operations.

HDFS Client

User applications access the file system using the HDFS client, a code library that exports the HDFS file system interface. 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.

When an application reads a file, the HDFS client first asks the NameNode for the list of DataNodes that host replicas of the blocks of the file. It then contacts a DataNode directly and requests the transfer of the desired block. When a client writes, it first asks the NameNode to choose DataNodes to host replicas of the first block of the file. The client organizes a pipeline from node-to-node and sends the data. When the first block is filled, the client requests new DataNodes to be chosen to host replicas of the next block. A new pipeline is organized, and the client sends the further bytes of the file.

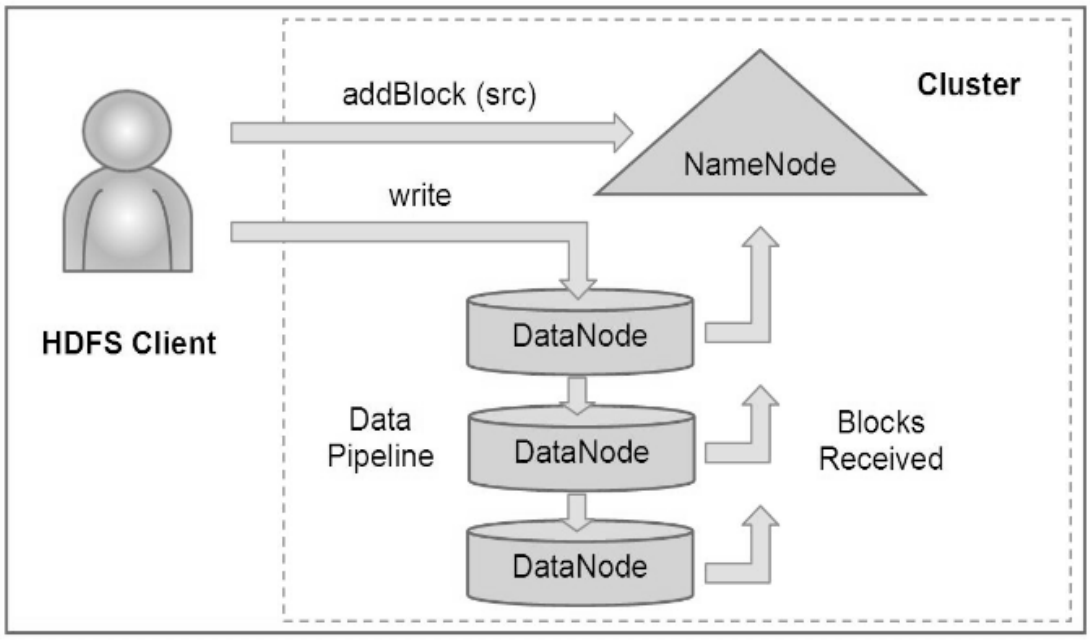


Fig 2.6: Interaction between the NameNode and the DataNodes

Unlike conventional file systems, HDFS provides an API that exposes the locations of a file blocks. This allows applications like the MapReduce framework to schedule a task to where the data are located, thus improving the read performance. It also allows an application to set the replication factor of a file. By default a file's replication factor is three. For critical files or files which are accessed very often, having a higher replication factor improves their tolerance against faults and increase their read bandwidth.

2.5 Job Scheduling

Job Scheduling plays an important role in the execution of data-intensive applications on cloud infrastructures. Choosing the appropriate job scheduling policy depends on the various criteria governing the application. Some applications want more real-time scheduling policies while some prefer low cost scheduling policies.

Current cloud infrastructures such as Microsoft Azure, Amazon EC2, Blue Cloud, and VCloud employ various scheduling policies depending upon their requirements.

VCloud uses Priority Scheduling policy to manage the jobs. Jobs are assigned priorities based on some criteria such as number of resources required, deadlines, complexity, etc. The

scheduler then arranges the jobs in the ready queue in order of their priority. Lower priority processes get interrupted by incoming higher priority processes.

Features of this scheduling policy are:

- Overhead is neither minimal, nor is it significant.
- Priority Scheduling has no particular advantage in terms of throughput over FIFO scheduling.
- Waiting time and response time depend on the priority of the process. Higher priority processes have smaller waiting and response times.
- Deadlines can be met by giving processes with deadlines a higher priority.

IBM'S BlueCloud uses the Hadoop parallel workload scheduling. By default Hadoop uses FIFO, and optional 5 scheduling priorities to schedule jobs from a work queue. Using FIFO scheduler runs the risk of blocking a small or critical job with an enormous ad-hoc jobs. In version 0.19 and above, the job scheduler was refactored, and added the ability to use an alternate scheduler such as the Fair scheduler or the Capacity scheduler.

Amazon EC2 has been the leading cloud provider for many years. It uses the Hadoop framework. As discussed above the default scheduling policy of Hadoop is the FIFO scheduling policy. Other schedulers such as Capacity and Fair are also used.

2.5.1 Fair Scheduler

2.5.1.1 Introduction

Fair scheduling is a method of assigning resources to jobs such that all jobs get, on average, an equal share of resources over time [5]. 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. Unlike the default Hadoop scheduler, which forms a queue of jobs, this lets short jobs finish in reasonable time while not starving long jobs. It is also a reasonable way to share a cluster between a number of 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.

The scheduler actually organizes jobs further into "pools", and shares resources fairly between these pools. By default, there is a separate pool for each user, so that each user gets the same share of the cluster no matter how many jobs they submit. However, it is also possible to set a job's pool based on the user's Unix group or any other jobconf property, such as the queue name property used by Capacity Scheduler. 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 the config file.

In addition to providing fair sharing, the Fair Scheduler allows assigning guaranteed minimum shares to pools [5], 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 config file. 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. Limiting the jobs does not cause any subsequently submitted jobs to fail, only to wait in the scheduler’s queue until some of the user's earlier jobs finish. Jobs to run from each user/pool are chosen in order of priority and then submit time, as in the default FIFO scheduler in Hadoop.

2.5.1.2 Fair Scheduler Goals

The Fair Scheduler was designed with four main goals:

- Run small jobs quickly even if they are sharing a cluster with large jobs. Unlike Hadoop’s built-in FIFO scheduler, fair scheduling lets small jobs make progress even if a large job is running, without starving the large job.
- Provide guaranteed service levels to “production” jobs, to let them run alongside experimental jobs in a shared cluster.

- Be simple to administer and configure. The scheduler should do something reasonable "out of the box," and users should only need to configure it as they discover that they want to use more advanced features.
- Support reconfiguration at runtime, without requiring a cluster restart.

2.5.1.3 Scheduler Features

Pools

The Fair Scheduler groups jobs into "pools" and performs fair sharing between these pools [11]. Each pool can use either FIFO or fair sharing to schedule jobs internal to the pool. The pool that a job is placed in is determined by a JobConf property, the "pool name property". By default, this is user.name, so that there is one pool per user. However, different properties can be used, e.g. group.name to have one pool per Unix group. The mapred-site.xml snippet below shows how to do this:

```
<property>
<name>mapred.fairscheduler.poolnameproperty</name>
<value>pool.name</value>
</property>

<property>
<name>pool.name</name>
<value>${user.name}</value>
</property>
```

Minimum Shares

Normally, active pools (those that contain jobs) will get equal shares of the map and reduce task slots in the cluster. However, it is also possible to set a minimum share of map and reduce slots on a given pool, which is a number of slots that it will always get when it is active, even if its fair share would be below this number. This is useful for guaranteeing that production jobs get a certain desired level of service when sharing a cluster with non-production jobs. When a pool is inactive (contains no jobs), its minimum share is not "reserved" for it – the slots are split up among the other pools.

Minimum shares have three effects:

- The pool's fair share will always be at least as large as its minimum share. Slots are taken from the share of other pools to achieve this. The only exception is if the minimum shares of the active pools add up to more than the total number of slots in the cluster; in this case, each pool's share will be scaled down proportionally.
- Pools whose running task count is below their minimum share get assigned slots first when slots are available.
- It is possible to set a preemption timeout on the pool after which, if it has not received enough task slots to meet its minimum share, it is allowed to kill tasks in other jobs to meet its share. Minimum shares with preemption timeouts thus act like SLAs.

Preemption

As mentioned above, the scheduler may kill tasks from a job in one pool in order to meet the minimum share of another pool. This is called preemption, although this usage of the word is somewhat strange given the normal definition of preemption as pausing; really it is the job that gets preempted, while the task gets killed. The feature explained above is called min share preemption. In addition, the scheduler supports fair share preemption, to kill tasks when a pool's fair share is not being met. Fair share preemption is much more conservative than min share preemption, because pools without min shares are expected to be non-production jobs where some amount of unfairness is tolerable. In particular, fair share preemption activates if a pool has been below half of its fair share for a configurable fair share preemption timeout, which is recommended to be set fairly high (e.g. 10 minutes).

In both types of preemption, the scheduler kills the most recently launched tasks from over-scheduled pools, to minimize the amount of computation wasted by preemption.

Running Job Limits

The fair scheduler can limit the number of concurrently running jobs from each user and from each pool. This is useful for limiting the amount of intermediate data

generated on the cluster. The jobs that will run are chosen in order of submit time and priority. Jobs submitted beyond the limit wait for one of the running jobs to finish.

Job Priorities

Within a pool, job priorities can be used to control the scheduling of jobs, whether the pool's internal scheduling mode is FIFO or fair sharing:

- In FIFO pools, jobs are ordered first by priority and then by submit time, as in Hadoop's default scheduler.
- In fair sharing pools, job priorities are used as weights to control how much share a job gets. The normal priority corresponds to a weight of 1.0, and each level gives 2x more weight. For example, a high-priority job gets a weight of 2.0, and will therefore get 2x the share of a normal-priority job.

Pool Weights

Pools can be given weights to achieve unequal sharing of the cluster. For example, a pool with weight 2.0 gets 2x the share of a pool with weight 1.0.

Administration

The Fair Scheduler includes a web UI displaying the active pools and jobs and their fair shares, moving jobs between pools, and changing job priorities. In addition, the Fair Scheduler's allocation file (specifying min shares and preemption timeouts for the pools) is automatically reloaded if it is modified on disk, to allow runtime reconfiguration.

2.5.1.4 Working of the Fair Scheduler

All schedulers in Hadoop, including the Fair Scheduler, inherit from the TaskScheduler abstract class. This class provides access to a TaskTrackerManager – an interface to the JobTracker – as well as a Configuration instance. It also asks the scheduler to implement three abstract methods: the lifecycle methods start and terminate, and a method called assignTasks to launch tasks on a given TaskTracker.

Task assignment in Hadoop is reactive. TaskTrackers periodically send heartbeats to the JobTracker with their TaskTrackerStatus, which contains a list of running tasks, the number of slots on the node, and other information. The JobTracker then calls assignTasks on the scheduler to obtain tasks to launch. These are returned with the heartbeat response.

Apart from reacting to heartbeats through assignTasks, schedulers can also be notified when jobs have been submitted to the cluster, killed, or removed by adding listeners to the TaskTrackerManager. The Fair Scheduler sets up these listeners in its start method. An important role of the listeners is to initialize jobs that are submitted – until a job is initialized, it cannot launch tasks. The Fair Scheduler currently initializes all jobs right away, but it may also be desirable to hold off initializing jobs if too many are submitted to limit memory usage on the JobTracker.

Selection of tasks within a job is mostly done by the JobInProgress class, and not by individual schedulers. JobInProgress exposes two methods, obtainNewMapTask and obtainNewReduceTask, to launch a task of either type. Both methods may either return a Task object or null if the job does not wish to launch a task. If the node containing a map task failed, the job will wish to re-run it to rebuild its output for use in the reduce tasks. Schedulers may therefore need to poll multiple jobs until they find one with a task to run.

Finally, for map tasks, an important scheduling criterion is data locality: running the task on a node or rack that contains its input data. Normally, JobInProgress.obtainNewMapTask returns the "closest" map task to a given node. However, to give schedulers slightly more control over data locality, there is also a version of obtainNewMapTask that allow the scheduler to cap the level of non-locality allowed for the task. The Fair Scheduler uses this method with an algorithm called delay scheduling to optimize data locality.

Fair Scheduler Basics

At a high level, the Fair Scheduler uses hierarchical scheduling to assign tasks. First it selects a pool to assign a task to according to the fair sharing algorithm. Then it

asks the pool obtain a task. The pool chooses among its jobs according to its internal scheduling order (FIFO or fair sharing). In fact, because jobs might not have tasks to launch (obtainNew(Map|Reduce)Task can return null), the scheduler actually establishes an ordering on jobs and asks them for tasks in turn. Within a pool, jobs are sorted either by priority and start time (for FIFO) or by distance below fair share. If the first job in the ordering does not have a task to launch, the pool will ask the second, third, etc jobs. Pools themselves are sorted by distance below min share and fair share, so if the first pool does not have any jobs that can launch tasks, the second pool is asked, etc.

Apart from the assign tasks code path, the Fair Scheduler also has a periodic update thread that calls update every few seconds. This thread is responsible for recomputing fair shares to display them on the UI, checking whether jobs need to be preempted, and checking whether the allocations file has changed to reload pool allocations (through PoolManager).

The Schedulable Class

To allow the same fair sharing algorithm to be used both between pools and within a pool, the Fair Scheduler uses an abstract class called Schedulable to represent both pools and jobs. Its subclasses for these roles are PoolSchedulable and JobSchedulable. A Schedulable is responsible for three roles:

- It can be asked to obtain a task through assignTask. This may return null if the Schedulable has no tasks to launch.
- It can be queried for information about the pool/job to use in scheduling, such as:
  - Number of running tasks.
  - Demand (number of tasks the Schedulable wants to run; this is equal to number of running tasks + number of unlaunched tasks).
  - Min share assigned through config file.
  - Weight (for fair sharing).
  - Priority and start time (for FIFO scheduling).
- It can be assigned a fair share through setFairShare.

There are separate Schedulable for map and reduce tasks, to make it possible to use the same algorithm on both types of tasks.

Fair Sharing Algorithm

A simple way to achieve fair sharing is the following: whenever a slot is available, assign it to the pool that has the fewest running tasks. This will ensure that all pool get an equal number of slots, unless a pool's demand is less than its fair share, in which case the extra slots are divided evenly among the other pools. Two features of the Fair Scheduler complicate this algorithm a little:

- Pool weights mean that some pools should get more slots than others. For example, a pool with weight 2 should get 2x more slots than a pool with weight 1. This is accomplished by changing the scheduling rule to "assign the slot to the pool whose value of runningTasks/weight is smallest."
- Minimum shares mean that pools below their min share should get slots first. When we sort pools to choose which ones to schedule next, we place pools below their min share ahead of pools above their min share. We order the pools below their min share by how far they are below it as a percentage of the share.

This fair sharing algorithm is implemented in FairShareComparator in the SchedulingAlgorithms class. The comparator orders jobs by distance below min share and then by runningTasks/weight.

Preemption

To determine when to preempt tasks, the Fair Schedulers maintains two values for each PoolSchedulable: the last time when the pool was at its min share, and the last time when the pool was at half its fair share. These conditions are checked periodically by the update thread in FairScheduler.updatePreemptionVariables, using the methods isStarvedForMinShare and isStarvedForFairShare. These methods also take into account the demand of the pool, so that a pool is not counted as starving if its demand is below its min/fair share but is otherwise met.

When preempting tasks, the scheduler kills the most recently launched tasks from over scheduled pools. This minimizes the amount of computation wasted by preemption and ensures that all jobs can eventuallyfinish (it is as if the preempted jobs just never got their last few slots). The tasks are chosen and preempted in preemptTasks.

Fair Share Computation

The scheduling algorithm achieves fair shares without actually needing to compute pools' numerical shares beforehand. However, for preemption and for displaying shares in the Web UI, we want to know what a pool's fair share is even if the pool is not currently at its share. That is, we want to know how many slots the pool would get if we started with all slots being empty and ran the algorithm in until we filled them.

One way to compute these shares would be to simulate starting out with empty slots and calling assignTasks repeatedly until they filled, but this is expensive, be cause each scheduling decision takes O(numJobs) time and we need to make O(numSlots) decisions. To compute fair shares efficiently, the Fair Scheduler includes an algorithm based on binary search in SchedulingAlgorithms.computeFairShares. This algorithm is based on the following observation. If all slots had been assigned according to weighted fair sharing respecting pools' demands and min shares, then there would exist a ratio r such that:

- Pools whose demand di is less than rwi (where wi is the weight of the pool) are assigned di slots.
- Pools whose min share mi is more than rwi are assigned min(mi , di) slots.
- All other pools are assigned rwi slots.
- The pools' shares sum up to the total number of slots t.

The Fair Scheduler uses binary search to compute the correct r. We define a function f (r) as the number of slots that would be used for a given r if conditions 1-3 above were met, and then find a value of r that makes f (r) = t. More precisely, f (r) is defined as:

$$f(r) = \sum_i \min(d_i, \max(rw_i, m_i)).$$

Note that f (r) is increasing in r because every term of the sum is increasing, so the equation f (r) = t can be solved by binary search. We choose 0 as a lower bound of our binary search because with r = 0, only min shares are assigned. To compute an upper bound for the binary search, we try r = 1, 2, 4, 8… until we find a value large enough that either more than t slots are used or all pools' demands are met. The steps of the

algorithm are explained in detail in SchedulingAlgorithms.java. This algorithm runs in time O(NP), where N is the number of jobs/pools and P is the desired number of bits of precision in the computed values (number of iterations of binary search), which we've set to 25. It thus scales linearly in the number of jobs and pools.

Running Job Limits

Running job limits are implemented by marking jobs as not runnable if there are too many jobs submitted by the same user or pool. This is done in FairScheduler.updateRunnability. A job that is not runnable declares its demand as 0 and always returns null from assignTasks.

Locking Order

Fair Scheduler data structures can be touched by several threads. Most commonly, the JobTracker invokes assignTasks. This happens inside a block of code where the JobTracker has locked itself already. Therefore, to prevent deadlocks, we always ensure that if both the FairScheduler and the JobTracker must be locked, the JobTracker is lockedfirst. Other threads that can lock the FairScheduler include the update thread and the web UI.

Unit Tests

The Fair Scheduler contains extensive unit tests using mock TaskTrackerManager, JobInProgress, TaskInProgress, and Schedulable objects. Scheduler tests are in TestFairScheduler.java. The computeFairShares algorithm is tested separately in TestComputeFairShares.java. All tests use accelerated time via a fake Clock class.

2.5.1.5 Code Guide

The following table lists the key source files in the Fair Scheduler:

FILE	CONTENTS
FairScheduler.java	Scheduler entry point. Also contains update thread, and logic for preemption, delay scheduling, and running job limits.

Schedulable.java	Definition of the Schedulable class. Extended by *JobSchedulable* and *PoolSchedulable.*
SchedulingAlgorithms.java	Contains FIFO and fair sharing comparators, as well as the *computeFairShares* algorithm.
PoolManager.java	Reads pool properties from the allocation file and maintains a collection of Pool *objects*. Pools are created on demand.
Pool.java	Represents a pool and stores its map and reduce Schedulables.
FairSchedulerServlet.java	Implements the scheduler's web UI.
FairSchedulerEventLog.java	An easy-to-parse event log for debugging. Must be enabled through *mapred.fairscheduler.eventlog.enabled*. If enabled, logs are placed in *$HADOOP LOG DIR/fairscheduler*.
TaskSelector.java	A pluggable class responsible for picking tasks within a job. Currently, *DefaultTaskSelector* delegates to *JobInProgress*, but this would be a useful place to experiment with new algorithms for speculative execution and locality.
LoadManager.java	A pluggable class responsible for determining when to launch more tasks on a TaskTracker. Currently, *CapBasedLoadManager* uses slot counts, but this would be a useful place to experiment with scheduling based on machine load.
WeightAdjuster.java	A pluggable class responsible for setting job weights. An example, *NewJobWeightBooster*, is provided, which increases weight temporarily for new jobs.

Table 2.2: Key source files in the Fair Scheduler

2.5.2 Capacity Scheduler

2.5.2.1 Introduction

The Capacity Scheduler is designed to run Hadoop Map-Reduce as a shared, multi-tenant cluster in an operator-friendly manner while maximizing the throughput and the utilization of the cluster while running Map-Reduce applications [4].

Traditionally each organization has it own private set of compute resources that have sufficient capacity to meet the organization's SLA under peak or near peak conditions. This generally leads to poor average utilization and the overhead of managing multiple independent clusters, one per each organization. Sharing clusters between organizations is a cost-effective manner of running large Hadoop installations since this allows them to reap benefits of economies of scale without creating private clusters. However, organizations are concerned about sharing a cluster because they are worried about others using the resources that are critical for their SLAs.

The Capacity Scheduler is designed to allow sharing a large cluster while giving each organization a minimum capacity guarantee [4]. The central idea is that the available resources in the Hadoop Map-Reduce cluster are partitioned among multiple organizations that collectively fund the cluster based on computing needs. There is an added benefit that an organization can access any excess capacity not being used by others. This provides elasticity for the organizations in a cost-effective manner. Sharing clusters across organizations necessitates strong support for multi-tenancy since each organization must be guaranteed capacity and safe-guards to ensure the shared cluster is impervious to single rouge job or user. The Capacity Scheduler provides a stringent set of limits to ensure that a single job or user or queue cannot consume disproportionate amount of resources in the cluster. Also, the Job Tracker of the cluster, in particular, is a precious resource and the Capacity Scheduler provides limits on initialized/pending tasks and jobs from a single user and queue to ensure fairness and stability of the cluster. The primary abstraction provided by the Capacity Scheduler is the concept of queues. These queues are typically setup by administrators to reflect the economics of the shared cluster.

2.5.2.2 Capacity Scheduler Features

- Support for multiple queues, where a job is submitted to a queue.
- Queues are allocated a fraction of the capacity of the grid in the sense that a certain capacity of resources will be at their disposal. All jobs submitted to a queue will have access to the capacity allocated to the queue.
- Free resources can be allocated to any queue beyond it's capacity. When there is demand for these resources from queues running below capacity at a future point in time, as tasks scheduled on these resources complete, they will be assigned to jobs on queues running below the capacity.
- Queues optionally support job priorities (disabled by default).
- Within a queue, jobs with higher priority will have access to the queue's resources before jobs with lower priority. However, once a job is running, it will not be preempted for a higher priority job, though new tasks from the higher priority job will be preferentially scheduled.
- In order to prevent one or more users from monopolizing its resources, each queue enforces a limit on the percentage of resources allocated to a user at any given time, if there is competition for them.
- Support for memory-intensive jobs, wherein a job can optionally specify higher memory-requirements than the default, and the tasks of the job will only be run on Task Trackers that have enough memory to spare.

The capacity scheduler was designed to have a number of named queues and an allotted appropriate capacity for each of the queue. This architecture for the capacity scheduler was possible because a previous issue in Jira HADOOP-3444 – The architecture for the Hadoop Resource Manager (V1) made changes to the Job Tracker to handle queues, guaranteed capacities, user limits and the functionality of scheduling a task on a Task Tracker. The Job Tracker contains a new component, the Job Queue Manager (JQM), to handle queues of jobs. Job queues are backed up by disk based storage.

2.5.2.3 Working of the Capacity Scheduler

Terminology used in Capacity Scheduler:

- A queue has excess capacity if it does not have enough jobs (queued or running) to take up its guaranteed capacity. Excess capacity needs to be distributed among all the queues that have lesser or no capacity.
- Queues that have given up excess capacity to other queues are called low queues. Queues that are running on additional capacity are called high queues.

For each queue, the Job Tracker keeps track of the following:

- *Guaranteed capacity (GC)*: is the capacity guaranteed to the queue. This parameter is set through the configuration file. The sum of all GCs is equal to the grid capacity. Since we're handling Map and Reduce slots differently, we will have a GC for each, i.e., a GC-M for maps and a GC-R for reducers. The sum of all GC-Ms is equal to the sum of all map slots available in the Grid, and the sum of all GC-Rs is equal to the sum of all reduce slots available in the grid.
- *Allocated capacity (AC)*: is the current capacity of the queue. This can be higher or lower than the Gauranteed Capacity, because of excess capacity distribution. The sum of all ACs is equal to the grid capacity. As above, each queue will have a AC-M for maps and AC-R for reduces.
- *Timer for claiming containers*: is the number of seconds the queue can wait till it needs its capacity back. There will be separate timers for claiming map and reduce slots
- *Number of containers being used*, i.e., the number of running tasks associated with the queue (C-RUN). Each queue will have a C-RUN-M for maps and C-RUN-R for reduces.
- *The number of Map and Reduce containers* used by each user.

Periodically or based on some events, the Job Tracker looks at redistributing the capacity. This can result in excess capacity being given to queues that need them, and capacity being claimed by queues.

Algorithm to redistribute excess capacity

The Job Tracker will run the following algorithm to redistribute excess capacity for both Maps and Reduces.

- The Job Tracker checks each queue to see if it has excess capacity. A queue has excess capacity if the number of running tasks associated with the queue is less than the allocated capacity of the queue (i.e., if C-RUN < AC) and there are no jobs queued. (C-RUN can also be taken as the number of tasks required by the waiting jobs).
- The total excess capacity is the sum of excess capacities of each queue. If there is at least one queue with excess capacity, the Job Tracker figures out the queues that this capacity can be distributed to. These are queues that need capacity, where C-RUN = AC (i.e., the queue is running at max capacity) and there are queued jobs.
- The Job Tracker now calculates how much excess capacity to distribute to each queue that needs it. This can be done in many ways.
  - Distribute capacity in the ratio of each queues guaranteed capacity. So if queue Q1, Q2, and Q3 have guaranteed capacities of GC1, GC2, and GC3, and if Q3 has N containers of excess capacity, Q1 gets (GC1*N)/(GC1+GC2) additional capacity, while Q2 gets (GC2*N)/(GC1+GC2).
  - You could use some other ratio that uses the number of waiting jobs. The more waiting jobs a queue has, the more its share of excess capacity.
- For each queue that needs capacity, the Job Tracker increments its AC with the capacity it is allocated. At the same time, the Job Tracker appropriately decrements the AC of queues with excess capacity.

Algorithm to reclaim excess capacity

The algorithm below is in terms of tasks, which can be map or reduce tasks. It is the same for both. The Job Tracker will run the algorithm to reclaim excess capacity for both Maps and Reduces.

- The Job Tracker determines which queues are low queues (if AC < GC). If a low queue has a job waiting, then we need to reclaim its resources. Capacity to be reclaimed = GC-AC.

- Capacity is re-claimed from any of the high queues (where AC > GC).
- Job Tracker decrements the AC of the high queue from which capacity is to be claimed, and increments the AC of the low queue. The decremented AC of the high queue cannot go below its GC, so the low queue may get its capacity back from more than one queue.
- The Job Tracker also starts a timer for the low queue (this can be an actual timer, or just a count, perhaps representing seconds, which can be decremented by the Job Tracker periodically).
- If a timer goes off, the Job Tracker needs to instruct some high queue to kill some of their tasks. The following algorithm is used to decide, from which the capacity has to be regained to give it back to the low queue:
  - The candidates are those high queues which are running more tasks than they should be, i.e., where C-RUN > AC.
  - Among these queues, the Job Tracker can pick those that are using the most excess capacity (i.e. queues with higher values for (C-RUN - AC)/AC).

- The next question is 'How does a high queue decide which tasks to kill? '. The Job Tracker uses the following method for deciding which task to kill.
  - It is expensive to kill tasks, so we need to focus on getting better at deciding which tasks to kill. Ideally, tasks that have started recently or made the least progress are the ones that are to be killed. The same algorithm used to decide which tasks to speculatively run can be used for this as well.

Within a queue, a user's limit can dynamically change depending on how many users have submitted jobs. This needs to be handled in a way similar to how we handle excess capacity between queues.

Task Scheduling in the Capacity Scheduler

The Task scehduling includes the process that happens when the task tracker has a free map slot or a free reduce slot. When a Task Tracker has a free Map slot the following steps take place:

- Task Tracker contacts Job Tracker to give it a Map task to run.

- Job Tracker finds out which queue to approach first (among all queues that have capacity, i.e., where C-RUN-M < AC-M). This is done in either one of the following ways:
    - Round-robin, so every queue has the same chance to get a free container.
    - Job Tracker can pick the queue with the maximum unused capacity.

- Job Tracker needs to pick a job which can use the slot. This is done in one of the following ways:
    - If the Job Tracker has no running jobs from that queue, it gets one from the Job Queue Manager.
        - Job Tracker asks for the first Job in the selected queue, via the Job Queue Manager. If the job's user's limit is maxed out, the job is returned to the queue and Job Tracker asks for the next job. This continues until the Job Tracker finds a suitable job.
        - Otherwise, the Job Tracker has a list of users in the queue whose jobs it is running, and it can figure out which of these users are over their limit. It asks the Job Queue Manager for the first job in the queue whose user is not in a list of maxed-out users it provides.
    - If the Job Tracker already has a list of running jobs from the queue, it looks at each (in order of priority) till it finds one whose user's limit has not been exceeded and returns the job to the task Tracker.

- If there is no job in the queue that is eligible to run (the queue may have no queued jobs), the Job Tracker picks another queue using the same steps as above.
- The Job Tracker finds out which Map task from the job to run on the free Task Tracker using the algorithm as finding a locality match using the job's cache, then look for failed tasks.
- Job Tracker increments C-RUN-M and the number of Map containers used by the job's user. It then returns the task to the Task Tracker.

When a Task Tracker has a free Reduce slot, similar steps as to what happens with a free Map slot, except that:

- Since there is no preemption of jobs based on priorities, we will not have the situation where a job's Reducers are blocking containers as they're waiting for Maps to run and there are no Map slots to run.

When a task fails or completes: Job Tracker decrements C-RUN and the number of containers used by the user.

2.5.2.4 Code Guide

The following table lists the key source files in the Capacity

FILE	CONTENT
CapacitySchedulerConf.java	Class providing access to resource manager configuration. Resource manager configuration involves setting up queues, and defining various properties for the queues. These are typically read from a file called capacity-scheduler.xml that must be in the *classpath* of the application. The class provides APIs to get/set and reload the configuration for the queues.
CapacitySchedulerQueue.java	Keeps track of scheduling information for queues. This scheduling information is used to decide how to allocate tasks, redistribute capacity, etc.
CapacityTaskScheduler.java	Scheduler entry point. Extends TaskScheduler and contains the core logic of the scheduler.
JobInitializationPoller.java	This class asynchronously initializes jobs submitted to the Map/Reduce cluster running with CapacityTaskScheduler
JobQueuesManager.java	The class extends JobInProgressListener and maintains the jobs being managed in one or more queues
CapacitySchedulerServlet.java	Servlet for displaying capacity scheduler

	information, at JobTracker URL/scheduler.(Supported only in 1.0)
MemoryMatcher.java	Check if a TT has enough memory to run of task specified from this job.

Table 2.3: Key source files in the Capacity Scheduler

2.5.3 Delay Scheduling

Hadoop Fair Scheduler has two main goals:

- *Fair sharing*: divide resources using max-min fair sharing to achieve statistical multiplexing. For example, if two jobs are running, each should get half the resources and if a third job is launched, each job's share should be 33%.
- *Data locality:* place computations near their input data, to maximize system throughput.

To achieve the first goal (fair sharing), a scheduler must reallocate resources between jobs when the number of jobs changes. A key design question is what to do with tasks (units of work that make up a job) from running jobs when a new job is submitted, in order to give resources to the new job. At a high level, two approaches can be taken:

1. Kill running tasks to make room for the new job.
2. Wait for running tasks to finish.

Killing reallocates resources instantly and gives control over locality for the new jobs, but it has the serious disadvantage of wasting the work of killed tasks. Waiting, on the other hand, does not have this problem, but can negatively impact fairness, as a new job needs to wait for tasks to finish to achieve its share, and locality, as the new job may not have any input data on the nodes that free up.

However, a strict implementation of fair sharing compromises locality, because the job to be scheduled next according to fairness might not have data on the nodes that are currently free [7]. To resolve this problem, we relax fairness slightly through a simple algorithm called delay scheduling, in which a job waits for a limited amount of time for a scheduling opportunity on a node that has data for it.

Locality Problems with Fair Sharing

The main aspect of MapReduce that complicates scheduling is the need to place tasks near their input data. Locality increases throughput because network bandwidth in a large cluster is much lower than the total bandwidth of the cluster's disks. Running on a node that contains the data (node locality) is most efficient, but when this is not possible, running on the same rack (rack locality) is faster than running off-rack [7]. The two locality problems that arise with native fair sharing: head-of-line scheduling and sticky slots.

- *Head-of-line scheduling*

  The first locality problem occurs in small jobs (jobs that have small input files and hence have a small number of data blocks to read). The problem is that whenever a job reaches the head of the sorted list in Algorithm 1 (i.e. has the fewest running tasks), one of its tasks is launched on the next slot that becomes free, no matter which node this slot is on. If the head-of-line job is small, it is unlikely to have data on the node that is given to it. For example, a job with data on 10% of nodes will only achieve 10% locality.

- *Sticky Slots*

  A second locality problem, sticky slots, happens even with large jobs if fair sharing is used. There is a tendency for a job to be assigned the same slot repeatedly. For example, suppose that there are 10 jobs in a 100-node cluster with one slot per node, and that each job has 10 running tasks. Suppose job j finishes a task on node n. Node n now requests a new task. At this point, j has 9 running tasks while all the other jobs have 10. Therefore, slots are assigned on node n to job j again. Consequently, in steady state, jobs never leave their original slots. This leads to poor data locality because input files are striped across the cluster, so each job needs to run some tasks on each machine.

The problems presented happen because following a strict queuing order forces a job with no local data to be scheduled. We address them through a simple technique called delay scheduling. When a node requests a task, if the head-of-line job cannot launch a local task, we skip it and look at subsequent jobs. However, if a job has been skipped long enough, we start allowing it to launch non-local tasks, to avoid starvation. The key insight behind delay scheduling is that although the first slot we consider giving to a job is unlikely to have data for

it, tasks finish so quickly that some slot with data for it will free up in the next few seconds

In delay scheduling, we scan jobs in order given by queuing policy, picking first that is permitted to launch a task. The jobs must wait before being permitted to launch non-local tasks. There is an increase in a job's time waited when it is skipped

Features

- Scan jobs in order given by queuing policy, picking first that is permitted to launch a task and the jobs must wait before being permitted to launch non-local tasks
- Increase a job's time waited when it is skipped.
    - Delay scheduling works well under two conditions that sufficient fraction of tasks are short relative to jobs and there are many locations where a task can run.
        - Blocks replicated across nodes, multiple tasks/node.

Chapter 3

Software Requirements Specification

3.1 Introduction

3.1.1 Purpose

Hadoop uses FIFO as its default scheduler, which treats a job's importance relative to when it was submitted. A comparative study of the Hadoop default scheduler with Fair Scheduler and Capacity Scheduler is done and the results are analyzed to show which scheduler is best suited to handle data-intensive applications.

Optimization of the existing Hadoop Scheduler using multiprocessor scheduling techniques is done for a stream of Hadoop jobs continuously submitted to a cloud cluster, where each job has a number of tasks. Scheduling is to be done on the cluster such that particular real-time objectives are achieved.

3.1.2 Scope of the Project

MapReduce applications are latency-sensitive, operating under demanding workloads that require fast response times for data-intensive computations under high data rates. Hadoop framework supports the execution of such MapReduce applications. Efficient scheduling of the jobs is an important factor in performance. Alternative scheduling policies such as Fair and Capacity scheduling polices are considered and their performance is analyzed on a cluster of three nodes. The problems in data locality are addressed by using delay scheduling policy.

3.2 General Description

3.2.1 Project Perspective

In this project we conduct a comparative study on the Hadoop default scheduler and the Fair scheduler and Capacity scheduler. The data from the analysis is used to determine which scheduling policy works better under different workloads. Next we try to optimize the Hadoop default scheduler so as to schedule MapReduce jobs more

efficiently in order to confer to the real-time constraints such as data-locality awareness.

3.2.2 General Constraints

Installing Hadoop. Hadoop is an open source project and is freely available as compared to the propriety software that employ usage-based payment model.

- Needs Java sun 6 jdk
- Contention-free network
- Good processing speed so as to handle huge amounts of data.

3.2.3 Assumptions and Dependencies

- Availability of appropriate data sets

3.3 Specific Requirements

3.3.1 Functional Requirements

The following constitute the functional requirements which have to be fulfilled:

Comparative analysis of the Hadoop default scheduler with Fair and Capacity Schedulers

- *Ubuntu installation on computers constituting the cluster.*
- *Hadoop installation on all cluster computers:* The computers are first configured to work as a single node cluster and later the required configuration files are changed to create a cluster.
- *Running MapReduce applications on the cluster:* Different data-intensive applications are developed and are run on the cluster under different workload conditions.
- *Change the Hadoop default scheduler to the Fair scheduler on the cluster:* Run the same data-intensive applications on the cluster and log the timing data for each workload.
- *Change the Hadoop default scheduler to the Capacity scheduler on the cluster:* Run the same data-intensive applications on the cluster and log the timing data for each workload.

- *Conduct a comparative analysis on the results obtained:* Plot graphs using the obtained data and determine which algorithm is better under which working conditions.

Optimization of Hadoop default scheduler

- The Hadoop default scheduler FIFO is modified.
- Update the cluster status and task tracker details.
- Calculate the available Map and Reduce capacity.
- Using Load Factor, determine if a Map or Reduce task should be assigned
- For each of the available map slots:
  - Get the status of each of the running jobs
  - Obtain either a new local or rack local or non local task and add it to assigned tasks.
    - If map padding exceeds for local task, do not schedule more maps.
- Repeat the same for Reduce tasks.
- Only 1 non-local map/reduce task is allocated to avoid depriving another node of its local task.

Optimization of Fair Scheduler

- The fair scheduler is modified.
- A new enumerator to find the level for each of the jobs is created.
- Only one of the three levels are allowed - NODE, RACK, OTHER
- If no local job is present, then the task tracker waits and is not assigned any task.
- If the task tracker has waited for a time greater than the node delay and a rack local task is available for the given job, then a rack local task is assigned to it.
- If no rack local task is available, then the task tracker continues to wait.
- If the task tracker has waited for a time greater than the Rack delay, then any random map task is assigned to the task tracker.

3.3.2 Software Requirements

Supported Platforms:

- GNU/Linux is supported as a development and production platform. Hadoop has been demonstrated on GNU/Linux clusters with 2000 nodes.
- Win32 is supported as a development platform. Distributed operation has not been well tested on Win32, so it is not supported as a production platform.

Required software for Linux and Windows include:

- JavaTM 1.5.x, preferably from Sun, must be installed.
- Sun Java JDK 1.6.x
- SSH (Secure Shell) must be installed and sshd must be running to use the Hadoop scripts that manage remote Hadoop daemons.
- Hadoop 0.20.20/x

Additional requirements for Windows include:

- Cygwin - Required for shell support in addition to the required software above.

3.3.3 Hardware Requirements

- 2 quad core processors @ 2.5GHz or higher per node
- Router with more than 4 ports
- 50GB memory space
- Ethernet cables

3.4 Interface Requirements

3.4.1 User Interface

The basic implementation of the scheduling policy is in Java with the MapReduce Hadoop environment. Hence the basic user interface is through the command prompt. The first example we consider will be the word count problem. The input to the problem will be through a file. Thus the basic user interface includes

a map and reduce program, input to the MapReduce program through a text file. The output for the program will be redirected to another file. All these files are specified in the command to run the wordcount program.

Hadoop comes with several web interfaces which are by default (defined in conf/hadoop-default.xml) available at these locations:

- http://localhost:50030/ – web UI for MapReduce job tracker(s)
- http://localhost:50060/ – web UI for task tracker(s)
- http://localhost:50070/ – web UI for HDFS name node(s)

These web interfaces provide concise information about what's happening in your Hadoop cluster.

The *job tracker* web UI provides information about general job statistics of the Hadoop cluster, running/completed/failed jobs and a job history log file. It also gives access to the "local machine's" Hadoop log files (the machine on which the web UI is running on). By default, it's available at http://localhost:50030/.

The *task tracker* web UI shows you running and non-running tasks. It also gives access to the "local machine's" Hadoop log files. By default, it's available at http://localhost:50060/.

The *name node* web UI shows you a cluster summary including information about total/remaining capacity, live and dead nodes. Additionally, it allows you to browse the HDFS namespace and view the contents of its files in the web browser. It also gives access to the "local machine's" Hadoop log files. By default, it's available at http://localhost:50070/.

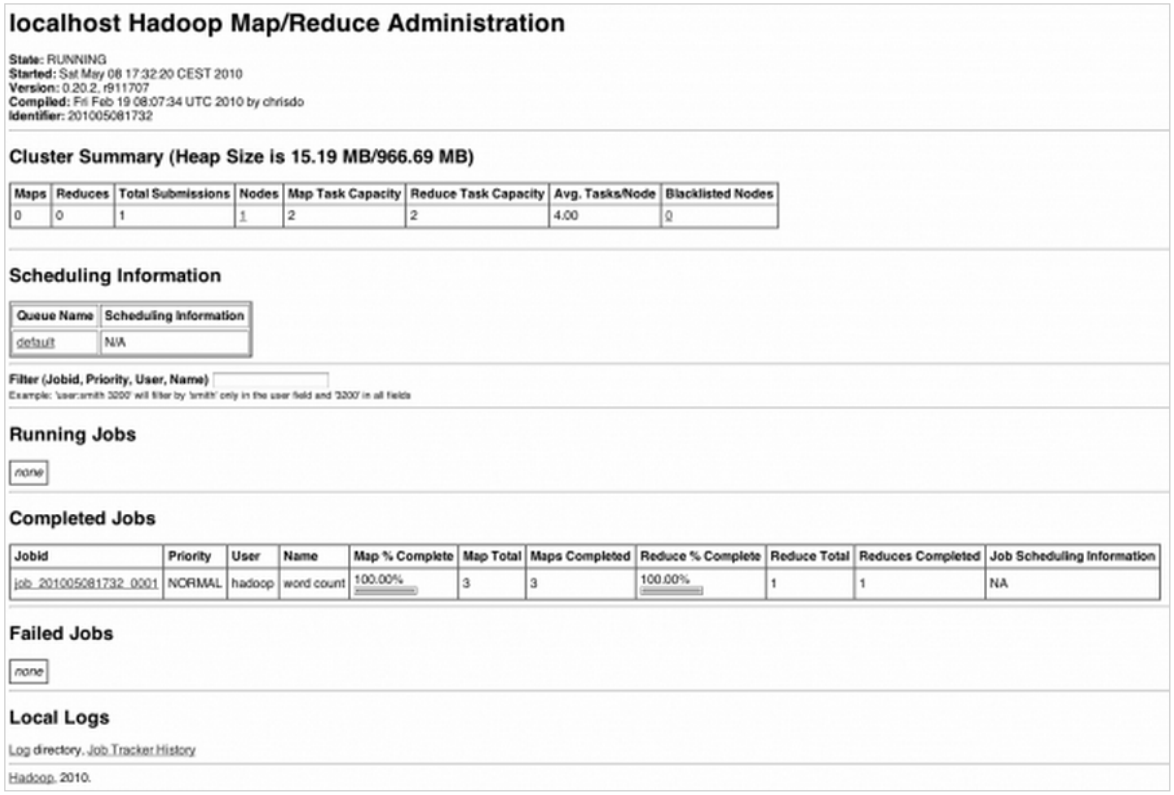


Fig 3.1: A screenshot of Hadoop's Job Tracker web interface.

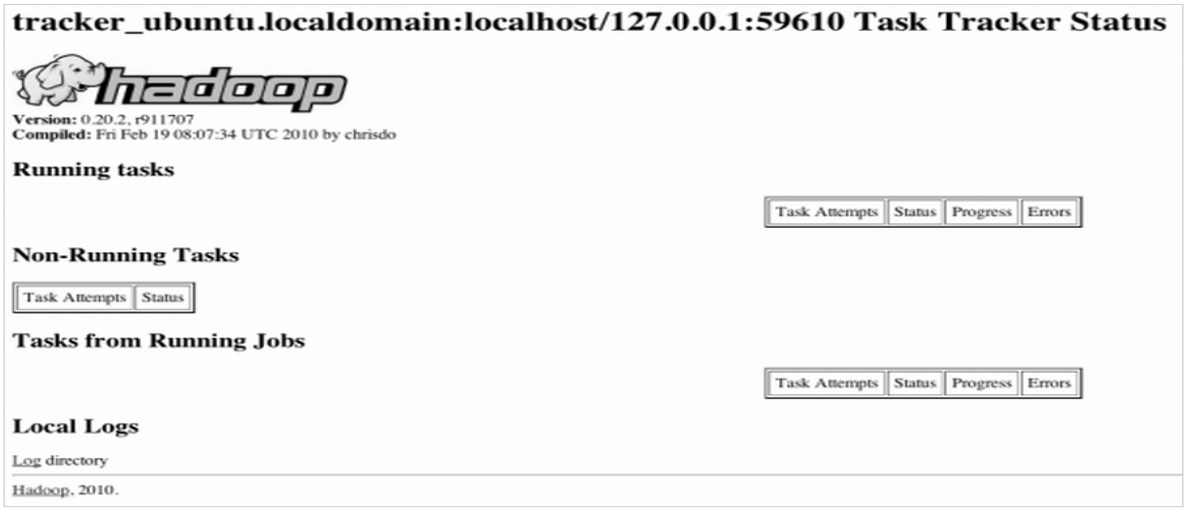


Fig 3.2: A screenshot of Hadoop's Task Tracker web interface.

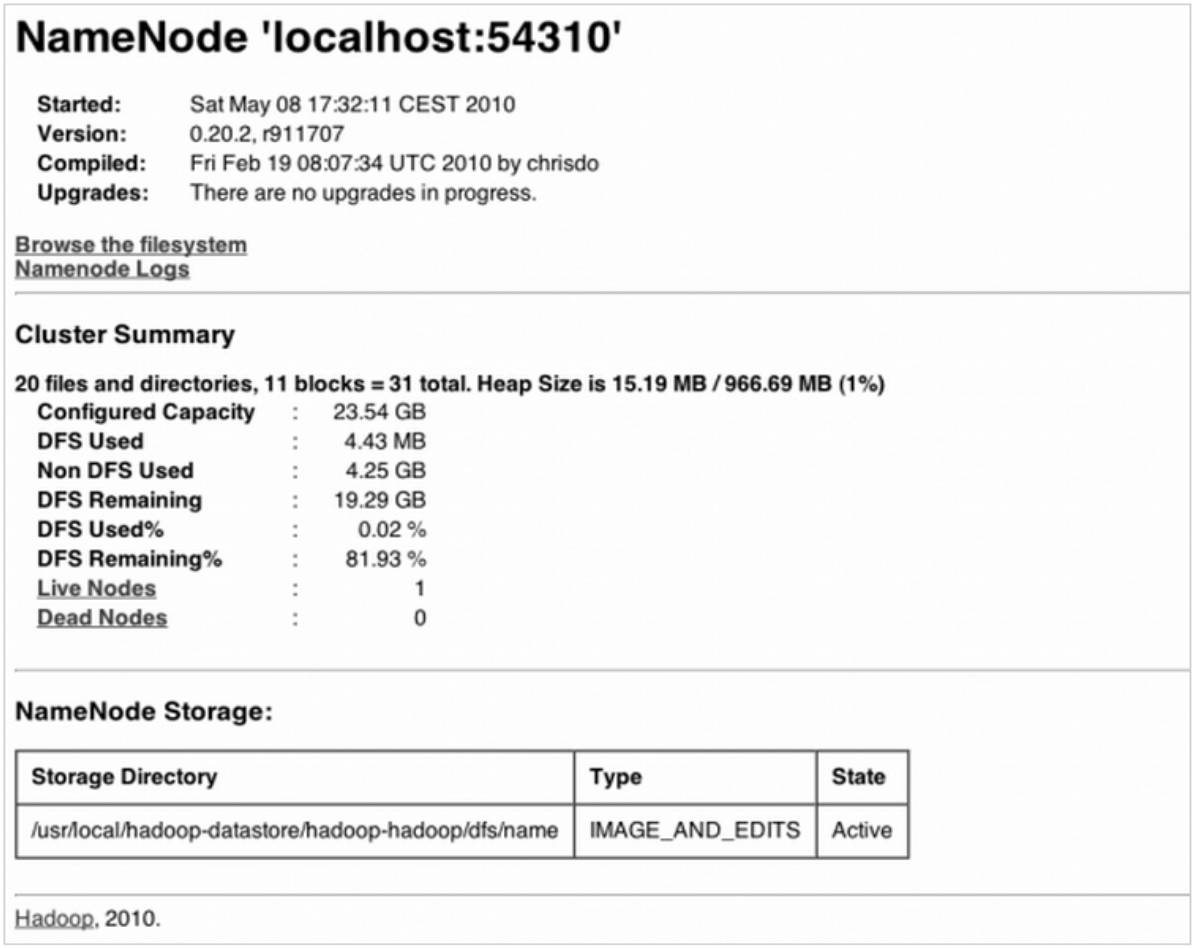


Fig 3.3: A screenshot of Hadoop's NameNode web interface.

3.5 Performance Requirements

- Real-time performance.
- Minimizing the total executing time of jobs.
- System load conditions and scheduling metrics.
- Impact of data skews and communication delays on scheduling.

Chapter 4

System Design

4.1 Cluster Design

A Hadoop cluster consists of a relatively simple architecture of masters and slaves (see Fig 5.1). The NameNode is the overall master of a Hadoop cluster and is responsible for the file system namespace and access control for clients. There also exists a JobTracker, whose job is to distribute jobs to waiting nodes. These two entities (NameNode and JobTracker) are the masters of the Hadoop architecture [6][10]. The slaves consist of the TaskTracker, which manages the job execution (including starting and monitoring jobs, capturing their output, and notifying the JobTracker of job completion). The DataNode is the storage node in a Hadoop cluster and represents the distributed file system (or at least a portion of it for multiple DataNodes). The TaskTracker and the DataNode are the slaves within the Hadoop cluster.

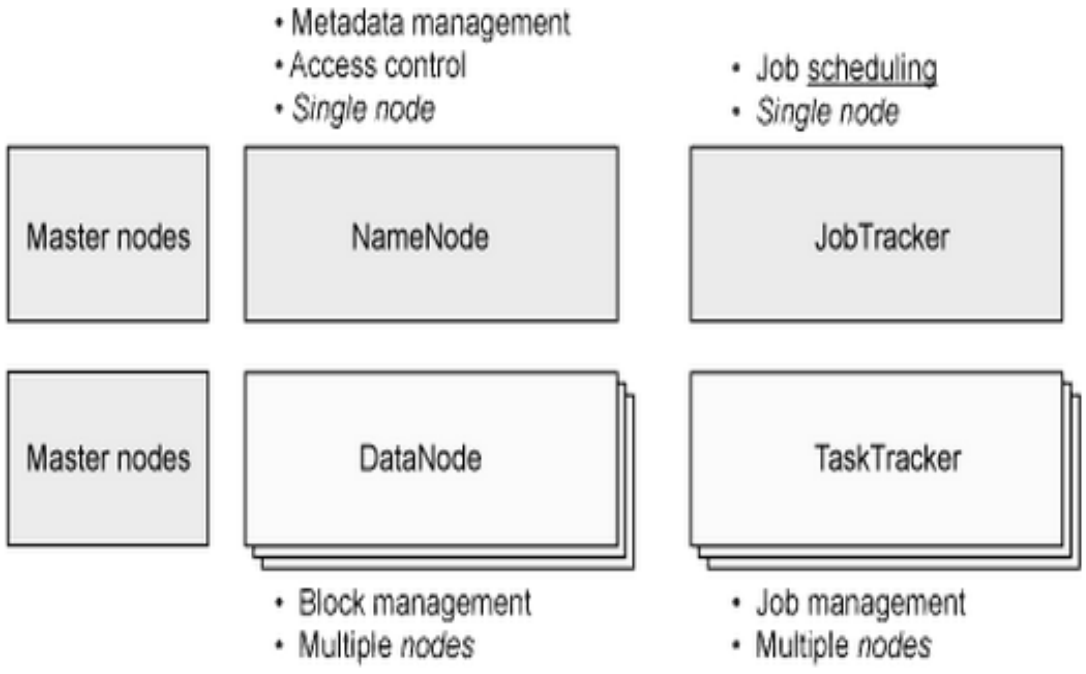


Fig 4.1: Elements of a Hadoop cluster

The cluster we setup consists of two connected slave nodes on which jobs are scheduled and a master node on which the scheduler resides. Each slave node/data node is a multicore processor, which is configured to have a number of slots for executing tasks. Each job submitted to the cluster consists of a set of independent map tasks, followed by a set of independent reduce tasks. The HDFS stores the input to the MapReduce application.The input is divided into smaller blocks and duplicated on multiple nodes. Each map task processes an input data block, which consists of a number of (key, value) tuples that are stored at one of the slave nodes. Each reduce task (containing also shuffle and sort phases) computes final results from the output data of all the map tasks [6].

4.2 Scheduler Design

The diagram below shows the architecture of the new Hadoop system. The Hadoop framework consists of the HDFS and the MapReduce framework. The Job Scheduling code falls between the HDFS and the MapReduce framework. The data for the map reduce program, is divided into pieces and stored in the HDFS file system. The scheduler schedules a task from the various map tasks and sends it for execution.

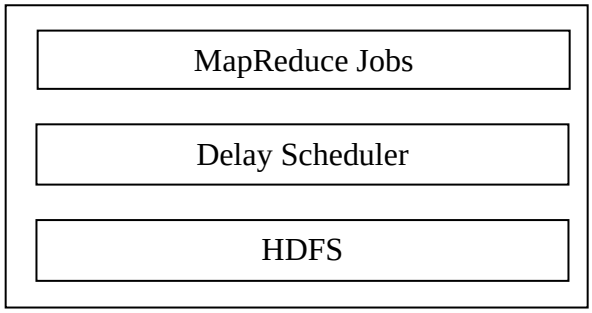


Fig 4.2: System Architecture

The flow of execution of the project is as shown in the following diagrams.

- Once Hadoop is setup and a cluster is formed, different data-intensive applications are run on the cluster using the Hadoop default scheduler (FIFO), Facebook's Fair scheduler and Yahoo!'s Capacity scheduler.
- Each application is run for different data sets and the time taken to complete the job is logged.
- The Hadoop default scheduler FIFO is optimized to include data locality awareness.

- Example programs are run on this Hadoop architecture and the time taken for job completion is logged.
- Next the same programs are executed with the Delay scheduler. The corresponding output parameters are recorded.
- The output parameters from the above scheduling policies are compared to analyze the performance improvement that can be achieved by using the new schedulers.

Flow of Execution:

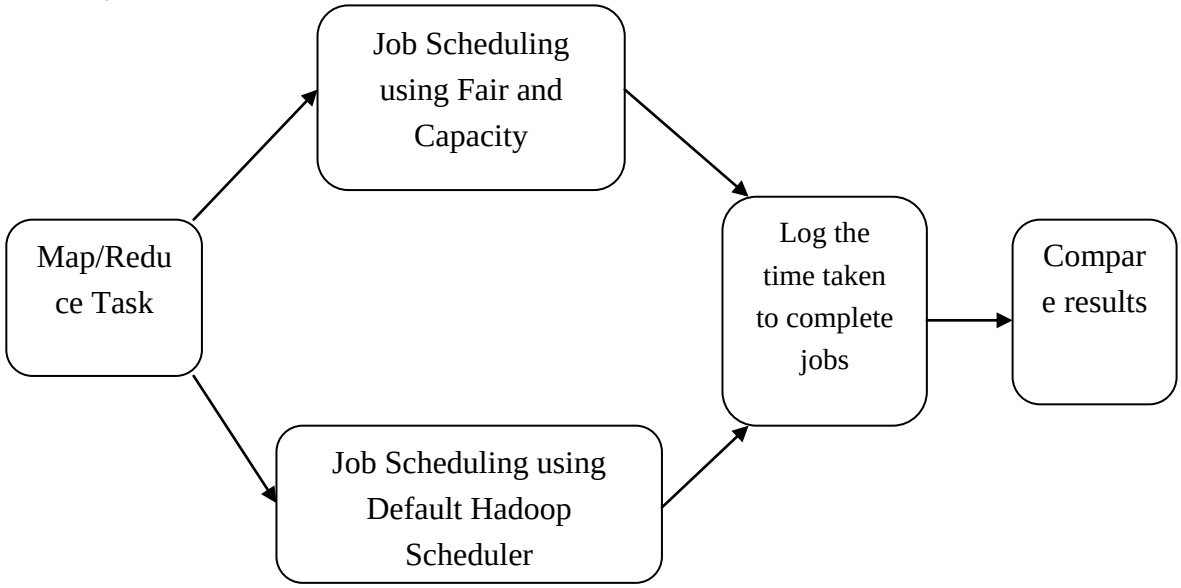


Fig 4.3: Comparative analysis of Hadoop default scheduler with Fair and Capacity Schedulers

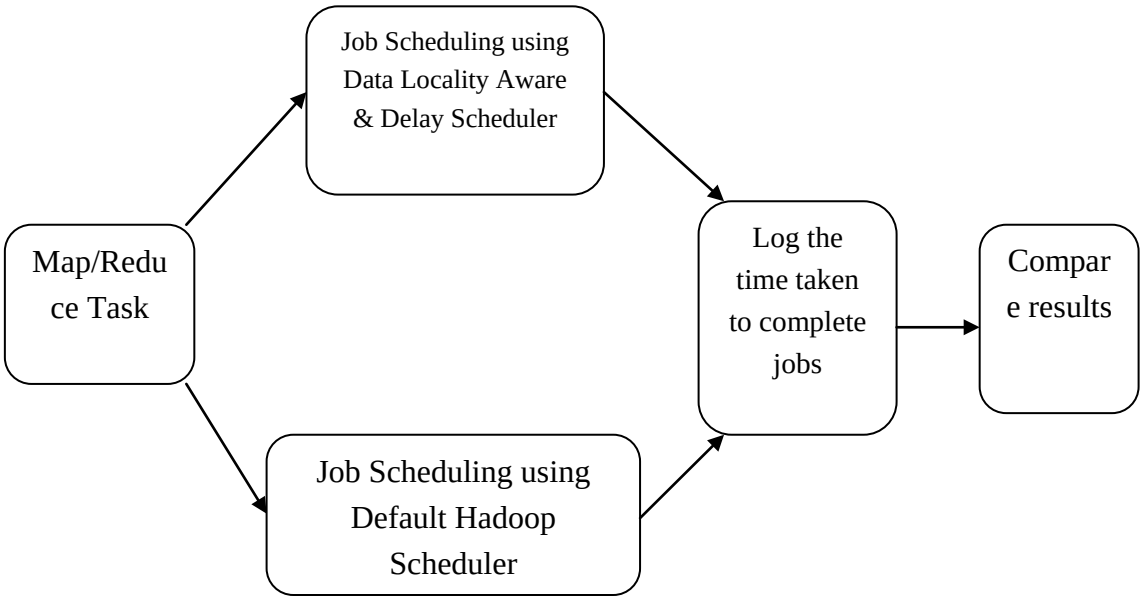


Fig 4.4: Comparison of the Hadoop default scheduler with an optimized schedulers

Chapter 5

Cluster Implementation

5.1 Running Hadoop on a Single Node Cluster (Setup)

5.1.1 Prerequisites

The single node cluster setup requires the following softwares:

- Ubuntu Linux 11.04 LTS, 10.10 LTS
- Hadoop 0.20.2/x, released February 2010

Sun Java 6

Hadoop requires a working Java 1.5.x (aka 5.0.x) installation. However, using Java 1.6.x (aka 6.0.x aka 6) is recommended for running Hadoop. Open Ubuntu terminal and execute the following commands:

$ sudo apt-get install python-software-properties
 # add repositories
$ add-apt-repository ppa:ferramroberto/
 # update source
$ sudo apt-get update
 # install Sun Java6 JDK
$ sudo apt-get install sun-java6-jdk sun-java6-plugins sun-java6-fonts

The full JDK will be placed in /usr/lib/jvm/java-6-sun of the Ubuntu Filesystem. After installation, make a quick check whether Sun's JDK is correctly set up:

user@ubuntu:~# java –version
java version "1.6.0_20"
Java(TM) SE Runtime Environment (build 1.6.0_20-b02)
Java HotSpot(TM) Client VM (build 16.3-b01, mixed mode, sharing)

Adding a dedicated Hadoop system user

A dedicated Hadoop user account for running Hadoop is created. This helps to separate the Hadoop installation from other software applications and user accounts running on the same machine (keeping in mind the security, permissions, backups, etc).

$ sudo addgroup hadoop

$ sudo adduser --ingroup hadoop hduser

This will add the user hduser and the group hadoop to the local machine.

Configuring SSH

Hadoop requires SSH access to manage its nodes, i.e. remote machines plus the local machine. For our single-node setup of Hadoop, we therefore need to configure SSH access to localhost for the hduser user created in the previously. First have SSH up and running on the machine and it should be configured it to allow SSH public key authentication. Next, we have to generate an SSH key for the hduser user.

```
hduser@ubuntu:~$ ssh-keygen -t rsa -P ""
Generating public/private rsa key pair.
Enter file in which to save the key (/home/hduser/.ssh/id_rsa):
Created directory '/home/hduser/.ssh'.
Your identification has been saved in /home/hduser/.ssh/id_rsa.
Your public key has been saved in /home/hduser/.ssh/id_rsa.pub.
The key fingerprint is:
d1:f0:27:59:82:a0:75:df:f5:a4:9b:15:d9:39:a7:c9 hduser@ubuntu
The key's randomart image is:
+--[ RSA 2048]----+
|     o.o.. . ..=|
|    o ..= = . B+|
|   .  . * o..o=|
|       . o  E+ |
|      S    o  |
+-----------------+
```

The second line will create an RSA key pair with an empty password. Next enable SSH access to the local machine with this newly created key.

hduser@ubuntu:~$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys

The final step is to test the SSH setup by connecting to the local machine with the hduser user. The step is also needed to save the local machine's host key fingerprint to the hduser user's known_hosts file.

hduser@ubuntu:~$ ssh localhost
The authenticity of host 'localhost (::1)' can't be established.
RSA key fingerprint is a0:a6:48:8f:bf:ab:88:d1:82:5b:05:a9:2b:8a:c4:2f.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (RSA) to the list of known hosts.
Linux ubuntu 2.6.35-31-generic #63-Ubuntu SMP Mon Nov 28 19:23:11 UTC 2011 i686 GNU/Linux
Ubuntu 10.10

Disabling IPv6

One problem with IPv6 on Ubuntu is that using 0.0.0.0 for the various networking-related Hadoop configuration options will result in Hadoop binding to the IPv6 addresses of the Ubuntu box. Disable IPv6 only for Hadoop as documented in HADOOP-3437. Do so by adding the following line to conf/hadoop-env.sh:
export HADOOP_OTPS=Djava.net.preferIPv4Stack=true

5.1.2 Hadoop

Installation

Download the latest stable version of Hadoop from the Apache Download Mirrors and extract the contents of the Hadoop package to /usr/local/hadoop. Change the owner of all the files to the hduser user and hadoop group as shown below

$ cd /usr/local
$ sudo tar/xfv Hadoop-0.20.2.tar.gz

$ sudo mv Hadoop-0.20.2 hadoop

$ sudo chown –R hduser:hadoop hadoop

Update $HOME/.bashrc

Add the following lines to the end of the $HOME/.bashrc file of user hduser.

```
# Set Hadoop-related environment variables
export HADOOP_HOME=/usr/local/hadoop

# Set JAVA_HOME (we will also configure JAVA_HOME directly for Hadoop later on)
export JAVA_HOME=/usr/lib/jvm/java-6-sun

# Some convenient aliases and functions for running Hadoop-related commands
unalias fs &> /dev/null
alias fs="hadoop fs"
unalias hls &> /dev/null
alias hls="fs -ls"

# If you have LZO compression enabled in your Hadoop cluster and
# compress job outputs with LZOP (not covered in this tutorial):
# Conveniently inspect an LZOP compressed file from the command
# line; run via:
#
# $ lzohead /hdfs/path/to/lzop/compressed/file.lzo
#
# Requires installed 'lzop' command.
#
lzohead () {
    hadoop fs -cat $1 | lzop -dc | head -1000 | less
}

# Add Hadoop bin/ directory to PATH
export PATH=$PATH:$HADOOP_HOME/bin
```

Configuration

In order to configure Hadoop make changes to the following documents.

hadoop-env.sh

The only required environment variable that has to be configured for Hadoop is JAVA_HOME. Open /conf/hadoop-env.sh in the editor and set the JAVA_HOME environment variable to the Sun JDK/JRE 6 directory.

Change

#export JAVA_HOME=/usr/lib/j2sdk1.5-sun

To

#export JAVA_HOME=/usr/lib/jvm/java-6-sun

Conf/-site.xml*

The ownerships and permissions required are set as follows. If the change in ownership is not made we get java.IOException when the namenode is formatted next time.

$ sudo mkdir –p /app/hadoop/tmp

$ sudo chown hduser:hadoop /app/hadoop/tmp

 # and if you want to tighten up security, chmod 755 to 750

$ sudo chmod 750 /app/hadoop/tmp

conf/core-site.xml

Here configuration parameters are included in the conf/core-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<property>
  <name>hadoop.tmp.dir</name>
  <value>/app/hadoop/tmp</value>
  <description>A base for other temporary directories.</description>
</property>

<property>
  <name>fs.default.name</name>
  <value>hdfs://localhost:54310</value>
  <description>The name of the default file system.  A URI whose
  scheme and authority determine the FileSystem implementation.  The
  uri's scheme determines the config property (fs.SCHEME.impl) naming
  the FileSystem implementation class.  The uri's authority is used to
  determine the host, port, etc. for a filesystem.</description>
</property>

</configuration>
~
~
```

conf/mapred-site.xml

The mapred configuration properties are defined in the conf/mapred-site.xml.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
  <name>mapred.job.tracker</name>
  <value>localhost:54311</value>
  <description>The host and port that the MapReduce job tracker runs
  at.  If "local", then jobs are run in-process as a single map
  and reduce task.
  </description>
</property>
</configuration>
~
~
~
```

conf/hdfs-site.xml

The replication value property is set.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
  <name>dfs.replication</name>
  <value>1</value>
  <description>Default block replication.
  The actual number of replications can be specified when the file is created.
  The default is used if replication is not specified in create time.
  </description>
</property>
</configuration>
~
~
~
```

Formatting the HDFS filesystem via the NameNode

By formatting the NameNode we initialize the directory specified by the df s.name.dir variable.To do this we run the following command:

hduser@ubuntu:/usr/local/hadoop$ bin/hadoop namenode -format

The following output is obtained.

12/05/02 12:10:12 INFO namenode.NameNode: STARTUP_MSG:

*/***

STARTUP_MSG: Starting NameNode

STARTUP_MSG: host -ubuntu/127.0.1.1

STARTUP_MSG: args = [-format]

STARTUP_MSG: version = 0.20-append-r1057313-123

STARTUP_MSG: build = http://svn.apache.org/repos/asf/hadoop/common/branches/branch-0.20-append -r ; compiled by 'root' on Wed May 2 11:38:58 IST 2012

***/

12/05/02 12:10:12 INFO namenode.FSNamesystem: fsOwner=hduser1,hadoop

12/05/02 12:10:12 INFO namenode.FSNamesystem: supergroup=supergroup

12/05/02 12:10:12 INFO namenode.FSNamesystem: isPermissionEnabled=true

12/05/02 12:10:12 INFO common.Storage: Image file of size 97 saved in 0 seconds.

12/05/02 12:10:12 INFO common.Storage: Storage directory /app1/hadoop/tmp/dfs/name has been successfully formatted.

12/05/02 12:10:12 INFO namenode.NameNode: SHUTDOWN_MSG:

*/***

SHUTDOWN_MSG: Shutting down NameNode at ubuntu/127.0.1.1

***/

Starting the single-node cluster

To start the cluster the following command has to be run. Starting the cluster will startup a Namenode, Datanode, Jobtracker and a Tasktracker on machine.

hduser@ubuntu:/usr/local/hadoop $ bin/start-all.sh

The following output is obtained:

hduser@ubuntu:/usr/local/hadoop $ bin/start-all.sh

starting namenode, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser1-namenode-ubuntu.out

localhost: starting datanode, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser1-datanode-ubuntu.out

localhost: starting secondarynamenode, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser1-secondarynamenode-ubuntu.out

starting jobtracker, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser1-jobtracker-ubuntu.out

localhost: starting tasktracker, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser1-tasktracker-ubuntu.out

hduser@ubuntu:/usr/local/hadoop$

To check whether the expected Hadoop processes are running, run the *jps* command.

hduser@ubuntu:/usr/local/hadoop$ jps

5065 JobTracker

4601 NameNode

4779 DataNode

5271 Jps

4955 SecondaryNameNode

5232 TaskTracker

Check with netstat if Hadoop is listening on the configured ports.

Running the MapReduce Job

Text files in Plain Text UTF-8 encoding are obtained from Project Gutenberg and stored in a temporary directory /tmp/gutenberg.

```
hduser@ubuntu:~$ ls -l /tmp/gutenberg/
total 3604
-rw-r--r-- 1 hduser hadoop  674566 Feb  3 10:17 pg20417.txt
-rw-r--r-- 1 hduser hadoop 1573112 Feb  3 10:18 pg4300.txt
-rw-r--r-- 1 hduser hadoop 1423801 Feb  3 10:18 pg5000.txt
hduser@ubuntu:~$
```

Restart the Hadoop cluster and copy the local data from /tmp/gutenberg onto the HDFS.

```
hduser@ubuntu:/usr/local/hadoop$ bin/hadoop dfs -copyFromLocal /tmp/gutenberg /user/hduser/
hduser@ubuntu:/usr/local/hadoop$ bin/hadoop dfs -ls /user/hduser
Found 1 items
drwxr-xr-x   - hduser supergroup          0 2010-05-08 17:40 /user/hduser/gutenberg
hduser@ubuntu:/usr/local/hadoop$ bin/hadoop dfs -ls /user/hduser/gutenberg
Found 3 items
-rw-r--r--   3 hduser supergroup     674566 2011-03-10 11:38 /user/hduser/gutenberg/pg20417
-rw-r--r--   3 hduser supergroup    1573112 2011-03-10 11:38 /user/hduser/gutenberg/pg4300.
-rw-r--r--   3 hduser supergroup    1423801 2011-03-10 11:38 /user/hduser/gutenberg/pg5000.
hduser@ubuntu:/usr/local/hadoop$
```

Now run the word count program using:

*hduser@master:/usr/local/hadoop$ bin/hadoop jar hadoop*examples*.jar wordcount /user/hduser/gutenberg /user/hduser/gutenberg-output*

... INFO mapred.FileInputFormat: Total input paths to process : 3

... INFO mapred.JobClient: Running job: job_0001

... INFO mapred.JobClient: map 0% reduce 0%

... INFO mapred.JobClient: map 28% reduce 0%

... INFO mapred.JobClient: map 57% reduce 0%

... INFO mapred.JobClient: map 71% reduce 0%

... INFO mapred.JobClient: map 100% reduce 9%

... INFO mapred.JobClient: map 100% reduce 68%

... INFO mapred.JobClient: map 100% reduce 100%

.... INFO mapred.JobClient: Job complete: job_0001

... INFO mapred.JobClient: Counters: 11

... INFO mapred.JobClient: org.apache.hadoop.examples.WordCount$Counter

... INFO mapred.JobClient: WORDS=1173099

... INFO mapred.JobClient: VALUES=1368295

... INFO mapred.JobClient: Map-Reduce Framework

... INFO mapred.JobClient: Map input records=136582

... INFO mapred.JobClient: Map output records=1173099

... INFO mapred.JobClient: Map input bytes=6925391

... INFO mapred.JobClient: Map output bytes=11403568

... INFO mapred.JobClient: Combine input records=1173099

... INFO mapred.JobClient: Combine output records=195196

... INFO mapred.JobClient: Reduce input groups=131275

The output can obtained using

hduser@ubuntu:/usr/local/hadoop$ bin/hadoop dfs -ls /user/hduser/gutenberg-output

The following output is obtained

Found 2 items

drwxr-xr-x - hduser supergroup /user/hduser/gutenberg-output/_logs

-rw-r--r-- 1 hduser supergroup /user/hduser/gutenberg-output/part-r-00000

hduser@ubuntu:/usr/local/hadoop$

We can increase the number of Reduce tasks by specifying the "-D" option as:

*hduser@ubuntu:/usr/local/hadoop$ bin/Hadoop jar Hadoop*examples*.jar wordcount -D user/hduser/gutenberg /user/hduser/gutenberg-output*

The Hadoop web interfaces can also be used to view the output, job and task statistics

- http://localhost:50030/ – web UI for MapReduce job tracker(s)
- http://localhost:50060/ – web UI for task tracker(s)
- http://localhost:50070/ – web UI for HDFS name node(s)

Stopping the single-node cluster

Run the following command to stop all the daemons running on the machine.

hduser@ubuntu:~$ /usr/local/hadoop/bin/stop-all.sh

The following output is seen.

hduser@ubuntu:/usr/local/hadoop$ bin/stop-all.sh

stopping jobtracker

localhost: stopping tasktracker

stopping namenode

localhost: stopping datanode

localhost: stopping secondarynamenode

5.2 Running Hadoop on a Multi-Node Cluster (Setup)

5.2.1 Prerequisites

Configuring single-node clusters first

Here 3 single-node clusters are configured, one as master and the other two as the slave1 and slave2. The designated master machine is called just the master from now on and the slave-only machines the slave1 and slave2. The two machines are given these respective hostnames in their networking setup in /etc/hosts. Shutdown each single-node cluster with /bin/stop-all.sh before continuing

5.2.2 Networking

All three machines must be able to reach each other over the network. The easiest is to put both machines in the same network with regard to hardware and software configuration. Setup a private network of your own using a separate router in order to minimize contention. Connect all the three single node clusters to the router and create a new wired connection on each of the computers. Define the gateway address, IP address and the network mask for each.

Update /etc/hosts on both machines with the following lines:

#/etc/hosts
172.1.6.122 master
172.1.6.106 slave1
172.1.6.98 slave2

5.2.3 SSH Access

The hduser user on the master must be able to connect

a) to its own user account on themaster – i.e. ssh master in this context and not necessarily ssh localhost – and

b) to the hduser user account on the slave via a password-less SSH login.

Next add the hduser@master's public SSH key manually or use the following SSH command:

hduser@master:~$ ssh-copy-id –I $HOME/.ssh/id_rsa.pub hduser@slave1

hduser@master:~$ ssh-copy-id –I $HOME/.ssh/id_rsa.pub hduser@slave2

Then test the SSH setup by connecting with user hduser from the master to the user account hduser on the slave1 and slave2. The step is also needed to save slave‘s host key fingerprint to the hduser@master‘s known_hosts file.

Connect master to master

```
hduser@master:~$ ssh master
The authenticity of host 'master (192.168.0.1)' can't be established.
RSA key fingerprint is 3b:21:b3:c0:21:5c:7c:54:2f:1e:2d:96:79:eb:7f:95.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'master' (RSA) to the list of known hosts.
Linux master 2.6.20-16-386 #2 Thu Jun 7 20:16:13 UTC 2007 i686
...
hduser@master:~$
```

Connect master to slave

```
hduser@master:~$ ssh slave
The authenticity of host 'slave (192.168.0.2)' can't be established.
RSA key fingerprint is 74:d7:61:86:db:86:8f:31:90:9c:68:b0:13:88:52:72.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'slave' (RSA) to the list of known hosts.
Ubuntu 10.04
...
hduser@slave:~$
```

5.2.4 Hadoop

Cluster Overview

The next sections will describe how to configure one Ubuntu box as a master node and the other Ubuntu box as a slave node. The master node will also act as a slave because we only have two machines available in our cluster but still want to spread data storage and processing to multiple machines.

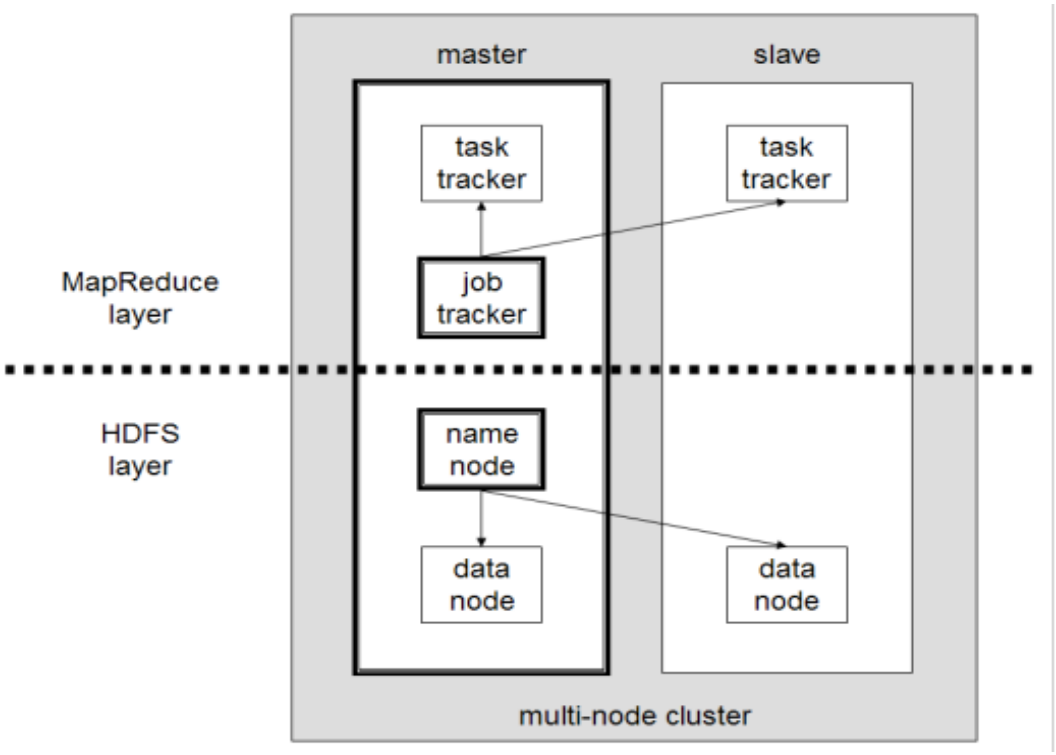


Fig 5.1: Multi-node cluster Organization

Configuration on master

- *Conf/masters*

The conf/masters file defines on which machines Hadoop will start secondary NameNodes in our multi-node cluster The primary NameNode and the JobTracker will always be the machines on which the bin/start-dfs.sh and bin/start-mapred.sh scripts are run, respectively ie. the master. The *conf/masters* is updated as:

master

- *Conf/slaves*

The conf/slaves file lists the hosts, one per line, where the Hadoop slave daemons (DataNodes and TaskTrackers) will be run. The conf/masters is updated as:

master

slave1

slave2

For configuring additional slave nodes add the following lines into conf/slaves file of all the machines in the cluster.

master

slave1

slave2

slave3

Configuring conf/-site.xml on master and slave*

- **conf/core-site.xml:**

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<property>
  <name>hadoop.tmp.dir</name>
  <value>/app/hadoop/tmp</value>
  <description>A base for other temporary directories.</description>
</property>
<property>
  <name>fs.default.name</name>
  <value>hdfs://master:54310</value>
  <description>The name of the default file system.  A URI whose
  scheme and authority determine the FileSystem implementation.  The
  uri's scheme determines the config property (fs.SCHEME.impl) naming
  the FileSystem implementation class.  The uri's authority is used to
  determine the host, port, etc. for a filesystem.</description>
</property>
</configuration>
```

- **conf/mapred-site.xml**

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
```

<property>

<name>mapred.job.tracker</name>

<value>master:54311</value>

<description>The host and port that the MapReduce job tracker runs

at. If "local", then jobs are run in-process as a single map

and reduce task.

</description>

</property>

</configuration>

- **conf/hdfs-site.xml**

<?xml version="1.0"?>

<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>

<property>

<name>dfs.replication</name>

<value>3</value>

<description>Default block replication.

The actual number of replications can be specified when the file is created.

The default is used if replication is not specified in create time.

</description>

</property>

</configuration>

Formatting HDFS via Namenode

Before the new multi-node cluster is started, format the Hadoop's distributed filesystem (HDFS) for the NameNode. This is done the first time when a Hadoop cluster is setup. The following command is executed.

hduser@ubuntu:~$ /usr/local/hadoop/bin/hadoop namenode -format

Start multi-node cluster

Starting the cluster is done in two steps.

- First, the HDFS daemons are started: the NameNode daemon is started on master, and DataNode daemons are started on all slaves.
- Second, the MapReduce daemons are started: the JobTracker is started on master, and TaskTracker daemons are started on all slaves

HDFS daemons

Run the command /bin/start-dfs.sh on the master machine. This will bring up HDFS with the NameNode running on the master machine, and DataNodes on the machines listed in the conf/slaves file.

hduser@master:/usr/local/hadoop$ bin/start-dfs.sh
starting namenode, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-namenode-master.out
slave1: Ubuntu 10.04
slave1: starting datanode, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-datanode-slave.out
slave2: Ubuntu 10.04
slave2: starting datanode, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-datanode-slave.out
master: starting datanode, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-datanode-master.out
master: starting secondarynamenode, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-secondarynamenode-master.out

Processes running:
The processes running are determined using the jps command.

- On master

    hduser@master:/usr/local/hadoop$ jps
    14799 NameNode
    15314 Jps
    14880 DataNode
    14977 SecondaryNameNode

- On slave

hduser@slave:/usr/local/hadoop$ jps

15616 Jps

15183 DataNode

Mapred deamons:

Run the command /bin/start-mapred.sh on the master machine. This will bring up the MapReduce cluster with the JobTracker running on the master machine, and TaskTrackers on the machines listed in the conf/slaves file.

hduser@master:/usr/local/hadoop$ bin/start-mapred.sh

starting jobtracker, logging to /usr/local/hadoop/bin/../logs/hadoop-hadoop-jobtracker-master.out

slave: Ubuntu 10.04

slave: starting tasktracker, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-tasktracker-slave.out

slave1: Ubuntu 10.04

slave1: starting tasktracker, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-tasktracker-slave.out

master: starting tasktracker, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-tasktracker-master.out

hduser@master:/usr/local/hadoop$

Processes running:

The processes running are determined using the jps command.

- On master

hduser@master:/usr/local/hadoop$ jps

16017 Jps

14799 NameNode

15686 TaskTracker

14880 DataNode

15596 JobTracker

14977 SecondaryNameNode

- On slave

  hduser@slave:/usr/local/hadoop$ jps

  15183 DataNode

  15897 TaskTracker

  16284 Jps

Running the MapReduce Job

Seven text files in Plain Text UTF-8 encoding are obtained from Project Gutenberg and stored in a temporary directory /tmp/gutenberg. Copy these to the HDFS and run the WordCount example MapReduce job on master, and retrieve the job result from HDFS to the local filesystem.

*hduser@master:/usr/local/hadoop$ bin/hadoop jar hadoop*examples*.jar wordcount /user/hduser/gutenberg /user/hduser/gutenberg-output*

... INFO mapred.FileInputFormat: Total input paths to process : 7

... INFO mapred.JobClient: Running job: job_0001

... INFO mapred.JobClient: map 0% reduce 0%

... INFO mapred.JobClient: map 28% reduce 0%

... INFO mapred.JobClient: map 57% reduce 0%

... INFO mapred.JobClient: map 71% reduce 0%

... INFO mapred.JobClient: map 100% reduce 9%

... INFO mapred.JobClient: map 100% reduce 68%

... INFO mapred.JobClient: map 100% reduce 100%

.... INFO mapred.JobClient: Job complete: job_0001

... INFO mapred.JobClient: Counters: 11

... INFO mapred.JobClient: org.apache.hadoop.examples.WordCount$Counter

... INFO mapred.JobClient: WORDS=1173099

... INFO mapred.JobClient: VALUES=1368295

... INFO mapred.JobClient: Map-Reduce Framework

... INFO mapred.JobClient: Map input records=136582

... INFO mapred.JobClient: Map output records=1173099

... INFO mapred.JobClient: Map input bytes=6925391

... INFO mapred.JobClient: Map output bytes=11403568

... INFO mapred.JobClient: Combine input records=1173099

... INFO mapred.JobClient: Combine output records=195196

... INFO mapred.JobClient: Reduce input groups=131275

... INFO mapred.JobClient: Reduce input records=195196

... INFO mapred.JobClient: Reduce output records=131275

hduser@master:/usr/local/hadoop$

On the master check the logs (hadoop/logs) of the Namenode and the Jobtracker to see how they work. On the slave machine check the Datanode and Tasktracker logs [3][9].

Chapter 6

Implementation

6.1 Fair Scheduler

6.1.1 Installation

To run the fair scheduler in the Hadoop installation, first put it on the CLASSPATH. The easiest way is to copy the hadoop-0.20.2-fairscheduler.jar from hadoop/contrib/fairscheduler to hadoop/lib. Alternatively you can modify HADOOP_CLASSPATH to include this jar, in Hadoop/conf/hadoop-env.sh [5].

- In order to compile fair scheduler, from sources execute ant package in source folder and copy the build/contrib/fair-scheduler/hadoop-0.20.2-fairscheduler.jar to HADOOP_HOME/lib.
- Next set the following property in the Hadoop config file Hadoop/conf /mapred-site.xml to have Hadoop use the fair scheduler:

```
<property>
  <name>mapred.jobtracker.taskScheduler</name>
  <value>org.apache.hadoop.mapred.FairScheduler</value>
</property>
```

- Once you restart the cluster, you can check that the fair scheduler is running by going to http://localhost: 50030/scheduler on the JobTracker's web UI. A "job scheduler administration" page should be visible there.

6.1.2 Configuring the Fair scheduler

The following properties can be set in mapred-site.xml to configure the fair scheduler [5]:

PROPERTY	DESCRIPTION
mapred.fairscheduler.allocation.file	Specifies an absolute path to an XML file which contains the allocations for each pool, as well as the per-pool and per-user

	limits on number of running jobs. If this property is not provided, allocations are not used.
mapred.fairscheduler.assignmultiple	Allows the scheduler to assign both a map task and a reduce task on each heartbeat, which improves cluster throughput when there are many small tasks to run. Boolean value, default: false.
mapred.fairscheduler.sizebasedweight	Take into account job sizes in calculating their weights for fair sharing.By default, weights are only based on job priorities.
mapred.fairscheduler.poolnameproperty	Specify which jobconf property is used to determine the pool that a job belongs in. String, default: user.name. Other values to set this to are: group.name and mapred.job.queue.name.
mapred.fairscheduler.weightadjuster	An extensibility point that lets you specify a class to adjust the weights of running jobs. This class should implement the *WeightAdjuster* interface.
mapred.fairscheduler.loadmanager	An extensibility point that lets you specify a class that determines how many maps and reduces can run on a given TaskTracker. This class should implement the LoadManager interface. By default the task caps in the Hadoop config file are used, but this option could be used to make the load based on available memory and CPU utilization for example.

mapred.fairscheduler.taskselector	An extensibility point that lets you specify a class that determines which task from within a job to launch on a given tracker. This can be used to change either the locality policy or the speculative execution algorithm. The default implementation uses Hadoop's default algorithms from JobInProgress.

Table 6.1: Properties that can be set in mapred-site.xml to configure the fair scheduler

6.1.3 Administration

The fair scheduler provides support for administration at runtime through two mechanisms:

- It is possible to modify pools' allocations and user and pool running job limits at runtime by editing the allocation config file. The scheduler will reload this file 10-15 seconds after it sees that it was modified.
- Current jobs, pools, and fair shares can be examined through the JobTracker's web interface, at http://localhost:50030/scheduler. On this interface, it is also possible to modify jobs' priorities or move jobs from one pool to another and see the effects on the fair shares (this requires JavaScript).

The following fields can be seen for each job on the web interface:

- Submitted - Date and time job was submitted.
- JobID, User, Name - Job identifiers as on the standard web UI.
- Pool - Current pool of job. Select another value to move job to another pool.
- Priority - Current priority. Select another value to change the job's priority
- Maps/Reduces Finished: Number of tasks finished / total tasks.
- Maps/Reduces Running: Tasks currently running.
- Map/Reduce Fair Share: The average number of task slots that this job should have at any given time according to fair sharing. The actual number of tasks will go up and down depending on how much compute time the job has had, but on average it will get its fair share amount.

In addition, it is possible to turn on an "advanced" view for the web UI, by going to http://localhost:50030/scheduler?advanced. This view show four more columns used for calculations internally:

- Maps/Reduce Weight: Weight of the job in the fair sharing calculations. This depends on priority and potentially also on job size and job age if the sizebasedweight and NewJobWeightBooster are enabled.
- Map/Reduce Deficit: The job's scheduling deficit in machine- seconds - the amount of resources it should have gotten according to its fair share, minus how many it actually got. Positive deficit means the job will be scheduled again in the near future because it needs to catch up to its fair share. The scheduler schedules jobs with higher deficit ahead of others. Please see the Implementation section of this document for details.

6.2 Capacity Scheduler

6.2.1 Installation

- The Capacity Scheduler is available as a JAR file in the Hadoop tarball under the contrib/capacity-scheduler directory. The name of the JAR file would be on the lines of hadoop-0.20.2-capacity-scheduler.jar.
- The Scheduler can also be built from source by executing ant package, in which case it would be available under build/contrib/capacity-scheduler.
- To run the Capacity Scheduler in your Hadoop installation, put it on the CLASSPATH. The easiest way is to copy the hadoop-0.20.2-capacity-scheduler.jar from to HADOOP_HOME/lib. Alternatively, you can modify HADOOP_CLASSPATH to include this jar, in conf/hadoop-env.sh.

6.2.2 Configuration

Using the Capacity Scheduler

To make the Hadoop framework use the Capacity Scheduler, set up the following property in the site configuration [4]:

```
<property>
 <name>mapred.jobtracker.taskScheduler</name>
 <value> org.apache.hadoop.mapred.CapacityTaskScheduler </value>
 </property>
```

Setting up queues

Multiple queues can be defined to which users can submit jobs with the Capacity Scheduler. To define multiple queues, you should edit the site configuration for Hadoop and modify the mapred.queue.names property. You can also configure ACLs for controlling which users or groups have access to the queues.

Configuring properties for queues

The Capacity Scheduler can be configured with several properties for each queue that control the behavior of the Scheduler. This configuration is in the *conf/capacity-scheduler.xml*. By default, the configuration is set up for one queue, named default. To specify a property for a queue that is defined in the site configuration, you should use the property name as *mapred.capacity-scheduler.queue.<queue-name>.<property-name>*.

The properties defined for queues and their descriptions are listed in the table below:

PROPERTY	DESCRIPTION
mapred.capacity-scheduler.queue.<queue-name>.capacity	Percentage of the number of slots in the cluster that are made to be available for jobs in this queue. The sum of capacities for all queues should be less than or equal 100.
mapred.capacity-scheduler.queue.<queue-name>.maximum-capacity	maximum-capacity defines a limit beyond which a queue cannot use the capacity of the cluster.This provides a means to limit how much excess capacity a queue can use. By default, there is no limit. The maximum-

	capacity of a queue can only be greater than or equal to its minimum capacity. Default value of -1 implies a queue can use complete capacity of the cluster.
mapred.capacity-scheduler.queue.<queue-name>.minimum-user-limit-percent	Each queue enforces a limit on the percentage of resources allocated to a user at any given time, if there is competition for them. This user limit can vary between a minimum and maximum value.
mapred.capacity-scheduler.queue.<queue-name>.user-limit-factor	The multiple of the queue capacity which can be configured to allow a single user to acquire more slots. By default this is set to 1 which ensure that a single user can never take more than the queue's configured capacity irrespective of how idle th cluster is.
mapred.capacity-scheduler.queue.<queue-name>.supports-priority	If true, priorities of jobs will be taken into account in scheduling decisions.

Table 6.2: Properties defined for queues in Capacity Scheduler

Reviewing the configuration of the Capacity Scheduler

Once the installation and configuration is completed, you can review it after starting the Map/Reduce cluster from the admin UI.

- Start the Map/Reduce cluster as usual.
- Open the JobTracker web UI.
- The queues you have configured should be listed under the Scheduling Information section of the page.
- The properties for the queues should be visible in the Scheduling Information column against each queue.

6.3 Data Locality Aware Scheduling

6.3.1 Data Locality Problems

The first locality problem occurs in small jobs (jobs that have small input files and hence have a small number of data blocks to read). The problem is that whenever a job reaches the head of the sorted list [7]. If the head-of-line job is small, it is unlikely to have data on the node that is given to it.

A second locality problem, sticky slots [7], happens even with large jobs if fair sharing is used. The problem is that there is a tendency for a job to be assigned the same slot repeatedly. For example, suppose that there are 10 jobs in a 100-node cluster with one slot per node, and that each job has 10 running tasks. Suppose job j finishes a task on node n. Node n now requests a new task. At this point, j has 9 running tasks while all the other jobs have 10. Therefore, slots are assigned on node n to job j again. Consequently, in steady state, jobs never leave their original slots. This leads to poor data locality because input files are striped across the cluster, so each job needs to run some tasks on each machine.

The problems we presented happen because following a strict queuing order forces a job with no local data to be scheduled. We address them through a simple technique called delay scheduling. When a node requests a task, if the head-of-line job cannot launch a local task, we skip it and look at subsequent jobs. However, if a job has been skipped long enough, we start allowing it to launch non-local tasks, to avoid starvation. The key insight behind delay scheduling is that although the first slot we consider giving to a job is unlikely to have data for it, tasks finish so quickly that some slot with data for it will free up in the next few seconds

6.3.2 Introduction

Since the existing Hadoop 0.20 scheduler does not consider locality of data when scheduling map tasks or reduce tasks to available slots, the FIFO scheduler is modified to accommodate locality awareness.

6.3.3 Goals

Reduce contention on the network by selecting local tasks in preference to rack local tasks or non local tasks. Local tasks are those for which data is available on the node itself.

Features:

- Reduction in execution time.
- When there are excess slots available the scheduler schedules tasks that are non local. The number of tasks that can be allocated in this manner is set to 1 so that other tasks are not deprived of local tasks.
- When a map task has to be allocated, the scheduler first looks for locally available tasks. Only if there are no locally available the next step is to search for rack local tasks. If there is any available task then it is allocated the slot. If no rack local tasks are present non-local tasks are searched for and allocated the slot.
- It uses Load factor to determine how many map tasks and reduce tasks are to be scheduled.

6.3.4 Implementation of the Data Locality Aware Scheduler

When a task has to be assigned to a free slot by the scheduler, assignTasks function is called. Here the cluster status from the taskTrackerManager is updated. The job queue from the JobInProgressListener and the map and reduce task counts of the current tracker are noted.

We then compute the remaining load for map and reduce tasks for each job across the pool. The load factor is calculated for the map and the reduce tasks. The load factor is used to determine the tracker's current map or reduce capacity. We assign tasks to the current taskTracker if the given machine has a workload that's less than the maximum load of that kind of task. However, if the cluster is close to getting loaded i.e. we don't have enough padding for speculative executions etc., we only schedule the "highest priority" task i.e. the task from the job with the highest priority.

For each of the available free slots we first try to schedule node local map task. A few slots are left free to accommodate future failures and speculative tasks. If map padding has exceeded, then we do not allocate the slot. If no node local task is found we find rack local or non local task and if found, the task is allocated a slot. The process of assigning a reduce task remains the same except that the new reduce task is randomly obtained.

6.3.5 Algorithm

Update the cluster status and task tracker details

Calculate the available Map and Reduce capacity

Using Load Factor, determine if a Map or Reduce task should be assigned

For each of the available map slots:

Get the status of each of the running jobs

Obtain either a new local or rack local or non local task and add it to assigned tasks

If map padding exceeds for local task, do not schedule more maps

Repeat the same for Reduce tasks

6.4 Delay Scheduling

6.4.1 Introduction

In delay scheduling, we scan jobs in order given by queuing policy, picking first that is permitted to launch a task. The jobs must wait before being permitted to launch non-local tasks. There is an increase in a job's time waited when it is skipped

6.4.2 Features

- Scan jobs in order given by queuing policy, picking first that is permitted to launch a task.and jobs must wait before being permitted to launch non-local tasks.
- Increase a job's time waited when it is skipped.
- Delay scheduling works well under two conditions:
    - Sufficient fraction of tasks are short relative to jobs
    - There are many locations where a task can run efficiently
        - Blocks replicated across nodes, multiple tasks/node

6.4.3 Implementation of Delay Scheduler

Allocation files are loaded for the pool. The cluster and tracker status for the running and runnable maps and reduces are obtained. We scan the jobs to assign tasks until neither maps or reduces can be assigned. A task is rejected if either the task reaches per heartbeat limit or number of running tasks reaches number of runnable tasks or if the tasks are rejected by the load manager. To determine which task type to assign, we first choose a task type that is not rejected. If both map and reduce are available then we choose the one with fewer running tasks. If the task is of type map, then the new task is obtained depending on the locality level otherwise a new reduce task is obtained.

For each of the jobs we obtain the status and sort them by using FifoJobComparator or DeficitComparator and update the locality wait time. The locality wait time keeps track of the total time waited for a local map task. Locality levels are set depending on the present locality level and a cache level cap is established to ensure that only jobs of a given locality level or lower are launched. A job is marked as visited if the map task was not launched.

We compute he locality level based on the time waited.
If the last launched task was node-local, then the locality level can be:
Non-local/Any: if timeWaitedForLocalMap > nodeLocalityDelay + rackLocalityDelay
Rack: if timeWaitedForLocalMap >= nodeLocalityDelay
Node: otherwise
If last task launched was rack-local:
Non-local/Any: if timeWaitedForLocalMap >= rackLocalityDelay
Rack: otherwise
The default locality level is Any.

6.4.4 Algorithm

when there is a free task slot on node n:
sort jobs according to queuing policy
for j in jobs:
    if j has node-local task t on n:

```
        j.level := 0; j.wait := 0; return t
    else if j has rack-local task t on n and (j.level ≥ 1 or j.wait ≥ T1):
        j.level := 1; j.wait := 0; return t
    else if j.level = 2 or (j.level = 1 and j.wait ≥ T2)
         or (j.level = 0 and j.wait ≥ T1 + T2):
        j.level := 2; j.wait := 0; return t
    else:
        j.wait += time since last scheduling decision
```

Each job begins at a "locality level" of 0, where it can only launch node-local tasks. If it waits at least W1 seconds, it goes to locality level 1 and may launch rack-local tasks. If it waits a further W2 seconds, it goes to level 2 and may launch off-rack tasks. Finally, if a job ever launches a "more local" task than the level it is on, it goes back down to a previous level.

6.5 MapReduce Applications

Even though the Hadoop framework is written in Java, programs for Hadoop need not to be coded in Java but can also be developed in other languages like Python or C++. The key behind the writing a Python code is that we will use HadoopStreaming for helping us passing data between our Map and Reduce code via STDIN (standard input) and STDOUT (standard output). We will simply use Python's sys.stdin to read input data and print our own output to sys.stdoutand the rest is taken care of by HadoopStreaming.

6.5.1 Hadoop Streaming

Hadoop streaming is a utility that comes with the Hadoop distribution. The utility allows you to create and run Map/Reduce jobs with any executable or script as the mapper and/or the reducer. For example:

```
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/hadoop-streaming.jar -input myInputDirs -output myOutputDir -mapper /bin/cat -reducer /bin/wc
```

How Streaming Works

In the above example, both the mapper and the reducer are executables that read the input from stdin (line by line) and emit the output to stdout. The utility will create a Map/Reduce job, submit the job to an appropriate cluster, and monitor the progress of the job until it completes.

When an executable is specified for mappers, each mapper task will launch the executable as a separate process when the mapper is initialized. As the mapper task runs, it converts its inputs into lines and feed the lines to the stdin of the process. In the meantime, the mapper collects the line oriented outputs from the stdout of the process and converts each line into a key/value pair, which is collected as the output of the mapper. By default, the prefix of a line up to the first tab character is the key and the rest of the line (excluding the tab character) will be the value. If there is no tab character in the line, then entire line is considered as key and the value is null. However, this can be customized, as discussed later.

When an executable is specified for reducers, each reducer task will launch the executable as a separate process then the reducer is initialized. As the reducer task runs, it converts its input key/values pairs into lines and feeds the lines to the stdin of the process. In the meantime, the reducer collects the line oriented outputs from the stdout of the process, converts each line into a key/value pair, which is collected as the output of the reducer. By default, the prefix of a line up to the first tab character is the key and the rest of the line (excluding the tab character) is the value. However, this can be customized, as discussed later.

6.5.2 MapReduce Programs

- *Wordcount:* This application is implemented as mapper and reducer where the mapper outputs the key-value pair < <word>, 1>. The reducer sums up the values, which are the occurrence counts for each key.
- *Inverted index:* The map function parses each document, and emits a sequence of <word, document ID> pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a <word, list(document ID)> pair.

The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

- *Count of URL Access Frequency:* The map function processes logs of web page requests and outputs <URL, 1>. The reduce function adds together all values for the same URL and emits a <URL, total count> pair.
- *Merge sort:* The map function sorts the input values based on the merge sort algorithm. The reducer combines the mapper outputs.

Inverted Index Example

- MapperIndex.py

```
#! /usr/bin/python
import os
from sys import stdin
import re
doc_id = ‘’
for line in stdin:
 content , extra = line.split('\n')
        var = str(content).startswith('docid')
 if var:
        doc_id = content
 else:
        words = re.findall(r'\w+', content)
        for word in words:
                if(len(doc_id)>1):
                        print("%s\t%s:1" % (word.lower(), doc_id))
```

- ReducerIndex.py

```
#! /usr/bin/python
import os
from sys import stdin
import re
```

```
index = {}
for line in stdin:
    word, postings = line.split('\t')
 index.setdefault(word, {})
    for posting in postings.split(','):
            doc_id, count = posting.split(':')
            count = int(count)
            index[word].setdefault(doc_id, 0)
            index[word][doc_id] += count
 for word in index:
    postings_list = ["%s:%d" % (doc_id, index[word][doc_id]) for doc_id in index[word]]
      postings = ','.join(postings_list)
      print('%s\t%s' % (word, postings))
```

Chapter 7

Testing

In order to carry out tests on the cluster we developed different data-intensive applications. Each of the applications is run with different sizes of data sets. The time taken to complete the jobs is noted and a graphical analysis is done. The same is repeated by changing the schedulers to Fair and Capacity schedulers.

7.1 MapReduce Examples

- Wordcount
- Inverted index
- Count of URL Access Frequency
- Merge sort

7.2 Comparisons of Schedulers

The tests were conducted extensively on a single node cluster, cluster of 3 nodes and a cluster of 10 nodes.

7.2.1 Single-node cluster statistics

- When a single job is submitted

Word Count Program:

The input given to the word count program was 3 e-books from the Gutenberg project of size 3.5 MB.

Scheduler	Execution time in seconds
FIFO	24
Fair Scheduler	32
Capacity Scheduler	38

Merge Sort Program:

The input to the merge sort consisted of a list of numbers. With different schedulers the following results were obtained.

Scheduler	Execution time in seconds
FIFO	27
Fair Scheduler	23
Capacity Scheduler	30

- When multiple jobs were submitted, the following statistics were obtained.

Word Count Program:

With FIFO Scheduler, in the first instance, 2 jobs were submitted one after the other in separate executions. In the second case, the jobs were submitted simultaneously in a single execution to FIFO, Fair and Capacity Schedulers. The size of the input data for the jobs was 25MB and 50 MB. The execution time tabulation is made as follows.

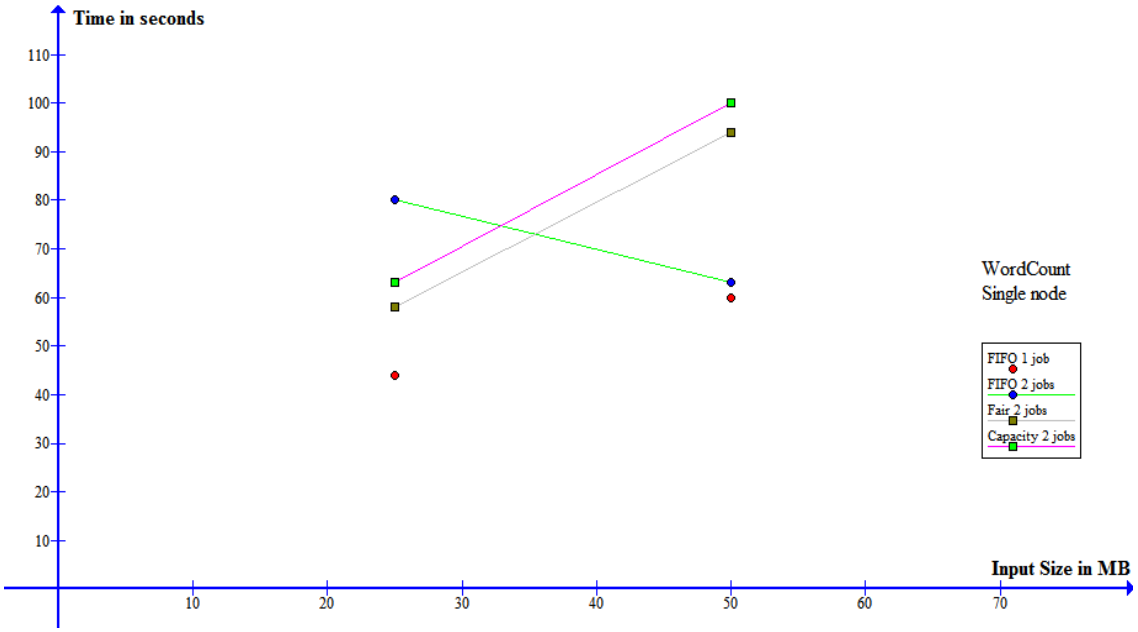
Case 1:

Job	Input Size	FIFO execution time in seconds
Job1	25MB	44
Job2	50MB	60

Case 2:

Job	Input Size	Execution time in seconds		
		FIFO	FAIR	CAPACITY
Job1	50MB	63	94	100
Job2	25MB	80	58	63

The graph below shows the different scenarios in which Wordcount program was run.



URL Count Program:

With FIFO Scheduler, in the first instance, 2 jobs were submitted one after the other in separate executions. In the second case, the jobs were submitted simultaneously in a single execution to FIFO, Fair and Capacity Schedulers. The size of the input data for the jobs was 25MB and 50 MB. The execution time tabulation is made as follows.

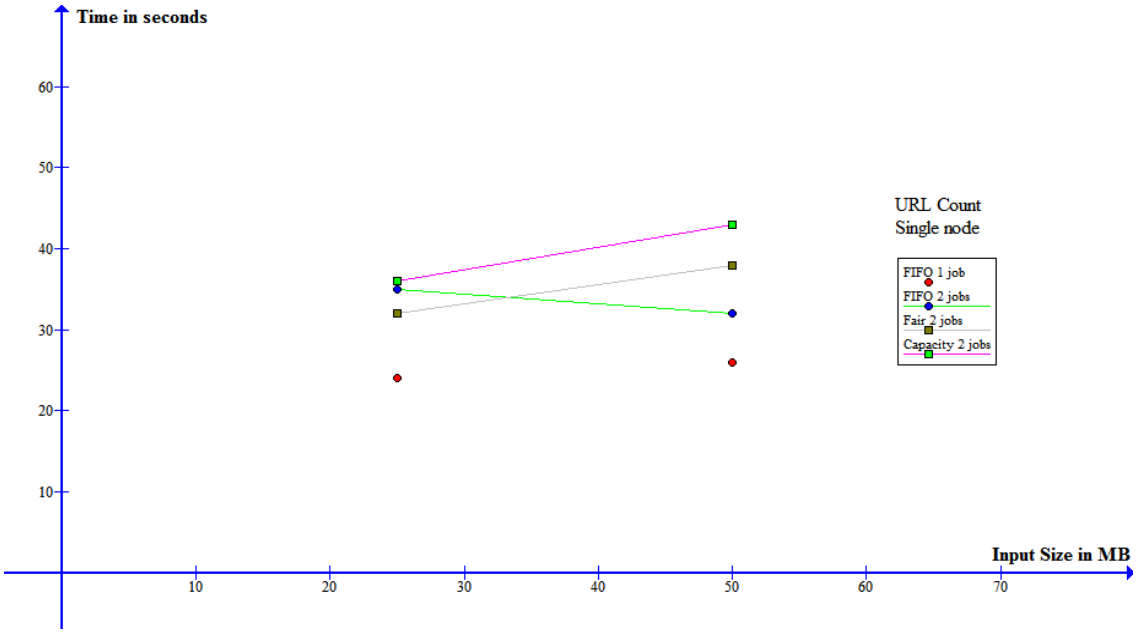
Case 1:

Job	Input Size	FIFO execution time in seconds
Job1	25MB	24
Job2	50MB	26

Case 2:

Job	Input Size	Execution time in seconds		
		FIFO	FAIR	CAPACITY
Job1	50MB	32	38	43
Job2	25MB	35	32	36

The graph below shows the different scenarios in which URL count program was run.



Merge Sort Program:

With FIFO Scheduler, in the first instance, 2 jobs were submitted one after the other in separate executions. In the second case, the jobs were submitted simultaneously in a single execution to FIFO, Fair and Capacity Schedulers. The size of the input data for the jobs was 25MB and 50 MB. The execution time tabulation is made as follows.

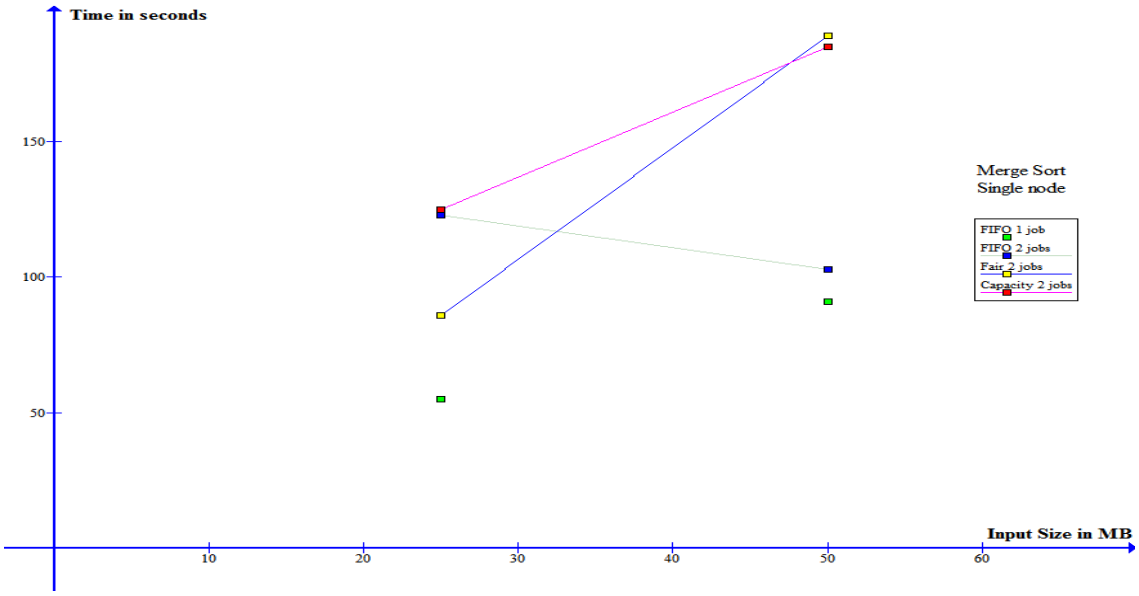
Case 1:

Job	Input Size	FIFO Execution time in seconds
Job1	50MB	91
Job2	25MB	55

Case 2:

Job	Input Size	Execution time in seconds		
		FIFO	FAIR	CAPACITY
Job1	50MB	103	189	185
Job2	25MB	123	86	125

The graph below shows the different scenarios in which Merge sort program was run.



Inverted Index Program:

With FIFO Scheduler, in the first instance, 2 jobs were submitted one after the other in separate executions. In the second case, the jobs were submitted simultaneously in a single execution to FIFO, Fair and Capacity Schedulers. The size of the input data for the jobs was 25MB and 50 MB. The execution time tabulation is made as follows.

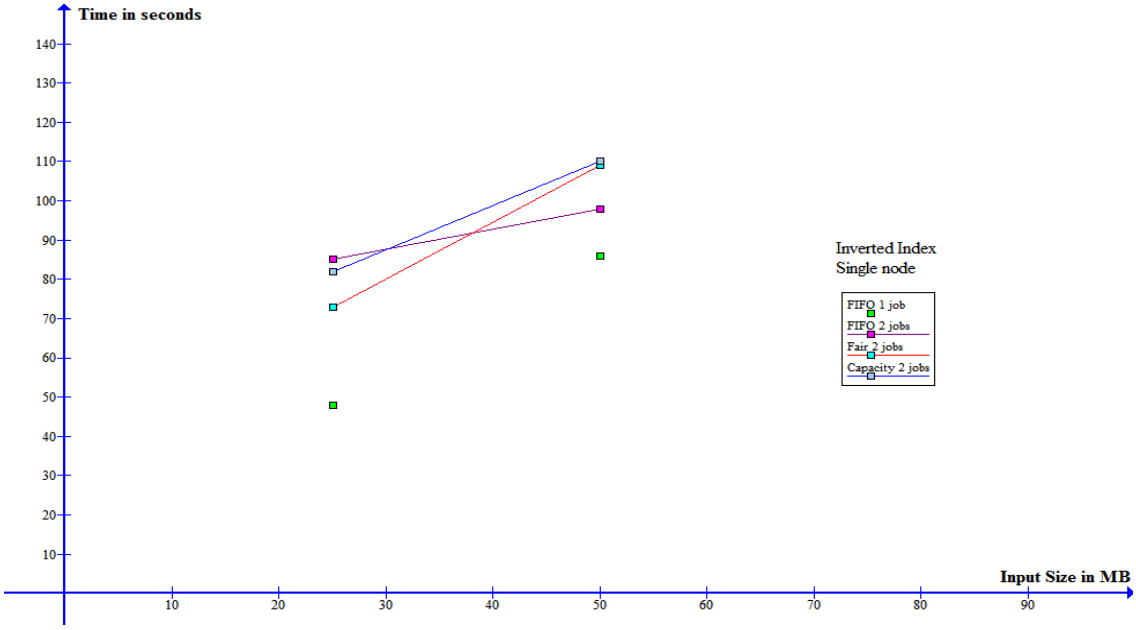
Case 1:

Job	Input Size	FIFO Execution time in seconds
Job1	50MB	86
Job2	25MB	48

Case 2:

Job	Input Size	Execution time in seconds		
		FIFO	FAIR	CAPACITY
Job1	50MB	98	109	110
Job2	25MB	85	73	82

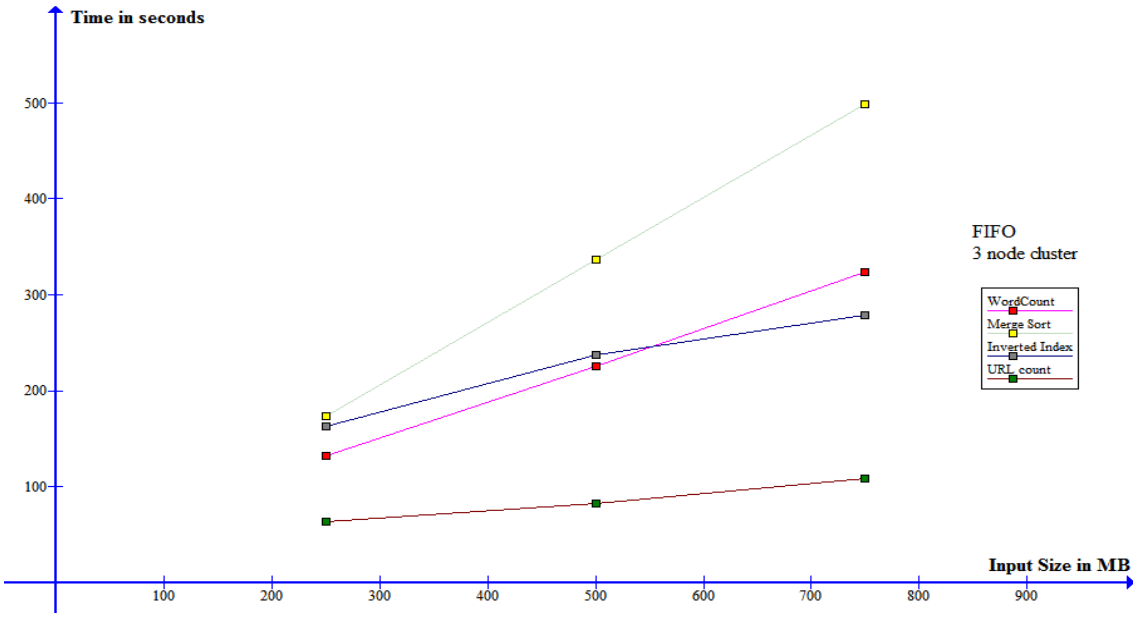
The graph below shows the different scenarios in which Inverted Index program was run.



7.2.2 Three-node Cluster Statistics

In FIFO Scheduling

The following two graphs show the time taken for job completion for wordcount ,url count , inverted index and merge sort applications when run using FIFO scheduler on a 3 node cluster.



In order to compare the FIFO results with that of Fair and Capacity Schedulers, we change the configurations files in Hadoop so that the scheduling policy is Fair and Capacity.

In Fair Scheduling:

- Submit two jobs. One job with data set of 750MB(job1) and the other of 250MB(job2).
- In FIFO though job2 is smaller it has to wait until job1 completes.
- In Fair both jobs are interleaved and hence time taken to complete the smaller job is less.

In Capacity Scheduling

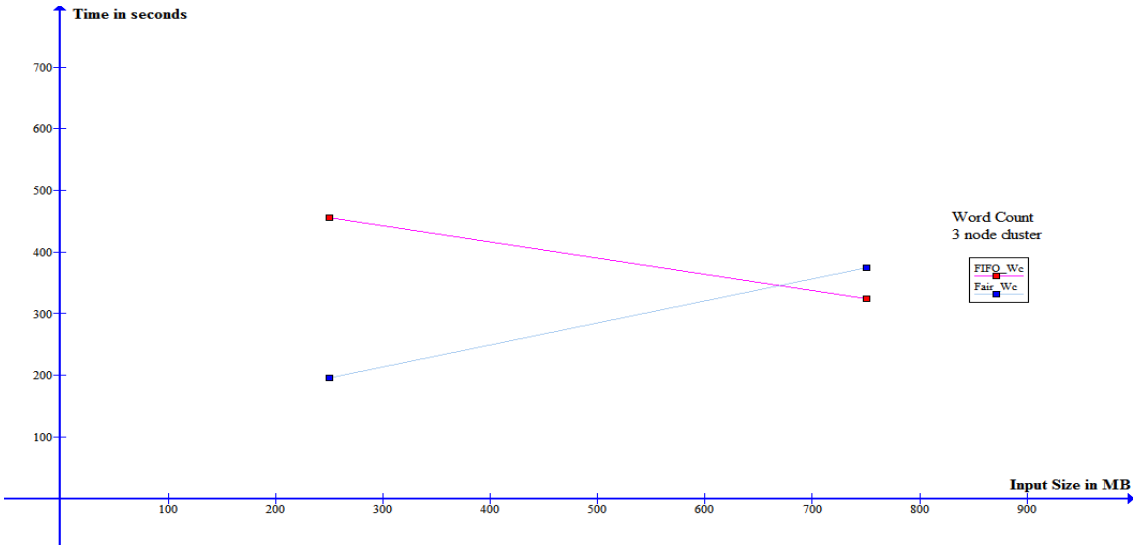
- Submit two jobs. One job with data set of 750MB(job1) and the other of 250MB(job2). Each job is sent to a different queue.
- In FIFO though job2 is smaller it has to wait until job1 completes.
- In Capacity both queues are interleaved and hence time taken to complete the jobs is less.

The following tables show the timings for FIFO vs Fair and FIFO vs Capacity Schedulers.

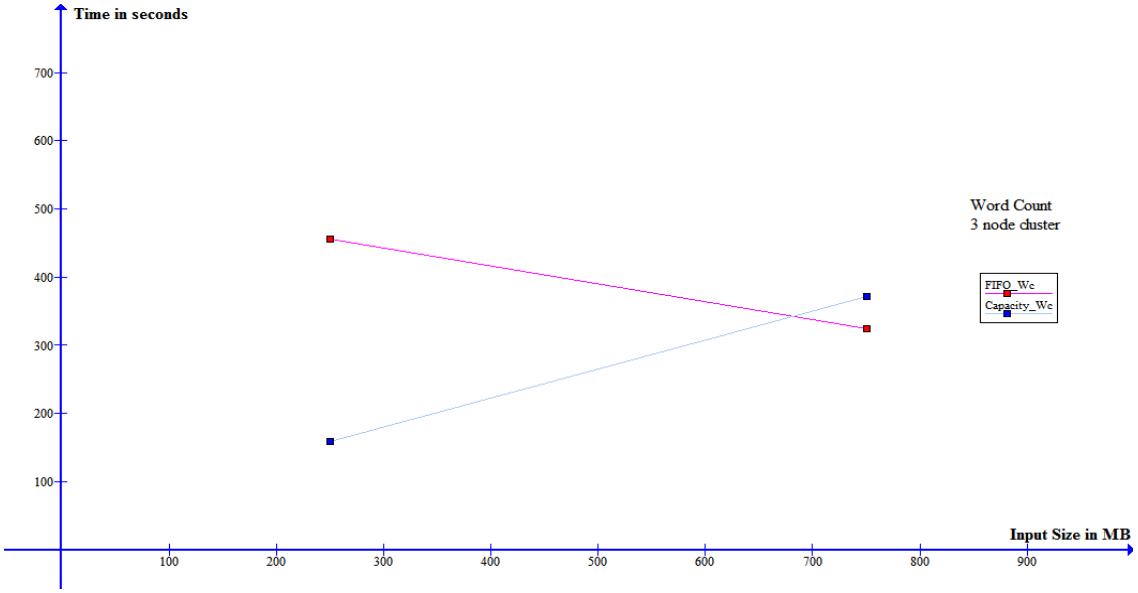
WordCount Program

Size	Scheduler	FIFO	FAIR	CAPACITY
750 MB		324	374	372
250 MB		456	196	159

The graph below shows the execution timings for FIFO and FAIR schedulers.



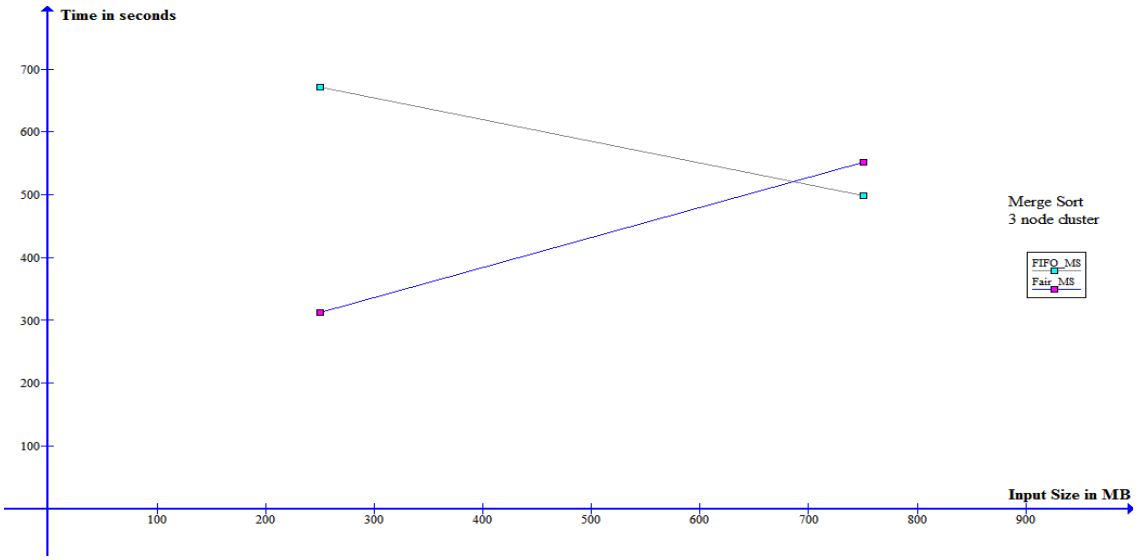
The graph below shows the execution timings for FIFO and CAPACITY schedulers.



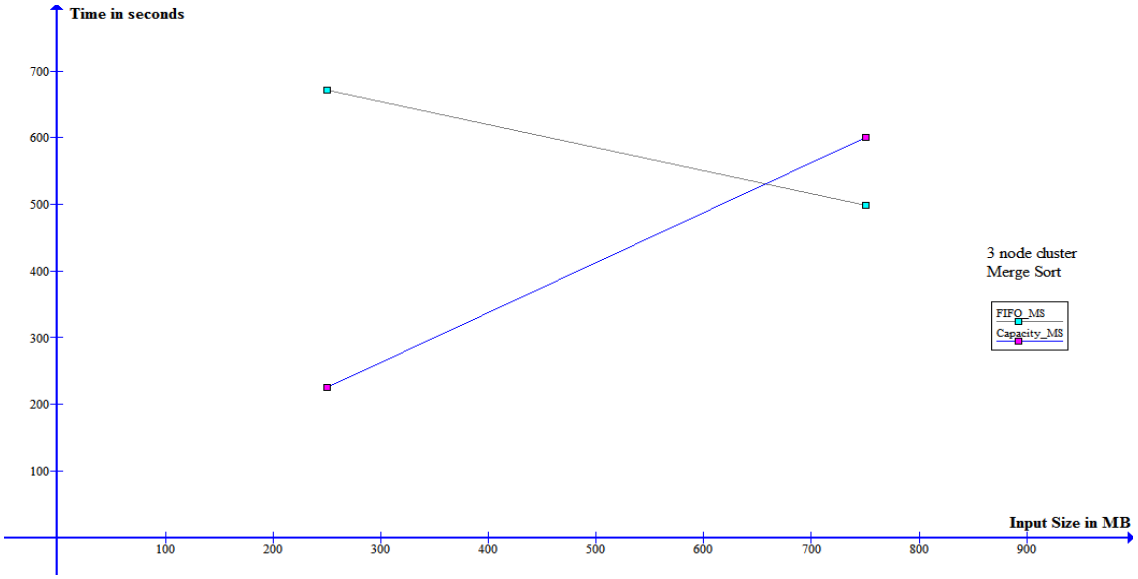
Merge sort program

Size	Scheduler	FIFO	FAIR	CAPACITY
750 MB		498	552	601
250 MB		672	312	225

The graph below shows the execution timings for FIFO and FAIR schedulers.



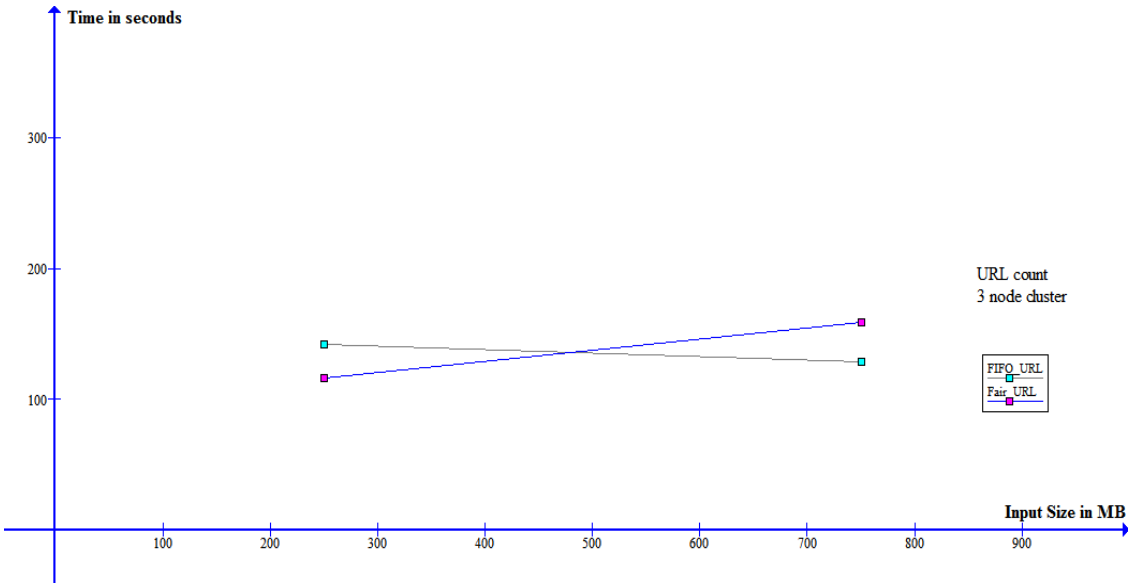
The graph below shows the execution timings for FIFO and CAPACITY schedulers.



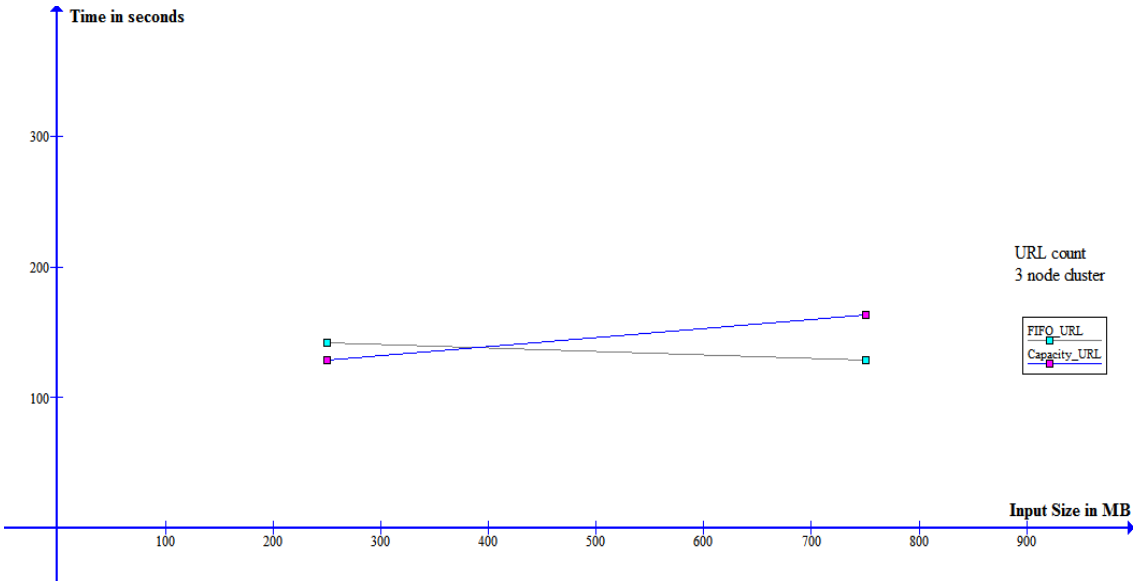
URL count program

Size	Scheduler	FIFO	FAIR	CAPACITY
750 MB		129	159	163
250 MB		142	116	129

The graph below shows the execution timings for FIFO and FAIR schedulers.



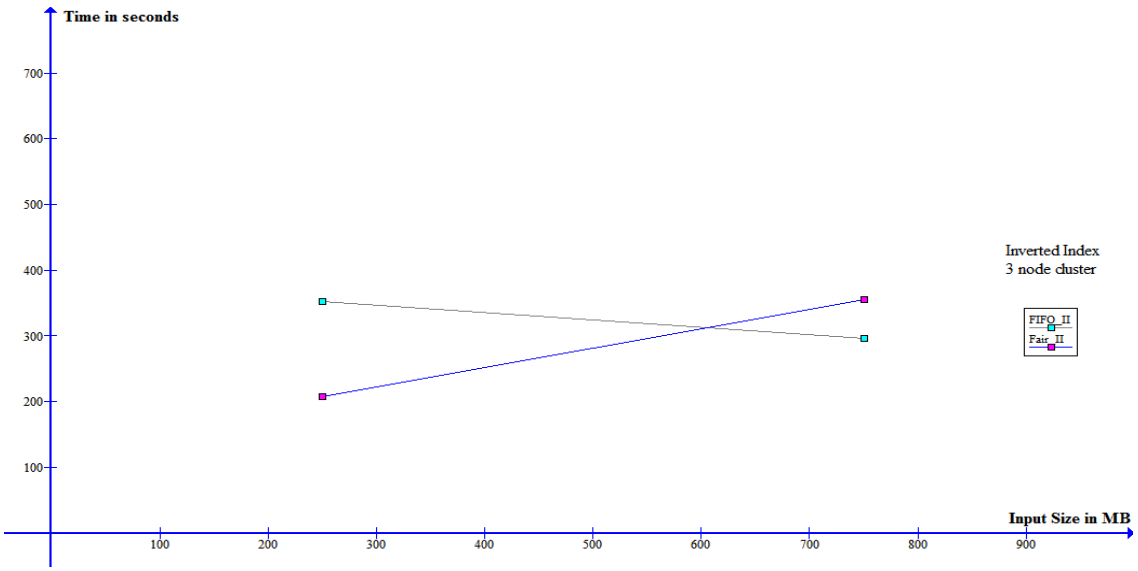
The graph below shows the execution timings for FIFO and CAPACITY schedulers.



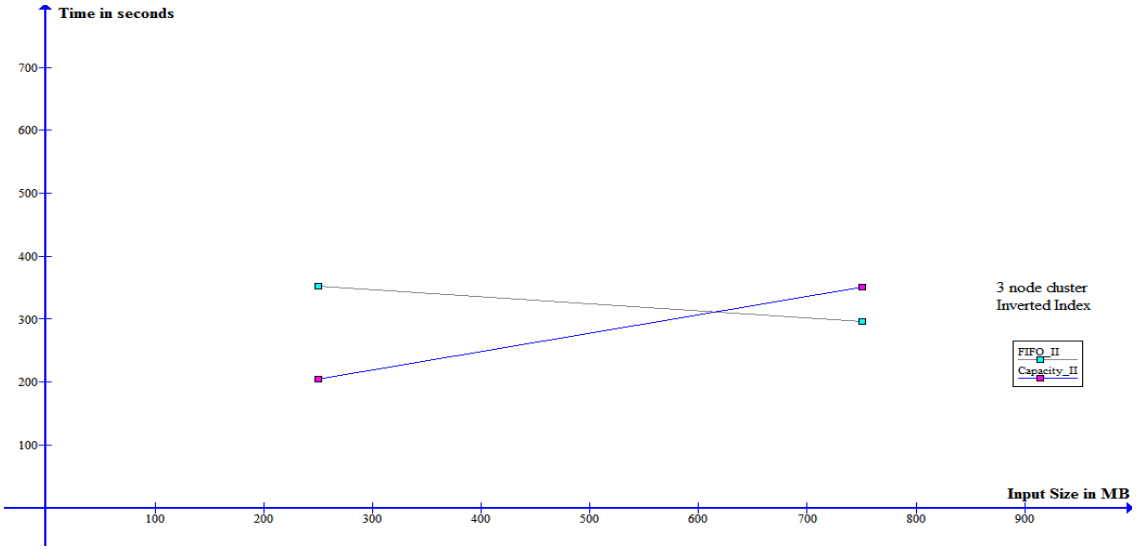
Inverted index program

Size	Scheduler	FIFO	FAIR	CAPACITY
750 MB		297	355	351
250 MB		353	207	204

The graph below shows the execution timings for FIFO and FAIR schedulers.



The graph below shows the execution timings for FIFO and CAPACITY schedulers.



6.2.3 Ten-node Cluster Statistics

In FIFO Scheduling

The following two graphs show the time taken for job completion for wordcount and merge sort applications when run using FIFO scheduler on a 10 node cluster.

- The results for wordcount program are as shown in the table below

Size in GB	Time in seconds
1.5	147
2	181
3	256

- The results for Merge Sort program are as shown in the table below

Size in GB	Time in seconds
1.5	270
2	344
3	524

In Fair Scheduling

- Submit two jobs. One job with data set of 3GB(job1) and the other of 1.5GB(job2).
- In FIFO though job2 is smaller it has to wait until job1 completes.
- In Fair both jobs are interleaved and hence time taken to complete the smaller job is less.

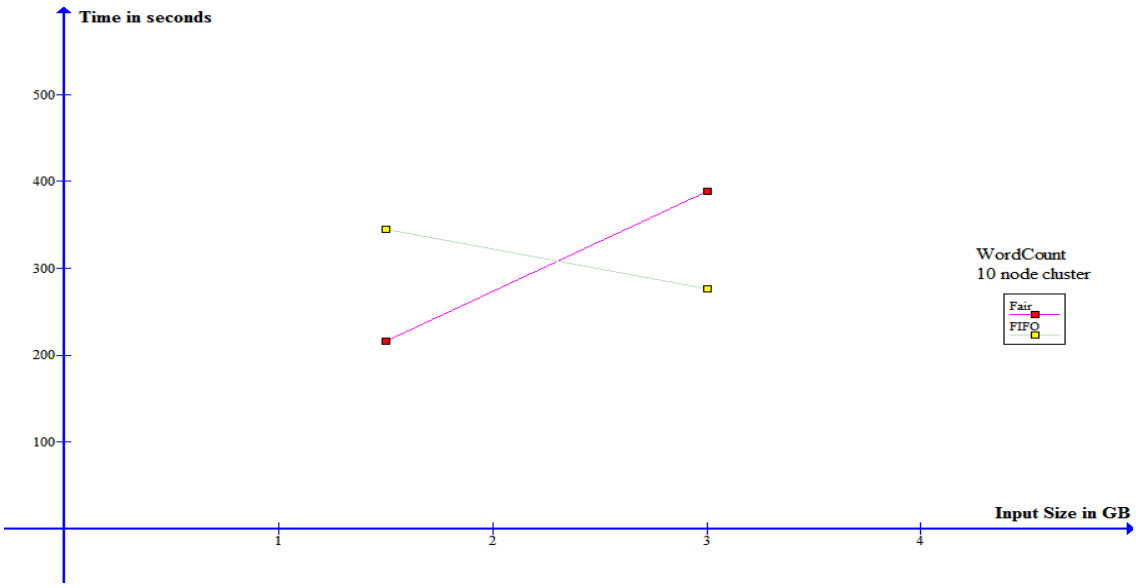
In Capacity Scheduling

- Submit two jobs. One job with data set of 3GB(job1) and the other of 1.5GB(job2). Each job is sent to a different queue.
- In FIFO though job2 is smaller it has to wait until job1 completes.
- In Capacity both queues are interleaved and hence time taken to complete the jobs is less.

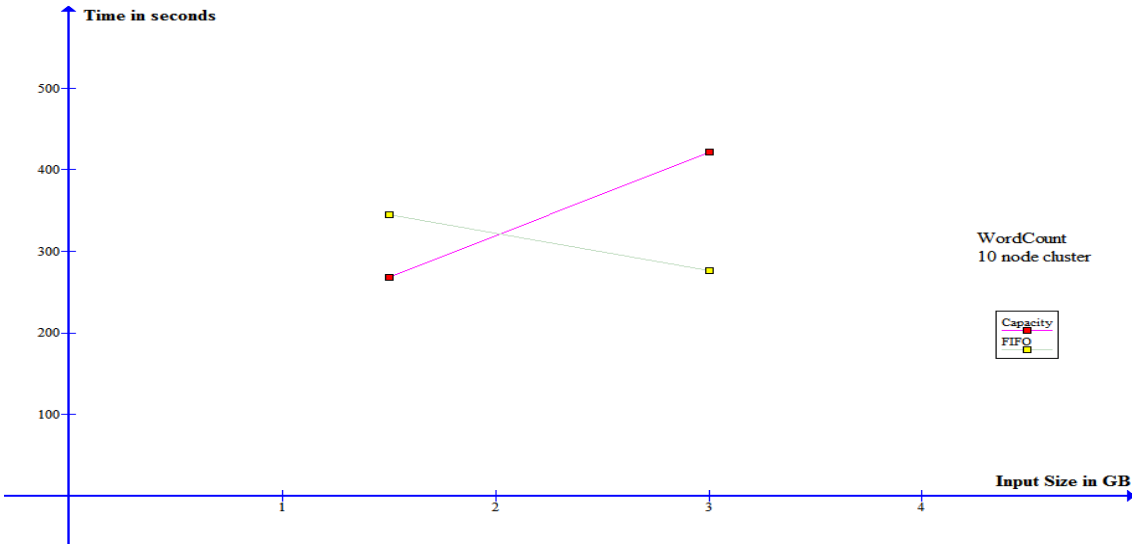
Wordcount program

Size	Scheduler	FIFO	FAIR	CAPACITY
1.5 GB		345	216	268
3 GB		276	388	421

The graph below shows the execution timings for FIFO and FAIR schedulers.



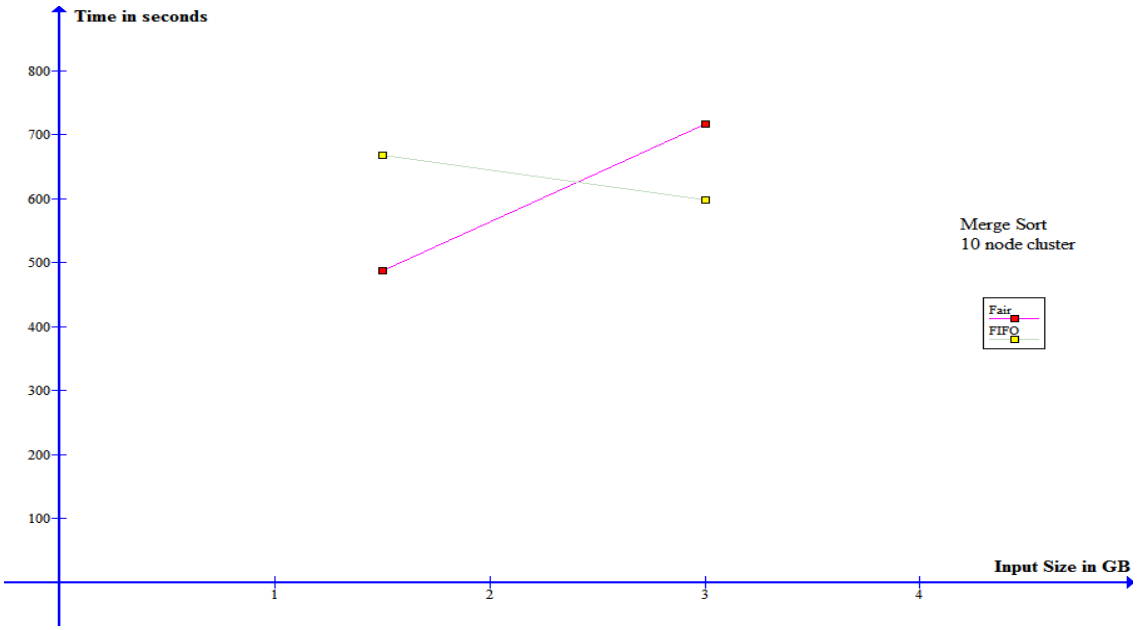
The graph below shows the execution timings for FIFO and CAPACITY schedulers.



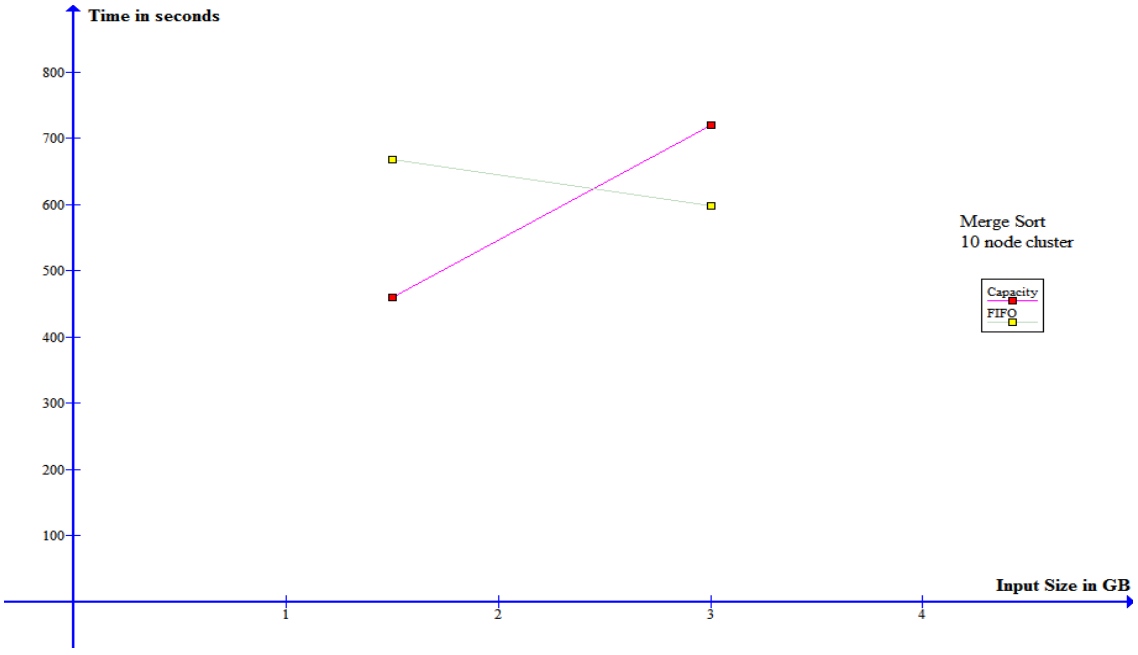
Merge sort program

Size	Scheduler	FIFO	FAIR	CAPACITY
1.5 GB		667	488	460
3 GB		597	717	720

The graph below shows the execution timings for FIFO and FAIR schedulers.



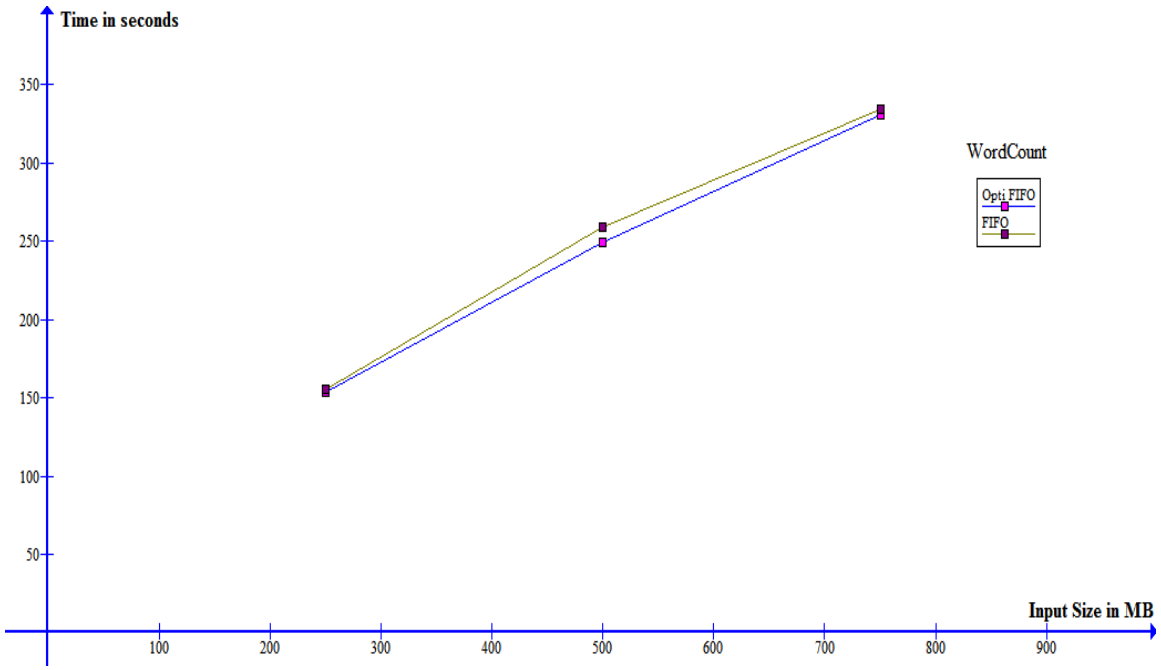
The graph below shows the execution timings for FIFO and CAPACITY schedulers.



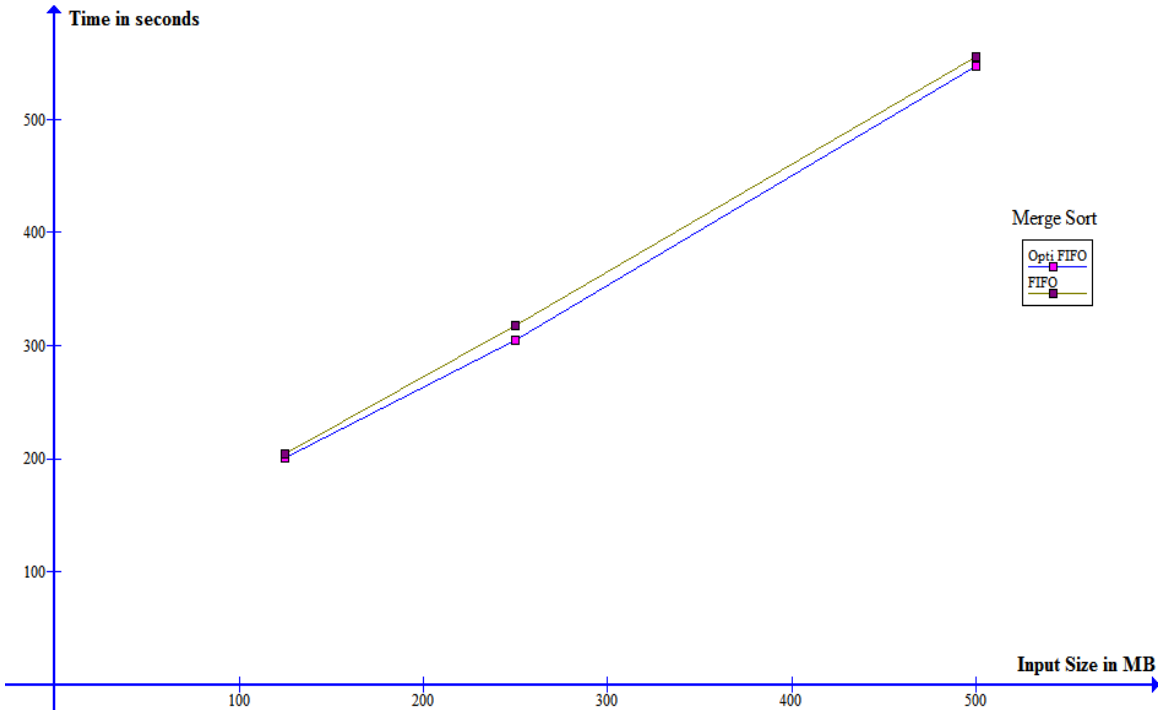
7.3 Testing the Optimized Hadoop Default Scheduler

The following graphs show the comparison in timing from the Hadoop default scheduler and our optimized Data locality aware scheduler.

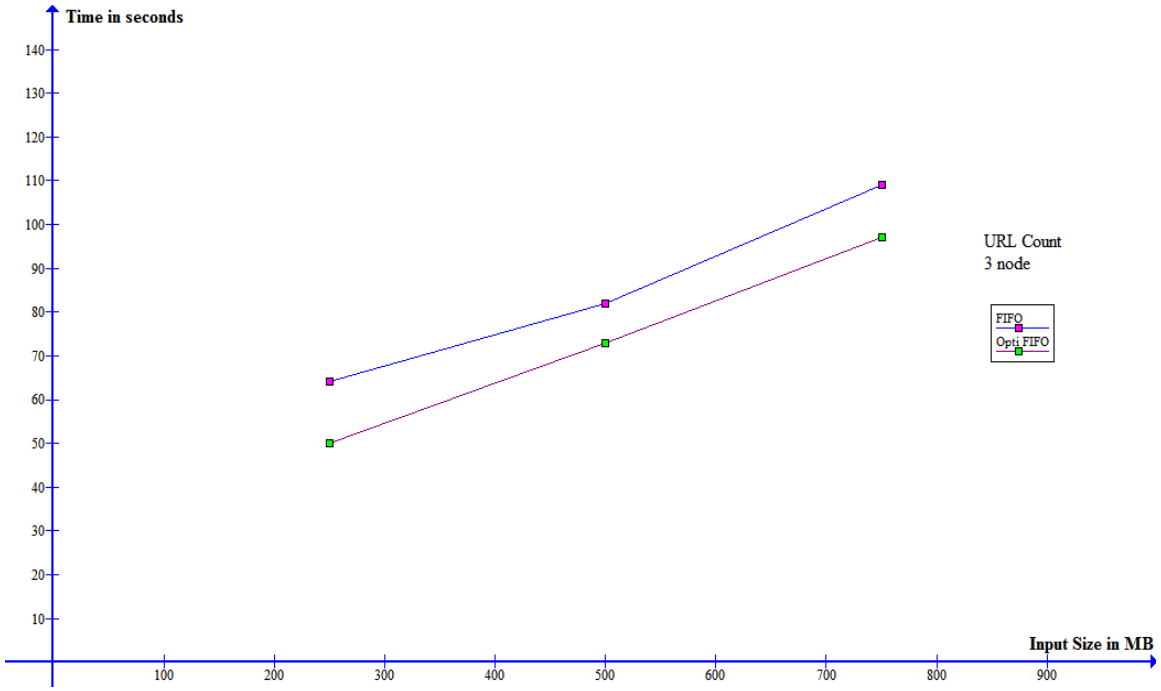
WordCount Program



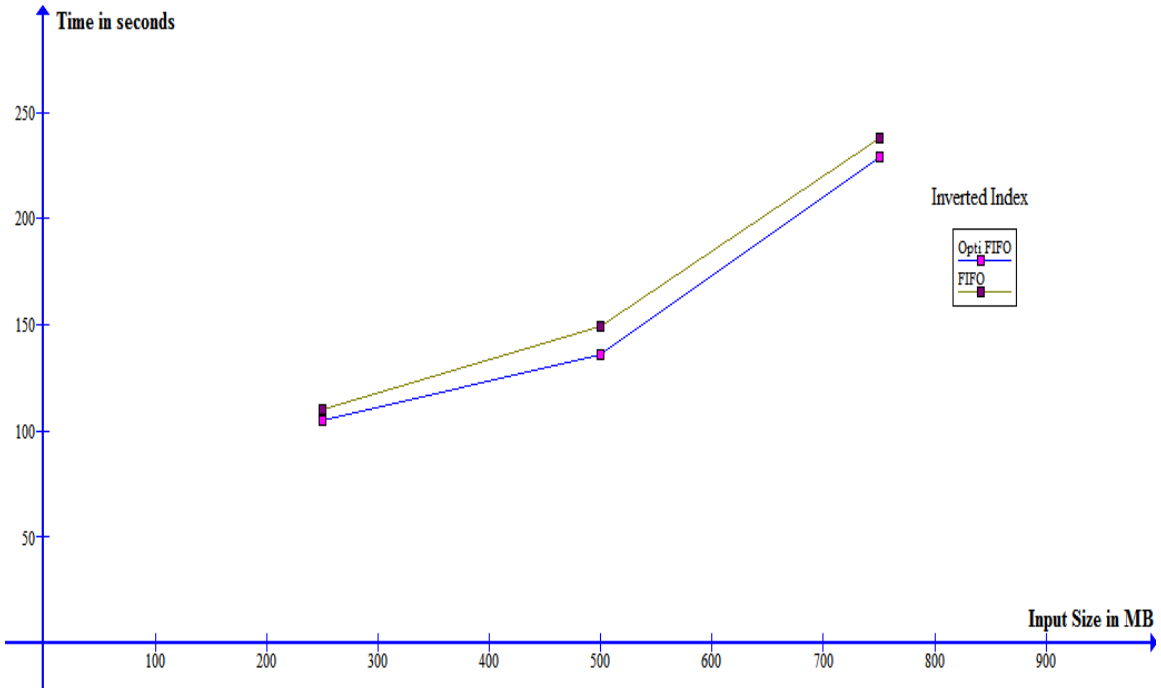
MergeSort Program



URL Count Program



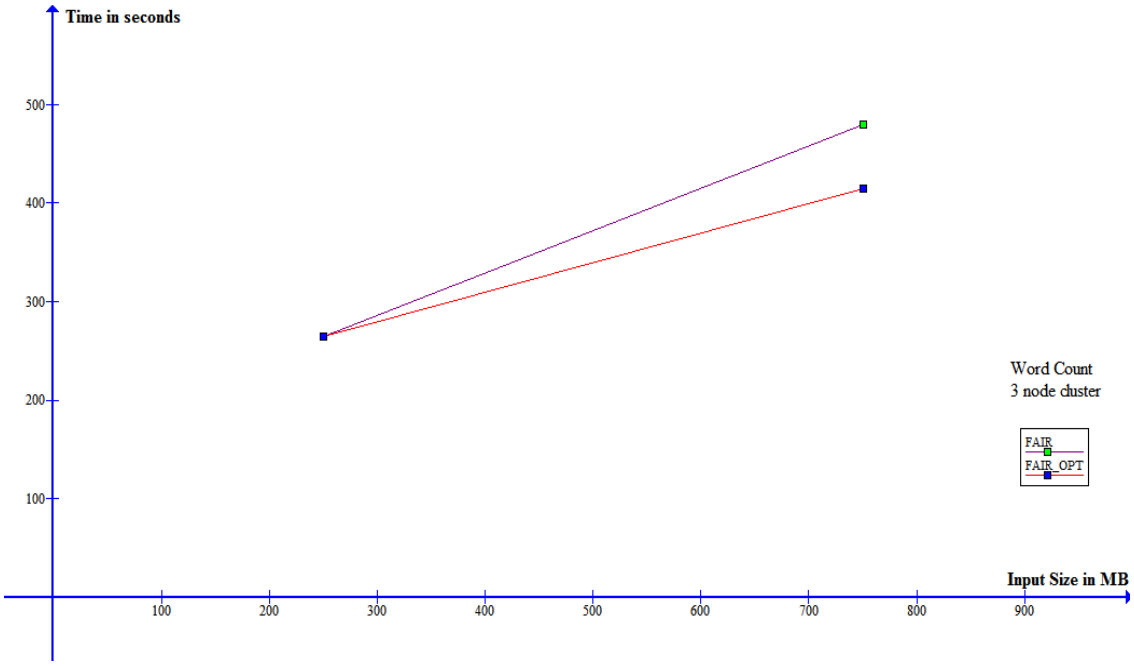
Inverted Index



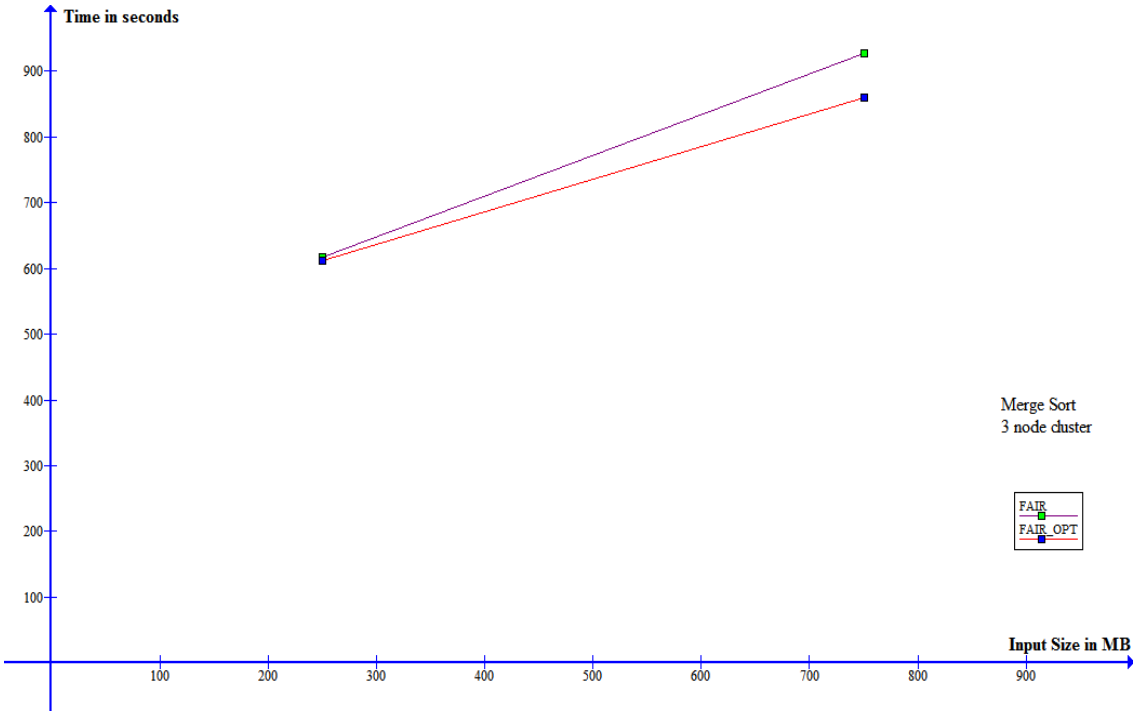
7.4 Testing the Optimized Fair Scheduler

The following graphs show the comparison in timing from the Hadoop default scheduler and our optimized Delay scheduler.

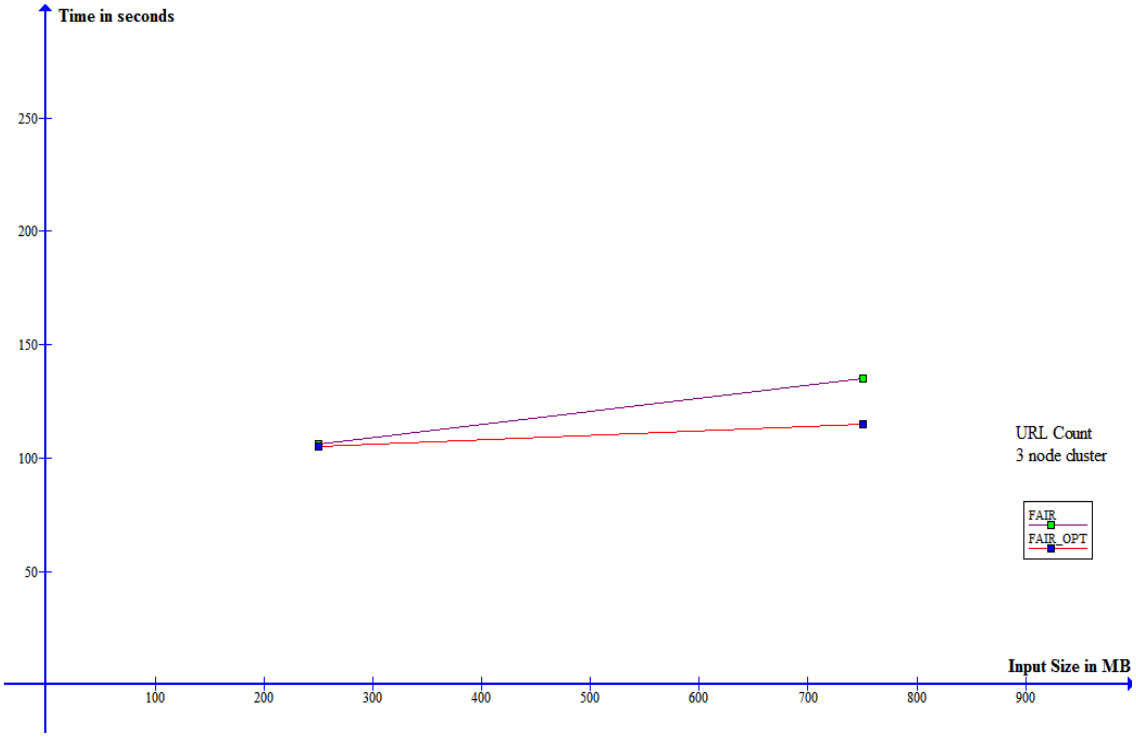
WordCount Program



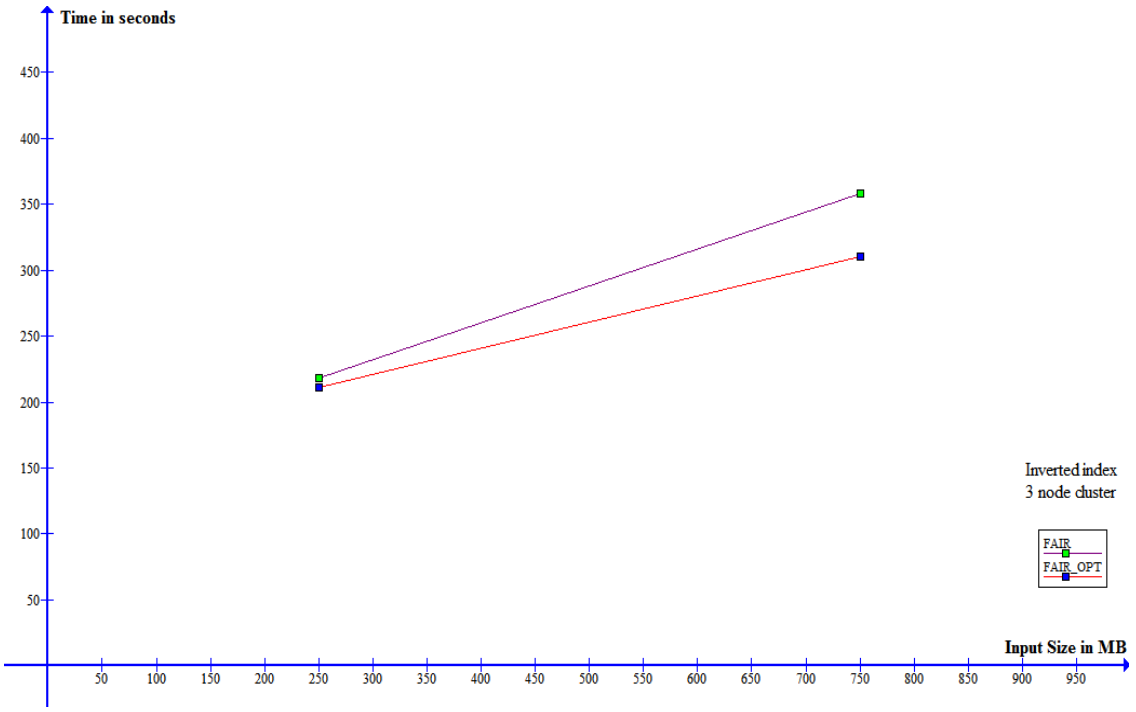
MergeSort Program



URL Count Program



Inverted Index



Chapter 8

Conclusion and Future Enhancements

From the data collected and analyzed we can infer that with respect to,

- FIFO and Fair Scheduler, the latter works better than the former when multiple jobs are submitted. Fair-share of resources are allocated to all the jobs hence the short jobs will finish fast whereas longer jobs are guaranteed not to get starved.
- FIFO and Capacity, the latter works well when there are many organizations or users sharing the cluster. Since each organization is assigned a queue with a minimum capacity guarantee, no organization is starved (FIFO) and there is effective use of resources.
- FIFO and Data Locality Aware Scheduler, the latter gives better scheduling results as it takes data locality into account when selecting tasks to schedule. By doing so, jobs are scheduled when their data is local to the node they are scheduled on hence reducing the time in data transfers.
- Fair and Delay Scheduler, the latter works better as it not only imbibes the good characteristics of fair scheduling, it also takes data locality into account. If data is not found on the local node, data is searched on the rack. This reduces the time taken to compete the job hence optimizing the performance of the scheduler.

The future enhancements to the project include:

- The scheduler in Hadoop can be enhanced to take into account the cost-effectiveness.
- The effectiveness of the scheduler can be tested by taking into account data skew-ness.

Chapter 9

References

[1] Apache Hadoop. http://hadoop.apache.org.

[2] Hadoop Map/Reduce tutorial, http://hadoop.apache.org/common/docs/current/mapred_tutorial.html

[3] Hadoop cluster setup tutorial, http://hadoop.apache.org/common/docs/current/cluster_setup.html

[4] Hadoop Capacity Scheduler tutorial, http://hadoop.apache.org/common/docs/current/capacity_scheduler.html

[5] Hadoop Fair Scheduler tutorial, http://hadoop.apache.org/common/docs/current/fair_scheduler.html

[6] T. White, Hadoop: The Definitive Guide. O'Reilly Media, Yahoo! Press, June 5, 2009.

[7] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, Ion Stoica, Delay Scheduling: A simple Technique for Achieving Locality ad Fairness in Cluster Scheduling, IEEE Paper,2010

[8] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, The Hadoop Distributed File System, IEEE paper, 2010

[9] Hadoop cluster setup, http://www.michael-noll.com/tutorials.

[10] HDFS Architecture, http://hadoop.apache.org/common/docs/current/hdfs_design.html

[11] Hadoop Fair Scheduler Design Document, August 15, 2009

[12] Mark Baker, University of Portsmouth, UK, Cluster Computing White Paper, Status – Final Release Version 2.0, 28th December 2000

[13] Mark Baker, Amy Apon, Rajkumar Buyya, Hai Jin, Cluster Computing and Applications, 18th September 2000

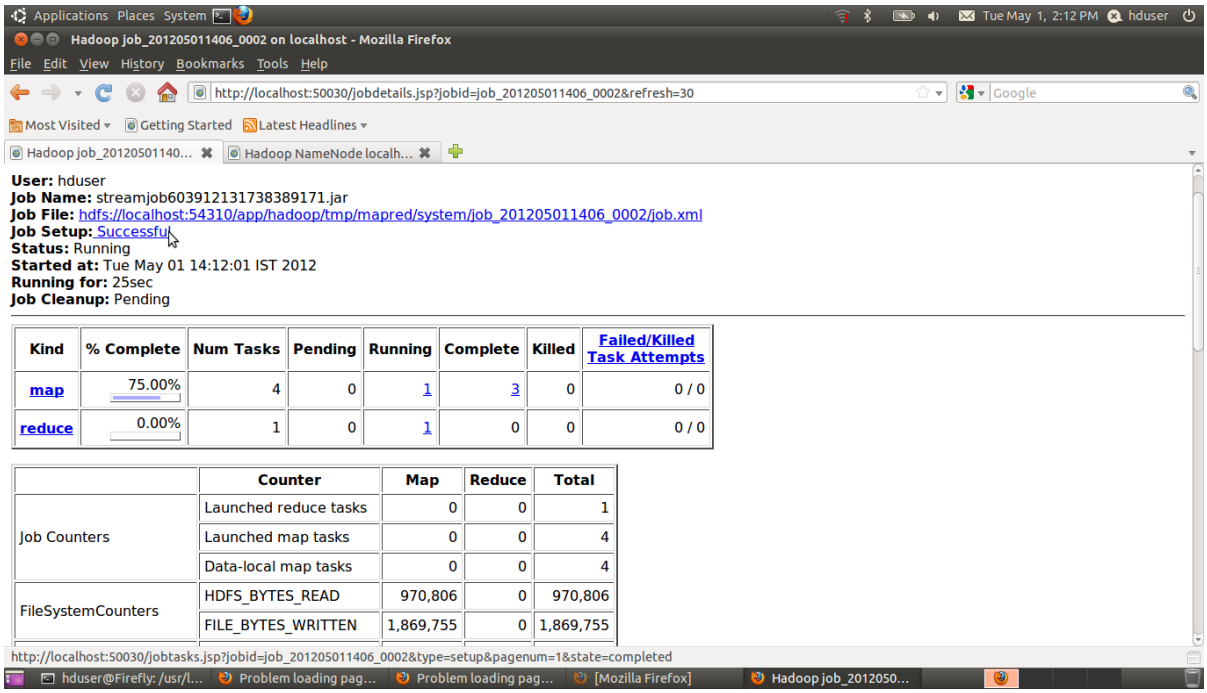
Chapter 10

Screenshots

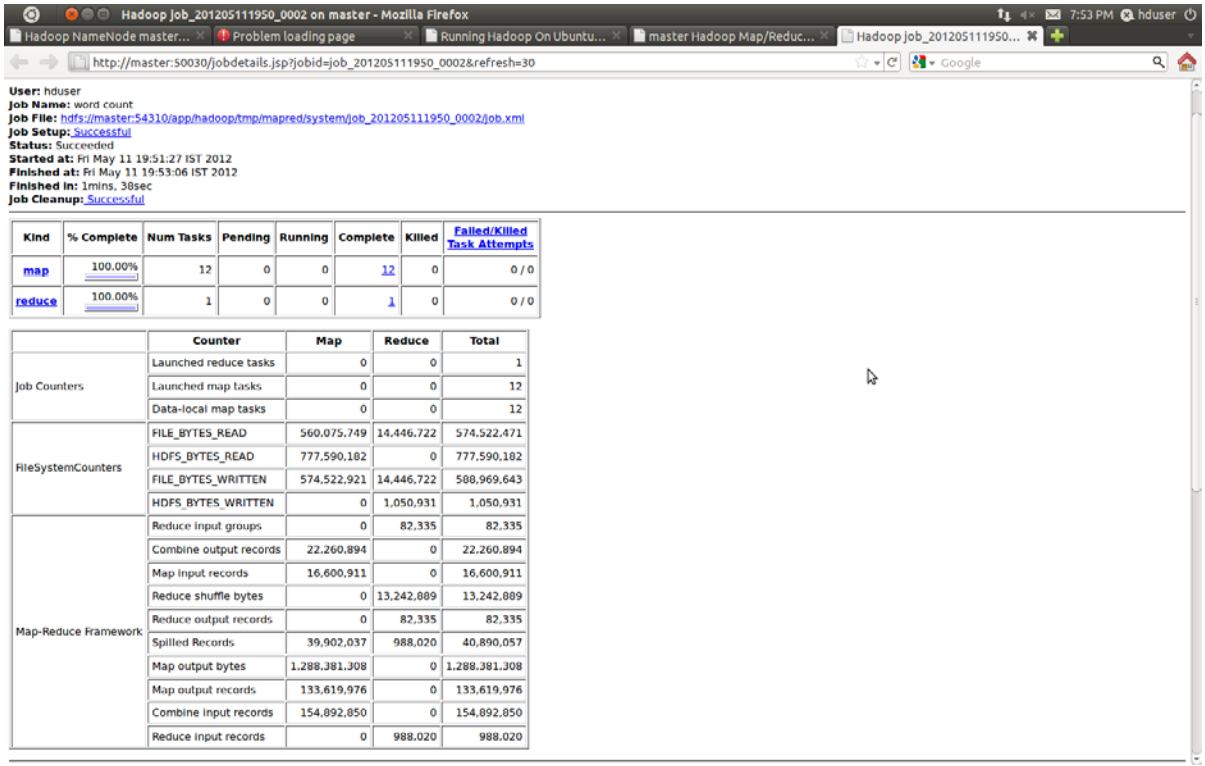
- Job Progress as seen on the terminal

```
hduser@ubuntu:/usr/local/hadoop$ bin/hadoop jar hadoop*examples*.jar wordcount /user/hduser
10/05/08 17:43:00 INFO input.FileInputFormat: Total input paths to process : 3
10/05/08 17:43:01 INFO mapred.JobClient: Running job: job_201005081732_0001
10/05/08 17:43:02 INFO mapred.JobClient:  map 0% reduce 0%
10/05/08 17:43:14 INFO mapred.JobClient:  map 66% reduce 0%
10/05/08 17:43:17 INFO mapred.JobClient:  map 100% reduce 0%
10/05/08 17:43:26 INFO mapred.JobClient:  map 100% reduce 100%
10/05/08 17:43:28 INFO mapred.JobClient: Job complete: job_201005081732_0001
10/05/08 17:43:28 INFO mapred.JobClient: Counters: 17
10/05/08 17:43:28 INFO mapred.JobClient:   Job Counters
10/05/08 17:43:28 INFO mapred.JobClient:     Launched reduce tasks=1
10/05/08 17:43:28 INFO mapred.JobClient:     Launched map tasks=3
10/05/08 17:43:28 INFO mapred.JobClient:     Data-local map tasks=3
10/05/08 17:43:28 INFO mapred.JobClient:   FileSystemCounters
10/05/08 17:43:28 INFO mapred.JobClient:     FILE_BYTES_READ=2214026
10/05/08 17:43:28 INFO mapred.JobClient:     HDFS_BYTES_READ=3639512
10/05/08 17:43:28 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=3687918
10/05/08 17:43:28 INFO mapred.JobClient:     HDFS_BYTES_WRITTEN=880330
10/05/08 17:43:28 INFO mapred.JobClient:   Map-Reduce Framework
10/05/08 17:43:28 INFO mapred.JobClient:     Reduce input groups=82290
10/05/08 17:43:28 INFO mapred.JobClient:     Combine output records=102286
10/05/08 17:43:28 INFO mapred.JobClient:     Map input records=77934
10/05/08 17:43:28 INFO mapred.JobClient:     Reduce shuffle bytes=1473796
10/05/08 17:43:28 INFO mapred.JobClient:     Reduce output records=82290
10/05/08 17:43:28 INFO mapred.JobClient:     Spilled Records=255874
10/05/08 17:43:28 INFO mapred.JobClient:     Map output bytes=6076267
10/05/08 17:43:28 INFO mapred.JobClient:     Combine input records=629187
10/05/08 17:43:28 INFO mapred.JobClient:     Map output records=629187
10/05/08 17:43:28 INFO mapred.JobClient:     Reduce input records=102286
```

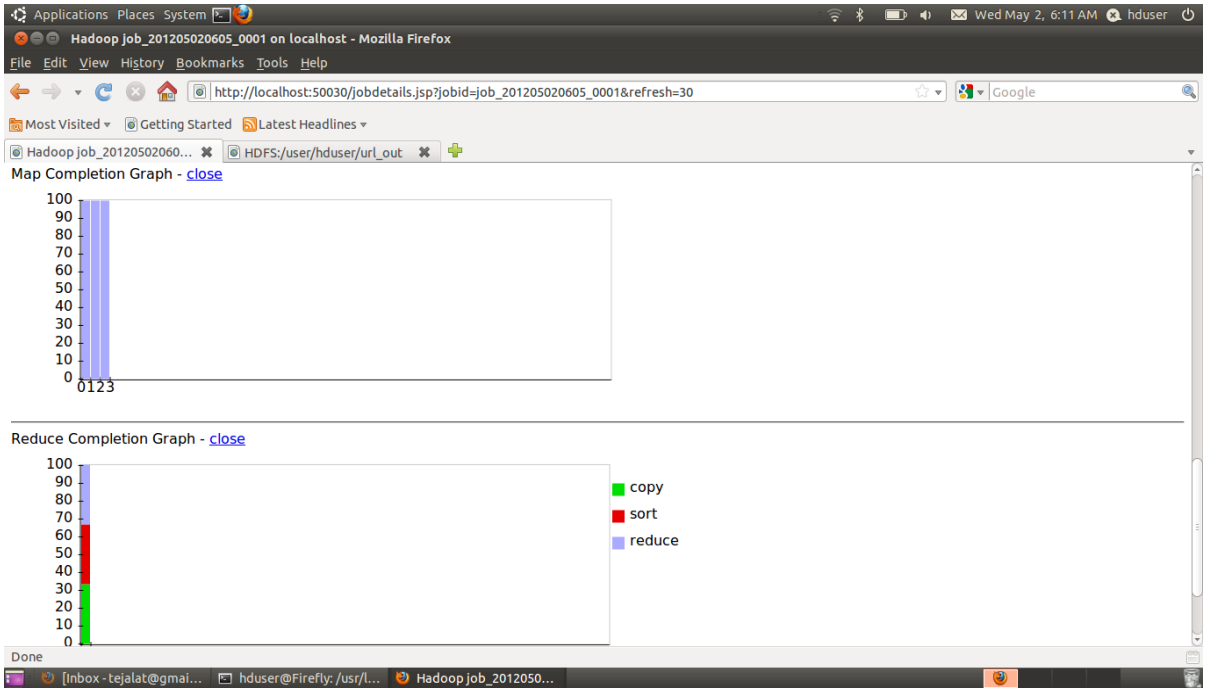
- Screenshot of a job in progress



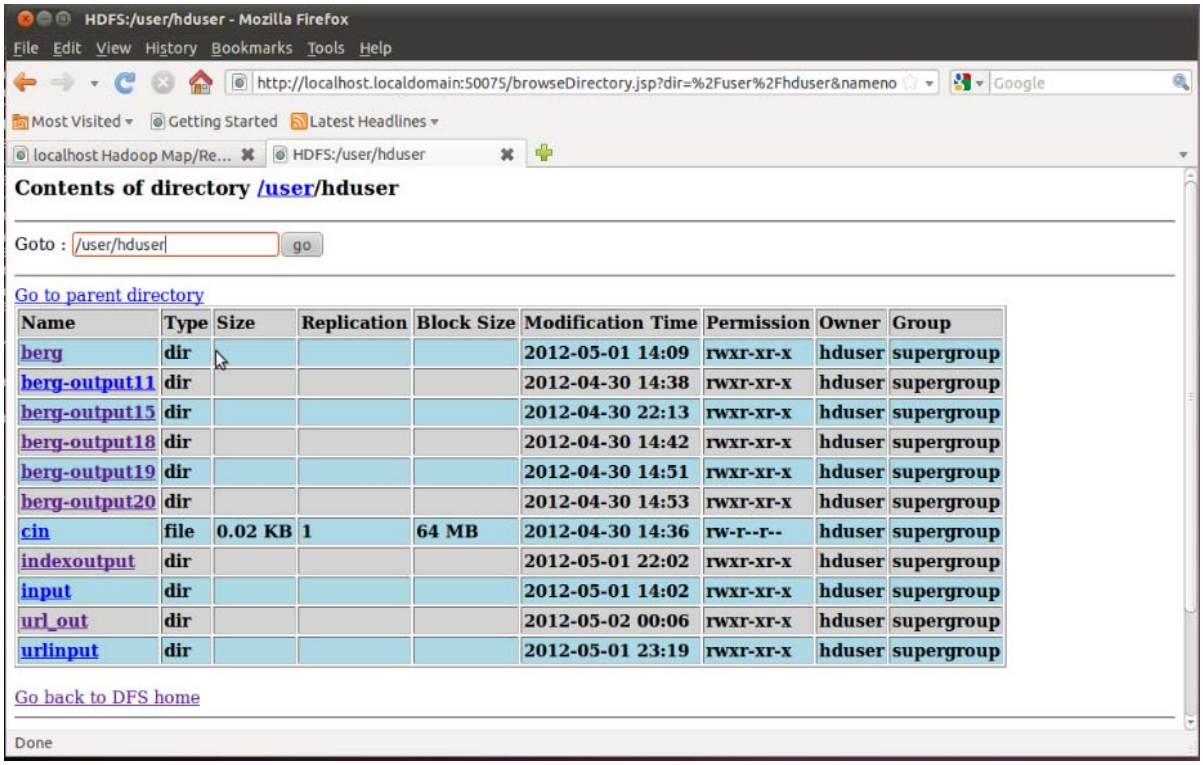
- Screenshot of a completed job



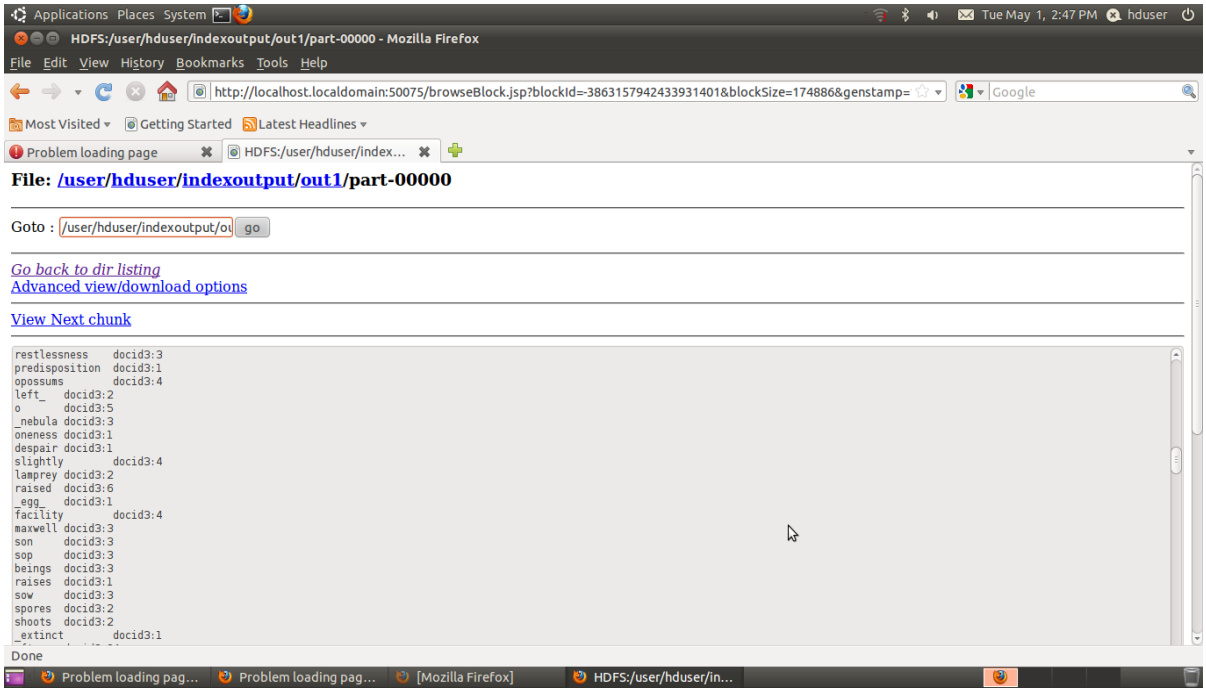
- Screenshot of the graphical representation of the Map and Reduce



- Screenshot of the HDFS



- Screenshot of the output file generated


Applications Places System
Tue May 1, 2:47 PM hduser
HDFS:/user/hduser/indexoutput/out1/part-00000 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost.localdomain:50075/browseBlock.jsp?blockId=-3863157942433931401&blockSize=174886&genstamp=
Most Visited Getting Started Latest Headlines
Problem loading page
HDFS:/user/hduser/index...
File: /user/hduser/indexoutput/out1/part-00000
Goto : /user/hduser/indexoutput/ou go
Go back to dir listing
Advanced view/download options
View Next chunk
restlessness docid3:3
predisposition docid3:1
opossums docid3:4
left_ docid3:2
o docid3:5
_nebula docid3:3
oneness docid3:1
despair docid3:1
slightly docid3:4
lamprey docid3:2
raised docid3:6
egg docid3:1
facility docid3:4
maxwell docid3:3
son docid3:3
sop docid3:3
beings docid3:3
raises docid3:1
sow docid3:3
spores docid3:2
shoots docid3:2
_extinct docid3:1
Done
Problem loading pag...
Problem loading pag...
[Mozilla Firefox]
HDFS:/user/hduser/in...