The Impact of Capacity Scheduler Configuration Settings on MapReduce Jobs

Jagmohan Chauhan, Dwight Makaroff and Winfried Grassmann
Dept. of Computer Science, University of Saskatchewan
Saskatoon, SK, CANADA S7N 3C9
Email: [jac735, makaroff, grassman]@cs.usask.ca

***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 influence an organization to 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 application execution time and resource allocation policy decisions.**

Existing work shows that the performance of MapReduce jobs depends on the cluster configuration, input data type and job configuration settings. However, that work fails to take into account the task scheduler settings. We show, through experimental evaluation, that task scheduler configuration parameters make a significant difference to the performance of the cluster and it is important to understand the influence of such parameters. Based on our findings, we also identified some of the open issues in the existing area of research.

***Keywords*-MapReduce, Task Scheduler, Performance**

I. Introduction

MapReduce [1] is a software framework for distributed processing of large data sets on compute clusters, where computation is divided into a map function and a reduce function. Hadoop[1] is an open source framework that implements the MapReduce parallel programming model. Hadoop consists of a MapReduce engine and a user-level filesystem [2] that manages storage resources across the cluster. These two components are briefly described below.

- MapReduce Engine: It consists of one master JobTracker and one slave TaskTracker per cluster-node. The master is responsible for scheduling and monitoring the jobs' component tasks on the slaves and re-executing failed tasks. The slaves execute tasks as directed by the master according to the task scheduling algorithm.
- HDFS: The Hadoop Distributed File System (HDFS) provides global access to files in the cluster [2]. HDFS is implemented by two services: the NameNode and DataNode. The NameNode is responsible for maintaining the HDFS directory tree, and is a centralized service in the cluster. The DataNode(s) store the blocks of data. The Secondary name node is another component of Hadoop and it performs periodic checkpoints. Figure 1 shows the Hadoop framework.[2]

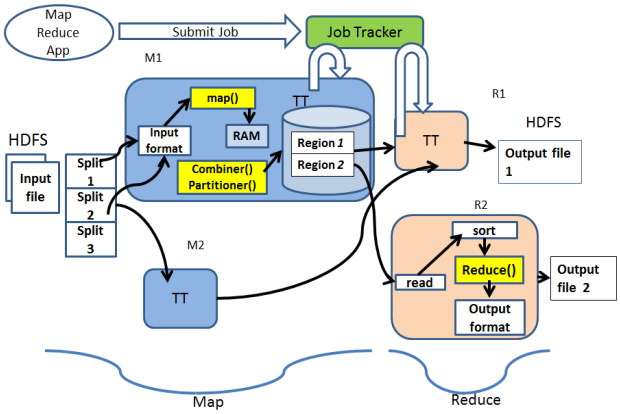


Figure 1. Hadoop Framework (http://4.bp.blogspot.com)

Existing research [3], [4] shows that the performance of MapReduce applications depends on the cluster configuration, job configuration settings and input data. The focus is mainly on the FIFO scheduler. We focus on the Capacity Scheduler, which shares the computing resources in a cluster between concurrent jobs. We performed simple experiments on representative MapReduce applications.

We study each major Capacity Scheduler configuration parameter in detail and demonstrate that it is important to understand their significance and fine tune the task scheduler settings to meet user needs. To the best of our knowledge, we are the first to study the impact of task scheduler settings on MapReduce applications with the Capacity Scheduler. However, one of the main intentions of this work is to bring to the fore some of the important open questions regarding MapReduce performance and the need for better tools to model and simulate scheduling behaviour.

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

[2]http://4.bp.blogspot.com/

It is not the goal of this work to optimize the task scheduler parameters to get optimal performance from the cluster for running MapReduce applications, but to show that these parameters have a significant impact on system performance and application response time. Through our experiments, we have found that some parameters have obvious effects (queue capacity and priority), but the effects differ between our benchmark applications. Some have no discernible effect and are not described in the paper, while yet other parameters interact in interesting and surprising ways. In particular, the user limit percentage had a non-intuitive effect on reduce time and interacted with the job scheduling parameters producing unexpected results.

The reminder of this paper is organized as follows. Section II introduces the Capacity Scheduler and its various configuration parameters. Related work is covered in Section III. Section IV contains details of experimental testbed and evaluation methodology. In Section V, we describe the experiments and evaluate the results. In section VI, we discuss the implications of our results. Finally, Section VII shows our conclusions and future work.

II. Background

The task scheduler runs on the job tracker 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 task schedulers in Hadoop:

- FIFO: Initially, Hadoop was used mainly for running large batch jobs such as web indexing. All the users submit jobs to a single queue, and jobs are executed sequentially.
- Fair Scheduler: The Fair Scheduler[3] arose out of Facebook's need to share its data warehouse between multiple users. Resources are allocated evenly between multiple jobs and capacity guarantees for production jobs are supported. It has pools where jobs are placed and each pool have a guaranteed capacity. Excess capacity is allocated between jobs using fair sharing.
- Capacity Scheduler: The Capacity Scheduler[4] from Yahoo offers similar functionality as the Fair Scheduler, but in a different way. In the Capacity Scheduler, one has to define a number of named queues. Each queue has a configurable number of map/reduce slots. The scheduler gives each queue its guaranteed capacity when it contains jobs, and shares unused capacity between the queues. However, within each queue, FIFO scheduling with priorities is used, except that one can limit the percentage of running tasks per user, so that users share a cluster equally. 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.

[3]http://hadoop.apache.org/common/docs/r0.20.2/fair_scheduler.html

[4]http://hadoop.apache.org/common/docs/r0.20.2/capacity_scheduler.html

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 are setup as min(number of queues, init-worker-threads). A single thread can handle multiple queues. All the jobs admitted into the system are not initialized instantly to reduce the memory footprints on the job tracker.
3) When a job is submitted to the cluster, the scheduler checks for job submission limits to determine if the job can be accepted or not (based on 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 the job can be initialized it is submitted to the assigned worker thread for the queue which initialize the job.
5) Whenever the job-tracker gets the heartbeat from a task tracker, a queue is selected from all the queues. 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. "max-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 task over rack-local task. This procedure is repeated until all jobs complete.

Both the Capacity Scheduler and the Fair Share Scheduler offer various configuration parameters, allowing administrators to tune scheduling parameters for the jobs. Table I shows the two types of configurable parameters available for the Capacity Scheduler:

1) Resource allocation (first 5 parameters), and
2) Job initialization (last 6 parameters).

There are other parameters which can be categorized under memory management. We have not taken them into account as they are only supported in Linux and only have an impact on memory intensive jobs.

Table I
CONFIGURABLE PARAMETERS FOR CAPACITY SCHEDULER

Parameter Name	Brief Description and Use	Default Value
queue.*q-name*.capacity	Percentage of the number of cluster's slots available for jobs in this queue.	1 queue@100%
queue.*q-name*.maximum-capacity	Limit beyond which a queue cannot use the capacity of the cluster.	-1
queue.*q-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.*q-name*.user-limit-factor	Queue capacity multiple to allow a single user to acquire more slots.	1
queue.*q-name*.supports-priority	If true, priorities of jobs will be taken into account in scheduling decisions.	False
maximum-system-jobs	Maximum number of jobs in the system which can be concurrently initialized	3000
queue.*q-name*.maximum-initialized-active-tasks	Maximum number of concurrently initialized tasks, across all jobs, all users	200000
queue.*q-name*.maximum-initialized-active-tasks-per-user	Limit of concurrently initialized tasks per-user, for every job of a user	100000
queue.*q-name*.init-accept-jobs-factor	The multiple of *(maximum-system-jobs*queue-capacity)* used to determine the number of jobs which are accepted by the scheduler.	10
init-poll-interval	Time between polls of the scheduler job queue to initialize new jobs	5000 ms
init-worker-threads	Number of worker threads used by Initialization poller to initialize jobs	5

III. RELATED WORK

Jiang, Ooi, Shi and Wu [5] studied the performance of MapReduce in a very detailed manner. They identified five design factors that affect the performance of Hadoop: 1) grouping schemes, 2) I/O modes, 3) data parsing, 4) indexing, and 5) block-level scheduling. By carefully tuning these factors, the overall performance of Hadoop improved by a factor of 2.5 to 3.5. Wang, Butt, Pandey and Gupta developed a simulation tool named MRPerf [4], [6]. MRPerf provides a means for analyzing application performance on a given Hadoop setup, enabling the evaluation of design decisions for fine-tuning and creating Hadoop clusters.

To ease the task of evaluating and comparing different MapReduce provisioning and scheduling approaches, another simulator named SimMR [7] was proposed. Babu [8] showed that the presence of too many job configuration parameters in Hadoop is cumbersome and highlighted the importance of an automated tool for optimizing parameter values. Herodotou *et al.* [9] introduced Starfish, which profiles and optimizes MapReduce programs, based on cost. The main aim of Starfish is to relieve users of trying to fine tune the job configuration parameters for MapReduce applications in any cluster settings and input data. MRSim [10] is yet another MapReduce simulator based on discrete event simulation, which can estimate job performance and capture job completion time/hardware utilization.

All the work described earlier pointed out that the task scheduler makes an important contribution to the performance of MapReduce applications but none of them delve into it in detail. We are trying to fill that gap with a detailed performance study. It is assumed that task scheduler choice has an impact but the nature and reasons for the effect it makes and the influence of task scheduler configuration parameters remains largely unexplored.

IV. EVALUATION METHODOLOGY

We conducted experiments on an isolated 6-node cluster. One node was designated as the job-tracker and name-node. The other 5 nodes carry out the tasks of data node and task tracker. All the nodes have 128Gb of hard disk with 2xQuad core Intel Xeon L5420 processors at 2.5GHz, 8 GB of RAM and a 1 Gbps network connection. RedHat 5.3 Linux is the OS on all the nodes, executing Hadoop 0.20.203. Measurements directly on hardware allows us to clearly isolate and identify the performance variations caused by the task scheduler settings.

TeraSort, *Sort*, and *WordCount* Sort were used as benchmark MapReduce applications for the experiments. *TeraGen*, *RandomWrite* and *RandomTextWrite* 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 number of task trackers is 5 and there are 10 map and 10 reduce slots (i.e. 2 slots/node for each phase). We chose short jobs because current studies shows that short jobs are popular in practice [11]. Nevertheless, we argue that the results obtained from our paper will also be valuable for longer jobs and for jobs running in big cluster. With longer jobs or a set of short and long jobs in a big shared cluster, the contention for resources will be prevalent, which shall lead to impact on the job's performance when the administrator changes the parameter settings present in the Capacity Scheduler. The effects can vary but they will definitely be present because the main concept behind the Capacity Scheduler is the allocation of tasks to slots.

For our experiments, we use the *execution time* as the performance metric. The execution time includes time from submission to the completion time. We also show the individual Map and Reduce execution time. The time shown in our results is an average of 3 runs. In all the runs for a particular setting, there was not much variation except while testing the minimum user limit percentage where we observed two modes of distribution. This phenomenon is described in more detail later in Section V. The experiments were designed keeping in mind the Capacity Scheduler configuration settings. Multiple users were supported to start

the jobs at the same time. We created a simple workload generator which submits the job in every defined queue of the scheduler.

V. EXPERIMENTAL RESULTS

This section discusses our experiments and results. In these experiments, we changed the Capacity Scheduler configuration settings one by one and observed their impact on the job execution time. Under these experiments the number of queues ranged from 1 to 3. Each queue has 3 users and each one of them submits one job, one after another at the interval of one second. 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 results shown in the paper. Subsequent experiments use the same settings in terms of the number of users per queue and their job submission pattern, unless stated otherwise.

The number of Map tasks depends on the size and type of input. The Hadoop framework decides at runtime how many Map tasks to create. In our experiments, we observed 70-80 Map tasks for each job. 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, we used the default values for job configuration parameters. The primary reason for choosing default values for job configuration parameters was that we want to show that task scheduler parameters have an effect on job performance even if one does not change job configuration parameters, and secondly, our main focus in this work was on task scheduling parameters.

The results graphs show the 2 components of execution time, as well as total execution time: *Map Time* and *Reduce Time*. The total execution time is not the sum of the 2 components, since there is overlap between the map and reduce phases of a job.

A. Impact of Resource Allocation Parameters

In these experiments, we changed the parameters of the Capacity Scheduler related to cluster resources, such as the number of queues, their capacity and user related queue configuration settings. 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. The wait time in these experiments for all the jobs was under 10 seconds.

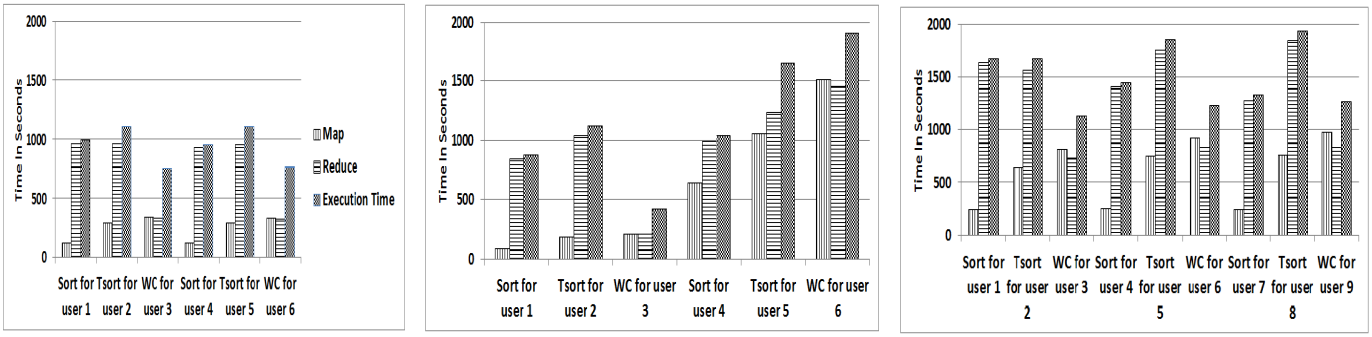
1) NumberofQueues and QueueCapacity: QueueCapacity is the guaranteed capacity which a queue will have at any time. Figure 2 shows the execution times for the various jobs under different settings. We can observe that the execution times of jobs depend on the value of these parameters. The time for each job increases with *NumberofQueues* and decreasing *QueueCapacity* due to increasing contention for shared resources like disk and network bandwidth.

To check the impact of increasing *NumberofQueues*, we compare Figure 2(a) and 2(c). The map, reduce and total execution times for all jobs increase significantly. The effect of *QueueCapacity* can also be clearly seen between Figure 2(a) and 2(b). For the same number of jobs, we observed different execution times. The jobs being executed in queue 1 get more slots and hence a reduction in their execution times. Note that this improvement is mainly because more map slots are available for that queue. The number of reduce tasks 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 2(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 times. The speculative execution was not enabled in our experiments.

2) MaximumCapacity: 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 MaximumCapacity (100% when the default value of -1 is used). The value of *MaximumCapacity* has to be at least *QueueCapacity*. Figure 3 shows the execution times for the various jobs under different settings. The first set of bars shows the execution times for 2 queues, each having 50% capacity while 2nd set has 2 queues with first having 90% and other queue having 10% *MaximumCapacity*. In both cases, *MaximumCapacity* is equal to *QueueCapacity*.

Comparing the first set of bars in Figure 3 with Figure 2(a), we observed that setting *MaximumCapacity* 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; it is available in both 50% configurations without or with *MaximumCapacity* enforcement. For second set of bars in Figure 3, the execution times for jobs in queue 1 remain similar to Figure 2(b) for the reasons explained before. However, for the second queue, the execution times increase 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.

We also observed in the second set of graphs that the Map phase execution time is greater than reduce phase execution time after the first job because, as mentioned earlier, a job typically has multiple map tasks, but one reduce task; when there is only one map slot present, 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 gets started when the



(a) 2 queue 50% each capacity (b) 2 queue 90%-10% capacity (c) 3 queue 33% each capacity

Figure 2. Effect of changing *NumberofQueues* and *QueueCapacity* (3 users/queue, 1 job/user)

first job finishes its map phase. So, 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.

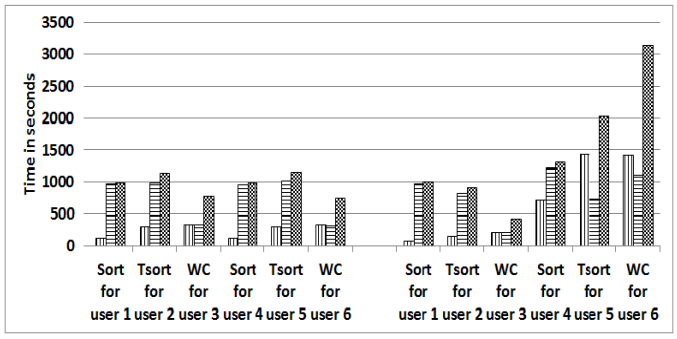


Figure 3. Effect of Changing *MaximumCapacity* on running time of jobs. First 6 bars represent 2 queues, 3 jobs/queue, 1 job/user, 50% capacity to each queue. Next 6 bars represent 2 queues, 3 jobs/queue, 1 job/user, 90%-10% capacity to queues.

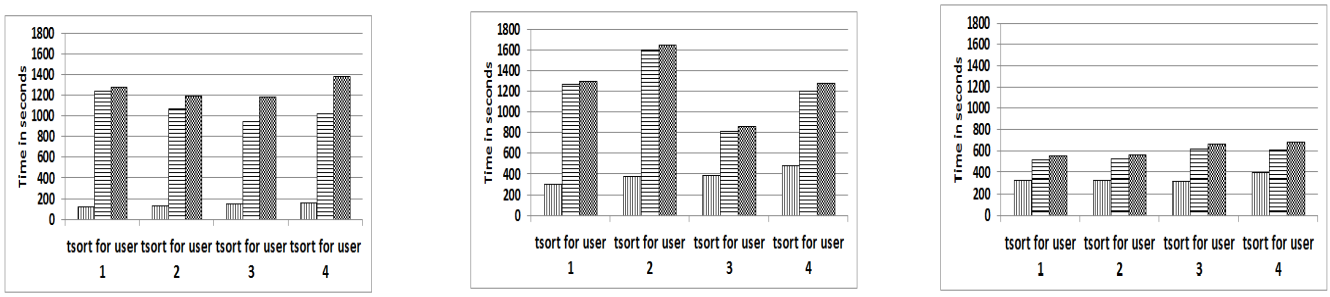
3) MinimumUserLimitPercent: This parameter allows limits to be defined on per user allocation of resources for a given queue. To check the impact of this parameter, we used a single queue, 100% capacity and 4 users. Each user simultaneously submitted one *TeraSort* job. Figure 4 shows the execution times for the various jobs. We can see that the Map execution times for all jobs increase significantly when *MinimumUserLimitPercent* is changed from 100% to 25%. This is because when *MinimumUserLimitPercent* is 100%, the first jobs 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 *MinimumUserLimitPercent* is 25%, however, each job starts the Map phase at the same time which also leads to increase in Map execution time as there are fewer Map slots.

We saw completely unexpected results for the Reduce time. We observed two different modes of distribution in *reduce* execution time in our results. The reduce execution time hovered around 14 minutes for the jobs in one mode of distribution and in the other the reduce time hovered around 25-29 minutes. A closer look at the logs revealed that it was caused by scheduling of two jobs simultaneously on two different slots on the same cluster node. Recall that every node has 2 map and 2 reduce slots. So, 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. Shuffle is a sub-phase during the reduce phase where sorted output of the mappers is transferred to the reducer over the network via HTTP. During sorting, the output from different mappers coming to the reducer is sorted based on keys.

In our experiments, we found that job 3 in Figure 4(b) was always executed on a cluster node where there was no other job's reduce task was running and hence we see a decrease in reduce time, compared to Figure 4(a). In Figure 4(a), the reduce task of job 3 was sometimes scheduled with other jobs on same cluster node. We consider this as a stochastic effect, because apart from the exception for job 3 in Figure 4(b), the reduce phase of all other jobs were scheduled with another job reduce task on same node at some time.

In figure 4(c), we showed the interaction between job configuration parameters (number of reduce tasks) and scheduling parameters (user limit factor). 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 times of jobs is reduced as more parallelism is achieved.

4) UserLimitFactor: This parameter allows a single user to acquire more slots than the configured queue capacity.



(a) User limit 100%

(b) User limit 25%

(c) User limit 25%, 5 reduce tasks/job

Figure 4. Effect of changing *MinimumUserLimitPercentage* (1 queue, 100% *QueueCapacity*, 4 user jobs)

Figure 5 shows the execution times for the various jobs. The number of reduce tasks per job and user limit factor was varied in these experiments. There is no change in execution times of jobs between Figure 5(a) and 2(a). This is because although *UserLimitFactor* 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 got it through its allocated queue capacity; *UserLimitFactor* was not relevant.

However, we see a big difference between Figure 5(b) and 5(c) here. The execution times for jobs in the second queue are much higher in 5(b) than in 5(c). The reason is the value of *MinimumUserLimitFactor*. In Figure 5(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(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 the 1^{st} queue only need 6 reduce slots and 4 are free) which leads to substantial reduction in execution time.

5) 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, HIGH, NORMAL and VERY_HIGH. We found that high priority jobs have shorter execution times than lower priority jobs, depending on the priority type. Figure 6 shows the execution times for the various jobs. Compared with Figure 2(a), we can see that priorities clearly affect job execution times.

B. Impact of Job Initialization Parameters

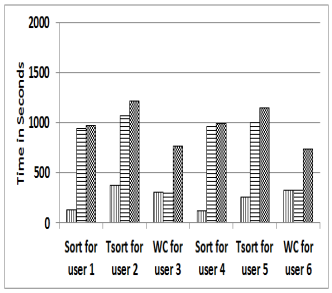
In these experiments, we changed the job initialization parameters in the Capacity Scheduler. These parameters determine the number of system jobs, tasks per queue and tasks per user which can be executed concurrently on the cluster. The *real execution time* (excluding from submission time to launch time) is used as a metric in some of the results in this section to isolate the effects of the waiting time.

1) MaximumSystemJobs: In this experiment, we varied the number of *MaximumSystemJobs* from 2 to 4, respectively, to observe the impact on the execution times of the jobs. Figures 7 and 8 show the execution times for the various jobs under different settings.

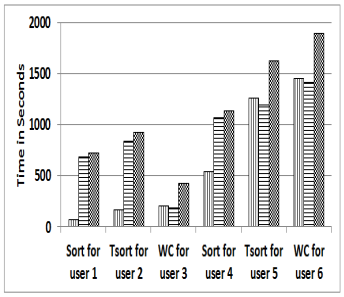
When the value of *MaximumSystemJobs* is 2, only a single job from each queue gets executed in parallel. As a result, the first *Sort* job in both queues has a better execution time than in Figure 8. When one job in a queue finishes execution, the second one starts executing, causing a long launching time for subsequent jobs. They have to wait a lot before they start executing. That is why we see long execution 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 times. 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 times because they do not have to wait as long as in the previous scenario. *WordCount* also has a faster response time as it was the last executed job in this scenario and did not face much competition from the other jobs which had completed. The real execution time for *WordCount* was similar in both scenarios and the difference observed is due to waiting time.

2) MaximumInitializedActiveTasks and MaximumInitializedActiveTasks/user: Both these parameters are related. *MaximumInitializedActiveTasks* 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. *MaximumInitializedActiveTasks/User* per user cannot be greater than *MaximumInitializedActiveTasks*. Figures 9 and 10 show the execution times for the various jobs under different settings. We can see the same trends for all the jobs in both the figures for the same reasons as explained earlier for *MaximumSystemJobs*.

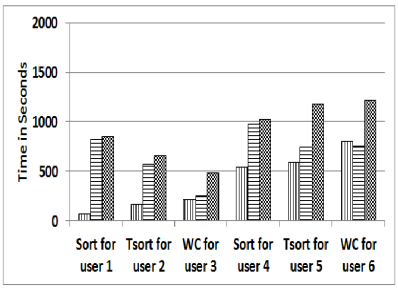
The reason for choosing 80 active tasks per user was that all the jobs had between 75 and 80 Map tasks. In scenario 1, when *MaximumInitializedActiveTasks* for a queue is 100, then only a single job can be run from that queue as $\lfloor \frac{100}{80} \rfloor =$



(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. Effect of changing user limit factor (3 users/queue and 1 job/user)

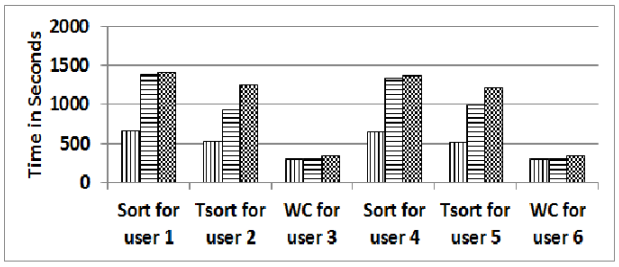


Figure 6. Execution times for 2 queues, 3 users/queue,1 job/user, 50% capacity each. All queues support priority. Priority of Sort=VERY_LOW, TeraSort=NORMAL, WordCount=VERY_HIGH

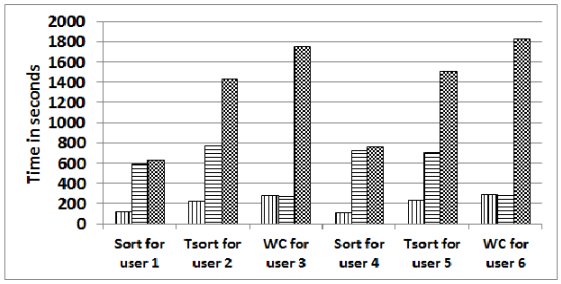


Figure 7. Execution times for 2 queues, 3 users/queue, 1 job/user, 50% capacity each, max system jobs=2

1. Hence, we get large waiting times for all jobs after the first job, but short real execution times. In scenario 2, 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 times, due to contention of resources except *WordCount* which was the last job to run in both the scenarios.

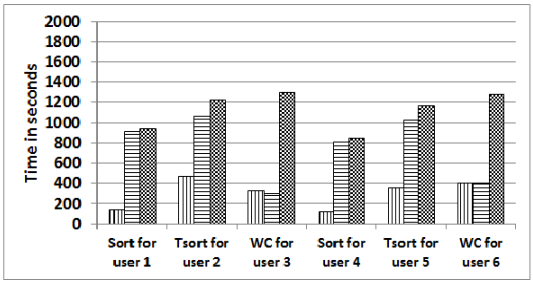


Figure 8. Execution times for 2 queues, 3 users/queue, 1 job/user, 50% capacity each, max system jobs=4

VI. DISCUSSION

Careful selection of scheduler configuration parameters is crucial to reduce the execution times 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 our experiments. The Capacity Scheduler has been around for a while 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[5] 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.

Yahoo has developed a simulator named Mumak,[6] which

[5]http://lucene.472066.n3.nabble.com/Hadoop-lucene-users-f647590.html
[6]https://issues.apache.org/jira/browse/MAPREDUCE-728

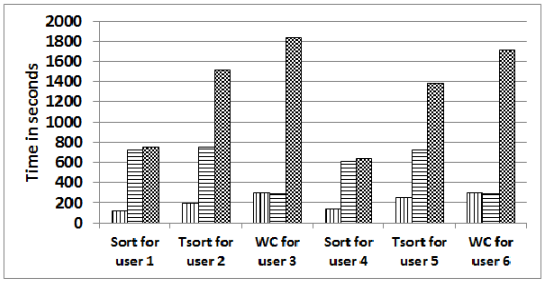


Figure 9. Execution time for 2 queues, 3 users/queue and 1 job/user, 50% capacity each, maximum-initialized-active-tasks=100, maximum-initialized-active-tasks-per-user=80

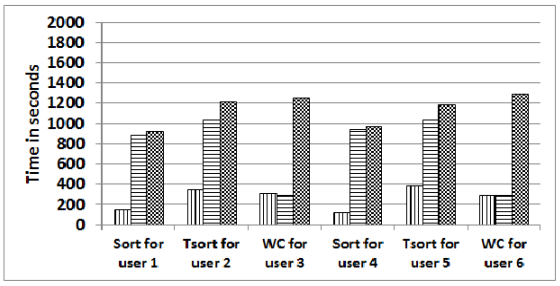


Figure 10. Execution time for 2 queues, 3 users/queue and 1 job/user, 50% capacity each, maximum-initialized-active-tasks=160, maximum-initialized-active-tasks-per-user=80

can do the task of identifying performance of MapReduce applications under different schedulers. Mumak takes a job trace derived from production workload and a cluster definition as input, and simulates the execution of the jobs as defined in the trace in the virtual cluster. However, this simulator cannot predict the performance for jobs when the cluster configuration is changed. It also does not simulate the shuffle phase, hindering its accuracy. Starfish [9] analyzes performance of MapReduce applications under changing cluster and job configuration settings. Unfortunately, it only supports the FIFO scheduler.

The performance modelling of Hadoop to date shows that the performance of a job in a cluster depends on Job configuration settings, cluster configuration and input data. However, through our experiments we showed that the execution time of jobs in certain environment depends on both job and task scheduler configuration parameters. Therefore, the performance model of Hadoop should include task scheduler configuration parameters as a key component.

VII. FUTURE WORK

Based on our findings, we are in the process of building a tool that will accurately predict the performance of jobs in a shared cluster under different settings for the Capacity Scheduler. It shall help administrators to find the optimal values and save their time before doing any resource allocation changes in the cluster. Due to economic challenges, it is possible that an organization sharing a cluster may want to switch to a different resource allocation scheme. In such cases, such a tool can be handy and let the organization know in advance that how much its work will be impacted if they change their budget.

ACKNOWLEDGMENT

The authors would like to acknowledge the support of HPC team at the University of Saskatchewan and the Natural Science and Engineering Research Council of Canada.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *CACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *2010 Symposium on Massive Storage Systems and Technologies*, Incline Village, NV, May 2010, pp. 1–10.

[3] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of MapReduce programs," *VLDB Endowment (PVLDB)*, vol. 4, no. 11, pp. 1111–1122, 2011.

[4] G. Wang, A. Butt., P. Pandey, and K. Gupta, "A simulation approach to evaluating design decisions in MapReduce setups," in *MASCOTS '09*, London, UK, Sep. 2009, pp. 1–11.

[5] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of MapReduce: an in-depth study," *VLDB Endowment (PVLDB)*, vol. 3, no. 1–2, pp. 472–483, Sep. 2010.

[6] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, "Using realistic simulation for performance analysis of MapReduce setups," in *Large-Scale System and Application Performance Workshop*, Garching, Germany, Jun. 2009, pp. 19–26.

[7] A. Verma, L. Cherkasova, and R. Campbell., "Play it again, SimMR!" in *CLUSTER 2011*, Austin, TX, Sep. 2011, pp. 253 –261.

[8] S. Babu, "Towards automatic optimization of MapReduce programs," in *SoCC '10*, Indianapolis, Indiana, Jun. 2010, pp. 137–142.

[9] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics," in *Conference on Innovative Data Systems Research*, Asilomar, CA, Jan. 2011, pp. 261–272.

[10] S. Hammoud, M. Li, Y. Liu, N. Alham, and Z. Liu, "MRSim: A discrete event based MapReduce simulator," in *International Fuzzy Systems and Knowledge Discovery Conference*, Yuntai, China, Aug. 2010, pp. 2993 –2997.

[11] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating MapReduce performance using workload suites," in *MASCOTS 2011*, Singapore, Singapore, Jul. 2011, pp. 390–399.