Zacheilas and Kalogeraki *EURASIP Journal on Embedded Systems* (2017) 2017:29
DOI 10.1186/s13639-017-0077-7

EURASIP Journal on Embedded Systems

RESEARCH Open Access

A Pareto-based scheduler for exploring cost-performance trade-offs for MapReduce workloads

Nikos Zacheilas* and Vana Kalogeraki

Abstract

In recent years, we are observing an increased demand for processing large amounts of data. The MapReduce programming model has been utilized by major computing companies and has been integrated by novel cyber physical systems (CPS) in order to perform large-scale data processing. However, the problem of efficiently scheduling MapReduce workloads in cluster environments, like Amazon's EC2, can be challenging due to the observed trade-off between the need for performance and the corresponding monetary cost. The problem is exacerbated by the fact that cloud providers tend to charge users based on their I/O operations, increasing dramatically the spending budget. In this paper, we describe our approach for scheduling MapReduce workloads in cluster environments taking into consideration the performance/budget trade-off. Our approach makes the following contributions: (i) we propose a novel Pareto-based scheduler for identifying near-optimal resource allocations for user workloads with respect to performance and monetary cost, and (ii) we develop an automatic configuration of basic tasks' parameters that allows us to further minimize the user's spending budget and the jobs' execution times. Our detailed experimental evaluation using both real and synthetic datasets illustrate that our approach improves the performance of the workloads as much as 50%, compared to its competitors.

Keywords: MapReduce, Scheduling, CPS, Big data

1 Introduction

In recent years, we observe a tremendous increase in the amount of data that needs to be stored and processed. Analyzing this large amount of data is a high priority task for many companies. Moreover, the sheer volume of data exceeds the capabilities of existing commercial databases. For this reason, novel distributed processing frameworks have been proposed. Google's MapReduce [1] is the most commonly used *parallel* and *distributed* programming model for performing Big Data processing on clusters of commodity hardware. Hadoop [2], which is MapReduce's most popular open-source implementation, is utilized by major companies including Twitter [3] and Yahoo [4] for analyzing their workloads, mainly due to its scalability features [5]. Furthermore, even traditional industries, such as banking and telecommunications, are adopting the use of Hadoop in their environments, due to their demand of processing fast-growing volumes of data [6].

One area where the MapReduce programming model is efficiently applied is in the context of Cyber Physical Systems (CPS). CPS are integrations of computational resources (e.g., commodity machines) with physical processes [7, 8]. In most cases, embedded networked computers control and monitor physical resources (i.e., sensors), usually with feedback loops, where the physical resources are strongly coupled and interact with the computational resources. Examples of Cyber Physical Systems range from traffic monitoring systems [9, 10] to smart grids [11]. In order to perform the feedback loop, it is necessary to process historical data and build models (e.g., using data mining techniques [12]) for the physical resources. Due to the high volume of data, it is common practice to use Big Data frameworks like Apache's Hadoop or Apache's Spark [13] for processing the historical data. Many CPS [12, 14] execute their MapReduce applications on public cloud infrastructures [15] like Amazon's EC2 [16] or

*Correspondence: zacheilas@aueb.gr
Department of Informatics, Athens University of Economics and Business, Athens, Greece

Microsoft's Azure [17] as these offer the necessary computing resources and enable the automatic deployment of the Hadoop framework. Usually, users of cloud infrastructures are charged based on the amount of processing and storage resources they reserve [18]. For example, in Amazon EC2 [16], users are charged on a per hour basis based on the amount of time they bind the reserved virtual machines (VMs) and also based on the amount of I/O operations performed by their applications [19].

Challenge 1. A challenging task for users in such environments is to efficiently determine how many VMs they should bind in order to satisfy their performance goals (e.g., minimize their jobs' execution times) without overspending. More specifically, a MapReduce job comprises multiple map/reduce tasks that execute on the available map/reduce slots. The number of map/reduce slots of the MapReduce job essentially correspond to the VMs' processing cores that will be used for the execution of the job. Increasing the number of slots used by a job, typically increases the number of VMs that will be reserved and this can impact the user's budget (assuming a typical pricing policy like Amazon's). The problem becomes more challenging when users submit multiple jobs (i.e., MapReduce workload) concurrently to the cluster. In this case, the possible slots' allocations can grow exponentially and thus determining the allocation that minimizes the workload's end-to-end execution time (i.e., makespan) and at the same does not overcharge the user, is not trivial.

Challenge 2. The second important challenge derives from the fact that the performance and the monetary cost of MapReduce jobs can be affected by a wide range of configuration settings. Hadoop has over 190 configuration parameters [20] (in Table 1, we present the most representative ones). Currently, the burden of tuning these parameters falls on the user who submits the jobs. Many users lack the expertise to properly configure these parameters, and this can easily lead to serious performance degradation [21]. Companies like Cloudera [22] provide rule of thumb recommendations for the basic configuration parameters; however, such approaches cannot be easily applied to a wide range of applications [20].

Table 1 Hadoop's basic configuration parameters

Parameter	Description	Default value
Mapred.reduce.tasks	Number of reduce tasks	1
Io.sort.mb	Map buffer size in MB	100
Io.sort.record.percent	The percentage of the map buffer's size used for metadata	0.05
Io.sort.spill.percent	Threshold in the map buffer's size that if exceeded in-memory data are stored in local files	0.80
Dfs.block.size	HDFS block size, determines the number of map tasks	128 m

In Fig. 1, we illustrate with an example, the effect of the configuration parameters on the performance of a typical Twitter friendship job and a Sort job. The Twitter job processes 1.2 GB tweets and identifies all unique mentioned and tagged friends per user, while the Sort job sorts 5 GB of randomly generated data (for more information about the jobs see Section 6). We run the jobs in our local 8-VM cluster (i.e., 1 Master and 7 Worker nodes), which consists of 14 map and 14 reduce slots. As you can observe in Fig. 1, with appropriate tuning of the reduce tasks parameter (i.e., *mapred.reduce.tasks* parameter in Table 1), we are able to reduce the execution time of a job by half. In the specific experiment, you can observe that by increasing the number of reduce tasks, we exploit the available parallelism (i.e., the available reduce slots) and thus we can decrease the execution time. However, increasing beyond the available resources can be sub-optimal as this can increase the execution rounds of the reduce tasks. More specifically, if we use more reduce tasks than the available 14 reduce slots, not all tasks can run in parallel and this increases the execution time. Figure 1 also illustrates that by tuning the buffer size in the map phase (i.e., *io.sort.mb* parameter), we can decrease the amount of data that is read and written by the map tasks, reducing considerably the performed I/O operations.

In MapReduce environments, a few recent works study the problem of scheduling MapReduce jobs on cluster resources in order to minimize standard scheduling theory metrics [23, 24]. More specifically, in [23], they minimize the workload's makespan while in [24], they focus on maximizing a profit metric which depends on the amount of jobs that satisfy their execution time deadlines. However, both techniques do not consider the impact of the scheduling decisions (i.e., allocation of map/reduce slots) on the user's spending budget. Furthermore, works like [20, 25] have been proposed for automatically tuning the job's configuration parameters (e.g., number of reduce tasks) in order to minimize its execution time. However, all these works do not examine how the available slots should be allocated among concurrently running jobs to satisfy both budget and performance constraints. Moreover, another parameter that is not taken into account in these works is how the I/O operations can affect the user's budget and how by controlling the buffer-related parameters we can minimize its impact. To the best of our knowledge, we are the first to focus on the multi-objective optimization problem of minimizing the makespan and required budget for the execution of MapReduce workloads in a single Hadoop cluster. Solving this problem is beneficial to all users of the Hadoop ecosystem including

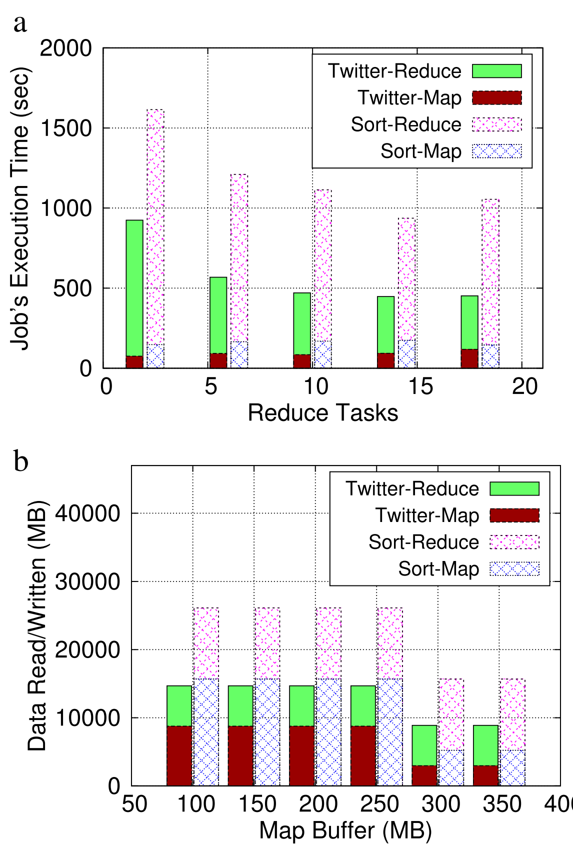


Fig. 1 Impact of configuration parameters (i.e., number of reduce tasks and map buffer size) on the job's end-to-end execution time and the amount of data that are read/written by the map and reduce tasks. **a** Reduced tasks impact on job's execution time. **b** Map buffer size impact on the job's I/O operations

CPS [12] that utilize Hadoop to generate performance models (i.e., using data mining MapReduce jobs) for the physical resources they monitor.

1.1 Contributions

In this work, we aim at providing a novel MapReduce scheduler that will enable the study of the makespan/monetary cost trade-off for MapReduce workloads that execute on homogeneous VMs in public cloud infrastructures like Amazon's EC2 [16], and at the same time will automatically tune the jobs' configuration parameters to improve further the jobs' performance and reduce the spending budget. The key contributions of our work are as follows:

- We introduce the design and implementation of a novel Pareto-based scheduler that, given a user's MapReduce workload and budget range, detects map/reduce slot allocations that minimize the workload's makespan without overcharging the user. Our approach is able to detect all valid map/reduce slots allocations for a given budget range and to present them to the user. To the best of our knowledge, our scheduler is the first that enables the user to examine the trade-off between the two metrics of interest and decide the slots' allocation that meets her requirements. Our scheduler differs from all previous schemes that schedule Hadoop jobs [23, 24], in the fact that we solve a multi-objective optimization problem that targets at minimizing both the workload's makespan and the user's budget, while all previous schedulers aim at minimizing only one performance objective. Furthermore, our proposed scheduler is the first that provides all near optimal slots' allocations. We argue that our scheduler can be helpful for all Hadoop users in multiple application domains (including CPS) that execute their applications (e.g., transportation [9], Mahout [26] applications) on public cloud infrastructures as they are able to observe the expected workload's makespan for different spending budgets and then determine the slots' allocations they consider most appropriate for their workload. We believe that our approach is more generic and more useful as it provides more choices to the user compared to a single solution which is the aim of the previous works (that minimize one of the two metrics of interest, either the makespan or the budget).
- We use a novel Pareto frontier greedy search algorithm for detecting near-optimal slots' allocations in a fast and scalable way, considering two different policies for traversing the frontier. The first policy, *Slots' Allocation*, assigns an extra slot (i.e., either map or reduce) to the job that affects the most the workload's makespan while the second, *VMs' Allocation*, reserves an extra VM for this job. Slots' Allocation searches the frontier more thoroughly as it gives one slot at a time, so we expect it to take longer to detect all the valid allocations. In contrast, VMs' Allocation is more lavish as it allocates one VM at each iteration step so we expect it to finish the search faster but it will detect less plans than the Slots' Allocation algorithm.
- We enhance the system's performance and decrease the user's spending budget through the automatic configuration of five basic MapReduce configuration parameters. More specifically, we tune the number of map and reduce tasks as we observed that by tuning these two parameters, we can decrease further the job's execution time. Furthermore, we consider three configuration parameters that control the map buffer size. These buffer-related parameters affect the number of I/O operations and thus the job's monetary cost. We formulate the impact of these five

parameters on the job's execution time and monetary cost and propose two greedy algorithms for automatically tuning them. We avoid more elaborate techniques like [27] as we want to tune the parameters as fast as possible in order to be able to utilize our techniques in real-time. For the adjustment of the map-buffer parameters, we use a Random Search algorithm in order to sample fast the search space and detect the parameters that minimize the I/O cost. While for the map/reduce task adjustment, we use a Hill climbing algorithm that gradually increases the number of map and reduce tasks used for the job, keeping the configuration that minimizes its execution time.

- We conduct an extensive evaluation study using both simulation and empirical workloads. For the simulation workloads, we use traces from Yahoo! [28] and Taobao [29], while for the empirical workloads, we execute short-running jobs from the PUMA benchmark [30] to display the applicability of our framework in commonly used workloads from major companies like Twitter. Our experimental results illustrate that our search algorithm outperforms state-of-the-art schemes by a few orders of magnitude in terms of execution time. Finally, our automatic parameters configuration approach minimizes further the required budget and the workload's end-to-end execution time.

2 Background

In this section, we first provide a brief introduction on the MapReduce programming model and then describe how MapReduce is utilized by current CPS.

2.1 MapReduce preliminaries

In this section, we provide a concise description of the MapReduce programming model and put emphasis on Hadoop's implementation.

2.1.1 Programming model

A MapReduce job is modelled as a sequence of two computational phases, the map and the reduce phase. Each phase comprises multiple tasks that execute in parallel (i.e., see Fig. 2). Tasks are modelled as follows: map$(k_1, v_1) \Rightarrow [k_2, v_2]$ and reduce$(k_2, [v_2]) \Rightarrow [k_3, v_3]$. Map tasks take as input (k_1, v_1) pairs and return a list of intermediate *(key,value)* pairs of possibly different types, k_2 and v_2. The values associated with the same key are grouped together into a list and are passed as input to the appropriate reduce task using a partitioning function. Finally, the reduce tasks emit arbitrary *(key,value)* pairs of a final type, k_3 and v_3.

2.1.2 Job execution

The cluster's processing resources determine the maximum number of map and reduce tasks that can execute concurrently. The cluster provider sets these resources by determining the number of slots that can run on the cluster nodes. Slots are used to execute map and reduce tasks. In most cases, each node is configured with one slot per processing core. Users in public clouds can reserve a fixed number of nodes (and the corresponding slots) for the execution of their jobs based on the budget they are willing to spend [18]. If the reserved slots are fewer than the job's tasks, the latter are added in a FIFO queue and wait for their execution. In Fig. 2, we illustrate how a job runs on a Hadoop cluster. The job consists of five map and four reduce tasks, while three map and two reduce slots have been reserved. Thus, the job requires two *rounds* (also called *waves*) of task execution in the reserved slots in order to finish.

2.1.3 Map task execution

Map tasks usually read their input data from the Hadoop's distributed file system (HDFS). For each input file, a map task is spawned. When the map function is invoked and starts generating output data, it does not simply write them on disk. As shown in Fig. 2, a more complex process takes place exploiting the use of in-memory buffers and some presorting. Each map task contains a circular memory buffer (with default size 100 MB) where the output data are kept. Apart from the actual data, for each output record, 16 bytes of metadata are kept and used for sorting, partitioning, and indexing. When the buffer reaches a certain threshold (by default 80% of its size) either due to the actual data or metadata sizes, a background thread spills the output records to disk (*spill files* in Fig. 2). Nowadays, we observe a trend from major cloud providers (e.g., Amazon) to charge users based on the amount of I/O operations in the local disks [19]. So, the buffers used by the map tasks can affect significantly this cost as they can minimize the required I/O operations by keeping more data in memory.

2.1.4 Reduce task execution

Initially, reduce tasks fetch the output data generated by the map tasks and stored on the local disks, using a partitioning function. This procedure is called *shuffle* and can affect significantly the execution time as it depends on the available cluster bandwidth. The retrieved data are stored on the memory buffer for reduce tasks. The size of the buffer is controlled via the *mapred.job.shuffle.input.buffer.percent* (set by default to 70%) parameter which specifies the proportion of the task's heap size that will be used for the buffering. When all data have been fetched, the reduce function is invoked and the output data are stored in the HDFS. The number of reduce tasks (by default the value is equal to 1) is user-specified via the *mapred.reduce.tasks* configuration parameter. Using the default, reduce task parameter may

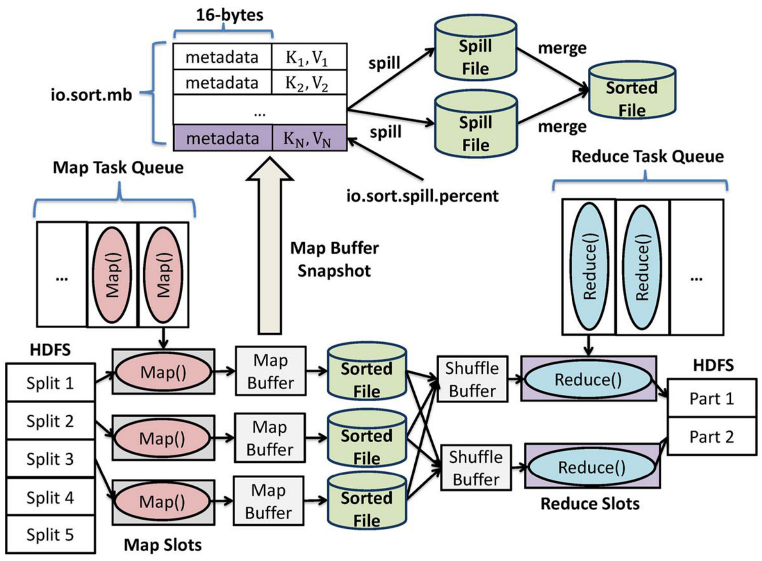


Fig. 2 MapReduce job execution in a Hadoop cluster

not be beneficial because it can lead to under-utilization of the reserved slots. For example in Fig. 2, if we used only one reduce task, we would not exploit fully the available parallelism (i.e., the other one reserved reduce slot).

2.2 CPS and MapReduce

In recent years, we see the wide adoption of novel Big Data processing frameworks by CPS mainly due to the large volume of data that they generate. As you can see in Fig. 3, CPS consist of two layers [7], the physical and the computational. The physical layer comprises a wide variety of input sensors that periodically send reports to the computational layer and also expect input from these computational resources in order to dynamically adapt to possible changes in the environment. In Fig. 3, we display a traffic monitoring CPS [9, 10, 31] that we have deployed in the city of Dublin. The physical layer receives input from sensors mounted on top of public buses, SCATS sensors mounted on road intersections, CCTV cameras, and mobile applications. All these sensors periodically report the traffic conditions in the city. The computational layer is responsible to process these reports and to inform the sensors to take precautionary actions if it is necessary. For example, it may inform the users of the mobile application to change their route if we expect heavy traffic in the upcoming minutes in specific traffic routes. Therefore, we have a feedback loop between the computational and the physical

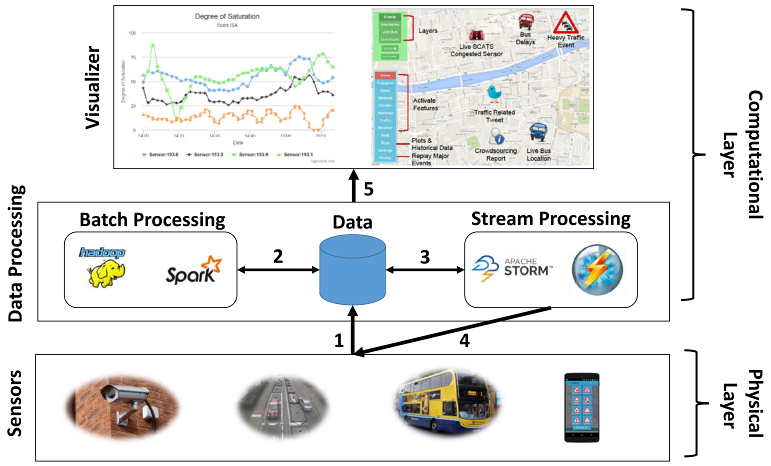


Fig. 3 An example of a traffic monitoring CPS which exploits Big Data frameworks

layer as physical resources affect the computations and vice versa.

The computational layer needs to exploit the improvements and benefits offered by the latest Big Data frameworks in order to be able to handle the sheer volume of data produced by the physical layer. More specifically, the system must be able to take actions as fast as possible and adapt to possible changes in the environment. For example, if a traffic accident is detected by the CCTV cameras, all the nearby users of the mobile app must be informed. So we need to be able to process all the incoming images in real-time and also inform users in case of such an event. For this reason, most CPS [12] adopt the use of distributed stream processing systems like Apache's Storm [32] or IBM's Streams [33]. Furthermore, the CPS must be able to detect abnormal behavior so we have to analyze past sensors' reports and model the expected behavior overtime using data mining techniques [12]. Novel batch processing frameworks based on the MapReduce programming model like Apache's Hadoop or Apache's Spark are used for these computations as they can handle the high volume of historical data by scaling the processing in multiple computing nodes.

Hadoop is the most widely used framework for the batch processing computations in CPS and has been applied in different settings like smart grids [11] and traffic monitoring [9]. Optimizations on top of Hadoop can improve the performance of the whole CPS. For example, because multiple jobs are executed concurrently (i.e., a MapReduce workload) in order to generate the sensors' performance models (e.g., different jobs compute models for the bus sensors and the CCTV cameras) if we minimize the workload's makespan, the CPS will be able to utilize the generated models faster in the stream processing component and thus inform the physical layer more quickly in case of a change in the current conditions. Finally, as we mentioned in Section 1, Hadoop usually runs in public cloud infrastructures where users are charged based on the amount of reserved resources so CPS will also benefit from techniques that target at minimizing the jobs' monetary cost. Therefore, we argue that the study of the budget/makespan trade-off is very important for CPS' that adopt the use of Hadoop as their batch processing component.

3 System model

In this section, we describe the basic parameters of our approach. Our model is based on MapReduce 1 but recently Hadoop has produced a new version, MapReduce 2 [34]. The main difference of the two versions is the fact that MapReduce 2 no longer considers separately the slots used for the map and reduce tasks, but for each node, it defines the number of containers it can support. In each container both map and reduce tasks can run. Our model can be easily extended to support this new version of Hadoop as we still have to determine how the containers should be allocated for the execution of map and reduce tasks (i.e., how many container will be used for map/reduce tasks).

Initially, the cluster administrator needs to provide the following information:

- *msPerVM*, *rsPerVM*: the number of map and reduce slots per VM. So in each VM, we have both map and reduce slots. This parameter depends on the VM's CPU cores. For example, if a VM has 4 CPU cores, we can use two cores as map slots (i.e., msPerVM = 2) and the other two cores as reduce slots (i.e., rsPerVM = 2). It should be clear that map slots are used for the execution of map tasks while reduce slots are utilized for the execution of reduce tasks. These parameters are not workload specific as the cluster may be used by multiple users. For example, in major companies like Yahoo!, multiple users submit their workloads concurrently in the cluster so the administrator has to determine how many map and reduce slots will be available per cluster's VM. Furthermore, when these parameters are changed, the Hadoop cluster needs to be restarted therefore it is not possible to modify their values during the execution of jobs.
- *Cost*: the per hour cost for reserving a VM in the cluster. We follow a cost policy like Amazon's EC2 [16] and Microsoft's Azure [17] where the budget that the user will pay depends on the amount of time she has reserved a VM multiplied by the per hour cost.
- *IOCost*: the monetary cost of an I/O operation. We consider as I/O operation the data/metadata reads or writes to the local disk.

Each user's workload comprises a set of Hadoop jobs (*Jobs*). A user in our setting can be CPS like [11] that use Hadoop for the data analysis of historical data and exploit the processing resources of a public cloud infrastructure like Amazon's EC2. The submitted workload is characterized by the following metrics:

- $Budget_{Min}$, $Budget_{Max}$: the minimum/maximum budget the user is willing to spend for executing the workload's jobs. In our system, we wanted to enable even a non-expert to provide us feedback in terms of the money he/she is willing to spend. In the worst case scenario, the user will provide a budget range where our scheduler cannot find a feasible solution (i.e., very small $\text{Budget}_{\text{Min}}$ and $\text{Budget}_{\text{Max}}$ parameters for the amount of work that needs to run) so in such cases he/she will be informed that no resource allocation is possible and the user will have to provide new values for these parameters. If the user does not provide $\text{Budget}_{\text{Min}}$, we solve a single-objective optimization problem that minimizes the user's spending budget for the current workload and then we use this value as $\text{Budget}_{\text{Min}}$. While this approach automatically determines the $\text{Budget}_{\text{Min}}$ parameter it also adds extra computational cost as we have to solve an extra optimization problem.

- *Budget*: the actual budget the user will spend for the execution of the workload.
- *Deadline*: the constraint imposed by the user on the workload's end-to-end execution time.
- *Makespan*: the workload's end-to-end execution time after we determined the map/reduce slots to use for the workload's jobs. Ideally, this value should be smaller than Deadline; if this is not achieved we assume that we have a service level agreement (SLA) violation as we are not able to satisfy the user's performance requirements.

Each job $j \in Jobs$ is characterized by the following parameters:

- ms_j, rs_j: the number of map/reduce slots that will be reserved for job j.
- VMs_j: the number of VMs used by job j calculated via the following formula:

$$\text{VMs}_j = \max\left(\lceil \frac{\text{ms}_j}{\text{msPerVM}} \rceil, \lceil \frac{\text{rs}_j}{\text{rsPerVM}} \rceil\right) \quad (1)$$

  In each VM, both map and reduce slots are available. If the job requires more map slots than reduce slots then the number of reserved VMs will depend on the map slots; otherwise, it will be determined by the number of reduce slots reserved by the job.
- mt_j, rt_j: the number of map/reduce tasks used by job j.
- $budget_j$: the monetary cost for running job j. This metric depends on the amount of occupied VMs, the time that these VMs are reserved and the cost of the I/O operations.
- $mtime_j, rtime_j, stime_j$: the estimated execution time of the map, reduce, and shuffle phases of job j.
- $jtime_j$: the estimated execution time in seconds of the whole job which depends on the estimations of the three previous metrics.
- $ioCost_j$: the monetary cost of the I/O operations performed by job j.
- $mBuffer_j$: the size of the buffer used by the map tasks (controlled by Hadoop's *io.sort.mb* parameter). This value depends on the memory that has been allocated to the JVM that will execute the task. For example, if the cluster consists of nodes that have 8 GB RAM and we want to use one map and one reduce slot in each node then the maximum value of mBuffer_j will be 4 GB.
- metaPercent_j: the percentage of the map buffer size that will be used for storing metadata, configured by the *io.sort.record.percent* parameter. This parameter is significantly smaller than the percentage used for the actual data; however, it is important as it can lead to increased I/O operations if its value is too small. The interval of this parameter is defined via the following equation:

$$0 < \text{metaPercent}_j < 1 \quad (2)$$

- $bThr_j$: the threshold in the mBuffer_j and $\text{metaPercent}_\text{j}$ parameters which if exceeded, data are spilled to disk. Similarly to the above parameter, this interval is defined via the following formula:

$$0 < bThr_j < 1 \quad (3)$$

- $outSize_j(mt_j)$: the size in bytes of a map task's intermediate data when mt_j map tasks are used. The number of map tasks will affect this metric as the output size will decrease when more map tasks are utilized.
- $outRec_j(mt_j)$: the amount of intermediate data records generated by a map task of job j when mt_j map tasks are used for the job. Each output record adds 16 bytes metadata information in the map buffer. We expect fewer output records per task when we increase the number of map tasks.
- $recSize_j(mt_j)$: the average record size in bytes which is computed via the following formula:

$$\text{recSize}_j(\text{mt}_j) = \frac{\text{outSize}_j(\text{mt}_j)}{\text{outRec}_j(\text{mt}_j)} \quad (4)$$

4 Our methodology

Our goal in this work is twofold: First, we aim at developing a novel Hadoop scheduler that will provide a set of possible slots' allocations for the users' workloads based on the makespan-cost trade-off so that they can choose the allocation that satisfies their requirements. Second, we seek at enhancing the performance of the users' jobs (i.e., reduce their end-to-end execution time) and decrease further the monetary cost by adjusting *five* basic configuration parameters. More specifically, we adjust the *number of map/reduce tasks* (i.e., mt_j, rt_j) as these parameters affect significantly the duration of the map/reduce phase. Furthermore, similar to Amazon's EBS [19], we consider the impact of the jobs' I/O operations on the observed monetary cost. For this reason, we tune the three basic *map-buffer* parameters (i.e., mBuffer_j, bThr_j, metaPercent_j) that control the amount of intermediate data that will be spilled to local disks, trying to minimize the impact of the storage cost on the total budget that will be spent by the user.

In the following sections, we describe the methodology we follow to tackle the aforementioned problems. Our approach consists of the following steps:

1. We formulate the multi-objective optimization problem we are solving (i.e., Section 4.2).
2. We propose a novel Pareto frontier search algorithm that generates map/reduce slots' allocations, given the user's budget range. Furthermore, we extend the algorithm by automatically tuning the number of map/reduce tasks, minimizing further the workload's makespan (i.e., Section 4.3).
3. Finally, we model the impact of the map-buffer configuration parameters on the user's budget and provide an algorithm for automatically tuning them (i.e., Section 4.4).

4.1 High level overview

In Fig. 4, we display the high level overview of our proposed scheduling approach. The user interacts with the system by providing a set of jobs to execute concurrently in the cluster and then our scheduler computes all the nearly optimal resource allocations so that the user can choose the one that best fits his needs. Furthermore, the proposed scheduler is capable of automatically tuning basic configuration parameters that impact both the user's budget and the workload's makespan.

As can be observed in Fig. 4, our scheduler consists of the following steps:

- The user submits a set of jobs to execute in the cluster. For each job, the user has to provide the implementation of the map and reduce functions.
- Our Pareto-search algorithm receives the user's workload and is responsible for determining all the slots' allocation plans that minimize both the makespan and the spending budget (discussed in detail in Algorithm 1 in Section 4.3).
- The Pareto-search algorithm traverses the search space using one of two policies, i.e., either the Slots' Allocation policy (i.e., Algorithm 2 in Section 4.3) or the VMs' Allocation policy (i.e., Algorithm 3 in Section 4.3). Depending on the policy used, we can end up with a different slots' allocation.
- When the Pareto-search algorithm has reserved resources (using either the Slots' or the VMs' allocation technique) for a job it invokes the Task Adjustment component in Fig. 4 to tune the number of map/reduce tasks (i.e., $\text{mt}_j, \text{rt}_j, \forall j \in \mathit{Jobs}$) for this job in order to minimize further the job's execution time (i.e., Algorithm 4 in Section 4.3).
- When all the near-optimal slot allocations plans have been generated by the Pareto-search algorithm, they are forwarded to the Buffer-Tuning component. The latter is responsible for tuning the $\text{mBuffer}_j, \text{bThr}_j, \text{metaPercent}_j, \forall j \in \text{Jobs}$ to minimize further the per job I/O cost and thus the total user's spending budget (i.e., Algorithm 5 in Section 4.4).

4.2 Our multi-objective optimization problem

In order to formally define our multi-objective optimization problem, we need to estimate the impact of the possible slots' allocation on the workload's *makespan* and user's *budget*. In our problem, we made the following assumptions: (i) there is no cap on the number of available VMs in the public cloud infrastructure, (ii) each VM is used for the execution of a single job (upon the job completion, the VM is turned-off if no other job needs to be scheduled), and (iii) the cluster comprises homogeneous VMs. We claim that our assumptions are valid as the number of VMs provided by major cloud providers like Amazon's EC2 are significantly larger than the amount of VMs that can be reserved by a single user. Regarding the second

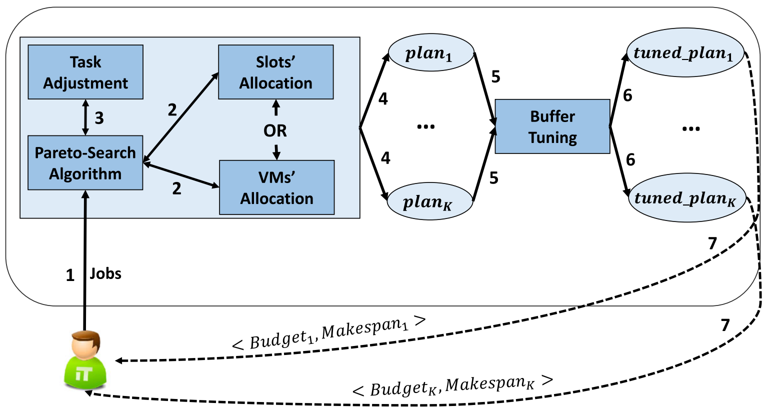


Fig. 4 Pareto-based scheduler's high level overview

assumption, we consider workloads where the jobs execute concurrently on the available resources. However, we argue that if this assumption does not hold, our approach can still be applied by splitting the workload in smaller sets of jobs and schedule each set by applying our proposed techniques.

4.2.1 Makespan computation

First, we describe how we compute the workload's makespan. There is a plethora of techniques for estimating this metric, such as building job profiles based on previous executions [35] or using debug runs prior to the actual job execution [36]. The main benefit of the job profiling approach is that it does not require the invocation of extra jobs as it exploits the already available historical data. The debug run techniques run the job twice and they estimate the execution time of the second invocation using the execution time observed in the first run. The benefit of this technique is that it does not require to keep historical data or train complex models for capturing the execution time. In our scheduler, we exploit the first approach as we focus on the execution of repetitive but aperiodic jobs so we assume that we have the required historical data for building the job profiles. Furthermore, we avoid the use of debug runs as they execute each job two times and thus they penalize the workload's makespan. For this reason, we applied the current state-of-the art job-profiling approach [35] which is effective both in homogeneous and heterogeneous environments [37] and has been efficiently applied by various recent works [38–40]. This prediction technique works well in cases where jobs' tasks have similar execution time requirements which is the vast majority of jobs executing on Hadoop clusters. The execution time of a MapReduce job is modelled as the sum of the execution times of the three different computation phases, specifically the map, the shuffle, and the reduce phases. Each phase is bounded by a lower and an upper limit.

In order to compute the lower and upper bounds, we exploit the following theorem (similarly to [35]):

Makespan Theorem: The makespan of a greedy task assignment is at least $\lceil n/k \rceil \cdot \mu$ and at most $\lceil (n-1)/k \rceil \cdot \mu + \lambda$

where $\mu = (\sum_{i=1}^{n} T_i)/n$ is the mean duration of n tasks (i.e., T_i the execution time of task i) and $\lambda = \max_i\{T_i\}$ the maximum task duration. The lower bound is trivial to compute as the best case is when all n tasks are equally distributed among the k available resources and require approximately the same execution time. Thus, the overall makespan is at least $\lceil n/k \rceil \cdot \mu$. For the upper bound, consider the worst case scenario, i.e., the longest task T' with duration λ is the last processed task. In this case, the time elapsed before the final task T' is scheduled is at most the following: $(\sum_{i=1}^{n-1} T_i)/k \leq \lceil (n-1)/k \rceil \cdot \mu + \lambda$. Thus, the makespan of the overall assignment is at most ($\lceil (n-1)/k \rceil \cdot \mu + \lambda$). The difference between lower and upper bound provides the range of possible completion times for the jobs due to non-determinism and scheduling. These bounds can be very useful in cases when $\lambda \ll \lceil n/k \rceil \cdot \mu$ (i.e., when the duration of the longest task is small as compared to the total makespan). We argue that the vast majority of MapReduce applications have similar behavior; thus, we exploit this prediction model in our setting.

We provide only the equations used for the estimation of the map phase's execution time but their applicability for the other two phases is trivial. The lower limit of the execution time of the map phase is given via the following formula:

$$\text{mtime}_j^{low} = \lceil \frac{\text{mt}_j}{\text{ms}_j} \rceil \times t_{avg}(\text{mt}_j), \forall j \in \text{Jobs} \tag{5}$$

where $t_{avg}(mt_j)$ is the average execution time of the map tasks and it includes both the CPU time and the time required for I/O operations (i.e., $t_{avg}(\text{mt}_j) = tcpu_{avg}(\text{mt}_j) + tio_{avg}(\text{mt}_j)$). It should be clear that jobs execute on separate VMs so there is no sharing of the I/O resources and the CPU cores, and thus there is no overhead on the map phase's execution time. The ratio $\lceil \frac{\text{mt}_j}{\text{ms}_j} \rceil$ depicts the impact of the execution *rounds* (map or reduce) on the expected execution time. Using fewer slots than the assigned tasks leads to an increase in the observed execution time as the phase will require multiple execution rounds. For example, assume that we have a job that consists of 10 map tasks and 3 available slots and the average task execution time is 5 s. The map phase will require 4 execution rounds. Each round will require approximately 5 s as the task execute concurrently on each round therefore the lower bound of the map execution time will be 20 s. Contrary to existing works such as [35] that assume a fixed average execution time of the tasks, we model t_{avg} as a function that depends on the number of tasks. Using more tasks decreases the amount of data that will be processed by each task so their average execution time will be smaller (as you observe in Fig. 5). Specifically for the reduce tasks, their impact is more evident when the jobs' intermediate data are skewed [41]. Because the jobs in our framework are repetitive, we apply extrapolation to approximate the $t_{avg}(\text{mt}_j)$ function. The upper limit in the execution time is computed with the following equation:

$$\text{mtime}_j^{\text{up}} = \lceil \frac{\text{mt}_j - 1}{\text{ms}_j} \rceil \times t_{avg}(\text{mt}_j) + t_{\max}(\text{mt}_j), \forall j \in \text{Jobs} \tag{6}$$

where $t_{\max}(\text{mt}_j)$ is the maximum execution time observed in the map tasks. The limits for the other two phases (i.e., shuffle and reduce) are computed in a similar way using the appropriate task/slots parameters and their mean and

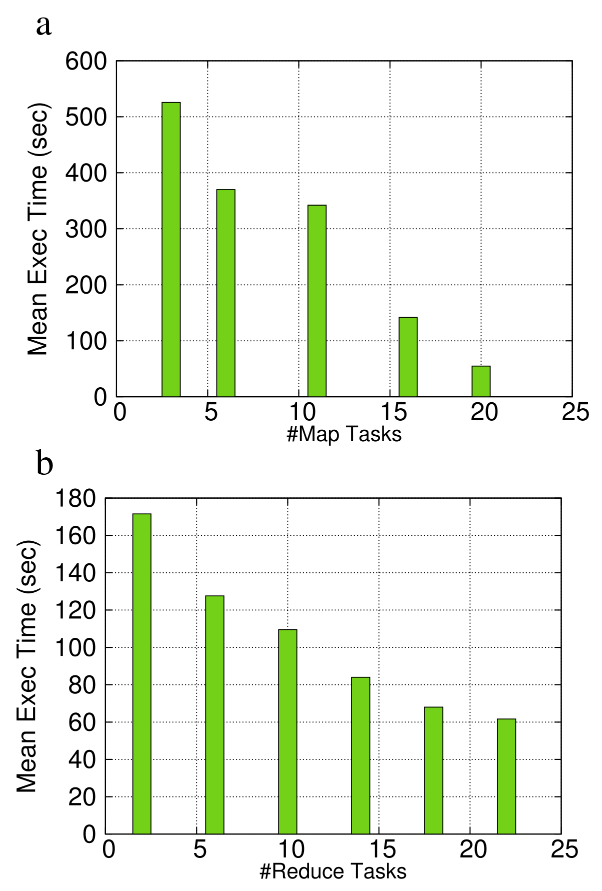


Fig. 5 Number of tasks impact on mean execution time of a Twitter job that processes 1.2 GB of input data. **a** Number of map tasks impact on task's mean execution time. **b** Number of reduce tasks impact on task's mean execution time

max execution times. Thus, we compute the upper and lower bound limits for the job's total execution time with the following two formulas:

$$\text{jtime}_j^{\text{low}} = \text{mtime}_j^{\text{low}} + \text{stime}_j^{\text{low}} + \text{rtime}_j^{\text{low}}, \forall j \in \text{Jobs} \tag{7}$$

$$\text{jtime}_j^{\text{up}} = \text{mtime}_j^{\text{up}} + \text{stime}_j^{\text{up}} + \text{rtime}_j^{\text{up}}, \forall j \in \text{Jobs} \tag{8}$$

Finally, the job's estimated end-to-end execution time as it was pointed out in [35] is given as the average of the two limits:

$$\text{jtime}_j = \lceil (\text{jtime}_j^{\text{up}} + \text{jtime}_j^{\text{low}})/2 \rceil, \forall j \in \text{Jobs} \tag{9}$$

With the above estimation, we compute the workload's end-to-end execution time as follows:

$$\text{Makespan} = \max_{j \in \text{Jobs}} \{\text{jtime}_j\} \tag{10}$$

Thus, the workload's end-to-end execution time depends on the slowest running job. In the case that no enough resources are available, the workload's jobs are split into *waves* of execution and the jobs that comprise a wave are executed in the available VMs. The workload's makespan is computed as the sum of Eq. 10 for the different execution waves. Before splitting the workload's jobs into waves, we sort them based on their execution time in ascending order to make sure that the jobs with the smaller execution time requirements will be scheduled first. Our goal is to satisfy the user's performance requirements; therefore, the workload's makespan should not exceed the user specified Deadline. So the following constraint should be satisfied when we determine the map/reduce slots to use per job:

$$\text{Makespan} \leq \text{Deadline} \tag{11}$$

4.2.2 *Budget computation*

The second metric we consider in our multi-objective problem is the spending budget. To compute this metric, we first calculate the per job monetary cost. We applied a pricing model similar to the one used by popular cloud providers like Amazon [16]. For each $j \in \text{Jobs}$, given a slots' allocation $(\text{ms}_j, \text{rs}_j)$, we compute the required VMs via Eq. 1 and then the budget can be calculated using the following equation:

$$\text{budget}_j = \text{VMs}_j \times \text{cost} \times \lceil \frac{\text{jtime}_j}{3600} \rceil + \text{ioCost}_j, \forall j \in \text{Jobs} \tag{12}$$

The per-job spending budget depends on the number of reserved VMs, the per VM cost as it was decided by the provider and the required execution time in hours. So similar to Amazon's EC2, the VM usage is charged based on the amount of time the VM is reserved. Amazon charges users on a per hour basis (e.g. $0.539 per hour for the m2.2xlarge instance [16]), so we divided the jtime_j metric that depicts the execution time in seconds with 3600 s. Amazon provides different VM types that can be reserved by the users and the pricing is adjusted in accordance to the VMs' processing capabilities (i.e., high-performance VMs have increased per-hour cost). In this work, we consider only *homogeneous* VMs but as future work, we plan to extend our approach to heterogeneous environments. Furthermore, the required budget for the I/O operations performed by the job needs to be added as extra cost. Our budget computation differs from other approaches like [18, 42] in the fact that we consider the impact of the amount of I/Os in the observed monetary cost, as there is a growing trend from cloud providers to charge users based on this metric [19]. The I/O cost is not negligible as the user pays approximately $0.05 per 1 million I/O requests. Assuming that the user's jobs process large datasets, there will be many I/Os which will lead to a larger cost. We compute the I/O cost for a job as follows:

$$\begin{aligned}&\mathrm{ioCost}_j = \mathrm{mt}_j \times \mathrm{spills}_j(\mathrm{mt}_j) \times \mathrm{spillSize}_j(\mathrm{mt}_j)\\ &\times \mathrm{IOCost}, \forall j \in \mathrm{Jobs}\end{aligned} \tag{13}$$

We consider the cost of the performed I/O operations when the intermediate data generated by the map tasks do not fit in their memory buffer and therefore are written in local files. The observed cost depends on the average number of these spill files per map task (i.e., $\mathrm{spills}_j(\mathrm{mt}_j)$), the average size of these files (i.e., $\mathrm{spillSize}_j(\mathrm{mt}_j)$), the per I/O operation cost as it was decided by the provider (i.e., IOCost), and the number of map tasks (i.e., mt_j). We explain in more details how the spill files are created and how we compute their sizes in Section 4.4. Currently, in our framework, we focus on the I/O operations performed by the map tasks due to the fact that map tasks usually process smaller sized input data (e.g., in the orders of *MB*). In contrast, reduce tasks process significantly larger input data as they are fewer than the map tasks so more data have to be spilled in the local files [25]. Therefore, adjusting the reduce tasks' buffer size will provide limited benefit to the observed ioCost_j. The total budget that will be spent by the user is computed as follows:

$$\mathrm{Budget} = \sum_{j\in\mathrm{Jobs}} \mathrm{budget}_j \tag{14}$$

The total budget for executing the workload is computed as the sum of the budget spent for all the user's jobs. The required budget needs to satisfy the following constraint based on the user's budget range:

$$\mathrm{Budget}_{\mathrm{Min}} \leq \mathrm{Budget} \leq \mathrm{Budget}_{\mathrm{Max}} \tag{15}$$

Allocating more slots than the assigned tasks is inefficient, as the extra slots will remain idle during the job's execution, leading to an unnecessary increase in the monetary cost. For this reason we bound the map/reduce slots with the following two constraints:

$$\mathrm{ms}_j \leq \mathrm{mt}_j, \forall j \in \mathrm{Jobs} \tag{16}$$

$$\mathrm{rs}_j \leq \mathrm{rt}_j, \forall j \in \mathrm{Jobs} \tag{17}$$

Based on the allocations of map and reduce slots (i.e., ms_j and $\mathrm{rs}_j, \forall j \in$ Jobs), we end up with different *Budget* and *Makespan* values. In Fig. 6, we illustrate with a simple example using a single job, how the different allocations of map/reduce slots can affect its execution time and corresponding budget. In this experiment, we used a typical Yahoo! job as described in [28] (for more details about the experiment see Section 6). Our problem is more complex as we have to consider multiple concurrently running jobs, so the search space is larger than the one displayed in Fig. 6. However, you can observe that there is a trade-off between the job's execution time and the required budget: as when we increase the number of slots, we decrease the job's execution time but this can lead to an increase in the spending budget; on the other hand, if the user utilizes fewer resources, the spending budget may decrease but at the same time the execution time increases. There are multiple slots' allocations with respect to the achievable Budget and Makespan values that are optimal. Therefore, our multi-objective optimization problem can be formally described as follows:

Problem Definition. Determine all $(\mathrm{ms}_j, \mathrm{rs}_j), \forall j \in$ Jobs allocations such that:

$$\begin{aligned}\textbf{minimize}:\ & \mathrm{Makespan} = \max_{j\in\mathrm{Jobs}}\{\mathrm{time}_j\}\\ & \mathrm{Budget} = \sum_{j\in\mathrm{Jobs}} \mathrm{budget}_j\\ \textbf{subject to}:\ & \mathrm{Makespan} \leq \mathrm{Deadline}\\ & \mathrm{Budget}_{\mathrm{Min}} \leq \mathrm{Budget} \leq \mathrm{Budget}_{\mathrm{Max}}\\ & \mathrm{ms}_j \leq \mathrm{mt}_j, \forall j \in \mathrm{Jobs}\\ & \mathrm{rs}_j \leq \mathrm{rt}_j, \forall j \in \mathrm{Jobs}\end{aligned}$$

4.3 Pareto-search algorithm

The problem we solve in this work is how to appropriately allocate map/reduce slots to a user's jobs in order to minimize the workload's makespan and at the same time minimize the user's spending budget. As we explained in Section 4.2, this is a multi-objective optimization problem due to the trade-off between the workload's makespan and the required budget. One of the most common ways of detecting appropriate solutions in such problems is by constructing the Pareto frontier [43]. Pareto-based search techniques have been commonly applied to similar problems [44] where the goal is to detect multiple optimal solutions and present them to the user in order to decide the solution that satisfies her requirements.

In our case, we consider as solutions the different slots' allocations (i.e., the allocation of map/reduce slots per job for the user's workload) that can be applied. In order to detect the optimal solutions in the examining search space, we must define the notion of dominance [44]. Given two slots' allocations SA_1 and SA_2, allocation SA_2 *dominates* SA_1 ($SA_2 \succeq SA_1$) if one of the two cases occurs: (1) the budget for SA_2 is less than equal to the one required for SA_1 and the makespan of SA_2 is less than SA_1's or (2) SA_2 requires strictly less budget than SA_1 and also the makespan of SA_2 is less than or equal to the makespan of SA_1.

The set containing all the non-dominated allocations constitutes the Pareto frontier of the available solution

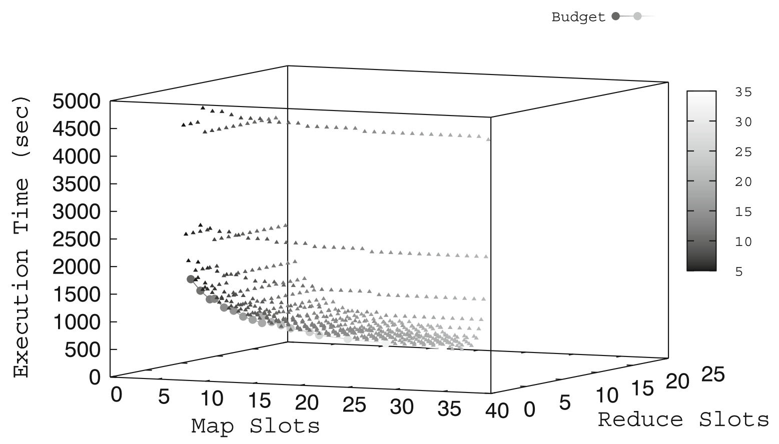


Fig. 6 Slots impact on the job's execution time and required budget

space as we illustrate in Fig. 7. Our aim is to construct this frontier and present it to the user so that she can decide the slots' allocation she is willing to use. The displayed Pareto points are optimal allocations with respect to both budget and workload's end-to-end execution time objectives. That is, there are no other slots' allocations that, for the given budget range and deadline, achieve smaller execution time for the workload. All the other possible allocations constitute the feasible region and are sub-optimal as you can observe in Fig. 7.

However, computing the Pareto frontier is a computationally costly process. For example, the simplest approach for finding the Pareto frontier is to enumerate all the possible combinations of jobs' and slots' allocations and keep the non-dominated ones. Each combination can be considered a slot allocation plan. To define it more formally, let $\mathrm{Jobs}^{'}$ be a set of jobs, where for each job $j \in \mathrm{Jobs}^{'}$ we have decided its ms_j and rs_j parameters. The $\mathrm{ms}_j, \mathrm{rs}_j$ parameters that will be allocated for a job will determine the number of VMs used (i.e., VMs_j parameter in Section 3). Based on the chosen ms_j and rs_j parameters for the jobs that comprise $\mathrm{Jobs}^{'}$ we will end up with different Budget and Makespan. Furthermore, the total number of such $\mathrm{Jobs}^{'}$ combinations that need to be examined by this simple exhaustive search approach will be $N^{|K|}$ where N is the maximum number of VMs that can be reserved by a job and K is the number of jobs that comprise the workload. Therefore, this exhaustive search algorithm has exponential complexity (i.e., $O(N^K)$) as it generates all these $N^{|K|}$ slot allocation plans and then keeps those that are non-dominated with respect to the Budget and Makespan metrics. We argue that the computation overhead of this algorithm is very high and so it cannot be used when we consider workloads that consist of a large number of jobs.

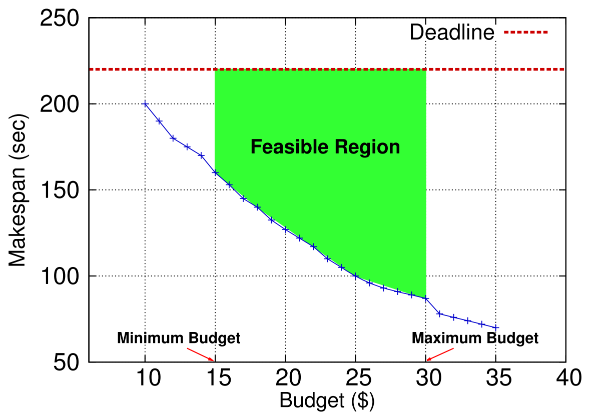


Fig. 7 The Pareto frontier and the feasible region of our multi-objective optimization problem

We propose a novel approach that detects near-optimal slots' allocations in an efficient and fast way without requiring to enumerate all the possible plans. Our greedy algorithm approximates the Pareto-optimal frontier by allocating new resources to the job that affects the most the workload's makespan. Our proposed technique does not enumerate all the possible slots' allocations but keeps searching for new slots' allocations as long as it has not exceeded the given budget range. As we illustrate in Algorithm 1, we traverse the frontier starting from the plan that uses minimum resources and at each step we try to improve the observed makespan and monetary cost. So initially, we reserve one map and one reduce slot for each job ($\mathrm{ms}_j = 1$ and $\mathrm{rs}_j = 1, \forall j \in \mathrm{Jobs}$). We keep increasing the jobs' slots trying to minimize the workload's makespan until we have not exceeded the available user's budget (i.e., $\mathrm{Budget}_{\mathrm{Max}}$ metric). In each step of the search, we

Algorithm 1 Pareto-search Algorithm

```
1: Input: $Jobs$: the user's workload, $Budget_{Min}$: the minimum user's budget, $Budget_{Max}$: the maximum user's budget, $Deadline$: the deadline in the workload's makespan.
2: Output: $Plans$: the Pareto-efficient slots' allocations.
3: $plan \leftarrow$ Get plan that uses $ms_j = 1, rs_j = 1, \forall j \in Jobs$
4: $Plans \leftarrow \{\}$
5: if $(plan.Budget \geq Budget_{Min} \wedge plan.Budget \leq Budget_{Max} \wedge plan.Makespan \leq Deadline)$ then
6:     $Plans \leftarrow Plans \cup \{plan\}$
7: while TRUE do
8:     $j' \leftarrow$ Find $j \in Jobs$ that affects the most $Makespan$
9:     $plan \leftarrow planDetection(j', Jobs)$
10:    if $(plan.Budget < Budget_{Min} \vee plan.Makespan > Deadline)$ then
11:        continue
12:    if $(plan.Budget > Budget_{Max} \wedge |Plans| > 0)$ then
13:        break
14:    $addPlan \leftarrow TRUE$
15:    for all $prevPlan \in Plans$ do
16:        if $(plan.Budget > prevPlan.Budget \wedge plan.Makespan < prevPlan.Makespan)$ then
17:            continue
18:        if $(plan.Budget \leq prevPlan.Budget \wedge plan.Makespan < prevPlan.Makespan)$ then
19:            $Plans \leftarrow Plans - \{prevPlan\}$
20:            continue
21:        $addPlan = FALSE$
22:        break
23:    if $addPlan == TRUE$ then
24:        $Plans \leftarrow Plans \cup \{plan\}$
25: return $Plans$
```

Algorithm 2 Slots' Allocation Algorithm

```
1: Input: $Jobs$: the user's workload, $j'$: the job that will receive the extra resources.
2: Output: $plan$: the detected slots' allocation plan.
3: $jtime'_{j'} \leftarrow$ Compute execution time of $j'$ using $ms_{j'} + 1$.
4: $jtime''_{j'} \leftarrow$ Compute execution time of $j'$ using $rs_{j'} + 1$.
5: if $(jtime'_{j'} < jtime''_{j'})$ then
6:     $ms_{j'} \leftarrow ms_{j'} + 1$
7: else
8:     $rs_{j'} \leftarrow rs_{j'} + 1$
9: $Makespan \leftarrow$ Compute makespan using $ms_{j'}$ and $rs_{j'}$
10: $Budget \leftarrow$ Compute budget using $ms_{j'}$ and $rs_{j'}$
11: $plan \leftarrow createPlan(Makespan, Budget)$
12: return $plan$
```

detect the job (j' in Algorithm 1) that affects the most the observed makespan and reserve extra resources for it (Line 9 in Algorithm 1).

We considered two policies for allocating slots for the job that affects the makespan:

Algorithm 3 VMs' Allocation Algorithm

```
1: Input: $Jobs$: the user's workload, $j'$: the job that will receive the extra resources, $msPerVM$: the number of map slots per VM, $rsPerVM$ the number of reduce slots per VM.
2: Output: $plan$: the detected slots' allocation plan.
3: $ms'_{j'} \leftarrow ms_{j'} + msPerVM$
4: $rs'_{j'} \leftarrow rs_{j'} + rsPerVM$
5: $Makespan \leftarrow$ Compute makespan using $ms'_{j'}$ and $rs'_{j'}$
6: $Budget \leftarrow$ Compute budget using $ms'_{j'}$ and $rs'_{j'}$
7: $plan \leftarrow createPlan(Makespan, Budget)$
8: return $plan$
```

Algorithm 4 Task Adjustment Algorithm

```
1: Input: $j$: the job for which task must be adjusted, $sl$: the number of slot's to use, $isMs$: the boolean variable that determines if the slots correspond to map or reduce slots.
2: Output: $tasks$: the number of tasks to use.
3: $tasks \leftarrow sl$
4: $jtime_j \leftarrow$ Compute execution time using Eq. 9
5: while TRUE do
6:     $tasks' \leftarrow tasks + 1$
7:     $jtime'_j \leftarrow$ Compute execution time using Eq. 9
8:     if $(jtime'_j \geq jtime_j)$ then
9:         break
10:    $jtime_j \leftarrow jtime'_j$
11:    $tasks \leftarrow tasks'$
12:    if $(isMs)$ then
13:        Apply buffer tuning for job $j$ using Algorithm 5
14: return $tasks$
```

1. *Slots' Allocation* (i.e., Algorithm 2): In this policy, we increase the amount of resources allocated to each job by one slot at a time. More specifically, we examine the impact on the job's execution time first if we increase the map slots by one, and then if we increase the number of reduce slots. Based on the type (map or reduce) of the slot that improves the execution time the most, we make the corresponding change.
2. *VMs' Allocation* (i.e., Algorithm 3). The second policy is more lavish as you can see in Algorithm 3, as it concurrently increases the map and reduce slots reserved by the job, allocating a new VM for the jobs' tasks. The *VMs' Allocation* policy offers a fast approach for distributing the available resources to the submitted jobs leading to faster search times. On the other hand, the Slots' Allocation policy reserves one slot at a time; although this may increase the search time of our approach, it can lead to better allocations as the two phases (i.e., map and reduce) are examined separately.

The two policies use a different atomic unit for detecting near-optimal plans (i.e., the Slots' allocation uses a single slot while the VMs' Allocations reserves a whole VM) so the two algorithms will traverse the search space differently. The Slots' Allocation policy will make more fine-grained decisions as it examines one slot at a time while the VMs' Allocation policy will assign a new VM to the job that has the highest execution time. Even if the two techniques end up detecting solutions that require the same number of VMs, the slots that have been allocated to the jobs will be different as the Slots' Allocation will distribute more efficiently the slots between the workload's jobs as it examines them one at a time.

When the extra slots for the job have been reserved, we create a new allocation plan that consists of new $ms_{j'}$ and $rs_{j'}$ allocations. The next step is to examine if this new plan should be added in the frontier. First, we check if the constraints are satisfied (lines 10–13 in Algorithm 1) and then we compare this new plan with the already detected plans to examine whether it is dominated or not (lines 15–22 in Algorithm 1). If the new plan dominates a previously detected plan in both metrics, we remove the previous plan from the detected solutions set (lines 18–20 in Algorithm 1). In case that the plan is non-dominated, we add it in the solutions set (lines 23–24 in Algorithm 1) that will be displayed to the user when the search has finished.

4.3.1 Task adjustment

We extend our Pareto-based scheduler to automatically adjust the map/reduce tasks while we allocate slots to jobs in order to minimize further the workload's end-to-end execution time. Each time we increase a job's map or reduce slot (or both of them if the VMs' Allocation policy is applied), we also adjust the corresponding tasks (i.e., map/reduce) that will be utilized. The goal is to minimize further the per job execution times (i.e., Eq. 9) and thus reduce the workload's makespan. For deciding the appropriate number of tasks, we apply a Hill Climbing search algorithm (i.e., Algorithm 4) that tries to minimize the job's execution time by gradually increasing the number of tasks. The algorithm initially sets the number of tasks equal to the number of slots (i.e., $\mathrm{mt}_j = \mathrm{ms}_j$ and $\mathrm{rt}_j = \mathrm{rs}_j$), and then keeps increasing the tasks by one as long as the job's execution time decreases. The rationale behind our approach comes from Eq. 5, where the average execution time of map/reduce tasks depends on the number of tasks. So an increase in the number of tasks can decrease their average execution time. However, when we increase the number of tasks, we also increase the number of execution rounds and this may penalize the job's execution time. So we increase the number of tasks only when the extra execution round does not deteriorate the task's execution time (i.e., the improvement in the phase's execution time due to the decrease of the tasks' average execution time is negligible). In most cases, the cost of the extra round will be larger so the algorithm will simply set the number of tasks equal to the number of the allocated slots. We argue that it is important to adjust the number of map/reduce task parameters as usually the users submit jobs with higher number of tasks than the slots that will be allocated to them. Therefore, Algorithm 4 will be helpful in such cases as it will decrease the number of execution rounds as it starts its search from the solution that performs a single execution round.

Finally, special care must be given when the number of map tasks for a job changes as it can lead to different I/O monetary cost (see Eq. 13 in Section 4.2) than the initially computed (i.e., via the algorithm proposed in Section 4.4). For this reason, we re-apply the algorithm described in Section 4.4 (i.e., Algorithm 1) to re-configure the map buffer parameters (i.e., $\mathrm{mBuffer}_j$, $\mathrm{metaPercent}_j$ and bThr_j) whenever we change the number of map tasks.

4.4 Buffer tuning

Our work puts emphasis on the monetary cost due to the I/O operations performed by a job's map tasks (as can be seen in Eq. 13 in Section 4.2). Once we model the impact of I/O operations on the user's budget, we minimize it by adjusting appropriately the parameters that affect the size of the map tasks' buffers. As we described in Section 2.1, the map buffer consists of two parts, the first part is responsible for storing metadata, specifically the record bounds in the blocks, while the second part keeps the actual data. The buffer is split and its data are stored in local files in two cases, either when the metadata exceed their allocated space or when the actual data exceed their own limit. The maximum size of actual data that can be kept in the buffer is defined via the following equation:

$$\mathrm{maxSize}_j = (1 - \mathrm{metaPercent}_j) \times \mathrm{mBuffer}_j \times \mathrm{bThr}_j \tag{18}$$

Similarly, the maximum metadata size in bytes that can be kept in the buffer is defined with the following formula:

$$\mathrm{maxMetaSize}_j = \mathrm{metaPercent}_j \times \mathrm{mBuffer}_j \times \mathrm{bThr}_j \tag{19}$$

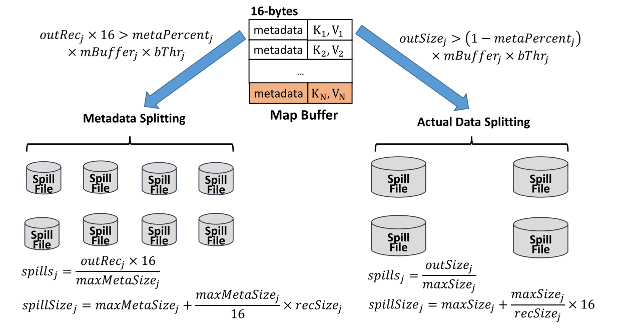
As we mentioned in Section 2.1, the same buffer threshold is used for determining when the available space (either for actual data or metadata) has been exceeded and thus data must be split in the local file system. The size of the metadata is small (i.e., 16 bytes per record); however, the percentage of the metadata that can be kept in the buffer is important as it affects the number of local files that will be created. Given the maximum size that can be kept on the buffer before a spill file is created, we can estimate the number of spills using the following equation:

$$\text{spills}_j(\text{mt}_j) = \max\left\{\left\lceil\frac{\text{outSize}_j(\text{mt}_j)}{\text{maxSize}_j}\right\rceil, \left\lceil\frac{\text{outRec}_j(\text{mt}_j)\times 16}{\text{maxMetaSize}_j}\right\rceil\right\} \tag{20}$$

So the number of spills depends heavily on the type of job we examine. For example, if the job has large-sized output records (i.e., large $\text{outSize}_j(\text{mt}_j)$) then the number of spills will be adjusted due to the actual data size. On the other hand, if map tasks generate many small-sized intermediate records (i.e., large $\text{outRec}_j(\text{mt}_j)$), the metadata part of the buffer will affect the number of spill files. In general, as you can see in Fig. 8, based on the splitting reason, we will end up either with a small number of large-sized files (i.e., spill occurred due to the actual data) or with a large number of small-sized files (i.e., data have been spilled due to the metadata). Depending on the reason of splitting, we end up with different spill file sizes that are computed using the following formulas:

$$\text{spillSize}_j(\text{mt}_j) = \begin{cases} \text{maxSize}_j + \frac{\text{maxSize}_j}{\text{recSize}_j(\text{mt}_j)} \times 16 & \text{(21a)} \\ \text{maxMetaSize}_j + \frac{\text{maxMetaSize}_j}{16} \times \text{recSize}_j(\text{mt}_j) & \text{(21b)} \end{cases}$$

The first case occurs when data are spilled due to the actual buffer size, while the second when the metadata space exceeds its limit. Given the spill size and the number of spill files, we can compute the cost of the job with respect to I/O operations via Formula 13. The spill size and the number of spills are computed using the $\text{recSize}_j(\text{mt}_j)$, $\text{outRec}_j(\text{mt}_j)$, and $\text{outSize}_j(\text{mt}_j)$ parameters. These parameters depend on the number of map tasks, so in order to compute them, we use historical data from previous execution runs. In general, we expect that the values of these metrics will decrease when the number of map tasks increases as the input data will be splitted into smaller parts. In order to minimize the impact of this cost, we must configure three different parameters, mBuffer_j, metaPercent_j, and bThr_j $\forall j \in$ Jobs. We solve the problem for each job that participates in the workload


16-bytes
metadata | K_1, V_1
metadata | K_2, V_2
...
metadata | K_N, V_N
Map Buffer
$outRec_j \times 16 > metaPercent_j \times mBuffer_j \times bThr_j$
$outSize_j > (1 - metaPercent_j) \times mBuffer_j \times bThr_j$
Metadata Splitting
Actual Data Splitting
Spill File
$spills_j = \frac{outRec_j \times 16}{maxMetaSize_j}$
$spillSize_j = maxMetaSize_j + \frac{maxMetaSize_j}{16} \times recSize_j$
$spills_j = \frac{outSize_j}{maxSize_j}$
$spillSize_j = maxSize_j + \frac{maxSize_j}{recSize_j} \times 16$
Fig. 8 Map buffer's splitting conditions

Algorithm 5 Buffer Tuning Algorithm

```
1:  Input: $j$: the job for which the buffer parameters must be adjusted, $iterations$: the number of iterations to perform.
2:  Output: $mBuffer_j$: the map buffer size, $metaPercent_j$: the metadata percentage, $bThr_j$: the map buffer's threshold.
3:  $itCount \leftarrow 0$
4:  $ioCost \leftarrow Double.MAX\_VALUE$
5:  while $itCount < iterations$ do
6:      $itCount \leftarrow itCount + 1$
7:      $mBuffer', metaPercent', bThr' \leftarrow$ Sample the parameters using three uniform distributions.
8:      $ioCost' \leftarrow$ Compute I/O cost using Eq. 13.
9:      if ($ioCost' > ioCost$) then
10:         continue
11:     if ($ioCost' == ioCost \wedge mBuffer' > mBuffer_j$) then
12:         continue
13:     $ioCost \leftarrow ioCost'$
14:     $mBuffer_j \leftarrow mBuffer'$
15:     $metaPercent_j \leftarrow metaPercent'$
16:     $bThr_j \leftarrow bThr'$
17: return $mBuffer_j, metaPercent_j, bThr_j$
```

by applying a greedy Random Search [45] algorithm on the parameter spaces of these variables and keeping the combination that minimizes Eq. 13. In Algorithm 5, we provide the pseudocode for tuning these parameters. The algorithm runs for a fixed number of iterations and uses three uniform distributions for sampling the three parameters of interest. We decided to use uniform distributions in order to be able to sample uniformly the search space for the three parameters and minimize the bias in our search [45]. For each parameter, a different uniform distribution is used and the upper and lower limits are the ones described in Section 3. Regarding the mBuffer_j, the upper bound depends on the memory that has been allocated for the map tasks. As we pointed out in Fig. 1, using all the available memory may not lead to a decrease in the I/O cost. For this reason, we argue that mBuffer_j should be also sampled and we have to keep its smallest value that minimizes the ioCost_j metric. The remaining free memory could be allocated for the tasks' processing (e.g., map tasks may use in-memory data structures). So if Algorithm 5 finds two solutions that lead to the same ioCost_j, we keep the one that uses the least mBuffer_j parameter. In each iteration, we compute the I/O monetary cost for the randomly sampled parameters (i.e., lines 7–8 in Algorithm 5) and update the configuration parameters that will be used by the job, only if we are able to reduce the currently minimum computed I/O cost (i.e., lines 9–14 in Algorithm 5).

5 Implementation

We have implemented our Pareto-based scheduler as a daemon running on the master node of a Hadoop

1.2.1 [2] cluster. The scheduler has been implemented in Java 1.7. In Fig. 9, we show at high level how our approach works. The user submits a MapReduce workload along with the budget range she is willing to spend to Hadoop's master node. More specifically, the user submits a JSON file where she specifies the necessary information for the Hadoop jobs that comprise the workload and also her budget and performance requirements. For each job, the user must also provide the classes that implement the map and reduce [46] Java interfaces. Then the Jobs Submitter component is responsible to forward this information to the daemon running in the Hadoop cluster. Our framework is able to support all types of Hadoop jobs including those offered by the Mahout machine learning library [26], without requiring any modification in their implementation. Our proposed scheduler replaces the default Hadoop scheduler (i.e., FIFO) and applications (e.g., Mahout [26], PUMA [30]) can execute unmodified on top of the Hadoop framework and take advantage of the scheduling optimizations provided by our Pareto-based scheduler to speed up their execution time and minimize their monetary cost.

The daemon invokes the search algorithm described in Section 4.3. The Pareto-Search Algorithm component initially interacts with the Parameters Tuning component in order to adjust the map-buffer related parameters to minimize the I/O cost (Section 4.4). Also, the Pareto-Search Algorithm component uses historical data of the already executed jobs in order to create the job's execution time estimation model we described in Section 4.2. We gather the necessary job's historical data using the Apache's Rumen [47] log analyzer when the all job's tasks have finished. Statistics are kept in a MySQL database (i.e., Jobs' statistics in Fig. 9). After the valid slots' allocations have been detected by the Pareto-Search Algorithm component, the daemon informs the user about the possible allocations and the latter responds with her chosen allocation based on the budget she is willing to spend.

When the scheduler has received the user's choice about the allocation that will be applied, the Queues Manager component is responsible to create a separate scheduling queue for each job. Queues Manager configures each queue to have the same number of map/reduce slots for the corresponding job as the number used in the chosen slots' allocation plan. We follow this approach to limit the number of slots used per job to the chosen number. For the actual execution of the jobs in the cluster, we use Hadoop's FAIR [2] scheduler in order to force the concurrent execution of jobs in the available resources (i.e., by setting the queue sizes in *fair_scheduler.xml* file [48]). Our approach cannot work with the default Hadoop FIFO scheduler as FIFO assumes that all resources are available to the submitted jobs, so a single job may occupy all the slots in the cluster. Our framework can be easily extended to support other Hadoop schedulers such as the Capacity Scheduler [2].

After the Queues Manager has configured the slots per queue, it informs the Jobs Submitter that the required slots for the execution of the workload have been reserved. The latter then uses the Hadoop API (i.e., *Job* class [46])

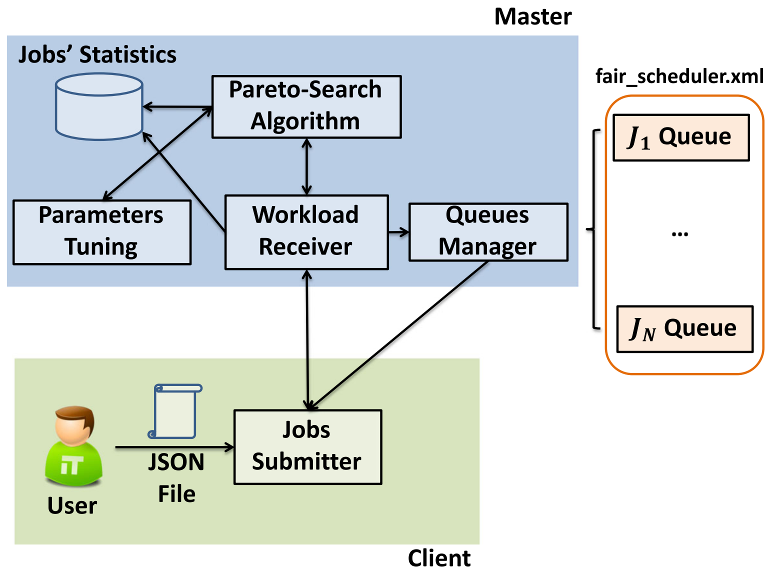


Fig. 9 Implementation details

to actually submit the jobs for execution in the cluster. For each job, a separate process is used for the submission so all the jobs that comprise the workload will be assigned simultaneously to the cluster. When all the jobs have finished, the Jobs Submitter informs the scheduler daemon so that the latter can retrieve the necessary statistics used for building the jobs' execution time models described in Section 4.2.

6 Evaluation

6.1 Setup

We have conducted a detailed experimental evaluation with both (a) simulations and (b) empirical workloads in our local cluster. Our cluster consisted of 8 VMs (Ubuntu 12.04), allocated with two CPU processors and 3096 MB RAM. One of the VMs was used as the Master node and the others were Worker nodes. Each Worker node was equipped with two map and two reduce slots. We demonstrate the performance of our approach performing both simulations in larger settings and also empirically running experiments on our local cluster, to illustrate the benefits of our approach. In the experiments we performed with simulation workloads, we assumed that we had no limit on the number of available VMs (i.e., the available VMs were not 8 as in our local cluster) because we wanted to examine the applicability of our approach when we have significantly larger workloads that execute on public cloud infrastructures like Amazon's EC2. So in these experiments, the resource allocations that can be detected were only constrained by the minimum and maximum budget. In our experiments, we defined the cost of reserving a VM to be \$1 per hour which is a typical value for high-performance nodes [16] and also is a value that makes the cost savings more pronounced. The I/O monetary cost was set to \$0.05 per 1 million I/O requests, a commonly used charging policy [19].

For the synthetic workloads, we did not have information about the I/Os performed by jobs so we did not use the buffer tuning optimization in the experiments we performed with the synthetic workloads. It should be clear that in all the experiments that depict the detected Pareto frontiers (i.e., Figs. 11, 14, 15, 16 and 17) we also tune the buffer parameters (i.e., only in the empirical workloads) and the number of tasks parameters, unless it is stated otherwise. Finally, we conducted detailed experiments to illustrate the performance of our tuning adjustment techniques (i.e., Figs. 17, 18, and 19) and the improvements they offer in terms of makespan (i.e., the task adjustment) and budget (i.e., the buffer adjustment technique).

6.2 Workloads

Our simulation workloads were based on the Hadoop workload analysis performed in the Yahoo! [28] and Taobao [29] Hadoop clusters. According to these works, in the Taobao workload, the map tasks execution times can be modelled by a log normal distribution with mean equal to 1.95 and standard deviation 1.67. Similarly, a log normal distribution can be used for the reduce tasks execution time with 3.52 mean and 1.56 standard deviation. The deadline when 10 Taobao jobs were submitted in the cluster was set to 3000 s. Regarding the Yahoo! workload, we simulated the task execution similarly to [23], so tasks were modelled via a normal distribution with 50 mean and 200 standard deviation for map tasks' execution times, while for reduce tasks, the mean was 100 and the standard deviation 300. For generating the number of tasks, we used a normal distribution with 154 mean and 558 standard deviation for the map tasks and 19 mean and 145 standard deviation for the reduce tasks. The budget range was set between \$10 and \$80. For the 10 Yahoo! jobs, we set the deadline to be equal to 4000 s. Furthermore, we examined the performance of our approach on our local cluster, running empirical workloads similar to the ones provided by the PUMA benchmark [30]. We used a subset of common jobs and generated different workloads (see Table 2 for more details). The user's budget for the real workloads was set between \$6 and \$20 as we are constrained by our computing resources (i.e., 8 VMs cluster).

6.3 Estimation error

First, we evaluated the accuracy of our prediction model when estimating the jobs' execution time. In this experiment, we considered the jobs in our empirical workloads. All jobs ran in our local 8 VMs cluster. We examined

Table 2 Empirical workloads

Workload	Jobs	No. of map tasks	No. of reduce tasks	Deadline (s)	Input size
Twitter friendship	3	20	10	1000	1.2 GB tweets[a]
Sort	3	80	10	2500	5 GB random data
Wordcount	3	47	10	1000	3.2 GB movie reviews
Twitter-Wordcount	1 Twitter and 2 Wordcount	20 Twitter and 47 Wordcount	10 Twitter and 10 Wordcount	1000	3.2 GB movie reviews and 1.2 GB tweets

[a]Extracted during the period of January 1, 2013 to April 30, 2013 using the Streaming API 2 of Twitter

how close the estimated execution time is to the actually observed one for different applications in order to prove that this prediction model can efficiently capture the required parameters for estimating the workload's makespan and budget. Furthermore, we display the lower and upper bounds of the execution as we described them in Section 4.2. We used 10 previous execution runs for gathering the data necessary for the prediction and then we estimated the execution time of a newly assigned job. As we illustrate in Fig. 10, the estimated execution time is very close to the actual, so applying this prediction model for estimating the jobs' execution time is a valid choice.

6.4 Simulation results

6.4.1 Allocation policies

The first experiment we conducted was to compare the two different policies (proposed in Section 4.3) for allocating resources to the job that affects the most the observed makespan. In Fig. 11, we present the detected frontiers when the VMs' and Slots' allocations policies are utilized both in the Yahoo! and the Taobao workloads. The Slots' Allocation policy outperforms the other approach as it makes a better decision in regards to where the slots will be assigned (i.e., for map or reduce tasks). More specifically, the Slots' Allocation policy can decide to assign more map than reduce slots if it observes a better performance in terms of the execution time. In contrast, the VMs' Allocation policy is more lavish as it assigns immediately a new VM instance and therefore it is not able to make more fine-grained decisions than the other approach which may use fewer VMs for achieving the exactly same Makespan and Budget. Also, it should be clear that throughout the search of the parameters' space, the two approaches make different decisions and thus may end up with different slots' allocations. It is possible that the two algorithms may utilize the same number of VMs

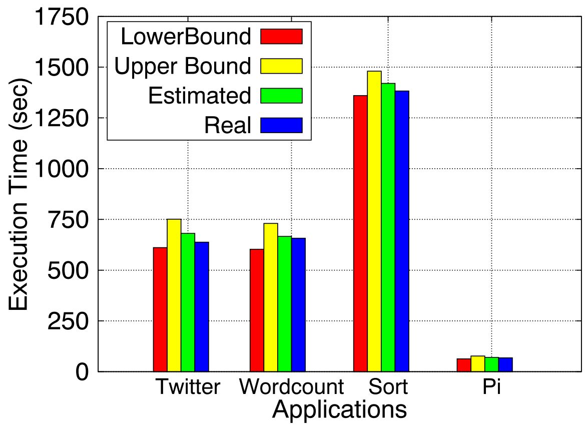


Fig. 10 Prediction model's accuracy

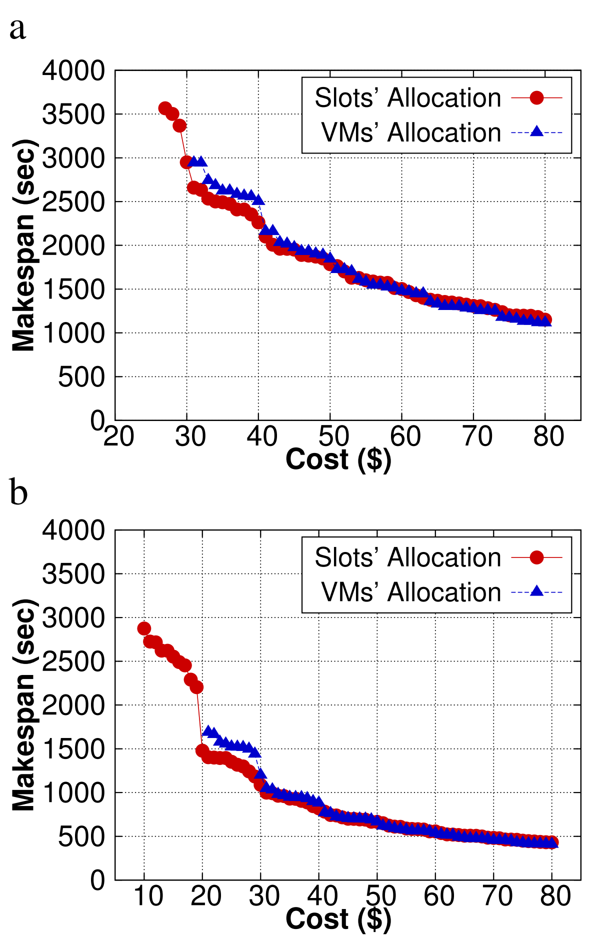


Fig. 11 Detected frontiers comparison using different allocation policies. **a** 10 Yahoo! jobs workload. **b** 10 Taobao jobs workload

but there will be a difference in terms of the slots used by the jobs so different values for the Makespan and Budget metrics will be observed. Furthermore, as you can observe in Fig. 12, the overhead of the Slots' Allocation in the execution time is negligible. Thus, in the rest of the experiments, we use the Slots' Allocation policy for allocating map/reduce slots.

6.4.2 Comparison with other search algorithms

We compared our Pareto frontier search algorithm with the following algorithms: (i) *NSGA-II* [49] which is a genetic-based multi-objective optimization algorithm, (ii) *GDE3* [50], an algorithm that extends the Pareto differential evolution method for global optimization, and (iii) *Starfish* [20] which uses the Recursive Random Search (RRS) [27] algorithm for detecting appropriate solutions. We applied RRS in our case by invoking it for each budget value in the given range, detecting the slots' allocations that minimize Eq. 10 from Section 4.2. Both genetic algorithms are well-known techniques for solving

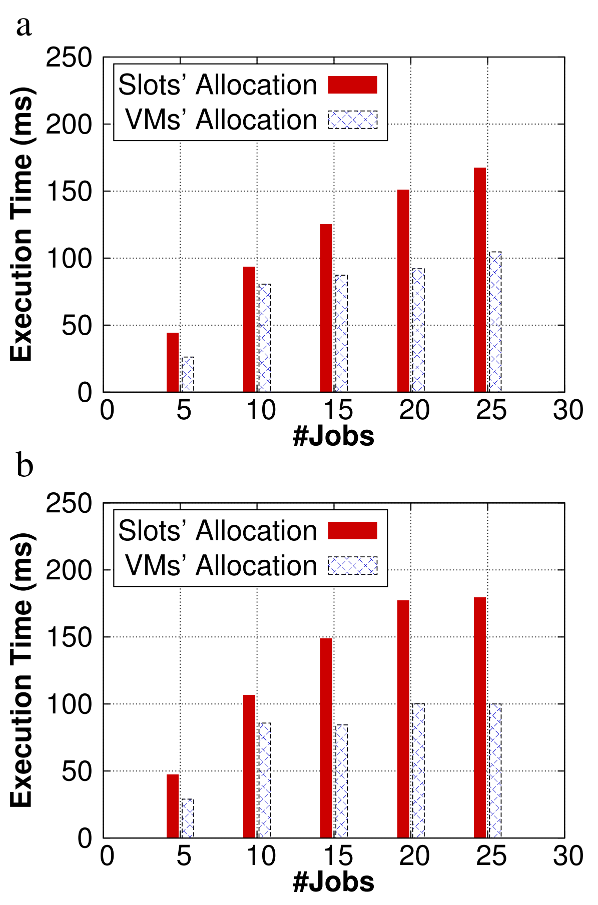


Fig. 12 Comparison of the execution times of the different allocation policies. **a** Yahoo! workload. **b** Taoabo workload

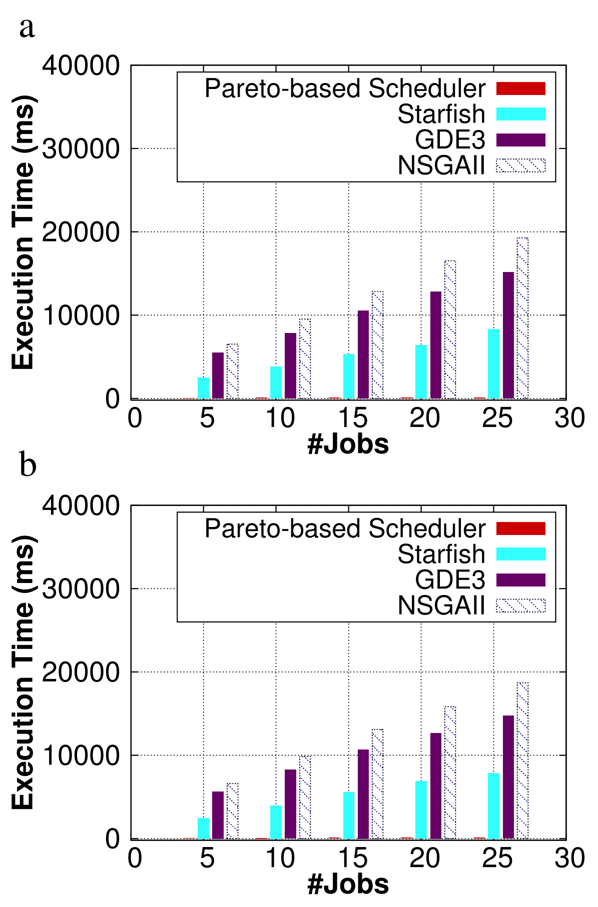


Fig. 13 Comparison of the execution times of the different search algorithms. **a** Yahoo! workload. **b** Taobao workload

multi-objective optimization problems and approximate the actual frontier by examining a sample of the different combinations.

In this set of experiments, we did not apply the buffer-tuning optimization so the default parameters were used and thus all approaches had the same I/O cost. We varied the number of jobs issued by the user to measure the execution time required for detecting the frontier. As Fig. 13 illustrates, our proposal requires the minimum execution time (i.e., less than 150 ms for 25 jobs) as it does not consider multiple combinations of the possible allocations. Furthermore, the detected frontiers in both workloads are very close to the frontiers detected by the two genetic algorithms as you can see in Fig. 14. Starfish is able to quickly detect valid allocations; however, as you can observe in Fig. 14, the detected solutions deviate significantly from ours in both workloads, as the randomly chosen allocations can be sub-optimal. Therefore, our proposal is able to find a good approximation of the actual frontiers and at the same time requires minimal execution time.

6.4.3 Comparison with Optimal

Finally, we examined the accuracy of our algorithm by comparing it with the Optimal algorithm that considers all the possible allocations of slots to jobs and then detects all the non-dominated allocations. More specifically, this algorithm enumerates all the possible slots' allocations for the jobs that comprise the workload and then keeps the allocations that are non-dominated. For example, in case that we want to schedule a workload of 2 jobs and each job consists only of map tasks that require at maximum 2 map slots then we have the following slots' allocation plans: $< 1,1 >$, $< 1,2 >$, $< 2,1 >$, $< 2,2 >$ to consider. As we pointed out in Section 4.3, this algorithm has exponential complexity so it can only be applied when the workload consists of a small number of jobs. In Fig. 15, we display the frontier found by our approach compared against the one found by the Optimal algorithm when 3 jobs are executing in the cluster. As you can observe there is a small difference between the two detected frontiers. However, the differences in the execution times are huge as our algorithm requires

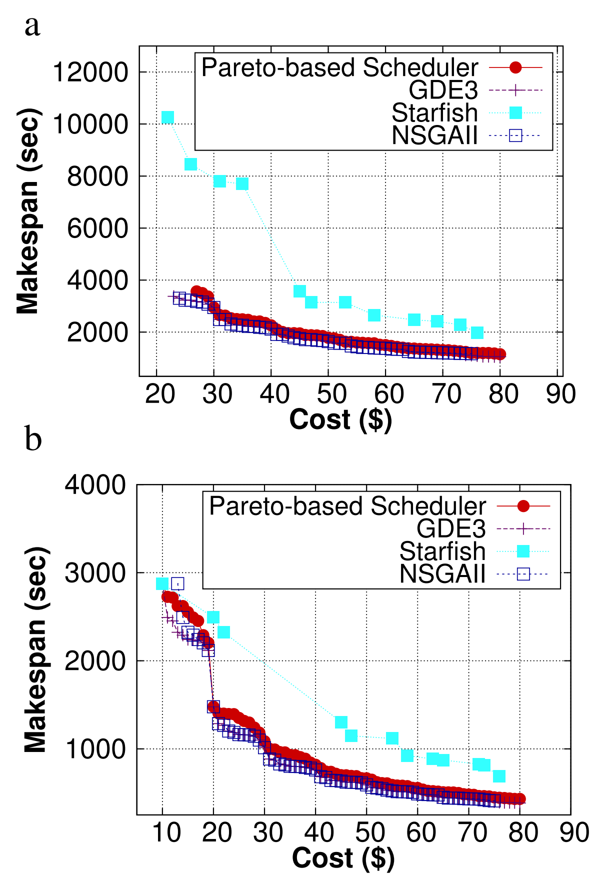


Fig. 14 Detected frontiers comparison using different search algorithms. **a** Ten Yahoo! jobs workload. **b** Ten Taobao jobs workload

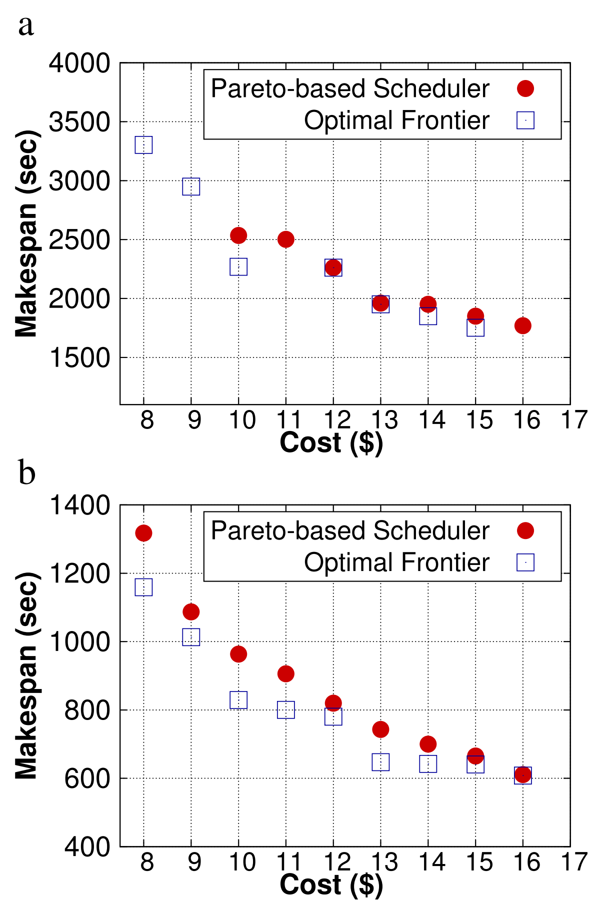


Fig. 15 Detected frontier comparison with the Optimal frontier. **a** Three Yahoo! jobs workload. **b** Three Taobao jobs workload

only 30 ms while the Optimal requires approximately 7.5 minutes.

6.5 Empirical results

6.5.1 Detected frontiers

In Fig. 16, we illustrate the detected frontiers for the different empirical workloads. As you can observe, our algorithm is able to find appropriate solutions for all the workloads. Furthermore, our scheduler utilizes more slots only if there is a benefit in the observed makespan. For example, in the Sort workload using more than $11 would lead to the same end-to-execution, so the corresponding points are not added in the displayed solutions.

6.5.2 Task adjustment

In regards to the automatic adjustment of the number of map/reduce tasks, in Fig. 17, we illustrate the benefits of our proposal. Jobs without the task optimization (i.e., the displayed frontiers in Fig. 16) used the number of tasks specified in Table 2. As you can observe in Fig. 17, the detected allocations require the same budget as in Fig. 16 but with significantly reduced end-to-end execution time for the workload. Specifically, for the Sort workload the benefits are in the order of 600 s. The reason for this is the fact that our algorithm minimizes the execution rounds and it increases the number of tasks

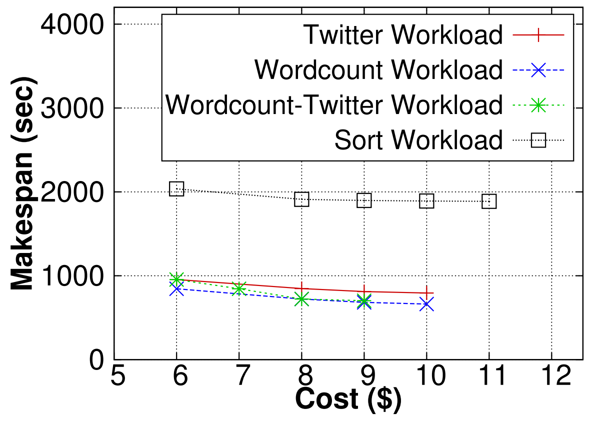


Fig. 16 Detected frontiers for empirical workloads

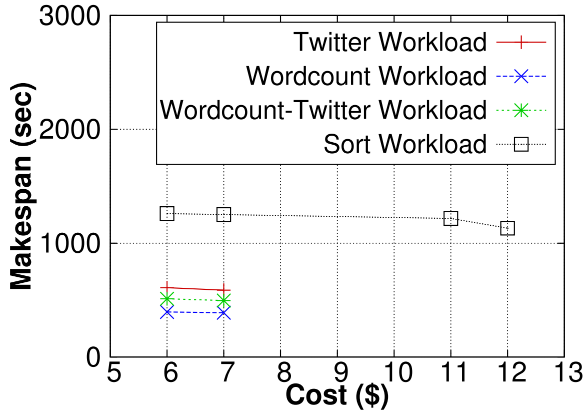


Fig. 17 Detected frontiers for empirical data with task adjustment enabled

only if we have a performance gain. Our cluster environment had minimum resources, so our algorithm adjusted the tasks to be equal to the allocated slots minimizing the required execution rounds and correspondingly the end-to-end execution time. So we argue that it is extremely important to adjust the number of jobs' tasks as we can gain significant performance improvements in terms of the workload's makespan.

6.5.3 Buffer adjustment

Finally, we evaluated the impact of the automatic map buffer size configuration. Our Random Search algorithm, described in Section 4.4, performed 10,000 iterations. We compared our proposed algorithm with the one that exhaustively enumerates all the possible configurations and keeps the setting that minimizes the I/O cost. The three buffer parameters are three float numbers, so in order to sample them, we use three small float as *step sizes* for the sampling procedure. For example, the mBuffer_j parameter had as initial value 128.0 MB while its maximum value was 512.0 MB. Then we sampled this range of values using 25.0 MB as the step size. It should be clear that, similarly to the Optimal algorithm, we used for the Pareto frontier construction in Fig. 15, this technique requires to generate all the possible combinations of the values of these three parameters. We considered this technique to display how close our approach is to the optimal configuration. Also, we compared our approach with the RRS algorithm used by Starfish and with an enumeration algorithm that puts the worst possible settings. The latter was used to depict the impact of these parameters in the observed cost. RRS [27] extends the Random Search technique by performing more iterations in areas of the solution space where a good solution has been detected. More specifically, in the beginning of the search, RRS performs sampling from the whole parameter space and thus detects a solution similar to the one found with the Random Search algorithm. Then the search continues by gradually shrinking the sample space to detect better configurations as we expect them to be near the already detected configuration. So in general it requires more time to tune the parameters but the detected configuration will be better than the one found with the simple Random Search technique. For the RRS algorithm we also set the maximum number of allowed iterations to be equal to 10,000.

In Fig. 18, we illustrate the benefits of our approach. We reduce significantly the required budget and our proposal is comparable to the optimal configuration. The execution time of our approach is rather small as it requires less than 100 ms (as you can see in Fig. 19) compared to RRS approach that performs a more thorough search of the search space until it converges to an optimal configuration. In our problem, we tune only three configuration parameters; therefore, the overhead imposed by RRS does not prove beneficial in the quality of the detected solutions and only increases the execution time of the search procedure.

7 Related work

7.1 Task scheduling

The problem of real-time scheduling of bag-of-tasks applications in cloud environments is a well-known and thoroughly studied problem [51], where different scheduling criteria have been considered including completion time [52], cost [53], and energy consumption [54]. However, we argue that the MapReduce environment is coherently different from the environments considered in these works. The MapReduce paradigm is probably the most widely adopted approach today to simplify the development and parallel execution of data processing jobs. The benefit over traditional schemes is that the MapReduce framework provides a more generic key/value data

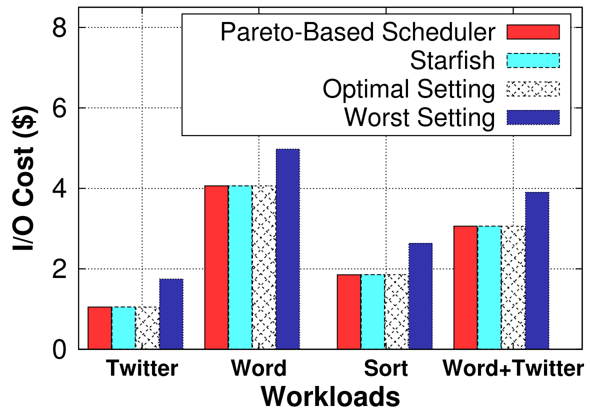


Fig. 18 I/O cost comparison using different buffer adjustment techniques

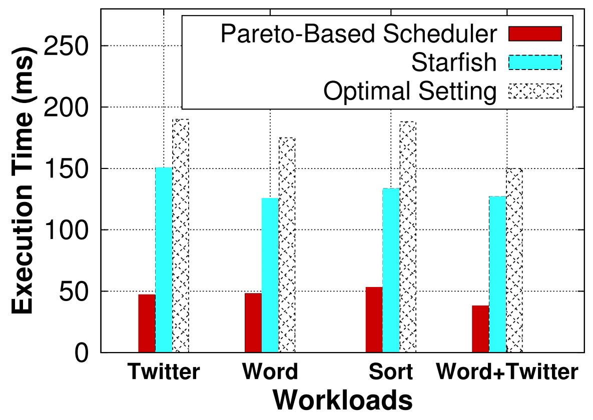


Fig. 19 Comparison of the execution times of the different buffer adjustment techniques

model which allows programmers to define arbitrarily complex user functions which then are wrapped as map and reduce tasks. This way, non-expert users or application programmers can rely on the semantics of the map/reduce functions and are not concerned with the concrete parallelization strategies. However, in MapReduce environments, it is necessary to tune a large number of configuration parameters that can affect the jobs' execution times so these parameters should be also considered during the scheduling process. The closest work in this domain that bares similarities to our problem was examined in [44]. The authors applied a Pareto frontier construction technique for detecting the task replication to use in a mixed cloud and grid environment. However, they examine different parameters as they aim at determining the optimal replication level and deadlines to use. Furthermore, their parameter space is limited so the Pareto frontier is constructed using exhaustive search. In contrast, our search space grows exponentially based on the number of jobs that comprise the workload therefore exhaustive search is not applicable.

7.2 MapReduce job scheduling

There has been prior work in regards to optimizing the slots' allocation for a single job minimizing the monetary cost while at the same time meeting deadline criteria [55]. We focus on multiple concurrently running jobs and not in a single job optimization. Furthermore, approaches like [23], [24], and [41] try to minimize standard scheduling theory metrics (e.g. makespan) in MapReduce workloads but they do not consider the reserved resources impact on the user's budget.

7.3 Autotuning MapReduce Jobs

Works regarding the automatic configuration and job profiling have mainly been done by [20] and [25]. In [20], they provide a profiler for the jobs and a what-if engine where queries can be issued to check the performance of the job for different configurations and resource allocations. They use the Recursive Random Search (RRS) algorithm for detecting the configuration parameters that maximize their objective function. Another recent proposal for Hadoop autotuning was described in [25] where they tune parameters to minimize the execution time of a single job. In their proposal, they focus on overlapping the execution time of map and reduce phases. However, they do not clearly specify how they allocate the slots to their jobs. We differ from these approaches in the fact that we examine a multi-objective optimization problem where the user submits multiple jobs and we want to provide her near-optimal slots' allocations.

7.4 Budget Allocation in MapReduce

Allocating the users' budget for executing MapReduce jobs in a cloud environment was first studied in [18], where they provide a schema that allocates slots to user's jobs based on the currently allocated budget. Our approach differs in the fact that the users submit multiple jobs each with different requirements. Authors in [42] examine the budget allocation problem in the context of MapReduce workflows, while in [56], they determine the optimal cluster setup for a Hadoop workload. Both works examine two different optimization problems, the first has as constraint the necessary budget while the second constraints the workload's makespan; in contrast, we focus on the multi-objective version of the aforementioned problems. In our previous work [57], we motivated the necessity of cost-effective slots' allocations and provided an overview of a framework for detecting them.

7.5 Multi-cluster environments

Finally, another problem that is studied by the research community is how to distribute the execution of a MapReduce workload among multiple clusters and balance the resource usage between them [58]. This is a different problem than ours as we focus on the execution of the workload in a dedicated Hadoop cluster, examining the budget/makespan trade-off. In our previous work [40], we also examined the scheduling problem in multiple-clusters environments focusing on the budget/performance trade-off of the possible jobs-to-clusters assignments. This is essentially a different problem than the one studied in this work. More specifically, our proposed Pareto-based scheduler can be used for the intra-job scheduling (i.e., determine the per cluster map and reduce slots' allocation) in the individual clusters while the scheduler proposed in [40] can be used for determining how jobs should be assigned to the available clusters.

8 Conclusions

In this paper, we present a novel Pareto-based scheduler for exploring cost-performance trade-offs for MapReduce workloads. Our proposal is beneficial to CPS and enterprises that execute their MapReduce workloads in public cloud infrastructures like Amazon's EC2. Our scheduler methodically searches and detects slots' allocations for the jobs that balance the workload's makespan and monetary cost, using a novel search algorithm. Furthermore, we show that by automatically tuning basic configuration parameters, such as the number of tasks and the map tasks' buffer size, we can greatly improve the applications' performance and decrease further the user's spending budget.

Our detailed experimental evaluation with both simulation and empirical workloads illustrates that our Pareto-based scheduler is able to detect valid slots' allocations and outperforms significantly current state-of-the-art techniques, like genetic algorithms and Starfish, in terms of the required execution time. More specifically, Pareto-based scheduler requires less than 100 ms for scheduling 25 concurrently submitted jobs while all the other techniques require more 5 s. Furthermore, in the empirical evaluation, we demonstrate that it is important to tune the map buffer parameters in all the examined workloads as we can reduce the I/O monetary cost more than 25%. Finally, our proposed task adjustment optimization enhances the performance of the Pareto-based scheduler by minimizing further the workload's makespan. For example, in a Sort workload, we are able to decrease its execution time more than 600 s. Therefore, we argue that task adjustment should be always applied when the scheduler determines the per job slots' allocations.

Acknowledgments
This research has been financed by the European Union through the FP7 ERC IDEAS 308019 NGHCS project and the Horizon2020 688380 VaVeL project.

Authors' information
Nikos Zacheilas is a PhD student at Athens University of Economics and Business under the supervision of Prof. Vana Kalogeraki. He received his BSc in Informatics from the University of Athens, followed by an MSc in Computer Science from Athens University of Economics and Business. His research interests include complex event processing, scheduling, and resource management in distributed systems.
Prof. Vana Kalogeraki received her PhD from the University of California, Santa Barbara in 2000. Previously, she has held positions as an Associate and Assistant Professor at the Department of Computer Science at the University of California, Riverside and as a Research Scientist at Hewlett-Packard Labs in Palo Alto, CA. She is currently at the Department of Informatics at Athens University of Economics and Business where she is leading the Distributed Systems research. Prof. Kalogeraki has been working in the field of distributed and real-time systems, distributed sensor systems, peer-to-peer systems, resource management and fault-tolerance for over 15 years. She has published over 100 journal and conference papers and contributions to books and has co-authored the OMG CORBA Dynamic Scheduling Standard. Prof. Kalogeraki was invited to give keynote talks and deliver tutorials and seminars on peer-to-peer computing. She has served as General Chair and Program Chair on many conferences in the area of distributed systems as well as program committee member on over 100 conferences. She was also awarded an ERC Starting Grant, a Marie Curie Fellowship, two best paper awards at the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009) and the 9th IEEE Annual International Symposium on Applications and the Internet (SAINT 2008), a Best Student Paper Award at the 11th IEEE/IPSJ International Symposium on Applications and the Internet(SAINT 2011), a UC Regents Fellowship Award, UC Academic Senate Research Awards and a research award from HP Labs. Her research has been supported by the European Union, NSF and gifts from SUN and Nokia.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Received: 14 October 2016 Accepted: 15 June 2017
Published online: 03 July 2017

References

1. J Dean, S Ghemawat, *MapReduce: Simplified data processing on large clusters*. (OSDI, San Francisco, 2004)
2. Hadoop (2016). http://hadoop.apache.org. Accessed 14 October 2016
3. Twitter (2016). http://twitter.com. Accessed 14 October 2016
4. Yahoo! (2016). http://www.yahoo.com. Accessed 14 October 2016
5. A Thusoo, JS Sarma, N Jain, Z Shao, P Chakka, S Anthony, H Liu, P Wyckoff, R Murthy, Hive—a warehousing solution over a Map-Reduce framework. PVLDB. **2**(2), 1626–1629 (2009)
6. JK Laurila, D Gatica-Perez, I Aad, J Blom, O Bornet, Do T-M-T, O Dousse, J Eberle, M Miettinen, From big smartphone data to worldwide research: The Mobile Data Challenge. Pervasive Mob. Comput. **9**(6), 752–771 (2013). Elsevier, Atlanta. https://doi.org/10.1016/j.pmcj.2013.07.014
7. EA Lee, in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), Orlando, Florida, USA*. Cyber physical systems: design challenges (IEEE, 2008), pp. 363–369
8. RR Rajkumar, I Lee, L Sha, J Stankovic, in *Proceedings of the 47th Design Automation Conference*. Cyber-physical systems: the next computing revolution (ACM, 2010), pp. 731–736
9. N Zygouras, N Zacheilas, V Kalogeraki, D Kinane, D Gunopulos, *Insights on a scalable and dynamic traffic management system*. (EDBT, Brussels, Belgium, 2015), pp. 653–664
10. N Panagiotou, N Zygouras, I Katakis, D Gunopulos, N Zacheilas, I Boutsis, V Kalogeraki, S Lynch, B O'Brien, in *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery (ECML/PKDD'16), Riva del Garda, Italy*. Intelligent urban data monitoring for smart cities (Springer, Riva del Garda, 2016)
11. Y Simmhan, S Aman, A Kumbhare, R Liu, S Stevens, Q Zhou, V Prasanna, Cloud-based software platform for big data analytics in smart grids. Comput. Sci. Eng. **15**(4), 38–47 (2013)
12. AJ Jara, D Genoud, Y Bocchi, in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2014 Eighth International Conference, Birmingham, UK*. Big data for cyber physical systems: an analysis of challenges, solutions and opportunities (IEEE, Birmingham, 2014)
13. Spark: 2016. http://spark.apache.org. Accessed 14 October 2016
14. Y Simmhan, A Kumbhare, Floe: a continuous dataflow framework for dynamic cloud applications (2014). arXiv preprint arXiv:1406.5977 http://adsabs.harvard.edu/abs/2014arXiv1406.5977S
15. Big Data & HPC. Powered by the AWS Cloud: 2016. http://aws.amazon.com/solutions/case-studies/big-data/. Accessed 14 October 2016
16. Amazon EC2: 2016. http://aws.amazon.com/ec2/. Accessed 14 October 2016
17. Microsoft Azure: 2016. http://azure.microsoft.com/en-us/. Accessed 14 October 2016
18. T Sandholm, K Lai, *Dynamic proportional share scheduling in Hadoop*. (JSSPP, Atlanta, GA, USA, 2010)
19. Amazon EBS: 2016. http://aws.amazon.com/ebs/. Accessed 14 October 2016
20. H Herodotou, S Babu, in *Proceedings of the VLDB Endowment*. Profiling, What-if analysis, and cost-based optimization of MapReduce programs, vol. 4 (VLDB, Seattle, 2011), pp. 1111–1122

21. S Babu, *Towards automatic optimization of MapReduce programs*. (SoCC, New York, USA, 2010)
22. Cloudera: 2016. http://www.cloudera.com/content/cloudera/en/home.html. Accessed 14 October 2016
23. A Verma, L Cherkasova, RH Campbell, Orchestrating an ensemble of MapReduce jobs for minimizing their Makespan. IEEE Trans. Dependable Secure Comput. **10**(5), 314–327 (2013)
24. X-Q Chai, Y-L Dong, J-F Li, Profit-oriented task scheduling algorithm in Hadoop cluster. EURASIP J. Embed. Syst. **2016**(1), 1 (2016)
25. J Shi, J Zou, J Lu, Z Cao, S Li, C Wang. MRTuner: a toolkit to enable holistic optimization for MapReduce jobs, vol. 7 (VLDB Endowment, Hangzhou, 2014), pp. 1319–1330
26. Apache Mahout: 2017. http://mahout.apache.org/. Accessed 31 January 2017
27. T Ye, S Kalyanaraman, A recursive Random Search Algorithm for large-scale network parameter configuration. ACM SIGMETRICS Perform. Eval. Rev. **1**, 196–205 (2003). ACM, San Diego
28. S Kavulya, J Tan, R Gandhi, P Narasimhan, in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. An analysis of traces from a production MapReduce cluster (IEEE, Melbourne, 2010), pp. 94–103
29. Z Ren, X Xu, J Wan, W Shi, M Zhou, in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. Workload characterization on a production Hadoop cluster: a case study on Taobao (IEEE, La Jolla, 2012), pp. 3–13
30. F Ahmad, S Lee, M Thottethodi, T Vijaykumar, *Puma: Purdue mapreduce benchmarks suite*, (2012)
31. N Panagiotou, N Zygouras, I Katakis, D Gunopulos, N Zacheilas, I Boutsis, V Kalogeraki, S Lynch, B O'Brien, D Kinane, in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Insight: dynamic traffic management using heterogeneous urban data (Springer, 2016), pp. 22–26
32. Apache's Storm: 2016. http://storm.apache.org/. Accessed 14 October 2016
33. IBM's Streams: 2016. http://www-03.ibm.com/software/products/en/ibm-streams. Accessed 14 October 2016
34. VK Vavilapalli, AC 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, E Baldeschwieler, in *Proceedings of the 4th annual Symposium on Cloud Computing*. Apache Hadoop YARN: yet another resource negotiator (Santa Clara, 2013), p. 5
35. A Verma, L Cherkasova, RH Campbell, in *Proceedings of the 8th ACM international conference on Autonomic computing*. ARIA: automatic resource inference and allocation for MapReduce environments (ACM, Karlsruhe, 2011), pp. 235–244
36. K Morton, A Friesen, M Balazinska, D Grossman, in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. Estimating the progress of MapReduce pipelines (IEEE, Long Beach, USA, 2010), pp. 681–684
37. Z Zhang, L Cherkasova, BT Loo, in *In Proceedings of the 6th IEEE International Conference on Cloud Computing, CLOUD'13*. Performance modeling of MapReduce jobs in heterogeneous environments (IEEE, Santa Clara, 2013)
38. B Sharma, T Wood, CR Das, in *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*. HybridMR: a hierarchical Mapreduce scheduler for hybrid data centers (IEEE, Philadelphia, 2013), pp. 102–111
39. A Verma, L Cherkasova, RH Campbell, in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. Play It Again, SimMR! (IEEE, Austin, 2011), pp. 253–261
40. N Zacheilas, V Kalogeraki, in *Autonomic Computing (ICAC), 2016 IEEE International Conference on*. Chess: Cost-effective scheduling across multiple heterogeneous mapreduce clusters (IEEE, Wurtzburg, 2016), pp. 65–74
41. N Zacheilas, V Kalogeraki, in *ICAC*. Real-time scheduling of skewed MapReduce jobs in heterogeneous environments (Usenix, Philadelphia, 2014), pp. 189–200
42. Y Wang, W Shi, Budget-driven scheduling algorithms for batches of MapReduce jobs in heterogeneous clouds. Cloud Comput. IEEE Trans. **2**(3), 306–319 (2014)
43. S Börzsönyi, D Kossmann, K Stocker, in *Data Engineering, 2001. Proceedings. 17th International Conference on*. The skyline operator (IEEE, Heidelberg, 2001), pp. 421–430
44. OA Ben-Yehuda, A Schuster, A Sharov, M Silberstein, A Iosup, in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. ExPERT: Pareto-efficient task replication on grids and a cloud (IEEE, Shanghai, 2012), pp. 167–178
45. AA Zhigljavsky, *Theory of Global Random Search*, vol. 65. (Springer & Business Media, 2012)
46. Hadoop's API: 2016. https://hadoop.apache.org/docs/r1.2.1/api/. Accessed 14 October 2016
47. Rumen: a tool to extract job characterization data from job tracker logs: 2016. https://issues.apache.org/jira/browse/MAPREDUCE-751. Accessed 14 October 2016
48. Hadoop's Fair Scheduler XML file: 2016. https://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html. Accessed 14 October 2016
49. K Deb, A Pratap, S Agarwal, T Meyarivan, A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. Evol. Comput. IEEE Trans. **6**(2), 182–197 (2002)
50. S Kukkonen, J Lampinen, GDE3: the third evolution step of generalized differential evolution. Evol. Comput. IEEE Trans. **1**, 443–450 (2005)
51. MA Netto, R Buyya, in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium On*. Offer-based scheduling of deadline-constrained bag-of-tasks applications for utility computing systems (IEEE, 2009), pp. 1–11
52. A Sulistio, W Schiffmann, R Buyya, in *International Conference on High-Performance Computing*. Advanced reservation-based scheduling of task graphs on clusters (Springer, 2006), pp. 60–71
53. S Selvarani, GS Sadhasivam, in *IEEE*. Improved cost-based algorithm for task scheduling in cloud computing, (2010), pp. 1–5
54. RN Calheiros, R Buyya, in *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference On*. Energy-efficient scheduling of urgent bag-of-tasks applications in clouds through dvfs (IEEE, 2014), pp. 342–349
55. A Verma, L Cherkasova, VS Kumar, RH Campbell, in *Network Operations and Management Symposium (NOMS), 2012 IEEE*. Deadline-based workload management for MapReduce environments: pieces of the performance puzzle (IEEE, Maui, 2012), pp. 900–905
56. Z Zhang, L Cherkasova, BT Loo, in *SIGMETRICS Performance Evaluation Review 42*. Exploiting cloud heterogeneity for optimized cost/performance mapreduce processing (ACM, Portland, 2015), pp. 38–50
57. N Zacheilas, V Kalogeraki, in *Autonomic Computing (ICAC), 2015 IEEE International Conference on*. A framework for cost-effective scheduling of MapReduce applications (IEEE, 2015), pp. 147–148
58. B Ghit, N Yigitbasi, A Iosup, D Epema, in *SIGMETRICS, Austin, TX, USA*. Balanced resource allocations across multiple dynamic mapreduce clusters (ACM, Austin, 2014), pp. 329–341