UNIVERSITETET I OSLO
Department of Informatics

Performance Analysis of Job-scheduling in Multi-User Hadoop Clusters

Joachim Seilfaldet

August 2012

Performance Analysis of Job-scheduling in Multi-User Hadoop Clusters

Joachim Seilfaldet

August 1, 2012

Abstract

The last decade has shown a drastic increase in data generation on computer systems and on the Internet. With an increasing number of Internet users the combined data available online is enormous. While search engines try to index all the worlds information available online and business try to get insight in user patterns across their data systems to provide a better understanding of their users. With this rapid increase of data available a system to process all this information is needed. Building large cluster to processes these amounts of data can be costly. With the size of clusters reaching the thousands computers. Therefor companies and institutions look to colocate data on a single processing cluster. Optimizing the efficiency of a processing cluster is wanted.

Apache Hadoop started as a open source project designed to process data in a large scale with reliability and scalability in mind. this project started out as a batch processing framework to process single jobs at a time, recent development have turned to sharing Hadoop clusters with other people to utilize the available resources within a Hadoop cluster. A few Task Schedulers have been written to facilitate this need. Languages such as Pig and Hive have been developed to quickly write analytic queries over data.

This thesis will look at the feasibility of running a processing cluster with multiple concurrent jobs running on a single cluster. It will provide a description of the system set up and shed some light upon the different obstacles that occurs when using each Task Scheduler. What can you expect in terms of performace from a system shared between multiple users.

Findings from experiments indicate that there are considerable differences between the different task schedulers benchmarked in this thesis. So, choosing a task scheduler that fits your needs is essential, as differences have a large impact on performance.

Acknowledgments

I would like to thank my supervisor during this thesis, Aleksander Øhrn for his guidance and help through this work. Also I would like to thank Stein H. Danielsen and Finn Arne Gangstad for introducing me to this interesting subject.

Finally, I would like to thank my friends and family for the support.

Joachim Seilfaldet
University of Oslo
August, 2012

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

The recent decade has shown a spectacular increase in the number of Internet users across the world. Studies from the International Telecommunication Union (ITU) show that in 2001 only 8 percent of the worlds population had access to the Internet, 29.4 percent users in the developed countries and 2.8 percent from the developing countries[20]. In 2011 around 34 percent of worlds population were able to access the Internet. Continuing this trend we can see a tremendous increase in Internet use the next years. 34 percent of the world population counts over 2 billion people, which is a staggering number of users.

With 34 percent of the world connected to the Internet the amount of data generated by all these users is enormous. In a white paper published by Cisco they estimate that the worlds internet IP traffic will reach 966 exabytes on a global scale by 2015[11]. Users using the Internet leave behind a large quantity of machine-generated data, which can be web server logs, call detail records, network event logs and such. But users are not only generating machine-generated data. An increasing trend of sharing information from their social life on social networking sites have become popular. Mark Zuckerberg, CEO of worlds large social network site Facebook said in a product launch that they see users sharing twice as much each year[13]. Users are sharing opinions in forums and blogposts. Pictures and videos in large sharing sites dedicated for sharing video. Commenting and contributing on factual sites such as wikis.

Companies are seeking to gain valuable knowledge about these traffic patterns of all these users to build better services and to reduce operational costs. To give them a better insight in how the people are using their products large amounts of logs and usage patterns have to be processed to see a clearer usage pattern across their product. Companies have to use large clusters of computers to process all this data in a reasonable amount of time. Distributed computing solves this by running multiple computers analyzing the same data in parallel to speed up the processing time. Apache Hadoop is a commonly used software framework for reliable data processing of vast amounts of data.

Since Hadoop was originally intended to only be used by a single user simultaneously some adjustments needed to be made. Support for multiple users have been implemented with a pluggable job-scheduler and implementation of queues have aided in making Hadoop a multi-user system. When companies or institutions invest in large clusters of computers to be used as a Hadoop cluster. They want to utilize them to their maximum potential. By that, it means utilizing processing power across the cluster and networking resources within the cluster.

1.2 Problem definition

the open-source software Apache Hadoop[1]. Apache Hadoop was originally written to support the Apache Nutch project, which is a open-source web-search software project. It provided Nutch with a large scale data-processing framework used on content downloaded by Nutch. Apache Hadoop have in its infancy only been able to process a single distributed task at a time, but as the software has become more wide-spread companies and institutions wanted to utilize the Hadoop cluster to its full extent. Exploiting the fact that the clusters often stay idle as well as colocation of data on a single cluster saves both time and money. Running multiple jobs simultaneously has been made possible with the pluggable job-scheduler. Facebook developed the Fair Scheduler and Yahoo developed the Capacity Scheduler, both are schedulers that are designed to share a Hadoop cluster with multiple users. Their goal is to efficiently share a large Hadoop cluster between many groups within their organization. Having different jobs submitted to a single cluster pose different challenges, such as how jobs should be scheduled based on their characteristics.

The intention with this study is to gain knowledge about how the Apache Hadoop framework treats multiple simultaneously running jobs. The Multi-User job schedulers designed by Facebook and Yahoo will be evaluated by running simulated test runs with different workloads. When multiple users share a single cluster you can not expect all jobs to be of the same type. Users have both different types of applications as well as different sizes of datasets. For this study we will create three different scenarios.

First scenario will run a set of jobs with a large dataset which will use a considerable long time to process. This can be jobs from real world scenarios such as machine learning jobs for spam detection and fraud detection, data transformation jobs such as processing large amounts of logs from different applications. Search indexing is also one example of long running jobs with much data to process. The second scenario will run a set of small jobs

Second scenario will be a set of shorter jobs running. Examples of this can be Hive or Pig scripts which converts job to smaller MapReduce jobs.

Last scenario will run a mix ranging from small jobs to the larger ones. The following problem definitions will be the main goal of this thesis:

- How well do a Hadoop cluster handle running multiple concurrent jobs?

- How well can the multi-user job schedulers utilize a Hadoop cluster when running multiple jobs.

- How do different workloads affect the performance of the task schedulers? Will the difference in jobs ran affect the overall performance of the task schedulers?
- What obstacles can occur by running multiple MapReduce jobs simultaneously?
- Which is the most efficient scheduler? How fast do jobs complete under the same conditions in a multi-user setting?

Can a Hadoop cluster run multiple concurrent jobs efficiently with the newly developed schedulers designed around this problem. How do the multi-user job schedulers compare to the standard job schedulers which is default in Apache Hadoop.

1.3 Chapter overview

Chapter 2 – Background

In chapter 2 details about the Hadoop framework is given, with its supporting software such as HDFS, MapReduce, Pig and the three job-schedulers used.

Chapter 3 – Experiment Design and Implementation

In chapter 3 descriptions on how the performance analysis will be executed, with details on what key aspects to include to gain an insightful look into the multi-user aspect of Hadoop. Implementation details on how the benchmarks are created is described.

Chapter 4 – Testing and Results

Chapter 4 presents results from the performance analysis run in this thesis. Results are presented and discussed.

Chapter 5 – Summary

Chapter 5 concludes and wraps up the findings from this thesis. We also discuss some future work.

Chapter 2

Background

2.1 Big Data Problems

Last few years the term big data have exploded in the technology community. Numbers from the Google Trends application show an rapid increase of traffic on the word "Big data" worldwide [6]. With this increasing trend a set of new companies and solutions emerge around the concept of solving big data problems.

The term "Big data" is used to describe data sets which in size are in the petabytes order. These data sets are often put together in a complex way, where normal databases are not able to handle complexity. Data sets can be found in areas such as meteorology, genomics and biological and environmental research. But, these data sets can also be found in technology areas such as internet search indexing, analyzing logs gathered from application and services, data gathered by sensor networks, complex social data gathered from social networks, photo editing and video rendering and eCommerce. Data sets have been able to grow into the size they are today because of the decreasing cost of storing data across commodity hardware.

IBM estimates that every day the worlds population creates 2.5 quintillion bytes of data [15]. In petabytes this amounts to a staggering 2220 petabyte every day. This data comes from every possible source in the world, such as sensors that gather climate information, to all the likes made on Facebook and such. Such data need to be processed to gain insight in the data set. Insight into the data set means that you derive some data from the data set, which will be beneficiary to you in some way.

As the world of Big Data have been getting more popular, so have the open-source framework for reliable data processing on a set of commodity hardware. Google Trends show an increasing trend for the search term Hadoop worldwide [7]. The increasing trend of Hadoop is caused by companies adapting this technology into their technology stack. Companies are created around the Hadoop eco-system providing a easy and simple way to access and get insight of their "Big data" problems.

Comapnies such as Hortonworks, Cloudera, MapR and Datameer to name a few have created a business around this emerging technology[8][4][9][5]. They provide easy installation and pre-configured Hadoop clusters, tools for analyzing and visualizing the results from queries, tools to import data into the Hadoop clusters and educational courses to learn how to best utilize the Hadoop eco-system.

2.2 Apache Hadoop

Apache Hadoop is an open-source project for reliable, scalable and distributed computing. The project consist of three individual projects, namely Hadoop Common, Hadoop Distributed File System and Hadoop MapReduce, while other projects exists around these.

Hadoop Distributed File System will be described in greater detail in section 2.3 and Hadoop MapReduce will be described in section 2.4.

Hadoop Common will not be described as this is only a supporting library.

There do of course exist counterparts to the Hadoop framework. Some notable projects are Disco, Spark and Storm which all are map-reduce frameworks[18].

2.2.1 Hadoop-related subprojects

Many applications has been implemented on top of the Hadoop platform and to supplement it. Many of which can be used completely without the Hadoop framework. Listed below are a few projects related to Hadoop.

Avro™
: Avro is a data serialization system. It provides rich data structures, remote procedure calls (RPC), a container file to store persistent data and a fast compact binary data format.

Cassandra™
: Apache Cassandra is a distributed storage system for storing large amounts of data across many commodity servers, while providing high availability with no single point of failure. Cassandra is a key-value store system.

Chukwa™
: Apache Chukwa is a large-scale log collection and analysis framework built upon Hadoop. Distributed File System and MapReduce.

HBase™
: Apache HBase is a distributed column oriented database modeled after Googles BigTable [10]. Tables in HBase may be input and output for MapReduce jobs run on Hadoop.

Hive™
: Apache Hive is a data warehouse infrastructure. It is designed to provide data summarization, query and analysis on top of the Hadoop platform[19].

Mahout™
: Apache Mahout is a framework for distributed scalable machine learning algorithms. Most implementations are implemented using the MapReduce paradigm.

Pig™

Apache Pig is a platform for analyzing large data using a high-level language called Pig Latin. Pig programs are broken down to small MapReduce tasks, run on a Hadoop cluster.

ZooKeeper™

Apache Zookeeper is a distributed coordination service for distributed applications.

2.3 Hadoop Distributed File System

Hadoop Distributed File System (HDFS) is the primary file system used by Apache Hadoop and its sub-projects. It was originally written to support Apache Nutch[2], which is an open-source web-search project. HDFS is for the most part based upon on a large distributed file system designed at Google for their production systems[14], The Google File System (GFS). The file system is designed to run on a large set of commodity computers, to reduce computing costs. In a cluster of commodity computers it is expected that hardware failure happens frequently. This is why the file system has been implemented with security measurements such as data replication and fast cluster rebalancing to avoid data loss. The default replication factor of HDFS is set to 3. Data is replicated on three different DataNodes in the cluster. Data placement in Hadoop clusters happen with a rack aware placement strategy to improve data reliability even further.

A typical Hadoop cluster is shown in figure 2.1. This contains a single NameNode explained in section 2.3.1 and multiple data nodes explained further in section 2.3.4. In larger clusters there is a need for a backup in case the name node crashes. Details about the backup node can be read in section 2.3.3.

2.3.1 NameNode

Acting as the master, in the master/slave relationship within HDFS the NameNode has the most important job in the cluster. It maintains the

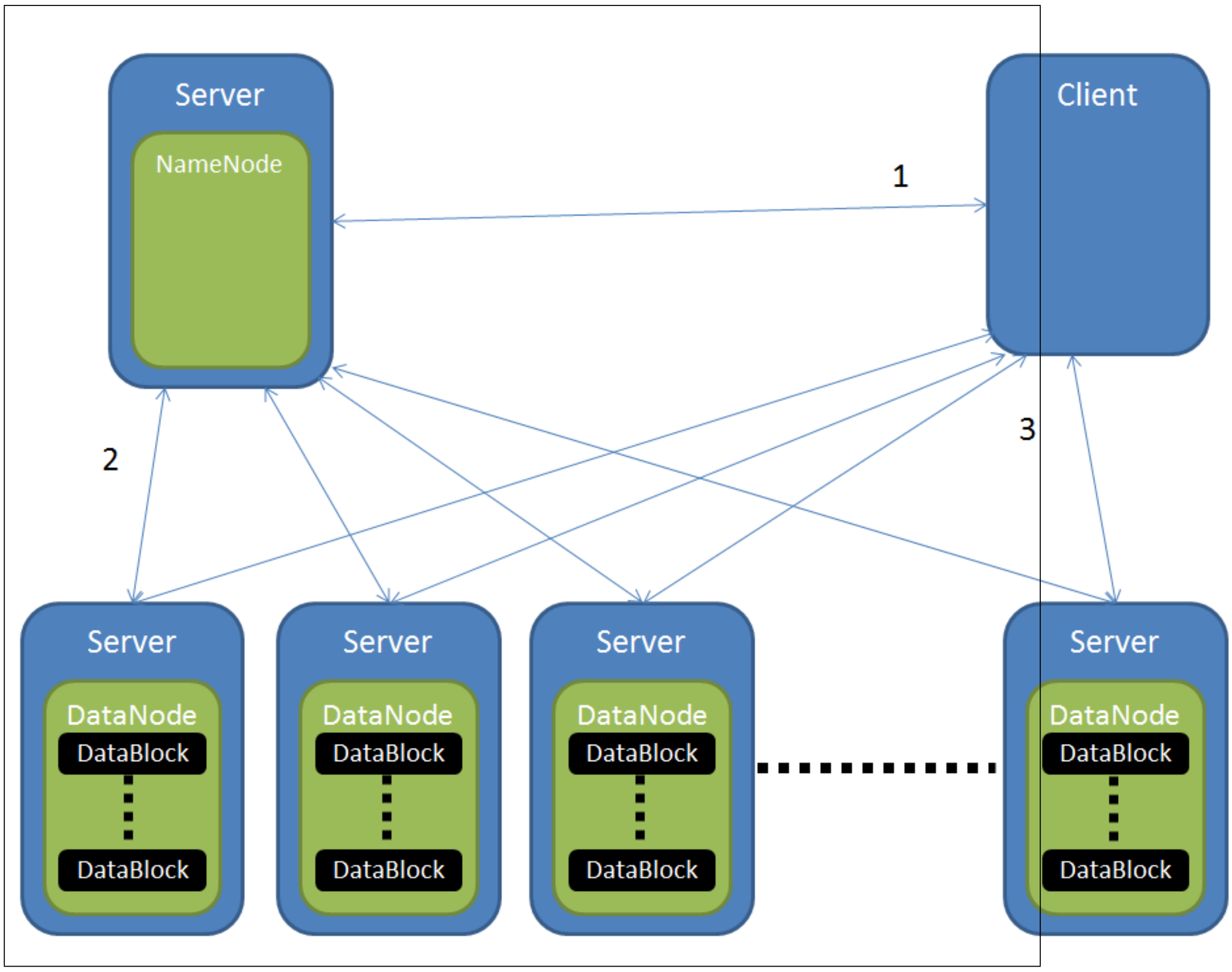

Figure 2.1: Overview of Hadoop Distributed File System

namespace for the entire file system across all nodes. The namespace in HDFS is organized in a traditional hierarchical file structure, which contains folders and files.

Maintaining the file system for a large cluster can be a cumbersome task, therefor the NameNode is kept on a separate computer than the data nodes. The name node is usually a computer with much memory as it keeps the entire namespace in memory at all time. This is done to increase the performance of file lookups.

Since there is only one NameNode in each cluster, this is a single point of failure. So, if the NameNode crashes the whole file system will be unavailable. To avoid this problem it is possible to have a secondary NameNode in the cluster called Checkpoint Node, explained in section 2.3.2 or a Backup Node explained in section 2.3.3

2.3.2 Checkpoint Node

Since the Name Node is a simple point of failure in HDFS, counter measurements such as the Checkpoint Node and Backup Node has been implemented.

The main responsiblilty of the Checkpoint Node is to download the namespace file fsimage and edits and merge them together and upload a fresh checkpoint file to NameNode.

The Checkpoint Node is designed to run on a separate machine from the Name Node in the Hadoop cluster. The checkpoint node will need to have the same amount of memory as Name Node, since their requirements in memory usage are equal.

2.3.3 BackupNode

Compared to the Checkpoint Node described in section 2.3.2 the Backup Node has many of the same responsibilities, with a few extra added. A added feature of the Backup Node is that it keeps a up-to-date copy of the file system namespace in its memory. It receives a stream of all edits made on the Name Node and then generates its own version of the namespace identical Name Nodes. In case of break down in the name node a fast switch to the backup node can be performed.

Since Backup Node keep a identical file system namespace in memory as Name Node it has the same requirements to the amount of memory.

2.3.4 DataNode

Hadoop File System is built up by a series of DataNodes located on different machines in a network. Each DataNode is responsible for storing files. The DataNode acts on commands from the NameNode which sends actions for the creation of blocks, deletion of blocks and replication of blocks.

When a file is read from the file system the NameNode gives the client the details on how to contact the appropriate DataNode for the block requested by the client. No data passes through the NameNode in any circumstance.

2.4 Hadoop MapReduce

Hadoop MapReduce is an open-source implementation of the MapReduce programming model introduced by Google, which is inspired by map reduce primitives from functional languages such as Lisp[12]. With it's relatively easy to use programming model, MapReduce has quickly become a mainstream way of handling large scale data processing.

MapReduce job splits input data into many pieces. MapReduce jobs are distributed throughput the cluster, so that data can be processed locally where data recides. Programs are written to be run be run in parallel. Jobs consist of Map and Reduce jobs.

Output from map jobs are sorted and sent as input for reduce jobs. A typical job is not finished before all reducers are finished up.

hadoop Mapreduce consist of two types of daemons running in the cluster. A master jobtracker and many slave tasktrackers. TaskTracker should be run on the same node as a DataNode from HDFS to achieve data locality. Jobtracker is explained in detail in section 2.4.1 and the tasktracker in section

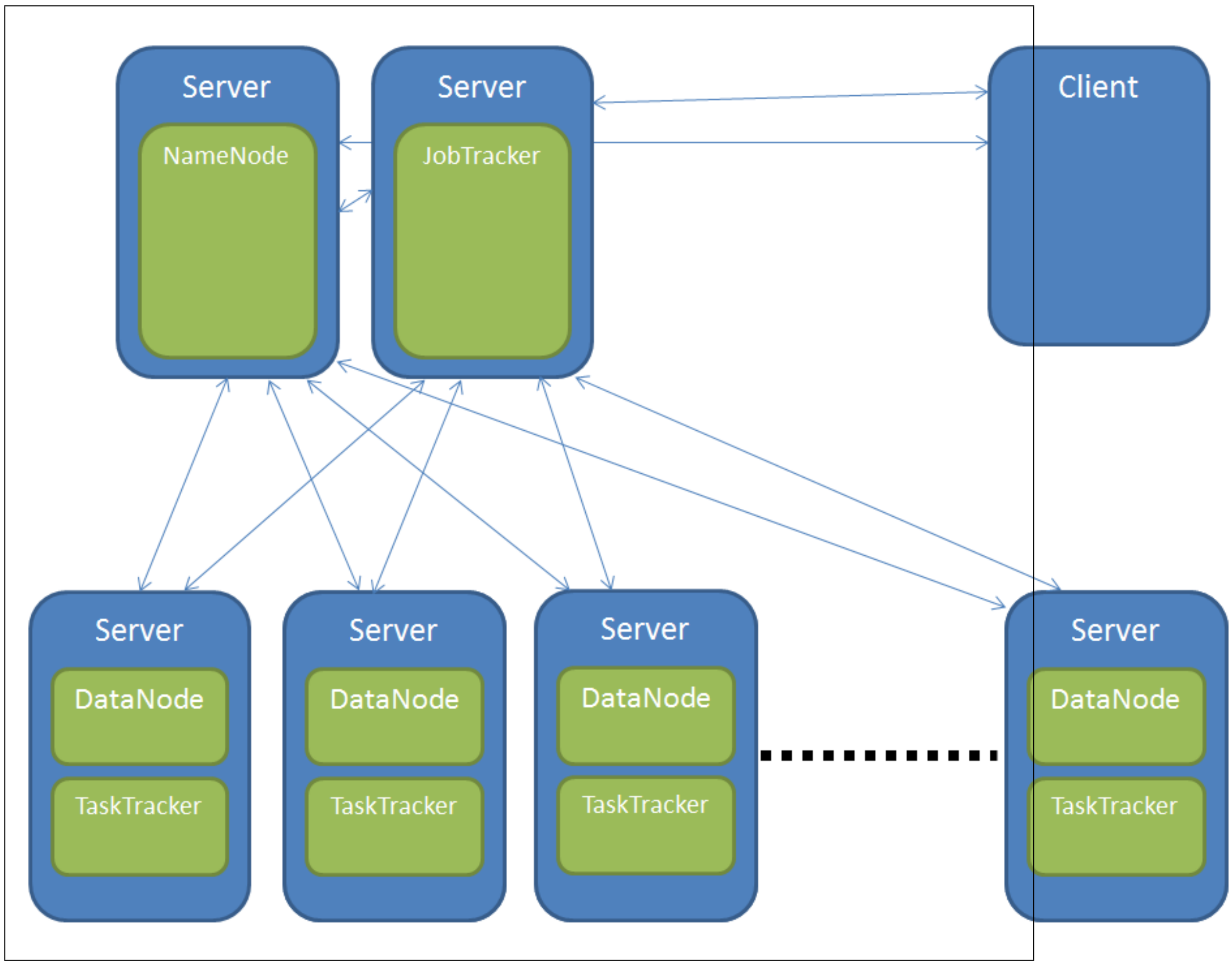

Figure 2.2: Overview of Hadoop MapReduce

2.4.2.

Listing 2.1: Word Count program in Java

```
public static class Map extends Mapper<LongWritable, Text, Text,
    IntWritable> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(LongWritable key, Text value, Context context)
      throws IOException,InterruptedException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
```

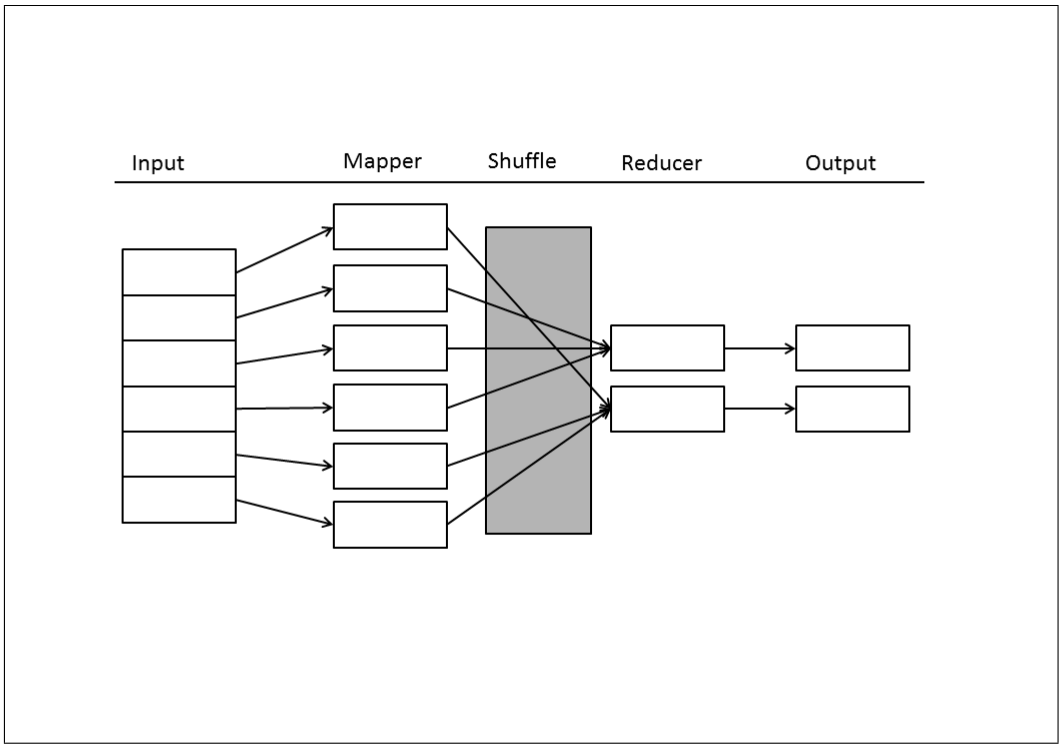

Figure 2.3: Execution flow of MapReduce

```
      while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class Reduce extends Reducer<Text, IntWritable,
      Text, IntWritable> {
    public void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      context.write(key, new IntWritable(sum));
    }
  }
```

2.4.1 JobTracker

The job tracker daemon receives MapReduce jobs from clients. It will look up the data location of input and then place jobs locally with the corresponding DataNode/TaskTracker. Running a local task is essential for the optimization of the cluster utilization. Clusters are usually connected with 1gbps connections, and links between racks have a slightly higher bandwidth. So data transfers between nodes are a scarce resource. Avoiding to run a non-local tasks is essential.

Communication between daemons inside the cluster happens through heartbeat messages which are sent out at given intervals. Intervals are specified with regards to a minimum interval configuration and the actual interval it operates with. Intervals are dependent upon the size of the cluster. Larger cluster have a much higher interval between heartbeat messages. Hadoop regulates this value as it sees fit when new nodes are added to the cluster.

JobTracker monitors all TaskTrackers with heartbeat messages. If it doesn't receive a heartbeat message from a task tracker it can consider this node as dead and blacklist this node. If it is blacklisted it will not be assigned new tasks.

2.4.2 TaskTracker

TaskTracker runs together with DataNodes on multiple nodes within the cluster. it receives tasks from JobTracker through heartbeat messages which is sent between the two daemons to communicated. Task Trackers message the Job Tracker in intervals telling it about its job status.

2.5 Job Scheduling in Hadoop Clusters

Task schedulers in Hadoop can be switched out by only adjusting one simple configuration setting. It allows for more complex task schedulers to be created. An example of how this is configured can be seen in listing 2.2.

Listing 2.2: Example configuration of a pluggable scheduler in Hadoop

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

The three task schedulers that are to be used in these experiments are described in the following subsections.

2.5.1 JobQueueTaskScheduler

A default installation of Apache Hadoop is set up with a "First-In, First-Out" (FIFO) scheduler. This is called JobQueueTaskScheduler. It only way of controlling the job flow is by priority on the jobs. Maps will execute in a first-in-first-out order, which prevents this task scheduler from running multiple jobs concurrently.

2.5.2 Capacity Scheduler

Designed at Yahoo to be used in a shared multi-user Hadoop cluster. Written to maximize throughput and the utilization of the cluster. Designed to give each user a minimum share of the cluster.

Jobs are submitted to queues. Each queue have been given a fraction of the cluster share to use. Jobs submitted to a specific queue will have access to resources allocated for that queue.

Each queue have security measurements implemented. Queue have access control list associated with each queue. Restricting users to put jobs in the wrong queue. Security also prohibits users from viewing and editing jobs in other queues if desired.

Excess cluster resources are distributed to each queue in a reasonable manner. Queues running below their given capacity will be allocated the excess resources as they become available. Queues can optionally be equipped with priority job selections. Higher prioritized jobs will get access to cluster resources before lower prioritized jobs will. Preemption of jobs is not currently supported.

2.5.3 Fair Scheduler

The FAIR Scheduler[21] was developed at Facebook, to be used with their data warehousing solutions. A need to share clusters to save money. not ideal to have two groups of developers having a private cluster each. Colocate both groups within a single cluster. Hadoop On Demand it good enough to share a cluster. Cluster utilization not good enough and data locality not efficient. Decided to develop a scheduler for multi-user purposes with improved sharing abilities.

Fair scheduler divides jobs into pools. Pools can be assigned a minimum share of a cluster. Users map-reduce jobs are placed within a pool. Multiple jobs within a pool can be run in FIFO order or with the fair scheduler. Resources within a pool would be divided between the multiple jobs running.

Choosing local tasks have effect on throughput of cluster. Fair scheduler need to relax its requirements a bit to achieve higher locality. Forcing a job to start without any local data to work on is not ideal. Implement delay scheduling to improve data locality. If a job cannot be assigned to work with local data, skip this

Jobs are placed in pools. Each pool have a guaranteed minimum share of the cluster. Any excess capacity is distributed within each pool. Aims to allocate the entire cluster at any point in time. Reassignment of task need to be done when new jobs arrive. Possible to kill tasks from other pools to make distribution fair, wait for a tasks to finish, or preempt tasks. FAIR scheduler use a combination of the two first options.

Allocation File

As per default FAIR is designed to create one pool for each user who submits a job to the Task Tracker and share the cluster evenly between the running jobs. But there is also a possibility if you want pools to have certain requirements.

The Fair Scheduler will read pool configurations from a separate XML file called fair-scheduler.xml by default. This file can should be placed in the configuration directory within the Hadoop installation.

Job allocations can be monitored by FAIRs web interface available at *http://<JobTracker URL>/scheduler.*

2.5.4 Summary

The two multi-user job schedulers work in a similar fashion. They both use queues or pools to divide up the resources available in the cluster. Each queue is then by configuration files given a certain capacity of the cluster.

The JobQueueTaskScheduler is the most primitive scheduler available and do not have any configuration settings to adjust.

2.6 Additional Hadoop projects

2.6.1 Apache Pig

Apache Pig [16] is high level data processing language. Created at Yahoo to ease the process of writing Map Reduce jobs with fast iteration. Map Reduce jobs in Pig are written with a built in data flow language called Pig Latin, explained in detail in section 2.6.1

Pig Latin

Pig Latin is a scripting language designed for large scale data analysis. Pig Latin is somewhat similar to SQL queries. Pig Latin can be extended with Under Defined Functions using languages such as Java and Python.

Listing 2.3: Word Count program in Pig Latin

```
-- whas
input_words = LOAD '/wordcount-input/' as (data:chararray);
-- asd
flatten_words = FOREACH input_words GENERATE FLATTEN(TOKENIZE(
    data)) AS word;
-- asd
grouped_words = GROUP flatten_words BY word;
-- aasdas
counted_words = FOREACH grouped_words GENERATE group, COUNT(
    flatten_words);
```

```
-- asda
STORE counted_words INTO '/wordcount-output/';
```

The Pig Lain language is fairly similar to the Sawzall language created at Google[17]. They are both declarative languages which supports statements as well.

2.7 Summary

We have in this chapter taken a closer look at what big data problems are all about, and the most common framework for processing big data, Apache Hadoop. Job schedulers are also explained in some detail.

Chapter 3

Experiment Design and Implementation

3.1 Introduction

This chapters describes the design process of creating set of benchmarks to be run in these experiments and how they are implemented.

Also an explanation of how the Hadoop cluster is set up with details surrounding this process. Lastly we describe the process of converting output data from the Hadoop clusters into illustrations by using a parser to extract information from the Hadoop logs.

3.2 Design Goals

Designing a benchmark suite based on real world requirement for testing multi-user Hadoop cluster can be a challenging task, because the usage areas of Hadoop is so many. Workloads can be very different for each Hadoop user. Some jobs might be machine learning jobs running for several days, while other can be short interactive jobs that require a output in a reasonable short time. Originally being developed for large batch jobs Hadoop now have been moving towards running smaller jobs within a reasonable time. Users of Hadoop are now running both small and large jobs on a shared cluster. Multi-user Hadoop clusters should have a job tracker running a task scheduler which distributes jobs evenly according to a predefined service level or a reasonable share of the resources. Jobs with long running times should be taken care of so that they didn't interfere to much with short MapReduce jobs.

Ideally a performance analysis should have capabilities to benchmark the most important aspects of having a multi-user Hadoop cluster. Developing such a benchmark suite one have to consider the following points.

Cluster Utilization Utilizing the entire cluster when the resources are available.

Fairness A scheduled job should be started within reasonable time, and should receive a fair amount of resources.

Deliver agreed service level Jobs with has been given a guarantee on resources should quickly be given these resources. Should not lead to starvation of jobs.

This list provides a set of problems when sharing a large Hadoop cluster between multiple users, but other points are also important when benchmarking Hadoop. Such as job completion time, data locality.

3.3 Measurements

This section descries each of the measurement we want to gather during the benchmarks we intend to run on the Hadoop cluster. Each of these measurements will be presented in a later chapter.

3.3.1 Execution Times

The execution time is defined to be the time from which the execution of a job starts to the very end of execution before the job is complete. In Hadoop terms this is defined to be from a job is submitted into the task trackers work queue. Then all the way through waiting for execution, to execution and then to job cleanup. Execution time stops when the results are stored back onto HDFS and ready to be used by the job issuer. The job tracker is responsible for guiding the job trough all these stages with its task scheduler. Having a Task scheduler which smoothly schedules all jobs with no overhead can be a key issue when choosing a scheduler.

When benchmarking task scheduler it is important to measure and compare the execution times between task schedulers to see how well they execute and organize their tasks. The execution time is affected by the amount of resources allocated to each specific job. Distributing the available resources within a cluster is key to achieving these fast execution times. With many jobs present in a queue of work it is key to distribute the resources between all jobs were applicable. Since there is only a predetermined amount of mapper- and reducerslots available on each node in the cluster the utilization of these are key.

With the Hadoop Frameworks extensible logging futures one can easily read and compute the time spent on each job. Timestamps for each event that happens through all jobs are logged into separate files and saved. By reading these log files we can pick out the timestamps from each job by reading the file. Each log file consist of two distict lines of information. These are from when the job was submitted to the task tracker and when the job finished.

From log file:

```
Job JOBID="" JOBNAME="" USER="" SUBMIT_TIME="1341249350626"
-> JOBCONF="" VIEW_JOB="*" MODIFY_JOB="*"
-> JOB_QUEUE="exampleQueue" .

Job JOBID="" FINISH_TIME="1341250651263" JOB_STATUS="SUCCESS"
-> FINISHED_MAPS="" FINISHED_REDUCES="" FAILED_MAPS=""
-> FAILED_REDUCES="" MAP_COUNTERS="" .
```

These timestamps are from the java function System.currentTimeMillis(); which get the amount of miliseconds from 1st of January 1970 to now. Knowing this we can convert the time to a more readable format, but most important calculate the total time spent on each job.

This is calculated with the formula:

$$\text{total time} = (\text{finish time} \mathbin{/} \text{submit time}) \mathbin{/} 1000$$

This gives us the total time spent on each job in seconds. we will later use this function when creating a small script to read through all logs.

3.3.2 Fast Response Times

Queues are used to share resources between in a Hadoop cluster. Mapper and reducer slots are then distributed between all these queues. The configuration file for the task scheduler resources are distributeds the resources between queues. To maximize the efficiency of the cluster idle resources are often released to other queues to speed of the performance of other queues. When the borrowed resources are needed by their original owner they need to be released and relocated. This need to happen in a efficient way to let users which submit jobs have their jobs processed quickly. specially if their submitted jobs are a smaller query which do not need processing power to complete. It is then better to free up some resources to let this process.

A worst case example on this can be if you run two jobs with names Job1 and Job2. Job1 is a long running job which would use 24 hours to complete using the entire cluster. Job2 is a much smaller job which would use 15 minutes to complete if the entire cluster were used. Job2 is submitted 15 minutes after Job1. In the traditional Hadoop configuration this would lead to Job1 being processed first by the entire cluster and then Job2 would get the entire clusters resources. The total time spent on Job1 is 24 hours. Job2 which spent 23,75 hours waiting, and then 15 minutes processing. The total time on both jobs are then 48 hours. It is not ideal for any user in the cluster for this to happen.

A unreasonable distribution of cluster resources might as well have a performance hit on the execution time of jobs. If a job should in any case not get a reasonable share of the cluster the progress of this job would stop and the execution time would suffer drastically.

To measure the time from which you submit a job to your designated queue to the time it begins to process on the job tracker we need to read through all map task that ran. This will let us determine which map task ran first, and also how long time it was between the first map task and the time of which the job was submitted. This can be read as the waiting time.

$$\text{waiting time} = (\text{first map start - submit time}) \ / \ 1000$$

In the log files values such as the launch time of a job can be found. Research showed that this was not a value that could be used to compare all schedulers, because the Capacity Scheduler from Yahoo launches a task soon as it is submitted, but do not reassign any resources to it after that. This is done to save time for when the job first start to run. Therefor the use of the first map task is used to calculate the waiting time for a job.

This waiting time can be measured across all benchmarks and can be compared in a fair way.

3.3.3 Data Locality

Since the Hadoop Framework features a underlying file system for storing data across the cluster launching a task close to where the data is stored is essential to keep the network bandwidth low. Increased bandwidth across the network may decrease the overall performance of all jobs running. Data transmission may delay map task to the amount of time used by transferring the data chunk from one node to another. Hadoop runs with a replication factor of three default. This means that a map task can be launch at three different place throughout the cluster. With replication factors lower the chance of an increased data transfer gets higher with many users using the cluster.

We want to measure the data locality to see how many times data have to be moved go get processed. In larger clusters nodes are often separated into racks. This description helps Hadoop place the data in places to make data available if a connection to a rack disconnects. By default Hadoop places data in two racks. Two blocks on one rack and one in a different rack. This ensures availability.

The Hadoop framework logs this through its internal job counters. There are counters which counts the number of rack-local maps and local maps. Rack-local values describe if data were moved between nodes on the same rack. Local maps describe a map were no data transfer were made in order to process this block. We can simply extract these numbers from each job in a simple script and compare them to other numbers.

3.3.4 Cluster Utilization

Cluster utilization means the utilization of the available resources a Hadoop cluster has to offer. When running with multiple users and multiple queues we want to efficiently utilize the resources across queues. Queues which have assigned a certain amount of resources should share this to other queues, when these resources are available. By sharing these resources we effectively utilize the processor of all nodes in a cluster. By having a large cluster shared amongst many users the execution time is reduced when the other users of the cluster is idle.

Having resources shared between many users we need to ensure fairness with respect to the amount of resources assigned to each queue. Queues should over time always receive the designated amount of resources. It should not take to much time before resources are relocated when needed. It is important to see how the scheduler reassigns new tasks to different queues.

This affects jobs if it is not prioritized correctly in a situation with many concurrent jobs running.

To measure this a script to read the status of each queue is needed. At a given interval this script will get the status of each queue available and write this. When we have each status we can see how tasks are distributed over all queues. Since each queue have by configuration a fixed set of mappers and reducers available we will monitor and see that during heavy load these queues get their appropriate resources.

3.4 Setup and Configuration

This section describes how the cluster, task schedulers and queues are set up for each benchmark.

3.4.1 Cluster

A Hadoop cluster was set up on Amazon Web Services using their Elastic Compute Cloud[3]. Apache Whirr 0.7.2 was used to spin up the Hadoop cluster. The Hadoop cluster was configured using the Whirr configuration file. In Listing 3.1 commands for launching and destroying a cluster is listed. Specifying the configuration file in the command redirects Whirr to the correct configuration. This file lets Whirr distribute the appropriate configuration variables to all the machines in the cluster. During the deployment of the cluster Whirr creates custom scripts based on the Whirr configuration file. These scripts are run on all nodes in the cluster, to install and configure the nodes. The complete start up script takes roughly 5 minutes before all Hadoop nodes are ready.

Listing 3.1: Configuration file used by Whirr

```
bin/whirr launch-cluster --config=masteroppgave.properties
bin/whirr destroy-cluster --config=masteroppgave.properties
```

The configuration used for all performance analysis runs are shown in Listing A.1. In the configuration file properties such as *whirr.provider*, *whirr.identity* and *whirr.credential* gives Whirr you access credentials to access resources available at Amazon Web Services. *whirr.instance-templates* lets you define which instances to run on each node, and the number of nodes. In Listing A.1 one node with NameNode daemon and JobTracker daemon will be run, together with 15 nodes running DataNode daemon and TaskTracker daemon running.

Amazon Web Services offers different type of hardware for each node. This can be configured using the option *whirr.hardware-id*. During experiments large standard hardware instances were used. Specification to this hardware type is shown in Table 3.1. All nodes ran an Amazon Machine Image (AMI) with Ubuntu 11.10 Oneiric. This were specified by using *whirr.image-id* to specify an ID for this image. *whirr.location-id* specified that the nodes were located in the US East region, which is located in Virginia, USA.

Value	Description
Memory	7.5 GB
CPU	4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each)
Storage	850 GB instance storage
Platform	64-bit Platform
I/O Performance	High

Table 3.1: Large Instance Specification

Appendix A.1 on page 76.

The configuration of Hadoop is done by specifying first which Hadoop configuration file you want to edit, and then adding the proper name and value for this configuration. Hadoop have three important configuration files, hadoop-common, hadoop-hdfs and hadoop-mapred.

¡hadoop config file¿.¡configuration name¿=¡configuration value¿

3.4.2 Hadoop Configuration

Many important configuration options in a Hadoop cluster is automatically generated by Apache Whirr during the bootstrap process, such as storage locations, address to NameNode and TaskTracker. These configuration parameters are distributed around to all data nodes, so that they are able to contact their master nodes.

Whirr do a good job bootstrapping and configuring nodes in a Hadoop cluster, it will only set general Hadoop configuration values to their defaults. Since Hadoop is designed to run in a single-user batch operation mode, not all values are great with a multi-user scenario. These will have to be written into the Whirr configuration file.

Performance benchmarks in this thesis were run with a few adjustments to facilitate multiple users using the Hadoop cluster.

mapred.tasktracker.map.tasks.maximum Set to "2". This setting defines how many map task can be run simultaneously on a TaskTracker, which is the slave node. This is the same as the default on Hadoop.

mapred.tasktracker.reduce.tasks.maximum Set to "2". This setting defines how many reduce task can be run simultaneously on a Task Tracker. This is the same as the default on Hadoop.

hadoop-mapreduce.mapred.queue.names Set to "production,research,test". This setting defined the names of each queue used in the experiments. Both the Fair Scheduler and Capacity Scheduler can use this Hadoop specific configuration to separate the jobs into queues.

mapred.reduce.slowstart.completed.maps Set to "0.9". This setting describes when the reducers are allowed to grab a reduce slot in the cluster. By default this is set "0.05". This means that when 5 percent of the maps are complete the task will request reducers to start to start spilling data into the reducers. In presence of both long and short jobs this can prove to have negative effects on the performance of short jobs. Initial tests showed that when a long running job had grabbed reduces slots, it would keep these occupied throughout its entire run, which could be a long time. Any short running job would then have to wait for a free reduce slot for quite some time. The reason for for it being set so low is mainly to reduce the overall runtime of jobs in a single user scenario. Maps and reduces would then run in parallel and complete earlier.

reduce.parallel.copies Set to "20". This setting lets the reducer run a higher number of parallel copies form maps. This speeds up the reduce phase of there is many maps that want to send data to reducers.

dfs.permissions Set to "false". This setting is normally set to "true" in Hadoop. During the experiments it was set to "false" to avoid any complications with storing files. In a multi-user scenario one would want this to set to "true" to avoid users overwriting files not belonging to other users.

3.4.3 Configuration of JobSchedulers

Hadoop was configured to use Capacity Scheduler shown in Listing 3.2 and Fair Scheduler shown in Listing 3.3 the default value for task scheduler in a standard installation of Hadoop is be seen in listing 3.4. The two multi-user schedulers will be configured as previously mentioned, while the JobQueue-TaskScheduler will run as a clean install of Hadoop.

The multi-user task schedulers use queues or pools to distribute resources between jobs. Each job is assigned a queue. It will use resources made available to this queue throughout its life. We configure queues that can be used in our benchmarks. The names of these do not matter, but for our test we have chosen Production, Research and Test to represent our queues. The names of the queues can typically be names of services that are dedicated some resources. For example hive or pig scripts. Another example of how the queues can be divided is by team within a company. Spam detection jobs can run on one queue, while log analyzer jobs can run in a separate queue.

To utilize the entire cluster we will try to specify the allocation files to utilize the whole cluster when possible. We can then from our tests see how well the jobs adapt to having to wait for resources, while other jobs are actively using them.

Listing 3.2: Configuration option for selecting Capacity Scheduler as Task Scheduler

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

Listing 3.3: Configuration option for selecting Fair Scheduler as Task Scheduler

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

Listing 3.4: Configuration option for default Task Scheduler

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

3.4.4 JobQueueTaskScheduler

The JobQueueTaskScheduler are not configured in any way. This will be a clean install, without any modification. There are no queues to be configured with this task scheduler.

CapacityScheduler

The production queue for the CapacityScheduler is set up with a capacity of 60 percent of the cluster. There is no maximum of how much of the cluster

capacity the queue can use. This will allow jobs submitted to this queue to use 100 percent of the queue if there are no other jobs jobs active. No other restrictions are set. The entire contents of configuration file can be found in appendix B.4 on page 79.
The research queue for the CapacityScheduler will only have an initial capacity of 20 percent available, while it can utilize 100 percent of the cluster when available. In order to achieve good execution times on jobs that run on the cluster we will allow all queues to utilize 100 percent. This will hopefully have a positive effect on the execution times. The entire contents of configuration file can be found in appendix B.5 on page 80.
The test queue for CapacityScheduler have a capacity of 20 percent, similar to the research queue. The entire contents of configuration file can be found in appendix B.6 on page 81.

FairScheduler

The production queue in the FairScheduler works similar to the CapacityScheduler, but capacity is not defined in percent, but instead is defined as minimum and maximum maps and reduces. These numbers are the maximum number of slots the queue can use during runtime. The values do not scale if the cluster increases in size, but the allocation file is reloaded on a regular basis. Because of this there is no need to restart the cluster to get an updated allocation file. When defining the production queue we want to have similar capacities as the CapacityScheduler so we have a minimum of 18 maps, which is 60 percent of the total maps available on the cluster. The cluster has 30 maps available. Two on each node in the cluster. We set the maximum of 30 maps to allow the queue to use the entire queue when possible. Same values are set for the reducers. The entire contents of the configuration file can be found in appendix B.1 on 78. Research queue have a minimum maps and reducers value set to 6. The value 6 is derived from the 20 percent which the CapacityScheduler queue got. The entire contents of the configuration file can be found in appendix B.2 on 78. The test queue for the FairScheduler receives 6 maps and reduces minimum, and a maximum of 30 to utilize during runtime. The entire contents of the configuration file can be found in appendix B.3 on 79.

3.5 Creating work loads

Three types of benchmarks were created. Each running a variety of jobs with given size. First benchmark consist of long running jobs. Second benchmark consist of short running jobs. The last benchmark consist of a mix between the two benchmarks. A mix of jobs ranging from short to longer running jobs.

Each benchmark consists of MapReduce jobs available from the Hadoop installation, some Pig scripts and some MapReduce developed for these tests.

3.5.1 MapReduceJobs and Pig scripts

TeraSort

The TeraSort job is an example which is bundled with the default installation of Hadoop. This is an standard MapReduce sorting example which is widely used when benchmarking Hadoop clusters. This job takes a input directory of files, which is generated from another bundled job TeraGen. The job then sorts the input in total order and output the data into an output directory.

To run this job from the commandline: hadoop jar hadoop-examples-1.0.1.jar terasort /inputdirectory/ /outputdirectory/

Wordcount

The Wordcount is an example which is bundled with the default installation of Hadoop. It can be found in hadoop-examples-1.0.1.jar located in the root directory. This example job takes a directory of text files. It reads trough files and count the occurrence of each word in the input data. It will output a file containing all the unique words together with the sum of all occurrences. Every mapper will read a portion of the input data and output key-value pairs. The word being counted is the key, while a value of 1 is outputted for each word. Results are sent along to a combiner. This sums up the output from the maps, and passes it along to the reducers. The reducers combine results from all maps and output the final values.

To run this job from the commandline: hadoop jar hadoop-examples-1.0.1.jar wordcount /inputdirectory/ /outputdirectory/

SecondarySort

The SecondarySort is an example mapreduce job which is bundled with the default installation of Hadoop. The job takes an input directory of text files where each line contains two integers separated with a space. The text input is then sorted. The first integer are sorted in ascending order, and then the second integers are sorted ascending. The job will output a text file containing all the numbers in sorted order.

To run this job from the commandline: hadoop jar hadoop-examples-1.0.1.jar secondarysort /inputdirectory/ /outputdirectory/

PopularSerches

The PopularSearches reads text files containing log data and counts how often popular search terms occurs. The input for this job is text files and it will output text files containing a key value pair for each search term. Each line in the outputted text file contains the key, which is the search term and a value which is the count of how many times the search term occurred in the input files. This job is written in Java.

To run this job from the commandline: hadoop jar Master.jar no.uio.ifi.joachse.PopularSearc
/inputdirectory/ /outputdirectory/

HourlyDistribution

The HourlyDistribution job reads text files containing log data and extracts the date from each entry. From the date of the query the hour is extracted and outputted as a key. Each key is counted and a total for each key is a outputted into a text file. The output file have one line for each hour in a day and the sum of queries for this query. This job is written in Java.

To run this job from the commandline: hadoop jar Master.jar no.uio.ifi.joachse.HourlyDistrib
/inputdirectory/ /outputdirectory/

BrowserCount

The BrowserCount is a job that reads through log files and extracts the browser from each log entry. For each browser found in the log files it will output a count of how many times this browser had been used. This job is written in Java.

To run this job from the commandline: hadoop jar Master.jar no.uio.ifi.joachse.BrowserCoun
/inputdirectory/ /outputdirectory/

CountryCount

The CountryCount is a similar job to the above. It will extract the country count for the log files. ...

To run this job from the commandline: hadoop jar Master.jar no.uio.ifi.joachse.CountryCoun
/inputdirectory/ /outputdirectory/

Unique Visitors - Pig script

The UniqueVisitors are a Pig script that extracts information from log files aswell. It will read through the log file and extract every unique visitor IP....

To run this job from the commandline: pig -x mapreduce -f UniqueVisitors.pig -p input=/inputdirectory/ -p output=/outputdirectory/

Unique Searches - Pig script

The UniqueSearches are a Pig script that extracts information from log files. It reads through all log files in a specified directory and outputs the unique search words. No count for how many times the search word occurred is outputted. To run this job from the commandline: pig -x mapreduce -f UniqueSearches.pig -p input=/inputdirectory/ -p output=/outputdirectory/

3.5.2 Benchmark run scripts

Large jobs

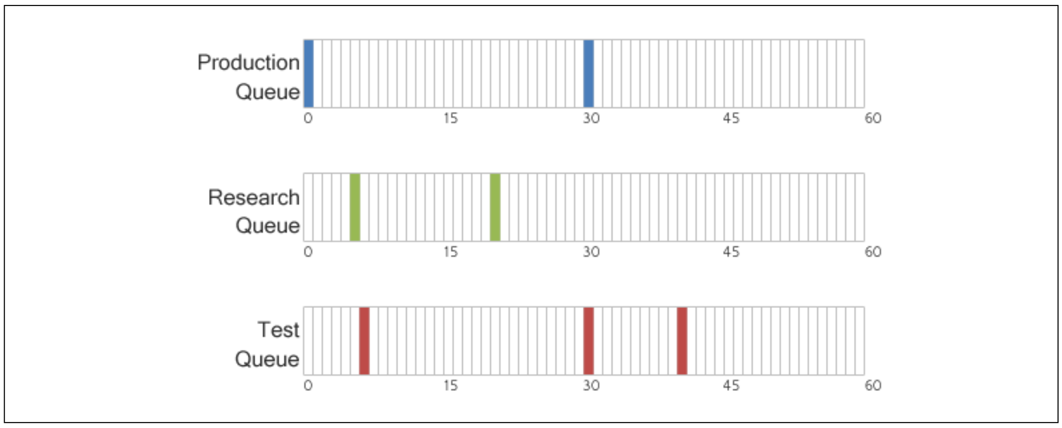

Figure 3.1: Job launch times in Large Benchmark

Figure 3.1 on 34 we can see at what time a job is launched during the Large job benchmark. In the production queue two jobs are launched. Both are TeraSort jobs which sorts and outputs a sorted set. The input size of these jobs are 70GB and 40GB. The output for both of these jobs will be the same sizes. They are executed 30 minutes apart.

The entire configuration for these jobs can be found in appendix C on page 84.

Small jobs

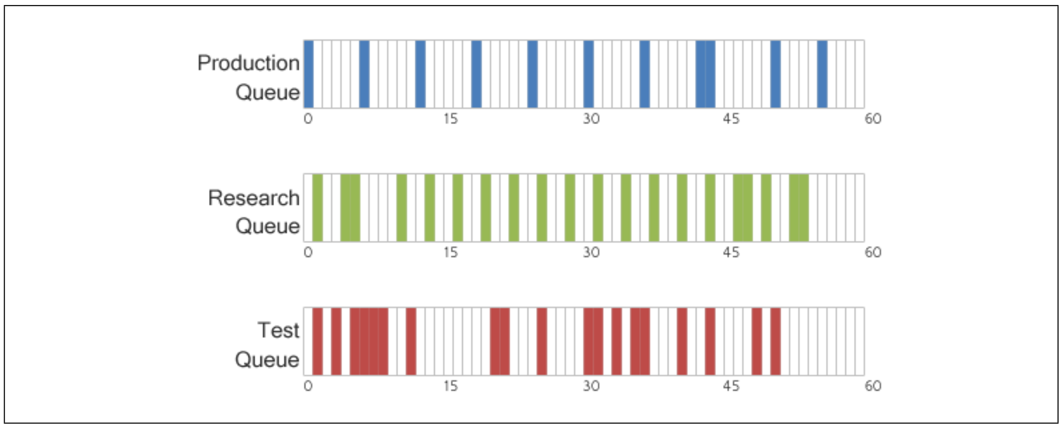

Figure 3.2: Job launch times in Small Benchmark

Figure 3.2 on page 35 displays the intervals of when the jobs are run using the small benchmark configuration. From the figure we see that there are many jobs launched within the hour displayed. The production queue holds a total of 11 TeraSort jobs with input sizes ranging from 5GB to 15GB. The research queue contains 20 jobs of different sizes. All these jobs are custom written Java scripts that counts from a log file. The last queue, which is test has a total of 23 jobs launched. These are Pig scripts, Wordcount and SecondarySort jobs. Their sizes are 1GB and 2GB of data.

The entire configuration for these jobs can be found in appendix C on page 84.

Mixed jobs

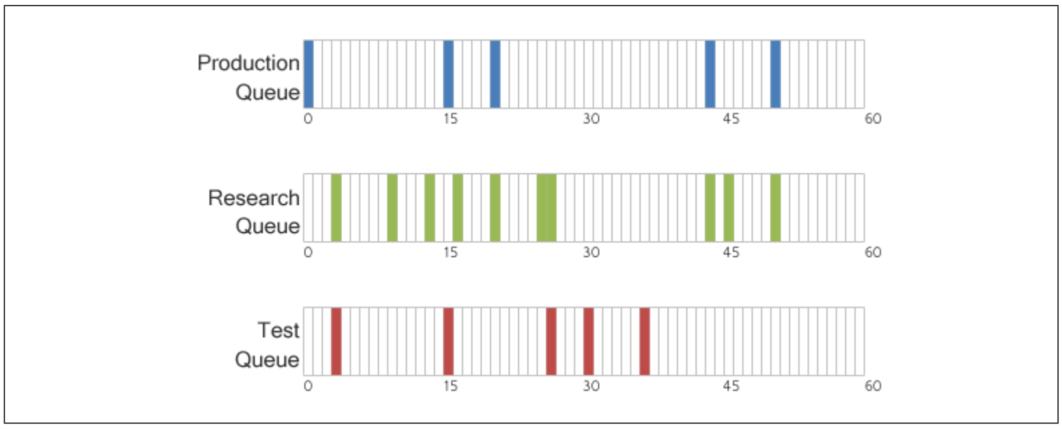

Figure 3.3: Job launch times in Mixed Benchmark

Figure 3.3 contains the points when new jobs are launched during the mixed benchmark. During this benchmark jobs are launched 21 times at different points in time. The production queue contains TeraSort jobs with sizes from 5GB to 40GB. Research queue runs 10 custom counter map-reduce jobs which reads log data and outputs counts. The test queue will only run 5 pig scripts.

The entire configuration for these jobs can be found in appendix C on page 84.

3.6 Executing Jobs

To assess the performance of three task schedulers the need for a normalized environment is needed. A Hadoop cluster have already been set up to host all the jobs about to run. The problem arises when trying to launch many jobs at a certain time during the benchmark. To achive a similar tests using all three task schedulers a simple launcher application was created in Java. This application is resposible for running the Hadoop jobs in the same order and at the same time across all benchmark runs.

The application reads from a configuration file how the benchmark run is supposed to run. The configuration file contains commands as well as the delay of the operation. After reading the configuration the application will start running the commands given to it. A sample of a configuration file can be seen in listing 3.5 on page 37.

Listing 3.5: Configuration job runner

```
<?xml version="1.0" encoding="UTF-8" ?>
<Benchmark name="Example">
  <Task>
    <Name>Example #1</Name>
    <Command>hadoop jar MapReduceJob.jar terasort</Command>
    <Delay>100</Delay>
  </Task>
</Benchmark>
```

One configuration file were created for each type of workload to be run on the cluster. These configuration files contain jobs with different sizes and applications.

3.7 Generate Inputdata

Since most data sets are not publicly available generating a fake data set to accompany the data suite is needed. Scripts to generate input data for each job in the benchmark suite will have to be developed. One script for each type of job. Data generation script should be able to generate different amount of input data. This is needed so that jobs with different sizes can be easily created. The number of mapping jobs created for each job is based on the size of the input data. HDFS has a default block size of 134217728 byte, which is specified by the configuration parameter *dfs.block.size*.

TeraSort data is generated beforehand with the TeraGen tool which is accompanied in the jar file which contains TeraSort.
Generating input data to accompany the counting map-reduce jobs is done by generating a fake log file with random generated data. This is not the most ideal way of creating logs, but other real world logs with different sizes are not available. The generated data is a text file with each log entry sepa-

rated by a tab. The map-reduce jobs will then decouple this data and extract the values wanted.

Data for the WordCount and SecondarySort is done with a script that writes random integers to a text file. We will use a the Wordcount example to count the number of unique integers, instead of counting the traditional words.

3.8 Visualising Results

To fully understand what happens during each benchmark a small Java application were created to parse the log files which is created during runtime from each Hadoop job. This application is just a quick tool to create output files for my personal use within this thesis, and not intended for anything else. The application goes through each log file within a directory to read its content. When reading trough the log files the application will successfully find each event described and add this to the applications memory. When all data is successfully read it will generate a CSV (Comma-separated values) file with the information contained within the logs file in a easy to use format. This format can then be imported into any statistical packages such as MATLAB and R, or even Microsoft Excel.

To successfully write output files the application need parameters regarding what output it should produce. In the list below a description of each parameter is given and also an example of the output from the application is displayed.

mapperqueue "inputdirectory" "outputfile"

Using the parameter mapperqueue the program will write a list with the status of each queue found within the log directory. It will sample every every two seconds to see the status of each queue. This interval is set within the program and cannot be changed from the command line. During testing this was found to give best results.

```
Example:
```

```
time,queuename1,queuename2,queuename3,queuesize
0,production,research,test,10
2,production,research,test,14
4,production,research,test,20
6,production,research,test,22
...
```

reducerqueue "inputdirectory" "outputfile"

This parameter acts similar to the mapperqueue parameter. It will create an output file containing the queue status at each interval. The interval for this application is hardcoded at 2 seconds.

```
Example:
time,queuename1,queuename2,queuename3,queuesize
0,production,research,test,10
2,production,research,test,14
4,production,research,test,20
6,production,research,test,22
...
```

localmaps "inputdirectory" "outputfile"

This parameter will display the amount of local map tasks launched with each job and the total amount of launched map tasks.

```
Example:
jobId,localMaps,launchedMaps
job_01,75,100
job_02,100,100
job_03,104,105
job_04,423,512
...
```

jobruntime "inputdirectory" "outputfile"

The first item

submit to finish

```
Example:
jobId,jobRuntime
job_01,620
job_02,230
job_03,402
job_04,107
...
```

waitingtime ”inputdirectory” ”outputfile”

Using this parameter will output the waiting time before a job started processing on at least one of the nodes within the Hadoop cluster. This parameter do not consider the launch time specified in the log file. It will calculate the time in seconds between the job was submitted and the time the first map job started.

```
Example:
jobId,waitingTime
job_01,0
job_02,102
job_03,32
job_04,10
...
```

These commands were used to generate the output which resulted in the graphs and charts in chapter 4 on page 41.

3.9 Summary

We have in this chapter presented the results from all of our benchmark results.

Chapter 4

Testing and Results

4.1 Introduction

This chapter describes the results and graphs from the benchmarks.

4.2 Execution time

A definition by execution time is described in an earlier chapter. This can be found in section 3.3.1 on page 20.

4.2.1 Running Large Jobs

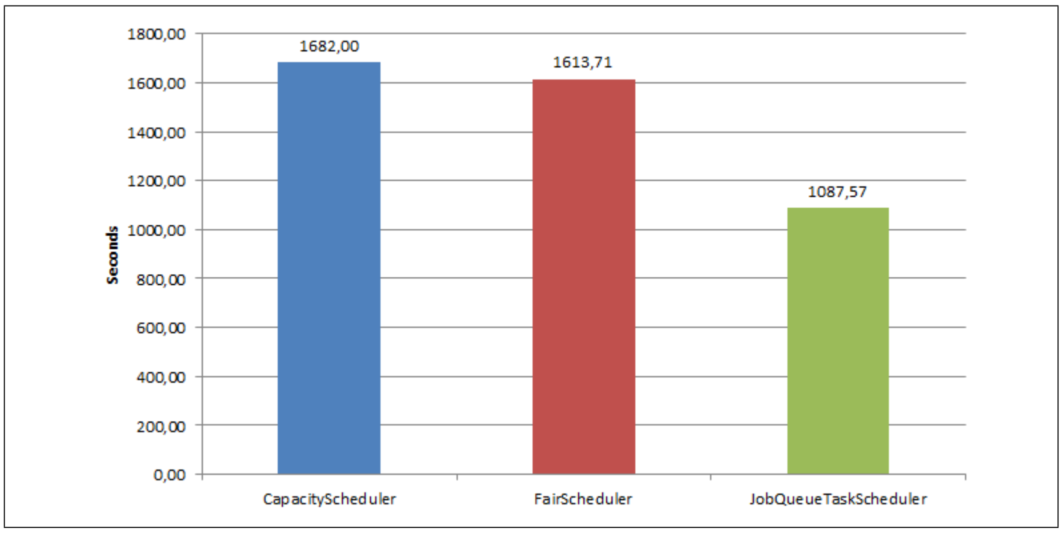

Figure 4.1: Average running time of jobs in the Large Benchmark test

Figure 4.1 presents the average run time from the large benchmark tests. The bar chart shows that the FairScheduler and CapacityScheduler which is designed for multi-user environment finish with roughly the same results. FairScheduler finishes just in front of the multi-user schedulers. The default installation of Hadoop with the JobQueueTaskScheduler finishes first with an average time spent per time around 1000 seconds for the large jobs. JobQueueTaskScheduler is about 60 percent faster than the two other.

4.2.2 Running Small Jobs

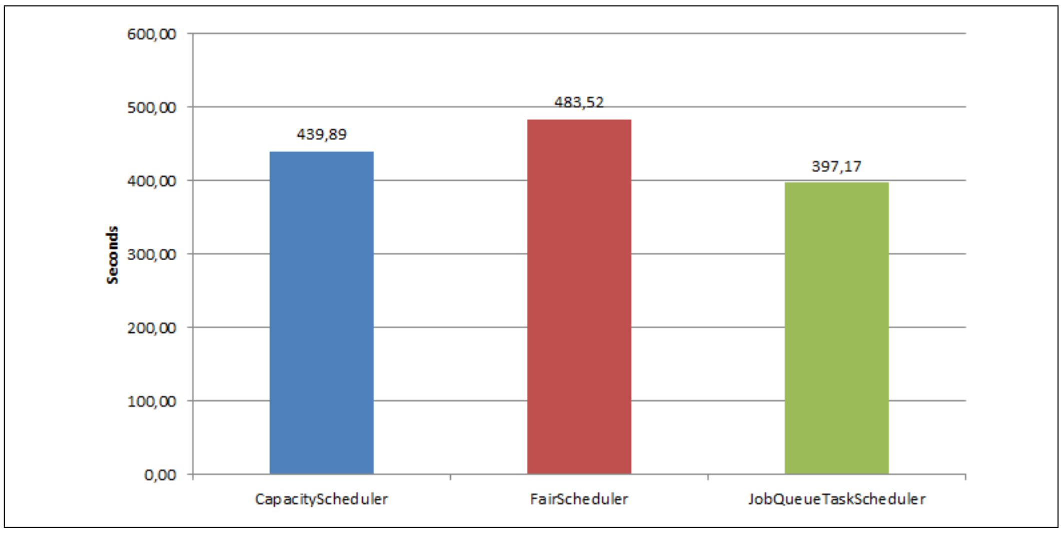

Figure 4.2: Average running time of jobs in the Small Benchmark test

In figure 4.2 results from the Small Benchmark is presented as a bar chart. The bar chart displays results which is around the same values, from 400 to 500 seconds. JobQueueTaskScheduler have the lowest average run times for the small benchmark with 397 seconds on average. CapacityScheduler have an average of 439 seconds for all jobs, while FairScheduler is last of all benchmark tests with 483 seconds. There is still a considerable difference between the results. JobQueueTaskScheduler is around 20 percent faster than FairScheduler.

4.2.3 Running Mixed Jobs

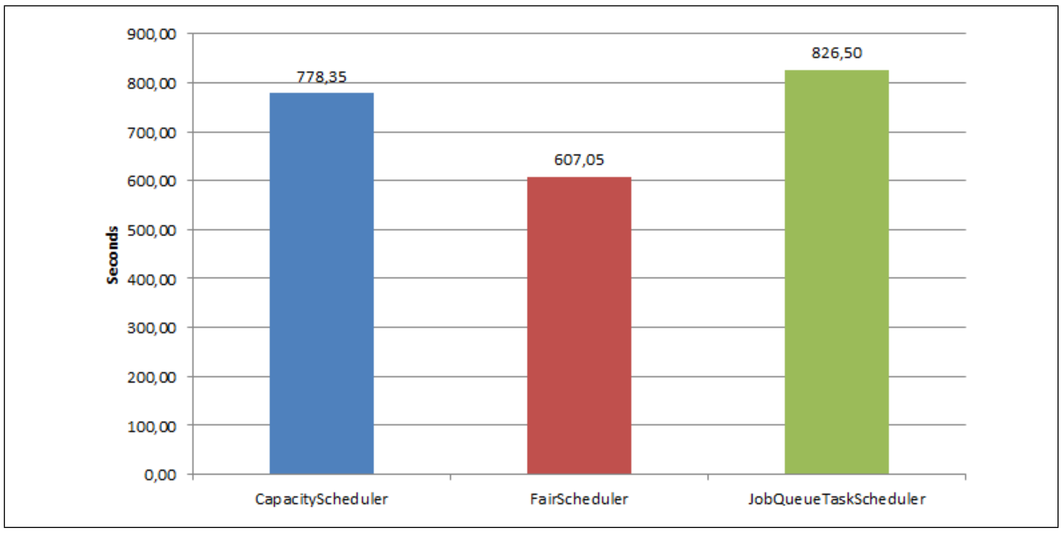

Figure 4.3: Average running time of jobs in the Mixed Benchmark test

Results from the Mixed Benchmark is shown in figure 4.3. Here we can see a different set of results that the two previous results. FairScheduler have the lowest average of all schedulers. The average for this scheduler is 607 seconds, while CapacityScheduler is the next with 778 seconds on average. This time the JobQueueTaskScheduler is clearly the slowest of them all. Averaging 826 seconds it is over 36 percent slower than the FairScheduler.

Summary

The results from the three benchmarks do not display any clear results. There is no scheduler which is fastest in all benchmarks. Even though the JobQueueTaskScheduler is quickest in both the large benchmark and the small benchmark it falls short in the mixed benchmark.

4.3 Waiting Time

The definition of waiting time is described in an earlier chapter, and can be found in a section called Fast Reponse Times on 22

4.3.1 Running Large Jobs

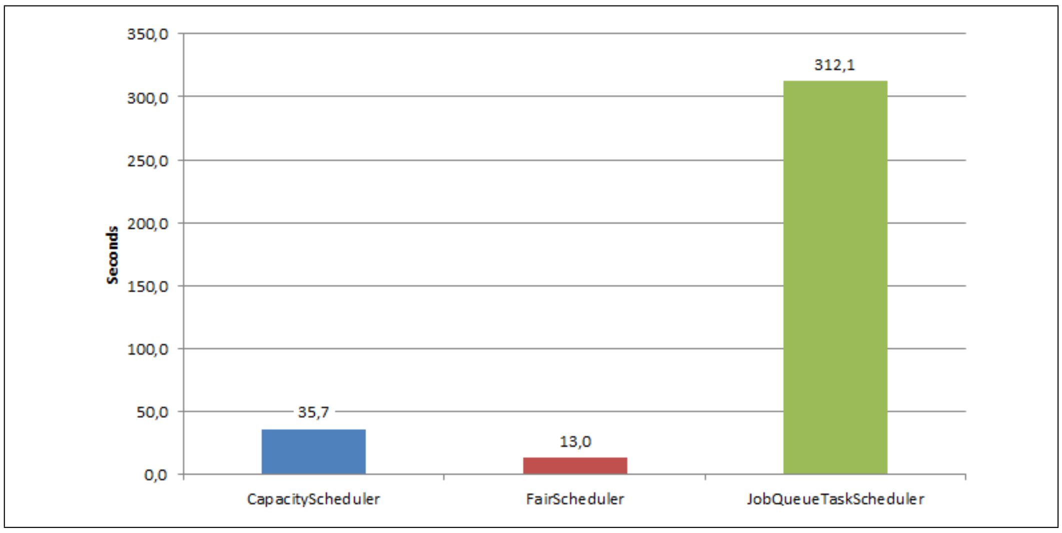

Figure 4.4: Average waiting time for jobs in the Large Benchmark

The average waiting time for each job displayed in figure 4.4 shows an large difference between the job schedulers. CapacityScheduler has on average half a minute of waiting time before a job is scheduled in this benchmark, while the FairScheduler have even better results with only 13 seconds on average before any job is started processing on the cluster. But the JobQueueTaskScheduler have an extremely high waiting time on average per job. An average job on a cluster with multiple concurrent users is 312 seconds, which is over 5 minutes. This means that if you have a small query you can end up having to wait 5 minutes on average before your job even start processing.

4.3.2 Running Small Jobs

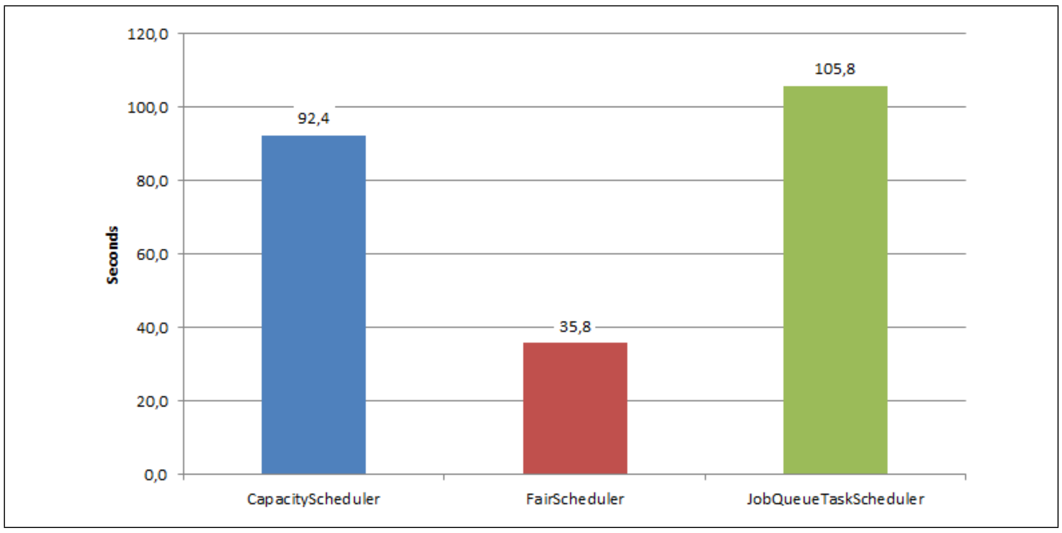

Figure 4.5: Average waiting time for jobs in the Small Benchmark

In figure 4.5 we see the average waiting time results from the Small Benchmark. Yet again the FairScheduler have the lowest amount of waiting time before a job is actually started processing. 30 second on average for the FairScheduler. The Capacity Scheduler is far slower than the FairScheduler in this benchmark with 92 seconds waiting time on average. Just slower than the CapacityScheduler the JobQueueTaskScheduler comes with an average of 105 seconds of waiting time per job.

4.3.3 Running Mixed Jobs

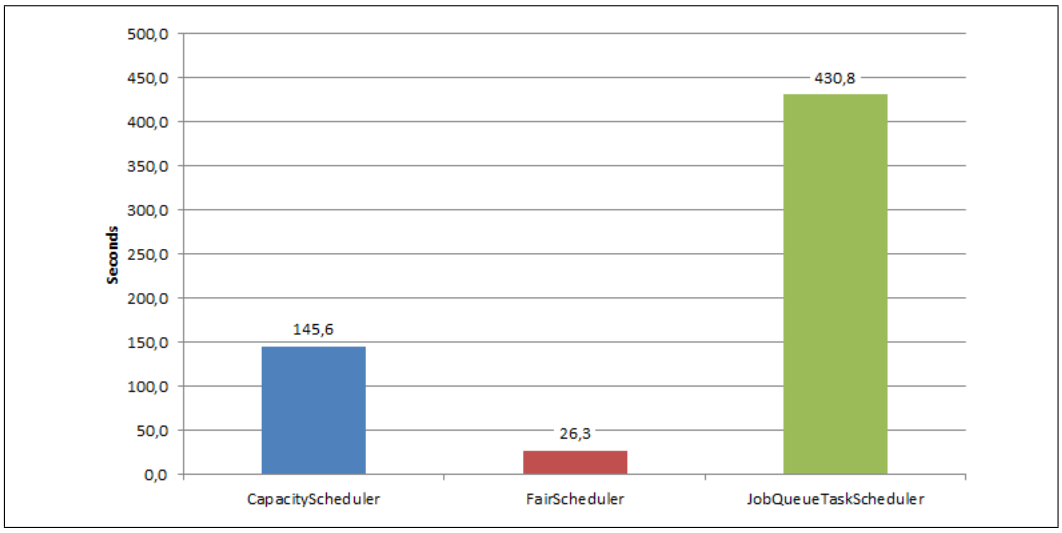

Figure 4.6: Average waiting time for jobs in the Mixed Benchmark

Lastly the Mixed Benchmarks result are shown in figure 4.6. Here we can see that the FairScheduler yet again has the lowest waiting time before a starts to process on the cluster. There is a 26 second waiting time in this benchmark for the FairScheduler. At around 150 seconds of waiting time we can find CapacityScheduler. The default task scheduler for Hadoop, the JobQueueTaskScheduler has a huge waiting time again. With 430 seconds of waiting time on average. This is an huge difference from the FairScheduler. The JobQueueTaskScheduler has a 16 time as long waiting time.

Summary

In the waiting time results we can see huge differences between the task schedulers. The default Hadoop task scheduler has long waiting periods simply because it do not run multiple jobs concurrently. This leads waiting time whenever there are two or more jobs submitted to the work queue.

In both the Large and Mixed benchmarks we see that the JobQueueTaskScheduler have waiting times up to 24 times as long as the FairScheduler. This is caused by the presence of long running jobs. Both benchmarks have jobs that last for very long times, and this clogs up the working queue for the JobQueueTaskScheduler.

4.4 Data Locality

The definition of Data Locaility is described in an earlier chapter and can be found in section 3.3.3 on page 23.

4.4.1 Running Large Jobs

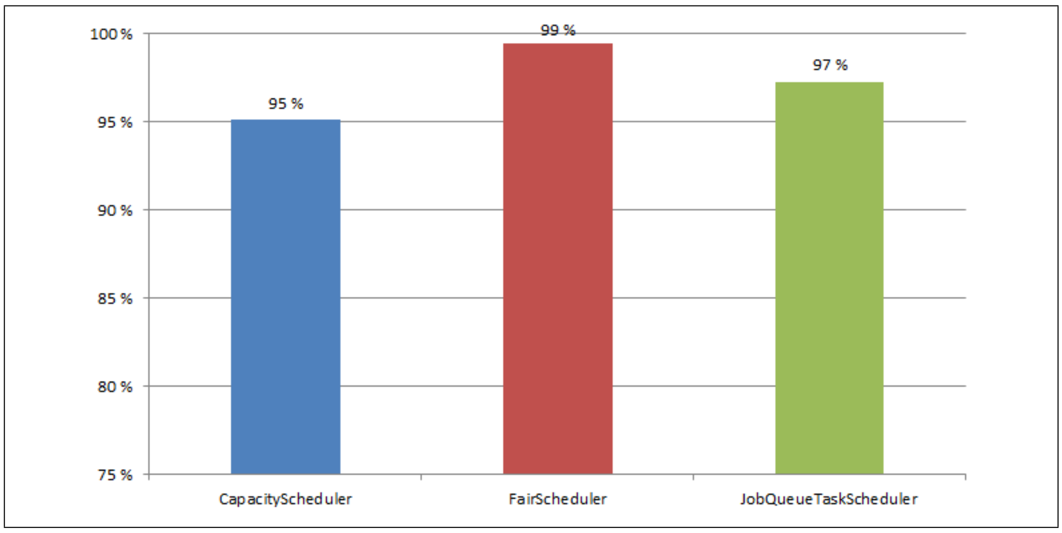

Figure 4.7: Local maps in percentage on Large Benchmark

The bar charts in figure 4.7 shows the percentage of local maps run from all jobs in the Large Benchmark. We see that the FairScheduler has 99 percent of its map tasks run on a node with the data local to it. JobQueueTaskScheduler comes behind with 97 percent local maps and then CapacityScheduler which only managed to run 95 percent of the task on a node with the data local. although the difference is only 4 percent between the schedulers this grows to a large amount of data transferred in the network.

The raw data from the Large Benchmark show that the task scheduler with the worst percentage of local maps had to transfer 95 blocks of data during the benchmark tests, which lasted around 90 minutes in total. When running with a block size of 128 mega byte this transfers to around 128 megabyte per minute.

4.4.2 Running Small Jobs

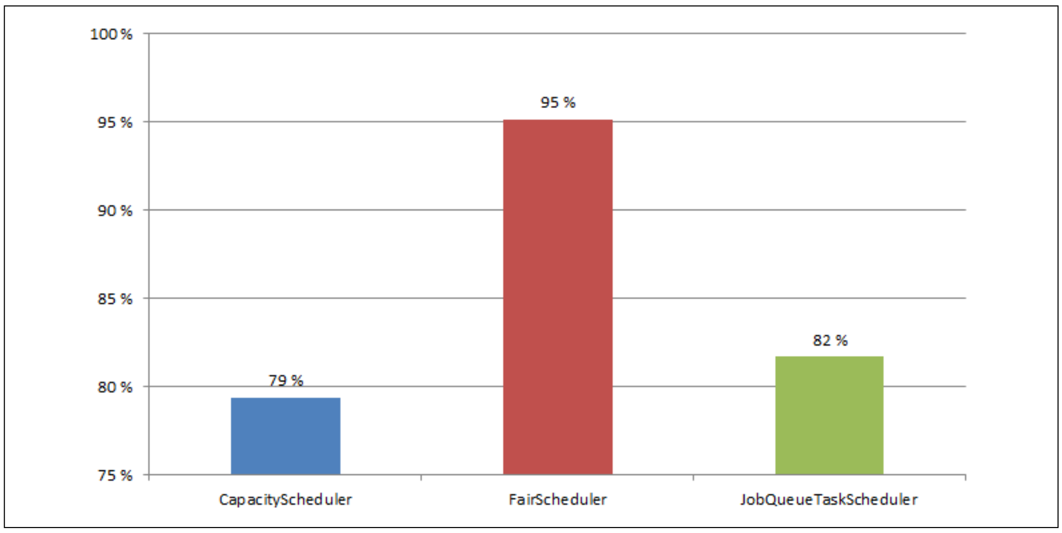

Figure 4.8: Local maps in percentage on Small Benchmark

In figure 4.8 we can see the results from the Small Benchmarks. This figure show a decreasing percentage data locality from the large benchmarks. We can see that the worst results are from the jobs run with the CapacityScheduler. This only manage to run 79 percent of its maps on a node with the data local. The JobQueueTaskScheduler manages 82 percent local maps in these tests. The FairScheduler manages an impressive 95 percent local maps on these tests.

Raw data reveals that the CapacityScheduler in the small benchmark has to move around 400 blocks of data to process. This puts a higher strain on the network that the other schedulers, specially the FairScheduler which only had to transfer 80 blocks of data.

4.4.3 Running Mixed Jobs

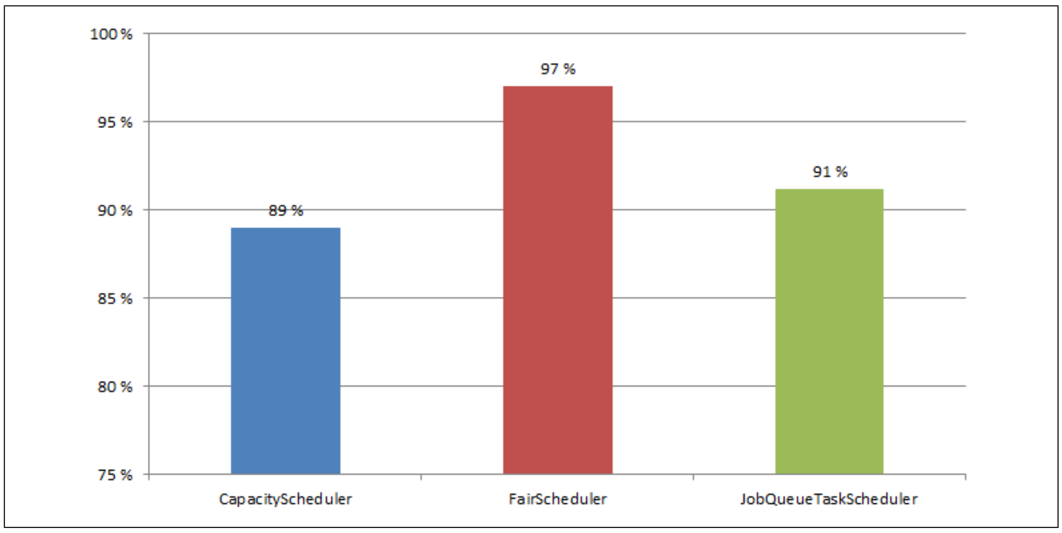

Figure 4.9: Local maps in percentage on Mixed Benchmark

Figure 4.9 is a bar chart displaying the results from the mixed benchmark. Again the FairScheduler has the higher percentage of local maps. It has 97 percent of local maps in this situation. The JobQueueTaskScheduler manages to achieve 91 percent of local maps, while the CapacityScheduler yet again has the worst amount of local maps. This time 89 percent.

Summary

The task schedulers performs quite differently in this task. The FairScheduler has a much higher percentage than any of the two other schedulers. This results in a network with less overall traffic for clusters running the FairScheduler.

The percentage amount of local maps drops with the amount of jobs processed by the cluster. The small benchmark is the worst with 54 jobs submitted within a 1 hour timeframe. Longer running jobs do not suffer as much from this results show.

4.5 Cluster Utilization

The cluster utilization definition is described in an earlier chapter and can be found in section 3.3.4 on page 24.

4.5.1 Running Large Jobs

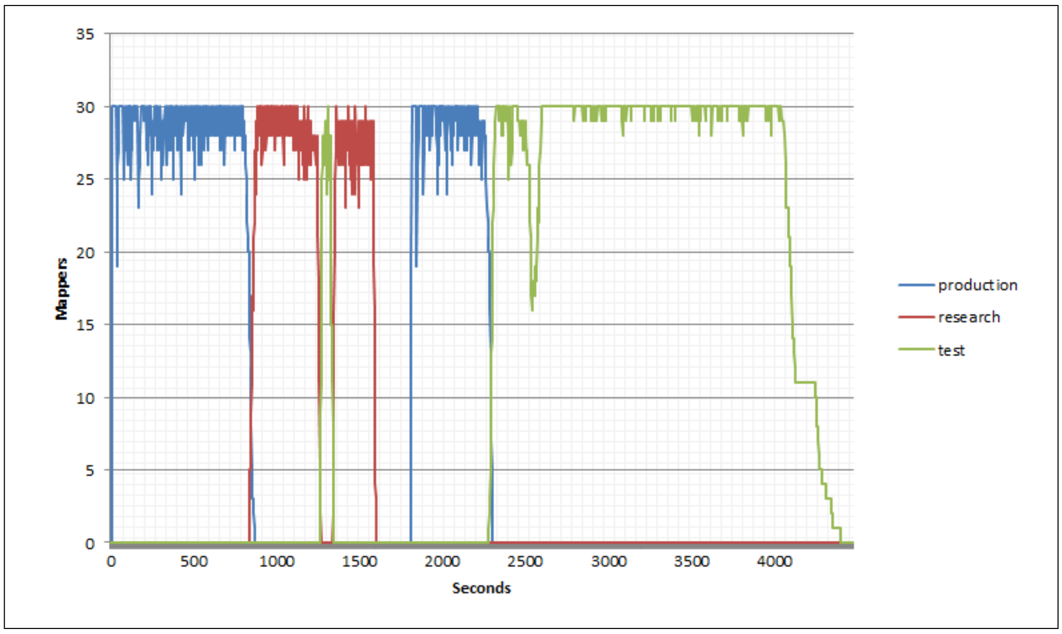

Figure 4.10: Mapper queue status from the Large Benchmark using JobQueueTaskScheduler

In figure 4.10 we can see how much of the mapper slots on the cluster is used throughout the entire benchmark. Since we are running with the JobQueue-TaskScheduler we see that mapper resources are not shared between any queues at the same time. Jobs run in order and get all the 30 available mapper slots available. The Map stage finishes quickly for all jobs and then moves on to new jobs. When 1600 seconds have passed we can see that the mappers have all finished the work assigned. Before a new job is submitted to the production queue at 1800 seconds.

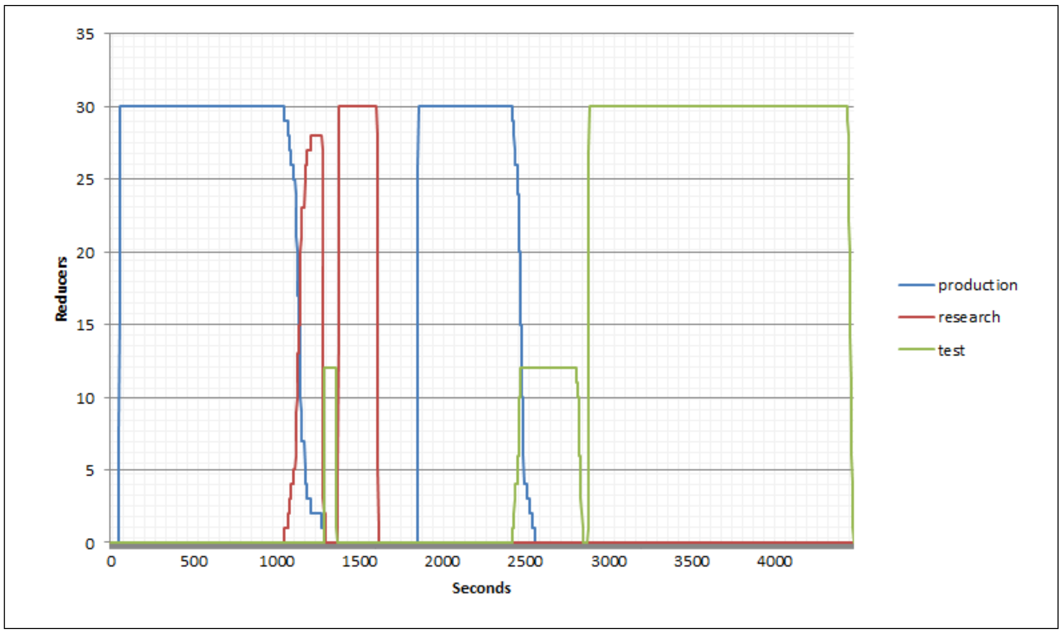

Figure 4.11: Reducer queue status from the Large Benchmark using JobQueueTaskScheduler

Figure 4.11 presents the reducer results from the large benchmark using the JobQueueTaskScheduler. This figure shows that the first job submitted into the production queue starts to reduce the results seconds after starting to process this job. It keep the reducers running to around 1300 seconds. Looking at the results in the previous figure 4.10 on page 51 we can see that this job finishes its mapper at around 800 seconds. We see that the reducers continually work on the output from the job throughout the entire mapper phase.

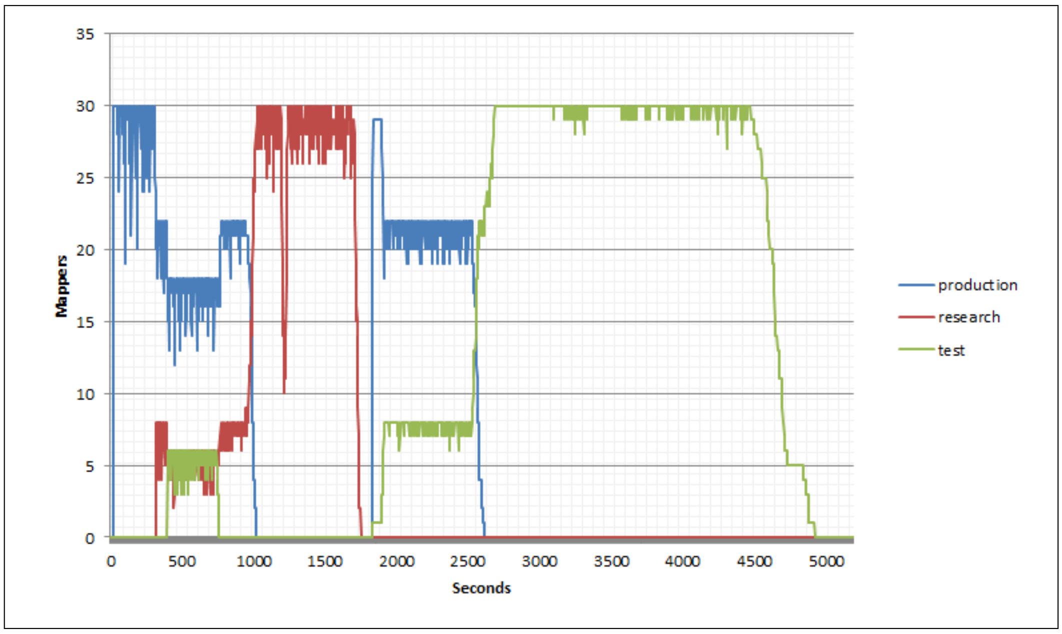

Figure 4.12: Mapper queue status from the Large Benchmark using CapacityScheduler

Displayed in the figure 4.12 are the results from Large Benchmarks using the CapacityScheduler developed by Yahoo. The results are the mapper queue status across the timeline of the benchmark. We can clearly see how this task scheduler distribute the mapper slots to the different jobs throughout the timeline. When there is only one job running we see that they get the entire clusters mapper slots. This is illustrated by the peaks in the graphs. All queues have at one point a job running alone on the cluster.

At around 500 seconds we see that there are three concurrent jobs running. At this point all three queues are represented with the amount of map slots given in the configuration file for the CapacityScheduler. At around 2000 seconds into the benchmark there are two jobs concurrently running. We can then see that both queue have more resources than what is configured. This is why both queue share the spare resources from the research queue. The unused resources are equally split between the remaining active queues.

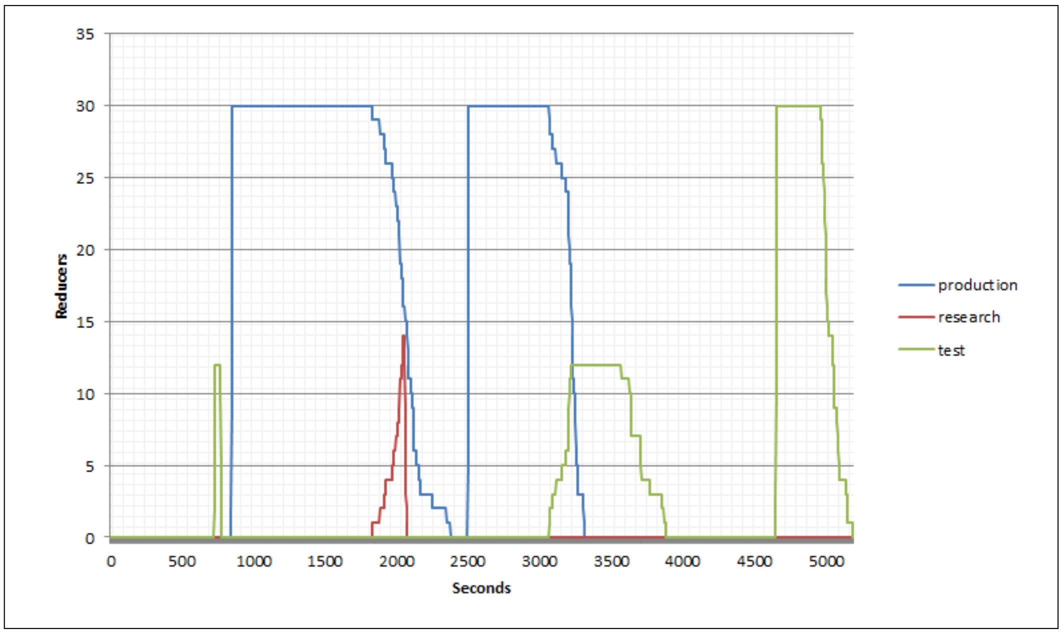

Figure 4.13: Reducer queue status from the Large Benchmark using CapacityScheduler

In figure 4.13 we the the status of the reducer queues for the CapacityScheduler. Here we can see that there are not many queues that runs reducer tasks concurrently. Reducers grab up all 30 available reducers slots were it is possible. Reducers slots are not made available until reducer tasks complete. For jobs with large data outputs this can take a long time. Such as the first job started in the production queue. This is a TeraSort job and has an input size of 70GB of data. The output of this job will also be 70GB. All data have to go through the reducers, so it will take a notable amount of time for it to complete. It can look like it forces jobs to wait until reducers complete. Examples of this can be seen at around 2000 seconds and 3200 seconds. Here the mappers of the jobs in the production queue finish and immediately reducers from other queues are started up. The waiting reducer tasks seem to be very small as they finish shortly after they are launched.

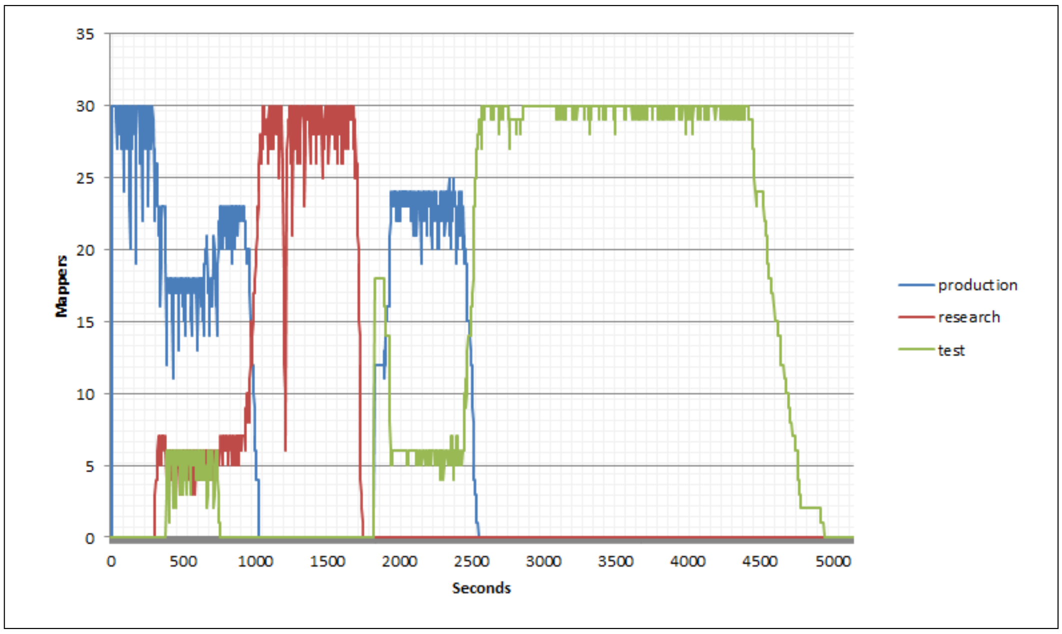

Figure 4.14: Mapper queue status from the Large Benchmark using FairScheduler

Results from the FairScheduler is presented in figure 4.14. Here we can see the mapper status from the large benchmarks. The results show similar results as the CapacityScheduler when regards to how the mapper slots are distributed between the queues. Mappers are distributed as new resources are available in the cluster. An example of this can be seen in the research queues first job. The job launches at around 250 seconds and is only given its configured share of resources. Then the job in the test queue finishes the unused resources are divided between the remaining queues. The FairScheduler divides these resources differently than the CapacityScheduler which only splits them. Since the FairScheduler is configured to use a weight on each queue the overflow of resources are divided unevenly on the remaining queues. This is why the research do only get a few extra mappers to work with, while the production queue receive a larger amount of mappers. When the production job is finished we see that all resources are routed to the research queue to speed up the only job running. During the high peak in the research queue we can see a drop on mappers down to 6 mappers. This is caused by another job launching just when the other job is finished.

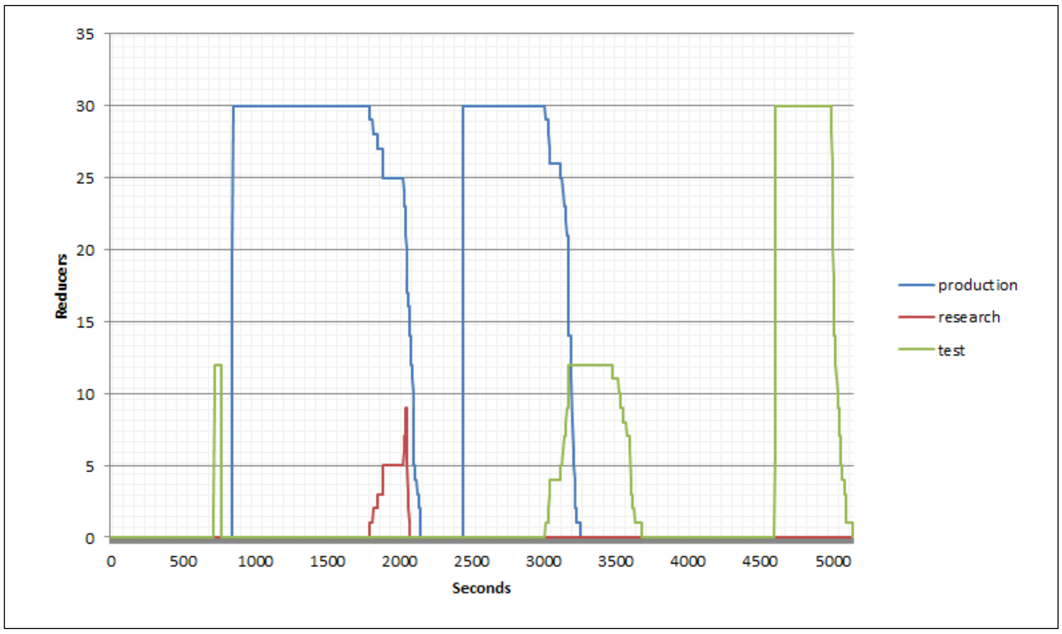

Figure 4.15: Reducer queue status from the Large Benchmark using FairScheduler

In figure 4.15 we can see the results from the FairScheduler run. The results are similar to the CapacitySchedulers results. The start of the reducers are delayed compared the to results from the JobQueueTaskScheduler shown in figure 4.11. Because of delay we can see that the first job in the test queue is allowed to run freely with the mappers it needs. The job is short so it finishes quickly. Had the first job of the production queue launched its reducers before the job in the test queue it would have had to wait until reducers were available. The production queue holds all 30 reducers for about 1000 seconds. When it starts to finish with its reducers a job in the research queue acquire reducers that become available.

Summary

In summary we have seen how the different task schedulers treat jobs that needs resources. The JobQueueTaskScheduler does so in its first-in-first-out way, while the two other try to facilitate jobs into queues and share the resources between jobs in these queues.

4.5.2 Running Small Jobs

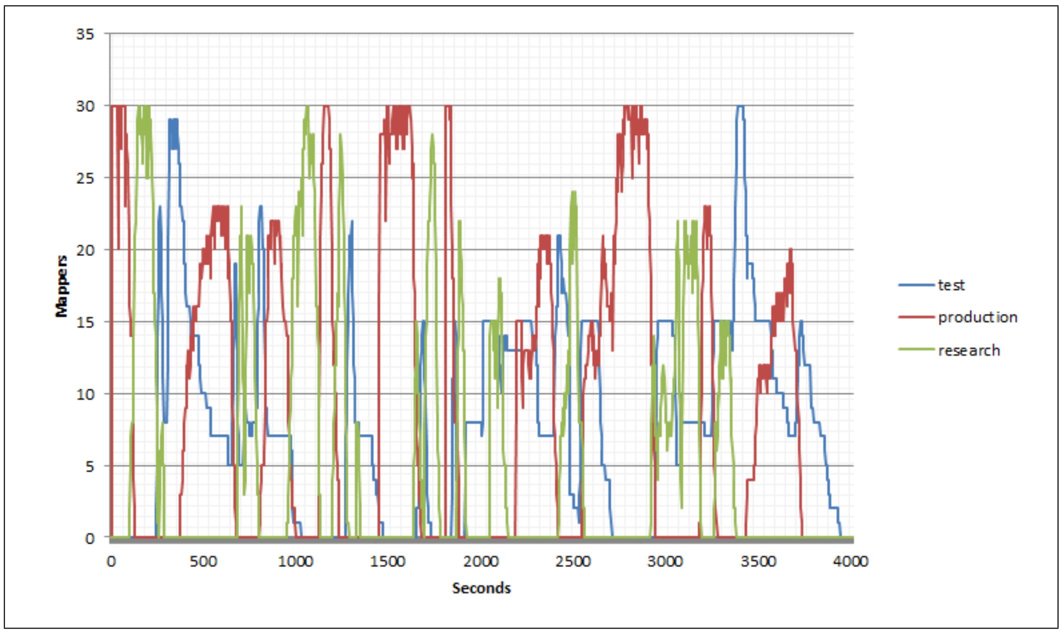

Figure 4.16: Mapper queue status from the Small Benchmark using JobQueueTaskScheduler

Figure 4.16 shows many peaked curves on each queue. Jobs are processed and finished quickly. We can see that some jobs are so small that they are not able to utilize all the mappers available on the cluster. To fully utilize the cluster a job would need to have a input set of at least 30 data blocks of data. Since the data blocks are set to be 128 megabyte the total input would need to be 3840 megabyte. It do not matter how big the input size is when running many jobs since the mapper slots are still being distributed to the next job in the queue. An example of this can be seen at around 500 seconds into the benchmark. The production job do not grab all the mapper slots so a job from the research queue is also started. At this point all the mappers are distributed between two concurrent jobs.

With small jobs there are not much waiting time for each job before it starts to process. But the graph can show that the cluster is busy most of the time by starting up new jobs constantly trough-out the whole benchmark.

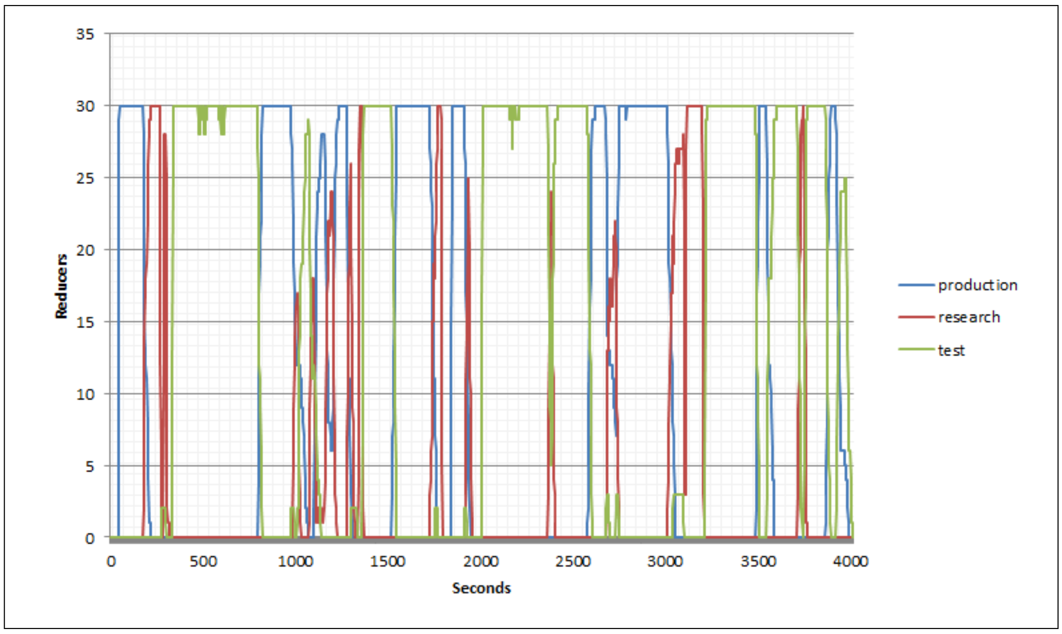

Figure 4.17: Reducer queue status from the Small Benchmark using JobQueueTaskScheduler

Shown in figure 4.17 are the results from the benchmark running small jobs. We see that most of the reduce phases are short and using the maximum amount of reduce slots per job. By using the maximum available reducer slots the job finishes quickly, but also disallows any other job in the queue to start outputting its results to the file system. From the figure we see that the cluster is operating at the maximum amount of reducers nearly all the time throughout the benchmark.

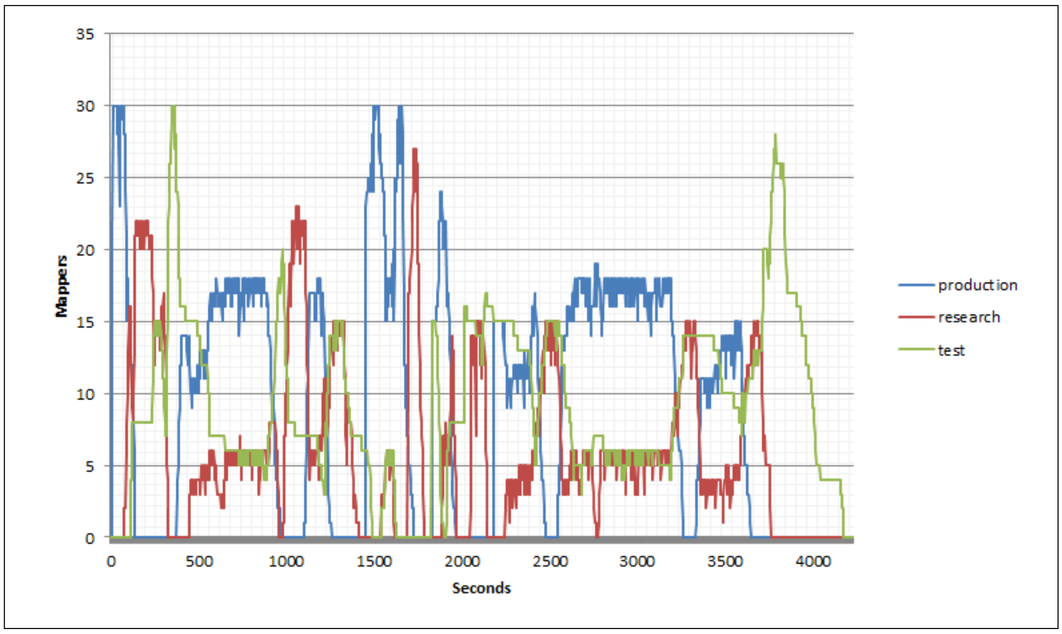

Figure 4.18: Mapper queue status from the Small Benchmark using CapacityScheduler

Running the benchmark on the CapacityScheduler yields the results shown in figure 4.18. Resources are divided across all queues with respect to how the configuration that was set up for the CapacityScheduler. The mapper stage finishes after roughly 70 minutes of processing on the cluster.

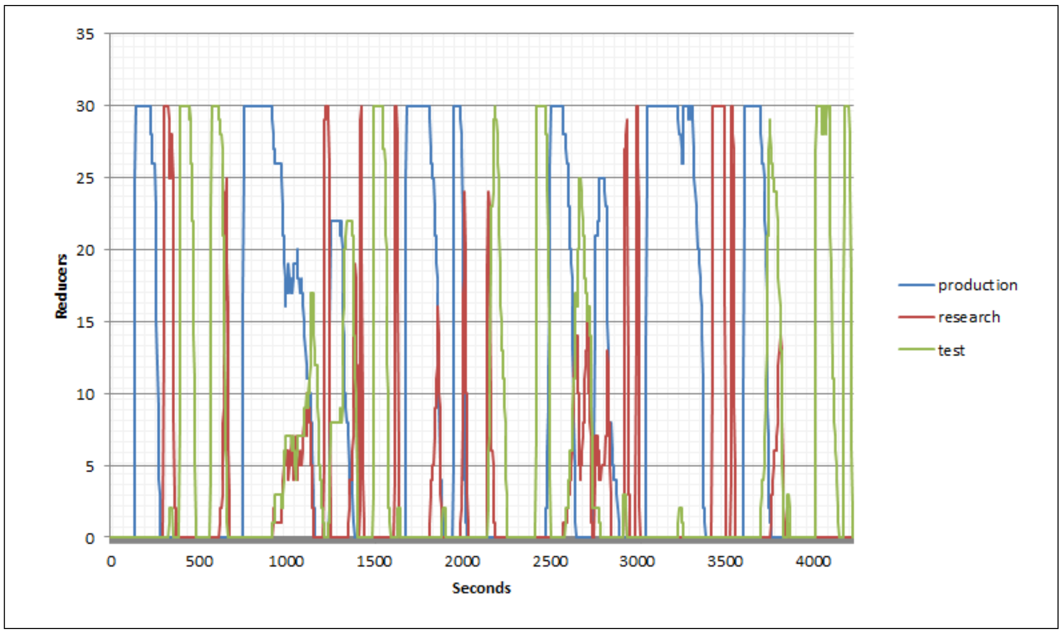

Figure 4.19: Reducer queue status from the Small Benchmark using CapacityScheduler

In figure 4.19 we can see a line chart similar to the JobQueueTaskSchedulers results. Allowing the reducers to start the maximum available amount of reducers causes some delay in the cluster. Since Hadoop only was configured to run a single job at a time this is a good choice. Using the maximum amount of reducers ensures the quickest reduce phase, but with multiple users this can be a challenge. Reducers are designed to run in only one wave, unlike the mappers which usually runs multiple waves. Mapper slots can then between two waves of mappers relocate the clusters resources. This is impossible with the reducers.

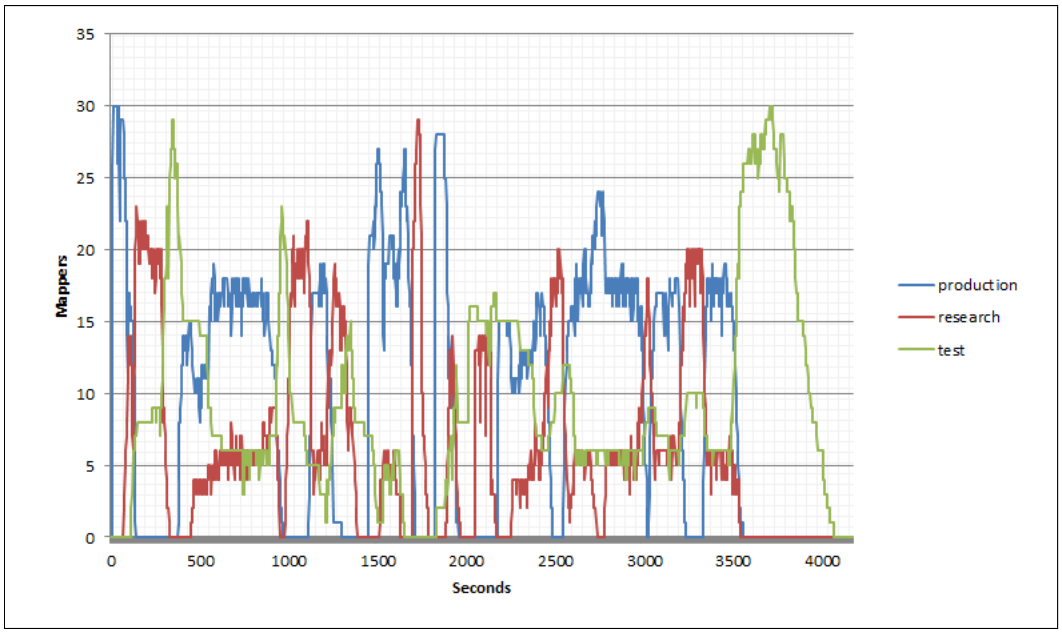

Figure 4.20: Mapper queue status from the Small Benchmark using FairScheduler

Figure 4.20 shows the results from the FairScheduler running the small benchmark. We see from the figure that it acts similar to the CapacityScheduler with distributing the jobs between queues. It follows the rules given in the allocation file for the FairScheduler. All jobs mappers finish just after 4000 seconds.

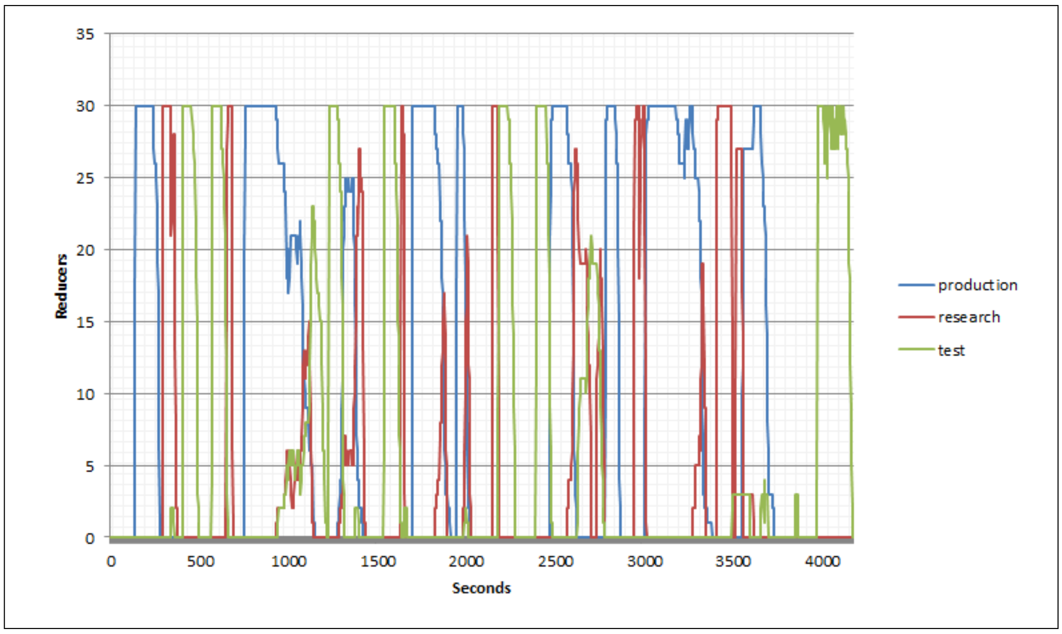

Figure 4.21: Reducer queue status from the Small Benchmark using FairScheduler

In figure 4.21 we see another example on how the jobs claim all the available reducers slots when running. These results from the FairScheduler is similar to those in the two previous task schedulers.

Summary

Summing up the results from the small benchmark we see that the JobQueue-TaskScheduler finishes the mapper phase faster than the two other task schedulers. The task schedulers intended for multi-user clusters shares the clusters resources between the queues according to their configurations. We saw that the reducers are quick when operating with small data, as they should. This do not cause the same problems as with the large jobs seen in the previous section were long running reducer task could interfere with other jobs output.

4.5.3 Running Mixed Jobs

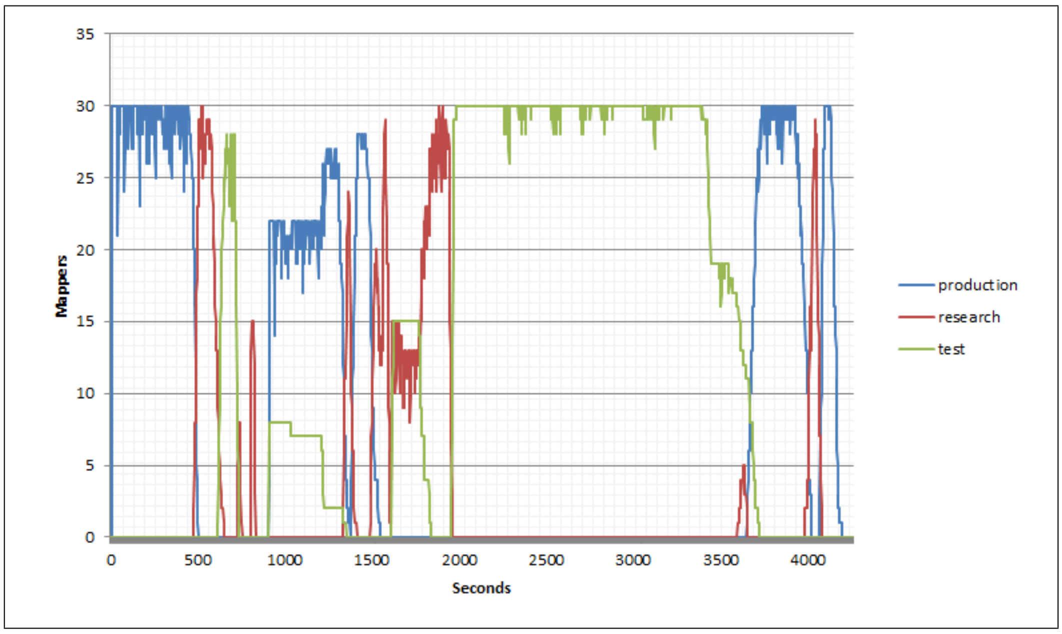

Figure 4.22: Mapper queue status from the Mixed Benchmark using JobQueueTaskScheduler

Running the benchmark with mixed jobs we get some interesting results shown in 4.22. Since the jobs vary in size some jobs do not use all mapper slots, while other jobs do. It will then run the two jobs seemingly in a concurrent order. This can be seen around 1000 seconds into the benchmarks results. Here we see a job from the production queue running together with a job from the test queue.

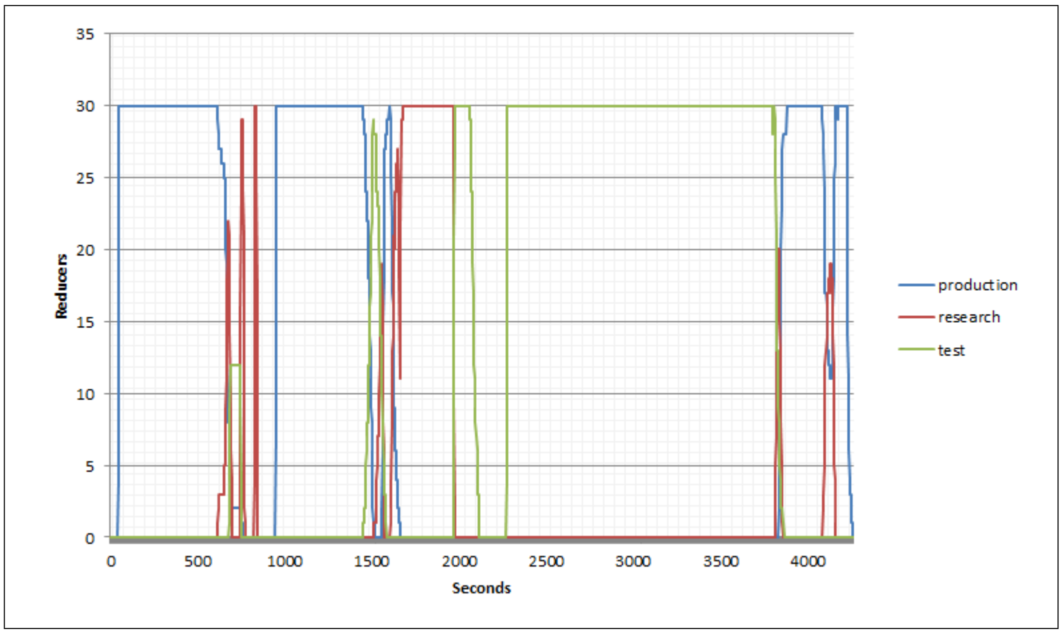

Figure 4.23: Reducer queue status from the Mixed Benchmark using JobQueueTaskScheduler

In figure 4.23 we see the reducer phase from the mixed benchmark with a default installation of Hadoop running the JobQueueTaskScheduler. We see the reducers start outputting data early on in the job. When the reducers from the job in the production queue finish we can see that two reducers start processing immediately after this, and they finish quickly. Through the rest of the benchmark we see how the reducers run one by one, often in rapid succession.

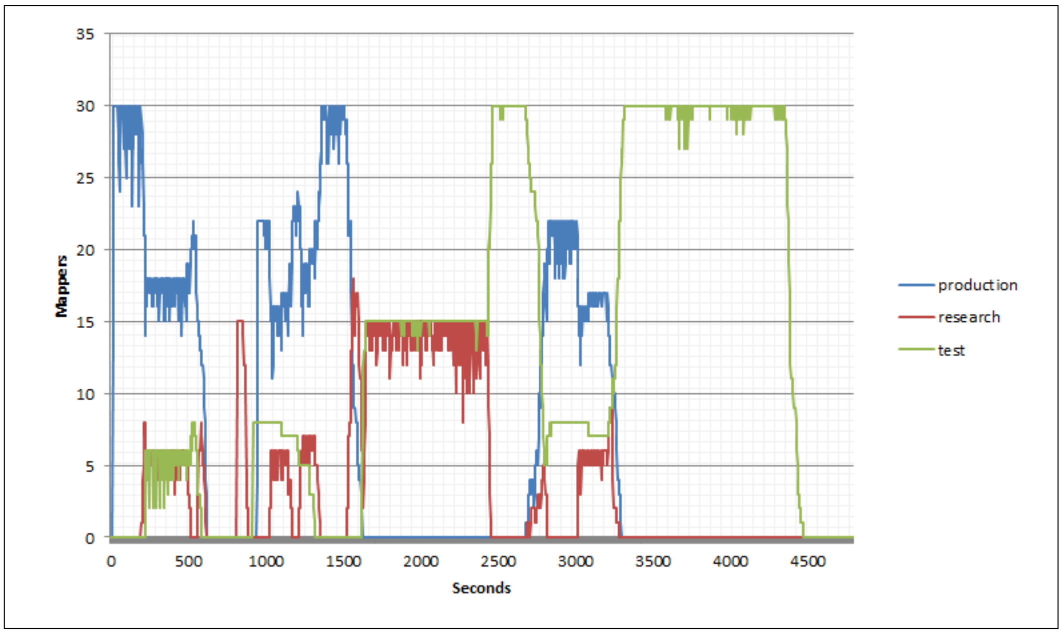

Figure 4.24: Mapper queue status from the Mixed Benchmark using CapacityScheduler

Figure 4.24 contains the results from the mixed benchmark using the CapacityScheduler. Here we can see that the scheduler schedules the small tasks in the presence of a longer running job into the appropriate queue. It uses the entire cluster whenever there are jobs available in the work queue. The entire map phase finishes at 4500 seconds.

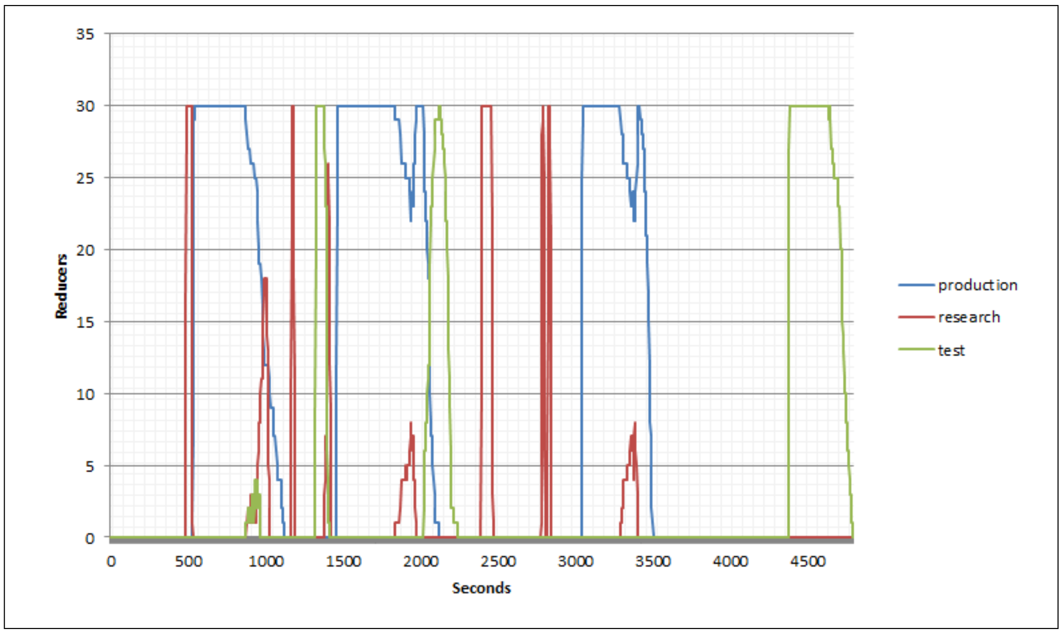

Figure 4.25: Reducer queue status from the Mixed Benchmark using CapacityScheduler

In figure 4.25 we see that the reducer task of the first job starts quite slow. All jobs initialize 30 reducers throughout the benchmark. The number of reducers an job creates can be set when the job is submitted through parameters. In cases with small output this parameter could be set to avoid the job in taking up so many reducers in the cluster.

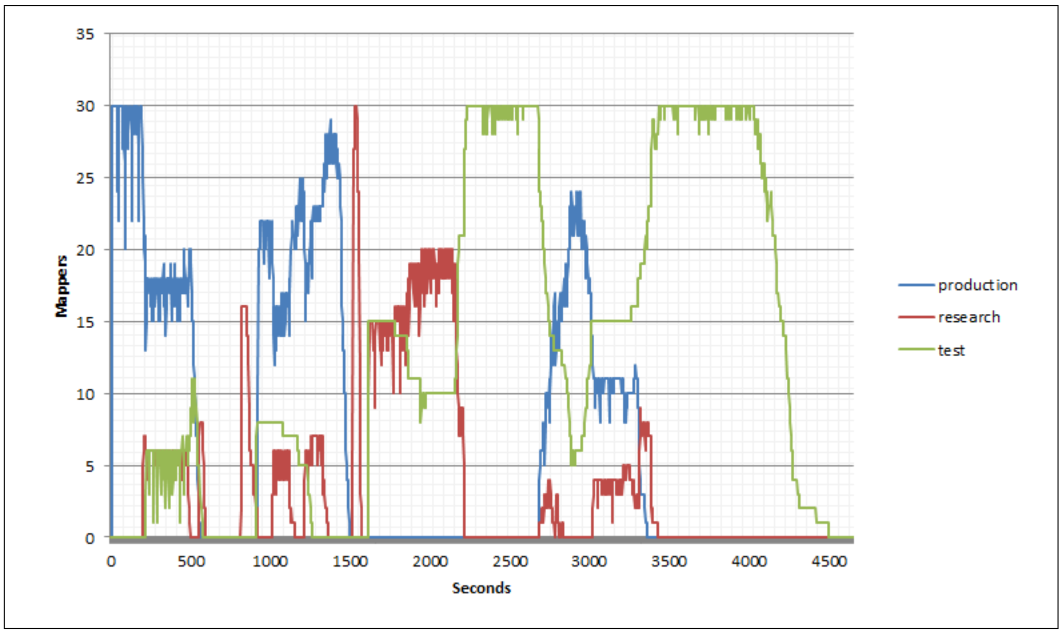

Figure 4.26: Mapper queue status from the Mixed Benchmark using FairScheduler

Figure 4.26 shows similar results as the CapacityScheduler with regards to how the jobs are scheduled. Jobs are start processing shortly after the jobs are submitted to the cluster. They are then put into the queue they belong and processed. Jobs receive the appropriate amount of resources based on the queues configuration. The mapper phase ends at around 4500 seconds, which is similar to the CapacityScheduler, but slower than the JobQueue-TaskScheduler.

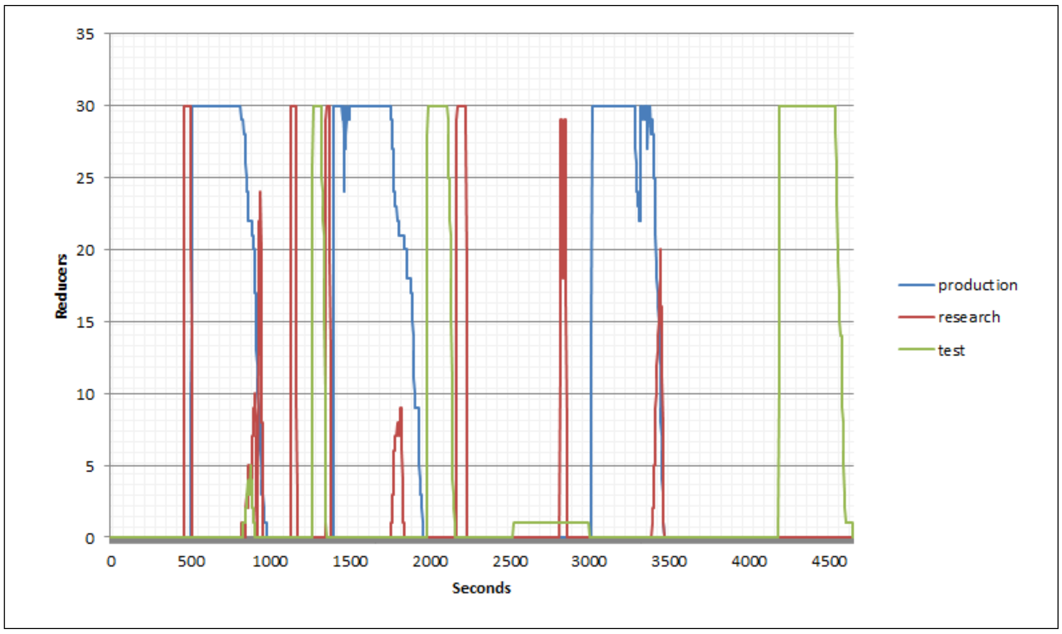

Figure 4.27: Reducer queue status from the Mixed Benchmark using FairScheduler

Another example in figure 4.27 were the reducers is not started until late in the job. Once a job reaches the point were they need to start the reducers they initialize 30 reducers per job.

Summary

In this section we have seen the results from the mixed benchmark. We have noticed the problem with jobs initializing 30 reducers each, every time. This happens regardless the amount of output data which is expected.

Chapter 5

Conclusion

5.1 Summary

In this thesis, we have taken a more in depth look at big data problems, and how to process them quickly in a multi-user scenario. We chose the Hadoop Framework for out tests, as this is one of the most popular frameworks in the open-source map-reduce world. The framework also have much documentation and research published around the problems of running a efficient Hadoop cluster in multi-user environments. We took an extensive look at the framework and the schedulers we were to use in our benchmark. The default task scheduler JobQueueTaskScheduler, as well as the two most common multi-user task schedulers, CapacityScheduler and FairScheduler. Both of them are described in this thesis.

We created a benchmark runner which reads configuration files and launches a set of Hadoop jobs onto our cluster, which we hosted in the cloud on Amazon EC2. Together with this benchmark runner we created thee types of benchmarks to run on our cluster, all running a variety of map-reduce jobs.

Results derived from our benchmarks expresses a need for a highly detailed configuration of Hadoop for it to run successfully on a multi-user Hadoop cluster. The Hadoop framework has a good configuration for running single jobs and it works adequade when running many jobs. You need to expect very long waiting times running a Hadoop cluster with the default task scheduler. The default task scheduler schedules tasks in a first-in-first-out order, which is great for processing map-reduce jobs fast utilizing the entire cluster, but in a multi-user setting there need to be a possible way to get a certain service level. Having your job processed without too much waiting is key in this aspect. Results from the multi-user task schedulers show that tasks usually are launched quite early compared to the JobQueueTaskScheduler.

During our multi-user benchmarks we configured the FairScheduler and CapacityScheduler to use a slow start configuration. Reduce tasks were allowed to run after the map phase had reached 90 percent completion. The default scheduler used a 5 percent slow start configuration. Results from the large and mixed benchmark show that this might be a bad idea in the presence of map-reduce jobs that output much data. By delaying the reduce phase the overall run time of jobs using the FairScheduler and CapacityScheduler suffered. When running small jobs there was nearly no effect from the slow start configuration.

Our jobs were launched without specifying any number of reducers. This caused the job to acquire the maximum possible reducers for the cluster, which were 30 in our cluster. Because of this many jobs were delayed as some reduce phases were running a long time. This could have been avoided with setting the allowed number of maps for a queue to a maximum, which was not the entire cluster. Also, the user submitting the job can and should need to specify a number of reducers which fits the amount of data the user want to output. Not specifying the number of reducers affects the entire clusters ability to handle multiple users concurrently as it congest the reducers for a long period of time.

The multi-user task schedulers do good job of re-assigning new jobs the needed resources that are available. They follow a configuration set by the administrator closely. Both task schedulers have the ability to deal with misbehaving jobs which will not complete in a timely order, altho this feature wasn't used in our tests. Running short map task increases the response time for the rescheduling of new jobs, but this has it tradeoff in the number of messages sent throughout the cluster. With small map task there is a need to send more heartbeat messages throughout the cluster assigning new tasks to nodes.

To successfully run a Hadoop cluster with multiple users there is a need to configure the cluster after how the usage of the cluster is. How to configure the cluster is highly dependent on the workload on the specific cluster. With multiple users the cluster should be configured to not hijack resources for longer periods of time. Reducers are possible abusers of resources as they can take long time to finish. Users should be encouraged to write jobs that respect the available resources in the cluster. From these benchmark runs the need to put a maximum cap on the number of reducers which can be obtained by each queue is highly encouraged. This reduces the impact on other queues jobs and also limits users of high-jacking all resources, either by a mistake or on purpose.

From our benchmarks we saw that the default JobQueueTaskScheduler was the most efficient task scheduler when looking purely at the run time of each job. The multi-user scheduler did finish at similar results. Due to the fact that we ran the multi-user task schedulers with a reducer slow start this impacted the running time greatly since the reducers started to process too slow into the job. This specially happened with the large jobs we ran on the cluster. When we ran smaller jobs this did not have such a big effect on the running time.

5.2 Future work

Due to time constraints, we were unable to thoroughly explore all scenarios around configuring and running benchmarks on a Hadoop cluster. Running a Hadoop cluster with multiple users submitting jobs throughout the day requires a cluster that can handle different types of jobs and timings. In our research we merely scratched the surface of what comes to configuring clusters to specification needed by the user. More extensive research could be done in the field of how to configure a cluster to certain types of workloads. Since we tried to keep the benchmarks within one to two hours each we did not have the chance on processing really big data sets. It could be interesting to see the results from experiments as such.

More work could be done on developing a standardized test suite and more configurable for multi-user Hadoop clusters. In our test suite simply contained a invented workload which might not be applicable to other users. Having a test suite that can simulate or recreate a workload from an actual history file would be useful when doing performance analysis on other workloads.

Also, running the test on a local system could be beneficial for the results in our tests since we are not in control both the network and the machines when running on Amazon EC2. Having a local system could prove useful for controlling the cluster.

Bibliography

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

[2] Apache Nutch. http://nutch.apache.org/.

[3] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/, 2012.

[4] Cloudera homepage. http://www.cloudera.com/, 2012.

[5] Datameer homepage. http://www.datameer.com/, 2012.

[6] Google Trends - Big Data. http://www.google.com/trends/?q=big+data, 05 2012.

[7] Google Trends - Hadoop. http://www.google.com/trends/?q=hadoop, 05 2012.

[8] Hortonworks homepage. http://www.hortonworks.com/, 2012.

[9] Mapr homepage. http://www.mapr.com/, 2012.

[10] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[11] Cisco. Cisco Visual Networking Index: Forecast and Methodology, 2010-2015. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360.pdf, 6 2011.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] Eric Eldon. Live-Blogging a Facebook Chat Product Launch at the Start of "Launching Season 2011". `http://www.insidefacebook.com/2011/07/06/live-blogging-a-facebook-product-launch-at-the-start-of-launching-season-20` 7 2011.

[14] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google File System. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[15] IBM. What is big data? `http://www-01.ibm.com/software/data/bigdata/`, May 2012.

[16] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[17] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[18] Ryan Rosario.

[19] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[20] International Telecommunications Union. Internet users per 100 inhabitants 2001-2011. `http://www.itu.int/ITU-D/ict/statistics/material/excel/2011/Internet_users_01-11.xls`, 2012.

[21] M. Zaharia, D. Borthakur, J.S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55, Apr*, pages 2009–55, 2009.

Appendix A

Apache Whirr Configuration

Listing A.1: Configuration file used by Whirr

```
# For EC2 set AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY
    environment variables.
whirr.provider=aws-ec2
whirr.identity=${env:AWS_ACCESS_KEY_ID}
whirr.credential=${env:AWS_SECRET_ACCESS_KEY}

# Change the cluster name here
whirr.cluster-name=Hadoop

# Change the number of machines in the cluster here
whirr.instance-templates=1 hadoop-namenode+hadoop-jobtracker,15
    hadoop-datanode+hadoop-tasktracker
# The size of the instance to use. See http://aws.amazon.com/ec2
    /instance-types/
whirr.hardware-id=m1.large
# Ubuntu 11.10 Oneiric instance-store ami-c2ba68ab
# See http://alestic.com/
whirr.image-id=us-east-1/ami-c2ba68ab
whirr.location-id=us-east-1

# Hadoop Configuration
hadoop-mapreduce.mapred.jobtracker.taskScheduler=org.apache.
    hadoop.mapred.FairScheduler
hadoop-mapreduce.mapred.jobtracker.taskScheduler=org.apache.
```

```
    hadoop.mapred.CapacityTaskScheduler

hadoop-mapreduce.mapred.tasktracker.map.tasks.maximum=2
hadoop-mapreduce.mapred.tasktracker.reduce.tasks.maximum=2
hadoop-mapreduce.mapred.fairscheduler.poolnameproperty=mapred.
    job.queue.name
hadoop-mapreduce.mapred.queue.names=production,research,test
hadoop-mapreduce.mapred.reduce.slowstart.completed.maps=0.9
hadoop-mapreduce.reduce.parallel.copies=20

hadoop-hdfs.dfs.permissions=false

# Expert: specify the version of Hadoop to install.
whirr.hadoop.version=1.0.1
whirr.hadoop.tarball.url=http://joachse.at.ifi.uio.no/hadoop-${
    whirr.hadoop.version}.tar.gz
```

Appendix B

Job Scheduler Configuration

B.1 JobQueueTaskScheduler

a

B.2 FairScheduler

Listing B.1: Configuration of Production Queue using Fair Scheduler

```
<allocations>
  <pool name="production">
    <minMaps>18</minMaps>
    <minReduces>18</minReduces>
    <maxMaps>30</maxMaps>
    <maxReduces>30</maxReduces>
    <weight>7</weight>
  </pool>
```

Listing B.2: Configuration of Research Queue using Fair Scheduler

```
<pool name="research">
  <minMaps>6</minMaps>
  <minReduces>6</minReduces>
  <maxMaps>30</maxMaps>
  <maxReduces>30</maxReduces>
  <weight>2</weight>
</pool>
```

Listing B.3: Configuration of Test Queue using Fair Scheduler

```
<pool name="test">
  <minMaps>6</minMaps>
  <minReduces>6</minReduces>
  <maxMaps>30</maxMaps>
  <maxReduces>30</maxReduces>
  <weight>1</weight>
</pool>
```

B.3 CapacityTaskScheduler

Listing B.4: Configuration of Production Queue using Capacity Scheduler

```
<property>
  <name>mapred.capacity-scheduler.queue.production.capacity</
      name>
  <value>60</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.production.maximum-
      capacity</name>
  <value>-1</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.production.supports-
      priority</name>
  <value>false</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.production.minimum-user-
      limit-percent</name>
  <value>100</value>
```

```
</property>

<property>
  <name>mapred.capacity-scheduler.queue.production.user-limit-
      factor</name>
  <value>100</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.production.maximum-
      initialized-active-tasks</name>
  <value>200000</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.production.maximum-
      initialized-active-tasks-per-user</name>
  <value>100000</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.production.init-accept-
      jobs-factor</name>
  <value>10</value>
</property>
```

Listing B.5: Configuration of Research Queue using Capacity Scheduler

```
<property>
  <name>mapred.capacity-scheduler.queue.research.capacity</name>
  <value>20</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.research.maximum-
      capacity</name>
  <value>-1</value>
</property>

<property>
```

```
  <name>mapred.capacity-scheduler.queue.research.supports-priority</name>
  <value>false</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.research.minimum-user-limit-percent</name>
  <value>100</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.research.user-limit-factor</name>
  <value>100</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.research.maximum-initialized-active-tasks</name>
  <value>200000</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.research.maximum-initialized-active-tasks-per-user</name>
  <value>100000</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.research.init-accept-jobs-factor</name>
  <value>10</value>
</property>
```

Listing B.6: Configuration of Test Queue using Capacity Scheduler

```
<property>
  <name>mapred.capacity-scheduler.queue.test.capacity</name>
  <value>20</value>
```

```
</property>

<property>
  <name>mapred.capacity-scheduler.queue.test.maximum-capacity</name>
  <value>-1</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.test.supports-priority</name>
  <value>false</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.test.minimum-user-limit-percent</name>
  <value>100</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.test.user-limit-factor</name>
  <value>100</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.test.maximum-initialized-active-tasks</name>
  <value>200000</value>
</property>

<property>
  <name>mapred.capacity-scheduler.queue.test.maximum-initialized-active-tasks-per-user</name>
  <value>100000</value>
</property>

<property>
```

```
  <name>mapred.capacity-scheduler.queue.test.init-accept-jobs-
      factor</name>
  <value>10</value>
</property>
```

Appendix C

Benchmark Configuration

Configuration files can be downloaded from `http://joachse.at.ifi.uio.no/files/bench-conf.tar.gz`.

Appendix D

Source code

Source code can be downloaded from `http://joachse.at.ifi.uio.no/files/`.