Learning Based Opportunistic Admission Control Algorithm for MapReduce as a Service

Jaideep Dhok
Internation Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad
jaideep@research.iiit.ac.in

Nitesh Maheshwari
Internation Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad
nitesh.maheshwari @research.iiit.ac.in

Vasudeva Varma
Internation Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad
vv@iiit.ac.in

ABSTRACT

Admission Control has been proven essential to avoid overloading of resources and for meeting user service demands in utility driven grid computing. Recent emergence of Cloud based services and the popularity of MapReduce paradigm in Cloud Computing environments make the problem of admission control intriguing. We propose a model that allows one to offer MapReduce jobs in the form of on-demand services. We present a learning based opportunistic algorithm that admits MapReduce jobs only if they are unlikely to cross the overload threshold set by the service provider. The algorithm meets deadlines negotiated by users in more than 80% of cases. We employ an automatically supervised Naïve Bayes Classifier to label incoming jobs as admissible and non-admissible. From the list of jobs classified as admissible, we then pick a job that is expected to maximize service provider utility. An external supervision rule automatically evaluates decisions made by the algorithm in retrospect, and trains the classifier. We evaluate our algorithm by simulating a MapReduce cluster hosted in the Cloud that offers a set of MapReduce jobs as services to its users. Our results show that admission control is useful in minimizing losses due to overloading of resources, and by choosing jobs that maximize revenue of the service provider.

Keywords

Cloud Computing, Software as a Service, MapReduce, Admission Control

1. INTRODUCTION

Cloud computing has been recognized as a one of the prominent new computing paradigms. The ability of cloud to provide on demand access to software, application platforms and infrastructure in the form of scalable services has attracted considerable interest in the academic community as well as in industry. MapReduce [14] has emerged as the paradigm of choice for developing large scale data intensive applications in the cloud. The state of the art in running MapReduce in the cloud is in the form of the Elastic MapReduce service by Amazon [2], which falls under the so called Platform as a Service (PaaS) paradigm, where users can submit their applications developed in the form of MapReduce jobs and resources necessary for performing computations are assembled on the fly.

Although PaaS has its own advantages, MapReduce when offered in the Software as a Service (SaaS) paradigm can prove useful to users as well as the service providers, as users can reuse MapReduce components and service providers can expose MapReduce jobs as pay-per-use services which are frequently in demand as building blocks for performing data intensive computations. Service providers rent computational resources from an infrastructure provider, and allow users to run ready to use services offered by the service provider. Effective admission control mechanisms are necessary in this setting in order to maximize utility from the perspective of service providers, and to ensure quality of the services for users [11, 12]. Admission control has been essential in preventing overload of computational resources thereby maintaining a guaranteed level of service.

In this paper we present a method for modeling MapReduce jobs as ready to use services, thus effectively bringing MapReduce in the Software as a Service paradigm. We extend the utility models proposed in previous work in order to adopt it to MapReduce. We also address the problem of admission control in this paradigm, particularly for Hadoop [4] which has emerged as the leading MapReduce implementation. The admission control algorithm that we propose uses a novel machine learning based approach for predicting job admission. The algorithm trains itself according to policy rules set by the service provider.

Most literature in this field has dealt with solving the problem of admission control for publicly available computational grids, where users can run arbitrary jobs and pay for the resources consumed by their applications [9, 13, 15, 16]. In our model, service providers offer a limited set of MapReduce jobs as web services. Existing resource management algorithms in Hadoop lay their focus on policies for sharing of resources and multiplexing job execution instead of focusing on maximizing user and service provider utility. Further, existing systems allow limited or no support for admission control.

The remainder of this paper is organized as follows. We

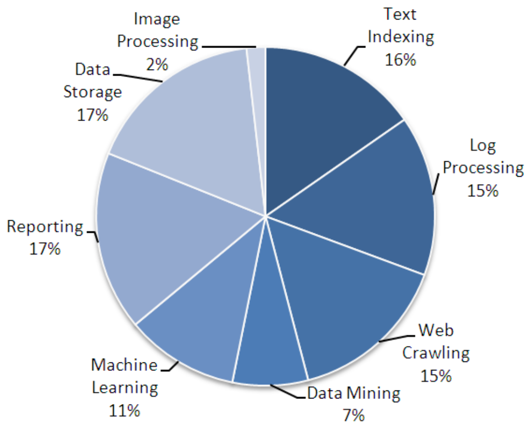

Figure 1: MapReduce has seen tremendous growth in the recent years especially for data intensive computing. The figure presents popular uses of MapReduce as gathered from the Hadoop PoweredBy page [7]. (Percentages in the figure are approximate)

present the MapReduce as a Service paradigm and service contracts in Section 2. Next, we give a brief overview of scheduling in Hadoop, and present our admission control algorithm in Section 3. Section 4 describes the methodology and simulation models used in evaluating our algorithm and the results of evaluation. We then compare our work with existing work in Section 5. Finally, we conclude by summarizing the key contributions, and touching on future topics in this area in Section 6.

2. MAPREDUCE IN SAAS PARADIGM

Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) are the three key paradigms that enable cloud computing. In these models, software applications, software platforms and infrastructure are provided to the users in the form of on-demand services, and they are charged according to the pay-per-use model. MapReduce, which has become a popular paradigm for large scale data processing in the Cloud, is usually associated with the PaaS paradigm, where the service provider offers a ready to use MapReduce cluster where users can run their jobs. An example of such a platform is the Amazon Elastic MapReduce service [2] where users can provision Hadoop clusters on the fly, and perform data intensive computation by providing the implementation of Map and Reduce components, and process data that is hosted in Cloud based storage services.

2.1 MapReduce Jobs as Services

We propose an extension to the above model that brings MapReduce in the SaaS paradigm. In our model, service providers offer a set of MapReduce applications as Web Services. For performing data intensive operations, users search from a repository of registered services and select a service that performs the desired operation. The service repository thus takes form of an online market place where users can choose from a range of MapReduce services. A market is beneficial to users, as the service providers are forced to provide better service at cheaper rates in order to overcome competition. Figure 2 shows the architecture.

MapReduce in the SaaS paradigm has following benefits from the users' perspective:

- Users can choose from a wide range of available applications to perform their computations, without having to invest in development of such applications.
- Users do not have to deal with establishing and maintaining MapReduce clusters, thus saving operational cost.
- Users can combine MapReduce services to form a data processing pipeline, where each unit in the pipeline could be offered by a different service provider, thus allowing the user to form a 'mashup' service.

After selecting a service, the users interact with it only through the web service interfaces exposed by the service provider. This helps in hiding implementation details from the uses, which could be beneficial to the service provider in cases where exposing application implementation details is not preferred. Messages exchanged between the user and the services are based on familiar transport mechanisms, for example, XML or JSON over HTTP. Users specify input data location in the cloud, metadata describing the computation to be performed on the input data and contractual service demands as parameters to the web service. Output data could either be obtained directly as a response from the service or it could be stored in the cloud. The latter approach is useful if the user desires to perform further computation by means of another service instance.

After accepting a request, a service provider then launches a MapReduce job corresponding to the request in her cluster that is hosted entirely in the cloud. The cloud in this case can be a private cloud that emulates cloud computing on privately owned infrastructure, or it could also be hosted in public cloud computing offerings such as Amazon EC2 [1], GoGrid, RackSpace Cloud etc. We assume that distinct requests are independent of each other, and thus could be completed in parallel. To increase revenue, the service provider processes multiple requests simultaneously by multiplexing job execution in the MapReduce cluster to achieve better resource utilization. The computational resources of the cluster are shared proportionately among the users. The proportion of resource allocated to a user's request depends upon the utility earned by the service provider after completing the user's request.

2.2 Example Use Cases

Having discussed the model, we now present few use cases of MapReduce as a service:

- *Ad-hoc querying on large datasets* - Consider a scenario where an online movie rental company is offering a large anonymized data set of its users' order history (of the order of few TBs) for analysis. The company can also offer several ready to use operations such as data selection, filtering, joins, pre-processing, etc. in the form of MapReduce jobs. Users can chain several such operations to extract useful informative patterns from the orders data such as set of genre of orders placed by users in a particular age group. Users do not have to own the data set, and they can reuse components developed by the service provider to extract desired information.

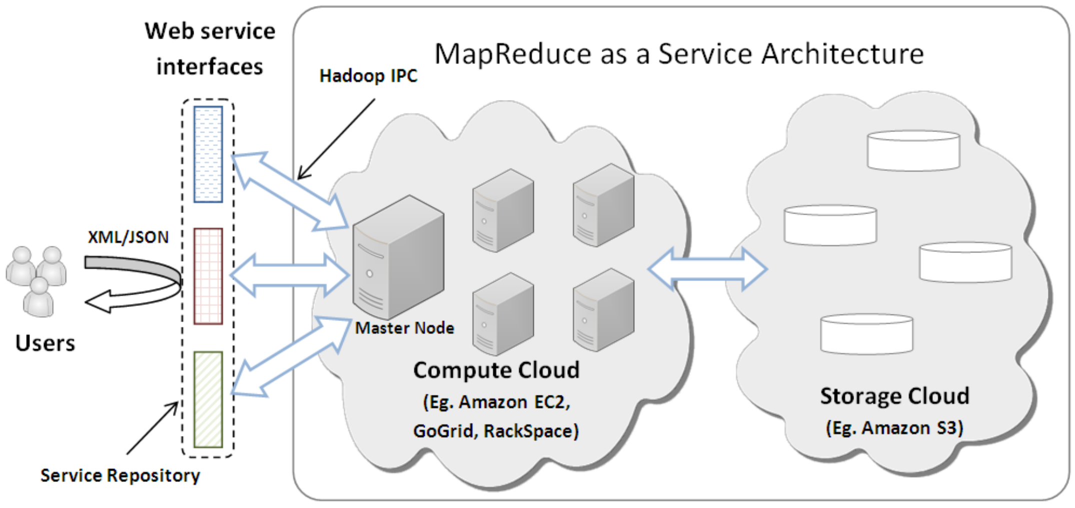

Figure 2: The architecture of MapReduce as a Service. Our model is based on the Hadoop open source MapReduce framework.

- *On demand crawling and indexing of web information sources* - A service provider could allow users to submit a list of seed URLs to be crawled using a domain specific crawling algorithm developed by the service provider. The crawling engine will utilize MapReduce jobs for distributing workload across multiple nodes. The service providers could also provide initial pre processing utilities in the form of MapReduce jobs such as jobs for extracting images and their alt-text and surrounding text from crawled web pages. Users can pay the provider only for the resources consumed during their crawl process, such as network bandwidth, disk space consumed by the crawled data etc.
- *Document Format Conversion Service* - In this example, users submit a list of documents stored in the cloud to the service provider, along with the desired output format for the documents. The service provider can then offer MapReduce jobs to convert documents in the desired file format. Example applications include on-demand video and audio conversion, generating thumbnails from video files etc.

2.3 Contractual Service Agreement

In MapReduce as a service model, users only pay for the share of resources consumed for their computation. Besides the demand of correctness of computation, deadline for performing the computation is also an integral part of users' expectation about the quality of service. Thus the price of the resources that the user is willing to pay and the deadline that the service provider agrees by, constitute the service contract in this model. The user and service providers must negotiate and mutually agree upon this contract. We do not address the problem of price determination in this paper; auctioning mechanisms such as the Dutch auction or the English auction could be used effectively for the purpose of judging the value of service.

Users specify utility functions that indicate the price they are willing to pay as a function of time taken to complete service request. We extend the generic three phase utility functions proposed by [9, 13, 15, 16]. In this framework, the users specify a soft deadline and a hard deadline. If the request completes before the soft deadline, a user pays the complete amount he/she agreed upon before submitting the request. After the soft deadline, the utility from the perspective of the user degrades, until the hard deadline, after which the user is no longer interested in the outcome of the request and is unwilling to pay for completion of the service. The decay in the utility could be linear, or the rate of decay could also vary with time passed since the soft deadline. The following set of parameters capture the set of utility functions that exhibit this behavior.

Formally, utility can be expressed as a function of time:

$$U(t) = \begin{cases} U_0 & \text{if } 0 < t \leq T_1 \\ U_0 - \alpha(t - T_1)^{\beta} & \text{if } T_1 < t \leq T_2 \\ U_P & \text{if } t > T_2 \end{cases}$$

where, t = 0 is the time when a service request is accepted. U_0 is the initial utility that the user is willing to pay if the request is completed before the soft deadline T_1, after which the utility decays until the hard deadline T_2. Users can control the values of decay parameters α and β. Finally, U_P gives the utility that the users are willing to pay after the hard deadline. A negative value of U_P implies a penalty to be incurred by the service provider for failing to meet the hard deadline. If U_P is zero after T_2, it means that the user is no longer interested in the outcome of the service, and thus will not pay any charges to the provider. The provider is thus free to cancel the request.

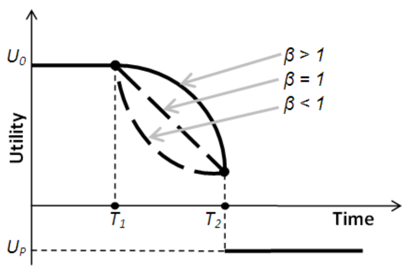

Figure 3: Utility Functions for different values of decay parameters.

The values of the decay parameters (α and β) represent the users interest in the outcome of the service request. A value of $\beta = 1$ gives a linear degradation in the utility if the job is not completed within the soft deadline. Similarly a value of $\beta = 0$ indicates a sharp drop off in the users interest if the soft deadline is missed. Decay functions for various values of α and β are shown in Figure 3.

The next section describes the need for admission control algorithms for the MapReduce as a Service model and our proposed algorithm.

3. PROPOSED ADMISSION CONTROL ALGORITHM

We attempt to solve the problem of admission control for Hadoop, which is a leading open source framework for MapReduce. We briefly mention the architecture of Hadoop MapReduce, and then proceed to our algorithm. First, let us consider the need for an admission control algorithm.

In our model, a service provider processes multiple requests simultaneously by multiplexing job execution in the cluster. Resources in the cluster are shared proportionately among the requests, and these proportions are decided by the utility that the service provider is expecting to earn after successful completion of a request. As a result, it becomes necessary to judiciously accept incoming jobs, so that incoming jobs do not affect the performance of already running jobs. Admission control also helps to prevent overloading of resources in the cluster. As the cluster is hosted in the cloud, the resources in the cluster could be scaled on-demand using auto-scaling capabilities. However, even if an auto-scaling facility is available, admission control can still prove viable because the rate of arrival of new requests could be much more than the rate of commissioning new nodes in the cluster.

3.1 Background: Hadoop Architecture

Hadoop's MapReduce implementation borrows much of its architecture from the original MapReduce system at Google [14]. Figure 4 depicts the architecture of Hadoop's MapReduce implementation. Although the architecture is centralized, Hadoop is known to scale well for small (single node) to very large (up to 4000 nodes) installations [8].

Scheduling decisions are taken by a master node (JobTracker), whereas the worker nodes (TaskTrackers) are responsible for task execution. The JobTracker keeps track of the heartbeat messages received periodically from the TaskTrackers and uses the information contained in them while assigning tasks to the TaskTracker. If a heartbeat is not received from a TaskTracker for a specified time interval, the TaskTracker is assumed to be dead. In such a case, the JobTracker re-launches all the incomplete tasks previously assigned to the dead TaskTracker. Task assignments are sent to the TaskTracker as a response to the heartbeat message. The TaskTracker spawns each MapReduce task in a separate process, in order to isolate itself from faults due to user code in other tasks.

3.2 Proposed Algorithm

The administrator specifies the maximum number of Map and Reduce task slots that control the number of simultaneously running tasks on a TaskTracker. Jobs compete for task slots in the cluster, and it is the responsibility of the scheduler to properly allocate slots so that jobs do not suffer from starvation, and they receive their fair share of the resources in the cluster.

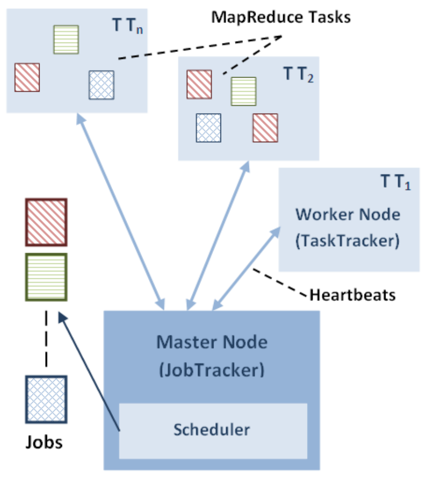

Figure 4: Architecture of MapReduce in Hadoop.

The admission controller runs at the master (JobTracker) node in the MapReduce cluster. Although user requests for services can arrive asynchronously, the algorithm considers them for admission only at fixed points in time. Time interval between two such admission points is referred to as an admission interval. Job requests arrived during an admission interval are maintained in the queue of candidate jobs. The algorithm takes this queue as input, and admits at most one job for execution in the cluster. All other requests are rejected and are not considered for further processing. The users are notified if their requested services have been accepted or rejected. Figure 5 summarizes the admission control block.

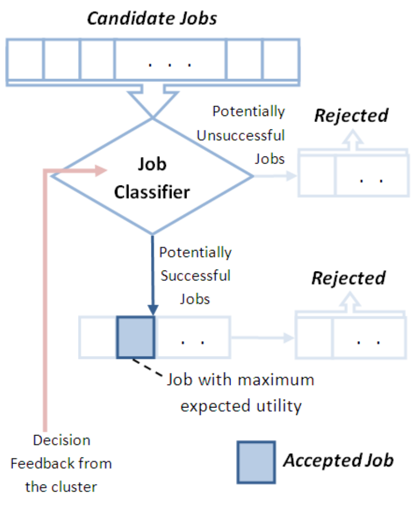
Candidate Jobs
Potentially Unsuccessful Jobs
Rejected
Job Classifier
Potentially Successful Jobs
Rejected
Job with maximum expected utility
Decision Feedback from the cluster
Accepted Job

Figure 5: Admission Controller.

To decide if and which request to accept, we use the Expected Utility Hypothesis from decision theory. This hypothesis states that given a set of choices with varying payouts and the likelihood of those payouts, a rational agent always prefers the option that maximizes the agent's expected utility. Applying this principle to the problem of selecting a job to be admitted, the algorithm chooses a job that maximizes expected utility from the perspective of the service provider. Formally,

$$Selected\ job = argmax_j(U_j \times P(J = Success|E))$$

where, U_j is the utility of the job as calculated from the utility function agreed upon by the user and the service provider in their service contract. While making the comparison, we consider only the utility that will be earned if the job is completed before the soft deadline specified by the user. $J = Success$ denotes the event that job admission is successful according to success criteria dictated by the service provider. The probability $P(J = Success|E)$ is conditional on the current state of the resources in the cluster, E.

The admission controller uses prior knowledge accumulated to make admission control decisions for predicting the outcome of admission of candidate jobs. To achieve this, we compute the posterior probability $P(J = Success|E)$ using Bayes Theorem:

$$P(J = Success|E) = \frac{P(E|J = Success) \times P(J = Success)}{P(E)}$$

The above equation forms the foundation of learning in our algorithm. The algorithm uses results of decisions made in the past to make the current decision. This is achieved by keeping track of past decisions and of their outcomes in the form of conditional probabilities.

The denominator $P(E)$ in the above equation is independent of candidate jobs and can be ignored safely as a constant while comparing the candidate jobs. For each job in the list, we estimate the probability of future success as well as future failure. A job is rejected if the likelihood of a failure is more than that of a success. If all jobs are likely to fail, none of the jobs are admitted. In other words, we classify the candidate jobs into potentially successful and potentially unsuccessful jobs, and then select the job that provides maximum utility from the set of potentially successful jobs. Figure 5 summarizes this process.

We thus select the job that maximizes the following quantity:

$$\frac{U_j \times P(E|J = Success) \times P(J = Success)}{P(E)}$$

The state of the environment E comprises of a number of factors describing the state of cluster resources such as cluster load, number of pending tasks currently in the cluster, the rate at which tasks are being completed, etc. We also extend the state of resources by including in it the properties of job request such as the size of request, mean run times observed in the past to complete similar requests, etc. Figures 6 and 7 list the cluster and job parameters and the reasoning for their inclusion in the state of the environment, E.

The quantity $P(E|J = Success)$ thus becomes:

$$P(E|J = Success) = P(e_1, e_2, e_3 ... e_n|J = Success)$$

where, e_1, e_2, . . . , e_n are the factors constituting the state of the environment E.

Parameter	**Description**
Used map slots	Ratio of number of map tasks currently running to the maximum allowed number of concurrent tasks in the MapReduce cluster (quantifies the availability of resources)
Used reduce slots	Same as above, but for reduce tasks
Pending maps	Number of map tasks currently waiting for slots to be allocated (quantifies the pending map workload)
Pending reduces	Same as above, but for reduce tasks
Finishing jobs	Number of jobs that are about to finish i.e. having very few pending tasks. If the value of this parameter is high, the newly accepted job is expected to have sufficient resources for its execution
Map time average	Moving average of map task runtimes (denotes the rate at which map tasks are being completed.
Reduce time average	Same as above, but for reduce tasks
Load	Ratio of number of tasks waiting to be assigned a slot to the maximum number of slots

Figure 6: Cluster Parameters included in E

We assume that the probabilities of these factors are conditionally independent of each other (the Naïve Bayes assumption). Thus,

$$P(E|J = Success) = \prod_{j=1}^{n} P(e_j|J = Success)$$

Service providers predefine the criteria for success or failure of a job. For example, the service provider could specify that any new admission that results in overloading of resources of the cluster beyond a specified threshold will be considered as a failure. Success and failure rules are used to validate a decision, based on the effects of the current decision. Validation rules cannot be applied until data about the impact of a decision is available. The results of these validations are sent as feedback to the admission controller. Upon receiving the feedback, the algorithm updates its probabilities so that mistakes made by the algorithm, if any, are not repeated in the future.

It is possible that an admission decision can adversely affect the makespan of already running jobs. However, the decision will be considered invalid only if it does not meet the success or failure criteria set by the service provider. Service providers could define success-failure criteria that consider the effect on makespan of other jobs as well.

Our algorithm is greedy, as we choose the job that seems to provide maximum utility from the immediately available choice. It is also opportunistic, as we are willing to suffer degradation of performance of existing jobs, if the newly admitted job can offer more utility compared to utility gained

Parameter	Description
Job maps	Number of map tasks in the candidate job (depends on the size of input data)
Job reduces	Number of reduce tasks in the candidate job
Mean map time	Mean map task runtime observed for this job in its past runs
Mean reduce time	Same as above, but for reduce tasks of the job

Figure 7: Job Parameters included in E

from these already executing jobs.

4. EVALUATION AND RESULTS

To verify the efficacy of our algorithm, we simulated the Hadoop MapReduce architecture and studied the behavior of our algorithm with the following baseline approaches:

- *Myopic* - In this approach, the job with maximum initial utility is accepted without other considerations
- *Random* - A job is admitted randomly from a given set of candidate jobs. The given set of jobs is appended with a null value to simulate job rejection.

4.1 Simulation Model

In our simulation model, the properties of a job are distributions specifying runtimes of map and reduce tasks of a job. To model the distribution of runtimes, we extracted and observed real world MapReduce job traces of MapReduce jobs run on actual Hadoop clusters. We observed that Map runtimes of a particular job follow the Normal distribution with the mean and standard deviation being the characteristic of the job. Similarly for reduce tasks the runtimes for Sort, Shuffle and Reduce phases also followed the Normal distribution.

Based on these observations, a map task modeled in our simulation occupies a slot for a random amount of time which is chosen from a Normal distribution which is the characteristic of the job. Similarly each of the three phases in a reduce task modeled in our simulation also occupies a slot in accordance with Normal distributions which are again properties of the job. Our simulation does not model task failures as the utility is earned only after successful completion of a job request. Thus it is the responsibility of the service provider to make sure that all accepted jobs are executed successfully, irrespective of individual task failures. We only use the information that can be obtained through the JobTracker in Hadoop as the JobTracker provides a unified view of the MapReduce cluster. All the parameters mentioned in Figure 6 and 7 could directly be obtained from the JobTracker itself. Figure 8 lists the simulation parameters and distributions used in generating simulation events.

For comparing the results across different runs, we keep the pseudo random distribution parameters constant between runs. All values reported in the results are averaged over 10 independent runs, unless otherwise specified.

4.2 Learning Algorithm Accuracy

To verify whether the admission controller is able to accept/reject jobs in order to maintain overload threshold as specified by the service provider, we measured the actual load average observed in a simulation run, and compare it against the desired load average as set by the service provider. The plot below summarizes results of these experiments.

Parameter	Decription
Job arrival distribution	Exponential
Job arrival rate (λ)	5 minutes
JobTracker heartbeat interval	3 seconds
Admission interval	3 minutes
JobTracker map slots	50
JobTracker reduce slots	20
Job map size	Uniform Random (51, 100)
Job reduce size	<Job map size>/10
Simulation time	500 minutes
Decay parameters (α and β)	$\alpha = 1$, $\beta = 1$
Soft deadline (T_1)	Time taken when all map tasks are executed in parallel + Time taken when all reduce tasks are executed in parallel
Hard deadline (T_2)	Time taken when only one task of the job is executed at a time

Figure 8: Simulation Parameters

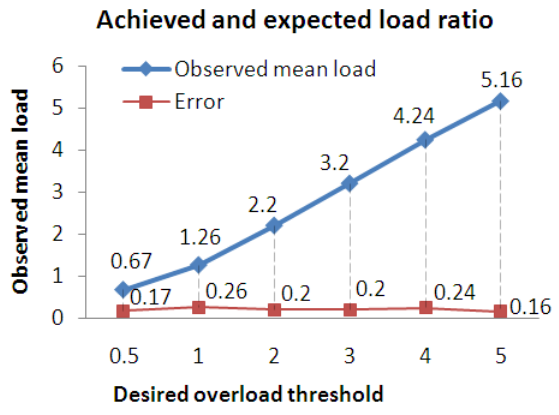

Figure 9: Achieved and expected load ratio

As we can see in the plot (Figure 9), the achieved load average value is fairly close, to the desired load average value. Further, the error rate is independent of the desired load average value. The errors may arise as a result of Naïve Bayes assumption made while computing posterior probabilities.

4.3 Comparison of Achieved Load Averages with Baseline Approaches

Next, we compare the performance of the learning our admission control algorithm with two baselines, as specified in the beginning of this section.

First, we compare the mean load averages observed in our algorithm to *Myopic* admission, and *Random* admission.

For this set of experiments, we kept the overload threshold to 100%. In other words, our admission controller rejected all those jobs which were predicted to cause the cluster load over 1.0. Figure 10 shows the results.

Algorithm	Achieved Load Average
Random	42.11
Myopic	42.09
Our algorithm	0.97

Figure 10: Comparison of Achieved Load Averages

As can be clearly seen in Figure 10, our admission control algorithm is very effective in preventing overload. This establishes the correctness of our algorithm, and proves our argument of the necessity of sophisticated admission control algorithms for MapReduce.

4.4 Service Contracts

The final experiments in our evaluation verify the ability of our algorithm in meeting user deadline guarantees. For this set of experiments, the values of decay parameters α and β were both set to 1, thereby making the decay rate linear. The soft deadline (T_1) in our case is the runtime of the job, if all tasks of the job are executed simultaneously. The hard deadline (T_2) is double the value of the soft deadline. To compare the algorithm with baseline approaches, we calculate the percentage of jobs that complete before the soft deadline, and the percentage of jobs that complete after the soft deadlines. We can see in Figure 11 that our algorithm is able to meet user QoS requirements in most of the cases, whereas the baseline approaches cause job runtimes to exceed soft deadlines in most of the cases.

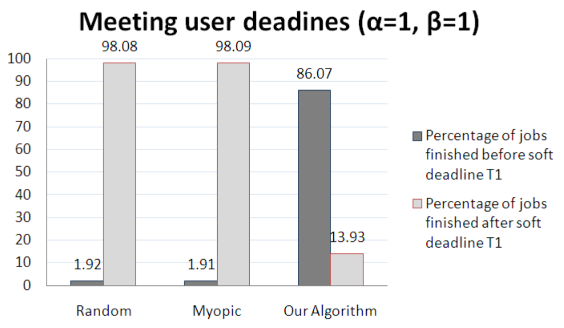

Figure 11: Performance while meeting user deadlines

5. RELATED WORK

We use the three stage utility model previously described in Risk Reward [15], Aggregate Utility [9], and Millennium [13]. These works discuss the problem of generic utility computing model where service providers rent resources from a third party in order to offer their services. Aggregate Utility allows users to control the behavior of the service providers by specifying aggregate utility functions. However, in our work we assume that individual requests of users are independent of each other. We also provide a more generic model for capturing user desires, especially to capture user disappointment, where as other works assume a linear decay in the utility. Unlike previous works, our algorithm employs machine learning, thus allowing the admission controller to make use of the effects of decisions made in the past, thereby continuously improving the performance of the algorithm.

An important difference between the works mentioned above and our work is that the previous works solve the problem of admission control where a service provider rents resources from an infrastructure provider, and allows users to execute arbitrary jobs. This model has become popular as the PaaS paradigm. We approach the problem for admission control for SaaS, where the service provider has complete knowledge about the services being offered.

MapReduce [14] presents MapReduce paradigm and its advantages for large scale data processing. Hadoop [4], which is a popular MapReduce implementation, bases much of its architecture and design decisions on the work in [14]. Our work extends the familiar usage of MapReduce for data processing by establishing a private cluster, to data processing by use of on-demand web services powered by MapReduce jobs. Amazon Elastic MapReduce [2] comes close to what we propose, however they offer MapReduce as a platform, whereas we present the model where MapReduce is offered as a service.

Work by Bichler and Setzer in [10] presents the problem of admission control for media on demand services, where they assume that service duration is of a fixed length, which is not the case for MapReduce jobs. They accept a service request if its revenue is higher than its opportunity cost, whereas in our approach we accept the service request with maximum expected utility.

Existing schedulers for Hadoop FAIR, Capacity, and Dynamic Priority [5, 6, 17] offer very limited facility of admission control. For example, the FAIR scheduler has a feature to suspend jobs until sufficient free resources are available. Existing schedulers focus on implementing resource sharing policies, within a MapReduce cluster owned internally by an organization. The dynamic priority scheduler uses market based approaches to control resources shared, by users in a cluster, however they also do not address the problem of admission control. Our approach borrows from the previous work on admission control in utility computing and provides a model more suited for cloud computing, in a scenario where MapReduce is offered as a service. In the next section we conclude by summarizing our results and mentioning future areas of research.

6. CONCLUSIONS AND FUTURE WORK

We presented a new paradigm of offering and utilizing MapReduce jobs in the cloud. The model we proposed is advantageous to users as well as potential service providers, willing to offer data intensive application as ready to use MapReduce services. We also presented a concrete mechanism for expressing user demands in the form of utility functions that capture users' perceived value of the service as a function of time.

We mentioned the need of an effective admission control algorithm, in order to meet our proposed paradigm. We have used a machine learning based approach, which utilizes experience gained in making admission control decisions the past. Our algorithm proved to be effective in achieving service provider expectations, as well as meeting quality of service requirements of users. Learning based approaches have rarely being tried for resource management in utility/grid/cloud computing.

Future directions of our research include investigating the use of machine learning in other resource management problems for example service brokering, and federation of resources across infrastructure owned by different organizations.

7. ACKNOWLEDGMENTS

This work is partly supported by a grant from Yahoo! India R&D, under the Nurture an area: Cloud Computing project. We would like to thank Chidambaran Kollengode, Jothi Padmanabhan and Preeti Priyadarshini of Yahoo! for their continual support and feedback. We would also like to thank ReddyRaja Annareddy of Pramati Technologies for his suggestions and Akshat Kumar for his help in debugging simulation source code.

8. REFERENCES

[1] Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/`.

[2] Amazon Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce/`.

[3] Amazon Simple Storage Service. `http://aws.amazon.com/s3/`.

[4] Apache Hadoop. `http://hadoop.apache.org`.

[5] Capacity Scheduler for Hadoop. `http://hadoop.apache.org/common/docs/current/capacity_scheduler.html`.

[6] Dynamic Priority Scheduler for Hadoop. `http://issues.apache.org/jira/browse/HADOOP-4768`.

[7] Hadoop PoweredBy. `http://wiki.apache.org/hadoop/PoweredBy`.

[8] Scaling Hadoop to 4000 nodes at Yahoo! `http://developer.yahoo.net/blogs/hadoop/2008/09/scaling_hadoop_to_4000_nodes_a.html`.

[9] A. AuYoung, L. Rit, S. Wiener, and J. Wilkes. Service contracts and aggregate utility functions. In *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pages 119–131, 0-0 2006.

[10] M. Bichler and T. Setzer. Admission control for media on demand services. *Service Oriented Computing and Applications*, 1(1):65–73, 2007.

[11] J. Broberg, S. Venugopal, and R. Buyya. Market-oriented grids and utility computing: The state-of-the-art and future directions. *Journal of Grid Computing*, 6(3):255–276, 2008.

[12] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009.

[13] B. N. Chun and D. E. Culler. User-centric performance analysis of market-based cluster batch schedulers. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 30, Washington, DC, USA, 2002. IEEE Computer Society.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 137–150, 2004.

[15] D. E. Irwin, L. E. Grit, and J. S. Chase. Balancing risk and reward in a market-based task service. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 160–169, Washington, DC, USA, 2004. IEEE Computer Society.

[16] F. I. Popovici and J. Wilkes. Profitable services in an uncertain world. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 36, Washington, DC, USA, 2005. IEEE Computer Society.

[17] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job Scheduling for Multi-User MapReduce Clusters. Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, Apr 2009.