Simulation and Performance Evaluation of Hadoop Capacity Scheduler

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By

Jagmohan Chauhan

Permission to Use

Abstract

MapReduce is a parallel programming paradigm used for processing huge datasets on certain classes of distributable problems using a cluster. Budgetary constraints and the need for better usage of resources in a MapReduce cluster often make organizations rent or share hardware resources for their main data processing and analysis tasks. Thus, there may be many competing jobs from different clients performing simultaneous requests to the MapReduce framework on a particular cluster. Schedulers like Fair Share and Capacity have been specially designed for such purposes. Administrators and users run into performance problems, however, because they do not know the exact meaning of different task scheduler settings and what impact they can have with respect to the resource allocation scheme across organizations for a shared MapReduce cluster. In this work, Capacity Scheduler is integrated into an existing MRPERF simulator to predict the performance of MapReduce jobs in a shared cluster under different settings for Capacity Scheduler.

A few case studies on the behaviour of Capacity Scheduler across different job patterns etc. using integrated simulator are also conducted.

Acknowledgements

I would like to take this opportunity to thank and express my gratitude to the people who helped me and made the successful completion of this thesis possible. First and foremost, I would like to express my genuine gratitude and sincere appreciation to my supervisors Dr. Dwight Makaroff and Dr. Winfried Grassmann who helped me all the way from the beginning of my study at U. of S. When I started my program at U. of S., I had very little idea about doing research. Both of my supervisors guided me in the right directions from the beginning of my program and helped me understanding the art of research. They provided me with lots of support, encouragement, motivation and ideas. They extended their helping hands whenever I needed any suggestions. I also appreciate their efforts in correcting my thesis which took a lot of their valuable time and yet they were patient. Besides my supervisors, I would like to thank the rest of the members of my thesis committee: Dr. Derek Eager, Dr. Nate Osgood and Dr. Keith Willoughby for their suggestions and insightful comments. I am very thankful to the DISCUS lab friends and also friends at U of S who provided unconditional support and encouragement for the successful completion of my thesis. I also thank the HPC team at the University of Saskatchewan to provide access to the Socrates cluster to do the real cluster and simulation experiments. Last but not the least, I would like to express my sincere gratitude to the almighty, my parents, siblings and my fiance who were always there for me.

Contents

List of Tables

List of Figures

List of Abbreviations

ARIA	Automatic Resource Inference and Allocation
DFS	Distributed File System
FIFO	First In First Out
GFS	Google File System
HDFS	Hadoop Distributed File System
HTTP	Hyper Text Transfer Protocol
LATE	Longest Approximate Time to End
MRPERF	MapReduce Performance Simulator
RAID	Redundant Array of Independent Disks
TCL	Tool Command Language
XML	Extensible Markup Language

Chapter 1

Introduction

MapReduce is a parallel programming paradigm used for processing big datasets on certain classes of distributable problems using a cluster [13]. Budgetary constraints and the need for better usage of resources in a MapReduce cluster often motivate different organizations to rent or share hardware resources for their big data processing and analysis tasks. Thus, there may be many competing jobs from different clients performing simultaneous requests to the MapReduce framework on a particular cluster. Schedulers like Fair Share[1] and Capacity[2] have been specially designed for such purposes. Capacity Scheduler offers a number of configuration parameters to control the scheduler behaviour in the cluster. The administrators have to often make decisions about changing these parameters to suit different organizational needs. In such cases, running the actual workload on the cluster and then fine tuning the cluster is a daunting task. Often, the administrators do not know how changing a certain set of parameters will affect the overall performance of the workload which normally runs in the cluster. Similarly, a cluster may be reduced or extended in terms of the number of nodes or their capabilities. In such cases as well, it is difficult for an administrator to predict the performance of MapReduce workload without actually running the jobs on the desired cluster. Hence, designing a Capacity Scheduler simulator which can help administrators in the above mentioned tasks without actually running the jobs will save both time and money, and will be a cost effective solution.

The other major benefit of having a simulator is that it can help to test new configuration parameters and/or functionalities if any are added in future in Capacity Scheduler to see their effect on MapReduce jobs. Further, there exist no studies which characterize the behaviour of Capacity Scheduler across different job submission patterns, and data layout scheduling algorithms, etc. The simulator designed in this work is used to undertake such a study. Interesting and important information is revealed about how Capacity Scheduler works under different scenarios. The rest of the chapter discusses the background, the thesis motivation, thesis statement, thesis contributions, and rest of the thesis organization.

[1]http://hadoop.apache.org/common/docs/r0.20.2/fairscheduler.html (5 June, 2013)
[2]http://hadoop.apache.org/common/docs/r0.20.2/capacityscheduler.html (5 June, 2013)

1.1 Data Intensive Computing

Companies, research and government organizations are producing huge amount of data along with World Wide Web and social networking sites [12]. The amount of such data is terabytes or petabytes in size and is often referred to as Big Data [1]. The applications working on Big Data in a parallel manner to produce meaningful information are called data intensive computing applications. Such applications devote most of their time to disk activity and network activities such as the movement and manipulation of data. Movement of data is quite common in data intensive applications where data is to be carried over the network between different nodes for further processing. For example, a MapReduce data intensive application needs a high bandwidth data access rate between the nodes placed on different racks. Manipulation of data is also often done in data intensive applications. For example, a MapReduce data intensive application invokes map and reduce functionality on huge amounts of data to produce the final output data. The analysis and processing of Big Data is crucial for the success of many organizations and data intensive computing is intended to address this need. Over the years, a number of system architectures have been developed for data-intensive computing. One such architecture is parallel and distributed relational databases [14], which is serving the industry for the last 20 years using clusters of nodes that perform no sharing. However, with the explosion of unstructured data, new processing frameworks were required. Several solutions emerged including the MapReduce architecture. MapReduce was developed by Google. An open source framework based on MapReduce called Hadoop[3] emerged in 2007 and is used by premier organizations like Yahoo!, Facebook, and others.

1.2 MapReduce

MapReduce was developed by Google in 2004. The power of MapReduce lies in its ease of use. The user defines a map and reduce function using languages like Java, Python etc. The other key components of distributed parallel processing environment like partitioning of the input data, scheduling and executing the tasks in the cluster and networking between the cluster nodes are automatically done and hidden from the user.

The map function operates on input data, which is in the form of key/value pairs. The map function produces output data in the form of a set of intermediate key/value pairs. All the values associated with a same intermediate key are grouped together and send to the reduce function. The reduce function takes intermediate key value pairs and merge these values to form a smaller set of values. Typically just at most one output value is produced per reduce invocation. For example, if the application is *Sort*, then the output after map and reduce processing will be of same size as the input. In Sort, the mapper is the predefined IdentityMapper and the reducer is the predefined IdentityReducer, both of which just pass their inputs

[3]http://hadoop.apache.org (5 June, 2013)

directly to the output. *Grep* is an application, where one has to find all the occurrences of a text string in a given input. If that text string does not exist in the given input, the output of map processing will be zero bytes. This will result in a reduce phase with no processing, as there will be no map output.

Example: For a word counting problem in a collection of documents, a user would write code similar to the following pseudo-code:

```
map(String key, String value):
key: document name
value: document contents
Parse the document content, word by word.
For each word, emit intermediate key value pair as (word,1).

reduce(String key, Iterator values):
key: a word
values: a list of counts
Count all the occurrences of every unique word and sum them up.
Output the result which shall contain every unique word with its number of occurrences.
```

A MapReduce job consists of the following processing stages and a typical execution is shown in Figure 1.1:[4]

1. Map Operations: Map tasks involve the following actions: [5]

    (a) Map Processing: HDFS is the user level filesystem present in Hadoop. HDFS splits the large input data set into smaller data blocks (64 MB by default). Data blocks are provided as an input to the map function. The block data is split into key value pairs based on the Input Format. The map function is invoked for every key value pair in the input. The output generated by the map function is written in a circular memory buffer, associated with each map. Before writing to the disk, the background thread divides the data into partitions (based on the partitioner used). Each map function output is allocated to a particular reducer by the application's partition function.

    (b) Spill: A background thread spills the contents to the disk when the buffer size reaches a threshold. During the spill process, Map continues to write data to the buffer unless it is full.

    (c) Sorting: An in-memory sort is performed on keys. The sorted output is provided to a combiner function (if used in the program). Combiners is a function which works to aggregate the Map output before it goes to the reducer. This minimizes the amount of data which will be shuffled across the network.

[4]http://developer.Yahoo!.com/hadoop/tutorial/module4.html (15th May, 2013)
[5]http://hadoop-toolkit.googlecode.com/files/White%20paper-HadoopPerformanceTuning.pdf (15th May, 2013)

(d) Merging: Before the map task is finished, the spill files are merged into a single partitioned and sorted output file.

(e) Compression: The map output can be compressed before writing to the disk to save disk space, and to minimizes the data which will be shuffled across the network. By default compression is not enabled.

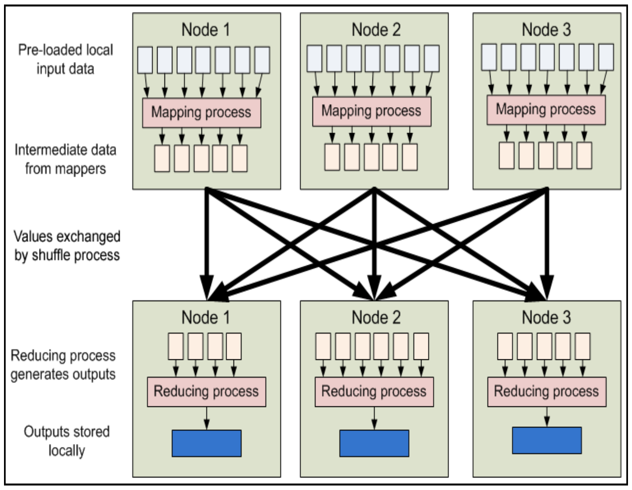

Pre-loaded local input data
Intermediate data from mappers
Values exchanged by shuffle process
Reducing process generates outputs
Outputs stored locally
Node 1
Node 2
Node 3
Mapping process
Reducing process

Figure 1.1: High-Level MapReduce Pipeline

2. Reduce Operations: The Reducer has three phases:

(a) Shuffle: The map outputs are transferred across the network to the reducers. The map output is copied to the reducer's memory buffer. When the data to be copied reaches above a certain threshold, it is merged and spilled to disk.

(b) Sort or Merge: This phase starts when all the map tasks have been completed and their output has been copied. Map outputs are merged and their sorting order is maintained.

(c) Reduce: In this phase, reduce function is invoked for each key in the sorted output. The output of this phase is written to the HDFS.

1.3 Hadoop

Hadoop is an open source platform that was designed to support data intensive applications by Yahoo!. Hadoop is based on Google's MapReduce and Google File System (GFS) [16]. A Hadoop cluster consists of 3 main components: a dedicated NameNode server to provide file system support, a secondary NameNode, which generates snapshots of the NameNode's metadata and a JobTracker server to manage job scheduling decisions. The architecture and job scheduling in Hadoop is explained in more detail in Chapter 2.

1.4 Thesis Motivation

The Task Scheduler is an important part of the MapReduce framework. Initially, MapReduce was designed to handle batch-oriented jobs and a FIFO task scheduler was suited to handle such jobs. However, later the need to have better data locality, better response time for short jobs and the need to share clusters among different organizations or between different groups of the same organizations led to the development of various other schedulers, like FairShare, [6] Delay aware [38], Capacity,[7] Quincy [21], etc.

The Capacity Scheduler was specially designed to share the capacity of the cluster among the different organizations fairly. Different organizations may be forced at times, due to budgetary constraints for example to, to change the resource allocation scheme in the cluster. Different configuration parameters exist in the Capacity Scheduler for the administrators to vary. The settings affect how the resources in the cluster are shared among different queues and users of the organization.

Preliminary investigating experiments were conducted on the Capacity Scheduler and found that these task scheduler settings can have significant impact on the performance of MapReduce jobs [9]. This is considered an important finding, because changing the resource allocation scheme for a cluster can have a big impact on the performance of running jobs of different organizations and hence it is important to estimate it before making any such changes. Therefore, one of the major motivations of this work is to design a Capacity Scheduler simulator to provide administrators with a tool to help them to find the estimated performance of MapReduce jobs if the resource allocation scheme is changed for a cluster.

Some of the other motivations which prompted this work are the following:

1. Organizations often do not have adequate resources to build a real cluster to test the Capacity Scheduler. Running jobs on a real cluster using the Capacity Scheduler to check their performance is also time consuming. In both these cases, the Capacity Scheduler simulator can be a handy tool which saves time and cost.

2. The Capacity Scheduler simulator also helps to characterize the behaviour of running jobs on a MapReduce cluster using the Capacity Scheduler across different job submission patterns, different job mixes, different network topologies and different cluster configurations.

1.5 Thesis Statement

The aim of this thesis is to determine how the MapReduce Capacity Scheduler parameter settings influence the performance of MapReduce applications and to create a Capacity Scheduler simulator. This is achieved through performance measurement, integration of the Capacity Scheduler into a MapReduce Simulator, and scalability experiments in the simulation environment.

[6]http://hadoop.apache.org/common/docs/r0.20.2/fairscheduler.html (19th May, 2013)
[7]http://hadoop.apache.org/common/docs/r0.20.2/capacityscheduler.html (19th May, 2013)

1.6 Thesis Contributions

This thesis explores the Capacity Scheduler and its parameters in detail. It makes the following contributions:

- This thesis is the first systematic study to understand Capacity Scheduler parameters and their impact on the performance of MapReduce applications.
- This thesis contributes a component of the simulator which can simulate Capacity Scheduler.
- The simulator developed in this thesis was verified against real cluster results on representative MapReduce applications.
- A large scale simulation was done in the thesis under different job submission patterns. The findings gathered from this simulation study can be used by the administrators to effectively manage a shared cluster using Capacity Scheduler.

1.7 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 describes related work. Chapter 3 discusses the design and implementation of the Capacity Scheduler integration into MRPERF (an existing MapReduce simulator). Chapter 4 contains the experimental design used in the thesis. Chapter 5 presents the initial results of the performance of MapReduce applications on a real cluster under varying the Capacity Scheduler parameters settings along with validation results, where simulation results are compared with real world experimental results for a small set of jobs. Chapter 6 contains the detailed analysis of a large scale simulation done on a 31 node cluster with different job submission pattern on a MapReduce workload containing a heterogeneous job mix. Finally, chapter 7 presents the conclusions along with future work.

Chapter 2

Background and Related Work

This chapter describes Hadoop in a more detailed fashion to give adequate background to the reader. Firstly, the architecture of Hadoop and its important components like JobTracker, NameNode, etc, are described. The discussion then moves on to the different task schedulers which exist in Hadoop, and in particular, on the Capacity Scheduler with its inner working mechanisms and its important properties. Numerous job schedulers have been designed by different research groups to optimize MapReduce task scheduler decisions in addition to the task schedulers present in Hadoop. Such job schedulers are also discussed. The remainder of this chapter contains evaluations of other research into performance modeling of MapReduce and simulation of distributed systems, including Grid and MapReduce.

Various efforts have been made in simulating the MapReduce framework. Mumak, Starfish, MRPERF etc. are a few examples. Some of these simulators are discussed in the final part of the chapter. The pros and cons and the different approaches taken by different MapReduce simulators are described.

2.1 Background

2.1.1 Hadoop Structure

Hadoop is an open source platform that supports the MapReduce parallel programming data model. Hadoop mainly consist of two components: MapReduce Engine and HDFS. These two components are briefly described below.

- The MapReduce Engine consists of a JobTracker and several TaskTrackers. The JobTracker primarily deals with scheduling the jobs' tasks on the TaskTrackers and monitoring them at regular intervals. The JobTracker also re-executes the tasks which get failed due to errors. The TaskTrackers execute the tasks as specified by the JobTracker according to some task scheduling algorithm. They also report periodically to the JobTracker when they finish executing a task and send a periodic heartbeat signal to inform the JobTracker to let it know that they are alive.

- HDFS: The Hadoop Distributed File System (HDFS) [26] provides global access to files in the cluster. The HDFS is coded in Java as a user level filesystem. The files in HDFS may be big and hence are split into smaller blocks of 64 MB. Each block is stored as a separate file in the local filesystem. HDFS is implemented by two services: the NameNode and the DataNode.

The NameNode maintains the HDFS directory tree. Like JobTracker, the NameNode is a single node in the cluster. Clients contact the NameNode to perform common filesystem operations. The NameNode keeps a mapping between HDFS file to a list of blocks in the file, and between blocks and the DataNode(s).

The Secondary NameNode task is to periodically perform checkpoints on the NameNode's persistent state. All other nodes in the cluster serve the purpose of DataNodes. Data blocks are stored in DataNode. A sample Hadoop architecture is shown in Figure 2.1. M1 and M2 are mappers. R1 and R2 are reducers. Mappers and Reducers are executed on TaskTrackers (TT). Split 1, 2 and 3 are splits of 64 MB from a HDFS input file.

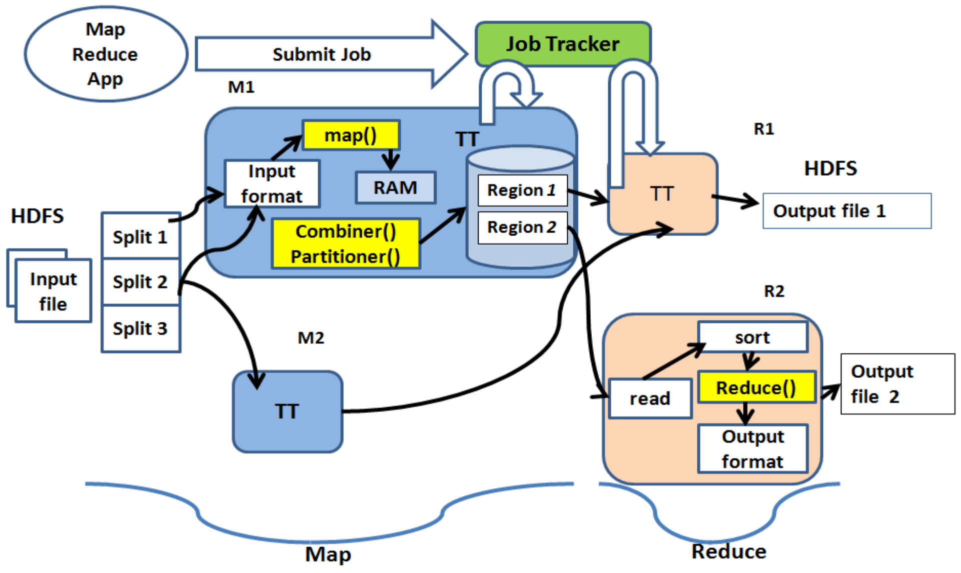


Figure 2.1: Hadoop Framework

2.1.2 Definitions

Data Locality

In the MapReduce framework, data locality is an important performance criterion. The system aims to reduce data transfer by processing distributed data as close as possible to the current storage location. Tasks have varying relationships with the data: node-local tasks have the data on the same machine's local disk, rack-local tasks have the data on a machine in the same rack and network data transfer to the local switch/router is necessary to complete the task, and, finally, remote tasks are neither node-local nor rack-local. Remote tasks incur greater latency, as the network transmission may be across multiple network links. Therefore, if a TaskTracker has to do map processing on the chunk of data which is available in its local storage, the processing will be fast. Otherwise, the TaskTracker has to fetch the required chunk of data from another node (may be on the same rack or on a different rack), that has the data over the network, increasing response time.

Off-Switch Task

An *off-switch task* is defined as a task in which data has to be fetched over the network. In the Capacity Scheduler, off-switch means remote.

Delay Scheduling in Capacity Scheduler

In Capacity Scheduler, delay scheduling is implemented for map tasks to achieve high data locality. There is one counter called *scheduling opportunities* for each job that gets incremented every time it gets the opportunity to schedule its map task. If node-local or rack-local tasks can be scheduled for the job, the *scheduling opportunities* counter is reset to zero. Initially, jobs can only schedule node-local or rack-local task. Hence, their map task scheduling is delayed if the data is not node-local to the TaskTracker from which the JobTracker gets the heartbeat to schedule the job's map tasks. Delay also occurs if no rack-local task can be found. However, the job's map task cannot be indefinitely delayed. So, after the *scheduling opportunities* counter becomes greater than the number of nodes in cluster, a remote task is scheduled for the job.

Straggler

A *Straggler* is a task (map or reduce) that takes an unusually long time to complete. This lengthens the total time for a MapReduce job to get finished. The reasons which contribute to stragglers , include static aspects such as hardware faults, heterogeneous hardware and dynamic aspects like CPU time variability, network traffic, disk contention, etc. [2].

MapReduce Slots

MapReduce Slots defines the maximum number of map and reduce tasks that can run in parallel on a cluster node. The number of slots on each cluster node can be different and depend on the node's hardware capacity.

Speculative Execution

Stragglers are a major issue in Hadoop. They cause a few cluster nodes to slow down the rest of the MapReduce application. "Hadoop does not try to diagnose and fix slow-running tasks. Instead, it tries to detect when a task is running slower than expected and launches another, equivalent, task as a backup. This is termed speculative execution of tasks. It is important to note that speculative execution does not work by launching two duplicate tasks at the same time. This would be wasteful of cluster resources. Rather, a speculative task is launched only after all the tasks for a job have been launched, and then only for tasks that have been running for some time and have failed to make as much progress, on average, as the other tasks from the job. When a task completes successfully, any duplicate tasks that are running are killed since they are no longer needed. So, if the original task completes before the speculative task, then the speculative task is killed. On the other hand, if the speculative task finishes first, then the original is killed [37]".

2.1.3 Task Schedulers in Hadoop

The task scheduler runs on the JobTracker and plays an important role in deciding where the tasks of a particular job will be executed in the cluster. There are three different well-known schedulers in Hadoop:

FIFO

Initially Hadoop was intended such as web indexing and log mining. All the users submit jobs to a single queue, and the FIFO task scheduler runs them sequentially in first in first out order. Jobs are executed as they arrive in the queue (as long as they have the same priority). However, FIFO does not strictly mean that the next job will wait until the current job is done. In cases where the current job does not completely consume the cluster capacity, the next job can run using the unused capacity of the cluster. FIFO jobs may not finish in the order they arrive. If there are multiple slots, and a small job arrives, it may get a slot or two and finish before a large job.

Fair Scheduler

Facebook's need to share its data warehouse between multiple users lead to the development of Fair Scheduler. Facebook initially used Hadoop to generate daily reports from the data it accumulated daily. Sooner other groups within Facebook started to use Hadoop as well and this increased the number of production jobs. In addition, analysts started using the data warehouse for short interactive jobs like Hive queries. Hive [31] is a data warehouse system for Hadoop that helps to analyse the large datasets stored in Hadoop-compatible file systems. Hive uses HiveQL to query the data. Hence, Facebook built the Fair Scheduler. Fair Scheduler allocates resources evenly between multiple jobs and also supports capacity guarantees for the different jobs. The Fair Scheduler is based on three concepts:

1. Jobs are placed into *pools* according to certain attribute like user name or group.

2. Each pool have a *guaranteed capacity*. Guaranteed capacity provides a minimum number of map slots and reduce slots to be given to the pool.

3. Excess capacity that is not used to satisfy the pool minimum is allocated between jobs using fair sharing. Fair Share is part of the Hadoop open source code.

Capacity Scheduler

The Capacity Scheduler from Yahoo! is similar to the Fair Scheduler. In Capacity Scheduler, a number of *named queues* are configured. Each queue has a guaranteed capacity in the form of map and reduce slots. The capacity is given to the queue when it contains jobs. Otherwise, any unused capacity is shared between the queues. FIFO scheduling with priorities is used in each queue. Limits can be placed on the percent of running tasks per user, so that users share a cluster equally. Both Capacity and Fair Share schedulers offer various configuration parameters, that allow administrators to tune scheduling parameters for the jobs. Table 2.1 shows the configurable parameters for the Capacity Scheduler. There are two types of parameters in the configuration settings:

1. **Resource allocation**: These parameters relate to the number of queues, their allocated capacity, Maximum-Capacity and per user limit within the queues. The first five parameters listed in Table 2.1 belong to this category.

2. **Job initialization**: These parameters are related to the number of jobs which can be in the system at a time, queue limit and the user limit on the number of tasks which can be spawned concurrently, etc. The last six parameters listed in the Table 2.1 belong to this category.

There are 2 other major parameters which affect the working of Capacity Scheduler, but these parameters are hidden in the code and not exposed to the users/administrators in the configuration file. These parameters are the following:

1. **maximum-tasks-per-heartbeat**: It specifies the maximum number of tasks which can be scheduled in a heartbeat. Typically, one map and one reduce task can be assigned per heartbeat by the JobTracker to the TaskTracker who sends the heartbeat, but with this parameter, multiple tasks of the same type can be assigned to the TaskTracker per heartbeat. One reason to not expose this parameter is that the number of tasks a TaskTracker can handle is defined by its hardware configuration. So, in certain cases, a user may give this parameter a huge value when the actual configuration on the TaskTracker cannot really support it. For example, if this parameter was exposed to the user, they could potentially set the value of maximum-tasks-per-heartbeat to 4 when a TaskTracker is configured to support only 2 map slots. In such cases, the values specified by the user could not be handled by the TaskTracker and will have no effect.

2. **MaxTasksToAssignAfterOffSwitch**: It defines the maximum number of tasks to schedule, per heartbeat, after an off-switch task has been assigned. Clearly, the more the off-switch tasks can be assigned, the more latency will be added to the MapReduce application execution time. One possible reason for not exposing this parameter to the user is that it is not well known how data locality affects the MapReduce applications performance under Capacity Scheduler. Setting this parameter to a large

Table 2.1: Configurable Parameters For Capacity Scheduler

Parameter Name	**Brief Description and Use**	**Default Value**
queue.*queue-name*.capacity	Percentage of the number of slots in the cluster available for jobs in this queue.	1 queue with 100%
queue.*queue-name*.Maximum-Capacity	Limit beyond which a queue cannot use the capacity of the cluster. This provides a means to limit how much excess capacity a queue can use. -1 means no limits are imposed on the maximum-capacity.	-1
queue.*queue-name*.Minimum-User-Limit-Percent	Each queue enforces a limit on the percentage of resources allocated to a user at any given time, if there is competition for them.	100
queue.*queue-name*.User-Limit-Factor	Allows user to acquire more slots than the queue's configured capacity if the cluster is idle.	1
queue.*queue-name*.Supports-Priority	Is priority considered in scheduling decisions.	False
Maximum-System-Jobs	Maximum number of concurrently initialized jobs	3000
queue.*queue-name*.Maximum-Initialized-Active-Tasks	Maximum number of concurrently initialized tasks in the queue for all the jobs.	200000
queue.*queue-name*.Maximum-Initialized-Active-Tasks-Per-User	Maximum number of concurrently initialized tasks per user in the queue.	100000
queue.*queue-name*.init-accept-jobs-factor	Multiple of *(Maximum-System-Jobs*queue-capacity)*. Determines the number of jobs accepted by the scheduler.	10
init-poll-interval	Time interval between job initializations.	5000 ms
init-worker-threads	Number of worker threads used to initialize jobs.	5

value means more remote tasks can be scheduled for a job, which will increase the execution time of the job.

The Capacity Scheduler works according to the following principles:

1. The existing configuration is read from `capacity-scheduer.xml` at cluster startup. It contains all the Task scheduler settings. The queues and other parameters are set using this information.

2. An initialization poller thread is started and worker threads are also initialized. The poller thread wakes up at specified intervals ("init-poll-interval"), distributes the work to worker threads and then goes to sleep. The number of worker threads is set up as `min(number_of_queues, init-worker-threads)`. A single thread can handle multiple queues. Not all the jobs admitted into the system are initialized instantly to reduce the memory footprints on JobTracker.

3. When a job is submitted to the cluster, the scheduler checks for job submission limits to check whether the job can be accepted for the queue and user.

4. If the job can be accepted, the initialization poller checks the job against initialization limits (e.g. "maximum-initialized-active-tasks"). If the job can be initialized, it is submitted to the assigned worker thread for the queue, which initializes the job.

5. Whenever the JobTracker gets the heartbeat from a TaskTracker, a queue is selected from all the queues whose jobs can be executed. A queue is selected by sorting the queues according to number of running tasks/capacity of the queue. Queue- and user- specific limits are checked to see if they are under appropriate limits (e.g. Maximum-Capacity). After selecting the queue, the first job is chosen from the queue unless its user is over the user limit. Next, a task is picked up from the job and the preference is given to node-local tasks over rack-local task s. This procedure is repeated until all the jobs complete execution.

2.2 Alternative Task Schedulers for MapReduce

In order to improve the performance of the jobs in MapReduce, different types of task schedulers were designed by different research groups. Some of the most prominent ones are discussed in this section.

2.2.1 Delay Scheduling

The Delay scheduler [38] has been designed to optimize the Fair share scheduler algorithm. The Fair share algorithm works effectively when it comes to fairness in sharing the resources amongst users. However, a strict implementation of fair sharing compromises locality, because the job to be scheduled next according to fairness might not have its data on the nodes that are currently free. Two locality problems that arise with naive fair sharing are the following: *head-of-line scheduling* and *sticky slots.* Head of Line scheduling refers to an issue where a job's task is launched on the next slot that has become free without considering job's data locality. The issue of sticky slots refers to the problem where there is a tendency for a job to be assigned the same slot repeatedly. In both cases, to maintain fairness, the scheduler is forced to launch a task from a job without local data on a node. These two issues in naive fair share scheduling algorithm influence throughput and response time for MapReduce jobs.

This problem of data locality is attacked by a simple technique called delay scheduling. In delay scheduling, the head-of-line job is not allowed to launch a task if it is not node local. The job is skipped and an attempt is made to search a node local task in the subsequent jobs. However, to avoid starvation, a job is allowed to launch a non-local task if that job has been skipped for a long time. The wait time to launch a non-local task is based on the rate at which slots free up in the cluster and the desired level of locality.

2.2.2 Quincy

Quincy [21] is another approach to accomplish data locality and fairness with multiple jobs on the Dryad clusters at Microsoft. It adopts flow-based scheduling and is a graph-based framework for cluster scheduling. The primary data structure used in this approach is a graph that encodes both the structure of the cluster's network and the set of waiting tasks along with their locality metadata. Appropriate weights and capacities are assigned to the edges in the graph to obtain, a declarative description of the scheduling policy. After this, Min-cost flow [4] is used as a standard solver to convert this declarative policy to a set of scheduling assignments. The scheduler updates the graph whenever an important event occurs (a task completes, a new task is submitted, etc.), and at regular timer events. Whenever the graph is updated, The scheduler computes a new min-cost flow on updation of the graph. The scheduler uses new flow to start or kill tasks.

2.2.3 Dynamic Priority Scheduler

The Dynamic Priority Scheduler [25] was designed by HP Labs. As seen with the Fair Share and the Capacity Scheduler, the administrator has to configure the parameters manually and start the cluster. Different organizations have to negotiate with other organizations or administrators if they need more resources. Sometimes this may be inconvenient (as it has to be done manually) and may not be possible. If a user's individual jobs vary in importance (critical if certain deadlines must be met) over time and between job types, severe allocation inefficiencies may occur. For example, a user with a high allocated capacity may run large, low- priority test jobs starving more important jobs from other users.

To solve this issue, the concept of dynamic regulated priorities was introduced in the Dynamic Priority Scheduler. In this scheme, each user is given a certain quota called *budget* over all the jobs at all times. However, this *budget* is not static, meaning that *budget* is not mapped to a particular capacity share value. Instead, users are allowed to specify how much of their *budget* they are willing to spend on each job at any given point in time, called the spending rate. Spending rates denote the willingness to pay a certain amount of the *budget* per map or reduce task slot per time unit. The time unit is configurable, and is referred to as an *allocation interval.* The share allocated to a specific user is calculated as their spending rate over the spending rates of all users. In each *allocation interval*, the scheduler performs the following operations:

- aggregates all spending rates s from all current users to calculate the cluster price, p.
- allocates $(s_i/p) \times c$ task slots (both mappers and reducers) to user i, where s_i, is the spending rate of user i, and c is the aggregate slot capacity of the cluster. This is done for all the users.
- deducts $s_i \times u_i$ from *budget* b, where u_i, is the number of slots used by user i. This is done for all the users.

The scheduler allows task preemption to guarantee the shares. However, at the same time, users can exceed their share if no other tasks are running. Furthermore, the users are only charged for the fraction of the shares they actually use. So if they don't run any jobs, their spending goes to 0. A single user is assigned a single queue due to security issues. However, queues can be shared between different users.

2.2.4 Other Research Efforts

There are numerous other attempts to create new scheduling algorithms in MapReduce. In ARIA (Automatic Resource Inference and Allocation) [33], a job can be submitted with a desirable job completion deadline. The scheduler then estimates and allocates the appropriate number of map and reduce slots to the job so that it meets the required deadline. The LATE (Longest Approximate Time to End) [39] scheduler has been developed to handle MapReduce jobs in the heterogeneous environment present in clouds. It was made to improve the speculative execution task scheduling in Hadoop, which performs poorly in a heterogeneous environment. The LATE algorithm uses a SlowTaskThreshold parameter to prevent unnecessary speculation. The LATE algorithm ranks tasks by estimated time remaining and starts a copy of the highest ranked task that has a progress rate lower than the SlowTaskThreshold. The advantage of the LATE algorithm is robustness to node heterogeneity, since only some of the slowest (not all) speculative tasks are restarted. Chao *et al.* [32] designed a new triple-queue scheduler which consist of a workload predict mechanism MR-Predict and three different queues (CPU-bound queue, I/O-bound queue and wait queue). They classified MapReduce workloads into three types, and their workload prediction mechanism automatically predicts the class of a new incoming job based on this classification. On the basis of the classification, jobs are assigned to the CPU-bound or I/O-bound queue.

2.3 MapReduce Performance Modeling

Krevat *et al.* [23] developed an optimistic performance model that predicts the idealized lower-bound runtime of a MapReduce workload. The model assumes that the data distribution is even, that data is perfectly pipelined through sequential operations, and that the underlying I/O resources are utilized at their full bandwidths. The model's input parameters describe basic characteristics of the job (e.g., amount of input data), the hardware (e.g., per-node disk and network throughput), and the framework configuration (e.g., replication factor). The goal of the model is not to accurately predict the runtime of a job on any given system, but to indicate what the runtime theoretically should be. The output is the idealized runtime. Evaluation shows that MapReduce implementations from both Google and Hadoop are far from optimal values estimated using the model. They also developed a minimal framework to run the applications to prove that the estimates are indeed achievable. The model works for one job rather than for a workload of jobs. The sources of inefficiency were straggler tasks, which arise due to disk-to-disk variability and network slowdown effects. Song [27] describes a model for MapReduce applications using queuing theory. The workload considered is homogeneous, with many instances of the same job, and the model focuses on predicting waiting time for map and reduce tasks. Another model [19] used by Starfish [20] divides tasks into stages and models each stage with a different model. It is further discussed in Section 2.5.1.

2.4 Simulation of Distributed Systems

A closely related large-scale distributed computing paradigm to MapReduce is Grid computing. Grid computing is well-known and has been used to solve large-scale problems using distributed resources. It addresses similar issues as MapReduce, but with a wider scope. In Grid computing a large computer network is formed by interconnecting lots of computer systems such that their resources are shared. A variety of simulators have been developed to model and simulate the performance of Grid systems, including Bricks [29], Microgrid [28], Simgrid [7], and GridSim [6].

Bricks [29] is a performance evaluation system based on simulation that allows analysis and comparison of various global computing systems under reproducible, and controlled environments. Its main focus was to evaluate different scheduling algorithms and schemes in global computing systems by simulating the network behaviour and resource scheduling algorithms. Bricks uses a queuing network model in which servers and networks are modeled as queuing systems. Bricks allows easy construction of a specific global computing system configuration because the users can specify network topologies, server machine architectures, communication models and scheduling framework components using the Bricks script.

MicroGrid [28] was the first notable effort to simulate and model large scale Grid structures. This project was conducted at the University of California at San Diego and was modeled after Globus [15]. MicroGrid creates a controlled virtual grid emulated environment. The main focus of MicroGrid was to allow researchers to run actual Grid applications on virtual Grid resources. The emulation was precise in producing results. However, the emulation of realistic applications was time consuming as applications run on emulated resources. Also, scheduling algorithms designers generally work with application models instead of constructing actual applications.

The Simgrid toolkit [7], was developed at the University of California at San Diego. It is a C language based toolkit for the application scheduling simulation, which can be used in large scale distributed systems like Grid. It supports modeling of resources that are time-shared and the load can be injected as constants or from real traces. Simgrid is a powerful event driven simulator that allows creation of tasks in terms of their execution time and resources with respect to a standard machine capability. Simgrid has been used for a number of real studies, and demonstrates the power of simulation. However, Simgrid is limited in scope due to its applicability to single scheduling entity and time-shared systems only. It is difficult to simulate multiple competing users, applications, and schedulers, each with their own policies in the toolkit.

GridSim [6] is Java-based toolkit for modelling and the simulation of computational resources for design and evaluation of schedulers and scheduling algorithms for network based high-performance clusters and Grid computing. It provides a comprehensive facility for the simulation of different classes of heterogeneous resources, users, applications, resource brokers, and schedulers. The GridSim toolkit resource modeling facilities are used to simulate the worldwide Grid resources managed as time- or space-shared scheduling policies.

2.5 Simulators for MapReduce

According to Babu [3] there exists one big problem with MapReduce: the large number of configuration parameters. The user has to configure these parameters manually to get optimized performance for their jobs. To run even a single job in MapReduce, the user has to set the parameters or rely on default settings, which may not be optimal. It has been proven that job configuration parameter settings can have a significant impact on the performance of the job. This issue makes the easy adoption of MapReduce difficult. For example, if the settings have to be done manually, then soon the need for special system administrators will arise, as happened for databases [8].

2.5.1 Starfish

Starfish [20] aimed to solve the problem of managing huge number of configuration parameters in the MapReduce framework by doing cost-based optimization of MapReduce programs. Starfish involves three components:

- **Profiling**: The Profiler is responsible for collecting job profiles. A job profile consists of the dataflow and cost estimates for a MapReduce job $j = (p, d, r, c)$, where p is a MapReduce program which runs on input data d on a set of cluster resources r using job configuration parameters c. The dataflow estimates represent information regarding flow of the data, while cost estimates represent resource usage and execution time. BtTrace[1] is used to collect profiles. The profiler can be turned off or on as demanded.

[1]http:://kenai.com/projects/btrace

- **What-If Engine**: This component predicts the performance statistics for any job which was executed previously and profiled but is now being run under different settings. The difference in settings can be in terms of input data, cluster configuration or job configuration parameters. The beauty of the What-if Engine lies in the fact that the job does not have to be executed on a cluster. Using its previous profile, a new virtual profile is created for this new job and white box models [19] are used to estimate new cost and dataflow related information. This is then fed into the simulator to get all run time performance metrics for the job. Dataflow proportionality and the cluster node homogeneity assumptions allow the new virtual profile to inherit the dataflow and cost statistics associated with the previous job profile.

- **Cost Based Optimization**: Here the optimal settings of different MapReduce configuration parameters is searched for a job. As searching a large parameter space is difficult, the approach taken here is of divide and conquer. The large search space is first divided into lower dimensional subspaces, and then each smaller subspace is solved independently to get approximately globally optimal values. The authors borrow techniques from simulated annealing [22] to solve this problem.

Starfish is one of a first kinds of tools which try to do self-optimization for MapReduce programs. However, currently it only supports FIFO based scheduling. So, the simulator used in Starfish can simulate only one running job at a time. Moreover, the changes in cluster settings (to see performance from the What-if Engine for the same job under different cluster settings) are only allowed in terms of adding and deleting the nodes of the same type.

2.5.2 MRPERF

MRPERF [35] aims to provide a fine-grained simulation of MapReduce setups throughout different phases. MRPERF is based on the popular ns-2 network simulator [5]. It models inter- and intra-rack task communications over the networks to simulate correct network behaviour. Some of the important motivations behind MRPERF were to find the performance of jobs running times under varying cluster configurations, different network topologies, different data placement algorithms and different task schedulers in MapReduce. FIFO, Fair Share and Quincy schedulers are present in MRPERF. For map/reduce task modeling, MRPERF creates a number of simulated nodes, where each node might have several processors and a single disk. There is a simplifying assumption about the application behaviour: a job has simple map and reduce tasks with compute time requirements that are proportional to the data size but not to the content of the data.

2.5.3 Mumak

Yahoo! has developed a discrete event simulator named Mumak,[2] which can estimate performance of MapReduce applications under different schedulers. It takes a job trace derived from production workload and a

[2]https://issues.apache.org/jira/browse/MAPREDUCE-728/

cluster definition as input, and simulates the execution of the jobs as defined in the trace in the virtual cluster. The detailed job execution trace (recorded in relation to virtual simulated time) is the output, which can be analyzed to understand various traits of individual schedulers (turn around time, throughput, fairness, capacity guarantee, etc.).

Mumak consists of the following components:

- **Discrete Event Simulator Engine with an event-queue**: It manages all the discrete events in virtual time and fires the appropriate handlers when the events occur.

- **Simulated JobTracker**: The JobTracker takes responsibility of simulating the MapReduce Scheduler.

- **Simulated Cluster (set of TaskTrackers)**: The simulated cluster consists of an appropriate number of simulated TaskTrackers, which respond to events generated by the simulation engine.

- **A Client for handling job-submission**: The client responds to job related events sent by the Engine and submits the appropriate jobs to the JobTracker.

The simulator is effective for analyzing different task scheduler behaviours. At the same time, it has some serious limitations. Mumak needs input traces from real production clusters to work upon and cannot predict the performance for jobs when the cluster configuration is changed. It also does not simulate the shuffle phase, which hinders its accuracy. These limitations restricts the applicability of this tool.

2.5.4 SimMR

SimMR [34] is yet another simulator which has been designed by HP Labs to evaluate different schedulers and different provisioning strategies. SimMR is comprised of three inter-related components:

- A Trace Generator which creates a replayable MapReduce workload. It can replay original JobTracker logs or a synthetic workload.

- A Simulator Engine which emulates the JobTracker functionality based on discrete event simulation. It manages all the discrete events in simulated time and performs the appropriate action on each event. The simulator engine fires events and runs the corresponding event handlers.

- A pluggable scheduling policy that controls the decisions on job ordering and resources allocation to different jobs over time.

SimMR focuses on simulating the JobTracker decisions and task/slot allocations among different jobs. It does not simulate the details of Tasktrackers like their hard disks or network packet transfers. It is not open source and hence cannot be extended by the research community.

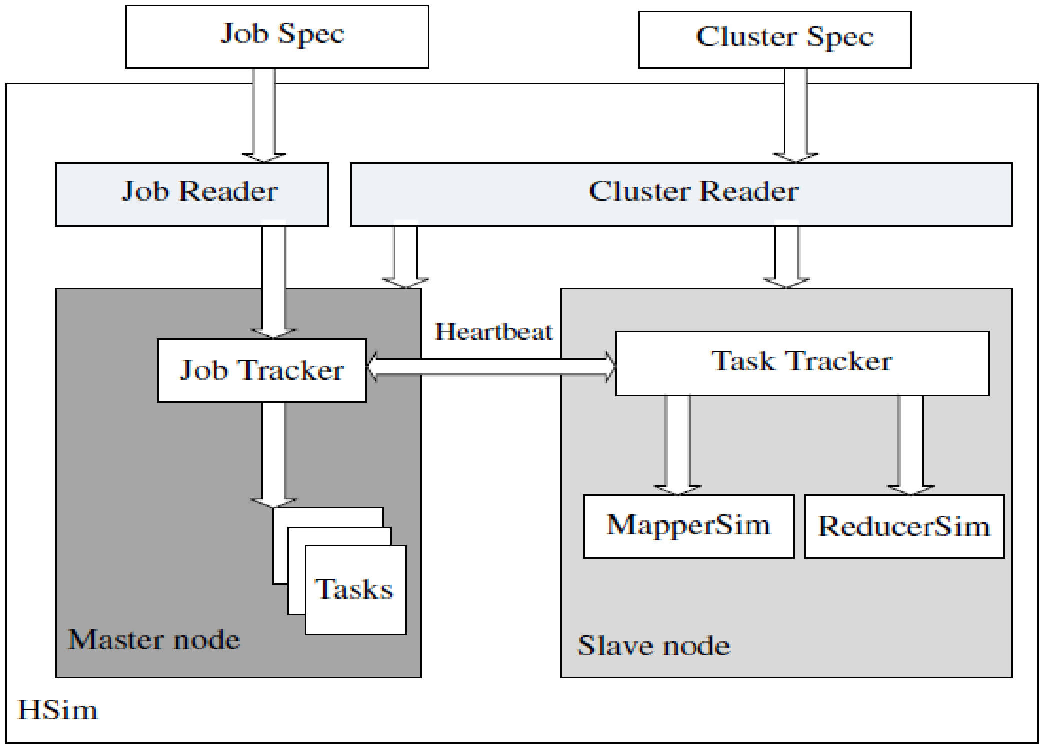


Figure 2.2: HSIM Architecture [24]

2.5.5 HSim

Hsim [24] is another simulator for MapReduce applications. It can simulate the dynamic behaviours of Hadoop environments and model a large number of Hadoop parameters such as Node parameters, which are related to processors, memory, hard disk, network interface, map and reduce instances, Cluster parameters, which include the number of nodes, node configurations, network routers, job queues and schedulers, and Hadoop system parameters. Figure 2.2 shows the architecture for HSIM.

To perform a simulation, the Cluster Reader component reads the cluster parameters from the Cluster Spec to create a simulated Hadoop cluster environment. A specified number of nodes are initialized and arranged using a certain type of topology. When the simulated cluster is ready, incoming jobs are retrieved from the job queue using a job scheduler. The Job Spec is processed by the Job Reader component and jobs are submitted to HSim for simulation. HSim follows a masterslave model. The simulated map instances (MapperSim), reduce instances (ReducerSim), JobTracker and TaskTrackers are located on Master and Slave nodes. The Master node performs the role of both NameNode and JobTracker. The Slave nodes serve as Datanodes and TaskTrackers. On Slave nodes, map instances and reduce instances perform data processing tasks.

As with previous work in the area, HSIM does not have a Capacity Scheduler component. There is also no support for load balancing. HSIM capabilities of supporting fair share scheduling, running multiple jobs in a cluster and supporting heterogeneous nodes in a cluster setup are debatable as the work lacked support in terms of experiments done.

2.5.6 MRSim

MRSim [17] is a MapReduce simulator based on discrete event simulation. The aim of MRSim is to measure scalability of MapReduce based applications easily and quickly and to capture the effects of different configurations of Hadoop setup on MapReduce based applications behaviour in terms of job completion times and hardware utilization. MRSim models and simulates network topology and traffic using GridSim. On the other hand, it models the rest of the system entities using SimJava discrete event engine. Figure 2.3 shows the architecture for MRSim.

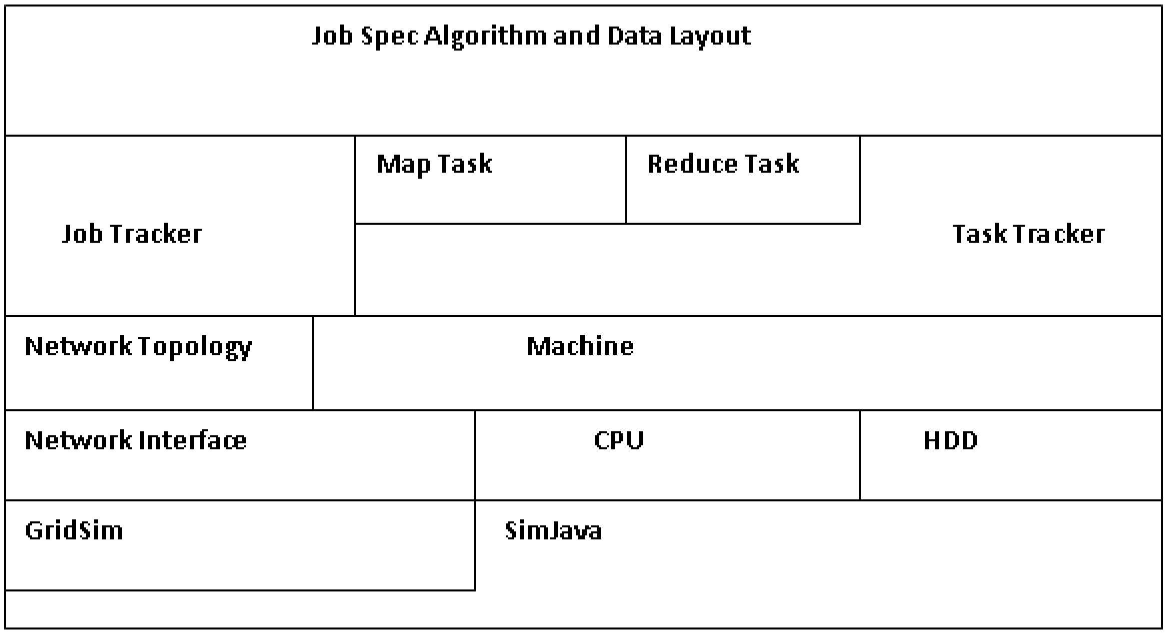


Figure 2.3: MRSim Architecture [17]

The system is designed using object-oriented models. CPU, Disk and Network Interface models were designed to be the basic blocks which can be grouped in a PC *machine entity.* Each machine is a part of network topology model. Each machine host a JobTracker process and a Task Tracker Process. The main component of the simulator is JobTracker that controls generating map and reduce tasks, monitors when different phases complete, and produces the final results. The user input needed by MRSim is divided into two different parts defined in text files of JSON format: cluster topology file and job specification file. The Topology file consists of several rack parts. Each rack consists of a group of machines linked with one router. The Job specifications file is comprised of a number of map and reduce tasks, data layout, algorithm description and the job configuration parameters. The data layout describes the location and the replications of the data splits on the simulated nodes. The input data for a map task is divided into 64 MB partitions and stored on different nodes. An algorithm description provides information about the MapReduce application, such as the number of CPU instructions per record, average record size in MapReduce tasks etc. The description helps the simulator to calculate how much time each task in the simulator would take.

MRSim is developed by the same people who developed HSIM. MRSim lacked capabilities to include new schedulers. There is no load capability. It only support one job in a system at a time and, similar to HSIM, it drew inspiration from MRPERF.

2.5.7 SimMapReduce

SimMapReduce [30] is a MapReduce simulator based on GridSim. It allows researchers to evaluate different scheduling algorithms and resource allocation policies. SimMapReduce has a multi-layer architecture as shown in Figure 2.4. SimMapReduce make use of existing packages as separate components. This makes the codes reusable, which save time and energy. SimMapReduce works on discrete events. At the lowest layer, SimJava serves to process the discrete events. A chain of events between different entities keep simulation going. Gridsim supports the basic provision of system components, such as grid resource, broker, gridlet, workload trace, networks and simulation calendar. SimMapReduce layer models the different entities of MapReduce. MRNode represents a MapReduce node in the cluster. MRBroker takes care of allocating nodes to the arriving users. After the user get nodes to execute their jobs, the job scheduler dispatches map/reduce tasks for user jobs to a specific node and monitors their execution. Each job has one corresponding MRMaster. FileManager is responsible for handling file activities. On the top layer, the description of system parameters, network topology and scheduling algorithms are described in XML files.

At the start of the simulation, nodes in the cluster report their characteristics to MRBroker. Every user has their own job sequences, and sends jobs sequentially depending on the arrival rate to the system. Each job has MapTask, ReduceTask, and a master, acting on behalf of the job. When a job starts, its master asks for the needed number of nodes from MRBroker. When MRBroker finds the requested number of nodes for the job, it allocates them to the job's MRMaster. MRMaster picks idle nodes on which to schedule job MapTasks. MapTask is executed on the scheduled node. When all MapTasks are finished for the job, MRMaster groups all key/value pairs and sends data with the same key to one ReduceTask. The reduce phase is carried out in ReduceTask and results are written to a final output file. MRMaster reports job completion to the user and gets destroyed when all the MApTasks and ReduceTasks are completed for the job. The simulation is finished when there are no more jobs present in the system.

This simulator supports multiple jobs but supports the FIFO scheduler only. There was no support to tune multiple Hadoop parameters. It is a very basic simulator which can run and execute incoming jobs in a FIFO manner on a simulated cluster setup.

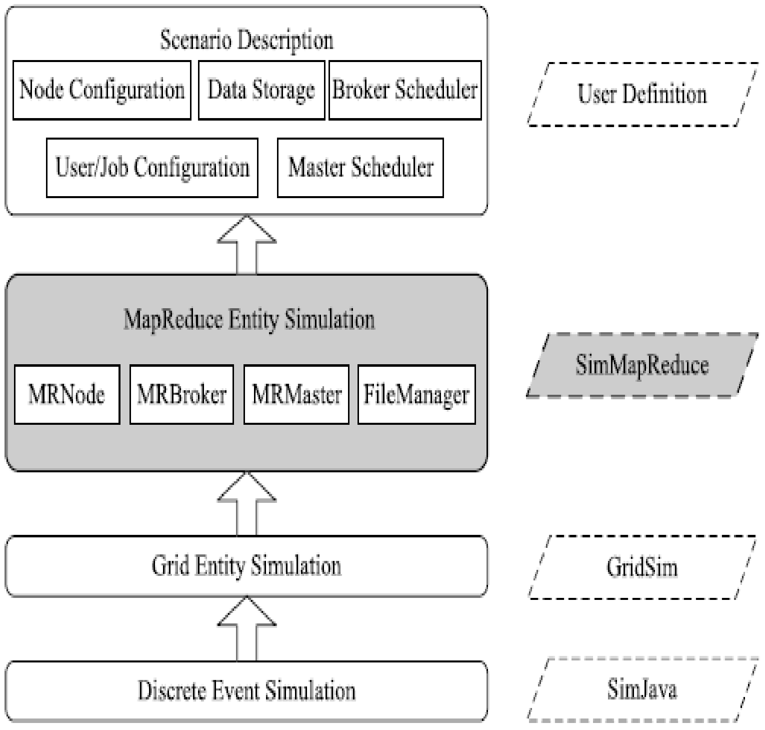


Figure 2.4: SimMapReduce Architecture [30]

2.6 Summary

Simulation has been used widely for performance prediction and characterization in different areas of computer science. MapReduce has been established as a preferred choice of framework for data intensive computing. Hence, a lot of different simulator tools have been developed over the last few years to model MapReduce. MRPERF, Starfish etc. are some examples. These simulator tools save time and help in performance prediction. However, one of the major drawbacks of all these simulators (except Mumak) is that they all support only the FIFO scheduler. By and large, the schedulers designed to support sharing of cluster resources among multiple users and organizations have been ignored in existing simulation tools for MapReduce, with Mumak being an exception. Mumak also has severe limitations like inaccurate performance modeling and the need for production level traces. One other important fact is that, despite the presence of the Capacity Scheduler for a long time, its behaviour has not been studied according to my knowledge.

Chapter 3
Design and Implementation of Capacity Scheduler simulator

MRPERF is an existing MapReduce simulator that is based on discrete event simulation. The first major work conducted in this thesis is to integrate the Capacity Scheduler in the MRPERF simulator tool. Therefore, in this chapter, the detailed design of MRPERF and the reasons to choose MRPERF are presented. Finally, the modifications for integrating the Capacity Scheduler into MRPERF are described in detail in the rest of the chapter.

3.1 MRPERF Design

MRPERF [36] is the earliest simulator tool for the MapReduce data processing paradigm. It was developed at Virginia Tech to capture various performance aspects of MapReduce setup and to then use this information to predict application performance. MRPERF was made open source to be used by the research community to enable it to explore design issues, validate new algorithms and optimization in MapReduce. Studying different aspects of MapReduce is very difficult due to the high cost in setting up a real cluster. Not everyone can afford speculative cluster implementations and hence a simulator of MapReduce can be of great use. For example, if a company is thinking of increasing the cluster size, it would be beneficial to see how much performance benefits they are going to obtain using a simulator.

MRPERF was designed to incorporate/study the following features in MapReduce:

- **Cluster configuration**: This includes exploration of performance when new nodes are added or deleted from the cluster, and node characteristics are changed in terms of cpu, and disks.

- **Network topologies**: This includes exploration of performance of MapReduce with different network topologies, (eg. tree, star, etc.)

- **Task Scheduling Algorithm**: Researchers can plug in a new algorithm for task scheduling and check resulting performance on MapReduce jobs. This is the aspect of MRPERF which is being leveraged to do the work in this thesis.

- **Data placement Algorithms**: Researchers can test new data placement algorithms that decide where to place data blocks in the cluster to enhance the performance of MapReduce applications.

3.1.1 Architecture of MRPERF

Figure 3.1 shows the high-level architecture of MRPERF. The user input needed by MRPERF can be classified into three parts: cluster topology specification, application job characteristics, and the data layout of the application input data. XML file is used to specify the topology of the cluster. The XML file is translated by MRPERF into a TCL file to be used by ns-2. The cluster topology consists of processor configuration (type and speed), memory configuration, disk bandwidth, number of DataNodes, racks and network topology. Specifying the processor type provides the name of the processor like Intel or ARM and its speed determines how much processing time an event would require to finish during the simulation. Each job to be executed in MRPERF has a configuration file that specifies different parameters such as Cycles per byte, filter ratio, etc., that are used during the map and reduce phase of that specific job. The fixed overheads, like connection setup times, are captured by measuring the overhead and using it in the simulator. Finally, the data layout deals with the NameNode's location, the location of data blocks on the simulated DataNodes, and replication factor, etc. All the different input files are read by their respective reader's modules. The ns-2 driver module simulates the network. Similarly, the disk simulator module models disk I/O. MapReduce Heuristics module (MRH) simulates Hadoops behaviour and all the other modules interacts with MRH.

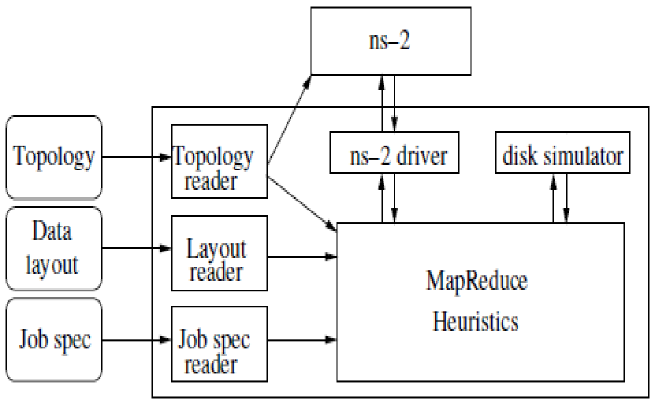


Figure 3.1: MRPERF Architecture. [36]

3.1.2 Working Mechanism in MRPERF

The main component of the simulator is a JobTracker. The JobTracker creates map and reduce tasks, lookout for different phase completions, and produces the final results. Figure 3.2 shows the control flow diagram for the JobTracker. MRPERF uses a heartbeat trigger to initiate JobTracker activities. This value is a parameter and can be set in the simulator. The default value is 300 milliseconds for small clusters. The heartbeat signals are sent every 300 milliseconds by the TaskTracker to tell the JobTracker that it has idle slots.

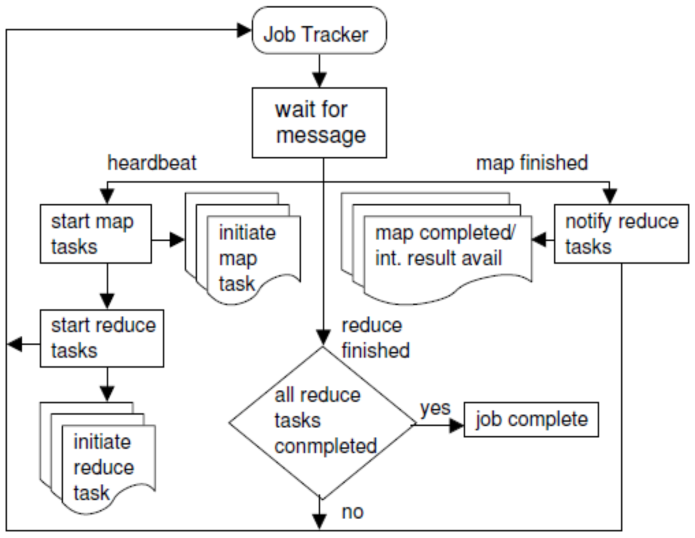


Figure 3.2: Control Flow In The JobTracker. [36]

“When the simulator receives a message to start a map task from the JobTracker, the following sequence of events happens, as shown in Figure 3.3 [35].

- A Java VM is started for the map task.
- Data access on a node is simulated through a separate process called the Data Manager. The main job of the Data Manager is to read data (input or intermediate) from the simulated local disk in response to a data request, and send the requested items back to the requester.
- Application-specific map, sort, and spill operations are simulated on the input data until all data has been consumed.
- A merge operation, if necessary, is simulated on the output data.
- A map task finished message is sent to the JobTracker to indicate completion of the map Task. The process then waits for the next assignment from the JobTracker.

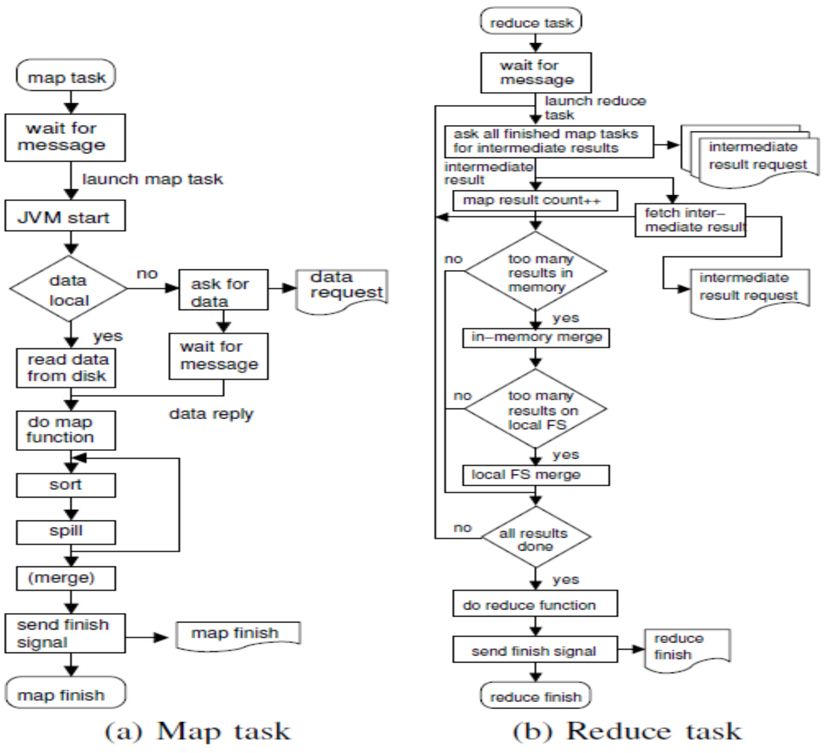


Figure 3.3: Control Flow When Running Map and Reduce Tasks. [36]

When simulator receives a message to start a reduce task from the JobTracker, the following sequence of events happens, as shown in Figure 3.3.

- A message is sent to all the corresponding map tasks to request intermediate data.
- Intermediate data is processed as it is received from the various map tasks. If the amount of data exceeds a pre-specified threshold, an in-memory or local file system merge is performed on the data. These two steps are repeated until all the associated map tasks finish, and the intermediate data has been received by the reduce task.
- The application-specific reduce function is performed on the combined intermediate data.
- A reduce task finished message is send to the JobTracker to indicate completion of the reduce task. The process then waits for the next assignment from the JobTracker."

3.1.3 Assumptions in MRPERF

MRPERF has some underlying assumptions to simplify its design. A job is restricted to having simple map and reduce tasks, and that the computing requirements are dependent on the size, and not content of the data. There is a single storage device per node. The job output data is proportional to the input data. The tasks assigned concurrently to use the resources of a node get equal share. MRPERF does not model

OS-level asynchronous prefetching. All the assumptions mentioned simplifies the design of MRPERF but pose some limitations. MRPERF cannot be used to run jobs which are complex and whose job output data is not proportional to the input size of data. The assumption to give equal share to all the tasks for a shared resource and absence of prefetching leads to prediction inaccuracy. The MRPERF limitation to only model a single disk per node does not allow it to work for RAID systems.

3.2 Reasons for Choice of MRPERF

There were various reasons to choose MRPERF as the framework to work upon in this thesis. It is open source and was made public by the developers to facilitate research. The code can be changed and tailored to meet specific research needs. MRPERF does not need actual logs from a production cluster. This is a huge advantage because getting the production cluster traces is not easy for everyone and companies often do not make their traces public. MRPERF supports synthetic jobs and also provides a mechanism to plug in a scheduler and see how it performs.

The main motivation of this work is to analyze the behavioural characteristics of job execution using the Capacity Scheduler – a task which is facilitated by MRPERF. MRPERF provides an appropriate framework to test the Capacity Scheduler under different cluster configurations, network topologies, job submission patterns, and data placement strategies.

3.3 Integration of the Capacity Scheduler into MRPERF

This section explains how the Capacity Scheduler is integrated into MRPERF, which constitutes the main implementation work of this thesis. This includes 1) setting up queues, 2) setting up the initialization poller 3) launching the jobs 4) job initialization, 5) logic for the main scheduler driver and 6) removing the jobs from the system. The ns-2 interfaces with TCL and hence each configuration file has to be converted into a TCL file to be used in simulator.

Integrating the Capacity Scheduler into MRPERF involves a lot of tasks. Capacity Scheduler is an already existing scheduler and is open source. The code consists of 4000 lines in 6 files and is written in Java. Two components – speculative execution and resource based scheduling parameter settings – were not to be included in the integration. Careful analysis of the Capacity Scheduler code and efforts to understand the exact requirements was undertaken. Speculative execution was left out because MRPERF does not support the modeling of task stragglers. Resource based scheduling parameters only work for Linux. Thus, these parameters were not incorporated in the simulator.

The integration activity also involves understanding how MRPERF works and its code. MRPERF consists of 4000 lines of code in 3 different languages: Python, TCL and C++. The front end consists of Python, XML and TCL files. The configuration related to cluster and jobs is contained in the XML files. A file named `capsim.xml` was created. This file contains the parameter settings for the Capacity Scheduler. The

XML files are translated to TCL files by a python program. As ns-2 interfaces with TCL, such conversions are necessary. Changes were undertaken on existing python files to convert newly added `capsim.xml` to `capsim.tcl`. Changes were made in existing TCL files to support new job types and multiple users. The existing main logic in MRPERF is contained in the file named `hsim.tcl` which interfaces with the scheduler code (present in the back end). The various stages of MapReduce processing are coded as TCL files in the front end and `hsim.tcl` also interfaces with them. Hence, code was added in the `hsim.tcl` file to interface with newly added Capacity Scheduler code. Code was also changed in `hsim.tcl` to process multiple heartbeats for reduce tasks.

One of the major issues with the MRPERF simulator was the lack of variability in simulation runs. A fixed set of jobs on a cluster setup with unchanged Capacity Scheduler parameter settings always produced the same results. The reason for this behaviour was that the heartbeat arrival time at the JobTracker in every simulation run was at fixed times and in same order of the DataNodes (first heartbeats from DataNodes). As there is some initial ordering of heartbeat arrival times, which never changes, the subsequent heartbeat arrivals are also unchanged in different simulation runs. Such a behaviour was not acceptable as there is always some randomness in heartbeat arrivals to the JobTracker in real cluster experiments. To add randomness in the simulator, a random delay was added to the time at which a node can generate its first heartbeat. This change added randomness in the system in two ways: a node can provide an initial heartbeat to the JobTracker at some random time, and no fixed order was placed on the nodes as to when they can generate the initial heartbeat. It should be noted, however, that after the first heartbeat, the subsequent heartbeat from a node comes after precisely 300 milliseconds if it is idle.

The Capacity Scheduler was implemented at the back end of MRPERF. 500 lines of code were written for the implementation at the front end in Python and TCL and 2000 lines at the back end in C++. The following sections describe the main logic of the implementation and what was integrated into MRPERF from the Capacity Scheduler.

3.3.1 Data Structures and Config Files

The Capacity Scheduler supports the following parameters in `Capacity-scheduler.xml` regarding queues:

- QueueCapacity
- Maximum-Capacity
- Minimum-User-Limit-Percent
- User-Limit-Factor
- Supports-Priority
- Maximum-System-Jobs

- Maximum-Initialized-Active-Tasks
- Maximum-Initialized-Active-Tasks-Per-User
- Init-Accept-Jobs-Factor.

A python program is then used to convert the XML into TCL file `capsim.tcl`. The `capsim.tcl` data file contains all the queue-related parameter information. This is later read during setting up of the queues by the simulator. The job-related configuration information is read from a TCL file `job.trace.tcl` before starting the simulator. Each job contains its submit time. This allows the simulator to launch these jobs at the correct time when the simulator runs. Each job also has an assigned user and queue.

The queue and initialization poller data structure forms the core of the simulator. Each queue in the system is represented by a queue data structure. Each queue data structure contains lists of waiting and initialized jobs. The queue data structure also contains a structure for map and reduce slot usage (Capacity, number of running tasks, number of slots occupied, maximum capacity, active users list, number of slots occupied by each user), and a list of users containing user information (like name and job count and number of active tasks). For each user in the queue data structure, a user info data structure is maintained. The user info structure contains information such as, the list of waiting jobs, running jobs and initializing jobs, and the number of active tasks.

The poller data structure represents the initialization poller and is created in the simulator after setting up of the queues. It maintains a list of jobs that are initialized and also simulates the logic to initialize the jobs from the different queues.

3.3.2 Capacity Scheduler Implementation

The step by step procedure of the Capacity Scheduler simulator inner workings is described next. It is divided into 6 main stages. Initially, all the queues and the initialization poller are created and initialized. The users then submit their jobs to their respective queues where they are launched and initialized. The main scheduler algorithm picks a task from a job and assigns the task a map/reduce slot. Finally, at the end, a job is completely removed from the system.

Queue Setup

Setting up the queues is done before actual simulation runs by reading `capsim.tcl`. One queue is created for each queue name defined in `capsim.tcl`. Each queue is checked for consistency in parameter settings. For example, the capacity should be between 1 and 100, Maximum-Capacity should be less than 100, Minimum-User-Limit-Percent should be less than 100 but greater than 0, etc. The added capacities of the queues in the system should be less than 100. In case any of the checks fail, the simulation stops. After this, other values from `capsim.tcl` are read for each queue and a queue structure is created. However, there are initial values for some queue state variables that are computed from the values read from `capsim.tcl`. For example: $maxjobstoinit = ceil(Maximum - System - Jobs \times QueueCapacity/100)$. Similarly, $maxjobsperusertoinit = ceil(Maximum - System - Jobs \times QueueCapacity/100 \times Minimum - User - Limit - Percent/100)$.

Setting up initialization poller

This module initializes the jobs in the simulator. In the Capacity Scheduler, not all the jobs are initialized as they are submitted. This is done to reduce the memory footprints on the JobTracker node. An initialization poller wakes up at regular time intervals and initializes the jobs. A periodic timer is initialized in the simulator to trigger the invocation of the initialization poller. As mentioned previously, the interval at which the poller does its work is called *poll time* and is configurable in the simulator.

Launching jobs

The following steps take place in the simulator to place the job into the waiting list of an assigned queue:

- The first step is to check if the job can be accepted into the system after it is submitted. For example, if $jobtotaltasks > maxactivetasksperuser$ or $queueinitialized + queuewaitingjobs \geq maxjobstoaccept$ or $userinitialized + userwaitingjob \geq maxjobsperusertoaccept$, then the job cannot enter the system.
- The job is added to the queue's waiting list.
- The user list of the job's queue is updated by adding the user to the list (if the user is not already present in the list). The user info structure of the queue is initialized.
- The job is added to the waiting list of the user info data structure.
- The job count of the user is initialized to 1 if it is 0 and also initialize number of slots occupied by this user to be 0 for both map and reduce slots. Increase the job count otherwise.

At this moment, the job is waiting in its queue to be initialized by the initialization poller.

Initializing jobs

At each polling interval, the initialization poller does the following work to initialize the jobs and remove the already initialized jobs:

- Remove the job from the initialized jobs list of the poller if it was already initialized during the last wakeup period (when poller last scheduled the jobs to be initialized).
- Select jobs which can be initialized from each queue waiting list and perform following actions on each one of them:

  - Check if *queueinitializedjobcount* $>$ *maxjobstoinit*. If true, than next queue is considered for selection.

  - Check if user is not allowed to have more jobs ($user\_initialized\_jobs > max\_job\_per\_user\_to\_init$). If true, then next job in the same queue is considered for selection.

  - The job is placed in initialized jobs list of the poller.

  - The job is placed in initialized job list of the queue.

  - The active task for the queue is incremented by the job's total (map and reduce) tasks.

  - The user info structure is updated. The job is added to the initialized job list of user info and the number of active tasks in user info is updated.

  - The job is removed from the waiting list of queue and user info data structure.

Then, the poller sleeps till the next *poll time* after initializing the jobs from every queue.

Main Task Scheduler algorithm

When a heartbeat is received by JobTracker from TaskTracker in MRPERF, it picks up and schedules a job using the following algorithm:

- The queues are sorted according to the number of running tasks in the queue/queue capacity.

- In sorted queue list, the queue's admissibility to get a cluster slot is checked. The admissibility criterion requires the queue's new slot usage (old slot usage + 1) to be less than the Maximum-Capacity of the queue. If admissibility criteria fails, then the next queue is considered.

- After picking up the queue, the user and queue limit are checked for each job sequentially in that queue. The sequential checking continues until a suitable job is found under the limits. The next step is to find a suitable task from the selected job.

- The highest preference is given to a node or rack-local task for the job. If no such task is found, then a remote task is scheduled. At each heartbeat, only one task from remote can be scheduled. A second remote task is not scheduled even if slots are available. The search for a task continues to the next job if a task cannot be found from the current job. The search continues to the next queue if no task can be scheduled with the current queue.

- If a task is found, it is scheduled to run. The different structure and variables associated with the queue are updated.

Removing completed jobs

When the jobs are completed, the following steps are taken:

- Remove the job from the queue's initialized job list. Remove it from user info data structure as well.
- The user is removed from the system if he/she does not have any more active tasks. The number of active tasks in queue and user info data structure is updated.
- The slot usage information for the user in the queue is updated.

Chapter 4
Experiments

This chapter contains the experimental design used in the thesis. It discusses the hardware setup, the parameters and factors varied in different experiments done for sensitivity analysis, the validation of the simulator and the simulation/scalability analysis. The experiments done in the thesis were conducted in 3 stages: Sensitivity Analysis Experiments, Simulator Validation Experiments and Simulation Experiments. Sensitivity Analysis Experiments were conducted as a preliminary investigation into Capacity Scheduler settings and their impact on a few representative MapReduce applications on a real cluster. Simulator Validation Experiments were conducted next to check the accuracy of simulator results (built in the thesis) versus real cluster results. Finally, a large simulation was performed to identify the impact of Capacity Scheduler parameters on jobs under different job submission patterns.

4.1 Hardware Environment

This section describes the hardware setup used in different experimental scenarios. The number of nodes and information about the processor, disk and network capabilities is provided.

4.1.1 Sensitivity Analysis Experiments

Socrates is a cluster which consists of 37 Sun Microsystem computers. The cluster has 28 *capacity nodes* and each node contains 8 cores with 8 GB of RAM. There are 8 *capability nodes* and each node contains 8 cores with 32 GB of RAM. *Capability nodes* have higher RAM than *capacity nodes. Capacity nodes* have higher storage than *capability nodes.* There is one head node which submits jobs to the other 36 compute nodes in the cluster. Socrates is used as a High Performance Computing (HPC) platform for teaching, training, and research assistance at the University of Saskatchewan. Exclusive access to the Socrates cluster was provided for a small time frame by the HPC team at the University to conduct the initial sensitivity analysis experiments.

The experiments were conducted on an isolated 6-node Socrates cluster. One node was designated as the JobTracker and NameNode. The other 5 nodes carried out the tasks of DataNode and TaskTracker. All the nodes had 128 Gb of hard disk with 2xQuad core Intel Xeon L5420 processors at 2.5 GHz, 8 GB of RAM and a 1 Gbps network connection. RedHat 5.3 Linux was the OS on all the nodes, executing Hadoop 0.20.203.

4.1.2 Simulator Validation Experiments

The real experimental cluster used for validation experiments consists of 5 nodes on local Discus lab machines at the University of Saskatchewan. One node was designated as JobTracker and NameNode while the other nodes served as TaskTracker and DataNodes. Each node consisted of a 2.4 Ghz processor, 4 GB of RAM and 250 GB hard disk. Ubuntu 10.04 Linux was the OS on all the nodes, executing Hadoop 0.20.203. All the nodes were on same rack and connected through a 1 Gbps switch. The validation experiments could not be done on Socrates Cluster as it was not possible to get exclusive access to the cluster for a very long period of time. The simulator validation was conducted on a laptop. The simulator contained the exact setup as the real cluster setup.

4.1.3 Simulation Experiments

The experiments with different configurations were done by simulation. The setup consists of 31 nodes where one node served as NameNode and JobTracker. The other 30 nodes act as DataNodes and TaskTrackers. Each node is modelled as a quad core processor with 2 CPUs, a single disk and 4 GB of memory. The amount of memory seems to be small for a quad core machine. However, it did not make a difference in the simulator. It turned out to be the case that although memory is a configurable parameter in the simulator, it is not used in the code. All the nodes are connected to same switch on one rack. All the nodes have 4 map and 4 reduce slots. The simulation was done on the Socrates cluster.

4.2 Definitions

4.2.1 Same-reduce-node Effect

The same-reduce-node effect describes the situation in which reduce tasks for two jobs are scheduled on same TaskTracker node. This leads to an increase in the reduce execution time for the following reasons: increased competition for network, cpu and disk resources. The major reasons tends to be the competition for disk and network. In a cluster, if each node has the same number of slots as the number of cores, then cpu is not a bottleneck. The same-reduce-node effect occurs due to the randomness present in the real system. The TaskTrackers send periodic heartbeats to the JobTracker if they have empty map/reduce slots. So, the JobTracker schedules the next pending map/reduce task for a job depending on when it gets the heartbeat and does not differentiate how many reduce/map tasks are already running on that particular TaskTracker. For example, if all the nodes in the cluster have 2 reduce slots, then every node can send at most 2 heartbeats for reduce tasks. Thus, in some cases when the JobTracker gets the 1st heartbeat for a reduce slot from a TaskTracker, it schedules a job's reduce task on that slot. If that same TaskTracker then sends a 2nd heartbeat and if the JobTracker has another pending reduce task for another job, then it will be scheduled on the same TaskTracker. So, it is a matter of chance as to which TaskTracker's heartbeat the JobTracker gets when it has a map/reduce task for a job to be scheduled.

4.2.2 Delayed Map Execution

In general (except Minimum-User-Limit-Percent setting), if there is a job 1 which is submitted before job 2, then job 2 map tasks do not start execution until job 1's map tasks are finished or there are extra slots remaining idle. However, in some runs, job 2 gets one or two map slots quickly (even though job 1's map tasks were not finished, because of a temporary condition where a slot was idle and the "wrong" job was selected). It means job 2 map execution started as soon as it was submitted to the system. Later, when the JobTracker received new heartbeats from the TaskTracker to assign jobs to the map slots, job 1 took preference as it was submitted earlier and until job 1 completes all its map tasks, there were no job 2 map tasks scheduled. This leads to unusually longer map times than expected in some cases. This effect is called delayed map execution in the rest of the thesis. Delayed Map Execution is not to be confused with Delay Scheduling. They both are different concepts.

4.2.3 Performance Measures

The various performance metrics used in the experiments are described next:

- **Data locality**: plays an important role in MapReduce environments. When more node-local tasks are scheduled over rack-local and remote tasks, performance of the jobs is better because of the reduced network activity.
- **Map time for jobs**: The time taken by map processing for each job.
- **Reduce time for jobs**: The time taken by reduce processing for each job.
- **Waiting time**: The waiting time in the queue for each job in the cluster until it gets its first map task started.
- **Execution time**: The execution time for each job in the cluster, excluding waiting time.
- **Elapsed time**: The turnaround time for each job in the cluster, including waiting time.
- **Response Ratio**: It is measured as $elapsedtime \div executiontime$ for each job. It is a fairly standard term in the Operating Systems performance community [18] and is frequently used as a metric in scheduling. The higher the value of the response ratio, the more relative waiting time for the job. For jobs whose execution time is 1000 seconds, a waiting time of 20 seconds would not affect its performance much. However, the same waiting time for a job whose execution time in only 40 seconds, can make a big difference. Hence, for better performance of the cluster, it is important to have short response time for the short jobs and thus, a lower response ratio.

- **Makespan**: The time in which all jobs are finished. "Makespan is a useful metric to understand the overall throughput of the cluster, and it is used to verify that improvements in response ratios do not come at the expense of being able to run fewer jobs per hour on the shared cluster [21].

Throughput is another very important metric to be measured in a performance evaluation study. However, because of time limitations it was not studied.

4.3 Parameters

This section describes the parameters which do not change in a given experimental scenario. Some of the things, like job types, number of jobs, number of queues, number of users, etc. remain constant and are presented in this section.

4.3.1 Sensitivity Analysis Experiments

In these experiments, each queue has 3 users and each one of them submits one job, one after another, at the interval of one second. Multiple queues were supported. Each user submits a specific job. The first user in each queue submits application *Sort*, the second submits *TeraSort* and third submits *WordCount*. *TeraGen*, *RandomWrite* and *RandomTextWrite*[1] were used to generate the input data for *TeraSort*, *Sort* and *WordCount*, respectively. In all executed MapReduce applications, the size of the input data file was 5 GB.

The *Sort* application uses the MapReduce framework to sort the input file into the output file. *TeraSort* is a MapReduce application which reads the input data and uses MapReduce to sort the data into a total order. It is different from *Sort* in the sense that it uses a custom partitioner rather than a default partitioner.[2] *WordCount* application reads text files and counts the total number of words in the files. The *RandomWriter* application writes random data to the HDFS using MapReduce. It is used to generate input data for the *Sort* application. The *Randomtextwriter* application uses MapReduce to generate files containing large random words. It is used to generate the input data for the *WordCount* application. *TeraGen* is used to generate the input data set for *TeraSort*.

4.3.2 Simulator Validation Experiments

In all the experiments, *Sort* and *TeraSort* were used as the jobs as their output data is proportional to the input data (a requirement for MRPERF). Two queues are used in the experiments, with 4 users per queue. Jobs in each queue were submitted with an inter-arrival time of 5 seconds, while jobs of same type in different queues were submitted at the same time. *TeraGen* and *RandomWriter* were used to generate the input data for the *TeraSort* and *Sort* jobs, respectively. *TeraSort* used 2 GB of input data, while *Sort* used 1 GB of

[1] http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/examples/ (22nd May, 2013)
[2] http://sortbenchmark.org/YahooHadoop.pdf (22nd May, 2013)

input data so as to have different input data sizes for different type of jobs. The workload generator generates the jobs in the following order:

- *TeraSort* submitted for user 1 and user 3 in Queue 1 and Queue 2, respectively.
- *TeraSort* submitted for user 2 and user 4 in Queue 1 and Queue 2, respectively, after 5 seconds.
- *Sort* submitted for user 1 and user 3 in Queue 1 and Queue 2, respectively, when their *TeraSort* job is finished.

4.3.3 Simulations

100 jobs of 5 different types were used. Every job has a different user. Existing research [10, 11] shows that in clusters used by Yahoo! and Facebook, the small jobs dominate the workload. Their percentage share is found to be 80%. The description of the jobs is shown in Table 4.1.

Table 4.1: Job Types For The Experiments

Job Type	**Nature of job**	**Percent in job mix**	**Num of Maps**	**Num of reducers**	**Shuffle ratio**	**Output ratio**	**Map compute**	**Reduce compute**
1	Data transformation	8	100	15	1	1	5	40
2	Aggregate and expand	6	200	1	0.025	3	40	5
3	Expand and aggregate	4	400	30	3	0.025	40	40
4	Data summary	2	800	8	0.075	0.0005	20	20
5	Small job	48	1	1	1	1	1	1
6	Small job	24	2	1	1	1	1	1
7	Small job	8	10	1	1	1	1	1

Small jobs have a small amount of input data and hence a small number of maps. The shuffle ratio for each job describes the amount of data which is produced after the map stage and serves as the amount of data for the reduce stage. The shuffle ratio is the ratio between the size of the map output data and size of the map input data. In the simulation, the total input data generated after the map stage is equally divided among the number of reducers for the job. The output ratio for the job types describes the final output data size after the reduce stage is done. The output ratio is the ratio between reduce output data and reduce input data. Long jobs, (those with more than 10 map tasks) are divided among 4 different categories.

For Data Transformation jobs, the data ratio remains similar at all stages. *TeraSort* is a good example of a Data Transformation job. The ratios for Data Summary type jobs are low, and decrease greatly at each stage (shuffle ratio for map and output ratio for reduce). In Aggregate and Expand, the data is aggregated during the map and thus generates comparatively small amounts of shuffle data, but expands during the reduce stage, producing a larger amounts of reduce output data. The process is reversed for the expand and aggregate job type. The number of maps and reduce tasks for each job was chosen to keep the cluster busy most of the time.

4.4 Factors Varied

This section contains the discussion abut the nature of the experiments and what was done in the experiments. The Capacity Scheduler settings changed for the experiments are listed. It also presents the main motivational ideas to undertake the simulation/scalability analysis.

4.4.1 Sensitivity Analysis Experiments

Single factor Experiments were done to determine if parameter settings had any effect and to estimate the magnitude of that effect. In these experiments, the Capacity Scheduler configuration settings were changed one by one and their impact on the job execution time was observed. The factors changed in the experiments are the following: NumberofQueues, QueueCapacity, Maximum-Capacity, Minimum-User-Limit-Percent, User-Limit-Factor, Supports-priority, Maximum-System-Jobs, Maximum-Initialized-Active-Tasks and Maximum-Initialized-Active-Tasks-Per-User. These experiments provide empirical evidence supporting the claim that these parameter settings influence performance and that the default parameters do not necessarily provide the desired performance.

4.4.2 Simulator Validation Experiments

These experiments were done after the implementation and integration of the Capacity Scheduler in MR-PERF. The main motive of these experiments was the verification of the real cluster results versus simulator results on representative MapReduce applications. The factors changed in the experiments are the following: NumberofQueues, QueueCapacity, Maximum-Capacity, Minimum-User-Limit-Percent, User-Limit-Factor, Maximum-System-Jobs, Maximum-Initialized-Active-Tasks and Maximum-Initialized-Active-Tasks-Per-User. Two different types of node configurations were used: One reduce slot per node and two reduce slots per node.

4.4.3 Simulation

These experiments were done on a cluster of 31 simulated nodes with different job submission patterns. There are three major questions of interest and the experiments were designed based on those questions:

- **Question 1**: How do the short jobs behave when they come after a long sequence of long jobs? What Capacity Scheduler settings can optimize the performance of short jobs under such conditions?
- **Question 2**: What are the effects of Capacity Scheduler settings for a job submission pattern where long job and a series of short jobs arrive in an interleaved fashion? Does it follow the results from previous question or not?
- **Question 3**: Does providing different queues for short and long jobs improve the performance of the system? Does it impact the performance of short jobs?

The 4 most important Capacity Scheduler parameters were changed under the experiments: Queue Capacity, Maximum-Capacity, Minimum-User-Limit-Percent and User-Limit-Factor.

To answer Question 1, all the long jobs were submitted to the cluster first, followed by all short jobs in the experiments. There were two queues, each containing 50 jobs and job mix as described in the Table 4.1. In both the queues, the jobs were submitted in same order and at the same time instant. The time gap between each job submission was 1 second. Full factorial experiments with 10 simulations were done. Description of the settings changed during each experiment is given in Table 4.2. For parameter definitions, please refer to Table 2.1.

To answer Question 2 and Question 3, two types of job submission patterns were submitted to the simulator. In the first type, 4 short jobs were submitted and then a long job followed by the same pattern. 2 queues with equal capacity (Queue 1 and Queue 2) of 50% were used. Each user has a single job in the system. 50 jobs were submitted to the first queue and 50 to the second. The inter-arrival time between the jobs was 1 second. In the second type of job submission pattern, short and long jobs were given separate queues. Queue 1 was divided into two sub-queues and its capacity of 50% was divided between them. Long jobs need more slots. So, 40% capacity was allocated to long job queue and 10% to the short job queue. The same allocation was applied to queue 2. The time of arrival for the jobs was kept as it was in interleaved case. The experiments conducted in first type of job submission pattern are refereed to as the "interleaved scenario" while the second type will be refereed as the "separate queue scenario" in the rest of the thesis. Description of the settings changed during each experiment is given in Table 4.3.

Experiments in the interleaved scenario and with separate queues further try to answer a few important questions:

- How does response ratio change between the two job submission patterns? Which Capacity Scheduler parameters improves response ratio for the jobs?
- How does Makespan (throughput) change between the two job submission patterns? Which Capacity Scheduler parameters improve Makespan for the jobs?
- Does allocating separate queues improve response ratio for short jobs?
- Does allocating separate queues improve Makespan for short jobs?

Table 4.2: Parameter Settings For The Experiments

Experiment No	**number of queues**	**Capacity**	**Maximum-Capacity**	**User-Limit-Factor**	**Minimum-User-Limit-Percent**
1	2	50-50	-1	1	100
2	2	50-50	50	1	100
3	2	50-50	-1	2	100
4	2	50-50	-1	1	25
5	2	50-50	50	2	100
6	2	50-50	50	1	25
7	2	50-50	-1	2	25
8	2	50-50	50	2	25
9	2	70-30	-1	1	100
10	2	70-30	50	1	100
11	2	70-30	-1	2	100
12	2	70-30	-1	1	25
13	2	70-30	50	2	100
14	2	70-30	50	1	25
15	2	70-30	-1	2	25
16	2	70-30	50	2	25

Table 4.3: Parameter Settings For The Experiments.

Experiment No	**number of queues**	**Capacity**	**Maximum-Capacity**	**User-Limit-Factor**	**Minimum-User-Limit-Percent**
17	2	50-50	-1	1	100
18	2	50-50	50-50	1	100
19	2	50-50	-1	2	100
20	2	50-50	-1	1	25
21	2	50-50	50-50	2	100
22	2	50-50	50-50	1	25
23	2	50-50	-1	2	25
24	2	50-50	50-50	2	25
25	4	40-40-10-10	-1	1	100
26	4	40-40-10-10	40-40-10-10	1	100
27	4	40-40-10-10	-1	2	100
28	4	40-40-10-10	-1	1	25
29	4	40-40-10-10	40-40-10-10	2	100
30	4	40-40-10-10	40-40-10-10	1	25
31	4	40-40-10-10	-1	2	25
32	4	40-40-10-10	40-40-10-10	2	25

Chapter 5 Sensitivity Measurement Results and Simulator Validation

Before implementing the Capacity Scheduler simulator in MRPERF, it was important to establish the fact that Capacity Scheduler parameters have an impact on the performance of MapReduce applications. The initial part of this chapter contains the single factor sensitivity measurement analysis. The experiments were conducted on a real world cluster with representative MapReduce applications and focusing on every single parameter present in the Capacity Scheduler. This analysis was necessary to give an initial insight into the working mechanism of Capacity Scheduler and to understand the significance of the parameters. In the second part of this chapter, the accuracy and functionality of the simulator components designed in this work are validated by comparing the simulator results against the real world cluster results.

5.1 Sensitivity Measurement Analysis

Measurements directly on hardware allows clear isolation and identification of the performance variations caused by the task scheduler settings. *TeraSort*, *Sort*, and *WordCount* were used as benchmark MapReduce applications for the experiments. End users are interested in the overall execution time of their jobs rather than individual map or reduce phase timings. Hence, execution time was used as the performance metric in the experiments. The individual map, reduce and wait times are also shown. The time shown in the results is an average of 3 runs. In all the runs for a particular setting, there was a small variation except while testing the Minimum-User-Limit-Percent, where results were not uniform. The experiments were designed while keeping in mind the Capacity Scheduler configuration settings. Multiple users started their jobs at the same time. A simple workload generator was created which submitted the job in each defined queue of the scheduler. In these experiments, the Capacity Scheduler configuration settings were changed one by one and their impact on the job execution time was observed. The number of queues ranged from 1 to 3. The first user in each queue submits application *Sort*, the second submits *TeraSort* and third submits *WordCount*. Users 1-3 are in queue 1, users 4-6 are in queue 2 and users 7-9 are in queue 3 in all the experiments done. Subsequent experiments use the same settings in terms of the number of users per queue and their job submission pattern, unless stated otherwise. There are 3 users per queue (except the User-Limit-Percent experiment) and each user has exactly one job to submit. In the User-Limit-Percent experiment, a single queue with 4 users were supported. The number of map tasks depended on the size and type of input. The Hadoop framework decides at runtime how many map tasks to create. In the experiments, each job had between 70-80 map tasks. Each job has one reduce task. The number of reduce tasks for a job is a job configuration parameter and its default value is 1. In almost all the experiments (unless specified otherwise), the default values of the job configuration parameters were used. The results graphs show the 2 components of execution time: map time and reduce time. The execution time is not the sum of the 2 components, since there is overlap between the map and reduce phases of a job. Wait time is shown as well.

5.1.1 Impact of Resource Allocation Parameters

In these experiments, the parameters of the Capacity Scheduler related to cluster resources were changed, such as the number of queues, their capacity and user related queue configuration settings.

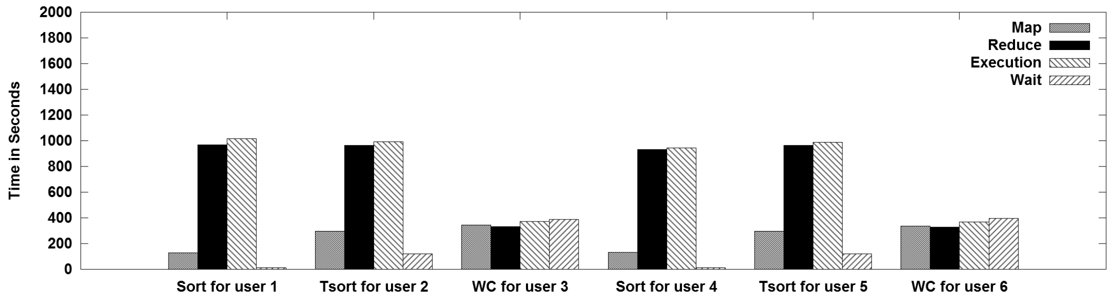
NumberofQueues and QueueCapacity

The time for each job increases with *NumberofQueues* due to increased contention for shared resources like disk and network bandwidth. Figure 5.1 shows the execution times for the various jobs under different *NumberofQueues*. To check the impact of increasing *NumberofQueues*, compare Figure 5.1(a) and 5.1(c). The execution time for all jobs increase almost by a factor 2. This is a natural outcome when the workload is doubled for the same set of resources.

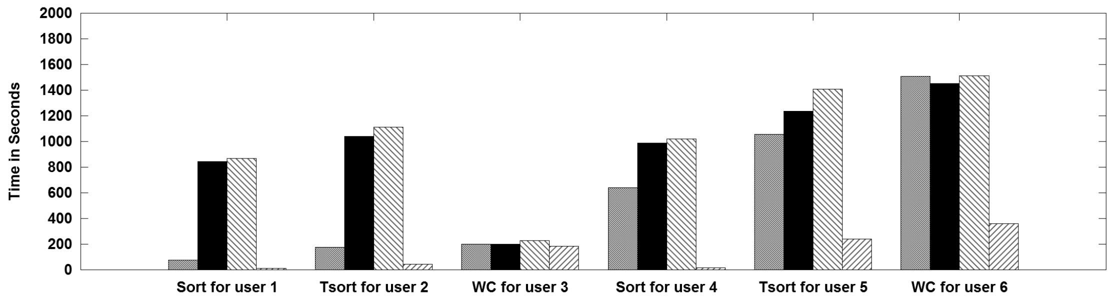
QueueCapacity is the guaranteed capacity which a queue will have at any time. The effect of *QueueCapacity* can be clearly seen between Figure 5.1(a) and 5.1(b). For the same number of jobs, different execution time were observed. The jobs being executed in queue 1 get more slots, and hence a reduction in their execution time. Note that this improvement is mainly because more map slots are available for that queue. The number of reduce tasks for each job is 1, requiring one reduce slot. More capacity means more map and reduce slots, decreasing map execution time for jobs in queue 1 and subsequently reduce time as well. However, sometime, the improvement may not be clearly visible because of stragglers as with *TeraSort* for user 2 in Figure 5.1(b). Due to stragglers in the map phase, tasks in the reduce phase do not continue smoothly and have to wait a lot before they finish, which eventually also affects total execution time. Under default settings, the jobs are executed in FIFO fashion. Hence, there is increasing waiting time according to the job arrival time. A job cannot start its execution until the earlier jobs in the queue have finished map processing. However, sometime this does not happen because of delayed map execution, as in Figure 5.1 (c).

Maximum-Capacity

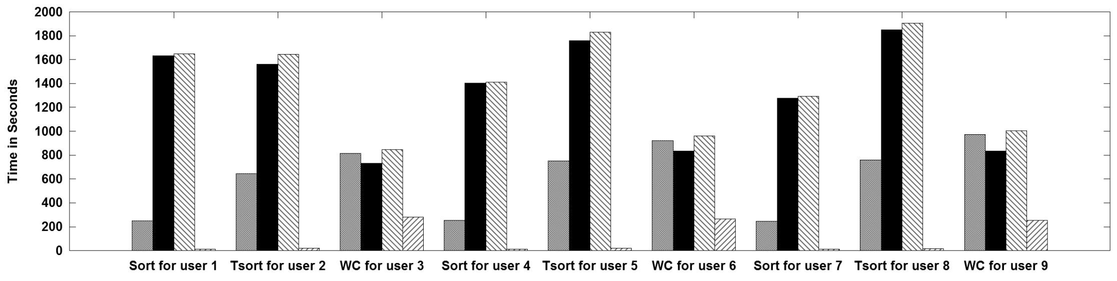
This parameter allows a queue to use unused capacity of other queues, if available. A queue can use resources in the cluster between the value of QueueCapacity and Maximum-Capacity (100% when the default value



(a) 2 queue: 50% each capacity



(b) 2 queue: 90%-10% capacity

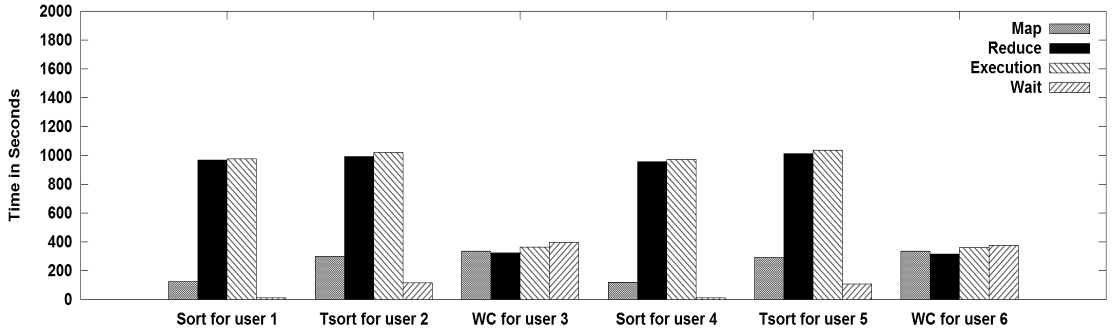


(c) 3 queue: 33% each capacity

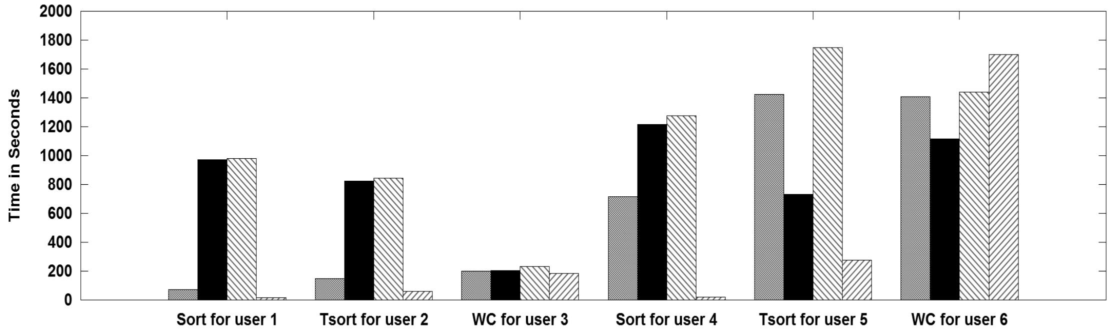
Figure 5.1: Effect Of *NumberofQueues* And *QueueCapacity*

of -1 is used). The value of *Maximum-Capacity* has to be at least *QueueCapacity*. Figure 5.2 shows the execution times for the various jobs under different settings. Figure 5.2 (a) shows the execution time for 2 queues, each having 50% capacity, while 5.2 (b) has 2 queues with first having 90% and other queue having 10% *Maximum-Capacity*. In both cases, *Maximum-Capacity* is equal to *QueueCapacity*.

Comparing Figure 5.2 (a) with Figure 5.1(a), setting *Maximum-Capacity* equal to *QueueCapacity* does not make much difference. No queue can use more than 50% of the map and reduce slots. In this case, no queue's jobs can interfere with jobs from another queue. This leads to slightly better map execution time. The reduce time is not affected by this change, as each job needs 1 reduce slot. A single reduce slot is available in both 50% configurations without or with *Maximum-Capacity* enforcement. In Figure 5.2 (b), the execution time for jobs in queue 1 remains similar to Figure 5.1(b) for the reasons explained before. However, for the second queue, the execution time increases for all jobs as it is allocated only 10% of the allocated capacity (1 map and 1 reduce slot) and cannot take more than its allocated capacity.



(a) 50-50% queues



(b) 90-10% queues

Figure 5.2: Effect Of *Maximum-Capacity* On Running Time Of Jobs.

Figure 5.2 shows that the map phase execution time is greater than reduce phase execution time for the subsequent jobs after the first job for a smaller capacity queue. This is because a job typically has multiple map tasks, but one reduce task. When there is only one map slot present (as is the case with 90-10% queue), it takes more time for map tasks to finish. The other reason for this phenomenon is that when a queue has only one reduce slot, the second job cannot start its reduce phase until the first job is finished. However, the second job's map phase is started when the first job finishes its map phase. Thus, by the time the second job gets its reduce slot, most of its map tasks are already finished. As a result, the reduce phase gets most of the data it needs to proceed immediately and does not have to wait much for map tasks to finish, which leads to faster reduce time. This also affects the total execution time.

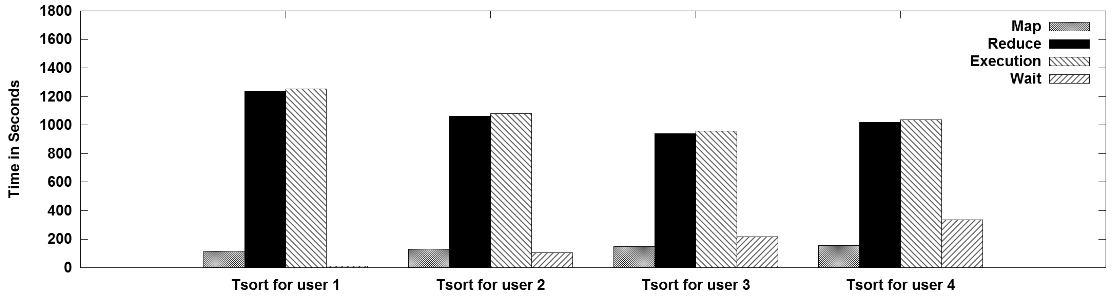
Minimum-User-Limit-Percent

This parameter allows limits to be defined on per user allocation of resources for a given queue. To check the impact of this parameter, a single queue, 100% capacity and 4 users was used. Each user simultaneously submitted one *TeraSort* job. Figure 5.3 shows the execution times for the various jobs. The map execution time for all jobs increases significantly when *Minimum-User-Limit-Percent* is changed from 100% to 25%. This is because when *Minimum-User-Limit-Percent* is 100%, the first job gets executed on all the map slots and when its map phase is finished, the map phase for the second job starts and so on. When *Minimum-User-Limit-Percent* is 25%, however, each job starts the map phase at the same time, which also leads to an increase in map execution time, as there are fewer map slots per job. Setting Minimum-User-Limit-Percent value to 25% improves waiting time for the jobs.

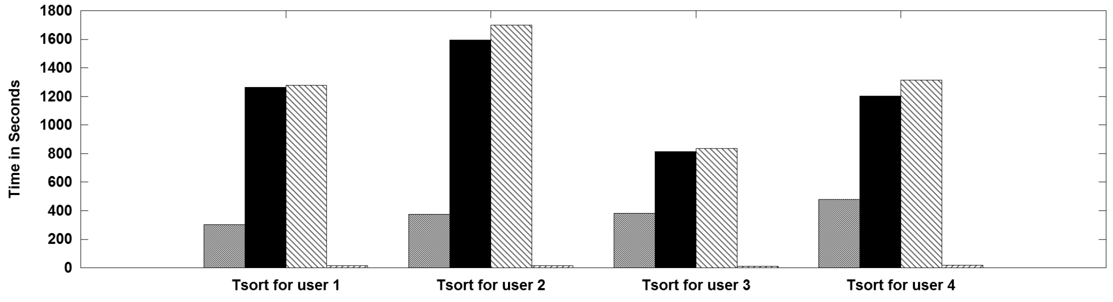
Completely unexpected results for the reduce time were seen in Figure 5.3 (b). The *reduce* execution times were not uniform for the different users as observed in the results. The reduce execution time hovered around 14 minutes for the jobs in one set of execution and in the other, the reduce time hovered around 28 minutes. A closer look at the logs revealed that this disparity was caused by scheduling two jobs simultaneously on two different slots on the same cluster node. Recall that every node has 2 map and 2 reduce slots. At times when the Capacity Scheduler selects a job's reduce phase to run on a node where no other job's reduce phase is running, then the job's running time was lower (i.e. 14 minutes). In the other case, however, as two jobs were running on the same cluster node, the reduce time increased, mainly due to increase in shuffle and sorting time. The shuffle time was increased because both jobs share network bandwidth and the sorting time was increased due to sharing of disk bandwidth.

In the experiments, job 3's reduce task (refer to Figure 5.3(b)) was never executed in parallel with some other job's reduce task on a cluster node. Hence, there was a decrease in reduce time, compared to Figure 5.3(a). In Figure 5.3(a), the reduce task of job 3 was sometimes scheduled with other jobs on same cluster node. It can be considered as a stochastic effect, because apart from the exception for job 3 in Figure 5.3(b), the reduce phase of all other jobs were scheduled with another job's reduce task on same node at some time.

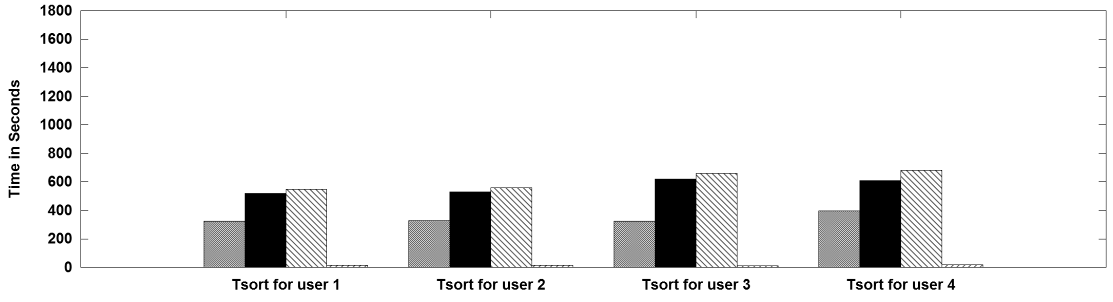
In Figure 5.3(c), the interaction between job configuration parameters (number of reduce tasks) and scheduling parameters (User-Limit-Factor) is shown. It is shown to emphasize that job execution time in MapReduce is not only dependent on job configuration but also on scheduling parameters. With more reduce tasks, the execution time of jobs is reduced as more parallelism is achieved. Shorter reduce time improves the



(a) Minimum-User-Limit-Percent 100%



(b) Minimum-User-Limit-Percent 25%



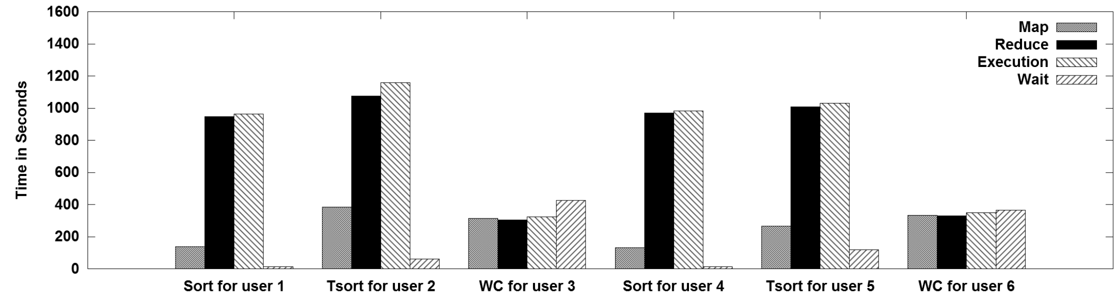
(c) Minimum-User-Limit-Percent 25%, 5 reduce tasks/job

Figure 5.3: Effect Of *Minimum-User-Limit-Percent*

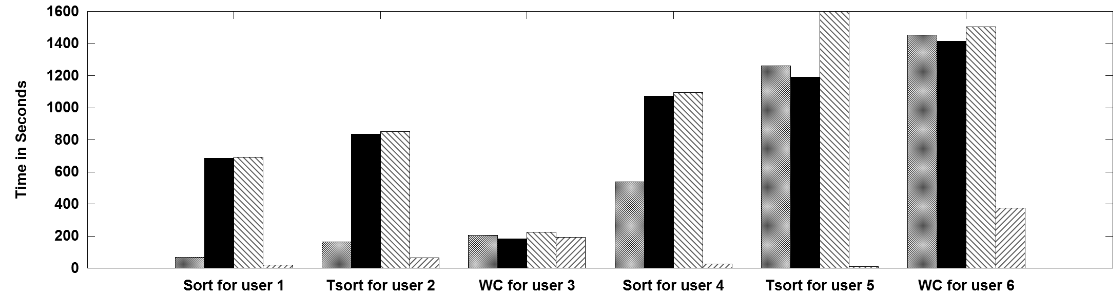
overall execution time of the jobs. Waiting time was small for all the jobs when *Minimum-User-Limit-Percent* setting was used. Under this setting, more jobs can execute concurrently, which leads to low waiting time.

User-Limit-Factor

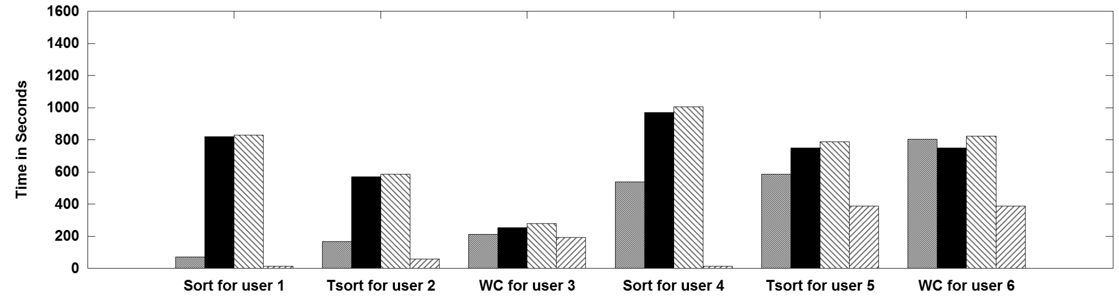
This parameter allows a single user to acquire more slots than the configured queue capacity. Figure 5.4 shows the execution times for the various jobs.



(a) 2 queues 50% each, ulf=2, 1 reduce task



(b) 2 queues 90%-10%, ulf=1, 2 reduce tasks



(c) 2 queues 90%-10%, ulf=2, 2 reduce tasks

Figure 5.4: Effect Of *User-Limit-Factor*

The number of reduce tasks per job and User-Limit-Factor was varied in these experiments. There is no change in execution time of jobs between Figure 5.4(a) and 5.1(a). This is because although *User-Limit-*

Factor is 2, it does not help as the number of reduce tasks for each job is 1. Each job needs one reduce slot and it was obtained through its allocated queue capacity. *User-Limit-Factor* was not relevant.

However, there is a big difference between Figure 5.4(b) and 5.4(c). The execution times for jobs in the second queue are much higher in 5.4(b) than in 5.4(c). The reason is the value of *User-Limit-Factor*. In Figure 5.4(b), the user from queue 2 cannot use more than 1 map and reduce slot as User-Limit-Factor is 1 and queue capacity is 10% (1 map and 1 reduce slot). In Figure 5.4(c), however, the user from queue 2 can get twice the queue capacity and hence it can get 2 reduce slots (from the other queue because the jobs in queue 1 only need 6 reduce slots and 4 are free) which leads to a substantial reduction in execution time.

Supports-priority

This parameter allows priority to be given to the users of any queue. The values of priority are of the following types: VERY_LOW, LOW, NORMAL, HIGH and VERY_HIGH. High priority jobs have shorter execution time than lower priority jobs, depending on the priority type. *Sort* was assigned VERY_LOW priority. *TeraSort* was given the NORMAL priority. *WordCount* was given VERY_HIGH priority. Figure 5.5 shows the execution times for the various jobs. Compared with Figure 5.1(a), it can be see that priorities clearly affect job execution time.

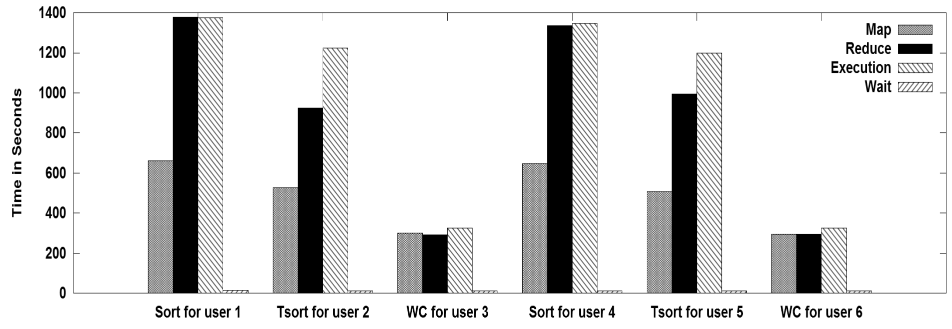


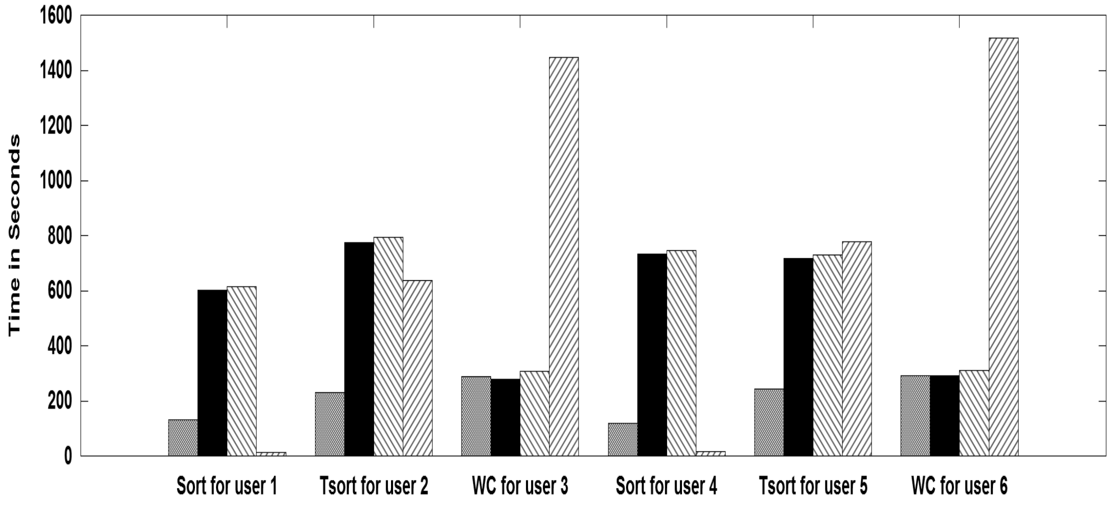
Figure 5.5: Effect Of Priority Settings

5.1.2 Impact of Job Initialization Parameters

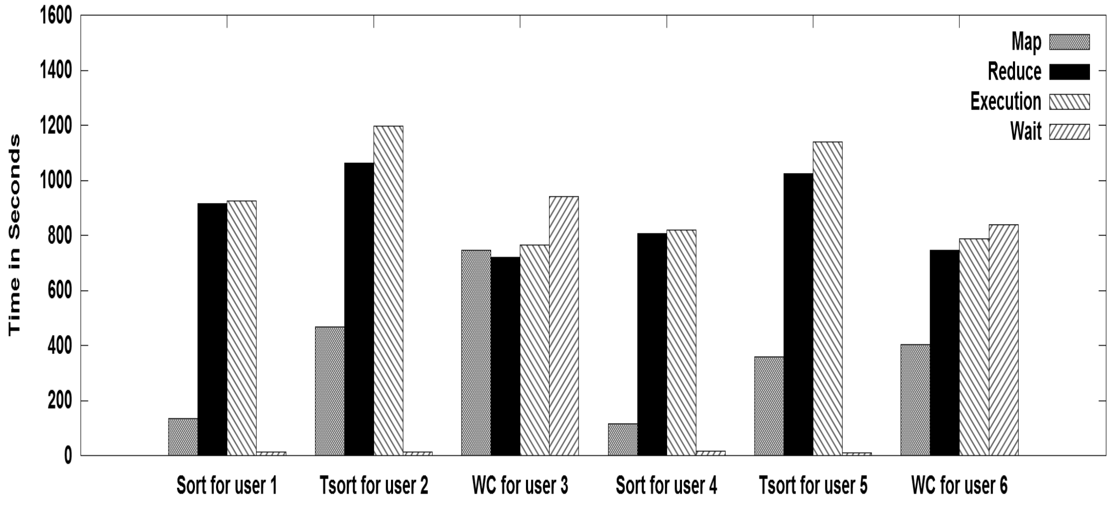
In the next set experiments, the job initialization parameters in the Capacity Scheduler were changed. These parameters determine the number of system jobs, tasks per queue and tasks per user which can be executed concurrently on the cluster.

Maximum-System-Jobs

In this experiment, the number of *Maximum-System-Jobs* was varied from 2 to 4, respectively, to observe the impact on the execution time of the jobs. Figures 5.6 shows the execution times for the various jobs under different settings.



(a) Maximum-System-Jobs=2



(b) Maximum-System-Jobs=4

Figure 5.6: Effect Of *Maximum-System-Jobs* On Running Time Of Jobs.

When the value of *Maximum-System-Jobs* is 2, only a single job from each queue gets executed in parallel. As a result, the first *Sort* job in both queues has a better execution time than in Figure 5.6 (b). When one job in a queue finishes execution, the next job starts executing, causing a long launching time for subsequent jobs. They have to wait a lot before they start executing. That is why there are large waiting times for the other two types of jobs. In the second scenario, the number of jobs which are being executed in each queue in parallel is 2. This leads to more resource contention among the running jobs and affects their execution time. The *Sort* job execution time increases because more parallel jobs can run in the system and each runs slower.

Alternatively, *TeraSort* and *WordCount* have faster real execution time because they do not have to wait as long as in the previous scenario. Real execution time does not include waiting time. The real execution time for *WordCount* was similar in both scenarios and the difference observed is due to waiting time.

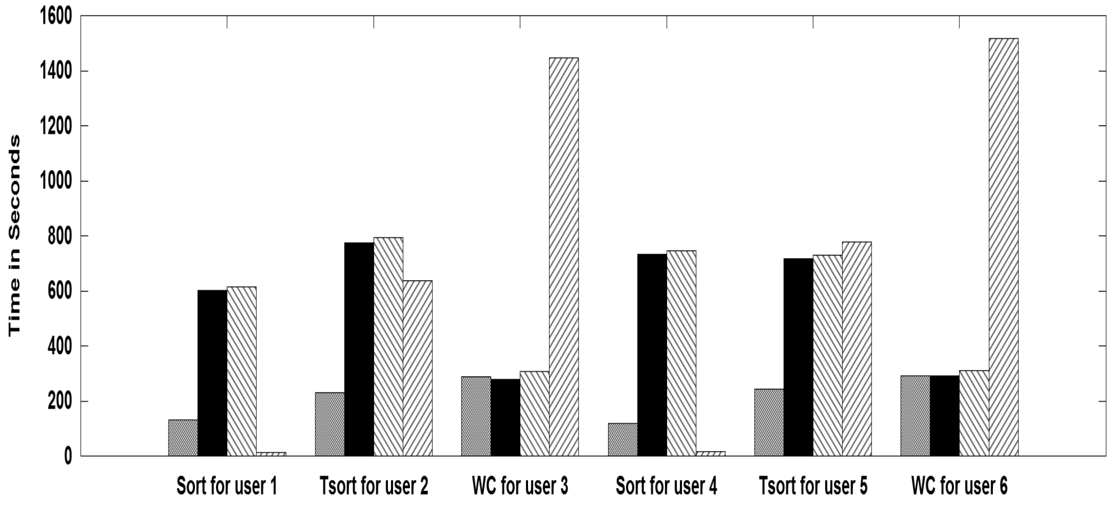
Maximum-Initialized-Active-Tasks and Maximum-Initialized-Active-Tasks-Per-User

These parameters are related. *Maximum-Initialized-Active-Tasks* defines the maximum number of tasks which can be executed concurrently for any queue, serving as an upper limit for maximum initialized active tasks per user. *Maximum-Initialized-Active-Tasks-Per-User* cannot be greater than *Maximum-Initialized-Active-Tasks*. Figure 5.7 shows the execution times for the various jobs under different settings. The same trends were observed for all the jobs in both the figures for the same reasons as explained earlier for *Maximum-System-Jobs*.

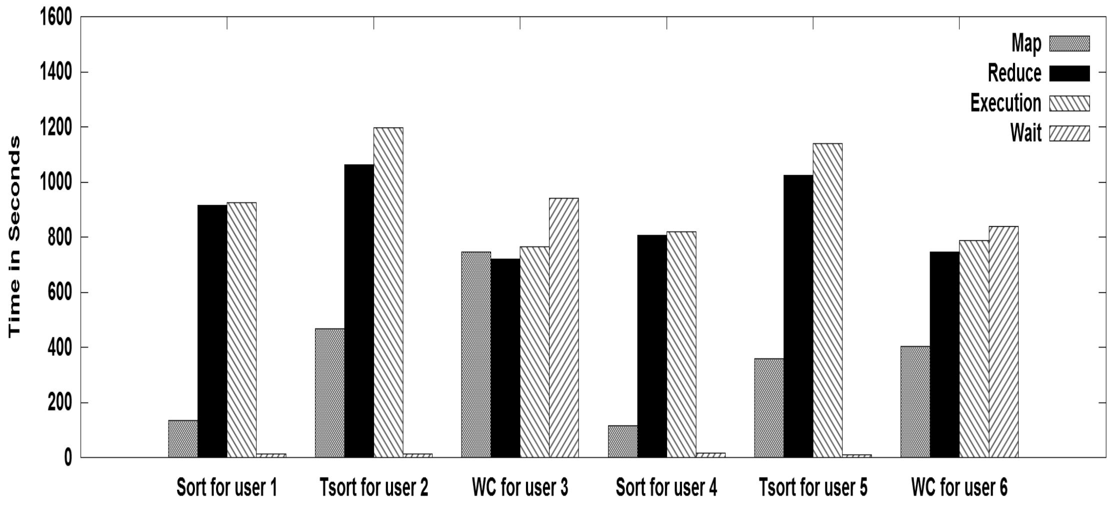
The reason for choosing 80 active tasks per user was that all the jobs had between 75 and 80 map tasks. In Figure 5.7 (a), when *Maximum-Initialized-Active-Tasks* for a queue is 100, then only a single job can be run from that queue as $\lfloor \frac{100}{80} \rfloor = 1$. Hence, there are large waiting times for all jobs after the first job, but short real execution time. In 5.7 (b), the number of jobs which can be executed simultaneously from any queue is $\lfloor \frac{160}{80} \rfloor = 2$. This leads to smaller waiting times for the jobs but large real execution time, due to contention of resources, except *WordCount* which was the last job to run in both of the scenarios.

5.1.3 Makespan

Makespan is an important metric to understand and merits discussion. It was found out that Makespan value was affected by the Capacity Scheduler settings across different experiments. But it did not change much when Capacity Scheduler settings were changed for the same workload. Table 5.1 shows the Makespan values for different experiments. The lower capacity queue has higher Makespan value than the larger capacity queue because jobs have fewer slots to execute on. In experiments with Maximum-System-Jobs and Maximum-Initialized-Active-Tasks, the Makespan value is determined by the jobs which executes at the end (as the jobs run sequentially and not in parallel).



(a) maximum-initialized-active-tasks=100, maximum-initialized-active-tasks-per-user=80



(b) maximum-initialized-active-tasks=160, maximum-initialized-active-tasks-per-user=80

Figure 5.7: Effect Of *Maximum-Initialized-Active-Tasks* On Running Time Of Jobs.

Table 5.1: Makespan For All Experiments

Experiment Description	**Queue 1 Makespan**	**Queue 2 Makespan**	**Queue 3 Makespan**
2 queue: 50% each capacity	1127	1123	-
2 queue: 90%-10% capacity	1131	1921	-
3 queue: 33% each capacity	1811	1866	1939
50-50% queues Maximum-Capacity	1149	1161	-
90-10% queues Maximum-Capacity	1292	2195	-
User-Limit-Factor 100%	1388	-	-
User-Limit-Factor 25%	1671	-	-
User-Limit-Factor 25%, 5 reduce tasks/job	691	-	-
2 queues 50% each, ulf=2, 1 reduce task	1234	1165	-
2 queues 90%-10%, ulf=1, 2 reduce tasks	1030	1899	-
2 queues 90%-10%, ulf=2, 2 reduce tasks	847	1249	-
Priority Settings	1454	1436	-
Maximum-System-Jobs=2	1774	1851	-
Maximum-System-Jobs=4	1724	1648	-
maximum-initialized-active-tasks=100, maximum-initialized-active-tasks-per-user=80	1852	1730	-
maximum-initialized-active-tasks=160, maximum-initialized-active-tasks-per-user=80	1262	1302	-

5.1.4 Discussion

Careful selection of scheduler configuration parameters is crucial to reduce the execution time of jobs in an environment where the Capacity Scheduler is used. The different values for Capacity Scheduler configuration parameters may have different impacts on the performance of the running jobs in the cluster as shown by the experiments conducted. The Capacity Scheduler has been around since 2009 and is used in Yahoo! clusters, but most Hadoop users and administrators do not know the precise meaning of the parameters and the kind of impact they can have on the execution time of the running jobs. A number of queries have been asked on Hadoop forums[1] regarding this issue. Finding the performance impact can be troublesome as well as time consuming. There is need for a tool which can not only help them to identify the performance of jobs after changing certain settings but also help them to find the optimal values of task scheduler configuration settings for a given cluster configuration and a set of jobs.

Experiments on the local cluster show that each single parameter present in Capacity Scheduler makes an impact on the performance of MapReduce applications, depending on the scenario. For example, *User-Limit-Factor* does not impact the performance of the jobs in an equal queue capacity scenario. However, with a 90-10% queue capacity scenario, the User-Limit-Factor shows improvement for the smaller capacity queue. Increasing the number of queues leads to an increase in the execution time of all the jobs almost by a factor of 2. In the Differential queue capacity scenario, the performance of smaller capacity queue jobs is worse than for the higher capacity queue. Maximum-Capacity limits a queue from using resources from other queues. It is beneficial if the queues in the system have continuous inflow of jobs and their allocated capacity does not remain unused. Minimum-User-Limit-Percent lowers the waiting time for jobs at the expense of increased execution time. Higher priority of the jobs allows them to execute and finish sooner than lower priority jobs in the queue. Job Initialization Parameters (Maximum-System-Jobs, Maximum-Initialized-Active-Tasks and Maximum-Initialized-Active-Tasks-Per-User) controls job admissibility into the system and controls concurrency. Lower concurrency leads to fewer jobs running in parallel. This leads to high waiting time, but faster execution time for the jobs arriving later in the queue.

5.2 Simulator Validation

The goal of the experiments in this section was to check the accuracy of the simulator against the real cluster results. The expectation was to get similar trends between real world and simulator results. From the results, the simulator was not numerically 100% accurate due to several different factors:

- Presence of straggler tasks in real world experiments. The simulator does not model the stragglers. Only map straggler tasks are shown in all the conducted experiments.

[1]http://lucene.472066.n3.nabble.com/Hadoop-lucene-users-f647590.html

- The simulator does not model all the different sub-phases of the map/reduce phase, which leads to some inaccuracy.
- Same-reduce-node effect as described in Section 4.2.1.

The above mentioned factors hindered the accuracy of the simulator and shall be described in detail in later sections of this chapter. Even so, the results do confirm the accuracy of the simulator in many other aspects.

The metrics observed in validation were the map time, the reduce time, the execution time and the waiting time for each job. Each real cluster experimentation was repeated 10 times. The graphs shows the average of 10 runs and the standard deviation among the runs. The trends were also compared.

5.2.1 Comparison of Results with one Reduce Slot per Node

This section shows the comparison of experimental results between the real cluster and the simulator on one reduce slot per node setup.

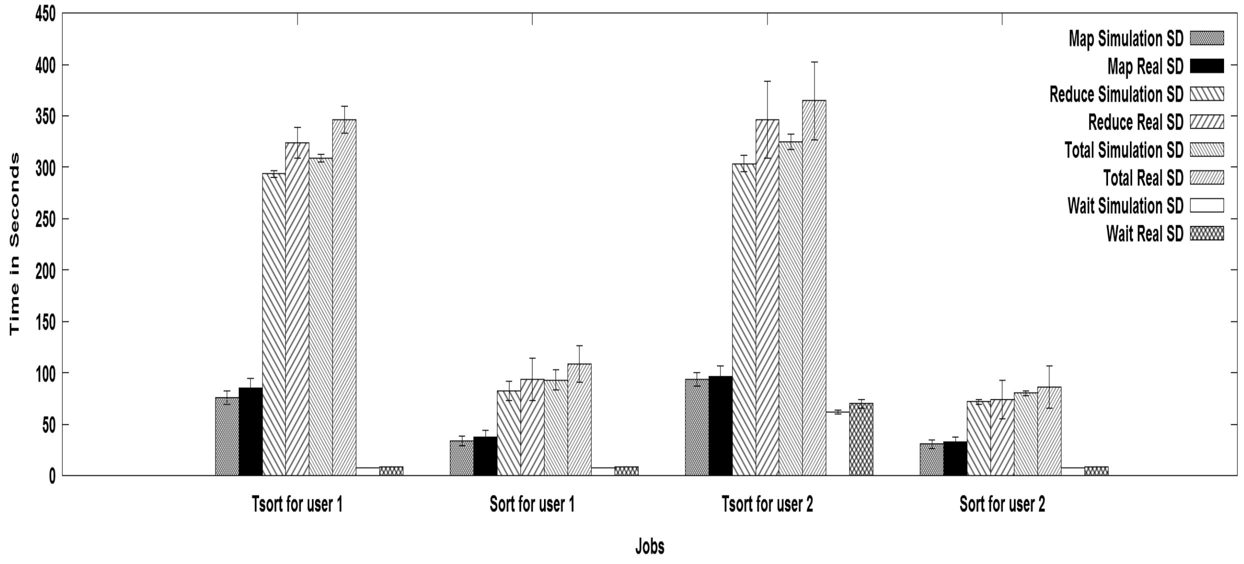
Effect of changing the *NumberofQueues*

In this first experiment, a single queue with 2 users and 4 jobs was used. In the second experiment, 2 queues with 2 users and 4 jobs each was used. The capacity of each queue was set at 50%. The results for both real cluster and simulator are shown in Figure 5.8.

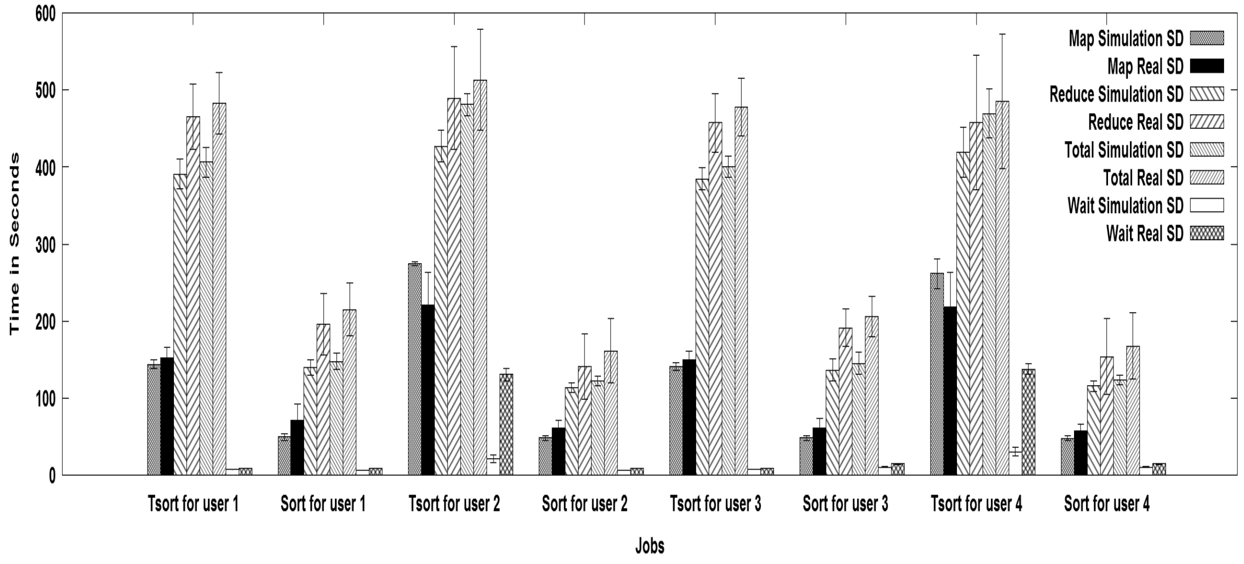
Differences exist between the simulator and the real cluster results. The real cluster results have a large standard deviation when there are 2 queues. This phenomenon shall be observed in all the results presented in this section.

The results for a single queue matched well for both the simulator and the real cluster experiments. There is a small standard deviation in single queue experiments. The reason for this was the presence of straggler tasks in some of the job's runs. This lengthens the total time for a MapReduce job to complete. More information on stragglers and their occurrences is provided at the end of the chapter.

As the number of queues increased from 1 to 2, the map, reduce and total execution time increases for all the jobs. This happened due to less cluster capacity allocated to each queue and a greater number of jobs present in the system. The standard deviation for the map execution time for user 2's TSort and user 4's TSort was high. The reason for this was the delayed map execution. This effect occurred only once for user 2's TSort and user 4's TSort in 10 runs. TSort for user 2 and 4 consistently exhibited delayed map execution behaviour in the simulator except one time. Hence, one can see marked differences in behaviour between real and simulator results for user 2's TSort and user 4's TSort. This also leads to huge differences in waiting time results for both the jobs. If the simulator would have consistently displayed non-delayed map execution behaviour, the trends would have been similar to real cluster results. The difference between the simulator and real cluster while exhibiting delayed map execution behaviour can be attributed to the timings in which



(a) 1 queue 100% capacity Cluster



(b) 2 queue 50% capacity Cluster

Figure 5.8: Effect Of *NumberofQueues* (2 Users/Queue and 2 Jobs/User)

the JobTracker gets the heartbeats from the DataNodes. The other major factor for the significant standard deviation for TSort was the straggler tasks in some of its runs (23 to be precise).

Delayed map execution also leads to increased reduce time, which causes significant standard deviation in reduce execution time results. The reduce task for a job is started in MapReduce when a certain number of map tasks are completed. By default, it is 0.05% of the total map tasks of a job and is called default_completed_maps_percent_for_reduce_slowstart. Thus, *Sort* with 1 GB of input data will have 16 map tasks and it will take just one map task to complete to trigger the reduce phase for it. Consider a case where two jobs are submitted to the system with small submission time difference between them and *Sort* is the second job with 16 map tasks. Initiation of the reduce phase in *Sort* after one map task completes will take a reduce slot from the cluster. As it was not the turn of job *Sort* to get the map slots before the first job is finished, the *Sort* job will not get any further map slots until the first job finishes its map tasks. Such a scenario not only leads to prolonged reduce execution time, but also to wastage of resources (reduce slot). Straggler map tasks yet again affect the reduce execution of all jobs, which leads to a significant standard deviation.

The differences in timings of the results obtained between the real and simulator can be ascribed to 2 points:

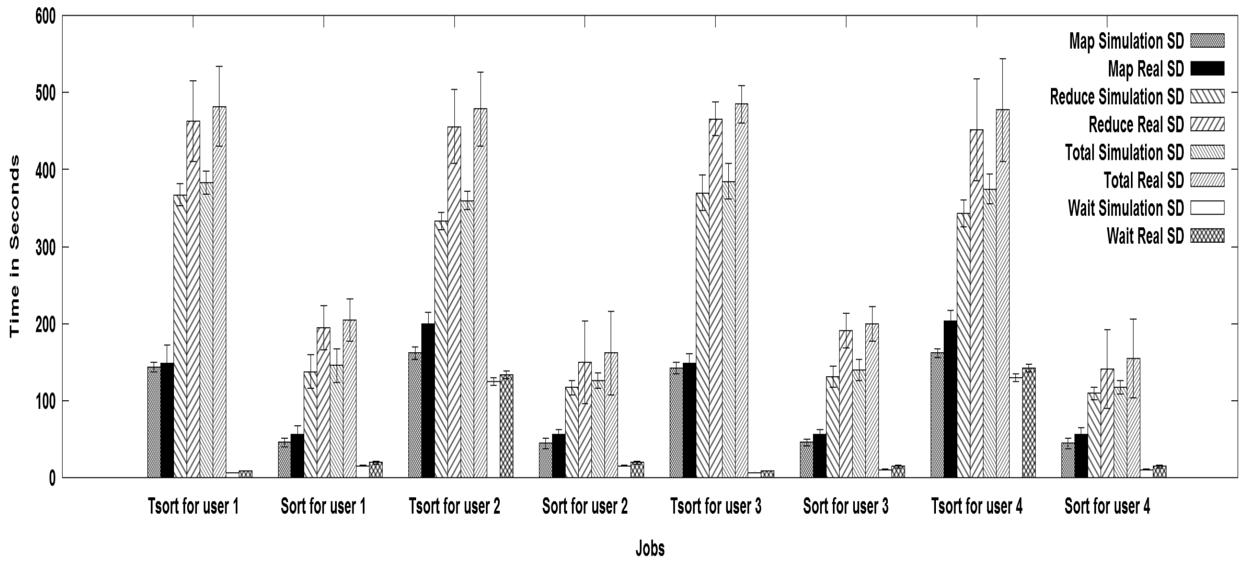
- Straggler tasks. The simulator does not model the straggler tasks.

- Some of the phases of map and reduce tasks are not present or modeled in the simulator. For example, the *collect* sub phase from the map phase is not present in the MRPERF simulator. During this phase, the intermediate (map-output) data is partitioned and collected into a buffer before spilling. MapReduce consists of multiple sub-phases. Hence, omitting any sub-phase leads to errors in the model and simulator prediction results.

Effect of changing the *Maximum-Capacity*

In this experiment, the max capacity of the queue was changed. The capacity and Maximum-Capacity of each queue was set at 50%. The results for both real cluster and simulator are shown in Figure 5.9.

The high standard deviation for map/reduce tasks is due to the presence of stragglers. The delayed map execution effect was not present in any of the runs in the real cluster or simulator results. The map/reduce timings for all the jobs remained the same or improved from earlier experiments done in section 5.2.1. Placing a limit on *Maximum-Capacity* of the queue prevents each queue from stealing resources from the other queue's quota. It also helps in removing Delayed map execution behaviour from the system. This led to improved map and reduce timings for user 2's TSort and user 4's TSort.



(a) 2 queues 50% capacity and max cap=50 % Cluster

Figure 5.9: Effect Of *Maximum-Capacity* (2 Users/Queue And 2 Jobs/User)

Effect of changing the *Minimum-User-Limit-Percent*

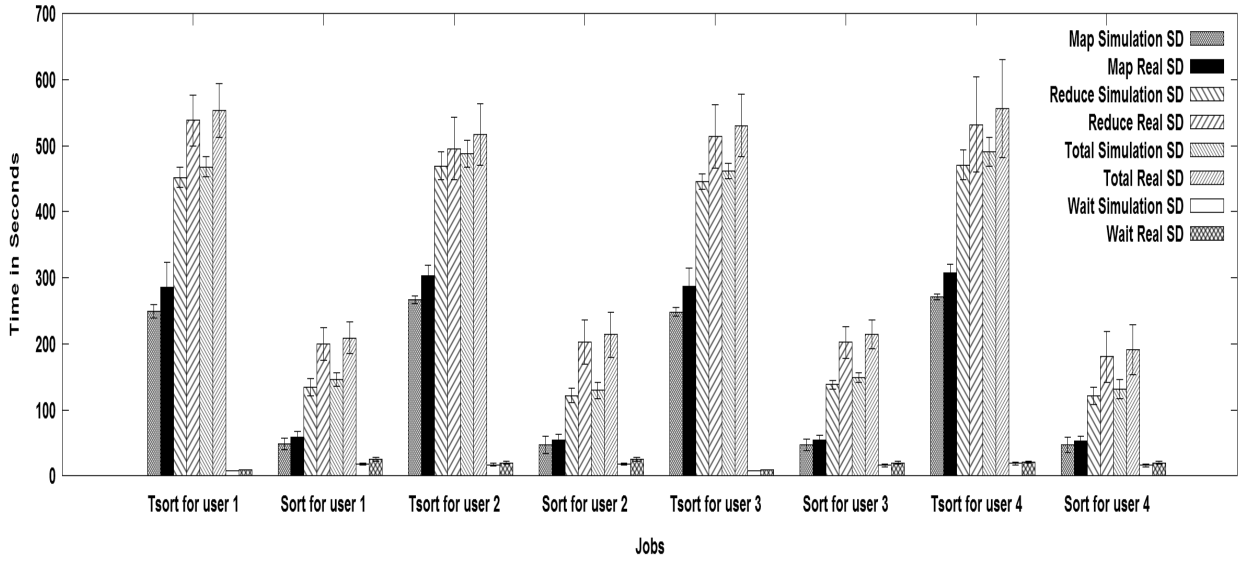
In this experiment, the Minimum-User-Limit-Percent of the queue was changed. The Minimum-User-Limit-Percent was changed to 50% instead of 100%. The results for both real cluster and simulator are shown in Figure 5.10.

Again, the high standard deviation for map/reduce tasks in the real cluster is due to the presence of stragglers. Changing the Minimum-User-Limit-Percent to 50% means the user's job can concurrently execute their map tasks and does not have to wait for the other user's job (map tasks) prior to it to finish. As a result, the map times for all the jobs increased. This also leads to increase in the reduce and overall execution time for the jobs. This trend was perfectly captured by the simulator as well.

Effect of changing the *Maximum-System-Jobs*

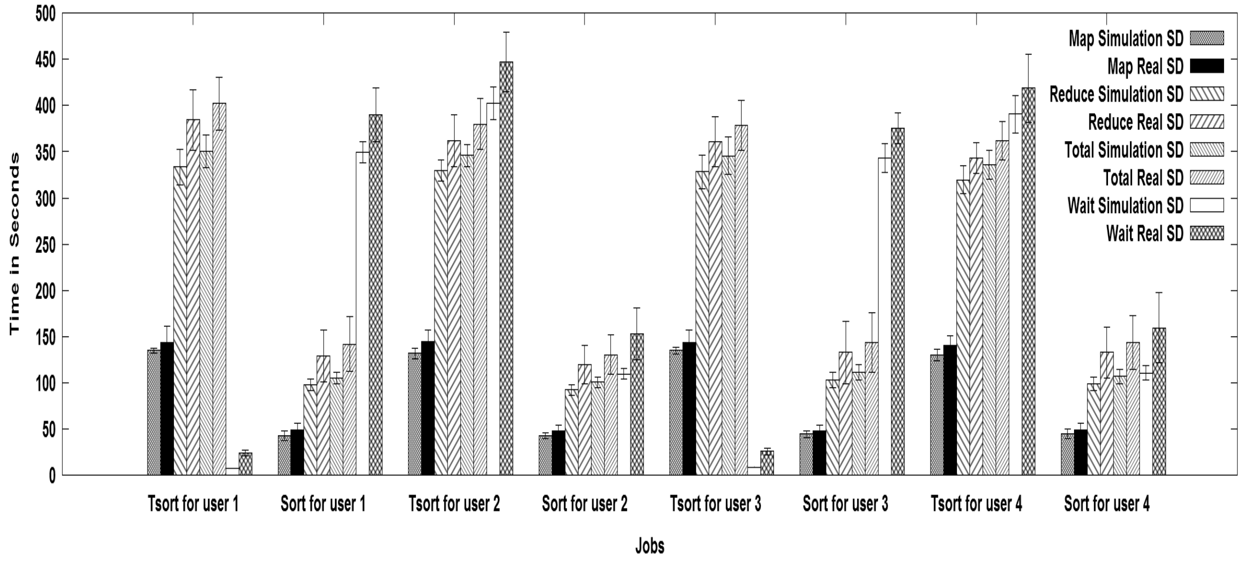
In this experiment, the maximum number of jobs that can be executed in parallel on the system was reduced to 2. The results for both real cluster and simulator are shown in Figure 5.11. No more than 1 job/queue was executing in parallel in either the real cluster or the simulator. This leads to huge waiting times for all the jobs after the first job in their respective queues.

In conclusion, it can be seen that the simulator depicts the trends well compared to the real cluster experiments. The accuracy of the simulator varied between 60% to 90% for map, reduce, execution and elapsed times. It also depicts all behaviour among different scenarios that were seen on the real cluster experiments.



(a) 2 queues 50% capacity and Minimum-User-Limit-Percent=50% Cluster

Figure 5.10: Effect of *Minimum-User-Limit-Percent* (2 Users/Queue and 2 Jobs/User)



(a) 2 queues Maximum-System-Jobs = 2 in Cluster

Figure 5.11: Effect Of *Maximum-System-Jobs* (2 Users/Queue And 2 Jobs/User)

5.2.2 Two Reduce Slots per Node

This section shows the comparison of experimental results between the real cluster and the simulator, when the number of reduce slots per node was increased from 1 to 2. As the number of reduce slots was increased to 2 on each node, some of the jobs' reduce tasks were simultaneously executed on the same node. Hence, in this section, 2 different graphs are shown for the same jobs. The one shows the jobs statistics for run-time when its reduce phase executes in isolation. The other one shows the jobs run-time statistics when it is scheduled with another job on the same node, but on different reduce slot. Tables 5.2 and 5.3 show the number of isolated-paired reduce slot runs for the real cluster and simulation respectively across different experiments. The discrepancy between the simulator and real cluster behaviour can be attributed to the timings in which JobTracker gets the heartbeat from the DataNodes.

Table 5.2: Isolated-Paired Reduce Runs For All Jobs In Real Cluster Experiments

Experiment	**TSort 1**	**Sort 1**	**TSort 2**	**Sort 2**	**TSort 3**	**Sort 3**	**TSort 4**	**Sort 4**
2 queue changing capacity	10-0	6-4	5-5	8-2	7-3	7-3	7-3	7-3
2 queue Maximum capacity	9-1	6-4	7-3	8-2	8-2	7-3	5-5	9-1
2 queue Minimum-User-Limit-Percent	6-4	8-2	7-3	7-3	6-4	9-1	7-3	9-1
2 queue Maximum-System-Jobs	9-1	10-0	9-1	9-1	9-1	10-0	9-1	9-1

Table 5.3: Isolated-Paired Reduce Runs For All Jobs In Simulation Experiments

Experiment	**TSort 1**	**Sort 1**	**TSort 2**	**Sort 2**	**TSort 3**	**Sort 3**	**TSort 4**	**Sort 4**
2 queue changing capacity	2-8	4-6	3-7	7-3	3-7	5-5	5-5	4-6
2 queue Maximum capacity	2-8	3-7	7-3	4-6	3-7	6-4	4-6	4-6
2 queue Minimum-User-Limit-Percent	4-6	3-7	3-7	5-5	2-8	4-6	4-6	6-4
2 queue Maximum-System-Jobs	5-5	6-4	5-5	5-5	5-5	5-5	6-4	6-4

Effect of changing the *NumberofQueues*

Two experiments were done to examine the effect of changing the number of queues in the experimental setup. In this first experiment, a single queue with 2 users and 4 jobs in each queue was used. In the second experiment, 2 queues with 2 users and 4 jobs in each queue were used. The capacity of each queue was set at 50%. The results for both real cluster and simulator are shown in Figure 5.12.

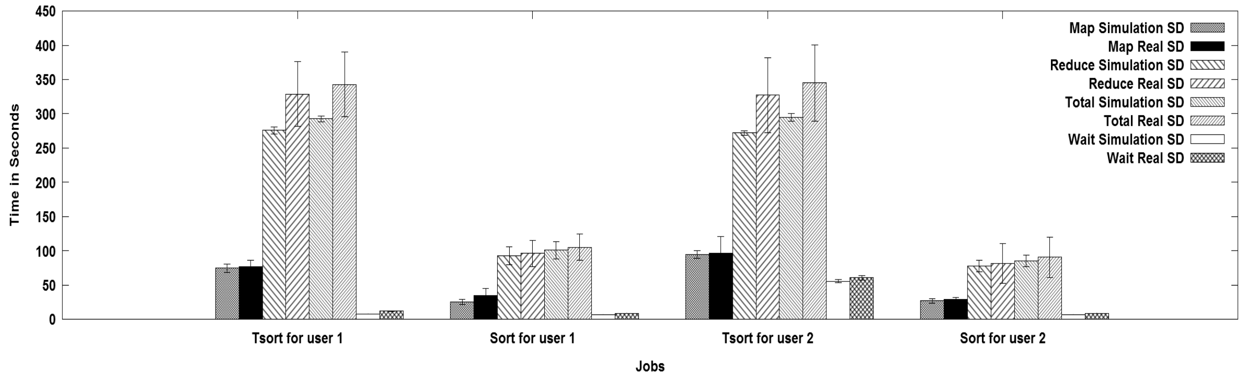
Differences exist between the simulator and the real cluster results. The real cluster results have a large standard deviation when there are 2 queues. This phenomenon was observed in all the results presented.

The results with a single queue experiments matched well for both the simulator and the real cluster. A small standard deviation exists in single queue experiments, caused by straggler tasks in some of the real cluster job runs. With an increasing number of queues, the map, reduce and total execution time increases for all the jobs. Delayed map execution happened once in the real cluster experiments for TSort 2 and TSort 4. Delayed map execution occurred in all the simulation runs. This explains the significant differences in the standard deviation as well for TSort 2 and TSort 4 between the real and simulation runs. The waiting times for TSort 2 and TSort 4 were markedly different because of delayed map execution. In the real cluster setup, TSort 1 and TSort 4 started when the map phase for TSort 2 and TSort 3 ended. However, in the simulation runs, they started their map execution as soon as they were submitted.

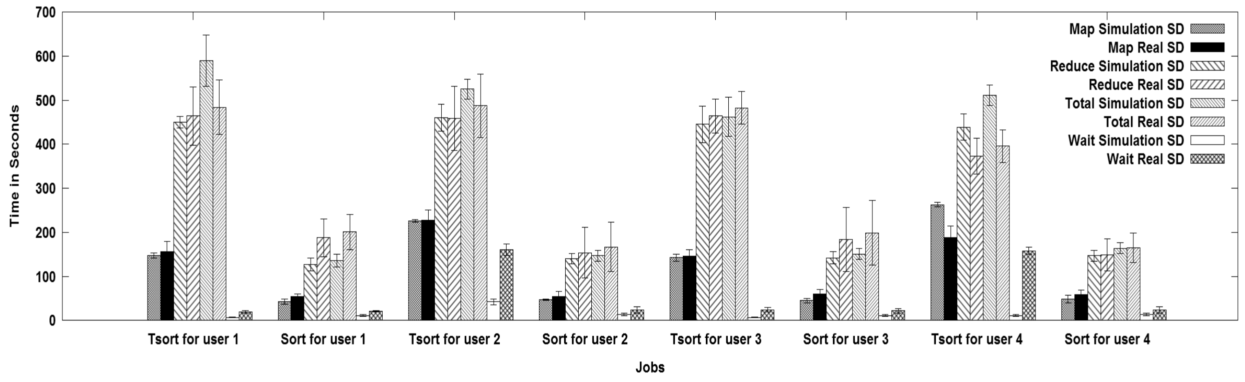
The increase in the reduce execution time in Figure 5.12 (c) is mainly attributed to the same-reduce-node effect. This effect also leads to bigger standard deviation on various occasions influencing each job differently. It depends on two major factors:

- Which job executes in parallel with which job. For example, in the real cluster results (refer to Figure 5.12 (c)), TSort 3 in particular was always paired with another TSort job and as TSort jobs execute longer, the reduce execution time for TSort 3 is much higher than all other jobs. The presence of stragglers also make it worst for TSort 3. All other TSort jobs were paired with TSort as well as *Sort* jobs during their runs, which make their reduce execution time less than TSort 3.

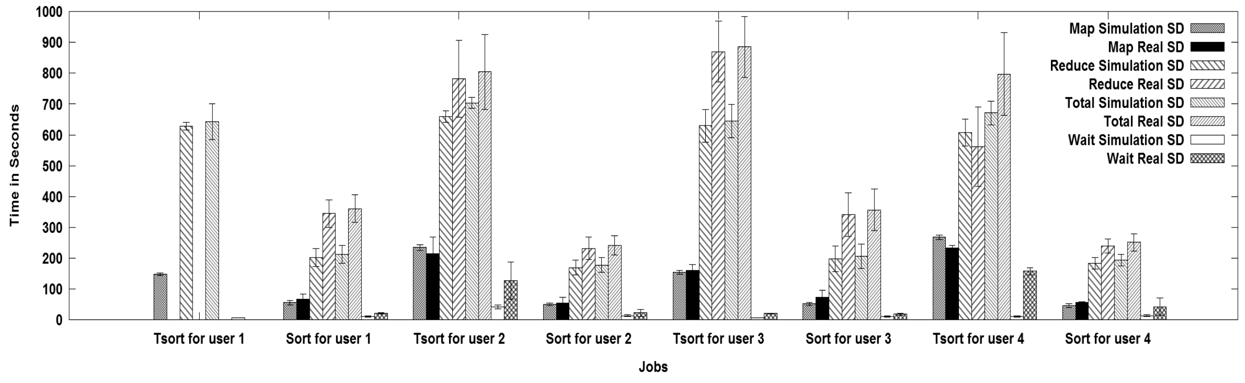
- When the jobs pair with each other. For instance, if two jobs start their reduce phase on the same node at the same time, then they will be affected much more than the two jobs if they are scheduled at different times. Consider a scenario, where the first *Sort* job starts executing its reduce phase on slot 0 of node 1 at time instance 0 and it takes 100 seconds to execute the reduce phase. Later, at time instant 5, another *Sort* job starts execution on node 1, slot 1. The two jobs started their reduce execution almost simultaneously, the reduce execution time will be almost twice (200 seconds) for both jobs (considering that the two jobs get a share of disk and network resources equally divided all the time). In a similar situation, if the second *Sort* job starts reduce execution phase at time 80, the first job's reduce execution time gets affected by 20%. This phenomenon can also be seen in the results. For job *Sort* 1, the reduce node effect was severe as it was scheduled almost simultaneously with other jobs on the same node. This was true for job *Sort* 3 also. However, in the simulation runs for the same jobs,



(a) 1 queue 100% capacity Cluster



(b) 2 queue 50% capacity Cluster - isolated reduce phase



(c) 2 queue 50% capacity Cluster - Paired reduce phases

Figure 5.12: Effect Of *NumberofQueues* (2 Users/Queue And 2 Jobs/User)

their reduce phase time does not overlap fully with the reduce phase time of other jobs (there was a small overlap). Such events caused the accuracy of results for these two jobs to not match accordingly with the real cluster results. The discrepancy between the simulator and real cluster behaviour can be attributed to the timings in which JobTracker get the heartbeats from the DataNodes.

The simulation waiting times can be higher than the real cluster waiting times for some jobs. This can be explained with the help of an example. Consider a situation where there are three jobs submitted to the system with job 1 in queue 1, job 2 in queue 2 and job 3 in queue 1. They execute in a FIFO basis if they are in the same queue. Consider scenario 1, where job 1 and 2 are both victims of same-reduce-node effect and hence take longer than usual to execute. In such a case, job 3's waiting time will be high. In scenario 2, job 1 and job 2 execute their reduce phase on different nodes and hence job 3's waiting time will be less than scenario 1, as job 1 will finish its execution on time. If scenario 1 happens in simulation more often for a job and scenario 2 happens more for that same job in a real cluster, then waiting time for the job will be higher in simulation than real cluster results.

Effect of changing the ***Maximum-Capacity***

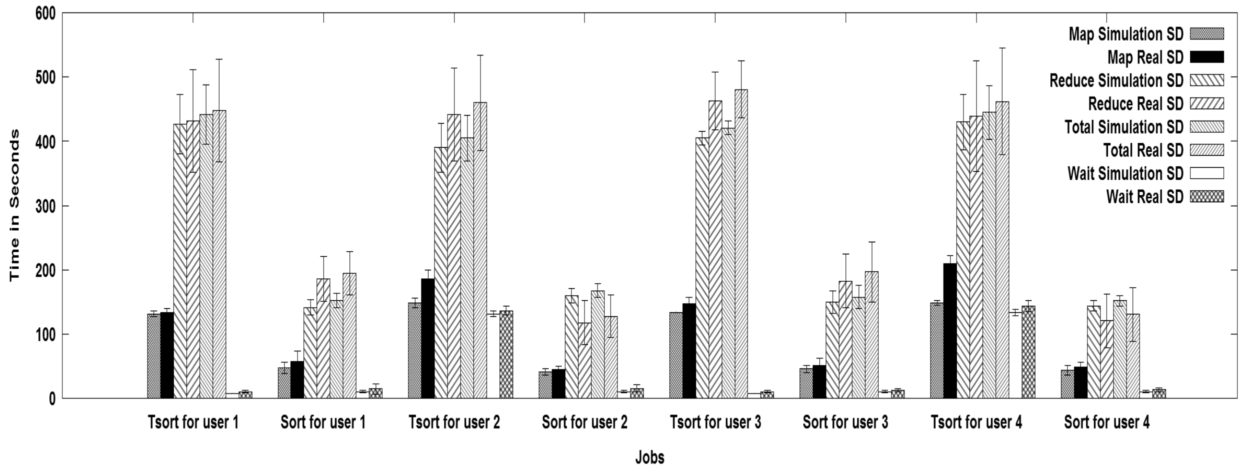
In the next experiment, the capacity and Maximum-Capacity of each queue was set at 50%. The results are shown in Figure 5.13.

The high standard deviation for map/reduce tasks is due to the presence of stragglers for isolated reduce cases. The Delayed map execution effect was not present in any of the runs in real cluster or simulator results. The map/reduce timings for all the jobs remained the same or improved from earlier experiment done in section 6.3.1 for isolated reduce cases. The high standard deviation for reduce tasks in paired reduce phase is due to the same-reduce-node effect for paired reduce cases.

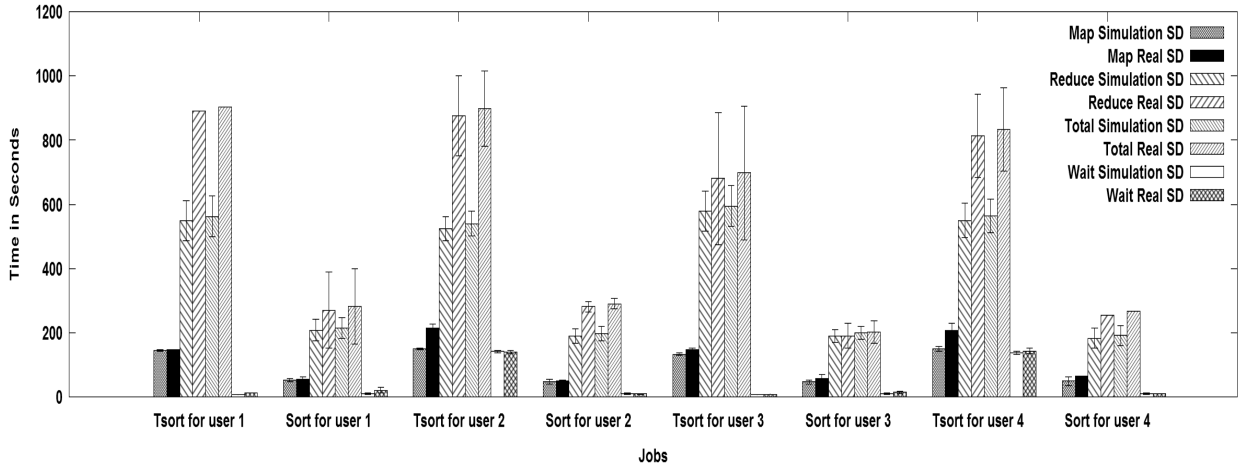
Effect of changing the ***User-Limit-Percent***

In this experiment, 2 values were used for Minimum-User-Limit-Percent: 50% and 100%. The results are shown in Figure 5.14.

The high standard deviation for map/reduce tasks is due to the presence of stragglers for isolated reduce cases. The reason of stragglers is the same as was present in the one reduce slot per node experiments. Changing the Minimum-User-Limit-Percent to 50% means the users can concurrently execute their map tasks. As a result, the map times for all the jobs increased. This also leads to an increase in the reduce time and overall execution time for the jobs. This trend was perfectly captured by the simulator as well. The high standard deviation for reduce tasks in paired reduce phase is due to the same-reduce-node effect for paired reduce cases.

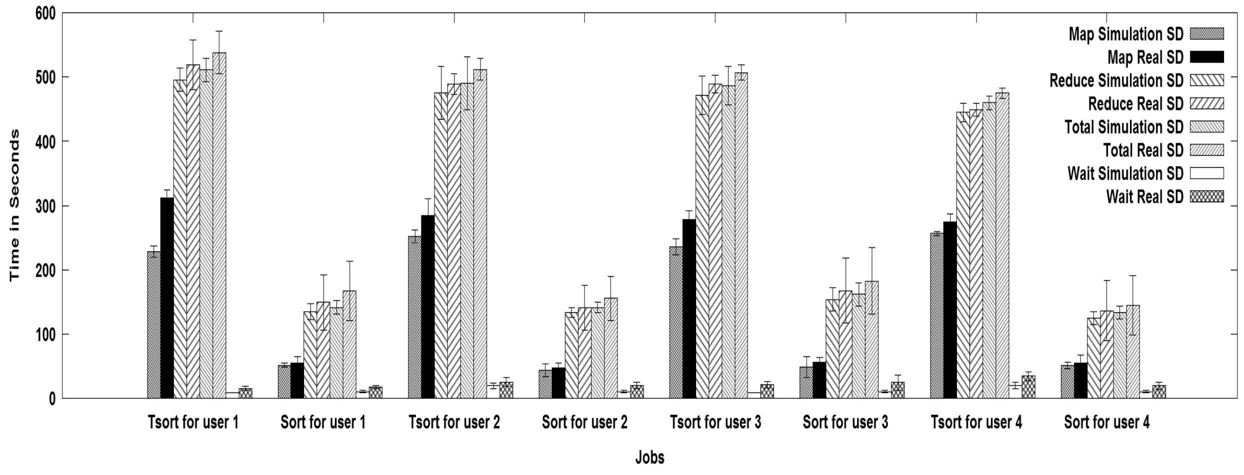


(a) 2 queues 50% capacity and max cap=50% Cluster - Isolated reduce phases

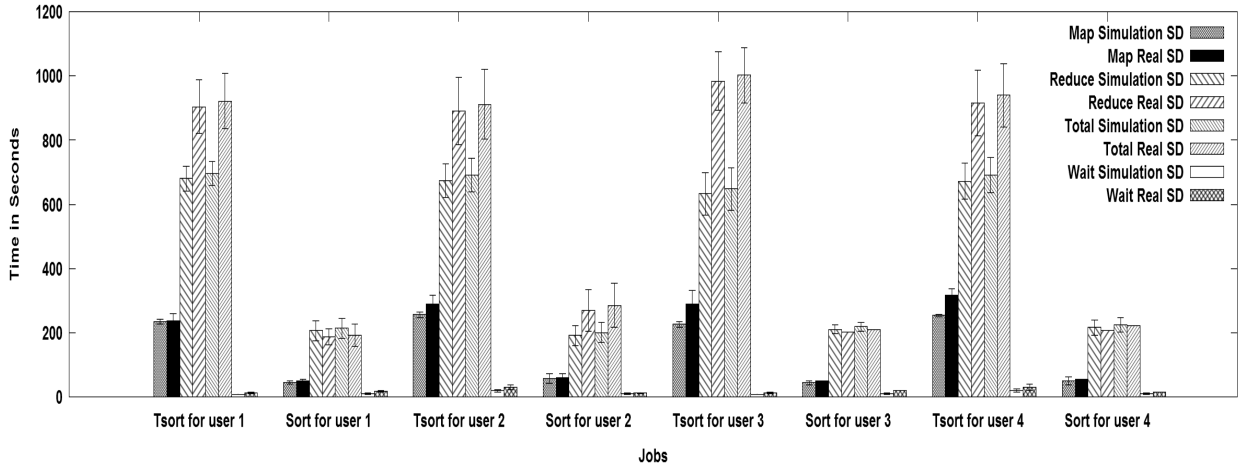


(b) 2 queues 50% capacity and max cap=50% Cluster - Paired reduce phases

Figure 5.13: Effect Of *Maximum-Capacity* (2 Users/Queue And 2 Jobs/User)



(a) 2 queues 50% capacity and Minimum-User-Limit-Percent=50% Cluster - isolated reduce phase

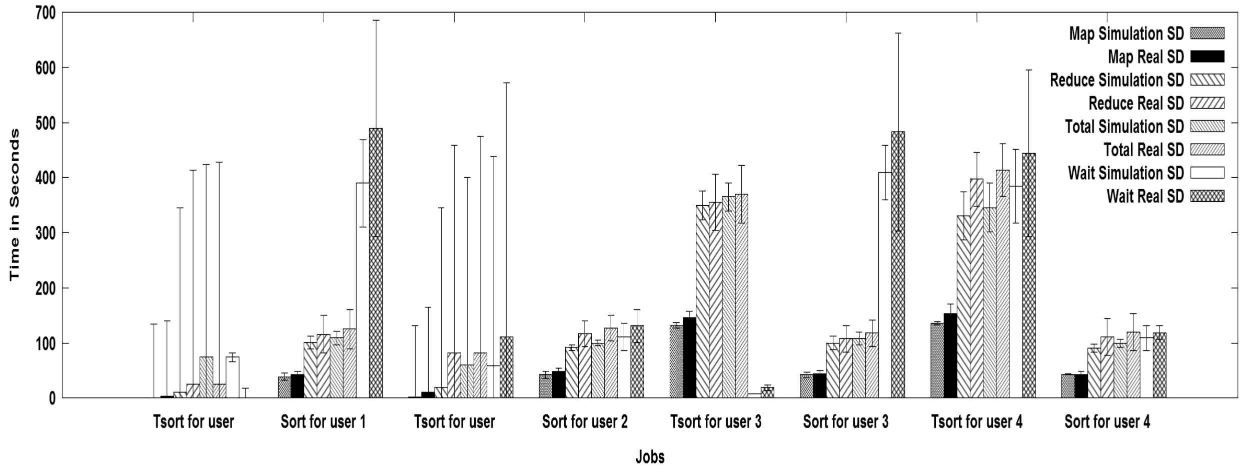


(b) 2 queues 50% capacity and Minimum-User-Limit-Percent=50% Cluster - paired reduce phase

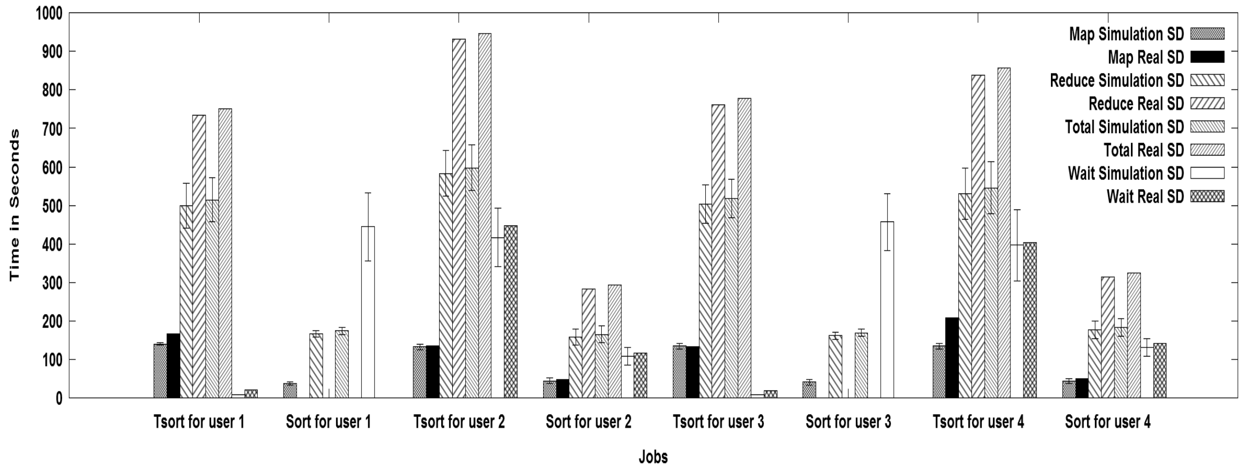
Figure 5.14: Effect Of *Minimum-User-Limit-Percent* (2 Users/Queue and 2 Jobs/User)

Effect of changing the *Maximum-System-Jobs*

In this experiment, the maximum number of jobs that can be executed in parallel on the system was reduced to 2. The results for both real cluster and simulator are shown in Figure 5.15.



(a) 2 queues Maximum-System-Jobs = 2 in Cluster - isolated reduce phase



(b) 2 queues Maximum-System-Jobs = 2 in Cluster - paired reduce phase

Figure 5.15: Effect Of *Maximum-System-Jobs* (2 Users/Queue and 2 Jobs/User)

No more than 1 job/queue was executing in parallel in either the real cluster or the simulator. It leads to huge waiting times for all the jobs after the first job in their respective queues.

5.2.3 Stragglers

This section discusses the reasons as to why stragglers were observed in the simulator validation experiments. Table 5.4 shows the number of map stragglers for both one and two reduce slots experimental scenario. Stragglers are often caused by hardware issues and software misconfiguration [2]. In the experiments done here, it appears likely that they are caused by resource contention as the system logs did not show any traces of hardware or software issue. The results from Table 5.4 point to that cause. It is to be noted that with more queues and jobs running in parallel, the number of stragglers goes up.

Table 5.4: Stragglers For All Experiments

Experiment Description	**No. of Stragglers in single reduce slot**	**No. of Stragglers in double reduce slot**
1 queue	7	4
2 queue changing capacity	23	27
2 queue Maximum-Capacity	23	24
2 queue Minimum-User-Limit-Percent	29	32
2 queue Maximum-System-Jobs	8	10

5.3 Analysis/Summary

In conclusion, it can be seen that the simulator showed similar trends as the real cluster experiments. The accuracy of the simulator varied between 60% to 90% for map, reduce, execution and elapsed times. It also depicts all behaviour among different scenarios that were seen on the real cluster experiments. However, the following reasons did not allow the simulator to achieve 100% accuracy are the following:

- Presence of straggler tasks in real world experiments. The simulator does not model the stragglers.
- The simulator does not model all the different sub-phases of the map/reduce phase, which leads to some inaccuracy in the prediction mechanism of the simulator.
- The same-reduce-node effect leads to high variability in results. The jobs are affected by the same-reduce-node effect in two different manners: Which job executes in parallel with which job, and when the jobs pair with each other.
- Some other factors like disk access time, CPU and network contention, prefetching etc. cannot be modeled precisely in a simulator model.

The simulator does a decent job of showing different trends among different experiments and this increases the confidence in the simulator. Through the simulator validation, some unexpected insights into the Capacity

Scheduler behaviour were discovered that can help system administrators. Under default parameter settings, some jobs can be victims of delayed map execution. In order to solve this issue, *Maximum-Capacity* of each queue should be set to the capacity of the queue. Other findings show that the same-reduce-node effect can deteriorate the execution time of jobs. This hurts even more when there are completely idle nodes in the cluster and still the new job is scheduled on an already busy node.

Chapter 6

Characterization of the Effect of Capacity Scheduler Parameter Settings using Simulation

This chapter contains the simulation results conducted on a cluster of 31 simulated nodes with different job submission patterns and provides more insights into the interaction of different parameters of the Capacity Scheduler under different job submission patterns. The chapter also provides a few important findings about the Capacity Scheduler settings, which can be used by Hadoop administrators to manage the running of jobs in their cluster efficiently. In all the experiments done in this chapter, the focus was on three important metrics: data locality, response ratio and Makespan. The variability and execution times for particular types of jobs is of interest as well.

In the following sections, Section 6.1 and Section 6.2 cover the experiments for a job submission pattern where a long sequence of large jobs is followed by a long sequence of short jobs. Section 6.3 covers the analysis of interleaved job submission pattern. In the interleaved job submission pattern, 4 short jobs were submitted and then a long job followed by the same pattern. In Section 6.4, the separate queue job submission pattern is discussed. Short and long jobs were given separate queues in separate queue scenario.

The Fair Share Scheduler was specifically designed to improve the performance of short jobs in presence of long jobs. In Section 6.1 and Section 6.2, experiments were done to see the performance of short jobs and how the Capacity Scheduler parameters interact with each other. The main motive of these experiments was to see under which Capacity Scheduler settings, the performance of the short jobs improves in terms of response ratio. A more extensive description about the job types, and factors varied are presented in Chapter 4. Experiments were done with 2 queues (Queue 1 and Queue 2).

6.1 Equal Queue Capacity: Job Types Separated

6.1.1 Data Locality

Data locality plays an important role in MapReduce environments. When more node-local tasks are scheduled over rack-local and remote tasks, performance of the jobs should be improved because of the reduced network activity. The experiments in the simulation were done on a single rack and hence there were no remote tasks. The observed metric was the percentage of rack-local tasks scheduled across different experiments.

The results shows that Data locality increases with the increasing map tasks for a job. Table 6.1 shows the percentage of rack-local tasks scheduled in different experiments across different job types for Queue 1.

Table 6.1: Percentage Of Rack-Local Tasks For All Job Types Across The Experiments

Job Type	**Exp 1**	**Exp 2**	**Exp 3**	**Exp 4**	**Exp 5**	**Exp 6**	**Exp 7**	**Exp 8**
1	11.25	11	10.75	11.92	11.3	13.6	12.2	13.1
2	5.0	5.0	5.0	6.1	5.2	6.7	6.4	6.2
3	2.4	2.63	2.4	2.8	2.7	3.6	3.2	3.0
4	1.21	1.0	1.2	1.5	1.0	1.9	2.0	2.4
5	91	80	94	88	92	88	90	88
6	87	75	85	83	90	82	78	82
7	61	53	61	56	56	58	56	59

The observed statistics were similar for Queue 2 and hence are not shown. For small jobs (Job Type 5, 6, 7), the percentage of total map tasks scheduled which are rack-local is very large. More rack-local tasks means less data locality (less node-local tasks). For small jobs with 1 map task, the distribution was bi-modal (either one of its map tasks is scheduled rack-local or none). Similarly, for small jobs with 2 map Tasks, the distribution was tri-modal, varying between 0, 50 and 100 percent. For large jobs, the percentage of rack-local tasks is low, which indicates high data locality. This shows that the Capacity Scheduler favors data locality for tasks having large number of maps. The main reason for this is the fact that as a job has more map tasks, the data is more widely distributed across the cluster. This makes it easy for the JobTracker to assign job to the nodes where the job's map input data is locally present in the disk.

The delay scheduling present in Capacity Scheduler does not help in improving data locality for the jobs. This is because delay scheduling comes into picture when remote tasks are scheduled (not when rack-local task is scheduled). This severely defeats the purpose of having delay scheduling. Node-local and rack-local are considered equivalent tasks in Capacity Scheduler and hence delay scheduling is not used. This assumption negatively influences the performance of the jobs. A distinction must be made between node-local, rack-local and remote tasks. Adding an extra level of delay scheduling when it comes to making a decision for JobTracker to schedule a rack-local task may improve the data locality for jobs.

Changing settings for the Scheduler does not make a big difference in terms of data locality under the experiments done in this section, as shown in Table 6.1. The changes in parameter settings in Capacity Scheduler provides a stringent set of limits to ensure that a single job or user or queue cannot consume a disproportionate amount of resources in the cluster and ensures fair sharing of resources. Data locality depends on which TaskTracker gets the opportunity to execute a task and whether it has the data locally in its disk. The decision to give a task to a TaskTracker is determined by the JobTracker running the task scheduler algorithm, and is designed to be independent of what the Capacity Scheduler parameter settings are at any instant of time. The delay scheduling added to have more data locality for jobs in Capacity Scheduler is independent of such settings as well.

6.1.2 Response Ratio

Changing Minimum-User-Limit-Percent improves response ratio in all the cases (Experiment 4, 6, 7 and 8) for the long jobs. Table 6.2 shows the response ratios across different experiments. The number in the table represents the average response ratio for all the jobs taken together for a single job type. Setting Minimum-User-Limit-Percent allows more users to execute their jobs concurrently. Jobs don't have to wait longer. Hence, the waiting time for the jobs decreases, as shown in Figure 6.1. The lower value for response ratio in experiment 1 for Job Type 4 is an artifact of delayed map execution.

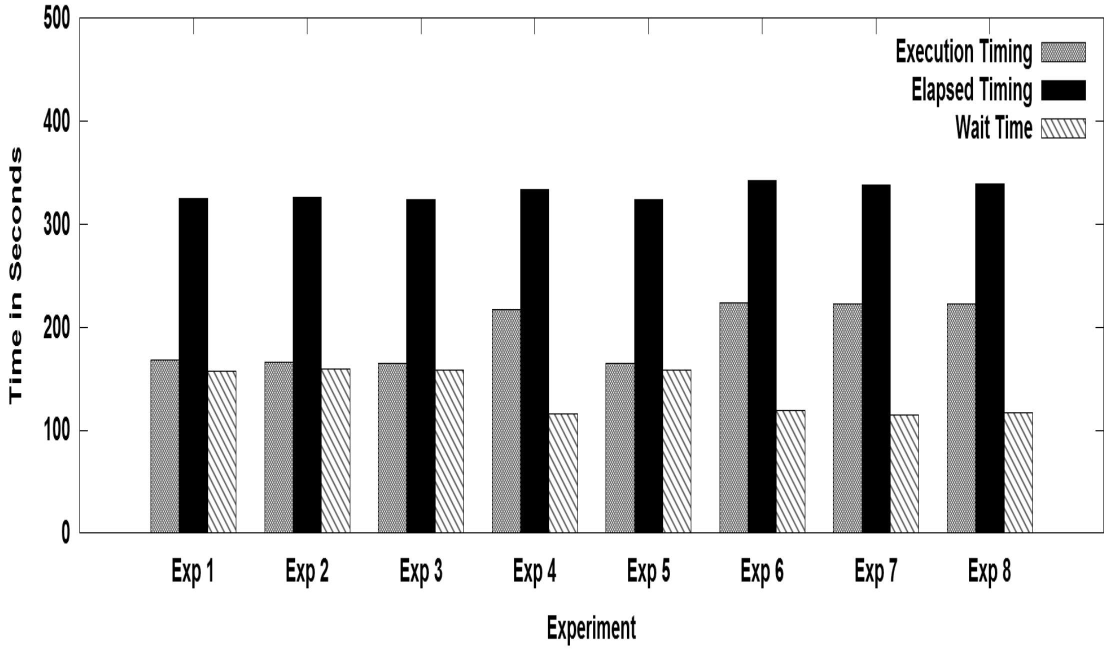
Table 6.2: Response Ratio For All Job Types For The Experiments

Job Type	**Exp 1**	**Exp 2**	**Exp 3**	**Exp 4**	**Exp 5**	**Exp 6**	**Exp 7**	**Exp 8**
1	1.2	1.21	1.2	1.01	1.21	1.01	1.01	1.01
2	1.93	1.96	1.96	1.53	1.96	1.53	1.51	1.52
3	1.24	1.51	1.51	1.14	1.51	1.15	1.14	1.14
4	2.1	6.70	6.7	1.54	6.75	1.54	1.54	1.54

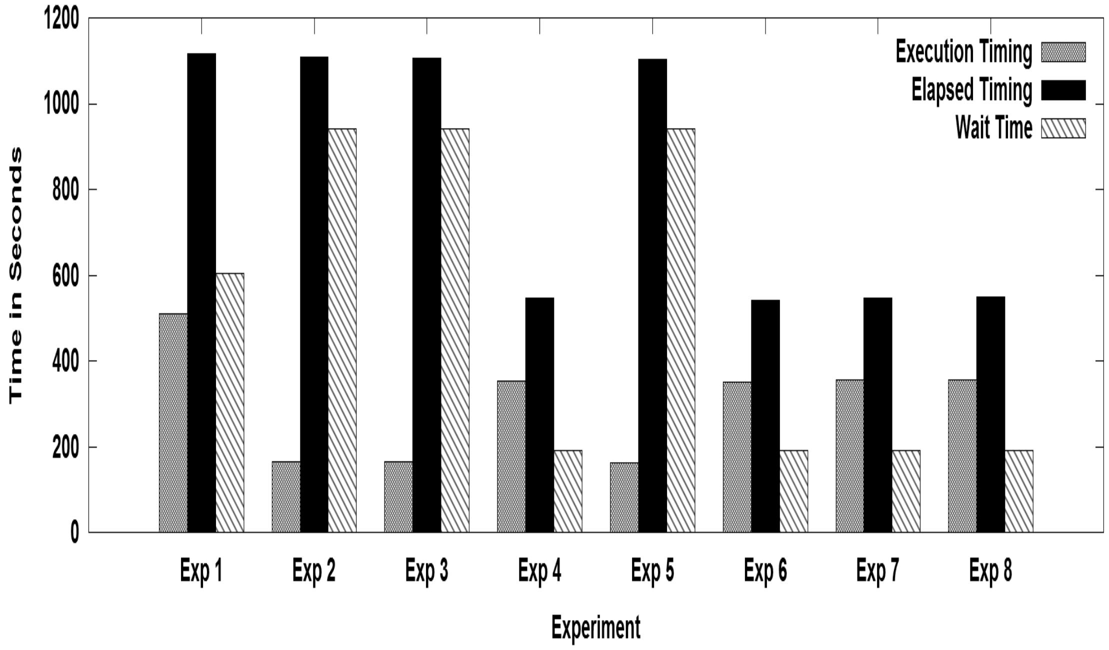
End users will always be more interested in the overall execution timings, along with elapsed timings for their jobs. As discussed, small values for the response ratio means lower waiting times for the jobs. However, the interesting question is: Does improving the response ratio also accompany improved execution time and elapsed time for the jobs? If not, then, improving response ratio does not mean anything for the jobs. Figure 6.1 give some insights.

The execution time increases for both the job types when the response ratio decreases (Minimum-User-Limit-Percent changes) because of increased competition among the jobs (lending to higher map and reduce timings and hence higher execution timings). The improvement in response ratio does not change elapsed timings for Job Type 2 (in fact, that increases slightly). However, for Job Type 4, elapsed time decreases significantly. The reason for this observation is the factor by which the response ratio improves for the two job types. Inspecting Table 6.2 tells that the response ratio changes marginally for Job Type 2. Hence, elapsed timings does not improve for it. The response ratio for Job Type 4 improves by a big factor and hence elapsed timings improved significantly for it.

There is another way to look at this phenomenon. Elapsed time is calculated as Execution time + Waiting time. Hence, a decrease in elapsed timing is possible under two conditions: if both the wait and execution



(a) Elapsed, Execution and Wait Timings for Job Type 2



(b) Elapsed, Execution and Wait Timings for Job Type 4

Figure 6.1: Elapsed, Execution And Wait Times For Job Types 2 And 4

time decrease for a job or one of the timing reduces significantly even if the other timing increases to offset the increase. More information can be obtained from Figure 6.1. Job Type 2 has earlier arrivals in the cluster and its waiting time does not improve much by changing Minimum-User-Limit-Percent. The little benefit it would have received from slightly decreased waiting time gets offset by the increase in the execution time. Job Type 4 arrives late in the system and hence has a very high waiting time under default settings. Changing Minimum-User-Limit-Percent for such jobs significantly improves their waiting time, which in turn leads to improved elapsed timings. The same logic applies for short job types.

Changing Minimum-User-Limit-Percent always improves the response ratio for the short job types too. Table 6.3 shows response ratio across different experiments.

Table 6.3: Response Ratio For Small Job Types For The Experiments

Job Type	**Exp 1**	**Exp 2**	**Exp 3**	**Exp 4**	**Exp 5**	**Exp 6**	**Exp 7**	**Exp 8**
5	15	15	14	3	14	3.6	3.2	3.5
6	12.8	12.9	12.79	3	12.7	3	2.7	3
7	10.3	10.4	10.3	1.8	10.5	1.83	1.82	1.93

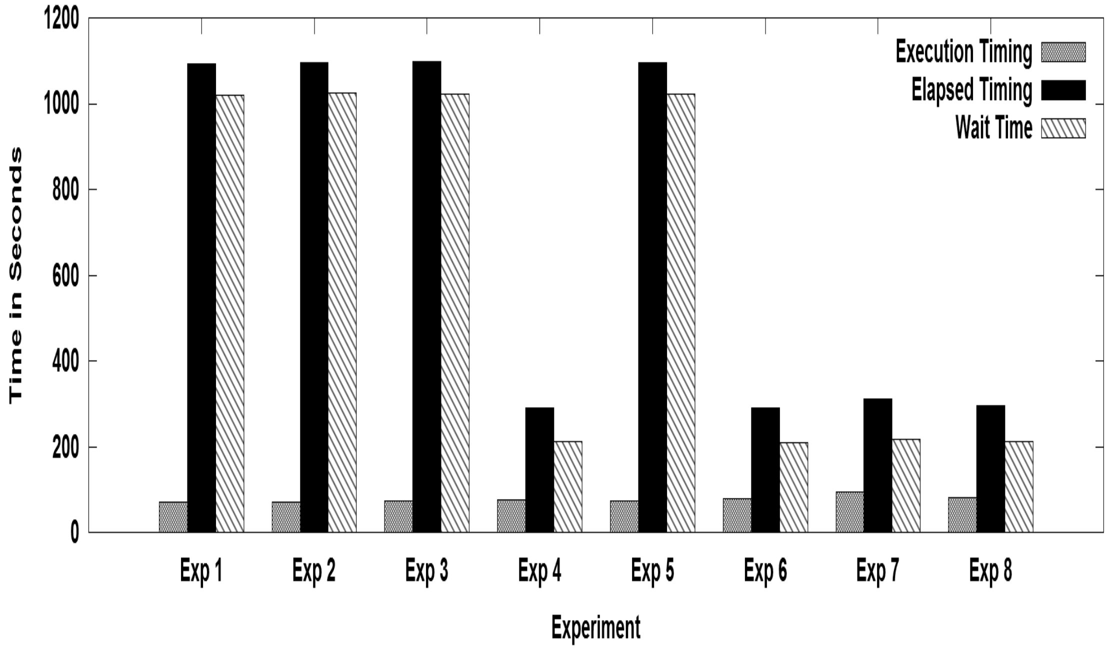
It can be seen that Minimum-User-Limit-Percent improves response ratio in all cases for the reasons explained earlier. Figure 6.2 gives some more insights. The execution time increases for both the job types 5 and 7, when response ratio improves, due to more competition between jobs. The improvement in response ratio significantly reduces the elapsed timings for Job Type 5 and 7 because of reduced waiting times. Inspecting Table 6.3 tells that the response ratio improves significantly for Job Type 5 and Job Type 7. However, as the improvement in elapsed timings also depends on the execution timing, the improvement is much higher for Job Type 5 than 7. The huge improvement in response ratio gets offset by the increased execution timings for Job Type 7. This discussion shows clearly that improving response ratio can bring varying benefits for different type of jobs. The jobs which arrive late in the arrival sequence get the most benefit.

6.1.3 Execution Time and Variation

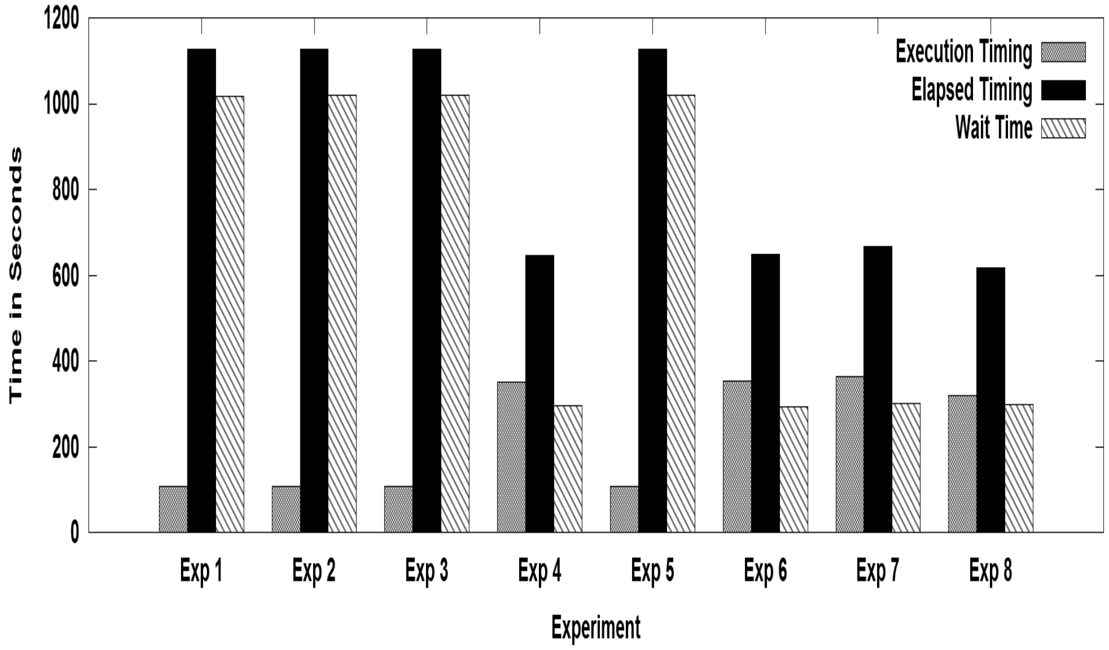
Variability in results for same experiment across different runs can yield interesting results. This was found to be true in this case study. The coefficient of variation for long jobs was found to be less than 0.1 for execution timing results. However, for short jobs it lies in the range of 0.1-0.5. This required a thorough investigation. Table 6.4 shows the variance between different experiments for short job types.

There are 3 different reasons for the large variation in short job execution time.

- **Data Locality**: It plays a major role in variation for short jobs with very few maps. It is the case that when data is node-local, then the execution time is low and this yields some of the lower end values for



(a) Elapsed, Execution and Wait Timings for Job Type 5



(b) Elapsed, Execution and Wait Timings for Job Type 7

Figure 6.2: Elapsed, Execution And Wait Times For Job Types 5 And 7

Table 6.4: Coefficient Of Variation For Short Job Types

Job Type	**Exp 1**	**Exp 2**	**Exp 3**	**Exp 4**	**Exp 5**	**Exp 6**	**Exp 7**	**Exp 8**
5	0.18	0.18	0.18	0.18	0.19	0.18	0.49	0.18
6	0.14	0.12	0.14	0.41	0.12	0.31	0.31	0.25
7	0.14	0.13	0.12	0.50	0.15	0.49	0.46	0.42

the execution time for short jobs. This is especially true for jobs having 1 or 2 map tasks.

- **Disk contention on local disk**: The higher end values for short jobs are due to the high contention on the disk on which a job's map or reduce task has been scheduled. The data from the experiments reveal that it is not always the case that a node-local map task will produce low map timings. Node-local map tasks can take longer than rack-local tasks if the contention on the node's local disk is too large due to the presence of other map/reduce tasks from the same or different jobs. This leads to variable execution timings.

- **Disk contention on remote disks**: Some other high end values for the execution timings of short jobs is due to the disk contention at remote nodes from which rack-local map data has to be fetched. The disk contention at the remote machine causes the map data to be fetched slowly. The data is to be read from the remote node disk and then fetched over the network for local map processing.

It is also observed from Table 6.4 that variance is very high for Experiment 4, 6, 7 and 8 (when Minimum-User-Limit-Percent is changed). This is obviously due to the change in the settings which causes one of the last two (disk contention at local disk and disk contention at remote disk) to happen and as a result high variability in results across different experiments is obtained.

6.1.4 Makespan

The term Makespan defines the time in which all jobs are finished. The Makespan is a useful metric to understand the overall throughput of the cluster, and it is used to verify that improvements in response ratios do not come at the expense of being able to run fewer jobs per hour on the shared cluster. The Makespan is shown in Table 6.5. It is shown for both of the queues.

The Makespan is similar for both queues because they have equal capacities. It can be seen that Makespan results forms a uniform trajectory across all the experiments. This confirms two facts:

- Improvement in response ratio for jobs due to change in Minimum-User-Limit-Percent (Experiments 4,6,7 and 8), does not come at the cost of increased Makespan time. This tells that the short jobs can enter the system earlier and finish faster without increasing the Makespan time.

Table 6.5: Makespan For Both Queues Across All The Experiments

Experiment	Makespan for Queue 1	Makespan for Queue 2
1	1232	1234
2	1197	1200
3	1230	1231
4	1229	1228
5	1229	1233
6	1229	1229
7	1227	1229
8	1230	1228

- The Makespan for the system is not affected by the Capacity Scheduler settings, but by the time it takes for the longest jobs to execute and finish.

6.2 Differential Queue Capacity: Job Types Separated

This section discusses the results when the queue capacity was changed to 70-30%. The capacity for both the queues was chosen such that smaller queue does not feel squeezed for the capacity and larger queue still holds a reasonably big chunk of cluster slots.

6.2.1 Data Locality

Data locality patterns were the same as in the previous section. Jobs having more map tasks have the largest data locality percentage. Tables 6.6 and 6.7 shows the data locality for the two queues. Small jobs have extremely low data locality, while large jobs continue to have a high number of node-local tasks. Data locality numbers were not influenced by different queue capacities due to the reasons explained in section 6.1.1.

6.2.2 Response Ratio

Minimum-User-Limit-Percent played the same role as it did in Section 6.1.2. It helps in reducing the waiting time for the jobs and improving response ratio. However, the benefit of improved response ratio depends on how much the execution time has been increased and how much the waiting time has been reduced for each job. Similar to the equal queue capacity scenario, Job Types 1, 2 and 3 did not show improvement in elapsed timings. For Job Types 4, 5, 6 and 7 there was significant improvement in elapsed timing for the same reasons as explained in the equal queue capacity scenario. Tables 6.8 and 6.9 shows the response ratio of the jobs for the two queues.

Maximum-Capacity plays a crucial role in the performance of the jobs in a cluster with queues having

Table 6.6: Percentage Of Rack-Local Task For All Job Types For Queue 1 (70%)

Job Type	**Exp 9**	**Exp 10**	**Exp 11**	**Exp 12**	**Exp 13**	**Exp 14**	**Exp 15**	**Exp 16**
1	11.3	13	10.3	11.6	13.9	12.9	11.2	14.2
2	4.9	5.4	4.8	5.8	5.2	7.1	5.9	6.5
3	2.5	2.6	2.5	2.3	2.5	2.7	2.6	2.6
4	1.3	1.2	1.1	1.7	1.3	1.5	1.6	1.7
5	85	89	92	92	92	90	89	89
6	80	84	86	80	84	88	83	84
7	57	62	62	59	63	59	55	57

Table 6.7: Percentage Of Rack-Local Task For All Job Types For Queue 2 (30%)

Job Type	**Exp 9**	**Exp 10**	**Exp 11**	**Exp 12**	**Exp 13**	**Exp 14**	**Exp 15**	**Exp 16**
1	11.2	11.2	10.9	11.5	10.8	13.6	12.0	12.5
2	5.3	6.7	4.5	6.0	7.5	7.7	6.1	7.7
3	2.6	3.6	2.5	2.6	2.8	3.2	2.8	3.0
4	1.2	1.9	1.2	1.6	2.2	2.4	1.6	2.3
5	88	90	91	88	92	88	90	92
6	85	88	85	83	90	81	88	86
7	58	66	59	56	68	60	55	61

Table 6.8: Response Ratio For All Job Types For The Experiments For Queue 1 (70%)

Job Type	**Exp 9**	**Exp 10**	**Exp 11**	**Exp 12**	**Exp 13**	**Exp 14**	**Exp 15**	**Exp 16**
1	1.2	1.15	1.20	1.06	1.14	1.02	1.07	1.01
2	1.92	1.6	1.93	1.57	1.67	1.39	1.5	1.38
3	1.43	1.42	1.51	1.16	1.43	1.12	1.16	1.12
4	5.18	5.38	6.7	1.5	5.5	1.45	1.57	1.45
5	15	12.5	14.6	3.6	12.4	1.9	3.4	1.8
6	12.6	10.28	12.94	1.75	10.43	1.9	1.8	1.9
7	9.1	5.2	10.2	1.75	5.2	1.81	1.86	1.78

Table 6.9: Response Ratio For All Job Types For The Experiments For Queue 2 (30%)

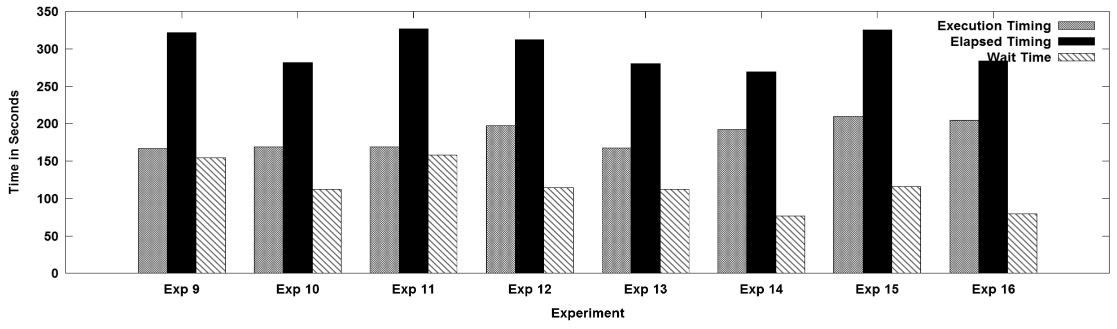
Job Type	**Exp 9**	**Exp 10**	**Exp 11**	**Exp 12**	**Exp 13**	**Exp 14**	**Exp 15**	**Exp 16**
1	1.14	1.32	1.20	1.02	1.32	1.01	1.01	1.01
2	1.78	2.3	1.95	1.5	2.28	1.71	1.5	1.69
3	1.2	1.87	1.51	1.16	1.86	1.22	1.16	1.22
4	3.74	10.78	6.7	1.5	10.8	1.64	1.55	1.64
5	14	26	14.6	3.6	26	6	3.4	6.1
6	13.5	22.8	13.0	2.5	22	5.3	2.6	5.3
7	10.1	14.6	10.3	1.8	14	3.3	1.84	3.4

unequal capacity. This result was different from the equal queue capacity scenario, where Maximum-Capacity did not play a significant role in any of the experiments because both the queues have equal capacity. The results from 70-30% queue capacity reveals the importance of setting up the Maximum-Capacity. As discussed, with default settings of -1 for Maximum-Capacity, there is no limit as to how much cluster could be used by a queue. Maximum-Capacity serves as a cap which forces a queue to respect its own capacity and do not claim extra capacity from the cluster. Figure 6.3 shows the execution and elapsed timings for the different Job Types across different experiments. The figure is shown to identify the role of Maximum-Capacity on the various timings on different type of jobs.

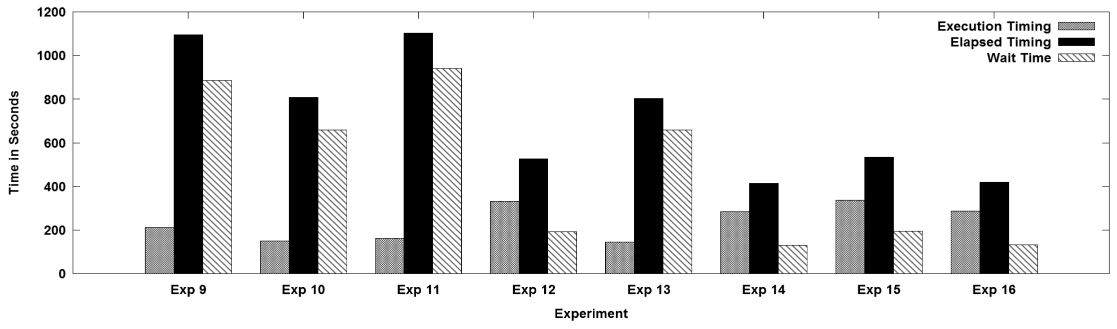
In experiments 10, 13, 14 and 16, where Maximum-Capacity was set to the capacity of the queue, both the queues did not interfere with each other. Hence, elapsed timings were better for the queue having more capacity than the lower capacity queue. In all other experiments, the queue with lower capacity interfered with the higher capacity queue and led to high elapsed times for the higher capacity queue jobs.

With the Minimum-User-Limit-Percent setting other than default, more users could execute tasks in parallel. This makes elapsed timings even worse for the high capacity queue, especially for small jobs. The lower capacity queue interferes more often with the higher capacity queue under such settings. It can be seen that for small jobs, performance of the higher capacity queue degrades so badly that it performs worse than the lower capacity queue. Figure 6.4 and Figure 6.5 shows the waiting time for the jobs in two different experimental settings as they get the resources from the system.

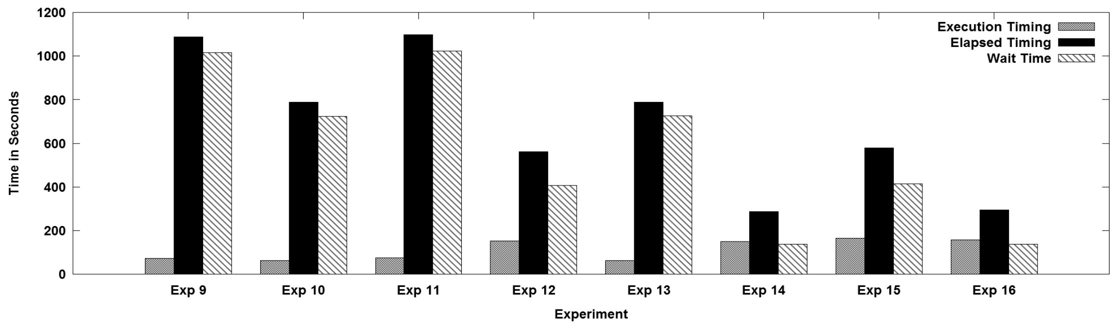
In experiment 14, the Maximum-Capacity keeps a check on the capacity which can be used by the queue at any time and hence the waiting time is low for high capacity queue jobs and very high for low capacity jobs (Minimum-User-Limit-Percent is 25% in both the experiments). The waiting time for small jobs (11-50 in the graph) in experiment 12 is very high, which leads to very high elapsed time. Keeping no limit on Maximum-Capacity in experiment 12 provides an easy opportunity for lower capacity queue to allocate itself the resources which were at the disposal of high capacity queue and hence depriving high capacity queue of its capacity.



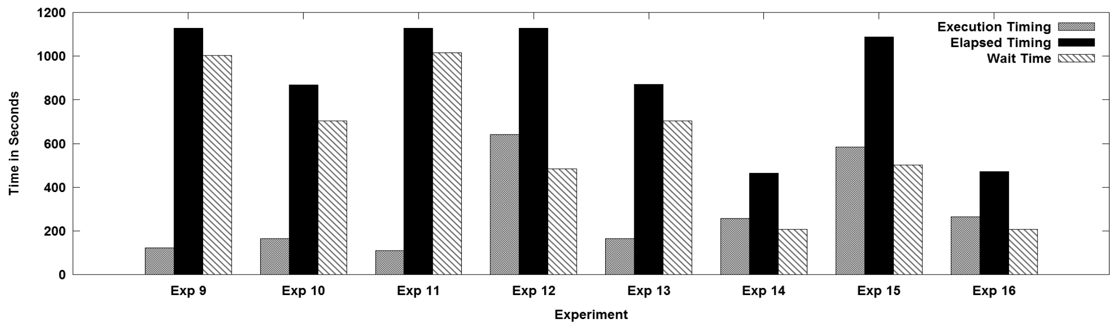
(a) Elapsed, Waiting and Execution Timings for Job Type 2



(b) Elapsed, Waiting and Execution Timings for Job Type 4

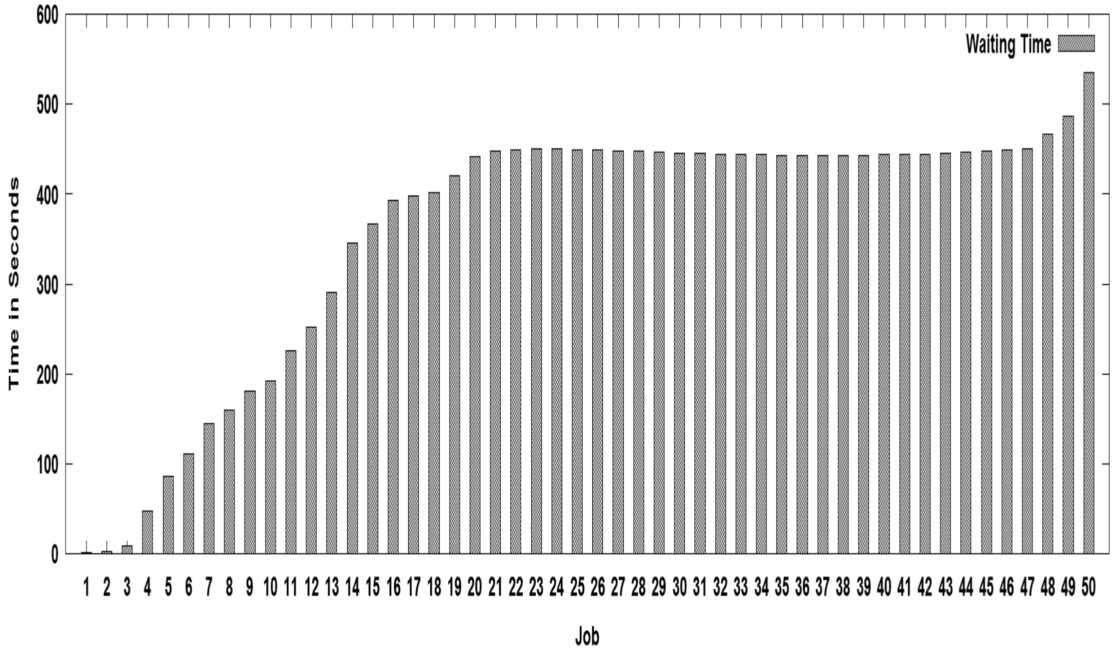


(c) Elapsed, Waiting and Execution Timings for Job Type 5

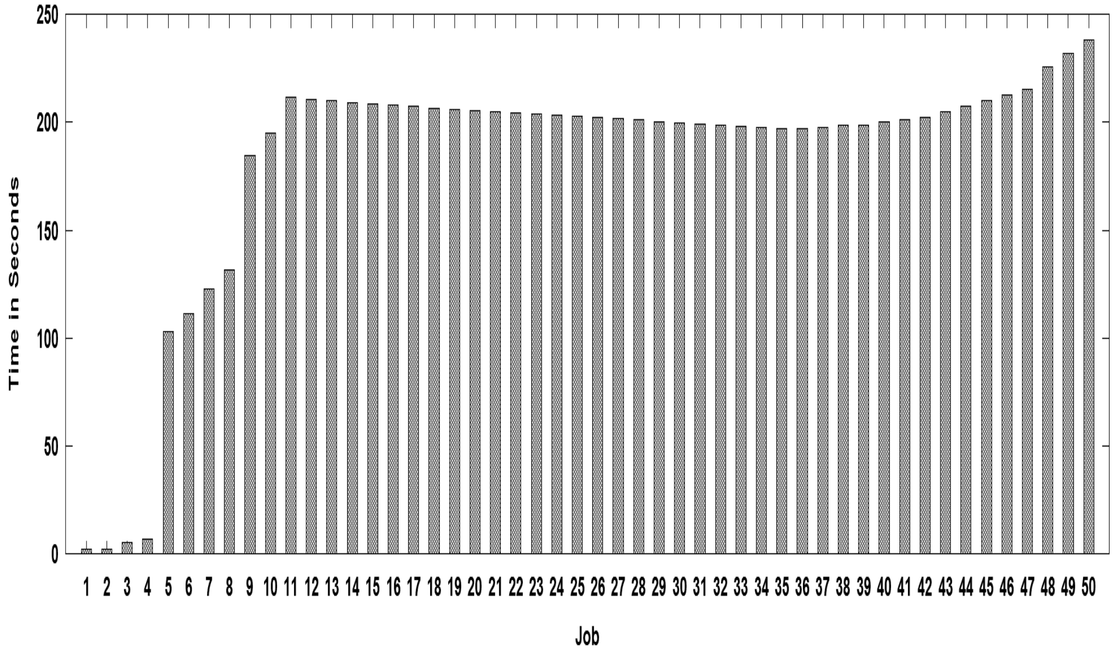


(d) Elapsed, Waiting and Execution Timings for Job Type 7

Figure 6.3: Elapsed, Execution And Wait Times For Job Types 2,4,5 And 7

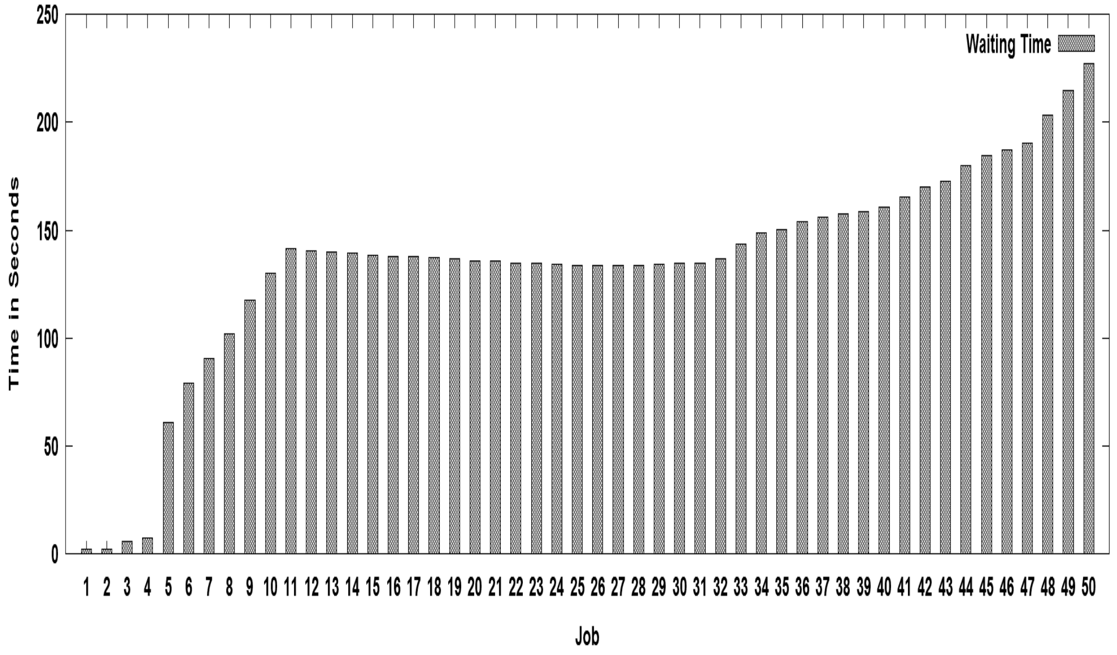


(a) Waiting Time for all Jobs for Queue 1 (70%) in Exp 12

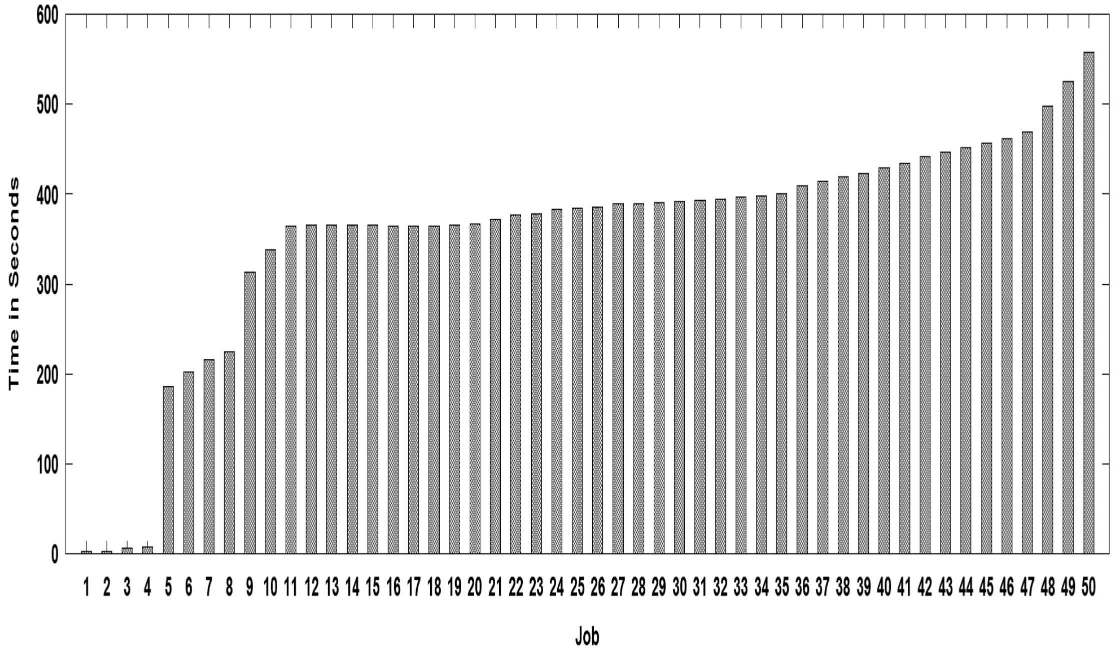


(b) Waiting Time for all Jobs for Queue 2 (30%) in Exp 12

Figure 6.4: Waiting Time For All Jobs In Exp 12



(a) Waiting Time for all Jobs for Queue 1 (70%) in Exp 14



(b) Waiting Time for all Jobs for Queue 2 (30%) in Exp 14

Figure 6.5: Waiting Time For All Jobs In Exp 14

6.2.3 Execution Time and Variation

The coefficient of variation for long jobs was found to be less than 0.1 for execution timing. However, for short jobs it was very high. Table 6.10 and 6.11 shows the variance between different experiments for short job types. Data locality, disk contention on local disk and disk contention on remote disk are three reasons leading to such a high coefficient of variation for short jobs. As discussed in Section 6.1.2 on response time, with Minimum-User-Limit-Percent setting and no Maximum-Capacity limit, the short jobs in Queue 1 are erratic in terms of execution time. On some occasions, they did not get required slot in time and had to wait. On other occasions, they did get the slots in time. This leads to a very high variability in their execution times.

Table 6.10: Coefficient Of Variation For Short Job Types For Queue 1 (70%)

Job Type	**Exp 9**	**Exp 10**	**Exp 11**	**Exp 12**	**Exp 13**	**Exp 14**	**Exp 15**	**Exp 16**
5	0.19	0.18	0.19	0.76	0.15	0.23	0.80	0.20
6	0.15	0.15	0.13	0.23	0.14	0.1	0.36	0.11
7	0.14	0.25	0.13	0.06	0.29	0.11	0.19	0.09

Table 6.11: Coefficient Of Variation For Short Job Types For Queue 2 (30%)

Job Type	**Exp 9**	**Exp 10**	**Exp 11**	**Exp 12**	**Exp 13**	**Exp 14**	**Exp 15**	**Exp 16**
5	0.16	0.20	0.19	0.24	0.19	0.14	0.16	0.14
6	0.16	0.16	0.13	0.31	0.13	0.11	0.20	0.11
7	0.1	0.13	0.12	0.3	0.14	0.14	0.32	0.13

6.2.4 Makespan

Makespan shows different findings compared to the equal queue capacity scenario. Makespan is shown in Table 6.12. The table depicts both queue 1 (higher capacity) and queue 2 (lower capacity). The Makespan is similar for both the queues when Maximum-Capacity is not fixed. Not setting Maximum-Capacity allows the lower capacity queue to get additional capacity from the higher capacity queue and hence both the queues have similar Makespan times. When Maximum-Capacity limit is used, the Makespan for the high capacity queue improves as it has more capacity and its jobs finish quickly. Both findings shown earlier for Makespan in the equal queue capacity scenario (Section 6.1.4) regarding Minimum-User-Limit-Percent and the longest jobs determining the Makespan time hold true here as well.

6.3 Interleaved Jobs

This section contains the discussion of the performance of jobs in a realistic scenario in Hadoop cluster: an interleaving job submission pattern, where a sequence of 4 short jobs is followed by 1 long job and so on. In the interleaved scenario, the queues are named as Queue 1 and Queue 2.

6.3.1 Data Locality

The data locality patterns were the same as Section 6.1.1. Jobs having a greater number of map tasks have the largest data locality percentage. Table 6.13 shows the data locality for Queue 1. Queue 2 has similar results. Small jobs have extremely low data locality, while large jobs have high data locality. Data locality numbers were not influenced by different queue capacity due to the reasons explained in earlier sections.

6.3.2 Response Ratio

For the interleaved scenario, *Minimum-User-Limit-Percent* is the most important setting to lower the response time. It helps in improving the response ratio for all the job types. Table 6.14 shows the response ratio for all job types across different experiments. The improvement in response ratio depends on the arrival timings of the job. A job coming earlier in the system under default settings gets minimum improvement in response ratio as its waiting time does not change much under changed *Minimum-User-Limit-Percent* parameter. For a job coming later to the system, the huge improvement in waiting time due to *Minimum-User-Limit-Percent* helps to improve response ratio significantly. In a system running over the longer-term, large gains could be expected. As discussed in previous sections, however, improved response ratio does not always mean improved elapsed time for a job type.

Table 6.12: Makespan For Both Queues Across All The Experiments

Experiment	**Makespan for Queue 1**	**Makespan for Queue 2**
1	1211	1244
2	1001	1399
3	1230	1227
4	1223	1226
5	997	1397
6	1029	1479
7	1233	1226
8	1021	1489

Table 6.13: Percentage Of Rack-Local Task For All Job Types Across The Experiments

Job Type	**Exp 17**	**Exp 18**	**Exp 19**	**Exp 20**	**Exp 21**	**Exp 22**	**Exp 23**	**Exp 24**
1	11.5	10.8	10.8	12.7	11.1	12.6	12.3	12.8
2	5.2	5.4	5.1	6.4	5.2	6.7	6.4	6.8
3	2.4	2.6	2.9	2.8	2.4	3.3	2.9	3.3
4	1.0	1.0	1.2	1.5	1.0	1.8	1.8	2.4
5	88	88	90	85	92	88	90	88
6	83	84	83	83	85	86	86	83
7	52	59	57	60	60	56	60	61

Table 6.14: Response Ratio For All Job Types For The Experiments - Interleaved Scenario

Job Type	**Exp 17**	**Exp 18**	**Exp 19**	**Exp 20**	**Exp 21**	**Exp 22**	**Exp 23**	**Exp 24**
1	1.18	1.18	1.18	1.02	1.18	1.02	1.01	1.02
2	1.90	1.94	1.93	1.46	1.91	1.46	1.47	1.47
3	1.35	1.51	1.51	1.15	1.50	1.15	1.15	1.15
4	8.46	8.70	8.91	1.79	8.48	1.79	1.77	1.80
5	2.03	2.02	2.03	1.41	2.02	1.43	1.43	1.42
6	3.19	4.16	4.24	2.47	4.10	2.53	2.53	2.48
7	4.20	6.91	6.85	1.90	6.70	1.89	1.90	1.90

6.3.3 Execution Time and Variation

The coefficient of variation for long jobs was found to be less than 0.1 for execution timing. However, for short jobs, it was very high. Table 6.15 shows the variance between different experiments for short job types. Data locality, disk contention on local disk and disk contention on remote disk are three reasons leading to such a high of variance for short jobs.

Table 6.15: Coefficient Of Variation For Short Job Types For Queue 1

Job Type	**Exp 17**	**Exp 18**	**Exp 19**	**Exp 20**	**Exp 21**	**Exp 22**	**Exp 23**	**Exp 24**
5	0.20	0.20	0.19	0.36	0.17	0.32	0.30	0.34
6	0.19	0.15	0.13	0.13	0.12	0.12	0.12	0.12
7	0.48	0.11	0.1	0.08	0.8	0.08	0.09	0.10

6.3.4 Makespan

The smaller value of Makespan for the same set of jobs means higher throughput. It indicates that the same number of jobs were processed in a lower amount of time, which means more jobs were processed per unit of time.

The Makespan for the interleaved scenario yet again shows that Makespan for both the queues is determined by long jobs. Changing Minimum-User-Limit-Percent improves the response ratios for all the jobs but does not decrease Makespan for the queues. Table 6.16 shows the Makespan for both the queues across different experiments.

Table 6.16: Makespan For All Experiments - Interleaved Scenario

Experiment Number	**Makespan for Queue 1**	**Makespan for Queue 2**
17	1250	1251
18	1251	1254
19	1249	1249
20	1229	1232
21	1252	1256
22	1233	1234
23	1232	1232
24	1228	1229

6.4 Separate Queue Jobs

This section tries to answer another important question: What happens when short and long jobs get their own respective queues? In the separate queue scenario, Queues 1 and 2 are short job queue and Queues 3 and 4 are long job queues. As short and long jobs have their own queue, Job Types 1, 2, 3 and 4 are submitted to Queues 3 and 4. Job Types 5, 6 and 7 are short jobs and are submitted to Queues 1 and 2.

6.4.1 Data Locality

Data locality patterns were the same as in previous sections. Jobs having a greater number of map tasks have the largest data locality percentage. Table 6.17 shows the data locality for Queue 1 and Queue 3.

Table 6.17: Percentage Of Rack-Local Task For All Job Types Across The Experiments In Queue 1 And Queue 3

Job Type	**Exp 25**	**Exp 26**	**Exp 27**	**Exp 28**	**Exp 29**	**Exp 30**	**Exp 31**	**Exp 32**
1	10.7	10.4	11.3	12.6	10.6	12.6	11.4	12.5
2	4.8	5.6	5.2	5.7	5.0	6.7	6.9	6.5
3	2.6	2.6	2.7	2.5	2.7	3.0	3.0	2.9
4	1.4	1.3	1.0	1.6	1.5	1.5	1.6	1.6
5	88	91	88	88	89	88	95	90
6	85	84	88	86	86	86	80	82
7	60	60	57	58	55	59	63	61

The other two queues have similar results. Small jobs have extremely low data locality, while large jobs have high data locality. Data locality numbers were not influenced by different queue capacity due to the reasons explained in Section 6.1.1.

6.4.2 Response Ratio

Minimum-User-Limit-Percent helps in improving the response ratio in all the conducted experiments so far. This raises an important question: Is it true for every job submission pattern? Surprisingly, the separate queue job submission pattern reveals something different. It comes to the fore that instead of Minimum-User-Limit-Percent, Maximum-Capacity helps in improving the response ratio for short jobs. But, for long jobs, Minimum-User-Limit-Percent helps in improving the response ratio in separate queue job submission pattern. Table 6.18 shows the response ratio for different Job Types across the different experiments.

It should be expected that with its own queue, short jobs will finish faster and hence their response

Table 6.18: Response Ratio For All Job Types For The Experiments - Separate Queue Scenario

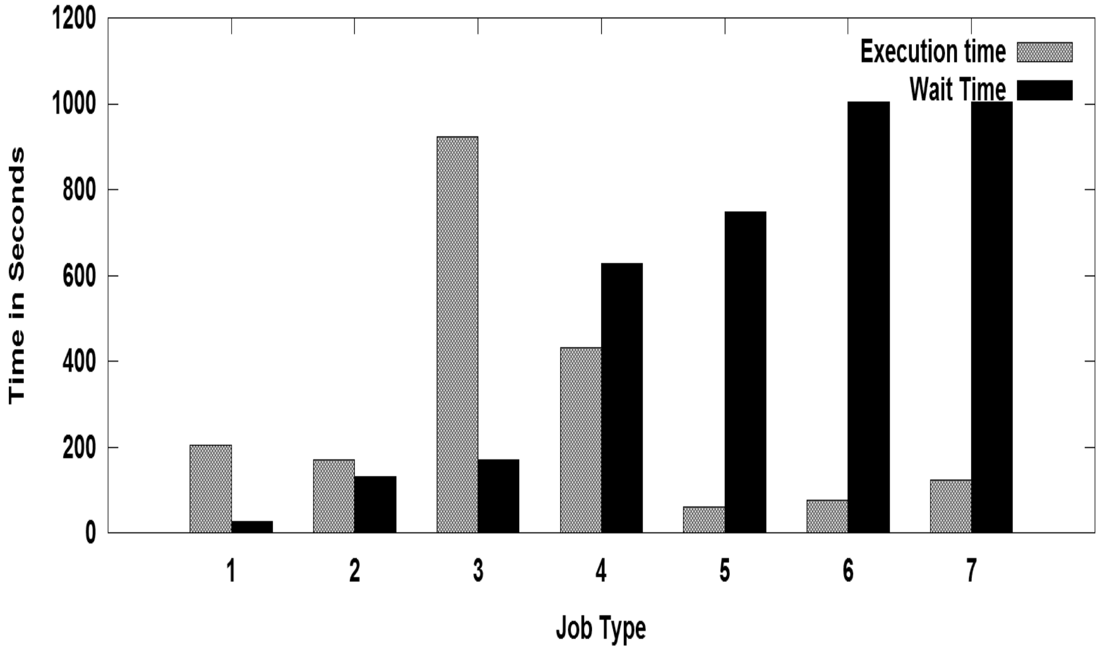
Job Type	**Exp 25**	**Exp 26**	**Exp 27**	**Exp 28**	**Exp 29**	**Exp 30**	**Exp 31**	**Exp 32**
1	1.14	1.23	1.25	1.01	1.25	1.01	1.02	1.02
2	1.76	2.21	1.94	1.39	2.18	1.56	1.38	1.54
3	1.19	1.59	1.51	1.12	1.58	1.16	1.12	1.15
4	2.46	9.36	6.51	1.55	9.06	1.73	1.56	1.72
5	13.05	1.2	5.85	4.84	1.18	1.06	5.0	1.08
6	14.11	1.65	7.03	5.22	1.48	1.20	5.18	1.20
7	9.04	1.53	4.18	4.88	1.45	1.27	4.87	1.25

ratio will have low values. However, in experiments 9, 11, 12 and 15 (where Maximum-Capacity limit is not imposed), the response ratio for short jobs (Job types 5, 6, and 7) is high. The reason is the presence of long running jobs in other queues. In absence of the Maximum-Capacity limits, the long jobs in the other queues take the slots from short jobs queues. This leads to huge waiting time for short jobs, which in turn leads to high elapsed timings and thus a high response ratio. Imposition of limits reverses the trend. In this case, long jobs suffers higher response ratios. The Maximum-Capacity limit puts an upper bound on the capacity a queue can use and hence short jobs have lower waiting time and also lower response ratio values.

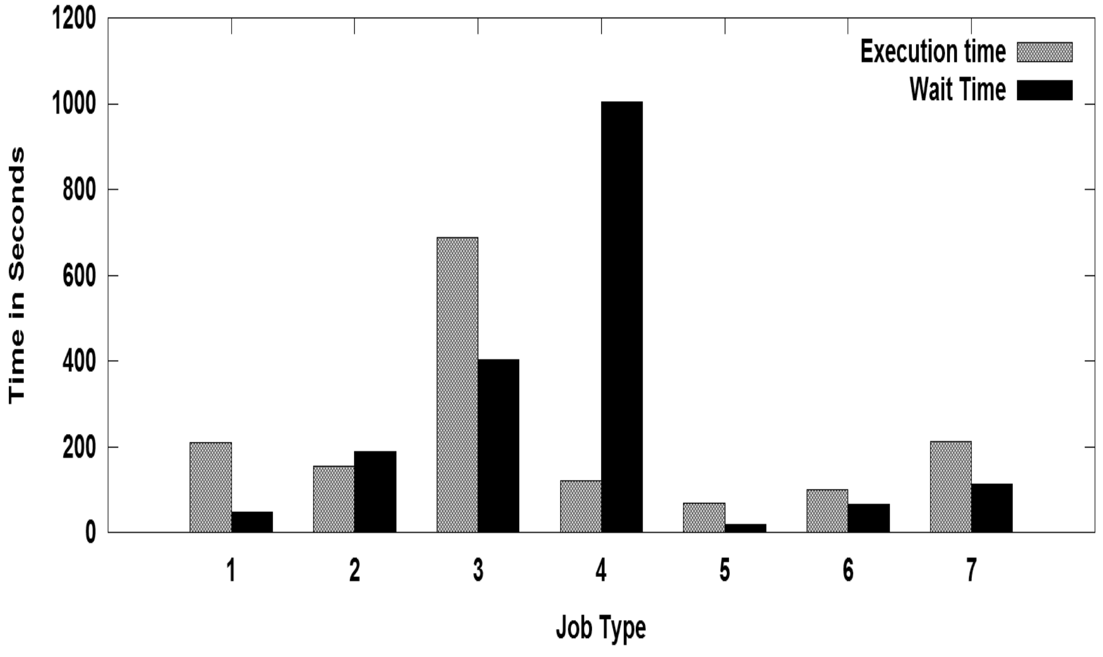
Figure 6.6 shows the execution and waiting time for all job types under two different settings. Two different situations occur. Under default settings, long jobs keep using the resources from the short job queue and hence don't have to wait long. This makes short jobs to wait very long. This explains the observed waiting time in 6.6 (a) for all different job types. In fact, most of the short jobs start after most of the long jobs finish. This leads to shorter execution time for short jobs because most of the nodes are idle (as most of the long jobs are finished) and hence there is less contention on each node. In the second experiment, long jobs have longer waiting times than for the default settings because of Maximum-Capacity limit. They cannot take resources from the short job queues. All short jobs have low waiting times and execute in parallel with long jobs, which leads to their high execution time. Also, as all short jobs finish quickly, and as long jobs have to wait longer, the execution time for such long jobs is lower.

6.4.3 Execution Time and Variation

The coefficient of variation for long jobs was found to be less than 0.1 for execution timing. However, for short jobs it was very high. Table 6.19 shows the variance between different experiments for short job types. Again, data locality, disk contention on local disks and disk contention on remote disks are three reasons leading to such a high coefficient of variation for short jobs.



(a) Waiting and Execution Timings for Job Types under Experiment 1 (Default Settings)



(b) Waiting and Execution Timings for Job Types under Experiment 2 (Max capacity limits)

Figure 6.6: Waiting And Execution Times For All Job Types: Separate Queues

Table 6.19: Coefficient Of Variation For Short Job Types For Queue 1 And Queue 3: Separate Queues

Job Type	Exp 25	Exp 26	Exp 27	Exp 28	Exp 29	Exp 30	Exp 31	Exp 32
5	0.20	0.20	0.19	0.36	0.17	0.32	0.30	0.34
6	0.19	0.15	0.13	0.13	0.12	0.12	0.12	0.12
7	0.48	0.11	0.1	0.08	0.8	0.08	0.09	0.10

6.4.4 Makespan

The Makespan for separate queue scenario reveals quite an interesting result. It should be expected that jobs present in a queue should be responsible for the value of the Makespan. This happened so far in all the earlier experiments. However, it was observed that in the separate queue scenario, the Makespan for the short job queue was affected by the long job queue (under no Maximum-Capacity limits). Table 6.20 shows the Makespan for all 4 queues across different experiments.

Table 6.20: Makespan For All Experiments - Separate Queue Scenario

Experiment Number	Makespan for Queue 1	Makespan for Queue 2	Makespan for Queue 3	Makespan for Queue 4
25	1196	1196	1249	1252
26	407	409	1242	1239
27	1200	1201	1239	1235
28	1242	1232	1207	1203
29	387	383	1262	1262
30	432	418	1243	1242
31	1235	1238	1202	1205
32	414	438	1259	1262

Limiting the Maximum-Capacity does not allow long jobs in their queues to take slots from short job queue. It also decreases the waiting time for short jobs in their queues. This makes Makespan for short job queues small compared to other experiments (when Maximum-Capacity was not limited).

6.5 Analysis

Based on the experiments conducted in this chapter, analysis is done and presented in this section. It is expected that the conclusions drawn here will be helpful for the Hadoop administrators.

6.5.1 Question 1

In presence of a long sequence of very big jobs followed by small jobs, Minimum-User-Limit-Percent plays the most important role. It helps in decreasing the waiting time for the jobs which are late in the arrival sequence to get the resources to run their tasks later under default settings. Changing Minimum-User-Limit-Percent also causes more competition for resources among jobs. This leads to increased execution times for the jobs.

Changing Minimum-User-Limit-Percent improves the response ratio for all the jobs. However, the performance benefits for the job depends on how much response ratio has been changed across the experiments under different settings. For a job, better response time also means better elapsed timings if the decreased wait timings does not get offset by the increased execution time (due to Minimum-User-Limit-Percent setting).

Data locality is not much influenced by the Capacity Scheduler settings. Data locality increases with the number of map tasks for a job. The Capacity Scheduler treats rack-local maps and node-local tasks as identical. Hence, adding an extra layer of delay scheduling at rack-local tasks and treating them different can improve performance for the jobs.

For queues with unequal capacity, it is a best practice to use Maximum-Capacity limit on the queues for real performance gains. Otherwise, the queue with lower capacity interferes more often with high capacity queue and takes the necessary resources away from it. This affects the response ratio for jobs present in the higher capacity queue.

6.5.2 Question 2

For the job submission pattern in which long and short jobs are submitted in an interleaved manner in the same queue, Minimum-User-Limit-Percent is the most important factor. Minimum-User-Limit-Percent helps in improving the response ratio for all the jobs and especially for the ones who are submitted later in the queue. In a system running over the longer-term, large gains could be expected. Makespan for the queue is determined by the long jobs. Data locality results are similar to those obtained in Question 1.

6.5.3 Question 3

For job submission patterns in which short jobs and long jobs are submitted in separate queues, two different behaviours were observed. With no Maximum-Capacity limits imposed, long jobs affect the short job queue behaviour by taking slots from them. This results in long waiting time for short jobs. Makespan and response ratio for short queues jobs are also affected by long jobs in such case. With Maximum-Capacity limits imposed, the short jobs are not affected by long jobs. Hence, their response time is small and their Makespan is not affected by the long jobs.

For job submission patterns where short jobs and long jobs are submitted in separate queues, Minimum-User-Limit-Percent helps in improving the response ratio but not as much as Maximum-Capacity. So, imposing Maximum-Capacity limits with Minimum-User-Limit-Percent settings can achieve the most optimal results for short jobs in terms of response time and Makespan.

If a queue capacity is to be divided among short and long jobs, long jobs must be given more capacity. This gives them more resources to run as they need more slots from the cluster.

Separating long jobs and short jobs in two different queues with Maximum-Capacity limits imposed and Minimum-User-Limit-Percent settings provides the optimal solution and is better than submitting both short and long jobs in the same queue. The waiting time for short jobs by maintaining separate queues improves drastically and hence they finish faster, giving a low Makespan value. This gives the administrator the capability to run more short jobs.

Chapter 7

Conclusions

The Capacity Scheduler is one of the task schedulers present in Hadoop. It is being used by a lot of premier organizations like Yahoo! and LinkedIn. Still, little is known about the characteristics of Capacity Scheduler and how its parameters impact MapReduce applications. This thesis helps in understanding the Capacity Scheduler characteristics with the help of simulation. This chapter contains a discussion on thesis contribution to the field of research, a broad summary of results and future work.

7.1 Thesis Summary

There is only limited knowledge in the published literature about the working of Capacity Scheduler and its parameters. This thesis highlights the need to understand and identify the importance of those parameters. It is important to understand how these parameters impact the MapReduce workload and which parameters are the most relevant under which type of workload. Such an understanding can help administrators of Hadoop Capacity Scheduler to fine tune the parameters according to their organizational needs.

An initial single factor sensitivity analysis on a simplistic workload consisting of a few MapReduce representative applications shows that all the capacity Scheduler parameters have an impact. Increasing the number of queues and changing queue capacity leads to change in the execution times of jobs. Lowering the queue capacity increases the execution time and raising the queue capacity decreases the execution time under the same workload. Maximum-Capacity limits a queue from using resources from other queues. Minimum-User-Limit-Percent lowers the waiting time for the jobs at the expense of increased execution time. Assigning a high priority to the jobs allows them to execute and finish sooner than lower priority jobs in the queue. Job Initialization Parameters (Maximum-System-Jobs, Maximum-Initialized-Active-Tasks and Maximum-Initialized-Active-Tasks-Per-User) controls job admissibility into the system and controls concurrency.

In order to study the impact of Capacity Scheduler in more detail, a simulator is needed. There exists no Capacity Scheduler simulator in the existing research with the exception of Mumak, which has several limitations with respect to the goals of this thesis. In particular, the need to have real workload traces from actual production cluster was a big roadblock. Hence an existing simulator named MRPERF (MapReduce simulator) was modified and the capability of Capacity Scheduler was integrated into it. With MRPERF being open source, it was easy to change and it allows the integration of the Capacity scheduler. MRPERF does not need real traces. After the thorough analysis of the code and its meaning, the implementation to integrate Capacity Scheduler in MRPERF was completed. To check if the new Capacity Scheduler simulator works, a validation study was undertaken. A simplistic workload was used on a real cluster and the impact of Capacity Scheduler parameters was studied. Single factor experiments were done. Later, the same workload was injected into the simulator to check the accuracy of the results vis-a-vis real cluster results. The numerical accuracy of the simulator was not 100% because of the following factors: stragglers, same-reduce-node effects and modeling errors already present in MRPERF. Many of these factors are random and thus unpredictable. However, the changing trends between different experiments was perfectly captured by the simulator in comparison to real cluster results.

In the final part of the thesis, a full blown simulation was performed to understand the impact of Capacity Scheduler parameters on a workload under different job submission patterns. Queue Capacity, Maximum-Capacity and Minimum-User-Limit-Percent were found to be the most relevant parameters under different job submission patterns. Minimum-User-Limit-Percent improves the response times for the jobs by decreasing their waiting time. This improvement is more prominent for the jobs which are submitted late in the queue. Data locality increases with more map input data. The Capacity Scheduler does not make a big impact on the data locality results for the same job under different settings and job submission patterns. It is beneficial to maintain separate queues for short and long jobs with a Minimum-User-Limit-Percent setting and Maximum-Capacity setting. This improves the response time for all the jobs and also prevents long job to steal capacity from the short job queues. In the absence of Maximum-Capacity limit, long jobs use the capacity from short job queues. It prevent short jobs from being executed in a timely manner.

7.2 Thesis contribution

This thesis explores Capacity Scheduler and its parameters in detail. It makes the following contributions:

- This thesis is the first systematic study to understand Capacity Scheduler parameters and their impact on the performance of MapReduce applications. The thesis contains experimental results performed on a real world cluster as well as simulation results. Capacity Scheduler was developed by Yahoo! to allow efficient sharing of the cluster resources among multiple organizations. It is heavily used by organizations like Yahoo!, LinkedIn, etc., but no study exists on its behaviour and how the various Capacity Scheduler parameters affect MapReduce applications. Users and administrators often get confused by the Capacity Scheduler settings in absence of such studies.

- This thesis contributes a component of the simulator which can simulate Capacity Scheduler. The existing Capacity Scheduler code was studied and then integrated in the MRPERF simulator. Existing MapReduce simulators that do not require detailed production traces do not permit the modeling of the Capacity Scheduler. The simulator developed in this thesis does not need real production traces. This greatly simplifies the testing of existing or new Scheduler parameters under simulation.

- The work in this thesis isolates and studies each Capacity Scheduler parameter independently and one by one on a real cluster to see which parameters are the most important ones in affecting MapReduce application behaviour. Representative MapReduce applications were used in experiments.

- The Capacity Scheduler component developed in this thesis was verified against real cluster results on representative MapReduce applications. The simulator was not 100 % numerically accurate due to various factors like stragglers, same-reduce-node effects, etc. These factors are random and thus unpredictable. However, the trends were predicted accurately by the simulator vis-a-vis real cluster results.

- Large scale simulations were done in the thesis under different job submission patterns with the most important Capacity Scheduler parameters (obtained on the basis of knowledge collected from previous experimental results). The observations from this simulation study can be used by administrators to effectively manage a shared cluster using Capacity Scheduler.

7.3 Future Work

Future work can be divided into three broad categories: improvement in the simulator, improvement in the integrated Capacity Scheduler simulation and in the experimental design space.

7.3.1 Improvement in MRPERF simulator

Many a lot of additional features and modifications can be realized in MRPERF. It was implemented based on ns-2, a packet-level network simulator, and its performance is much worse than other simulators. Simulating a 30 node cluster with 100 jobs takes 4 hours. By porting the existing MRPERF framework onto a faster network simulator, it can be made faster. MRPERF can also be extended to have multiple-disk support on a single node. Memory configuration can also be implemented in MRPERF. There are a couple of sub phases which are not modeled in MRPERF, such as Collect. Such sub-phases omissions leads to prediction errors in the simulator. MRPERF can be extended to include omitted sub-phases. Straggler support can be put into MRPERF.

7.3.2 Improvement in Capacity Scheduler simulation

The integration performed to include Capacity Scheduler into MRPERF does not contain code for speculative execution. Speculative execution helps to reduce lost time due to stragglers. Since MRPERF does not support stragglers, speculative execution was not integrated. Hadoop developers found out the same problem with low data locality as described in this thesis. Surprisingly, the timings of the finding of this issue [1] was coincidental with this thesis work. Hadoop developers fixed the issue with a new level added for delay scheduling at the rack level. This fix can be added to the existing Capacity Scheduler simulation to test if it improves the data locality for the jobs. As observed in the experimental results, allocating separate queues to small and large jobs with Maximum-Capacity limits imposed gives optimal results. In such a scenario, small jobs finish faster. The queue capacity of small jobs queue remains unused after they finish execution, as long jobs cannot use small jobs queue capacity. A new improvement can be made to the simulator code by allowing large jobs to use the small jobs queue capacity when the small job's queue does not have any more jobs. This will involve changing of the Capacity Scheduler parameters on the fly.

7.3.3 Experimental Design

The experimental design can be extended to include measuring Capacity Scheduler characteristics under different network topologies, and data layout algorithms. The workload can include new types of jobs such as map only jobs. Experiments can be performed having job submission patterns with bursty workloads. The simulation can be conducted under different cluster configurations to check the impact of Capacity Scheduler settings under changing cluster sizes. The cluster can vary in terms of the number of nodes, map slots and reduce slots. Steady state simulation and experiments with different job arrival patterns can be undertaken as future work.

Future work can also be extended in the area of optimization. What and how values should be assigned to multiple Capacity Scheduler parameters to get the best running time for jobs will be a good research question to pursue. For example, how to determine how much more capacity should be allocated to the long job queue?

[1] https://issues.apache.org/jira/browse/MAPREDUCE-4305

References

[1] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big data and cloud computing: new wine or just new bottles? *Proceedings of the VLDB Endowment*, 3(1-2):1647–1648, September 2010.

[2] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 1–16, Vancouver, BC, Canada, 2010.

[3] Shivnath Babu. Towards automatic optimization of MapReduce programs. In *SoCC '10*, pages 137–142, Indianapolis, IN, June 2010.

[4] Stephen P. Bradley, Arnoldo C. Hax, and Thomas L. Magnanti. *Applied Mathematical Programming.* Addison-Wesley, 1977.

[5] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *Computer*, 33(5):59 –67, May 2000.

[6] Rajkumar Buyya and Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1175–1220, January 2002.

[7] Henri Casanova. Simgrid: a toolkit for the simulation of application scheduling. In *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 430–437, Brisbane, Australia, May 2001.

[8] Surajit Chaudhuri and Gerhard Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 1–10, Cairo, Egypt, September 2000.

[9] Jagmohan Chauhan, Dwight Makaroff, and Winfried Grassmann. The impact of capacity scheduler configuration settings on MapReduce jobs. In *Proceedings of the 2012 Second International Conference on Cloud and Green Computing*, pages 667–674, Xiangtan, China, November 2012.

[10] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, August 2012.

[11] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. The case for evaluating mapreduce performance using workload suites. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 390–399, Singapore, 2011.

[12] Jeffrey Cohen, John Eshleman, Brian Hagenbuch, Joy Kent, Christopher Pedrotti, Gavin Sherry, and Florian Waas. Online expansion of largescale data warehouses. *Proceedings of the VLDB Endowment*, 4(12):1249–1259, August 2011.

[13] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[14] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[15] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[16] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, Bolton Landing, NY, October 2003.

[17] Suhel Hammoud, Maozhen Li, Yang Liu, Nasullah K. Alham., and Zelong Liu. MRSim: A discrete event based MapReduce simulator. In *Proceedings of the 7th International Fuzzy Systems and Knowledge Discovery (FSKD) Conference*, volume 6, pages 2993–2997, Yuntai, China, August 2010.

[18] Per Brinch Hansen. *Operating System Principles.* Upper Saddle River, NJ, 1973.

[19] Herodotos Herodotou. Hadoop Performance Models. Technical Report CS-2011-05, Duke CS, 2011.

[20] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, pages 261–272, Asilomar, CA, January 2011.

[21] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 261–276, Big Sky, Montana, October 2009.

[22] Scott Kirkpatrick, Charles D. Gelatt, and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[23] Elie Krevat, Tomer Shiran, Eric Anderson, Joseph Tucek, Jay J Wylie, and Gregory R Ganger. Applying simple performance models to understand inefficiencies in data-intensive computing. Technical report, UC Berkeley CS, 2011.

[24] Yang Liu, Maozhen Li, Nasullah Khalid Alham, and Suhel Hammoud. Hsim: A MapReduce simulator in enabling cloud computing. *Future Gener. Comput. Syst.*, 29(1):300–308, January 2013.

[25] Thomas Sandholm and Kevin Lai. Dynamic proportional share scheduling in hadoop. In *Proceedings of the 15th International Conference on Job Scheduling Strategies for Parallel Processing*, pages 110–131, Atlanta, GA, May 2010.

[26] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File system. In *Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies*, pages 1–10, Incline Village, NV, May 2010.

[27] Hou Song. Performance analysis of MapReduce computing framework. Masters thesis, National University of Singapore, 2011.

[28] HyoJung Song, Xin Liu, Denis Jakobsen, Ranjita Bhagwan, Xianan Zhang, Kenjiro Taura, and Andrew A. Chien. The micrgrid: a scientific tool for modeling computational grids. In *Proceedings of the 13th International Conference On Supercomputing*, pages 53–53, Dallas, TX, November 2000.

[29] Atsuko Takefusa, Satoshi Matsuoka, Hidemoto Nakada, Kento Aida, and Umpei Nagashima. Overview of a performance evaluation system for global computing scheduling algorithms. In *Proceedings of the 8th International Symposium on High Performance Distributed Computing*, pages 97–104, Redondo Beach, CA, August 1999.

[30] Fei Teng, Lei Yu, and Frederic Magoules. Simmapreduce: A simulator for modeling MapReduce framework. In *Proceedings of the 5th FTRA International Conference on Multimedia and Ubiquitous Engineering*, pages 277–282, Crete, Greece, June 2011.

[31] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a MapReduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, August 2009.

[32] Chao Tian, Haojie Zhou, Yongqiang He, and Li Zha. A dynamic MapReduce scheduler for heterogeneous workloads. In *Proceedings of the 8th International Conference on Grid and Cooperative Computing, 2009*, pages 218–224, Lanzhou, China, August 2009.

[33] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: automatic resource inference and allocation for MapReduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, pages 235–244, Karlsruhe, Germany, June 2011.

[34] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. Play it Again, SimMR! In *Proceedings of the IEEE CLUSTER 2011*, pages 253–261, Austin, TX, September 2011.

[35] Guanying Wang, Ali R. Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in MapReduce setups. In *Proceedings of the 2009 IEEE 17th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 1 –11, London, UK, September 2009.

[36] Guanying Wang, Ali R. Butt, Prashant Pandey, and Karan Gupta. Using realistic simulation for performance analysis of MapReduce setups. In *Proceedings of the Large-Scale System and Application Performance Workshop*, pages 19–26, Garching, Germany, June 2009.

[37] Tom White. *Hadoop: The Definitive Guide*. O'Reilly, 2009.

[38] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, pages 265–278, Paris, France, April 2010.

[39] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 29–42, San Diego, CA, December 2008.