Optimal Map Reduce Job Capacity Allocation in Cloud Systems

Marzieh Malekimajd
Sharif University of Technology, Iran
malekimajd@ce.sharif.edu

Danilo Ardagna
Politecnico di Milano, Italy
danilo.ardagna@polimi.it

Michele Ciavotta
Politecnico di Milano, Italy
michele.ciavotta@polimi.it

Alessandro Maria Rizzi
Politecnico di Milano, Italy
alessandromaria.rizzi@polimi.it

Mauro Passacantando
Università di Pisa, Italy
mauro.passacantando@unipi.it

April 16, 2015

Abstract

We are entering a Big Data world. Many sectors of our economy are now guided by data-driven decision processes. Big Data and business intelligence applications are facilitated by the MapReduce programming model while, at infrastructural layer, cloud computing provides flexible and cost effective solutions for allocating on demand large clusters. Capacity allocation in such systems is a key challenge to provide performance for MapReduce jobs and minimize cloud resource costs. The contribution of this paper is twofold: (i) we provide new upper and lower bounds for MapReduce job execution time in shared Hadoop clusters, (ii) we formulate a linear programming model able to minimize cloud resources costs and job rejection penalties for the execution of jobs of multiple classes with (soft) deadline guarantees. Simulation results show how the execution time of MapReduce jobs falls within 14% of our upper bound on average. Moreover, numerical analyses demonstrate that our method is able to determine the global optimal solution of the linear problem for systems including up to 1,000 user classes in less than 0.5 seconds.

1 Introduction

Nowadays, many sectors of our economy are guided by data-driven decision processes [14]. In complex systems that do not lend themselves to intuitive models (e.g., natural sciences, social and engineered systems [11]), data-driven

modeling and hypothesis generation have a key role to understanding system behavior and interactions.

The adoption of data intensive applications is well recognized as able to enhance efficiency of enterprises and the quality of our lives. A recent McKinsey analysis [19] has shown, for instance, that Big Data could produce $300 billion potential annual value to US health care. The analysis has also shown how Europe public sector could potentially reduce expenditure of administrative activities by 15–20%, with an increase of value ranging between $223 and $446 billion [11, 19].

From the technological perspective, the MapReduce programming model is recognized to be the most prominent solution for Big Data applications [16]. Its open source implementation, Hadoop, is able to manage large datasets over either commodity clusters and high performance distributed topologies [29]. MapReduce has attracted interest of both industry and academia, since analyzing large amounts of unstructured data is a high priority task for many companies and overtakes the scalability level that can be achieved by traditional data warehouse and business intelligence technologies [16].

Likewise, cloud computing is becoming a mainstream solution to provide very large clusters on a pay-per-use basis. Cloud storage provides an effective and cheap solution for storing Big Data as modern NoSQL databases demonstrated good extensibility and scalability in storing and accessing data [15]. Moreover, the pay-per-use approach and the almost infinite capacity of cloud infrastructures can be used efficiently in supporting data intensive computation. Many cloud providers already include in their offering Map Reduce based platforms such as Google MapReduce framework, Microsoft HDinsight, and Amazon Elastic MapReduce [2, 4, 5]. IDC estimates that by 2020, nearly 40% of Big Data analyses will be supported by public cloud [6], while Hadoop is expected to touch half of the world data by 2015 [15].

A MapReduce job consists of two main phases, Map and Reduce; each phase performs a user-defined function on input data. MapReduce jobs were meant to run on dedicated clusters to support batch analyses. Nevertheless, MapReduce applications have evolved and it is not uncommon that large queries, submitted by different user classes, need to be performed on shared clusters, possibly with some guarantees on their execution time. In this context the main drawback [17, 26] is that the execution time of a MapReduce job is generally unknown in advance. In such systems, capacity allocation becomes one of the most important aspects. Determining the optimal number of nodes in a cluster, shared among multiple users performing heterogeneous tasks, is an important and challenging problem [26]. Moreover, capacity allocation policies need to decide jobs execution and rejection rates in a way that users' workloads meet their deadlines and the overall cost is minimized.

Capacity and Fair schedulers have been introduced in the new versions of Hadoop to address capacity allocation challenges and effective resource management [1, 3]. The main goal of Hadoop 2.x [25] is maximizing cluster utilization, while avoiding short (i.e., interactive) job starvation.

Our focus in this paper is on dynamic capacity allocation. First, we determine

new upper and lower bounds for MapReduce job execution times in shared Hadoop clusters adopting capacity and fair schedulers. Next, we formulate the capacity allocation problem as an optimization problem, with the aim of minimizing the cost of cloud resources and penalties for jobs rejections. We then reduce our minimization problem to a Linear Programming (LP) problem, which can be solved very efficiently by state of the art solvers.

We validate the accuracy of our bounds through the YARN Scheduler Load Simulator (SLS) [7]. The scalability of our optimization approach is demonstrated by considering a very large set of experiments. The largest instance we consider, including 1,000 user classes, can be solved to optimality in less than 0.5 seconds. Moreover, simulation results show that average job execution time is around 14% lower than our upper bound.

To the best of our knowledge, the only work providing upper and lower bounds for MapReduce jobs execution time is [26], where only dedicated clusters and FIFO scheduling are considered (that are not able to fulfill job concurrency and resource sharing requirements for current MapReduce applications).

This paper is organized as follows. MapReduce job execution time lower and upper bounds are presented in Section 2. In Section 3 the Capacity Allocation (CA) problem is introduced and its linear formulation is presented in Section 4. The accuracy of the bounds and the scalability of the solution are evaluated in Section 5. Section 6 describes the related work. Conclusions are finally drawn in Section 7.

2 Estimating job execution times in shared clusters

In large clusters, multiple classes of MapReduce jobs can be executed concurrently[1]. In such systems we need to estimate job execution times for determining the configuration of minimum cost, while providing service level agreement (SLA) guarantees. Previous works, e.g., [26], provided theoretical bounds to design performance models for Hadoop 1.0, considering in particular the FIFO scheduler. Those bounds can be used to predict job completion times only for dedicated clusters.

Nowadays, large shared clusters are ruled by newer schedulers, i.e., Capacity and Fair [1, 3]. In the following, we derive new bounds for such systems. In particular, Section 2.1 introduces preliminaries and provides a tighter bound with respect to [26] for a single-phase (either Map or Reduce) job. Section 2.2 extends the analysis to the case of two single-phase jobs, while Section 2.3 provides bounds for the case of multiple (single-phase) jobs involved. Ultimately, we complete our analysis using the bounds in Section 2.4 to derive execution time bounds for multiple classes of complete MapReduce jobs. Such results are used in the remaining sections to define the constraints of the CA problem that

[1] A job class is a set of jobs characterized by the same profile in terms of map, reduce and shuffle duration.

guarantee job deadlines are met. For space limitation, some proofs are omitted and reported in [18].

2.1 Single job bounds

Let us consider the execution of a single-phase MapReduce job J and let us denote with k, n, μ, and λ the number of available slots, the number of tasks in a Map or Reduce phase of J, the mean and maximum task duration, respectively. In the following, we suppose that the assignment of tasks to slots is done using an on-line greedy algorithm that assigns each task to the slot with the earliest finishing time.

Proposition 2.1. *The execution time of a Map or Reduce phase of J under a greedy task assignment is at most*

$$U = \frac{n\,\mu - \lambda}{k} + \lambda.$$

Proof. By contradiction, we assume the execution time is $U + \epsilon$ with $\epsilon > 0$. Note that $n\,\mu$ is the phase total workload, that is the duration of considered phase in the case of only one slot available. Let the last processed task has duration t. All slots are busy before the starting of the last task (otherwise it would have started earlier). The time that has elapsed before starting the last task is $(U + \epsilon - t)$. Since all slots are busy for $(U + \epsilon - t)$ time, the total workload until that point is $(U + \epsilon - t)\ k$. At the end of the execution, the whole phase workload must be unchanged, hence

$$\begin{aligned} (U + \epsilon - t)\ k + t = n\,\mu \qquad &\Leftrightarrow \left(\frac{n\mu - \lambda}{k} + \lambda + \epsilon - t\right) k + t = n\mu \Leftrightarrow \\ (k-1)\,\lambda + \epsilon\,k + t(1-k) = 0 \qquad &\Leftrightarrow \epsilon\,k = (t-\lambda)(k-1). \end{aligned}$$

Since $t \leq \lambda$, we get $\epsilon\,k \leq 0$, that is a contradiction because we assumed $\epsilon > 0$ and $k \geq 1$.

The worst case scenario is illustrated in Figure 1, where job J starts with k slots such that for $\frac{n\mu-\lambda}{k}$ time units all slots are busy. After that time only one task with duration λ is left to be executed. One slot performs the last task while all other slots are free. Finally, after $\frac{n\mu-\lambda}{k} + \lambda$ time units, all tasks are executed and the phase is completed. □

Note that a similar upper bound has been proposed in [26]. Our contribution improves the previous result by $\lambda - \mu$.

2.2 Two job bounds

In order to provide fast response times to small jobs and maximize the throughput and utilization of Hadoop clusters, Fair and Capacity schedulers have been devised. Fair scheduler organizes jobs in pools such that every job gets, on

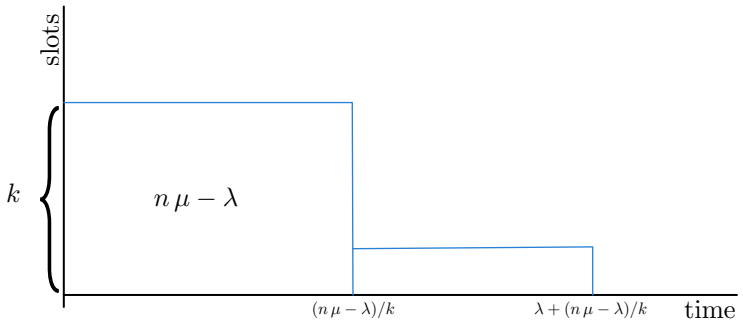

Figure 1: Worst case of one job execution time

average, an equal amount of resources over time. A single running job uses the entire cluster however, if other jobs are submitted, the slots that are progressively released are assigned to the new jobs. In addition, the Fair scheduler can guarantee minimum shares, enables preemption and limits the number of concurrent running jobs/tasks. Capacity schedulers have similar functionalities. The feature set of the Capacity scheduler includes minimum shares guarantee, security, elasticity, multi-tenancy, preemption and job priorities.

A scheduler is defined to be *work-conserving* if it never lets a processor idle while there are runnable tasks in the system. Both Fair and Capacity schedulers can be configured in *work-conserving* or *non-work-conserving* (which vice versa, let available resources idle) mode.

Let us consider the execution of two jobs J_i and J_j. If the system is configured in *non-work-conserving* mode, available slots are divided statistically and J_i idle slots are not allocated to J_j. Note that the upper bound defined in Proposition 2.1 and the lower bound provided in [26] are still valid, since resources are partitioned. Vice versa, if the system is configured according to *work-conserving* mode, when J_i finishes, its slots are allocated to J_j if it still has tasks waiting to start. In this situation, the bounds proposed in Propositions 2.2 and 2.3 hold. We assume both jobs start at the same time and J_i has α_i percent of all the available k slots whereas α_j percent of slots are reserved to J_j, i.e., $\alpha_i, \alpha_j \in (0,1)$ and $\alpha_i + \alpha_j = 1$.

Proposition 2.2. *The execution times of a greedy task assignment of two jobs (J_i,J_j) in work-conserving mode are at least* $\min\left\{\frac{n_i\,\mu_i}{\alpha_i\,k}, \frac{n_j\,\mu_j}{\alpha_j\,k}\right\}$ *and* $\frac{n_i\,\mu_i + n_j\,\mu_j}{k}$, *respectively.*

Proof. The analysis of the execution of the first finished job is equivalent to the case with a single job in the system (the best lower and upper bound known in the literature are given by [26] and Proposition 2.1). As regards the second job, the number of slots changes at some point of its execution, in other words when the first job finishes, the second job gets all the slots of the system.

Let us suppose that J_i terminates first, hence J_j receives all slots after at least $\frac{n_i\,\mu_i}{\alpha_i\,k}$ time units (i.e., after J_i lower bound [26]). Let us denote with t_f the

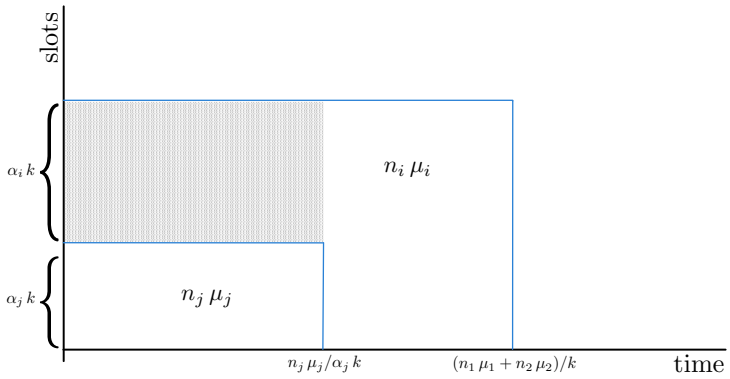

Figure 2: Lower bound of two jobs in work-conserving mode.

lower bound for J_j execution time. First J_j has $\alpha_j\, k$ slots until time instant $\frac{n_i\, \mu_i}{\alpha_i\, k}$ (see the dotted area in Figure 2), then J_j receives all k slots for a period of time equal to $\left(t_f - \frac{n_i\, \mu_i}{\alpha_i\, k}\right)$. The maximum workload that can be executed according to the number of slots is greater than or equal to the workload of job J_j:

$$\frac{n_i\, \mu_i}{\alpha_i\, k}\, \alpha_j\, k + \left(t_f - \frac{n_i \mu_i}{\alpha_i\, k}\right) k \geq n_j\, \mu_j.$$

Thus, by replacing α_j with $1 - \alpha_i$ we get

$$\frac{n_i\, \mu_i}{\alpha_i\, k}\, (1 - \alpha_i)\, k + \left(t_f - \frac{n_i\, \mu_i}{\alpha_i\, k}\right) k \geq n_j\, \mu_j,$$

that is equivalent to $t_f \geq (n_i\, \mu_i + n_j\, \mu_j)/k$. □

Proposition 2.3. *In a system with two jobs J_i and J_j in work-conserving mode, the upper bound of the execution time of job J_i is*

$$T_i = \begin{cases} \dfrac{n_i \mu_i - \lambda_i}{k \alpha_i} + \lambda_i, & \text{if } \frac{n_j\, \mu_j}{k\, \alpha_j} \geq \frac{n_i\, \mu_i - \lambda_i}{k\, \alpha_i}, \\ \dfrac{n_j\, \mu_j + n_i\, \mu_i - \lambda_i}{k} + \lambda_i, & \text{otherwise.} \end{cases}$$

Proof. Here we want to know the upper bound for a job when *conserving-mode* policy allows using idle slots. Hence, the upper bound is achieved when the minimum idle slots become available and it happens when the other job makes its slots busy. If $\frac{n_j\, \mu_j}{k\, \alpha_j} \geq \frac{n_i \mu_i - \lambda_i}{k \alpha_i}$ holds (see Figure 3), then the slots of other job can be busy such that upper bound of this job does not change. If the inequality does not hold, then slots of the other job become available before this job finishes (see Figure 4). Likewise the previous proof, in the worst case the last task (with maximum duration) can only start after a period of time in which all slots have been busy that is: $(n_j \mu_j + n_i \mu_i - \lambda_i)/k$. □

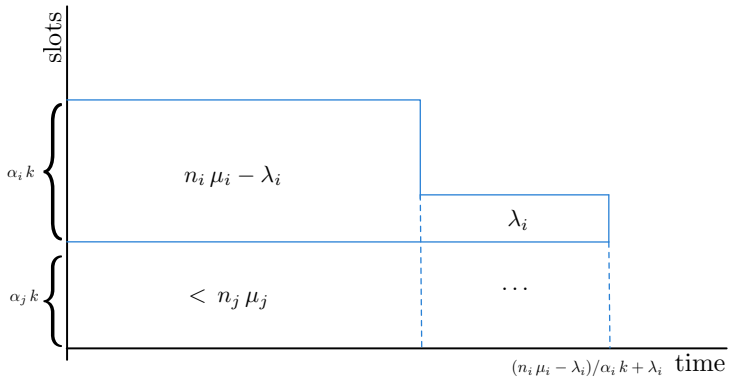

Figure 3: Upper bound of two jobs in work-conserving mode for the job that ends the earliest

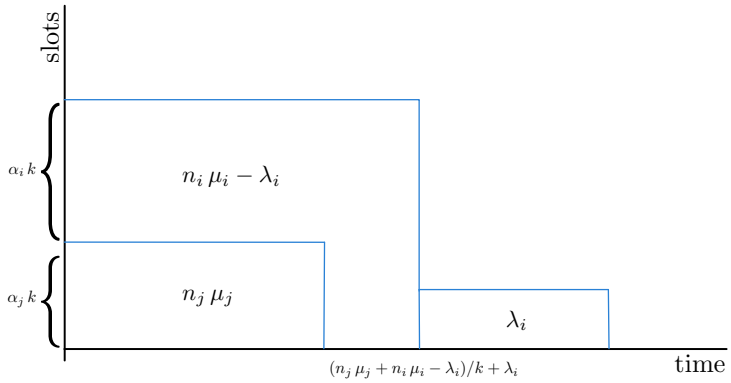

Figure 4: Upper bound of two jobs in work-conserving mode for the job that end the latest

2.3 Multiple class bounds

In a shared system, let k be the number of slots and $\mathcal{U}$ be the set of job classes. In each class $i \in \mathcal{U}$, h_i concurrent jobs are executed by using α_i percent of system slots. Each job J_i in class i has n_i tasks with mean task duration μ_i and maximum task duration λ_i.

Proposition 2.4. *The lower bound for the execution time of job J_i in presence of multiple classes of jobs is*

$$\frac{n_i\, \mu_i\, h_i}{k\, \alpha_i}.$$

Proof. Each class has $k\alpha_i$ slots and h_i concurrent jobs, so each job has overall $k\alpha_i/h_i$ slots and, using the bound provided in [26], we get as lower bound $\frac{n_i \mu_i}{\frac{k\alpha_i}{h_i}} = \frac{n_i \mu_i h_i}{k\alpha_i}$. □

Proposition 2.5. *The upper bound for the execution time of job J_i in presence of multiple classes of jobs is*

$$\frac{(n_i\mu_i - 2\lambda_i)h_i}{k\alpha_i} + 2\lambda_i.$$

Proof. Figure 5 shows a system where slots are shared among several classes of jobs. The max number of slots dedicated to a single job of class i is $k\alpha_i/h_i$.

Let us illustrate the worst case scenario for job J_i. We assume that job J_i^- is executed before J_i and that each slot freed up from J_i^- is dedicated to J_i. We also assume $k\alpha_i/h_i - 1$ slots in the last wave of job J_i^- start performing a task with maximum duration, and the first slot freed up from job J_i^- is dedicated to J_i. In the worst case, this slot also performs a task with maximum duration. After duration λ_i, remaining slots in J_i^- are freed up and are dedicated to job J_i. $k\alpha_i/h_i$ slots perform tasks of J_i for $\frac{(n_i\mu_i-2\lambda_i)h_i}{k\alpha_i}$ time and after that there is just one task with max duration λ_i. So time $\frac{(n_i\mu_i-2\lambda_i)h_i}{k\alpha_i} + 2\lambda_i$ is spent for performing job J_i

To prove there is no larger upper bound we use contradiction. Let us assume that job J_i in Figure 6 is executed in time $\epsilon + \frac{n_i\mu_i-2\lambda_i}{\frac{k\alpha_i}{h_i}} + 2\lambda_i, \epsilon > 0$. Let after time $t_1 \leq \lambda_i$ of the considered job starts, all possible slots $k\alpha_i/h_i$ (fair share) are allocated to J_i and the duration of the last task is $t_2 \leq \lambda_i$. The duration $\epsilon + \frac{n_i\mu_i-2\lambda_i}{\frac{k\alpha_i}{h_i}} + 2\lambda_i - (t_1 + t_2)$ is the minimum amount of time that the assumed job has $k\alpha_i/h_i$ slots. We calculate a bound by computing the minimum amount of workloads that can be done W^1 and the amount of workload that has to be done $W^2 = n_i\mu_i$. The minimum amount is

$$W^1 = (\epsilon + \frac{n_i\mu_i - 2\lambda_i}{\frac{k\alpha_i}{h_i}} + 2\lambda_i - t_1 - t_2)\frac{k\alpha_i}{h_i} + t_1 + t_2$$

as shown by the dotted area in Figure 6. Note that, the first term is the workload performed when $k\,\alpha_i/h_i$ slots are available, while t_1 and t_2 are the workloads performed when there is at least one single slot. The following relation between W^1 and W^2 holds:

$$\left(\epsilon + \frac{n_i\,\mu_i - 2\lambda_i}{\frac{k\,\alpha_i}{h_i}} + 2\lambda_i - t_1 - t_2\right)\frac{k\,\alpha_i}{h_i} + t_1 + t_2 \leq n_i\,\mu_i.$$

Since $1 - \frac{k\,\alpha_i}{h_i} \leq 0$ and $t_1 + t_2 - 2\lambda_i \leq 0$, we get

$$\epsilon\frac{k\,\alpha_i}{h_i} + n_i\,\mu_i - 2\lambda_i + (2\lambda_i - t_1 - t_2)\frac{k\,\alpha_i}{h_i} + t_1 + t_2 \leq n_i\,\mu_i,$$

i.e., $\epsilon \leq t_1 + t_2 - 2\,\lambda_i \leq 0$, which is impossible since $\epsilon > 0$. □

2.4 Bounds for MapReduce Jobs Execution

In this section, we extend the results presented in [26] for a MapReduce system with S_M Map slots and S_R Reduce slots using Fair/Capacity scheduler. Similar jobs are grouped together in a job class $i \in \mathcal{U}$ and α_M^i and α_R^i are the percentage

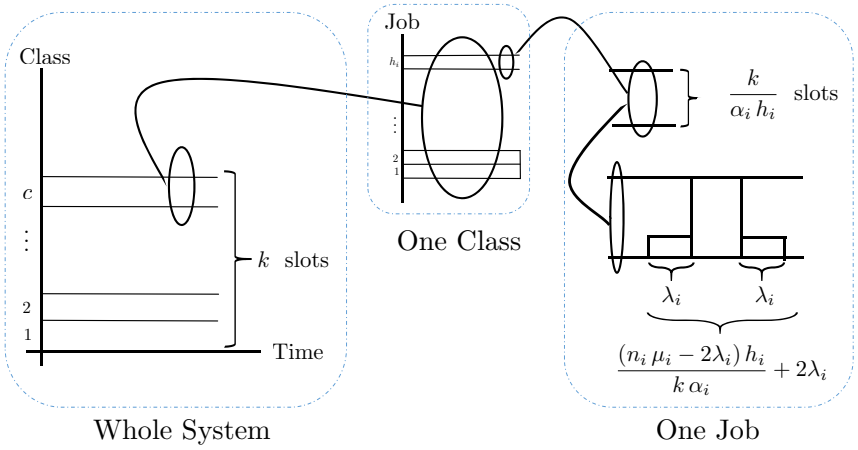

Figure 5: Slots sharing in a system with several classes of jobs

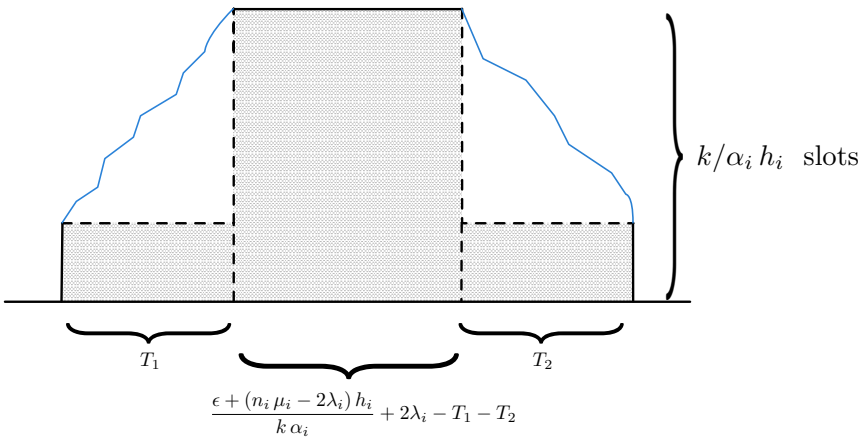

Figure 6: Execution of a single job in considering the proof by contradiction

of all Map and Reduce slots dedicated to class i, while there are h_i jobs running concurrently. Let us denote with M^i_{avg}, M^i_{max}, R^i_{avg}, R^i_{max}, $Sh^{1,i}_{avg}$, $Sh^{1,i}_{max}$, Sh^i_{avg} and Sh^i_{max} the average and maximum durations of Map, Reduce, first Shuffle and typical Shuffle tasks, respectively. These values define an empirical performance profile for each job class i, while N^i_M and N^i_R are the number of Map and Reduce tasks of job J_i profile. By using the bounds defined in the previous sections, a lower and an upper bound on the duration of the entire Map phase can be estimated as follows:

$$T^{low}_{M^i} = \frac{N^i_M\, M^i_{avg}\, h_i}{S_M\, \alpha^i_M},$$
$$T^{up}_{M^i} = \frac{(N^i_M\, M^i_{avg} - 2M^i_{max})\, h_i}{S_M\, \alpha^i_M} + 2\, M^i_{max}.$$

Similar results can be obtained for the Reduce stage, that consists of the Reduce and part of the Shuffle phase. In fact, according also to the results discussed in [26], we distinguish the non-overlapping portion of the first shuffle wave from the duration of the remaining tasks in the typical shuffle. The time of the typical shuffle phase can be estimated as:

$$T^{low}_{Sh^i} = \left(\frac{N^i_R\, h_i}{S_R\, \alpha^i_R} - 1\right) Sh^i_{avg},$$
$$T^{up}_{Sh^i} = \frac{(N^i_R\, Sh^i_{avg} - 2\, Sh^i_{max})\, h_i}{S_M\, \alpha^i_R} + 2\, Sh^i_{max}.$$

Finally, by putting all parts together, we get:

$$T^{low}_i = A^{low}_i \frac{h_i}{S_M \alpha^i_M} + B^{low}_i \frac{h_i}{S_R \alpha^i_R} + C^{low}_i, \tag{1}$$

where $A^{low}_i = N^i_M M^i_{avg}$, $B^{low}_i = N^i_R(Sh^i_{avg} + R^i_{avg})$ and $C^{low}_i = Sh^{1(J_i)}_{avg} - Sh^i_{avg}$.

In the same way, the execution time of job J_i is at most:

$$T^{up}_i = A^{up}_i \frac{h_i}{S_M \alpha^i_M} + B^{up}_i \frac{h_i}{S_R \alpha^i_R} + C^{up}_i, \tag{2}$$

where:

$$A^{up}_i = N^i_M M^i_{avg} - 2M^i_{max},$$
$$B^{up}_i = N^i_R Sh^i_{avg} - 2Sh^i_{max} + N^i_R R^i_{avg} - 2R^i_{max},$$
$$C^{up}_i = 2Sh^i_{max} + Sh^{1(i)}_{max} + 2M^i_{max} + 2R^i_{max}.$$

According to the guarantees to be provided to the end users, we can use T^{up}_i upper bound (being conservative) or the approximated formula

$$T^{avg}_i = (T^{low}_i + T^{up}_i)/2 \tag{3}$$

to bound the execution time of class i jobs in the Capacity Allocation problem described in the next section.

3 Capacity Allocation Problem

In this section, we consider the joint Capacity Allocation and Admission Control problem for a cloud based shared Hadoop 2.x system. We assume that the system runs the fair or capacity scheduler, serving a set of user classes, requesting the concurrent execution of jobs with similar execution profile. Each class i is executed with $s^i_M = \alpha^i_M S_M$ Map slots and $s^i_R = \alpha^i_R S_R$ Reduce slots with a concurrency degree of h_i (i.e., h_i jobs with the same profile are executed concurrently). We also assume that the system implements an admission control mechanism bounding the number of concurrent jobs h_i executed by the system, i.e., some jobs can be rejected. H^{up}_i denotes a prediction for the number of jobs of class i to be executed and we have $h_i \leq H^{up}_i$. Furthermore, in order to avoid job starvation, we also impose h_i to be greater than a given lower bound H^{low}_i. Finally, a (soft) deadline D_i is associated with each class i.

Note that, given s^i_M, s^i_R and h_i, the execution time of a class i job can be approximated by:

$$T_i = \frac{A_i\, h_i}{s^i_M} + \frac{B_i\, h_i}{s^i_R} + C_i, \tag{4}$$

where A_i, B_i and C_i are positive constants computed as discussed in the previous section.

We can use Equations (2)–(3) to derive (4), considering conservative upper bounds. In this latter case D_i can be considered as hard deadline. In alternative, as in [26], (4) can be obtained from (1), (2), and (3). In that case, (4) is not a bound but an approximated formula and D_i becomes soft deadline. In this work, we follow this latter more flexible approach. We assume that our MapReduce implementation is hosted in a Cloud environment that provides *on-demand* and *reserved* (see, e.g., Amazon EC2 pricing model [2]) homogeneous virtual machines (VMs). Moreover, we denote with c^i_M and c^i_R the number of Map and Reduce slots hosted in each VM, i.e., each instance supports c^i_M Map and c^i_R Reduce concurrent tasks for each job J_i in class i. As a consequence, let x_m and x_r be the number of Map and Reduce slots required by a certain job J_i, the number of VMs to be provisioned has to be equal to $x_m/c^i_M + x_r/c^i_R$.

Let us denote with δ and with $\rho < \delta$ the cost of on-demand and reserved VMs, respectively and with $\bar{r}$ the number of reserved VMs available (i.e., the number of VMs subscribed with a long term contract). Let d and r be the number of on-demand and reserved VMs, respectively, used to serve end users' requests. The aim of the Capacity Allocation (CA) problem we consider here is to minimize the overall execution cost meeting, at the same time, all deadlines. The execution cost includes both the VM allocation cost and the penalty cost for job rejection. Given p_i, the penalty cost for rejection of a class i job, the overall execution cost can be calculated as follows:

$$\delta\, d + \rho\, r + \sum_{i \in \mathcal{U}} p_i\, (H^{up}_i - h_i)\,, \tag{5}$$

where decision variables are d, r, h_i, s^i_M and s^i_R, for any $i \in \mathcal{U}$, i.e., we have to decide the number of on-demand and reserved VMs, concurrency degree, and

System Parameters	
c_M^i	Number of Map slots hosted in a VM of class i
c_R^i	Number of Reduce slots hosted in a VM of class i
$\mathcal{U}$	Set of job classes
p_i	Penalty for rejecting jobs from class i
D_i	Makespan deadline of jobs from class i
A_i	CPU requirement for the Map phase which can be derived by input data and job class i
B_i	CPU requirement for the Reduce phase which can be derived by input data and job class i
C_i	Time constant factor depends on Map, Copy, Shuffle and Reduce phases that derived by input data and job class i
$\bar{r}$	Number of available reserved VMs
δ	Cost of on-demand VMs
ρ	Cost of reserved VMs
H_i^{up}	Upper bound on the number of class i jobs to be executed concurrently
H_i^{low}	Lower bound on the number of class i jobs to be executed concurrently
Decision Variables	
s_M^i	Number of slots to be allocated to class i for executing Map task
s_R^i	Number of slots to be allocated to class i for executing Reduce task
h_i	Number of jobs of class i to be executed concurrently
r	Number of reserved VMs to be allocated for job execution
d	Number of on-demand VMs to be allocated for for job execution

Table 1: Optimization model: parameters and decision variable.

the number of Map and Reduce slots for each job class i. The notation adopted in this paper is summarized in Table 1.

4 Optimization Problem

In this section, we formulate the CA optimization problem and propose a suitable and fast solution technique for the execution of MapReduce jobs in Cloud environments. The objective is to minimize the execution cost, while meeting job (soft) deadlines. The total cost includes VM provisioning costs and a penalty due to job rejection. In equation (5) the term $\sum_{i=1}^{c} p_i H_i^{up}$ is a constant independent from decision variables and can be dropped. The optimization problem can then be defined as follows:

$$\text{(P0)} \qquad \min \ \delta\, d + \rho\, r - \sum_{i \in \mathcal{U}} p_i\, h_i$$

subject to:

$$\frac{A_i\,h_i}{s_M^i} + \frac{B_i\,h_i}{s_R^i} + E_i \leq 0, \quad \forall i \in \mathcal{U}, \tag{6}$$

$$r \leq \bar{r}, \tag{7}$$

$$\sum_{i \in \mathcal{U}} \left(\frac{s_M^i}{c_M^i} + \frac{s_R^i}{c_R^i} \right) \leq r + d, \tag{8}$$

$$H_i^{low} \leq h_i \leq H_i^{up}, \quad \forall i \in \mathcal{U}, \tag{9}$$

$$r \geq 0, \tag{10}$$

$$d \geq 0, \tag{11}$$

$$s_M^i \geq 0, \quad \forall i \in \mathcal{U}, \tag{12}$$

$$s_R^i \geq 0, \quad \forall i \in \mathcal{U}, \tag{13}$$

where constraints (6) are derived from equation (4) by imposing the execution of each job to end before its deadline (i.e., $E_i = C_i - D_i < 0$). Constraint (7) ensures that no more than the available reserved VMs can be allocated. Constraint (8) guarantees that enough VMs are allocated to execute submitted jobs within their deadlines. Constraints (9) bound the job concurrency level for each user.

We remark that, in the above problem formulation, variables r, d, s_M^i, s_R^i, h_i are not integer as in reality they should be. In fact, requiring variables to be integer makes the problem much more difficult to solve. However, this approximation is widely used in the literature (see, e.g., [9, 30]) since relaxed variables can be rounded to the closest integer at the expense of a generally very small increment of the overall cost (this is intuitive for large-scale MapReduce systems that require tens or hundreds of relatively cheap VMs), justifying the use of a relaxed model. Therefore, we decided to deal with continuous variables, considering a relaxation of the real problem. However, this restriction will be removed in the numerical analyses reported in Section 5.

Problem (P0) has a linear objective function but constraints (6) are non-linear and non-convex (the proof is reported in [18]). To overcome the non-convexity of the constraints, we introduce new decision variables $\Psi_i = 1/h_i$, for any $i \in \mathcal{U}$, to replace h_i. Then, problem (P0) is equivalent to problem (P1) defined as follows:

(P1) $$\min\ \delta\, d + \rho\, r - \sum_{i \in \mathcal{U}} \frac{p_i}{\Psi_i}$$

subject to:

$$\frac{A_i}{s_M^i \Psi_i} + \frac{B_i}{s_R^i \Psi_i} + E_i \leq 0, \qquad \forall\ i \in \mathcal{U}, \tag{14}$$

$$r \leq \bar{r}, \tag{15}$$

$$\sum_{i \in \mathcal{U}} \left(\frac{s_M^i}{c_M^i} + \frac{s_R^i}{c_R^i} \right) \leq r + d, \tag{16}$$

$$\Psi_i^{low} \leq \Psi_i \leq \Psi_i^{up}, \qquad \forall\ i \in \mathcal{U}, \tag{17}$$

$$r \geq 0, \tag{18}$$

$$d \geq 0, \tag{19}$$

$$s_M^i \geq 0, \qquad \forall\ i \in \mathcal{U}, \tag{20}$$

$$s_R^i \geq 0, \qquad \forall\ i \in \mathcal{U}, \tag{21}$$

where $\Psi_i^{low} = 1/H_i^{up}$ and $\Psi_i^{up} = 1/H_i^{low}$. We remark that now constraints (14) are convex (the proof is reported in [18]). The convexity of all the constraints of problem (P1) allows to prove the following result.

Theorem 4.1. *In any optimal solution of problem (P1), constraints (14) hold as equalities and the number of slots to be allocated to job class i, s_M^i and s_R^i, can be evaluated as follows:*

$$s_M^i = -\frac{1}{E_i \Psi_i} \left(\sqrt{\frac{A_i\, B_i\, c_M^i}{c_R^i}} + A_i \right), \tag{22}$$

$$s_R^i = -\frac{1}{E_i \Psi_i} \left(\sqrt{\frac{A_i\, B_i\, c_R^i}{c_M^i}} + B_i \right). \tag{23}$$

The proof of Theorem 4.1 is reported in [18]. The results of Theorem 4.1 allow to transform (P1) into an equivalent linear programming problem, which can be solved very quickly by state of the art solvers.

Theorem 4.2. *(P1) is equivalent to the following problem:*

(P2) $$\min\ \delta\, d + \rho\, r - \sum_{i \in \mathcal{U}} p_i\, h_i$$

subject to:

$$r \leq \bar{r}, \tag{24}$$

$$\sum_{i \in \mathcal{U}} \gamma_i\, h_i \leq r + d, \tag{25}$$

$$H_i^{low} \leq h_i \leq H_i^{up}, \qquad \forall\ i \in \mathcal{U}, \tag{26}$$

$$r \geq 0, \tag{27}$$

$$d \geq 0, \tag{28}$$

where $\gamma_i = \gamma_i^1 + \gamma_i^2$ *with:*

$$\gamma_i^1 = -\frac{1}{E_i\, c_R^i}\left(\sqrt{\frac{A_i\, B_i\, c_R^i}{c_M^i}} + B_i\right), \tag{29}$$

$$\gamma_i^2 = -\frac{1}{E_i\, c_M^i}\left(\sqrt{\frac{A_i\, B_i\, c_M^i}{c_R^i}} + A_i\right), \tag{30}$$

and the decision variables are r, d *and* $h_i = 1/\Psi_i$, *for any* $i \in \mathcal{U}$.

The proof of Theorem 4.2 is reported in [18]. Since (P2) is a linear problem, commercial and open source solvers currently available are able to solve efficiently very large instances. A scalability analysis is reported in the following section.

The Karush-Kuhn-Tucker (KKT) conditions corresponding to problem (P2) guarantee that any optimal solution of (P2) has the following important properties.

Theorem 4.3. *If* (r^*, d^*, h^*) *is an optimal solution of problem (P2), then the following statements hold:*

a) $r^* > 0$, *i.e., reserved instances are always used.*

b) $\sum_{i\in\mathcal{U}} \gamma_i\, h_i^* = r^* + d^*$, *i.e.,* γ_i *can be considered a computing capacity conversion ratio that allows to translate class* i *concurrency level into VM capacity resource requirements.*

c) If $p_i/\gamma_i > \delta$, *then* $h_i^* = H_i^{up}$, *i.e., class* i *job are never rejected.*

d) If $p_i/\gamma_i < \rho$, *then* $h_i^* = H_i^{low}$, *i.e., class* i *concurrency level is set to the lower bound.*

e) If $\bar{r} > \sum_{i\in\mathcal{U}} \gamma_i\, H_i^{up}$, *then* $d^* = 0$, *i.e., for property b), if the total capacity requirement can be satisfied through reserved instances, on demand VMs are never used.*

f) If $\bar{r} < \sum_{i\in\mathcal{U}} \gamma_i\, H_i^{low}$, *then* $r^* = \bar{r}$ *and* $d^* > 0$, *i.e., for property b), if the minimum job requirements exceed reserved instance capacity, then on demand VMs are needed.*

Proof. The KKT conditions associated to (P2) are:

$$\rho - \nu + \mu_r - \lambda_r = 0, \tag{31}$$

$$\delta - \nu - \lambda_d = 0, \tag{32}$$

$$-p_i + \gamma_i\, \nu + \mu_i - \lambda_i = 0, \quad \forall\, i\ \in \mathcal{U}, \tag{33}$$

$$\nu\left(\sum_{i\in\mathcal{U}} \gamma_i\, h_i^* - r^* - d^*\right) = 0, \tag{34}$$

$$\lambda_r\, r^* = 0, \tag{35}$$

$$\mu_r\,(r^* - \bar{r}) = 0, \tag{36}$$

$$\lambda_d\, d^* = 0, \tag{37}$$

$$\lambda_i\,(h_i^* - H_i^{low}) = 0, \quad \forall\, i\ \in \mathcal{U}, \tag{38}$$

$$\mu_i\,(h_i^* - H_i^{up}) = 0, \quad \forall\, i\ \in \mathcal{U}, \tag{39}$$

$$\nu, \lambda_r, \mu_r, \lambda_d \geq 0, \tag{40}$$

$$\lambda_i, \mu_i \geq 0, \quad \forall\, i\ \in \mathcal{U}. \tag{41}$$

a) Assume, by contradiction, that $r^* = 0$. Then

$$d^* \geq \sum_{i \in \mathcal{U}} \gamma_i \, h_i^* \geq \sum_{i \in \mathcal{U}} \gamma_i \, H_i^{low} > 0,$$

thus $\lambda_d = 0$ and $\nu = \delta$. On the other hand, (36) implies that $\mu_r = 0$ and $\lambda_r = \rho - \nu = \rho - \delta < 0$ which is impossible.

b) Since $r^* > 0$, we have $\lambda_r = 0$, hence (31) implies $\nu = \rho + \mu_r \geq \rho > 0$, thus constraint (25) is active at (r^*, d^*, h^*).

c) It follows from (32) that $\nu = \delta - \lambda_d \leq \delta$, hence we have

$$\mu_i = \lambda_i + p_i - \gamma_i \, \nu \geq p_i - \gamma_i \, \nu \geq p_i - \gamma_i \, \delta > 0.$$

Therefore $h_i^* = H_i^{up}$.

d) Since $\nu \geq \rho$, we get

$$\lambda_i = \mu_i + \gamma_i \, \nu - p_i \geq \gamma_i \, \nu - p_i \geq \gamma_i \, \rho - p_i > 0,$$

hence $h_i^* = H_i^{low}$.

e) We have

$$r^* = \sum_{i \in \mathcal{U}} \gamma_i \, h_i^* - d^* \leq \sum_{i \in \mathcal{U}} \gamma_i \, H_i^{up} < \bar{r},$$

thus $\mu_r = 0$ and $\nu = \rho$. Therefore, $\lambda_d = \delta - \rho > 0$ implies $d^* = 0$.

f) We have

$$d^* = \sum_{i \in \mathcal{U}} \gamma_i \, h_i^* - r^* \geq \sum_{i \in \mathcal{U}} \gamma_i \, H_i^{low} - \bar{r} > 0,$$

hence $\lambda_d = 0$ and $\nu = \delta$. Therefore, $\mu_r = \delta - \rho > 0$ implies $r^* = \bar{r}$.

□

Property *a)* is obvious, since reserved instances are the cheapest ones. Property *b)* and Theorem 4.2 lead to an important theoretical result. Indeed, γ_i parameters can be interpreted as a computing capacity conversion ratio that allows to estimate VM capacity requirements in terms of class i concurrency level. Accordingly, also properties *c)* and *d)* become intuitive. The product $\gamma_i \, \delta$ is the unit cost for class i job execution with on-demand instances. If $\gamma_i \, \delta$ is lower than the penalty cost, then class i jobs will always be executed. Vice versa, if $\gamma_i \, \rho$, i.e., the class i per unit reserved cost, is larger than the penalty, class i jobs will always be rejected. Finally, properties *e)* and *f)* relate the overall minimum $\sum_{i \in \mathcal{U}} \gamma_i \, H_i^{low}$ and maximum $\sum_{i \in \mathcal{U}} \gamma_i \, H_i^{low}$ capacity requirements to reserved instance capacity and allow to establish a priory if on demand VMs will or will not be used.

5 Experimental Results

In this section we: (i) validate job execution time bounds, (ii) evaluate the scalability of the CA problem solution, and (iii) investigate how different (P2) problem settings impact on the cloud cluster cost.

Our analyses are based on a very large set of randomly generated instances. Bound accuracy is evaluated through the YARN Scheduler Load Simulator (SLS) [7]. In the following section, the design of experiments is presented. Bound accuracy and scalability analyses are reported in Sections 5.2 and 5.3. Finally, the analysis of how (P2) problem parameters impact on cost is reported in Section 5.4.

5.1 Design of experiments

Analyses in this section intend to be representative of real Hadoop systems. Instances have been randomly generated by picking parameters according to values observed in real systems and logs of MapReduce applications. Afterwards, we use uniform distributions within the ranges reported in Table 2.

In our model, the cloud cluster consists of on-demand and reserved VMs. We considered Amazon EC2 prices for VM hourly costs [2]. On demand and reserved instance prices varied in the range ($0.05,$0.40), to consider the adoption of different VM configurations.

Regarding MapReduce applications parameters, we used the values reported in [27], which consider real log traces obtained from four MapReduce applications: Twitter, Sort, WikiTrends, and WordCount.

Moreover, as in [27] we assume that deadlines are uniformly distributed in the range (10, 20) minutes. We use the job profile from [27] to calculate a reasonable value for penalties. First, the minimum cost for running a single job (let it be cj_i) is evaluated by setting $H_i^{up} = H_i^{low}$ and solving problem $(P2)$, disabling the admission control mechanism. Then, we set the penalty value for job rejections $p_i = 10\, cj_i$ as in [8]. We varied H_i^{up} in the range (10, 30), and we set $H_i^{low} = 0.9\, H_i^{up}$.

Job Profile		**Cluster Scale**	
N_M^i	(70, 700)	H_i^{up} (¢)	(10, 30)
N_R^i	(32, 64)		
M_{max}^i (s)	(16, 120)	**Job Rejection Penalty**	
$Sh_{max}^{typ^i}$ (s)	(30, 150)	p_i (¢)	(250, 2500)
R_{max}^i (s)	(15, 75)		
$Sh_{max}^{1(i)}$ (s)	(10, 30)	**Cloud Instance Price**	
c_M^i, c_R^i	(1, 4)	ρ (¢)	(5, 20)
D_i (s)	(600, 1200)	δ (¢)	(5, 40)

Table 2: Cluster characteristics and Job Profiles

5.2 Accuracy of Execution Time Bounds

The aim of this section is to compare our time bounds (1) and (2) against the execution times obtained through YARN SLS [7], the official simulator provided within Hadoop 2.3 framework.

YARN SLS requires an Hadoop deployment and it interacts with it by means of mocked *NodeManagers* and *ApplicationMasters* with the purpose of simulating

Twitter			Sort		
No. of users	T_1^{up} gap	m_1 gap	No. of users	T_2^{up} gap	m_2 gap
4	7.78%	1.24%	10	6.04%	0.61%
6	6.35%	-0.09%	8	8.83%	3.26%
5	18.53%	7.68%	4	19.79%	10.42%
4	16.10%	6.43%	6	12.79%	4.79%
8	6.70%	-11.98%	7	2.85%	-7.48%
3	17.04%	7.12%	7	14.24%	5.64%
6	4.65%	-9.80%	10	6.35%	-10.80%
6	2.26%	-5.07%	6	5.07%	-1.58%
9	0.49%	-4.94%	7	2.43%	-2.44%
4	8.24%	1.56%	10	5.28%	-0.45%

Table 3: Two job classes analysis (Twitter and Sort)

both a set of cluster nodes and the relative workload. Those entities interact directly with Hadoop YARN, simulating a whole running environment with a one to one mapping between simulated and real times (i.e., the simulation of 1 second of the Hadoop cluster requires 1 second simulation).

SLS requires as input a cluster configuration file and an execution trace. This trace can be provided either in Apache Rumen[2] format or in the SLS proprietary format (the one we adopted), which is a simplified version containing only the data strictly needed for simulation. In particular, among other information, it provides for each job and each task the start and end times.

In our evaluation we consider the MapReduce job profiles extracted from log traces available from Twitter, Sort, WikiTrends, and WordCount reported in [27]. In order to use the SLS tool, we generated synthetic job traces representing these workloads. First of all, since SLS does not provide shuffle phase execution time, we have to use a simplified version of equations (1) and (2). Therefore, we partially removed the shuffle phase, by ignoring the first shuffle wave (to a certain extent overlapped with the last Map wave, though) and by including the remaining part (e.g., Sh^i_{avg}) in the Reduce phase. We also consider the total number of available slots as shared between the Map and Reduce tasks, being unable to assign them to a specific phase. In particular, we used a number of slots equals to the number of virtual cores allocated in the simulator. These slots have been used in both phases so we set S_M and S_R equal to the available cores. Then, we set the ratios $\frac{\alpha^i_R}{h_i} = \frac{\alpha^i_M}{h_i}$ equal to $1/\sum_{k\in\mathcal{U}} h_k$. This because the available resources are equally shared among the different users, so each class i will have a ratio of resources proportional to its users h_i: $\alpha^i_R = \alpha^i_M = h_i/\sum_{k\in\mathcal{U}} h_k$.

In order to validate our bounds, we must compute job durations, i.e., for each job, the difference between its submission and completion time.

Since SLS is a trace based simulator, we must generate a trace that interleaves

[2] A tool for extracting traces from Hadoop logs http://hadoop.apache.org/docs/r1.2.1/rumen.html

WordCount			WikiTrends		
No. of users	T_1^{up} gap	m_1 gap	No. of users	T_2^{up} gap	m_2 gap
2	23.12%	5.27%	4	37.46%	23.93%
4	8.35%	-4.20%	4	26.46%	16.78%
3	28.30%	7.08%	2	57.48%	39.47%
2	14.04%	-0.65%	3	23.28%	12.65%
4	19.15%	3.80%	3	48.68%	35.86%
5	17.32%	5.06%	4	34.95%	25.61%
3	21.97%	4.34%	3	35.58%	22.15%
3	37.22%	14.59%	2	62.11%	43.47%
5	15.89%	2.52%	3	37.19%	26.62%
2	17.50%	2.41%	5	26.01%	15.08%

Table 4: Two job classes analysis (WordCount and WikiTrends)

for each user the submission of jobs by their average duration. However, we do not know this duration (that is the goal of this simulation), but we can obtain it by relying on a fixed-point iteration method. We consider a closed model in which for each class i, h_i users can concurrently submit multiple jobs. Let approximate the average job duration T_i with an initial guess $A_{i,0}$ for each class $i \in \mathcal{U}$ and run the simulation of the generated trace. Then, we can refine our guess of T_i iteratively with the value $A_{i,n}$, computed as follows:

$$A_{i,n} = \beta \widetilde{T}_{i,n-1} + (1-\beta) A_{i,n-1}, \tag{42}$$

for each class $i \in \mathcal{U}$ (we experimentally set $\beta = 0.07$), where $\widetilde{T}_{i,n-1}$ is the average job duration obtained by SLS for class i at the previous run $n-1$.

We iterate this procedure until $A_{i,n}$ and $\widetilde{T}_{i,n}$ are close enough for each class $i \in \mathcal{U}$. At that point $A_{i,n} \approx \widetilde{T}_{i,n} \approx T_i$ for each job class i. We stop the fixed-point iteration method when the ratio $\max_{i \in \mathcal{U}} |A_{i,n} - \widetilde{T}_{i,n}|/\widetilde{T}_{i,n}$ is below a given threshold τ (set experimentally equal to 0.1). We then evaluate how far our bounds are from this value, by comparing $\widetilde{T}_{i,n}$ with the upper bound T_i^{up} and the average of the two bounds $m_i = (T_i^{low} + T_i^{up})/2$.

Each simulation trace has been built by considering different user classes (drawn from WorkCount, Sort, Twitter and WikiTrends traces) setting $A_{i,0} = T_i^{up}$ for any $i \in \mathcal{U}$. In order to avoid that jobs start simultaneously (unrealistic in real systems), we delay each job submission by a random exponentially-distributed time value (i.e., the user think time set equal to a tenth of the estimated job execution time). Ultimately, we scaled down by a factor of 10 the original execution times in order to achieve a simulation speedup.

We considered different test configurations with two and three job classes and with a random number of users in the range $[2, 10]$. Those scenarios represents light load conditions that correspond to the worst case for the evaluation of our

Twitter			Sort			WordCount		
N.	T_1^{up} gap	m_1 gap	N.	T_2^{up} gap	m_2 gap	N.	T_3^{up} gap	m_3 gap
5	16.99%	7.24%	3	17.82%	9.46%	2	16.20%	5.07%
4	10.25%	1.06%	3	15.01%	6.85%	3	10.88%	0.26%
5	6.21%	-1.26%	3	2.84%	-3.30%	4	5.26%	-3.30%
5	8.71%	-7.89%	4	10.24%	-4.06%	2	8.84%	-10.32%
5	3.92%	-3.39%	5	2.66%	-3.46%	2	3.82%	-4.62%
5	14.32%	5.43%	4	14.31%	6.44%	2	14.37%	4.34%
3	10.10%	2.21%	4	13.58%	6.38%	5	8.24%	-0.53%
4	21.64%	11.33%	2	17.84%	8.97%	4	18.58%	7.26%
2	11.51%	2.06%	3	8.37%	0.21%	5	9.74%	-0.74%
4	9.53%	1.68%	4	10.46%	3.46%	4	7.37%	-1.32%

Table 5: Three job classes analysis (Twitter, Sort and WordCount)

bounds. Indeed, under light load conditions the probability that any user class is temporarily idle can be significant and, the Fair and Capacity scheduler, would assign the idle user class slots to other classes to boost their performance. Vice versa, under heavy loads our upper bounds become tighter.

Tables 3-6 report the results we achieved. For each run the number of users and the gap between T_i and both T_i^{up} and m_i are reported (a negative m_i gap means that $T_i > m_i$). All the simulations have been performed considering a cluster with 128 cores and using the YARN fair scheduler.

Overall, for the two job classes, the gap between the upper bound and the jobs mean execution time is around 19% on average, while the gap with respect to m_i is only 10% on average. For three classes the average between the upper bound and the jobs mean execution time gap is 11%, while the gap with respect to m_i is 5%. Over all the set of experiments the average between the upper bound and the jobs mean execution time is 14%.

Simulations run on Microsoft Azure Linux small instances (i.e., single core, 1.75GB VMs). The fixed-point iteration procedure converges in 4.4 iterations on average. The simulation time of each fixed-point procedure iteration was around 31 minutes.

5.3 Scalability analysis

In this section, we evaluate the scalability of our optimization solution. We performed our experiment on a VirtualBox virtual machine based on Ubuntu 12.04 server running on an intel Xeon Nehalem dual socket quad-core system with 32 GB of RAM. Optimal solution to problem (P2) was obtained by running CPLEX 12.0 where we also restricted decision variables r, d and h_i to be integer, i.e., we considered the Mixed Integer Linear Programming (MILP) version of (P2). We performed experiments considering different numbers of user classes. We varied the cardinality of the set $\mathcal{U}$ between 20 and 1,000 with step 20, and

Sort			WordCount			WikiTrends		
N.	T_1^{up} gap	m_1 gap	N.	T_2^{up} gap	m_2 gap	N.	T_3^{up} gap	m_3 gap
4	5.15%	-1.51%	4	6.28%	-2.33%	4	15.77%	9.73%
4	8.01%	-0.12%	3	9.48%	-0.97%	3	23.12%	15.46%
5	2.56%	-4.51%	2	2.48%	-6.51%	4	14.40%	7.90%
4	8.79%	-0.22%	2	9.02%	-2.38%	3	16.17%	8.19%
2	3.54%	-4.25%	3	7.03%	-3.18%	5	11.32%	4.39%
2	13.98%	5.07%	3	13.05%	1.18%	4	19.55%	11.28%
4	14.74%	6.60%	2	14.79%	3.80%	4	21.14%	13.55%
5	8.64%	1.60%	4	6.90%	-2.51%	2	14.52%	7.97%
5	0.91%	-5.64%	2	2.26%	-6.75%	4	9.64%	3.37%
4	11.02%	4.40%	4	7.84%	-0.93%	4	14.43%	8.42%

Table 6: Three job clasess analysis (Sort, WordCount and WikiTrends)

run each experiment ten times.

The results show that the time required to determine global optimal solution for the MILP problem is, on average, less than 0.08 seconds. The instances of maximum size including 1,000 user classes can be solved in less than 0.5 second in the worst case.

5.4 Case Studies

In this section, we investigate how different (P2) problem settings impact on the cloud cluster cost. In particular, we analyse three case studies to address the following research questions: (1) Is it better to consider a shared cluster or to devote a dedicated cluster to individual user classes? (2) What is the effect of job concurrency on cluster cost? (3) Which is the cost impact of more strict deadlines? (is there a linear relation between the cost and job deadlines?). Instances have been generated according to Sections 5.1 and 5.3. Furthermore, to ease the results interpretation we excluded reserved instances and assumed there is a single type of VM available from the cloud provider.

5.4.1 Effect of sharing cluster

In this case study, we want to examine the effect of cluster resource sharing. In particular, we consider two scenarios. The first one is our baseline, which corresponds to (P2) problem setting. The second one considers the same resource demand (in terms of job profiles, deadlines, etc.) but $|\mathcal{U}|$ (P2) problems are solved independently, i.e., assuming a dedicated cluster is devoted to each user class. To perform the comparisons, we consider different numbers of user classes. We vary the cardinality of the set $\mathcal{U}$ between 20 and 1,000 with step 20 and randomly generate ten instances for each cardinality value. For each instance we calculate two values: the first one is the objective function of the baseline

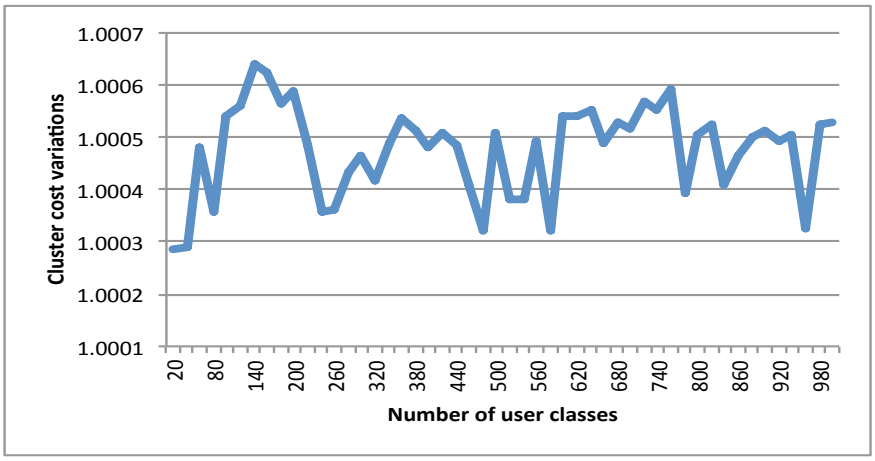

Figure 7: Effect of assuming all user classes together.

scenario, that we refer to as *dependent objective function*; the second value, that we call *independent objective function*, is evaluated by summing up the $|\mathcal{U}|$ objective functions of the individual problems. The comparison is performed by considering the ratio between the *dependent* and *independent objective function*. Figure 7 reports the average of this ratios for different numbers of user classes. Overall, the cluster cost marginally decreases by assuming all user classes together and on average we have 0.48% variation on the overall cluster cost. We can conclude that, thanks to cloud elasticity, the adoption of shared or dedicated clusters leads to the same cost. Note that, shared cluster can lead to benefits thanks to HDFS (e.g., better disk performance and node load balancing) but this can not be captured by our cost model.

5.4.2 Effect of job concurrency degree

In this case study we want to analyze the effect of the job concurrency degree on the cost of one single job. To perform the experiment, we assume there is just one user class in the cluster. We vary the job concurrency degree h_i from 10 to 30 and, for each value, we randomly generate 10 instances of problem (P2). For each instance we disable the admission control by setting up $H^{low} = H^{up}$ and we solve the optimization problem. We calculate the cost of one single job for each instance by dividing the objective function by the job concurrency degree.

Figure 8 shows how the per-job cost varies with different job concurrency degrees for a representative example. Overall, the analysis demonstrates that the cost variance for different job concurrency is negligible, i.e., the different job concurrency degree leads to less than 0.002% variation of the cost of one job. Hence, in a cloud setting, elasticity allows to obtain a constant per-job execution cost independently of the number of users in a class. This result is in line with Theorem 4.3 b).

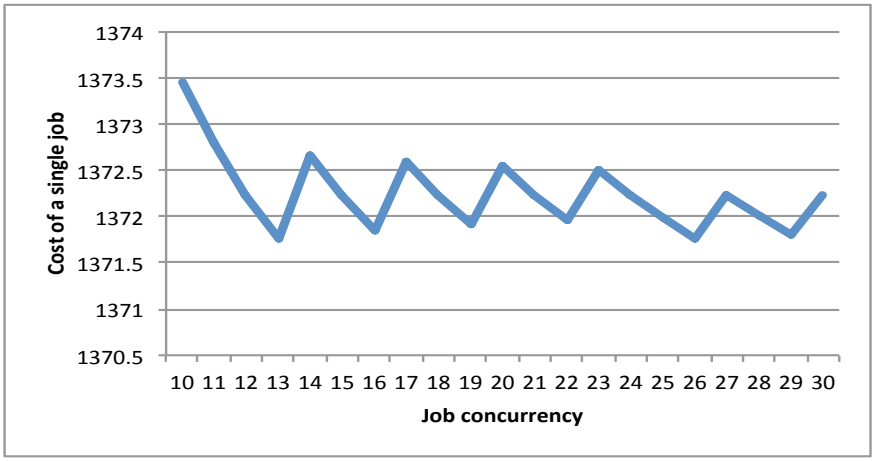

Figure 8: Effect of job concurrency degree on single job cost.

5.4.3 Effect of tightening the deadlines

Here we want to examine the relation between cost and deadlines. In particular, we check the effect of reducing the deadlines on the cluster cost. We vary the cardinality of the set $\mathcal{U}$ between 20 and 1,000 and for each cardinality we generate several random instances as described in Section 5.3. For each instance, we iteratively tighten the deadlines of every user class to observe how the changes are reflected on the cost. In each step, we decrease the deadlines by 5% of the initial value. The reduction process continues until the instance with new deadlines does not have a feasible solution. After each reduction, we calculate the increased cost ratio, i.e., the ratio between the objective function for the problem with the new deadlines and the objective function of the problem with the initial deadlines. Figure 9 illustrates the trend of the increase cost ratio for a representative instance with 20 user classes: the reduction is not linear and the cost to pay for reducing the deadlines by a 60% is more than three times with respect to the base case.

6 Related Work

Capacity management and optimal scheduling of Hadoop clusters received a lot of interest by the research community. Authors in [13] propose *Starfish*, a self-tuning System for analytics on Hadoop. Indeed, rarely Hadoop exhibits the best performance as it is, without a specific tuning phase. Starfish, collects at runtime some key informations about the job execution generating a profile that is eventually exploited to automatically configure Hadoop without human intervention. The same tool has been successful employed to solve cluster sizing problems [12].

Tian and Chen [24] face the problem of resource provisioning optimization

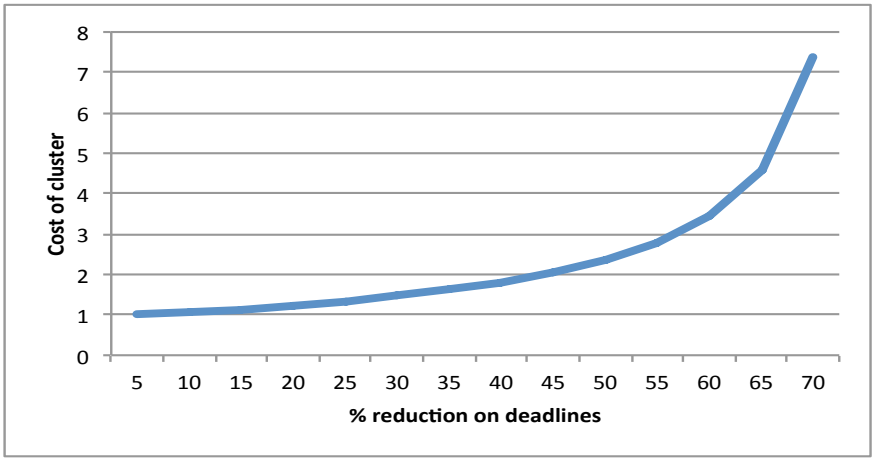

Figure 9: Effect of reducing deadlines on cluster cost.

minimizing the cost associated with the execution of a job. This work presents a cost model that depends on the amount of input data and on the considered job characteristics. A profiling regression-based analysis is carried out to estimate the model parameters.

A different approach, based on closed queuing networks, is proposed in [20] that considers also contention and parallelism on compute nodes to evaluate the completion time of a MapReduce job. Unfortunately, this approach concerns the execution time of the map phase only. Vianna et al. [28] propose a similar solution, which however, has been validated for cluster exclusively dedicated to the execution of a single job.

The work in [17] models the execution of Map task through a tandem queue with overlapping phases and provides very efficient run time scheduling solutions for the joint optimization of the Map and copy/shuffle phases. Authors show how their runtime scheduling algorithms match closely the performance of the offline optimal version.

The work in [10] introduces a novel modeling approach based on mean field analysis and provide very fast approximate methods to predict the performance of Big Data systems.

Deadlines for MapReduce jobs are considered also in [23]. The authors recognize the inability of Hadoop schedulers to handle properly jobs with deadlines and propose to adapt to the problem some well-known multiprocessor scheduling policies. They present two versions of the Earliest Deadline First heuristic and demonstrate they outperform the classical Hadoop schedulers.

The problem of progress estimation of parallel queries is addressed in [21]. The authors present Parallax, a progress estimator able to predict the completion time of queries representing MapReduce jobs. The estimator is implemented on Pig and evaluated with PigMix benchmark.

ParaTimer [22], an extension of Parallax, is a progress estimator that can

predict the completion of parallel queries expressed as Directed Acyclic Graph (DAG) of MapReduce jobs. The main improvement with respect to the previous work, is the support for queries where multiple jobs work in parallel, i.e., have different path in the DAG. Authors in [31] investigate the performance of MapReduce applications on homogeneous and heterogeneous Hadoop cloud based clusters. They consider a problem similar to the one we faced in our work and provide a simulation-based framework for minimizing infrastructural costs. However, admission control is not considered and a single type of workload (i.e., user class) is optimized.

In [26] the ARIA framework is presented. This work is the closest to our contribution and focuses on clusters dedicated to single user classes running on top of a first in first out scheduler. The framework addresses the problem of calculating the most suitable amount of resource (slots) to allocate to Map and Reduce tasks in order to meet a user-defined soft deadline for a certain job and reduce costs associated with resource over-provisioning. A MapReduce performance model relying on a compact job profile definition to calculate a lower bound, an upper bound and an estimation of job execution time is presented. Finally, such model, improved in [32], is validated through a simulation study and an experimental campaign on a 66-nodes Hadoop cluster.

7 Conclusions and Future Work

In this paper, we provided an optimization model able to minimize the execution costs of heterogeneous tasks in cloud based shared Hadoop clusters. Our model is based on novel upper and lower bounds for MapReduce job execution time. Our solution has been validated by a large set of experiments. Results have shown that our method is able to determine the global minimum solutions for systems including up to 1,000 user classes in less than 0.5 seconds. Moreover, the average execution time of MapReduce jobs obtained through simulations is within 14% of our bounds on average. Future work will validate the considered time bounds in real cloud clusters. Moreover, a distributed implementation of the optimization solver able to exploit the YARN hierarchical architecture will be developed.

Acknowledgement

The work of Marzieh Malekimajd has been supported by the European Commission grant no. FP7-ICT-2011-8-318484 (MODAClouds). Danilo Ardagna and Michele Ciavotta's work has been partially supported by the the European Commission grant no. H2020-644869 (DICE). The simulations and numerical analyses have been performed under the Windows Azure Research Pass 2013 grant.

References

[1] Capacity Scheduler. http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[2] Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2.

[3] Fair Scheduler. http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[4] MapReduce: Simplified Data Processing on Large Clusters. http://research.google.com/archive/mapreduce.html.

[5] Microsoft Azure. http://azure.microsoft.com/en-us/services/hdinsight/.

[6] The digital universe in 2020. http://idcdocserv.com/1414.

[7] YARN Scheduler Load Simulator (SLS). http://hadoop.apache.org/docs/r2.3.0/hadoop-sls/SchedulerLoadSimulator.html.

[8] J. Anselmi, D. Ardagna, and M. Passacantando. Generalized Nash Equilibria for SaaS/PaaS Clouds. *European Journal of Operational Research*, 236(1):326–339, 2014.

[9] D. Ardagna, B. Panicucci, and M. Passacantando. Generalized Nash Equilibria for the Service Provisioning Problem in Cloud Systems. *IEEE Transactions on Services Computing*, 6(4):429–442, 2013.

[10] A. Castiglione, M. Gribaudo, M. Iacono, and F. Palmieri. Exploiting mean field analysis to model performances of big data architectures. *Future Generation Computer Systems*, 37(0):203–211, 2014.

[11] C. P. Chen and C.-Y. Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences*, 275(0):314 – 347, 2014.

[12] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *SOCC '11*, pages 18:1–18:14, 2011.

[13] 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 *CIDR '11*, pages 261–272, 2011.

[14] H. V. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, and C. Shahabi. Big data and its technical challenges. *Commun. ACM*, 57(7):86–94, 2014.

[15] K. Kambatla, G. Kollias, V. Kumar, and A. Grama. Trends in big data analytics. *Journal of Parallel and Distributed Computing*, 74(7):2561–2573, 2014.

[16] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: A survey. *SIGMOD Rec.*, 40(4):11–20, 2012.

[17] M. Lin, L. Zhang, A. Wierman, and J. Tan. Joint optimization of overlapping phases in MapReduce. *SIGMETRICS Performance Evaluation Review*, 41(3):16–18, 2013.

[18] M. Malekimajd, A. M. Rizzi, D. Ardagna, M. Ciavotta, M. Passacantando, and A. Movaghar. Optimal Capacity Allocation for executing Map Reduce Jobs in Cloud Systems. Technical Report n. 2014.11, Politecnico di Milano, `http://home.deib.polimi.it/ardagna/MapReduceTechReport2014-11.pdf`.

[19] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data: The next frontier for innovation, competition, and productivity. McKinsey Global Institute, 2012.

[20] D. A. Menascé and S. Bardhan. Queuing Network Models to Predict the Completion Time of the Map Phase of MapReduce Jobs. In *38th International Computer Measurement Group Conference*, 2012.

[21] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: A Progress Indicator for MapReduce DAGs. In *SIGMOD '10*, pages 507–518, 2010.

[22] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of MapReduce pipelines. In *ICDE '10*, pages 681–684, 2010.

[23] L. T. X. Phan, Z. Zhang, Q. Zheng, B. T. Loo, and I. Lee. An empirical analysis of scheduling techniques for real-time cloud-based data processing. In *SOCA'11*, pages 1–8, 2011.

[24] F. Tian and K. Chen. Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds. In *CLOUD '11*, pages 155–162, 2011.

[25] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SOCC '13*, pages 5:1–5:16, 2013.

[26] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments. In *ICAC '11*, pages 235–244, 2011.

[27] A. Verma, L. Cherkasova, and R. H. Campbell. Resource Provisioning Framework for Mapreduce Jobs with Performance Goals. In *Middleware'11*, pages 165–186, 2011.

[28] E. Vianna, G. Comarela, T. Pontes, J. M. Almeida, V. A. F. Almeida, K. Wilkinson, H. A. Kuno, and U. Dayal. Analytical Performance Models for MapReduce Workloads. *International Journal of Parallel Programming*, 41(4):495–525, 2013.

[29] F. Yan, L. Cherkasova, Z. Zhang, and E. Smirni. Heterogeneous cores for MapReduce processing: Opportunity or challenge? In *NOMS '14*, pages 1–4, 2014.

[30] Q. Zhang, Q. Zhu, M. Zhani, and R. Boutaba. Dynamic Service Placement in Geographically Distributed Clouds. In *ICDCS '12*, pages 526–535, 2012.

[31] Z. Zhang, L. Cherkasova, and B. T. Loo. Exploiting Cloud Heterogeneity for Optimized Cost/Performance MapReduce Processing. In *CloudDP '14*, pages 1:1–1:6, 2014.

[32] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Automated Profiling and Resource Management of Pig Programs for Meeting Service Level Objectives. In *ICAC '12*, pages 53–62, 2012.