Dynamic Configuration of Partitioning in Spark Applications

Anastasios Gounaris, Georgia Kougka, Rubèn Tous, Carlos Tripiana, and Jordi Torres

Abstract—Spark has become one of the main options for large-scale analytics running on top of shared-nothing clusters. This work aims to make a deep dive into the parallelism configuration and shed light on the behavior of parallel spark jobs. It is motivated by the fact that running a Spark application on all the available processors does not necessarily imply lower running time, while may entail waste of resources. We first propose analytical models for expressing the running time as a function of the number of machines employed. We then take another step, namely to present novel algorithms for configuring dynamic partitioning with a view to minimizing resource consumption without sacrificing running time beyond a user-defined limit. The problem we target is NP-hard. To tackle it, we propose a greedy approach after introducing the notions of dependency graphs and of the benefit from modifying the degree of partitioning at a stage; complementarily, we investigate a randomized approach. Our polynomial solutions are capable of judiciously use the resources that are potentially at user's disposal and strike interesting trade-offs between running time and resource consumption. Their efficiency is thoroughly investigated through experiments based on real execution data.

Index Terms—data repartitioning, data flow optimization, data flow profiling, Spark

1 Introduction

Spark has become one of the main options for end-to-end processing of large volumes of data. Initially proposed in [1], [2], it has nowadays evolved as one of the most active open-source Apache projects with a huge developer and user community. Its remarkably wide adoption stems from several characteristics that are of high practical significance. Spark provides a common framework for processing both batch and streaming data. It is compatible with other broadly used technologies, such as R, NoSQL databases, HDFS, YARN and MESOS to name a few. It tackles several key performance limitations of traditional MapReduce through the exploitation of the main memory that is available to the largest possible extent. In addition, it is bundled with components for processing structured data in an SQL-like fashion and applying built-in machine learning algorithms.

Spark abstracts data collections as Resilient Distributed Datasets (RDDs), which are partitioned across several nodes so that they can be operated on in parallel. As reported in several sources, e.g., [3], [4], it can run jobs up to two order of magnitude faster than Hadoop MapReduce.

Nevertheless, its configuration and fine-tuning remains a big challenge. The work in [5] has started to scratch the surface of this issue and, after extensive experimentation in a high-performance computing (HPC) platform, namely the Marenostrum III (MN3), the supercomputer at Barcelona Supercomputing Centre (BSC), and an additional commercial infrastructure, has reached to two main conclusions: (i) for each infrastructure, a different setting of number of cores managed by a single executor yields the highest performance in terms of execution time, while suboptimal settings may lead to performance degradation of several factors despite using exactly the same number of cores; and (ii) for each application and depending on the intensity of data shuffling, a different number of data partitions need to be allocated to each CPU core; this number ranges from 1 to 4 for applications running on MN3 while inn applications where CPU cost dominates, having a single partition per core yields lower running times.

Building on these early results and further evidence that CPU usage is (rather counter-intuitively) the key aspect in the performance of Spark jobs [6], in this work we take a deeper look at the issue of data partitioning. Data partitioning refers to the number of chunks that the datasets to be processed are split, and strongly relates to the degree of parallelism of execution, i.e., the number of cores and machines used for execution.

The motivation behind our effort is the fact that lower performance due to suboptimal partitioning is inherently related to resource waste, i.e., resources are commonly used unnecessarily longer to perform the same tasks. As such, our main goal is to investigate the relationships and trade-offs between performance and resource consumption in Spark applications and propose techniques that modify the data partitioning dynamically, i.e., during execution.

To this end we make the following contributions:

1) *Profiling of Spark programs:* we perform profiling of Spark applications and we explain how we can describe their behavior as a function of the number of machines used. The analytical cost models that we propose are more accurate and improve upon current models for MapReduce, like the one in [7].

• *A. Gounaris and G. Kougka are with the Aristotle University of Thessaloniki, Greece.*
Email: gounaria, georkoug@csd.auth.gr
• *R. Tous is with the Universitat Politècnica de Catalunya, Spain*
Email: rtous@ac.upc.edu
• *C. Tripiana and J. Torres are with the Barcelona Supercomputing Centre, Spain*
Email: carlos.tripiana, jordi.torres@bsc.es

2) *Proposal of novel algorithms for the selection of the degree of partitioning taking into account both the resource consumption and running time:* we present novel solutions for configuring the level of data partitioning dynamically, advocating to depart from the usual usage pattern of either sticking to the level of partitioning of the source data or employ all the machines available. Our solutions make judicious choices regarding the degree of partitioning for different stages in the same Spark application. We investigate two main approaches, namely greedy techniques based on the notions of dependency graph and stage benefit, and a randomized technique. These solutions manage to strike different balances between the two objectives. Formally, we study the problem of minimizing resource consumption under the constraint that running time does not increase more than a user-defined threshold. No other solutions for the same problem exist to date.
3) *Thorough experiments:* we evaluate our proposals using real data. The results are particularly promising. We show that we can reach interesting trade-offs between resource consumption and running time. We further provide evidence that we can improve on resource consumption up to more than 6 times at the expense of 50% higher running time. We show that *"more is less"* holds, i.e., it is possible to use less resources to execute faster, but even if we first optimize for running time, we can trade a small amount of performance for significant savings in resource consumption.

Structure of the remainder of the paper: In Section 2, we analyze the behavior of Spark applications and we present the formulas that describe running time as a function of the machines used. Section 3 formally introduces the problem of trading performance for resource consumption and discusses the techniques we propose in detail. In Section 4, we use real execution data and a wide range of applications types to evaluate our proposals. We discuss end-to-end processing issues and related work in Sections 5 and 6, respectively, and we conclude in Section 7.

2 Profiling of Spark Programs

In this section, we first provide execution details of Spark programs and we present our benchmarking applications. The main part of this section consists of our profiling results, which explain the behaviour of Spark applications as a function of the number of the machines used.

2.1 Background

A Spark application consists of two types of operations, namely *transformations* and *actions*. Transformations, such as *map* and *filter*, apply a function on each RDD element and result in a new RDD. Actions trigger the execution of such functions and produce meaningful results. Example actions include *reduce*, *count* and *collect*, which return a value to the coordinator program, called *driver*. For each action in the application, a *job* is performed, which includes several RDD transformations.

Spark's scheduler creates a physical execution plan for the job based on the directed acyclic graph (DAG) of transformations. The physical plan is divided into *stages*. A *stage* is a sequence of transformations that can be pipelined (executed, without data movement, in parallel over all data partitions). Pipelining transformations are deliberately grouped together in a single stage to speed-up performance. The sequence of computations defined by a *stage* instantiated over a single data partition is called a *task*. A *task* is the actual unit of execution of the physical plan. The task scheduler assigns tasks to workers via the cluster manager The degree of partitioning at each stage is equal to the number of tasks for each operation included in that stage.

Data may be *repartitioned* across RDDs. This may be done as a result of a specific transformation. For example, *groupByKey* and *sortByKey* are applied to a key-value pair RDD and result in a new RDD, where data are partitioned across machines differently. This data re-distribution is commonly referred to as data *shuffling*. Data shuffling is an expensive operation, but it is important to note that the cost is not only network-related. Data shuffling serializes data and stores temporary data on disk if they cannot fit in main memory; temporary files are kept as long as they are needed for fault tolerance purposes. As such, the cost is also CPU and disk I/O related. In the generic case, when data are repartitioned, the degree of partitioning can change as well.[1]

2.2 Our setting and the benchmarking applications

We ran our profiling experiments on the MN3 supercomputer provided by BSC in the city of Barcelona. With the last upgrade, MN3 has a peak performance of 1.1 Petaflops. At June 2013, MareNostrum was positioned at the 29th place in the TOP500 list of fastest supercomputers in the world, whereas according to the latest TOP500 list in November 2015, MareNostrum is 93rd. A full technical description of MN3 and how it supports Spark applications is in [5]. Spark allows for several cluster managers: standalone, YARN and MESOS. Spark4MN employs the former, and before each execution, exclusively reserves the amount of machines requested by the user, i.e., there are no multi-tenancy issues during Spark application execution.

Our benchmarking applications are elementary ones, i.e., they can be expressed as a single built-in function in the context of larger applications:

sort (by key): this application includes data shuffling and its two main stages between which shuffling takes place are shown in Figure 1(left).

aggregate (by key): this application also includes data shuffling. Its distinctive characteristic is that it reduces the result size on the reducer side.

k-means: this is an example of an iterative CPU-intensive algorithm, where, in each iteration, local processing is followed by transmission of summary information about new cluster centers. The DAGs of the two main stages in each iteration are shown in Figure 1(right).

1. In the remainder, when the term repartitioning is used, it will denote change in both data distribution and the degree of partitioning, whereas shuffling will denote re-distribution without modifying the degree of partitioning.

naive-bayes: this is a simple machine learning algorithm with no iterations and less inter-communication between machines participating in the execution.

shuffling: this benchmarking application just reshuffles data across machines. Its execution resembles that of sort, but without performing any meaningful processing.

In our experiments, the number of machines (resp. cores) used ranges from 8 (resp. 128) to 96 (resp. 1536). This is in line with reports in [8], [9], [10] that the majority of clusters are rather small and have fewer than 50 nodes. Also, we experimented with raw datasets at the orders of hundreds gigabytes (the corresponding RDDs are 1-5 times larger), since, this is the typical dataset processed even in companies that are notorious for their big data application demands [11]. The Spark version was 1.5.2.

We divide profiling programs into three categories: those that do not change the degree of partitioning, those that change the degree of partitioning without performing other operations, and those that both perform meaningful operations and change the degree of partitioning.

Degree of partitioning vs. degree of parallelism: In general, the degree of parallelism, i.e., the number of machines/cores employed is different from the degree of partitioning. To avoid imbalanced execution, the degree of partitioning must be equal to the number of cores multiplied by a small integer. However, based on (i) the evidence in [5] that, for CPU-intensive applications on MN3, the most efficient configuration of the degree of partitioning is to be set equal to the number of cores, and (ii) the evidence in [6], where the main performance bottlenecks are the CPU ones, in this work, we always set the degree of partitioning to the number of cores. Also, we allow the usage of complete machines, each consisting of 16 cores. Consequently, configuring the number of machines defines the degree of partitioning, too, and vice versa.

2.3 Simple Application Profiling

We first examine Spark programs that do not modify the degree of partitioning during RDD manipulation. For k-means, we study 5 variants. The first three operate on 100-dimensional records, and the sizes are 100M (*k-meansI*), 50M (*k-meansII*) and 150M (*k-meansIII*) records, respectively (the sizes in bytes are 100GB, 50GB and 150GB, respectively). The two other k-means programs operate on 10M 1000-dimensional (*k-meansIV*) and 1000M (*k-meansV*) 10-dimensional records, respectively. The other datasets are

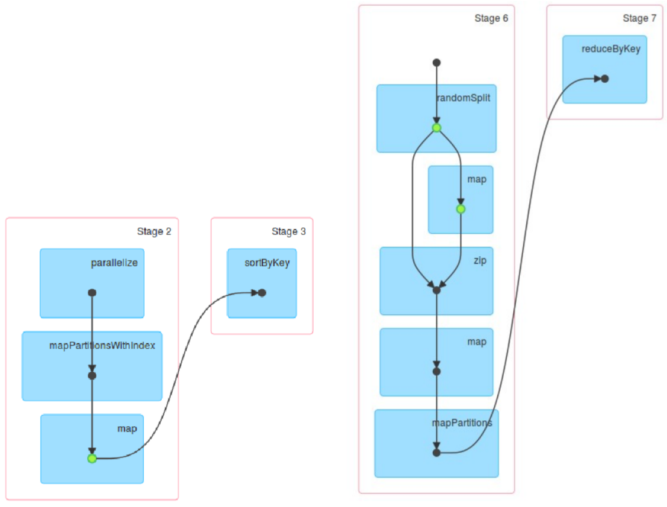


Fig. 1. Example DAGs of part of the stages from the execution of sort (left) and k-means (right).

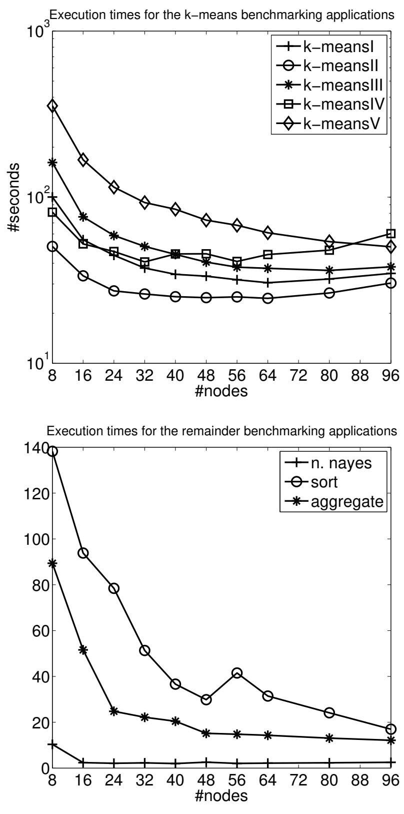


Fig. 2. Execution times for the benchmarking applications.

TABLE 1
Fitting errors

	Eq. (1,2)		Eq. (3)	
program	R^2	*SSE*	R^2	*SSE*
k-meansI	0.998	6.967	0.998	6.501
k-meansII	0.992	4.828	0.996	2.313
k-meansIII	0.995	71.881	0.996	50.339
k-meansIV	0.949	67.191	0.961	51.456
k-meansV	0.999	224.155	0.999	59.42
n. bayes	0.914	5.088	0.964	2.512
sort	0.963	486.123	0.97	387.536
aggregate	0.985	79.188	0.986	77.701
shuffle	0.692	16.851	0.922	4.242

	Eq. (4)		Eq. (5)	
program	R^2	*SSE*	R^2	*SSE*
k-meansI	0.999	2.399	0.999	2.49
k-meansII	0.996	2.116	0.996	2.085
k-meansIII	1	0.954	1	3.007
k-meansIV	0.970	39.626	0.969	41.26
k-meansV	1	24.38	1	21.06
n. bayes	0.995	0.326	0.973	1.636
sort	0.97	387.536	0.97	387.536
aggregate	0.986	77.701	0.986	77.701
shuffle	0.967	1.79	0.996	0.23

also 100GB in size. The dataset for naive-bayes consists of 100M 100-dimensional records, whereas the datasets for the last two programs consist of 1B records of 100 bytes each.

Figure 2 shows the plots of runs on MN3. All runs were repeated 5 times and the median value is presented. From this figure, two main observations can be drawn. Firstly, for small degrees of parallelism, i.e., 8-24 nodes, the executions times are drastically reduced when increasing the number of machines employed. Super-linear speed-ups can be observed as well since increasing the memory available decreases the possibility to use the disk during shuffling. For example, for naive bayes, when going from 8 machines to 16, the execution time is reduced to less than a half. Secondly, after a saturation point, employing new machines yields either negligible performance improvements or even performance degradation.

The main question we try to answer is: *can we propose a formula that can describe the behavior illustrated in Figure 2?* More specifically, we ask for a formula $T(n)$, which can express the execution time of a Spark application as a function of the number of nodes used n. The starting point is the formula for the empirical speedup metric encountered in Hadoop applications as explained in [7]:

$$\frac{T(1)}{T(n)} = \frac{n}{1 + \sigma n + \kappa n(n-1)} \tag{1}$$

where σ and κ are the contention and data distribution incoherency coefficients, respectively. The interesting note about this formula is that, in massively parallel settings, σ can be negative, thus leading to super-linear scalability for specific ranges of n values. Simple algebraic manipulation of Eq. (1) gives the following generic template:

$$T(n) = a + \frac{b}{n} + cn \tag{2}$$

where a, b and c are constants.

A weak point of the above equation is that it essentially ignores the fact that each Spark applications includes some shuffling phases and the cost of shuffling depends on the number of pairs of senders and receivers, which is $O(n^2)$. So, an initial improvement on [7] towards more accurate cost formulas is to add a shuffling coefficient d as follows:

$$T(n) = a + \frac{b}{n} + cn + dn^2 \tag{3}$$

In Eq. (3), the coefficients a, b and d correspond to the non-parallelizable cost, the parallelizable cost and the overhead due to shuffling communication. As such, they are expected to be non-negative. Coefficient c captures the contention factor, similar to σ in Eq. (1). Since adding new nodes increases the aggregate memory that is available and thus decreases the possibility to use the disk for intermediate storage, it is common, if not likely, c to be negative.

We proceed one step further and we add a new coefficient. There are two ways to do that. The first is to add a coefficient e in the denominator of the second component yielding the following cost model (n_{min} is the lowest possible degree of partitioning):

$$T(n) = a + \frac{b}{n-e} + cn + dn^2,\ e \in [0, n_{min} - 1] \tag{4}$$

TABLE 2
Coefficients for Eq. (4) and (5)

Coefficients for Eq. (4)					
program	a	b	c	d	e
k-meansI	33.626	343.61	-0.33	0.0032	3.019
k-meansII	20.896	192.12	-0.097	0.0018	1.664
k-meansIII	53.314	358.17	-0.659	0.0048	4.841
k-meansIV	54.336	52.186	-0.588	0.0066	6.327
k-meansV	36.38	1774	-0.545	0.0000	2.439
n. bayes	1.404	8.861	0.001	0.0000	7
sort	71.586	626.39	-1.335	0.0076	0
aggregate	1.335	716.4	-0.048	0.0009	0
shuffle	30.312	-9.622	-0.6311	0.0057	7

Coefficients for Eq. (5)					
program	a	b	b'	c	d
k-meansI	37.581	224.36	2393	-0.384	0.0034
k-meansII	22.232	162.13	569.89	-0.12	0.0019
k-meansIII	60.14	0.000	6821	-0.678	0.0046
k-meansIV	50.167	0.000	2185	-0.445	0.0056
k-meansV	42.84	1503	8004	-0.099	0.0000
n. bayes	0.616	0	600.1	0.022	0.0000
sort	71.586	626.39	0	-1.335	0.0076
aggregate	1.335	716.4	0	-0.048	0.0009
shuffle	-19.731	805.533	-4424	0.546	-0.0034

This formula better captures the fact that, for small degrees of parallelism, the memory shortage, which necessitates the usage of disk, is more intensive, and the decrease in the running time for small additions of new nodes is steeper.

Another option is to split the parallelizable cost into two components, one for the cpu and I/O-bound processing, and another for the shuffling-bound part:

$$a + \frac{b}{n} + \frac{b'}{n^2} + cn + dn^2 \tag{5}$$

In Table 1, we compare the fitting errors of the above formulas. The fitting errors are measured in two ways, namely R^2 and sum of square errors *SSE*. We can see that our proposals in Eq. (3)-(5) can significantly reduce the fitting errors of the proposal in Eq. (1). In addition, the last two formulas, which employ five coefficients, are significantly more accurate compared to Eq. (3), which comprises one coefficient less. The differences in the accuracy between Eq. (4) and (5) are less significant. Finally, in Table 2, we present the values of the coefficients of the two last formulas, where it can be shown that c is typically negative. The coefficients are computed using standard curve fitting techniques, such as those in the `cftool` of Matlab.

2.4 Repartitioning Profiling

Shuffling redistributes data in order to perform meaningful processing, e.g., to apply an aggregate function of data that have been previously grouped according to their key. Repartitioning simply redistributes data without performing any processing and, in general, modifies the degree of parallelism. In the following set of experiments, we redistribute data from n_s sender nodes to n_r receiver nodes.

Figure 3 shows the repartitioning cost when we repartition 100M 100-dimensional vectors. The shape of the plots

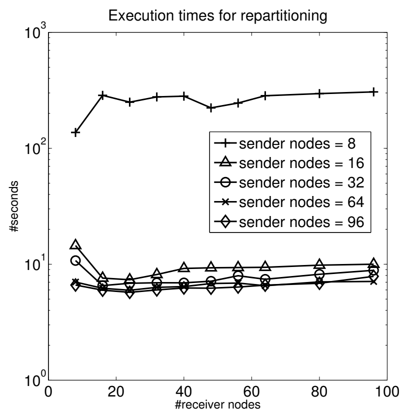


Fig. 3. Execution times for repartitioning.

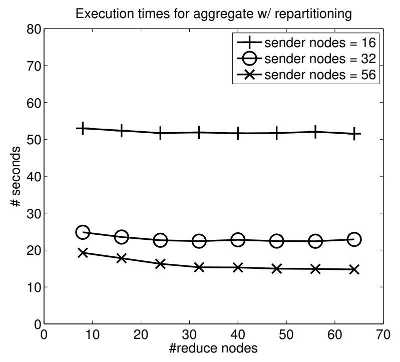


Fig. 4. Execution times for repartitioning within the application.

differs significantly between very low and higher numbers of sender nodes. In order to express the execution time as a function of both n_s and n_r we extend Eq. (4) as follows:

$$T(n) = a + \frac{b}{n_s - e} + c_s n_s + c_r n_r + d n_s n_r \qquad (6)$$

where $e \in [0, n_{min} - 1]$. Of course, it would be possible to apply the one-variable functions of the previous subsection assuming either n_s or n_r fixed, but a two-variable function better describes the behavior of repartitioning. For the specific example, R^2 is 0.9553, and the values of the parameters a, b, c_s, c_r and d are -2.377, 260.6, -0.0001, -0.0001, and 0.001132, respectively.

2.5 Profiling of Applications Coupled with Repartitioning

Certain applications can perform repartitioning while performing computations. Such applications typically contain transformations like *aggregateByKey*, *combineByKey*, *foldByKey*, *reduceByKey* and *groupByKey*, which all can receive as an optional parameter the number of the partitions on the reducer side. In Figure 4, we show three representative examples about how the *aggregate* benchmarking application behaves when we modify the number of reducer nodes from 8 to 64. The main observation is that the plots can be approximated as horizontal lines. This means that the number of reducers does not have a big impact on the running time. This, in turn, implies that we can safely conclude that, in such applications, repartitioning can be done for free, and the Equations Eq. (3)-(5) still apply.

2.6 Additional Remarks and Open Issues

In the previous subsections, we showed that we can accurately model the execution time of a parallel Spark program as a function of the nodes it employs. Although the fitting errors are small, we have observed that they can be further reduced if we allow the coefficients b and d to be negative. However, in that case, the meaning of the coefficients would lack an intuitive interpretation.

In addition, it is unclear how the parameters are modified when we keep the same applications but we modify the distribution or the volume of the dataset processed. For example, *k-meansI* processes twice as much data as *k-meansII*, whereas *k-meansIII* processes 50% more data than *k-meansI*. An open research question is whether, given the coefficients in the first row of Tables 2 and **??** for *k-meansI*, one could *estimate* the coefficients for *k-meansII* or *k-meansIII* without sacrificing accuracy. Similarly, if the distribution changes, the behavior changes as well; broadly, skewed distributions behave as if the effective degree of partitioning is lower, but there is no straightforward way to tune the coefficients automatically. A more challenging topic is to derive coefficients for an application based on known coefficients for different applications provided that they share some characteristics. We further discuss these issues in Section 5.

3 Configuration of the Partitioning Degree

The results of the previous section reveal (i) that running an application on all machines available is not necessarily the most efficient choice in terms of running time, and (ii) big decreases in the number of machines for some jobs may not result in significant performance degradation. In this section, we examine the problem of repartitioning the data throughout the application execution with a view to saving resource consumption without sacrificing performance.

A typical real-world Spark program may contain several elementary applications, like those examined as benchmarking applications. For simplicity, we consider the case where repartitioning takes place with the help of the *repartition* transformation, which is artificially inserted between other transformations. That is, we do not examine repartitioning between any pair of connected stages, as shown in Figure 1, but only between sets of stages (jobs).

3.1 Motivating Example

Consider the simplified example in Figure 5, where we process an RDD initially partitioned across 96 nodes. Each vertex in the figure, shows a distinct group of stages within the whole example application. The degree of partitioning/parallelism in each stage group remains the same. The upper part of the example shows the execution times when the degree of partitioning stays fixed at 96 nodes throughout execution. Overall, 96 nodes were occupied for 920 secs. At the bottom part of the figure, we show the execution times when we repartition data for some of the stages. The

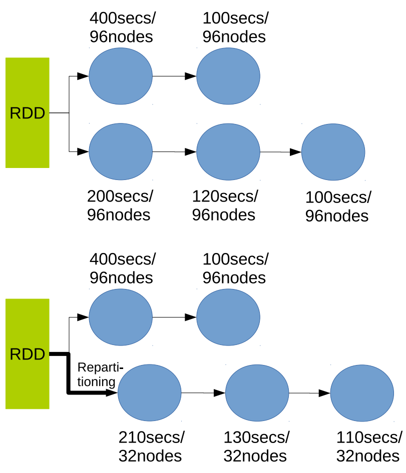


Fig. 5. Example of repartitioning with benefits for resource consumption at the expense of a slight increase in execution time.

execution time becomes 950 secs, but for 450 secs, only 32 machines were employed.

We informally define resource consumption rc as the product of the number of machines and the time they were occupied. In the first case, $rc = 920 \times 96 = 88320$. In the latter case, the resource consumption drops to $rc = 500 \times 96 + 450 \times 32 = 62400$. That is, rc drops by 29.9% at the expense of 3.3% higher execution time.

As another example from real runs on MN3, consider a Spark program that consists of the following five phases: it (i) applies naive bayes to a dataset; (ii) applies k-means; (iii) filters out the two thirds of the records; (iv) re-apples naive bayes to the remaining records; and (v) applies k-means to the filtered records. We experimented in MN3 with the following three configurations in a scenario where 96 nodes were at our disposal and the initial dataset was partitioned across all available node. *conf-a:* steps (i)-(v) ran on 96 nodes; *conf-b:* steps (i)-(iii) ran on 96 nodes and during step (iii) repartitioning took place so that steps (iv)-(v) continued on 64 nodes; and *conf-c:* steps (i)-(iii) ran on 96 nodes and steps (iv)-(v) continued on 32 nodes.

The running times of the three configurations were 75.8, 76.8, and 87.4 secs, respectively. The corresponding rc values were 7275, 6598, and 6163. Comparing *conf-a* to *conf-b*, we observe that we can trade 1.3% of running time for 9.3% less resource consumption. Comparing *conf-a* to *conf-b*, we also observe that we can trade 15% of response time for 15% less resource consumption. Further, if the initial data was partitioned across 64 nodes and this partitioning was kept throughout execution, the running time was 93 secs and rc dropped to 5952. Moreover, these times include also the cost of repartitioning, which is not shown in Figure 5. The above examples make evident that, by configuring the degree of partitioning (which, in our work, is the same as defining the degree of parallelism), we can strike a configurable balance between running time and resource utilization.

3.2 Problem Formulation

Here, we provide the formal definition of the problem of configuring the degree of partitioning. The prerequisites are as follows:

1) A DAG of a Spark data flow execution plan denoted as *G(V,E)*. Each vertex $v \in V$ describes a set of stages executed with the same degree of partitioning n_v and each edge $e = (v, v') \in E$ is a pair of vertices and denotes data transmission (with or without repartitioning) as shown in Figure 5.[2]
2) A predefined range D of possible amounts of nodes allocated to each vertex, s.t. $n_v \in D, \ \forall v \in V$.
3) Profiles for execution times for each vertex of the DAG as a function of the degree of parallelism, $cost_v(n_v)$. This cost encapsulates any shuffling taking place among different tasks within a stage, as explained in the previous section.
4) Profiles for repartitioning cost for each edge of the DAG as a function of the degree of parallelism, of the adjacent vertices $cost_{(v,v')}(n_v, n_{v'})$. Apparently, if $n_v = n_{v'}$ then $cost_{(v,v')}(n_v, n_{v'}) = 0$.

During execution, we assume that the system schedules as many resources as required to run a single vertex. As such, the vertices are executed sequentially. This is orthogonal to the fact that within a vertex, execution is performed in a both pipelined and partitioned parallelism manner. According to the above assumptions, the running time *RunningTime (RT)* of a Spark data flow G is given by the following formula:

$$RunningTime(G) = \sum_{v \in V} cost_v(n_v) + \sum_{(v,v') \in E} cost_{(v,v')}(n_v, n_{v'}) \tag{7}$$

Analogously, we define the resource consumption *ResourceConsumption (RC)* of G as follows:

$$ResourceConsumption(G) = \sum_{v \in V} n_v cost_v(n_v) + \sum_{(v,v') \in E} max\{n_v, n_{v'}\} cost_{(v,v')}(n_v, n_{v'}) \tag{8}$$

Goal: find the partitioning of each vertex n_v, so that *ResourceConsumption(G)* is minimized, while the execution time does not exceed $(1 + \varepsilon) RunningTime(n_{initial})$, where $n_{initial}$ is the unmodified degree of partitioning and $\varepsilon > 0$.

3.2.1 Problem Analysis

The problem is NP-hard. A sketch of the proof is as follows. If we build a solution to the problem of finding the degree of partitioning for each vertex that minimizes RC under no constraint on RT, then this solution can be applied to minimize RT as well, due to the linear relationship between Equations (8) and (7). Both these unconstrained problems can be deemed as instances of the *Flow Activity Allocation (FAA)* problem, which is proven to be NP-hard in [12]. Also, according to the analysis in [12], FAA cannot even be approximated by a polynomial-time algorithm with approximation error bound less than 8/7. As such, our problem is NP-hard as well, because, if it could be solved in polynomial time, then it could be used to solve FAA, by setting $\varepsilon \to \infty$. But this cannot happen as explained above.

2. Since the vertices denote sets of stages, these DAGs provide a higher-level view of the execution than the DAGs, excerpts of which are shown in Figure 1.

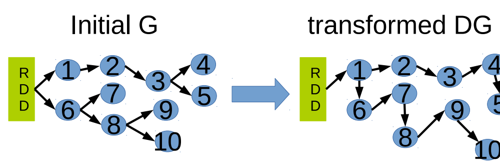


Fig. 6. Example of producing the dependency graph.

3.3 Solutions

The main rationale of our solutions is to perform repartitioning in an informed way taking into account overheads. The tricky point is that changing the partitioning of a vertex may also benefit its siblings. So, as a first step, to understand the effects of repartitioning on subsequent vertices, we introduce the dependency graph.

We assume that each RDD is cached according to its latest partitioning. As such, changing the degree of partitioning of a specific vertex impacts on both its descendants in the initial graph and its siblings that have not been processed yet. These dependencies can be better understood by producing a dependency graph DG from G, where a vertex v' depends on v, if and only if there is a path from v to v' in DG. Producing DG can be performed in linear time through processing G. The edges that differ are those connecting a parent node with multiple child nodes: each child vertex that has a sibling processed earlier is disconnected from its parent and connected to the sibling that has most recently processed; Figure 6 shows an example.

Then, we introduce the notion of the vertex *benefit*, which captures the positive effect of repartitioning a vertex on itself and other connected vertices. The exact way the befit is quantifies is flexible. We propose three distinct greedy techniques that differ in the way the benefit is calculated.

In addition, motivated by the results of [12], where randomized solutions performed well, we investigate a randomized solution in this work as well. The latter extends the solutions for single-objective resource allocation in [12] through applying a flavor *Iterated Local Search (ILS)*, which, as verified by our experiments, is capable of offering different trade-offs between running time and resource consumption than the greedy techniques.

In all our techniques, we assume that we can estimate the running time of the initial partitioning, and as such, we know the threshold $RTthreshold = (1 + \varepsilon)RunningTime(n_{initial})$.

3.3.1 Greedy Techniques

The high-level description of our greedy approach is in Algorithm 1. The input of the algorithm is: (i) the initial partitioning of each vertex and (ii) the threshold on the RT. The output of the algorithm is a new partitioning, optimized for lower RT. Holding the partitioning information requires no special data structure, i.e., an array is sufficient.

Algorithm 1 consists of three loops. The two internal loops iterate over all combinations of a vertex i and a degree of parallelism j. For each combination (i, j), the benefit is calculated. When all combinations have been examined, the combination of vertex and degree of parallelism with the highest benefit is selected. This procedure is then repeated k times, where $k \leq |V|$.[3]

Algorithm 1 Greedy specification of degree of partitioning

```
Require: $partitioning[], RTthreshold$
  for $k \leftarrow 1$ to $|V|$ do
    $maxBenefit \leftarrow 0$
    $best \leftarrow (NULL, NULL)$ //holds the best repartitioning
    for $i \leftarrow 1$ to $|V|$ do
      for $j \leftarrow 1$ to $|D|$ do
        // check partitioning $i$ by degree $j$
        Calculate $benefit(i, j)$
        Calculate $RT$ of new partitioning
        if $RT \leq RTthreshold$ then
          if $benefit(i, j) > maxBenefit$ then
            $maxBenefit \leftarrow benefit(i, j)$
            $best \leftarrow (i, j)$
          end if
        end if
      end for
    end for
    apply $best$ to $partitioning[]$
  end for
  return $partitioning[]$
```

In this work, we propose and investigate the behavior of the following three techniques that differ in the way the benefit is computed, which also impacts on what kind of repartitioning are performed in each iteration:

1) **G-local:** this technique checks the impact on RC if vertex i is partitioned in j partitions. The potential cost of repartitioning with other vertices that are connected to i through an edge is considered. If the difference of the previous RC with the new one is positive, this difference denotes $benefit(i, j)$. Otherwise, $benefit(i, j)$ is set to 0. When the most beneficial repartitioning is found, termed as $best(i, j)$ in the algorithm, $partitioning[best.i] = best.j$. The complexity is $O(|V|^2|D|)$.
2) **G-ext:** this technique extends the repartitioning of vertex i to its descendant vertices in DG, as long as RC decreases. It starts like G-local. If $benefit(i, j) > 0$, it inserts all the children vertices of i in DG into a queue. Then, it recursively applies the same procedure to each element in the queue. The final benefit is the sum of all benefits. As such, the benefit of vertex i may entail changing the partitioning of several other vertices downstream. Since for each vertex, up to $|V|$ other vertices are considered, the complexity is $O(|V|^3|D|)$.
3) **G-full:** this technique resembles G-ext in that it considers the benefits for downstream vertices as well. However, instead of checking up to which vertex the repartitioning should be applied, it enforces all vertices on the paths from vertex i to a sink vertex in DG (i.e., a vertex with no outgoing edges) to get degree of parallelism equal to j. This aims to

3. It is important to recalculate the benefits from scratch rather than selecting the k highest benefits.

Algorithm 2 ILS-based specification of degree of partitioning

```
Require: $partitioning[], RTthreshold$
  for $i \leftarrow 1$ to $iter1$ do
    //tempPart[] holds a temp partitioning
    $tempPart[] \leftarrow$ perturb $partitioning[]$
    for $j \leftarrow 1$ to $iter2$ do
      // try local optimizations
      select a random vertex $v$
      select randon degree $n_v$
      if repartitioning $(v, n_v)$ improves $RT$ then
        $tempPart[v] \leftarrow n_v$
      end if
    end for
    if $RC(tempPart) < RC(partitioning)$ then
      if $RT(tempPart) < RTthreshold$ then
        $partitioning \leftarrow tempPart$
      end if
    end if
  end for
  return $partitioning[]$
```

minimize repartitioning costs. As previously, the complexity is $O(|V|^3|D|)$.

3.3.2 Iterated Local Search (ILS)

Our ILS algorithm is tailored to the specificities of our bi-objective problem and the types of Spark profiles presented in Section 2. It consists of two iterations as shown in the high-level description of Algorithm 2. The external loop, repeated $iter1$ times, directly targets the minimization of RC. The internal loop, repeated $iter2$ times, conforms to a different rationale. First, it is stochastic in that it randomly perturbs the current partitioning array. More specifically, m vertices are randomly changed to another degree of parallelism. Second, it tries to guide the partitioning of each vertex towards the area of its minimum RT, as a means to decrease both RC and RT. The latter is important so that further repartitionings can be investigated. As can be observed from Figure 2, this area may be towards the middle of the range $|D|$. In the following, we set $m = 3$, $iter1 = 100$ and $iter2 = 300$, while we evaluate the impact of other values of $iter1$ as well. The complexity of ILS is $O(iter1 iter2)$. As shown in the evaluation section, ILS exhibits a different behavior than the greedy techniques and is capable of trading RT for RC in an interesting manner.

4 EXPERIMENTS

We split the evaluation section in two parts. First, we examine a real Spark dataflow, which is capable of providing insights into the real-world benefits of our solutions. Then, in order to evaluate our solutions more systematically, we resort to synthetic flows that belong to a wide range of types and sizes; in these flows, the vertex profiles conform to the benchmarking applications of Section 2.

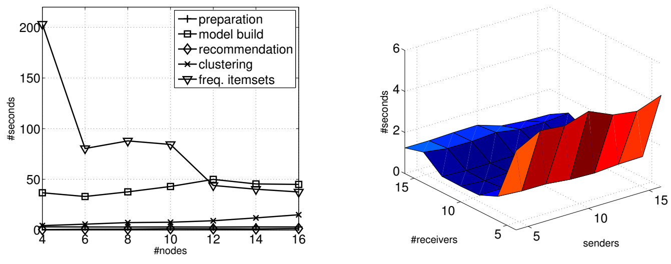


Fig. 7. Profiles for the steps (left) and repartitioning (right) of the case study

4.1 Real case-study

We consider a dataflow, which processes the Audioscrobbler dataset[4] in the following 5-step manner: i) it transforms the data accordingly in order to apply the next steps; (ii) it builds a collaborative filtering model; (iii) it produces recommendations for all users; (iv) it clusters the dataset in 12 clusters; and (v) it finds the frequent itemsets with support 0.05.[5] The profiles for these five steps and the potential repartitioning are shown in Figure 7 when the flow is executed on a cluster from 4 to 16 nodes on MN3.

Using all machines yields $RT = 106$secs and $RC = 1696$. Our solutions, for any $\varepsilon >$ yield as low running time as $RT = 84.61 secs$ and $RC = 845.8$. In other words, this is an example that both running time and consumption are decreased due to the specific form of the profiles. The former is decreased by 20% and the latter by more than 50%.

4.2 Synthetic Flows using Real Traces

For more thorough and systematic evaluation, we use a series of synthetic DAGs, where the behavior of each vertex is according to the real execution times, as presented in Section 2. We consider five types of DAGs in three sizes, as depicted in Figure 8.

The considered DAG types cover a wide range of flow execution plans. Type (A) refers to a flow in the form of a chain, thus covering applications that manipulate an initial dataset in a single way, which consists of several steps producing a single result dataset. Type (B) generalizes (A) in that the same dataset is subject to several analyses, and as such, the desired output comprises more than one result datasets. Both these types employ unary vertices, i.e., vertices with a single input edge. Type (C) extends type (A) through the support of multiple input datasets, and binary vertices, i.e., vertices with more than a single input. Type (B) and (C) have the form of a tree. Type (D) includes stages with three inputs and mesh graph structures. Finally, type (E) extends (B) through the consideration of generic DAGs rather than trees only. In addition, we consider 3 sizes for each type: small, medium and large, corresponding to DAGs with 5, 10, and 15 non-source vertices, respectively.

The behavior of each vertex is according to real execution data. More specifically, the profile of each vertex is a perturbed version of one of the profiles in Table 2. For each ver-

4. available from http://www-etud.iro.umontreal.ca/~bergstrj/audioscrobbler_data.html

5. Example implementation of the first three steps can be found at https://github.com/sryza/aas/tree/master/ch03-recommender

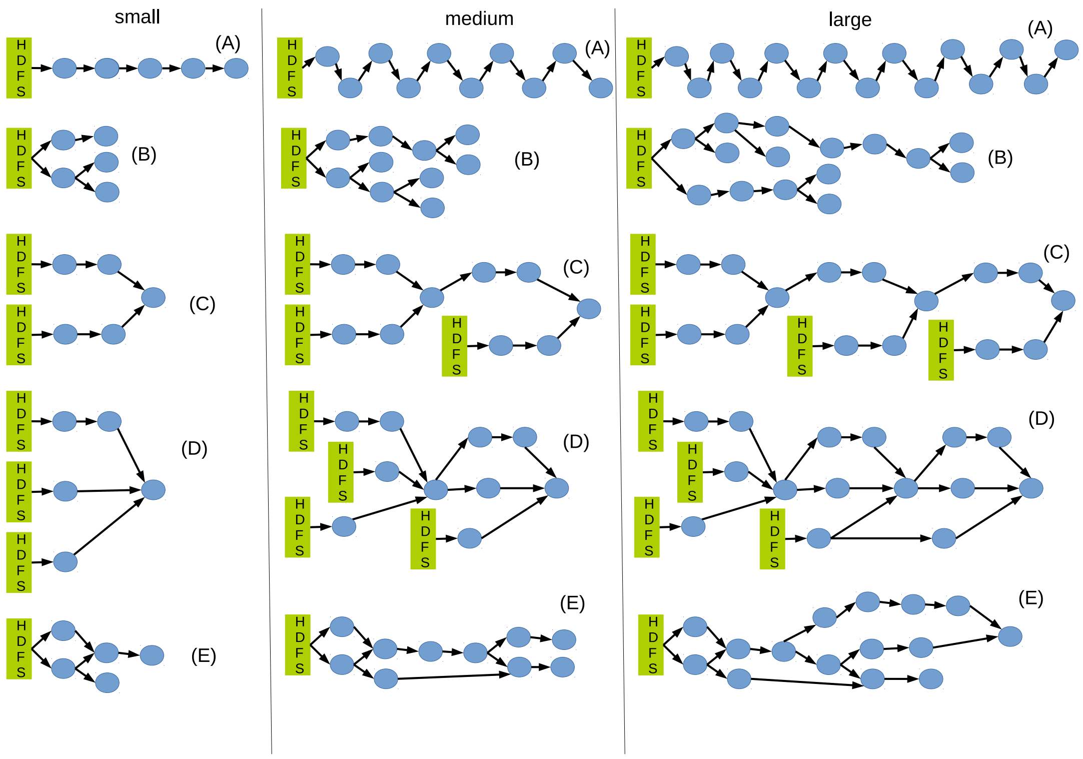


Fig. 8. DAGs considered in the experiments.

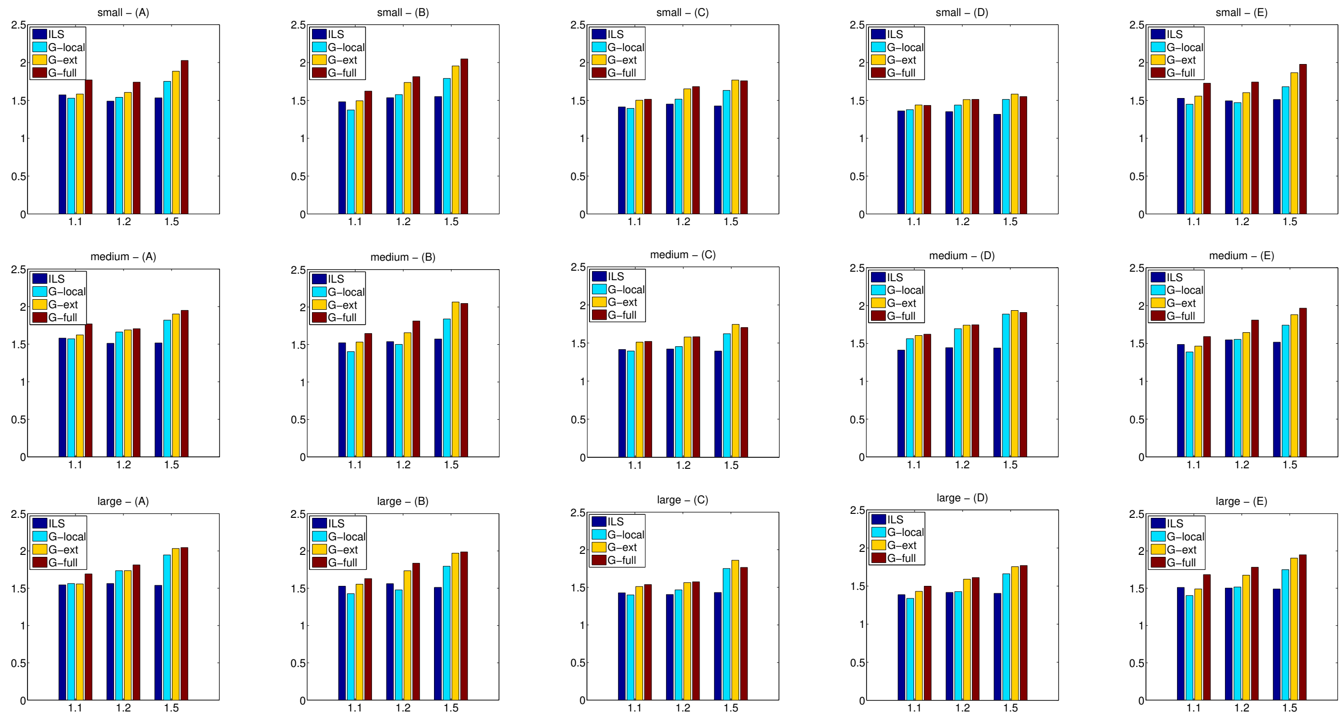


Fig. 9. Average number of times the initial resource consumption is higher than the optimized for $\varepsilon =$ 1.1, 1.2 and 1.5.

tex, first, one of the rows in the table is randomly selected, and then each parameter is randomly perturbed by $\pm 30\%$. Similarly, the cost of repartitioning is also based on the empirical measurements in Section 2, artificially perturbed by $\pm 30\%$ as well. In all experiments, the range of degree of parallelism is from 8 to 96 nodes and the initial datasets are partitioned in all 96 nodes without allowing their degree of parallelism to be modified. Based on these profiles, we expect that, in the generic case, resource consumption is minimized at the expense of an increase in running time (contrary to the case in the previous section).

TABLE 3
The highest improvements on the resource consumption compared to the initial

	best average			best maximum	
DAG	avg	stdev	techn.	max	techn.
$\varepsilon = 1.1$					
small - (A)	1.77	0.71	G-full	4.13	G-full
small - (B)	1.62	0.57	G-full	3.77	G-full
small - (C)	1.51	0.44	G-full	4.02	G-full
small - (D)	1.44	0.41	G-ext	3.14	G-full
small - (E)	1.73	0.65	G-full	3.73	G-full
medium - (A)	1.77	0.65	G-full	4.49	G-full
medium - (B)	1.64	0.42	G-full	3.18	G-ext
medium - (C)	1.52	0.37	G-full	2.99	G-full
medium - (D)	1.62	0.32	G-full	2.8	G-full
medium - (E)	1.59	0.42	G-full	2.95	G-full
large - (A)	1.69	0.53	G-full	3.32	G-full
large - (B)	1.63	0.37	G-full	3.11	G-full
large - (C)	1.54	0.31	G-full	2.6	G-ext
large - (D)	1.5	0.3	G-full	2.81	G-full
large - (E)	1.68	0.41	G-full	2.89	G-full
$\varepsilon = 1.2$					
small - (A)	1.74	0.71	G-full	4.98	G-full
small - (B)	1.81	0.69	G-full	5.23	G-full
small - (C)	1.68	0.74	G-full	5.96	G-full
small - (D)	1.51	0.37	G-full	2.72	G-full
small - (E)	1.74	0.71	G-full	5.18	G-full
medium - (A)	1.75	0.58	G-full	4.28	G-full
medium - (B)	1.81	0.59	G-full	4.03	G-full
medium - (C)	1.58	0.38	G-full	2.95	G-ext
medium - (D)	1.75	0.34	G-full	2.74	G-full
medium - (E)	1.81	0.62	G-full	4.96	G-full
large - (A)	1.81	0.48	G-full	3.17	G-full
large - (B)	1.84	0.47	G-full	3.91	G-full
large - (C)	1.57	0.3	G-full	2.62	G-ext
large - (D)	1.61	0.33	G-full	2.65	G-full
large - (E)	1.78	0.49	G-full	3.6	G-full
$\varepsilon = 1.5$					
small - (A)	2.03	0.9	G-full	6.08	G-full
small - (B)	2.05	0.91	G-full	4.95	G-full
small - (C)	1.77	0.68	G-ext	3.25	G-full
small - (D)	1.58	0.4	G-ext	2.98	G-full
small - (E)	1.98	0.8	G-full	5.24	G-full
medium - (A)	1.95	0.7	G-full	5.17	G-full
medium - (B)	2.07	0.64	G-ext	4.0	G-full
medium - (C)	1.74	0.4	G-ext	3.88	G-full
medium - (D)	1.94	0.41	G-ext	3.24	G-ext
medium - (E)	1.97	0.65	G-full	3.81	G-full
large - (A)	2.04	0.78	G-full	6.77	G-full
large - (B)	1.99	0.55	G-full	3.67	G-full
large - (C)	1.86	0.37	G-ext	4.4	G-full
large - (D)	1.77	0.36	G-full	2.67	G-full
large - (E)	1.95	0.5	G-full	3.62	G-full

4.2.1 *Results*

In the first set of experiments, we initially assess the efficiency of the techniques when ε=1.1, 1.2, and 1.5. Each combination of i) DAG type, ii) DAG size and iii) technique is repeated 100 times. Each time, the vertex and edge costs are randomly instantiated as explained above.

Figure 9 shows how many times the initial RC is higher than the optimized on average. The initial RC is the one when no repartitioning takes place throughout execution. The information in the figure is complemented by Table 3, where, for each combination of DAG type and size, the number of times the initial RC is higher than the RC of best performing technique is presented. For the latter, both the average best performance and the best performance in isolated runs is considered.

The main observations can be summarized as follows:

1) The improvements on RC due to the proposed techniques are significant. Even for small values of ε, such as $\varepsilon = 1.1$, the average RC of the initial execution is at least 1.44 times higher than the RC of our solutions and it can reach up to 1.77 times. For larger values of ε, $\varepsilon = 1.5$ the initial RC can more than 2 times higher, which means that our techniques are capable of dropping the resource consumption to the half. The improvements in isolated cases, as shown in the second column of Table 3 from right, are even more significant. In certain cases, e.g., for DAGs of type "large - (A)", the improvements of RC are by a factor of more than 6 times.
2) G-full and G-ext fully dominate the randomized solution ILS and G-local in terms of RC. The low efficiency of ILS is somewhat of a surprise, given the good behavior of similar solutions to the problems that tackle a single objective as in [12].
3) On average, G-full is more efficient than G-ext. As shown in the figure, G-ext is superior in DAG types (C) and (D) of large size, but by a small margin compared to G-full. To the contrary, in the cases where G-full is superior, which are far more frequent, the margin is wider.
4) Counter-intuitively, the size of the DAG seems to play a minor role in the efficiency of the techniques. This is more evident with the help of Figure 10, where the behavior of G-full is shown with a different grouping than the one in Figure 9. From the figure, we can also observe that the type of the DAG significantly impacts on the RC improvement

We close the discussion of this experiment with a look at the RT (see Figure 11). Combining the results about the increase of RT and RC reveals a distinctive feature of the ILS technique: ILS is capable of decreasing the RC by at least 30% on average at the expense of small, if not negligible, increase in RT. In other words, the trade-off between RC and RT achieved by ILS is of a very different nature than that of G-full. More specifically, the highest average increase in RT is for type (E) of medium and large size and is approximately 5%. For the remainder DAGs it is even less, and in certain cases, such as medium-(D) and small-(C) ILS manages to decrease both RT and RC. For the G-local, G-ext and G-full techniques, the increase in RT is closely defined by ε, as expected (with an exception for small-(C), where all techniques improve RT as well).

In light of the above analysis, we do not consider that G-full and G-ext dominate ILS in general, since the latter exhibits a different type of behavior with an interesting trade-off between RC and RT. Finally, this analysis explains why the ILS bars in Figure 9 are similar for a given combination of DAG type and size regardless of the value of ε.

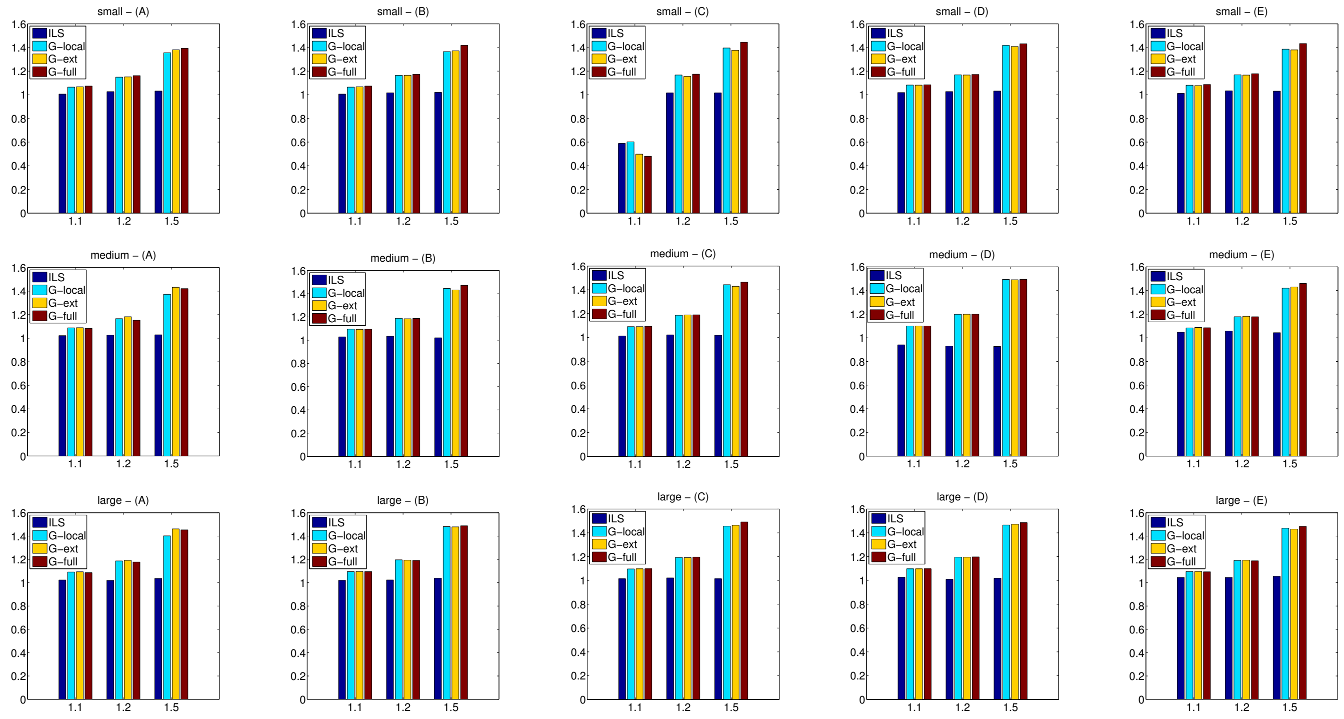


Fig. 11. Actual increase factor of running time for $\varepsilon = 1.1$, 1.2 and 1.5.

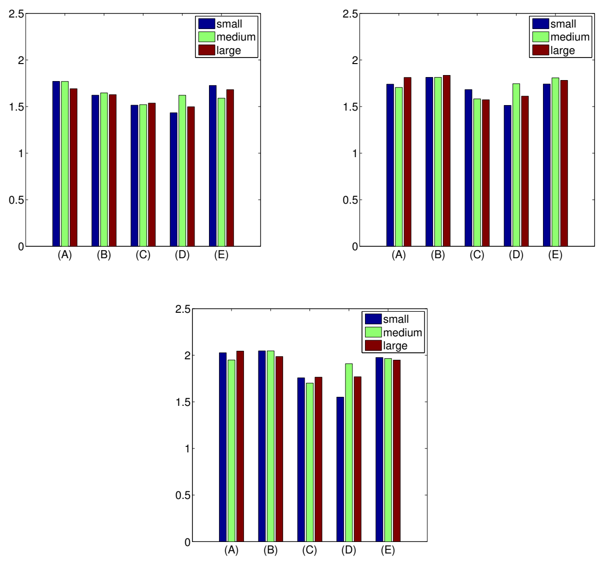


Fig. 10. The behavior of G-full for $\varepsilon = 1.1$(top left), 1.2 (top right) and 1.5(bottom).

TABLE 4
The highest improvements on the resource consumption compared to the flow with optimized RT

	best average vs.		best maximum vs.	
DAG	initial	best RT	initial	best RT
$\varepsilon = 1.1$				
small - (A)	1.77	1.24	4.13	1.56
small - (C)	1.51	1.21	4.02	1.58
medium - (A)	1.77	1.18	4.49	1.46
medium - (C)	1.52	1.19	2.99	1.43
large - (A)	1.69	1.18	3.32	1.36
large - (C)	1.54	1.18	2.6	1.39
$\varepsilon = 1.2$				
small - (A)	1.74	1.36	4.98	1.94
small - (C)	1.68	1.25	5.96	1.71
medium - (A)	1.75	1.28	4.28	1.68
medium - (C)	1.58	1.25	2.95	1.57
large - (A)	1.81	1.26	3.17	1.75
large - (C)	1.57	1.25	2.62	1.54
$\varepsilon = 1.5$				
small - (A)	2.03	1.46	6.08	2.66
small - (C)	1.77	1.4	3.25	2.12
medium - (A)	1.95	1.47	5.17	2.46
medium - (C)	1.74	1.38	3.88	1.72
large - (A)	2.04	1.44	6.77	1.95
large - (C)	1.86	1.42	4.4	1.79

4.2.2 Results After Optimizing for RT

In the second set of experiments, we extend the problem statement and we assume that we first find the partitioning of each vertex so that RT is minimized, before trying to minimize RC under a bounded increase in RT. To this end, we combine the rationale of [12] with the techniques in Section 3. The expectation is that both the initial RT and RC will improve. The improvements on RT are due to the concave shape of the vertex profiles, which imply that executing on the maximum number of nodes available does not necessarily imply faster execution. For many cases, this benefit outweighs the overhead of repartitioning, so overall, the average RT decreases. In addition, RC decreases because not all nodes are used.

In order to minimize RT, we employ the dynamic programming algorithm and the heuristics in [12]. Moreover, on top of their initial solutions, as a meta-optimization step, we run a modified version of the ILS technique. The modifications are such so that the aim is to minimize RT rather than RC under a constraint on RT. As expected, RT and RC both decrease. For example, for the (A) type of DAG, regardless of the size, RT drops by approximately 5% and RC drops by 20-25%. For type C, the decrease in RT is similar, but the decrease in RC is slightly less, 18-20%.

The important notice is that our techniques are capable of yielding significant improvements on RC even when applied to these optimized partitionings. The general trend of each technique remains the same as in the previous experiment. Table 4 summarizes the results of the best performing technique, which in almost all cases is G-full (for brevity, only DAG types (A) and (C) are presented).

From the table, we can see that when allowing up to 10% increase in RT the benefits of RC are 18-24% for type (A) and 18-21 for type (C). For $\varepsilon = 1.2$, these benefits increase to 26-36% and 25%, respectively. When $\varepsilon = 1.5$, they become 44-48% and 38-42%, respectively. Table 4 also presents the maximum observed improvements in any of the 100 repeats of each combination of technique and DAG type and size.

4.2.3 Impact of Profile Inaccuracy

Given that our techniques require detailed profiling information, a question may arise as to whether our techniques are sensitive to metadata inaccuracies. To investigate this issue, we run a new set of experiments, where the actual profile parameters are further perturbed by a factor of $\pm 30\%$. These perturbed values are unknown to the optimization techniques, and they are used only for the computation of the RT and RC values of the final solution. According to the results, there are negligible changes in all the numbers presented above. The maximum deviations observed do not exceed $\pm 1.2\%$. The reason is that the perturbations are of such a magnitude that the relative differences between the coefficients, which differ by orders of magnitude, are affected to a small extent only.

4.3 Optimization Overhead

The optimization overhead of our solutions is low. For example, optimizing the large DAG of type (C) on a machine with i7-4770 CPU at 3.4GHz with 16GB of RAM using the Greedy techniques takes less than 0.2 secs. ILS is slower and takes 2.55 secs on average. Increasing the number of iterations in ILS 3 times results in time overhead of 8.18 secs. Based on the above overhead information, it can be concluded that the number of external iterations of ILS has a tangible impact on optimization time. In Figure 12, we show 5 example runs for ILS. We can observe that, after 100 external iterations, the decrease rate in RC is significantly lower than in the first iterations.

5 Discussion on the provision of end-to-end solutions

Thus far, we have explained what is the type of the profiles of Spark applications in terms of analytical cost models (Section 2) and we provided solutions for deciding on the degree of partitioning (Section 3) along with experimental evidence regarding their efficiency (Section 4). These two modules are essential for building practical end-to-end solutions, but arguably need to be coupled with techniques that are capable of deriving the profiles in an efficient manner.

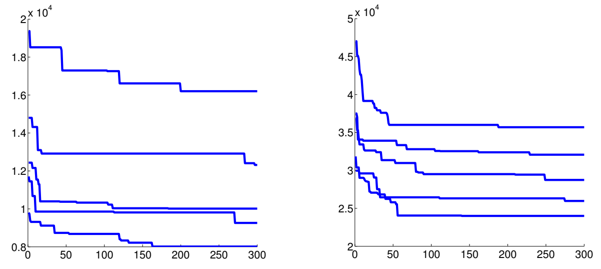


Fig. 12. Resource consumption as a function of the number of ILS external iterations in 5 example runs for small-(A)(left) and large-(C)(right).

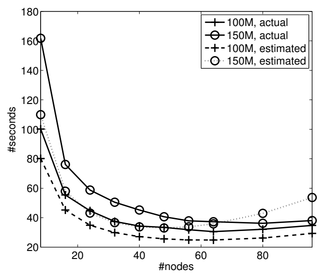


Fig. 13. Sampling-based profile estimation.

A natural choice to this end is to employ sampling, e.g., as in [13], [14]. However, sampling-based automated profile generation seems to be a particularly challenging task in Spark. Consider the case in Figure 13, where we profile *k-means* using exactly the same type of data in three sizes. The sample is 50M records, and we want to use it to estimate the profiles for datasets 2 and 3 times larger (i.e., the sample is relatively large). Although this seems to be an easy thing to do, the results are poor. In the figure, we show how much the estimates deviate from the actual profiles if we employ the template in Eq. (4) and we multiply the coefficients b, c and d of the sample, accordingly. The deviation can reach several factors. We have investigated several ways to tune the coefficients. The one shown in the figure is the following: for a dataset n times larger, the multiplication factors are n, n and $nlog(n)$, respectively. Other tuning ways behave even worse. The main lesson learnt is that the problem of inexpensively deriving the necessary profiles is challenging and there seems to be no trivial way to transfer the results of smaller datasets (i.e., samples) and/or applications to larger or different datasets and applications. This constitutes a main direction for future work.

6 Related Work

The closest work to us is the work in the context of the Cumulon system [15]. Cumulon extends Hadoop MapReduce and focuses on matrix-based analyses, which are common in many scientific problems. It tries to minimize the monetary cost when renting cloud virtual machines while respecting a constraint on execution time. Micro-benchmarking is used to obtain performance profiles for each task. However, the main differences, which do not allow us to transfer their techniques to our problem, are that the degree of partitioning is exhaustively searched for each vertex, which incurs unacceptably high overhead, and no repartitioning cost is considered. Enumerating over all combinations is also employed in [16]. The proposal in [17] targets a MapReduce environment and derives cost formulas for each vertex as a function of map and reduce slots. It then finds the minimum number of slots that are required to meet a running time constraint using Lagrange multipliers. Extensions have appeared in [18], where trade-off curves between running time and monetary cost are provided to the user, who makes the final choice, and in [19], where the number of map and reduce slots are optimally decided. All these techniques are specific to a MapReduce setting running on cloud machines and assume an analytical cost model that is even simpler than the one in [7], which is extended by our work.

A shortened single-objective version of our problem has been investigated in the past under several descriptions. In [12], [20], the problem of changing the type of the execution engine for each task while taking into account engine switching costs is tackled. Even if we treat different execution engines as different degrees of partitioning, the solutions in [12], [20] are inadequate for our bi-objective problem, while also the solutions in [20] cannot scale. Also, there are several techniques that try to allocate a DAG to the appropriate number of resources, but, typically, they do not consider any aspect that can correspond to repartitioning overhead, e.g., [21], [22]. Bi-objective allocation in cloud settings have been considered in several other works, but making different assumptions, e.g., when clients are potentially antagonistic to the provider and jobs consist of standalone tasks as in [23], or when there are multiple candidate cloud infrastructures, as in [24].

Regarding profiling, the work in [6], provides evidence about the importance of mitigating CPU bottlenecks in Spark applications and proposes a methodology to detect them. An analytic model for MapReduce tasks to compute latencies is presented in [25]. [26] proposes a stochastic cost model for generic workflow tasks but does not consider different degrees of parallelism. Finally, [27] proposes a technique to predict the number of resources in scientific workflows in order to obtain a desirable level of performance. However, all these cost modeling and profiling techniques do not cover specific phenomena in Spark execution, such as super-linear speed-ups for small degrees of parallelism and performance degradation for large ones. The proposals in [13], [14] present a sampling-based approach to estimate the profile of a single embarrassingly parallel task, based on the behavior of some of its partitions. However, they assume that partitions are scheduled in multiple waves, whereas we have adopted a configuration, where all partitions are scheduled in a single wave but there are multiple interdependent tasks. In [28], MapReduce profiles are transferred from one cluster to another, but still, extensive benchmarking on both clusters and detailed profiles on one cluster are required. [16] leverages past traces to estimate the profiles of new jobs based on the similarity of job profiles. The techniques in [29], [30], [31] estimate the behavior of reduce tasks when changing the degree of parallelism with the help of a FIFO scheduler; however, this approach cannot reflect the execution complex Spark jobs with internal shuffling at a reasonable accuracy. In general, advances in profiling and instantiating the cost models of spark applications at runtime are orthogonal to our dynamic partitioning techniques.

7 Conclusions and Future Research

In this work, we investigate the problem of resource waste when executing a Spark application on all the machines available. To solve this, we propose solutions that modify the degree of data partitioning and parallelism during execution, so that we decrease the amount of time resources are occupied while bounding any increase in execution time. We also perform a detail analysis of the running time of simple spark applications under different degrees of parallelism and we derive analytical formulas that describe the application performance. Using real data from the profiling activity, we evaluate our dynamic partitioning solutions. The results show that we can trade small amounts of running time for big reductions in resource consumption, e.g., by a factor more than 6X.

This work can also be extended in several directions, which call for further research. It assumes a standalone setting of Spark cluster manager, as the one in the Barcelona Supercomputing Center. The impact of additional cluster managers, such as YARN and MESOS, needs to be investigated as well. Also, the trend in Spark program development is to depart from RDDs and work on top of Dataframes and Datasets (as from Spark v.1.6). Repartitioning and scalability behavior on top of these interfaces needs to be explored.[6] However, the main problem according to our view is the need of application profiles in order to run our dynamic partitioning techniques. Deriving reasonably accurate profiles with as few test runs as possible is an important avenue for future work.

Acknowledgment. We would like to thank Mrs E. Katsiantoni, student at the Aristotle Univ. of Thessaloniki, for helping with initial versions of the ILS algorithm code. Also, this work is partially supported by the Spanish Ministry of Economy and Competitivity under contract TIN2015-65316-P and by the SGR programme (2014-SGR-1051) of the Catalan Government.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," ser. HotCloud'10.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012, pp. 15–28.

[3] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Sql and rich analytics at scale," in *SIGMOD*, 2013, pp. 13–24.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: relational data processing in spark," in *SIGMOD*, 2015, pp. 1383–1394.

[5] R. Tous, A. Gounaris, C. Tripiana, J. Torres, S. Girona, E. Ayguade, J. Labarta, Y. Becerra, D. Carrera, and M. Valero, "Spark deployment and performance evaluation on the marenostrum supercomputer," in *IEEE BigData*, 2015, pp. 299–306.

6. At the time of writing, these interfaces suffer from certain technical limitations regarding the support for dynamic partitioning, e.g., https://issues.apache.org/jira/browse/SPARK-9872.

[6] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun, "Making sense of performance in data analytics frameworks," in *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2015, pp. 293–307.

[7] N. J. Gunther, P. Puglia, and K. Tomasette, "Hadoop superlinear scalability," *Commun. ACM*, vol. 58, no. 4, pp. 46–55, 2015.

[8] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, "Hadoop's adolescence," *PVLDB*, vol. 6, no. 10, pp. 853–864, 2013.

[9] B. Graham and M. R. Rangaswami, "Do you hadoop? a survey of big data practitioners," Tech. Rep., 2013.

[10] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Çetintemel, and S. Zdonik, "An architecture for compiling udf-centric workflows," *PVLDB*, vol. 8, no. 12, pp. 1466–1477, 2015.

[11] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas, "Nobody ever got fired for using hadoop on a cluster," in *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, ser. HotCDP '12, 2012, pp. 2:1–2:5.

[12] G. Kougka, A. Gounaris, and K. Tsichlas, "Practical algorithms for execution engine selection in data flows," *Future Generation Computer Systems*, vol. 45, no. 0, pp. 133 – 148, 2015.

[13] M. Sahli, E. Mansour, T. Alturkestani, and P. Kalnis, "Automatic tuning of bag-of-tasks applications," in *31st IEEE International Conference on Data Engineering, ICDE*, 2015, pp. 843–854.

[14] M. Sahli, E. Mansour, and P. Kalnis, "ACME: A scalable parallel system for extracting frequent patterns from a very long sequence," *VLDB J.*, vol. 23, no. 6, pp. 871–893, 2014.

[15] B. Huang, S. Babu, and J. Yang, "Cumulon: optimizing statistical data analysis in the cloud," in *SIGMOD Conference*, 2013, pp. 1–12.

[16] J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang, "MRTuner: A toolkit to enable holistic optimization for mapreduce jobs," *PVLDB*, vol. 7, no. 13, pp. 1319–1330, 2014.

[17] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: automatic resource inference and allocation for mapreduce environments," in *Proc. of the 8th International Conference on Autonomic Computing, ICAC*, 2011, pp. 235–244.

[18] Z. Zhang, L. Cherkasova, and B. T. Loo, "Optimizing cost and performance trade-offs for mapreduce job processing in the cloud," in *2014 IEEE Network Operations and Management Symposium, NOMS*, 2014, pp. 1–8.

[19] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo, "Automated profiling and resource management of pig programs for meeting service level objectives," in *9th International Conference on Autonomic Computing, ICAC'12*, 2012, pp. 53–62.

[20] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal, "Optimizing analytic data flows for multiple execution engines," in *SIGMOD Conference*, 2012, pp. 829–840.

[21] M. Rahman, M. R. Hassan, R. Ranjan, and R. Buyya, "Adaptive workflow scheduling for dynamic grid and cloud computing environment," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 13, pp. 1816–1842, 2013.

[22] R. Duan, R. Prodan, and T. Fahringer, "Performance and cost optimization for multiple large-scale grid workflow applications," in *Proc. of the ACM/IEEE Conf. on Supercomputing*, 2007, pp. 1–12.

[23] J. Simao and L. Veiga, "Partial utility-driven scheduling for flexible sla and pricing arbitration in clouds," *IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1–1, 2016.

[24] L. Sharifi, L. Cerdà-Alabern, F. Freitag, and L. Veiga, "Energy efficient cloud service provisioning: Keeping data center granularity in perspective," *J. Grid Comput.*, vol. 14, no. 2, pp. 299–325, 2016.

[25] B. Li, Y. Diao, and P. J. Shenoy, "Supporting scalable analytics with latency constraints," *PVLDB*, vol. 8, no. 11, pp. 1166–1177, 2015.

[26] A. M. Chirkin, A. Belloum, S. V. Kovalchuk, and M. X. Makkes, "Execution time estimation for workflow scheduling," in *Proceedings of the 9th Workshop on Workflows in Support of Large-Scale Science*. IEEE Press, 2014, pp. 1–10.

[27] I. Pietri, G. Juve, E. Deelman, and R. Sakellariou, "A performance model to estimate execution time of scientific workflows on the cloud," in *Proceedings of the 9th Workshop on Workflows in Support of Large-Scale Science*. IEEE Press, 2014, pp. 11–19.

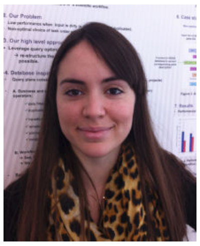
[28] A. Verma, L. Cherkasova, and R. H. Campbell, "Profiling and evaluating hardware choices for mapreduce environments: An application-aware approach," *Perform. Eval.*, vol. 79, pp. 328–344, 2014.

[29] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of mapreduce programs," *PVLDB*, vol. 4, no. 11, pp. 1111–1122, 2011.

[30] H. Herodotou, F. Dong, and S. Babu, "No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ser. SOCC '11, 2011, pp. 18:1–18:14.

[31] H. Lim, H. Herodotou, and S. Babu, "Stubby: A transformation-based optimizer for mapreduce workflows," *PVLDB*, vol. 5, no. 11, pp. 1196–1207, 2012.

Anastasios Gounaris is an assistant professor at the Dept. of Informatics of the Aristotle University of Thessaloniki, Greece. A. Gounaris received his PhD from the University of Manchester (UK) in 2005. His research interests are in the areas of autonomic, adaptive and wide-area data management, massive parallelism, flow and query optimization, data mining and resource scheduling. More details can be found at http://delab.csd.auth.gr/~gounaris/.



Georgia Kougka is a PhD candidate in the Informatics Dept. of the Aristotle University of Thessaloniki since 2012. Her Ph.D. project has been partially funded by the Thales Research Funding Program of the Hellenic General Secretariat for Research and Technology. Her research interests lie in the field of data flow optimization, flow task allocation and multi-objective flow optimization. More details can be found at http://delab.csd.auth.gr/~georkoug/.

Rubèn Tous is an associate professor at the Dept. of Computer Architecture of UPC. He has also been participating as spanish delegate in ISO/MPEG and ISO/JPEG. He is editor of Part 12 of the MPEG-7 standard and Parts 1, 3 and 6 of the JPEG's JPSearch standard. His main research interests are multimedia big data computing, multimedia information retrieval, knowledge representation, machine learning and computer vision.

Carlos Tripiana Montes holds a MSc in Computer Science working as permanent staff in the HPC User Support Group of the Operations Department at the Barcelona Supercomputing Center since 2010. His expertise comprises Computer Graphics, Data Processing, Visualization, and Big Data, all focused on HPC ecosystems.

Jordi Torres holds a PhD from UPC (Best UPC Computer Science Thesis Award, 1993). Currently he is a full professor in the Computer Architecture Department at UPC. He has more than twenty years of experience in research and development of advanced distributed and parallel systems in the High Performance Computing Group at UPC. More details can be found at http://www.jorditorres.org/.