A fast solution for bi-objective traffic minimization in geo-distributed data flows

Anna-Valentini Michailidou
Aristotle University of Thessaloniki
Greece
annavalen@csd.auth.gr

Anastasios Gounaris
Aristotle University of Thessaloniki
Greece
gounaria@csd.auth.gr

ABSTRACT

Geo-distributed analytics is becoming an increasingly commonplace as IoT, fog computing and big data processing platforms are nowadays integrating with each other. In this work, we deal with a problem encountered when complex Spark workflows run on top of geographically dispersed nodes, either data centers or individual machines. There have been proposals that optimize the execution of such workflows in terms of the aggregate traffic generated or the latency (which is due to data transmission), or both metrics. However, the state-of-the-art solutions that target both objectives are either significantly sub-optimal or suffer from high optimization overhead. In this work, we address this limitation. The main solutions that we propose are both efficient and effective; based on either the extremal optimization or the greedy algorithm design paradigm, they can yield significant improvements having an optimization overhead of a few tens of seconds even for Spark workflows of 15 stages running on 15 distributed nodes. We also show the inadequacy of evolutionary optimization solutions, such as genetic algorithms, for our problem.

CCS CONCEPTS

• **Information systems → Query optimization**; • **Computer systems organization → Distributed architectures**;

KEYWORDS

data flows, workflows, Spark, optimization

ACM Reference Format:
Anna-Valentini Michailidou and Anastasios Gounaris. 2019. A fast solution for bi-objective traffic minimization in geo-distributed data flows. In *23rd International Database Engineering & Applications Symposium (IDEAS'19), June 10–12, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3331076.3331107

1 INTRODUCTION

Geo-distributed analytics, such as fog computing solutions [1, 22], is an emerging area boosted by the maturity of big data analytics platforms supporting data streams, e.g., Flink [8] and Spark [2], along with the prevalence of IoT devices in modern applications [5, 7, 18]. The execution model builds upon and extends the one in distributed [21] and parallel [9] databases. In short, the execution plan is typically a directed acyclic graph of operators and benefits from the main types of query plan parallelism, namely partitioned, pipelined and independent.

In this work, we consider Spark running over separate physical nodes with distinct data transmission capacities; as reported in [10], the applications of such a setting span several fields, such as climate science, multinational companies, bio-informatics and log analysis. Spark execution plan inherently benefits from pipelined and partitioned parallelism [3] with the underlying cluster management layers, e.g., YARN, Mesos and so on, being responsible for the actual runtime task scheduling. A typical assumption is that the cluster on which the execution runs is characterized by abundant memory and fast node interconnection speeds, and the whole processing takes place in a single geographical area. However, this assumption becomes a limitation, when the data to be processed are physically stored in multiple places and/or processing needs to occur close to the data source. To overcome this limitation, several geo-distribution-aware extensions to MapReduce-based solutions have been proposed [10].

Optimization techniques for geo-distributed Spark execution plans directly affect the manner partitioned parallelism is enforced through specifying the portion of the tasks in each Spark stage that each processing node should become responsible for. Current techniques to this end aim to minimize either the total traffic between the nodes, e.g.,[27], or the latency, e.g. [23]. In a recent previous proposal of ours, we present bi-objective solutions that target both criteria [19]. The proposal in [19] challenges the validity of a main motivation behind geo-distributed data flows, namely that it is too costly to gather data in a single place, e.g., [10, 16, 28], and is tailored to multi-stage workflows rather than simple two-stage MapReduce ones. It comprises two techniques: a greedy one that is fast but not very effective in terms of the quality of the derived solutions and another one based on iterated local search that is effective but takes longer time, in the order of couple of minutes, to compute the proposed task distribution.

In this work, we make a twofold contribution. Firstly, we combine the best attributes of the solutions mentioned above. The main bi-objective solutions that we propose are capable of running much faster that the best performing one in [19] and still yield significant improvements over the main competitor, as evidenced by the results of a thorough evaluation. The solution is based on either the extremal optimization (EO) paradigm [4, 6] or the greedy algorithm design strategy but in a less shortsighted than in [19]. Secondly, also in light of the well-known *No Free Lunch* theorem [29], it is important to find which optimization paradigm fits better to

IDEAS'19, June 10–12, 2019, Athens, Greece

ACM ISBN 978-1-4503-6249-8/19/06…$15.00
https://doi.org/10.1145/3331076.3331107

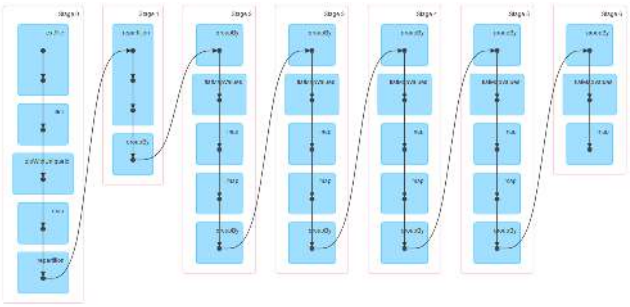


Figure 1: A real Spark DAG

our specific problem; to this end, we show that evolutionary optimization solutions, such as genetic algorithms, are inferior to the solutions we propose hereby.

Paper structure. The remainder of the paper is structured as follows. In the next section, we give a motivation scenario. In Section 3, we give the formal problem definition and we outline the solutions in [19] to make this work self-contained. We present our new solution in Section 4. The evaluation aims to cover a wide range of scenarios and is presented in Section 5. We conclude with the discussion of the related work and the open issues in Sections 6 and 7, respectively.

2 A MOTIVATION EXAMPLE

Our work is highly inspired by performance issues in modern data analytics platforms, such as Spark, which is arguably the most-widespread framework for data-intensive cluster computing to date. The distinctive feature of our work is that we do not assume a centralized, homogeneous setting; on the contrary we consider that a cluster may consist of physical machines that are geo-distributed, have heterogeneous uplink and downlink speed capacities, and communicate through sending data across a network in order to complete an application. Our algorithms can make such applications run faster by offering a task placement plan that minimizes the data transfer over the network, while we consider both of these two objectives. Next, we showcase how our algorithms, namely Greedy-full and Extremal to be presented in Section 4, can improve a Spark application.

In a geo-distributed setting, it is reasonable to assume that data transmission is the dominant factor for the application latency. Focusing on the data transmission capacities, we employ three machines (noted as M1, M2 and M3 in Table 1) with uplink speeds of 5 MB/sec, 2 MB/sec, and 5 MB/sec, respectively. The downlink speeds are 5, 3 and 2 MB/sec, respectively. The execution plan of the application we try to optimize, in the form of a Directed Acyclic Graph (DAG), is a linear one, as shown in Figure 1. Each node (bounded rectangle in the figure) is a stage that consists of tasks, the placement of which is decided by our algorithms. The edges between the nodes represent the data movement between the stages. The overall input is set to 287.6 MB and the selectivity between the stages is always equal to 1; i.e., the total amount of data being reshuffled and flowing across the stages remains the same.

We first compute the task allocation offline and then enforce the task allocation in Spark. Then, we compare the estimated running time reduction with the actual one. The offline computation ignores the CPU overhead that a real setting has even when transmitting data [20] and thus the time it refers to is only the overhead of moving data. Note that the data movement reduction is the same in the offline computation and the real run. Table 1 shows the allocations decided by each algorithm, namely Iridium [23], our main competitor, *Extremal* and *Greedy-full* (i.e., the contributions of this work) for each stage and for each machine. Note that for the first two stages of Figure 1, we do not choose a new placement because we assume that the initial data placement, on which the task placement of the first two stages depends, is fixed; these two stages just read and parallelize the initial dataset evenly. Thus we start from the task placement of Stage 2. With these new task allocations, Extremal is estimated to achieve a 50.5% reduction in the running time over Iridium, while Greedy-full achieves 9.86% reduction. The real reduction achieved in the Spark environment was 47.75% and 13% respectively, indicating that our algorithms can indeed reduce the running time of a real application in Spark. More importantly, the amount of data transmitted over the network drops by 74.97% due to Extremal (from 799.7 to 200.1MBs) and by 25.3% due to Greedy-full (from 799.7 to 597.3MBs).

Implementation Details. In order to enforce our task placement in the Spark engine, we have rebuilt Apache Spark 2.3.2 with the following changes; we override the `TaskSchedulerImpl` class, where we disable the shuffling of the offers the executors make for a task and we edit the `TaskSetManager` class to set the task locality to ``Any'' and thus prevent Spark from deciding a placement for the tasks based on the data location. Finally, we can easily emulate a geo-distributed setting with machines characterized by different downlink and uplink speeds by using machines connected to a local network and set the bandwidth limits of the executors using a tool, such as the *Wonder Shaper script*[1]

3 BACKGROUND

We first present the problem statement, which is kept the same as in [19], and then we present the two existing solutions, the strong points of which we combine in this work. The problem is stated in a system-agnostic manner; i.e., it is not applicable to Spark solely.

3.1 Problem Statement

A geo-distributed data flow is represented as a DAG $G(V, E)$. Each node $v_j \in V$, where $j = 1 \ldots N$ and $N = |V|$, represents a *job* and each edge represents a shuffle data movement between the jobs. For example, in Spark data flows, we consider a job to be a *Spark stage* (note that Spark uses the terminology job to refer to a set of stages); in between such stages, data shuffling takes place. Each job runs in parallel in M data centers (DCs): i.e., each DC becomes responsible for a fraction of the job execution with the magnitude of the fraction devised by our algorithms. DCs generalize the notion of physical machines used in the motivation example.

Conceptually, the workload of a job is split into small units of work, each allocated to a specific processing element, e.g., a multicore server of a specific DC, as an atomic unit. We refer to these

[1] available from https://github.com/magnific0/wondershaper

Table 1: Task placement decision (proportion of tasks allocated) of Iridium, Extremal and Greedy-full for each stage and machine

Algorithm/Stage-Machine	stage 2			stage 3			stage 4			stage 5			stage 6		
	M1	M2	M3	M1	M2	M3	M1	M2	M3	M1	M2	M3	M1	M2	M3
Iridium	0.286	0.429	0.285	0.188	0.529	0.283	0.098	0.628	0.274	0.038	0.717	0.245	0.208	0.792	0.0
Extremal	0.0041	0.993	0.0029	0.0	0.995	0.005	0.003	0.997	0.0	0.002	0.998	0.0	0.001	0.999	0.0
Greedy-full	0.333	0.477	0.19	0.116	0.606	0.278	0.032	0.706	0.262	0.0	1.0	0.0	0.0	1.0	0.0

splits as *tasks*. Due to shuffling, in the generic case, it is necessary to move data between DCs before the execution of each task. This data movement is the dominant factor regarding the running time of the jobs, while the actual execution time of the job is considered to be negligible.

In this work, we deal with the allocation of sets of tasks to each DC for each job. Let I^j be the input dataset size of v_j. If the selectivity of the job is a^j, then the output dataset is of size $S^j = a^j * I^j$; the job selectivity is defined as the ratio of its output to input size. If v_j has outgoing edges in G, S^j is divided into M parts to be sent to the jobs downstream, denoted by $r_i^j S^j,\ i = 1 \ldots M$, s.t. $\sum r_i^j = 1$. Essentially, r_i^j corresponds to the fraction of tasks of the *children nodes* of v_j assigned to the i^{th} DC (tasks are assumed to be infinitesimally divisible). In other words, r_i^j values affect the workload allocation of jobs v_k, where $(j,k) \in E$. Overall, each DC has to transfer a fraction of $(1 - r_i^j)$ of its local output data S_i^j, and to receive a total of $r_i^j * (S^j - S_i^j)$ data from all the other DCs.[2] Following the rationale in [23], we specify the uplink (resp. downlink) bandwidth of the i^{th} DC as U_i (resp. D_i). Table 2 summarizes the main notation.

Based on the above, the time for a site to send data regarding the output of a job is $TU_i^j = (1 - r_i^j) * S_i^j / U_i$, and the time to receive data is $TD_i^j = r_i^j * (S^j - S_i^j)/D_i$. The running time RT_j of v_j is $max\{TU_i^j, TD_i^j\}$.

The total data movement from a node v_j is equal to $DM_j = \sum_{i=1}^{M} (1 - r_i^j) * S_i^j$. The total data movement is $DM(G) = \sum_{j=1}^{N} DM_j$, where v_j has at least one outgoing edge.

The running time of a G, $RT(G)$ is the maximum sum of RT_j values across any path from a source job (v_j without incoming edges) to a sink one (v_j without outgoing edges); sink nodes have zero running time by default.

More formally, the problem we target is defined as follows:

Problem Statement: Given a dataflow G, a fixed distribution of the initial data across M DCs, and a running time value $RTbase$, compute the r_i^j values s.t. $DM(G)$ is minimized and $RT(G)$ is always less than $(1+\varepsilon)RTbase$, where ε is a small constant $\varepsilon > -1$. If $0 > \varepsilon > -1$, then we enforce the solutions to seek improvements regarding both $DM(G)$ and $RT(G)$; when ε is positive, we tolerate increases in $RT(G)$ compared to $RTbase$. We can also regard positive values of ε as the percentage of the performance degradation that is tolerated.

[2]Note that in general, $S_i^j \neq r_i^j S^j$, i.e., the distribution of the intermediate results in a job is not necessarily the same as the way these results are shuffled in the next jobs. However, assuming a uniform distribution of results, it holds that $S_i^j = mean(r_i^k) * S^j$, where $(k, j) \in E$.

Table 2: Notations used in the paper.

Symbol	Meaning
$G(V,E)$	the data flow DAG
N, M	number of jobs and DCs
I^j	amount of input data of a job $v_j \in V$
α^j	selectivity of a job v_j
S^j	amount of intermediate output data of a job ($S^j = a^j * I^j$)
U_i	uplink bandwidth on DC i
D_i	downlink bandwidth on DC i
S_i^j	amount of intermediate data of v_j on DC i
r_i^j	fraction of tasks executed on DC i for jobs succeeding v_j
TU_i^j, TD_i^j	running time of intermediate data transfer on up and down link of DC i
$RT(G)$	total running time of G
$DM(G)$	total data movement between DCs in G
RT_j	running time of job v_j
DM_j	total data movement between DCs of job v_j
allocations	A $N \times M$ array holding in each row *allocations*$[j]$ the $r_i^j,\ j = 1 \ldots N,\ i = 1 \ldots M$ values

Note that the higher we set ε, the more the problem tends to be a single-objective optimization (that of minimizing $DM(G)$) in practice.

3.2 Existing Solutions and Limitations

In [19], a two-step approach was followed:

(1) Use Iridium [23] as the guideline for the initial assignment of tasks, i.e., computation of the r_i^j values, to the DCs. Iridium decides the allocation for each job separately, after performing a topological sorting on G, and considers the nodes from the upstream to the downstream ones. In this way, $RTbase$ is derived.
(2) Re-arrange the allocations with a view to decreasing the total movement cost while not allowing running time degradation more than ε times.

Then, for the second step, two techniques were proposed. The first one, is a fast greedy one. In the next section, we introduce another greedy technique explaining the differences. The second one is an Iterated Local Search (ILS) algorithm that uses Stochastic Hill Climbing (SHC) internally. It randomly perturbs the initial solution, and then looks for additional randomly chosen small changes in the perturbed configuration, so that $DM(G)$ improves,

Algorithm 1 Greedy-full algorithm

Require: *allocations*, *RTthreshold*, $DM(G)$, $RT(G)$, *iterations*
  bestAllocations ← *allocations*
  bestRT ← $RT(G)$
  bestDM ← $DM(G)$
  for i ←1 to iterations **do**
    for each job **do**
      bottleneckDC ← findBottleneckDC(job)
      Reallocate tasks regarding the current job through distributing a proportion of β of bottleneckDC's fraction to the other DCs
      tempAllocations ← apply changes to all downstream jobs in G
      Calculate $RT(G)'$ using *tempAllocations*
      Calculate $DM(G)'$ using *tempAllocations*
      if $DM(G)' < bestDM$ && $RT(G)' \leq RTthreshold$ **then**
        bestAllocations ← *tempAllocations*
        bestRT ← $RT(G)'$
        bestDM ← $DM(G)'$
      end if
    end for
  end for
  return *bestAllocations*, *bestRT*, *bestDM*

while $RT(G)$ remains under the threshold. The ILS-based solution is shown to be capable of yielding much better results at the expense of overhead that is higher by an order of magnitude; e.g., in large flows it took 2-3 minutes on a modern PC to check 75 random perturbations, each running SHC 75 times. The extremal optimization-inspired technique and the new greedy that we introduce in the next section manage to achieve similar quality in the results running much closer to the initial greedy technique, as discussed in the experiments.

4 OUR PROPOSAL

The aim is to devise fast algorithms being as effective as the ILS-one in [19].

4.1 A greedy solution that is less shortsighted

The first algorithm we implemented is a greedy one described in Algorithm 1 (termed *Greedy-full*). The algorithm works using an initial solution derived by Iridium [23] (we also examine using a random solution in Section 5.2.3). The input of the algorithm is (i) the initial allocation of tasks on the DCs, (ii) the running time threshold $RTthreshold = (1 + \varepsilon)RTbase$, where *RTbase* is the initial $RT(G)$, (iii) the initial $DM(G)$, (iv) the initial RT(G) and (v) the number of iterations. The output is the new allocation of tasks optimized for lower $DM(G)$ with the new $RT(G)$ to be under the threshold.

The algorithm consists of two loops. The external one is repeated 20 times while the internal one iterates over all the jobs. The number of the external iterations is configurable but unless otherwise stated, we set them to 20 (see Section 5.2.2). For each job in topological order it finds the bottleneck DC. More specifically, the *findBottleneckDC(job)* function in the algorithm returns the DC that has the least, non zero, task placement ratio. For this task placement ratio, the algorithm further removes a proportion of β and distributes it to the rest of the DCs that already have tasks proportionally. In this work, we set β equal to 1/3. Then, it assesses the global impact of such a local change. It re-calculates the task placement of the downstream nodes and if the new $RT(G)$ is under the threshold and the $DM(G)$ is minimized, then the solution becomes the best one.

In our previous work [19], we also implemented a greedy algorithm. The main difference with the algorithm of this work is that, when a job's task placement is altered, the affects are not transferred to the downstream nodes in the internal loop; i.e., the initial greedy solution focuses on local changes in a shortsighted manner. However, addressing this limitation comes at the expense of higher optimization times to derive the final task allocation, but, as shown later, the trade-off is interesting.

4.2 An EO-based solution

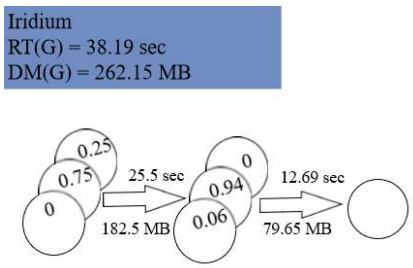
We propose an EO-based solution that will be referred to as *Extremal* (see Algorithm 2). Extremal uses also an initial solution, like Greedy-full. The input and the output remain the same for the two algorithms. Algorithm 2 consists of one loop. In each iteration, it finds the slowest job of the graph (through the *findSlowestJob(G)*) and rearranges its task placement fractions by removing a β fraction of the task ratio of randomly picked DCs. We set β equal to 1/3 and the probability is set to 1/2. This reallocation affects the downstream nodes as the S^j is re-arranged to the DCs; this reallocation is computed using the Linear Programming technique (LP) from [23]. Then the new $RT(G)$ and $DM(G)$ are calculated and the solution becomes the best one so far only if the $DM(G)$ improves and if the $RT(G)$ is under the given threshold. The number of the iterations is configurable but unless otherwise stated, we set them to 100 (see Section 5.2.2). Compared to the Greedy-full solution, its main difference is that it focuses on the slowest job overall rather than examining all jobs in turn; then, for the slowest jobs, examines more extensive random changes.

4.2.1 Example. Suppose a linear G with three nodes and three DCs on which each node is executed in parallel. The uplink and downlink of the DCs are U=(10, 1, 10), D=(10, 5, 5). The S_i^1 values are S_i^1=(120, 100, 50) and α =1 for both jobs. Figure 2a shows the result of Iridium. In the figure, each circle corresponds to a job-DC pair annotated by the corresponding r_i^j value.

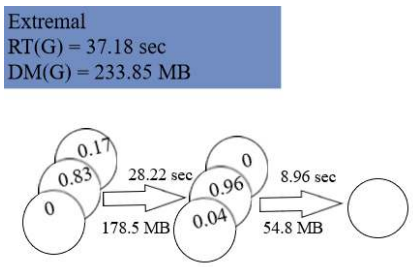
We execute the loop of Algorithm 2 a single time. We set ε = 0.1. Thus the threshold is set to $RTthreshold = (1 + 0.1)38.19 =$ 42 sec. First, the algorithm searches for the slowest job which in that case, is the first one. Then, it chooses a random DC that has a fraction of tasks larger than 0, let's say that this DC is the third one. Then the algorithm removes 1/3 of its workload and transfers it to the 2nd DC, which is the only other DC with non-zero allocation; this results in r_i^1=(0, 0.83, 0.17), which, in turn yields S_i^2=(0, 224.1, 45.9) and r_i^2=(0.04, 0.96, 0). The new $RT(G)$ is 37.18 sec. The benefit in the data movement is 28.3 MBs (Figure 2b). This new $RT(G)$ is under the threshold so the solution is accepted. The final reduction over Iridium is 2.6% in terms of $RT(G)$ and 11% in $DM(G)$ which is the main metric we try to minimize. In this example, Greedy-full can reach the same outcome as Extremal, but the latter has

Algorithm 2 Extremal algorithm

Require: *allocations*, *RTthreshold*, $DM(G)$, $RT(G)$, *iterations*
  bestAllocations ← *allocations*
  bestRT ← $RT(G)$
  bestDM ← $DM(G)$
  for i ←1 to iterations **do**
    slowestJob ← findSlowestJob(G)
    for *eachDC* **do**
      With probability p, reallocate tasks regarding the slowest job through distributing a proportion of β of DC's fraction to the other DCs
    end for
    tempAllocations ← apply changes to G
    Calculate $RT(G)'$ using *tempAllocations*
    Calculate $DM(G)'$ using *tempAllocations*
    if $DM(G)' < bestDM$ && $RT(G)' \leq RTthreshold$ **then**
      bestAllocations ← *tempAllocations*
      bestRT ← $RT(G)'$
      bestDM ← $DM(G)'$
    end if
  end for
  return *bestAllocations*, *bestRT*, *bestDM*



(a) Iridium task allocation



(b) Extremal algorithm task allocation

Figure 2: Example using extremal algorithm

performed only one reallocation (for the first job) while Greedy-full has checked all the jobs.

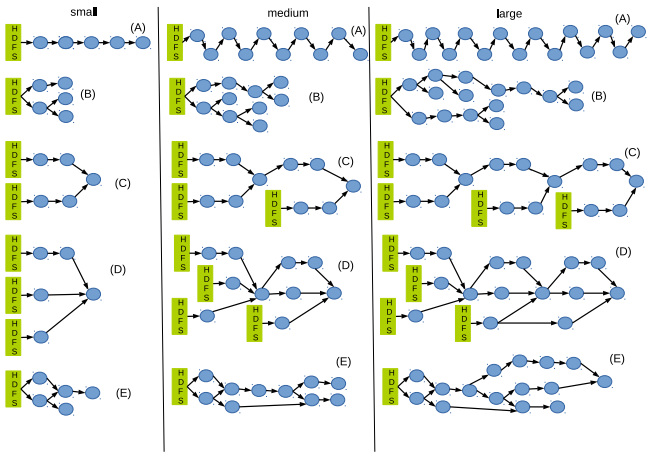


Figure 3: DAGs considered in the experiments (taken from [11])

5 EXPERIMENTS

5.1 Setting

We have already shown in Section 2 that estimated improvements correspond to improvements in real runs as well. To cover a broad range of scenarios, we resort to simulations. We use the simulation setting presented in our previous work [19], which includes five types of DAGs from [11] (presented in Figure 3) in three sizes each. The DAGs cover a very broad range of real applications, including DAGs produced when running TPC-H on Spark. To allow for a direct comparison against the results in [19], we experiment with 3 values of M = 5; 10; 15 and 3 values of ε = 0.1 and 0.2 and 0.5. The experiments were performed for every combination of DAG, number of DCs and ε value. Unless otherwise stated, $p = 0.5$, $iterations = 20$ for Greedy-full and 100 for Extremal, and $\beta = 1/3$. For the remainder of the variables, we resort to a setting similar to the one in [23]. The initial dataset I^j of the source nodes is randomly generated in the range [100MB, 1GB]. The U_i and D_i of each DC fall into the range of [100MB/sec, 2GB/sec]. The selectivities α of the jobs are between 0.01 and 2 with 50% of the job selectivities ranging from 0.01 to 0.5, 25% of them ranging from 0.5 to 1 and the rest 25% ranging from 1 to 2 (similar to the selectivities in Facebook production analytics according to [23]). For each combination of DAG type, M and ε, we created random instances according to the parameters above, and we report the average values.

5.2 Main Experiments

5.2.1 Main comparison. In the first set of experiments, we compare our algorithms namely Extremal and Greedy-full to the ones presented in our previous work [19], Iterated Local Search and Greedy, regarding the reduction in $RT(G)$ and $DM(G)$ they achieve over Iridium, when we set $\epsilon = 0.2$. The results are presented in Figure 4 and Figure 5 for $DM(G)$ and $RT(G)$, respectively. On average, Extremal reduces Iridium's $DM(G)$ by 28.16%, Greedy-full by

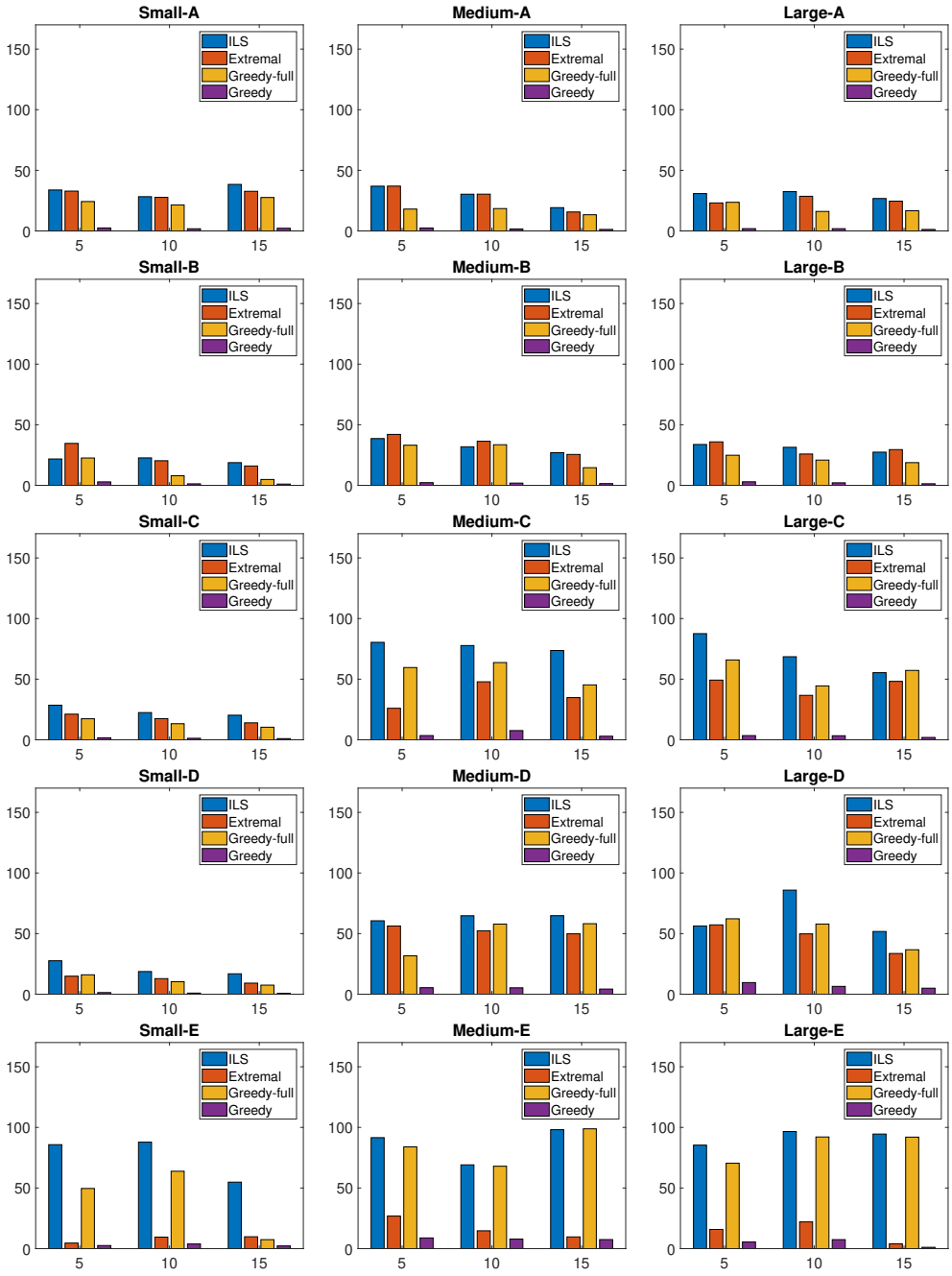


Figure 4: Percentage of $DM(G)$ reduction for M =5, 10 and 15 when ε=0.2.

37.83%, ILS by 50.12% and Greedy by 3.25%. In most cases, Greedy-full reduces the $RT(G)$ as well by a mean reduction of 28.26%, Extremal by 11.03%, ILS by 44.31%, while Greedy increases the $RT(G)$ by 7.5%.

As we can observe from Figure 4, Extremal is outperformed by ILS and Greedy-full by a large margin in the DAGs where the slowest job turns out to be one close to the sink nodes, e.g., Small E, where a single node collects data from two previous nodes. In the other cases, the behavior of Extremal and ILS is similar, whereas there exist several combinations of DAG types and sizes where Extremal is the best performing approach in average. The performance of Greedy-full is closer to that of ILS, but, as explained later, with slightly higher overhead than Extremal.

5.2.2 Convergence Rate. In this section, we compare the convergence rate of Extremal and Greedy-full. In order to find the

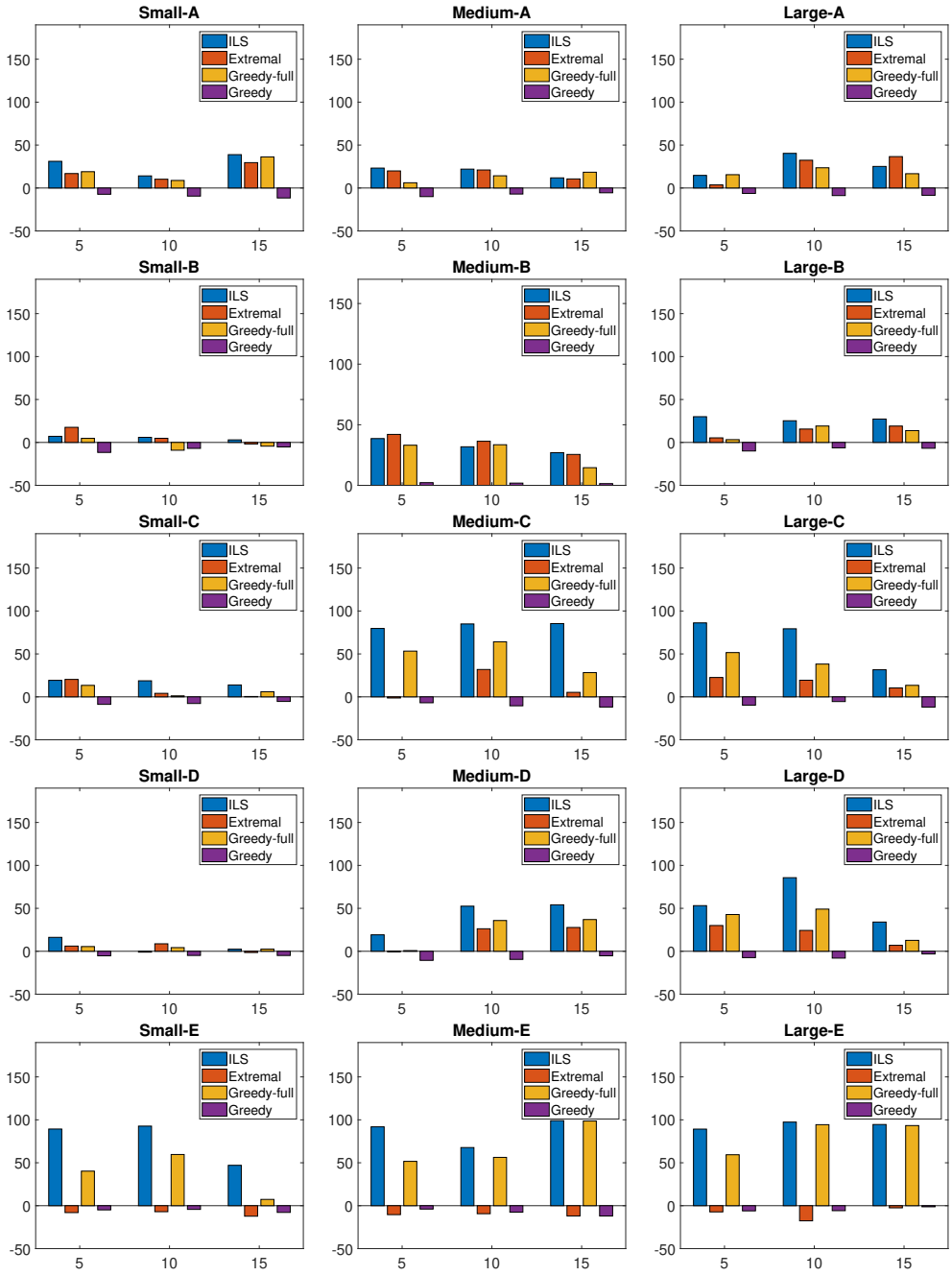


Figure 5: Percentage of $RT(G)$ reduction for M =5, 10 and 15 when ε=0.2.

convergence rate of Greedy-full we set the number of iterations to $N * M = 6 * 10 = 60$ while Extremal iterates 100 times. Figure 6 shows the results for the Small-A DAG and 10 machines. We can observe that Greedy-full converges at around the 10th iteration, which is faster than Extremal which converges at around 50th iteration. We should also consider the running time of the algorithms. While Extremal iterates more times, it only takes 5.7 sec but Greedy-full takes 13.08 sec. Taken that into consideration, Extremal converges at around 2.85 sec and Greedy-full at around 2.18 sec (machine specifications are given when discussing time overheads in more detail). This explains our choice to set the number of iterations of the two algorithms to 20 and 100, respectively. We further investigate the behavior of Extremal, ranging the number of iterations from 25 to 150. The results are presented in Figures 7

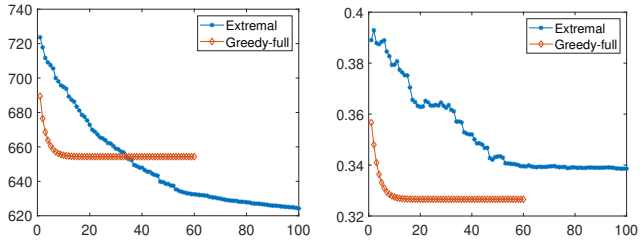


Figure 6: $DM(G)$ (left) and $RT(G)$ (right) convergence rate for the Small-A (top) DAG when running Extremal and Greedy-full (M=10, ε=10%)

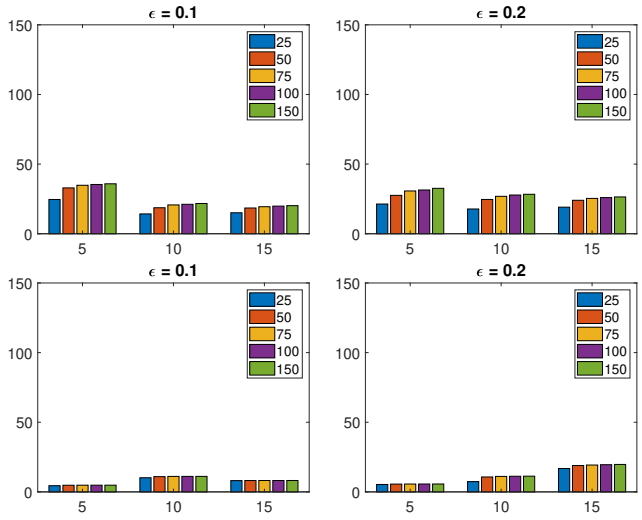


Figure 7: Percentage of $DM(G)$ reduction for the Small-A (top) and Large-E (bottom) DAGs when running Extremal for different M (horizontal axis), ε and number of iterations

and 8. Setting the iterations to 100 offers a good trade-off between the quality of the output and the running time of the algorithm.

5.2.3 Impact of initial allocation. In this set of experiments, we tried initializing the Greedy-full and Extremal algorithms with a random solution rather than the Iridium one. The results show that the algorithms are quite sensitive to the initial allocation as they cannot produce a plan that improves on the Iridium's $RT(G)$ and $DM(G)$ (no figures are shown due to space constraints). In most cases, the final results of the algorithms that were initialized with the random solution are worse than the Iridium ones. Therefore, the initialization phase in our solution that first optimizes for RT (though employing Iridium's approach) and then proceeds to DM minimization is crucial in the bi-objective optimization solution.

5.2.4 Time overheads. The running time of each algorithm is presented in Table 3. The experiments were performed on a machine with i7-4510U CPU at 2.00GHz with 8 GB of RAM. Two main observations can be drawn: (i) Extremal and Greedy-full incur lower overhead than ILS by an order of magnitude; and (ii) Greedy-full is slower than Extremal regarding non-small flows.

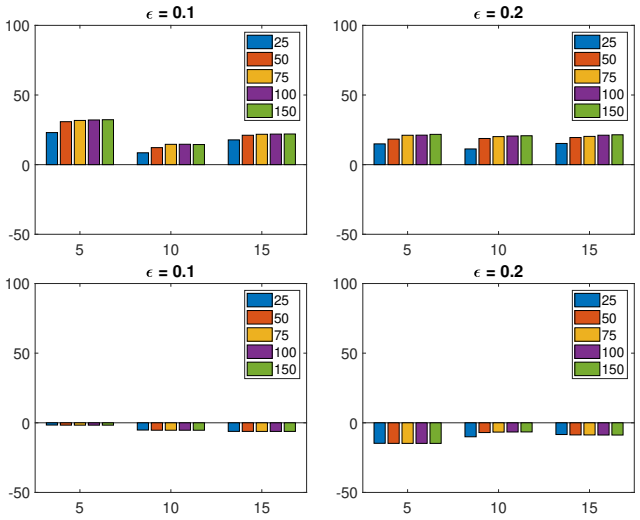


Figure 8: Percentage of $RT(G)$ reduction for the Small-A (top) and Large-E (bottom) DAGs when running Extremal for different M (horizontal axis), ε and number of iterations

Algorithm 3 Genetic algorithm

Require: $populationSize, recombinationProb, mutationProb, generations$

```
population ← initializePopulation(populationSize)
best, bestRT, bestDM ← getBest(population)
for i ← 1 to generations do
  parents ← selectParents(population)
  children ← ∅
  for each pair in parents do
    child1, child2 ← recombination(pair, recombinationProb)
    children ← mutate(child1, mutationProb)
    children ← mutate(child2, mutationProb)
  end for
  population ← combine(population, children)
  population ← population(1 : populationSize)
  best, bestRT, bestDM ← getBest(evaluatedPopulation)
end for
return best, bestRT, bestDM
```

5.3 Comparison against an Evolutionary Solution

In this section, we present how an evolutionary algorithm performs in our setting. Specifically, we have implemented the genetic algorithm described in Algorithm 3 and compared it to Extremal and Greedy-full. First, the Genetic algorithm initializes the population and finds the best solution among them. In our implementation, we initialized the population of size 400 with the Iridium's solution, about 10% greedy solutions over Iridium and 90% random ones. Then, the population is divided into pairs from the recombination of which new solutions (children) are produced. Then, the children are mutated with a small probability, inserted in the population and the best solution is found. This is repeated for a number

Table 3: Running times of the algorithms for different M values (in sec).

	Small A			Medium C			Large E		
Algorithm \M	5	10	15	5	10	15	5	10	15
Iridium	0.13	0.17	0.18	0.28	0.29	0.3	0.3	0.31	0.36
Extremal	5.44	5.7	5.71	4.37	4.73	5.95	13.91	18.17	18.77
Greedy-full	3.66	3.99	4.12	10.8	11.39	11.81	20.93	22.32	24.51
Greedy	0.16	0.19	0.21	0.8	1.12	1.31	2.45	3.29	3.6
ILS (75 iterations)	59.99	69.7	71.98	87.73	91.16	93.44	130.98	131.26	159.04

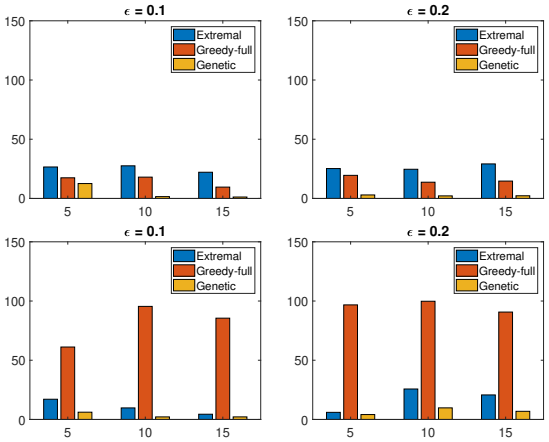


Figure 9: Percentage of $DM(G)$ reduction for the Small-A (top) and Large-E (bottom) DAGs when running Extremal, Greedy-full and Genetic for different M (horizontal axis) and ϵ

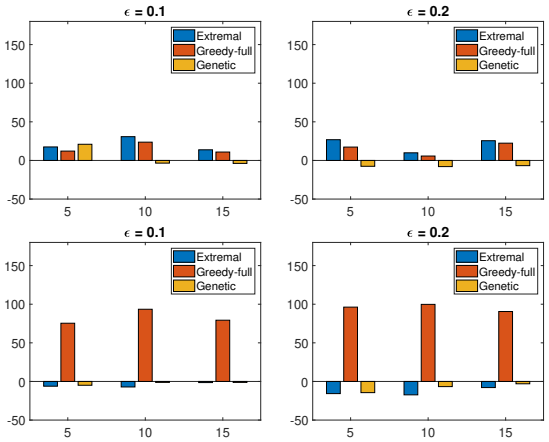


Figure 10: Percentage of $RT(G)$ reduction for the Small-A (top) and Large-E (bottom) DAGs when running Extremal, Greedy-full and Genetic for different M (horizontal axis) and ϵ

of iterations called generations. The output of the algorithm is the best $RT(G)$ and $DM(G)$.

Figures 9 and 10 show the results, when the number of generations is set to 400. As can be seen, Genetic is the least beneficial algorithm for our setting. That indicates that it is more effective to work on a single solution rather than having a collection of them, e.g the population in the Genetic. Moreover the random combination of components from different solutions does not lead to a good outcome either.

6 RELATED WORK

As the amount of jobs that need to be executed in geo-distributed data centers is increasing, there have been several proposals for optimized task placement. Many works focus on minimizing the total traffic. For example, WANalytics [27] deals with the task placement in this regard, but does not consider the overall running time. [17] offers a prediction of job execution time but focuses only on the minimization of the data movement as well. Clarinet [26] is a query optimizer that chooses the best execution plan among the ones provided by multiple query optimizers, considering the WAN-consumption during scheduling and task placement.

On the other hand, there are solutions that employ the minimization of the running time as their objective. Two earlier proposals, include Nebula [24] and Tetris [12] that overlook issues regarding total data movement. Heintz et al. [13] developed a framework that optimizes the data and task placement of each phase of a mapreduce job focusing on minimizing the makespan of the query but not on the overall data movement either. Iridium [23], which is the work against which we compare our solution, also focused only on the running time. However, Iridium can modify the placement of the initial data as well. In our work, we assume that initial data allocation is fixed. Tetrium [15] also tries to improve upon Iridium, as we do, in two ways. Firstly, through considering the time spent due to computations and not only data transmission. Secondly, through making scheduling decisions at a lower level than simple decision of the fraction of the tasks to run on each site to account for the case when the slots available are less than the allocated tasks. Both these extensions are interesting and we plan to investigate them in the future. Contrary to our proposal, it focuses mostly on response time but supports constraints on data movement (we treat the two metrics as of equal importance through first optimizing for response time and then for data movement); also, in our solutions, we manage to handle stage dependencies better through not running a stage-by-stage technique only once.

There are also works that consider both metrics. For example, Flutter [14] is a system that performs bi-objective task placement online but all tasks of the same stage are allocated to a single data

center. Works on multi-objective query optimization, such as [25], suffer from the same limitation. Finally, the work in [30] targets both metrics but is tailored to a single MapReduce flow with the reducer being executed on a single DC. In summary, none of these works can be applied to a generic DAG, where each DAG vertex is distributed across several nodes.

7 DISCUSSION

In this work, we proposed a fast solution that decides the task placement in complex analytics workflows targeting the minimization of both response time and data movement. The thorough experiments show that we can yield significant improvements over our main competitor with much less overhead than the previous proposal to the same end.

In general, there are further open issues in the multi-objective problem we deal with. Taking into consideration the processing costs and capacity constraints of the participating nodes, in line with the work in [15], is a promising direction for future work. Also, investigating how the required metadata can be efficiently monitored online is an open issue. Finally, further research is required for taking into account aspects such as scheduling decisions when multiple workflows run on the same infrastructure concurrently.

ACKNOWLEDGMENT

This research has been co-financed by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH - CREATE - INNOVATE (project code:T1EDK-01944)

REFERENCES

[1] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. 2017. Real-Time Video Analytics: The Killer App for Edge Computing. *IEEE Computer* 50, 10 (2017), 58–67.

[2] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. 601–613.

[3] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 1383–1394.

[4] Stefan Boettcher. 2000. Extremal Optimization: Heuristics via Coevolutionary Avalanches. *Computing in Science and Engg.* 2, 6 (Nov. 2000), 75–82.

[5] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing (MCC '12)*. ACM, New York, NY, USA, 13–16. https://doi.org/10.1145/2342509.2342513

[6] Jason Brownlee. 2011. *Clever algorithms: nature-inspired programming recipes*. Jason Brownlee.

[7] C. C. Byers. 2017. Architectural Imperatives for Fog Computing: Use Cases, Requirements, and Architectural Techniques for Fog-Enabled IoT Networks. *IEEE Communications Magazine* 55, 8 (Aug 2017), 14–20. https://doi.org/10.1109/MCOM.2017.1600885

[8] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *PVLDB* 10, 12 (2017), 1718–1729.

[9] David J. DeWitt and Jim Gray. 1992. Parallel Database Systems: The Future of High Performance Database Systems. *Commun. ACM* 35, 6 (1992), 85–98.

[10] S. Dolev, P. Florissi, E. Gudes, S. Sharma, and I. Singer. 2017. A Survey on Geographically Distributed Big-Data Processing using MapReduce. *IEEE Transactions on Big Data* (2017), 1–1.

[11] Anastasios Gounaris, Georgia Kougka, Rubén Tous, Carlos Tripiana Montes, and Jordi Torres. 2017. Dynamic Configuration of Partitioning in Spark Applications. *IEEE Trans. Parallel Distrib. Syst.* 28, 7 (2017), 1891–1904.

[12] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource Packing for Cluster Schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 455–466. https://doi.org/10.1145/2619239.2626334

[13] B. Heintz, A. Chandra, R. K. Sitaraman, and J. Weissman. 2016. End-to-End Optimization for Geo-Distributed MapReduce. *IEEE Transactions on Cloud Computing* 4, 3 (July 2016), 293–306. https://doi.org/10.1109/TCC.2014.2355225

[14] Z. Hu, B. Li, and J. Luo. 2016. Flutter: Scheduling tasks closer to data across geo-distributed datacenters. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2016.7524469

[15] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. 2018. Wide-area Analytics with Multiple Resources. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 12, 16 pages. https://doi.org/10.1145/3190508.3190528

[16] Konstantinos Kloudas, Margarida Mamede, Nuno Preguiça, and Rodrigo Rodrigues. 2015. Pixida: Optimizing Data Parallel Jobs in Wide-area Data Analytics. *Proc. VLDB Endow.* 9, 2 (Oct. 2015), 72–83.

[17] P. Li, S. Guo, T. Miyazaki, X. Liao, H. Jin, A. Y. Zomaya, and K. Wang. 2017. Traffic-Aware Geo-Distributed Big Data Analytics with Predictable Job Completion Time. *IEEE Transactions on Parallel and Distributed Systems* 28, 6 (June 2017), 1785–1796. https://doi.org/10.1109/TPDS.2016.2626285

[18] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, and W. Zhao. 2017. A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications. *IEEE Internet of Things Journal* 4, 5 (Oct 2017), 1125–1142. https://doi.org/10.1109/JIOT.2017.2683200

[19] Anna-Valentini Michailidou and Anastasios Gounaris. 2019. Bi-objective traffic optimization in geo-distributed data flows. *Big Data Research* https://doi.org/10.1016/j.bdr.2019.04.002 (2019).

[20] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*. 293–307.

[21] M. Tamer Özsu and Patrick Valduriez. 2011. *Principles of Distributed Database Systems, Third Edition*. Springer.

[22] Pankesh Patel, Muhammad Intizar Ali, and Amit P. Sheth. 2017. On Using the Intelligent Edge for IoT Analytics. *IEEE Intelligent Systems* 32, 5 (2017), 64–69.

[23] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. 2015. Low Latency Geo-distributed Data Analytics. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 421–434. https://doi.org/10.1145/2785956.2787505

[24] M. Ryden, K. Oh, A. Chandra, and J. Weissman. 2014. Nebula: Distributed edge cloud for data-intensive computing. In *2014 International Conference on Collaboration Technologies and Systems (CTS)*. 491–492.

[25] E. Tsamoura, A. Gounaris, and K. Tsichlas. 2013. Multi-objective Optimization of Data Flows in Multi-cloud Environment. In *Proceedings of the 2nd International Workshop on Data Analytics in the Cloud (DanaC'2013) (in conjunction with ACM SIGMOD/PODS'2013)*. New York, NY, 6–10. http://delab.csd.auth.gr/papers/DANAC2013tgt.pdf

[26] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. 2016. CLARINET: WAN-Aware Optimization for Analytics Queries. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 435–450. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/viswanathan

[27] Ashish Vulimiri, Carlo Curino, Brighten Godfrey, Konstantinos Karanasos, and George Varghese. 2015. WANalytics: Analytics for a Geo-Distributed Data-Intensive World. In *CIDR*.

[28] Ashish Vulimiri, Carlo Curino, P. Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. 2015. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 323–336.

[29] D. H. Wolpert and W. G. Macready. 1997. No Free Lunch Theorems for Optimization. *Trans. Evol. Comp* 1, 1 (1997), 67–82.

[30] Wenhua Xiao, Weidong Bao, Xiaomin Zhu, and Ling Liu. 2017. Cost-Aware Big Data Processing Across Geo-Distributed Datacenters. *IEEE Trans. Parallel Distrib. Syst.* 28, 11 (2017), 3114–3127.